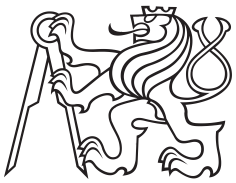
Master Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Comparison of path planning methods for a multi-robot team

Bc. Jakub Hvězda

Supervisor: RNDr. Miroslav Kulich, Ph.D.
May 2017

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Computer Science

DIPLOMA THESIS AGREEMENT

Student: Hvězda Jakub

Study programme: Open Informatics
Specialisation: Artificial Intelligence

Title of Diploma Thesis: Comparison of path planning methods for a multi-robot team

Guidelines:

1. Get acquainted with current approaches to collision-free path planning for a team of cooperating robots.
2. Choose most promising methods and implement them. The selection should be made mainly with respect to computational complexity of methods or/and quality of generated solutions.
3. Design and create a set of testing scenarios.
5. Experimentally evaluate and compare the implemented algorithms. Describe and discuss obtained results.
6. Discuss applicability of the particular algorithms to real-world problems.

Bibliography/Sources:

[1] W. Wang and W. B. Goh. A stochastic algorithm for makespan minimized multi-agent path planning in discrete space. Appl. Soft Comput. 30, C, May 2015, 287-304.
[2] Peasgood, M.; Clark, C.M.; McPhee, J. A Complete and Scalable Strategy for Coordinating Multiple Robots Within Roadmaps, in Robotics, IEEE Transactions on , vol.24, no.2, pp.283-292, April 2008
[3] A. W. ter Mors, "Conflict-free route planning in dynamic environments," 2011 IEEE/RSJ International Conference on Intelligent Robots and Systems, San Francisco, CA, 2011, pp. 2166-2171.
[4] Guni Sharon, Roni Stern, Meir Goldenberg, Ariel Felner, The increasing cost tree search for optimal multi-agent pathfinding, Artificial Intelligence, Volume 195, February 2013, Pages 470-495, ISSN 0004-3702,
[5] G. Wagner, Minsu Kang and H. Choset, "Probabilistic path planning for multiple robots with subdimensional expansion," 2012 IEEE International Conference on Robotics and Automation, Saint Paul, MN, 2012, pp. 2886-2892.
[6] K. Solovey, O. Salzman, O. and D. Halperin, Finding a needle in an exponential haystack: Discrete RRT for exploration of implicit roadmaps in multi-robot motion planning. In Algorithmic Foundations of Robotics XI (pp. 591-607). Springer International Publishing.

Diploma Thesis Supervisor: RNDr. Miroslav Kulich, Ph.D.

Valid until the end of the summer semester of academic year 2017/2018

L.S.

prof. Dr. Michal Pěchouček, MSc.
Head of Department

prof. Ing. Pavel Ripka,CSc.
Dean

Prague, February 23, 2017

Acknowledgements

I would like to thank my supervisor RNDr. Miroslav Kulich, Ph.D. for his guidance and valuable advice.

My thanks also goes to my family for all their support.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, date .....

.....
signature

Abstract

This master thesis discusses the topic of multi-agent pathplanning. For this reason several algorithms were picked and described in the first part of this thesis. All algorithms were implemented in C++ and from experience from working with these algorithms several modifications and improvements were proposed and implemented. The second part of the thesis elaborates on the results of experiments performed on the basic versions of the algorithms as well as the improvements and discusses their effect. This part discusses the potential applications the algorithms as well. All algorithms were tested on the map of robotic warehouse as well as grid maps from pc games.

Keywords: multi-robot pathfinding, multi-agent, algorithm comparison

Supervisor: RNDr. Miroslav Kulich, Ph.D.
Czech Institute of Informatics, Robotics, and Cybernetics, Zikova street 1903/4,166 36 Prague 6

Abstrakt

Tato diplomová práce se zabývá tématem multi-agentního plánování. Za tímto účelem bylo vybráno několik algoritmů které byly popsány v první části této práce. Všechny algoritmy byly poté implementovány v C++ a ze zkušeností získaných prací na těchto algoritmech byly navrženy a implementovány změny a vylepšení jejichž účelem bylo vylepšit vlastnosti algoritmů. Druhá část práce se zabývá výsledky získanými z experimentů na implementovaných algoritmech stejně tak jako jejich vylepšeních. Tato část práce také probírá potenciální využití těchto algoritmů v reálném světě. Všechny algoritmy byly testovány na mapách robotického skladiště spolu s čtvercovými mapami z počítačových her.

Klíčová slova: hledání cest pro tým robotů, multi-agentní, porovnání algoritmů

Překlad názvu: Porovnání metod plánovaní cest pro tým robotů

Contents

Chapter 1

Introduction

In today's age several fields of the industry that deal with coordination of multiple entities such as airports are faced with situations where traffic is higher than the actual capacity. This leads to reliance on path optimizations to increase their throughput. Another example might be a robotic warehouse with several robots trying to retrieve the desired object from the warehouse in as short time as possible. For these reasons the field of multi-agent pathfinding is extensively studied field with many different approaches to the topic.

This thesis focuses on the comparisson of several pathfinding algorithms for multi robot coordination, implementing them and performing modifications to them to improve their overall performance. The thesis is structured into several chapters.

Chapter 2 introduces the addressed pathfinding problem and discusses several approaches to this problem. It also introduces necessary terminology that is used in the following chapters. The end of this chapter is dedicated to discussion about the algorithms that were chosen to be implemented.

Chapter 3 then introduces decoupled algorithm introduced by A. W. ter Mors in [12][11] that uses a variation of A* algorithm to perform search through a free time-window graph to find a solution for one agent at a time using the other agents as obstacles moving in time.

Chapter 4 is dedicated to the coupled optimal algorithm introduced by G. Sharon et al. in [9]. The main idea of this algorithm is to split the pathfinding process into two main searches. The high-level one searches the space of combinations of costs of each individual agents while the low-level search takes the high-level costs as constraints on the lengths of paths for each individual agent and tries to find solution that satisfies these constraints.

In Chapter 5 the decoupled algorithm introduced by M. Peasgood et al. [6] is presented whose main idea is to plan agents on a spanning tree of a given graph and to divide the pathfinding into a number of phases. Several improvements are proposed to this algorithm in this section such as different methods of concurrent plan construction, implementation of loop removal from plans and different choices of leaf node in the first phase of the algorithm.

Chapter 6 introduces the coupled sampling algorithm that is a modification of a well known RRT algorithm [4] that works on explicitly given graphs. This thesis also proposes a number of modifications to this algorithm as

well as a novel approach that is inspired by the RRT* algorithm[2]. Among the modifications are different sampling methods for the random sample generation used in expansion phase of this algorithm, implementing version of this algorithm that grows two trees instead of one whose goal is to reduce the number of iterations required to find a solution or improving the local connector by implementing a modification that reduces the time cost of generated path. Because this algorithm is concerned about finding any solution as fast as possible the quality of found solution is often far from optimum. The goal of the novel approach proposed in this thesis makes changes to the algorithm to produce plans that are much closer to optimum.

Chapter 7 contains experiments with the implemented algorithms. Algorithms are evaluated on publicly available maps as well as a supplied robotic warehouse map.

Chapter 8 contains the summary of this thesis along with the possible future work.

Chapter 2

Pathfinding problem

The pathfinding problem is a problem of finding a path between two vertices on a graph. It is an extensively researched topic in the field of AI. Its use varies from traffic routing, GPS navigation, robot routing to solving combinatorial problems or even pathfinding in computer games. The problem can be divided into single-agent and multi-agent pathfinding problems. Optimal solutions to both of these problems are usually found using algorithms that are derived from A* algorithm [8]. These algorithms usually use a cost function $f(n) = g(n) + h(n)$ where $g(n)$ is a value of the shortest path found so far from the start node s_a to the node n and $h(n)$ is the heuristic estimate of a value of a path from the node n to the terminal node t_a. If this optimization function never overestimates the shortest path from the start node s_a to the terminal node t_a then the heuristic function $h(n)$ is called admissible and algorithms that use such function are guaranteed to find an optimal solution. Single-agent pathfinding problems consist of problems where one agent is given a start position s_a and an end position t_a and his goal is to find the path between the two. The multi-agent pathfinding problem which is generalization of the single-agent pathfinding problem for $a > 1$ agents. Each of these agents is assigned a start location s_a and a terminal location t_a. The task is to find paths for all agents from their start locations to terminal locations with the goal to avoid all collisions with obstacles and other agents. The goal of many algorithms can be concerned about finding any solution in the shortest time possible, while other algorithms concern themselved with minimizing the cumulative cost function. An example of such cumulative cost function can be a sum of distances traveled by each agent, minimizing waiting time of agents or minimizing the realization time of a plan. Solving multi-agent pathfinding problem in general form is an NP-complete task. The multi-agent pathfinding algorithms can be divided into two main groups:

- The centralized approach assumes that there exists a central unit that gathers information from all agents and whose task is to find paths for all agents. It enables an effective way of cooperation between the agents and usually leads to better solutions but its main disadvantage is the reliability on the central unit.
- The decentralized approach assumes that every agent is equiped with a

processing unit and has its own responsibilities.

This thesis is concerned about centralized approaches and as such the decentralized approaches are out of scope. The algorithms using the centralized approach can be divided into coupled and decoupled subgroups.

Coupled algorithms define the multi-agent pathfinding problem as a global, single-agent pathfinding problem which means that n agents are considered to be a single unit with $n \times k$ degrees of freedom, where k is number of degrees of freedom of each individual agent. These algorithms are typically used for a small number of agents and use *A*-based* algorithms to find solutions.

Decoupled algorithms on the other hand find paths for each agent individualy and combine them to form the final plan. These algorithms differ in the way they handle colliding paths and are used typically for a larger number of agents in a system. Their main disadvantage is that they are usually not complete and the produced plans are not optimal.

The following sections define the multi-agent pathfinding problem and related terminology used in the next chapters. The last sections introduce the chosen algorithms that are implemented and described in Chapters 3, 4, 5 and 6.

2.1 Problem definition and terminology

Multi-agent pathfinding is a problem that is concerned about finding paths for multiple agents from their given start locations to their target locations without colliding with each other or obstacles in the environment while also optimizing a global cost function.

2.1.1 Problem inputs and outputs

Inputs into multi-agent pathfinding problem are:

- A graph $G(V, E)$ where $|V| = N$. The vertices V of the graph are all the possible locations for agents and the edges E are all possible transitions between the locations.
- k agents each labeled $a_1, a_2, ..., a_k$. Each of these agents has a start location $s_i \in V$ and a target location $t_i \in V$.

For the problem simplification the time is discretized into time points.

The output of this problem is a plan, that specifies location of every agent for all time points where at the beginning all agents are at their initial locations and at the end all agents are located in their goal locations.

2.1.2 Actions

Every agent can perform two types of action at each time point: It can move into one of neighbouring nodes or it can wait at its current location. Every

algorithm can make different assumptions regarding the cost of these actions but this thesis assumes that staying idle has zero cost of distance traveled, but costs time. Another assumption is that once an agent reaches his terminal node it waits for other agents to finish.

2.1.3 Constraints

The main constraints on agent movement assumed in the thesis are:

- No two agents a_1 and a_1 can occupy one node $v \in V$ at the same time.
- Assume two agents a_1,a_2 located in two neighbouring nodes $v_1, v_2 \in V$ respectively, they can not travel across the same edge (v_1, v_2) at the same time in opposite directions. In other words two neighboring agents cannot swap positions. However, this thesis assumes that it is possible for agents to follow one another. For example if agent a_1 moves from $v_2 \in V$ to $v_3 \in V$ then agent a_2 can at the same time move from $v_1 \in V$ to $v_2 \in V$.

2.1.4 Composite roadmaps

The algorithm presented in Chapter 6 uses *composite roadmap* to find a plan for all agents.

The *composite roadmap* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a graph that is defined as follows. The vertices $\mathcal{V}$ are all combinations of placements of m agents on original graph G that are without any collision. These vertices can also be viewed as m agent configurations $C = (v_1, v_2, ..., v_m)$, where an agent a_i is located in a vertex v_i and all agents are pairwise collision free. The edges of $\mathcal{G}$ can be created using either Cartesian product or Tensor product. For the purposes of this thesis the Tensor product is used and thus for two m agent configurations $C = (v_1, v_2, ..., v_m)$, $C' = (v'_1, v'_2, ..., v'_m)$ the edge (C, C') exists if $(v_i, v'_i) \in E_i$ for every i and no two agents collide with each other during the traversal of their respective edges.

What this definition says is that the edge between two agent configurations C,C' in the composite roadmap exists if for every pair of locations v_i,v'_i exists an edge $(v_i, v'_i) \in E$ in the original graph G and that during the simultaneus transition of agents from C to C' no collision occurs.

2.1.5 Chosen algorithms

Total of four algorithms were chosen to be implemented in this thesis.

The first algorithm is the representative of decoupled algorithms that uses derived A* algorithm to find the paths for each agent individually. The main idea of this algorithm is that the planning is carried out on free time window graph that is constructed from the original graph for each agent individually. Agents are planned one after another and after each agent finds its path the free time window graph is updated to remove the time intervals that were

used by the agent from the free time window graph. This algorithm is not complete but on open maps produces resutts close to optimum.

The second algorithm is an optimal, complete coupled algorithm whose idea is to split the pathfinding into two search problems. The first search called high-level search explores the space of combinations of costs for each agent in breadth first manner by gradually increasing cost restrictions of every agent. This search creates constraints on path costs for individual agents and the second search called low-level search then uses these constraints to try to find a solution that does not violate these constraints. If such solution exists it is returned and the algorithm terminates.

The third algorithm uses a decoupled approach and its main idea is to find paths for each agent on spanning tree of a graph G. This algorithm is complete if the number of agents that need to be coordinated is lower than the number of leafs in the spanning tree. The main idea is that if all agents are located in leaf nodes of this spanning tree and at least one leaf node is free then all internal nodes of the tree can be traversed without collision and any two agents can switch their locations. The search itself is divided into three main phases. In the first phase all agents are moved into leaf nodes, in the second phase the agents are moved into positions from which they can move to their respective goals without causing collisions and the last third phase moves agents to their goals.

While all previously mentioned algorithms are deterministic the last algorithm is a representative of coupled sampling algorithms and is a modification of a well known *Rapidly-exploring random tree* algorithm. This sampling based algorithms is probabilistically complete and as such the probability to find solution approaches one as time is spent. However, the algorithm can not determine if solution exists. The search is done by exploring a *composite roadmap* using the *oracle technique* while also trying to connect the current state with the goal state.

Chapter 3

Conflict-Free Route Planning in Dynamic Environments

The first presented algorithm was first introduced by Adriaan W. ter Mors in [12][11]. This decoupled approach to multi-agent planning adapts A* algorithm through a graph of free time intervals or windows to find the shortest path for each agent in terms of time required. This chapter includes two major sections, the first of which introduces the model of multi-agent planning that is assumed by this algorithm, while the second section presents the route planning algorithm.

3.1 Model

The input of the algorithm is a set A of agents each of which has to find the fastest path from his initial position to his goal position. The next input is a roadmap which is modeled as a resource graph $G_R = (R, E_R)$ where resources R can be the paths in a robot warehouse, lanes on airports or roads, intersections etc. The path the agent can follow is restricted to edges E_R which limit the transition of robots such that agent can get from resource r_1 to resource r_2 only if r_2 is the neighbour of r_1 in graph G meaning that edge $(r_1, r_2) \in E_R$. Each resource has also two main attributes associated with it. These are capacity $c(r)$ which corresponds to the maximum number of agents that can occupy a resource at the same time and duration $d(r) > 0$ that represents the minimal time it takes the agent to traverse a given resource. Plans for every agent then contain not only sequence of resources on its path but also time intervals during which the agent visits them.

For general purpose planning the algorithm assumes that the resource graph is constructed such that resources are of two types: intersection resources with capacity 1 and lane resources with capacity 1 or greater. Another assumption is also that if multiple agents are present on the same resource then they are all traveling in the same direction and their order does not change, meaning they cannot overtake each other. The idea is that the lanes are not wide enough for two agents to drive in parallel but long enough so that agents can drive behind each other.

The capacity is a simplifying assumption introduced because it eliminates

the need to calculate collisions during the planning phase and only capacity of resource and free time windows overlap are thus taken into consideration.

Several terms such as *route plan*, *resource load*, *free time window* and *free time window graph* must be defined for further description.

3.1.1 Route plan

Given a start resource r and a goal resource r' the *route plan* is a sequence $\pi = (\langle r_1, \mathcal{T}_1 \rangle, ..., \langle r_n, \mathcal{T}_n \rangle)$ of n plan steps where $\mathcal{T}_i = [t_i, t_i')$, such that $r_1 = r$, $r_n = r'$, $t_1 \geq t$ and $\forall j \in \{1, ..., n\}$:

1. interval $\mathcal{T}_j$ meets interval $\mathcal{T}_{j+1} (j < n)$. This constraint means that exit time from j^{th} resource must be equal to the entry time to the $j + 1th$ resource.
2. $\mid r_j \mid \geq d(r_j)$ meaning that the agents occupation time of a resource is at least sufficient to travel across the resource in the minimum travel time.
3. $(r_j, r_{j+1}) \in E_R$ means that if two resources follow each other in the plan, then there must be an edge between them in the resource graph.

3.1.2 Resource load

Given a set of agents plans $\mathcal{P}$ and a set of all time points T, the resource load is a function $\lambda : R \times T \to \mathcal{N}$ which returns a number of agents that are located at resource r in a given time $t \in T$:

$$\lambda(r, t) = |\{(r, \mathcal{T}) \in \pi \mid \pi \in \mathcal{P} \wedge t \in \mathcal{T}\}|$$

What this means is that an agent can use a resource only in such time intervals where the resource is occupied by less agents than its capacity. By entering the resource only in these *time windows* it is ensured that no conflicts with other agents occur in agents plan.

3.1.3 Free time windows

Given a resource load function λ, a free time window on resource r is a maximum interval $w = [t_1, t_2)$ such that:

1. $\forall t \in w : \lambda(r, t) < c(r)$
2. $(t_2 - t_1) \geq d(r)$

What these conditions mean is that for an interval to be a free time window the capacity in each time point of the interval must be always sufficient and also long enough so that the agent can travel across the resource. Because every agent that wants to traverse a free time window on a resource must enter the window, travel across and then leave it, it cannot enter at the end of a free time window or leave at the start of one because of non zero traversal

time. For this reason every free time window w has entry window w_{entry} and exit window w_{exit} associated with it. These are limited by a time window w by the minimum traversal time through the resource:

- $w_{entry} = [t_1, t_2 - d(r))$
- $w_{exit} = [t_1 + d(r), t_2)$

If agent desires to travel from one resource r_1 to the neighbouring resource r_2 it needs to find free time windows w,w' on both of these resources. Due to constraints on route plan discussed in Section 3.1.1 for w' to be reachable from w the entry window w'_{entry} must overlap with exit window w_{exit}.

3.1.4 Free time window graph

Free time window graph $G_W = (W, E_W)$ is a directed graph where vertices $w \in W$ are a set of free time windows and edges E_W specify reachability between free time windows of W. This means that given two free time windows w,w' on resources r,r' respectively it holds that $(w, w') \in E_W$ only if:

- $(r, r') \in E_R$
- $w_{exit} \cap w'_{entry} \neq \emptyset$

Each agent uses its own free time window graph for planning his route as every free time window graph contains only information about $n - 1$ previous agents. It does not contain information about movements of such agents. For this reason some assumptions need to be made about the graph of start and end resources for each agent because otherwise it would be possible that some agent i could make it impossible for agent $i + 1$ to find his plan. These assumptions are for example that no two agents can have the same destination, the destination resources have sufficient capacity to hold all agents that have them as their goal or that once each agent reaches his destination he vanishes from the infrastructure.

Our implementation has two versions with different assumptions. The first implementation assumes that once an agent reaches his goal it departs from the infrastructure. This is inspired by the use on airports where once aircraft reaches the start of runway it initiates take off and can be ignored by the rest of the agents(assuming the runway is not a part of the graph G). The second implementation assumes that no two agents can share the same goal and do not depart the infrastructure once they reach it.

3.2 Planning algorithm

The classical shortest path planning expects that if a node v lies on the shortest path from s to t, then the shortest path to v can be expanded to

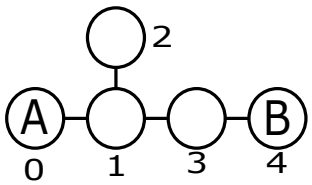


Figure 3.1: Example problem where classical planning approach can not find solution. Agent A has node 4 and agent B has his goal in node 0. Agents are planned in B,A order.

shortest path to t. However, this approach may run into difficulties in certain scenarios. Figure 3.1 shows one such scenario. The agent A has his goal node in node 4 and the agent B has his goal in node 0. Consider that agent B is already planned. In this case agent A cannot be planned because partial path to node 1 cannot be further expanded towards goal because it would cause collision with agent B. What is required of agent A is to move to node 2, wait for agent B to pass and then move towards goal node 4. However, this is not possible in classical shortest planning, but is possible for the algorithm described in this section.

The main idea of the algorithm is that it considers only partial plans leading to the free time window on a resource as opposed to a classical planning approach that considers partial plans to whole nodes. If the partial plan arrives to resource r at time t which lies in the free time window w then every other partial plan that arrives to the same time window on the resource r in time $t' > t$ can be simulated by waiting in resource r from t to t'. This approach allows agent A in the example on Figure 3.1 to move to node 2, wait for agent B to pass to his goal and then move directly to goal node 4.

Algorithm 1 Path planning algorithm

1: **if** $\exists w\,[w \in W \mid t \in w_{entry} \wedge r_1 = resource(w)]$ **then**
2: mark(w,open)
3: entryTime(w) $\leftarrow$ t
4: **while** $open \neq \emptyset$ **do**
5: $w \leftarrow \underset{w' \in open}{\operatorname{argmin}} f(w')$
6: $mark(w, closed)$
7: $r \leftarrow resource(w)$
8: **if** $r = r_2$ **then return** $followBackPoints(w)$
9: $t_{exit} \leftarrow g(w) = entryTime(w) + d(resource(w))$
10: **for** $w' \in \{\rho(r, t_{exit}) \setminus closed\}$ **do**
11: $t_{entry} \leftarrow max(t_{exit}, start(w'))$
12: **if** $t_{entry} < entryTime(w')$ **then**
13: $backpointer(w') \leftarrow w$
14: $entryTime(w') \leftarrow t_{entry}$
15: $mark(w', open)$
return *null*

The main algorithm performs a search through the free time window graph in a similar way to A*. The algorithm keeps track of open partial plans with their values $f = g + h$ where g is the actual time cost of the partial plan and h is heuristic estimate of a cost of a plan to goal resource from the end of the partial plan. Our implementation assumes that each edge is traversed in one unit of time which enables the heuristic estimate to be the Euclidean distance to the goal node. The search process can be seen in Algorithm. 1. The first step is to check whether there exists a time window w on a resource r such that $t \in w_{entry}$ (line 1). In case no such window exists then no plan exists and thus $null$ is returned. If such window exists it is marked as open and a time t is marked as an entry time into the window w (lines 2-3). On line 5 a partial plan with the minimum cost $f(w) = g(w) + h(w)$ is selected and marked as closed (line 6). If a resource r that is associated with the window w is the goal resource r_2 then the shortest path to r_2 has been found and it is returned through following back pointers. If the heuristic used to estimate h is consistent then no other partial plan on the open list have higher cost and expansion of these partial plans would never create a plan with lower cost. If a resource r is not the goal resource then an exit time t_{exit} from the window w is found as $entryTime(w) + d(r)$. Once the exit time t_{exit} is found then the algorithm iterates over all reachable time windows $w' \in \rho(r, t_{exit})$ where $\rho(r, t_{exit})$ is a set of all reachable time windows from w and earliest exit time t_{exit}. For each of these windows an entry time t_{entry} is found as a maximum of t_{exit} and a start of window $start(w')$. If the entry time t_{entry} is smaller than the entry time to w' then w' is marked as open and added to the open list as well as update the entry time into w' to t_{entry} and back pointer to w. In case where no plan to the goal r_2 exists the algorithm returns null.

At the start of the algorithm all resources start with one available free time window $[0, \infty)$. After finding a plan for each agent the free time window graph is updated by removing the time intervals used in the plan of previous agent.

Chapter 4

Increasing Cost Tree Search

This chapter describes a coupled optimal multi-agent planning algorithm first introduced by Guni Sharon, Roni Stern, Meir Goldenberg and Ariel Felner in [9]. It is based on a different approach as opposed to *A*-based* algorithms that rely heavily on the used heuristic that is guiding the search. Increasing cost tree search (ICTS) relies on the fact that the complete solution is made of paths for each individual agent. Based on this, ICTS splits the multi agent pathfinding search into two parts:

1. High-level search which searches through the space of combinations of costs for each individual agent. This search provides the constraints for the low-level search.

2. Low-level searches for a valid solution given the constraints on costs of individual agents provided by the high-level search. This phase can also we viewed as a goal check for the high-level search.

4.1 High-level search

The high-level search performs a search on a tree structure called Increasing Cost Tree. For k agents the nodes in this tree are k-element vectors of cost values for each agent. Each such node represents all possible solutions with a given path lengths for every agent. Sum of elements of this vector represents cost of every possible plan with given lengths of paths for each agent and is always the same for every node in the same level of the tree. A node with optimal path costs for each individual agent without taking other agents into consideration is chosen to be root of the tree. For k agents the successors are generated by increasing the cost of every agent by 1 always generating one new successor. This procedure always generates k new nodes. For example, a node with costs $[C_1, C_2, ..., C_k]$ generates these successors: $[C_1 + 1, C_2, ..., C_k]$, $[C_1, C_2 + 1, ..., C_k]$,..., $[C_1, C_2, ..., C_k + 1]$. For the example problem seen in Fig. 4.1 the high-level search goes through a tree depicted in Figure 4.2. The root of the tree has costs $(2, 2)$ because the optimal paths for agents A,B are 1,4 and 1,5 respectively both of which have cost 2. Node $(2, 2)$ is thus chosen to the the root node. Because the low level search fails for this case as the

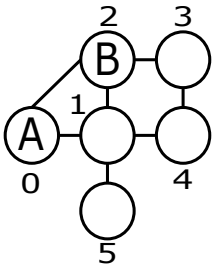


Figure 4.1: Example problem. Agent A has node 4 as his goal and agent B has node 5 as his goal.

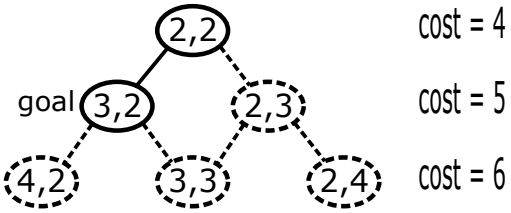


Figure 4.2: Tree searched in the high level search.

only paths with length 2 to the goal would result in a collision the search continues and successors $(3, 2)$ and $(2, 3)$ are created. The successor $(3, 2)$ is visited first because the search is performed in the breadth-first manner. The low-level search succeeds for this node, because agent A can wait in the node 0 until agent B gets to his goal and then move towards the goal in node 4. Other nodes in the high-level search are then ignored as the solution has been found.

4.2 Low-level search

The input into low level search are the constraints on path lengths for each agent. The first step of low level search is for each agent to find all paths with a given length. The number of such paths is exponential and so the original algorithm in [9] stores these paths in a special structure called Multi-value decision diagrams (MDDs).

4.2.1 Multi-value decision diagrams

All paths of the given length l are stored in MDD in the original algorithm. This structure generalizes Binary Decision Diagrams by allowing more than two choices for every decision node. MDD_i^c is an MDD for agent i which stores all paths with the cost c. It has one source node s and one node terminal node t. All nodes in the MDD have a depth d below the source node s. Every node at set depth d corresponds to a possible location of a_i at the $d - th$ step on a set path. For the example problem in Fig. 4.1 the MDDs of agents A,B can be seen in Figure 4.3

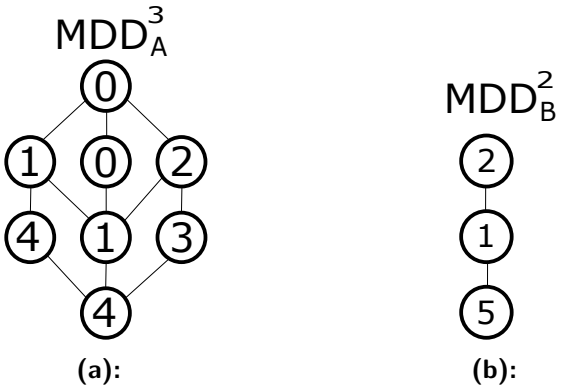


Figure 4.3: MDDs for agents A,B respectively for the problem in Fig. 4.1.

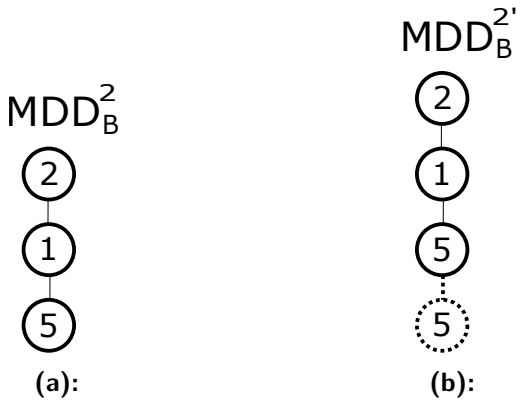


Figure 4.4: Dummy node added to the MDD.

4.2.2 k-agent MDD space searching algorithm

To find the final plan this algorithm iterates over the MDDs to find a set of non-conflicting paths. If no such set is found, then it is returned to the high-level search that the current node is invalid. Without the loss of generality this section describes the case for two agents. Considering two agents A,B located in their start positions, the global 2-agent search space is defined as the state space created by moving these two agents simultaneously to all possible directions. Given MDDs for agents A,B that correspond to the high-level node c,d. No loss of generality is achieved if $c = d$ is considered. Because if $c > d$ the difference of length can be achieved by adding dummy nodes to the "shorter" MDDs terminal node to match their lengths. This operation can be seen in Figure 4.4

The cross product of MDD_A and MDD_B is used to generate a subspace of the global 2-agent search space. This subspace is called 2-agent-MDD search space and is only a subset of global 2-agent search space because it is constrained only to moves contained in the single agent MDDs. Using this the 2-agent-MDD can be defined as MDD_{AB} for agents A and B. Every node in MDD_{AB} corresponds to a valid non-conflicting pair of locations for the two agents. An example of 2-agent-MDD created by cross product of

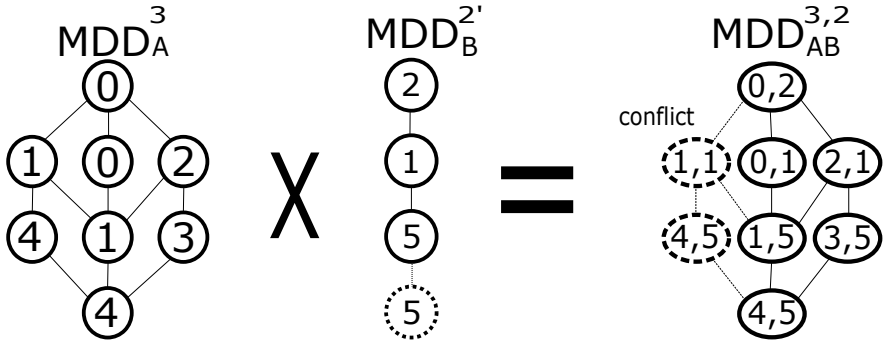


Figure 4.5: Example of cross product for MDD_A^3 and $MDD_B^{2'}$

MDD_A and MDD_B from Figure 4.3 can be seen in Figure 4.5.

As seen in this picture the nodes that contain conflict of the two agents are omitted along with all their successors.

The algorithm itself can be seen in Algorithm 2. The first step is to create an initial node for the high-level search by finding shortest paths to the goal for each agent and using their length(line 1). To guarantee optimality the tree is then searched in breadth-first manner and every node is checked if its the end node by performing low level search on it and using the lengths of paths for each agent as constraints. In case the low level search succeeds the solution found is returned. The low level search can be seen in Algorithm 3. The first step of the low level search is to create MDD for each individual agent. After that the n-agent-MDD is searched for a path to the terminal node. If such path exists then the low level search succeeded and the solution is obtained by backtracking the path from the terminal node. If no path exists then it means that no solution exists and an empty set is returned.

The n-agent-MDD can be extremely large but it does not need to be wholly created and stored in memory in order to be searched. What can be done is to systematically search the n-agent-MDD using kind of search as no particular type of search is required to guarantee optimality. In our implementation the depth first search was used.

Algorithm 2 The ICT search algorithm

```
1: T ← getInitialCosts
2: for each ICT node searched in breadth-first manner do
3:     Solution = lowLevelSearch(node)
4:     if Solution was found then return Solution
```

Algorithm 3 Low-level search algorithm

```
MDDs ← create MDD for every agent
for node searched in cross product of MDDs do
    if node contains conflict then
        skip node
    if node is terminal then return backtrack(node)
if no solution exists then return ∅
```

Chapter 5

A Complete and Scalable Strategy for Coordinating Multiple Robots Within Roadmaps

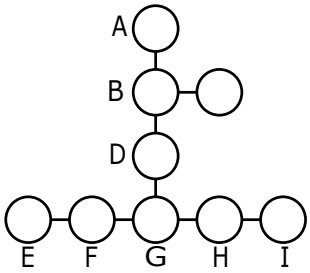
Mike Peasgood, Christopher Michael Clark, John McPhee first introduced this centralized decoupled algorithm in [6]. The main principle of this algorithm is to find nodes for agents to move to while maintaining such a state of the graph that does not block other agents. The paths between these points are then found using standard one-agent planning algorithms such as A* while looking at other agents as obstacles. The main advantage of this algorithm is its scalability and also the ability to check whether it can find solution for given number of agents. The main disadvantage is the dependance on spanning tree and its generation because it either has to be generated before every start of the algorthm to fit given problem i.e use goal and start nodes as leafs or compute the spanning tree once, use it for every given problem but with worse quality solutions.

5.1 Map representation and spanning tree selection

The original algorithm assumes that the given graph G is undirected. While it is possible to modify the algorithm to work on directed graphs, it is beyond the scope of this thesis. Given the graph G the algorithm first finds spanning tree T in this graph i.e. a subset of edges connecting all vertices without creating any loops. Spanning tree with N nodes has L leaf nodes, $N - L$ internal nodes and one of the internal nodes is always the root node. An example of a graph and its spanning tree can be seen in Fig. 5.1 where graph nodes G,F,D,H,B are the interion nodes of tree rooted in G while nodes E,A,C,I are the leaf nodes. Every node in the tree had its depth d associated with it meaning the number of nodes that are encountered on the way through the tree to the root. Illustration of a depth for each node from the graph on Fig. 5.1 can be seen in Fig. 7.1b.

Any algorithm that can generate a spanning tree of a graph can be used as the main algorithm is not dependent on the way the spanning tree is found.

(a) : Example on input graph



(b) : Spanning tree $\mathcal{T}$ of graph on picture (a) rooted in node G with subtree rooted in B

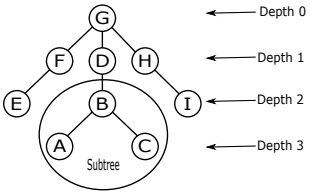


Figure 5.1: Example graph with corresponding spanning tree

But when choosing the algorithm to generate such a spanning tree several things need to be kept in mind. The algorithm moves agents usually only between leaf nodes of a spanning tree with the exception of final phases where agents are moved to their final positions. Because of this fact the maximum number of agents that can be planned is directly reliant on the number of leafs in the tree. Last thing to note is that the internal nodes are kept free to move through. Result of these limitations is that the spanning tree should have maximal number of leafs. Our implementation uses algorithm that tries to maximize number of leafs in the spanning tree. Further information about this algorithm can be found in Section 5.3.

5.2 Planning algorithm description

The algorithm breaks the multi-agent pathfinding problem into a sequence of 4 phases. Because the graph is represented as a spanning tree the algorithm can utilize following two properties of this representation if number of agents r is smaller than number of leafs L:

1. Any agent is able to move to any internal node in the graph G if all agents are located in leaf nodes.

2. It is possible for any two agents to swap their positions if all agents are located in leaf nodes.

The first property is obvious as there must always be a non-coliding path from a leaf node to an interior node if all agents stay in leaf nodes. Since this property property holds and because $r < L$ there is always at least one free leaf N_{free} that can be used for the swap. For any two agents A_1 and A_2 standing on respective leaf nodes N_1, N_2 and a free leaf node N_{free} the swap operation can be achieved by moving A_1 to N_{free}, A_2 to N_1 and then A_1 to N_2.

Both previously mentioned properties ensure that if all agents are in leaf nodes it is possible to move agents towards their goal nodes without risk of deadlock. The multiphase algorithm consists of the following phases:

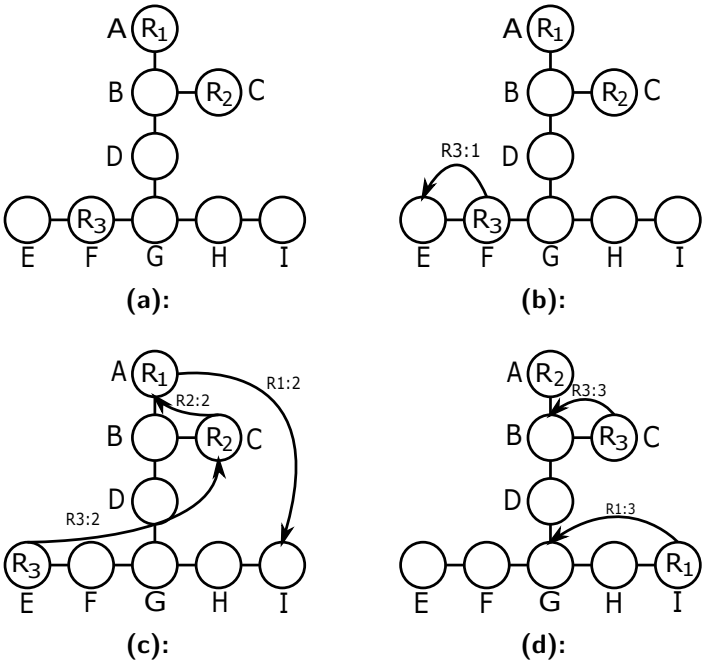


Figure 5.2: Example of a run of the algorithm. An initial configuration can be seen on Figure (a). (b) depicts the first phase of the algorithm where all agents are moved to leaf nodes. Second phase is depicted on Figure (c) where agents are moved to subtrees of their goals. (d) shows the final phase of the algorithm that moves all agents to their respective goal nodes.

- Phase 1: Moving all agents to leaf nodes.
- Phase 2: Moving agents in order of their goals tree depth to leaf nodes from which they can move to goal position without collision.
- Phase 3: Moving agents to their goal positions.
- Phase 4: Building concurrent plan from generated movement sequences for each agent.

For the purposes of following sections it is required to define several functions as they are frequently used in the pseudocodes for the previously listed phases.

$currentNode\,(agent)$: this function returns node that given agent occupies in current step of the algorithm.

$goalNode(agent)$: returns goal node of given $agent$

$freeLeafNode\,()$: returns free leaf node within the spanning tree.

$freeLeafInSubtree\,(node)$: returns free leaf in subtree of the spanning tree rooted in $node$. $freeLeafNotInSubtree\,(node)$: works in a similar manner but returns free leaf node outside of the subtree rooted in $node$.

$astarPath(start, end)$: this function returns shortest path between $start$ and end node while assuming there are no obstacles in the graph

$freeAstarPath(start, end)$: this function functions the same as $astarPath$ with the exception of considering already occupied nodes.

$findObstacleAgent(path)$: function searches for all agents that have their position in the current time on any node of path. List of agents is returned in reversed order meaning the agent that stands the closest to end node is the first. If no agent stands on the path empty set is returned.

$addPath(path, agent)$: this method serves to add $path$ sequence to the plan for given agent. It also updates the agents position to the last node of the path.

$planAgentToNode(agent, node)$: is a function that combines $freeAstarPath$ function to find a path between agents current location and given node and then adds found path to agents plan using $addPath$ function

$subTreeContains(root, node)$: queries spanning tree is its subtree rooted in $root$ contains $node$.

$getBlockedAgent(node)$: finds all agents that are currently located within subtree rooted in $node$ whose goal node is outside of the subtree.

$sortAgentsByDepthOfGoal()$: returns order of agents in the ascending order of the depth of their goal node within the spanning tree from deepest to shallowest.

$isLeafNode(node)$: returns true if $node$ is leaf of the spanning tree.

5.2.1 Phase 1: Reaching leaf nodes

Algorithm 4 Phase 1 of the multiphase algorithm. Serves to move all agents to leaf nodes of the spanning tree

```
1:  order = sortAgentsByDepthOfGoal()
2:  for each agent in order do
3:      start = currentNode(agent)
4:      if isLeafNode(start) then
5:          continue
6:      leaf = freeLeafNode()
7:      path = astarPath(start,leaf)
8:      obstacle = findObstacleAgent(path)
9:      if no obstacle found then
10:         addPath(path,agent)
11:     else
12:         start = currentNode(obstacle)
13:         path = astarPlan(start,leaf)
14:         addPath(path, obstacle)
```

Goal of the first phase is to move every agent into a leaf node of the spanning tree. This is illustrated in Alg.4. Agents are first sorted in ascending order of their goal nodes depth(line 1). Each agent if then checked if he is already located in leaf node. If that is the case it skips this agent and goes to the next one (line 5). When an agent is not located in leaf node the algorithm

finds available free leaf node L_i of the spanning tree (line 6). This operation is guaranteed to succeed because $r < L$ which implies that at least one free leaf is available node even when all agents are located in leafs. For the choice of leaf any heuristic can be used or left to be random. Choice of leaf is further discussed in the implementation Section 5.3 of this chapter. In the next step the path P from agents R_i current location is found to the selected free leaf node using all available edges of the graph and ignoring all obstacles(line 7). Success of this phase is independent on the choice of the pathfinding algorithm but quality of solution is. For the purposes of describing the algorithm, A* is assumed to be used for the path finding. When a path P is found it is examined whether any other agent has his current position on any of its nodes (line 8). If no agent is found on the path P then the found path segment is added to the plan for the agent R_i (line 10). In case there are one or more agents found on any of the nodes of the found path then agent R_j that is the furthest along the path is selected as he has obstacle free path if he follows rest of the path P to leaf L_i.

These steps are repeated until all agents are located in the leaf nodes. It is guaranteed that this process terminates because in each iteration there is at least one agent moved to leaf node. If the moved agent was already on leaf node then closer node is freed for agent R_i to plan to. The only case in which this procedure would fail is if $r \geq L$ in which case failure is detected.

Initial configuration of an example problem can be seen on Figure 5.2a. Agents R_1, R_2, R_3 are placed in nodes A, C, F and their goal nodes are in G, A, B respectively. Spanning tree of this given graph is in Figure 7.1b. Because agents R_1 and R_2 are already located in the leafs of the spanning tree, it is not necessary to move them. On the other hand, agent R_3 is not located in a leaf node as thus is moved to node E which is his nearest leaf node. This operation can be seen in Figure 5.2b.

5.2.2 Phase 2: Sorting agents by depth of goals

In Phase 2 the algorithm attempts to move all agents into nodes from which they can move to the goal without collisions. This is done by moving them into leaf nodes that are close to their goal positions. However, a problem occurs when they try to move to their goal positions and it is required for them to swap positions or agents R_i goal position blocks path to agents R_j goal position. This problem is solved by utilizing relative positions of agent and his goal in the subtree.

Let T_G be a subtree of a spanning tree rooted in goal node G_i for agent R_i. The previously mentioned problem can occur only if any of these conditions are met

- G_i is occupied, another agent R_j is inside the subtree of T_G and his goal is outside the subtree.
- G_i is occupied, another agent R_j is outside the subtree of T_G but his goal is inside T_G

Algorithm 5 Phase 2 of the algorithm. Its purpose is to move agents into such positions from which it is possible for them to move to their goal node without any collisions

```
1:  order = sortAgentsByDepthOfGoal()
2:  for each agent in order do
3:      start = currentNode(agent)
4:      goal = goalNode(agent)
5:      if subTreeContains(goal,start) then
6:          continue
7:      blockedAgent = getBlockedAgent(goal)
8:      if blockedAgent exists then
9:          blockedNode = currentNode(blockedAgent)
10:         leaf = freeLeafNotInSubtree(goal)
11:         if leaf exists then
12:             planAgentToNode(blockedAgent,leaf)
13:             planAgentToNode(agent, blockedNode)
14:         else
15:             leaf = freeLeafInSubtree(goal)
16:             planAgentToNode(agent,leaf)
17:             planAgentToNode(blockedAgent,goal)
18:             continue
19:     else
20:         leaf = freeLeafInSubtree(goal)
21:         planAgentToNode(agent,leaf)
```

- G_i is occupied, another agent R_j and hit goal G_j are both in the T_G but path from agents R_j current locations contains node G_i

Both cases 1 and 3 can be solved by moving agents into leaf nodes within the subtrees of their goals. Case 2 can be solved by ordering the depth of agents within the subtree based on the depth of their goal within the spanning tree. The Algorithm. 5 solves these issues by first sorting agents by the depth of their goal from deepest to shallowest(line 1). In the next step the algorithm checks whether agent is already within subtree of their goal. In the case he already is within the subtree of his goal it is not required to move him and as such the agent is skipped (line 6). If agent R_i is not located within subtree of his goal G_i, the algorithm first checks if there is any other agent within subtree T_{G_i} that has his goal outside of this subtree. If such agent does not exist then the algorithm finds free leaf node within T_{G_i} and moves the agent to it (lines 20-21). However, in our testing it occured that it is possible that T_{G_i} does not contain aby free leafs. Because agents are taken by the depth of their goal from the deepest to the shallowest it is possible to move an agent safely to his goal position instead, as all agents that have goal nodes within subtree T_{G_i} must have already been processed and as such there cannot exist any agent R_j outside T_{G_i} with goal G_j inside T_{G_i}. In case there is at least one agent inside T_{G_i} in position N_j that has his goal outside

T_{G_i} then the one R_j whose goal node has the highest depth is chosen and algorithm attempts to find free leaf that is not in T_{G_i} (line 10). If such leaf N_{leaf} exists, then agent R_j is moved to N_{leaf} and agent R_i is moved to N_j (lines 11-13). However, if such leaf does not exist the algorithm finds a leaf N_{leaf} inside the subtree T_{G_i} and moves agent R_i to it while blocked agent R_j is then moved to his goal node G_j (lines 14-18). This however proved inefficient in our testing as it is not guaranteed that there will not be any other agents that will need to pass through G_j and as such caused failure of the algorithm. For this reason it proved to be the best that blocked agent R_j was not moved in our implementation.

Figure 5.2c depicts the process of phase 2. Agents move in order of the depth of their goal which in this case means that agent R_2 moves first as his goal node A is in depth 3 of the spanning tree. Node A is at the given time occupied by agent R_1 and because node A is a leaf node and as such is the only node in its own subtree, another free leaf node I outside of subtree of A is found. Agent R_1 is then moved to node I and after that agent R_2 is moved to A. In the next iteration agenr R_3 is moved to leaf node C which is in subtree of node B. After initial movement agent R_1 is no longer moved as node I is in subtree of node G.

5.2.3 Phase 3: Filling remaining goals

Algorithm 6 Phase 3 of the algorithm. All agents that are still not in their goal nodes are moved to them.

```
1: order = reverse(sortAgentsByDepthOfGoal())
2: for each agent in order do
3:     goal = goalNode(agent)
4:     if agent not at goal node then planAgentToNode(agent,goal)
```

The last phase of the main algorithm moves any remaining agents that are not in their goal positions to them. Agents are first sorted by the depth of their goal node in the reverse order than the one used in previous phases i.e. from the shallowest to the deepest (line 1). This order guarantees obstacle free paths for agents moving to their goal nodes because of their configuration created in Phase 2 where they were ordered by the depth of their goal from deepest to shallowest.

Process of phase 3 can be see on Figure 5.2d. Agents are moved in reverse order of their goals depth in the tree. This means that Agent R_1 is first, agent R_3 second and R_2 is the last one. In this phase agent R_1 moves on collision free path towards his goal node G. Agent $R3$ moves to his goal node B and agent R_2 doesnt move as he is already in his goal A.

5.2.4 Optimizing created plan

Because of the assumption made for the previous three phases the resulting plan can be extremely suboptimal. The movement to leaf nodes in Phase

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
R_1	A		B	D	G	H	I								H	G	
R_2	C							B	A								
R_3	F	E								F	G	D	B	C			B

Table 5.1: Plan segments created in phases 1-3

	0	1	2	3	4	5	6	7	8	9	10
R_1	A	B	D	G	H						G
R_2	C					B	A				
R_3	F							G	D	B	

Table 5.2: Plan segments after removing unnecessary cycles

1 and positioning of agents to the subtree of their goals creates a lot of maneuvers that can be unnecessary because the space they occupy can be unutilized. Another issue with the generated plan is that in every moment only one of the agents is moving.

To lessen the impact of these issues and to bring solution closer to optimum some postprocessing optimizations can be made. First optimization that can be done is to remove redundant cycles in agents plans. The second optimization is then to create concurrent plan from generated plan segments from phases 1-3.

5.2.5 Loop removal

As stated in the previous section it is possible that several loops were introduced to the solution during the first 3 phases. To remove the loops it is possible to go over the every time step of the solution and keep track of occupancy in each node. If the same node is encountered by one agent it is certain that there is a loop in his plan and all steps between these two visits of the same node can be ommited from his plan. The plan for a problem solution from Figure 5.2 can be seen in Table 5.1. As stated there can be loops in the base solution from first three phases for example the agent R_3 moves to node E from F as his first step which is unnecessary because he moves to F again in his next plan segment. Because no other agent has stepped on node F when agent R_3 moved to E and back to F it is possible to remove this cycle. Plan after removing all cycles can be seen in Table 5.2

5.2.6 Phase 4: Building a concurrent plan

First 3 phases generate a plan in which only one agent moves at a time. This can be represented as a sequence of path windows for each agent and to lower the time required to plan to complete, these windows need to be overlaped in time as much as possible. This is done by first considering initial positions of agents as windows that are set to time zero. After that the path sequences are considered one by one and shifted back in time until collision with previously

	0	1	2	3	4	5	6
R_1	A	B	D	G	H	G	
R_2	C		B	A			
R_3	F				G	D	B

Table 5.3: Final plan after overlaping segments in time

placed sequences occurs. In this context two agents located in the same node at the same time or agents switching positions between two neighbouring nodes are considered as a conflict.

Non overlaping plan can be seen in Table 5.1 and its version with removed cycles in Table 5.2. Table 5.3 shows final plan after overlaping all segments in time as much as possible. Both cycle removal operation and overlaping segments in time managed to shorten the original plans length from 17 steps to 7 steps and thus bringing it closer to optimum.

5.3 Implementation

Basic implementation followed steps described in the previous sections. Because the quality of solution is affected by the heuristic to choose leaf nodes for agents to go to in the first phase, several methods of leaf choice were implemented and tested such as:

- A random free leaf node
- The nearest leaf to agents position
- A random free leaf in agents goals subtree
- The nearest free leaf in subtree of agents goal
- The nearest free leaf in subtree of agents position

Implementation of the second phase followed the pseudocode of Algorithm 5 but solved issues presented in the corresponding Section 5.2.2. First issue considered the situation when there was an agent R_b that was within subtree of agents R_1 goal but his goal was outside of this subtree. The original algorithm tries to find a free leaf outside of this subtree. If such leaf does not exist, the algorithm then finds a free leaf in said subtree and moves agent R_1 to it. Agent R_b is then moved to his goal. This proved to cause problems as it is not guaranteed that no other agent will not use agents R_b goal for his plan. The lines 15-17 in the original Algorithm 5 were then substituted for pseudocode seen in Algorithm 7. As can be seen the solution was to not move agent R_b as his depth of node must be lower than that of agent R_1 and thus meaning he is moved to subtree of his goal in one of future iterations.

The next issue with pseudocode in Algorithm 5 was that when there was no blocked agent the algorithm tried to move agent R_1 into one of free leafs of his goal but such free leaf would sometimes not exist. The solution to this

problem was to check if free leaf was found. If free leaf was found then no change was necessary and the agent is moved to his allocated free leaf node. Conversely if no free leaf exists in the subtree, agent R_1 is moved to his goal instead. This is possible because if there was any agent that would need to go through agents R_1 goal it would have gone before him as agents move according to ordering by the depth of their goal and as such it is guaranteed that no agent needs to go through agents R_1 goal. These changes modify the lines 20-21 of Algorithm 5 by swaping them for the lines in Algorithm 8.

Algorithm 7 Handling no free leaf outside of subtree

```
leaf = freeLeafInSubtree(goal)
planAgentToNode(agent,leaf)
continue
```

Algorithm 8 Handling no free leaf in subtree

```
leaf = freeLeafInSubtree(goal)
if leaf exists then
    planAgentToNode(agent,leaf)
else
    planAgentToNode(agent,goal)
```

Rest of this section discusses implementation details of necessary data structure for the algorithm as well loop removal and concurrent plan building phases.

5.3.1 Data structure

The algorithm utilizes spanning tree of a graph to plan agent movements. The spanning tree implementation also requires to accommodate fast querying for free leaf nodes and also if certain node is in a subtree of a different node. Implementation of the spanning tree that can be queried for nodes location in subtree is inspired by implementation suggested in [5].

The spanning tree is created from the initial graph as a graph object itself by only selecting used edges and skipping the rest. The algorithm used for this process in our implementation is suggested in [6]. It performs informed search through the whole graph and always expands node that has currently maximal number of neighbours and then closes it. This approach tries to maximize the number of leafs in the spanning tree and thus maximize number of agents that can be planned.

To query the basic tree if node A is in the subtree of node B it is required to move from node A to the root of the tree and check whether node B is encounted along the path. However, this approach is extremely inefficient and because of that the approach proposed and implemented in [5] is used. The main idea of this approach is to assign unique index to each node of the spanning tree and use only these indices I_A,I_B to determine if A is in the

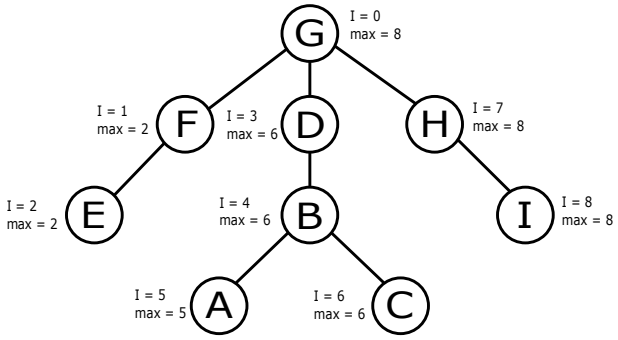


Figure 5.3: Example of indexed spanning tree

subtree of B. The indices are assigned in such a way that subtree of node A contains only nodes with higher indices than that of A. This is achieved by performing DFS search through the created spanning tree and marking the nodes in the order they are visited. Root of the tree has always index 0. Example of indexed spanning tree 7.1b can be seen in Figure 5.3. If subtree is then queried for relation of nodes A and B, the only following three cases can happen:

1. Index of A is lower than that of B and A is not in the subtree of B.
2. Index A is higher than B and node A is in the subtree of B.
3. Index A is higher than B and node A is not in the subtree of B.

The first case can clearly determine the relation between A and B. The second and third cases are then left to be distinguished. This can be achieved if each node also saves the index of the highest node in its subtree. If this information is known then the the following rule can be applied:

- If $I_A \geq I_B \wedge I_A \leq I_{B_{max}}$ then node A lies in the subtree of B.

5.3.2 Implementation of loop removal

As discussed in the Section 5.2.5 the plan created in by the first three phases of the algorithm may contain a large number of maneuvers for each agent that are unnecessary. Loop removal procedure tries to mitigate this issue by looking for these loops in the plan and removing them. The procedure can be seen in Algorithm 9. The process keeps information about node visits in structure $nodeVisits$ which is empty at the start (line 1) The algorithm then iterates over all plan segments, extracts information about which agent it belongs to and then iterates over all of its steps (lines 2-5). For every step the algorithm checks which agents already stepped on the corresponding node. If no agent stepped on or if other agent has stepped on said node the current step is marked in $nodeVisits$ structure along with the position in current segment (lines 7-10). On the other hand, if the agent who previously visited current node is the current agent, then the algorithm has found a cycle. This

cycle is them removed and structure $nodeVisits$ is updated for every node in the removed cycle accordingly (lines 11 - 16).

Algorithm 9 $LoopRemoval(planSegments)$

```
1:  nodeVisits = ∅
2:  for planSegment ∈ planSegments do
3:      agent ← planSegment.first
4:      segment ← planSegment.second
5:      for i = 1 → length(segment) do
6:          node ← segment[i]
7:          if nodeVisits[node] isempty then
8:              nodeVisits[node].addToTop(agent, segment, i)
9:          if previous visit in node is not from agent then
10:             nodeVisits[node].addToTop(agent, segment, i)
11:         else
12:             firstSeg ← nodeVisits[node].top()
13:             secondSeg ← segment
14:             startLoop ← nodeVisits[node].top().pos
15:             endLoop ← i
16:             removeLoop(firstSegment, startLoop, secondSeg, endLoop)
```

5.3.3 Implementation of concurrent plan building

As described in Section 5.2.6 the concurrent plan building is an essential part of the algorithm because otherwise the agents always move one at a time which in many instances results in plans that are longer than necessary because of agents that are waiting.

In our implementation several versions of concurrent plan building were tested. These implementations can be divided into two groups:

1. The first type of concurrent plan building is the one described in Section 5.2.6. For this type two versions were implemented. Both versions consider each segment individualy and then shift it in time until suitable start time is found. First version places the segment at the end of the plan and then shifts it back in time until collision occurs. Second version on the other hand places the segment to the time when the corresponding agent last stopped and then shifts it later in time until no collision occurs.

2. The second type of proposed concurrent plan building techniques are heavily inspired by algorithm previously described in Chapter 3. Three versions of this approach were implemented but all follow the same idea. The original algorithm is given a start and end points and tries to find a path through time intervals on given resource graph to connect these points. The main idea of this proposed concurrent building is to modify this algorithm in such a way that instead finding path through time intervals on nodes of the entire graph, the algorithm is given sequence

of resource nodes i.e. segments through which it needs to find a path. These three versions differ in the type of search used: Depth first search, Breadth first search and A*.

Chapter 6

Multi-robot Discrete Rapidly-Exploring Random Tree

This chapter describes sampling-based algorithm called Multi-Robot discrete Rapidly-Exploring Random Tree [3] (MRdRRT) which is an adaptation of a well known rapidly-exploring random tree (RRT) algorithm [4] for a discrete space (graph) embedded in Euclidean space. As opposed to *A*-based* approaches the sampling algorithms work well in high-dimensional configuration spaces. The dRRT algorithm goes through composite roadmap that may possibly have an exponential (in a number of agents to be coordinated) number of neighbours. High efficiency of traversal is achieved by using only partial information about the roadmap. Only one neighbouring node is considered in each step, which enables to find solutions for given scenarios while exploring only small fraction of composite space.

The RRT algorithm is discussed in the Section 6.1 of this chapter. The Section 6.2 is dedicated to description of necessary modifications to RRT that allow it to work on discrete graph. The Section 6.3 follows up on the second section and explains further modifications that enable dRRT to be utilized in multi-agent use resulting in MRdRRT. Implementation of the basic algorithm is discussed in the Section 6.4 as well as several modifications proposed by this thesis which were inspised by modifications of RRT such as growing two trees instead of one or "smarter" generation of random sample. The main contributions of this thesis are proposed modifications to MRdRRT that are inspired by changes introduced in RRT*[2] that bring multi-agent solution of MRdRRT closer to optimum. These modifications include new version of oracle and a new step called rewiring, that attempts to improve the structure of the tree in every step. All of these changes are described in the Section 6.5.

6.1 Rapidly-exploring random tree

A Rapidly-exploring random tree(RRT)[4] is a well-known sampling based algorithm which efficiently searches nonconvex high-dimensional spaces by building a tree which grows towards randomly generated samples from the search space. The main loop of the algorithm can be seen in Alg. 10. It starts with a tree containing only initial configuration s (line 1) and the algorithm

expands the tree (Alg. 11) in every step. The next step of the algorithm is the connector whose function is to attempt to connect the newly added configuration q_{new} with the terminal state t. If the connector succeeds the final path is obtained by concatenating the path obtained by connector and backtracking steps in the tree from q_{new} to the initial configuration s (line 5). For example the easiest connector can be implemented as a check if any obstacle intersects the line between newly added vertex q_{new} and terminal node t. If no intersection if found then the line from q_{new} to t is returned.

The expansion function is seen in Algorithm 11. The procedure repeats following steps N times where N is a parameter. The first step is randomly drawing a sample q_{rand} from search space C (line 2). RRT then finds the nearest neighbour q_{near} of the generated sample in the tree (T) (line 3) and generates a new configuration by expanding the tree from q_{near} towards the sampled point q_{rand} without violation any constraints, e.g. avoiding collisions with obstacles (line 4). This configuration can be generated in several ways such as having a fixed step size and making the step towards q_{rand} towards r_{near} or linking q_{rand} and q_{near} directly in which case q_{rand} becomes q_{new}.

Algorithm 10 RRT

1: $\mathcal{T}.init\,(s)$
2: **loop**
3: $\quad EXPAND\,(\mathcal{T})$
4: $\quad \mathcal{P} \leftarrow CONNECT\_TO\_TARGET\,(\mathcal{T}, t)$
5: $\quad$ **if** $not\_empty(\mathcal{P})$ **then return** $RETRIEVE\_PATH\,(\mathcal{T}, \mathcal{P})$

Algorithm 11 $EXPAND\,(\mathcal{T})$

1: **loop**$i = 1 \leftarrow N$
2: $\quad q_{rand} = RANDOM\_SAMPLE\,()$
3: $\quad q_{near} \leftarrow NEAREST\_NEIGHBOUR\,(\mathcal{T}, q_{rand})$
4: $\quad q_{new} \leftarrow newConf\,(q_{near}, q_{rand})$
5: $\quad \mathcal{T}.add\_vertex\,(q_{new})$
6: $\quad \mathcal{T}.add\_edge\,(q_{near}, q_{new})$

6.2 Discrete RRT

A discrete rapidly-exploring random tree (dRRT)[3] is a modification of the RRT algorithm for pathfinding in implicitly given graphs embedded in Euclidean space. The graph can be viewed as an approximation of the relevant portion of Euclidean space and its traversal as an exploration of its subspace. Let $G = (V, E)$ be a graph, where every $v \in V$ is embedded in a point in Euclidean space $\mathbb{R}^d$ and every edge $(v, v') \in E$ is a line segment connecting the points. Given two vertices $s, t \in V$ the dRRT searches for a path in G from s to t. Just like RRT, the dRRT grows a tree rooted in

s by iteratively adding new points to the tree while also trying to connect to t without violating any constraints, e.g. collision with environment. The growth is achieved by randomly sampling a point in the composite space and then extending the current tree towards this sample. In the discrete case the newly added vertices and edges are taken from G as there are no new vertices nor edges created during the process. Given implicitly represented graph G, the information about neighbours of already visited nodes is retrieved by a technique called oracle.

6.2.1 Oracle technique for querying the implicit graph

In order to generate neighbor nodes of already visited nodes dRRT uses technique called oracle. Without loss of generality consider that G is embedded in $[0,1]^d$. For two points $v, v' \in [0,1]^d$ the $\rho(v, v')$ denotes a ray that begins in v and goes through v'. $\angle_v(v', v'')$ given three points $v, v', v'' \in [0,1]^d$ denotes the (smaller) angle between $\rho(v, v')$ and $\rho(v, v'')$. The way the oracle is used is given sample point u it returns the neighbour v' of v such that angle between rays $\rho(u, v')$ and $\rho(v, v')$ is minimized. This can be defined as

$$\mathcal{O}_D(v, u) := \operatorname*{argmin}_{v' \in V} \{\angle_v(u, v') \mid (v, v') \in E\}.$$

6.2.2 dRRT description

At the first glance dRRT has similar structure to the RRT(see alg.10). The algorithm starts in the initial node s (line 1) and iteratively grows a tree (which is a subgraph of G). The growth is driven by expansion towards a randomly generated sample while avoiding all conflicts (line 3). Additionaly, the algorithm tries to connect to end node t in every iteration (line 4), if connection to t is possible, meaning there exists a path from newly added node to t, then the algorithm terminates and retrieves the path from built tree.

On the other hand, the expansion step (see Alg.12) of the algorithm is different from classical RRT in a sense that it only expands to vertices of G. Every time expansion is called it runs following steps N times where N is set parameter: Random sample point $r_{rand} \in [0,1]^d$ is generated (line 2) and the nearest neighbour q_{near} of that sample that is already in the tree is found (line 3). After that, the oracle $\mathcal{O}_\mathcal{D}$ is queried (line 4) to find a new point $q_{new} \in V$ that extends the tree towards q_{rand} from q_{near}. Once q_{new} is obtained, it is checked (line 5) whether it is already present in the tree. In case it is not present it is added together with the respective edge (q_{near}, q_{new}) to the tree (lines 6,7).

After each expansion step, the algorithm tries to connect to the node t using $CONNECT\_TO\_TARGET$(Alg.13) operation. This operation tries to connect t with its nearest neighbours $q \in \mathcal{T}$ using $LOCAL\_CONNECTOR$ method. Once the $CONNECT\_TO\_TARGET$ operation succeeds the

$RETRIEVE\_PATH$ operation is called, which concatenates path from s to q with the path $\mathcal{P}$.

Algorithm 12 $EXPAND\,(\mathcal{T})$ for dRRT

```
for i = 1 → N do
    q_rand = RANDOM_SAMPLE()
    (q_near ← NEAREST_NEIGHBOUR(T, q_rand))
    q_new ← O_D(q_near, q_rand)
    if q_new ∉ T then
        T.add_vertex(q_new)
        T.add_edge(q_near, q_new)
```

Algorithm 13 $CONNECT\_TO\_TARGET\,(\mathcal{T}, t)$

```
for q ∈ NEAREST_NEIGHBOURS(T, ⊔, K) do
    P ← LOCAL_CONNECTOR(q, t)
    if not_empty(P) then return P
return ∅
```

6.2.3 Local connector

It is possible that the tree $\mathcal{T}$ will, if given sufficient time, eventually reach t during the expansion phase. It is however unlikely for larger problems and because of that it is necessary to use $LOCAL\_CONNECTOR$. Given two vertices $q_0, q_1 \in G$ this method tries to search for a path between q_0 and q_1 without violating any constraints. It is assumed that connecting two nearby samples requires less effort, than solving the whole initial problem. The assumption is also that the local connector is effective on restricted pathfinding problems only.

6.3 Multi-Robot discrete Dapidly-exploring Random Tree

This section describes multi-agent adaptation of dRTT called Multi-Robot discrete Rapidly-exploring Random Tree (MRdRRT). Specifically necessary changes are described to each step of dRRT for it to work in pathfinding in a composite roadmap G which is embedded in joint $C - space$ of m agents.

6.3.1 MRdRRT description

On high level, MRdRRT executes the same operations as dRRT. Steps necessary in each operation, however, slightly vary. A new sample s for multi-agent scenario is generated in the expansion phase in a similar manner as for a single agent. The only difference is that a sample $s_{1,n} \in [0,1]^d$ is generated for each agent and all of them are then concatenated into s. Oracle

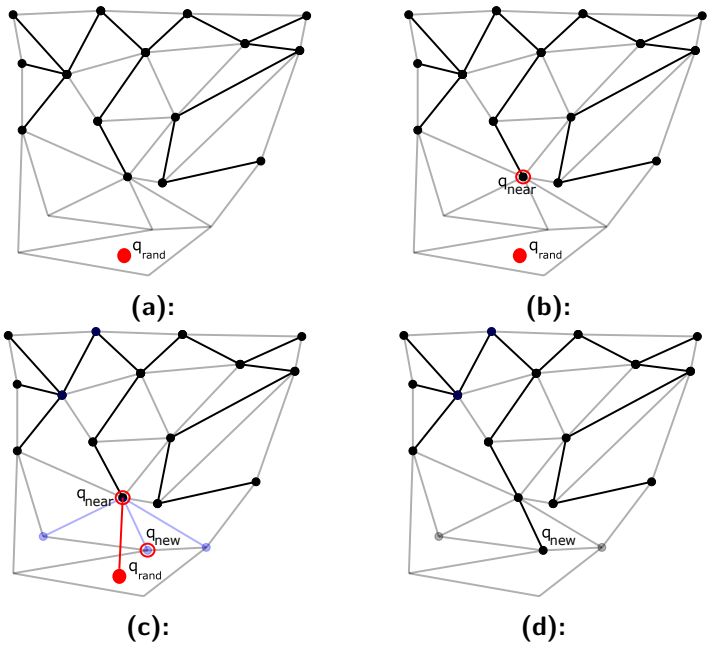


Figure 6.1: The illustration of expansion phase of $dRRT$ algorithm. The tree $\mathcal{T}$ is depicted in black edges and vertices. Gray vertices and edges depict unexplored portions of the graph. In (a) a random sample q_{rand}(red) is generated. (b) nearest vertex q_{near} already in $\mathcal{T}$ using Euclidean distance is found. (c) neighbour vertex q_{new} of q_{near} is found such that its direction from q_{near} is closest to the direction to q_{rand} from q_{near}. (d) vertex q_{new} is added to the tree $\mathcal{T}$

and the $CONNECT\_TO\_TARGET$ methods for multi-agent scenario are discussed in more detail in the following sections.

6.3.2 Oracle $\mathcal{O}_D$ for multi-agent scenario

As discussed in section 6.2.1 the oracle given $q \in V$ and a random sample s, $\mathcal{O}_{\mathcal{D}}(C, q)$ returns C' such that C' is a neighbouring node of C in $\mathcal{G}$ and any other C'' of C the $\rho(C, q)$ forms a smaller angle with $\rho(C, C')$ than with $\rho(C, C'')$. It is needed for multiple agents to define $C(r_i)$ as the $C - space$ of a agent r_i. Let $q = (q_1, ..., q_m)$ where $q_i \in C(r_i)$ and let $C = (c_i, ..., c_m)$ where $c_i \in V_i$. In order to find a neighbour of C it is necessary to first find the best neighbour for each agent and concatenate them into a candidate neighbour C' for C. The next step is to validate whether (C, C') forms a valid edge in G i.e. if no constraint is invalidated during this transition. If valid, the new node C' is returned, an empty set is returned otherwise, the sample s is ignored and a new sample is generated in the expansion phase.

6.3.3 Local connector for multi-agent scenario

The goal of the local connector is to connect two given vertices of a graph as discussed in section 13. For a multiple agent scenario a framework described by van den Berg et. al [13] is used. Given two vertices $\mathcal{V} = (v_1, ..., v_m) \in G$

and $\mathcal{V}' = (v'_1, ..., v'_m) \in G$ the path π_i is found for each agent r_i on G_i from v_i to v'_i. After this the connector attempts to find an ordering of agents in which agents move one at a time meaning agent r_i does not leave its starting position v_i untill all agents with higher priority reached their final positions on their respective paths while avoiding collisions. When all agents with higher priority reach their destinations the agent r_i moves along its path while other agents stand still. Any algorithm that finds such ordering can be used, but in this case the technique discussed in [3] was used in our implementation.

The priorities are assigned according to Algorithm 14. The procedure starts with a graph $\mathcal{I}$ containing m vertices representing m agents (line 2) but no edges. For each agent r_i it is then checked whether any other agent has its start position or end position on a path of the agent r_i (line 4). After that for every colliding agent r_j it is checked if his start position v_j is on the path of agent r_i in which case agent r_j must have higher priority than r_i so edge $(r_j \rightarrow r_i)$ is added to the graph $\mathcal{I}$ (line 7). Otherwise the coliding agent must have its end position v'_j on the path and in that case the edge $(r_i \rightarrow r_j)$ is added because the agent r_i must have higher priority than r_j (line 9). When all agents are processed it is checked if the graph $\mathcal{I}$ is acyclic (line 10), in which case the topological sort of the graph is returned. An empty set is returned otherwise (line 11).

Algorithm 14 Ordering of agents for local connector

```
1:  paths ← find_paths(V, V')
2:  I ← (R, ∅)
3:  for i = 1 → m do
4:      π ← colliding_agents (paths [i])
5:      for r_j ∈ π do
6:          if pos (r_j) == V [j] then
7:              I.add_edge ((r_j, r_i))
8:          else
9:              I.add_edge ((r_i, r_j))
10: if I.is_acyclic () then return I.topological_sort ()
11: elsereturn ∅
```

6.4 Implementation

This section discusses implementation of MRdRRT algorithm along with changes made to the basic algorithm. Because the basic oracle implementation ran into situations that caused it to repeatedly create conflicting configurations which caused a large number of samples to be discarded, an improved version of oracle was implemented that purposefully tries to generate new valid node and only when it cannot generate it, the sample is discarded. Different methods of getting random samples were also implemented and are described in this section. The modifications of local connector were heavily inspired

by overlaping path segments from algorithm which is described in Chapter 5. The last major improvement that was implemented was method that grows two trees instead of one. This improvement was suggested because in the original MRdRRT the distance in the composite space C between newly created node q_{new} and terminal node t can be large and thus lowers the chance of local connector success. The idea behind two trees is that by trying to always connect two newly generated nodes both of which are generated towards the same sample shortens the distance that needs to be bridged by local connector and thus increases the chance of local connector success.

6.4.1 Basic implementation

The basic high level implementation of MRdRRT followed the structure of the pseudocode in Alg. 15. The main difference is that the expansion phase always generates one new node instead of N and this local connector is always called only on this newly generated node instead of on N nearest neighbours of terminal node t. This change was proposed because the base version tended to check the same nodes multiple times and always checked the same number of them. These modifications allow that all newly added nodes are checked if they can be connected to the terminal node and thus increasing the chance to success.

Algorithm 15 MRdRRT implementation

```
1: T.init(s)
2: loop
3:     P ← CONNECT_TO_TARGET(T, t)
4:     if not_empty(P) then
5:         Break
6:     EXPAND(T)
   return RETRIEVE_PATH(T, P)
```

Random sample generation

The expansion phase generates random sample as its first step to get a sample node towards which the generated tree is extended. For this reason it depends on random sample generation and because of this several possible ways how to generate random samples in the expansion phase were implemented including:

- Random samples from the bounding box of $\mathcal{G}$
- Random samples from the bounding box of $\mathcal{G}$ with addition of chance to wait
- Randomly chosen vertices of $\mathcal{G}$
- Randomly sampled points from neighbourhood of shortest paths for every agent. Every such point q has the property that $shortestPath(s, q) +$

$shortestPath(q,t) \leq shortestPath(s,t) + N$ where N is a parameter that can be arbitrary non-negative value.

The first approach is what the original MRdRRT uses but it proved inefficient in maps with tight spaces as it would not allow agents to stay put as their next action and would not find solution in situations where standing still was required for one of the agents. Because of this the slightly improved version was implemented, where for every agent there is a chance for the oracle to make a given agent stay put in newly generated vertex. The third version of random sample generation tried to incorporate the chance to stay put directly into the random sample generation by generating random vertices from $\mathcal{G}$. If the same point is generated as in the nearest neighbour then the agent stays put. The last iteration of random sample generation is the improvement on the previous one with the change to vertices that are generated from G. Before the main loop of path finding this method finds shortest paths for every agent and after that it finds points from which the sample is generated by going over all points p and checking if

$$shortestPath(s,q) + shortestPath(q,t) \leq shortestPath(s,t) + N$$

where N can be arbitraty non-negative value.

Nearest neighbour search

Because of the significant impact on performance of brute-force search for the nearest neighbour and also because exact nearest neighbour is not required the Fast Library for Approximate Nearest Neighbour(FLANN)[7] was used. This library was specificaly chosen because of its performance but also because of its ability to add new points to the search index during the algorithm run.

Oracle $\mathcal{O}_D$

The first implementation of Oracle $\mathcal{O}_D$ was made according to the Section 6.3.2 with the addition of chance for a agent to stay put discussed in the Subsection 6.4.1. After generating a sample this way the new node had to be checked for collisions. This method proved inefficient along with its associated random sample generation. Implementation of the next version of a random sample generation resulted in improvements in the second oracle version. This version iterates over positions of all agents v and tries to generate a new step v' for them towards sample point u while avoiding collisions and also minimizing the $\angle_v(u, v')$ by keeping a list of collision configurations that need to be avoided. The last iteration of the method Oracle only adds to the previous one random ordering of agents in each query to avoid getting stuck.

6.4.2 Local connector

Local connector implementation was made in the same way as described in Alg. 6.2.3 while also including postprocessing which improves a number

of overall plan steps by reducing waiting required by the agents. The first step is to find shortest paths for all agents from the current agent position v to the end position v'. In the case where end positions are also target positions t, the preprocessing before the start of algorithm was implemented. This preprocessing includes calculation and storing of paths leading from all nodes of the graph to all target positions. The next step is generating the dependancy graph and acyclicity check described in Section 6.3.3. In the case when the graph is acyclic the order in which the agents move is generated by topologically sorting the nodes in the dependancy graph. Kahn's algorithm[1] was used to topologically sort the graph in our implementation. This process creates the order in which agents are able to get into their end positions without collisions if they move one by one. The main drawback of this is that agents might wait unnecessarily. The standard connector implementation can be seen in Fig. 6.4a. The goal of improvement is to shift the respective intervals p_i representing individual paths as early in time as possible. This can be done by considering each interval p_i individualy, setting its position to its start position (line 2) v_i and then shifting it to the right later in time (line 6) until no conflict arises(see Alg.16). The result of this procedure can be seen in Fig. 6.4b.

Algorithm 16 Ovelaping local connector intervals

1: $\mathcal{I} = LOCAL\_CONNECTOR\,(v, v')$
2: $OverlapingPlan = SetStartPositions\,(v)$
3: **for** *interval* i $\in \mathcal{I}$ **do**
4: $\quad t = 1$
5: $\quad$ **while** $CONFLICT\,(OverlapingPlan, i, t)$ **do**
6: $\quad\quad t = t + 1$
7: $\quad OverlapingPlan.add\,(i, t)$

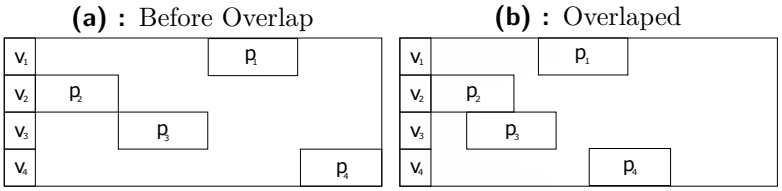


Figure 6.3: Shifting path segments

6.4.3 Two-tree version implementation

The local connector in the basic version of MRdRRT may sometimes have troubles connecting terminal node t with the newly added node q_{new} because their distance may be too big in the composite space. The main idea of two-tree implementation is to grow two trees instead of one. One from the initial configuration s and one from terminal configuration t. Main differences

to the basic MRdRRT are in the expansion phase (see Alg.17) where one random sample is generated and new nodes $q_{newStartNode}$ and $q_{newEndTree}$ are created using the oracle. If none of them is already in its respective tree they are added to them. Local connector then tries to connect these two new nodes together instead of one new node and terminal node. This process should help bring the nodes that need to be connected closer together and increase the chance of success for local connector.

Algorithm 17 $EXPAND\_2TREES\,(\mathcal{T}_{start}, \mathcal{T}_{end})$

1: **for** $i = 1 \rightarrow N$ **do**
2: $q_{rand} = RANDOM\_SAMPLE\,()$
3: $q_{nearStartTree} \leftarrow NEAREST\_NEIGHBOUR\,(\mathcal{T}_{start}, q_{rand})$
4: $q_{nearEndTree} \leftarrow NEAREST\_NEIGHBOUR\,(\mathcal{T}_{end}, q_{rand})$
5: $q_{newStartTree} \leftarrow \mathcal{O}_D\,(q_{nearStartTree}, q_{rand})$
6: $q_{newEndTree} \leftarrow \mathcal{O}_D\,(q_{nearEndTree}, q_{rand})$
7: **if** $q_{newStartTree} \notin \mathcal{T}_{start}\ AND\ q_{newEndTree} \notin \mathcal{T}_{end}$ **then**
8: $\mathcal{T}.add\_vertex\,(q_{new})$
9: $\mathcal{T}.add\_edge\,(q_{near}, q_{new})$

6.5 Steps towards optimality - RRT*

The main strength but also weakness of the RRT algorithm comes from the fact that it searches for any path leading from an initial configuration s to a terminal configuration t. This fact combined with the random nature of RRT often results in longer paths than necessary. For this reason the algorithm RRT*[2] was introduced which improves this behavious. RRT* (Alg.18) which is known to converge to optimal solution makes improvements to the expansion phase of RRT and it also introduces a new step called rewiring, which locally improves the structure of the tree $\mathcal{T}$.

The improved expansion phase(see Alg.19) adds one additional step to the expansion phase of original RRT. The algorithm after generating new random sample q_{rand} (line 2), finding a nearest neighbour q_{near} (line 3) and generating a new configuration q_{new} (line 4) iterates over all nodes of $\mathcal{T}$ that are within radius r centered in q_{new} and connects it to the one which minimizes the path length to the root of the tree from q_{new} if chosen as a predecessor. This steps purpose is to drive the expansion in such a way that tries to eliminate unnecesarily long paths.

The Rewiring (see Alg. 20) is the new step introduced in RRT* and its purpose is to localy revise the structure of the tree $\mathcal{T}$ by considering the newly added node q_{new} from expansion phase as a transit node. Rewiring iterates over all nodes within the radius r of node q_{new} and checks if a path leading to them from the root would be shorter if their predecessor was the node q_{new}.

Algorithm 18 RRT* algorithm

1: $\mathcal{T}.init\,(s)$
2: **loop**
3: $\quad EXPAND\,(\mathcal{T}, r)$
4: $\quad REWIRE\,(\mathcal{T}, r, q_{new})$
5: $\quad \mathcal{P} \leftarrow CONNECT\_TO\_TARGET\,(\mathcal{T}, t)$
6: $\quad$ **if** $not\_empty(\mathcal{P})$ **then return** $RETRIEVE\_PATH\,(\mathcal{T}, \mathcal{P})$

Algorithm 19 RRT^* $EXPAND\,(\mathcal{T}, r)$

1: $q_{rand} = RANDOM\_SAMPLE\,()$
2: $q_{near} \leftarrow NEAREST\_NEIGHBOUR\,(\mathcal{T}, q_{rand})$
3: $q_{new} \leftarrow newConf\,(q_{near}, q_{rand})$
4: $q_{bestPred} = -1$
5: $d_{best} = \infty$
6: **for** $n \in IN\_RADIUS(\mathcal{T}, q_{rand}, r)$ **do**
7: $\quad$ **if** $RootDist\,(n) + length\,(edge\,(n, q_{new})) < d_{best}$ **then**
8: $\quad\quad d_{best} = RootDist\,(n) + length\,(edge\,(n, q_{new}))$
9: $\quad\quad q_{bestPred} = n$
10: $\mathcal{T}.add\_vertex\,(q_{new})$
11: $\mathcal{T}.add\_edge\,(q_{bestPred}, q_{new})$

6.5.1 RRT* modifications towards discrete multi-agent scenario

For the RRT* to work in a multi-agent discrete scenario the expansion phase of MRdRRT algorithm (see Section 6.3) needs to be modified. As discussed in the Section 6.5 the change to expansion phase consists of connecting the new node q_{new} to a node already in the tree $\mathcal{T}$ that minimizes the distance traveled from the initial configuration s. For the purpose of distance measurement between nodes in a multi-agent scenario the sum of Euclidean distances traveled by each agent was used. In the original modification of the expansion phase the additional step consists of checking nodes in the radius around the new node q_{new} for the "best" predecessor and then connecting q_{new} to it. However, in the multi-agent discrete scenario (Alg. 21) the computational requirements to perform a similar task are much higher because it would require to run a local connector method from section 6.3.3 on each node in the radius and then perform the distance to root check. The expansion phase was thus modified in such a way that used nearest neighbour search instead of radius (line 2). The key difference is that in the first step of expansion the random sample q_{rand} (line 1) is generated but after that the new node q_{new} is not created from the nearest neighbour of q_{rand}. Instead, N nearest neighbours of q_{rand} are iterated over (lines 6-11), a new node q_{new} is generated from them using oracle $\mathcal{O}_D$, but not added into the tree. Each q_{new} is checked for the distance traveled through tree $\mathcal{T}$ towards the root s and only a node that minimizes this distance is connected to its corresponding predecessor.

Algorithm 20 $REWIRE\,(\mathcal{T}, r, q_{new})$

for $n \in IN\_RADIUS\,(\mathcal{T}, r, q_{added})$ **do**
  if $RootDist\,(v) + length\,(edge\,(q_{new}, n)) < RootDist\,(n)$ **then**
    $n.predecessor = q_{new}$

Rewiring step of RRT* localy revises a structure of $\mathcal{T}$ by checking whether nodes in the radius r around newly added node q_{new} had distance traveled towards the root node shorter if they were reconnected to q_{new}. This step was modified for the use in a multi-agent discrete case by omitting the radius and using N nearest neighbours search instead. Because these neighbouring configurations q_{near} might not be direct neighbours of q_{new} in the composite graph G the local connector is used to obtain a path between these two nodes. If local connector fails to find the path, the neighbour is immediately skipped. In the case of local connectors success in finding a path p between q_{new} and q_{near} it is checked if a length of path from the root to q_{new} concatenated with the path p and node the q_{near} is shorter than a distance traveled through $\mathcal{T}$ from the root to q_{near}. If it is shorter then all nodes of p are added to $\mathcal{T}$. The first node of p is connected as successor of q_{new} and the last node of p is chosen as a new predecessor of q_{near}.

Algorithm 21 $MRdRRT^*\ EXPAND\,(\mathcal{T}, r)$

1: $q_{rand} \leftarrow RANDOM\_SAMPLE\,()$
2: $NNs \leftarrow getNearestNeighbours(q_{rand})$
3: $q_{bestPred} = -1$
4: $d_{best} = \infty$
5: $q_{newBest} = \emptyset$
6: **for** $q_{near} \in NNs$ **do**
7: $\quad q_{new} \leftarrow \mathcal{O}_D\,(q_{near}, q_{rand})$
8: $\quad$ **if** $RootDist\,(q_{near}) + distance\,(q_{near}, q_{new}) < d_{best}$ **then**
9: $\quad\quad d_{best} = RootDist\,(q_{near}) + distance\,(q_{near}, q_{new})$
10: $\quad\quad q_{bestPred} = q_{near}$
11: $\quad\quad q_{newBest} = q_{new}$
12: $\mathcal{T}.add\_vertex\,(q_{newBest})$
13: $\mathcal{T}.add\_edge\,(q_{bestPred}, q_{newBest})$

Algorithm 22 $REWIRE\,(\mathcal{T}, r, q_{new})$

$NNs \leftarrow getNearestNeighbours(q_{new})$
for $q_{near} \in NNs$ **do**
  $p \leftarrow LOCAL\_CONNECTOR\,(q_{new}, q_{near})$
  if $RootDist\,(v) + length\,(edge\,(q_{new}, n)) < RootDist\,(n)$ **then**
    $n.predecessor = q_{new}$

Chapter 7

Experiments

This chapter elaborates on the experiments performed on the implemented algorithms from Chapters 3, 4, 5 and 6. These will be further referenced to as *mors*, *icts*, *peasgood* and *MRdRRT*. Section 7.1 elaborates on the methodology of the experiments performed in Chapters 7.2, 7.3,7.4 and 7.5. Section 7.2 discusses the results obtained from the experiments done on the *mors* algorithm. Results of experiments on the *icts* algorithm can be found in Section 7.3. Section 7.4 presents the results from experiments on *peasgood* algorithm, while Section 7.5 introduces the results obtained from experiments on different versions of *MRdRRT* algorithm and discusses the potential real-world use of the implemented algorithms.

The experiments described in this section were all performed on three types of maps that are because of their size available on the enclosed CD.

1. The first type of maps are the grid benchmark maps that are freely accessible at [10]
2. Three versions of maps from a real-world robot warehouse that differ in size.
3. Map used for testing of *icts* algorithm which is a 5x5 grid.

7.1 Methodology

At first every algorithm is tested how it scales with the number of agents placed on one map. For the algorithms that have had several versions of their steps implemented the experiments first deduce their best setting that is then used for the scaling experiments. Several performance indicators are measured during these experiments, such as:

- Number of time steps required until the last agent gets to his goal. This is further referenced as the length of the plan.
- Sum of distances traveled by particular agents.
- Number of iterations of the algorithm required if applicable.

- Number of failed plans. This means number of times the algorithm overstepped the maximum number of iterations or also if algorithm managed to plan only a partial number of agents.
- Time required to obtain the plan. For algorithms that include preprocessing the time is split into preprocessing time and planning time.

Once all scaling experiments are done, all algorithms are tested and compared on the same assignments on a map of a large warehouse and other selected maps.

7.2 Results of the mors algorithm

The experiment on the *mors* algorithm was performed on the largest map of the robotic warehouse with 3315 vertices. The starting number of agents is 10 and this number is incremented in every cycle by 5 until 300 agents are present. The algorithm was tested on 50 randomly generated assignments for every number of agents where every agent starts at time 0 and two agents can not have same start point or end point.

Because the algorithm is not complete it can happen that it will not be able to find a path for every agent in the assignment. The success rate of the algorithm was therefore also measured. For the testing purposes it is considered that if a path is not found for all agents, then the algorithm failed to find a plan for the given assignment.

One of the causes for a failure was in most cases non-sufficient length of the free time window in which an agent enters the free time window graph that caused its inability to expand into free time windows on neighbouring resources.

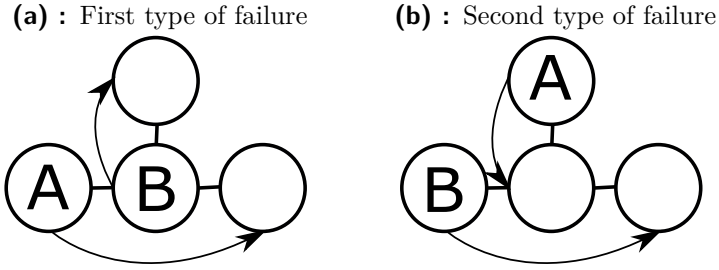


Figure 7.1: Examples of two types of situations where the *mors* algorithm fails to find a plan.

Another cause for a failure to find a path for an agent can be non-existence of such path. Because the algorithm always considers only one agent at a time it is likely to happen that currently planned agent can block a path of subsequent agents if he stays in his goal position and does not disappear from the map. An example of both types of a failure can be seen in Figure 7.1. The first figure shows the first type of a failure as if the agent A moves to his goal with each step taking 1 unit of time then the middle node will

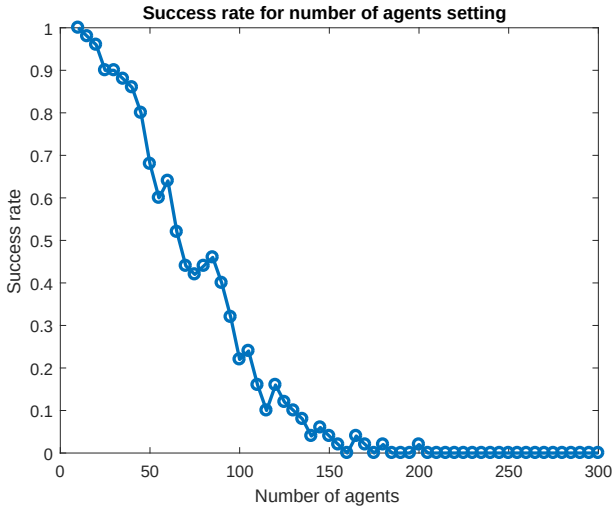


Figure 7.2: The *mors* algorithm success rate scaling

have two free time windows for the agent B - $[0, 1)$ and $[2, \infty)$. Because both agents have entry time 0 the agent B has to use the free time window $[0, 1)$ for his entry but because the step length is 1, this time window can not be used as its length is shorter and thus resulting in a failure. The second figure shows the second type of a failure. The agent A moves to its goal node in the middle which will leave only one free time window - $[0, 1)$. But the agent B can not use this window as it is not long enough for him to enter, traverse and leave which leads to the second type of failure where the path does not exist in the free time window graph for the current agent.

As seen in Figure 7.2. the success rate of the algorithm decreases rapidly with the increasing number of agents. This issue can be mitigated however by ensuring for example minimal distance between the start points of agents or if the agents disappear from the graph once they reach their goal positions.

Considering only the plans that contain a path for all agents the algorithm scales really well in terms of summed distance traveled by each agent as seen in Figure 7.3

The experiments show that the *mors* algorithm scales incredibly well both in sum of distances traveled but also in runtime but has a low success rate for higher number of agents.

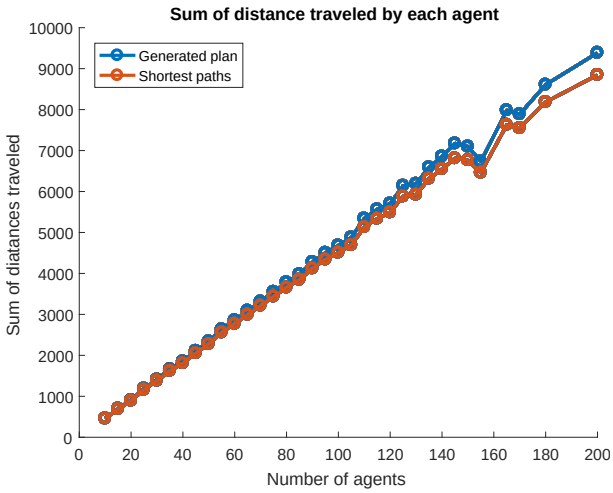


Figure 7.3: The *mors* algorithm scaling of a mean sum of distances traveled by each agent in *mors* algorithm. The shortest paths values are obtained by summing the shortest path length for each agent obtained by A* that ignore all other agents in the assignment.

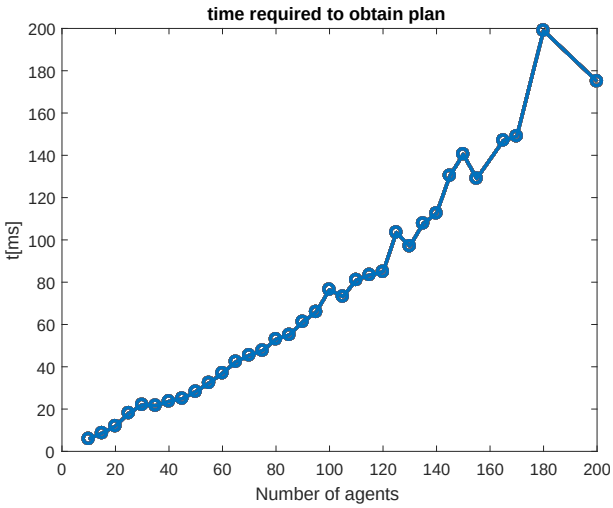


Figure 7.4: The *mors* algorithm scaling in terms of runtime.

7.3 Results of the icts algorithm

This section describes the experiment performed on *icts* algorithm. Due to its computational complexity it is unsuitable for bigger problems and thus it was tested on a 5x5 grid map with randomly generated start and end positions for increasing number of agents. The number of agents starts at 1 and is incremented by 1 until 10 agents are on the map. For every number of agents 5 different assignments were generated. No two start positions overlaped as well as no two end positions overlaped in these assignments. The results of this can be seen in Figure 7.5.

Because *icts* offers optimal solutions for given assignments the interesting thing to see in the results is the difference between optimal solution and the shortest one where all agents only follow their shortest paths to goal nodes. The algorithm also does not scale really well in terms of runtime which is expected because during the algorithm run all paths leading to goal with fixed length have to be found, stored and then plan is found in their cross-product which is a taxing operation.

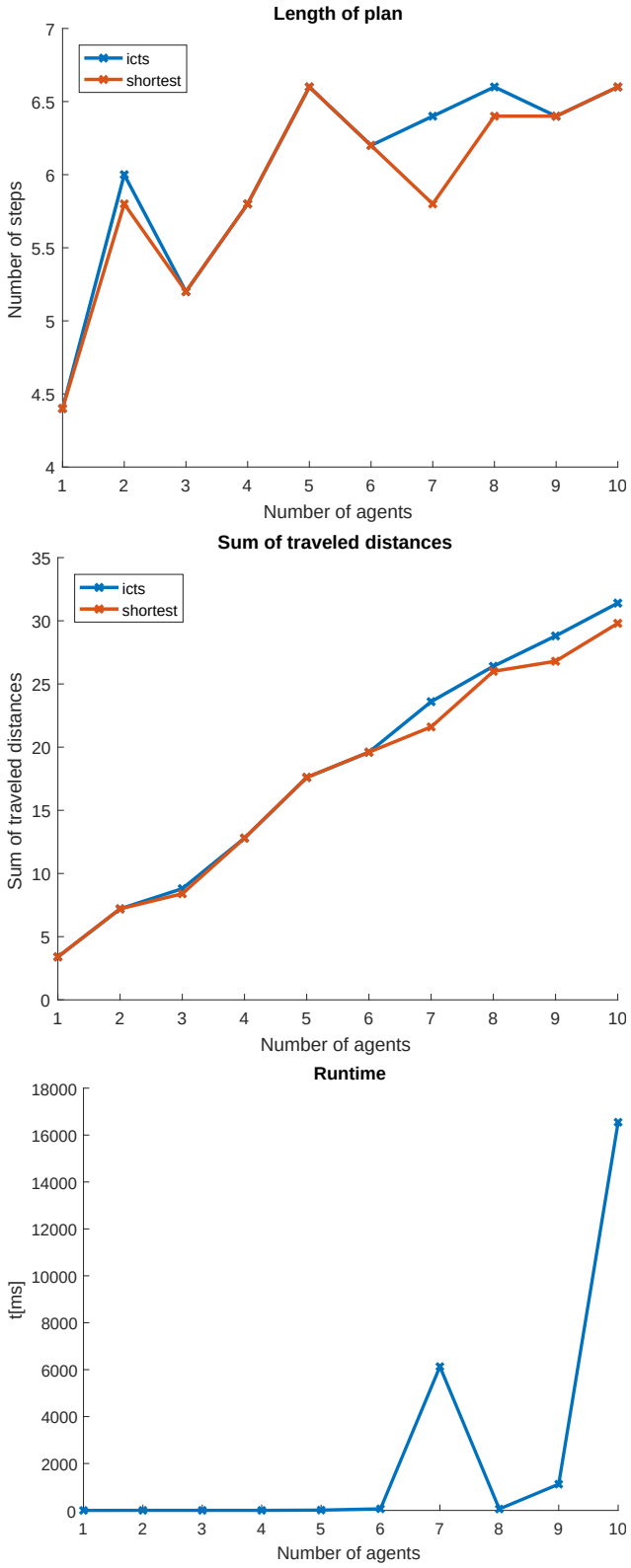


Figure 7.5: Scaling of the *icts* algorithm.

7.4 Results of the peasgood algorithm

Experiments on the *peasgood* were made on the largest map for the robotic warehouse which consists of 3315 vertices. The number of agents started at 10 and was increased in increments of 10 until 500 agents were reached. For each number of agents 10 different assignments were generated where no two start positions and no two terminal positions overlap. Because Chapter 5 introduced several modifications to the algorithm the goal of this section is to determine the ideal setting which is then used for the experiments in later sections. The first modifications are to the choice of free leaf used in the first phase of the algorithm. There are five versions for the leaf selection in total:

1. Random free leaf node
2. Nearest leaf to agent's position
3. Random free leaf in agent's goal subtree
4. Nearest free leaf in subtree of agent's goal
5. Nearest free leaf in subtree of agent's position

The results for this setting can be seen in Figure 7.6. For the purposes of this experiment the choice of concurrent plan building was set to use the version that places the segment to the end of the route of the corresponding agent and then shifts it later in time until no conflict is present.

It can be seen that the setting where every agent is sent to his nearest leaf performs the best out of the five variants because in all three observed metrics it performed the best. What is interesting to observe is the runtime performace of each setting. It was assumed that by moving agents directly into the free nodes of their subtrees the number of necessary steps would be reduced and thus the runtime would reduce. What can be seen is the opposite, which might be caused by the second part of the first phase that checks the paths to selected nodes for presence of other agents. If such agents are found, then the one furthest along the path is selected and moved to the selected free leaf node instead of the first considered agent. As a result of this operation the agents can be moved further from their goal nodes than they were at the start which causes the negative performance hit.

The second modifications are different versions of concurrent plan building which has also five different variants:

1. Place segments to the end and shift them earlier in time until conflict
2. Place segments to the end of the route of their respective agent and shift them later in time untill no conflict occurs
3. Depth-first search through time windows inspired by algorithm in Chapter 3

4. Breadth-first search variant of setting 3
5. A* search variant of setting 3

Results for the different settings in phase 4 of the algorithm can be seen in Figure 7.7. Sum of distances traveled is the same for every setting because concurrent plan building does not alter the paths agents take and thus is not included in this comparisson. The results show that placing each segment to the track end of its respective agent and then shifting it later in time offers the best performace out of the five options in terms of plan quality and second best performace in terms of runtime. The interesting part is the fact that in terms of plan quality the depth first search is identical to A* search and breadth first search is almost the same as placing the segment at the end of the plan and then shifting it back untill no collision occurs.

It is possible to say from the previous experiments that the best settings are using nearest free leaf node search in the first phase and placing the path segments to the end of their respective agent's path and shifting them later in time untill no conflict arises. All previous experiments were performed with the optional removal of loops from the plan. The results of experiments with the best setting that compare the results of the algorithm with and without the loop removal can be seen in Figure 7.8. Interesting observation in these results is that while loop removal helps to reduce the distances traveled by the agents, it starts to have a negative impact on the plan length as the number of agents in the system increases. This behaviour seems counter intuitive but possible explanation is that when the necessary maneuvers are removed from the individual agent plans it also decreases the maneuverability for the concurrent plan building and forces the individual path segments to be pushed much later in time then if the loops were not removed.

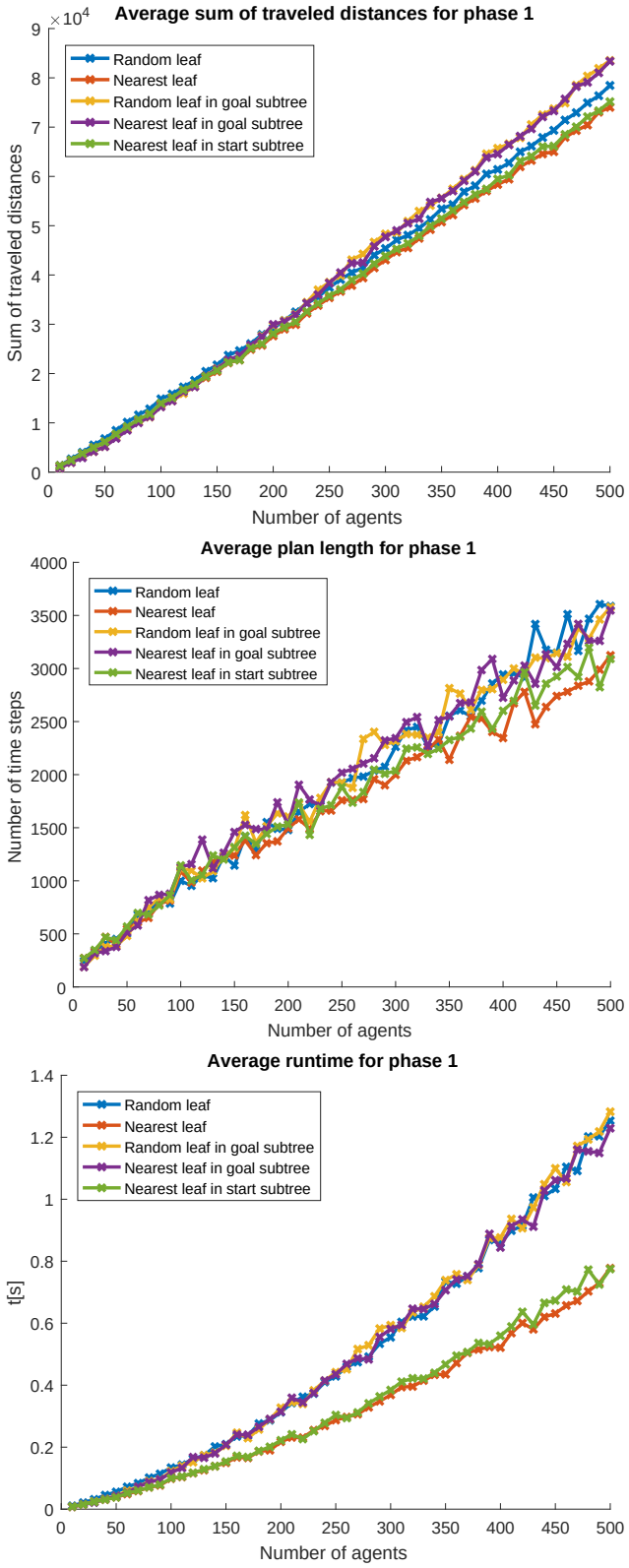


Figure 7.6: The *peasgood* algorithm comparison of different settings for phase 1.

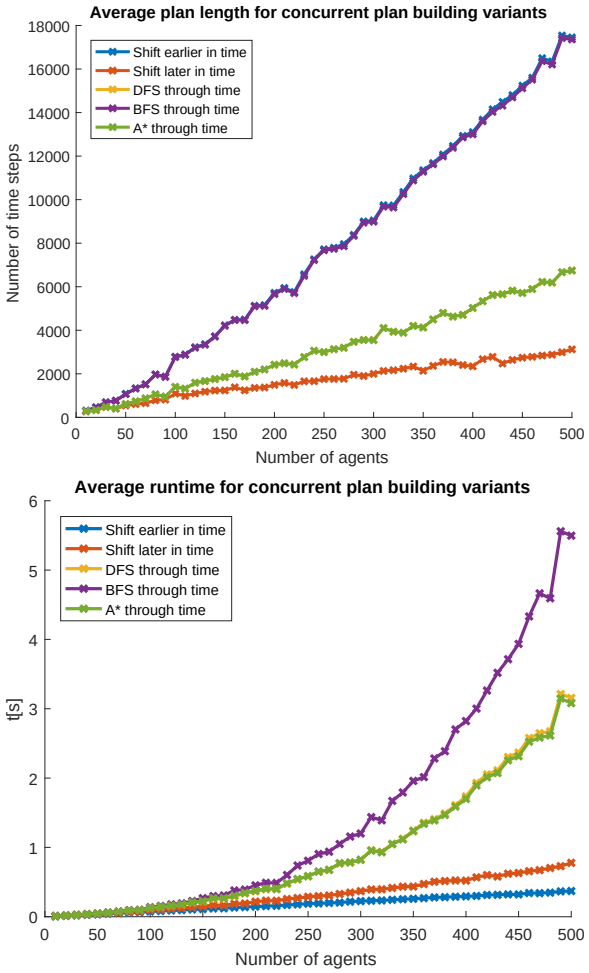


Figure 7.7: The *peasgood* algorithm comparison of different settings for concurrent plan building

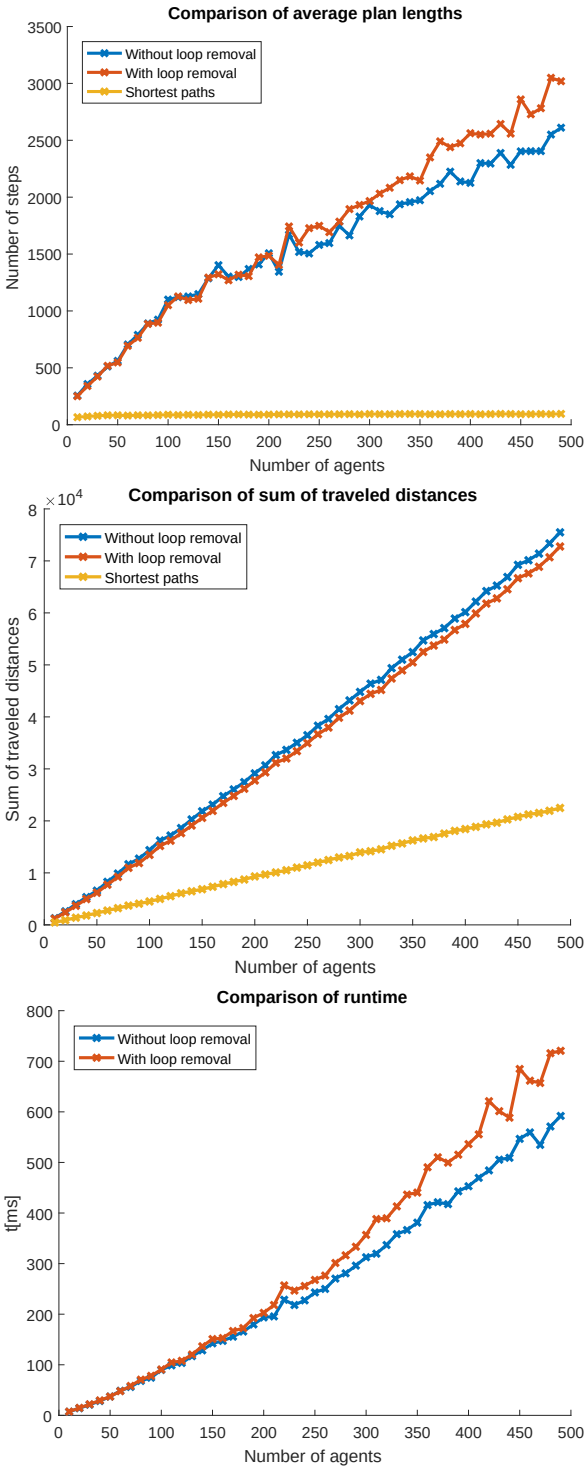


Figure 7.8: The *peasgood* algorithm scaling with increasing number of agents

7.5 Results of the MRdRRT algorithm

This section describes experiments performed on the *MRdRRT* algorithm. The Chapter 6 described several implemented versions of this algorithm such as the basic version that uses one tree, modified version that uses two trees that try to connect to each other and lastly the new approach presented in this Thesis that was inspired by the *RRT** algorithm. These variants are further refered to as *1-tree*, *2-tree* and *star* variants.

The experiments are performed by generating 10 assignments for 10 different numbers of agents from 10 to 100. Each of these assignments is then run 10 times and the displayed results are averages over each number of agents. All versions of the algorithm are tested on the *large* map of the robotic warehouse. The obtained results for the first two versions of the algorithm can be seen in Figure 7.9.

The results show that the basic *(*1-tree) implementation performs better than *2-tree* version on the selected map in terms of quality of the plan but at the cost of several times higher runtime and number of iterations.

To compare the proposed *star* version of the algorithm it is required to first determine the best setting for the parameter N, which is number of nearest neighbours that are checked during the search and substitutes the radius parameter used in the original RRT algorithm. The comparison of different settings for this parameter is displayed in Figure 7.10.

For this specific map the settings performed similarly but $N = 20$ is chosen for the future experiments as it provides good ratio between the complexity and quality.

Once the parameter N is chosen it is possible to make comparison between all three versions of the algorithm. This comparison can be seen in Figure 7.11.

The results show that the proposed *star* version of the algorithm performs significantly better than *1-tree*, *2-tree* variants but at the cost of a higher number of iterations and runtime. During the experiments the maximum number of iterations the algorithms were able to perform was set to one milion. Once one milion iterations is reached, it is considered that the algorithm failed. *1-tree*, *2-tree* variants never failed while the failure rate of the *star* variant can be seen in Figure 7.12.

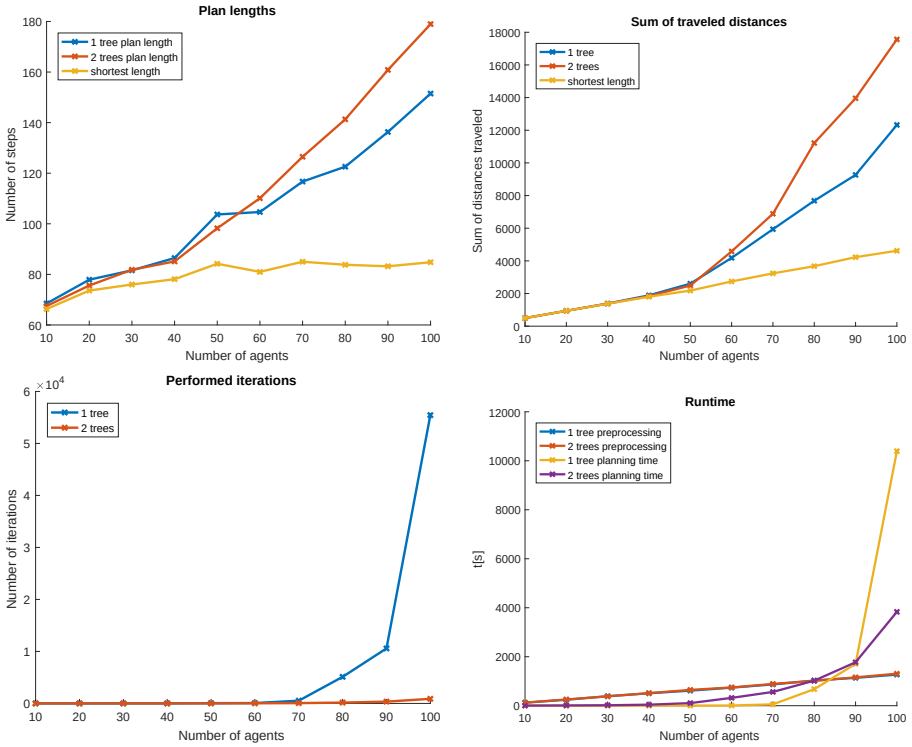


Figure 7.9: MRdRRT scaling with increasing number of agents

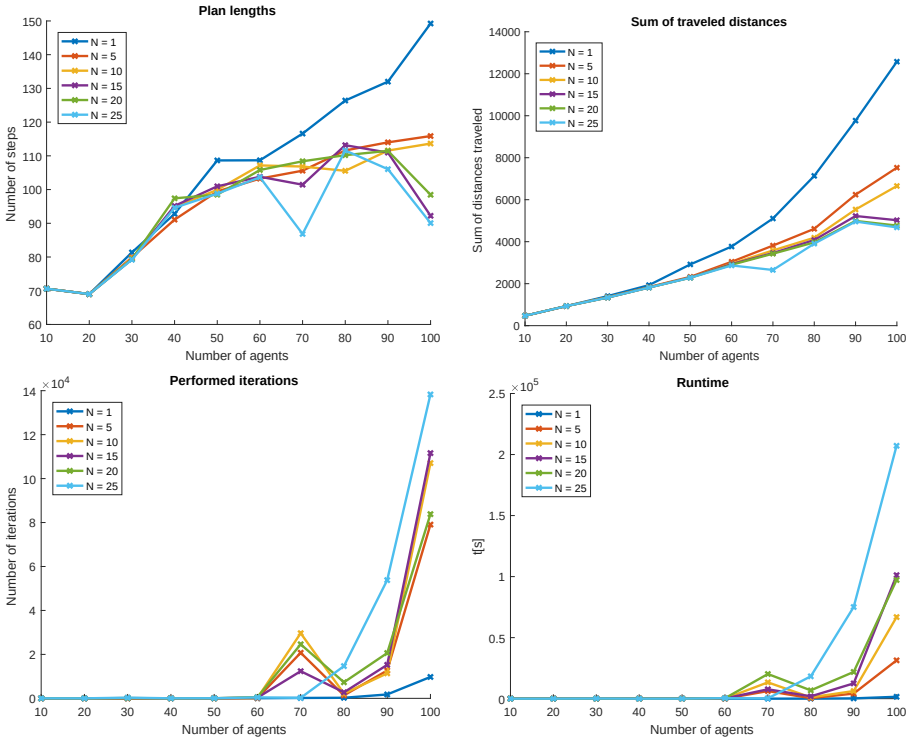


Figure 7.10: *MRdRRT star* scaling with increasing number of agents for different values of N

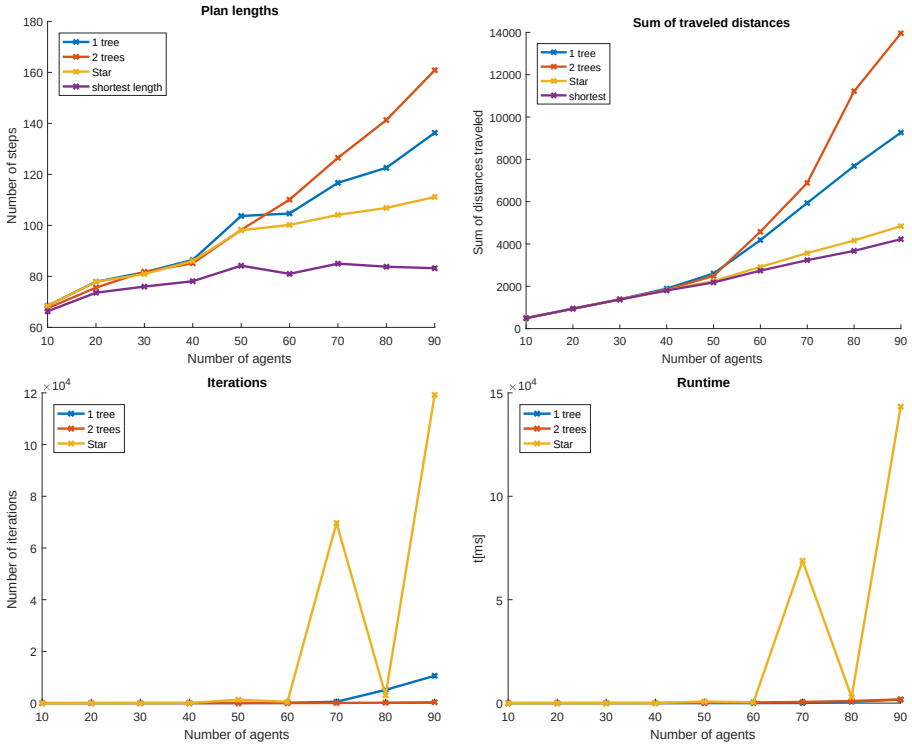


Figure 7.11: Scaling of all three versions of *MRdRRT* algorithm

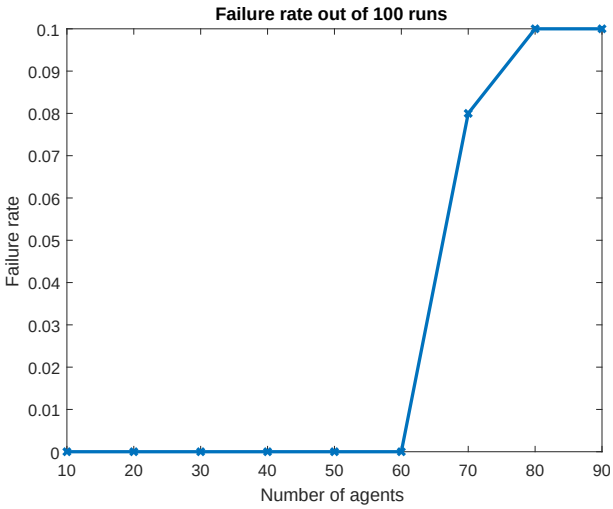


Figure 7.12: MRdRRT *star* variant failure rate

7.6 Comparison

This section shows the comparison of the implemented algorithms on three selected maps - large and medium sized maps of the robotic warehouse and open arena map from the open data set [10]. These are further denoted as *arena*,*medium* and *large*. Only the *mors*, *peasgood* and all three variants of *MRdRRT* algorithm are tested in this part because *icts* is extremely inefficient for pathfinding in these problems.

The measured performance indicators are the same as in the previous section. The quality of the plans is also compared in terms of *PDB* and *PDM* because they offer fairer comparison and insight into the quality of found solutions. *PDB* is percent deviation from the best known solution, in case of this thesis the best solution is considered to be the solution that would occur if all agents moved from their start positions to their goal positions without considering any other agents (further refered to as *shortest*). *PDB* is thus defined as a difference between the best solution found by the algorithm in consecutive runs on the same assignment (denoted as *best*) and the *shortest* divided by the *shortest* i.e

$$PDB = \frac{best - shortest}{shortest}$$

PDM is a simular measure but instead of the best found solution considers the mean value of the solution(denoted *mean*). It can be then defined as

$$PDM = \frac{mean - shortest}{shortest}$$

In case of *mors* and *peasgood* algorithms the *PDB* and *PDM* are the same for every assignment because they are executes only once. On the other hand, *MRdRRT* is a probabilistic algorithm and as such is executed multiple times on each assignment and the *PDB* and *PDM* are calculated from the results.

Every algorithm is tested on the same set of assignments. These are 8 different number of agents in the system from 10 to 80 in the increments of 10. For every number of agents 5 assignments are generated. Because *MRdRRT* is a probabilistic algorithm each assignment is ran 5 times. The limit for iterations for all versions of *MRdRRT* algorithm is set to 500 000. If the algorithm can not find solution by then, it is considered as a failure.

The results on the *arena* map can be found in Figure 7.13 and the results on *medium* and *large* map are to be seen in Figures 7.14 and 7.15 respectively. More details about the gathered results can be seen in Tables 7.1, 7.2 and 7.3 respectively. The results are average values from the successful runs of the algorithms.

The gathered results show that when *mors* algorithm finds a solution it is extremely close to optimum. In these experiments the difference between the shortest solution and the found by *mors* was zero. The problem with *mors* on the other hand is its low success rate for higher numbers of agents. The *peasgood* algorithm performed as expected with lower quality plans in

terms of performed steps but 100% success rate and decent quality in terms of distances traveled by each agent. Its main advantage is also the time required to obtain the plan which showed to be low in comparisson to other algorithms.

The variants of *MRdRRT* performed as expected with the *star* variant offering the best quality of plans which were close even to solutions found by *mors*. On the contrary its performance in terms of time and iterations required to obtain the solution it performed the worst out of the three variants even failing to find a plan for all assignments with 80 agents on *medium* map, because the number of iterations always exceeded 500000 which was set as a limit. Because the algorithm is probabilistic the algorithm would eventually find solution if no limit of iterations was set. The *2-tree* variant performed the worst in terms of plan quality but offered the best results in terms of success rate to find the solution. The results for higher numbers of agents are skewed by the fact that the results are averages and *2-tree* variant has 100% success rate on all 3 maps, therefore the average is calculated over more results than for other two variants. The *1-tree* variant offers better quality solutions than *2-tree* version but for the tradeoff of lower success rate and higher number of iterations.

Because the *mors* algorithm scales really well with the number of agents in the system in terms of distance traveled the real-world application can be any highly dynamic environment that can ensure the spacing between starting points, a low number of choke points or end points not in choke points, or for example that the agents disappear once they reach their goal node. A choke point can be seen as point through which most agents have to pass in order to get to their goals. If such choke point is the goal destination of one of the agents A then all agents that arrive to this point after A can not pass through and as such no path exists for them.

Robotic warehouse or airports can be ideal examples of such environments because these usually have fixed entry points and goal nodes that are usually node choke points of the graph and in case of airports once agent reaches the goal point, which can be for example runway it disappears from the graph because it takes off.

The *icts* algorithm showed extreme problems in terms of scaling with increasing number of agents and map sizes. As such it is unusable for large scale problems such as planning robots in roadmaps. But for smaller problems with limited number of agents and small number of potential states for each agent where the the main focus if optimality of the solution the *icts* algorithm can provide great results.

While the *peasgood* algorithm provides plans that with lower quality than other solvers, its main strength is that when the number of planned agents is lower than the number of leafs in the spanning tree it plans on, the algorithm has 100% success rate and scales incredibly well in terms of time required to obtain the plan. This algorithm is applicable in any field where the main focus is maximal success rate and low time to obtain the plan rather than overall plan quality. It is also worth to mention that this algorithm performs

extremely well in maps that are not as open as the maps that were tested i.e. maps with tunnels.

The *MRdRRT* algorithm offers middle ground between the previous algorithms with better solutions then *peasgood* but worse than *mors* with higher success rate depending on the variant. Therefore the algorithm could be used in any field the previous algorithms would be used with the exception of environment that requires maximal success rate which may cause a problem because of *MRdRRT* random nature.

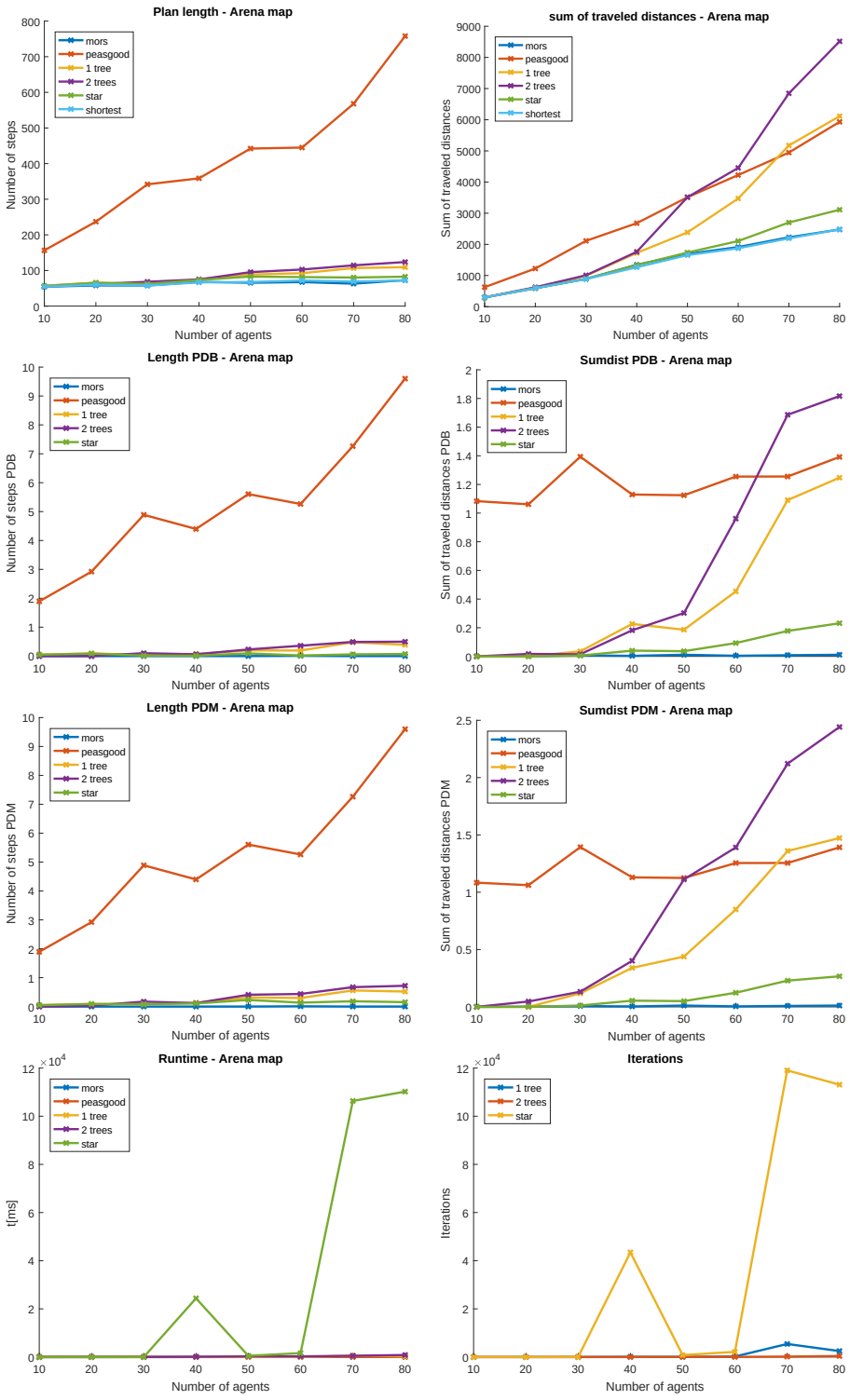


Figure 7.13: Comparison of the algorithms on the arena map

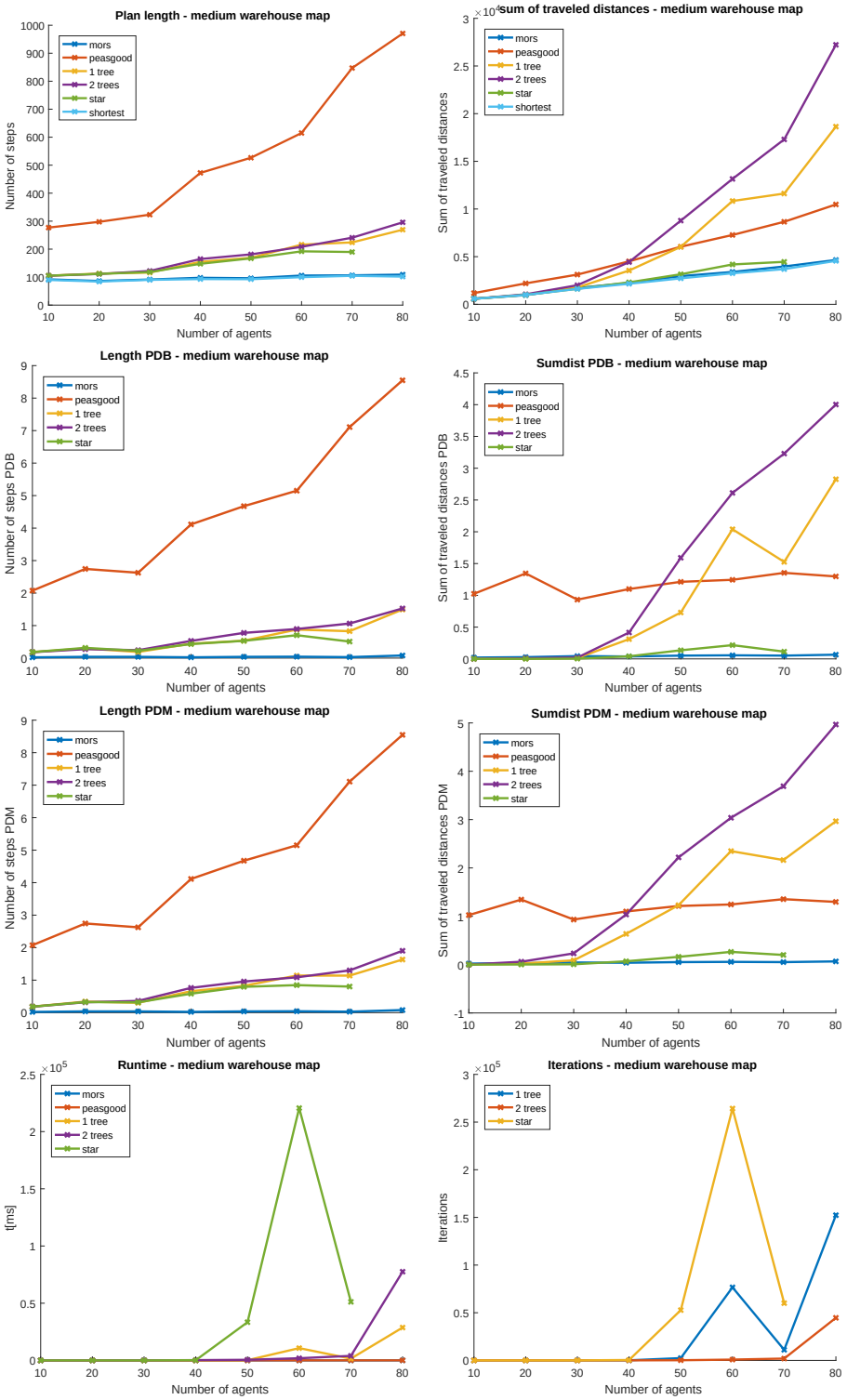


Figure 7.14: Comparison of the algorithms on the medium map

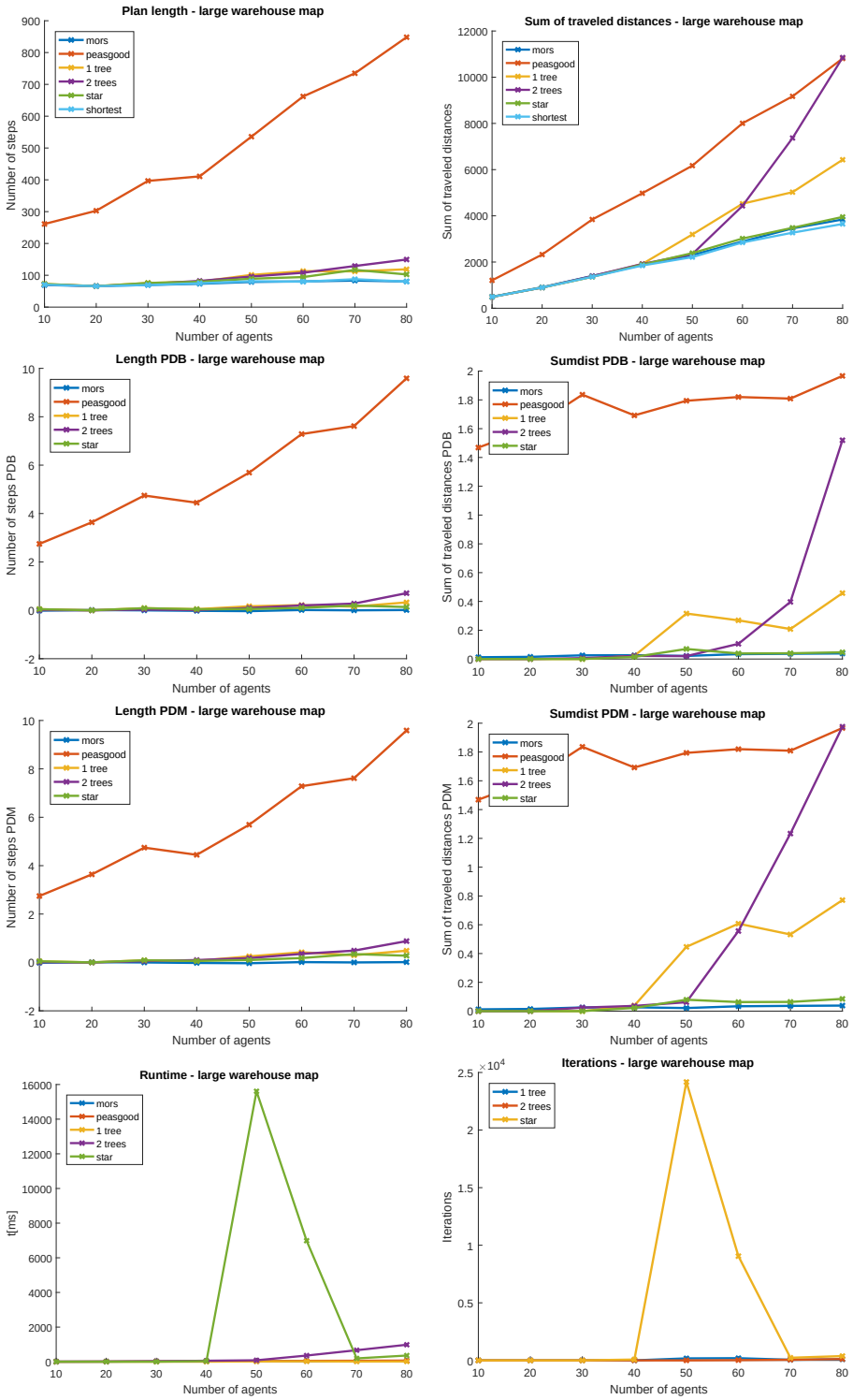


Figure 7.15: Comparison of the algorithms on large map

n-agents	Criterion	Mors	Peasgood	1 tree	2 trees	Star
10	length PDB	0.00	1.89	0.05	0.00	0.05
	length PDM	0.00	1.89	0.05	0.00	0.05
	sumdist PDB	0.00	1.08	0.00	0.00	0.00
	sumdist PDM	0.00	1.89	0.00	0.00	0.00
	runtime[ms]	4.00	3.60	0.04	2.84	11.60
	success rate	1.00	1.00	1.00	1.00	1.00
20	length PDB	0.00	2.92	0.09	0.03	0.09
	length PDM	0.00	2.92	0.09	0.03	0.09
	sumdist PDB	0.00	1.06	0.00	0.02	0.00
	sumdist PDM	0.00	2.92	0.00	0.03	0.00
	runtime[ms]	10.00	7.80	0.20	19.12	6.48
	success rate	0.80	1.00	1.00	1.00	1.00
30	length PDB	0.00	4.89	0.04	0.17	0.02
	length PDM	0.00	4.89	0.08	0.17	0.08
	sumdist PDB	0.01	1.39	0.04	0.02	0.01
	sumdist PDM	0.01	4.89	0.12	0.17	0.01
	runtime[ms]	15.00	13.60	1.48	34.20	17.40
	success rate	1.00	1.00	1.00	1.00	1.00
40	length PDB	0.00	4.40	0.07	0.12	0.01
	length PDM	0.00	4.40	0.13	0.12	0.11
	sumdist PDB	0.00	1.13	0.23	0.18	0.04
	sumdist PDM	0.00	4.40	0.34	0.12	0.05
	runtime[ms]	28.00	17.20	6.72	96.76	24386.44
	success rate	0.40	1.00	1.00	1.00	1.00
50	length PDB	0.00	5.61	0.20	0.41	0.09
	length PDM	0.00	5.61	0.31	0.41	0.23
	sumdist PDB	0.01	1.12	0.19	0.30	0.04
	sumdist PDM	0.01	5.61	0.44	0.41	0.05
	runtime[ms]	47.00	23.00	6.24	265.08	509.12
	success rate	0.40	1.00	1.00	1.00	1.00
60	length PDB	0.01	5.26	0.19	0.44	0.02
	length PDM	0.01	5.26	0.30	0.44	0.14
	sumdist PDB	0.00	1.25	0.45	0.96	0.09
	sumdist PDM	0.00	5.26	0.85	0.44	0.12
	runtime[ms]	41.50	27.80	19.56	265.60	1618.00
	success rate	0.40	1.00	1.00	1.00	1.00
70	length PDB	0.00	7.26	0.48	0.67	0.05
	length PDM	0.00	7.26	0.56	0.67	0.15
	sumdist PDB	0.01	1.26	1.09	1.69	0.14
	sumdist PDM	0.01	7.26	1.36	0.67	0.18
	runtime[ms]	59.00	33.40	588.84	522.44	85075.75
	success rate	0.20	1.00	1.00	1.00	0.76
80	length PDB	0.00	9.60	0.39	0.72	0.06
	length PDM	0.00	9.60	0.52	0.72	0.12
	sumdist PDB	0.01	1.39	1.25	1.82	0.18
	sumdist PDM	0.01	9.60	1.47	0.72	0.21
	runtime[ms]	64.00	42.40	287.56	838.12	88170.44
	success rate	0.40	1.00	1.00	1.00	0.76

Table 7.1: Comparison of algorithms on *arena* map

n-agents	Criterion	Mors	Peasgood	1 tree	2 trees	Star
10	length PDB	0.02	2.07	0.18	0.18	0.18
	length PDM	0.02	2.07	0.18	0.18	0.18
	sumdist PDB	0.02	1.02	0.00	0.00	0.00
	sumdist PDM	0.02	2.07	0.00	0.18	0.00
	runtime[ms]	3.20	4.80	0.28	2.96	1.60
	success rate	1.00	1.00	1.00	1.00	1.00
20	length PDB	0.04	2.74	0.30	0.32	0.31
	length PDM	0.04	2.74	0.34	0.32	0.33
	sumdist PDB	0.03	1.35	0.01	0.01	0.00
	sumdist PDM	0.03	2.74	0.02	0.32	0.00
	runtime[ms]	4.25	8.40	1.00	13.60	2.24
	success rate	0.80	1.00	1.00	1.00	1.00
30	length PDB	0.04	2.63	0.19	0.36	0.22
	length PDM	0.04	2.63	0.30	0.36	0.32
	sumdist PDB	0.04	0.93	0.01	0.02	0.00
	sumdist PDM	0.04	2.63	0.09	0.36	0.01
	runtime[ms]	11.75	12.40	2.12	56.40	6.96
	success rate	0.80	1.00	1.00	1.00	1.00
40	length PDB	0.02	4.11	0.44	0.76	0.43
	length PDM	0.02	4.11	0.66	0.76	0.58
	sumdist PDB	0.04	1.10	0.31	0.42	0.04
	sumdist PDM	0.04	4.11	0.64	0.76	0.07
	runtime[ms]	12.67	18.80	8.24	241.88	207.24
	success rate	0.60	1.00	1.00	1.00	1.00
50	length PDB	0.04	4.67	0.53	0.96	0.53
	length PDM	0.04	4.67	0.82	0.96	0.80
	sumdist PDB	0.05	1.21	0.73	1.59	0.14
	sumdist PDM	0.05	4.67	1.23	0.96	0.16
	runtime[ms]	24.00	23.80	196.80	536.24	33387.44
	success rate	0.40	1.00	1.00	1.00	0.92
60	length PDB	0.04	5.15	0.88	1.36	0.53
	length PDM	0.04	5.15	1.14	1.36	0.64
	sumdist PDB	0.06	1.24	2.04	3.26	0.16
	sumdist PDM	0.06	5.15	2.35	1.36	0.20
	runtime[ms]	27.33	31.40	10737.50	2380.45	165418.30
	success rate	0.60	1.00	0.68	1.00	0.44
70	length PDB	0.03	7.11	0.83	1.30	0.30
	length PDM	0.03	7.11	1.14	1.30	0.48
	sumdist PDB	0.05	1.35	1.53	3.23	0.07
	sumdist PDM	0.05	7.11	2.16	1.30	0.12
	runtime[ms]	24.50	37.40	1578.12	3963.92	30709.04
	success rate	0.40	1.00	1.00	1.00	0.60
80	length PDB	0.08	8.55	1.50	3.17	*N/A*
	length PDM	0.08	8.55	1.63	3.17	*N/A*
	sumdist PDB	0.07	1.30	2.83	6.67	*N/A*
	sumdist PDM	0.07	8.55	2.97	3.17	*N/A*
	runtime[ms]	42.50	50.40	28722.20	129270.00	*N/A*
	success rate	0.40	1.00	0.32	1.00	0.00

Table 7.2: Comparison of algorithms on *medium* map

n-agents	Criterion	Mors	Peasgood	1 tree	2 trees	Star
10	length PDB	-0.01	2.74	0.05	0.01	0.05
	length PDM	-0.01	2.74	0.05	0.01	0.05
	sumdist PDB	0.01	1.47	0.00	0.00	0.00
	sumdist PDM	0.01	2.74	0.00	0.01	0.00
	runtime[ms]	6.20	6.80	0.44	5.76	2.96
	success rate	1.00	1.00	1.00	1.00	1.00
20	length PDB	0.01	3.64	0.00	0.00	0.00
	length PDM	0.01	3.64	0.00	0.00	0.00
	sumdist PDB	0.02	1.60	0.00	0.00	0.00
	sumdist PDM	0.02	3.64	0.00	0.00	0.00
	runtime[ms]	10.00	13.60	0.76	8.56	2.88
	success rate	1.00	1.00	1.00	1.00	1.00
30	length PDB	0.00	4.74	0.09	0.05	0.09
	length PDM	0.00	4.74	0.09	0.05	0.09
	sumdist PDB	0.03	1.84	0.00	0.01	0.00
	sumdist PDM	0.03	4.74	0.00	0.05	0.00
	runtime[ms]	17.20	23.00	1.16	27.28	3.24
	success rate	1.00	1.00	1.00	1.00	1.00
40	length PDB	-0.02	4.45	0.05	0.10	0.05
	length PDM	-0.02	4.45	0.07	0.10	0.06
	sumdist PDB	0.03	1.69	0.02	0.02	0.01
	sumdist PDM	0.03	4.45	0.04	0.10	0.02
	runtime[ms]	22.50	29.00	2.72	59.60	35.40
	success rate	0.80	1.00	1.00	1.00	1.00
50	length PDB	-0.03	5.69	0.17	0.18	0.05
	length PDM	-0.03	5.69	0.25	0.18	0.10
	sumdist PDB	0.02	1.79	0.32	0.02	0.07
	sumdist PDM	0.02	5.69	0.45	0.18	0.08
	runtime[ms]	31.33	36.80	17.28	85.20	15609.88
	success rate	0.60	1.00	1.00	1.00	1.00
60	length PDB	0.01	7.28	0.21	0.35	0.10
	length PDM	0.01	7.28	0.42	0.35	0.18
	sumdist PDB	0.03	1.82	0.27	0.11	0.04
	sumdist PDM	0.03	7.28	0.61	0.35	0.06
	runtime[ms]	43.00	52.80	22.68	358.12	6986.80
	success rate	0.60	1.00	1.00	1.00	1.00
70	length PDB	0.00	7.62	0.15	0.49	0.20
	length PDM	0.00	7.62	0.30	0.49	0.34
	sumdist PDB	0.04	1.81	0.21	0.40	0.04
	sumdist PDM	0.04	7.62	0.53	0.49	0.06
	runtime[ms]	42.67	58.60	10.44	667.04	193.24
	success rate	0.60	1.00	1.00	1.00	1.00
80	length PDB	0.01	9.59	0.33	0.88	0.14
	length PDM	0.01	9.59	0.48	0.88	0.28
	sumdist PDB	0.04	1.97	0.46	1.52	0.05
	sumdist PDM	0.04	9.59	0.77	0.88	0.08
	runtime[ms]	44.00	69.80	16.72	981.60	359.20
	success rate	0.40	1.00	1.00	1.00	1.00

Table 7.3: Comparison of algorithms on *large* map

Chapter 8

Conclusion

This thesis discussed the task of multi-agent pathfinding. Four different algorithms were chosen from the different categories of multi-agent pathfinding algorithms. These were then studied and implemented in C++. Each algorithm was discussed and described in a separate chapter. Several improvements of the *peasgood* and the *MRdRRT* algorithms based on the experience of their function were proposed and tested in experiments in Chapter 7. Among the improvements to the *peasgood* algorithm are different techniques of obtaining leaf node in the first phase of the *peasgood* algorithm, implementing an algorithm to remove loops in the resulting plan and proposing different techniques for concurrent plan building.

The *MRdRRT* algorithm was enhanced by different sampling methods for the random sample in addition with accompanying versions of oracle implementation that reduce the number of thrown away samples by trying to generate a valid new configuration of agents from a given sample. Another addition for the *MRdRRT* algorithm is the *2-tree* version that modified the original algorithm to use 2 trees, one from the start and one from the target configuration that are actively trying to connect to each other with the purpose of reducing the required number of iterations to find solution. Lastly from the experience with the two previous versions a new approach inspired by the known *RRT** algorithm is proposed in this thesis whose goal is to improve the quality of the solution obtained by the *MRdRRT* algorithm.

All algorithms and their improvements were thoroughly tested in the Chapter 7 of this thesis along with comparing their desired properties with the experimental results.

The results showed that the *2-tree* variant increased the success rate of the algorithm in assignments where *1-tree* variant had low success rate while the *star* variant found better quality solutions than the previous two variants in almost all cases but for the cost of longer runtime which in some cases exceeded the given limit of iterations.

The *star* version of the *MRdRRT* algorithm showed promise in bringing the obtained solution closer to optimum but for the cost of increased execution time. For this reason the future work should focus on improvement of this proposed algorithm in terms of reducing the number of required iterations to find the first solution. This could be done for example by reducing the

dimensionality of the problem by planning smaller groups of agents in batches and then considering them as obstacles moving in time for following groups. Another major improvement would be finding more powerful local connector and devising more intelligent expansion phase.

Bibliography

[1] A. B. Kahn. Topological sorting of large networks. *Commun. ACM*, 5(11):558–562, November 1962.

[2] Sertac Karaman and Emilio Frazzoli. Incremental sampling-based algorithms for optimal motion planning. *CoRR*, abs/1005.0416, 2010.

[3] Dan Halperin Kiril Solovey, Oren Salzman. Finding a needle in an exponential haystack: Discrete rrt for exploration of implicit roadmaps in multi-robot motion planning. *Algorithmic Foundations of Robotics XI*, pages 591–607, 2014.

[4] Steven M. LaValle. Rapidly-exploring random trees: A new tool for path planning. Technical report, October 1998.

[5] Martin Makovička. Koordinace v systémech s více roboty. Master's thesis, Czech Technical University in Prague, Czech Republic, 2012.

[6] John McPhee Mike Peasgood, Christopher Michael Clark. A complete and scalable strategy for coordinating multiple robots within roadmaps. *IEEE Transactions on Robotics*, pages 283–292, 2008.

[7] Marius Muja and David G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *International Conference on Computer Vision Theory and Application VISSAPP'09)*, pages 331–340. INSTICC Press, 2009.

[8] Bertram Raphael Peter E. Hart, Nils J. Nilsson. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics SCC-4*, pages 100–1007, 1968.

[9] Guni Sharon, Roni Stern, Meir Goldenberg, and Ariel Felner. The increasing cost tree search for optimal multi-agent pathfinding. *Artif. Intell.*, 195:470–495, February 2013.

[10] N. Sturtevant. Benchmarks for grid-based pathfinding. *Transactions on Computational Intelligence and AI in Games*, 4(2):144 – 148, 2012.

[11] Adriaan ter Mors, Cees Witteveen, Jonne Zutt, and Fernando A. Kuipers. Context-aware route planning. In Juergen Dix and Cees Witteveen, editors, *Multiagent System Technologies, 8th German Conference, MATES 2010, Leipzig, Germany*, volume 6251 of *Lecture Notes in Computer Science*, pages 138–149. Springer, 2010.

[12] Adriaan W. ter Mors. Conflict-free route planning in dynamic environments. In Nancy M. Amato, editor, *Proceedings of the 2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2166–2171. IEEE, September 2011.

[13] J. van den Berg, J. Snoeyink, M. Lin, and D. Manocha. Centralized path planning for multiple robots: Optimal decoupling into sequential plans. In *Proceedings of Robotics: Science and Systems*, Seattle, USA, June 2009.

Chapter 9

Enclosed CD contents

The root directory on the enclosed CD contains the following items:

- **thesis.pdf:** The PDF file of this thesis.
- **[text_source]:** Directory containing latex source files of the document.
- **[source]:** Directory containing source codes written in C++. The project can be built with Visual Studio.
- **[maps]:** Directory containing files of used maps.