Context-Aware Multi-Stage Route Planning

Master's Thesis

Jeroen van Belle

Context-Aware Multi-Stage Route Planning

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Jeroen van Belle
born in Vlissingen, the Netherlands

Algorithmics Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

© 2010 Jeroen van Belle

Cover picture:
An airplane is de-iced before it can take off; see chapter 1.

Context-Aware Multi-Stage Route Planning

Author: Jeroen van Belle
Student id: 9047397
Email: `j.p.vanbelle@student.tudelft.nl`

Abstract

One way to prevent collisions and deadlocks between agents on infrastructures with capacity constraints is to use *context-aware* route planning algorithms. In context-aware route planning, there is a set of agents each planning a conflict-free route from a start location to a destination location on a common infrastructure. If a sequential approach is used for route planning, finding an optimal conflict-free route for a single agent can be done in polynomial time by allowing agents to place reservations on infrastructure resources for the intended periods of occupation.

In the *multi*-stage variant of the context-aware route planning problem, there is a sequence of locations to be visited by an agent instead of just a start and destination location as in *single*-stage route planning. A straightforward approach to the multi-stage route planning problem would be concatenating subsequent routes found by a single-stage algorithm. However, this solution is *incomplete* and *sub-optimal* in case of context-aware route planning. Therefore, we developed three multi-stage route planning algorithms that always return an optimal route for a single agent, given a set of reservations made by previous agents. The results of our experiments show that one of the multi-stage algorithms is suitable for use in practice, since the cpu-time needed to make a route is at most a few tenths of a second.

Thesis Committee:

Chair:	Prof. Dr. C. Witteveen, Faculty EEMCS, TU Delft
University supervisor:	Dr. Ir. A.W. ter Mors, Faculty EEMCS, TU Delft
Committee Member:	Dr. K. van der Meer, Faculty EEMCS, TU Delft
Committee Member:	Dr. M.M. de Weerdt, Faculty EEMCS, TU Delft

Contents

Contents iii

List of Figures v

1 Introduction 1
- 1.1 Problem Statement . . . 2
- 1.2 Overview and Contributions . . . 3

2 Context-Aware Single-Stage Route Planning 5
- 2.1 Infrastructure Model . . . 5
- 2.2 Agent Plans . . . 6
  - 2.2.1 Additional constraints . . . 7
- 2.3 Reservations and Free Time Windows . . . 10
  - 2.3.1 Additional constraints and free time windows . . . 12
- 2.4 Single-Stage Route Planning . . . 14
  - 2.4.1 The A* algorithm . . . 14
  - 2.4.2 A single-stage route planning algorithm . . . 19
  - 2.4.3 Additional constraints and single-stage route planning . . . 23

3 The Context-Aware Multi-Stage Route Planning Problem 25
- 3.1 Linear Concatenation Approach . . . 25
  - 3.1.1 Complexity . . . 26
- 3.2 Incompleteness and Sub-Optimality of the Linear Concatenation Approach . . . 27
- 3.3 Concluding Remarks . . . 27

4 Context-Aware Multi-Stage Route Planning Algorithms 31
- 4.1 Breadth-First Concatenation Approach . . . 31
  - 4.1.1 Correctness . . . 36
  - 4.1.2 Complexity . . . 36
  - 4.1.3 Additional constraints and breadth-first concatenation . . . 36

4.2 A* Concatenation Approach 37
4.2.1 Correctness 40
4.2.2 Complexity 41
4.3 A* Multi-Layer Approach 41
4.3.1 Correctness 47
4.3.2 Complexity 50
4.4 Concluding Remarks 50

5 Experiments **53**
5.1 Test Procedure 53
5.2 Results 54
5.2.1 The linear concatenation algorithm and its plan quality . . . 54
5.2.2 CPU-time analysis of the multi-stage route planning algorithms 57
5.3 Concluding Remarks 61

6 Conclusions and Future Work **63**
6.1 Conclusions 63
6.2 Future work 64

Bibliography **67**

List of Figures

1.1 Two agents travel towards each other. 1

2.1 An infrastructure graph G, and the corresponding resource graph G_R. . 6
2.2 Agent A_1 and agent A_2 exchange resources at time 5. 8
2.3 Agent A_1 is leading A_2. 10
2.4 Resource load for resource r with capacity 3. 11
2.5 Agent A_1 will enter lane resource r from intersection v, and must therefore exit r via intersection w. 13
2.6 A graph with five nodes, and the edges have costs as depicted. 18
2.7 Free time window f' is reachable from free time window f, but not from the earliest possible exit time $g(f)$. 20

3.1 Infrastructure graph and free time window graph. 28

4.1 Breadth-first concatenation approach with four stages. The long-dashed arrow (π_8) represents a plan that will not be found by the breadth-first concatenation algorithm. The dashed arrows (π_5 and π_9) represent plans that will be found, but are not considered for further expansion. 32
4.2 A visiting sequence of size three results in a free time window graph with three layers. 43

5.1 Type of plans found by the linear concatenation algorithm with visiting sequence size 4 for an increasing number of agents on the random graph. 55
5.2 Type of plans found by the linear concatenation algorithm with visiting sequence size 6 for an increasing number of agents on the random graph. 55
5.3 Type of plans found by the linear concatenation algorithm with visiting sequence size 8 for an increasing number of agents on the random graph. 56
5.4 Type of plans found by the linear concatenation algorithm with visiting sequence size 4 for an increasing number of agents on the Schiphol infrastructure. 56

5.5 Average CPU-time for the single-stage route planning algorithm for an increasing number of agents on the random graph. 58
5.6 Average CPU-time for the breadth-first concatenation algorithm for several visiting sequence sizes for an increasing number of agents on the random graph. 59
5.7 Average CPU-time for the A* concatenation algorithm for several visiting sequence sizes for an increasing number of agents on the random graph. 59
5.8 Average CPU-time for the A* multi-layer algorithm for several visiting sequence sizes for an increasing number of agents on the random graph. 60
5.9 Average CPU-time for the multi-stage algorithms with visiting sequence size 6 for an increasing number of agents on the random graph. 60
5.10 Average CPU-time for the multi-stage algorithms with visiting sequence size 4 for an increasing number of agents on the Schiphol infrastructure. 62
5.11 Average CPU-time for the A* multi-layer algorithm with visiting sequence size 4 for an increasing number of agents on the Schiphol infrastructure. 62

Chapter 1

Introduction

When a set of agents have to travel on a common infrastructure consisting of resources with a limited capacity, collisions and deadlocks between the agents can occur. An example of a collision is when two agents travel on a resource in opposite directions (see figure 1.1). If there is not enough room for the agents to pass each other, they will run into each other. However, if the agents are for example equipped with sensors, then the example results in a deadlock situation. The agents are forced to stop in front of each other, and no further transport is possible.

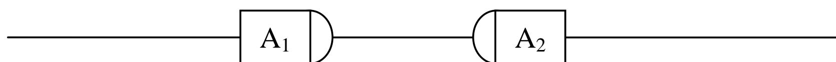


Figure 1.1: Two agents travel towards each other.

One way to prevent collisions and deadlocks on infrastructures with capacity constraints is to use *context-aware* route planning algorithms. In context-aware route planning [8, 14], there is a set of agents each planning a conflict-free *route* from a start location to a destination location on a common infrastructure. A route, also called a *plan*, consists of a *path* and a corresponding *schedule*. A path is a sequence of resources, which have to be visited by the agent, and a schedule gives arrival and departure times for the resources of the path. Context-awareness means that an agent that is computing a plan has to take into account the plans of other agents that also want to use the infrastructure. As such, context-aware route planning is an attempt to resolve possible conflicts between agents before the execution of the agent plans. Other approaches to handle the problem of collisions and deadlocks are described in [15].

An example of an application where context-aware route planning plays a role is an automated guided vehicle system (AGVS). An AGVS is a driverless vehicle-based transportation system. Traditionally, they are used in manufacturing plants [17] and warehouses [10, 16] to transport materials betweens locations of the facility. Currently, automated guided vehicles (AGVs) are also used for transportation tasks in

other areas, such as container terminals (e.g. in Singapore, Rotterdam, or Hamburg), where AGVs carry containers to and from ships [9, 11].

Another application domain is taxi route planning for airplanes at airports [14]. In taxi route planning, conflict-free routes have to be found for airplanes on the infrastructure of the airport. This infrastructure consists of runways, taxiways, aprons, gates, etc. For example, when an airplane is about to land to pick up or drop off some passengers, a route needs to be found from a runway, where the airplane can land, to a gate, where passengers can leave and enter the airplane, and then from the gate to a runway to take off. It is obviously important that the airplane never comes into contact with other airplanes on his route.

1.1 Problem Statement

In *single*-stage route planning an agent has a start location and a destination location. A route has to be found for this agent in such a way that this agent doesn't interfere with other agents on the infrastructure. In *multi*-stage route planning there is a sequence of locations to be visited by an agent instead of just a start and destination location as in single-stage route planning. The multi-stage route planning problem can occur in all of the aforementioned problem domains, since agents frequently have more than one task to perform. At airports, for example, wintry conditions sometimes require snow and ice to be removed from wings and fuselage shortly before take off. This means that an airplane cannot taxi directly from the gate to the runway, because it must first make a stop at a de-icing station, which may be located elsewhere at the airport. In manufacturing, an automated guided vehicle (AGV) may have a sequence of transportation tasks to perform, and it must also make the occasional trip to the battery charging station in between orders.

As far as we know, the multi-stage route planning problem hasn't been studied previously in the literature. The reason might be that often, the shortest route along a sequence of resources is simply the concatenation of shortest routes between successive resources. However, in context-aware route planning, this concatenation approach might return a non-optimal route, or even no route at all (even when a route does exist). Therefore, finding a complete and optimal algorithm for the multi-stage route planning problem becomes the subject of our research.

Note that a related, but more general, problem that has been studied extensively is the Traveling Salesperson Problem (TSP) [2], in which there is a *set* (i.e., unordered) of locations that must be visited with minimum total cost. However, the generality afforded by the TSP is not required in our intended applications; for instance, in the airport de-icing scenario, an agent need not consider route plans where the airplane takes off prior to de-icing.

1.2 Overview and Contributions

Having briefly introduced the multi-stage route planning problem, the remainder of this introduction presents a brief overview of its treatment, and a description of our most important contributions.

The first contribution of this master's thesis is the identification of the multi-stage route planning problem. When concatenation of shortest routes between successive resources is sufficient to find a multi-stage route, the multi-stage problem is trivial. However, we found out that this concatenation approach is incomplete and non-optimal in case of context-aware route planning. The second and main contribution of this thesis is the formulation of three complete and optimal multi-stage route planning algorithms, and the analysis of these algorithms. The third contribution is an empirical evaluation of the multi-stage route planning algorithms.

In chapter 2 we will describe a framework for context-aware route planning, and we will explain a single-stage route planning algorithm. The algorithm is based on the concept of *free time windows*: time intervals in which an agent can use a resource without introducing conflicts with other agents. In chapter 3 we will explain the multi-stage route planning problem. We will demonstrate why the concatenation of plans found by the single-stage route planning algorithm is an incomplete and sub-optimal approach to multi-stage route planning. During our research, we have developed three complete and optimal multi-stage algorithms, which we will present in chapter 4. For each algorithm a specification is given together with a proof of correctness and an analysis of its complexity. In chapter 5 we tested the need for a complete and optimal multi-stage route planning algorithm by empirically investigating the failure rate and plan quality of the trivial concatenation approach from chapter 3. Furthermore, we examined the execution times of the complete and optimal multi-stage algorithms from chapter 4. Finally, chapter 6 will give a summary and a conclusion. Furthermore, some ideas for future work will be discussed.

Chapter 2

Context-Aware Single-Stage Route Planning

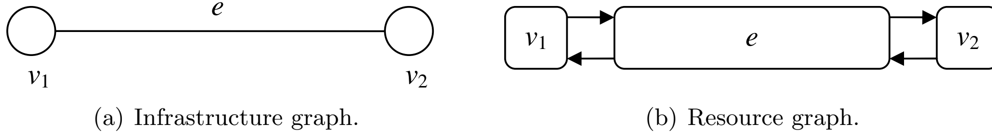
In general, context-aware route planning is about finding a collectively optimal set of conflict-free plans for agents on a common infrastructure. In [13] is proved that this problem is NP-hard. However, if a sequential approach is used for route planning, which means that agents make plans one after the other, finding an optimal conflict-free route plan for a single agent can be done in polynomial time. Therefore, we will focus on this sequential approach.

In this chapter a framework for context-aware route planning is presented, and a single-stage route planning algorithm is discussed. The first section describes the model that is used for the infrastructure. In section 2.2 a definition is given for an agent plan. Furthermore, we look at the cost of an agent plan, and several constraints for agent plans are discussed. Section 2.3 describes the framework that models the solution method that is based on placing reservations on resources. The idea for the route planning algorithm is that an agent makes a plan, and then it places reservations on the resources for the intended periods of occupation. During planning, an agent is only allowed to use time intervals that do not conflict with the set of existing reservations on the resources. These time intervals are called *free time windows*. The set of free time windows and the reachability between the free time windows form a graph structure: *the free time window graph*. In the last section a single-stage route planning algorithm is explained. This algorithm uses the free time window graph to find a shortest-time, conflict-free route plan for an agent.

2.1 Infrastructure Model

An *infrastructure* is a graph $G = (V, E)$, where V is a set of vertices representing *intersections* and *locations*, and $E \subseteq V \times V$ is a set of edges. These edges can be both directed or undirected, and we will use the term *lane* to refer to both types of edge. Because collisions between agents should be avoided on intersections, locations, and lanes, and not only lanes have non-zero travel time, we treat all elements of the

infrastructure in the same way. Therefore, we transform the infrastructure graph G to a *resource graph* $G_R = (R, E_R)$. The set of vertices of the resource graph is the set of (infrastructure) *resources*: $R = V \cup E$. We derive the set of arcs E_R in the following manner: for each undirected edge $e = \{v, w\} \in E$, the set E_R contains the pairs (v, e), (e, w), (w, e), and (e, v); for each arc (directed edge) $(v, w) \in E$, E_R contains the pairs (v, e) and (e, w). See figure 2.1 for a translation of a simple infrastructure to a resource graph. The set of arcs E_R can be interpreted as a successor relation: if $(r, r') \in E_R$, then an agent can go directly from resource r to resource r'.



(a) Infrastructure graph. (b) Resource graph.

Figure 2.1: An infrastructure graph G, and the corresponding resource graph G_R.

We associate two attributes with every resource $r \in R$: a capacity and a minimum travel time. The function $tt : R \rightarrow T^+$ gives the minimum travel time of a resource. The set T is the set of possible time points, and it consists of the set of non-negative real numbers; the set T^+ consists of only positive real numbers. The capacity is a function $cap : R \rightarrow \mathbb{N}^+$ that specifies the maximum number of agents that can simultaneously occupy a resource. We will assume that all intersections and locations (i.e., resources in V) have a capacity of one.

2.2 Agent Plans

We define a set $\mathcal{A}$ of agents that can traverse the infrastructure. Each agent $A_i \in \mathcal{A}$ has one start location $r \in R$ and one destination location $r' \in R$. In order to determine a set of conflict-free plans, agents have to specify exactly for each time point which resource they will occupy.

Definition 2.2.1 (Agent Plan)**.** *Given an agent $A_i \in \mathcal{A}$, a start location $r \in R$, a destination location $r' \in R$, and a start time $t \in T$, a plan for A_i is a sequence $\pi = (\langle r_1, \tau_1 = [t_1, t'_1)\rangle, \ldots, \langle r_n, \tau_n = [t_n, t'_n)\rangle)$ of n* plan steps *such that $r_1 = r$, $r_n = r'$, $t_1 \geq t$, and $\forall j \in \{1, \ldots, n\}$:*

1. *interval τ_j meets interval τ_{j+1} $(j < n)$;*
2. *$|\tau_j| \geq tt(r_j)$;*
3. *$(r_j, r_{j+1}) \in E_R$ $(j < n)$.*

The first constraint in the above definition makes use of Allen's interval algebra [1][1], and states that the exit time out of the jth resource in the plan must be equal to the entry time into resource $j+1$. The second constraint requires that the agent's occupation time of a resource is at least sufficient to traverse the resource in the minimum travel time. The third constraint states that if two resources follow each other in the agent's plan, then they must be adjacent in the resource graph.

The single objective in route planning that we will consider is to minimize completion time. Hence, we define the cost of an agent plan as the end time of the plan. The cost of a set of agent plans is simply the maximum of the individual plan costs. This is called the *makespan* of the plans, i.e., the time at which all agents have completed their plan.

Definition 2.2.2 (Plan Cost)**.** *Given an agent plan* $\pi = (\langle r_1, \tau_1 = [t_1, t'_1)\rangle, \ldots, \langle r_n, \tau_n = [t_n, t'_n)\rangle)$*, the cost of* π *is defined as* $c(\pi) = t'_n$*. The cost of a set* Π *of agent plans is defined as* $c(\Pi) = \max_{\pi_i \in \Pi}(c(\pi_i))$*.*

As described above, an agent plan defines for each point in time which resource the agent occupies. From a set of agent plans, we can now derive the number of agents that occupy a particular resource at a certain point in time.

Definition 2.2.3 (Resource Load)**.** *Given a set* Π *of agent plans, the* resource load λ *is a function* $\lambda : R \times T \rightarrow \mathbb{N}$ *that returns the number of agents occupying a resource* r *at time point* t*:*

$$\lambda(r,t) = |\{\langle r, \tau\rangle \sqsubseteq \pi \mid \pi \in \Pi \wedge t \in \tau\}|$$

In context-aware route planning it is crucial that the resource load never exceeds the capacity of a resource, because agents need to find plans that do not interfere with each other. In other words, to avoid collisions we need to be sure that there are never more agents on a resource than its capacity allows. This is the resource load constraint:

$$\forall r \forall t : \lambda(r,t) \leq cap(r) \tag{2.1}$$

In many realistic application domains, there are additional constraints that a set of agent route plans must satisfy. We will now discuss three constraints that may be considered in addition to the basic resource load constraint.

2.2.1 Additional constraints

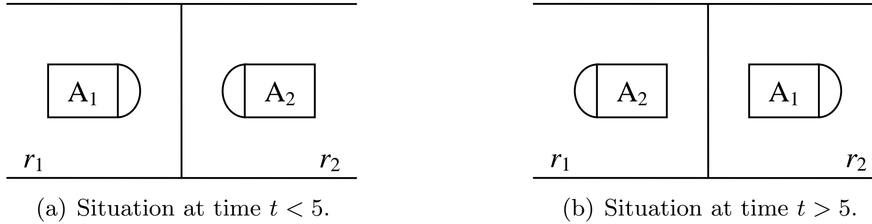
The first additional constraint concerns how agents go from one resource to another. This constraint, which was first considered by Hatzack and Nebel [7], is necessary in some application domains to rule out head-on conflicts [3]. A head-on conflict

[1] We make use of the *meets* predicate, which means that the end of one interval is equal to the start of the second, and the *precedes* predicate, which means that the end of one interval is earlier than the start of the second.

between two agents will occur if they exchange resources at the same point in time, and there is not enough space at the intersection of the resources involved. The second constraint specifies that a bidirectional lane resource may only be used in one direction at the same time, to rule out the possibility of oncoming traffic [8, 13]. This way no head-on conflicts can occur on lane resources. This constraint can be needed if there is not enough room for the agents to pass each other on the resource. The third constraint prevents agents from overtaking each other on a lane resource, so no catching-up conflicts [3] can occur on lane resources [8, 13]. The last two constraints are commonly relevant when lane resources are long and narrow, and there is no room for two agents to drive side-by-side.

Simultaneous resource exchanges

The first constraint that we will discuss is a constraint to prevent simultaneous resource exchanges between agents. When there is not enough space at the intersection of two resources, a head-on conflict between two agents can occur if they exchange resources at the same point in time. Consider figure 2.2 for an example situation. We have two resources r_1 and r_2 of unit capacity, and $E_R = \{(r_1, r_2), (r_2, r_1)\}$, i.e., travel is possible in both directions. Furthermore, imagine we have two agents A_1 and A_2, with the following respective plans: $\pi_1 = (\langle r_1, [0, 5)\rangle, \langle r_2, [5, \infty)\rangle)$ and $\pi_2 = (\langle r_2, [0, 5)\rangle, \langle r_1, [5, \infty)\rangle)$. Note that the union of these plans does not at any time exceed the capacity of the resources, and these plans therefore seem conflict-free.



(a) Situation at time $t < 5$.　(b) Situation at time $t > 5$.

Figure 2.2: Agent A_1 and agent A_2 exchange resources at time 5.

The problem with these plans is that at time 5, the two agents 'exchange' resources. Unless there is enough space to maneuver at the intersection of resources r_1 and r_2, this attempted resource exchange will result in a collision. This collision can be prevented by the following constraint:

$$\begin{aligned} \forall i \forall j : (\langle r, [_, t_1)\rangle, \langle r', [t_1, _)\rangle) \sqsubseteq \pi_i \wedge \\ (\langle r', [_, t_2)\rangle, \langle r, [t_2, _)\rangle) \sqsubseteq \pi_j \rightarrow \\ t_1 \neq t_2 \end{aligned} \tag{2.2}$$

Note that because locations and intersections are connected by lanes, and locations and intersections by definition have a capacity of one (see section 2.1), we do not have to take into account the situation that two connected resources both

have a greater capacity than one. In that case the situation of figure 2.2 would not necessarily result in a collision, and we would need another constraint that considers the capacity of the resources involved (cf. [13]).

Bidirectional lane traversal

The second constraint prevents a bidirectional lane resource being used by multiple agents in both directions at the same time. This constraint is necessary if there is not enough room for the agents to pass each other on the resource. If a resource r is a bidirectional lane connecting intersections v and w (i.e., $\{v, w\} \in E$), then agents can travel both from v to w via r, and also from w to v. The constraint states that if there is an agent that occupies r in the interval $[t, t')$, and this agent is going from v to w, then no agent may be going from w to v along the *same resource* r, between t and t'. This implies that if two agents have overlapping occupation intervals on r, then they must enter r from the same intersection:

$$\begin{aligned} \forall i \forall j : (\langle v, _\rangle, \langle r, \tau\rangle, \langle w, _\rangle) \sqsubseteq \pi_i \wedge \\ (\langle w, _\rangle, \langle r, \tau'\rangle, \langle v, _\rangle) \sqsubseteq \pi_j \rightarrow \\ \tau \cap \tau' = \varnothing \end{aligned} \tag{2.3}$$

Note that if intervals τ and τ' from equation 2.3 meet, e.g. $\tau = [0, 3)$ and $\tau' = [3, 6)$, then a simultaneous resource exchange takes place at time 3 between resources r and w. If simultaneous resource exchanges are not allowed, then τ and τ' cannot meet.

Overtaking

The third constraint prevents agents from overtaking each other on a lane resource. If overtaking on lane resources is forbidden, then the interval in which an agent may use the resource is constrained by the agents that already occupy the resource. That is, an agent that wants to use the resource has to take into account the agent that enters the resource directly before it, called the *leading agent*, and the agent that enters the resource directly after it, called the *trailing* agent. Consider the following example.

Suppose we have a resource with a capacity of 3. Agent A_1 traverses the resource during interval $[20, 50)$, and agent A_2 traverses the resource during interval $[50, 70)$ (see figure 2.3). If now a third agent A_3 wants to use the resource somewhere during interval $[20, 70)$, then agent A_3 needs to drive in between agents A_1 and A_2; agent A_1, which enters the resource first, is A_3's leading agent, whereas agent A_2, which enters the resource last, is the trailing agent.

The interval in which agent A_3 can make use of the resource is constrained by agents A_1 and A_2, despite the fact that the capacity of the resource is never exceeded. Agent A_3 has to enter the resource later than time 20, but earlier than time 50 to traverse the resource in between agents A_1 and A_2. Furthermore, agent A_3 needs to exit the resource later than time 50 to avoid a catching-up conflict with agent A_1,

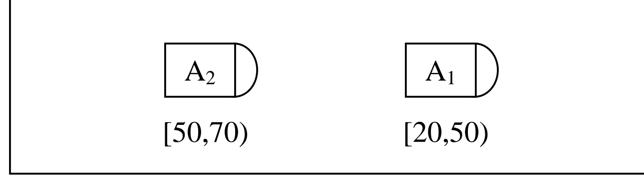


Figure 2.3: Agent A_1 is leading A_2.

and agent A_3 has to exit the resource earlier than time 70 to avoid a catching-up conflict with agent A_2. Hence, the entry and exit times of agent A_3 are governed by the following equations:

$$\begin{aligned} \text{entry}(A_1) < \text{entry}(A_3) < \text{entry}(A_2) \\ \text{exit}(A_1) < \text{exit}(A_3) < \text{exit}(A_2) \end{aligned}$$

The general overtaking constraint specifies that if two plan steps make use of the same resource, then the one that starts earlier should also finish earlier:

$$\begin{aligned} \forall i \forall j : \langle r, [t_1, t_2) \rangle \sqsubseteq \pi_i \wedge \langle r, [t_3, t_4) \rangle \sqsubseteq \pi_j \rightarrow \\ (t_1 > t_3 \wedge t_2 > t_4) \vee (t_3 > t_1 \wedge t_4 > t_2) \end{aligned} \tag{2.4}$$

Equation 2.4 ensures that agents exit resources in the order that they entered them, and as a result it allows agents to execute their plans without overtaking maneuvers taking place.

2.3 Reservations and Free Time Windows

As said in the introduction of this chapter, we take a sequential approach to route planning. The idea for route planning is that the first agent makes a plan, and then it places reservations on the resources for the intended periods of occupation. The second agent to make a plan has to take into account these reservations to avoid conflicts with the first agent. In general, agent n (i.e., the nth agent to make a plan) has to plan 'around' the reservations of agents $1, \ldots, n-1$ to avoid conflicts with the first $n-1$ agents. Hence, an agent that wants to make a plan is only allowed to use time intervals that do not conflict with the set of existing reservations on the resources. These time intervals are called *free time windows*.

Given a set of agent plans, the resource load of a resource tells us exactly when a resource is free to be used by other agents. During a time interval when the resource load is at least one less than the capacity, another agent may enter the resource. Figure 2.4 depicts the resource load for some resource r in the interval $[0, 10)$. The capacity of r is 3, and we see that during the intervals $[4, 6)$ and $[7, 8)$ this capacity is fully utilized. During the intervals $\tau = [0, 4)$, $\tau' = [6, 7)$, and $\tau'' = [8, 10)$ there is room for (at least) one more agent. However, if we assume that the minimum travel time of r is 2, then the interval $\tau' = [6, 7)$ is too short to be of use to any agent. Therefore, the only free time windows are τ and τ''.

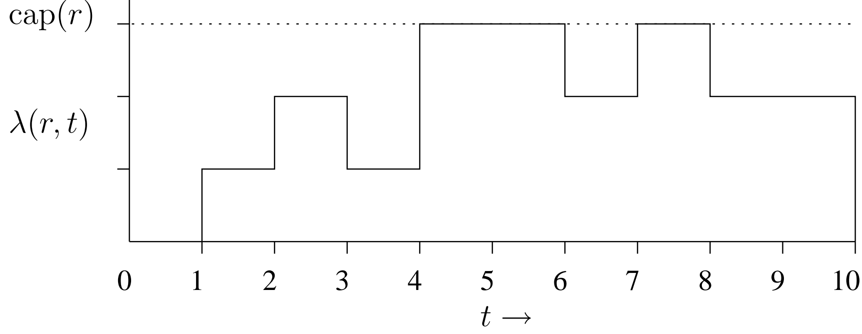


Figure 2.4: Resource load for resource r with capacity 3.

Definition 2.3.1 (Free Time Window). *Given a resource-load function λ, a free time window on resource r is a maximal interval $f = [t_1, t_2)$ such that:*

1. $\forall t \in f : \lambda(r, t) < cap(r)$;
2. $(t_2 - t_1) \geq tt(r)$.

The above definition states that for an interval to be a free time window, there should not only be sufficient capacity at any moment during that interval (condition 1), but it should also be long enough for an agent to traverse the resource (condition 2). The set of all free time windows $F = (F_1, \ldots, F_{|R|})$ is partitioned into $|R|$ sets: one set of free time windows F_i for every resource $r_i \in R$. Note that the set of free time windows F_i on resource r_i is a vector $(f_{i,1}, \ldots, f_{i,m})$ of disjoint intervals such that for all $j \in \{1, \ldots, m-1\}$, $f_{i,j}$ precedes $f_{i,j+1}$.

Within a free time window, an agent must enter a resource, traverse it, and exit the resource. Because of the (non-zero) minimum travel time of a resource, an agent cannot enter a resource right at the end of a free time window, and it cannot exit the window at the start of one. We therefore define for every free time window f an *entry window* $\tau_{\text{entry}}(f)$ and an *exit window* $\tau_{\text{exit}}(f)$. The sizes of the entry and exit windows of a free time window $f = [t_1, t_2)$ on resource r are constrained by the minimum travel time of the resource:

$$\tau_{\text{entry}}(f) = [t_1, t_2 - tt(r)) \quad (2.5)$$

$$\tau_{\text{exit}}(f) = [t_1 + tt(r), t_2) \quad (2.6)$$

An agent that wants to go from resource r to resource r' should find a free time window for both of these resources. By definition 2.2.1 of an agent plan, the exit time out of r should be equal to the entry time into r'. Hence, for a free time window f' on r' to be *reachable* from free time window f on r, the *entry* window of f' should overlap with the *exit* window of f.

Definition 2.3.2 (Free Time Window Reachability). *Given a free time window f on resource r, and a free time window f' on resource r', free time window f' is reachable from f, denoted $(f, f') \in E_F$, if:*

1. $(r, r') \in E_R$*;*

2. $\tau_{exit}(f) \cap \tau_{entry}(f') \neq \varnothing$*.*

The set of free time windows F together with the reachability relation E_F form a graph structure: the *free time window graph.*

Definition 2.3.3 (Free Time Window Graph)**.** *The* free time window graph $G_F = (F, E_F)$ *is a directed graph where the set of vertices, given by* $F = \bigcup_{i=1}^{|R|} F_i$*, is the set of free time windows, and the set of edges is given by the reachability relation* E_F*.*

The vertices (i.e., the free time windows) of the free time window graph represent the times at which resources can be entered by an agent without introducing conflicts with other agents. The edges of the free time window graph specify the reachability between the free time windows. In section 2.4 we will show how an agent can find an optimal conflict-free route by performing a search through this free time window graph.

Definition 2.3.1 of a free time window is only based on the resource load constraint: an agent may enter a resource if there is enough capacity left. In section 2.2.1 additional constraints for agent plans were discussed. We will now discuss how the definition of a free time window can be extended to take these additional constraints into account.

2.3.1 Additional constraints and free time windows

In this section, we show how each of the additional constraints from section 2.2.1 can be 'encoded' into the definition of a free time window.

Simultaneous resource exchanges

The first constraint that we discussed was a constraint that prevented a head-on conflict when two agents exchange resources at the same point in time. We will show that under definition 2.3.1 of a free time window, a simultaneous resource exchange is not possible.

Take a look at figure 2.2 again. Imagine now that agent A_1 computed his plan $\pi_1 = (\langle r_1, [0, 5)\rangle, \langle r_2, [5, \infty)\rangle)$, and reserved this plan on resources r_1 and r_2. After reserving plan π_1 there is one free time window on resource r_1: $f_{1,1} = [5, \infty)$ that starts after A_1 has left the resource, and one free time window on resource r_2: $f_{2,1} = [0, 5)$ prior to A_1's traversal.

If agent A_2 now computes his plan, and wants to perform a simultaneous resource exchange with agent A_1, it would make the plan $\pi_2 = (\langle r_2, [0, 5)\rangle, \langle r_1, [5, \infty)\rangle)$. The free time windows used by this plan are $f_{2,1}$ and $f_{1,1}$ respectively. However, $f_{1,1}$ is not reachable from $f_{2,1}$ according to definition 2.3.2, since $[0, 5) \cap [5, \infty) = \varnothing$. Hence, if we make use of the free time window graph to find a plan for A_2 from r_2 to r_1, then we cannot find the plan π_2 that would perform a simultaneous resource exchange with A_1.

Bidirectional lane traversal

The second constraint that we discussed ensured that a lane resource is only used in one direction at the same time to prevent head-on conflicts. We can prevent bidirectional lane traversal by maintaining one set of free time windows for each direction of the bidirectional lane resource. We can deduce the direction of an agent plan step from the preceding plan step. Suppose we have a lane resource r that is connected to intersections v and w. If an agent plan contains the subsequence (v, r), then the agent will exit r at w (see figure 2.5). If an agent plan contains the subsequence (w, r), then the agent travels in the opposite direction, and will exit r at v.

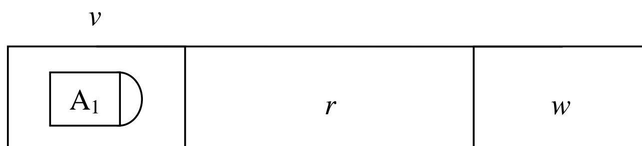


Figure 2.5: Agent A_1 will enter lane resource r from intersection v, and must therefore exit r via intersection w.

We introduce the augmented resource load function $\lambda^a : R \times R \times T$, such that $\lambda(r', r, t)$ specifies the number of agents that are on resource r at time t, and have come directly from resource r'.

Definition 2.3.4 (Directed Free Time Window)**.** *Given an augmented resource-load function λ^a, and two intersection resources v and w connected by lane resource r, a free time window on resource r is a maximal interval $f = [t_1, t_2)$ such that:*

1. $\forall t \in f : \lambda^a(v, r, t) < cap(r)$;
2. $\forall t \in f : \lambda^a(w, r, t) = 0$;
3. $(t_2 - t_1) \geq tt(r)$.

In definition 2.3.4, the free time window f is implicitly defined for the direction from v to r. To obtain the free time window in the other direction, we can exchange v and w.

Overtaking

The third constraint that we discussed ensured that agents cannot overtake each other on a lane resource to prevent catching-up conflicts. The constraint stated that if an agent enters a resource before another, it should also exit the resource first. For this constraint, we need to redefine a free time window in terms of the plan steps of the leading and trailing agents. Let $\tau_{lead}(r, t)$ be the plan step of the first agent to enter resource r before time t, and let $\tau_{trail}(r, t)$ be the plan step of the first agent to enter resource r after time t. We will also assume that there is

a minimum separation time δ between two successive agents. Hence, if an agent traverses a resource in the interval $[t_1, t_2)$, then no agent may enter the resource in $(t_1 - \delta, t_1 + \delta)$, and no agent may exit the resource during $(t_2 - \delta, t_2 + \delta)$.

Definition 2.3.5 (Free Time Window no Overtaking)**.** *Given a resource-load function λ, an intended entry time t^*, a leading-agent interval $\tau_{lead}(r, t^*) = [t_1, t_2)$, a trailing-agent interval $\tau_{trail}(r, t^*) = [t_3, t_4)$, and a minimum separation time δ, a free time window on resource r that does not allow overtaking is a maximal interval $f = [t, t')$ such that:*

1. $t = t_1 + \delta$;
2. $t' = t_4 - \delta$;
3. $\forall t_i \in f : \lambda(r, t_i) < cap(r)$;
4. $(t' - t) \geq tt(r)$.

The entry and exit windows of f are specified by

$$\tau_{entry}(f) = [t_1 + \delta, t_3 - \delta) \quad (2.7)$$

$$\tau_{exit}(f) = [t_2 + \delta, t_4 - \delta) \quad (2.8)$$

In case there is no leading agent, we can set $\tau_{lead}(r, t) = [-\delta, tt(r) - \delta]$, and if there is no trailing agent, we can set $\tau_{trail}(r, t) = (\infty, \infty)$.

2.4 Single-Stage Route Planning

In this section a single-stage route planning algorithm by Ter Mors, Zutt, and Witteveen [14] is explained. They present an algorithm that uses the free time window graph $G_F = (F, E_F)$ to find a shortest-time route plan for a single agent, while respecting the reservations of previous agents. The algorithm searches for a shortest-time path through the free time window graph, and finds an optimal conflict-free route plan on both unidirectional and bidirectional infrastructures. The algorithm is based on the A* search algorithm [4, 6, 12]. Therefore, we will start with an explanation of the A* algorithm.

2.4.1 The A* algorithm

Firstly, we will give a description of the A* algorithm. Then, we will mention some interesting properties of the A* algorithm, and give a specification. We will conclude with a simple example.

Algorithm description

The A* algorithm is a best-first graph search algorithm that finds the least-cost path from a start node to a destination node. It first searches the nodes that appear to be most likely to lead to a shortest path to the destination node. To accomplish this, A* uses an estimated cost function $y(v)$ to determine the order in which the search visits nodes in the graph. In the traditional A* notation the symbol f is used for the estimated cost function, but we already use the symbol f for a free time window, so instead we will use y for the estimated cost function. The function $y(v)$ is the sum of two functions:

- $g(v)$: The partial cost function that calculates the cost from the start node to the current node v.
- $h(v)$: A heuristic function that estimates the cost from the current node v to the destination node.

Hence, the function $y(v)$ is the estimated cost of the cheapest solution through v.

The A* algorithm maintains a list of nodes that can be expanded, known as the *open list*[2]. The open list can be seen as a list of partial paths from the start node to the destination node. For every node on the open list the g, h, and y values, and a pointer to its predecessor node are stored. Hence, a node on the open list can be represented as a 5-tuple $\langle v, g, h, y, p \rangle$, where:

- v is the node;
- g is the g value of v;
- h is the h value of v;
- y is the y value of v;
- p is a pointer to the node's predecessor.

Maintaining pointers is only necessary if you want to construct the actual path from the start node to the destination node when the algorithm terminates. In each iteration the most promising node is expanded, i.e., the node v with the lowest value of $y(v)$. If there are more candidate nodes for expansion, meaning that they have the same y value, it does not matter which node is chosen. Expanding a node consists of several steps. Firstly, the node is removed from the open list. Then, for all successor nodes the g, h, and y values are calculated, and for every successor node the pointer is set to the node currently being expanded. Finally, if a successor node is not already on the open list, it is added to the open list. If a successor node is already on the open list, we need to check if the newly found path to the node is cheaper than the path to the node already on the open list. In other words, if the g value for the node on the open list is higher than the newly found g value for the

[2] The open list is usually implemented as a priority queue.

node, then the newly found path to the node is better. In that case we replace the node on the open list. If the g value for the node on the open list is lower or equal than the newly found g value for the node, then the newly found path to the node is worse or equal, and nothing is done. The algorithm continues until the destination node is taken off the open list or until the open list is empty. In the first case a least-cost path is found, in the latter case there doesn't exist a path from the start node to the destination node. The actual path can be constructed by following the pointer of the destination node to its predecessor node, and from that node working backwards to the start node.

Properties of the A* algorithm

The A* algorithm is *complete* in the sense that it will always find a solution if there is one [12]. For A* to be *optimal*, meaning that it will always find an optimal solution, the heuristic function $h(v)$ needs to be *admissible* [6, 12]. A heuristic is admissible if it never overestimates the cost of reaching the destination node. Additionally, the heuristic function $h(v)$ can be *consistent* [6, 12]. A heuristic is consistent if for any pair of adjacent nodes v and v', where $d(v, v')$ denotes the cost of getting from v to v', we have:

$$h(v) \leq d(v, v') + h(v') \tag{2.9}$$

Note that a consistent heuristic is always admissible, but an admissible heuristic need not be consistent. In route planning, a consistent heuristic function can be e.g. the straight-line distance from the current node to the destination node. We now give a short proof that the straight-line distance is indeed a consistent heuristic function. By the triangle inequality from mathematics we have:

$$d_{STL}(v, \text{destination}) \leq d_{STL}(v, v') + d_{STL}(v', \text{destination})$$

The step cost $d(v, v')$ between two nodes v and v' is always greater or equal to the straight-line distance $d_{STL}(v, v')$ between v and v': $d(v, v') \geq d_{STL}(v, v')$. This leads to:

$$\begin{gathered} d_{STL}(v, \text{destination}) \leq d(v, v') + d_{STL}(v', \text{destination}) \Rightarrow \\ h(v) \leq d(v, v') + h(v'). \end{gathered}$$

If a heuristic is consistent, then A* evaluates nodes with non-decreasing y values [12]. Russell and Norvig give the following short proof:

> Suppose that v' is a successor of v; then $g(v') = g(v) + d(v, v')$, and we have:
>
> $$y(v') = g(v') + h(v') = g(v) + d(v, v') + h(v') \geq g(v) + h(v) = y(v).$$

An important property of A* is that, if a consistent heuristic function is used, a node needs to be expanded at most once [12]. This means that when a node is expanded,

we have the guarantee that we have found a cheapest path from the start node to this node. In that case a *closed list* of nodes that have already been expanded can be used to make the search more efficient. When a node is expanded, it is put on the closed list, and for every successor node it is checked if it is on the closed list, and when it is, that successor node can be ignored. Finally, we mention that A* is computationally optimal for any heuristic function, meaning that no other algorithm employing the same heuristic will expand fewer nodes than A* [4].

Specification of the A* algorithm

For the sake of completeness, we will now give a specification of the A* algorithm. In this thesis we will only treat algorithms that use consistent heuristic functions. Therefore, we included the use of a closed list in the specification. Furthermore, note that we use the straight-line distance as the heuristic function, but any other consistent heuristic could also be used.

Algorithm 1 A*

Require: start node v_1, destination node v_2; graph $G = (V, E)$.
Ensure: shortest path from v_1 to v_2.

```
 1: g(v_1) ← 0
 2: h(v_1) ← d_STL(v_1, v_2)
 3: y(v_1) ← g(v_1) + h(v_1)
 4: pointer(v_1) ← nil
 5: add(v_1, open)
 6: while open ≠ ∅ do
 7:     v ← argmin_{v'∈open} y(v')
 8:     remove(v, open)
 9:     add(v, closed)
10:     if v = v_2 then
11:         return followPointers(v)
12:     for all v' ∈ {successors(v) \ closed} do
13:         g(v') ← g(v) + d(v, v')
14:         h(v') ← d_STL(v', v_2)
15:         y(v') ← g(v') + h(v')
16:         pointer(v') ← v
17:         if ∃v''[v'' ∈ open | v'' = v'] then
18:             if g(v'') > g(v') then
19:                 replace(v'', v', open)
20:         else
21:             add(v', open)
22: return nil
```

In lines 1 to 3 the g, h, and y values are calculated for the start node v_1. The start node needs no pointer, so in line 4 the pointer of v_1 is set to nil. Then, in

line 5 node v_1 is added to the open list. Next, as long as the open list is not empty, in each iteration the most promising node is expanded. In line 7 the node v with the lowest value of $y(v)$ is selected from the open list. Then, the node is removed from the open list, and added to the closed list. If the selected node v equals the destination node v_2, a shortest path to v_2 has been found. The path is constructed in line 11 by following a series of pointers from the destination node backwards to the start node. If v is not the destination node v_2, the selected node v is expanded to all successor nodes that are not on the closed list. In lines 13 to 16 the g, h, and y values are calculated for successor node v', and the pointer is set to the selected node v. In line 17 it is checked if the successor node is already on the open list. If it is, we check in line 18 if the g value for the node on the open list is higher than the newly found g value for the node. If the g value for the node on the open list is higher, the node on the open list is replaced in line 19. If the open list doesn't contain the successor node, the node is added to the open list in line 21. Then, the next node on the open list is expanded in the next iteration. Finally, if the open list is empty before a path is found, then no path exists, and nil is returned in line 22.

Example of the A* algorithm

Consider figure 2.6 for an example of how A* works. Suppose we want to find a path from node a to e. We will use the following heuristic function: $h(a) = 5$, $h(b) = 3$, $h(c) = 4$, $h(d) = 1$, and $h(e) = 0$. Note that this heuristic is consistent.

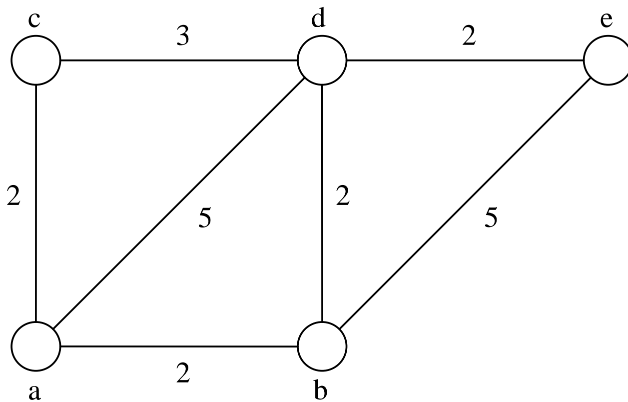


Figure 2.6: A graph with five nodes, and the edges have costs as depicted.

We start by calculating the g, h, and y values for the start node a: $y(a) = g(a) + h(a) = 0 + 5 = 5$, and we put it on the open list. Because node a is the start node, no pointer needs to be set. The open list now looks like: $(\langle a, 0, 5, 5, - \rangle)$. Then, we take node a off the open list, and put it on the closed list. The successors of node a are nodes b, c, and d. We calculate the g, h, and y values for these nodes: $y(b) = 2 + 3 = 5$, $y(c) = 2 + 4 = 6$, and $y(d) = 5 + 1 = 6$, and for every node the pointer is set to node a. Then, we put the successor nodes on the open list. The

open list now looks like: $(\langle b, 2, 3, 5, a\rangle, \langle c, 2, 4, 6, a\rangle, \langle d, 5, 1, 6, a\rangle)$. Node b has the lowest y value, so we take this node off the open list, and put it on the closed list. The closed list now contains nodes a and b. The successors of node b are nodes a, d, and e. Node a is on the closed list, so we can ignore it. We calculate the g, h, and y values for the other nodes: $y(d) = 4 + 1 = 5$ and $y(e) = 7 + 0 = 7$, and for every node the pointer is set to node b. Node d is already on the open list, and its g value is higher than the newly found g value, so we need to replace node d, because we found a shorter path. Node e is not on the open list yet, so it is added. The open list now looks like: $(\langle c, 2, 4, 6, a\rangle, \langle d, 4, 1, 5, b\rangle, \langle e, 7, 0, 7, b\rangle)$. Node d has the lowest y value, so we take this node off the open list, and put it on the closed list. The closed list now contains nodes a, b, and d. The successors of node d are nodes a, b, c, and e. Nodes a and b are on the closed list, so we can ignore them. We calculate the g, h, and y values for the other nodes: $y(c) = 7 + 4 = 11$ and $y(e) = 6 + 0 = 6$, and for every node the pointer is set to node d. Nodes c and e are both already on the open list. We replace node e, because the newly found path is shorter than the one on the open list ($6 < 7$). The newly found path for node c is longer than the one on the open list ($7 > 2$), so nothing is done with the newly found path. The open list now looks like: $(\langle c, 2, 4, 6, a\rangle, \langle e, 6, 0, 6, d\rangle)$. Both nodes have the same y value. It does not matter which node we take off for expansion. We choose to take node e off the open list. This node is the destination node, so we now found a shortest path from node a to e, and its cost is 6. By following the pointer of node e back to the start node, we can construct the actual path. The pointer of node e leads to node d, node d leads to node b, and node b leads to node a, so the actual path is $a - b - d - e$.

2.4.2 A single-stage route planning algorithm

Ter Mors, Zutt, and Witteveen [14] present a single-stage route planning algorithm that performs a search through the free time window graph $G_F = (F, E_F)$. The route planning algorithm is based on the idea of going from one free time window on one resource, to another free time window on the next resource. In principle they apply the A* algorithm on the free time window graph. The algorithm uses an estimated cost function $y(f)$ to determine the order in which free time windows are selected for expansion. The function $y(f)$ is the sum of two functions:

- $g(f)$: The partial cost function that calculates the earliest possible exit time out of free time window f.
- $h(f)$: The heuristic function that estimates the cost from the current resource associated with free time window f to the destination resource.

The earliest possible exit time out of free time window f is equal to the time an agent enters free time window f plus the travel time of the resource r associated with f: $g(f) = \text{entryTime}(f) + tt(r)$. The heuristic function $h(f)$ calculates the cost of the shortest path from the current resource r to the destination resource r' without

taking into account the presence of other agents: $h(f) = \text{shortestPath}(r, r')$. Note that this heuristic function is consistent. To save computation time the heuristic function $h(f)$ can be precalculated, and the results can be stored in a matrix. This can be done by performing a search with A* through the resource graph for every combination of resources or by a specialized algorithm that finds the shortest paths between all pairs of resources like the Floyd-Warshall algorithm [5]. Hence, the estimated cost of a route through free time window f with destination resource r' is:

$$y(f) = g(f) + h(f) = \text{entryTime}(f) + tt(r) + \text{shortestPath}(r, r').$$

To determine if f' is a successor of free time window f, it is not sufficient to check if f' is reachable from f as described in definition 2.3.2. Take a look at figure 2.7. Free time window f' is reachable from free time window f, because the exit window of f overlaps with the entry window of f'. However, from the earliest possible exit time $g(f)$ out of free time window f the entry window of f' cannot be reached, so f' is no successor of f. Hence, to determine if $f' \in \text{successors}(f)$ with $f = [t_1, t_2)$, we have to check if f' is reachable from f from the earliest possible exit time $g(f)$ out of f: $[g(f), t_2) \cap \tau_{\text{entry}}(f') \neq \varnothing$.

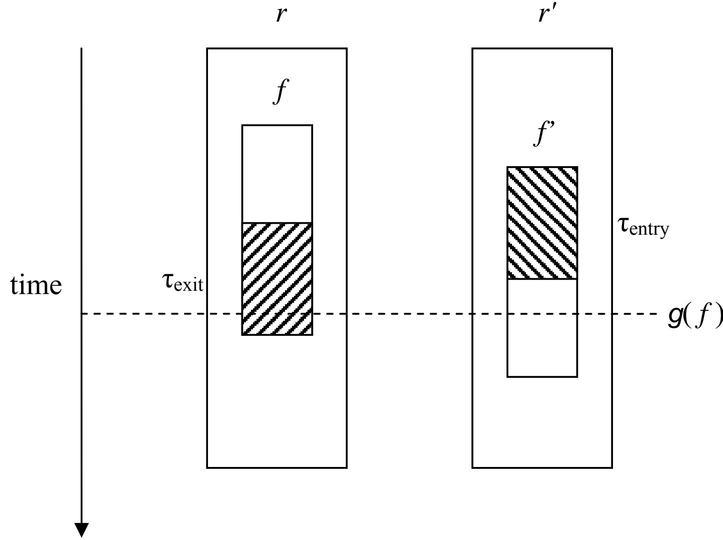


Figure 2.7: Free time window f' is reachable from free time window f, but not from the earliest possible exit time $g(f)$.

The single-stage route planning algorithm maintains an open list of free time windows that can be expanded, and in each iteration the free time window that is most likely to lead to a shortest route to the destination resource is expanded, i.e., the free time window f with the lowest value of $y(f)$. The open list can be seen as a list of partial routes from the start resource to the destination resource. For every free time window on the open list the entry time t, the g, h, and y values, and a

pointer to its predecessor free time window are stored. Hence, a free time window on the open list can be represented as a 6-tuple $\langle f, t, g, h, y, p \rangle$, where:

- f is the free time window;
- t is the entry time into f;
- g is the g value of f;
- h is the h value of f;
- y is the y value of f;
- p is a pointer to the free time window's predecessor.

Algorithm 2 is a specification of the algorithm. In line 1 it is checked if there exists a free time window on the start resource r_1 that has an entry window that contains the start time t. If there isn't such a free time window f, then no plan exists, and nil is returned in line 25. Otherwise, in line 2 the entry time into f is recorded as the start time t. Then, in lines 3 to 6 the g, h, and y values are calculated for the start window f, and the pointer is set to nil. In line 7 the free time window f is added to the open list. Next, as long as the open list is not empty, in each iteration the most promising free time window is expanded. In line 9 the free time window f with the lowest value of $y(f)$ is selected from the open list. Then, the free time window is removed from the open list, and added to the closed list. If the resource associated with the selected free time window f equals the destination resource r_2, a shortest route to r_2 has been found. The optimal plan is constructed in line 13 by following a series of pointers from the current free time window f backwards to the start window. Note that the cost of the plan as defined in definition 2.2.2 is equal to $g(f)$. If the resource associated with f is not the destination resource r_2, the selected free time window f is expanded to all successor free time windows that are not on the closed list. Recall, as described above, that a free time window f' is a successor of the selected free time window f if f' is reachable from f from the earliest possible exit time $g(f)$ out of free time window f. In line 15 the entry time into successor free time window f' is determined by the maximum of the earliest possible exit time out of f, and the start of free time window f'. Then, in lines 16 to 19 the g, h, and y values are calculated for successor f', and the pointer is set to the selected free time window f. In line 20 it is checked if the successor free time window is already on the open list. If it is, we check in line 21 if the entry time into the free time window on the open list is later than the newly found entry time into the free time window. Note that this check is equivalent to checking if $g(f'') > g(f')$ as is done in the A* algorithm. If the entry time into the free time window on the open list is later, the free time window on the open list is replaced in line 22. If the open list doesn't contain the successor free time window, the free time window is added in line 24. Then, the next free time window on the open list is expanded in the next iteration. Finally, if the open list is empty before a route is found, then no route exists, and nil is returned in line 25.

Algorithm 2 Single-Stage Route Planning

Require: start resource r_1, destination resource r_2, start time t; free time window graph $G_F = (F, E_F)$.

Ensure: shortest-time, conflict-free route plan from (r_1, t) to r_2.

```
1: if ∃f [f ∈ F | t ∈ τ_entry(f) ∧ r_1 = resource(f)] then
2:     entryTime(f) ← t
3:     g(f) ← t + tt(r_1)
4:     h(f) ← shortestPath(r_1, r_2)
5:     y(f) ← g(f) + h(f)
6:     pointer(f) ← nil
7:     add(f, open)
8: while open ≠ ∅ do
9:     f ← argmin_{f'∈open} y(f')
10:    remove(f, open)
11:    add(f, closed)
12:    if resource(f) = r_2 then
13:        return followPointers(f)
14:    for all f' ∈ {successors(f) \ closed} do
15:        entryTime(f') ← max(g(f), start(f'))
16:        g(f') ← entryTime(f') + tt(resource(f'))
17:        h(f') ← shortestPath(resource(f'), r_2)
18:        y(f') ← g(f') + h(f')
19:        pointer(f') ← f
20:        if ∃f''[f'' ∈ open | f'' = f'] then
21:            if entryTime(f'') > entryTime(f') then
22:                replace(f'', f', open)
23:        else
24:            add(f', open)
25: return nil
```

Ter Mors et al. prove that the algorithm is guaranteed to return an optimal solution. The algorithm has a run-time complexity of $O((|E_F| + |F|)\log(|F|))$. If the heuristic function $h(f)$ is set to zero, which is a consistent heuristic, they prove that the first visit to a free time window is optimal. This means that each free time window needs to be added to the open list at most once. In that case we can add a free time window to the closed list directly after it is added to the open list instead of adding a free time window to the closed list after it is removed from the open list. Algorithm 3 is a specification of the single-stage route planning algorithm without a heuristic function. Removing the heuristic function results in a run-time complexity of $O(|F|\log(|F|) + |E_F|)$.

Algorithm 3 Single-Stage Route Planning without Heuristic

Require: start resource r_1, destination resource r_2, start time t; free time window graph $G_F = (F, E_F)$.

Ensure: shortest-time, conflict-free route plan from (r_1, t) to r_2.

```
 1: if ∃f [f ∈ F | t ∈ τ_entry(f) ∧ r_1 = resource(f)] then
 2:     entryTime(f) ← t
 3:     g(f) ← t + tt(r_1)
 4:     pointer(f) ← nil
 5:     add(f, open)
 6:     add(f, closed)
 7: while open ≠ ∅ do
 8:     f ← argmin_{f'∈open} g(f')
 9:     remove(f, open)
10:     if resource(f) = r_2 then
11:         return followPointers(f)
12:     for all f' ∈ {successors(f) \ closed} do
13:         entryTime(f') ← max(g(f), start(f'))
14:         g(f') ← entryTime(f') + tt(resource(f'))
15:         pointer(f') ← f
16:         add(f', open)
17:         add(f', closed)
18: return nil
```

2.4.3 Additional constraints and single-stage route planning

In section 2.3.1 we showed how the definition of a free time window could be extended to take the additional constraints from section 2.2.1 into account. If overtaking is not allowed, some changes are needed to the single-stage route planning algorithm. In the previous section the partial cost function was defined as $g(f) = \text{entryTime}(f) + tt(r)$. However, if overtaking is forbidden, then the start of the exit window $\tau_{\text{exit}}(f)$ may be later than the entry time plus the travel time, because of the leading agent's exit time. Hence, the partial cost function has to be changed to $g(f) = \max(\text{start}(\tau_{\text{exit}}(f)), \text{entryTime}(f) + tt(r))$.

Furthermore, if the entry time into a successor free time window was earlier than the entry time into the same free time window that was already on the open list, the free time window on the open list was replaced in line 22. However, if the earliest possible exit time out of the free time window on the open list was determined by the start of the exit window $\tau_{\text{exit}}(f)$, the earlier entry time into the free time window doesn't matter, because we can still leave the free time window no earlier than the start of the exit window $\tau_{\text{exit}}(f)$, even though we enter it earlier than before. Hence, in line 21 it should be checked whether $g(f'') > g(f')$ in order to avoid unnecessary replace operations. Note that the new check is an optimization, and not necessarily required for the correct operation of the algorithm.

Chapter 3

The Context-Aware Multi-Stage Route Planning Problem

In the previous chapter a framework for context-aware route planning was presented. The idea for the single-stage route planning algorithm was to find a shortest-time path through the free time window graph, rather than directly through the infrastructure graph itself. In the multi-stage variant of the context-aware route planning problem, each agent has a *sequence* ϕ of resources it must visit instead of just a start and destination resource. This *visiting sequence* must be visited in a fixed order, that is, if $\phi = (r_1, r_2, \ldots, r_m)$, then for any plan π it must hold that $\phi \sqsubseteq \text{resources}(\pi)$ (the visiting sequence is a sub-sequence of the resources in the plan).

A straightforward approach to the multi-stage route planning problem is to partition the problem into a sequence of 'single-stage' route planning problems, and then to concatenate the resulting plans. The first section of this chapter describes this *linear*[1] concatenation approach. Then, in section 3.2, we will show the problem of this approach. We will discuss an example that not only demonstrates that the linear concatenation approach is *sub-optimal*, but also that it is *incomplete*, i.e., it can fail to find a route plan even if one exists.

3.1 Linear Concatenation Approach

Algorithm 4 is a specification of the linear concatenation algorithm. In this algorithm we represent a plan as a tail end X of the plan, and a head consisting of a tuple $\langle r, t \rangle$, where:

- r is the last visited resource of the plan;
- t is the entry time into r.

In line 1 we try to find a plan from the start resource r_1 to the second resource of the visiting sequence ϕ with the single-stage route planning algorithm (planRoute

[1] We call this approach the linear concatenation approach to clarify the difference with the optimal and complete concatenation approaches dealt with in the next chapter.

means a call to the single-stage route planning algorithm (algorithm 2)). If no plan is found, then no multi-stage plan can be found, and we return π, which is currently nil, in line 11. Otherwise, we will try, as long as the final stage m hasn't been reached, to expand the current partial plan π to the next stage. In line 5 we try to find a plan from the current stage i to the next stage $i+1$ with the single-stage route planning algorithm. Note that the earliest possible start time we can use to find a plan to the next stage is the time t' at which the current stage is entered. If no plan is found, then no multi-stage plan can be found, and we return nil in line 9. Otherwise, the plan that has been expanded in this iteration is concatenated with the newly found plan in line 7. Then, in line 10 the counter i is updated to the next stage, and we try to expand the concatenated plan in the next iteration. When we reach the final stage m in the visiting sequence ϕ, we return the multi-stage plan in line 11.

Algorithm 4 Multi-Stage Linear Concatenation

Require: visiting sequence $\phi = (r_1, r_2, \ldots, r_m)$, start time t; free time window graph $G_F = (F, E_F)$.

1: $\pi = (\langle r_2, t' \rangle, X) \leftarrow \text{planRoute}(r_1, r_2, t, G_F)$
2: **if** $\pi \neq \mathsf{nil}$ **then**
3: $\quad i \leftarrow 2$
4: $\quad$ **while** $i < m$ **do**
5: $\quad\quad \pi_{(i,i+1)} \leftarrow \text{planRoute}(r_i, r_{i+1}, t', G_F)$
6: $\quad\quad$ **if** $\pi_{(i,i+1)} \neq \mathsf{nil}$ **then**
7: $\quad\quad\quad \pi = (\langle r_{i+1}, t' \rangle, X) \leftarrow \pi \frown \pi_{(i,i+1)}$
8: $\quad\quad$ **else**
9: $\quad\quad\quad$ **return** nil
10: $\quad\quad i \leftarrow i + 1$
11: **return** π

3.1.1 Complexity

The computational complexity of algorithm 4 is determined by the number of calls that are made to the single-stage route planning algorithm.

Proposition 3.1.1. *Algorithm 4 has a run-time complexity of* $O(|\phi| \cdot (|E_F| + |F|) \log(|F|))$.

Proof. When a multi-stage plan is found, the algorithm uses exactly $|\phi| - 1$ calls to the single-stage route planning algorithm, so the run-time complexity of the algorithm is $O(|\phi|)$ times the complexity of algorithm 2. □

3.2 Incompleteness and Sub-Optimality of the Linear Concatenation Approach

In this section we will demonstrate that the linear concatenation approach to the context-aware multi-stage route planning problem is incomplete (i.e., it does not always find a solution when there exists one) and sub-optimal. This has the following reason: if we are looking for a route from resource r_a to resource r_c, then it does not hold that the shortest route to an intermediate resource r_b (i.e., resource r_b is on all shortest routes from r_a to r_c) can be expanded to the shortest route to r_c[2]. Instead, it may be necessary to take a slower route to r_b in order to find the shortest route to r_c. This is illustrated in the following example.

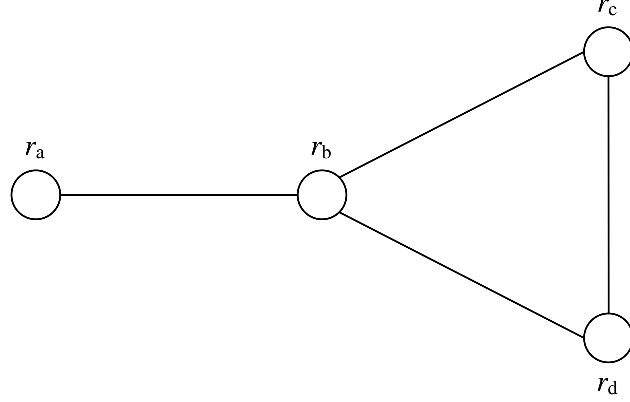
Example 3.2.1. *In figure 3.1, there are four resources r_a, r_b, r_c, and r_d, and there are some reservations on r_b and r_d. The travel times of r_a, r_b, and r_d are 2, and the travel time of r_c is 5. Suppose that an agent wants to go from r_a to r_d. In r_a, there is only a single free time window, and from that free time window it can reach both free time windows on r_b. Obviously, the shortest route to r_b makes use of the earlier free time window. However, since the free time window $f_{b,1} = [0, 4)$ ends before the start of the only free time window $f_{d,1} = [5, \infty)$ of resource r_d, the shortest route to r_b can only be expanded to r_c. Traversing r_c requires 5 time units, so r_d is entered at time 9. However, the shortest route to r_d enters r_b at time 6, the start of the second free time window $f_{b,2}$, and then goes directly to resource r_d, which it enters at time 8.*

From this example we conclude that the linear concatenation approach to multi-stage route planning is sub-optimal: for the visiting sequence (r_a, r_b, r_d), the linear concatenation approach would return the plan that enters resource r_d at time 9 which is clearly not optimal. To demonstrate that the linear concatenation approach is also incomplete, only a small modification to the example is required: if we remove resource r_c altogether, then the shortest route to r_b, ending in $f_{b,1}$, cannot be expanded at all, so the linear concatenation approach will not find a plan for the visiting sequence (r_a, r_b, r_d). Hence, as a result of the reservations in the system, the linear concatenation approach is incomplete and sub-optimal.

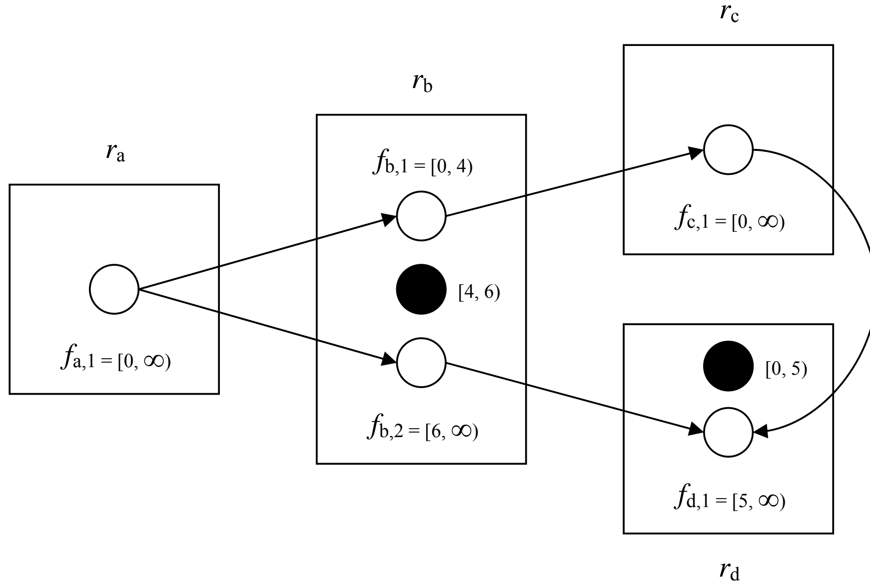
3.3 Concluding Remarks

In this chapter we demonstrated that the trivial solution (i.e., concatenating subsequent plans) to the multi-stage route planning problem doesn't work in case of context-aware route planning. The question is if there are polynomial algorithms that can guarantee an optimal multi-stage route plan if one exists. The answer to this question is yes. In the next chapter we will present three complete and optimal algorithms for the multi-stage route planning problem. Then, the next question is

[2]By contrast, in classical shortest path planning, it does hold that the shortest path to an intermediate node can be expanded to the shortest path to the destination node.



(a) An infrastructure graph with four intersection resources r_a, r_b, r_c, and r_d. For the sake of simplicity the lane resources (i.e., the edges) have a zero travel time.



(b) The associated free time window graph. Blacks circles represent reservations, open circles represent free time windows, and the arrows represent the reachability between the free time windows.

Figure 3.1: Infrastructure graph and free time window graph.

how the execution times of these three multi-stage route planning algorithms relate to the execution time of the single-stage route planning algorithm. We know that the single-stage route planning algorithm can be used in real-time applications, because the execution time of the algorithm lies in the range of several milliseconds [14]. Ideally, the execution time of a new multi-stage algorithm would be in the same range as that of the single-stage algorithm. This way we can easily extend existing real-time systems, which make use of the single-stage route planning algorithm, with the new multi-stage algorithm. Take for example the de-icing problem at an airport as mentioned in the introduction of this thesis, and suppose that the

making of a multi-stage plan takes several minutes. Now, when plans have to be generated for a lot of airplanes under wintry conditions, calculations may take up to hours of time. Then, the multi-stage algorithm is clearly not suitable for use in a real-time environment. Other interesting questions are how often the linear concatenation algorithm fails to find a plan, how often it finds a sub-optimal plan, and when it finds a sub-optimal plan, what is the plan quality. The answers to these questions actually determine to what extent there is a need for a complete and optimal multi-stage algorithm. We can assume that the linear concatenation algorithm is pretty fast, since it only requires $|\phi| - 1$ calls to the single-stage route planning algorithm. Suppose now that a complete and optimal algorithm is very slow, and the linear concatenation algorithm quite often finds a plan that also has a good quality, then it might be interesting to use the linear concatenation algorithm as multi-stage algorithm. However, if the complete and optimal algorithm is also fast, then there is no reason to use the linear concatenation algorithm. In chapter 5 we will empirically examine the linear concatenation approach. Furthermore, we will look into the execution times of the complete and optimal multi-stage algorithms.

Chapter 4

Context-Aware Multi-Stage Route Planning Algorithms

During our research on multi-stage route planning, we have developed three complete and optimal algorithms. The first two algorithms make use of the single-stage route planning algorithm in order to find partial plans from a certain stage to the next stage. The first algorithm is the most simple approach. The algorithm explores the free time windows on the several stages of the visiting sequence in a breadth-first way, and concatenates subsequent partial plans to find a multi-stage plan. A 'smart' administration of visited free time windows is kept to avoid finding useless plans. This approach is called the breadth-first concatenation approach, and it is explained in section 4.1. The second algorithm is an optimized version of the first algorithm. It is a best-first search algorithm, which is similar to the A* algorithm. An optimal conflict-free multi-stage plan is found by expanding the most promising partial plan in each iteration. This approach is called the A* concatenation approach, and it is explained in section 4.2. The third algorithm doesn't make use of the concatenation of partial plans found by the single-stage route planning algorithm. The algorithm uses a separate free time window graph for every stage, and it is similar to the single-stage route planning algorithm. This approach is called the A* multi-layer approach, and it is treated in the last section of this chapter.

4.1 Breadth-First Concatenation Approach

As explained in the previous chapter, the problem with the linear concatenation approach is that only the shortest-time plan between two successive resources of the visiting sequence is considered for concatenation. Sometimes it is necessary to take a slower plan for concatenation as shown in example 3.2.1. The breadth-first concatenation approach simply tries to find all possible plans between two successive resources of the visiting sequence with the single-stage route planning algorithm, and then expand these plans to the next stage. It performs a complete breadth-first search through the free time windows on the stages of the visiting sequence.

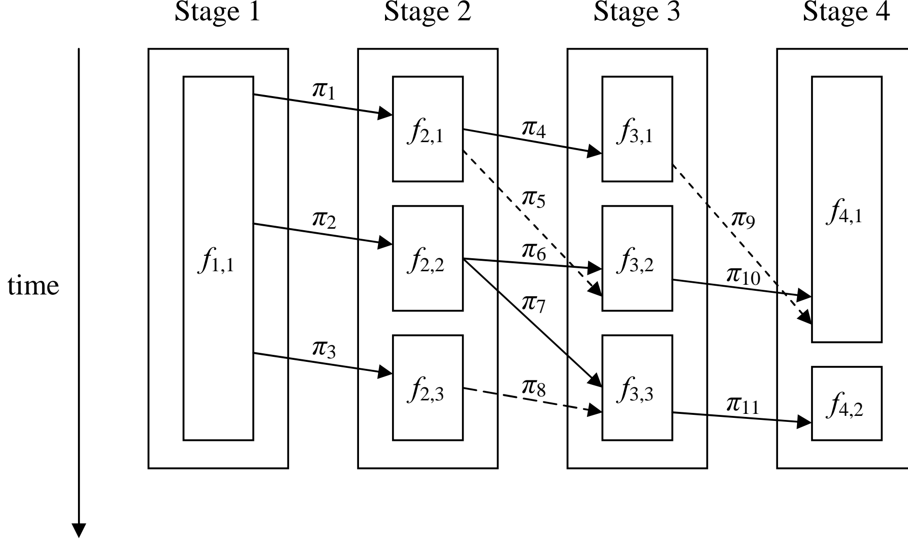


Figure 4.1: Breadth-first concatenation approach with four stages. The long-dashed arrow (π_8) represents a plan that will not be found by the breadth-first concatenation algorithm. The dashed arrows (π_5 and π_9) represent plans that will be found, but are not considered for further expansion.

Consider figure 4.1 for an example of the breadth-first concatenation approach. We start by making a plan from the start resource to the second stage. This results in plan π_1, which ends in free time window $f_{2,1}$. Then, we try again to find a plan from the start resource to the second stage that doesn't make use of free time window $f_{2,1}$. This results in plan π_2, which ends in free time window $f_{2,2}$. We repeat this process until no more plans to the second stage can be found. This results in all possible plans from the first stage to the second stage, which are plans π_1, π_2, and π_3. Next, we take the best (shortest-time) plan to the second stage, which is plan π_1, and try to expand this plan to the third stage. Note that the earliest possible start time we can use to find a plan to the third stage is the time at which plan π_1 enters the second stage. The expansion of plan π_1 results in plans π_4 and π_5. Then, we take the next plan to the second stage, which is plan π_2, and try to expand this plan to the third stage. This results in plans π_6 and π_7. Note that plan π_6 arrives earlier within free time window $f_{3,2}$ than plan π_5, which previously visited the free time window. We repeat this process until each plan to the second stage has been fully expanded. This results in all possible plans from the second stage to the third stage, which are plans π_4, π_5, π_6, π_7, and π_8.

It is not necessary to expand all these plans to the fourth stage. It is sufficient to take only the best partial plans to each free time window on a certain stage into consideration for further expansion. This can simply be explained as follows. If a partial plan π arrives earlier in the same free time window than a partial plan π', then expansion of π' can never lead to a better plan than the expansion of π. Any

plan π'' that results from expanding π' can be simulated using π: the only thing we have to do is wait in the resource in which plan π (and π') ends from the start of the interval of the last plan step of plan π to the start of the interval of the last plan step of plan π', and then follow plan π''. In other words, the set of plans that can be expanded from π' is a subset of the plans that can be expanded from π. Hence, we only need to take plans π_4, π_6, and π_7 into consideration for expansion to the fourth stage. We repeat the process of expansion for the third stage to the fourth stage, which results in plans π_9, π_{10}, and π_{11}. Plan π_{10} is the shortest-time plan to the destination resource. If we now concatenate plans π_2, π_6, and π_{10}, then an optimal multi-stage plan has been found.

We just explained that it is sufficient to take only the best partial plans to each free time window on a certain stage into consideration for further expansion. This means that, when we make a plan from stage i to stage $i+1$, we only want to find a plan that:

1. makes use of a 'non-discovered' free time window on stage $i+1$, or
2. makes use of a previously-visited free time window, but arrives at an earlier time within this free time window.

To accomplish that the single-stage route planning algorithm only finds plans that meet the two requirements mentioned above, we use a copy of the free time windows F_i^{copy} for every stage $i \in \{2, \ldots, m\}$ in the visiting sequence. In the set F_i^{copy} we record the entry time of a visited free time window on stage i. Initially, we set the entry times into all free time windows of F_i^{copy} to infinity. When we make a call to the single-stage route planning algorithm to find a plan from stage i to the next stage $i+1$, we pass the set $F' = F \oplus F_{i+1}^{\text{copy}}$ as an argument. The expression $F' = F \oplus F_{i+1}^{\text{copy}}$ indicates that a set F' of free time windows is created in which the entry times of F_{i+1}^{copy} are added to the corresponding free time windows in F. Note that when making a plan from stage i to stage $i+1$, the single-stage route planning algorithm only takes into account the entry times on stage $i+1$, not the entry times on any other stage j, even if stages $i+1$ and j correspond to the same resource. Furthermore, we add a line of code to the single-stage route planning algorithm: after line 15 of algorithm 2 we check if the entry time into a successor free time window f' is earlier than the recorded entry time into f': $\text{entryTime}(f') < \text{entryTime}(F_{i+1}^{\text{copy}}, f')$. If the entry time into the free time window is earlier, we continue. Otherwise, we discard the free time window, and we continue with the next successor free time window. This way, it is ensured that a plan is found that will not make use of a previously-visited free time window, unless it can reach it at an earlier time.

Take a look at figure 4.1 again. Now, when the breadth-first concatenation algorithm is executed, the single-stage route planning algorithm will not find plan π_8, because plan π_7 already visited free time window $f_{3,3}$ at an earlier time. However, the rest of the plans are found, because they enter a 'non-discovered' free time window (plans π_1, π_2, π_3, π_4, π_5, π_7, π_9, and π_{11}), or arrive earlier within a previously-visited free time window (plans π_6 and π_{10}). Furthermore, plans π_5 and π_9 are discarded,

because these plans are not the best plans possible to free time windows $f_{3,2}$ and $f_{4,1}$ respectively.

The breadth-first concatenation algorithm maintains for every stage $i \in \{2, \ldots, m\}$ in the visiting sequence a list of best partial plans Π_i to this stage i, and in each iteration all plans on the list of partial plans Π_i are expanded to the next stage. Only the best partial plans to each reachable free time window on the next stage are stored, and we continue with the next iteration until we reach the final stage.

Algorithm 5 is a specification of the breadth-first concatenation algorithm. In this algorithm we represent a plan as a tail end X of the plan, and a head consisting of a tuple $\langle f, t \rangle$, where:

- f is the free time window associated with the last visited resource of the plan;
- t is the entry time into f.

In line 2 we make a copy of the free time windows associated with every stage (except for the first stage) in the visiting sequence ϕ. In line 3 we try to find a plan from the start resource r_1 to the second resource of the visiting sequence ϕ with the single-stage route planning algorithm. If no plan is found, then no multi-stage plan exists, and we return nil in line 24. Otherwise, we record the entry time t' into free time window f reached by the found plan in the set F_2^{copy} in line 5. In line 6 we add the found plan to the list of partial plans Π_2 to stage 2. Then, in line 7 we try to find another plan to stage 2. We keep repeating this process until all plans to stage 2 have been found. Then, we will try, as long as the final stage m hasn't been reached, to expand all the partial plans to the current stage to the next stage. In line 11 we select the partial plan with the smallest cost from the list of partial plans Π_i to stage i. Then, the partial plan is removed from the list of partial plans. In line 13 we try to find a plan from the current stage i to the next stage $i+1$ with the single-stage route planning algorithm. Note that the earliest possible start time we can use to find a plan to the next stage is the time t' at which the current stage is entered. If no plan is found, we try to expand the next partial plan on the list of partial plans Π_i. Otherwise, the plan that has been expanded in this iteration is concatenated with the newly found plan in line 15. Then, in line 16 we record the entry time t'' into free time window f' reached by the concatenated plan in the set F_{i+1}^{copy}. In line 17 we check whether a plan π'' already exists on the list of partial plans Π_{i+1} to stage $i+1$ that visits the same free time window as the concatenated plan. Recall that the single-stage route planning algorithm only finds plans that are better than any previously found plans to stage $i+1$, because of the set with recorded entry times F_{i+1}^{copy} into the free time windows on stage $i+1$. Hence, if such a plan π'' exists, we know that the concatenated plan π' is a plan that arrives earlier within a previously-visited free time window, and we replace plan π'' with the concatenated plan π' on the list of partial plans Π_{i+1} to stage $i+1$ in line 18. If the list of partial plans Π_{i+1} to stage $i+1$ doesn't yet contain a plan to free time window f', we add the concatenated plan π' to Π_{i+1} in line 20. Then, in line 21 we try to find a another plan to stage $i+1$. We keep repeating this process until

the current partial plan has been fully expanded to stage $i+1$. Then, we try to expand the next partial plan on the list of partial plans Π_i. We keep repeating this process until all plans to stage $i+1$ have been found. Then, in line 22 the counter i is updated, and we continue with the next stage in the visiting sequence. When the counter becomes equal to m, the final stage has been reached. The list of partial plans Π_m contains all multi-stage plans to the final stage m. If the list of partial plans Π_m is empty, then no multi-stage plan exists, and nil is returned in line 24. Otherwise, the plan in Π_m with the smallest cost is returned in line 27.

Algorithm 5 Multi-Stage Breadth-First Concatenation

Require: visiting sequence $\phi = (r_1, r_2, \ldots, r_m)$, start time t; free time window graph $G_F = (F, E_F)$.
Ensure: shortest-time, conflict-free route plan π, such that $\phi \sqsubseteq \text{resources}(\pi)$.

1: **for all** $r_i \in \{\phi \setminus \{r_1\}\}$ **do**
2: $\quad F_i^{\text{copy}} \leftarrow F_i$
3: $\pi = (\langle f, t' \rangle, X) \leftarrow \text{planRoute}(r_1, r_2, t, (F \oplus F_2^{\text{copy}}, E_F))$
4: **while** $\pi \neq \mathsf{nil}$ **do**
5: $\quad \text{entryTime}(F_2^{\text{copy}}, f) \leftarrow t'$
6: $\quad \text{add}(\pi, \Pi_2)$
7: $\quad \pi = (\langle f, t' \rangle, X) \leftarrow \text{planRoute}(r_1, r_2, t, (F \oplus F_2^{\text{copy}}, E_F))$
8: $i \leftarrow 2$
9: **while** $i < m$ **do**
10: $\quad$ **while** $\Pi_i \neq \varnothing$ **do**
11: $\quad\quad \pi = (\langle f, t' \rangle, X) \leftarrow \text{argmin}_{\pi' \in \Pi_i} c(\pi')$
12: $\quad\quad \text{remove}(\pi, \Pi_i)$
13: $\quad\quad \pi_{(i,i+1)} \leftarrow \text{planRoute}(r_i, r_{i+1}, t', (F \oplus F_{i+1}^{\text{copy}}, E_F))$
14: $\quad\quad$ **while** $\pi_{(i,i+1)} \neq \mathsf{nil}$ **do**
15: $\quad\quad\quad \pi' = (\langle f', t'' \rangle, X) \leftarrow \pi \frown \pi_{(i,i+1)}$
16: $\quad\quad\quad \text{entryTime}(F_{i+1}^{\text{copy}}, f') \leftarrow t''$
17: $\quad\quad\quad$ **if** $\exists \pi'' = (\langle f', _ \rangle, _) \in \Pi_{i+1}$ **then**
18: $\quad\quad\quad\quad \text{replace}(\pi'', \pi', \Pi_{i+1})$
19: $\quad\quad\quad$ **else**
20: $\quad\quad\quad\quad \text{add}(\pi', \Pi_{i+1})$
21: $\quad\quad\quad \pi_{(i,i+1)} \leftarrow \text{planRoute}(r_i, r_{i+1}, t', (F \oplus F_{i+1}^{\text{copy}}, E_F))$
22: $\quad i \leftarrow i + 1$
23: **if** $\Pi_m = \varnothing$ **then**
24: $\quad$ **return** nil
25: **else**
26: $\quad \pi \leftarrow \text{argmin}_{\pi' \in \Pi_m} c(\pi')$
27: $\quad$ **return** π

4.1.1 Correctness

We must prove that if a solution to the multi-stage route planning problem exists, then algorithm 5 finds an optimal solution.

Proposition 4.1.1. *Algorithm 5 returns an optimal solution.*

Proof. The algorithm is complete, because it considers all possible shortest-time partial plans between the free time windows of two successive stages in the visiting sequence. Previously, we explained that it is sufficient to take only the best partial plans to each free time window on a certain stage into consideration for further expansion, because the expansion of a worse partial plan can be simulated using the best plan by simply waiting in the resource. What remains to be proven is the correctness of the replacement of a partial plan on the list of partial plans Π_{i+1} to stage $i+1$ in line 18. In line 17 we check if there is already a plan to free time window f' on the list of partial plans Π_{i+1} to stage $i+1$. If such a plan π'' exists, then it *must* be that this plan has a later entry time into f' than the newly found plan π': the contrary would imply that the entry time of plan π'' into stage $i+1$ (and therefore into free time window f') were earlier than the entry time of plan π' into stage $i+1$. But the single-stage route planning algorithm can only find the plan π' that makes use of free time window f' if the entry time of plan π' into free time window f' is earlier than any previously recorded entry time. Hence, the entry time of plan π'' into free time window f' must be later, and we can safely replace plan π'' with π', since the set of plans that can be expanded from plan π'' is a subset of the plans that can be expanded from plan π'. This way only the best partial plans to the next stage are found, and therefore algorithm 5 is optimal. □

4.1.2 Complexity

The computational complexity of algorithm 5 is determined by the number of calls that are made to the single-stage route planning algorithm.

Proposition 4.1.2. *Algorithm 5 has a run-time complexity of* $O(|\phi||F|^2 \cdot (|E_F|+|F|) \log(|F|))$.

Proof. The list of partial plans Π_i to stage i can contain at most $|F_i|$ plans: one partial plan to each free time window on stage i. Every partial plan in Π_i can be expanded, in the worst case, to every free time window on the next stage $i+1$, which results in $|F_{i+1}|$ calls to the single-stage route planning algorithm. Hence, for every stage at most $|F_i| \times |F_{i+1}|$ calls are made to the single-stage route planning algorithm. There are $|\phi|$ stages, so the run-time complexity of the algorithm is $O(|\phi||F|^2)$ times the complexity of algorithm 2. □

4.1.3 Additional constraints and breadth-first concatenation

If overtaking is forbidden, an earlier entry time into a free time window doesn't necessarily mean an earlier possible exit time out of the free time window, because

of the leading agent's exit time as we have seen in section 2.4.3. To avoid that plans are found that arrive in a free time window at an earlier time than before, but cannot leave it earlier, and thereby avoiding unnecessary replace operations in line 18 of algorithm 5, we check if the partial cost of a successor free time window f' is smaller than the partial cost resulting from the recorded entry time into f': $g(f') < \max(\text{start}(\tau_{\text{exit}}(f')), \text{entryTime}(F_{i+1}^{\text{copy}}, f') + tt(\text{resource}(f')))$ after line 16 of algorithm 2 instead of checking if $\text{entryTime}(f') < \text{entryTime}(F_{i+1}^{\text{copy}}, f')$ after line 15. This way, it is ensured that plans are found that will not make use of a previously-visited free time window, unless they can leave it at an earlier time. Note that the new check is an optimization, and not necessarily required for the correct operation of algorithm 5.

4.2 A* Concatenation Approach

The breadth-first concatenation approach finds a lot of partial plans that are not interesting, i.e., plans that arrive in a free time window at a time that exceeds the cost of the eventually found multi-stage plan. The second complete and optimal multi-stage route planning algorithm we present tries to improve on the breadth-first concatenation algorithm by finding an optimal solution using fewer calls to the single-stage route planning algorithm. The A* concatenation algorithm is based on the A* search algorithm. It uses an estimated cost function $y(\pi)$ to determine the order in which partial plans are selected for expansion. The function $y(\pi)$ is the sum of two functions:

- $g(\pi)$: The partial cost function that calculates the minimal plan cost of partial plan π.
- $h(\pi)$: The heuristic function that estimates the cost of completing π to the final stage in the visiting sequence.

The partial cost $g(\pi)$ for a partial plan π is initially equal to the plan cost of plan π: $g(\pi) = c(\pi)$. The heuristic cost $h(\pi)$ for π is initially equal to the cost of the shortest path from the current stage i to the final stage m along the stages in the visiting sequence ϕ that have yet to be visited without taking into account the presence of other agents: $h(\pi) = \sum_{k=i}^{m-1} \text{shortestPath}(r_k, r_{k+1})$. Note that this heuristic function is consistent.

The A* concatenation algorithm only maintains one list of partial plans Π that can be expanded as opposed to the breadth-first concatenation algorithm that maintains a list of partial plans for every stage in the visiting sequence. In each iteration the partial plan that is most likely to lead to a shortest multi-stage route to the final stage is expanded to the next stage, i.e., the partial plan π with the lowest value of $y(\pi)$. However, a partial plan π is only expanded with one new plan, instead of making plans to all reachable free time windows on the next stage as in the breadth-first concatenation algorithm. This means that a partial plan is only

partially expanded, and needs to be put back on the list of partial plans to guarantee completeness. For every partial plan on the list of partial plans the g, h, and y values, and a stage number are stored. Hence, a partial plan on the list of partial plans can be represented as a 5-tuple $\langle \pi, g, h, y, i \rangle$, where:

- π is the partial plan;
- g is the g value of π;
- h is the h value of π;
- y is the y value of π;
- i is the stage number of π.

Maintaining stage numbers is necessary to keep track of the stage reached by a partial plan (a partial plan to stage i has stage number i). Again, we use the set F_i^{copy} to ensure that the single-stage route planning algorithm only finds plans that will not make use of a previously-visited free time window, unless they can reach it at an earlier time.

Algorithm 6 is a specification of the A* concatenation algorithm. In line 2 we make a copy of the free time windows associated with every stage (except for the first stage) in the visiting sequence ϕ. In line 3 we try to find a start plan that only contains the start resource r_1 of the visiting sequence ϕ with the single-stage route planning algorithm. If no plan is found, then no multi-stage plan exists, and we return nil in line 34. Otherwise, in lines 5 to 8 the g, h, and y values are calculated for the start plan π, and the stage number for π is set to 1. In line 9 the start plan π is added to the list of partial plans Π. Next, we will try, as long as the list of partial plans is not empty, to expand the most promising partial plan in each iteration. In line 11 the partial plan π with the lowest value of $y(\pi)$ is selected from the list of partial plans. Then, the partial plan is removed from the list of partial plans. If the stage number of the selected plan π equals the final stage m, an optimal multi-stage plan has been found, and it is returned in line 15. If the stage number of the selected plan π is not equal to m, we try to find a plan from the current stage i to the next stage $i+1$ with the single-stage route planning algorithm in line 16. Note that the earliest possible start time we can use to find a plan to the next stage is the time t' at which the current stage is entered. If no plan is found, we try to expand the next partial plan on the list of partial plans. Otherwise, the plan that has been expanded in this iteration is concatenated with the newly found plan in line 18. In line 19 we record the entry time t'' into free time window f' reached by the concatenated plan in the set F_{i+1}^{copy}. Then, in lines 20 to 23 the g, h, and y values are calculated for the concatenated plan π', and the stage number for π' is set to $i+1$. In line 24 we check whether a plan already exists on the list of partial plans Π that visits the same free time window and has the same stage number as the concatenated plan. If such a plan π'' exists, we replace the plan with the concatenated plan π' on the list of partial plans Π in line 25. If the list of partial plans Π doesn't yet contain

Algorithm 6 Multi-Stage A* Concatenation

Require: visiting sequence $\phi = (r_1, r_2, \ldots, r_m)$, start time t; free time window graph $G_F = (F, E_F)$.

Ensure: shortest-time, conflict-free route plan π, such that $\phi \sqsubseteq \text{resources}(\pi)$.

1: **for all** $r_i \in \{\phi \setminus \{r_1\}\}$ **do**
2: $\quad F_i^{\text{copy}} \leftarrow F_i$
3: $\pi \leftarrow \text{planRoute}(r_1, r_1, t, G_F)$
4: **if** $\pi \neq \textsf{nil}$ **then**
5: $\quad g(\pi) \leftarrow c(\pi)$
6: $\quad h(\pi) \leftarrow \sum_{k=1}^{m-1} \text{shortestPath}(r_k, r_{k+1})$
7: $\quad y(\pi) \leftarrow g(\pi) + h(\pi)$
8: $\quad \text{stage}(\pi) \leftarrow 1$
9: $\quad \text{add}(\pi, \Pi)$
10: **while** $\Pi \neq \varnothing$ **do**
11: $\quad \pi = (\langle f, t' \rangle, X) \leftarrow \text{argmin}_{\pi' \in \Pi}\, y(\pi')$
12: $\quad \text{remove}(\pi, \Pi)$
13: $\quad i \leftarrow \text{stage}(\pi)$
14: $\quad$ **if** $i = m$ **then**
15: $\quad\quad$ **return** π
16: $\quad \pi_{(i,i+1)} \leftarrow \text{planRoute}(r_i, r_{i+1}, t', (F \oplus F_{i+1}^{\text{copy}}, E_F))$
17: $\quad$ **if** $\pi_{(i,i+1)} \neq \textsf{nil}$ **then**
18: $\quad\quad \pi' = (\langle f', t'' \rangle, X) \leftarrow \pi \frown \pi_{(i,i+1)}$
19: $\quad\quad \text{entryTime}(F_{i+1}^{\text{copy}}, f') \leftarrow t''$
20: $\quad\quad g(\pi') \leftarrow c(\pi')$
21: $\quad\quad h(\pi') \leftarrow \sum_{k=i+1}^{m-1} \text{shortestPath}(r_k, r_{k+1})$
22: $\quad\quad y(\pi') \leftarrow g(\pi') + h(\pi')$
23: $\quad\quad \text{stage}(\pi') \leftarrow i + 1$
24: $\quad\quad$ **if** $\exists \pi''[\pi'' \in \Pi \mid \pi'' = (\langle f', _ \rangle, _) \wedge \text{stage}(\pi'') = \text{stage}(\pi')]$ **then**
25: $\quad\quad\quad \text{replace}(\pi'', \pi', \Pi)$
26: $\quad\quad$ **else**
27: $\quad\quad\quad \text{add}(\pi', \Pi)$
28: $\quad\quad f'' \leftarrow \text{nextFreeTimeWindow}(f')$
29: $\quad\quad$ **if** $f'' \neq \textsf{nil}$ **then**
30: $\quad\quad\quad g(\pi) \leftarrow \text{start}(\tau_{\text{exit}}(f''))$
31: $\quad\quad\quad h(\pi) \leftarrow h(\pi')$
32: $\quad\quad\quad y(\pi) \leftarrow g(\pi) + h(\pi)$
33: $\quad\quad\quad \text{add}(\pi, \Pi)$
34: **return** nil

a plan to free time window f', we add the concatenated plan π' to Π in line 27. Then, in line 28 we get the free time window succeeding the free time window f' that is reached by the concatenated plan π'. If there isn't such a window f'', then

plan π has been fully expanded, and we don't need to put plan π back on the list of partial plans. Otherwise, the g value for the plan π that has been expanded in this iteration is set to the start of the exit window of f'', the h value for π is set to the h value of π', and the y value for π is updated accordingly. In line 33 plan π is put back on the list of partial plans. Then, we try to expand the next partial plan on the list of partial plans Π in the next iteration. Finally, if the list of partial plans Π is empty before a multi-stage plan is found, then no multi-stage plan exists, and nil is returned in line 34.

Finally, note that if overtaking is forbidden, the check for the single-stage route planning algorithm described in section 4.1.3 can also be applied when using the A* concatenation algorithm.

4.2.1 Correctness

We must prove that if a solution to the multi-stage route planning problem exists, then algorithm 6 finds an optimal solution.

Proposition 4.2.1. *Algorithm 6 returns an optimal solution.*

Proof. Algorithm 6 is complete and optimal for the same reason that a standard A* algorithm is: in each iteration the most promising plan is expanded, and it is not possible that expansion of a plan π results in a plan π' such that $y(\pi') < y(\pi)$. What remains to be proven is the correctness of:

1. the replacement of a partial plan on the list of partial plans in line 25;
2. putting back a partial plan that has been partially expanded on the list of partial plans, and the associated calculation of the new g, h, and y values in lines 30 to 33.

Ad 1: In the proof of correctness of the breadth-first concatenation algorithm, we proved the correctness of the replacement of a partial plan on the list of partial plans to a certain stage. The same proof applies to line 25 of algorithm 6.

Ad 2: In each iteration a partial plan π is only partially expanded. To guarantee completeness a partial plan needs to be put back on the list of partial plans after it has been expanded. In line 28 we determine the free time window that succeeds the free time window f' reached by the concatenated plan π'. If such a window f'' exists, we first update the g, h, and y values for plan π before we put the plan back on the list of partial plans to avoid expanding the same plan in the next iteration. The next time we expand π, we only can find a plan π'' that is more expensive than π', because no expansion of plan π can make use of a free time window earlier than f', or f' itself, because π' is an optimal plan from π to the next stage $i+1$. The plan cost of plan π'' becomes at least equal to the start of the exit window of free time window f'', because π cannot reach the next stage earlier than the start of the next free time window f'', and therefore π'' cannot leave the next stage earlier than the start of the exit window of the next free time window f''. Therefore, we

set the minimal plan cost $g(\pi)$ to the start of the exit window of f'' in line 30. In line 31 the heuristic cost $h(\pi)$ is set to $h(\pi')$, because the minimal plan cost $g(\pi)$ is determined in such a way that it is like plan π has already been expanded again to the next stage. This way the new value of $y(\pi)$ does not *overestimate* the cost of completing π to the final stage, which is required for optimality. If there isn't a free time window that succeeds the free time window f' reached by the concatenated plan π', then plan π has been fully expanded, and we don't need to put plan π back on the list of partial plans.

We conclude that algorithm 6 is correct, and because of the 'A* property', it is optimal. □

4.2.2 Complexity

The computational complexity of algorithm 6 is determined by the number of calls that are made to the single-stage route planning algorithm.

Proposition 4.2.2. *Algorithm 6 has a run-time complexity of* $O(|\phi||F|^2 \cdot (|E_F| + |F|) \log(|F|))$.

Proof. The list of partial plans Π can contain at most $|F_i|$ plans to stage i: one partial plan to each free time window on stage i. Every partial plan that ends in some free time window on stage i can be expanded, in the worst case, to every free time window on the next stage $i + 1$, which results in $|F_{i+1}|$ calls to the single-stage route planning algorithm. Hence, for every stage at most $|F_i| \times |F_{i+1}|$ calls are made to the single-stage route planning algorithm. There are $|\phi|$ stages, so the run-time complexity of the algorithm is $O(|\phi||F|^2)$ times the complexity of algorithm 2. □

Note that, although the A* concatenation algorithm is an improvement of the breadth-first concatenation algorithm, they have the same worst-case run-time complexity. Finally, we have to make a side note that concerns both the breadth-first concatenation algorithm and the A* concatenation algorithm. The complexity factor of $O(|F|^2)$ is caused by the fact that, sometimes, a plan π to stage i that is expanded later than a plan π' to stage i leads to a shorter plan to stage $i + 1$. However, to actually use $O(|F|^2)$ calls to the single-stage route planning algorithm, it must occur $O(|F|)$ times that such a plan π results in an earlier plan to stage $i + 1$, which seems unlikely. It is therefore an open question whether a better complexity bound for both the breadth-first concatenation algorithm and the A* concatenation algorithm can be found, or whether indeed pathological examples exist where $O(|F|^2)$ calls to algorithm 2 are required.

4.3 A* Multi-Layer Approach

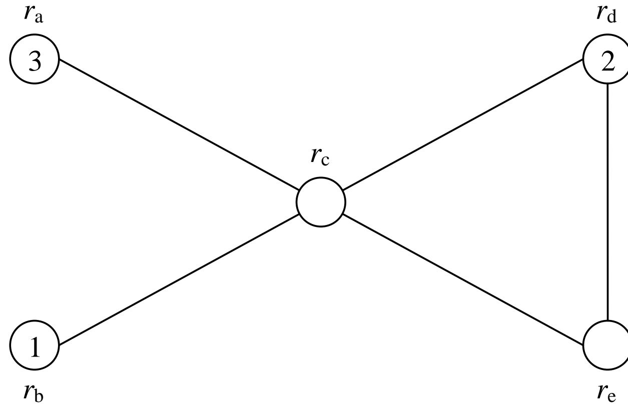
In this section we will explain an algorithm that doesn't make use of the concatenation of plans found by the single-stage route planning algorithm like the previous two algorithms explained in this chapter. When the concatenation approaches make

a plan from a certain stage to the next stage, they probably find a plan that is similar to some of the previously found plans, i.e., many of the visited resources and their free time windows will be the same. The problem with the concatenation approaches is that a plan is calculated from scratch every time a call to the single-stage route planning algorithm is made. The only information that is used from previously found plans are the entry times in the free time windows on the next stage. The rest of the information from previous plans isn't used in the planning process. The A* multi-layer algorithm tries to overcome this problem by expanding free time windows the same way as is done in the single-stage route planning algorithm.

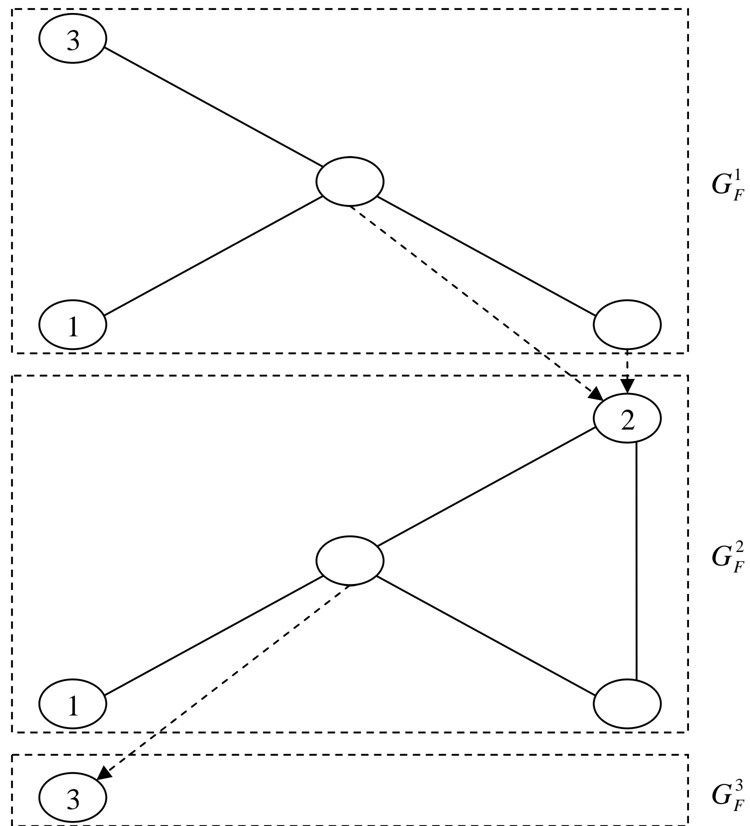
In the single-stage route planning algorithm, a partial plan is completely characterized by the last free time window and the associated entry time: together they determine how the partial plan can be expanded. In multi-stage route planning, we should also know which resources of the visiting sequence have been reached yet. For example, if a partial plan ends in the final resource of the sequence, but not all other stages have been visited yet, then we should continue expanding the plan. This also means that we may have to visit a free time window more than once. Take a look at the infrastructure in figure 4.2(a), where we have a visiting sequence $\phi = (r_b, r_d, r_a)$. Any valid multi-stage plan will have to pass intersection r_c at least twice. In case there are no reservations in the system yet, then each resource has a single free time window, and we need to allow more than one visit to the free time window on resource r_c. However, as we have seen in chapter 2, the single-stage route planning algorithm only expands a free time window at most once, which results in a plan that visits a free time window at most once.

The solution we present in this section is to create an augmented free time window graph $G_F^\alpha = (F^\alpha, E_F^\alpha)$ that consists of multiple layers. We construct the augmented free time window graph using the original free time window graph G_F and the visiting sequence ϕ. For every stage in the visiting sequence we make a separate free time window subgraph, and these subgraphs are connected to each other. The vertices (i.e., the augmented free time windows) of subgraph G_F^i for stage $i \in \{1, \ldots, m-1\}$ consist of all free time windows in the original free time window graph G_F, except for the free time windows of the next stage $i+1$. The vertices of subgraph G_F^m for the last stage m only consist of the free time windows of stage m. The edges of subgraph G_F^i for stage $i \in \{1, \ldots, m-1\}$ are the same as in the original free time window graph G_F, except for the augmented free time windows in G_F^i corresponding to the free time windows in G_F that can reach the free time windows on stage $i+1$. Those augmented free time windows are connected to the augmented free time windows on stage $i+1$ in subgraph G_F^{i+1}. We connect the subgraph for stage i to the subgraph for stage $i+1$ in only one direction, meaning that once we reach subgraph G_F^{i+1}, we cannot reach the subgraphs for stages i or smaller anymore.

Consider figure 4.2 for an example of the creation of the augmented free time window graph. There is an infrastructure with five intersection resources, and a visiting sequence consisting of three stages, which are marked 1, 2, and 3. We can see that subgraph G_F^1 doesn't contain any free time windows of stage 2, G_F^2



(a) An infrastructure graph with five intersection resources. For the sake of simplicity the lane resources (i.e., the edges) have a zero travel time. The visiting sequence is set to (r_b, r_d, r_a).



(b) The associated augmented free time window graph. Ovals represent augmented free time windows on the corresponding resources of the infrastructure graph. Dashed arrows represent the reachability between the free time window subgraphs.

Figure 4.2: A visiting sequence of size three results in a free time window graph with three layers.

doesn't contain any free time windows of stage 3, and G_F^3 only contains the free time windows of stage 3. Furthermore, we can see how the free time window subgraphs are connected to each other.

The A* multi-layer algorithm, which is similar to the single-stage route planning algorithm, performs a search through the augmented free time window graph G_F^α to find an optimal multi-stage plan. If we expand the augmented free time windows of the augmented free time window graph, we always know which stages in the visiting sequence we have already visited: when we reach a certain subgraph G_F^i, we know that we have visited the first i stages in the sequence. Furthermore, it is sufficient to expand an augmented free time window of a certain subgraph at most once, because a multi-stage plan needs to visit the corresponding free time window in G_F at most once when going from stage to stage.

Each augmented free time window of the augmented free time window graph G_F^α belongs to a certain subgraph G_F^i as we have seen above. Therefore, we represent an augmented free time window as a pair $\langle f, i \rangle$, where:

- f is the free time window that corresponds to free time window f in the original free time window graph G_F;
- i is the stage number of f.

The A* multi-layer algorithm uses an estimated cost function $y(\langle f, i \rangle)$ to determine the order in which augmented free time windows are selected for expansion. The function $y(\langle f, i \rangle)$ is the sum of two functions:

- $g(\langle f, i \rangle)$: The partial cost function that calculates the earliest possible exit time out of augmented free time window $\langle f, i \rangle$.
- $h(\langle f, i \rangle)$: The heuristic function that estimates the cost from the current resource associated with augmented free time window $\langle f, i \rangle$ to the final stage in the visiting sequence.

Note that the partial cost function is the same as in the single-stage route planning algorithm: $g(\langle f, i \rangle) = \text{entryTime}(\langle f, i \rangle) + tt(r)$. The heuristic function $h(\langle f, i \rangle)$ calculates the cost of the shortest path from the current resource r to the next stage $i+1$ plus the cost of the shortest path from the next stage $i+1$ to the final stage m along the stages in the visiting sequence ϕ that have yet to be visited without taking into account the presence of other agents:

$$h(\langle f, i \rangle) = \text{shortestPath}(r, r_{i+1}) + \sum_{k=i+1}^{m-1} \text{shortestPath}(r_k, r_{k+1})$$

Note that this heuristic function is consistent.

To determine the successors of an augmented free time window $\langle f, i \rangle$, we first have to define the augmented reachability relation E_F^α. Augmented free time windows with stage number i can only reach (and be reached from) other augmented free time windows with stage number i. The exception is for augmented free time windows on

'stage' resources: augmented free time window f with stage number i is connected to f' with stage number $i+1$ if $(f, f') \in E_F$, and the resource associated with f' is stage $i+1$. Therefore, we define the augmented reachability relation E_F^α as follows, given a visiting sequence $\phi = (r_1, r_2, \ldots, r_m)$:

$$E_F^\alpha = \{(\langle f, i\rangle, \langle f', i\rangle) \mid (f, f') \in E_F \wedge \text{resource}(f') \neq r_{i+1}\} \cup \{(\langle f, i\rangle, \langle f', i+1\rangle) \mid (f, f') \in E_F \wedge \text{resource}(f') = r_{i+1}\}$$

As in the single-stage case, the existence of a pair $(\langle f, i\rangle, \langle f', i'\rangle) \in E_F^\alpha$ does not guarantee that augmented free time window f with stage number i can be expanded to augmented free time window f' with stage number i'. To determine if $\langle f', i'\rangle \in \text{successors}(\langle f, i\rangle)$ with $\langle f, i\rangle = [t_1, t_2)$, we have to check if $\langle f', i'\rangle$ is reachable from $\langle f, i\rangle$ from the earliest possible exit time $g(\langle f, i\rangle)$ out of $\langle f, i\rangle$: $[g(\langle f, i\rangle), t_2) \cap \tau_{\text{entry}}(\langle f', i'\rangle) \neq \varnothing$.

The A* multi-layer algorithm maintains an open list of augmented free time windows that can be expanded, and in each iteration the augmented free time window that is most likely to lead to a shortest multi-stage route to the final stage is expanded, i.e., the augmented free time window $\langle f, i\rangle$ with the lowest value of $y(\langle f, i\rangle)$. For every augmented free time window on the open list the entry time t, the g, h, and y values, and a pointer to its predecessor window are stored. Hence, an augmented free time window on the open list can be represented as a 6-tuple $\langle\langle f, i\rangle, t, g, h, y, p\rangle$, where:

- $\langle f, i\rangle$ is the augmented free time window;
- t is the entry time into $\langle f, i\rangle$;
- g is the g value of $\langle f, i\rangle$;
- h is the h value of $\langle f, i\rangle$;
- y is the y value of $\langle f, i\rangle$;
- p is a pointer to the augmented free time window's predecessor.

Algorithm 7 is a specification of the algorithm. In line 1 the augmented free time window graph is created on basis of the original free time window graph G_F and the visiting sequence ϕ. In line 2 we check if there exists an augmented free time window with stage number 1 on the start resource r_1 of the visiting sequence ϕ that has an entry window that contains the start time t. If there isn't such an augmented free time window $\langle f, 1\rangle$, then no plan exists, and nil is returned in line 28. Otherwise, in line 3 the entry time into $\langle f, 1\rangle$ is recorded as the start time t. Then, in lines 4 to 7 the g, h, and y values are calculated for the start window $\langle f, 1\rangle$, and the pointer is set to nil. In line 8 the augmented free time window $\langle f, 1\rangle$ is added to the open list. Next, as long as the open list is not empty, in each iteration the most promising augmented free time window is expanded. In line 10 the augmented free time window $\langle f, i\rangle$ with the lowest value of $y(\langle f, i\rangle)$ is selected from the open

Algorithm 7 Multi-Stage A* Multi-Layer

Require: visiting sequence $\phi = (r_1, r_2, \ldots, r_m)$, start time t; free time window graph $G_F = (F, E_F)$.

Ensure: shortest-time, conflict-free route plan π, such that $\phi \sqsubseteq \text{resources}(\pi)$.

1: $G_F^\alpha = (F^\alpha, E_F^\alpha) \leftarrow \text{generateFTWG}(G_F, \phi)$
2: **if** $\exists \langle f, 1\rangle \, [\langle f, 1\rangle \in F^\alpha \mid t \in \tau_{\text{entry}}(\langle f, 1\rangle) \wedge r_1 = \text{resource}(\langle f, 1\rangle)]$ **then**
3: $\quad \text{entryTime}(\langle f, 1\rangle) \leftarrow t$
4: $\quad g(\langle f, 1\rangle) \leftarrow t + tt(r_1)$
5: $\quad h(\langle f, 1\rangle) \leftarrow \sum_{k=1}^{m-1} \text{shortestPath}(r_k, r_{k+1})$
6: $\quad y(\langle f, 1\rangle) \leftarrow g(\langle f, 1\rangle) + h(\langle f, 1\rangle)$
7: $\quad \text{pointer}(\langle f, 1\rangle) \leftarrow \mathsf{nil}$
8: $\quad \text{add}(\langle f, 1\rangle, \mathsf{open})$
9: **while** $\mathsf{open} \neq \varnothing$ **do**
10: $\quad \langle f, i\rangle \leftarrow \text{argmin}_{\langle f', i'\rangle \in \mathsf{open}} \, y(\langle f', i'\rangle)$
11: $\quad \text{remove}(\langle f, i\rangle, \mathsf{open})$
12: $\quad \text{add}(\langle f, i\rangle, \mathsf{closed})$
13: $\quad$ **if** $i = m$ **then**
14: $\quad\quad$ **return** $\text{followPointers}(\langle f, i\rangle)$
15: $\quad$ **for all** $\langle f', i'\rangle \in \{\text{successors}(\langle f, i\rangle) \setminus \mathsf{closed}\}$ **do**
16: $\quad\quad \text{entryTime}(\langle f', i'\rangle) \leftarrow \max(g(\langle f, i\rangle), \text{start}(\langle f', i'\rangle))$
17: $\quad\quad g(\langle f', i'\rangle) \leftarrow \text{entryTime}(\langle f', i'\rangle) + tt(\text{resource}(\langle f', i'\rangle))$
18: $\quad\quad h_1(\langle f', i'\rangle) \leftarrow \text{shortestPath}(\text{resource}(\langle f', i'\rangle), r_{i+1})$
19: $\quad\quad h_2(\langle f', i'\rangle) \leftarrow \sum_{k=i+1}^{m-1} \text{shortestPath}(r_k, r_{k+1})$
20: $\quad\quad h(\langle f', i'\rangle) \leftarrow h_1(\langle f', i'\rangle) + h_2(\langle f', i'\rangle)$
21: $\quad\quad y(\langle f', i'\rangle) \leftarrow g(\langle f', i'\rangle) + h(\langle f', i'\rangle)$
22: $\quad\quad \text{pointer}(\langle f', i'\rangle) \leftarrow \langle f, i\rangle$
23: $\quad\quad$ **if** $\exists \langle f'', i''\rangle [\langle f'', i''\rangle \in \mathsf{open} \mid f'' = f' \wedge i'' = i']$ **then**
24: $\quad\quad\quad$ **if** $\text{entryTime}(\langle f'', i''\rangle) > \text{entryTime}(\langle f', i'\rangle)$ **then**
25: $\quad\quad\quad\quad \text{replace}(\langle f'', i''\rangle, \langle f', i'\rangle, \mathsf{open})$
26: $\quad\quad$ **else**
27: $\quad\quad\quad \text{add}(\langle f', i'\rangle, \mathsf{open})$
28: **return** nil

list. Then, the augmented free time window is removed from the open list, and added to the closed list. If the stage number i of the selected augmented free time window equals the final stage m, an optimal multi-stage plan has been found. The optimal plan is constructed in line 14. Following the series of pointers from the current augmented free time window $\langle f, i\rangle$ backwards to the start window results in a sequence of augmented free time windows in G_F^α. This sequence can be translated into a sequence of free time windows in the original free time window graph G_F to give the optimal plan. Note that the cost of the plan as defined in definition 2.2.2 is equal to $g(\langle f, i\rangle)$. If the stage number of the selected augmented free time window

is not equal to m, the selected window $\langle f, i\rangle$ is expanded to all successor windows that are not on the closed list. Recall, as described above, that an augmented free time window $\langle f', i'\rangle$ is a successor of the selected window $\langle f, i\rangle$ if $\langle f', i'\rangle$ is reachable from $\langle f, i\rangle$ from the earliest possible exit time $g(\langle f, i\rangle)$ out of window $\langle f, i\rangle$. In line 16 the entry time into successor window $\langle f', i'\rangle$ is determined by the maximum of the earliest possible exit time out of $\langle f, i\rangle$, and the start of window $\langle f', i'\rangle$. Then, in lines 17 to 22 the g, h, and y values are calculated for successor $\langle f', i'\rangle$, and the pointer is set to the selected window $\langle f, i\rangle$. Note that the first part h_1 of the heuristic function h is equal to zero when $i' = i + 1$. In line 23 it is checked if the successor window is already on the open list. If it is, we check in line 24 if the entry time into the augmented free time window on the open list is later than the newly found entry time into the window. If the entry time into the augmented free time window on the open list is later, the window on the open list is replaced in line 25. If the open list doesn't contain the successor window, the augmented free time window is added in line 27. Then, the next augmented free time window on the open list is expanded in the next iteration. Finally, if the open list is empty before a multi-stage plan is found, then no multi-stage plan exists, and nil is returned in line 28.

Finally, note that if overtaking is forbidden, we need to change the partial cost function as described in section 2.4.3.

4.3.1 Correctness

We must prove that if a solution to the multi-stage route planning problem exists, then algorithm 7 finds an optimal solution. We begin with a lemma.

Lemma 4.3.1. *For any $1 \leq i \leq m$ it holds that when subgraph G_F^i is reached, the first i resources of the visiting sequence ϕ have been visited.*

Proof. We will prove this by induction. We start the path on the first stage in subgraph G_F^1, so the proposition holds for $i = 1$. Suppose now that for some $i \geq 1$, we have visited the first i stages, when we reach subgraph G_F^i. The only way to reach the next subgraph G_F^{i+1} is via subgraph G_F^i by the structure of the augmented free time window graph: we reach the next subgraph when we visit the next stage $i + 1$. Hence, when we reach subgraph G_F^{i+1}, we know that we have visited the first $i + 1$ stages. We now have proved by induction that for any $1 \leq i \leq m$ it holds that when we reach subgraph G_F^i, we have visited the first i resources of the visiting sequence ϕ. □

Proposition 4.3.2. *Algorithm 7 returns an optimal solution.*

Proof. Algorithm 7 searches for a path through the augmented free time window graph G_F^α, starting on the first stage in subgraph G_F^1 and ending in subgraph G_F^m. In order to prove the correctness of algorithm 7 we have to prove that:

1. such a path through the augmented free time window graph G_F^α is indeed a multi-stage plan;

2. such a path is a *shortest* path through the augmented free time window graph G_F^α.

Ad 1: Two conditions have to be satisfied for a path through the augmented free time window graph G_F^α, starting on the first stage in subgraph G_F^1 and ending in subgraph G_F^m, to be a multi-stage plan:

1. *The path should visit all stages in the visiting sequence*
   Lemma 4.3.1 implies that when we reach subgraph G_F^m, we have visited all resources of the visiting sequence ϕ. Hence, a path starting on the first stage in subgraph G_F^1 and ending in subgraph G_F^m visits all stages in the visiting sequence.

2. *It should be possible to translate the path through G_F^α into a path through G_F*
   Each augmented free time window corresponds to a certain free time window in the original free time window graph G_F, so a sequence of augmented free time windows can be translated into a sequence of free time windows in the original free time window graph G_F.

Hence, we can conclude that a path through the augmented free time window graph G_F^α, starting on the first stage in subgraph G_F^1 and ending in subgraph G_F^m, is a multi-stage plan.

Ad 2: We will prove that algorithm 7 always returns an optimal path through the augmented free time window graph G_F^α (if one exists) by showing that prior to termination, there is always an augmented free time window on the open list that is on an optimal path. Because any sub-optimal (partial or full) path has a higher y-value than a (partial or full) optimal path, the optimal augmented free time window on the open list will be expanded before a sub-optimal solution can be returned. Expansion of an optimal partial path will result in another optimal (partial or full) path.

1. *Path p^* is an optimal path through G_F^α*
   Let p^* be an optimal path through the augmented free time window graph G_F^α, starting from time t on the first stage in subgraph G_F^1 and ending in subgraph G_F^m. We can characterize p^* as a sequence of n augmented free time windows with optimal entry times: $p^* = (\langle\langle f_1, 1\rangle, t_1^*\rangle, \ldots, \langle\langle f_n, m\rangle, t_n^*\rangle)$, where $\langle f_1, 1\rangle$ is an augmented free time window on the first stage, t_1^* equals the start time t, and $\langle f_n, m\rangle$ is an augmented free time window on the last stage m. Note that the stage numbers of the augmented free time windows in the sequence are non-decreasing. We will show that, prior to termination of the algorithm, there exists an augmented free time window on the open list $\langle f_w, j\rangle \in p^*$ with optimal entry time t_w^*[1].

[1] Showing the existence of an augmented free time window on the open list that is on an optimal path is similar to lemma 1 in Hart et al. [6].

2. *There exist optimal windows on the closed list*
   Let Δ be the set of all augmented free time windows from the optimal path p^* that are on the closed list, such that for all $\langle f_v, i_v \rangle \in \Delta$, $t_v = \text{entryTime}(\langle f_v, i_v \rangle)$ is optimal. Note that Δ is non-empty, since after the first iteration it contains the start window $\langle f_1, 1 \rangle$ with entry time t.

3. *Optimal window $\langle f_{w-1}, i \rangle$ on the closed list has been expanded to window $\langle f_w, j \rangle$ on the open list that is on optimal path p^**
   Let $\langle f_{w-1}, i \rangle$ be the augmented free time window from Δ with the highest index. Since $\langle f_{w-1}, i \rangle$ is on the closed list, it must have been expanded to $\langle f_w, j \rangle$, unless $\langle f_w, j \rangle$ were already on the closed list. Because we use a consistent heuristic function h, prior expansion of $\langle f_w, j \rangle$ is impossible, which we will now demonstrate. If $\langle f_w, j \rangle$ were on the closed list, it would, by definition of Δ, have a sub-optimal entry time $t' > t^*_w$. Although j can be either equal to i or $i + 1$, in both cases we have:

$$\begin{aligned} y(\langle f_{w-1}, i \rangle) &= t^*_{w-1} + tt(\text{resource}(\langle f_{w-1}, i \rangle)) + h(\langle f_{w-1}, i \rangle) \\ &\leq t^*_w + tt(\text{resource}(\langle f_w, j \rangle)) + h(\langle f_w, j \rangle) \text{ (consistency)} \\ &< t' + tt(\text{resource}(\langle f_w, j \rangle)) + h(\langle f_w, j \rangle) \end{aligned}$$

   Hence, $\langle f_{w-1}, i \rangle$ would be expanded before $\langle f_w, j \rangle$, so $\langle f_w, j \rangle$ cannot be on the closed list.

4. *Window $\langle f_w, j \rangle$ on optimal path p^* has been entered with optimal entry time t^*_w*
   We will now show that the expansion of $\langle f_{w-1}, i \rangle$ to $\langle f_w, j \rangle$ has resulted in the optimal entry time t^*_w into $\langle f_w, j \rangle$. The entry time of an augmented free time window is determined in line 16 of algorithm 7; there are two cases to consider:

   case a: $\text{entryTime}(\langle f_w, j \rangle)$ equals the start of the augmented free time window $\langle f_w, j \rangle$. Since a window cannot be entered earlier than its start time, $\text{entryTime}(\langle f_w, j \rangle)$ equals the optimal entry time t^*_w.

   case b: $\text{entryTime}(\langle f_w, j \rangle) = t^*_{w-1} + tt(\text{resource}(\langle f_{w-1}, i \rangle))$. Since t^*_{w-1} is optimal, so is $\text{entryTime}(\langle f_w, j \rangle)$.

Having shown that there exists an augmented free time window on the open list with the optimal entry time, it is easy to show that algorithm 7 can never return a sub-optimal solution. As explained in section 2.4.1, a consistent heuristic is also admissible, i.e., it never overestimates the cost of reaching the destination. This implies that at the final stage, we have $h = 0$. A sub-optimal path to the final stage therefore has a larger y-value than any (partial or full) optimal path; an optimal path will thus be retrieved from the open list first.

Finally, note that algorithm 7 always terminates: there is only a finite number of augmented free time windows, and in each iteration, one augmented free time

window is added to the closed list (i.e., it can never be expanded again). Hence, if a solution exists, algorithm 7 always returns an optimal solution. □

4.3.2 Complexity

Proposition 4.3.3. *Algorithm 7 has a run-time complexity of* $O(|\phi| \cdot (|E_F| + |F|) \log(|F|))$.

Proof. First of all, note that we need to expand each augmented free time window at most once, because we use a consistent heuristic function (see section 2.4.1). Therefore, an augmented free time window $\langle f, i \rangle \in F^\alpha$ can be retrieved from the open list at most once, so the while loop runs for at most $|F^\alpha|$ iterations. If we implement the open list as a priority queue, removing the cheapest element from the list takes $O(\log(|F^\alpha|))$ time.

The for loop in line 15 could inspect every connection between two augmented free time windows exactly once, so lines 16 to 27 can run at most $|E_F^\alpha|$ times. Within the for loop, line 23 checks whether the successor augmented free time window is already on the open list. If it is, and its entry time is later, we replace the window in line 25. Otherwise, the window is added in line 27. Replacing or adding a window on the open list takes $O(\log(|F^\alpha|))$ time. This results in a run-time complexity of $O((|E_F^\alpha| + |F^\alpha|) \log(|F^\alpha|))$.

Furthermore, there are $|\phi| \cdot |F|$ augmented free time windows, i.e., $|F^\alpha| = |\phi| \cdot |F|$, and the augmented reachability relation E_F^α is around $|\phi|$ times as large as the reachability relation E_F, i.e., $|E_F^\alpha| = |\phi| \cdot |E_F|$. Hence, the run-time complexity of algorithm 7 becomes $O(|\phi| \cdot (|E_F| + |F|) \log(|F|))$. □

4.4 Concluding Remarks

In this chapter we presented three complete and optimal multi-stage route planning algorithms. The breadth-first concatenation algorithm and the A* concatenation algorithm both make use of the single-stage route planning algorithm. They concatenate plans between stages of the visiting sequence to find an optimal multi-stage plan. Although the A* concatenation algorithm is an improvement of the breadth-first concatenation algorithm, they both have a worst-case run-time complexity of $O(|\phi||F|^2 \cdot (|E_F| + |F|) \log(|F|))$. The A* multi-layer algorithm doesn't make use of the single-stage route planning algorithm. A new free time window graph is created on basis of the original free time window graph and the visiting sequence. The algorithm searches for a path through this new free time window graph to find an optimal multi-stage plan the same way as the single-stage route planning algorithm searches for a path through the free time window graph. The A* multi-layer algorithm is the algorithm with the best run-time complexity. It has a complexity of $O(|\phi| \cdot (|E_F| + |F|) \log(|F|))$. Note that this is the same complexity as the complexity of the incomplete and sub-optimal linear concatenation algorithm from chapter 3.

In the next chapter we will empirically investigate the algorithms presented in this chapter. We will examine the required CPU-time, and how this CPU-time depends on the size of the visiting sequence. Additionally, we will compare the CPU-time of the multi-stage algorithms with the CPU-time of the single-stage route planning algorithm. We expect that the A* multi-layer algorithm will be the fastest algorithm among the three complete and optimal multi-stage route planning algorithms. Furthermore, we expect the A* concatenation algorithm to be faster than the breadth-first concatenation algorithm.

Chapter 5

Experiments

In chapter 3 we explained that the linear concatenation approach is incomplete and sub-optimal. The need for a complete and optimal multi-stage route planning algorithm is determined by the failure rate and plan quality of the linear concatenation approach. Therefore, in this chapter we will empirically investigate (*i*) how often the linear concatenation algorithm fails to find a solution, (*ii*) how often it finds a sub-optimal solution, and (*iii*) when it finds a sub-optimal solution, what is the plan quality, i.e., how much more expensive is the sub-optimal solution than an optimal plan, found by one of the optimal algorithms. We will also investigate the CPU-time required by the three complete and optimal multi-stage route planning algorithms from the previous chapter, and how the CPU-time depends on the length of the visiting sequence. We will start by describing our test setup for the experiments in section 5.1. Then, in section 5.2, we will describe the results.

5.1 Test Procedure

We tested the algorithms on two different infrastructures. The first infrastructure is a random graph consisting of 180 lanes and 100 intersections (i.e., 280 resources)[1]. For each agent a randomly generated visiting sequence ϕ was determined. The size of the visiting sequence $|\phi|$ was set to 4, 6, and 8.

To investigate how multi-stage route planning performs in a real-life application, we looked at the de-icing problem at an airport as mentioned in the introduction of this thesis. Our infrastructure is a model of Amsterdam Airport Schiphol, which consists of 1219 resources. For each airplane (agent), the visiting sequence consisted of (*i*) one of six available runways, (*ii*) one of 187 gates, (*iii*) one of two de-icing stations, located at the center of the airport, and (*iv*) one of five remaining runways (i.e., different from the arrival runway).

We used a total of 900 agents to perform the experiments. The first agent to make a plan has no reservations to take into account, whereas agent 900 has to

[1] We have actually performed the experiments on several random graphs both smaller and larger, which showed similar results.

respect the reservations of the previous 899 agents (we chose a random order in which to let the agents plan). For each agent, we ran four algorithms: the linear concatenation algorithm (algorithm 4), the breadth-first concatenation algorithm (algorithm 5), the A* concatenation algorithm (algorithm 6), and the A* multi-layer algorithm (algorithm 7). We reserved the plan that was found by the A* multi-layer algorithm. All of the following experiments have been repeated 100 times with different (randomly chosen) visiting sequences, and were performed on an AMD Opteron 250 2.4 Ghz with 4 GB of memory.

5.2 Results

In the first part of this section we will discuss the results regarding the linear concatenation approach. The second part discusses the results concerning the execution times of the three complete and optimal multi-stage route planning algorithms from chapter 4.

5.2.1 The linear concatenation algorithm and its plan quality

Figures 5.1 through 5.4 show the percentage of plans produced by the linear concatenation algorithm that are optimal (line with circles), sub-optimal (line with triangles), and the percentage of null plans (i.e., the percentage of runs that the linear concatenation algorithm finds no plan — line with squares). Note that to obtain the percentage of runs in which the complete and optimal multi-stage route planning algorithms outperform the linear concatenation algorithm, we need to add the percentage of sub-optimal plans to the percentage of null plans.

The first conclusion we can draw from figures 5.1 through 5.4 is that the percentage of null plans not only increases with the number of reservations in the system, but also with the length of the visiting sequence. Although it is true that 200 agents produce more reservations when they have a plan along 8 resources rather than 4, we also see that 80% or more null plans is never reached when $|\phi| \leq 6$, even for 900 agents, whereas it is already reached for around 200 agents when $|\phi| = 8$. Also, for all sizes of visiting sequence we see that the percentage of null plans increases quickly at first, but then it starts to level out. The exception is the Schiphol infrastructure, where the levelling of the percentage of null plans was not (yet?) observed for 900 agents. The behaviour of the null-plan lines leads us to believe that there is a small probability of failure at each intermediate resource of the visiting sequence, and that this probability increases with the number of reservations in the system, up to a point where no more reservations can be made for a certain time period.

For all sizes of visiting sequence, and for both Schiphol and the random graph, we see that the percentage of sub-optimal plans stays relatively constant. For $|\phi| \geq 6$, the percentage is higher when the percentage of null plans is still small. The quality of the sub-optimal plans is quite high, however, as shown in table 5.1. The differences in plan cost between the sub-optimal plans and the optimal plans are small indeed for the Schiphol infrastructure, and we suspect there are two reasons for this. First

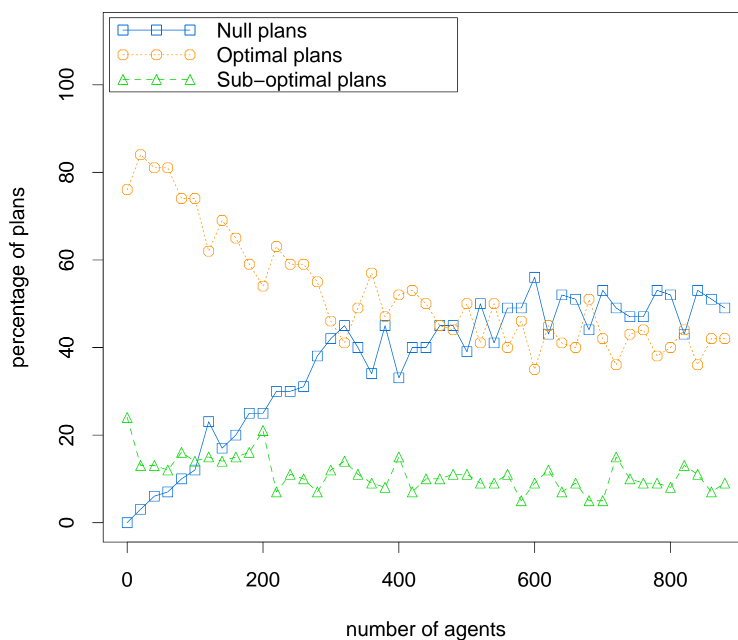


Figure 5.1: Type of plans found by the linear concatenation algorithm with visiting sequence size 4 for an increasing number of agents on the random graph.

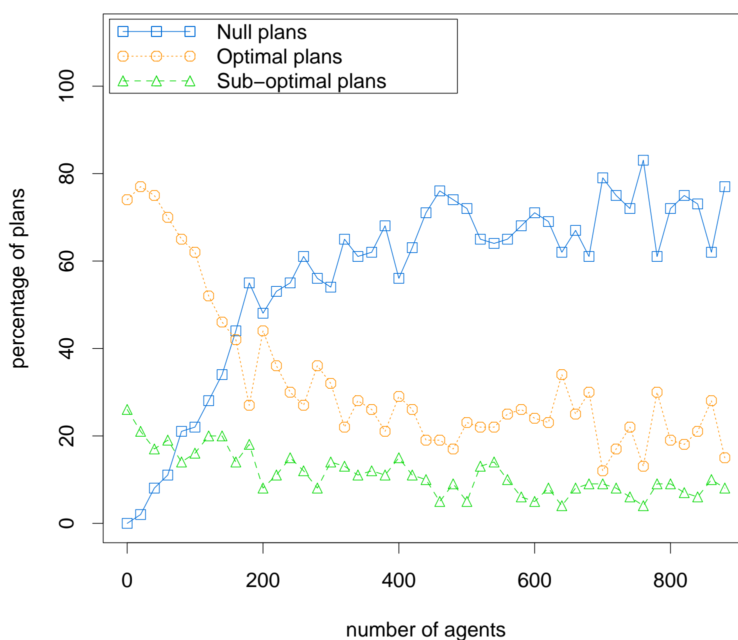


Figure 5.2: Type of plans found by the linear concatenation algorithm with visiting sequence size 6 for an increasing number of agents on the random graph.

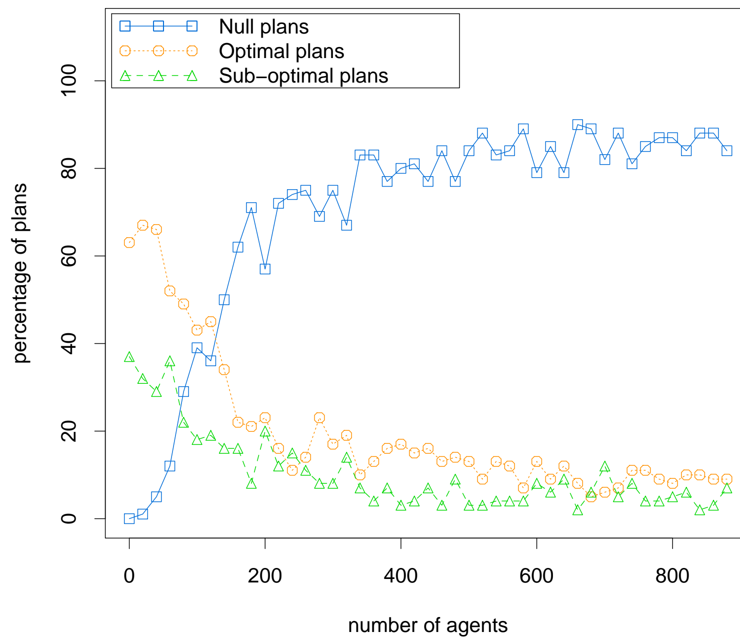


Figure 5.3: Type of plans found by the linear concatenation algorithm with visiting sequence size 8 for an increasing number of agents on the random graph.

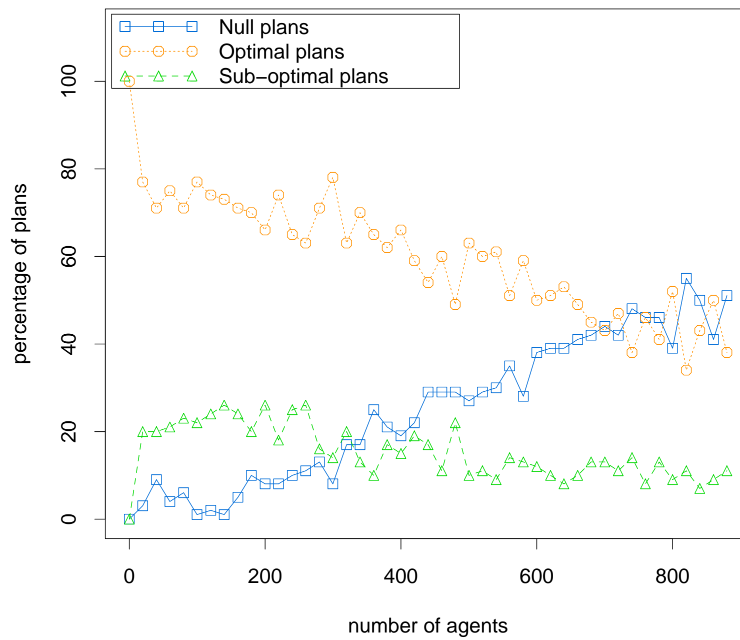


Figure 5.4: Type of plans found by the linear concatenation algorithm with visiting sequence size 4 for an increasing number of agents on the Schiphol infrastructure.

of all, the distances between stages on the Schiphol infrastructure are larger, so the loss of quality is divided by a greater total plan cost. Second, and perhaps more importantly, the final stage in any plan (a departure runway) is shared by many agents, and will therefore become a bottleneck resource. Hence, any time lost after the penultimate stage (the de-icing station) can be made up because the agent has to wait for entry into the runway. A final conclusion is that the linear concatenation approach suffers more from incompleteness than from sub-optimality.

Table 5.1: Plan cost of sub-optimal plans found by the linear concatenation algorithm.

infrastructure	$\lvert\phi\rvert$	plan cost compared to optimum
random graph	4	102.83%
random graph	6	102.35%
random graph	8	102.11%
Schiphol	4	100.14%

5.2.2 CPU-time analysis of the multi-stage route planning algorithms

We expect that the A* multi-layer algorithm will be the fastest multi-stage algorithm, because this algorithm has the best computational complexity as we have seen in the previous chapter. The breadth-first concatenation algorithm and the A* concatenation algorithm have the same complexity, but since the A* concatenation algorithm is an improvement of the breadth-first concatenation algorithm, we expect that the A* concatenation algorithm is faster than the breadth-first concatenation algorithm.

We started with testing the single-stage route planning algorithm to see how this algorithm performs. This way we will be able to compare the execution times of the multi-stage algorithms with the execution time of the single-stage route planning algorithm. Figure 5.5 shows the average CPU-time required by the single-stage route planning algorithm along with 95% confidence intervals. Note that the CPU-time is measured in milliseconds. We see that the single-stage algorithm is very fast. Computing a plan only takes a couple of milliseconds. The required CPU-time increases slowly with the number of agents in the system; when almost all agents have reserved their plans, the time to make a plan is almost three times as large as the time it takes to find a plan for one of the first agents. However, the confidence intervals are very large, which means that there is a lot of variation in execution times. Note that this experiment has been performed on the random graph, but we can report that the same test on the Schiphol infrastructure showed similar results.

Figure 5.6 through 5.8 show the average CPU-time for each algorithm along with 95% confidence intervals for the several sizes of the visiting sequence on the

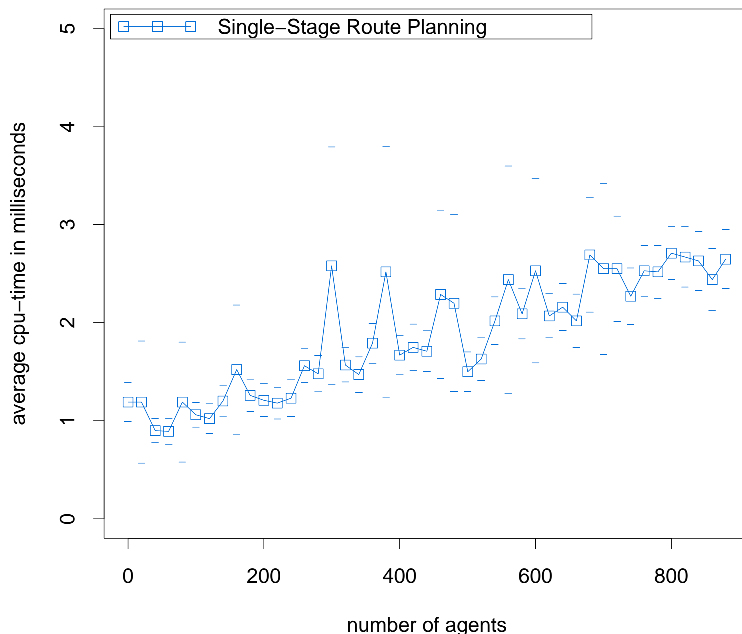


Figure 5.5: Average CPU-time for the single-stage route planning algorithm for an increasing number of agents on the random graph.

random graph. We didn't plot the CPU-time required by the linear concatenation algorithm, but we can report that, when a plan was found, the linear concatenation algorithm required the same amount of CPU-time as the A* multi-layer algorithm. Note that the CPU-time of the breadth-first concatenation algorithm and the A* concatenation algorithm is measured in seconds, but the CPU-time of the A* multi-layer algorithm in milliseconds. The confidence intervals grow both with the number of agents, but also with the size of the visiting sequence. This means that for larger $|\phi|$ there is more variation in execution times. We can see that the relation between the CPU-time and the number of agents is more or less linear for all algorithms. Note that the worst-case complexity of algorithm 5 and algorithm 6, which would require $O(|F|^2)$ calls to algorithm 2 (for a total complexity that is at least quadratic), is not observed. This means that in practice it is rare that $O(|F|^2)$ calls to the single-stage route planning algorithm are required. Furthermore, we see that the relation between the CPU-time and the length of the visiting sequence is more than linear for all algorithms; doubling the size of the visiting sequence causes the execution time to increase more than twice. This can be explained by the factor $|F|\log(|F|)$ present in the complexity of each algorithm combined with the fact that with a longer visiting sequence an agent produces more reservations, and thus there are more free time windows.

Figure 5.9 shows the average CPU-time for the multi-stage algorithms for visiting sequence size 6 on the random graph. In this figure we can see how the three multi-stage algorithms perform relative to each other. It demonstrates that the A* multi-

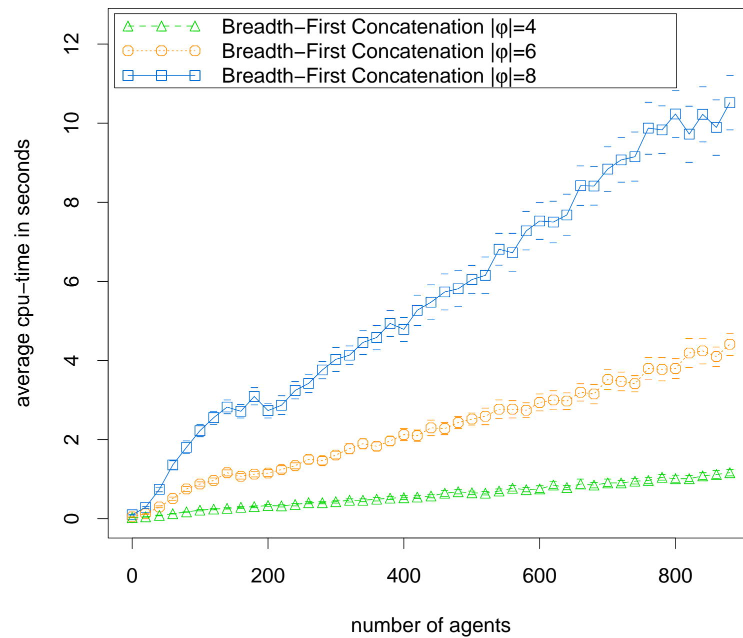


Figure 5.6: Average CPU-time for the breadth-first concatenation algorithm for several visiting sequence sizes for an increasing number of agents on the random graph.

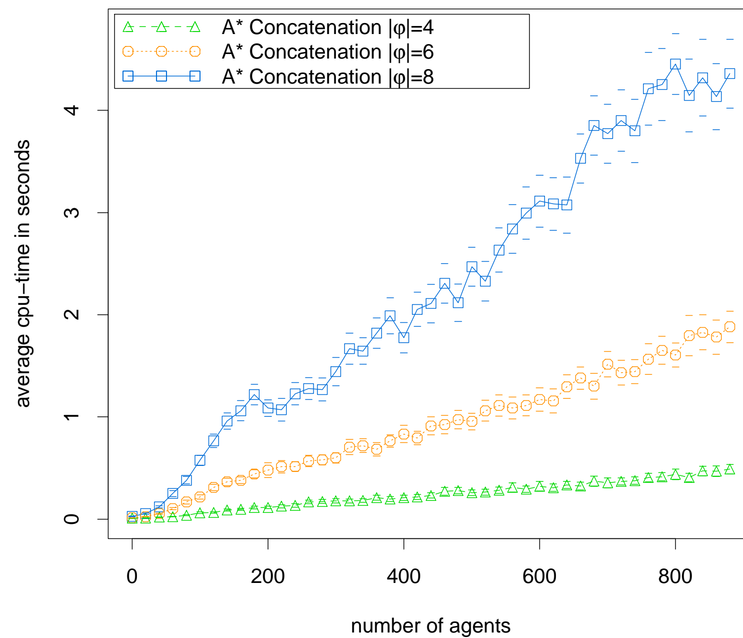


Figure 5.7: Average CPU-time for the A* concatenation algorithm for several visiting sequence sizes for an increasing number of agents on the random graph.

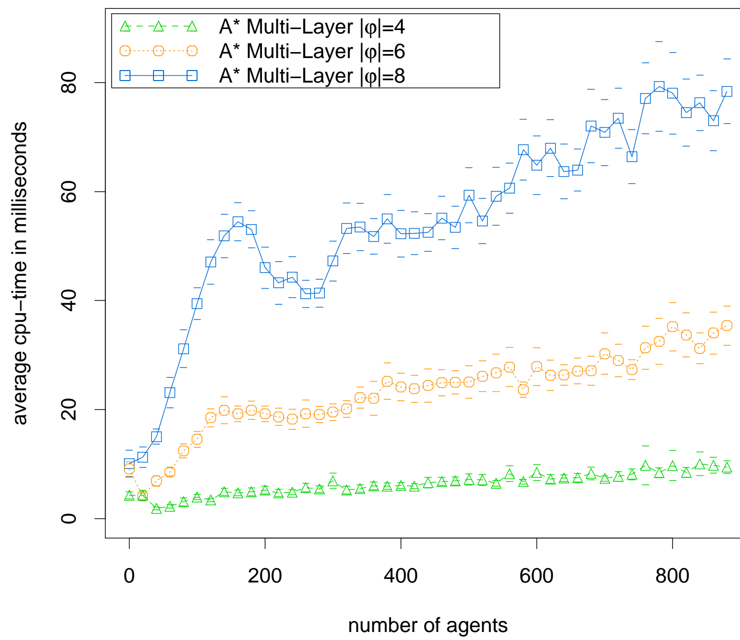


Figure 5.8: Average CPU-time for the A* multi-layer algorithm for several visiting sequence sizes for an increasing number of agents on the random graph.

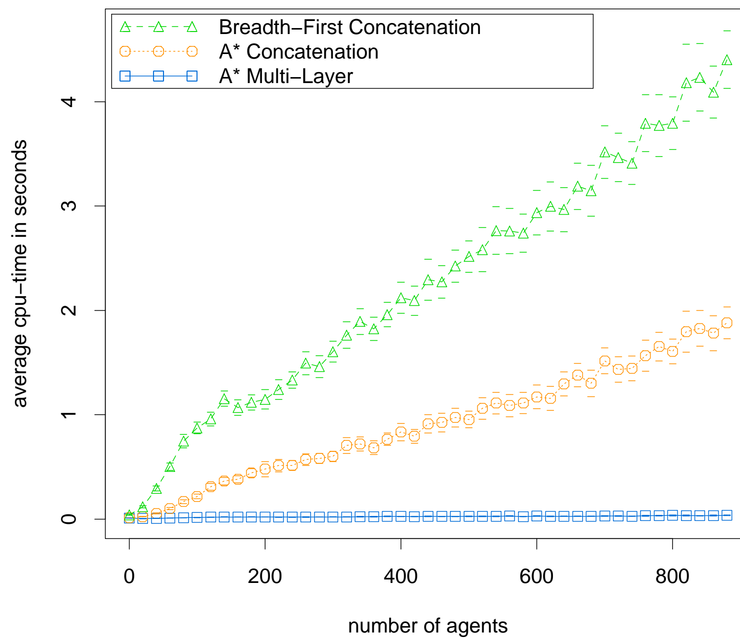


Figure 5.9: Average CPU-time for the multi-stage algorithms with visiting sequence size 6 for an increasing number of agents on the random graph.

layer algorithm is by far the fastest algorithm; the A* concatenation algorithm is about three times faster than the breadth-first concatenation algorithm.

Figure 5.10 shows the average CPU-time for the multi-stage algorithms on the Schiphol infrastructure. Figure 5.10 looks the same as figure 5.9, although the execution times of the A* concatenation algorithm and the breadth-first concatenation algorithm are closer to each other than on the random graph. We also see that each algorithm considerably takes more time to find a plan on the Schiphol infrastructure than on the random graph with visiting sequence size 6, even though the visiting sequence size on the Schiphol infrastructure is just 4. The reason for this is that the Schiphol infrastructure consists of many more resources than the random graph. Hence, a lot more possible route plans need to be considered by the multi-stage algorithms. In figure 5.10 we cannot clearly see the performance of the A* multi-layer algorithm. Therefore, we included figure 5.11 that only shows the average CPU-time for the A* multi-layer algorithm on the Schiphol infrastructure.

5.3 Concluding Remarks

In this chapter we have seen that the linear concatenation algorithm really is no option to use for multi-stage route planning, because its failure rate is too high. As expected, the A* multi-layer algorithm is the fastest multi-stage algorithm among the three complete and optimal multi-stage route planning algorithms from the previous chapter, followed by the A* concatenation algorithm, and then the breadth-first concatenation algorithm. The difference in execution time between the A* multi-layer algorithm and the other two algorithms is quite large. The CPU-time required by the A* multi-layer algorithm is at most a few tenths of a second, while the other two algorithms sometimes need several seconds to find a plan depending on the size of the visiting sequence and the number of agents that already reserved a plan. We conclude by stating that the execution time of the A* multi-layer algorithm is a factor larger than the execution time of the single-stage route planning algorithm, but it still lies in the range of milliseconds.

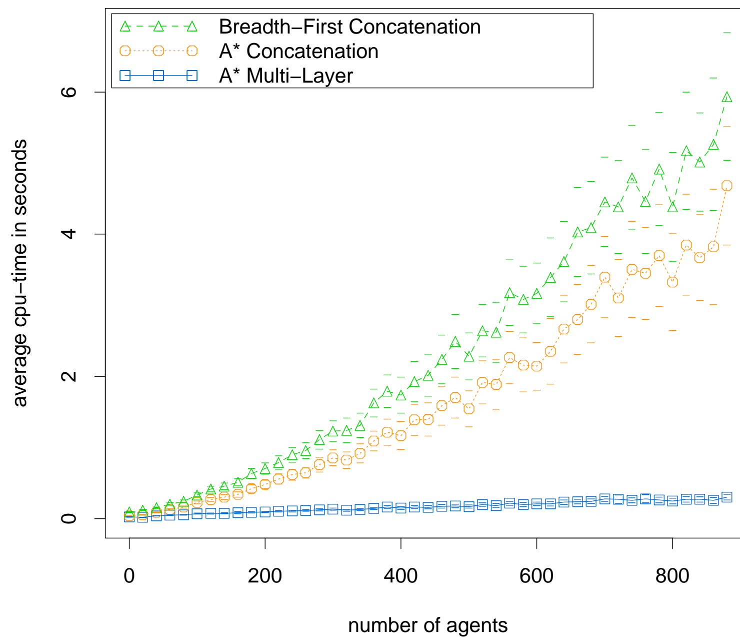


Figure 5.10: Average CPU-time for the multi-stage algorithms with visiting sequence size 4 for an increasing number of agents on the Schiphol infrastructure.

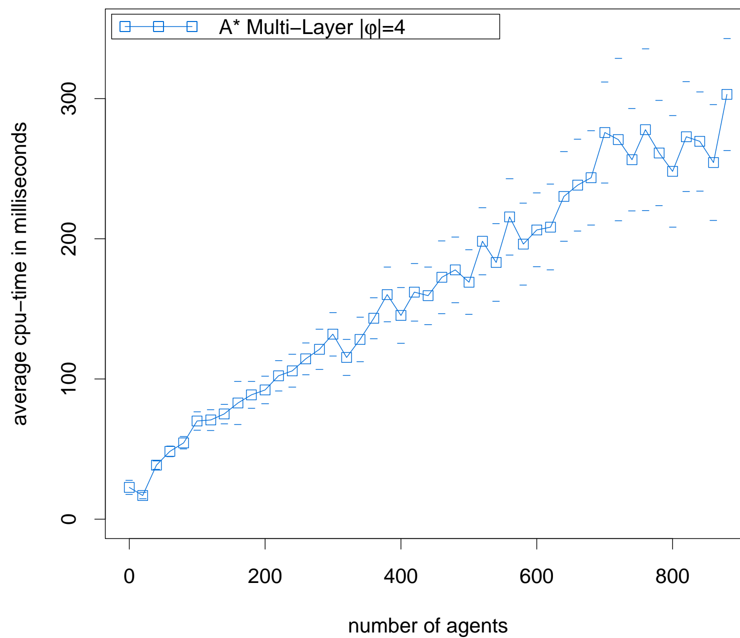


Figure 5.11: Average CPU-time for the A* multi-layer algorithm with visiting sequence size 4 for an increasing number of agents on the Schiphol infrastructure.

Chapter 6

Conclusions and Future Work

This final chapter gives a summary of the matter discussed in the previous chapters. Then, some interesting ideas for future research will be discussed.

6.1 Conclusions

In this thesis we did research on context-aware multi-stage route planning. In general, context-aware route planning is about finding a collectively optimal set of conflict-free plans for agents on a common infrastructure. However, this problem is NP-hard as proved in [13], but if a sequential approach is used, which means that agents make plans one after the other, finding an optimal conflict-free route plan for a single agent can be done in polynomial time. Therefore, we focused on this sequential approach. The idea for route planning is that an agent makes a plan, and then it places reservations on the resources for the intended periods of occupation. During planning, an agent is only allowed to use time intervals that do not conflict with the set of existing reservations on the resources: the *free time windows*. In the multi-stage variant of the context-aware route planning problem, a shortest route has to be found along a sequence of resources instead of just from a start resource to a destination resource as in single-stage route planning.

First of all, we identified the multi-stage route planning problem. It appeared that the solution to the problem is not as trivial as it first seems; concatenating subsequent plans found by the single-stage route planning algorithm turned out to be an incomplete and sub-optimal approach as a result of the reservations in the system. Therefore, our main goal was to find a complete and optimal multi-stage route planning algorithm. We developed three multi-stage route planning algorithms that always return an optimal route for a single agent, given a set of reservations made by previous agents. The A* multi-layer algorithm is the algorithm with the best run-time complexity. In fact, it has the same complexity as the trivial linear concatenation algorithm, which is incomplete and sub-optimal. The experiments concerning the execution times of the algorithms showed that the A* multi-layer algorithm really outperforms the other two complete and optimal multi-

stage algorithms. The CPU-time required by the A* multi-layer algorithm is at most a few tenths of a second as opposed to the other two multi-stage algorithms, which sometimes require several seconds to find a plan depending on the size of the visiting sequence and the number of agents that already reserved a plan. The CPU-time of the A* multi-layer algorithm turns out to be a factor larger than the CPU-time of the single-stage route planning algorithm, but it is still in the range of milliseconds. The experiments regarding the linear concatenation algorithm showed that this algorithm is no option to use for multi-stage route planning, because its failure rate is too high. Furthermore, it appeared that the A* multi-layer algorithm is as fast as the linear concatenation algorithm in case the linear concatenation algorithm does find a multi-stage plan. We conclude that our research has been successful, because we developed an efficient complete and optimal multi-stage algorithm that has the same complexity and requires the same amount of CPU-time as the trivial approach, which is incomplete and sub-optimal.

6.2 Future work

In this section we briefly mention some promising topics for future research.

In chapter 2 we discussed several plan constraints in addition to the basic resource load constraint that a set of agent plans may also be required to satisfy. For example, we may require that agents are not allowed to overtake each other on a resource; or, a lane resource that allows bidirectional traffic may only be traversed in one direction at the same time. It might be interesting to examine other plan constraints that are required by certain problems. Take for example the de-icing problem at airports. We didn't take into account the constraint that an airplane must take off within a certain time limit of de-icing (called the *holdover time*, which is typically 15 minutes), to prevent ice from re-forming. If a plan is found that exceeds the holdover time, then the plan is basically worthless. To solve this problem effectively, we first have to examine the reason for exceeding the holdover time. The reason could be that agents wait too long in some 'bottleneck' resources and thereby block the road for other agents. Another reason can be e.g. that some parts of the infrastructure are simply too congested. In the first case, a possible solution might be to identify the bottleneck resources. Then, in the planning phase we can make sure that an agent only makes use of a bottleneck resource for a limited time by e.g. shortening the duration of the free time windows on that resource. In the second case, a possible solution might be to plan with a heuristic that distributes agents more uniformly over the infrastructure, so that the infrastructure becomes less congested. In both these cases the individual plan quality of the agents may deteriorate slightly in favor of meeting the holdover time. There could be even other reasons to consider, and maybe there are other types of solutions possible, so this is an interesting direction for future research.

In this thesis we used a sequential approach to route planning, meaning that agents make plans one after the other. This means that the set of reservations on

the resources is fully known to a planning agent. However, this sequential approach is restrictive for two reasons: (*i*) in a multi-agent system each agent has its own thread of control, and (*ii*) agents have always to respect the reservations of other agents completely, which limits their autonomy. An algorithm that allows agents to make plans simultaneously, takes the first objection directly away, and paves the way for a solution to the objection of autonomy. We can think of a scenario in which agents have preferences about which agent uses a resource, and when, and that both making and revising reservations is subject to negotiation between the agents. This can be an interesting topic for future work.

As explained in chapter 2, the sequential approach to route planning finds an optimal plan for each individual agent, but it does not guarantee a globally optimal solution for all route planning agents together. The total performance of the resulting route plans will be dependent on the exact sequence in which the agents will plan their individual route. In [13] there has been investigated what the effect of different sequences of planning agents is on the total performance of the set of route plans. They found that the impact of the order in which the agents plan is influenced by a number of factors, such as the density of the infrastructure. An interesting subject for future research is to find heuristic functions to determine efficient planning sequences. Their experiments showed that agents lose much time waiting for agents traveling in the opposite direction, so a heuristic function could be aimed at creating *flows* of agents traveling in the same direction. Note that planning sequences might also be interesting to the de-icing problem with holdover constraint discussed above.

Bibliography

[1] James F. Allen. Maintaining knowledge about temporal intervals. *Commun. ACM*, 26(11):832–843, 1983.

[2] David Applegate, Robert Bixby, Vasek Chvatal, and William Cook. On the solution of traveling salesman problems. In *Documenta Mathematica Journal der Deutschen Mathematiker-Vereinigung, International Congress of Mathematicians*, pages 645–656, 1998.

[3] A. J. Broadbent, C. B. Besant, S. K. Premi, and S. P. Walker. Free ranging AGV systems: promises, problems, and pathways. In *2nd International Conference on Automated Materials Handling*, pages 221–237, 1985.

[4] Rina Dechter and Judea Pearl. Generalized best-first search strategies and the optimality of A*. *Journal of the ACM*, 32:505–536, 1985.

[5] Robert W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6), 1962.

[6] 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 SSC4*, (2):100–107, 1968.

[7] Wolfgang Hatzack and Bernhard Nebel. The operational traffic problem: Computational complexity and solutions. In A. Cesta, editor, *Proceedings of the 6th European Conference on Planning (ECP'01)*, pages 49–60, 2001.

[8] Chang W. Kim and J.M.A. Tanchoco. Conflict-free shortest-time bidirectional AGV routeing. *International Journal of Production Research*, 29(1):2377–2391, 1991.

[9] Kap Hwan Kim, Su Min Jeon, and Kwang Ryel Ryu. Deadlock prevention for automated guided vehicles in automated container terminals. *OR Spectrum*, 28(4):659–679, October 2006.

[10] Tuan Le-Anh and M.B.M De Koster. A review of design and control of automated guided vehicle systems. *European Journal of Operational Research*, 171(1):1–23, May 2006.

[11] Matthias Lehmann, Martin Grunow, and Hans-Otto Günther. Deadlock handling for real-time control of AGVs at automated container terminals. *OR Spectrum*, 28(4):631–657, October 2006.

[12] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, second edition, 2003.

[13] Adriaan ter Mors. *The world according to MARP*. PhD thesis, Technische Universiteit Delft, 2010.

[14] Adriaan W. ter Mors, Jonne Zutt, and Cees Witteveen. Context-aware logistic routing and scheduling. In *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling*, pages 328–335, 2007.

[15] Jeroen van Belle. Traffic management for agents on infrastructures with limited-capacity resources: A literature survey, 2009.

[16] Iris F.A. Vis. Survey of research in the design and control of automated guided vehicle systems. *European Journal of Operational Research*, 170(3):677–709, May 2006.

[17] N. Viswanadham, Y. Narahari, and Timothy L. Johnson. Deadlock prevention and deadlock avoidance in flexible manufacturing systems using Petri net models. *IEEE Transactions on Robotics and Automation*, 6(6), December 1990.