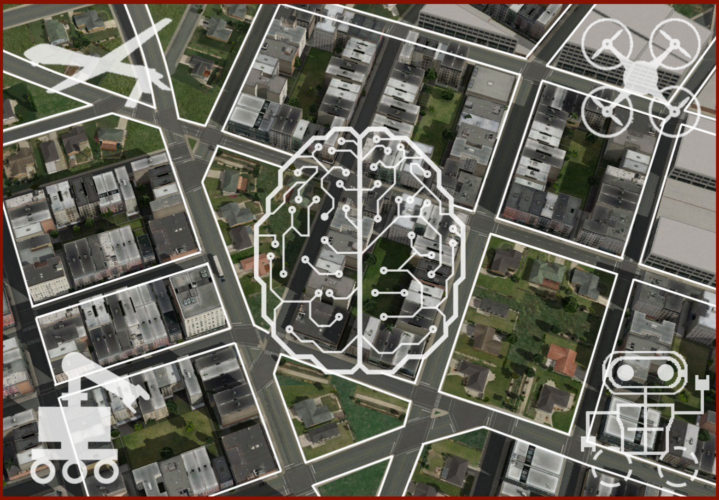
Tesis Doctoral

Architecture for Planning and Execution of Missions with Fleets of Unmanned Vehicles



Autor: Jorge Juan Muñoz Morera

Directores: Jesús Iván Maza Alcaníz

Aníbal Ollero Baturone

Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2019

Tesis Doctoral

Architecture for Planning and Execution of Missions with Fleets of Unmanned Vehicles

Autor:

Jorge Juan Muñoz Morera

Directores:

Jesús Iván Maza Alcaníz

Profesor Titular de Universidad

Aníbal Ollero Baturone

Catedrático de Universidad

Ingeniería de Sistemas y Automática

Escuela Técnica Superior de Ingeniería

Universidad de Sevilla

2019

Tesis Doctoral: Architecture for Planning and Execution of Missions with Fleets of Unmanned Vehicles

Autor: Jorge Juan Muñoz Morera
Directores: Jesús Iván Maza Alcaníz, Aníbal Ollero Baturone

El tribunal nombrado para juzgar la Tesis arriba indicada, compuesto por los siguientes doctores:

Presidente:

Vocales:

Secretario:

acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Fecha:

A mis padres Alfonso y Carmen

Agradecimientos

Esta tesis doctoral es fruto de todo el trabajo realizado a lo largo de los 5 años que ha durado. Durante todo este tiempo he tenido que compaginarla no sólo con mi propio trabajo, sino también con mi vida personal. Hacer una tesis es difícil no sólo por el esfuerzo necesario durante la investigación, sino también por el tiempo dedicado. Es inevitable pasar menos tiempo con aquellos a los que quieres, y eso ha sido especialmente duro para mí.

Por todo lo anterior, quiero agradecer todo el apoyo recibido a mi familia, puesto que ellos han sido los que me han animado a finalizar mis estudios. En especial, agradecerle a mis padres Alfonso y Carmen por haber estado presente en los momentos mas duros y dar la talla siempre. Espero poder devolverles todo lo que me han dado algún día y de alguna forma. También a mi hermana Sandra, por ser mi confidente, escuchar todos y cada uno de mis problemas y ayudarme a contar hasta diez cuando ha sido necesario. Y por supuesto a mis sobrinos, por alegrarme el alma con solo mirarlos.

También tengo que agradecer a los profesores Iván Maza y Aníbal Ollero, por permitirme entrar en el Grupo de Robótica, Visión y Control y aprender todo lo que significa y rodea al mundo de la investigación y la docencia.

No quiero olvidarme de agradecer a mi compañero en CATEC Antonio Jiménez Bellido, puesto que fué él quién me permitió entrar por primera vez en el mundo de los vehículos aéreos allá en el año 2012, cuando aún no sabía ni que existían, descubriendo una línea de trabajo poco común para un ingeniero informático. Agradecimientos a él y también a mi antiguo compañero Luis Díaz, puesto que actuaron como tutores durante mi primer paso por el centro y me enseñaron mucho. También a Antidio Viguria, por haberse acordado de mí y confiar en mis cualidades para la segunda etapa. Y por supuesto, gracias también a mis compañeros actuales de Altran.

Finalmente, agradecerle a Inma su paciencia conmigo y sobre todo, su compañía. Porque gracias a personas como ella la vida es un poquito más fácil.

Jorge Juan Muñoz Morera
Investigador en Altran Defence and Space

Sevilla, 2018

Acknowledgements

This Thesis is the result of all the work done through five years of duration. During all this time I had to combine my Ph.D. studies not only with my job, but also with my personal life. Doing a Thesis is hard not only for the needed effort but also for the time spent. It is unavoidable to share less time with those you love, and this has been specially hard for me.

For all the above, I would like to thank the received support to my family, because they encouraged me to finish my studies. Specially, to my parents Alfonso and Carmen for being with me in the hard moments and making the grade always. I hope I can return them all their love someday and somehow. Thanks to my sister Sandra, for being my confident, hearing all my problems and helping me count to ten when needed. And of course to my nephews, for gladding my soul.

I have to thanks professors Iván Maza and Anibal Ollero for letting me in the Robotics, Vision and Control Group and learn all that surrounds the field of research and teaching.

I do not want to forget my companion in CATEC Antonio Jiménez Bellido, because he brought me the opportunity to enter the field of unmanned aerial vehicles for the first time in 2012, when I still did not know they existed, opening an uncommon work line for a computer science engineer. Thanks to him and also to my old companion Luis Díaz, because both acted as my tutors during my first stage at CATEC and I learned a lot from them. Thanks also to Antidio Viguria for remembering me and trusting in my qualities for the second stage. And of course, thanks to all my current colleagues in Altran.

Finally, I would like to thank Inma for her patience and especially, for her company. Because people like her make the life a little bit easier.

Jorge Juan Muñoz Morera
Researcher in Altran Aerospace and Defence

Seville, 2018

Resumen

Esta tesis presenta contribuciones en el campo de la planificación automática y la programación de tareas, la rama de la inteligencia artificial que se ocupa de la realización de estrategias o secuencias de acciones típicamente para su ejecución por parte de vehículos no tripulados, robots autónomos y/o agentes inteligentes. Cuando se intenta alcanzar un objetivo determinado, la cooperación puede ser un aspecto clave. La complejidad de algunas tareas requiere la cooperación entre varios agentes. Mas aún, incluso si una tarea es lo suficientemente simple para ser llevada a cabo por un único agente, puede usarse la cooperación para reducir el coste total de la misma. Para realizar tareas complejas que requieren interacción física con el mundo real, los vehículos no tripulados pueden ser usados como agentes. En los últimos años se han creado y utilizado una gran diversidad de plataformas no tripuladas, principalmente vehículos que pueden ser dirigidos sin un humano a bordo, tanto en misiones civiles como militares.

En esta tesis se aborda la aplicación de planificación simbólica de redes jerárquicas de tareas (HTN planning, por sus siglas en inglés) en la resolución de problemas de enrutamiento de vehículos (VRP, por sus siglas en inglés) [18], en dominios que implican múltiples vehículos no tripulados de capacidades heterogéneas que deben cooperar para alcanzar una serie de objetivos específicos.

La planificación con redes jerárquicas de tareas describe dominios utilizando una descripción que descompone conjuntos de tareas en subconjuntos más pequeños de subtareas gradualmente, hasta obtener tareas del más bajo nivel que no pueden ser descompuestas y se consideran directamente ejecutables. Esta jerarquía es similar al modo en que los humanos razonan sobre los problemas, descomponiéndolos en subproblemas según el contexto, y por lo tanto suelen ser fáciles de comprender y diseñar.

Los problemas de enrutamiento de vehículos son una generalización del problema del viajante (TSP, por sus siglas en inglés). La resolución del problema del viajante consiste en encontrar la ruta más corta posible que permite visitar una lista de ciudades, partiendo y acabando en la misma ciudad. Su generalización, el problema de enrutamiento de vehículos, consiste en encontrar el conjunto de rutas de longitud mínima que permite cubrir todas las ciudades con un determinado número de vehículos. Ambos problemas cuentan con una fuerte componente combinatoria para su resolución, especialmente en el caso del VRP, por lo que su presencia en dominios que van a ser tratados con un planificador HTN clásico supone un gran reto.

Para la aplicación de un planificador HTN en la resolución de problemas de enrutamiento de vehículos desarrollamos dos métodos. En el primero de ellos presentamos un sistema de optimización de soluciones basado en puntuaciones, que nos permite una nueva forma de conexión entre un software especializado en la resolución del VRP con el planificador HTN. Llamamos a este modo de conexión el método desacoplado, puesto que resolvemos la componente combinatoria del problema de enrutamiento de vehículos mediante un solucionador específico que se comunica con el planificador HTN y le suministra la información necesaria para continuar con la descomposición de tareas. El segundo método consiste en mejorar el planificador HTN utilizado para que sea capaz de resolver el problema de enrutamiento de vehículos de la mejor forma posible sin tener que depender de módulos de software externos. Llamamos a este modo el método acoplado. Con este motivo hemos desarrollado un nuevo planificador HTN que utiliza un algoritmo de búsqueda distinto del que se utiliza normalmente en planificadores de este tipo.

Esta tesis presenta nuevas contribuciones en el campo de la planificación con redes jerárquicas de tareas para la resolución de problemas de enrutamiento de vehículos. Se aplica una nueva forma de conexión entre dos planificadores independientes basada en un sistema de cálculo de puntuaciones que les permite colaborar en la optimización de soluciones, y se presenta un nuevo planificador HTN con un algoritmo de búsqueda distinto al comúnmente utilizado. Se muestra la aplicación de estos dos métodos en misiones civiles dentro del entorno de los Proyectos ARCAS y AEROARMS financiados por la Comisión Europea y se presentan extensos resultados de simulación para comprobar la validez de los dos métodos propuestos.

Abstract

This thesis presents contributions in the field of automated planning and scheduling, the branch of artificial intelligence that concerns the realization of strategies or action sequences typically for execution by unmanned vehicles, autonomous robots and/or intelligent agents. When trying to achieve certain goal, cooperation may be a key aspect. The complexity of some tasks requires the cooperation among several agents. Moreover, even if the task is simple enough to be carried out by a single agent, cooperation can be used to decrease the overall cost of the operation. To perform complex tasks that require physical interaction with the real world, unmanned vehicles can be used as agents. In the last years a great variety of unmanned platforms, mainly vehicles that can be driven without a human on board, have been developed and used both in civil and military missions.

This thesis deals with the application of Hierarchical Task Network (HTN) planning in the resolution of vehicle routing problems (VRP) [18] in domains involving multiple heterogeneous unmanned vehicles that must cooperate to achieve specific goals.

HTN planning describes problem domains using a description that decomposes set of tasks into subsets of smaller tasks and so on, obtaining low-level tasks that cannot be further decomposed and are supposed to be executable. The hierarchy resembles the way the humans reason about problems by decomposing them into sub-problems depending on the context and therefore tend to be easy to understand and design.

Vehicle routing problems are a generalization of the travelling salesman problem (TSP). The TSP consists on finding the shortest path that connects all the cities from a list, starting and ending on the same city. The VRP consists on finding the set of minimal routes that cover all cities by using a specific number of vehicles. Both problems have a combinatorial nature, specially the VRP, that makes it very difficult to use a HTN planner in domains where these problems are present.

Two approaches to use a HTN planner in domains involving the VRP have been tested. The first approach consists on a score-based optimization system that allows us to apply a new way of connecting a software specialized in the resolution of the VRP with the HTN planner. We call this the decoupled approach, as we tackle the combinatorial nature of the VRP by using a specialized solver that communicates with the HTN planner and provides all the required information to do the task decomposition. The second approach consists on improving and enhancing the HTN planner to be capable of solving the VRP without needing the use of an external software. We call this the coupled approach. For this reason, a new HTN planner that uses a different search algorithm from these commonly used in that type of planners has been developed and is presented in this work.

This thesis presents new contributions in the field of hierarchical task network planning for the resolution of vehicle routing problem domains. A new way of connecting two independent planning systems based on a score calculation system that lets them cooperate in the optimization of the solutions is applied, and a new HTN planner that uses a different search algorithm from that usually used in other HTN planners is presented. These two methods are applied in civil missions in the framework of the ARCAS and AEROARMS Projects funded by the European Commission. Extensive simulation results are presented to test the validity of the two approaches.

Contents

Nomenclature

$\mathscr{M}$	Mission definition for a domain
$\mathscr{T}$	Set of tasks that compose a mission
$\mathscr{H}$	Set of home locations of the aerial vehicles
$\mathscr{L}$	Set of stock parts locations or locations where the tasks have to be done
$\mathscr{L}'$	Set of locations where the parts have to be assembled
V	Set of vertices of a graph
E	Set of edges of a graph
$G(V,E)$	Graph composed of V vertices and E edges
$G_N(V,E)$	Graph with N number of vertices
P_i	Set of preconditions for the i-th task
R_k	Route for the k-th aerial vehicle
c_{r_i,r_j}	Non-negative travel time between vertex r_i and r_j
$\mathscr{R}$	Set of routes for the aerial vehicles
$C(R_k)$	Cost for the route of the k-th aerial vehicle
$g(n)$	Cost for getting from the start node to the n-th node
$h(n)$	Heuristic cost function or cost for getting from the n-th node to the closest goal node
$f(n)$	Sum of $g(n)$ and $h(n)$

1 Introduction

This thesis presents contributions in the applications of unmanned aerial vehicles to civilian missions. More precisely, the goal of this thesis is the application of Hierarchical Task Network (HTN) planning in the resolution of vehicle routing problems (VRP) [18] in domains involving unmanned vehicles that must cooperate to achieve specific goals.

This chapter presents the motivation and main objectives of the research carried out. Then, the outline and main contributions are presented. Finally, the chapter ends describing the framework in which the work has been developed.

1.1 Motivation and Objectives

To perform complex tasks that go beyond the execution of software and that requires physical interaction with the real world, unmanned vehicles can be used as agents. Tasks such as data and image acquisition of areas, map building, target tracking, infrastructure inspection or even human interaction and aiding, among many others, require the agents to have the capability to interact with the world, not only by moving or sensing along the scene where the action takes place but also by modifying and adapting it with the purpose of achieving their goals. To completely reduce the execution costs of their tasks it is desirable to avoid the needs of having a human operator that drives the behaviour or movements of the vehicles, or at least to reduce as much as possible its interaction. In the last years this has been accomplished by the development of a great variety of unmanned platforms, mainly vehicles that can be driven without a human on board. Unmanned vehicles can either be remotely controlled or remotely guided but the ones for our interest are the autonomous vehicles, which are capable of sensing their environment and navigating on their own. From here on we refer to these autonomous vehicles simply as unmanned vehicles.

Unmanned Aerial Vehicles (UAVs) are self-propelled air vehicles capable of conducting autonomous operations. UAVs have been used in military applications and, in general, for classified purposes. Nowadays it is clear that UAVs have a wide range of civil applications.

Ground vehicles still have limitations to reach the locations that humans specify, both indoor and outdoor. This, along with the needs to act as fast as possible, impose large restrictions to the use of ground vehicles. The higher mobility and maneuverability of UAVs with respect to ground vehicles makes them a natural approach for tasks like information gathering or even the deployment of instrumentation, among many others. Many advances have been developed in positioning and navigation systems and the on-board sensorial and computational capabilities have been greatly improved. Although the research done on this thesis can be applied to any kind of unmanned vehicles, due to their high versatility we will center our attention on unmanned aerial vehicles.

When trying to achieve a certain goal, cooperation may be a key aspect. The complexity of some tasks often requires cooperation among several agents. Moreover, even if the task is simple enough to be carried out by a single agent, cooperation can be used to decrease the overall cost. Cooperation among multiple unmanned vehicles may be needed depending on the complexity of the tasks. For example:

- When it is necessary to operate simultaneously in different locations.
- When the target area is very large.
- When there are payload limitations for the unmanned vehicles to execute the mission.
- When there are restrictions related with the energy consumption of the unmanned vehicles.

Restrictions related with energy consumption, weight, payload or size play an important role in the design of unmanned vehicles, specially those of low cost, lightweight and small dimensions. For this reason, the use of multiple cooperative unmanned vehicles is the most suitable and versatile approach for many applications.

In addition, unmanned vehicles need to exhibit intelligent behaviour in order to be able to cooperate. Reasoning about the near future is needed to reach such intelligence. The reasoning process is called planning and, in the artificial intelligence community, it has been divided in smaller problems: plan the next actions with task planning, plan the motions of the vehicles with motion planning, plan the pose of the end effectors with grasp planning, and many other forms of planning.

One of the goals of this thesis is to study symbolic planning in domains involving the Vehicle Routing Problem (VRP). The main objective of this work is to address the use of a HTN planner to solve a problem that has a combinatorial nature such as the VRP. As we want to center our attention mainly in the symbolic level, motion planning is not a key aspect of our research, although the literature has been studied to learn the different possibilities and techniques for connecting symbolic planners with motion planners. The geometric component in this thesis has been modelled at a high level of abstraction, using cartesian coordinate systems for locations, graphs with costs for navigation maps, euclidean distances and so on.

Among the different possibilities for the symbolic planning field we have chosen Hierarchical Task Network (HTN) planning. HTN planning describes problem domains using a description that decomposes set of tasks into subsets of smaller tasks and so on, until obtaining low-level tasks that cannot be further decomposed and are supposed to be executable. The hierarchy resembles the way the humans reason about problems by decomposing them into sub-problems depending on the context and therefore tends to be easy to understand and design. One of the main advantages of HTN planning is that it uses domain-dependent knowledge to guide the search process. To solve a problem, a domain expert has to design the decomposition tree for the task network and that tree will serve as a heuristic for the search. In addition to this, its expressivity helps to understand the task and the hierarchy decomposition even by people outside the planning field, which is very valuable when different fields of expertise must cooperate.

One of the counterparts of HTN planning is its limitations when reasoning in domains that have a combinatorial nature such as the VRP. The search process is guided by the domain-dependent knowledge provided in the decomposition tree, but the algorithm of the planner does not take part in the decision of which node of the search tree to expand, as this is usually given by the proper decomposition tree. As a result, finding the optimal solution in problems such as the VRP may not be possible, as at the present moment there is no known heuristic to optimally find a solution for this problem. In these cases, the solutions found by HTN planners depend on the order on which the search nodes are expanded. Thus, it is needed to enhance the efficiency of the planners and to seek alternatives to overcome these limitations.

In this thesis we formulate a way to enhance the performance of HTN planning in the resolution of vehicle routing problems in domains involving unmanned vehicles. Two approaches have been developed and tested. The first approach consists on a score-based optimization system that allows us to apply a new way of connecting a software specialized in the resolution of the VRP with the HTN planner. We call this the *decoupled approach*, as we tackle the combinatorial nature of the VRP by using a specialized solver that communicates with the HTN planner and provides all the required information to do the task decomposition. The second approach consists on improving and enhancing the HTN planner to be capable of solving the VRP without needing the use of an external software. We call this the *coupled approach*. For this reason, a new HTN planner that uses a different search algorithm from these commonly used in that type of planners has been developed and is presented in this work.

One of the key objectives of this thesis was the application of the two approaches in civil missions involving the use of unmanned vehicles in the context of vehicle routing problems. The software implementation for the two approaches has been tested and validated in simulations with autonomous vehicles in the framework of the ARCAS and AEROARMS Projects funded by the European Commission. The structure of the thesis is presented in the next section.

1.2 Outline and Main Contributions

Main contributions provided by this thesis are related to HTN planning in domains that involve the vehicle routing problem. Particularly, the work presented in this thesis tackles the performance of HTN planners for solving VRP-based domains, presenting two approaches that extend the research previously done in this field and applying it for its use in civil missions involving the use of unmanned vehicles in the context of vehicle routing problems.

This thesis applies a decoupled approach based on a score optimization system to integrate a symbolic HTN planner with a constraint satisfaction solver for VRP domains involving assembly operations. The approach uses a bi-directional communication between the VRP planner and the HTN planner, in a closed-loop that receives feedback from the symbolic level to improve the search in the VRP solver, optimizing the solutions found. The main contribution is a new way of connecting two independent planning systems based on a score calculation system that lets them cooperate in the optimization of the solutions found and its application in the context of structure assembly missions.

A coupled approach for solving VRP domains with a single HTN planner in the context of structure inspection operations is also studied in this thesis and as a result, a new HTN planner has been developed. Our main contribution here is the development and implementation of a new heuristic-guided algorithm to drive the HTN planner search towards the optimal solutions. This new algorithm is completely different from the depth-search algorithm that is commonly used in other HTN planners.

The contents of the different chapters are summarized in the following paragraphs.

We start in **Chapter 2** by presenting the different plan search algorithms that are more commonly used to solve classical planning problems, followed by a review of developed planning software. Then, the integration problem between different planning levels that arises when trying to solve complex problems is presented. This is followed by a study of related work, sorted by categories of approaches developed to solve the integration between the different planning levels.

Chapter 3 explains the motivations of our decision to use HTN planning over other valid task planning techniques and presents an overview of HTN planning. Then, we define more precisely the HTN formalism used in this thesis. After that, we present the HTN planner used in our research: JSHOP2 [94]. We start by explaining the Planning Domain Definition Language (PDDL) on which JSHOP2 is based, then the different elements that compose a JSHOP2 problem are explained, and finally the JSHOP2 planning algorithm is presented.

The decoupled approach for our new score-based way of connecting the symbolic HTN planner with a constraint satisfaction solver for VRP domains is presented in **Chapter 4**. We present an overview of the solver used to address the VRP and which communicates with the symbolic HTN planner: OptaPlanner [100]. Then we formalize the solved problem, a variant of the Vehicle Routing Problem [18] that appears in the context of the ARCAS Project. After that, the domains designed for both planners are explained and the simulation results are presented. The contents of this chapter have resulted in several publications, two journal papers[91, 88] and three conference papers[90, 89, 80], and present a new way of connecting two independent planners based on a score calculation systems that lets them collaborate in the solution optimization.

In **Chapter 5** we describe a coupled approach to enhance the performance of a HTN planner in the context of VRP domains involving structure inspection operations, presenting a new developed HTN planner based on the replacement of the HTN depth-first search algorithm of JSHOP2 by an heuristic search like the one used in the A* search algorithm. First, the motivations and expectations for our new HTN algorithm are described. A brief overview of the A* algorithm and its properties, which inspired the development of our new planner, is also presented. Then, the description of our new heuristic-guided SHOP* HTN planner and its properties are given. In the context of the AEROARMS Project, an use-case to test the optimality of our new planner is presented and a deeper study is done by applying a benchmark and comparing the results with the original implementation of the JSHOP2 planner and the decoupled approach used in the previous chapter. The results of this chapter have been submitted to the Engineering Applications of Artificial Intelligence Journal (EAAI) and is currently under review.

Finally, **Chapter 6** summarizes the conclusions of the thesis and proposes some guidelines for further research.

It is worth to mention that the work presented in this thesis has been mainly developed in the framework of the ARCAS Project and AEROARMS Project funded by the European Union (see Section 1.3), and the **ARM-EXTEND** State Project.

1.2.1 Summary of Publications

Part of this thesis has been published in the following journals, book chapter and international conferences, including two journals indexed in the Journal Citation Reports (JCR) database:

- I. Maza, J. Muñoz-Morera, F. Caballero, E. Casado, V. Perez-Villar, and A. Ollero, *Architecture and tools for the generation of flight intent from mission intent for a fleet of unmanned aerial systems*, Unmanned Aircraft Systems (ICUAS), 2014 International Conference on, IEEE, May 2014, pp. 9–19.
- J. Muñoz-Morera (as speaker), I. Maza, C. J. Fernandez-Agüera, F. Caballero, and A. Ollero, *Assembly planning for the construction of structures with multiple UAS equipped with robotic arms*, Unmanned Aircraft Systems (ICUAS), 2015 International Conference on, IEEE, June 2015, pp. 1049–1058.
- J. Muñoz-Morera, I. Maza, F. Caballero, and A. Ollero, *Architecture for the automatic generation of plans for multiple UAS from a generic mission description*, Journal of Intelligent and Robotic Systems 84 (2016), no. 1, 493–509 (English).
- J. Muñoz-Morera, I. Maza, C. J. Fernandez-Agüera, and A. Ollero, *Task allocation for teams of aerial robots equipped with manipulators in assembly operations*, Advances in Intelligent Systems and Computing, vol. 417, pp. 585–596, Springer International Publishing, 2016.
- J. Muñoz-Morera, F. Alarcon, I. Maza, and A. Ollero, *Combining a hierarchical task network planner with a constraint satisfaction solver for assembly operations involving routing problems in a multi-robot context*, International Journal of Advanced Robotic Systems 15 (2018), no. 3, 1–13 (English).
- J. Muñoz-Morera and I. Maza and A. Ollero. *SHOP*: a heuristic guided implementation for dealing with the Vehicle Routing Problem.* Engineering Applications of Artificial Intelligence Journal (under review).

  It is important to note that the author of this thesis was the speaker of the conference paper presented in the International Conference on Unmanned Aircraft Systems (ICUAS) in 2015.

1.3 Framework

An important part of the work presented in this thesis has been developed within the framework of the European Projects **ARCAS** and **AEROARMS**, and the State Project **ARM-EXTEND**.

The ARCAS (Aerial Robotics Cooperative Assembly System) project lasted from November 2011 till November 2015 and is probably the first research project involving load transportation, manipulation and deployment with multiple autonomous aerial vehicles equipped with robotics arms.

The general objective of ARCAS[1] was the development and experimental validation of the first cooperative free-flying robot system for assembly and structure construction. The ARCAS project provided integrated and consolidated scientific foundations for flying robot perception, planning and control, and produced a framework for the design and development of cooperating flying robots for assembly operations. The integration of all these functionalities made a solid ground for new applications and services in aerial and space robotics. The building of platforms for the evacuation of people in rescue operations or the installation of platforms in uneven terrains for landing of manned and unmanned VTOL aircrafts are some examples of aerial robotics potential.

The ARCAS robotic system involved transportation of parts by means of one and several (joint transportation) flying robots, and the precise placement and assembly of the parts with appropriate manipulation devices to build a structure or to assembly an object. The project used experimental testbeds for the validation of concepts and algorithms in practical applications.

The project had the following scientific and technological objectives:

- **Motion control.** The development of new techniques and methodologies for motion control of a free-flying robot with a mounted manipulator in contact with a grasped object as well as for coordinated control of multiple cooperating flying robots with manipulators in contact with the same object.
- **Perception.** Development of new perception techniques to model, identify and recognize the scenario and for use in guidance in the assembly operations by means of flying robots.
- **Cooperative assembly planning.** The development of new methods for cooperative assembly by means of multiple flying robots with application to the assembly of objects, in sites that are very difficult to access from the ground.
- **Operator assistance.** Development of strategies for operator assistance in manipulation tasks involving multiple cooperating flying robots. Identification of the appropriate way to provide information concerning the system state to the operator and for his/her interaction with the automatic control system using a previously developed VR-haptic system.

As it can be read, the ARCAS project has two main axis: structure assembly by a team of UAVs and space manipulation. The latter is a side-part of the project and primarily

[1] http://www.arcas-project.eu

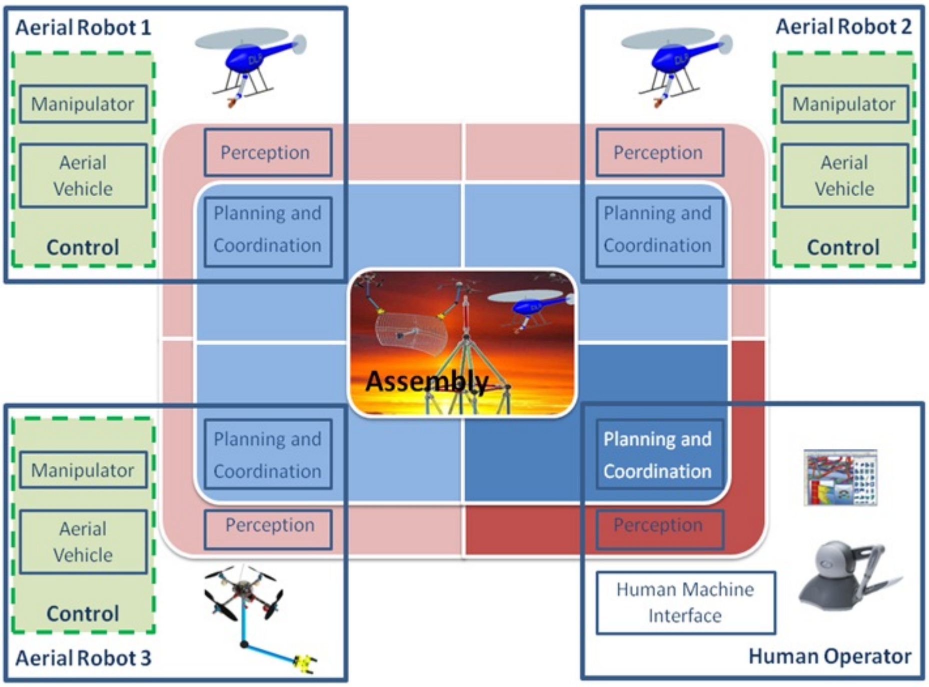


Figure 1.1 Overview of the ARCAS project. The objectives of the project included new methods for motion control and coordination of free-flying robots equipped with manipulators, new flying robot perception methods for identification and recognition of objects, new methods for cooperative assembly planning and structure construction and new strategies for operator assistance.

involved the control community. The structure assembly is the part combining work on assembly sequence planning, task planning, motion planning and execution supervision.

In the project, the planning level was composed of three planners: an assembly sequence planner that computes the assembly sequence from the CAD model of the structure to be assembled, the symbolic planner which is responsible for the task allocation and sequencing, and finally a motion planner to compute the trajectories for each action in the symbolic plan.

The AEROARMS European Project (AErial RObotic system integrating multiple ARMS and advanced manipulation capabilities for inspection and maintenance) was launched on June 2015 and is coordinated by Prof. Aníbal Ollero. The project is funded by the European Commission program Horizon 2020 for Research and Development.

AEROARMS[2] is a 4 years project. The challenging objective of AEROARMS is the development of the first aerial robots in the world with multiple articulated arms and advanced manipulation capabilities. The aerial robots will be able to fly to inaccessible sites

[2] http://www.aeroarms-project.eu/

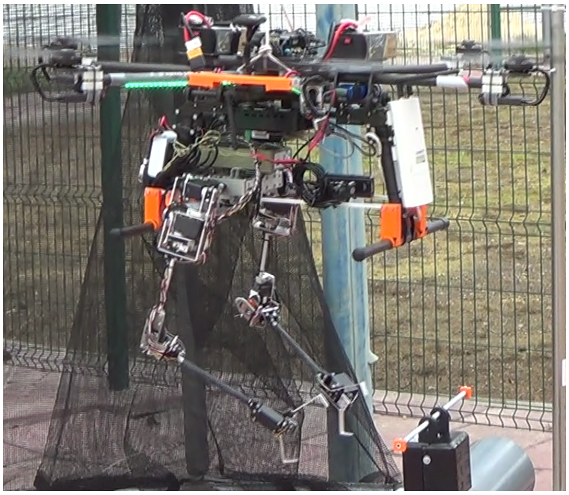


Figure 1.2 The objective of the AEROARMS Project is the development of the first aerial robots in the world with multiple articulated arms and advanced manipulation capabilities. The aerial robots will be able to fly to inaccessible sites and perform manipulation tasks with two arms while flying or they will anchor to a structure, using other arm for accurate manipulation tasks.

and perform manipulation tasks with two arms while flying or they will anchor to a structure, using other arm for accurate manipulation tasks. The robots will be applied to industrial inspection and maintenance tasks that currently require the use of scaffolds, cranes and manned aircraft. These aerial robotic manipulators are an evolution of the unmanned aerial systems with the added possibility of physical interaction with the environment by means of articulated arms.

AEROAMS includes the demonstration of the developed technologies in oil and gas industries that currently involves very high maintenance costs (approximately €56.000 million a year). Then, the future use of a system as AEROARMS might have an important economic impact, in addition to performing tasks that can be dangerous for workers. Two particular applications will be demonstrated:

- Installation and maintenance of permanent Non Destructive Tests (NDT) sensors on remote components such as pipes, fire flares or structural components. The application involves the preparation of structures to install the sensors (drilling a hole into insulation, removing paint, etc.), the installation of the sensors and the finishing of the structure.
- Deploying and maintaining a mobile robotic system permanently installed on a remote structure. Assuming the presence of a newly designed mobile robot allowing easy exchange and maintenance of components (e.g., batteries etc.), the application consists of the use of the aerial robot to maintain the robot permanently installed in the structure without costly and dangerous human operations. The results of

AEROARMS could be also applied to the inspection and maintenance of generation plants, critical infrastructures and others.

AEROARMS continues the work carried out in the FP7 European project ARCAS, in which the first aerial robots with 6 and 7 degrees of freedom arms and perception and planning capabilities have been developed and are being demonstrated in structure construction.

The ARM-EXTEND project proposes the development of the first robotic manipulation system with aerial and ground locomotion capabilities in an industrial environment and the first specifically designed for inspection and maintenance purposes in locations with very difficult access. The system will perform works at height that today are carried out manually with important risks of the persons and high operational costs. This system will be able to fly near to the intervention location, land on very constrained surfaces, such as pipes, and roll on the surface performing inspection and maintenance tasks and reaching sites not directly accessible by the flying system. Thus, it will be possible to perform manipulation task with a fixed base overcoming the accuracy limitations of manipulation with a floating base and decreasing the energy consumption allowing to increase the duration of the missions, which is very constrained in aerial manipulation.

The scientific and technological objectives of ARM-EXTEND are the following:

- Development of the first aerial robotic manipulator with both aerial and ground locomotion capabilities increasing accuracy in manipulation (millimitre accuracy) and improving drastically the duration of the missions of the aerial manipulation systems based on multi-rotor which is only few minutes. The rotary wing configuration will have multiple rotors with tilted axis (vectorized thrust) to provide more agility overcoming the limitation of conventional multi-rotor platforms with parallel axis.
- Develop autonomous landing on pipes and other constrained surfaces by using 3D environment perception system with both visual and laser sensors. The landing system will benefit from the agility of the multi-rotor with tilted axis.
- Manipulation by using light robotic arms overcoming the accuracy and dexterity that can be achieved while the robot is flying without any contact point. Thus, the manipulator will be able to perform wall thickness measurements and installation of permanent sensors.
- Collaborate with other aerial robots in inspection and maintenance activities. Thus, a fleet of aerial robotic inspectors will be able to detect and localise leaks and determine the locations to be inspected by the aerial robotic manipulator to perform wall thickness and other measurements requiring contact.

The above objectives are complex but can be achieved by using the very relevant previous results of the research team that has pioneered the aerial robotic manipulation and play a leadership role in this area and its applications to inspection. The ARM-EXTEND system will be able to perform contact and non-contact inspection in energy generation plants decreasing dramatically maintenance costs and avoiding contamination due to leakages or even explosions due to gas losses. The project will perform experiments in pipe inspection an maintenance, which is a very relevant activity with a huge economic impact.

This thesis has been created in the context of structure assembly and inspection, with the intention of researching different ways of giving the task planner all the knowledge needed to tackle the routing problems encountered in assembly and inspection operations.

2 Automated Planning Background

The general problem of finding a solution to a given task has been tackled mainly following three approaches. First, programming-based approaches count on programmers to encode the method to solve a problem. In learning-based approaches, it is the program that improves itself by learning the adequate solution, and this can be done by trial-and-error or based on the information provided by an instructor. Model-based methodologies rely on a general program which infers automatically a solution, starting from a suitable description of the actions, sensors, and goals.

Automated Planning, sometimes denoted as simply planning, is a model-based approach to autonomous behaviour, and is a discipline of Artificial Intelligence that studies the different aspects of what makes intelligent a behaviour. The ability to correctly reason about the actions to perform before acting is certainly a central point to define intelligence, and is the main focus of automated planning.

Automated planning concerns the realization of strategies or action sequences, typically for execution by intelligent agents, autonomous robots and unmanned vehicles. The solutions in automated planning are complex and are found usually in a multidimensional space. Automated planning involves a wide range of research fields. The central problems include reasoning, knowledge, task planning, learning, natural language processing, perception and the ability to move and manipulate objects. Approaches include statistical methods, computational intelligence, machine learning and traditional symbolic planning. For robots to solve real-world tasks, they need to reason about both symbolic actions and continuous-valued paths in the geometric world. Indeed, one of the main issues of symbolic task planning is its difficulty to reason about the geometry of the environment on which the action takes place.

This chapter presents how symbolic planning may be connected to its geometric counterpart in the world. This inevitably brings us to the Motion Planning field and all its related work. As the goal of this thesis is the application of HTN planning in the resolution of vehicle routing problems for structure assembly and inspection domains, the geometric reasoning in this study has been modelled at a high level of abstraction, using cartesian coordinate systems for locations, graphs with costs for navigation maps, euclidean distances and so on. Although this thesis is not centred in Motion Planning, understanding the related literature to know the different techniques that have been applied to connect symbolic and motion planners gave us some ideas to fill the gap between our symbolic and geometric levels.

In this chapter we discuss works related to the general symbolic task planning problem and systems that integrate symbolic and geometric reasoning. Thus a short classification of the methods usually applied to tackle this problem is presented.

2.1 Plan Search Algorithms

Solving classical planning problems can be cast as a path-finding problem in a directed graph whose nodes represent states, and whose edges represent state transitions due to actions. The whole graph is usually called the State Transition Graph [73]. Classical planning problems can then be solved by using graph search algorithms to find a path from the initial state to a goal state. This graph search approach is not trivial because the size of the graph may grow exponentially with the size of the planning problem.

Some search algorithms find new states generating a search tree without using any domain specific knowledge. The algorithm does not have any additional information about the states beyond the problem definition, no information is used to determine the preference of one child over other when discovering and expanding nodes on the search tree, thus looking the total search space for the solution. This kind of strategy is called *uninformed search*. Some examples of popular algorithms that follow this strategy are the **Breadth-first** and **Depth-search** algorithms [73]. Breadth first uses a First-In First-Out (FIFO) queue to put all new nodes that have been discovered and are candidates to be expanded in the search tree, thus selecting states using the *first-come first-serve* principle. All plans that have k steps are exhausted before plans with $k + 1$ steps are investigated, guaranteeing that the first solution found will use the smallest number of steps. If instead of using a FIFO we use a LIFO (Last-In, First-Out) queue, then an aggressive exploration of the state transition graph occurs. This variant is the depth-first search algorithm, which dives quickly into the search graph. In that case the preference is toward investigating longer plans very early, as opposed of looking first to shorter plans.

An approach that has been proved to be effective relies on the use of heuristics. Heuristic search uses heuristic functions to evaluate the cost-to-go from a node to a goal, or more generally, to provide a ranking of nodes in order of their relative desirability [96]. This estimation of the distance in the search space is then used by the search algorithm to drive the state-space search, preferring to visit nodes considered more promising from their heuristic value. As stated in [8], planning with heuristic search is sound and complete by construction, as far as the used search algorithm is complete, given that the state space contains exactly all the possible plans as paths from the initial state to any goal state. This strategy is called *informed search*, and some popular algorithms that follow the strategy are **A*** [49, 50] and **Best-first** [73]. The A* search algorithm is an extension of Dijkstra's algorithm that tries to reduce the total number of states explored by incorporating a heuristic estimate of the cost to get to the goal from a given state, called *cost-to-go*. If that estimate is an underestimate of the true optimal cost-to-go for all states in the state space, then the algorithm is guaranteed to find the optimal plan. For Best-first search, the priority queue is also sorted according to an estimate of the optimal cost-to-go. The solutions obtained in this way are not necessarily optimal, it does not matter whether the estimate exceeds the true optimal cost-to-go. Although optimal solutions are not found, in some cases fewer vertices are explored which results in faster running times.

The search in the state space can be directed forward (progression search) or backward (regression search). Both search techniques must handle the exponential growth of the state space with respect to the number of actions. In some scenarios, backward search can use partially uninstantiated actions and keep the branching factor lower, but still requires further heuristics for good efficiency. The general approach to a heuristic search is to apply domain-independent relaxations. Standard ways to relax actions are to ignore some or all of their preconditions, or to ignore effects that delete literals, as proposed by [57]. Other heuristics involve abstracting states by decomposition of the goal into independent subgoals [107]. Another popular approach is the generation of a planning graph, which is a subset of the full state transition graph and allows reachability analysis to guide the search. A description of planning graphs can be found in [6].

Another way of efficiently drive the state-space search is the use of *metaheuristics*. In computer science and mathematical optimization, a metaheuristic is a higher-level procedure designed to find, generate or select a heuristic that may provide a good enough solution to an optimization problem, especially with incomplete or imperfect information or limited computation capacity. Metaheuristics sample a set of solutions which is too large to be completely sampled. They may make few assumptions about the optimization problem being solved, and so they may be usable for a variety of problems. Compared to optimization algorithms and iterative methods, metaheuristics do not guarantee that a globally optimal solution can be found on some problems, and in fact they may stuck on a local minimum or maximum. Many metaheuristics implement some form of optimization so that the solution found is dependent on the set of random variables generated. In combinatorial optimization, by searching over a large set of feasible solutions, metaheuristics can often find good solutions with less computational effort than optimization algorithms, iterative methods, or simple heuristics. As such, they are useful approaches for optimization problems [4]. Several books and survey papers have been published on the subject [43, 40, 116] but most literature on metaheuristics is experimental in nature, describing empirical results based on computer experiments with the algorithms. Some formal theoretical results are also available, often on convergence and the possibility of finding the global optimum [7].

The properties of metaheuristics can be summarized in the following:

- Metaheuristics are strategies that guide the search process.
- The goal is to efficiently explore the search space in order to find near–optimal solutions.
- Techniques which constitute metaheuristic algorithms range from simple local search procedures to complex learning processes.
- Metaheuristic algorithms are approximate and usually non-deterministic.
- Metaheuristics are not problem-specific.

Metaheuristics will be covered in greater detail on Chapter 4.

2.2 Planning Software

Through the use of Artificial Intelligence planning, humans can program computers to automatically formulate a plan, given a problem description and its related data as input. Research in automated planning started in the late 1960s. From the early days on, automated planning was motivated by and applied to controlling autonomous robots. The first intelligent mobile robot, *Shakey*, was demonstrated to the public in 1969 [98] and could navigate itself, avoiding obstacles and pushing objects to achieve a given task. Both its automated planner and its pathfinding components were seminal to research in symbolic planning and robot motion planning. Their combination may be considered as the first task and motion planning system. The Shakey system included Fikes and Nilsson's STRIPS planner [33], which is regarded as the first major planning system. As an automated planner, it solves a symbolic task, defined by a goal criterion, and generates a valid sequence of actions which fulfils preconditions and entail effects. Using a symbolic planner the Shakey robot could plan discrete motion, such as moving from one room to another while respecting discrete geometric constraints, such as the connectivity of rooms. After that, symbolic actions were refined by a path planner and executed by the mobile platform.

Since then and in contrast to traditional planning representations, more expressive languages were proposed. An example is the *situation calculus language*, as introduced by [81] and refined in [105]. Situation calculus is designed to model linear time with branching situations, and exhibits some features of a second-order language. As an example, the action language *Golog*, which is based on the situation calculus, has been applied to multi-robot task planning [31]. However, in situation calculus planning the efficiency to solve practical problems is considerably sacrificed for the expressiveness of problem definitions [107]. Other generalizations include temporal planning, where actions require a certain time, concurrent planning, where multiple actions may be performed simultaneously, or probabilistic planning, where the state can only be observed up to an uncertainty.

Besides the already mentioned STRIPS formulation for domain descriptions, PDDL [82] gained popularity and wide acceptance among planning software, especially through the International Planning Competition (ICAPS). PDDL separates the domain description with its action schema from the problem definition, which includes initial and goal states for a particular instance. Its core syntax is stable and a common format for software planners. In addition, many extensions and variants were proposed to accommodate more expressive features, for instance numerical fluents and multiple agents.

Among planning software systems, several implementations have been applied to robotics tasks. The *Fast Forward* planner (FF) by Hoffmann and Nebel [57] was the first to combine a hill-climbing search with a goal distance heuristic. The heuristics of FF and its variants have been proven to be particularly efficient for practical benchmark problems [107]. Similar to FF, the *Fast Downward* planner (FD), developed by Helmert [54], progresses the search following several heuristics. However, it translates the problem to a multi-valued task planning representation instead of operating in a conventional propositional representation. The heuristics of the FF planner have been adapted to work in a robotics task and motion planning system for multi-object manipulation [37]. With interface implementation and symbolic mapping of continuous variables, multi-object robot manipulation can directly use domain-independent planners, such as FF and FD planner, as demonstrated by [115]. Other planners are dedicated to hierarchical task planning, where larger-scale domains can be handled when a domain-specific hierarchy of actions is provided, instead of a classical STRIPS domain. An implementation that has been applied to robot task planning is the Simple Hierarchical Ordered Planner (SHOP), which performs ordered task decomposition. SHOP was described and made available by Nau et al. in 1999 [93], and has been applied in several hierarchical task and motion planning systems since, for instance Bidot's forward-chaining combined task and path planner [5].

While the above-mentioned task planners apply general-purpose artificial intelligence planners, several integrated task and motion planning systems use domain-specific planners, and many of the discrete search schemes directly encode a fixed action schema rather than separating the domain description from the code. As an example, Kaelbling and Lozano-Pérez belief-space hierarchical planner [61, 63] includes a domain-tailored backtracking search that directly evaluates actions implemented in a scripting language. One of the first hybrid task and motion planning systems, the *aSyMov* planner by Gravot, Cambon, and Alami [44] included a hand-crafted search, and was later revised to a version that can parse more generic domain descriptions [12].

2.3 Ways to Integrate Symbolic and Geometric Reasoning

One of the problems of symbolic task planning is the difficulty to reason about the geometry of the world that many task planners have. To solve complex problems that require some knowledge about the geometry of the environment the planner usually has to relax some of the preconditions that are present in the problem domain. Different approaches from diverse research directions have been proposed since years ago to combine symbolic and geometric reasoning. In this section we aim to briefly categorize related works with respect to their general approach, search strategy and main characteristics.

Along the years, the problem of combining symbolic and geometric reasoning came from the need of efficiently combine task and motion planners. Although our work is not focused specifically on motion planning, we rely and reason on symbolic and geometric consequences and how they interfere one with each other, so the different connections between task and motion planners are of our interest. The problem of combining symbolic and geometric reasoning has been given different names, such as 'manipulation planning' in [72], 'hybrid planning' in [46], 'Combining Planning and Motion Planning' (CPMP) from [14], 'Task And Motion Planning' (TAMP) in [77] and its variants [97, 52, 17, 70].

Task and motion planners perform searches through symbolic and geometric spaces. Search schemes may be progressions directed forward from a starting configuration to a goal [44, 114, 25, 103], regressions directed backwards from a goal to the start [61], or may do recursion from both start and goal [3]. Some task and motion planners are built on general-purpose automated planners and allow generic domain definitions and multiple types of actions [114, 12, 24, 103], for which the Planning Domain Definition Language

(PDDL) is a frequent choice [12, 24]; others use domain-specific planners, where adding new types of actions would require manual implementation [61, 63].

An important strategy for a task and motion planner is how to explore discrete and continuous search spaces, how to sample in the continuous space, and how to coordinate search in both spaces. In order to categorize search strategies, one can distinguish between symbolic searches that are mostly refined by a geometric planner [10, 114, 27] and geometric motion planners that are guided to fulfil a certain symbolic task [103, 53]. The simplest solution is to plan only symbolical actions and later refine these by motion planning, as in the Shakey system [99]. Gravot, Cambon, and Alami [12] realized that a tighter integration is required to achieve completeness, and search in the product space of discrete and continuous states. Kaelbling and Lozano-Pérez [62] proposed a hierarchical planner in the belief space, which can generate new symbols that represent geometric constraints. Several other robot task planners search in the symbolic space and interface a geometric planner to evaluate preconditions and apply effects, with the latter strategy proposed by Dornhege et al. [25]. Several other planners [114, 74, 5] also belong to this category.

In order to reduce the search space, some approaches propose to plan hierarchically to interleave planning and execution, or to generate semantic maps. Hierarchical task networks (HTNs) require a full definition of primitive actions and levels of compound tasks, up to a goal task. The HTN planning approach is taken by the state-abstracted HTN planner [123] and others [64]. Interleaved planning and execution may greatly reduce the search space, since it only needs to solve for the next action. Interleaved planning is proposed by several authors [48, 62], and is suitable for domains with sensor perception, where the robot should react to measurements. It may be complete for domains that contain only reversible actions, among other criteria [62]. On the task level, semantic maps can represent spatial relations and domain knowledge [35].

Through the years many different approaches have been proposed to combine symbolic and geometric reasoning. In the work done in [71] a good classification of the approaches is presented taking into account how the communication is done between the planners. Based on this classification, it is possible to distinguish three main categories:

- **Symbolic layer calls the geometric layer**. The symbolic planner performs the search and verifies the plan by asking the geometric planner about its feasibility.
- **Geometric layer calls the symbolic layer**. The geometric planner knows the possible solutions and uses the symbolic layer to determine which ones to further explore.
- **Sample in the compound state**. The search space is a compound space between the geometric and the symbolic spaces and the search is done simultaneously.

In the following subsections these categories are further explained.

2.3.1 Symbolic layer calls the geometric layer

Many developed approaches can fit inside this category. Some researchers propose to compute first all symbolic plans and then use the geometry level to find a feasible one.

Others try to compute first a complete symbolic plan and then test it in the geometric level, computing another complete symbolic plan if it is unfeasible. Another option is to call the geometric planner while the symbolic planner is computing the plan.

The work done in [16] introduce the concept of a task motion multigraph, a data structure that can be used to reveal the possibility of planning in different state spaces in order to achieve the same goal. The different options reflect the mobile manipulator's ability to use different hardware components to perform a required task, as for example, opening a door with the left arm or with the right arm, having thus two possible state spaces for planning. Given the specification of a task, it shows how to encode the available motion planning options in a task motion multigraph. In this case, from a list of possible plans they are able to find a feasible set of motions that fulfil the given symbolic goal. The authors have extended this work in [17] by introducing uncertainties and using Markov Decision Processes to guide the search. Hierarchical Task Networks have also been used in this respect. In [69] it is argued that the domain of humanoid robot manipulation has specific features which allow to delegate a part of the geometric reasoning to the task planner, enriching the planning domain with geometric representations. With this approach, the task planner generates alternative sequences of actions which lead to the same geometric result. They use a HTN where they broke the geometric actions into basic primitives, to find all the possible plans, and then the geometric reasoner is used to test the geometric feasibility of the plan.

One example of finding first a symbolic plan and then computing a geometrically feasible solution can be found on [77]. They describe a strategy for integrated task and motion planning based on performing a symbolic search for a sequence of high-level operations, such as pick, move and place, while postponing geometric decisions. In [115], once they find a task plan they try to plan the geometric actions, and if they fail an error is returned to update the symbolic state and a new task plan is computed. A different solution is proposed in [11] where a formal framework (the action description language C+) is used to provide robots with high-level reasoning, such as planning. They introduce a method that bridges the high-level discrete action planning and the low level continuous behaviour by trajectory planning. When they encounter a problem (collision) they report it to the reasoner and a new symbolic plan is computed where they try to extract the trajectories again. In [27] the authors keep nearly the same framework but use, in place of the action description language, a causal reasoner to find the symbolic plans. If the geometric resolution fails, it changes the planning problem by adding constraints to the causal reasoner in order to take the causes of failure into account.

Some studies ensure geometric feasibility while computing the symbolic plan. In [25] the authors decompose the manipulation problem into a symbolic and a geometric part. The symbolic part is implemented as a classical symbolic planner that tightly integrates a geometric planner enabling to generate correct plans. A probabilistic roadmap planner constitutes the geometric part. During the computation of the roadmap they utilize proximity queries to determine non-colliding configurations and to verify collision free paths between configurations. The same authors in [23] present a study about soundness and completeness of their approach in addition to multiple examples and relevant results. Hertle et al. [55] propose a new planning language, the Object-oriented Planning Language, where the task descriptions are written in an easy object-oriented-like form similar to C++

and which can handle semantic attachments. In addition to describing the domain, their language is also used to produce a domain-specific interface that allows the integration of external modules.

Approaches based on calls to external procedures can be found. In [32] the authors build a planner that makes use of an expressive variable-free, first-order planning language called Functional STRIPS where constraints, functions and numerical variables are accommodated, and extend it to handle also state constraints. They use functions for encoding the geometrical dimensions and poses of objects, and state constraints to express that no pair of objects, including the robot, can overlap in space. Gaschler et al. [39] also uses external calls at symbolic level combined with a detailed symbolic state of the world to compute feasible plans. They extend their work on [38] by adding specific geometric predicates to their actions, which results in a improvement of the search speed.

In relation with HTNs, [123] presents a hierarchical planning system and its application to robotic manipulation. The system is capable of finding kinematic solutions to task-level problems, taking advantage of subtask-specific information and reusing optimal solutions. They present results on discrete problems as well as pick-and-place tasks for a mobile robot. The GoDel planning system [109] tackle one drawback of Hierarchical Task Network planning, which is the difficulty of providing complete domain knowledge, i.e., a complete and correct set of HTN methods for every task. To provide a principled way to overcome this difficulty, the authors define a simple formalism that extends classical planning to include problem decomposition using methods, and a planning algorithm based on this formalism. The methods specify ways to achieve goals rather than tasks, as in conventional HTN planning, and goals may be achieved even when no methods are available.

2.3.2 Geometric layer calls the symbolic layer

In this category fall all the approaches that guide a geometric search with information that comes from queries to the symbolic layer. In [125] the search is done in the geometric state space and for each state, symbolic information is computed to guide the geometric search to the goal. An interesting approach is used by [14], where a motion planner is used to explore the world. The generated graph is used to check if any of its edges modify the state of any object, in which case the motion caused by the edge is considered to be an action. Later, a symbolic planner is used to find a plan with these actions. In the work done in [97] a synthesis algorithm complements continuous motion planning algorithms with calls to a Satisfiability Modulo Theories (SMT) solver. From the scene description given by the user, a motion planning algorithm is used to construct a placement graph, an abstraction of a manipulation graph whose paths represent feasible low-level motion plans. An SMT-solver is then used to symbolically explore the space of all integrated plans that correspond to paths in the placement graph and also satisfy constraints demanded by the user requirements. Aware of the fact that symbolic task planners can efficiently construct plans involving many entities but cannot incorporate the constraints from geometry and kinematics, the FFRob planner [37] shows how to extend the heuristic ideas from the FastForward (FF) planner [57] to motion planning, using a multi-query roadmap structure that can be conditioned to model different placements of movable objects. In [36] the same authors build a reachability graph from a simplified version of the problem in a backward fashion. From this graph they compute a heuristic that drives the forward search to the final solution. The search is a enforced hill climbing: they remember the minimum heuristic value and if a state with a lower value is reached then the queue of tasks is popped.

A different approach is done in [103] where the authors sample the continuous space guided by the symbolic level, until reaching a state which satisfies the goal previously given to the geometric planner. A tree is created and expanded at each iteration, choosing the more relevant node and exploring the space from there. The same authors expand this approach in [102] replacing the symbolic planner by an automata described using Linear Temporal Logic (LTL).

2.3.3 Sample in the compound state

In this last category the search is carried out in parallel at geometric and symbolic levels, using a compound state comformed by the union of the states of both levels. One of the first works on combining symbolic and geometric reasoning was the aSyMov planner [44]. The aSyMov planner was specially designed to address intricate robot planning problems where geometric constraints cannot be abstracted in a way that has no influence on the symbolic plan. The authors manage to establish an effective link between the representations used by a symbolic task planner and the representations used by a motion and manipulation planning library. At each step of the planning process both symbolic and geometric constraints are considered. The planning process tries to arbitrate between finding a plan with the level of knowledge it has already acquired, or investing more in a deeper knowledge of the topology of the different configuration spaces it manipulates.

A key problem in integrating task and motion planning is to deal with the fact that many motion planning queries are unfeasible. In high-dimensional configuration spaces no effective motion planning techniques exist to detect that a query is unfeasible. Probabilistic Road Maps (PRM) [66] planners can solve feasible queries efficiently but in variable running times. In [52] it is considered that robots can move inside a feasible space only, and can switch between 'feasible spaces' through transitions: inside a feasible space the robot cannot change his contacts with the outside world (as when the robot is moving an object) but can do it through a 'transition space' (by placing the object on a table). They create and use a PRM in each 'feasible space'. The work is further extended in [51] by creating a symbolic language able to make requests to their previous system, obtaining a larger range of possible actions. Similar methods are used in [3], applying a RRT algorithm instead of a PRM.

2.4 Conclusions

In this Chapter we have presented different plan search algorithms that are commonly used for solving classical planning problems. Metaheuristics have been also introduced as an efficient way to guide the search of plans. Several examples of planning software and systems from the automated planning literature have been also studied.

From the related work, the different ways to integrate symbolic and geometric reasoning researched and developed along the last years have been categorized. Three different categories have been clearly distinguished and identified: symbolic layer calls the geometric layer, geometric layer calls the symbolic layer, and sample in the compound state.

3 Hierarchical Task Network Planning

Hierarchies are one of the most common structures used to understand and conceptualize the world. In fact, the way the humans think when trying to accomplish an action is hierarchical. When someone needs to buy food because the refrigerator is running empty, the first thought is to go to the supermarket. After that, one thinks on getting into the car, drive to the supermarket, buy the food and return. As soon as each action on the plan is getting closer to its execution, they are further decomposed. For example, to get into the car one first need to dress, search and pick the car key and then exit home and go to the car. This way of thinking not only affects the way the humans act, it also affects the way the humans organize the world. Governments, military or police forces, enterprises and in general, the different entities of our world are all organized in a hierarchical structure of persons and positions. This organization is not casual, it is a consequence of the way the humans think.

Within the field of Artificial Intelligence planning, which deals with the automation of world-relevant problems, Hierarchical Task Network (HTN) planning is the branch that represents and handles hierarchies. In particular, the requirement for rich domain knowledge to characterize the world enables HTN planning to be very useful, and also to perform very well on computation terms.

In recent years, hierarchical task networks have emerged as a powerful framework for representing and organizing knowledge about actions. With access to such content, a robot can generate or execute plans far more effectively than it can from other classical approaches, because the knowledge specifies how to decompose complex tasks into simpler ones.

HTN approaches have been applied successfully to a variety of challenging domains, and in some cases they seem to scale to complexity much better than classical planning methods. Despite their clear advantages, hierarchical task networks can be difficult and time consuming to construct manually, as they need a specialized person that gives the initial domain knowledge.

This chapter presents the HTN planner on which our research is based. We start giving an introduction where some details in the history of HTN are presented: its origins, some of the most relevant HTN planners and practical applications. After that, the mathematical model for the HTN formalism is presented. Finally an overview of JSHOP2, the HTN planner used in our research, is presented.

3.1 Introduction

Automated planning systems can be classified into three categories, based on the amount of knowledge they need to be configured to work in different planning domains: domain-independent planners, domain-dependent or domain-specific planners and domain-configurable planners [92].

Domain-independent planning systems have as input a description of a planning problem to solve, and the planning engine is general enough to work in any planning domain that satisfies some set of simplifying assumptions. For nearly the entire time that automated planning has existed, it has been dominated by research on domain-independent planning. The main limitation of this approach is that it is not possible to develop a domain-independent planner that works efficiently in all kind of planning domains. Because of this difficulty, most research has focused on a set of classical planning domains.

Domain-dependent planners are planning systems that are customized for its use on a single specific planning domain and are unlikely to work in other domains unless major modifications are made to the planning system. In a domain-dependent planner, the domain specific information may be encoded into the planning engine itself. Such a planner can be quite efficient in creating plans for the target domain but will not be usable in any other planning domain. If the planner is needed to work with another domain, then it is necessary to build an entirely new planner.

Domain-configurable planners are planning systems in which the planning engine is domain-independent but the input to the planner includes domain-specific knowledge to constrain the planner's search so that the planner searches only a small part of the search space. In domain-configurable planners, the planning engine is domain independent but the input to the planner includes a domain description, that is, a collection of domain-specific knowledge written in a language that is understandable to the planning engine. The planning engine can be reconfigured to work in another problem domain by giving it a new domain description. A special type of domain-configurable planners are HTN planners.

Hierarchical Task Network (HTN) planning was first developed several years ago [108, 117]. The work done by Earl D. Sacerdoti in 1975 was the first on thinking in the non-linearity of plans, and introduced the concept of *procedural net* as a new information structure to represent a plan as a partial ordering of actions with respect to time, instead of the classical view of a plan as a linear sequence of actions that are executed one at a time. The procedural net was a network of nodes, with each node representing a particular action at some level of detail. The nodes were linked in a partially ordered time sequence by predecessor and successor links to form hierarchical descriptions of operations and to form plans of action, so that nodes at each level of the hierarchy conformed a sequence that represented a plan at a particular level of detail. The developed NOAH planner [108], whose name stands for Nets of Action Hierarchies, was the first on using a non-linear representation of plans.

One year later and based on NOAH, the NONLIN [117] hierarchical partial-order AI planning system was developed by Austin Tate at the University of Edinburgh. It was developed to work on a project aimed at producing an interactive program for the construction of project networks, such as in house building tasks. The NONLIN system was a development made over the research done with NOAH, but improved the planner in several ways. It introduced a task formalism as a powerful and flexible language for the users to describe domains. Also, it had the capability of generating and storing all the alternatives at the choice points of the search for a future use, either on the fail of some approach tried by the planner or to generate more than one solution. Some other improvements were done but, regardless of that, the works of Sacerdoti and Tate supposed the beginning of HTN planning.

HTN planning is an Artificial Intelligence planning technique that differs from the main ideas conceived in classical planning. A task network is a hierarchy of tasks where each can be executed if the task is simple enough, or decomposed into lower-level subtask networks otherwise. HTN planning includes an initial state description, an initial task network that has to be decomposed and that represents the goal to achieve, and a domain knowledge consisting on how to decompose the task networks. The planning process starts by decomposing the goal task network, reducing it into several subtask networks until all compound tasks are decomposed and thus, a solution is found. The solution consists on a set of non-decomposable primitive tasks that are applicable to the initial world state.

The main characteristic of HTN planning is the need of a well-structured domain knowledge to decompose the task networks into smaller subtask networks. This knowledge contains all the information needed to solve a specific planning problem, and represents a guide on how it must be solved. Such knowledge encodes much more information than that needed and present in other classical planning techniques and has its advantages and disadvantages [92]. Mainly, this knowledge gives a boost in terms of performance and coverage across many domains to HTN planners compared to classical planners, but the domain formalization is more complex and it requires an expert person who knows how to solve the planning problem, so the human effort required to configure the planner is high.

Most of the HTN planning research have been focused on practical applications [28, 58]. Examples include production-line scheduling [121], planning and scheduling for spacecraft [30, 2] and evacuation planning [86]. Many other real-world applications have been developed by using the natural knowledge-modelling framework of HTN, including military planning [85, 87], manufacturing processes [93, 111] and even interactive dialogue generation [13].

An especially interesting field of application for HTN planning is strategy formulation and character behaviour in computer games due to real-time or fast response requirements, in addition to the generation of human-like demeanour. In [95] a modified version of HTN structures was used to represent multi-agency and uncertainty in the bridge card-playing game, applying HTN decomposition to produce a game tree on which each branch represents a move that fits into some coherent strategy. Another example of using HTN representations to model strategic game AI can be found in [56], where HTNs were used to model effective team strategies for bots while finite state machines were used to encode individual bot behaviour. This allowed the bots to react properly in a highly dynamic environment while contributing to the team task. The SHPE (Simple Hierarchical Planning Engine) planner [83] is a hierarchical task network planning system designed to generate dynamic behaviours for real-time video games, being able to return relevant plans in few milliseconds for several problem instances of the presented planning domain. On the way of the creation of virtual autonomous characters that are lifelike and intelligent, a strategic planning architecture is presented in [78] in the domain of fire fighting and emergency response. This architecture used a visual perception system and one HTN planner to generate an online plan based on the information gathered from the visual perception system, generating a believable non-player character behaviour.

As commented before, the NOAH and NONLIN were the two first developed HTN planners but they have been followed by many others HTN planners in the AI community. Among the most relevant we can find two categories: planners without monitoring features (re-planning) and planners with monitoring features.

The UMCP (Universal Method Composition Planner) [29] is a pure planning system, implemented in Lisp. UMCP has been proven to be sound and complete and offers both and automatic (or interactive) search and a graphical user interface to navigate through the search space. Years later of its development, the same authors developed the SHOP2 [94] planner, a partially-ordered domain-independent planning system. SHOP2 starts its planning process from a problem file which contains the initial state of the world as well as the tasks that must be accomplished. It needs to know how the decomposition of tasks has to be done, so an additional file must be given containing this information. SHOP2 generates the steps of each plan in the same order that those steps will be later executed, so it knows the current state of the world at each step of the planning process. This reduces the complexity of reasoning by eliminating a great deal of uncertainty about the world, thereby making it easier to incorporate substantial expressive power into the planning system. A multi-agent version of SHOP2, called *A-SHOP* [21] has also been developed. It is an integration of the SHOP HTN planning system with the IMPACT multi-agent environment [22]. The A-SHOP algorithm is an agentized adaptation of the SHOP planning algorithm that takes advantage of IMPACT's capabilities for interacting with external agents, performing mixed symbolic/numeric computations, and making queries to distributed heterogeneous information sources, such as specialized data structures or external databases.

O-Plan2 [118] (Open Planning Architecture) provides a generic domain-independent computational architecture suitable for command planning and execution applications. The main contribution of O-Plan2 has been a complete vision of a modular and flexible planning and control system incorporating artificial intelligence methods. It offers an architecture not only for planning but also for scheduling, controlling and monitoring the actions. Another example of a HTN-based plan generation and execution system is SIPE-2 [122], a performance-oriented, general-purpose software system for generating and monitoring the execution of plans. SIPE-2 plans hierarchically using different levels of abstraction, and provides a formalism for describing actions as operators. Given an arbitrary initial situation and a set of goals, SIPE-2 either automatically or under interactive control combines operators to generate plans to achieve the prescribed goals in the given world. SIPE-2 includes heuristics for reducing computational complexity, and is capable of generating a novel sequence of actions that responds to the current situation: it has execution-monitoring techniques that accept new information about the world and modify the plan minimally to respond to unexpected events. SIADEX [19] is a complex framework that integrates several AI techniques able to design fighting plans against forest fires, also allowing to respond to the uncertainty in the execution of a plan by offering plan repairing (replanning) and revision procedures. It is based on four main components, a web server that centralizes all the flow of information between the system and the user, the ontology server that is the basis for knowledge sharing and exchange between all the components, and the planning and monitoring servers that are offered as intelligent services through the web server.

3.2 HTN Formalism

3.2.1 Mathematical Model

The formalism presented here describes the mathematical model [41] of HTN planning, composed of a planning language, operators, methods, tasks, task networks, planning problem and solution.

The HTN planning language is a first-order language that contains several mutually disjoint sets of symbols. A *predicate*, which may takes as values *true* or *false*, consists of a predicate symbol $p \in P$, where P is a finite set of predicate symbols, and a list of terms $(\tau_1,...,\tau_k)$. A *term* is either a constant symbol $c \in C$, where C is a finite set of constant symbols, or a variable symbol $v \in V$, where V is a finite set of variable symbols. The set of predicates is denoted by Q. A predicate is *ground* if its terms do not contain variable symbols.

In formal systems of logic for knowledge representation, the *Closed-World Assumption* (CWA) is the presumption that a statement that is true is also known to be true and conversely, what is not currently known to be true is false. For our formalism, a *state* $s \in 2^Q$ is a set of ground predicates in which the CWA is adopted, so only all the predicates that are true are specified in a state. A *primitive task* is an expression $t_p(\tau)$ where $t_p \in T_p$ and T_p is a finite set of primitive task symbols, and $\tau = \tau_1,...,\tau_k$ are terms. A primitive task is represented by a *planning operator*.

Definition 3.2.1 (Operator) An operator o is a triple $(p(o); pre(o); eff(o))$, where $p(o)$ is a primitive task, $pre(o) \in 2^Q$ are the preconditions to apply the operator, and $eff(o) \in 2^Q$ are the effects of applying the operator. The subsets $pre^+(o)$ and $pre^-(o)$ denote the positive and negative preconditions of o, respectively. In addition, the subsets $eff^+(o)$ and $eff^-(o)$ denote the positive and negative effects of applying o, respectively.

A transition from one state to another is accomplished by applying an instance of an operator whose precondition is a logical consequence of the current state. An operator o is said to be *applicable* in the state s, if $pre^+(o) \subseteq s$ and $pre^-(o) \cap s = \emptyset$. Applying o to s results in the state $s[o] = (s \setminus eff^-(o)) \cup eff^+(o)$. From now, the notations $s[o] = s'$ and $s \xrightarrow{o} s'$ will be used interchangeably.

A *compound task* is an expression $t_c(\tau)$ where $t_c \in T_c$ and T_c is a finite set of compound-task symbols, and $\tau = \tau_1,...,\tau_k$ are terms. We denote the set of task names as T_n, which is composed by the union of T_p and T_c.

Definition 3.2.2 (Task Network) A task network tn is a pair (T,ϕ), where T is a finite set of tasks, and ϕ is a finite set of constraints.

The constraints in ϕ specify restrictions over T that must be satisfied during the planning process to decompose the task network. A task network over T_p is a *primitive task network*. The set of all task networks over T_n is denoted as TN.

Definition 3.2.3 (Method) A method m is a pair $(c(m),tn(m))$, where $c(m)$ is a compound task, and $tn(m)$ is a task network to decompose $c(m)$.

Definition 3.2.4 (Planning Problem) A planning problem P is a tuple (Q,T_p,T_c,O,M,tn_0,s_0), where:

- Q is a finite set of predicates
- T_p is a finite set of primitive task symbols
- T_c is a finite set of compound task symbols
- $O \subseteq T_p \times 2^Q \times 2^Q$ is a finite set of operators
- $M \subseteq T_c \times TN$ is a finite set of methods
- tn_0 is the initial task network
- s_0 is the initial state .

An operator sequence $o_1,...,o_n$ is *executable* in s, if there are states $s_0,...,s_n$ such that $s_0 = s$, o_i is applicable in s_{i-1} and $s_{i-1}(o_i) = s_i$ for all $i \leqslant n$. Given a problem P, a *solution* to P is an operator sequence executable in s_0 by decomposing tn_0.

Given the previous definitions, the work-flow for an HTN planner would be the following: the input to the planner consists on an initial task network tn_0 representing the problem to be solved, along with an initial state s_0. The task network is a finite set of tasks T restricted by a finite set of constraints ϕ that represents some action that needs to be done. Each task on the network can be primitive, meaning that it can be performed directly, or compound, meaning that the planner needs to figure out how to perform it. The set of operators O tell the effects of executing each primitive task while the set of methods M tell how to perform compound tasks. Each method m pairs a compound task $c(m)$ with a task network $tn(m)$, and tells how to achieve the compound task by performing the tasks specified in the network, supposing that this can be done in a way that satisfies all the constraints $\phi(tn_0)$ of the task network.

As commented before, the planning process starts with the initial task network tn_0 and the initial state s_0, and repeats the following steps: until no compound tasks are left, it finds a method m in M and a compound task c_i in tn_0 such that $(c_i(m),tn'(m))$, i.e. a method that pairs c_i with tn'. Then it modifies tn_0 by replacing $c_i(m)$ with the tasks in $tn'(m)$ and adding the constraints $\phi(tn')$ into $\phi(tn_0)$.

3.2.2 Search Space

In the mathematical model presented before, the main concepts that describe the HTN planning formalism have been defined. All these concepts affect the structure of the space on which the search takes place. There are two possible structures of search spaces that can be created by HTN planners: *plan space* and *state space*.

The first type of search space, the *plan space*, consists of task networks and task decompositions as evolutions from one task network to another. Given some planning problem, at the beginning of the search, a task decomposition is imposed on the initial task network, and the process continues by repeatedly decomposing tasks from a newly created task network until a primitive task network is produced. A *linearisation* of this primitive task network executable in the initial state represents a solution to the planning problem.

The second type of search space is called the *state space*. It consists on explicitly described states restricted by task decompositions. The search begins in the initial state with an empty plan, but instead of searching for a state that will satisfy the goal, the search is for a state that will accomplish the initial task network. In particular, if a task from the task network is compound, a task decomposition is performed and the search continues on the next decomposition level, but in the same state. If the task is primitive, it is executed and the search continues into a successor state. This task is then added to the plan. When there are no more tasks in the task network to be decomposed, the search has finished. The solution to the planning problem is the plan containing a sequence of *totally ordered* actions.

We refer to HTN planners that search in the plan space as *plan-based HTN planners*, and to the model of HTN planning as *plan-based HTN planning*. The term state space is used to refer to the second type of search space. Thus, we refer to HTN planners searching in this space as *state-based HTN planners*, and to the model of HTN planning as *state-based HTN planning*.

3.2.3 Plan-Based HTN Planning

To formalize the plan-based HTN planning, Definition 3.2.2 must be further complemented and also some new concepts must be introduced.

Definition 3.2.5 (Task network) A task network tn is a triple (T, φ, ψ), where

- T is a finite and non-empty set of tasks
- $\varphi : T \to T_n$ labels a task with a task name
- ψ is a formula composed by conjunction, disjunction of negation of the following set of constraints:
  - $\prec \subseteq T \times T$ is a strict partial order on T (irreflexive, transitive, asymmetric)
  - $\mapsto \subseteq V \times V \cup V \times C$ is a restriction on bindings of task network variables
  - $O \subseteq T_p \times 2^Q \times 2^Q$ is a partial order on tasks and state predicates.

Definition 3.2.6 (Decomposition) Let m be a method and $tn_c = (T_c, \varphi_c, \psi_c)$ be a task network. Method m decomposes tn_c into a new task network tn_n by replacing task t, denoted

as $tn_c \xrightarrow{t,m}_D tn_n$, if and only if $t \in T_c$, $\varphi_c(t) = c(m)$, and there exists a task network $tn' = (T',\varphi',\psi')$ such that $tn' \equiv tn(m)$ and $T' \cap T \neq \emptyset$, and
$tn_n := ((T_c \backslash t) \cup T', \varphi_c \cup \varphi', \psi_c \cup \psi' \cup \psi_D)$ where
$\psi_D := \{(t_1,t_2) \in T_c \times T' \mid (t_1,t) \in \prec_c\} \cup \{(t_1,t_2) \in T' \times T_c \mid (t,t_2) \in \prec_c\} \cup$
$\{(p,t_1) \in Q \times T' \mid (p,t) \in \vdash_{\prec_c} \cup \{(t_1,p) \in T' \times Q \mid (t,p) \in \vdash_{\prec_c}\} \cup$
$\{(t_1,p,t_2) \in T' \times Q \times T' \mid (t,p,t_2) \in \vdash_{\prec_c}\}$.

Given a planning problem P, $tn_c \rightarrow^*_D tn_n$ indicates that tn_n results from tn_c by an arbitrary number of decompositions using methods from M.

Definition 3.2.7 (Executable Task Network) Given a planning problem P, a task network $tn = (T,\varphi,\psi)$ is *executable* in state s, if and only if it is primitive and there exists a linearisation of its tasks $t_1,\ldots,t_n$ that is compatible with ψ and the corresponding sequence of operators $\varphi(t_1),\ldots,\varphi(t_n)$ is executable in s.

Finally, a *solution* for a HTN planning problem can be formalized.

Definition 3.2.8 (Solution) A task network tn_s is a solution to a planning problem P, if and only if tn_s is executable in s_0, and $tn_0 \rightarrow^*_D tn_s$ for tn_s being a solution to P.

Definition 3.2.9 (Plan Space) Given a plan-based HTN planning problem P, a plan space PG is a directed graph $(V;E)$ if and only if $tn_0 \in V$, and for each $tn \rightarrow_D tn' : tn,tn' \in V$ and $(tn,tn') \in E$.

3.2.4 State-Based HTN Planning

To formalize state-based HTN planning, Definition 3.2.2 and Definition 3.2.3 must be further complemented while some concepts that were defined for plan-based HTN planning must be reformulated.

Definition 3.2.10 (Task network) A task network tn is a pair $(T, \prec)$, where

- T is a finite set of tasks
- $\prec$ is a strict partial order on T (irreflexive, transitive and asymmetric).

Definition 3.2.11 (Method) A method m is a triple $(c(m),pre(m),tn(m))$, where $c(m)$ is a compound task, $pre(m) \in 2^Q$ is a precondition, and $tn(m)$ is a task network. The subsets $pre^+(m),pre^-(m)$ denote the positive and negative precondition of m, respectively.

A method m is applicable in state s, if and only if $pre^+(m) \subseteq s$ and $pre^-(m) \cap s = \emptyset$. Applying m to s results in a new task network.

Definition 3.2.12 (Decomposition) Let m be an applicable method in s and $tn_c = (T_c, \prec_c)$ be a task network. Method m decomposes tn_c into a new task network tn_n by replacing task t, written $tn_c \xrightarrow{s,t,m}_D tn_n$, if and only if $t \in T_c, t = c(m)$ and
$tn_n := ((T_c \backslash \{t\}) \cup T_m, \prec_c \cup \prec_m \cup \prec_D)$ where
$\prec_D := \{(t_1,t_2) \in T_c \times T_m \mid (t_1,t) \in \prec_c\} \cup \{(t_1,t_2) \in T_m \times T_c \mid (t,t_2) \in \prec_c\}$.

Finally, the state space can be defined as follows.

Definition 3.2.13 (State Space) Given a state-based HTN planning problem P, a state space SG is a directed graph (V,E) if and only if $s_0 \in V$, and there is a state s_i and $t_k \in tn$ such that

- if t_k is primitive, then $s_i \xrightarrow{t_k} s_{i+1}$ such that $k = i+1, s_i, s_{i+1} \in V$ and $(s_i, s_{i+1}) \in E$; or
- if t_k is compound, then $tn \rightarrow_D tn'$ is a self-transition such that $s_i \in V$ and $(s_i, s_i) \in E$.

3.3 JSHOP2 HTN Planner

As commented before in 3.2.2, the search space in HTN planning can be classified into two different types, the *plan space* and the *state space*. Based on this, HTN planners can also be classified in *plan-based HTN planners* or *state-based HTN planners*. From the work done in [41], it can be seen that plan-based HTN planners need to search more complex spaces than state-based HTN planners, which also affects some of the concepts used during the search in terms of the number of needed techniques, their technical complexity and their interconnection. Plan-based HTN planners seem to require smaller domain knowledge than state-based HTN planners, but it appears that both categories of planners have similar levels of practical expressiveness.

Due to the complexity of the problems to solve in this thesis, it is important the use of a planner whose search space is the less complex as possible, as it will reduce computation times. Also, the amount of domain knowledge needed by the planner is not important in our case, as this knowledge is designed and given beforehand and does not affect the performance of the planning process. For these reasons, state-based HTN planners are more suitable for our problems. Specifically, the JSHOP2 HTN planner has been chosen to solve the problem, as from all the state-based HTN planners it is one of the most applied in the research community [41].

3.3.1 Planning Domain Definition Language: The Origin

The Planning Domain Definition Language (PDDL) is an attempt to standardize Artificial Intelligence planning languages. Inspired by the STRIPS [33] and ADL [101] languages, it was first developed by Drew McDermott in 1998 to make possible the 1998/2000 International Planning Competition (IPC). Since then, the language has evolved with each competition, resulting on its different official versions.

The International Planning Competition is an event organized every two years in the context of the International Conference on Planning and Scheduling (ICAPS). Among its goals are to: analyse and advance the state-of-the-art in automated planning systems, provide new data sets for the research community to be used as benchmarks for evaluating different approaches to automated planning, or emphasize new research issues in planning, among others. For the competition, different planners (among which SHOP2 was present in previous sessions) are registered to try to solve the problems proposed by the organization, producing different solutions that are later evaluated by following some criteria such as the quality of the solution, its optimality, the computation time, etc.

Due to the needs of having each of the registered planners being capable of solving the same problems, a formal language definition was needed. The development of PDDL supposed the adoption of a common formalism for describing planning domains. This adoption has improved the reuse of research and allows a direct comparison of different systems and approaches of implementations, fastening the progress in the field.

PDDL has become a standard encoding language to model classical planning problems. Its syntax, inspired by the Lisp programming language, is mainly composed by logical expressions and relations. The main components of a problem modelled with PDDL are the following:

- Objects: the things in the world that are of interest for the problem.
- Predicates: the logical properties of objects that are of interest, and which can take *true* or *false* as values.
- Initial state: the state of the world where the problem starts in, given by the initial values of the predicates. Those initial values for the predicates are also called *facts*.
- Actions: the different ways of changing the current state of the world, the actions that are applicable over the objects and that result in some effect that changes the value of some predicates.
- Goal specification: the predicates that must be true to consider the problem as solved.

The model for a planning problem is defined in PDDL by using two separated files: the domain file and the problem file.

The domain file is used to model the world. It is composed of the different predicates that can be present in a planning problem. The different actions that are applicable over those predicates are also defined. An important feature of a domain file is that it is shared by all the problem instances of a planning problem, so once specified this file remains unchanged.

The problem file is used to create an instance of a planning problem. Inside the problem file are defined the objects that are present in that specific problem instance, along with its initial state and the goal specification. The domain file needed to solve the problem is also specified.

All the official versions of PDDL are composed by the same basic defining elements, but big changes have been done since the release of the first official version. These changes have been focused in increasing the expressive power of the language and some of them have finally made the language suitable for expressing time-based domains, expanding greatly the type of problems that the language can manage. However, it must be noted that PDDL is used to model planning problems and not to solve them, so the implementation of a planner capable of solving all the problems that PDDL can express is not trivial. In fact, many of the existing planners are only capable of interpreting a subset of expressions for a specific version of PDDL, and some others even have a custom variant for the language.

The main characteristics of the different versions that have been published since the release of the first official specification of the language are summarized below:

- PDDL 1.2: this was the first official release and the one which was used on the first two IPC competitions in 1998 and 2000. It defined the basic elements of the language (objects, predicates, initial state, actions and goals) and separated the model of the planning problem in the domain file and the problem file.
- PDDL 2.1: this version was used on the third IPC competition in 2002 and represented a great change from the previous version, bringing the language closer to more real-world problems. To model non-binary resources such us fuel-level, distance or weight, it introduced the concept of *numeric-fluents* to associate a numeric value to an object or tuple of objects. It also defined *plan-metrics* to allow quantitative evaluation of plans, oriented to optimization and metric minimization/maximization. Finally, the introduction of *durative actions*, which could have non-discrete length conditions and effects, allowed the language to manage the time variable in the domains.
- PDDL 2.2: this version was used on the fourth IPC in 2004 and extended the language with some important elements, although it was not a great evolution from the previous version. To model the dependency of given facts from other facts, such as in the transitive relation, it introduced the concept of *derived predicates*. Also, to model external events that occur at a given time independently from the plan execution, the concept of *timed-initial literal* was created.
- PDDL 3.0: used on the IPC competition of 2006, it introduced the concepts of state-trajectory constraints and preferences. The former represent hard-constraints which should be true for the state-trajectory produced during the execution of the plan, i.e. for the solution computed for a given planning problem. The latter define soft-constraints whose satisfaction is not necessary but that can take part into the plan metric to measure the quality of the plan.
- PDDL 3.1: this is the last version of the language and the one used on the IPC competitions from 2008 until today. It introduced the concept of *object-fluents* to associate an object to another object or to a tuple of objects.

3.3.2 JSHOP2 Overview

JSHOP2, the Java implementation of SHOP2 [94], is a sound and complete domain-independent planning system based on Hierarchical Task Networks. JSHOP2 starts its planning process from a problem file which contains the initial state of the world. This file defines all the entities that are present in the problem and their initial state, as well as the tasks that should be accomplished. The objective of JSHOP2 is the decomposition of these tasks into smaller subtasks, until obtaining primitive tasks which are the actions at the lowest level that compose the final plan. JSHOP2 needs to know how this decomposition has to be done, so an additional file should be given containing the domain of the problem. In this domain the structures that represent the high level tasks, called methods, are defined, as well as the structures that represent the primitive tasks, called operators. The methods contain the list of subtasks on which they can be decomposed, either another methods or operators. Thus, for solving a planning problem these two files, the problem file and the domain file, should be available.

JSHOP2 generates the steps of each plan in the same order that those steps will be later executed, so it knows the current state at each step of the planning process. As it has been previously mentioned, this reduces the complexity of reasoning by eliminating a great deal of uncertainty about the world, thereby making it easy to incorporate substantial expressive power into the planning system. In addition to that, some of the main characteristics of JSHOP2 are the following:

- JSHOP2 allows tasks and subtasks to be partially ordered. Plans may interleave subtasks from different tasks.
- JSHOP2 incorporates features from PDDL, such as quantifiers and conditional effects.
- If there are alternative ways to satisfy the preconditions of a method, JSHOP2 can sort the alternatives according to a criterion specified in the definition of the method. By this way it is possible to tell the planner which parts of the search space to explore first.
- JSHOP2 can handle temporal planning domains, so it is possible to maintain information for multiple timelines within the current state.

Along this Thesis, we will use the terms JSHOP2 and SHOP2 to refer to the same planning system, but it must be taken into account that JSHOP2 refers to the Java implementation while SHOP2 refers to the Lisp implementation of the planning engine. In addition, the Python implementation of the engine, called PYHOP, will be presented and explained in the next chapters.

3.3.3 Elements of a Domain Description

In JSHOP2, the description of a planning domain is composed by a set of tasks, operators, methods, axioms and external function calls. These elements are described briefly in this subsection.

Tasks

A task represents an activity to perform. Syntactically, a task consists of a task symbol followed by a list of arguments. A task may be either primitive or compound. A primitive task is one that is supposed to be accomplished by a planning operator: the task symbol is the name of the planning operator to use, and the task arguments are the parameters for the operator. A compound task is one that needs to be decomposed into smaller tasks using a method.

Operators

An operator indicates how a primitive task can be performed. JSHOP2 operators are similar to PDDL operators: each operator *o* has a head *head(o)* consisting on the operator name and a list of parameters, a precondition expression *pre(o)* indicating what should be true in the current state in order for the operator to be applicable, and a delete list *del(o)* and add list *add(o)* giving the negative and positive effects of applying the operator. Like in PDDL, the preconditions and effects may include logical connectives and quantifiers.

Operators can also do numeric computations and assignments to local variables. As in PDDL, two operators can not have the same name, so for each primitive task, all applicable actions are instances of the same operator. Operators also have an optional cost expression. This expression can use any of the variables that appear in the head and preconditions. The total cost of a plan is the sum of the costs of the operator instances.

Methods

Methods tell how to decompose a compound task into a partially or totally ordered set of subtasks, each of which can be compound or primitive. The simplest version of a method has three parts: the task for which the method is to be used (the *head* of the method), the preconditions that the current state must satisfy in order for the method to be applicable, and the subtasks on which the task will be decomposed. In general, a method m may have the form

$$(: method\ head(m)\ p_1\ t_1\ p_2\ t_2\ ...),$$

where *head(m)* is the task name and arguments for m, each p_i is a precondition expression and each t_i is a partially or totally ordered set of subtasks. The meaning of this is analogous to an *if-then-else* construct: it tells JSHOP2 that if p_1 is satisfied then t_1 should be used, otherwise if p_2 is satisfied then t_2 should be used, and so on.

Axioms

The preconditions of each method or operator may include conjunctions, disjunctions, negations, universal and existential quantifiers, implications, numerical computations, and external function calls. Axioms can be used to infer preconditions that are not explicitly asserted in the current state, simplifying the definition of methods and operators. Axioms are generalized versions of *Horn clauses*, with the form

$$(: -\ head\ tail)$$

which means that *head* is true if *tail* is true. The tail of the clause may contain anything that may appear in the preconditions of an operator or method.

External Function Calls

With the external function calls feature it is possible to make external Java function calls from the preconditions of a method or operator. These calls allow the execution of external code or libraries to run calculations or evaluations that in other ways could not be done from inside the JSHOP2 planning engine. This greatly increases the planning capabilities of JSHOP2, as it allows the interconnection among JSHOP2 and other types of specialized planners to do more specific computations.

3.3.4 JSHOP2 Algorithm

Algorithm 1 shows a simplified version of the JSHOP2 planning algorithm. The algorithm receives three parameters: the initial state s, a partially ordered set of tasks T and a domain description D. The algorithm runs while the set T contains any task, and on each iteration chooses one task t that is known of not being preceded by any other task. Then two possible cases appear: t being primitive or t being compound.

If t is a primitive task, then a set A of operator instances is created. This set is composed with instances of all the operators from D whose head match the head of the primitive task t and whose preconditions are satisfied by the state s. If the set A is non void, then an operator a from the set is chosen and added to the final plan P, the state s is updated by applying the delete list and add list from a, and the task t is removed from the set T. If A is void then this branch of the search space fails.

If t is a compound task, then a set M of method instances is created. This set is composed with instances of all the methods from D whose head match the head of the primitive task t and whose preconditions are satisfied by the state s. If the set M is non void, then a method m from the set is chosen, its subtasks *sub(m)* are added to the final plan P preceding the tasks that t already preceded, and the task t is removed from the set T. If M is void then this branch of the search space fails.

3.3.5 Multi-Timeline Preprocessing

JSHOP2 does not have an explicit mechanism for reasoning about durative and concurrent actions. However, it still has enough expressive power to represent durative and concurrent actions because it knows the current state at each step of the planning process and since its operators can assign values to variables and do numeric calculations. These features allowed the development of a preprocessing technique called *Multi-Timeline Preprocessing* (MTP) [94]. MTP is a technique for translating PDDL operators into JSHOP2 operators that keep track of temporal information in the current state.

Let us suppose that in each state s, every logical atom $(p\ c_1\ ...\ c_n)$ (also known as *predicates* or *functions* in PDDL terminology) represents a property for an object. A property is said to be *dynamic* if an operator may change the value c_n. For example, if the initial state for a problem contains $(at\ robot1\ location1)$ but there is an operator that moves $robot1$ to a different location, then the location of $robot1$ is said to be dynamic. For each property p that changes over time, MTP modifies the operators to keep track, within the current state, of the times at which the property changes and the times at which various preconditions depend on the property. For each dynamic property p, the current state will contain two time-stamps: $read-time(p)$, which is the last time that any action read the value of p, and $write-time(p)$, which is the last time that any action modified the value of p. MTP modifies the operators in such a way that whenever an operator accesses a dynamic property, the operator will update the property's read-time and, if an operator modifies a dynamic property, it will update the property's write-time. By this way, the current state will contain different timelines, consisting on a read-time and write-time for each dynamic property.

Algorithm 1: Simplified JSHOP2 planning algorithm

```
Data: $S$, $T$ and $D$
Result: Final plan $P$, or $FAILURE$
begin
    $P =$ The empty plan;
    $T_0 \longleftarrow \{t \in T :$ no other task in T is constrained to precede t$\}$;
    Loop
        if $T = \emptyset$ then
            return $P$;
        choose any $t \in T_0$;
        if $t$ is a primitive task then
            $A \longleftarrow \{a : a$ is an operator instance from $D$ whose head match $t$ and $s$
              satisfies $a's$ preconditions$\}$;
            if $A = \emptyset$ then
                return $FAILURE$;
            choose any $a \in A$;
            modify $s$ by deleting $del(a)$ and adding $add(a)$;
            append $a$ to $P$;
            modify $T$ by removing $t$;
            $T_0 \longleftarrow \{t \in T :$ no other task in $T$ is constrained to precede $t\}$;
        else
            $M \longleftarrow \{m : m$ is a method instance from $D$ whose head match $t$ and $s$
              satisfies $m$'s preconditions$\}$;
            if $M = \emptyset$ then
                return $FAILURE$;
            choose any $m \in M$;
            modify $T$ by removing $t$ and adding $sub(m)$, constraining each task in
              $sub(m)$ to precede the tasks that $t$ preceded;
            if $sub(m) \neq \emptyset$ then
                $T_0 \longleftarrow \{t \in sub(m) :$ no other task in $T$ is constrained to precede $t\}$;
            else
                $T_0 \longleftarrow \{t \in T :$ no other task in $T$ is constrained to precede $t\}$;
```

3.4 Conclusion

In this Chapter we introduce JSHOP2, the HTN planner on which our research is based. We have presented an overview of Hierarchical Task Network planning, including its formalism, as the base on which JSHOP2 relies.

It has been shown that hierarchies resemble the way the humans think and act. Also, the way the humans organize the world are reflected in the use of hierarchies. That makes the use of HTN planners for solving real-world problems a natural approach. Modelling the domain requires a bigger effort compared to other planning techniques, because the knowledge must be provided by one human that has some expertise in the matter, but the domain is more easily understandable by any person, expert or not. Also, this knowledge gives a boost in terms of performance and coverage across many domains to HTN planners compared to classical planners. Several examples have been presented that show that HTN planning can be used and applied in a large variety of real-world problems beyond a classical set of domains. It requires a greater effort in the knowledge definition for the HTN planner, but the higher performance of this planning technique makes the difference against other classical choices, justifying our selection of HTN planning for problem solving.

The JSHOP2 HTN planner has been presented as our selected HTN planner for the research in this thesis. Its modelling language is a direct translation of PDDL, making it easier to model domains and problems of any kind. It is widely extended and accepted in the research community and has been also adapted to be integrated in multi-agent environments, being capable of interacting with external agents and making queries to distributed heterogeneous information sources. That makes this planner a good selection among the different possibilities.

4 Decoupled Geometric and Symbolic Reasoning

This chapter addresses the combination of symbolic Hierarchical Task Network (HTN) planning and geometric reasoning in a multi-vehicle context involving the Vehicle Routing Problem (VRP) [18] for assembly operations with aerial robots. Each planner has its own problem domain and search space, and the description of how both planners interact in a loop sharing information in order to improve the solutions is presented. The HTN planner estimates the cost of the sequence of actions needed in the mission execution for each assignment computed for the VRP and gives feedback to the VRP solver in the search for a better assignment. This interaction scheme has been tested with different VRP solver configurations at the geometric reasoning level.

The main contribution presented in this chapter is a new way of connecting two independent planning systems based on a score calculation system that lets them cooperate in the optimization of the solutions found and its application in the context of structure assembly missions.

The chapter initially presents a short description of the addressed problem. The geometric VRP planner is described along with the connection between the geometric level and the symbolic planner. Simulation results in a scenario with a team of aerial vehicles assembling a structure are presented, showing the feasibility of the approach and the performance obtained with the different configurations of the VRP solver.

4.1 Introduction

The main goal of the European project that inspired our work, the ARCAS project, was constructing one structure defined in a CAD model by using multiple drones equipped with robotic arms. This kind of system is of great interest in situations where the assembly of a structure is required but the characteristics of the terrain or the environment make the assembly operation difficult. This type of situations may arise in civilian missions such a mountain rescue or fire, but also in military missions like building a bridge. Different scheduling and planning problems are involved in this context: assembly planning, multi-robot task allocation, symbolic planning and motion planning. There is a huge amount of related work in any of these topics independently, but the problem of combining these different planning levels has been less addressed in the literature until the last ten years.

Assembly planning can be defined as the process of constructing a specific structure given a set of parts, by computing a plan that is composed of different assembly operations and the order on which they must be executed to build the structure. During the plan generation different variables are taken into account such as the geometry of the parts, the geometry of the final structure, the resources to handle the parts and build the structure, the tools available, etc. It has been proven in [65] that all assembly problems have an NP-complete nature. A complete survey on assembly sequencing was presented in [60], taking into account the geometry of the problem and its combinatorial nature. The most complete and recent taxonomy on the topic can be found in [42]. In the context of multiple aerial robots, a team that cooperatively constructs a structure is presented in [76]. In this study, the different parts had a simple geometry and the tools used by the aerial robots were grippers, so picking and placing the parts did not need any kind of manipulation planning. In addition, the parts were placed sequentially, so the benefits of using a team of robots for parallelization were not fully exploited and the assembly tasks were done sequentially. On the other hand, an automated system that uses a team of robots equipped with different tools for the assembly of furnishing is presented in [67]. In that work, a symbolic planner determines the order of operations over the parts for the assembly operation. However, the allocation of tasks to robots is done at the symbolic level by using preconditions and postconditions in an object-oriented symbolic planning specification language, so the task allocation does not use any optimization heuristic.

It can be seen that the assembly planning and sequencing topics have been addressed since many decades ago but it still remains as an interesting research field and in fact, nowadays the need to have robots with precise assembly capabilities is increasing. One of the trends is to enhance the precision of robots by using new data models and sensors with better precision in their measurements. In [119] the authors present a system for the automatic generation of *'depth-maps'* for peg-in-hole assembly operations. Depth-maps are two-dimensional arrays that contain the perpendicular distances of a peg with respect to its mating hole and are commonly used in assembly operations. The framework presented automatically generates a depth-map given as input the CAD model of the peg and hole. Another way of improving the precision of assembly operations is by measuring the sound produced by mating parts, as presented in [75]. In this study, an acoustic contacting detection is presented to substitute the traditional use of strain gauge load cells. By putting a receiver into a part, when two parts come into contact part of the sound wave energy is transmitted from the part to the receiver, making it possible to detect the contacting event. The work presented in [45] performs object localization using a monocular camera. The authors use an eye-in-hand manipulator and a mobile platform for the task. Initially, a SURF algorithm is applied for feature detection and initial localization. Then, a new probability-based natural right angle detection algorithm is applied, and finally, a 2D template matching algorithm is used to fine-tune the object localization. Another interesting trend is the use of augmented reality (AR) to improve the accuracy of the assembly process in teleoperation. In [9] the effects of using an AR system are evaluated with the intention of overcoming the differences in perception between telepresence and real presence. The system used an RGB-D camera and a head-mounted display for the operator, and a Baxter robot on the other side. By using this setup, the authors demonstrated that by using their AR system the accuracy and efficiency of the robot in the assembly tasks were improved. Regarding high precision measurements, a case study of the error chain is done in [124]. In that work, a robotized assembly system is studied and an assembly accuracy analysis model for misalignment errors is proposed. This model provides an assembly accuracy estimation and has been tested in different assembly experiments, giving a reliable worst-case accuracy estimation.

An interesting application of assembly planning can arise from the use of multiple robots. Self-assembly is a process in which a disordered system of pre-existing components forms an organized structure or pattern as a consequence of specific local interactions among the components themselves, without external direction. Some studies can be found in this matter. In [59] the authors have addressed the self-assembly process for swarm robots. In that study they propose an enhanced self-assembling morphology distributed control algorithm for swarm robots, enabling dynamic local navigation according to the distance of seed robot and docking robot. The work also presents time measurements for line-shape, arrow-shape, T-shape and star-shape morphologies. The case of using a heterogeneous group of robots for self-assembling is studied in [26]. In that work, each agent can only become the neighbour of a specific set of agents in the target pattern. A constrained bipartite graph-matching algorithm is used to allocate the agents to spots in the target pattern. The allocation is done in a way that adjacent agents are allocated only if they are compatible. The presented algorithm is also compared with other optimal matching algorithms, showing lower run times.

Another application of assembly planning that also requires cooperation among the different robots is collaborative assembly. In [79] the authors address the problem of moving objects with a group of autonomous robots. Instead of using different planning strategies described in the literature for pick and place, object passing, object re-grasping, etc, they propose a planning scheme that aims to unify the different solutions. The implemented planner can exploit support surfaces if required in order to reach the goal, for example putting an object in a region of a table that lets another robot to pick it. The planner relies on the geometric information stored in a database about support surfaces, possible approximations and feasible grasps, among others. In [34] a model for an assembly/disassembly line that uses two wheeled robots working in parallel is presented. One of the robots has a robotic manipulator used for part manipulation while the other robot is used for transporting the parts. During the assembly, if a part does not pass the quality test the whole assembly is cancelled and the disassembly starts to recover the different parts. The work is focused on task planning, modelling and simulation of the assembly line, and use Synchronized Hybrid Petri Nets to control the assembly/disassembly. Regarding safety, some human-robot cells trigger a safety stop when humans leave the safety zone. Collaborative human-robot assembly requires further research to avoid completely stopping robot operations when humans are near the working area of the robots. In [120] a human-aware robotic system is presented. The system is capable of predicting human motion and plan in time to execute safe motions during automotive assembly tasks, without needing to trigger a safety stop. The main interest of this system lies in its ability to adapt the behaviour of the robot to the behaviour of the human. The robot can operate in a *'Planning with Prediction'* mode, without knowing the task of the human. The robot uses the detected human position and a set of predictions to adapt its motion to the motion of the human. By this way, the robot can pause its task or move to another zone to let the human move freely, without needing to trigger a safety stop.

In our research it was required to deal with the combination of a symbolic state and a geometric configuration, where a trajectory can modify the symbolic state. Hence, when an action is applied, both the symbolic and geometric states can change. In the literature, there are different possible decompositions for the composition of the symbolic and geometric states: in [16, 77, 70, 114, 63] the symbolic level calls the geometric level, the geometric level calls the symbolic level in [36, 103, 14] and the compound state is used directly in [12, 51, 52]. The approach adopted in our project belongs to the second group where the geometric level calls the symbolic level. However, this thesis is focused on the combination of a symbolic HTN planner and a constraint satisfaction solver for the Vehicle Routing Problem (VRP). The scheme proposed for this connection in a multi-robot context for assembly operations with aerial robots is the main novelty of this work.

This chapter is focused on the combination of multi-vehicle symbolic planning with geometric reasoning in the context of structure assembly. The geometric layer in this thesis has been modelled at a high level of abstraction, using cartesian coordinate systems for locations, graphs with costs for navigation maps and Euclidean distances. As it was explained in Chapter 2, there are three different ways of connecting the symbolic and geometric levels: the symbolic level calls the geometric level, the geometric level calls the symbolic level, and using the compound state. Our approach belongs to the second group, where the geometric planner calls the symbolic level. The novelty of our contribution compared to previous work in that group is the use of a VRP planner to implement the geometric layer. Thus, in this chapter the terms *VRP planner* and *geometric planner* refer to the same thing.

4.2 Problem Statement

Given a set of parts that compose a structure, with the parts distributed along a scenario, and given a set of aerial robots also distributed along the scenario, our goal is to assemble the whole structure using the aerial robots, minimizing the total assembly time and maximizing the potential parallelism of using a team of robots in a collaborative way.

The parts have a simple geometry (rectangular parallelepiped), with a handle on top which makes it possible to pick the part, and with a cavity beneath which allows stacking the parts. All the robots are equipped with robotics manipulators that let them pick and place the parts. In addition, the assembly plan is known in advance, so for each part it is known which other parts must be assembled first. The locations and orientations where the parts must be picked and where the parts must be assembled are already known.

The problem of assembling the structure can be seen as two problems highly coupled one with each other: the problem of assigning the parts to the robots, and the problem of scheduling the different task of the robots in time. They are highly coupled because changing the assignment will lead to a different scheduling, and changing the scheduling, that is, changing the time on which the different tasks are planned to be executed, will probably invalidate the assignment.

The assignment problem consists in assigning the different parts to the available robots by following some criterion. One valid criterion could be trying to minimize the routes of the used aerial robots, because it is reasonable to think that minimizing the routes of the robots the total assembly time will be minimized, as the robots will travel the shortest paths possible. However this is not always true. If we think in the assembly plan, we know that the parts must be assembled following some order, and one part may need other parts to be assembled first. If the criterion of minimizing the path is the only one used, this could result in one 'unbalanced' assignment, where some robots were assigned lots of parts because they were near them, and the other robots will have to wait until these unbalanced aerial robots assemble their parts. So, it seems clear that the criterion of minimizing the routes of the robots must be accompanied by another criterion. The other criterion could be to do the assignment in a 'balanced' way, by inspecting the dependencies of each part and assigning them in a way that the amount of time that the robots have to wait for the other is also minimized. In any case, this problem may be seen as a variant of the Vehicle Routing Problem (VRP) described in [18].

On the other hand, once an assignment has been computed, the aerial robots must have a detailed plan to execute the assembly of the parts. The tasks that compose their plans must be correctly scheduled. These tasks include actions such as taking-off, travelling to points of interests, synchronizing with the other aerial robots, picking the parts, placing the parts, and so on. The start time and end times of each action must be planned and computed.

Let us consider a mission $\mathscr{M}$ consisting on the assembly of a structure composed by several parts initially located around the environment. The parts have to be assembled on specific locations by a team of n aerial vehicles starting the mission from their home locations. Then the mission is composed by a set of assembly tasks $\mathscr{T}$. Each of the parts has a weight and a dependency list consisting on the tasks that must be done prior to its assembly. Let us define $\mathscr{L}$ as the set of stock parts locations, $\mathscr{L}'$ as the set of locations where the parts have to be assembled and $\mathscr{H}$ as the home locations of the aerial vehicles. The objective is to assemble all the parts on their locations minimizing the travel flight times of the vehicles and exploiting the potential parallelism that can be achieved using multiple aerial vehicles.

The implicit combinatorial problem can be expressed by the edges of a graph $G(V,E)$ considering the following notation:

- $\mathscr{T} = \{t_1, t_2, ..., t_m\}$ is a set of m assembly tasks.
- $P_i \subseteq \mathscr{T}$ is the set of preconditions for the i-th task, i.e. the subset of tasks that must have been done prior to the execution of that task.
- n is the number of aerial vehicles.
- $\mathscr{L} = \{l_1, l_2, ..., l_m\}$ is the set of stock parts locations and $\mathscr{L}' = \{l'_1, l'_2, ..., l'_m\}$ is the set of final assembly locations.
- $\mathscr{H} = \{h_1, h_2, ..., h_n\}$ is the set of aerial vehicle home locations.

- $V = \{\mathscr{L} \cup \mathscr{L}' \cup \mathscr{H}\} = \{v_1, v_2, ..., v_{2m+n}\}$ is the set of vertices of the G graph.
- $E = \{(v_i, v_j) | v_i, v_j \in V; i < j\}$ is the edge set.
- $R_k = \{r_1, r_2, ..., r_s\} \subseteq V$ is the route for the k-th aerial vehicle, composed by a subset of s_k vertices from V.
- Cost c_{r_i, r_j} is a non-negative travel time between vertex r_i and r_j, where $c_{r_i, r_j} = c_{r_j, r_i}$.
- $p = \{p_1, p_2, ..., p_n\}$ is a vector with the maximum payloads of the aerial vehicles.
- $w = \{w_1, w_2, ..., w_m\}$ is a vector containing the weights of the parts.
- $q = \{q_1, q_2, ..., q_m\}$ is a vector containing the times on which the parts are finally assembled.

The problem consists in determining a set $\mathscr{R}$ of routes with minimal cost and a vector q of minimal task assembly times, with each route starting at the home locations of the vehicles, such that every vertex in $\mathscr{L}$ is visited at least by one vehicle and followed by its subsequent vertex in $\mathscr{L}'$, without exceeding the payload of each vehicle and respecting the preconditions for each of the parts. The same location can be visited by several aerial vehicles because some parts must be transported cooperatively by more than one aerial vehicle if they are too heavy.

For the k-th aerial vehicle, the cost of a route is given by

$$C(R_k) = \sum_{i=1}^{s_k - 1} c_{r_i, r_{i+1}}, \tag{4.1}$$

where $r_1 \in \mathscr{H}$, $r_i \in V$ and $r_j \in \mathscr{L} \implies r_{j+1} \in \mathscr{L}'$. Considering that up to two aerial vehicles can cooperatively transport a single heavy part, this route R_k is feasible if:

$$(p_k \geq w_j) \vee (\exists R_z | r_j \in R_z \wedge (p_k + p_z) \geq w_j),$$

i.e. the weight of each part does not exceed the maximum payload of the aerial vehicle transporting it or there is another available aerial vehicle so that both can transport it cooperatively. For each part w_j in this route it must be met in addition that $\forall t_s \in P_j : q_s < q_j$, i.e. all the parts from its set of preconditions must have been assembled before that part.

The goal is to minimize the total travel time $\sum_{i=1}^{n} C(R_i)$ of the feasible routes executing all the assembly tasks of the mission and to balance the workload of the different aerial vehicles.

4.3 Geometric Planner

4.3.1 Overview

The problem presented in the previous section has a high combinatorial nature. This kind of problems are not suited for HTN planners as due to their implementation they usually get stuck during the search phase, as the search tree is usually very huge in combinatorial problems of medium and high sizes. To solve this problem we need to help the HTN planner with a layer that frees the symbolic planner from the computation needed to solve the VRP part. We do this by solving the VRP part with a separate planner, presenting a new way of connecting two independent planning systems based on a score calculation method that lets them cooperate in the optimization of the solutions found.

To solve the VRP problem we used OptaPlanner [100], an open source, multi-platform planning engine written in Java and released under the Apache Software License. OptaPlanner is aimed to solve planning problems with resource usage optimization. It is a lightweight, embeddable planning engine that allows to solve optimization problems efficiently, applying constraints on plain domain objects and reusing existing code from other previously implemented problem domains. One of its main characteristics is that it has been developed to solve real-life problems, having a big applicability in several already known problem types such as vehicle routing, educational timetabling, sport competition scheduling, etc.

OptaPlanner is capable of generating near-optimal plans by applying optimization heuristics and meta-heuristics combined with score calculation. Its main advantage is that the solver's algorithm is highly configurable. In OptaPlanner it is possible to use different heuristics and metaheuristics algorithms, also called optimization algorithms, applied in sequence so that the user can select the most suitable algorithm combination for the problem in question. The optimization is done in base of a score calculation that is computed after a solution is found. This score determines the suitability of the last computed solution: if after searching for a new solution the new score is worse than the score computed for the previous solution, then the last solution is discarded and the process continues trying to generate a solution with a better score.

Every new solution found by the solver is usually computed from a previously computed solution, so the search process can be seen as evolving an initial solution into a mostly better and better solution. OptaPlanner uses a single search path of solutions, not a search tree as is usual in other planning engines. At each solution in the path it evaluates a number of changes and applies them to take a step to the next solution, doing that during a high number of iterations until the search process finishes, usually because a timer set by the user has run out. OptaPlanner acts like a human planner, using a single search path and moving facts around to find a good feasible solution. That way of working gives a high scalability.

In OptaPlanner the entities of the real world that must be assigned are called *planning entities*, and are represented as Java classes. These planning entities have one or more *planning variables* represented as Java attributes that take different *planning values* over the planning process. To solve an assignment problem, each of the planning entities must have been given a valid planning value for each of its planning variables. This requires the implementation of several Java classes that model the problem domain and the solution, being also possible to implement comparator classes to sort the planning entities by its assigned planning value or by other criteria to aid the planning process. After implementing the domain, the solver must be configured by writing an XML file were the different optimization algorithms desired to be used in the search process are specified and customized. Then, the problem data set has to be read as an XML file. For that purpose, OptaPlanner has a module that parses the XML file of the data set and automatically instantiates the objects of the domain with all the data contained in the file. Finally, the solver starts the search process and finishes when the search is exhausted or when a timer expires. Then the best solution found can be recovered.

4.3.2 Moves

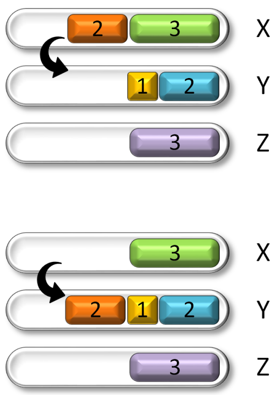
OptaPlanner acts like a human planner. When an initial solution is found, it evaluates a number of changes on that solution and applies them to take a step to a new solution.

A *move* is a change or a set of changes from a solution A to a new solution B. The new solution is said to be a neighbour of the original solution. A single move can change a single planning entity or multiple planning entities. Whenever the case, all optimization algorithms use moves to transition from one solution to another. The number of moves that is possible to do from one solution may be extremely large. Thus, in addition of generating moves, they must be selected to conform a reduced set. Discarding moves implies that the solutions that could be reached by applying these moves will never be inspected, but the reduction is necessary in order to keep the performance of the solver.

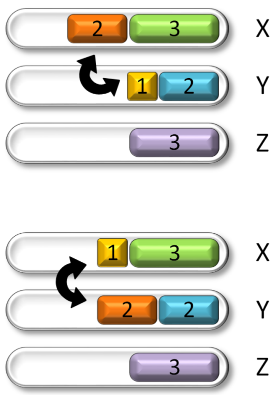
The way of generating moves in OptaPlanner is by using a special Java class named *MoveSelector*. The purpose of a MoveSelector is to create a move iterator. The optimization algorithm will then use it to iterate over the set of possible moves. To generate a move, the MoveSelector needs to select one or more planning entities and planning values for their planning variables. They are declared and configured in the XML configuration file of the solver. One interesting characteristic is that they can be nested so the children MoveSelectors can feed moves to their parent MoveSelectors, conforming a tree structure.

From all the moves generated by a MoveSelector or by a combination of MoveSelectors, the optimization algorithm must select only one. The selected move is then applied to the current solution, producing a new one. This move is called the *step*. The criterion to select the winning move depends on the optimization algorithm used. Some of them may select the first produced move or the move that leads to the best score. For example, the Hill Climbing optimization algorithm will always select the move that leads to the best score but the Tabu Search optimization algorithm will refuse to select that move if it is included in the Tabu list, so a move that leads to a lower score will be selected instead. An explanation of the different optimization algorithms will be presented in the next subsection.

Figures 4.1 and 4.2 show a short explanation of four of the most commonly used types of MoveSelectors that can be used within the OptaPlanner framework. In addition, Figure 4.3 shows an example of nesting multiple MoveSelectors and how it can be customized to give priority to some MoveSelectors over others.

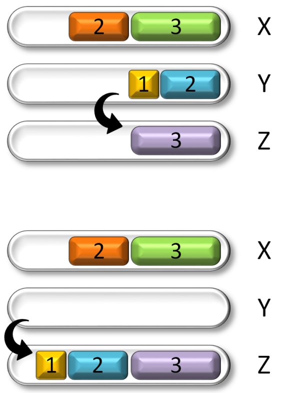


(a) Change MoveSelector.

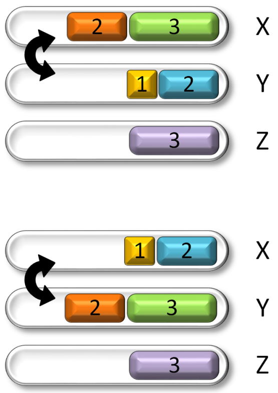


(b) Swap MoveSelector.

Figure 4.1 Example of the Change and Swap MoveSelectors. In the images, three lists labelled as X, Y and Z are shown. All the lists have the same limited capacity to store generic objects. The objects that must be stored within the lists represent the planning entities and are shown in boxes of different colors, having each a length whose value is shown inside. The planning variable for the boxes would be of type list, and the possible planning values would be X, Y or Z. Figure 4.1a shows an example of a Change MoveSelector, which moves planning entities by changing the planning values of their planning variables. In that case, it is moving the orange box from list X to list Y. Figure 4.1b shows an example of a Swap MoveSelector, which swaps planning entities by swapping the planning values of their planning variables. In that case, it is swapping the orange and yellow boxes between lists X and Y.

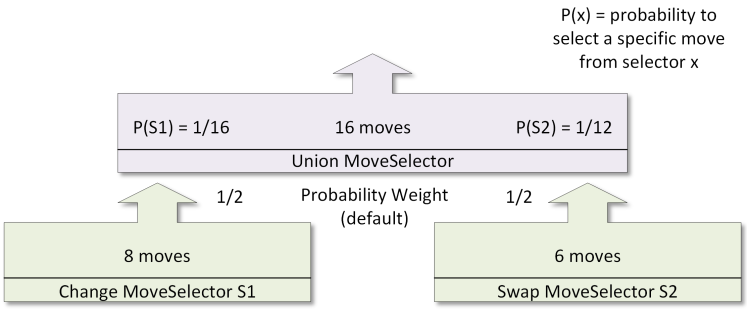


(a) Change Pillar MoveSelector.

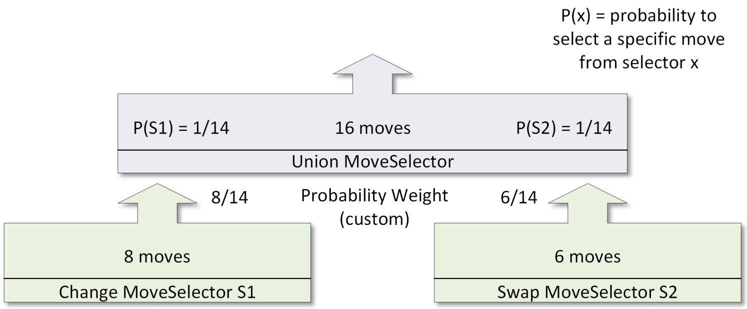


(b) Swap Pillar MoveSelector.

Figure 4.2 Example of the Change and Swap Pillar MoveSelectors. In the OptaPlanner terminology, a *pillar* is a set of planning entities that have the same planning value for their planning variables. In the example shown it is traduced as boxes that are stored in the same list. Figure 4.2a shows an example of a Change Pillar MoveSelector, which moves a pillar of planning entities. In that case, it moves the pillar from list Y to list Z. Figure 4.2b shows an example of a Swap Pillar MoveSelector. In that case, it swaps the pillars between lists X and Y.



(a) Union MoveSelector with default probability weights.



(b) Union MoveSelector with customized probability weights.

Figure 4.3 Example of nesting MoveSelectors by using a Union MoveSelector. The Union MoveSelector takes all the moves that are generated by all of its child MoveSelectors in the XML configuration file. Inside a child, all the moves have the same probability of being selected, but a probability weight can be configured for each MoveSelector child to make more or less possible from which MoveSelector a move will come. In Figure 4.3a the probability weights are the same, which means that the move finally selected by the solver has the same probability of coming from the Change MoveSelector or from the Swap MoveSelector. However, as each of the children MoveSelector may generate a different number of moves, the probability of selecting a specific move from one child MoveSelector differs from the other child. Inside the Change MoveSelector that generates 8 moves, the probability of selecting a specific move is 1/8, but due to the probability weight of 1/2 its final probability of being selected is 1/16. For the Swap MoveSelector a specific move has a probability of 1/12 for being selected. Figure 4.3b shows the opposite case. The probability weight for the Change MoveSelector is greater, but the probability of selecting a specific move from the Change MoveSelector is the same as the probability of selecting a specific move from the Swap MoveSelector.

4.3.3 Solver Phases

As it was mentioned before, the OptaPlanner solver can be configured to use multiple optimization algorithms. Each of the optimization algorithms used is called a *solver phase*. During the execution of the solver there is never more than one solver phase executing at the same time, so a solver phase only starts when the previous phase has finished, i.e. they are executed sequentially. There are three different types of solver phases that can be used in the OptaPlanner solver: Construction Heuristics (CH), Metaheuristics (MH) and Exhaustive Search (ES).

Construction Heuristics

The CH solver phase builds an initial solution in a short time. The solution computed is not always feasible, but it tries to find it fast so that the following solver phases can finish the search of a feasible one by starting from that initial solution. There are different algorithms that can be used as CH. One common characteristic of them is that when a CH assigns a planning entity, that assignment remains unchanged until the end of the algorithm. This is the main reason that makes the CH algorithms find solutions that may be unfeasible: no re-planning is done at this phase.

The available CH algorithms that can be used to configure the solver are: First Fit, First Fit Decreasing, Weakest Fit, Weakest Fit Decreasing, Strongest Fit, Strongest Fit Decreasing and Cheapest Insertion. A detailed description of them can be found in [100].

Metaheuristics

The MH solver phase is based on different types of local search algorithms. Local search starts from the initial solution computed by the CH phase and evolves it into a mostly better and better solution. At each solution, it evaluates a number of moves between the planning entities and applies the most suitable to step to the next solution, whose score may be better, equal or worse than the previous. Allowing as solution a new one which has a worse score than the previous is important because it avoids getting stuck in local minimum. The local search does not use a search tree, but a search path. When finding a new solution all possible moves are evaluated but unless it is the chosen move, it does not investigate further the rest of possible solutions. That makes the local search very scalable, but it may not find never the optimal solution.

Five different local search algorithms can be used to configure the solver:

- Hill Climbing: tries different moves among the planning entities and then takes the best move, which is the move which leads to the solution with the highest score.
- Tabu Search: like Hill Climbing but maintains a tabu list to avoid getting stuck in local optima. The tabu list holds recently used objects that are taboo to use for now. Any move in the current solution that involve an object in the tabu list is not accepted.
- Simulated Annealing: a move is accepted if it does not decrease the score or, in case it does decrease the score, it passes a random check. The chance that a decreasing move passes the random check decreases relative to the size of the score decrement of the new solution and the time the phase has been running.

- Late Acceptance: accepts a move if it does not decrease the score or if it leads to a score that is at least the score of a fixed number of steps ago.
- Step Counting Hill Climbing: for a number of steps, it keeps the step score as a threshold. A move is accepted if it does not decrease the score, or if it leads to a score that is at least the threshold score.

Exhaustive Search

The ES solver phase does not depend on previous phases and is configured alone. The Brute Force or the Branch and Bound algorithms are available. These methods guarantee the find of the optimal solution for a problem, but are poorly scalable so are not usually chosen to solve real problems. However, for very small size problems they can be a good choice.

4.3.4 Score Calculation

To compare the suitability of the different solutions computed along the task allocation process, a score-based calculation is done after a solution is found. This score is based on the definition of three types of constraints with different levels of relevance. Given a solution, its score consists on the sum of the broken constraints for each of the constraint types defined for the problem. Thus, a sum of zero for each of the constraints is the best possible score for a solution. The constraint types are:

- Hard-constraints: these are constraints that must not be broken in any case. A broken hard-constraint will lead to an unfeasible plan, so its sum must be zero.
- Medium-constraints: these are constraints that are desirable to be broken the less as possible. Its importance is under the importance of the hard-constraints but above the importance of the soft-constraints.
- Soft-constraints: these are the constraints with lower priority. They have the lowest impact when broken, but still they must be minimized.

For a given problem, the type of constraints that will be used in the score calculation is previously specified in the XML configuration file of the solver. The meaning of each of the constraints for a specific problem is defined in the implementation of the problem domain, where a Score class is defined to be executed each time a solution is found. Inside that class, the sum of the different types for the used constraints is done taking into account the state of the solution.

As commented before, the score calculation is done after a solution is found. That is the case when a MoveSelector generates a move. Every move generated leads to a solution but this solution will only be the new solution if the move is selected by the optimization algorithm. Whenever the case, the score calculation process is executed after each move is generated to pair the move with the score of the solution it leads to. This is necessary for the optimization algorithms to compare the score of the solutions for every move generated and select the most appropriate move.

4.3.5 Geometric Domain

In OptaPlanner the entities of the real world that must be assigned are called planning entities and are represented as Java classes. For the problem defined in Section 4.2 the geometric planner domain has been defined as follows: the planning entities are the assembly tasks. Each assembly task represents a part that must be assembled by one or more vehicles, depending on the part weight and the payload capabilities of the vehicles. In addition to the weight, each part has a dependency list that contains all the assembly tasks that must be executed before the assembly of that specific part. Each of the vehicles has a list on which the assigned assembly tasks will be stored. The same assembly task can appear in the list of multiple vehicles if the weight of the part requires it to be assembled by more than one, but the sum of the payload weights of the given vehicles must be equal or greater than the part weight.

4.4 Symbolic Planner

4.4.1 PDDL Domain Definition

In Section 4.2 a planning problem consisting on the assembly of an structure by the cooperation of a team of unmanned vehicles equipped with robotic manipulators was presented. The part of the problem consisting on the assignment of assembly tasks to vehicles was previously presented in Section 4.3, describing the solver and domain designed to model that part of the problem.

The problem to solve in this section consists on, given an assignment of assembly tasks to vehicles, finding a low-level plan for each of the vehicles and scheduling the different low-level operations minimizing the total assembly time. The low-level plans must contain the correct sequence of assembly tasks needed to assemble the structure successfully. In this subsection the domain designed in PDDL to model that part of the problem is presented.

Restrictions

Although recent versions of PDDL offer the greatest expressiveness for the domain definition, the version of the language used to define the model is PDDL 2.2 as it offers enough expressiveness for our specific domain, simplifying its development. As commented in Chapters 2 and 3, the PDDL language is used to model planning problems and not to solve them, so this PDDL domain has been developed with the purpose of enhancing the understanding of the problem and easing the later use of the HTN planner to finally solve the problem described.

To model the problem, the following restrictions have been taken into account:

- Each of the assembly tasks contains a dependency list consisting on the assembly tasks that must be done prior to its execution. At any given time, only those assembly tasks that have all its dependencies met can be selected to be executed.
- To find a low-level plan for each of the vehicles, and assignment of assembly tasks to vehicles is needed to know which parts have been assigned to each of the vehicles.

This assignment is supposed to be previously known, as it is part of the problem that solves the planner presented in Section 4.3.

- The maximum number of vehicles needed to transport a single part has been limited to two. This is reflected on the assignment of assembly tasks to vehicles, where some parts appear assigned to only one vehicle and some others to two vehicles. The information related, such as the payload capabilities of the unmanned vehicles or the part weight is not needed at this level, as it has been previously used to compute the assignment.
- Due to difficulties on the transportation, it is not allowed to make subassemblies with the parts and assemble them on the final structure. All the parts must be directly assembled on their assembly locations in the structure.
- Parallelization among the assembly tasks execution is needed when assembling the structure in order to minimize the total assembly time. This implies that the actions in the low-level plans may be executed concurrently.
- All the unmanned vehicles have a battery that limits their operation time and decreases as the time goes by, along the execution of the different actions.
- All the actions that the unmanned vehicles can perform have a duration. This duration is fixed for some of the actions and computed for others.
- The time variable must be taken into account in the domain definition, as the actions in the low-level plans must be correctly scheduled and may be executed concurrently, the resources such as the batteries must decrease over the time, and the total assembly time must be minimized.

Types, Predicates and Functions

Although not mandatory for the definition of a domain, typing clarifies the kind of entities that may be present, so the domain has been defined using this feature. PDDL has few built-in types, mainly the *object* and *number* types to represent things and numbers, which are the base for the definition of new types. Usually with these two types suffice for the domain definition, but for clarifying purposes some new types have been defined. These types can be seen on Table 4.1.

Predicates represent the logical properties of the entities that are of interest. They define the state at which the entities of a problem instance may be. A predicate is a logical sentence that may be checked to be true or false in a specific time instant or during a time interval. For example, during a *LAND* action a quadrotor is on a *landing* state, but after finishing the action the state of the quadrotor is *landed* and the previous landing state is no longer true. Table 4.2 shows the different predicates that are present in the domain. In addition to these, a special type of predicates named *derived-predicates* have been used. Those special predicates model the transitive relations that can appear among normal predicates, and are mainly used to simplify logical expressions. The list of derived-predicates can be seen on Code 4.1.

Predicates associate a logical value to a single object or to a tuple of objects. To be able to model non-binary resources such as fuel-level, speed or distance, among others, PDDL

Table 4.1 Types defined for the PDDL domain. In addition to the two built-in types, five new types have been defined to model the different objects that may be present in a problem instance for the domain. The aerial vehicles that may be present in a problem instance are of the specific type *quadrotor*, and all the parts are considered to be of the same type. The *location* type defines the places where the different objects of the problem instance are situated. This type serves as the parent of two special location types, the *pick_location* and *assembly_location* types, which define the location from where a part has to be picked and the location where a part has to be placed in the structure, respectively. When reaching a specific pick/place location, a vehicle may need additional data on how to finally pick or place the related part, so this additional data requirement has been modeled by defining these two special location types.

Type Name	Parent Type
quadrotor	object
part	object
location	object
pick_location	location
assembly_location	location

Code 4.1 Derived predicates defined for the PDDL domain. These predicates have been defined to infer logical expression that automatically become true after some other logical expression changes its value to true. Three derived predicates have been defined. The first derived predicate tells that if a part is assembled, then it must exists an assembly location on which the part is placed. The second derived predicate tells that if a part is assembled, then all the parts that are dependencies for this part must also be assembled. The third derived predicate tells that if a part is at a quadrotor, then the quadrotor is transporting the part.

```
(:derived
  (assembled ?prt)
    (exists (?loc - assembly_location) (at ?prt ?loc))

  (assembled ?prt1)
    (forall (?prt2 - part)
      (imply (depends ?prt1 ?prt2) (assembled ?prt2)))

  (at ?prt - part ?uav - quadrotor)
    (transporting ?uav ?prt)
)
```

2.1 introduced the concept of *numeric-fluents*, also called *functions*. Functions are used to associate a numerical value to an object or tuple of objects. This value does not need to be constant, and thus can be decreased or increased over time. Functions have been used in this domain to model concepts such as the batteries of the vehicles or the distance

Table 4.2 Predicates defined for the PDDL domain. These predicates model the state at which the quadrotors or parts may be, or the relations between different entities. For example, a specific quadrotor may be in the *landing* state during a *LAND* action or in the *moving* state while travelling from one location to another. Similarly, a part will be in the *assembled* state after it has been assembled on the final structure. In the case of the *assemble_at* predicate, it tells the location on which a specific part must be assembled, establishing a logical relation between both entities. The *at* predicate has been defined to have a second form to cover the case of a part that has been picked by a quadrotor and that is considered to be at the vehicle. Both forms differ on the type of their arguments.

Predicate	Description
(at ?obj - object ?loc - location)	Object *obj* is at location *loc*
(at ?prt - part ?uav - quadrotor)	Part *prt* is onboard quadrotor *uav*
(assemble_at ?prt - part ?loc - assembly_location)	Part *prt* must be assembled on location *loc*
(depends ?prt1 ?prt2 - part)	Part *prt1* depends on part *prt2*
(assigned ?prt - part ?uav - quadrotor)	Part *prt* is assigned to quadrotor *uav*
(assembled ?prt - part)	Part *prt* is assembled
(landed ?uav - quadrotor)	Quadrotor *uav* is landed
(landing ?uav - quadrotor)	Quadrotor *uav* is landing
(taking_off ?uav - quadrotor)	Quadrotor *uav* is taking-off
(hovering ?uav - quadrotor)	Quadrotor *uav* is hovering
(moving ?uav - quadrotor)	Quadrotor *uav* is going to some location
(picking ?uav - quadrotor ?prt - part)	Quadrotor *uav* is picking part *prt*
(placing ?uav - quadrotor ?prt - part)	Quadrotor *uav* is placing part *prt*
(transporting ?uav - quadrotor ?prt - part)	Quadrotor *uav* is carrying part *prt*

between locations, among others. The complete list of the defined functions can be seen on Table 4.3.

Table 4.3 Functions defined for the PDDL domain. Four different functions have been defined to model the battery and speed of the quadrotors, the distance between locations and the parts that remain to be assembled. The battery is expressed in seconds, as it represents the time that is left until the battery runs out. The remaining_parts function represents the parts that are left to be assembled in a given time.

Function	Description
(battery ?uav - quadrotor)	Remaining battery life for quadrotor *uav*, expressed in seconds
(speed ?uav - quadrotor)	Medium speed at which quadrotor *uav* moves, expressed in meters per second
(distance ?loc1 ?loc2 - location)	Distance in meters needed to go from *loc1* to *loc2*
(remaining_parts)	The number of parts that remain disassembled

Durative Actions

The values of predicates, derived-predicates and functions define the state on which the entities of a problem instance are in a given time instant. To advance in the planning process, a planner needs a way of modifying the value of these expressions to find new states, in a search process that leads to the goal state. In PDDL, the way of modifying the states of the entities is by the application of *actions*. As it name suggests, actions are the different operations that is possible to do over the entities that are present in the world. For an aerial vehicle, possible actions would be the take-off, move or land operations.

Actions usually have a parameter list which contains the arguments for the action. These arguments are the objects that take part in the execution of the action. For a take-off action, it would be the quadrotor that will execute the action, for example. Actions also have conditions and effects. Conditions are the predicates whose value must be true for the action to be applicable. If any of the predicates that appear in the condition list is false, then the action will not be applicable for the given arguments. Effects are the consequences of applying the action, and are also expressed as predicates with a given logical value. If the conditions are satisfied, then the values of the predicates that appear in the effects list will take effect. This can include setting to true or false the value of previous existing predicates or setting new ones.

Due to the needs of taking into account the time variable, durative actions have been used instead of the standard actions. Durative actions were first included in PDDL 2.1 to model actions that can take place during a time interval, instead of only on specific moments. These actions have a duration expression that computes the exact or bounded duration of the action, in time units. The duration expression may be:

- An equality, on which case the duration takes a fixed value.
- An inequality, on which case the duration will be bounded among some values.
- Void, on which case the value of the duration can not be fixed or bounded beforehand and is determined by external events that make the action stop.

In addition to the duration, durative actions have some modifiers that are possible to be applied over the conditions and effects of the action. These modifiers make possible to express the exact moment on which a condition must be true for the action in order to be applicable, and the exact moment on which an effect becomes applied. The available modifiers are:

- At start: determines that the condition or the effect must be checked or applied at the start of the action, respectively.
- Over all: determines that the condition or the effect must be checked or applied during the action, respectively. This range does not include the start and end moments of the action.
- At end: determines that the condition or the effect must be checked or applied at the end of the action, respectively.

The durative actions defined for the domain model the different operations that the vehicles can perform during a mission. Seven different durative actions have been defined. Codes 4.2, 4.3 and 4.4 show the take-off, land and move durative actions. Codes 4.5, 4.6, 4.7 and 4.8 show the pick and place durative actions. In the case of the pick and place actions, two different versions have been defined, one for the case of a part that needs to be transported by a single vehicle and another for the case of a part that needs to be transported by two vehicles.

Finally, a special consideration must be taken into account for the PDDL domain. Each assembly task has a list of dependencies that must be met prior to its execution. One part can not be assembled if all parts that appear on its dependency list are not yet assembled. It may be the case that, in a given moment, a quadrotor could not assemble any of its assigned parts until the dependencies of any of these parts are finally met. If that is the case, then the quadrotor must wait until the dependencies of its assigned parts are met. A similar case could be when picking a part that must be transported among two quadrotors. The part may have all its preconditions met and one of the quadrotors may be located at the pick location, however the other may be executing other operation. Again, the first quadrotor has to wait until the second is ready and positioned at the pick location. Code 4.9 shows the definition of a *wait* durative action. This action has been defined to take into account that it may be necessary for a quadrotor to wait an amount of time until some condition is met prior to the execution of its current action. This definition may not be necessary depending on how the planner used to solve the PDDL problem behaves when an agent is locked, because some of them automatically introduce *wait* or *NOP* (No OPeration) actions to fit in the final plan or automatically increase the duration of other previously executed actions if possible. As it will be seen in Subsection4.4.3, that is not the case of JSHOP2, and the action definition will be needed.

Code 4.2 *Take-off* durative action. The quadrotor must be landed at the start of the action, and during the action its state must be *taking-off* and its battery must be equal or greater than zero. If these conditions are met, then the state *landed* is removed at the start, the *taking-off* state is set and the battery is decreased. At the end (when the action finishes), the *taking-off* state is removed and the quadrotor enters in *hovering* state. The duration has been set to fifteen seconds.

```
(:durative-action TAKE-OFF
  :parameters
  (?uav - quadrotor)
  :duration
  (= ?duration 15)
  :condition
  (and
    (at start (landed ?uav))
    (over all (taking_off ?uav))
    (over all (>= (- (battery ?uav) ?duration) 0))
  )
  :effect
  (and
    (at start (decrease (battery ?uav) ?duration))
    (at start (not (landed ?uav)))
    (at start (taking_off ?uav))
    (at end (not (taking_off ?uav)))
    (at end (hovering ?uav))
  )
)
```

Code 4.3 *Land* durative action. The quadrotor must be hovering at the start of the action, and during the action its state must be *landing* and its battery must be equal or greater than zero. If these conditions are met, then the state of hovering is removed at the start, the *landing* state is set and the battery is decreased. At the end (when the action finishes), the *landing* state is removed and the quadrotor enters in a *landed* state. The duration has been set to fifteen seconds.

```
(:durative-action LAND
  :parameters
  (?uav - quadrotor)
  :duration
  (= ?duration 15)
  :condition
  (and
    (at start (hovering ?uav))
    (over all (landing ?uav))
    (over all (>= (- (battery ?uav) ?duration) 0))
  )
  :effect
  (and
    (at start (decrease (battery ?uav) ?duration))
    (at start (landing ?uav))
    (at start (not (hovering ?uav)))
    (at end (not (landing ?uav)))
    (at end (landed ?uav))
  )
)
```

Code 4.4 *Move* durative action. The quadrotor must be hovering at the start of the action, must be located in the start location, and during the action its state must be *moving* and its battery must be equal or greater than zero. If these conditions are met, then the state of hovering is removed at the start, the quadrotor is no longer located at the start location, the moving state is set and the battery is decreased. At the end (when the action finishes), the moving state is removed, the quadrotor enters in a hovering state and finishes located at the end position. The duration has been computed from the distance to travel and the average speed of the quadrotor.

```
(:durative-action MOVE
  :parameters
  (?uav - quadrotor ?from ?to - location)
  :duration
  (= ?duration (/ (distance ?from ?to) (speed ?uav)))
  :condition
  (and
    (at start (hovering ?uav))
    (at start (at ?uav ?from))
    (over all (moving ?uav))
    (over all (>= (- (battery ?uav) ?duration) 0))
  )
  :effect
  (and
    (at start (decrease (battery ?uav) ?duration))
    (at start (moving ?uav))
    (at start (not (hovering ?uav)))
    (at start (not (at ?uav ?from)))
    (at end (not (moving ?uav)))
    (at end (hovering ?uav))
    (at end (at ?uav ?to))
  )
)
```

Code 4.5 *Pick* durative action. The quadrotor must be hovering at the start of the action, it must not be transporting any part, the part must be located on its pick location and not assembled. During the action, the state of the quadrotor must be *picking* and its battery must be equal or greater than zero. If these conditions are met, then the state of *hovering* is removed at the start, the part is no longer located at its pick location, the *picking* state is set and the battery is decreased. At the end (when the action finishes), the *picking* state is removed, the quadrotor enters in *hovering* state and the part finishes located at the quadrotor. The duration has been set to twenty seconds.

```
(:durative-action PICK
  :parameters
  (?uav - quadrotor ?prt - part)
  :duration
  (= ?duration 20)
  :condition
  (and
    (at start
      (exists (?loc - pick_location)
        (and ((at ?prt ?loc)(at ?uav ?loc)))))
    (at start (not (exists (?piece - part) (at ?piece ?uav))))
    (at start (hovering ?uav))
    (at start (not (assembled ?prt)))
    (over all (picking ?uav ?prt))
    (over all (>= (- (battery ?uav) ?duration) 0))
  )
  :effect
  (and
    (at start (decrease (battery ?uav) ?duration))
    (at start (picking ?uav ?prt))
    (at start (not (hovering ?uav)))
    (at start (not (at ?prt ?loc)))
    (at end (not (picking ?uav ?prt)))
    (at end (hovering ?uav))
    (at end (at ?prt ?uav))
  )
)
```

Code 4.6 *Dual Pick* durative action. This action has been defined to model the case of a part that needs to be picked by two robots due to its weight. The conditions and effects that appeared in Code 4.5 are now duplicated, having one for each of the vehicles. When the action is finished, the part is considered to be at both vehicles. The duration has been set to thirty seconds.

```
(:durative-action DUAL_PICK
  :parameters
  (?uav1 ?uav2 - quadrotor ?prt - part)
  :duration
  (= ?duration 30)
  :condition
  (and
    (at start
      (exists (?loc - pick_location)
        (and ((at ?prt ?loc)(at ?uav1 ?loc))(at ?uav2 ?loc))))
    (at start (not (exists (?piece1 - part) (at ?piece1 ?uav1))))
    (at start (not (exists (?piece2 - part) (at ?piece2 ?uav2))))
    (at start (hovering ?uav1))
    (at start (hovering ?uav2))
    (at start (not (assembled ?prt)))
    (over all (picking ?uav1 ?prt))
    (over all (picking ?uav2 ?prt))
    (over all (>= (- (battery ?uav1) ?duration) 0))
    (over all (>= (- (battery ?uav2) ?duration) 0))
  )
  :effect
  (and
    (at start (decrease (battery ?uav1) ?duration))
    (at start (decrease (battery ?uav2) ?duration))
    (at start (picking ?uav1 ?prt))
    (at start (picking ?uav2 ?prt))
    (at start (not (hovering ?uav1)))
    (at start (not (hovering ?uav2)))
    (at start (not (at ?prt ?loc)))
    (at end (not (picking ?uav1 ?prt)))
    (at end (not (picking ?uav2 ?prt)))
    (at end (hovering ?uav1))
    (at end (hovering ?uav2))
    (at end (at ?prt ?uav1))
    (at end (at ?prt ?uav2))
  )
)
```

Code 4.7 *Place* durative action. The quadrotor must be hovering at the start of the action, it must be transporting the part and located on its assembly location, and the part must have all its dependencies assembled. During the action, the state of the quadrotor must be *placing* and its battery must be equal or greater than zero. If these conditions are met, then the state of *hovering* is removed at the start, the part is no longer located at the quadrotor, the *placing* state is set and the battery is decreased. At the end (when the action finishes), the *placing* state is removed, the quadrotor enters in *hovering* state and the part finishes assembled at its assembly location. The number of remaining parts is decreased in one unit. The duration has been set to twenty seconds.

```
(:durative-action PLACE
  :parameters
  (?uav - quadrotor ?prt - part)
  :duration
  (= ?duration 20)
  :condition
  (and
    (at start
      (exists (?loc - assembly_location)
        (and ((at ?uav ?loc)(assemble_at ?prt ?loc)))))
    (at start (at ?prt ?uav))
    (at start
      (forall (?prt2 - part)
        (imply (depends ?prt ?prt2) (assembled ?prt2))))
    (at start (hovering ?uav))
    (at start (not (assembled ?prt)))
    (over all (placing ?uav ?prt))
    (over all (>= (- (battery ?uav) ?duration) 0))
  )
  :effect
  (and
    (at start (decrease (battery ?uav) ?duration))
    (at start (placing ?uav ?prt))
    (at start (not (hovering ?uav)))
    (at start (not (at ?prt ?uav)))
    (at end (not (placing ?uav ?prt)))
    (at end (hovering ?uav))
    (at end (at ?prt ?loc))
    (at end (assembled ?prt))
    (at end (decrease (remaining_parts) 1))
  )
)
```

Code 4.8 *Dual Place* durative action. This action has been defined to model the case of a part that needs to be placed by two vehicles due to its weight. The conditions and effects that appeared in Code 4.7 are now duplicated, having one for each of the vehicles. When the action starts, the part is considered to be at both vehicles. The duration has been set to thirty seconds.

```
(:durative-action DUAL_PLACE
  :parameters
  (?uav1 ?uav2 - quadrotor ?prt - part)
  :duration
  (= ?duration 30)
  :condition
  (and
    (at start
      (exists (?loc - assembly_location)
        (and ((at ?uav1 ?loc)(at ?uav2 ?loc)(assemble_at ?prt ?loc)))))
    (at start (at ?prt ?uav1))
    (at start (at ?prt ?uav2))
    (at start
      (forall (?prt2 - part)
        (imply (depends ?prt ?prt2) (assembled ?prt2))))
    (at start (hovering ?uav1))
    (at start (hovering ?uav2))
    (at start (not (assembled ?prt)))
    (over all (placing ?uav1 ?prt))
    (over all (placing ?uav2 ?prt))
    (over all (>= (- (battery ?uav1) ?duration) 0))
    (over all (>= (- (battery ?uav2) ?duration) 0))
  )
  :effect
  (and
    (at start (decrease (battery ?uav1) ?duration))
    (at start (decrease (battery ?uav2) ?duration))
    (at start (placing ?uav1 ?prt))
    (at start (placing ?uav2 ?prt))
    (at start (not (hovering ?uav1)))
    (at start (not (hovering ?uav2)))
    (at start (not (at ?prt ?uav1)))
    (at start (not (at ?prt ?uav2)))
    (at end (not (placing ?uav1 ?prt)))
    (at end (not (placing ?uav2 ?prt)))
    (at end (hovering ?uav1))
    (at end (hovering ?uav2))
    (at end (at ?prt ?loc))
    (at end (assembled ?prt))
    (at end (decrease (remaining_parts) 1))
  )
)
```

Code 4.9 *Wait* durative action. The duration of the action is bounded to be in the range (0-1). A planner could choose a value in the given range to make a locked agent wait some time until it is capable of executing any other action, and repeating the wait if necessary. As specified in the conditions, at the start and during the wait, the quadrotor must not be doing any other action that implies some kind of move, such as landing or taking-off. The *hovering* state is allowed, as a quadrotor can wait in this state. The effect part of the action is empty, as waiting only affects the increase of the time variable.

```
(:durative-action WAIT
  :parameters
  (?uav - quadrotor)
  :duration
  (and (<= ?duration 1) (> ?duration 0))
  :condition
  (and
    (at start (not (landing ?uav))
    (at start (not (taking_off ?uav))
    (at start (not (moving ?uav))
    (at start (not (exists (?prt - part)(picking ?uav ?prt))
    (at start (not (exists (?prt - part)(placing ?uav ?prt))
    (over all (not (landing ?uav))
    (over all (not (taking_off ?uav))
    (over all (not (moving ?uav))
    (over all (not (exists (?prt - part)(picking ?uav ?prt))
    (over all (not (exists (?prt - part)(placing ?uav ?prt))
    (over all (>= (- (battery ?uav) ?duration) 0))
  )
  :effect
  (
  )
)
```

4.4.2 Problem Description

Once a planning domain has been defined, then a problem description is needed to make a specific instance of a planning problem.

The PDDL problem description contains the goal specification and the instances of the entities, along with the initial states and values that conform a specific problem instance that has to be solved. Thus, the PDDL problem description is used to define the initial state at which a planning problem instance starts, as well as the goals that have to be achieved.

Every PDDL problem description is linked with a PDDL domain definition that tells the planners how to solve the problem instances. Thus, a reference to the domain file must appear in all the problem description instances.

A PDDL problem description is mainly composed by four parts: the declaration of objects, the initialization of predicates, the initialization of functions and the goal and metrics specifications. Codes 4.10, 4.11, 4.12 and 4.13 show some examples on how these elements are defined.

Code 4.10 Objects declaration for a PDDL example problem. The different quadrotors, parts and locations that are present in the problem instance are declared.

```
(:objects
  uav1 uav2 - quadrotor
  prt1 prt2 prt3 prt4 - part
  loc1 loc2 - location
  loc3 loc4 loc5 loc6 - pick_location
  loc7 loc8 loc9 loc10 - assembly_location
)
```

Code 4.11 Initialization of values for the different predicates that conform the initial state of a PDDL example problem. Uninitialized predicates are supposed to be false.

```
(:init
  (at uav1 loc1) (at uav2 loc2)
  (landed uav1) (landed uav2)
  (at prt1 loc3)...(at prt4 loc6)
  (assemble_at prt1 loc7)...(assemble_at prt4 loc10)
  (assigned prt1 uav1)
  (not (assembled prt1))
  ...
  (depends prt3 prt1)
  (depends prt3 prt2)
  ...
)
```

Code 4.12 Initialization of function values for a PDDL example problem. Functions are treated as predicates, so they are defined in the same *:init* section of the problem description.

```
(= (speed uav1) 1)
(= (speed uav2) 1)
(= (battery uav1) 600)
(= (battery uav2) 600)
(= (distance loc1 loc2) 10)
...
(= (distance loc10 loc9) 15)
(= (remaining_parts) 4)
```

Code 4.13 Goal and metric declaration for a PDDL example problem. The goal represents the logical expression that must be true in order to consider the problem as solved. The metrics are additional values that can be used to measure the quality of the solutions. For the PDDL domain designed in the previous subsection, this definition remains unaltered among all the problem instances, as all have as goals assembling all the parts minimizing the total assembly time.

```
; goal definition
(:goal (== (remaining_parts) 0))
; metric definition
(:metric minimize (total-time))
```

4.4.3 SHOP2 Problem Domain

The JSHOP2 domain designed to solve the decomposition and scheduling problem described in Section 4.2 is presented here. This domain is a translation of the PDDL domain presented in the previous section. As stated before in this chapter, PDDL gives a language to formalize and define a planning domain, but the problem must be solved by a planner. Because of this, some additional elements have been added to the domain definition. These elements represent the knowledge needed by JSHOP2 to solve the planning problems, and are basically the definition of the different methods (task networks) that will be used to decompose the high-level tasks to obtain the final plan.

To explain the designed JSHOP2 domain, a top-down approach will be used. First, the high-level task that represents the problem to solve will be presented. This task represents the assembly of a structure composed by multiple parts. The task of assembling a complete structure can be seen as multiple subtasks consisting on the assembly of single parts. To decompose the high-level task into subtasks, a recursive method has been defined to try to assemble a single part on each call. This recursive method calls a lower-level method which in turns try to decompose a subtask consisting on the assembly of a single part into lower-level subtasks related to the operations that the different aerial vehicles need to effectively assemble the part, such as taking-off, moving, picking or placing the part, etc. Each of these operations is also represented by a method that finally calls a JSHOP2 operator and finishes the decomposition.

High-level Method Definition

In JSHOP2, the high-level tasks represent the goals to accomplish. For our domain definition, only one high-level task has been defined. This task represents the assembly of the whole structure, which is composed of several parts and is considered the goal to reach in the planning process. The planning process ends when this task has been completely decomposed and thus, a low-level plan has been computed.

The task of assembling a structure can be divided on smaller subtasks consisting on the assembly of single parts. With the purpose of decomposing the high-level task on these smaller subtasks, the high-level method shown in Code 4.14 has been applied. The method has been defined to be recursive, so that on each call it selects one part from the set of parts that can be eligible to be assembled, tries to assemble that part and again makes a call to itself. The method have three pairs *preconditions-subtasks*, that cover the following cases:

- The first pair covers the case of a part that needs to be assembled by using two aerial vehicles. If there is a part that has been assigned to two different aerial vehicles and it has all its preconditions met, then it is selected to be assembled and the recursive method is called again.
- The second pair covers the case of a part that needs to be assembled by using only one aerial vehicle. If there is a part that has been assigned to one aerial vehicle and it has all its preconditions met, then it is selected to be assembled and the recursive method is called again.
- The third pair covers the case of having all the assembly tasks done. There are no parts left to be assembled, so no more recursive calls are made and an operator is called to exit and flag the finish of the planning process.

The high-level method shown in Code 4.14 is a simplified version of the final method implemented for the domain. In this simplified method, the only requirements needed to choose a part is that the chosen part has all its preconditions met. If all the parts that are present in its precondition list are assembled, then that part can be chosen to be assembled. However, that way of selecting parts does not lead to optimal plans and do not minimize the total assembly time. An example of this can be seen in Figure 4.4.

To minimize the total assembly time, a priority has been established among all the parts that can be chosen to be assembled. From that set, those parts that are known to 'unlock' other parts after being assembled are chosen first. By this way, situations as the shown in Figure 4.4 are avoided and thus, the potential parallelism of using multiple aerial vehicles is increased. Code 4.15 shows the changes made to the simplified high-level method to establish a priority among all the eligible parts.

Code 4.14 Simplified high-level method definition for the decomposition of the high-level task. The high-level task consists on the assembly of a complete structure, composed of several parts. The method to decompose that task has been defined to be recursive, so that on each call it selects from the set of parts, one that satisfies any of the *preconditions-subtasks* pairs.

```
(:method (mission ?missionType)
  ;preconditions
  ((remaining_tasks ?remaining)(call > ?remaining 0)
   (object ?part) (not(assembled ?part))
   (depends ?part ?dependency_list)
   (forall (?otherPart)((object ?otherPart) (call Member ?otherPart ?
        dependency_list)) (assembled ?otherPart))
   (location ?loc ?x1 ?y1 ?z1) (assembly_location ?part ?loc)
   (quadrotor ?uav1) (quadrotor ?uav2) (call != ?uav1 ?uav2)
   (assigned ?part ?uav1) (assigned ?part ?uav2))

  ;subtasks
  ((assemble ?uav1 ?uav2 ?loc ?part)
   (mission ?missionType))

  ;preconditions
  ((remaining_tasks ?remaining)
   (call > ?remaining 0)
   (object ?part) (not(assembled ?part))
   (depends ?part ?dependency_list)
   (forall (?otherPart)((object ?otherPart) (call Member ?otherPart ?
        dependency_list)) (assembled ?otherPart))
   (quadrotor ?uav) (assigned ?part ?uav)
   (location ?loc ?x1 ?y1 ?z1) (assembly_location ?part ?loc))

  ;subtasks
  ((assemble ?uav ?loc ?part)
   (mission ?missionType))

  ;preconditions
  ((remaining_tasks 0))

  ;subtasks
  ((!assembly_plan_finished))
)
```

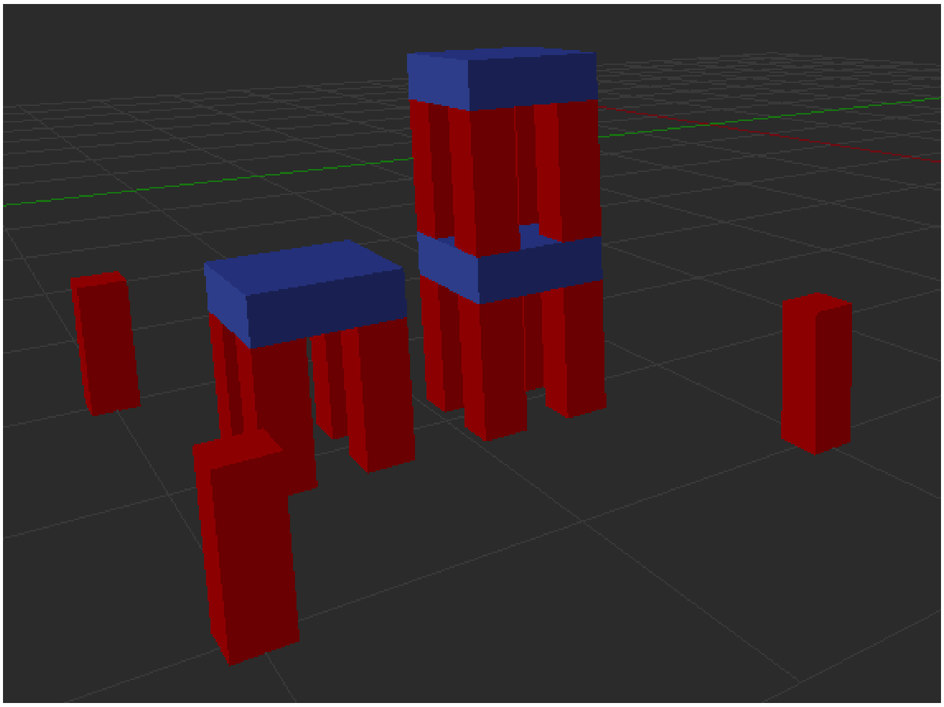


Figure 4.4 Example structure showing the importance of establishing a priority when selecting the parts. The outer parts can be selected and assembled at any time, as they do not depend on any other part. However, selecting them first will cause that other parts that are needed for other robots to start their work will be assembled later. That is the case of the blue parts, as they need to wait until all the parts below them are assembled. If the parts below the blue parts are chosen to be assembled after the outer parts, then the total assembly time will be increased as some vehicles will be waiting more time to start their work and the potential parallelism of using multiple vehicles will not be exploited.

Code 4.15 Modified high-level method definition for the decomposition of the high-level task by using priorities. The pairs *preconditions-subtasks* of a JSHOP2 method are analogous to an *if-then-else* construct, and thus the first pairs are checked first. By this way, it is possible to model preconditions that give priority to a subset of parts. In that case, the pairs added before those already shown in Code 4.14 serve to choose first the parts that have all its preconditions met but also that are known to be dependencies for other parts that are not assembled yet. In this manner, the unlocking of other parts is favoured.

```
(:method (mission ?missionType)
  ;precondition
  ((remaining_tasks ?remaining) (call > ?remaining 0)
   (object ?part) (not(assembled ?part))
   (depends ?part ?dependency_list)
   (location ?loc ?x1 ?y1 ?z1) (assembly_location ?part ?loc)
   (forall (?otherPart)((object ?otherPart) (call Member ?otherPart ?
        dependency_list)) (assembled ?otherPart))
   (quadrotor ?uav1) (quadrotor ?uav2) (call != ?uav1 ?uav2)
   (assigned ?part ?uav1) (assigned ?part ?uav2)
   (object ?part2) (call != ?part ?part2)
   (not(assembled ?part2))
   (depends ?part2 ?dependency_list2)
   (call Member ?part ?dependency_list2))

  ;subtasks
  ((assemble ?uav1 ?uav2 ?loc ?part)
   (mission ?missionType))

  ;precondition
  ((remaining_tasks ?remaining) (call > ?remaining 0)
   (object ?part) (not(assembled ?part))
   (depends ?part ?dependency_list)
   (location ?loc ?x1 ?y1 ?z1) (assembly_location ?part ?loc)
   (forall (?otherPart)((object ?otherPart) (call Member ?otherPart ?
        dependency_list)) (assembled ?otherPart))
   (quadrotor ?uav) (assigned ?part ?uav)
   (object ?part2) (call != ?part ?part2)
   (not(assembled ?part2))
   (depends ?part2 ?dependency_list2)
   (call Member ?part ?dependency_list2))

  ;subtasks
  ((assemble ?uav ?loc ?part)
   (mission ?missionType))
  ...
)
```

Submethods Definition

Previously, a high-level method defined to do the decomposition of the high-level task was presented. This task was decomposed by this method into several subtasks consisting on the assembly of single parts. These subtasks must be also decomposed to obtain a lower level plan, so other submethods have been defined to do the decomposition.

Code 4.16 shows the submethods defined to decompose a task consisting in the assembly of a single part into several subtasks. In the previous case, where a task representing the assembly of a complete structure was decomposed into several subtasks consisting in assembling single parts, the different aerial vehicles did not appear in the methods definition. In this case the different aerial vehicles appear, as they are needed to execute the assembly of single parts. Thus, the submethods are decomposed again into several submethods that represent the different operations that the aerial vehicles can execute and that combined, can lead to the execution of the assembly tasks. Each of these have been also defined in the JSHOP2 domain. However, for the sake of simplicity, most of them are not showed in this chapter, as their preconditions are the same as the preconditions that appear in the operators they call. They have been implemented as a 'bridge' between the method that decomposes the single assembly tasks and the operators that represent the aerial robot's actions. Code 4.17 shows the implementation of the *synchro_wait* submethod.

Code 4.16 Methods definition for the decomposition of the high-level task into subtasks. The upper method is for the case of a part that can be transported by a single aerial robot. The method checks that the part is in a specific location, not assembled and that all the parts it depends on are assembled, and also that the aerial vehicle is not transporting any other part. The method is then decomposed into several submethods, consisting on moving the robot to the part location, picking the part, moving to the assembly location and placing the part. The method below corresponds to the case of a part that must be transported by two aerial vehicles. Its submethods are cooperative versions of the formers.

```
(:method (assemble ?uav ?targetLocation ?part)
  ;precondition
  ((quadrotor ?uav) (object ?part)
   (location ?loc ?x1 ?y1 ?z1) (at ?part ?loc)
   (location ?targetLocation ?x2 ?y2 ?z2) (not (transporting ?uav))
   (not (transported ?part)) (not (assembled ?part))
   (depends ?part ?dependency_list)
   (forall (?otherPart)((object ?otherPart) (call Member ?otherPart ?
        dependency_list))
    (assembled ?otherPart)))
  ;subtasks
  ((move ?uav ?loc) (pick ?uav ?part)
   (move ?uav ?targetLocation) (place ?uav ?targetLocation ?part)))

(:method (assemble ?uav1 ?uav2 ?targetLocation ?part)
  ;precondition
  ((quadrotor ?uav1) (quadrotor ?uav2) (object ?part)
   (location ?loc ?x1 ?y1 ?z1) (at ?part ?loc)
   (location ?targetLocation ?x2 ?y2 ?z2) (not (transporting ?uav1))
   (not (transporting ?uav2)) (not (transported ?part))
   (not (assembled ?part)) (depends ?part ?dependency_list)
   (forall (?otherPart)((object ?otherPart) (call Member ?otherPart ?
        dependency_list))
    (assembled ?otherPart)))
  ;subtasks
  ((move ?uav1 ?loc) (move ?uav2 ?loc)
   (synchro_wait ?uav1 ?uav2 )
   (pick ?uav1 ?uav2 ?part)
   (move ?uav1 ?uav2 ?targetLocation)
   (place ?uav1 ?uav2 ?targetLocation ?part)
  )
)
```

Code 4.17 Submethod definition for the *synchro_wait* task. As the rest of the submethods that represent the different operations of the aerial vehicles, this submethod serves as a bridge between the method that decomposes the single assembly tasks and the operators that represent the aerial vehicle actions, in this case the *sync* action that will be better explained later. This action is needed in cooperative operations, where the aerial vehicles must synchronize to be at the same location at the same time and to coordinate to execute other operations. If one aerial vehicle is placed in a specific location and the other vehicle is needed to be at this location to execute between both a cooperative operation, then the first vehicle must wait until the second finishes its current operation and arrives. As JSHOP2 plans for the tasks in the order they appear, at this moment of the search it is possible for the planner to know if the second vehicle is executing a previous operation. Also, as the MTP technique explained in 3.3.5 is used in our domain, the start time and duration of this previous operation is already known, so based on these two timestamps the planner can estimate the time on which the second vehicle will arrive to the location where the first vehicle is, and the *wait time* for the first vehicle can be computed. All this can be seen on the preconditions of the method, which uses two pairs preconditions-subtasks to know which of the vehicles must wait. The *write-times* of the *at* property for both vehicles are checked. In a specific moment, the vehicle who has the greater write-time will not be available until that time is reached by the global timeline, so if the vehicle with lower write-time wants to do a cooperative operation then it will have to wait until the write-time of the second vehicle. As it can be seen, the *start* and *duration* times for the *synchro_wait* task are computed in the preconditions and are sent to the called operator so that it will be correctly scheduled.

```
(:method (synchro_wait ?uav1 ?uav2)
  ;precondition
  ((quadrotor ?uav1) (quadrotor ?uav2)
   (write-time at ?uav1 ?t1) (write-time at ?uav2 ?t2)
   (call >= ?t1 ?t2) (assign ?duration (call - ?t1 ?t2))
   (assign ?start ?t2) (battery ?uav1 ?r1) (battery ?uav2 ?r2)
   (call >= (call - ?r2 ?duration) 0))

  ;subtasks
  ((!sync ?uav2 ?uav1 ?start ?duration)
   (!sync ?uav1 ?uav2 ?t1 0))

  ;precondition
  ((quadrotor ?uav1) (quadrotor ?uav2)
   (write-time at ?uav1 ?t1) (write-time at ?uav2 ?t2)
   (call < ?t1 ?t2) (assign ?duration (call - ?t2 ?t1))
   (assign ?start ?t1) (battery ?uav1 ?r1) (battery ?uav2 ?r2)
   (call >= (call - ?r1 ?duration) 0))

  ;subtasks
  ((!sync ?uav1 ?uav2 ?start ?duration)
   (!sync ?uav2 ?uav1 ?t2 0))
)
```

Operators Definition

The operators defined for the JSHOP2 domain are presented in the codes below. Operators represent tasks at the lowest-level of the task hierarchy, and are defined to model tasks that can be executed directly.

The operators presented here are a direct translation of the PDDL operators that were defined in Subsection 4.4.1 as *durative-actions*. The purpose of our domain is to solve planning problems not only by decomposing high-level tasks to lower-level tasks, but also by computing the scheduling of the resulting lower-level tasks. This requires that each of the lowest-level tasks must have two timestamps, one to mark the start of the operation and one to tell its duration so that the start and end times for the operation are well defined. If all the operators have these two timestamps defined, then the resulting plan will be correctly scheduled.

As it was explained in Code 4.17, the methods that are above the operators in the hierarchy compute the start and duration times on its preconditions and, after that, they call the operators by passing these values as arguments. This is one of the steps required to apply the MTP technique explained in Section 3.3.5. The second step is done within the operators. They use these two values with two purposes:

- The first purpose is to check if the battery will last until the end of the operation. If not, or if the rest of preconditions for the operator are not met, then it is not possible to apply the operator. The failing will be notified to the upper level method and the JSHOP2 planning engine will start a backtracking process to try to find another task decomposition and another state that leads to a correct operator execution.
- The second purpose is to update the *read-time* and *write-time* of all the dynamic properties that are used by the operator. Two dynamic properties are defined in our domain: the *at* predicate and the *battery* resource, that is modelled also as a predicate. The update of dynamic properties is important to generate a valid scheduling and avoid concurrent access to these properties, which will lead to operators that overlap in time and change predicate values while other operators are using them.

Codes 4.18, 4.19 and 4.20 show the *take-off*, *land* and *move* operators. Codes 4.22, 4.23, 4.24 and 4.25 show the *pick* and *place* operators in its versions for one and two vehicles. Finally, Codes 4.26 and 4.27 show the *sync* and *finish* operators.

Code 4.18 *Take-off* operator defined for the domain.

```
(:operator (!takeoff ?uav ?start ?duration)
  ;precondition
  ((quadrotor ?uav) (landed ?uav)
   (battery ?uav ?r) (read-time battery ?uav ?t0)
   (write-time battery ?uav ?t1)
   (assign ?end (call + ?start ?duration))
   (call >= (call - ?r ?duration) 0))

  ;delete list
  ((landed ?uav) (battery ?uav ?r)
   (read-time battery ?uav ?t0)
   (write-time battery ?uav ?t1))

  ;add list
  ((hovering ?uav) (battery ?uav (call - ?r ?duration))
   (read-time battery ?uav ?end)
   (write-time battery ?uav ?end))
)
```

Code 4.19 *Land* operator defined for the domain.

```
(:operator (!land ?uav ?start ?duration)
  ;precondition
  ((quadrotor ?uav) (hovering ?uav)
   (battery ?uav ?r) (read-time battery ?uav ?t0)
   (write-time battery ?uav ?t1)
   (assign ?end (call + ?start ?duration))
   (call >= (call - ?r ?duration) 0))

  ;delete list
  ((hovering ?uav) (battery ?uav ?r)
   (read-time battery ?uav ?t0)
   (write-time battery ?uav ?t1))

  ;add list
  ((landed ?uav) (battery ?uav (call - ?r ?duration))
   (read-time battery ?uav ?end)
   (write-time battery ?uav ?end))
)
```

Code 4.20 *Move* operator defined for the domain. The result of applying the operator is the aerial vehicle being placed at the end location.

```
(:operator (!move ?uav ?source ?destination ?start ?duration)
  ;precondition
  ((quadrotor ?uav) (hovering ?uav)
   (location ?source ?x1 ?y1 ?z1) (location ?destination ?x2 ?y2 ?z2)
   (at ?uav ?source) (battery ?uav ?r)
   (read-time battery ?uav ?t0) (write-time battery ?uav ?t1)
   (read-time at ?uav ?t2) (write-time at ?uav ?t3)
   (assign ?end (call + ?start ?duration))
   (call >= (call - ?r ?duration) 0))

  ;delete list
  ((at ?uav ?source) (battery ?uav ?r)
   (read-time battery ?uav ?t0) (write-time battery ?uav ?t1)
   (read-time at ?uav ?t2) (write-time at ?uav ?t3))

  ;add list
  ((at ?uav ?destination)
   (battery ?uav (call - ?r ?duration))
   (read-time battery ?uav ?end) (write-time battery ?uav ?end)
   (read-time at ?uav ?end) (write-time at ?uav ?end))
)
```

Code 4.21 Two-vehicles version of the *move* operator. This operator has been specially defined to cover the case of one part that is being transported by two aerial vehicles. In that case, the part is supposed to be at both vehicles and they must move in a synchronized way along all the path while carrying the part. Thus, this operator is only used after a *dual-pick* operator execution.

```
(:operator (!move ?uav1 ?uav2 ?source ?destination ?start ?duration)
  ;precondition
  ((quadrotor ?uav1) (hovering ?uav1)
   (quadrotor ?uav2) (hovering ?uav2)
   (location ?source ?x1 ?y1 ?z1)
   (location ?destination ?x2 ?y2 ?z2)
   (at ?uav1 ?source) (at ?uav2 ?source)
   (battery ?uav1 ?r1)
   (read-time battery ?uav1 ?t0) (write-time battery ?uav1 ?t1)
   (read-time at ?uav1 ?t2) (write-time at ?uav1 ?t3)
   (battery ?uav2 ?r2) (read-time battery ?uav2 ?t4) (write-time battery ?uav2 ?t5)
   (read-time at ?uav2 ?t6) (write-time at ?uav2 ?t7)
   (assign ?end (call + ?start ?duration))
   (call >= (call - ?r1 ?duration) 0) (call >= (call - ?r2 ?duration) 0))

  ;delete list
  ((at ?uav1 ?source) (at ?uav2 ?source)
   (battery ?uav1 ?r1) (battery ?uav2 ?r2)
   (read-time battery ?uav1 ?t0) (write-time battery ?uav1 ?t1)
   (read-time at ?uav1 ?t2) (write-time at ?uav1 ?t3)
   (read-time battery ?uav2 ?t4) (write-time battery ?uav2 ?t5) (read-time at ?uav2 ?t6) (write-time at ?uav2 ?t7))

  ;add list
  ((at ?uav1 ?destination) (at ?uav2 ?destination) (battery ?uav1 (call - ?r1 ?duration)) (battery ?uav2 (call - ?r2 ?duration))
   (read-time battery ?uav1 ?end) (write-time battery ?uav1 ?end) (read-time at ?uav1 ?end) (write-time at ?uav1 ?end)
   (read-time battery ?uav2 ?end) (write-time battery ?uav2 ?end) (read-time at ?uav2 ?end) (write-time at ?uav2 ?end))

)
```

Code 4.22 *Pick* operator defined for the domain. The applying of this operator results in one single part being picked by a single aerial vehicle.

```
(:operator (!pick ?uav ?part ?start ?duration)
  ;precondition
  ((quadrotor ?uav) (object ?part)
   (hovering ?uav) (location ?loc ?x1 ?y1 ?z1)
   (at ?part ?loc) (at ?uav ?loc)
   (not (transported ?part)) (not (transporting ?uav))
   (assign ?end (call + ?start ?duration))
   (read-time at ?uav ?t0) (read-time at ?part ?t1)
   (write-time at ?part ?t2)
   (battery ?uav ?r) (call >= (call - ?r ?duration) 0))

  ;delete list
  ((read-time at ?uav ?t0) (read-time at ?part ?t1)
   (write-time at ?part ?t2) (at ?part ?loc)
   (battery ?uav ?r))

  ;add list
  ((transporting ?uav) (transported ?part)
   (at ?part ?uav) (read-time at ?uav ?end)
   (read-time at ?part ?end)
   (write-time at ?part ?end)
   (battery ?uav (call - ?r ?duration)))
)
```

Code 4.23 Two-vehicles version of the *pick* operator.

```
(:operator (!pick_object ?uav1 ?uav2 ?part ?start ?duration)
  ;precondition
  ((quadrotor ?uav1) (quadrotor ?uav2)
   (object ?part) (hovering ?uav1)
   (hovering ?uav2)
   (location ?loc ?x1 ?y1 ?z1) (at ?part ?loc)
   (at ?uav1 ?loc) (at ?uav2 ?loc)
   (not (transported ?part)) (not (transporting ?uav1))
   (not (transporting ?uav2))
   (assign ?end (call + ?start ?duration))
   (read-time at ?uav1 ?t1) (read-time at ?uav2 ?t2)
   (read-time at ?part ?t3) (write-time at ?part ?t4)
   (battery ?uav1 ?r1) (call >= (call - ?r1 ?duration) 0)
   (battery ?uav2 ?r2) (call >= (call - ?r2 ?duration) 0))

  ;delete list
  ((read-time at ?uav1 ?t1) (read-time at ?uav2 ?t2)
   (read-time at ?part ?t3)
   (write-time at ?part ?t4) (at ?part ?loc)
   (battery ?uav1 ?r1) (battery ?uav2 ?r2))

  ;add list
  ((transporting ?uav1) (transporting ?uav2)
   (transported ?part) (at ?part ?uav1)
   (at ?part ?uav2)
   (read-time at ?uav1 ?end) (read-time at ?uav2 ?end)
   (read-time at ?part ?end) (write-time at ?part ?end)
   (battery ?uav1 (call - ?r1 ?duration)) (battery ?uav2 (call - ?r2 ?duration))
       )
)
```

Code 4.24 *Place* operator defined for the domain. The applying of this operator results in one single part being assembled.

```
(:operator (!place ?uav ?loc ?objectName ?start ?duration)
  ;precondition
  ((quadrotor ?uav) (object ?objectName)
   (hovering ?uav) (location ?loc ?x1 ?y1 ?z1)
   (at ?uav ?loc)
   (transported ?objectName) (at ?objectName ?uav)
   (transporting ?uav)
   (depends ?objectName ?dependency_list)
   (forall (?z)((object ?z) (call Member ?z ?dependency_list))
    (done_with ?z))
   (read-time at ?uav ?t0) (read-time at ?objectName ?t1)
   (write-time at ?objectName ?t2)
   (assign ?end (call + ?start ?duration))
   (remaining_tasks ?remaining)
   (call > ?remaining 0) (assign ?left (call - ?remaining 1))
   (battery ?uav ?r) (call >= (call - ?r ?duration) 0))

  ;delete list
  ((read-time at ?uav ?t0) (read-time at ?objectName ?t1)
   (write-time at ?objectName ?t2)
   (transported ?objectName)
   (at ?objectName ?uav) (transporting ?uav)
   (remaining_tasks ?remaining) (battery ?uav ?r))

  ;add list
  ((read-time at ?uav ?end) (read-time at ?objectName ?end)
   (write-time at ?objectName ?end) (at ?objectName ?loc)
   (done_with ?objectName)
   (remaining_tasks ?left) (battery ?uav (call - ?r ?duration)))
)
```

Code 4.25 Two-vehicles version of the *place* operator.

```
(:operator (!place ?uav1 ?uav2 ?loc ?part ?start ?duration)
  ;precondition
  ((quadrotor ?uav1) (quadrotor ?uav2)
   (object ?part) (hovering ?uav1)
   (hovering ?uav2) (location ?loc ?x1 ?y1 ?z1)
   (at ?uav1 ?loc) (at ?uav2 ?loc)
   (transported ?part) (at ?part ?uav1)
   (at ?part ?uav2) (transporting ?uav1) (transporting ?uav2)
   (depends ?part ?dependency_list)
   (forall (?z)((object ?z) (call Member ?z ?dependency_list)) (done_with ?z))
   (read-time at ?uav1 ?t0) (read-time at ?uav2 ?t1)
   (read-time at ?part ?t2) (write-time at ?part ?t3)
   (assign ?end (call + ?start ?duration))
   (remaining_tasks ?remaining)
   (call > ?remaining 0) (assign ?left (call - ?remaining 1))
   (battery ?uav1 ?r1) (call >= (call - ?r1 ?duration) 0)
   (battery ?uav2 ?r2) (call >= (call - ?r2 ?duration) 0))

  ;delete list
  ((read-time at ?uav1 ?t0) (read-time at ?uav2 ?t1)
   (read-time at ?part ?t2) (write-time at ?part ?t3)
   (transported ?part)
   (at ?part ?uav1) (at ?part ?uav2)
   (transporting ?uav1) (transporting ?uav2)
   (remaining_tasks ?remaining)
   (battery ?uav1 ?r1) (battery ?uav2 ?r2))

  ;add list
  ((read-time at ?uav1 ?end) (read-time at ?uav2 ?end)
   (read-time at ?part ?end) (write-time at ?part ?end)
   (at ?part ?loc) (done_with ?part)
   (remaining_tasks ?left)
   (battery ?uav1 (call - ?r1 ?duration))
   (battery ?uav2 (call - ?r2 ?duration)))
)
```

Code 4.26 *Sync* operator defined for the domain. The purpose of this operator is to make an aerial vehicle to wait an amount of time, so it only has effect on the *battery* and *at* dynamic properties of the vehicle.

```
(:operator (!sync ?uav1 ?uav2 ?start ?duration)
  ;precondition
  ((assign ?end (call + ?start ?duration))
   (read-time at ?uav1 ?t0) (battery ?uav1 ?r1)
   (battery ?uav2 ?r2) (call >= (call - ?r1 ?duration) 0))

  ;delete list
  ((read-time at ?uav1 ?t0) (battery ?uav1 ?r1))

  ;add list
  ((read-time at ?uav1 ?end) (battery ?uav1 (call - ?r1 ?duration)))
)
```

Code 4.27 *Finish* operator defined for the domain. This operator does not appear in the durative-actions defined for the PDDL domain. It has been defined as a *base case* for the recursive method defined in Section 4.4.3 to signal the end of the planning process and therefore to stop the recursion. It checks in the preconditions that all the parts have been assembled and has no effects.

```
(:operator (!assembly_plan_finished)
  ;precondition
  ((remaining_tasks 0))

  ;delete list
  ()

  ;add list
  ()
)
```

4.5 Connecting the Geometric and Symbolic Planners

The solution for the problem presented in Section 4.2 involves two parts: an assignment of assembly tasks to aerial vehicles and an assembly tasks decomposition and scheduling over time for each aerial vehicle. In order to compare the suitability of different solutions computed along the whole planning process, a score-based calculation is done after a new solution is found.

The solution score is based on three types of constraints with different levels of relevance. Given a new solution, its score consists on the number of broken constraints for each of the constraint types defined, thus they are represented as negative values. The constraint types were explained in Section 4.3.4.

The domain presented in Section 4.3 has been designed to solve the assignment part of the problem. In that domain, only the aerial vehicles and assembly tasks along with their dependencies are considered as entities, but the temporal aspects of the problem are not present. The values of the hard and medium constraints are computed within this domain. The hard-constraint value indicates if the weight of the assigned parts does not exceed the sum of the payloads of the assigned vehicles. Once an assembly task is allocated, the medium-constraint value indicates how many of its dependencies (parts that should be already assembled) are also allocated to the same vehicle.

The symbolic domain presented in Section 4.4 is designed to compute the assembly tasks decomposition and scheduling of the problem. In this case, the temporal domain is considered and the soft-constraint value is computed as the total assembly time for the whole structure within this symbolic domain.

The whole score calculation needs both planners to be connected and to communicate in a bidirectional way. The pseudo-code for the whole planning process can be seen in Algorithm 2. First, the VRP planner must solve the assignment problem and compute the related hard and medium constraints values. After that, and only if the hard-constraints for the given assignment are zero, it sends the computed assignment to the symbolic planner, which solves the decomposition and scheduling problem and computes the soft-constraint value. Then this value is sent back to the VRP planner, closing the score calculation loop. With the total score of the whole solution, the VRP planner can compare different solutions and optimize the search to try to find new assignments that lead to better scores and improved solutions. Hence, the optimization is done cooperatively between both planners, preserving each of them its own domain and solving a different part of the whole problem.

Figure 4.5 represents a summary of the interconnection between the geometric and symbolic planners, showing a simplified version of Algorithm 2 in a visual way.

Algorithm 2: Pseudo-code showing the connection between the involved planners. The inputs for the system are a pre-computed assembly plan composed of assembly tasks, the list of available robots, the list of locations and the time limit specified for the computations. Initially, the values for the current hard (H), medium (M) and soft (S) constraints are set to the minimum possible negative value. The best values computed during the planning process for these variables are also kept and set to the same minimum possible value. On each iteration, the VRP solver calls the *computeRoutes* function, which computes an assignment of assembly tasks to robots and defines the routes for each one. After that, it calls the *computeScore* function to compute the related hard and medium-constraints values. If the hard-constraints are zero, it calls the symbolic planner through the *HTN_Planner* function, which in turn computes the decomposition of the assembly tasks (the low-level plan for each aerial robot) and the related soft-constraints value. If the decomposition was possible, then the VRP solver compares the new values for the hard, medium and soft-constraints with the best values that have been found by calling the *comparePlans* function, and updates the best values if the new ones are better, also storing the decomposition computed by the symbolic planner, which is then the best plan found. The process is repeated until the time is exhausted.

```
Data: Assembly tasks list $A$, robots list $R$, locations list $L$, time limit $T$
Result: plan $P$ for all the robots
begin
    $P \leftarrow \emptyset, P_{\text{new}} \leftarrow \emptyset$ ;
    $H \leftarrow -\infty, M \leftarrow -\infty, S \leftarrow -\infty$;
    $H_{\max} \leftarrow -\infty, M_{\max} \leftarrow -\infty, S_{\max} \leftarrow -\infty$;
    Loop
        routes ← VRP_Planner.computeRoutes($A$, $R$, $L$) ;
        ($H$,$M$) ← VRP_Planner.computeScore(routes) ;
        if $H \neq 0$ then
            continue ;
        ($P_{\text{new}}$,$S$) ← HTN_Planner(routes) ;
        if $P_{new} \neq \emptyset$ then
            if VRP_Planner.comparePlans($P$,$P_{new}$) then
                $H_{\max} \leftarrow H, M_{\max} \leftarrow M, S_{\max} \leftarrow S$;
                $P \leftarrow P_{\text{new}}$;
        if timeReached($T$) then
            exit loop ;
    return $P$;
```

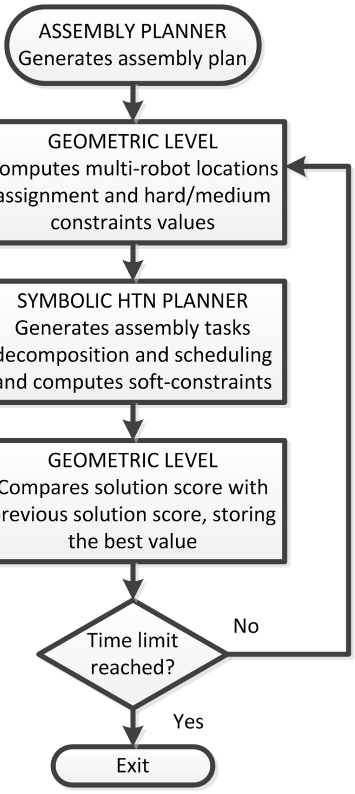

ASSEMBLY PLANNER
Generates assembly plan
GEOMETRIC LEVEL
Computes multi-robot locations assignment and hard/medium constraints values
SYMBOLIC HTN PLANNER
Generates assembly tasks decomposition and scheduling and computes soft-constraints
GEOMETRIC LEVEL
Compares solution score with previous solution score, storing the best value
Time limit reached?
No
Yes
Exit

Figure 4.5 Work-flow of the whole planning process where the role of the different planners is highlighted.

4.6 Use Case: Testing the Architecture

In this section a representative mission will be used to illustrate the operation of our decoupled planning approach in a multi-vehicle context. In the mission presented, a fleet composed of four unmanned aerial vehicles equipped with robotic manipulators is available, where all of the vehicles are simulated. The environment contains twenty-six locations of interest (see Figure 4.6) where eleven of them are the initial locations for the parts, another eleven are the assembly locations and four are home locations for the UAVs. The structure to be assembled can be viewed in Figure 4.7.

The entry point to the system is the 3D CAD environment model that contains the initial state (stock parts and home locations of the UAVs) and the 3D CAD model of the structure to be built. An external assembly planner [90] reads the 3D CAD model of the structure and generates a valid assembly plan. Each of the assembly tasks contains two nodes: one with the *effects* of the operation and one with the *preconditions* for the operation. The effect of the operation is the part that will be assembled after the task execution, whereas the preconditions for the operation are the parts that must be assembled prior to the execution of the task. Part of the assembly plan generated by the assembly planner is shown in Code 4.28.

In order to exploit the capabilities of a team with multiple vehicles, at the geometric planner level, tasks can be divided to be shared among different UAVs. That is the case of the assembly tasks associated to *Box001*, *Box002* and *Box003*, whose weight of 900 grams is higher than the payload of a single vehicle, which is of 500 grams. The task for each of

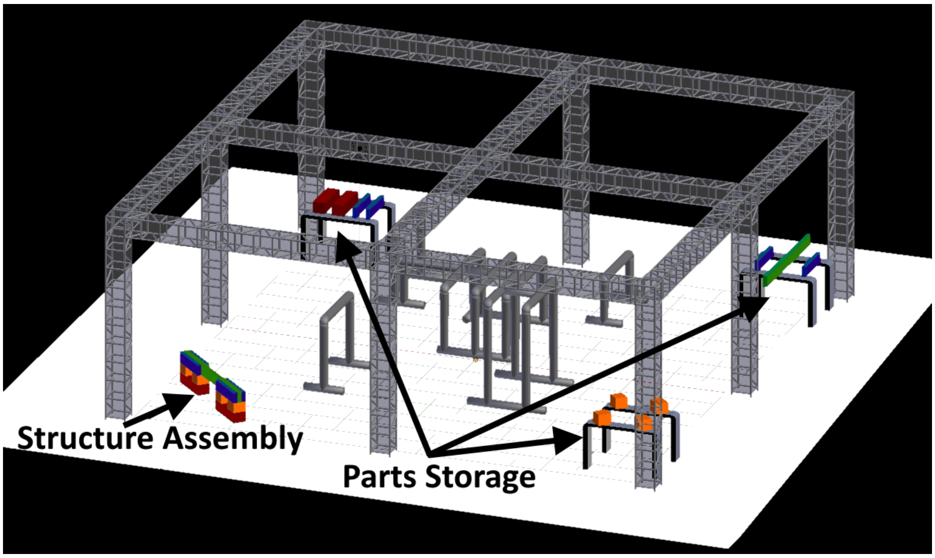


Figure 4.6 CAD model of the indoor testbed used for the experiments of the ARCAS Project. In this arena, the parts are initially stored over tables in different stock areas and are finally assembled on a designated location.

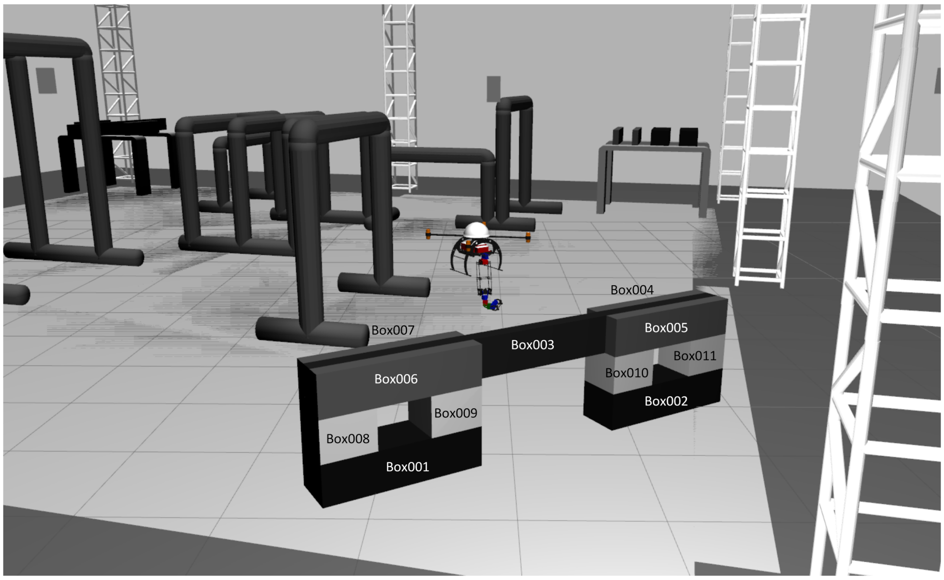


Figure 4.7 Assembly structure for the mission. The structure is composed of eleven parts, enumerated from *Box001* to *Box011*. All the parts are supposed to have a flat handle from which the robotic arms of the UAVs can pick them.

these parts is automatically divided into two different tasks of 450 grams. In this manner, a single part can be assigned to multiple UAVs.

The geometric planner solves the assignment problem. The solver was configured to apply two phases: a Construction Heuristic and a Metaheuristic. The Construction Heuristic chosen was the so called *First Fit Decreasing*, which assigns the more difficult planning entities first (those tasks with a higher part weight in our case), so it sorts the planning entities on decreasing difficulty. The Metaheuristic chosen was the *Late Acceptance*, a variant of the Hill Climbing local search. Late Acceptance does, for the assignment initially computed by the Construction Heuristic, some moves between the planning entities, one per iteration, and accepts any move that leads to a score that is better than the best score of a number of moves ago. This allows to do one move that initially leads to a worse score than the previous to improve the score computed some moves ago.

After an assignment of parts to UAVs is done, the geometric planner generates a planning problem in a format suitable for the symbolic planner. The JSHOP2 planning problem contains the high-level task that represent the whole structure assembly and that should be decomposed into primitives, as well as the initial states of all the entities involved. Code 4.29 shows the generated JSHOP2 planning problem that led to the best solution found, containing among others, the assignment from parts to UAVs generated by the geometric planner and the dependencies computed for each of the parts by the assembly planner. It must be remembered that on each iteration of the geometric planner, a new JSHOP2 planning problem is generated, so we only show the one that led to the best solution.

Code 4.28 First tasks of the assembly plan generated by the external assembly planner [90]. The tasks are partially ordered meaning that a single vehicle could do the assembly correctly by executing the tasks in that order. Tasks that appear later in the file may be executed before some of the previous as is the case of part *Box001* which constitutes a part of the base and thus do not depend on any other.

```xml
<plans>
  <assemblyPlan>
    <assemblyOperation action="base">
      <effect>
        <at part="Box002"/>
      </effect>
    </assemblyOperation>
    <assemblyOperation action="connection">
      <precondition>
        <at part="Box002"/>
      </precondition>
      <effect>
        <at part="Box011"/>
      </effect>
    </assemblyOperation>
    <assemblyOperation action="connection">
      <precondition>
        <at part="Box002"/>
      </precondition>
      <effect>
        <at part="Box010"/>
      </effect>
    </assemblyOperation>
    <assemblyOperation action="base">
      <effect>
        <at part="Box001"/>
      </effect>
    </assemblyOperation>
    ...
  </assemblyPlan>
</plans>
```

The plan computed by the symbolic planner contains all the primitives for each of the vehicles involved in the mission execution and it has been represented with a Gantt chart in Figure 4.8. Again, we only show the Gantt chart for the best solution found. The symbolic planner produced a schedule of all the assembly tasks computed by the assembly planner. In the cases where multiple choices could be done, the planner decided to assemble first those parts that were dependencies for other parts that could not be assembled yet, trying to maximize the potential parallelism of using multiple vehicles. That produced, for the assignment computed by the geometric planner, a correct scheduling of the assembly tasks.

Code 4.29 JSHOP2 planning problem. It contains the assignment from parts to UAVs generated by the geometric planner and the dependencies computed for each of the parts by the assembly planner. The state of the different vehicles and parts is also included.

```
; FACTS
(
  ; UAV defs
  (quadrotor uav1)(quadrotor uav2)
  (quadrotor uav3)(quadrotor uav4)
  ; location defs
  (location 1)
  ...
  (location 26)
  ; object defs
  (object Box001)
  ...
  (object Box011)
  ; object state defs
  (at Box001 1)
  ...
  (at Box011 21)
  ; ObjectDependencies
  (depends Box001 ())
  (depends Box002 ())
  (depends Box003 (Box008 Box009 Box010 Box011))
  (depends Box004 (Box011 Box010))
  ...
  (depends Box010 (Box002))
  (depends Box011 (Box002))
  ; assembly locations
  (assembly_location Box001 2)
  ...
  (assembly_location Box011 22)
  ; part assignments
  (assigned Box003 uav1)
  (assigned Box004 uav1)
  (assigned Box001 uav1)
  (assigned Box008 uav1)
  (assigned Box002 uav2)
  ...
  (assigned Box002 uav3)
  (assigned Box010 uav4)
  (assigned Box005 uav4)
  (assigned Box001 uav4)
  ; UAS state defs
  (battery uav1 1200)
  (at uav1 23)(landed uav1)
  ...
  (battery uav4 1200)
  (at uav4 26)(landed uav4)
  ...
  ; remaining tasks
  (remaining_tasks 11)
)
; GOALS
((mission assemble))
```

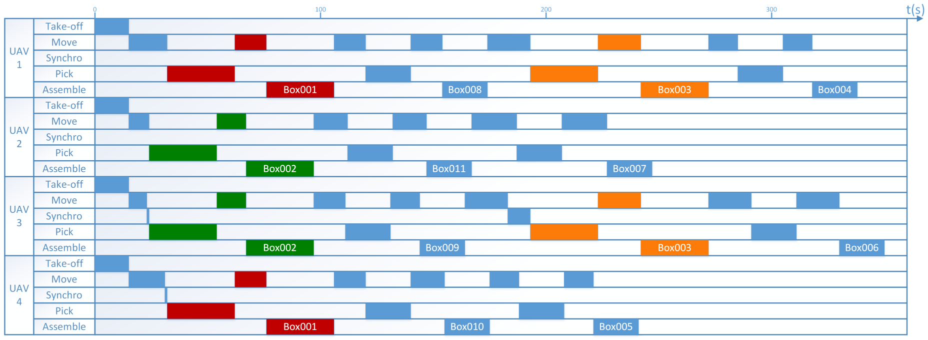


Figure 4.8 Gantt chart of the best solution found by the system. Each rectangle represents a primitive task. Primitives with a string on top represent the assembly task of the part with the given name. Primitives of blue color are independent and are executed by the vehicles individually. Primitives of red, green and orange color are cooperatives and thus are executed by the UAVs on which they appear simultaneously. Cooperative primitives appear on parts that must be managed by two vehicles simultaneously due to their weight. Red primitives are executed cooperatively and simultaneously by *UAV1* and *UAV4* in order to assemble the part *Box001*, green primitives by *UAV2* and *UAV3* to assemble *Box002* and orange primitives by *UAV1* and *UAV3* to assemble *Box003*.

4.7 Simulation Results

Different simulations have been carried out in the environment shown in Figure 4.6, which is the 3D model of the indoor testbed used for the experiments. Within the testbed, localization of the robots is given by the Vicon motion tracking system with millimetre precision. In the study done by Merriaux et al. [84], it has been proven that the Vicon system can achieve errors below two millimetres at common speeds, and below one millimetre for static objects. For the simulations, the ROS global coordinate system is used. The tests have been done on a machine with an Intel i7 CPU at 2 GHz and 8GB RAM. The goal of the simulations is to compare different solvers.

A team of four aerial robots equipped with manipulators has to assemble a given structure. Figure 4.9 shows one of the prototypes developed in the context of the project that is modelled and used in the simulations of this section.

Three structures with a different number of parts (see Figure 4.10) have been considered and, for each of the structures, ten data sets have been generated changing randomly the initial locations of the parts.

The solver has been configured to use one Construction Heuristic phase followed by one Metaheuristic phase. The purpose of the former is to obtain an initial solution for the assignment problem, which will be later optimized by the second solver phase. Three CH algorithms have been applied to our problem: First Fit, First Fit Decreasing and Cheapest Insertion. A detailed description of each one can be found in [100]. The results shown in Table 4.4 have been computed with a time limit of ten minutes if the search does not

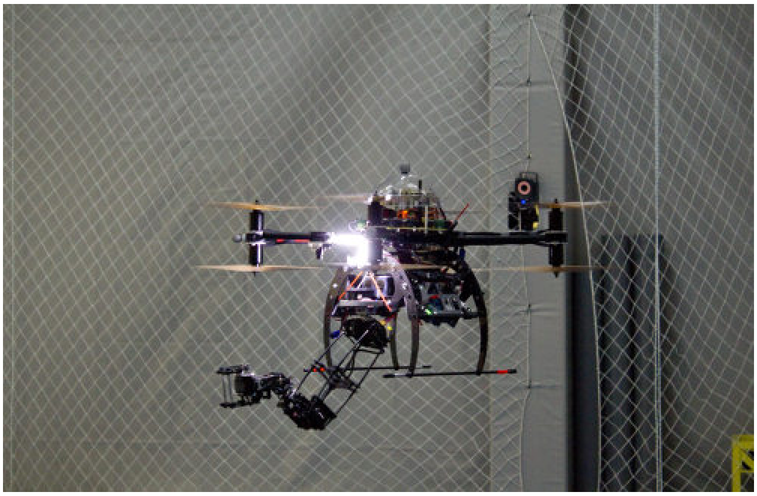


Figure 4.9 Aerial robot prototype equipped with a robotic arm in the indoor testbed located in the FADA-CATEC facilities in Seville (Spain). The model of this prototype has been used in the simulations of the missions.

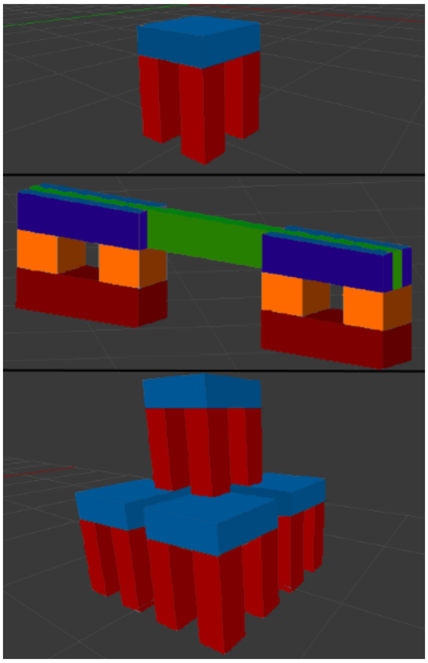


Figure 4.10 Structures used for the benchmark with sizes of five, eleven and twenty-five parts (structures 1, 2 and 3 from top to bottom for later reference).

finish before. It can be seen that the First Fit algorithm obtained slightly better values, even reducing to zero the medium constraints. In addition, its computation times are lower than the others.

Table 4.4 Construction Heuristic solver phase results for 30 simulations. For each algorithm, the mean and standard deviations for the hard, medium and soft constraint values are presented, as well as the mean computation time. The broken constraints are represented as negative values.

CH Algorithm	Hard (SD)	Medium (SD)	Soft (SD)	t(s)
First Fit	-0.33 (0.48)	0 (0)	-625.80 (418.34)	3.12
First Fit Decreasing	-0.33 (0.48)	-0.33 (0.48)	-630.15 (422.36)	9.56
Cheapest Insertion	-0.33 (0.48)	-0.33 (0.48)	-630.15 (422.36)	21.33

The second phase is the MH phase, which tries to optimize the initial locations assignment computed by the previous CH phase. Five local search algorithms, whose description can also be found on [100], have been compared: Hill Climbing, Tabu Search, Simulated Annealing, Late Acceptance and Step Counting Hill Climbing. As this phase requires the use of a previous CH phase, the First Fit algorithm was configured as CH. The results are shown in Table 4.5. All the MH algorithms reduced to zero the values of the hard and medium constraints, so only the mean and standard deviations of the soft constraint values are shown. The results show that the Late Acceptance and Step Counting Hill Climbing algorithms tie, obtaining better values than the others.

To study the effects over the solutions of changing the number of aerial vehicles used, we have decided to focus our attention on the Late Acceptance algorithm, although the Step Counting Hill Climbing would have been also a good choice. We have used only the third structure since it is the most complex one with the higher number of parts (25). Five parts need to be transported between two aerial robots due to its high weight, so these five parts are divided into two assembly tasks, resulting in 30 different assembly tasks. Then, the difficulty in solving the problem is greater than using the other structures. To test the scalability of the system when increasing the number of available aerial vehicles, five datasets for the given structure have been created with a number of available aerial vehicles of 10, 20, 30, 40 and 50 respectively. The results of the tests are presented in Figure 4.11. Figure 4.11a shows the score (assembly time) obtained for each of the datasets, whereas Figure 4.11b shows the number of aerial robots used in the solutions.

As it is shown in Figure 4.11a, increasing the number of available aerial robots leads to better plans, as the assembly time tends to decrease. However, from the 30 available aerial robots dataset onwards the differences are not so high and the assembly time starts to decrease more slowly than for the previous datasets. In fact, the 50 dataset gets worse assembly times than the 30 and 40 aerial robots datasets.

The resulting number of aerial robots used for each dataset is displayed in Figure 4.11b.

Table 4.5 Meta-heuristics solver phase results of the soft constraints generated after 30 simulations with three different structures. The solver was configured with a time limit of ten minutes if the search did not finish before. However, all the algorithms reached the time limit without exhausting the search.

CH Algorithm	Struct.1 (SD)	Struct.2 (SD)	Struct.3 (SD)	Total (SD)
Hill Climbing	-205.80 (14.19)	-415.80 (37.91)	-1020.0 (54.33)	-547.27 (353.15)
Tabu Search	-205.80 (14.19)	-403.10 (29.64)	-1020.9 (55.95)	-543.27 (354.99)
Simulated Annealing	-203.70 (15.29)	-413.40 (31.34)	-1031.9 (49.81)	-549.67 (359.18)
Late Acceptance	-203.80 (15.33)	-410.80 (31.27)	-991.30 (73.75)	-535.30 (342.06)
Step Counting H.C.	-203.80 (15.33)	-410.80 (31.27)	-991.30 (73.75)	-535.30 (342.06)

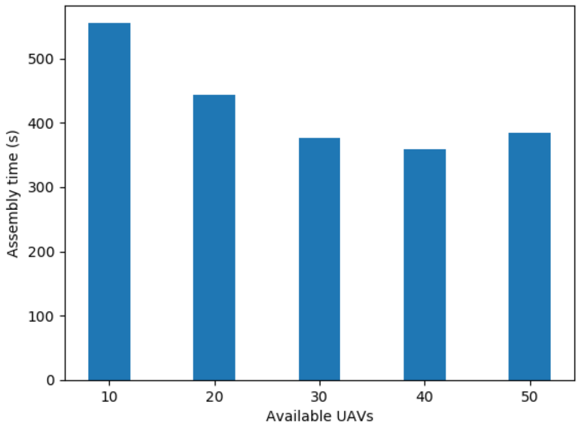
As the number of available aerial robots is increased, the number of aerial robots used in the solutions tends also to increase. For the datasets that have a number of available aerial robots lower or equal than the number of assembly tasks (30), the solver uses a number of vehicles that is near the maximum number of vehicles available, as it can be seen in the 10, 20 and 30 aerial robots datasets. For a higher number of available aerial robots, the number of used aerial robots stabilizes near 30, which is the number of assembly tasks for the structure. This fact tells us that the solver will always try to use the maximum number of available aerial robots, even using one aerial robot per assembly task if there are enough aerial robots available. This may seem logical because it is an (extreme) way of maximizing parallelism: in fact, many people will think on this as the optimal solution. However, two associated issues should be also taken into account:

- Having many aerial robots working in our testbed with a size of tens of meters is unrealistic due to the associated air traffic density. As the combined planner will always try to use the maximum number of available resources, the usage of these resources should be limited for instance introducing hard-constraints that saturate the maximum number of used vehicles.
- Increasing the number of available aerial robots also increases indirectly the problem size. The VRP planner has a greater number of options to choose when assigning assembly tasks, which can lead to obtaining better plans but can also have the opposite effect since the search tree size is increased and many more options would be available to be check. We can see this in the results for the 50 available aerial robots dataset, whose assembly time is slightly worse than the times for the 30 and 40 available aerial robots datasets. Thus, if the number of available aerial robots is increased, then the solver's computing time should also be increased.

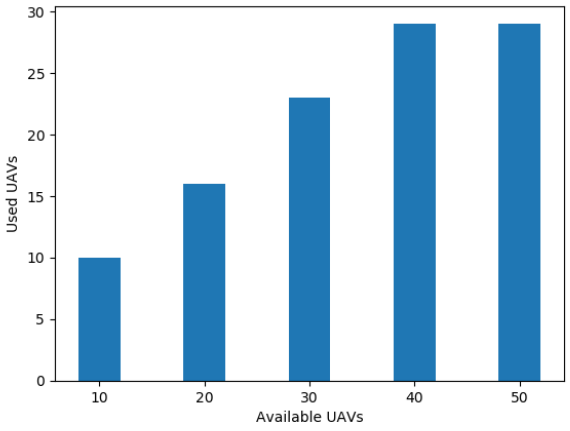
Although it is out of the scope of this paper, our planning framework includes the possibility to simulate the execution of the low-level plans computed for each vehicle. An execution layer has been implemented as a C++ graphical user interface application to read the low-level plans. The interface, implemented using the Qt framework, checks for the correct execution and synchronization of the tasks and generates Gantt charts to display the different timelines of the aerial vehicles. The application communicates with a middleware developed by using the ROS (Robot Operating System) framework that connects with the Gazebo simulator. Figure 4.12 shows a screenshot of a mission execution on the Gazebo simulator by one aerial robot. A video of the execution can be downloaded from *https://grvc.us.es/symballoc#simulationPaper*.

It should be mentioned that the motion planning, multi-robot collision avoidance and the control levels have been also implemented in ROS. In particular, the approach followed at the control level is described in [106], whereas for multi-robot collision avoidance the techniques implemented are presented in [1]. Regarding motion planning, a comparative study was presented in [104] that lead to the use of the RRT-Connect algorithm in our simulations.

The whole ROS stack developed for the integrated planning framework has been used in the real aerial robots equipped with manipulators. However, the implementation details of the other planners and their interconnection are out of the scope of this paper. As a reference, there are some videos available also in *https://grvc.us.es/symballoc* that show these additional planning capabilities and the execution of plans with several aerial robots both in simulation and in the testbed located in FADA-CATEC.



(a) Assembly times for tests in a range of available aerial robots between 10 and 50.



(b) Number of aerial robots used for tests in a range of available aerial robots between 10 and 50.

Figure 4.11 Results of the scalability tests done in a range of available aerial robots between 10 and 50. The third structure from Figure 4.10 has been used in the tests since it is the most complex one with a higher number of parts. First Fit and Late Acceptance algorithms have been configured in the solver. Five datasets have been created with a number of available aerial vehicles of 10, 20, 30, 40 and 50 respectively. Increasing the number of available aerial robots leads to better plans since the assembly times tend to decrease. As the number of available aerial robots grows, the number of aerial robots used in the solutions tends to increase.

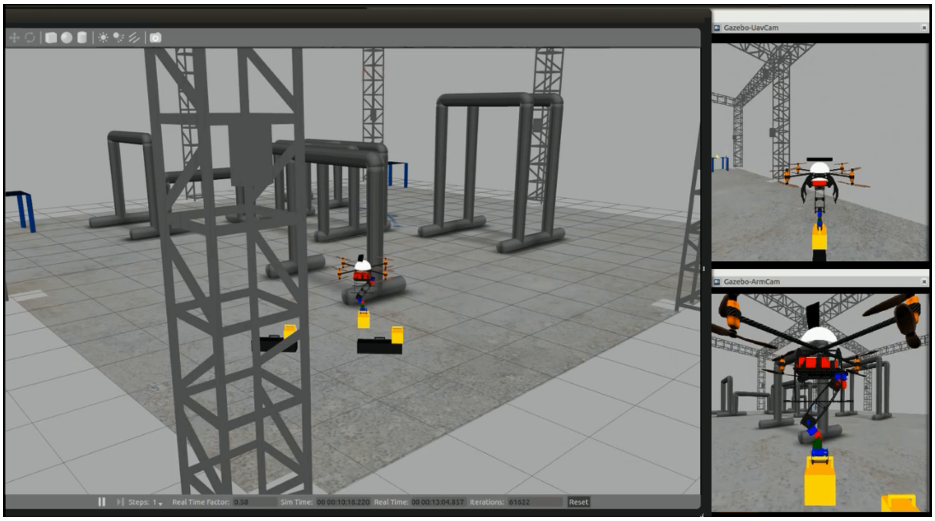


Figure 4.12 Screenshot of the simulation of an assembly action executed by one aerial vehicle during one of the missions. All the parts have a handle to hold and move them and can be stacked.

4.8 Conclusions

In this chapter we perform task assignment and scheduling to improve cooperation and maximize parallelism in a domain that mix symbolic reasoning with the VRP. The main contribution is a new way of connecting two independent planning systems based on a score calculation system that lets them cooperate in the optimization of the solutions found and its application in the context of structure assembly missions.

The approach has been tested successfully in missions involving multiple simulated aerial vehicles. The bi-directional communication between the planners has allowed the optimization of the solutions found by the VRP planner, and thus, the feedback of the symbolic layer has been a key aspect to drive the search towards better solutions.

Different meta-heuristic algorithms have been applied. Although these algorithms have not been able to guarantee optimal solutions, they have computed feasible solutions in short times proving their effectiveness.

One of the problems found in our new approach arises at the VRP level, as the OptaPlanner engine may not find a solution that satisfies all the imposed hard-constraints. In addition to this, the planning time is set beforehand by the user, so when the timer expires OptaPlanner may not have found yet a feasible solution. These problems can be solved by increasing the planning time and relaxing the hard-constraints for the problem domain. Nevertheless, in all the simulations we have been able to find and generate a feasible solution without increasing the planning time or relaxing the initial constraints.

There are several advantages in using an aerial robot with manipulation capabilities and practical applications of the research presented in this chapter can be found in different industry fields. The AEROARMS [1] European project aims to develop the first aerial robotic platform equipped with multiple arms and advanced manipulation capabilities, with the intention to be used in inspection and maintenance tasks in industrial plants. This project is based on the results obtained from the ARCAS [2] European project that inspired the work presented in this chapter, and one of its main objectives is the development of systems which are able to grab and dock with one or more arms and perform dexterous accurate manipulation with another arm. Another practical application can be found in the ARM-EXTEND [3] Spanish project, which is also based on the results of the ARCAS project. ARM-EXTEND proposes the development of the first robotic manipulation system with aerial and ground locomotion capabilities in an industrial environment and the first specifically designed for inspection and maintenance purposes in locations with very difficult access. The system aims to be used particularly in the maintenance of solar power plants.

In future work, the goal is to enhance the planning domain based on the realistic conditions with the prototypes developed in our project. Modifying the architecture to ensure the completeness of the system is one of our current goals. To achieve this, a new sound and complete symbolic HTN planner with geometric reasoning capabilities has

[1] https://aeroarms-project.eu
[2] http://www.arcas-project.eu
[3] https://grvc.us.es/national-projects/

been developed, with the intention of replacing the OptaPlanner planning engine. It will be presented in the next chapter.

5 Coupled Geometric and Symbolic Reasoning

In the previous chapter, the combination of a symbolic Hierarchical Task Network (HTN) planner and a VRP planner was used to solve the problem domain modelled for the ARCAS project. The composition of the geometric and symbolic states was solved by making the VRP planner call the HTN planner when needed: after the VRP planner made an initial assignment of parts to aerial vehicles based on the locations of the aerial vehicles and the locations of the parts (as well as their dependencies), it called the HTN planner to obtain the scheduling of the assembly operations and to get the estimated duration of the assembly. With this estimation, it was possible for the VRP planner to optimize the computed assignment and try to get another one that could improve the overall score and thus, the plan quality.

Our new score-based interconnection between the VRP and HTN planners made it possible to solve the modelled ARCAS domain but left some questions opened. Is it possible to solve this kind of problems, where a certain degree of geometric reasoning is needed, by using only the given HTN planner? If that is the case, how good the solutions will be? Is it possible to optimize the solutions or to obtain the optimal solutions? Is there a way to ensure the completeness of the system?

In order to be able to answer these question and based on the idea of a guided heuristic search, a new HTN planner has been developed and tested in the context of the AEROARMS project. The algorithm for our new HTN planner is based on the A* algorithm [49, 50] and offers some additional features that are not present in the score-based interconnection presented in Chapter 4. These features are novelties that are not present in the original JSHOP2 implementation and are the result of replacing the depth-first search algorithm implemented in JSHOP2, which is commonly used in many others HTN planners as the search strategy, by a guided heuristic search. We call this new planner SHOP*.

The chapter is structured as follows. The motivations and expectations for our new HTN algorithm are described in Section 5.1. In Section 5.2 the description of our new heuristic-guided SHOP* HTN planner and its properties is given along with a short overview of the A* algorithm and its properties, which inspired the development of our new planner. An use-case in the context of the AEROARMS project to test the optimality of our new planner is presented in Section 5.3, where the Travelling Salesman Problem [47] is involved. In Section 5.4 a benchmark is applied and the previous problem is solved by using our new planner. Its results are compared to the results obtained by using the original implementation of JSHOP2 and our original planner interconnection used in the previous chapter. Finally, the chapter ends by presenting the conclusions obtained during the research and after the tests.

5.1 Replacing the Depth-first search: Motivations and Expected Behaviour

In Section 3.3.4 we presented and explained the JSHOP2 algorithm. This algorithm was implemented by its authors using a depth-first search [73]. As soon as the subtasks of a given task are identified, they are inserted in a queue by following a First-In First-Out way (FIFO). By this way, when a task can be decomposed, the first subtask is explored first. With that subtask the process is repeated, exploring again first the first subtask on which it can be decomposed, and so on.

Along the research done in this thesis we have found that the use of a depth-first search seems to be a common scheme in the implementation of HTN planners. The SHOP planner [93] was implemented using this search algorithm, and it results interesting that its evolution, the JSHOP2 planner used in this thesis, still uses that kind of search. The HATP hierarchical planner [110], developed by researchers of the LAAS-CNRS, also uses a depth-first search on its implementation. In the field of video games, where the use of a well implemented Artificial Intelligence is important to model the behaviour of Non-Player Characters (NPCs), the trend keeps the same. The Simple Hierarchical Planning Engine (SHPE) [83] is a planning system based on HTN planning techniques currently implemented in games, and uses a depth-first search. Many other examples that use the same search pattern for the implementation of their HTN planning systems can be found [112, 113].

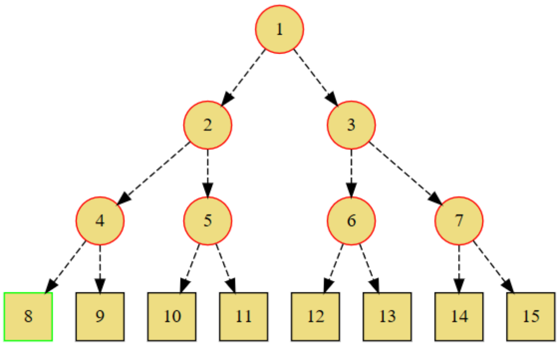
Which can be the reason that makes some researchers choose the depth-first search algorithm over other popular search schemes like breadth-first [73] when implementing their HTN planning systems? We know that in the field of graph search algorithms, the use of depth-first gives preference toward investigating longer plans very early while the use of breadth-first guarantees that the first solution found will always use the smallest number of steps. So, why not to use always a breadth-first in the implementation of HTN planning systems?

To make an idea of the possible answer for this last question let us consider the graphs shown in Figure 5.1. In this figure, two possible traversals of a decomposition tree for a high-level task are shown. The decomposition for the high-level task (the root node) and all the subtasks has been supposed to be binary (each task can be decomposed only into two subtasks). Although in HTN planning a given task can be modelled to be decomposed into 1 to N subtasks, the decomposition into a single subtask is not a very frequent case, as the child decomposition of the subtask could be directly attached to the parent task, so for a simplification in our explanation let us suppose a completely binary decomposition tree. For each task (circle node) in the tree, two possible decompositions can be chosen. The lowest-level subtasks on which the high-level task can be decomposed are represented by the leafs of the tree (square nodes). These nodes represent the final decomposition of the high-level task, so they can be seen as the goal states to reach in the search tree. Figure 5.1a shows the decomposition tree traversal if a breadth-first search is applied, while Figure 5.1b shows the traversal if a depth-first is applied. The nodes have been labelled with the order on which they are explored. Red nodes represent nodes that are explored before visiting a leaf node, while the green node represent the first visited leaf. By looking into the figures two interesting things can be seen. The first is that in the given example, the depth-first search is capable of decomposing the parent task faster than breadth-first, as it reaches a leaf node faster. The second thing is that breadth-first needs to visit a greater number of nodes before reaching a leaf node in comparison with depth-first.

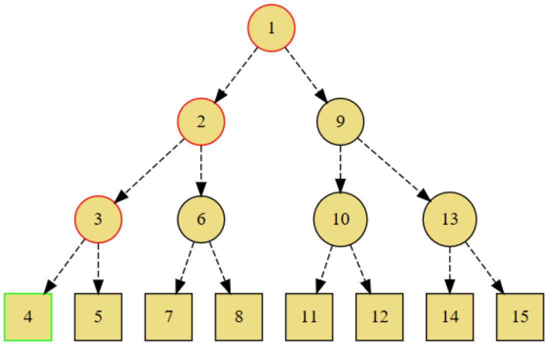
From what can be seen on Figure 5.1 two conclusions can be made. The first is that depth-first makes an aggressive exploration in the depth of the decomposition tree, and this seems to fasten reaching a goal state in the search tree, as in HTN trees the goals are located in the leaf nodes. The second conclusion is that breadth-first needs to visit more nodes before reaching a leaf node. In HTN systems this is not only translated in higher computation times, but also in a greater consumption of the available memory. A HTN system requires that each visited node keeps stored in memory until the algorithm finishes, as if some decomposition fails then a backtrack mechanism is triggered to go back to the parent node of the failing node to try the other possible decomposition. That makes breadth-first consume a considerable high amount of memory compared with depth-first, and as it will be seen later on this chapter, this can be an issue when solving problems of medium or high sizes.

The decomposition tree on Figure 5.1 has been used as an example to see the main differences between two classical graph search algorithms, the depth and breadth-first algorithms, that could be used in the implementation of HTN planners. Of course, we could find specific examples where breadth-first finds a solution sooner than depth-first. But in the field of HTN, guiding the search towards finding a plan using depth is natural, so in general the depth-first algorithm is a better choice than breadth-first.

After knowing one of the reasons why a depth-first search was used in the JSHOP2 HTN planner, we thought different ways of improving its performance. One of the drawbacks of the JSHOP2 planner is that it stops after finding the first solution. The domain description is used to drive the symbolic search toward a good enough solution by providing methods in their order of quality and using ordering functions when binding variables. By this way, we usually consider that finding the first feasible plan in a reasonable time is enough. However, that makes its use impractical in problems that, for example, depend on any kind of assignment, as the cost of the solutions highly depends on the initial assignment. That was one of the reasons that lead us to use an external VRP planner to solve the ARCAS domain presented in Chapter 4. The plan quality for the ARCAS domain was highly dependent on the initial assignment of parts to aerial vehicles. When trying to solve the ARCAS domain by using only the JSHOP2 planner, we realized that after making an initial assignment, JSHOP2 was unable to modify it if a solution was found. In addition to this, we encountered problems when facing the geometrical part of the problem. Although the geometric level of the ARCAS domain was modelled at a high level of abstraction, as we



(a) Breadth-first search.



(b) Depth-first search.

Figure 5.1 Example of breadth-first and depth-first search applied to a HTN decomposition. The tree root represents the task to decompose while leaf nodes represent the lowest-level subtasks on which the root task can be decomposed (goal states). High-level tasks (those that can be decomposed) are represented as circles while low-level tasks are represented as squares. The decomposition tree has been supposed to be binary, so each task only has two possible decompositions. Figure 5.1a shows the decomposition tree traversal for breadth-first, while Figure 5.1b shows the traversal if a depth-first is applied. Red nodes are visited prior reaching a leaf node, while the green node is the first visited leaf. Depth-first reaches a goal state sooner and exploring a lower number of nodes than breadth-first.

used a graph with costs and Euclidean distances, its encoding in the JSHOP2 domain was difficult and hardened the readability of the domain and problem description.

When planning in JSHOP2 we stop at the first solution. To find the optimal solution would require to explore the complete decomposition tree and try all the alternatives, which would require very long computation times, specially if using the depth-first search. Indeed, a great number of problems would not be solvable, at least in a reasonable time.

In this chapter we propose to replace the depth-first search algorithm implemented in JSHOP2 and use the domain encoding as a form of heuristic to drive the search, focusing on nodes that open promising solutions. In specific, we propose to use the ideas of the A* algorithm (which will be explained in the next section). For each problem domain, an additional heuristic function will be given. The purpose of this heuristic function is to give a cost estimation to go from a decomposition node to the goal node, which is the one where all tasks would have been decomposed. By using this heuristic function (which is problem dependent), all the possible decompositions of a task will be scored with their costs and will be ordered to explore first the more "promising" ones, those with the lowest cost and that would drive to a better plan. If the explored nodes are stored in a priority queue as they are found, ordered by their costs, recovering and expanding on each iteration the more promising ones, and if the heuristic function meets some mathematical properties (which will be also explained in the next section), then the best solution will be found. To be capable of handling more easily problem domains that have some sort of geometry, a geometric projection of the decomposition nodes based on geometric computation, such as using Euclidean distances and so on, could be used in the heuristic function.

What we first expect with this proposal is to give JSHOP* the ability to improve the solution found. The search engine will still stop after finding the first solution, but the quality of this solution should be significantly higher than before, because the search will be driven by the heuristic function towards a better solution. And as we will see in the next section, if the heuristic function is *admissible*, then we can guarantee that the solution found is the optimal solution.

In performance terms, the algorithm should be slower than before applying the changes, as we will evaluate a higher number of nodes before reaching the solution. Also, the memory usage should be higher than using the depth-first, as the discovered nodes will be kept on memory to update their score in those cases where different decomposition or variable bindings drive to the same search state.

5.2 SHOP*: the A*-based HTN Planner

The A* search algorithm [49, 50] is an extension of Dijkstra's algorithm [20] that tries to reduce the total number of states explored by using a heuristic estimate of the cost to go from a given state to the goal. It is widely used in path-finding and graph traversal problems, and enjoys widespread use due to its performance and accuracy. It is very common to see A* usages in the field of video-games, due to its easy implementation.

A* is proven to be *complete*: it finds a solution if one exists and otherwise, it correctly reports that no solution is possible. In addition, under certain conditions A* is also proven to be *optimal*: if a solution is found, then it is guaranteed that this solution is the optimal solution.

A* solves problems by searching among all possible paths to the solution for the one that incurs the smallest cost. Among these paths it first considers the ones that appear to lead more quickly to the solution. Starting from a specific node of a graph, it constructs a tree of paths starting from that node, expanding paths one at a time, until one of its paths ends at the predetermined goal node.

At each iteration, A* needs to determine which of its partial paths to expand into one or more longer paths. It does so based on an estimate of the *cost to go* to the goal node. A* selects the path that minimizes:

$$f(n) = g(n) + h(n) \tag{5.1}$$

where n is the last node on the path, *g(n)* is the cost of the path from the start node to n, and *h(n)* is a heuristic function that estimates the cost of the cheapest path from n to the goal. For the algorithm to find the shortest path (the optimal solution), the heuristic function must be *admissible* [73], meaning that it never overestimates the actual cost to go to the nearest goal node. This heuristic function is problem-specific. As an example of admissible heuristic, when searching for the shortest route on a map, *h(n)* might represent the straight-line distance from the current node n to the goal, since that is physically the smallest possible distance between any two points. In some problems it is difficult to find a heuristic that is efficient and provides a good search guidance, and in some cases finding an admissible heuristic may not be possible, so finding the optimal plans is not always possible.

Algorithm 3 shows the pseudocode for the A* algorithm. Some implementations of A* use a priority queue to perform the repeated selection of the minimum estimated cost nodes to expand, while others use a simple queue. The queue is known as the *open set*. At each iteration, the node with the lowest *f(n)* value is removed from the queue, the f and g values of all its neighbours are updated, and these neighbours are added to the open set. The algorithm continues until a goal node is reached and has a lower f value than any node in the open set, or until the set is empty. If h is an admissible heuristic, then the f value of the goal is guaranteed to be the length of the shortest path (the best solution). To find the actual sequence of steps, the algorithm can be implemented so that each node on the path keeps track of its predecessor. By this way, it is possible to reconstruct the path that leads to the goal node.

It is necessary to remark four important functions that appear in Algorithm 3 and that are problem specific:

- The *Is_Goal* function receives the current state and the goal state, and checks if the current state is indeed a goal state, on which case the algorithm finishes and the path to the solution is reconstructed and returned.
- The *Get_Neighbor_List* function receives one state and returns the list of all possible states that can be reached from the input state.
- The *Dist* function computes the distance (cost) of going from the current node to one of its neighbours. This function is needed to compute the *g(n)* value.
- The *Heuristic_Cost* function computes the estimated distance of going from a node to the closest goal node, and represents the value of *h(n)*.

For each problem domain, these four functions, as well as the data structure for the nodes, need to be implemented.

The PYHOP planner (see *https://bitbucket.org/dananau/pyhop*) is a Python implementation developed by the same author of the JSHOP2 HTN planning system, and was developed

Algorithm 3: Pseudocode for the A* algorithm. At each iteration, the node with the lowest *f(n)* value is removed from the queue, the f and g values of all its neighbours are updated, and these neighbours are added to the open set. To compute the *gScore* of neighbours, a *Dist* function that returns the cost of getting from the current node to the neighbour is needed. The algorithm continues until a goal node is reached and has a lower f value than any node in the open set, or until the set is empty. If the *Heuristic_Cost* function is an admissible heuristic, then the f value of the goal is the length of the shortest path. If each node keeps track of the previous node, when the algorithm finds a solution then the path to the goal node can be reconstructed and returned by calling the function *Reconstruct_Path*.

```
Data: Start node $start$, end node $goal$
Result: Plan path $P$, or $FAILURE$
begin
    $openSet = new\ Empty\_Set(NODE)$;
    $cameFrom = new\ Empty\_Map(NODE- > NODE)$;
    $gScore = new\ Empty\_Map(NODE- > DOUBLE)$;
    $fScore = new\ Empty\_Map(NODE- > DOUBLE)$;
    $neighborList = new\ Empty\_List(NODE)$;
    $gScore[start] = 0$;
    $fScore[start] = gScore[start] + Heuristic\_Cost(start, goal)$;
    $openSet.put(start)$;
    while $NOT\ openSet.empty()$ do
        $current =$ the node in openSet having the lowest fScore value;
        if $Is\_Goal(current, goal)$ then
            return $Reconstruct\_Path(cameFrom, current)$;
        $openSet.remove(current)$;
        $neighborList = Get\_Neighbor\_List(current)$;
        for $each\ neighbor\ of\ neighborList$ do
            $tentative\_gScore = gScore[current] + Dist(current, neighbor)$;
            if $neighbor\ not\ in\ openSet$ then
                $openSet.add(neighbor)$;
            else
                if $tentative\_gScore >= gScore[neighbor]$ then
                    $continue$;
            $cameFrom[neighbor] = current$;
            $gScore[neighbor] = tentative\_gScore$;
            $fScore[neighbor] =$
              $gScore[neighbor] + Heuristic\_Cost(neighbor, goal)$;
    return $FAILURE$;
```

to be easy to understand and implement, as it has less than 150 lines of code. This code simplicity made as to take the decision of replacing its depth-first search algorithm with our new A*-based heuristic-guided search algorithm and to see how the solutions would be improved.

In PYHOP, the states of the world are represented using ordinary variable bindings and not logical propositions as in JSHOP2. A state is just a Python object that contains the variable bindings, and one can add to a state as many variables of any type as desired. Also, the HTN operators and methods for PYHOP are not defined using a specialized planning language. Instead, they are written as ordinary Python functions that receive the current state of the world passed as an argument and modify it as needed. Inside these functions a programmer can code anything that he wants, but the nature of JSHOP2 must be kept: the purpose of operators and methods still is to decompose tasks into subtasks and to update the current state of the world. PYHOP, like JSHOP2, constructs plans in the same order that they will be later executed, so at each planning step the current state of the world is known.

We chose PYHOP for its minimalist implementation, easier to understand and modify than any other HTN planning system. Also, we did not had to handle with any specialized planning-language software module, as in PYHOP the domain description (operators and methods) and the problem description (the state of the world and the tasks to decompose) are encoded in the same code file with Python functions, objects and variable declarations. As in PYHOP all is done using the Python language, that gave us the ability to do almost anything we wanted when modifying the algorithm. During our research we also noticed that both PYHOP and JSHOP2 had similar computation times when solving problems, with the advantage of PYHOP being simpler and more easily customizable

Code 5.1 shows the main Python code for the PYHOP algorithm implementation. We have omitted the originally included debug code and other pieces of code related to how methods and operators are declared for clarification purposes. The main algorithm of PYHOP is defined as a recursive function that receives the current state of the world, a list with the tasks that must be decomposed, and a list that represents the final plan, which is initially empty. On each recursive call, the algorithm initially checks if the task list is empty, on which case the algorithm finishes returning the plan list. If that is not the case, then the first task of the list is retrieved and two possibilities may arise.

The first possibility is that the task can be executed directly by an operator, on which case the new state resulting of applying the operator is created, the task is removed from the task list and added to the plan list, and a recursive call is done with the updated variables.

The second possibility is that the task can be executed by some methods. In that case, the methods that can decompose the task are tried in order of appearance. The subtasks resulting of applying the methods to the task are computed and added to the first positions of the task list, and a recursive call is done. In that case, the plan list is not modified.

One of the most important parts of the code is when applying a method to a task. It must be remarked that, although there may be multiple methods that can decompose the task, they are only applied if the recursive call fired by the previously applied method failed in finding a plan. In other words, with that implementation, the only way to try the different decomposition branches of the different methods that can be applied to a specific task is that a backtrack is triggered, and that situation only occurs when the recursive call for the

Code 5.1 Python code of the main algorithm for the PYHOP HTN planner. The algorithm is defined as a recursive function that receives the current state of the world, a list with the tasks that must be decomposed, and a list that represents the final plan, initially empty. Each task of the task list is also a list where the first element (*task1[0]* in the code) is a string that represents the task name and the rest of elements are the arguments or parameters for the task (*task1[1:]* in the code). This is the same for the plan list. If the task list is empty, the algorithm finishes returning the plan list. If not, then the first task of the list is retrieved. If the task can be executed directly by an operator, the current state of the world is updated, the task is removed from the task list and added to the end of the plan list. If the task can be executed by some methods, they are tried in order of appearance, the resulting subtasks are computed and added to the first positions of the task list.

```
def seek_plan(state, tasks, plan):
    if tasks == []:
        return plan
    task1 = tasks[0]
    if task1[0] in operators:
        operator = operators[task1[0]]
        newstate = operator(copy.deepcopy(state),*task1[1:])
        if newstate:
            solution = seek_plan(newstate, tasks[1:], plan+[task1])
            if solution != False:
                return solution
    if task1[0] in methods:
        relevant = methods[task1[0]]
        for method in relevant:
            subtasks = method(state,*task1[1:])
            if subtasks != False:
                solution = seek_plan(state, subtasks+tasks[1:], plan)
                if solution != False:
                    return solution
    return False
```

chosen method failed in finding a plan. If a plan is found, no other decomposition branch will be checked, so the algorithm finishes when finding the first feasible plan.

Another important thing to be taken into account is that always the first task of the task list is retrieved, and that when inserting new subtasks, these are inserted in the first positions of the task list. In this manner PYHOP constructs plans in the same order that they will be later executed, knowing at each planning step the current state of the world. This also implies that a depth-first search algorithm is being used.

As it has been explained in Section 5.1, we want to apply the idea of a guided heuristic search from the A* algorithm to PYHOP, so several modifications were done to the PYHOP code to achieve this. To avoid distracting the attention of the reader on implementation details related with the Python language, from now all the code related with these modifications will be shown as pseudocode.

We want an algorithm that on each iteration retrieves and expand the node with the lowest *f(n)* value. So the first decision made over PYHOP was to avoid using a recursive

Code 5.2 Pseudocode for the State class definition. One state is initially composed of two variables of type List, one for the tasks that need to be decomposed and one for the tasks that have been decomposed and conform the resulting plan. Additional variables may be added as needed, depending on the problem domain.

```
Class State {
  List task_list ;
  List plan_list ;
};
```

function and use instead an iterative function. This decision does not affect the behaviour of PYHOP, as any recursive function has an iterative counterpart and vice versa.

After thinking in the iterative counterpart of PYHOP we realized that adding the characteristics of the A* algorithm to the PYHOP algorithm could be more easily done if the code of the PYHOP algorithm is inserted instead on the skeleton of a Python implementation for the A* algorithm. As explained previously, the A* algorithm has four function definitions that are problem specific and that characterize how the different problems are solved, so that the rest of the A* pseudocode shown in Algorithm 3 remains unchanged for any kind of problem. To insert the PYHOP functionality into the A* algorithm we only have to center our attention into the *Is_Goal* function and the *Get_-Neighbor_List* function, introducing the PYHOP functionality into these two functions and leaving the implementation of the other two for later. But before explaining the implementation of functions it is important to define first how the states are defined in our new planner.

As commented before, in PYHOP the current state of the world is represented as a Python object with variable bindings that is copied and modified into a new state object when an operator is applied. We have kept this decision in our planner, but added the requirement that each state object must have two required variable bindings: the current task list and the current plan list. We saw in Code 5.1 that on each recursive call, a copy of the task and plan lists is done and modified to be passed as arguments for the following recursive call. By this way, if a backtrack is triggered, then the original task and plan lists are found unaltered. Since we now want an iterative version of the algorithm and need to store the different states in a priority queue, we need to store for each state its related task and plan lists as a variable binding. As it was seen in Code 5.1, each task element of the task list is indeed a list where the first element is a string that represents the name of the task and the rest of elements represent the parameters or arguments for the task. The same happens for the plan list. Code 5.2 shows the pseudocode definition for our State object. This object initially has two variable bindings, the task list and the plan list. Depending on the problem domain, the user can add as many variable bindings as needed.

The implementation of the *Is_Goal* function was trivial. Since we know that in HTN the goal is to have all tasks decomposed, in this function we only have to check that the task list is empty. Algorithm 4 shows the pseudocode for the function. It checks that the state that is passed as argument has an empty task list, which means that all tasks have been decomposed and thus, a plan has been found.

Algorithm 4: Pseudocode for the *Is_Goal* function. The function only checks that the task list for the given state is empty, which means that all tasks have been decomposed and thus, the given state is a goal state.

```
Function Is_Goal(State state):
    return state.task_list.is_empty();
```

The implementation of the *Get_Neighbor_List* function was also easy. In this function is where the main functionality of PYHOP has been inserted, with some modifications. The purpose of this function is, given an input node, to return the list of all its neighbour nodes, that is to say, the list of all nodes that can be visited from the input node. In HTN terms, this is equivalent to returning all possible decompositions that can be done from the input node. Algorithm 5 shows the pseudocode for the implementation of the *Get_Neighbor_List* function.

With the implementation of the *Is_Goal* function and the *Get_Neighbor_List* function we now have our new planning system complete. The SHOP* HTN planner is a combination of the PYHOP and A* algorithms. A* brings its main scheme, the guided heuristic search, while PYHOP brings its HTN planning system, embedded on the A*. Since we are embedding the PYHOP functionality into the A* algorithm, our new planner SHOP* is inheriting two of the most important characteristics of A*:

- The algorithm is proven to be **complete**. If a solution exists, then it is guaranteed to be found. Otherwise, it correctly reports that no solution is possible.
- If the heuristic function *h(n)* is admissible, meaning that it never overestimates the distance from the current node to the closest goal node, then the solution found by the planner is **optimal**.

All of this have been possible by implementing the *Is_Goal* and *Get_Neighbor_List* functions but, what happens with the *Dist* function and *h(n)* heuristic function from A* and the operators and methods from PYHOP? These functions are problem specific, meaning that their implementation depend on the type of problem that needs to be solved. For this reason these functions have not been shown here. They will be implemented in the next section, where a specific problem is presented, defined and solved: the Travelling Salesman Problem [47] in the context of structure inspection operations of the AEROARMS project.

Algorithm 5: Pseudocode for the *Get_Neighbor_List* function. This function receives an input state and returns a list of all neighbour states that can be reached from the input state. In HTN terms, this is equivalent of returning all possible decompositions that can be done from the input state. The first task in the task list of the input state is recovered. If an operator can be applied to the task, then the task is removed from the task list of the input state and added to the plan list of the input state, and the new state is updated accordingly, returning a neighbour list composed of only one neighbour. If any method can be applied to decompose the task, then the subtasks resulting of applying each of the applicable methods to the task are computed, and the task is replaced from the task list with the subtasks, updating the new states accordingly. In that case, each applicable method and variable binding generates a new state that is inserted in the neighbour list.

```
Function Get_Neighbor_List(State state):
    $neighbor\_list = new\ Empty\_List(State)$;
    $task = state.task\_list[0]$ if an operator $\rho$ can be applied to task then
        $newstate$ = resulting state of applying $\rho$ to $task$;
        $newstate.task\_list$ = result of removing $task$ from $state.task\_list$;
        $newstate.plan\_list$ = result of adding $task$ to the end of $state.task\_list$;
        $neighbor\_list.put(newstate)$;
    else
        for each method $\phi$ and variable binding $\theta$ that can be applied to task do
            $subtasks$ = result of decomposing $task$ by applying $\phi$ with $\theta$;
            $newstate$ = new copy of $state$;
            $newstate.task\_list$ = result of replacing $task$ with $subtasks$ in
              $state.task\_list$;
            $neighbor\_list.put(newstate)$;
    return $neighbor\_list$;
```

5.3 Use Case: Testing the Optimality of SHOP*

In the previous sections we have detailed and explained the steps done to implement our new planner, the SHOP* HTN planning system. The planner has been implemented by using the idea of the A* algorithm, the use of a guided heuristic search instead of the traditional use of a depth-first search for discovering new states. Two of the main advantages that we claim by using the SHOP* HTN planner are its completeness and, under certain conditions, its optimality.

In this section we present an use case in the context of the AEROARMS project to test the optimality of the planner.

5.3.1 Problem Statement

Let us consider a mission $\mathscr{M}$ consisting on inspecting a given structure to check the condition and integrity of the building, such as in the AEROARMS project. There are several ways of doing this, depending on the type of the structure. For example, if the structure consists on pipes that transport hot gases, a thermal camera can be used to find gas leaks. Or more generally, a laser profilometer can be used to check the surface condition of the pipes. In any case, the inspection has to be done by using an aerial vehicle that starts the mission on its home location, previously known, and the inspection will be done in specific key points of the structure, also previously known. The aerial vehicle is equipped with different sensors, each one suitable for different pipe types: a thermal camera for pipes with hot gases, a profilometer for pipes with gases at room temperature, etc. Before inspecting each key point, the aerial vehicle must decide which sensor to use and execute the actions needed to ensure that the inspection is correctly done, such as sending a camera focus command, triggering the photo, starting the profilometer, etc. Let us define $\mathscr{L}$ as the set of locations where the inspection has to be done, that is, the key points, and $\mathscr{H}$ as the starting home location of the aerial vehicle. The objective is to inspect the whole structure visiting each of the locations of $\mathscr{L}$ only once, starting from the home location $\mathscr{H}$ and finishing in the same home location, minimizing the total distance travelled during the mission.

As can be seen, the implicit combinatorial problem can be expressed by the edges of a graph $G_N(V,E)$ with N vertices, where the vertices V are the union of $\mathscr{L}$ and $\mathscr{H}$ and the edges E are the weighted straight line connections between every two locations, with the weight representing the Euclidean distance between the locations. Let us define more formally the problem by introducing some definitions.

Definition 5.3.1 (Complete Graph) A complete graph G_N is a graph with N vertices and an edge between every two vertices.

Definition 5.3.2 (Weighted Graph) A weighted graph is a graph in which each edge is assigned a weight representing the cost of traversing that edge.

Definition 5.3.3 (Hamilton Circuit) A Hamilton circuit is a circuit that uses every vertex of a graph once.

Given the previous definitions, we can define our problem as the problem of finding a minimum-weight Hamilton circuit in G_N that starts in $\mathscr{H}$. Part of the problem described here is very similar to the well-known Travelling Salesman Problem (TSP) [47].

5.3.2 SHOP* Problem Domain

To model the SHOP* domain for the problem presented in the previous subsection we need first to design a hierarchical task network to represent the domain. Figure 5.2 shows the task network designed. The ellipse shaped nodes represent methods, the rectangle shaped nodes represent operators and the diamond shaped nodes represent preconditions. Nodes grouped by a box represent subtasks of the same decomposition. Purple arrows represent the order on which subtasks have to be executed inside a decomposition, left to right in the figure. *S'* and *S''* represent modified states over the input state *S*. Only operators can modify states. The input arguments for the methods and operators are inside parenthesis. The state is always an argument for each method and operator.

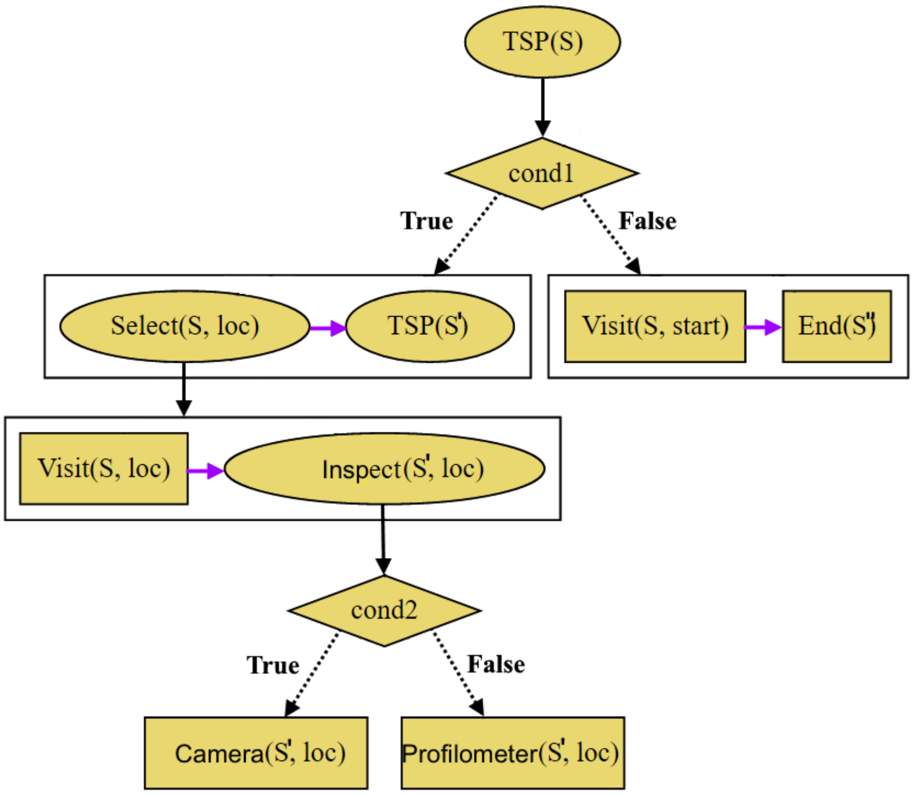


Figure 5.2 Task network designed for the TSP-derived problem presented in Subsection 5.3.1. The ellipse shaped nodes represent methods, the rectangle shaped nodes represent operators and the diamond shaped nodes represent preconditions. Nodes grouped by a box represent subtasks of the same decomposition. Purple arrows represent the order on which subtasks have to be executed inside a decomposition, left to right in the figure. *S'* and *S''* represent modified states over the input state *S*. Only operators can modify states. The input arguments for the methods and operators are inside parenthesis. The state is always an argument for each method and operator. Precondition *cond1* checks if in the current state *S* there are locations that remain unvisited. Precondition *cond2* checks the pipe conditions, which is needed to choose the most appropriate sensor between the two available to do the inspection.

Some variable bindings have been added to the State class to ease the implementation of the domain: one object of type *Set* with the locations that have been visited and another *Set* object with the locations that remain unvisited. In addition to these two sets, the starting location and the current location of the aerial vehicle have been included, as well as a reference to a dictionary that holds for each location its cartesian coordinates. At the starting state, the set of visited locations is empty and the set of unvisited locations contains all target locations, with the exception of the start location of the aerial vehicle from which the mission starts.

The top-level method is called *TSP* and receives as input the current state of the world. This method can generate two types of decompositions depending on the logical value of the precondition *cond1*. This precondition checks if in the current state *S* there are locations that remain unvisited. If that is not the case, then the aerial vehicle has to move from its current location to the start location and finish the mission; these subtasks conform a decomposition group, thus the returned decomposition contains two subtasks, *Visit* and *End*, for which both operators have been implemented. Only one decomposition is returned, as there is no need for variable binding. The *Visit* operator modifies the state to exclude from the unvisited set the location passed as argument and to include that location in the set of visited, also updating the current location of the aerial vehicle. The *End* operator is a convenience operator that signals the end of a mission, and does nothing over the received state. Its only purpose is to appear at the end of the resulting plan.

In the case that there are locations that remain unvisited, then the TSP method returns a different decomposition, composed of two subtasks. The first is the *Select* subtask, for which a method has been implemented, and the second is a TSP subtask. The *Select* subtask has the purpose of selecting one location from the unvisited set. After selecting a location and computing the decomposition for the *Select* task, the input state *S* is modified, so the following call for the *TSP* method is done with the new state *S'*, which now contains one less location to visit. So, the TSP method can be seen as a recursive method that on each call decomposes a problem of smaller size (one location less to visit). It is needed to say that for this decomposition group, a variable binding for the *loc* variable is needed, as it can take as value any of the locations that appear in the unvisited set. So, the *Select* method will generate as many decompositions as values can take the *loc* variable.

Given a variable binding, the *Select* method generates a third decomposition group. This group is composed of a *Visit* subtask for the given location, and a *Inspection* subtask. For this last, a method has been implemented that generates two possible subtasks depending on the value of *cond2*. The purpose of this condition is to check the pipe type and choose the most appropriate sensor between the two available: the thermal camera of the profilometer. For each sensor, an operator has been implemented that models the actions needed to use the specific sensor type. These operators do not modify the state received as argument.

After explaining the HTN network implemented, from the A* algorithm shown in Algorithm 3 there are two functions left to complete the domain for our SHOP* planner: the *Dist* function and the *Heuristic_Cost* function.

The *Dist* function computes the distance (cost) of going from the current node to one of its neighbours. This function is needed to compute the *g(n)* value of A*. In our problem, the cost of going from one state *state1* to a neighbour state *state2* is the cost of going from the current location where the aerial vehicle is in *state1* to the location where the aerial vehicle is in *state2*. This cost has been modelled as the Euclidean distance between locations. Algorithm 6 shows the pseudocode for our *Dist* function.

The *Heuristic_Cost* function computes the estimated distance of going from a node to the closest goal node, and represents the value of *h(n)* in A*. As we want to show the

Algorithm 6: Pseudocode for the *Dist* function. The cost of going from one state *state1* to a neighbour state *state2* is the cost of going from the current location where the aerial vehicle is in *state1* to the location where the aerial vehicle is in *state2*. This cost has been modelled as the Euclidean distance between locations.

```
Function Dist(State state1, State state2):
    return $euclidean\_distance(state1.current\_loc, state2.current\_loc)$;
```

optimality of our planner, we need to choose an admissible heuristic. So, let us first define the concept of a *Minimum Spanning Tree* (MST).

Definition 5.3.4 (Spanning Tree) A spanning tree of a graph is a subgraph that contains all the vertices of the graph and is a tree.

Definition 5.3.5 (Minimum Spanning Tree) From all spanning trees of a graph, the minimum spanning tree is the one with the minimum sum of the edge costs of the tree.

Figure 5.3 shows an example of a minimum spanning tree. The graph from the left has multiple spanning trees, but the one with the minimum sum of edge cost is shown at the right of the figure.

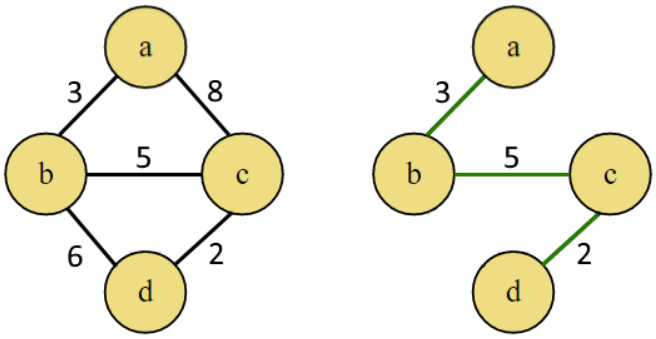


Figure 5.3 Minimum spanning tree example graph. The graph from the left has multiple spanning trees, but the one with the minimum sum of edges cost is shown at the right of the figure. In that case, its sum value is of 10 units.

Given the definition of a minimum spanning tree, we have implemented the heuristic function for our problem as follows: given a state n, the heuristic function *h(n)* is computed as the sum of the distance to the nearest unvisited location from the current location plus the estimated distance to travel all the unvisited locations plus the nearest distance from an unvisited location to the start location. The estimated distance to travel all the unvisited locations is computed by creating a weighted graph from all the unvisited locations and computing the sum of the edge costs of its minimum spanning tree. By this way, our heuristic function *h(n)* never overestimates the cost of going to the closest goal node, so we have an admissible heuristic function.

Algorithm 7 shows the pseudocode for our *Heuristic_Cost* function. To compute the minimum spanning tree of a graph there are multiple algorithms, but in this thesis we have used the Python implementation of the Kruskal's algorithm [68, 15].

Algorithm 7: Pseudocode for the *Heuristic_Cost* function or *h(n)*. Given an input state, the heuristic function is computed as the sum of the distance to the nearest unvisited location from the current location plus the estimated distance to travel all the unvisited locations plus the nearest distance from an unvisited location to the start location. The estimated distance to travel all the unvisited locations is computed by creating a weighted graph from all the unvisited locations and computing the sum of the edge costs of its minimum spanning tree. This is an admissible heuristic as it never overestimates the cost to go to the closest goal node.

```
Function Heuristic_Cost(State state):
    $cost = 0$;
    $cost += distance\_to\_nearest\_unvisited(state.current\_loc, state.unvisited)$;
    $cost += minimum\_spanning\_tree\_cost(state.unvisited)$;
    $cost += distance\_to\_nearest\_unvisited(state.start\_loc, state.unvisited)$;
    return $cost$;
```

5.3.3 Solving a Specific Case

In this subsection we will present a first use-case to test the optimality of our new planner. Although a deeper study with simulation results where three different techniques for solving the problem in Subsection 5.3.1 has been done and is presented in the next section, presenting and solving here a simple example can be useful to understand how our new planner works.

Given an aerial vehicle that starts the mission on a specific location, our objective is to check the integrity of a given structure that is primary composed by gas pipes. The mission presented here is an inspection task, such those targeted by the AEROARMS project. The locations of interest or key points where the aerial vehicle has to do the inspection are known before the mission starts. From its starting position, the aerial vehicle has to visit each location once, returning from the last visited location to the start location, and minimizing the distance travelled during the mission.

Figure 5.4 shows the input problem. Ten locations where the aerial vehicle has to do the inspections have been defined, marked in red color. The starting location is marked in green color. All locations are expressed in cartesian coordinates, and the distances between locations are computed as Euclidean distances. The measurement unit has been omitted in the axes because it depends on the coordinate system used, and this can be configured for the domain (the measurement unit can be configured to be meters, kilometres, and so on).

To test if SHOP* was capable of computing the optimal solution by using the heuristic function defined in Algorithm 7, a brute force algorithm was used to solve the problem. The brute force algorithm checks all the possible routes for the aerial vehicle and returns the one with the minimum length.

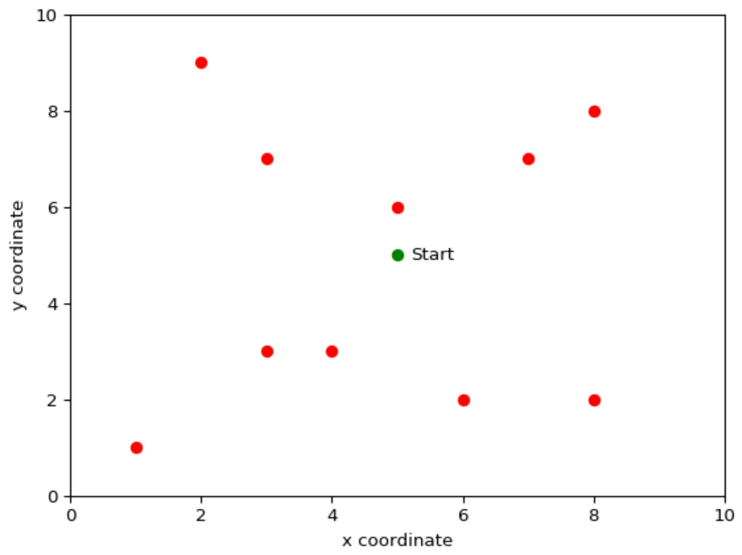
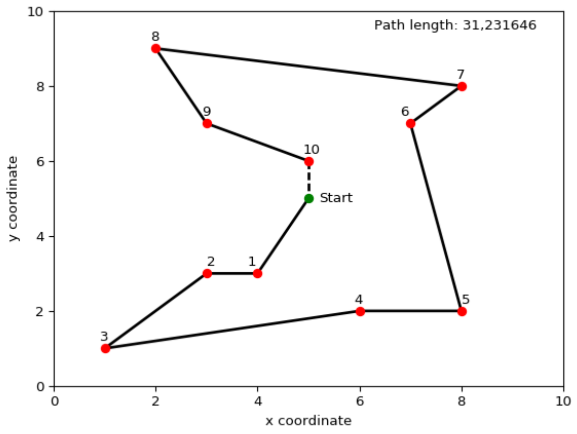
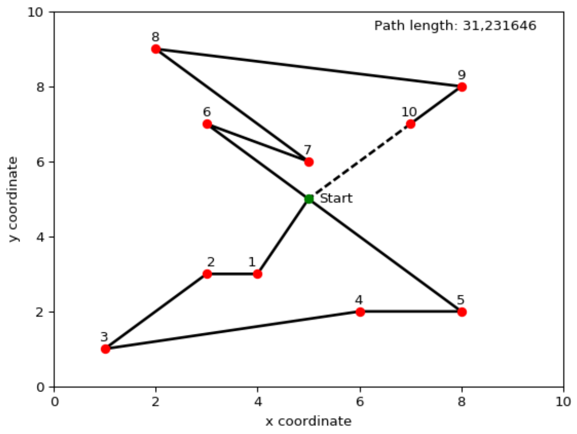


Figure 5.4 Use case for testing the optimality of SHOP*. Ten locations where the aerial vehicle has to do the inspections have been defined, marked in red color. The starting location is marked in green color. All locations are expressed in cartesian coordinates, and the distances between locations are computed as Euclidean distances. The measurement unit has been omitted in the axes because it depends on the coordinate system used, and this can be configured for the domain (the measurement unit can be configured to be meters, kilometres, and so on).

Figure 5.5 shows the graphs resulting of applying the brute force algorithm and the SHOP* algorithm to the input problem. The numbers over the red locations represent the order on which the locations are visited on each graph. The dashed line represents the return of the aerial robot from the last visited location to the start location. Both algorithms returned a path length of 31,231646 units (rounded to six decimals, but a higher precision is achieved), so from all possible solutions to the problem, our SHOP* algorithm was capable of computing the optimal solution. It is interesting to see that, although both algorithms returned the same path length, the graphs are different, as the order on which the locations are travelled varies from one to another. That means that, for the problem presented in Subsection 5.3.1, there may be different optimal solutions, and our planner only returns one of them.



(a) Brute force graph result for the input problem of Figure 5.4.



(b) SHOP* graph result for the input problem of Figure 5.4.

Figure 5.5 Resulting graphs of applying the brute force algorithm and SHOP* to the input problem. Figure 5.5a shows the solution returned by the brute force algorithm. Figure 5.5b shows the solution returned by the SHOP* planner. The numbers over the red locations represent the order on which the locations are visited on each graph. The dashed line represents the return of the aerial vehicle from the last visited location to the start location. Both algorithms returned a path length of 31,231646 units (rounded to six decimals, but a higher precision is achieved), so from all possible solutions to the problem, our SHOP* algorithm was capable of getting the optimal solution. Although both algorithms returned the same path length, the graphs are different, as the order on which the locations are travelled varies from one to another. That means that, for the problem presented in 5.3.1, there may be different optimal solutions and our planner only returns one of them.

5.4 Simulation Results

Different simulations have been carried out to compare three different options to solve the problem presented in Section 5.3. The options that have been evaluated are:

- The coupled approach presented in this chapter, using our new SHOP* HTN planner.
- The use of the original JSHOP2 planner.
- The decoupled approach presented in Chapter 4, consisting on the dual-planner interconnection OptaPlanner-JSHOP2.

The domain designed for the coupled approach, our new SHOP* HTN planner, was explained in Section 5.3. In that section, the hierarchical task network designed for solving the problem was presented and shown in Figure 5.2. This hierarchical task network have been reused in the second option, the original JSHOP2 planner.

For the third option, the dual-planner interconnection OptaPlanner-JSHOP2, some modifications have been done. In first place, now OptaPlanner is used as a TSP solver instead of a VRP solver as it was used in Chapter 4. The score for the OptaPlanner domain has been configured to use only one type of constraint: a soft-constraint that measures the distance travelled by the aerial vehicle. The solver has been configured to use one Construction Heuristic phase plus a Metaheuristic Phase. As Construction Heuristic the First Fit algorithm was used. As Metaheuristic, the five local search algorithms tested in Section 4.7 have been used: Simulated Annealing, Late Acceptance, Tabu Search, Hill Climbing and Step-Counting Hill Climbing. Additionally, a Brute Force algorithm has been used. For all the algorithms, a time limit of five minutes was configured. In the figures presented in this section they are referenced as SA, LA, TS, HC, SC and BF respectively. After computing the order on which the locations must be travelled to minimize the distance (the Euclidean distance between each pair of locations is used), OptaPlanner calls the JSHOP2 process passing the ordered list of locations as argument. Then, from this ordered list, a task network is decomposed to obtain the low-level plan. As the task network knows the order on which the cities have to be visited, for this purpose we have changed slightly the task network used in the other two approaches. The new hierarchical task network is shown in Figure 5.6. The main difference with respect to the original network is that now the TSP method receives an additional argument, a list *order* with the unvisited locations in the order on which they must be visited. The first consequence of this change is that now the *Select* method does not need a variable binding for the *loc* variable, as it will take as value the first location on the list. The second consequence is that the *Visit* operator must modify the *order* list to remove the first location from the list, so that the following call to the TSP method receives correctly the updated *order* list. Henceforth, when we talk about OptaPlanner we are making reference to the dual planner interconnection OptaPlanner-JSHOP2.

For the tests, six data sets have been generated with problem sizes of 10, 15, 20, 25, 30 and 35. The problem size is given by the number of locations that the aerial vehicle must visit. Each of the data sets is composed of five different problems where the coordinates for the locations have been generated randomly to be included in the range *[0, 100]* on

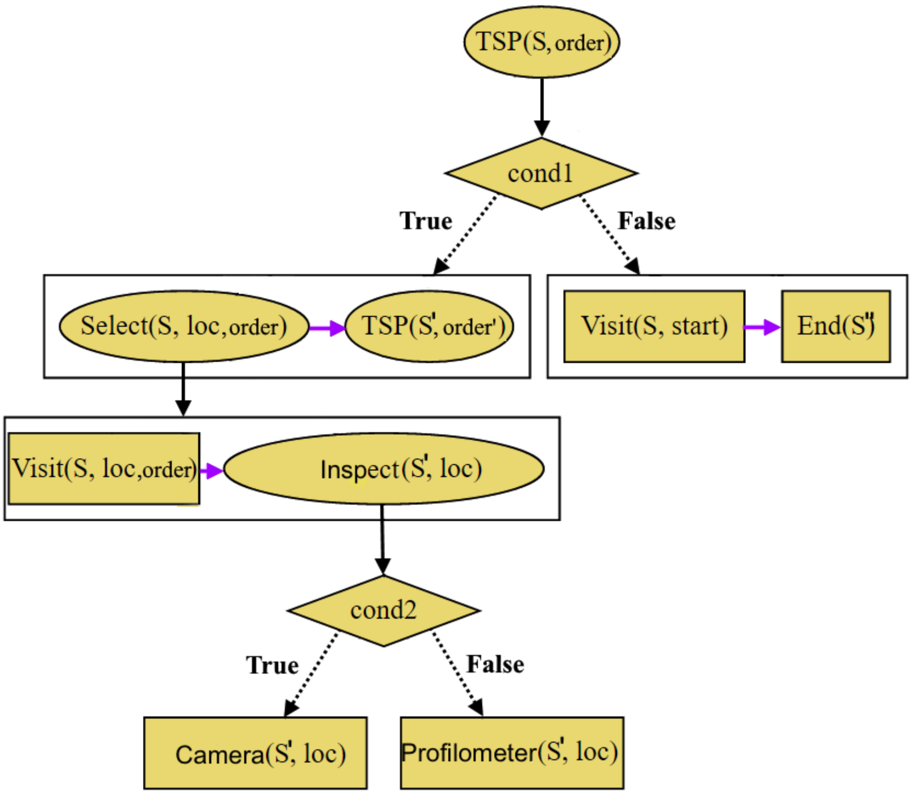
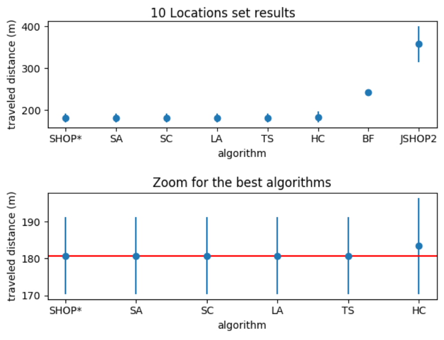


Figure 5.6 Redesigned task network for the OptaPlanner-JSHOP2 test. The new hierarchical task network differs slightly from the task network of Figure 5.2. The main difference is that now the TSP method receives an additional argument, a list *order* with the unvisited locations in the order on which they must be visited. The first consequence of this change is that now the *Select* method does not need a variable binding for the *loc* variable, as it will take as value the first location on the list. The second consequence is that the *Visit* operator must modify the *order* list to remove the first location from the list, so that the following call to the TSP method receives correctly the updated *order* list.

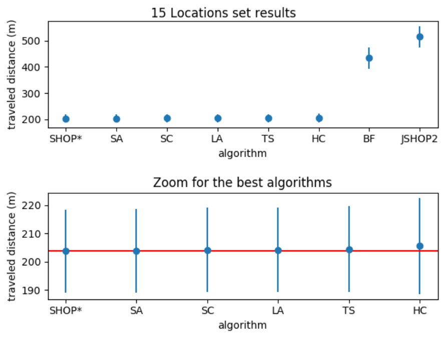
each axis. Thus, we have 30 different problems to solve. The tests have been done on a machine with an Intel i7 CPU at 2 GHz and 8GB RAM. The goal of the simulations is to compare the performance of the three different options (OptaPlanner-JSHOP2, SHOP* and JSHOP2).

The measurement unit in the simulations are meters. One purpose of the simulations was testing the quality of the computed solutions. The quality of the solutions is given by its score, which measures the total distance travelled in meters by the aerial vehicle when traversing the locations. We use the Euclidean distance between each pair of locations.

The lower the distance, the better the score is. Figures 5.7, 5.8 and 5.9 show the mean and standard deviations values for the algorithms tested. Note that as the SHOP* algorithm always computes the optimal solutions as it uses an admissible heuristic, its mean and standard deviation values are the best possible, so they are considered as the optimal mean and standard deviation and thus, serve as reference for the other algorithms. The Brute Force and JSHOP2 algorithms are far from the others, so those are the two algorithms that offered the worst results. In the case of Brute Force, this bad result is because it reached the time limit without exhausting the search, so it did not find the optimal solution. For the remaining, it can be seen that the OptaPlanner algorithms computed results that are near to the optimal mean computed by SHOP* in all data sets, but the difference seems to grow slowly as the problem size increases. So, the SHOP* planner computed the best solutions.

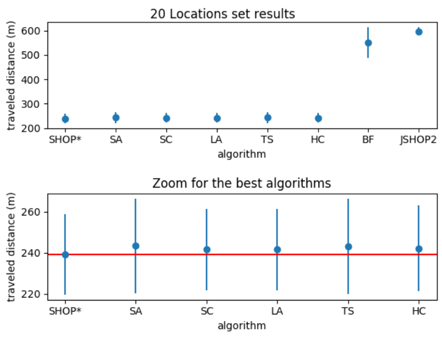


(a) 10 Locations set result.

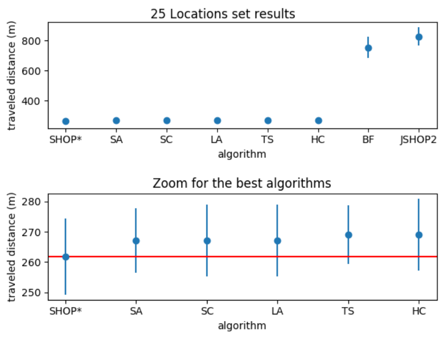


(b) 15 Locations set result.

Figure 5.7 Results for the 10 and 15 locations data sets. For every set, the upper graph shows the results for all the algorithms tested, the computed travelled distance. As the results of two of the algorithms, Brute Force and JSHOP2, were very far in value from the others, an additional graph is shown below showing a zoom for the six best algorithms and displaying a red line that represents were the optimal mean value for the data set is located (the optimal mean is the mean computed from the optimal solutions of a data set). This line will always contains the point that represent the mean of the SHOP* algorithm, as it always gets the optimal solutions due to the use of an admissible heuristic. For every algorithm, the mean and standard deviation are shown. The SHOP* mean and standard deviation represent the optimal mean and standard deviation, so it is used as a reference. Brute Force and JSHOP2 computed the worst results, very far from the other algorithms in both data sets. In the case of Brute Force, this is because it exhausted the time limit before finding the optimal solution. The rest of algorithms computed solutions that were optimal or very near to the optimal. For the 10 locations data set, all got the optimal mean and standard deviation, with the exception of the Hill Climbing algorithm. For the 15 locations data set, in addition to Hill Climbing, the Tabu Search algorithm did not get the optimal mean.

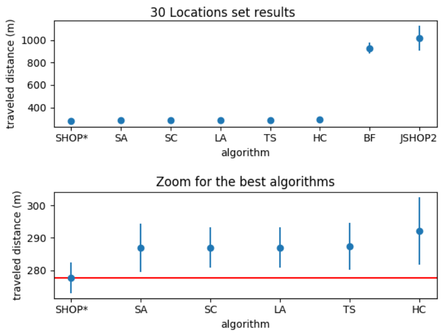


(a) 20 Locations set result.

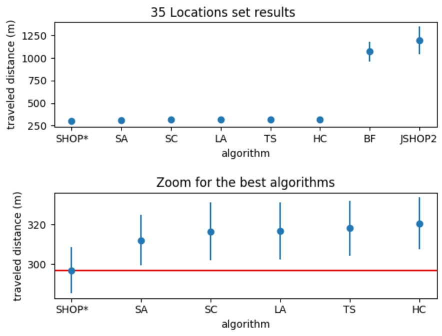


(b) 25 Locations set result.

Figure 5.8 Results for the 20 and 25 locations data sets. Again, the results computed by the Brute Force and JSHOP2 algorithms are the worst, very far from the rest in both data sets. On this time, any of the other algorithms was capable of getting to the optimal mean of SHOP*. This can be better seen for the 25 locations data set, where the distance to the optimal mean starts to grow.



(a) 30 Locations set result.



(b) 35 Locations set result.

Figure 5.9 Results for the 30 and 35 locations data sets. As it happens with the data sets shown in Figure 5.8 and 5.7, the Brute Force and JSHOP2 algorithms are even further from the rest, so at this point we can affirm that those are the two algorithms that offered the worst results. For the rest, it can be seen that they still computed results that are near to the optimal mean computed by SHOP* in both data sets, but the difference seems to grow slowly as the problem size increases, as it can be seen comparing the results in Figure 5.9a with the results in Figure 5.9b.

The consumption of memory has been measured in Figures 5.10 and 5.11. It can be seen that the algorithm that consumes the least amount of memory is JSHOP2, using near zero MBytes of memory in most of the data sets (it usually consumed few kilobytes).

Following that are the OptaPlanner algorithms, which consumed amounts of memory near the 100 Mbytes. Those algorithms have shown a very similar behaviour in memory consumption, having very similar means and standard deviation values, so we have omitted additional plots for them. As JSHOP2, the memory consumption for OptaPlanner does not increase with the problem size, which traduces in a high scalability. Finally, SHOP* was the algorithm that used a higher amount of memory, which increases with the problem size. SHOP* memory consumption has been proven to increase linearly with the execution time for this problem domain, as it can be seen in Figure 5.11, being therefore the worst in memory consumption.

Finally, as Simulated Annealing has been proven to be the best from all the OptaPlanner algorithms for this domain (we also got this conclusion in Section 4.7, where it tied with Step Counting Hill Climbing), and as JSHOP2 is unable to optimize its solutions, we have finally centred our attention in comparing the optimization curves for SHOP* and Simulated Annealing, shown in Figure 5.12.

Each curve is an approximation made with a third degree polynomial from the results of the six data sets, and represent how each algorithm optimizes the solutions as time goes by. Both algorithms tend to decrease the computed best scores as time increases (lower values on the *y* axis are better), but from a given time instant, the SHOP* curve gets better (lower) values. That means that our guided heuristic search drives the search faster to compute better values than Simulated Annealing.

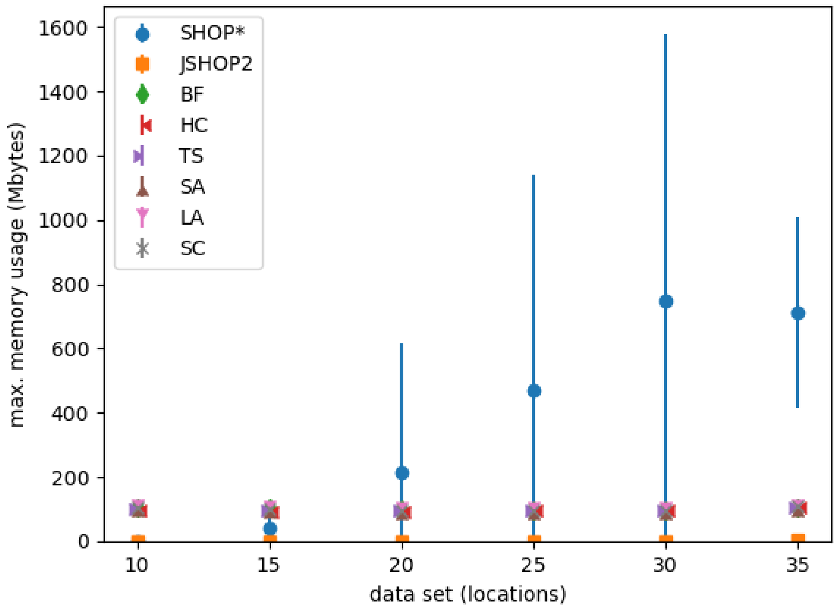


Figure 5.10 Maximum memory usage per data set. From all the algorithms, the one that needed the least amount of memory was the JSHOP2 algorithm, having checked that it used no more than 23 Mbytes in its worst case. All of the algorithms used by OptaPlanner (Brute Force, Hill Climbing, Tabu Search, Simulated Annealing, Late Acceptance and Step Counting Hill Climbing) showed a very similar usage of memory, with almost identical mean values and very low standard deviations. For this reason, they appear overlapped. For clarity purposes, we omit a zoom for these algorithms because its behaviour in memory use is almost the same, but it is interesting to see that they always use more or less the same amount of memory, around 100 Mbytes independently of the problem size. This gives us an idea of their good scalability. SHOP* was the algorithm that used the greatest amount of memory, which increased with the problem size so it is the algorithm with worst scalability. It is interesting to see its high standard deviation values. As it will be explained later (see Figure 5.11, the SHOP* memory consumption increases linearly with the execution time. So, for a given problem, the amount of memory used is given by the execution time needed to solve it: problems that are solved soon use less memory than problems that are solved later. As in a specific problem the distance of the goal state from the root of the search tree is random (as the visit and start locations have been generated randomly), then for each data set we can have problems that are solved very soon or very late, which is traduced in high standard deviation values for memory usages.

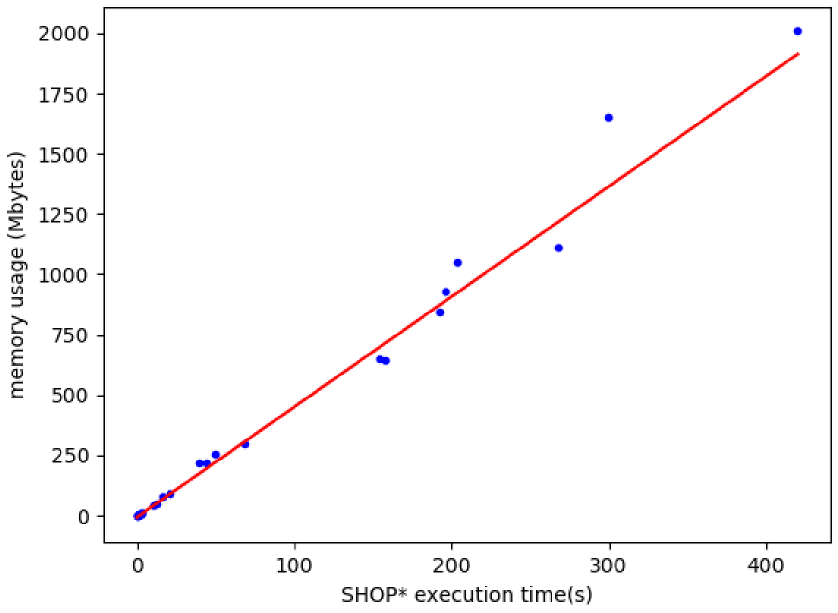


Figure 5.11 SHOP* memory consumption per execution time. From all the solved problems for the six data sets, a total of 30 points (5 per data set) that represent the memory usage per execution time is represented. The points have been approximated with a least squares line. As it can be seen, the memory usage of SHOP* increases linearly with the execution time. Each discovered state must be kept in memory until the algorithm finishes, because each state can be re-discovered and its score improved, so the use of memory increases with the execution time. It is important to clarify that the amount of memory increased depends on the problem: for example, problems with a large combinatorial nature will produce more states on each iteration and consume more memory. Thus, the plot shown in this figure must be seen as the memory consumption curve for the specific problem solved, and not as the general memory consumption curve for SHOP*.

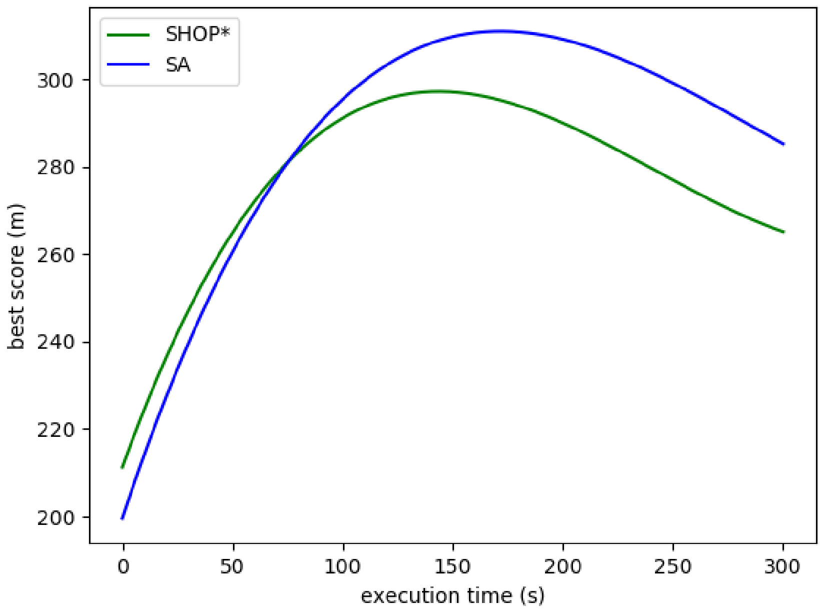


Figure 5.12 SHOP* and Simulated Annealing Optimization Curves. Each curve represents the best score as a function of the execution time. Each curve is an approximation made with a third degree polynomial from the results of the six data sets, and represent how each algorithm optimizes the solutions as time goes by. As from all the tested OptaPlanner algorithms Simulated Annealing offered the best results, for clarity purposes we omit the curves for the rest of algorithms. The lower the distance in meters, the better the solution is, so lower values in the *y* axis are better. Both algorithms tend to decrease the computed best scores as time increases, but from a given time instant (near second 75 approximately), the SHOP* curve gets better (lower) values. That means that the guided heuristic search drives the computed solutions faster to better values than Simulated Annealing.

5.5 Conclusions

In this chapter we have addressed the combination of geometric and symbolic reasoning by following a coupled approach.

A new HTN planner has been developed, based on the ideas of the A* algorithm. The depth-search algorithm of the Python implementation of JSHOP2 has been replaced by our new A*-based search algorithm.

The main idea of our new SHOP* HTN planner is to perform a guided heuristic search. The purpose of the guided heuristic search is to replace the depth-first algorithm present in most HTN planners and drive the search towards the best possible solution expanding first the more promising nodes. For domains that involve some kind of geometric reasoning, a geometric projection of the decomposition nodes based on geometric computation, such as using Euclidean distances and so on, can be used, avoiding the needs of using an external geometric planner.

Our new HTN planner is capable of computing the optimal solution if the heuristic function provided to the system is admissible. Although the planner is capable of working in any type of domains, single or multi-vehicle, for clarification purposes we have tested the planner in a single-vehicle domain example against a brute force algorithm to check that effectively, when the admissibility criterion is satisfied by the heuristic function, then the optimal solution is returned.

Several tests have been additionally done to measure the performance of our new HTN planner against the original JSHOP2 implementation and the decoupled approach OptaPlanner-JSHOP2 tested in the previous chapter, in the context of the AEROARMS project for a single aerial vehicle. We have demonstrated that only SHOP* was capable of computing the optimal solution in all data sets. SHOP* has proven to be the fastest algorithm after the original implementation of JSHOP2, which computed the worst score results.

Future work include the optimization of the SHOP* algorithm to decrease the memory consumption. In addition, the performance of our new planner has to be measured in multi-vehicle domains, as they usually involve higher computational loads.

6 Conclusions and Future Developments

This chapter summarizes the main contributions of the thesis and highlights its main results. Advantages and disadvantages of the proposed approaches are discussed. Finally, in order to overcome the drawbacks, some guidelines are introduced for future improvements.

6.1 Conclusions

The application of symbolic Hierarchical Task Network (HTN) planning in the resolution of Vehicle Routing Problems (VRP) in domains involving unmanned vehicles is addressed in this thesis, in the context of assembly and inspection operations .

The different ways to integrate symbolic and geometric reasoning researched and developed along the last years are categorized in Chapter 2. From the literature, three different categories have been clearly distinguished and identified: symbolic layer calls the geometric layer, geometric layer calls the symbolic layer, and sample in the compound state. When connecting geometric and symbolic layers, one of these three categories needs to be applied.

Even though there are many powerful symbolic planning methods, we focused our work in HTN because hierarchies resemble the way the humans think, act and organize the world, as we explained in Chapter 3. That makes the use of HTN planners for solving real-world problems a natural approach. Modelling the domain requires a bigger effort compared to other planning techniques, because the knowledge must be provided by a human that has some expertise in the matter, but the domain is more easily understandable by any person. This knowledge gives a boost in terms of performance and coverage across many domains to HTN planners compared to classical planners. The higher performance of this planning technique makes the difference against other classical choices, justifying our selection of HTN planning for problem solving. In addition, the JSHOP2 HTN planner has been presented as our selected HTN planner for the research. Its modelling language is a direct translation of PDDL. It has been shown to be sound and complete, and it has been also adapted to be integrated in multi-agent environments, being capable of interacting with external agents and making queries to distributed heterogeneous information sources. Because it is widely extended and accepted in the research community, we chose this planner among the different possibilities.

Two approaches to integrate geometric reasoning with the HTN planner have been studied, presented and tested. The first approach consists on a new score-based optimization method for connecting a VRP planner with the HTN planner. We call this the *decoupled approach* as we solve the lack of geometric reasoning of the symbolic HTN planner by connecting it to a separate planner that communicate with it and feeds it with the missing geometric information. The second approach, which we call the *coupled approach*, consists on the development of a new HTN planner to perform a guided heuristic search that enhances the performance of the planner and eases the use of geometric information in the HTN planning engine, without needing to connect it to a VRP planner.

6.1.1 Decoupled Approach

The decoupled approach was presented in Chapter 4 and consists on a new score-based optimization method that allowed us to connect two independent planning systems, the geometric planner and the HTN planner, in the context of structure assembly missions that involve the VRP. We solve the lack of geometric reasoning of the symbolic HTN planner by connecting it to a VRP planner that communicates with it and feeds it with the missing geometric information. Both planners communicate to optimize the solutions found based on a score method, and the solutions are improved over time. The planning engine presented performs task assignment and scheduling to increase parallelism and cooperation in the domain of the ARCAS project, mixing geometric and symbolic reasoning.

A quantitative study for each of the possible configurations for the VRP planner was made, running multiple simulations. The decoupled approach was able to compute a valid assignment of structure parts for the aerial vehicles on each simulation. In addition, a correct scheduling for the different actions of the resulting plan for each of the vehicles was generated in all the simulations. The bi-directional communication between the geometric planner and the symbolic HTN planner allowed the optimization of the solutions found by the VRP planner. Different metaheuristic algorithms were tested and compared. Although these algorithms are not capable of finding the optimal solutions, they can compute reasonably good solutions in short times, showing their effectiveness.

6.1.2 Coupled Approach

The coupled approach consists on the development of a new HTN planner based on the idea of a guided heuristic search and inspired by the A* algorithm. The purpose of the guided heuristic search is to replace the depth-first algorithm present in most HTN planners and drive the search towards the best possible solution expanding first the more promising nodes. For domains that involve some kind of geometric reasoning, such as the VRP, a geometric projection of the decomposition nodes based on geometric computation, such as using Euclidean distances and so on, can be used, avoiding the needs of using an external geometric planner. We call this planner the SHOP* HTN planner.

Our new HTN planner is capable of computing the optimal solution if the heuristic function provided to the system is admissible. We have tested it in a single-vehicle domain example in the context of the VRP and against a brute force algorithm. The results show that when the admissibility criterion is satisfied, then the solution returned from our new planner is optimal.

Several tests have been additionally done to measure the performance of our new HTN planner against the original JSHOP2 implementation and the decoupled approach from Chapter 4, in the context of the AEROARMS project. In the tests it is shown that only SHOP* was capable of computing the optimal solution in all the data sets. The speed of our SHOP* planner was also compared with the fastest solver configuration for the Optaplanner-JSHOP2 connection, showing that SHOP* is the fastest algorithm.

6.2 Future Developments

One of the main drawbacks of the decoupled approach is the impossibility of ensuring the completeness of the system. In the decoupled approach, the search is mainly driven by OptaPlanner, which in turns calls JSHOP2 to get feedback from the symbolic level and tries to optimize the solution found at the geometric level. JSHOP2 is sound and complete, but that is not the case of OptaPlanner so our decoupled approach cannot ensure that it will always find a solution if one exists. Further research on this is needed. One possible solution is to let JSHOP2 drive the main search and call the geometric planner to get feedback from the geometric level. If OptaPlanner is not capable of finding a solution in the geometric level, we can let JSHOP2 continue and finish the search at the symbolic level, making some assumptions to bypass the missing geometric information or even trying to compute a solution to the geometrical problem when OptaPlanner fails. As the main search will be done by JSHOP2, which we know is sound and complete, we can then ensure the completeness of the system. Also, in future work, the goal is to execute the missions with the prototypes developed in the ARCAS project in order to find more realistic aspects to enrich the decoupled approach.

For the coupled approach, an optimization of the SHOP* main algorithm to decrease the memory consumption is needed, as in domains with a large combinatorial nature, the system that is executing the planner may run out of memory very fast. In addition, the performance of our new HTN planner has to be measured in multi-vehicle domains with the prototypes from the AEROARMS project, as having multiple agents usually involve higher computational loads. Finding an admissible heuristic that lets the planner compute the optimal plans is also more difficult. In addition to this, the quality of the solutions when using a non-admissible heuristic has to be studied and compared with other approaches.

List of Figures

List of Tables

List of Codes

Bibliography

[1] D. Alejo, J. A. Cobano, G. Heredia, and A. Ollero, *A reactive method for collision avoidance in industrial environments*, Journal of Intelligent & Robotic Systems **84** (2016), no. 1, 745–758.

[2] M. M. Arentoft, Y. Parrod, J. Stader, I. Stokes, and H. Vadon, *Optimum-AIV: a planning and scheduling system for spacecraft {AIV}*, Telematics and Informatics **8** (1991), no. 4, 239 – 252.

[3] J. Barry, L. P. Kaelbling, and T. Lozano-Perez, *A hierarchical approach to manipulation with diverse actions*, IEEE Internation Conference on Robotics and Automation (ICRA), 2013.

[4] L. Bianchi, M. Dorigo, L. M. Gambardella, and W. J. Gutjahr, *A survey on metaheuristics for stochastic combinatorial optimization*, Natural Computing **8** (2009), no. 2, 239–287.

[5] J. Bidot, L. Karlsson, F. Lagriffoul, and A. Saffiotti, *Geometric backtracking for combined task and motion planning in robotic systems*, vol. 247, 2017, Special Issue on AI and Robotics, pp. 229 – 265.

[6] A. L. Blum and M. L. Furst, *Fast planning through planning graph analysis*, Artificial Intelligence **90** (1995), no. 1, 1636–1642.

[7] C. Blum and A. Roli, *Metaheuristics in combinatorial optimization: overview and conceptual comparison*, ACM Computing Surveys **35** (2003), no. 3, 268–308.

[8] B. Bonet and H. Geffner, *Planning as heuristic search*, Artificial Intelligence **129** (2001), 5–33.

[9] F. Brizzi, L. Peppoloni, A. Graziano, E. D. Stefano, C. A. Avizzano, and E. Ruffaldi, *Effects of augmented reality on the performance of teleoperated industrial assembly tasks in a robotic embodiment*, IEEE Transactions on Human-Machine Systems **48** (2018), no. 2, 197–206.

[10] C. Burbridge and R. Dearden, *An approach to efficient planning for robotic manipulation tasks*, Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS), 2013.

[11] O. Caldiran, K. Haspalamutgil, A. Ok, C. Palaz, E. Erdem, and V. Patoglu, *Bridging the gap between high-level reasoning and low-level control*, Logic Programming and Non-monotonic Reasoning, Lecture Notes in Computer Science, vol. 5753, Springer, 2009, pp. 342–354.

[12] S. Cambon, R. Alami, and F. Gravot, *A hybrid approach to intricate motion, manipulation and task planning*, International Journal of Robotics Research (IJRR) **28** (2009), no. 1, 104–126.

[13] M. Cavazza and F. Charles, *Dialogue generation in character-based interactive storytelling*, Proceedings of the Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE), AAAI Press, 2005, pp. 21–26.

[14] J. Choi and E. Amir, *Combining planning and motion planning*, IEEE International Conference on Robotics and Automation (ICRA), 2009, pp. 238–244.

[15] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*, 3rd ed., The MIT Press, 2009.

[16] I. A. Şucan and L. Kavraki, *Mobile manipulation: encoding motion planning options using task motion multigraphs*, IEEE International Conference on Robotics and Automation (ICRA), 2011, pp. 5492–5498.

[17] I. A. Şucan and L. E. Kavraki, *Accounting for uncertainty in simultaneous task and motion planning using task motion multigraphs*, IEEE International Conference on Robotics and Automation (ICRA), 2012, pp. 4822–4828.

[18] G. B. Dantzig and J. H. Ramser, *The truck dispatching problem*, Management Science **6** (1959), no. 1, 80–91.

[19] M. de la Asunción, L. Castillo, J. Fdez-Olivares, O. García-Pérez, A. González, and F. Palao, *SIADEX: an interactive knowledge-based planner for decision support in forest fire fighting*, AI Communications **18** (2005), no. 4, 257–268.

[20] E. W. Dijkstra, *A note on two problems in connexion with graphs*, Numerische Mathematik (1959), 269–271.

[21] J. Dix, H. Muñoz-Avila, D. Nau, and L. Zhang, *IMPACTing SHOP: putting an AI planner into a multi-agent environment*, Annals of Mathematics and Artificial Intelligence **37** (2003), no. 4, 381–407.

[22] J. Dix and Y. Zhang, *Impact: a multi-agent framework with declarative semantics*, pp. 69–94, Springer US, 2005.

[23] C. Dornhege, P. Eyerich, T. Keller, M. Brenner, and B. Nebel, *Integrating task and motion planning using semantic attachments*, Bridging the Gap Between Task and Motion Planning, Papers from the AAAI Workshop, 2010.

[24] C. Dornhege, P. Eyerich, T. Keller, S. Trüg, M. Brenner, and B. Nebel, *Semantic attachments for domain-independent planning systems*, International Conference on Automated Planning and Scheduling (ICAPS), 2009.

[25] C. Dornhege, M. Gissler, M. Teschner, and B. Nebel, *Integrating symbolic and geometric planning for mobile manipulation*, International Workshop on Safety, Security and Rescue Robotics (SSRR), 2009.

[26] A. Dutta, *Self-assembly in heterogeneous multi-agent system using constrained matching algorithm*, 2016 IEEE/WIC/ACM International Conference on Web Intelligence (WI), Oct 2016, pp. 351–358.

[27] E. Erdem, K. Haspalamutgil, C. Palaz, V. Patoglu, and T. Uras, *Combining high-level causal reasoning with low-level geometric reasoning and motion planning for robotic manipulation*, IEEE International Conference on Robotics and Automation (ICRA), 2011, pp. 4575–4581.

[28] K. Erol, J. Hendler, and D. Nau, *HTN planning: complexity and expressivity*, Proceedings of the National Conference on Artificial Intelligence (AAAI), AAAI Press, 1994, pp. 1123–1128.

[29] K. Erol, J. A. Hendler, and D. Nau, *UMCP: a sound and complete procedure for hierarchical task-network planning*, Proceedings of the International Conference on AI Planning & Scheduling (AIPS), 1994, pp. 249–254.

[30] T. A. Estlin, S. A. Chien, and X. Wang, *An argument for a hybrid HTN/operator-based approach to planning*, Recent Advances in AI Planning, Lecture Notes in Computer Science, vol. 1348, Springer Berlin Heidelberg, 1997, pp. 182–194.

[31] A. Ferrein, C. Fritz, and G. Lakemeyer, *Using Golog for deliberation and team coordination in robotic soccer*, Künstliche Intelligenz **19** (2005), no. 1, 24–.

[32] J. Ferrer-Mestres, G. Francès, and H. Geffner, *Planning with state constraints and its application to combined task and motion planning*, PlanRob - Workshop on Planning and Robotics, International Conference on Automated Planning and Scheduling (ICAPS), 2015.

[33] R. E. Fikes and N. J. Nilsson, *STRIPS: a new approach to the application of theorem proving to problem solving*, Proceedings of the 2nd International Joint Conference on Artificial Intelligence (IJCAI), Morgan Kaufmann Publishers Inc., 1971, pp. 608–620.

[34] A. Filipescu, A. Filipescu, A. Voda, and E. Minca, *Hybrid modeling, balancing and control of a mechatronics line served by two mobile robots*, 2016 20th International Conference on System Theory, Control and Computing (ICSTCC), Oct 2016, pp. 234–239.

[35] C. Galindo, J. A. Fernández-Madrigal, J. González, and A. Saffiotti, *Robot task planning using semantic maps*, Journal of Robotics and Autonomous Systems (RAS) **56** (2008), no. 11, 955–966.

[36] C. R. Garret, T. Lozano-Perez, and L. P. Kaelbling, *Backward-forward search for manipulation planning*, IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2015.

[37] C. R. Garrett, T. Lozano-Pérez, and L. P. Kaelbling, *FFRob: An efficient heuristic for task and motion planning*, pp. 179–195, Springer International Publishing, 2015.

[38] A. Gaschler, I. Kessler, R. A. Petrick, and A. Knoll, *Extending the knowledge of volumes approach to robot task planning with efficient geometric predicates*, Proceedings of the IEEE International Conference on Robotics and Automation (ICRA), 2015.

[39] A. Gaschler, R. P. A. Petrick, M. Giuliani, M. Rickert, and A. Knoll, *KVP: a knowledge of volumes approach to robot task planning*, IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2013, pp. 202–208.

[40] M. Gendreau and J.Y. Potvin, *Handbook of metaheuristics*, 2nd ed., Springer Publishing Company, Incorporated, 2010.

[41] I. Georgievski and M. Aiello, *HTN planning: overview, comparison, and beyond*, Artificial Intelligence **222** (2015), no. 0, 124–156.

[42] S. Ghandi and E. Masehian, *Review and taxonomies of assembly and disassembly path planning problems and approaches*, Computer-Aided Design **67** (2015), no. C, 58–86.

[43] D. E. Goldberg, *Genetic algorithms in search, optimization and machine learning*, 1st ed., Addison-Wesley Longman Publishing, 1989.

[44] F. Gravot, S. Cambon, and R. Alami, *aSyMov: a planner that deals with intricate symbolic and geometric problems*, pp. 100–110, Springer Berlin Heidelberg, 2005.

[45] J. Gu, H. Wang, W. Chen, and R. Wu, *Monocular visual object-localization using natural corners for assembly tasks*, 2016 IEEE International Conference on Robotics and Biomimetics (ROBIO), Dec 2016, pp. 1383–1388.

[46] J. Guitton and J. Farges, *Towards a hybridization of task and motion planning for robotic architectures*, International workshop on Hybrid Control of Autonomous Systems (HYCAS), 2009, pp. 21–24.

[47] G. Gutin and A. P. Punnen, *The traveling salesman problem and its variations*, Combinatorial optimization, Kluwer Academic, 2002.

[48] K. Zita Haigh and M. M. Veloso, *Interleaving planning and robot execution for asynchronous user requests*, Autonomous Robots **5** (1998), no. 1, 79–95.

[49] P. E. Hart, N. J. Nilsson, and B. Raphael, *A formal basis for the heuristic determination of minimum cost paths*, IEEE Transactions on Systems Science and Cybernetics **4** (1968), no. 2, 100–107.

[50] ______, *Correction to "a formal basis for the heuristic determination of minimum cost paths"*, SIGART Bull. (1972), no. 37, 28–29.

[51] K. Hauser, *Task planning with continuous actions and non-deterministic motion planning queries*, 2010.

[52] K. Hauser and J. C. Latombe, *Integrating task and PRM motion planning: dealing with many infeasible motion planning queries*, ICAPS Workshop on Bridging the Gap between Task and Motion Planning, 2009.

[53] K. Hauser, V. Ng-Thow-Hing, and H. Gonzalez-Baños, *Multi-modal motion planning for a humanoid robot manipulation task*, pp. 307–317, Springer Berlin Heidelberg, 2011.

[54] M. Helmert, *The Fast Downward planning system*, Journal of Artificial Intelligence Research (JAIR) **26** (2006), no. 1, 191–246.

[55] A. Hertle, C. Dornhege, T. Keller, and B. Nebel, *Planning with semantic attachments: an object-oriented view*, Proceedings of the European Conference on Artificial Intelligence, 2012, pp. 402–407.

[56] H. Hoang, S. Lee-Urban, and H. Muñoz-Avila, *Hierarchical plan representations for encoding strategic game AI*, Artificial Intelligence and Interactive Digital Entertainment Conference, 2005, pp. 63–68.

[57] J. Hoffmann and B. Nebel, *The FF planning system: fast plan generation through heuristic search*, Journal of Artificial Intelligence Research (JAIR) **14** (2001), no. 1, 253–302.

[58] C. Hogg, H. Muñoz-Avila, and U. Kuter, *Learning hierarchical task models from input traces*, Computational Intelligence (2014).

[59] J. JianJu and G. YunJian, *A enhanced self-assembly morphology distributed control algorithm of swarm robots*, 2017 2nd International Conference on Robotics and Automation Engineering (ICRAE), Dec 2017, pp. 57–62.

[60] P. Jiménez, *Survey on assembly sequencing: a combinatorial and geometrical perspective*, Journal of Intelligent Manufacturing **24** (2013), no. 2, 235–250.

[61] L. P. Kaelbling and T. Lozano-Perez, *Unifying perception, estimation and action for mobile manipulation via belief space planning*, IEEE International Conference on Robotics and Automation (ICRA), 2012, pp. 2952–2959.

[62] L. P. Kaelbling and T. S. Lozano-Perez, *Hierarchical task and motion planning in the now*, IEEE International Conference on Robotics and Automation (ICRA), 2011, pp. 1470–1477.

[63] L. P. Kaelbling and T. Lozano-Pérez, *Integrated task and motion planning in belief space*, International Journal of Robotics Research (IJRR) **32** (2013), no. 9-10, 1194–1227.

[64] L. Karlsson, J. Bidot, F. Lagriffoul, A. Saffiotti, U. Hillenbrand, and F. Schmidt, *Combining task and path planning for a humanoid two-arm robotic system*, Proceedings of TAMPRA: Combining Task and Motion Planning for Real-World Applications (ICAPS workshop), 2012, pp. 13–20.

[65] L. Kavraki, J. C. Latombe, and R. H. Wilson, *On the complexity of assembly partitioning*, Information Processing Letters **48** (1993), 229–235.

[66] L. Kavraki, P. Svestka, J. C. Latombe, and M. Overmars, *Probabilistic roadmaps for path planning in high-dimensional configuration spaces*, IEEE International Conference on Robotics and Automation (ICRA), 1996, pp. 566–580.

[67] R. A. Knepper, T. Layton, J. Romanishin, and D. Rus, *Ikeabot: an autonomous multi-robot coordinated furniture assembly system.*, IEEE International Conference on Robotics and Automation (ICRA), 2013, pp. 855–862.

[68] J. B. Kruskal, *On the shortest spanning subtree of a graph and the traveling salesman problem*, Proceedings of the American Mathematical Society **7** (1956), no. 1, 48–50.

[69] F. Lagriffoul, *Delegating geometric reasoning to the task planner*, Workshop on Planning and Robotics, International Conference on Automated Planning and Scheduling (ICAPS), 2013, pp. 54–59.

[70] F. Lagriffoul, D. Dimitrov, J. Bidot, A. Saffiotti, and L. Karlsson, *Efficiently combining task and motion planning using geometric constraints*, International Journal of Robotics Research (IJRR) **33** (2014), no. 14, 1726–1747.

[71] R. Lallement, *Symbolic and geometric planning for teams of robots and humans*, Theses, INSA de Toulouse, 2016.

[72] J.P. Laumond and R. Alami, *A geometrical approach to planning manipulation tasks in robotics*, First Canadian Conference on Computational Geometry, 1989.

[73] S. M. LaValle, *Planning algorithms*, Cambridge University Press, 2006.

[74] D. Leidner, A. Dietrich, F. Schmidt, C. Borst, and A. Albu-Schaffer, *Object-centered hybrid reasoning for whole-body mobile manipulation*, IEEE International Conference on Robotics and Automation (ICRA), 2014, pp. 1828–1835.

[75] S. Li and H. Gu, *Acoustic contacting detection in robotic accurate assembly*, 2017 IEEE International Conference on Robotics and Biomimetics (ROBIO), Dec 2017, pp. 1829–1832.

[76] Q. Lindsey, D. Mellinger, and V. Kumar, *Construction of cubic structures with quadrotor teams*, Robotics: Science and Systems, MITP, 2012.

[77] T. Lozano-Perez and L. P. Kaelbling, *A constraint-based method for solving sequential manipulation planning problems*, IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2014, pp. 3684–3691.

[78] I.M. Mahmoud, L. Lianchao, D. Wloka, and M. Z. Ali, *Believable npcs in serious games: HTN planning approach based on visual perception*, IEEE Conference on Computational Intelligence and Games (CIG), 2014, pp. 1–8.

[79] H. Marino, M. Ferrati, A. Settimi, C. Rosales, and M. Gabiccini, *On the problem of moving objects with autonomous robots: A unifying high-level planning approach*, IEEE Robotics and Automation Letters **1** (2016), no. 1, 469–476.

[80] I. Maza, J. Muñoz-Morera, F. Caballero, E. Casado, V. Perez-Villar, and A. Ollero, *Architecture and tools for the generation of flight intent from mission intent for a fleet of unmanned aerial systems*, International Conference on Unmanned Aircraft Systems (ICUAS), IEEE, 2014, pp. 9–19.

[81] J. McCarthy and P. J. Hayes, *Readings in nonmonotonic reasoning*, Morgan Kaufmann Publishers Inc., 1987, pp. 26–45.

[82] D. Mcdermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins, *PDDL - The Planning Domain Definition Language*, Tech. Report TR-98-003, Yale Center for Computational Vision and Control, 1998.

[83] A. Menif, E. Jacopin, and T. Cazenave, *SHPE: HTN planning for video games*, Computer Games, Communications in Computer and Information Science, vol. 504, Springer International Publishing, 2014, pp. 119–132.

[84] Pierre Merriaux, Yohan Dupuis, Rémi Boutteau, Pascal Vasseur, and Xavier Savatier, *A study of Vicon system positioning performance*, Sensors **17** (2017), no. 7.

[85] S. W. Mitchell, *A hybrid architecture for real-time mixed-initiative planning and control*, Proceedings of the National Conference on Artificial Intelligence and Conference on Innovative Applications of Artificial Intelligence, AAAI Press, 1997, pp. 1032–1037.

[86] H. Muñoz-Avila, D. W. Aha, D. Nau, R. Weber, L. Breslow, and F. Yamal, *SiN: integrating case-based reasoning with task decomposition*, Proc of the International Joint Conference on Artificial Intelligence (IJCAI), 2001.

[87] H. Muñoz-Avila, D. C. Mcfarlane, D. W. Aha, L. Breslow, J. A. Ballas, and D. Nau, *Using guidelines to constrain interactive case-based HTN planning*, International Conference on Case-Based Reasoning and Development (ICCBR), 1999.

[88] J. Muñoz-Morera, F. Alarcon, I. Maza, and A. Ollero, *Combining a hierarchical task network planner with a constraint satisfaction solver for assembly operations involving routing problems in a multi-robot context*, International Journal of Advanced Robotic Systems (IJARS) **15** (2018), no. 3, 1–13.

[89] J. Muñoz-Morera, I. Maza, F. Caballero, and A. Ollero, *Architecture for the automatic generation of plans for multiple UAS from a generic mission description*, Journal of Intelligent & Robotic Systems **84** (2016), no. 1, 493–509.

[90] J. Muñoz-Morera, I. Maza, C. J. Fernandez-Agüera, F. Caballero, and A. Ollero, *Assembly planning for the construction of structures with multiple UAS equipped with robotic arms*, International Conference on Unmanned Aircraft Systems (ICUAS), IEEE, 2015, pp. 1049–1058.

[91] J. Muñoz-Morera, I. Maza, C. J. Fernandez-Agüera, and A. Ollero, *Task allocation for teams of aerial robots equipped with manipulators in assembly operations*, Advances in Intelligent Systems and Computing, vol. 417, pp. 585–596, Springer International Publishing, 2016.

[92] D. Nau, *Current trends in automated planning*, Artificial Intelligence Magazine **28** (2007), 43 – 43.

[93] D. Nau, Y. Cao, A. Lotem, and H. Muñoz-Avila, *SHOP: Simple hierarchical ordered planner*, Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), Morgan Kaufmann Publishers Inc., 1999, pp. 968–973.

[94] D. Nau, O. Ilghami, U. Kuter, J. W. Murdock, D. Wu, and F. Yaman, *SHOP2: an HTN planning system*, Journal of Artificial Intelligence Research (JAIR) **20** (2003), 379–404.

[95] D. Nau, S. J. J. Smith, and K. Erol, *Control strategies in HTN planning: theory versus practice*, Proceedings of the National Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence, American Association for Artificial Intelligence, 1998, pp. 1127–1133.

[96] N. Nau, M. Ghallab, and P. Traverso, *Automated planning: theory & practice*, Morgan Kaufmann Publishers Inc., 2004.

[97] S. Nedunuri, S. Prabhu, M. Moll, S. Chaudhuri, and L. E. Kavraki, *SMT-based synthesis of integrated task and motion plans for mobile manipulation*, IEEE International Conference on Robotics and Automation (ICRA), 2014, pp. 655–662.

[98] N. J. Nilsson, *A mobile automaton: an application of artificial intelligence techniques*, Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), 1969, pp. 509–520.

[99] ______, *Shakey the robot*, Tech. Report 323, AI Center, SRI International, 1984.

[100] Red Hat open source community, *OptaPlanner*, 2018.

[101] E. Pednault, *ADL: exploring the middle ground between STRIPS and the situation calculus*, Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning, 1989, pp. 324–332.

[102] E. Plaku, *Planning robot motions to satisfy linear temporal logic, geometric, and differential constraints*, ICAPS Workshop on Combining Task and Motion Planning for Real-World Applications, 2012, pp. 21–28.

[103] E. Plaku and G. D. Hager, *Sampling-based motion and symbolic action planning with geometric and differential constraints*, IEEE International Conference on Robotics and Automation (ICRA), 2010, pp. 5002–5008.

[104] R. Ragel, I. Maza, F. Caballero, and A. Ollero, *Comparison of motion planning techniques for a multi-rotor UAS equipped with a multi-joint manipulator arm*, 2015 Workshop on Research, Education and Development of Unmanned Aerial Systems (RED-UAS), Nov 2015, pp. 133–141.

[105] R. Reiter, *Artificial intelligence and mathematical theory of computation*, Academic Press Professional Inc., 1991, pp. 359–380.

[106] F. Ruggiero, M. A. Trujillo, R. Cano, H. Ascorbe, A. Viguria, C. Perez, V. Lippiello, A. Ollero, and B. Siciliano, *A multilayer control for multirotor UAVs equipped with a servo robot arm*, 2015 IEEE International Conference on Robotics and Automation (ICRA), May 2015, pp. 4014–4020.

[107] S. Russell and P. Norvig, *Artificial intelligence : a modern approach*, 3 ed., Prentice Hall, 2010.

[108] E. D. Sacerdoti, *The non-linear nature of plans*, Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), Morgan Kaufmann Publishers Inc., 1975, pp. 206–214.

[109] V. Shivashankar, K. Kaipa, D. Nau, and K. S. Gupta, *Towards integrating hierarchical goal networks and motion planners to support planning for human-robot teams*, (2014).

[110] L. D. Silva, R. Lallement, and R. Alami, *The HATP hierarchical planner: formalisation and an initial study of its usability and practicality*, IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2015, pp. 6465–6472.

[111] S. J. J. Smith, K. Hebbar, D. Nau, and I. Minis, *Integrating electrical and mechanical design and process planning*, Knowledge intensive CAD, IFIP — The International Federation for Information Processing, Springer US, 1997, pp. 269–288.

[112] D. Soemers and M. Winands, *Hierarchical task network plan reuse for video games*, IEEE International Conference on Computational Intelligence and Games (CIG), 2016, pp. 1–8.

[113] S. Sohrabi, J. A. Baier, and S. A. McIlraith, *HTN planning with preferences*, Proceedings of the International Joint Conference on Artifical Intelligence (IJCAI), Morgan Kaufmann Publishers Inc., 2009, pp. 1790–1797.

[114] S. Srivastava, E. Fang, L. Riano, R. Chitnis, S. Russell, and P. Abbeel, *Combined task and motion planning through an extensible planner-independent interface layer*, IEEE International Conference on Robotics and Automation (ICRA), 2014, pp. 639–646.

[115] S. Srivastava, L. Riano, S. Russell, and P. Abbeel, *Using classical planners for tasks with continuous operators in robotics*, ICAPS Workshop on Planning and Robotics, 2013.

[116] E. Talbi, *Metaheuristics: from design to implementation*, Wiley Publishing, 2009.

[117] A. Tate, *Generating project networks*, Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), Morgan Kaufmann Publishers Inc., 1977, pp. 888–893.

[118] A. Tate, B. Drabble, and R. Kirby, *O-Plan2: an open architecture for command, planning and control*, Intelligent Scheduling, Morgan Kaufmann Publishers Inc., 1994, pp. 213–239.

[119] A. Dayal Udai and S. K. Saha, *A framework for CAD-based offline depth-map preparation for automated assembly tasks*, 2016 International Conference on Robotics and Automation for Humanitarian Applications (RAHA), Dec 2016, pp. 1–6.

[120] V. V. Unhelkar, P. A. Lasota, Q. Tyroller, R. D. Buhai, L. Marceau, B. Deml, and J. A. Shah, *Human-aware robotic assistant for collaborative assembly: Integrating human motion prediction with planning in time*, IEEE Robotics and Automation Letters **PP** (2018), no. 99, 1–1.

[121] D. E. Wilkins, *Practical planning: extending the classical AI planning paradigm*, Morgan Kaufmann Publishers Inc., 1988.

[122] ______, *Using the SIPE-2 planning system: a manual for version 4.17*, (1997).

[123] J. Wolfe, B. Marthi, and S. Russell, *Combined task and motion planning for mobile manipulation*, Interational Conference on Automated Planning and Scheduling (ICAPS), 2010.

[124] F. Zhao, H. Gu, C. Li, and C. Chen, *Accuracy analysis for robotized assembly system*, 2017 IEEE International Conference on Robotics and Biomimetics (ROBIO), Dec 2017, pp. 1850–1855.

[125] S. Zickler and M. Veloso, *Efficient physics-based planning: sampling search via non-deterministic tactics and skills*, Proceedings of the International Conference on Autonomous Agents and Multiagent Systems, 2009, pp. 27–33.