FACULTY OF INFORMATION TECHNOLOGY CTU IN PRAGUE

ASSIGNMENT OF BACHELOR’S THESIS

Title:	Adapting the Conflict-based Search Algorithm for Alternative Objectives
Student:	Berker Katipoglu
Supervisor:	doc. RNDr. Pavel Surynek, Ph.D.
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of summer semester 2020/21

Instructions

The task is to adapt the Conflict-based Search (CBS) algorithm for optimal multi-agent path finding (MAPF) for different objectives other than the sum-of-cost. While the sum-of-costs objective focuses on minimizing the overall cost of actions performed by agents it is sometimes more reasonable to use different objectives such as the makespan (the time when the last agent arrives to its goal). One possible direction is to adapt and evaluate the CBS algorithm for the makespan objective. The specific tasks for the student are as follows:

1. Study relevant materials concerning the CBS algorithm.
2. Suggest adaptations of CBS for alternative objective and analyze theoretically soundness of the adapted algorithm.
3. Evaluate the new algorithm experimentally on a relevant set of benchmarks.
4. Compare the original version of CBS using the sum-of-cost objective and the modified version and discuss impact of different objectives on the performance of the algorithm.

References

[1] Eli Boyarski, Ariel Felner, Roni Stern, Guni Sharon, David Tolpin, Oded Betzalel, Solomon Eyal Shimony: ICBS: Improved Conflict-Based Search Algorithm for Multi-Agent Pathfinding. IJCAI 2015: 740-746

[2] Guni Sharon, Roni Stern, Ariel Felner, Nathan R. Sturtevant: Conflict-based search for optimal multi-agent pathfinding. Artif. Intell. 219: 40-66 (2015)

[3] Ariel Felner, Jiaoyang Li, Eli Boyarski, Hang Ma, Liron Cohen, T. K. Satish Kumar, Sven Koenig: Adding Heuristics to Conflict-Based Search for Multi-Agent Path Finding. ICAPS 2018: 83-87

Ing. Michal Valenta, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague January 8, 2020

Czech Technical University in Prague
Faculty of Information Technology
Department of Software Engineering

Bachelor's thesis

Adapting the Conflict-Based Search Algorithm for Alternative Objectives

Berker Katipoglu

Supervisor: doc. RNDr. Pavel Surynek, Ph.D.

9th January 2020

Acknowledgements

I would like to thank my supervisor for providing feedback on my thesis and also I would like to thank my parents for their support and encouragement.

Declaration

In Prague on 9th January 2020 . . . .

Citation of this thesis

Katipoglu, Berker. *Adapting the Conflict-Based Search Algorithm for Alternative Objectives.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2020.

Abstrakt

Hledání cesty pro více agentů (MAPF) je důležitým typem problému plánování v umělé inteligenci. Existuje mnoho aplikací MAPF a každá aplikace má své vlastní priority, což dává MAPF mnoho různých variací. S rostoucím zájmem vědců o toto téma byly vyvinuty nové algoritmy pro řešení různých variací MAPF v posledních letech. Konfliktní vyhledávání (CBS) je jedním z těchto algoritmů, který je v současné době nejmodernějším řešením řešení instancí MAPF pomocí objektivní funkce součtu nákladů. V tomto článku budeme diskutovat o tom, jak přizpůsobit CBS pro objektivní funkci značky *makespan*, emprické srovnání obtížnosti řešení instancí MAPF s CBS v rámci součtu nákladů a objektivních funkcí značky a způsoby, jak zlepšit chování CBS pomocí objektivní funkce značkypanpan. Experimentální výsledky ukazují, že řešení usnadňuje řešení než řešení součtu nákladů, a je možné upravit algoritmus pro objektivní funkci tak, aby se dosáhlo lepšího výkonu.

Klíčová slova Umělá inteligence, hledání cest s více agenty, vyhledávání založené na konfliktech, objektivní funkce, makespan

Abstract

The Multi-Agent Path Finding (MAPF) is an important type of planning problem in artificial intelligence. There are many applications of MAPF and each application has its own priorities, which gives MAPF many different variations. With the increasing interest of researchers on this topic, new algorithms developed for solving different variations of MAPF in the recent years. Conflict Based Search (CBS) is one of those algorithms, which is currently the state-of-art for solving MAPF instances with *sum-of-cost* objective function. In this paper, we will discuss how to adapt CBS for *makespan* objective function, emprical comparison of the difficulty of solving MAPF instances with CBS under sum-of-costs and makespan objective functions, and ways to improve the behaviour of CBS with makespan objective function. Experimental results show that solving makespan is easier than solving sum-of-costs, and it is possible to adjust the algorithm for the objective function to obtain better performance.

Keywords Artificial Intelligence, Multi-Agent Path Finding, Conflict Based Search, Objective Function, Makespan

Contents

List of Figures

List of Tables

Introduction

Planning is an important topic in artificial intelligence, which focuses on finding sequence of actions for an agent to make it reach its goal state. MAPF is a cooperative planning problem for multiple agents, and it aims to find a path for each agent on a given graph such that agents will be able to follow their path without colliding with each other. [1]. There are many applications of MAPF such as robotics [2], video games, vehicle routing [3], aviation [4]. With the emerging new technologies interest in MAPF has been increasing among research groups.

There are two main approaches of solving MAPF problems, *coupled* and *decoupled*. Solvers using decoupled approach plan each agent separately. This makes decoupled solvers relatively fast, however optimality and completeness of the solution is not guaranteed. On the other hand it is possible to obtain an optimal solution with a coupled solver, but with usually significant computational expenses. CBS algorithm combines coupled and decoupled approaches and guarantees to provide optimal solution.

MAPF has many variations depending on the set of assumptions of the agent behavior, and problem objectives. Two common variations for objectives are minimizing *sum-of-costs* and minimizing *makespan*. Sum-of-costs focuses on total cost of agent paths in a solution, whereas makespan focuses on the maximum cost of agent paths. CBS algorithm works with sum-of-costs objective fucntion. In this paper we will investigate how to adapt CBS algorithm for makespan objective function and suggest improvements to obtain solutions faster. Also background knowledge will be provided regarding to pathfining problem, MAPF and CBS algorithm.

Chapter 1

Path Finding Problem

Graphs in computer science are abstraction of real-world problems. They are used to simplify the problem so that it can be represented and manipulated by a computer. Idea of graphs were introduced in 18th century by Swiss mathematician Leonhard Euler. He used graphs to solve the seven briges of Königsberg problem. Königsberg was situated by the river Pregel and it was connected by seven bridges. The problem was whether it is possible to take a walk through the town in such a way that each bridge would be crossed exactly once. Graph theory was born with this problem and it is now one of the major areas of mathematical research.

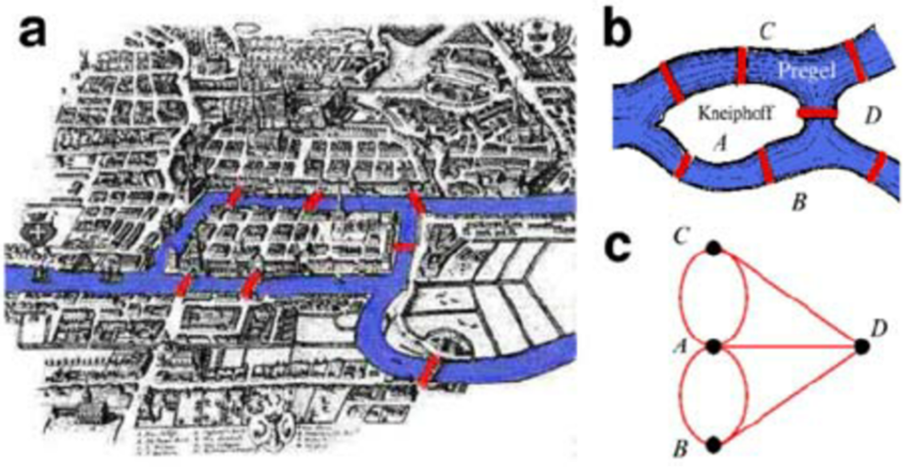


Figure 1.1: a) The town of Königsberg. (b) Schematic representation of the area. (c) Euler's graph representation of the problem.

Formally, we define a graph G as an ordered pair $G = (V, E)$ where, V is a set of nodes (vertices) and E is a set of edges (links) between the vertices such that $E \subseteq \{(u, v)|u, v \in V\}$. Graphs are used in many real world problems such as social networking, transportation, biology, simulations, robotics, artificial intelligence etc. In some of those problems it is important to find a *path* between two given vertices. For example if the problem is about moving a robot from point A to point B, then in the graph representation of this problem, first we need to find a path from A to B in order to program our robot. In graph theory a *path* p is defined as a sequence of distinct vertices connected by edges. Formally for a graph $G = (V, E)$, $p = (V_1, ..., V_n)$ is a path such that $V_i \neq V_j, \forall i \neq j \wedge (V_i, V_{i+1}) \in E$.

1.0.1 Path finding algorithms

Depending on the graph there may be more than one path between two vertices, and some of those may be better than others. Again referring to our robotics example, if we want to move our robot from vertex A to vertex B and if it is important to find the shortest path in that particular problem, then the results are comparable and some of them may be better than others. This problem is usually referred as the shortest path problem and it is a well studied optimization problem in computer science. There are various algorithms to solve the shortest path problem like Greedy Best-First Search (GBFS), Dijkstra, Floyd-Warshall, A*. Among these algorithms, A* is a widely used due to its flexibility and adaptability for wide range of context.

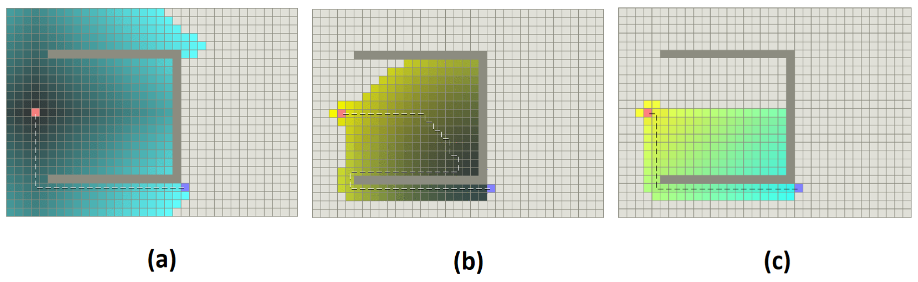


Figure 1.2: Visualization of shortest path finding algorithms. (a) Dijkstra. (b) Greedy Best-First Search. (c) A*.

A* is a combination of Dijkstra's and GBFS algorithms. Dijkstra's algorithm favors nodes that are close to the source node, whereas GBFS algorithm favors nodes that are close to the goal node. In the standard terminology, $g(n)$ is the exact cost of the path from the starting node to any node n

(which is used by Dijkstra), and $h(n)$ represents the *heuristic* estimated cost from node n to the goal node (which is used by GBFS). A* balances the two as it moves from the starting point to the goal and it favors nodes with the lowest $f(n) = g(n) + h(n)$ value. [5]

In A*, we need to keep track of several information about the nodes. First one is the nodes to be examined. For this purpose a set called $OPEN$ can be used. $OPEN$ will contain all the nodes to be examined. At the beginning of the algorithm $OPEN$ only contains the source node. Second information to keep track of is the nodes that are already examined. This can be realized by another set called $CLOSED$. At the beginning, $CLOSED$ is an empty set. We also need to keep track of g value of each node. We can use a map to keep track of this information. Finally, we need to keep track of the parent node for each node. Again using a map will be convenient for this purpose.

Algorithm will run in a loop, while $OPEN$ set is not empty. In each iteration a node N with lowest $f(N)$ value will be removed from $OPEN$ for examination. If N is the target node, then algorithm calculates the path using the parent mapping and returns it. Otherwise, N is added to $CLOSED$ and each neighbor of N will be examined. For a neighbor N' of N, first we check if it is already in $CLOSED$. If so, we ignore this neighbor. Else we calculate $g(N')$ and save it to a variable called *cost*. Then we check if N' is already in $OPEN$. If it is in $OPEN$ and *cost* value is less than $g(N')$, then we update the $g(N')$ value with the new *cost* value. If it is not in $OPEN$, we save the cost value in $g(N')$, we map N as parent of N' and add N' to $OPEN$. If loop ends, then it means there is no path between source and target.

Algorithm 1 A*

```
    Input: $G(V, E)$, $Source$, $Target$
 1: $OPEN \leftarrow \{Source\}$
 2: $CLOSED \leftarrow \{\}$
 3: $G\_VALUES \leftarrow$ empty map
 4: $PARENTS \leftarrow$ empty map
 5: while $OPEN$ not empty do
 6:     $N \leftarrow$ best node from $OPEN$ // lowest $f(N)$ value
 7:     if $N = Target$ then
 8:         return $find\_path(N)$
 9:     end if
10:     Insert $N$ to $CLOSED$
11:     for each neighbor $N'$ of $N$ do
12:         if $N'$ in $CLOSED$ then
13:             continue
14:         end if
15:         $COST \leftarrow G\_VALUES[N] + cost(N, N')$
16:         if $N'$ in $OPEN$ and $COST < G\_VALUES[N']$ then
17:             $G\_VALUES[N'] \leftarrow COST$
18:         else
19:             $G\_VALUES[N'] \leftarrow COST$
20:             Insert $N'$ to $OPEN$
21:         end if
22:     end for
23: end while
```

Chapter 2

Multi Agent Path Finding

MAPF is an extension of path finding problem with k agents instead of 1. Each agent will have a start and a goal location and the goal of the problem is again finding a path for each agent from its start to goal location. The important point is that these paths should be collision free. Usually there also exists an *objective function*, which needs to be optimized. Optimal MAPF is NP-Complete because it is a generalization of sliding puzzle problem. [6]

2.1 MAPF definition

In classical Multi-Agent Path Finding (MAPF), problem with k agents is defined by a tuple (G, s, t), where:

- $G = (V, E)$ is a an undirected graph,
- $s : [1, ..., k] \rightarrow V$ is a mapping for each agent to its source vertex,
- $t : [1, ..., k] \rightarrow V$ is a mapping for each agent to its target vertex.

Time is considered to be discrete and edge weights are considered to be 1. In every time step each agent will be located in one of the vertices and they can perform a single action. An action is a function $a : V \rightarrow V$ such that $a(v) = v'$. Which means, if an agent is at vertex v and performs action a then it will be in vertex v' in the next time step. Agents can either *wait* in their current vertex v or can *move* to an adjacent vertex v' in the graph (i.e. $(v, v') \in E$).

A plan for an agent is a sequence of actions such that execution of these actions on source vertex of that agent will end up at its target vertex. More formally, $\pi_i = (a_1, ..., a_n)$ is a **single-agent plan**, iff $t(i) = a_n(a_{n-1}(...(a_1(s(i)))))$. A **solution** to MAPF problem is a set of **collision-free**, k single-agent plans, one for each agent. [1]

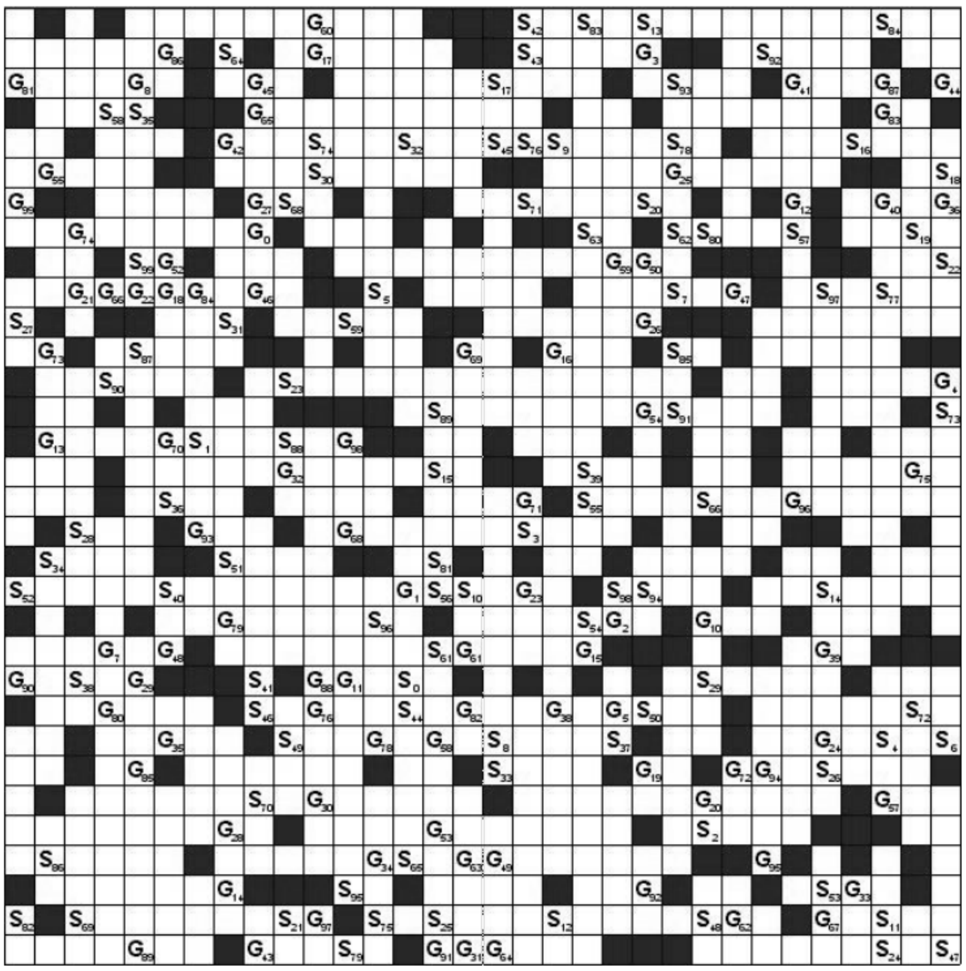


Figure 2.1: Visualization of MAPF instances with many agents.

[7]

2.2 Type of conflicts

For a MAPF solution to be valid, there should be no conflicts between any two single-agent plans within that solution. Conflict is a restriction based on the environment and problem description. Common types of conflicts are listed below.

2.2.1 Vertex conflict

Vertex conflict occurs between two single-agent plans π_i and π_j iff according to these plans agents i and j are occupying the same vertex at the same time step. Formally, there exist a vertex conflict between π_i and π_j iff, there exists a time step t such that $\pi_i[t] = \pi_j[t]$.

2.2.2 Edge conflict

Edge conflict occurs between two single-agent plans π_i and π_j iff according to these plans agents i and j are traversing the same edge at the same time step and in the same direction. Formally, there exist an edge conflict between π_i and π_j iff, there exists a time step t such that $\pi_i[t] = \pi_j[t]$ and $\pi_i[t+1] = \pi_j[t+1]$.

2.2.3 Swapping conflict

Swapping conflict occurs between two single-agent plans π_i and π_j iff according to these plans agents i and j are traversing the same edge at the same time step and in the opposite direction. Formally, there exist a swapping conflict between π_i and π_j iff, there exists a time step t such that $\pi_i[t] = \pi_j[t+1]$ and $\pi_i[t+1] = \pi_j[t]$.

2.2.4 Following conflict

Following conflict occurs between two single-agent plans π_i and π_j iff according to these plans agent i is occupying a vertex which was occupied by agent j in the previous time step. Formally, there exist a following conflict between π_i and π_j iff, there exists a time step t such that $\pi_i[t+1] = \pi_j[t]$.

2.2.5 Cycle conflict

Cycle conflict occurs between a set of single-agent plans $\pi_i, \pi_{i+1}, ..., \pi_j$ iff according to these plans each agent is occupying a vertex which was occupied by another agent in the previous time step in a rotational pattern. Formally, there exist a cycle conflict between $\pi_i, \pi_{i+1}, ..., \pi_j$ iff, there exists a time step t such that $\pi_i[t+1] = \pi_{i+1}[t]$ and $\pi_{i+1}[t+1] = \pi_{i+2}[t]$ and ... and $\pi_{j-1}[t+1] = \pi_j[t]$ and $\pi_j[t+1] = \pi_i[t]$.

2.3 Agent behavior at target vertex

In a MAPF solution different agents may end up at their target vertex at different time steps. This situation may or may not cause further conflicts according to how agent behavior is defined, therefore problem definition should

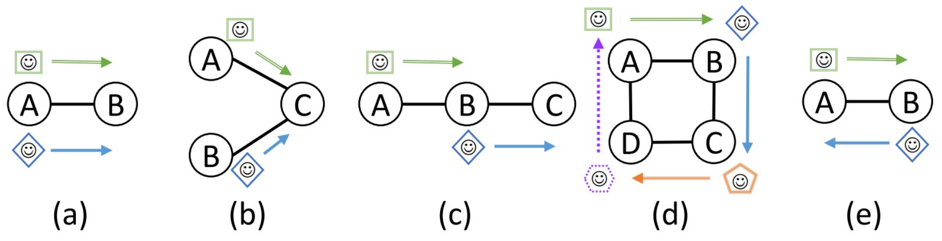


Figure 2.2: Visualization of common conflict types. (a) Edge conflict. (b) Vertex conflict. (c) following conflict. (d) Cycle conflict. (e) Swapping conflict. [1]

include the agent behavior at the target location. There are two main behavior types.

2.3.1 Stay at target

In the stay at target type, whenever an agent reaches at its target location it will keep staying there until the end of whole planning. That is to say, it will occupy that vertex and if any other agent is planned to pass that vertex, that will cause a vertex conflict. Formally, if agents are assumed to stay at target, then single-agent plans π_i and π_j will have a vertex conflict if there exists a time step $t >= |\pi_i|$ such that $\pi_j[t] = \pi_i[|\pi_i|]$.

2.3.2 Disappear at target

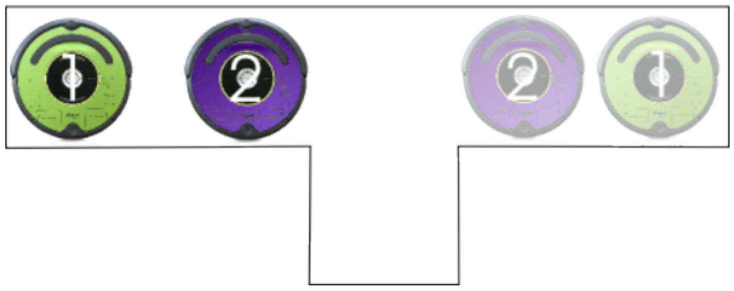
In the disappear at target type, whenever an agent reaches at its target location it will disappear from the graph and is assumed that it will not occupy any vertex. That will avoid this an agent to have any conflicts with another agent after it reaches the target vertex.

2.4 Objective functions

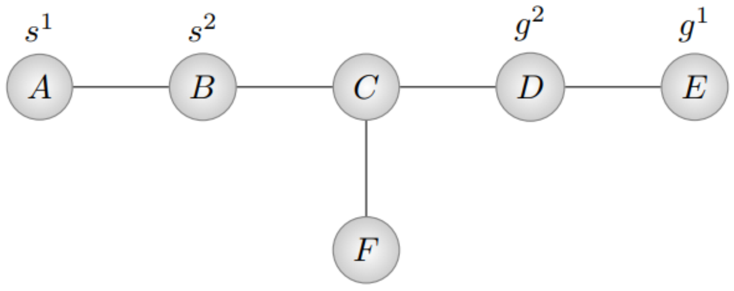
To be able to talk about an optimal solution, there should be an evaluation criteria to determine which solution is better than the other. Objective functions are used for this purpose. There are two main objective functions, *makespan* and *sum of costs*, and the goal is to minimize them. However, it is possible to introduce new objective functions according to the problem definition. For example if there is a strict time limit for the planning, maximum number of agents reaching target within a given makespan can be used as an objective function. Or one problem may be focus on the total number of non-waiting actions (which also referred as sum-of-fuel).

2.4.1 Makespan

In makespan objective function, criteria is when the whole planning finishes. This is determined by the longest *single-agent plan* in the solution. Formally, for a MAPF solution $\pi = \{\pi_1, ..., \pi_k\}$, the makespan is defined as $max_{1 \leq i \leq k} |\pi_i|$.



(a) Example environment with two agents. The transparent agent images on the right mark the goal locations of the agents with corresponding colors.



(b) Graph representation of the example environment.

Agent	$t = 1$	$t = 2$	$t = 3$	$t = 4$
1	$A \rightarrow B$	$B \rightarrow C$	$C \rightarrow D$	$D \rightarrow E$
2	$B \rightarrow C$	$C \rightarrow F$	$F \rightarrow C$	$C \rightarrow D$

(c) Optimal MAPF plan with makespan four.

Figure 2.3: Running example with makespan objective function.

[8]

2.4.2 Sum of costs

In sum of costs objective function, criteria is the sum of single-agent plan lengths. Formally, for a MAPF solution $\pi = \{\pi_1, ..., \pi_k\}$, the sum of costs is defined as $\sum_{1 \leq i \leq k} |\pi_i|$. Sum of costs objective function is known as *flowtime* as well.

While calculating sum of costs, it is important to specify the agent behavior at target location and how it effects the plan length. If the agent behavior is *stay at target*, then the common assumption is, agents plan ends if it is staying at its target location and not moving to any other adjacent vertex. For example, if an agent reaches its target vertex at time t and afterwards it needs to move again from its target vertex due to a conflict at time t' and comes back to the target vertex at t'' and stays there until the whole plan finishes, then this agent's plan length is t''.

Chapter 3

Conflict Based Search

There are two main approaches to solve a MAPF problem, *decoupled* and *coupled* approach. In *decoupled* approach, paths are planned separately. For example in HCA* algorithm, single-agent paths are saved in *global reservation table.* During planning each agent must avoid the location and time steps in the global reservation table, which were reserved by previous agents. [9] Although decoupled algorithms run relatively fast, optimality and completeness is not always guaranteed. On the other hand, coupled approaches usually return optimal solution. In this approach, MAPF problem is treated as a single-agent problem and can be solved by an A*-based algorithm that searches the state space. Main issue is the state space grows exponentially with the number of agents. [7]

Conflict based search (CBS) is an algorithm which combines both coupled and decoupled algorithms. CBS guarantees optimal solution while performing single-agent searches for path finding, like decoupled algorithms. CBS is a two-level algorithm, on the high level, a search is performed on a *constraint tree* (CT), whose nodes have the information about constraints on time and location for each agent plan. According to these constraints, in each tree node, low-level search is performed to find the single-agent plans for the agents subjected to the constraint of that node. Unlike A*-based search algorithms where state space grows exponential with the number of agents, high level of CBS is exponential with the conflicts encountered during the solving process. [10]

3.1 Definitions

In CBS, the term *path* is used only for a single agent. A *solution* is the set of k paths for a given k agents. A *constraint* is a tuple (a_i, v, t), where agent a_i is restricted to be at vertex v at time step t. During the execution of the algorithm, agents are linked with set of constraints. A *consistent path* for an agent is a path that satisfies all of the constraints for that agent. Similarly, a *consistent solution* is a set of k consistent paths for given k agents. A *conflict* is a tuple (a_i, a_j, v, t), where agent a_i and agent a_j are planned to occupy the vertex v at time step t. A solution is *valid* if there are no conflicts between any of the paths within that solution. [10]

3.2 High-level

At the high level, CBS searches for a valid solution in a *constraint tree* (CT). A CT is a binary tree, and each node N in CT contains the following information:

- A **set of constraints** ($N.constraints$). At root, constraints set is empty. Every child node will inherit the constraints set from its parent and add a new constraint for the given agent.
- A **solution** ($N.solution$). A set of k paths, one for each agent. Each path should be consistent with the constraints of the node and all of its predecessors.
- A **total cost value** ($N.cost$), is the sum of path lengths of that node's solution. It is also referred as the *f-value* of the node.

At the high-level, algorithm performs a best-first search, where nodes are ordered according to their total cost value, until it finds the goal node which is a node with valid solution. When a node N is expanded during the search, first a *validation* is performed to check if the solution ($N.solution$) is valid or not. If it is a valid solution, then N is a goal node and $N.solution$ is returned. If a conflict is encountered during the validation, then N is declared as a non-goal node.

To be able to solve a conflict $C = (a_i, a_j, v, t)$, at least one of the two constraints (a_i, v, t) and (a_j, v, t) should be added to $N.constraints$. To guarantee optimality, both options should be investigated. Therefore, node N is split into two child nodes. Each child node inherits the constraints from N, one of them adds the constraint (a_i, v, t) and the other one adds the constraint (a_j, v, t). Then low-level search is invoked for both of the child nodes and paths for agent a_i and a_j will be updated in the related child node.

3.3 Low-level

At the low-level, CBS is searching for a consistent and optimal path for a given agent and its associated constraints. This search is performed in a *decoupled manner*, i.e., ignoring other agents. The search is three dimensional, two for

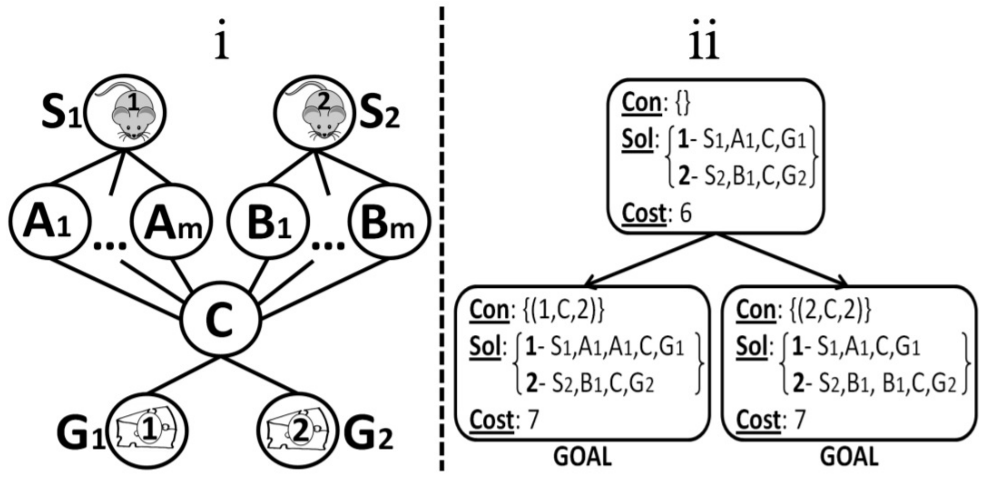


Figure 3.1: (i) visualization of MAPF instance, (ii) example of CT.

[10]

spatial dimensions and another dimension of time. For the search, any single-agent path finding algorithm can be used. In order to satisfy all constraints, whenever a state x is generated with $g(x) = t$, and if there exists a constraint (a_i, x, t), then this state is discarded.

3.4 Optimality

If there are solutions to a MAPF instance, CBS returns the optimal one for that. In order to prove the optimality of CBS, first we will provide some supporting claims.

Definition 1

Let $CV(N)$ be the set of all solutions for a *node* N in a *constraint tree*, that are: (1) consistent with the set of constraints of N and (2) also valid, such that they don't have any conflicts. If N is not a goal node, then solution at the node N ($N.solution$)will not be in $CV(N)$ because it is not valid.

Definition 2

For any solution $p \in CV(N)$, node N permits the solution p.

For example, root node in CT does not have any constraints. Any valid solution will satisfy the empty set of constraints. Therefore, we can say that the root node in CT permits all valid solutions.

Algorithm 2 CBS - High Level

```
Input: MAPF Instance
 1: $R.constraints = \emptyset$
 2: $R.solution$ = find individual paths using low-level()
 3: $R.cost = SOC(R.solution)$
 4: insert $R$ to $OPEN$
 5: while $OPEN$ not empty do
 6:     $N \leftarrow$ best node from $OPEN$ // lowest solution cost
 7:     if $N$ has no conflict then
 8:         return $N.solution$
 9:     end if
10:     $C \leftarrow$ first conflict $(a_i, a_j, v, t)$ in $N$
11:     for each agent $a_i$ in $C$ do
12:         $A \leftarrow$ new node
13:         $A.constraints \leftarrow N.constraints + (a_i, v, t)$
14:         $A.solution \leftarrow N.solution$
15:         Update $A.solution$ by invoking low-level($a_i, A.constaints$)
16:         $A.cost = SOC(A.solution)$
17:         Insert $A$ to $OPEN$
18:     end for
19: end while
```

The objective function used in CBS is sum of costs. Let $minCost(CV(N))$ be the minimum cost over all solutions in $CV(N)$.

Lemma 1

The cost of a node N in the CT is a lower bound on $minCost(CV(N))$.

Proof

For each node N in CT, paths for the associated agent is calculated by using an optimum path finding algorithm. That is to say, for a node N, the solution $N.solution$ contains the shortest paths that satisfy $N.constraints$ for each agent. Solution may not be valid, however it makes a lower bound on the set of valid solutions for N, since no agent can reach to its goal with a shorter path satisfying the constraints.

Lemma 2

At all time steps, there exists a CT node N in $OPEN$, which permits a valid solution p.

Proof

We are going to prove this lemma by induction. For the base case only root node is in $OPEN$, which does not have any constraints. We know that root node permits all valid solutions. Assuming this is true for the first i expansion cycles.In cycle $i + 1$ node N, which permits the optimal solution

p, had collisions and needed to be expanded to N'_1 and N'_2 with the new constraints. Any valid solution must satisfy either one the new constraints set. Any valid solution from $VS(N)$ is either in $VS(N'_1)$ or $VS(N'_2)$. Therefore, at all times there is at least one CT node in $OPEN$ that permits the optimal solution.

Theorem 1
CBS returns the optimal solution.
Proof
Let's assume a goal node G is chosen for expansion in the high-level expansion cycle, with a cost of $c(G)$. We know that at that point, all valid solutions are permitted by the nodes in $OPEN$ (lemma 2). Let p be a valid solution with the cost of $c(p)$ and let $N(p)$ be the node that permits p in $OPEN$, with the cost of $c(N(p))$. $c(N(p)) \leq c(p)$ (lemma 1). High-level in CBS expands the nodes of CT in a best-first manner. Therefore $c(G) \leq c(N(p)) \leq c(p)$. [10]

Chapter 4

Contributions

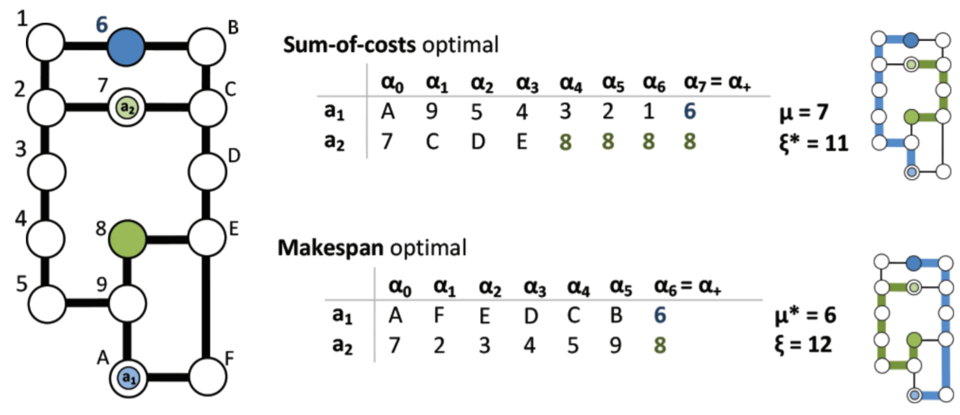
Optimal solution for a MAPF instance depends on the objective function defined for that problem. CBS algorithm is used with *sum of costs* objective function and it returns the optimal solution for that. This chapter will focus on how to adapt CBS for *makespan* objective function and possible improvements on the algorithm to find the optimal solution faster.

4.1 CBS with makespan

First thing to clarify is weather an optimal solution for a MAPF instance may differ between makespan and sum of costs objective functions or not. As discussed previously, makespan for a MAPF instance is the time until the last agent reaches its destination, whereas sum of costs is the sum of each path length in the solution. The figure 4.1, visualization of a MAPF instance with two agents a_1 and a_2, and two optimal solutions for *sum of costs* (denoted as ξ) and *makespan* (denoted as μ). This example proves that it is possible to have MAPF instances where *sum of costs* and *makespan* optimal solutions differ.

Intuitively, in order to adapt CBS algorithm for makespan objective function, we can change the total cost value for a CT node. In the original CBS, total cost value of a CT node is given as the sum of path lengths of that node's solution. If we change this function definition from sum of path lengths to maximum of the path lengths, this will make the nodes in $OPEN$ to get sorted by the longest path length in their solution. During the expansion cycle in the high-level of CBS, whenever we get a node from $OPEN$, we will know that there is no other node in $OPEN$ with a better solution in terms of makespan objective function.

We need to discuss if changing the objective function will effect the optimality of CBS or not. Referring back to section 3.4, *lemma 1* will hold since changing cost function does not affect the low-level search of CBS. Therefore



Sum-of-costs optimal

	α_0	α_1	α_2	α_3	α_4	α_5	α_6	$\alpha_7 = \alpha_+$	
a_1	A	9	5	4	3	2	1	6	$\mu = 7$
a_2	7	C	D	E	8	8	8	8	$\xi^* = 11$

Makespan optimal

	α_0	α_1	α_2	α_3	α_4	α_5	$\alpha_6 = \alpha_+$	
a_1	A	F	E	D	C	B	6	$\mu^* = 6$
a_2	7	2	3	4	5	9	8	$\xi = 12$

Figure 4.1: Visualization of a MAPF instance where *sum-of-costs* and *makespan* optimal solutions differ.

[11]

the cost of a node N in the CT is still a lower bound on $minCost(CV(N))$. Lemma 2 will hold because changing total cost function does not affect the if a node N permits a valid solution or not. Since lemma 1 and lemma 2 holds, and CBS expands the nodes of CT in a best-first manned in high-level no matter what the cost function is, it holds that CBS returns the optimal solution for makespan objective function as well.

We will formally prove CBS with makespan objective function will return optimal solution too. Referring back to the definitions in section 3.4 proof of CBS returning optimal solution is as follows.

Lemma 1
The cost of a node N in the CT is a lower bound on $minCost(CV(N))$.
Proof
For each node N in CT, paths for the associated agent is calculated by using an optimum path finding algorithm in decoupled manner since makespan is a minimization task. That is to say, for a node N, the solution $N.solution$ contains the shortest paths that satisfy $N.constraints$ for each agent. Since during the calculation of the paths, other agents are ignored no agent can reach to its goal with a shorter path satisfying only its own constraints. That is to say for a given constraint set $N.constraints$, agent N cannot reach its target location faster.

Lemma 2
At all time steps, there exists a CT node N in $OPEN$, which permits a valid solution p.

Proof

We are going to prove this lemma by induction. For the base case only root node is in $OPEN$, which does not have any constraints. We know that root node permits all valid solutions. Assuming this is true for the first i expansion cycles.In cycle $i+1$ node N, which permits the optimal solution p, had collisions and needed to be expanded to N_1' and N_2' with the new constraints. Any valid solution must satisfy either one the new constraints set. Any valid solution from $VS(N)$ is either in $VS(N_1')$ or $VS(N_2')$. Therefore, at all times there is at least one CT node in $OPEN$ that permits the optimal solution.

Theorem 1

CBS with makespan objective function returns the optimal solution.

Proof

Let's assume a goal node G is chosen for expansion in the high-level expansion cycle, with a cost of $c(G)$. We know that at that point, all valid solutions are permitted by the nodes in $OPEN$ (lemma 2). Let p be a valid solution with the cost of $c(p)$ and let $N(p)$ be the node that permits p in $OPEN$, with the cost of $c(N(p))$. $c(N(p)) \leq c(p)$ (lemma 1). High-level in CBS expands the nodes of CT in a best-first manner. Therefore $c(G) \leq c(N(p)) \leq c(p)$.

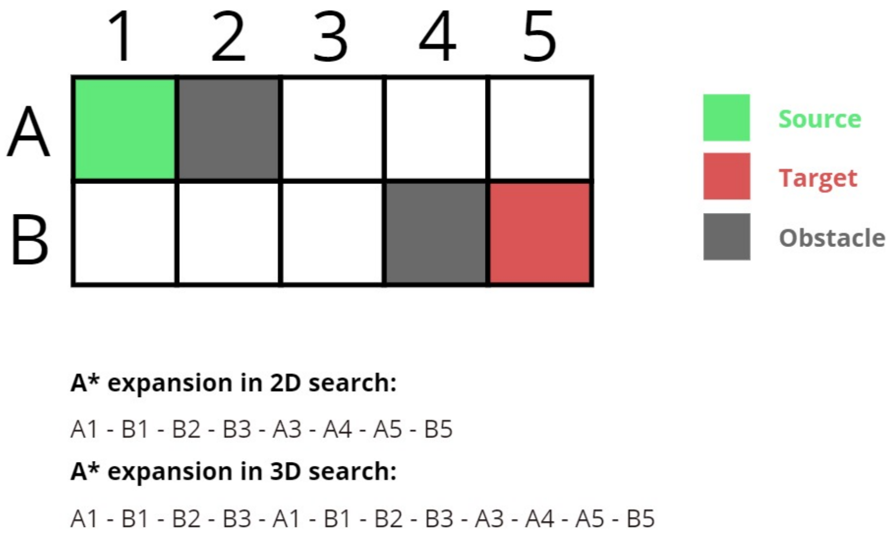
4.2 Improvements on high-level

At high-level, CBS performs an expansion cycle on a priority queue. Nodes in the queue are sorted by their total cost function and the node with lowest cost value will be removed from the queue in each turn. However, the algorithm makes no distinctions between two nodes with equal cost value. That is to say, algorithm work similar to BFS within same cost value nodes. This may increase the number of nodes expanded until a goal node is found since CT grows exponentially with the number of conflicts and goal nodes are the leaf nodes of a CT. More conflicts resolved, greater the chance is to find a goad node, that is why we may increase the possibility to find a goal node faster by using DFS for the nodes with same cost value.

4.3 Improvements on low-level

Each time a conflict occurs between two agents, low-level CBS is invoked for each agent to find a new optimal path with the new constraints. So, except for the root node, for each node in CT low-level is invoked once. Although there are no restrictions for the algorithm to be used in low-level, usually A* is used. This procedure seems to be efficient, since A* is one of the fastest path finding algorithms. However when searching for a path in three dimensions, two for spatial dimensions and one dimension for time, number of expanded nodes might increase dramatically.

To have a better understanding of this situation, let's focus on the example in figure 4.2, where we will demonstrate three dimensional A* search. In this example, we assume there is one agent with source node as $A1$ and target node as $B5$. As heuristic value, we will use the manhattan distance. Before we start, we can see that target node is located on the right and bottom of the source node. So ideally, agent should move right and bottom directions to be able to reach the target. However there are some obstacles, which forces the agent to move in opposite direction.



A* expansion in 2D search:

A1 - B1 - B2 - B3 - A3 - A4 - A5 - B5

A* expansion in 3D search:

A1 - B1 - B2 - B3 - A1 - B1 - B2 - B3 - A3 - A4 - A5 - B5

Figure 4.2: Example of A* expansion in 2D and 3D search

At the beginning, first node to be expanded will be the source node, $A1$, with $g = 0$, $h = 5$ and $f = 5$. After processing $A1$, $A1$ and $B1$ will be added to $OPEN$. $A1$ will have $g = 1$, $h = 5$ and $f = 6$, and $B1$ will have $g = 1$, $h = 4$ and $f = 5$. So after expansion of a node N, a copy of that node N' will be added to $OPEN$. f value of N' will be one more than f value of N since h value will remain same and g value will increase by one. Because of the higher f value, copied node will be added at the end of queue, and algorithm will keep expanding nodes with lower f values. This pattern will continue until agent will come across an obstacle, which forces it to move opposite direction of the target node. In this example, when node $B3$ is expanded, successors will be $B3$ with $g = 4$, $h = 2$, $f = 5$ and $A3$ with $g = 4$, $h = 3$ and $f = 7$. We know that for the shortest path, we supposed to expand $A3$, however since agent moved opposite direction, f value of $A3$ is higher than all the previous duplicate nodes (each with $f = 6$). This situation repeats each time agent is forced to move opposite direction by either an obstacle or a constraint.

To lower the number of expanded nodes in low-level CBS, we will build a **multi-value decision diagram** (MDD), and perform DFS on that MDD. MDD is a directed acyclic graph, which contains all paths for a given agent and a given maximum path length. Nodes at depth t of MDD for an agent a, corresponds to all the nodes that agent a can occupy at time step t. In figure 4.3 an example of MDDs for two agents with 4 time steps. Since searching path is two dimensional on MDD, it will return the path faster. However it is computationally costly to build MDDs, and also the length of the path should be known prior to constructing MDD. [12]

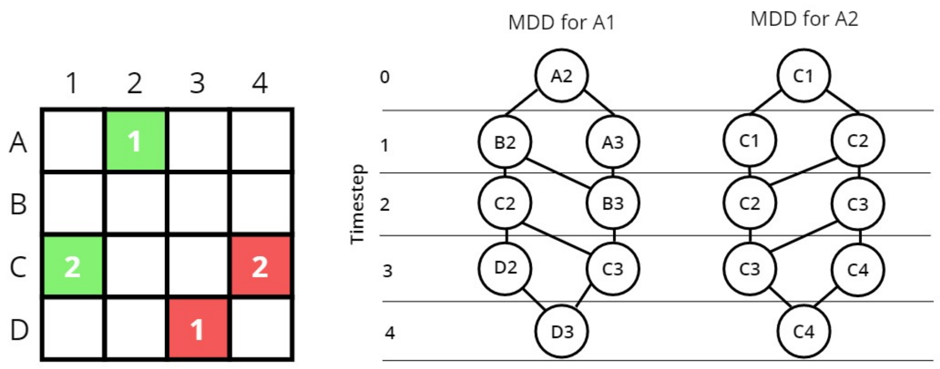


Figure 4.3: Example MDDs for 4 time steps

Choosing wrong path lengths may result in not returning the optimal solution for the MAPF instance. Since our objective function is *makespan*, we only care about the longest path in the solution. That is to say, path lengths of other agents, which have shorter path length in the solution, does not effect the quality of the solution. Using this information, at the beginning of the algorithm we will use A* on two dimensions, to find minimum paths for each agent in a *decoupled* manner. After finding all the paths, we will use the longest path length to construct all MDDs. We save this value in each node as threshold value, and if low-level CBS fails to return a path for a new CT node, we increase the threshold value of that node and add it to $OPEN$. If during the expansion cycle in high-level, a node has bigger threshold value than the depth of already constructed MDDs, then we construct new MDDs with this threshold value.

Chapter 5

Experiments

In this chapter we will discuss the experimental results of CBS, CBS with makespan (CBS_MS), CBS_MS with high-level improvements (CBS_MS_HL) and CBS_MS with both high-level and low-level improvements (CBS_MS_HLLL). We will focus on three metrics, success rate, average runtime and average number of node extension in high level and low-level. Success rate is the ratio of successfully solved instances to total number of instances in a given time limit. Average runtime will be calculated for successfully solved instances by all solvers among different number of agents. Similarly average number of nodes will also be calculated among successfully solved instances by all solvers for all given number of agents.

5.1 Dataset

For the experiments 5 grid type 2D MAPF benchmark maps were used. [13] One small map (8x8) without obstacles (empty), three medium size maps (32x32) with different types of obstacles (random, room and maze) and one big map (194x194) with obstacles (lak303d). Visualization of the maps can be seen in the figure 5.1.

5.2 Setup

Experiments were run under following assumptions of MAPF instances:

- Maps used are 4 connected grids, i.e. an agent can move up, down, left, right or stay at its node.
- Each agent occupies only one grid in a time step.
- Edge weights are 1.
- Vertex conflict, edge conflict and swapping conflicts are not allowed.

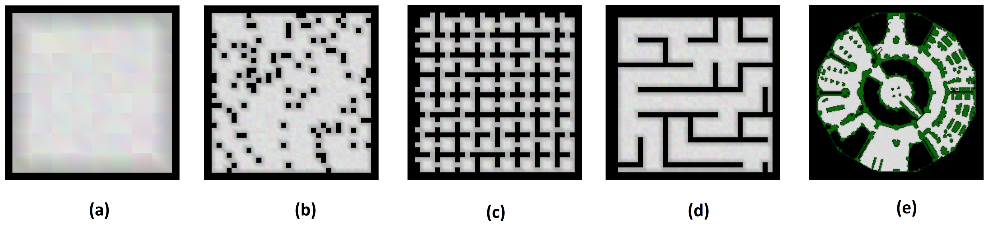


Figure 5.1: Maps used in experiments. (a) Empty, (b) Random, (c) Room, (d) Maze, (e) lak303d.

- Agents will stay at their targets.

Experiments were run on a computer with AMD Ryzen 1700 at 3.9Ghz CPU and 16GB DDR4 at 3433Mhz RAM.

5.3 Implementation

Algorithms and tests were implemented using Ptyhon 3.7. PyCharm was used as IDE since it provides easy refactoring and debugging tools. Implementation of the algorithms was done in *object oriented* manner. Solver classes were *inherited* from the base CBS solver class, which saved time and also resulted in clean and readable code. During tests *polymorphism* was used to solve and evaluate different algorithms. Software was designed to be flexible to evaluate custom MAPF instances with different time limits and number of agents. Results of the experiments are automatically saved in CSV files under the *results* folder.

5.4 Results

In this section comparison results of CBS algorithm under sum-of-costs and makespan objective functions will be presented. Also comparison of the suggested improvements and their effectiveness will be discussed. Besides giving numerical results, pros and cons of the algorithms will be analyzed based on the experiments.

5.4.1 Small map

Comparison CBS and CBS-MS in small map shows that, CBS-MS performs better. In figure 5.3 we can see that runtime of CBS increase much faster than CBS-MS. Also figure 5.2 show that success rate of CBS drops more

Table 5.1: Average High Level Node Expansion in 8x8 Empty Grid.

k	CBS	CBS-MS	CBS-MS-HL	CBS-MS-HLLL
5	2.5	2.8	2.2	2.7
10	65.5	11.7	7.2	8.7
15	NA	36.5	20.4	24.9
20	NA	89	45	50
25	NA	258	103	114

dramatically than CBS-MS. The reason can be seen in the table 5.1 more clearly. Although number of nodes expanded for CBS and CBS-MS for 5 agents is similar, CBS expands almost six times more nodes, than CBS-MS for 10 agents.

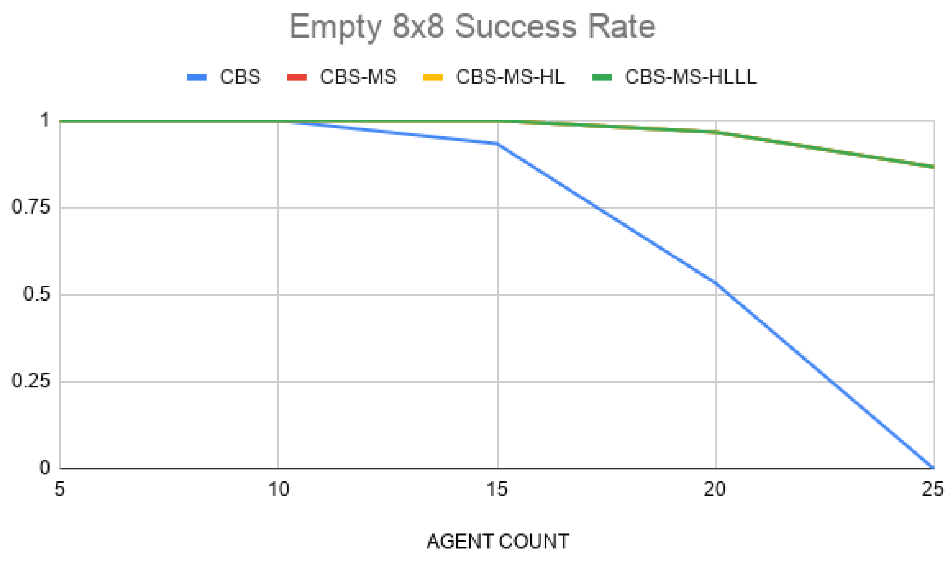


Figure 5.2: Success rate in 8x8 empty grid.

Comparing CBS-MS-HL and CBS-MS-HLLL, as figure 5.3 indicates, low level improvements gain more importance as the number of agents increase. Although until 20 agents CBS-MS-HL outperforms CBS-MS-HLLL, the pattern shows us that for higher number of agents CBS-MS-HLLL will perform better.

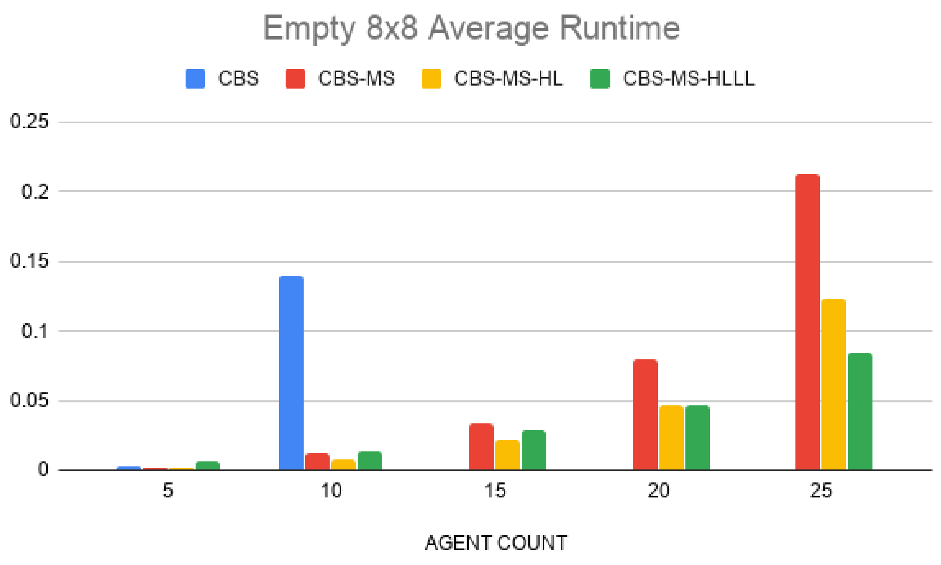


Figure 5.3: Average runtime (sec) in 8x8 empty grid.

5.4.2 Medium maps

CBS-MS performs better than CBS in all tested medium size maps. In success rate figures 5.4, 5.5 and 5.6 we can see that success rate of CBS is dominated by CBS-MS. However the gap of success rate of CBS and CBS-MS differs between maps. Success rate figures shows that performance of CBS effected by the type of the map more than CBS-MS. For example, maze map is much more error prone than random map, since it consist of narrow corridor which limits the movement of agents more than a 10% randomly blocked map. That is why the gap between the performances of CBS and CBS-MS in maze increases faster in maze than random.

Results of high level node expansion in table 5.2 and table 5.3 is parallel with the results of success rates between CBS and CBS-MS. Even for small number of agents, number of nodes expanded in high level CBS is double in maze than random map. Furthermore, the increase in number of expanded nodes with increasing number of agents is much higher in CBS than CBS-MS. For example for the random map, from 5 agents to 10 agents, CBS-MS expands 5 more nodes in average, whereas CBS expands 34 more nodes in average. This gap gets bigger as the number of agents increase, which makes CBS to solve the problem slower than CBS-MS.

Except for the maze map, CBS-MS-HL and CBS-MS-HLLL seems to perform similarly according to success graphs. However, when average runtime graphs, graph 5.7 and graph 5.8 are inspected, we can see that average runtime of CBS-MS-HL increases faster than average runtime of CBS-MS-HLLL, with the increasing number of agents. Difference is more dramatic in maze grid, since in a maze an agent may need to move opposite direction of its goal, which makes low level search to expand much more nodes with conventional A* search as it was explained in section 4.3.

High level node expansion comparison of CBS-MS-HL and CBS-MS-HLLL shows that CBS-MS-HLLL may expand more nodes than CBS-MS-HL as it can be seen in the table 5.3 and table 5.2. Although the difference is much more significant in maze grid, CBS-MS-HLLL performs much better in average runtime, as it can be seen in figure 5.8. That is to say, low level node expansion performance of CBS-MS-HLLL more than enough to compensate for the worse high level performance.

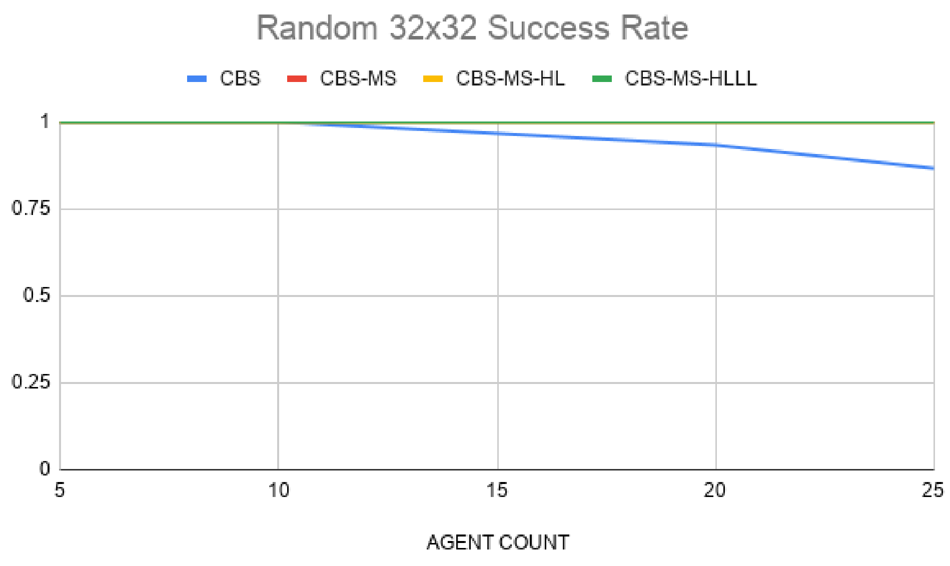


Figure 5.4: Success rate in 32x32 random obstacle grid.

5.4.3 Big map

Lak303d was the most difficult map to solve among the selected dataset. Maze like structure of lak303d not makes it difficult for low level algorithm to find the individual paths, also it increases the possibility of having conflicts and makes it difficult for high level too. Comparison results show more clearly

Table 5.2: Average High Level Node Expansion in 32x32 Random Grid.

k	**CBS**	**CBS-MS**	**CBS-MS-HL**	**CBS-MS-HLLL**
5	1.1	1.8	2.3	1.5
10	35	6.7	4.3	5.9
15	721	18.8	8.1	13.2
20	732	27	11.6	13.8
25	NA	51.3	21.7	36.7

Table 5.3: Average High Level Node Expansion in 32x32 Maze Grid.

k	**CBS**	**CBS-MS**	**CBS-MS-HL**	**CBS-MS-HLLL**
5	2.2	6.9	5.2	7.2
10	52.4	26	16.2	27.7
15	NA	60.5	35.3	87.8
20	NA	134	63.9	173.4
25	NA	250.8	107.1	255.5

Table 5.4: Average Low Level Node Expansion (in thousands) in 32x32 Random Grid.

k	**CBS**	**CBS-MS**	**CBS-MS-HL**	**CBS-MS-HLLL**
5	0.6	0.6	0.6	0.4
10	16	4	3	1
15	168	12	9	2
20	189	18	12	3
25	NA	28	18	5

Table 5.5: Average Low Level Node Expansion (in thousands) in 32x32 Maze Grid.

k	**CBS**	**CBS-MS**	**CBS-MS-HL**	**CBS-MS-HLLL**
5	NA	74	54	2
10	NA	271	168	5
15	NA	580	355	19
20	NA	1181	748	210
25	NA	2057	1267	229

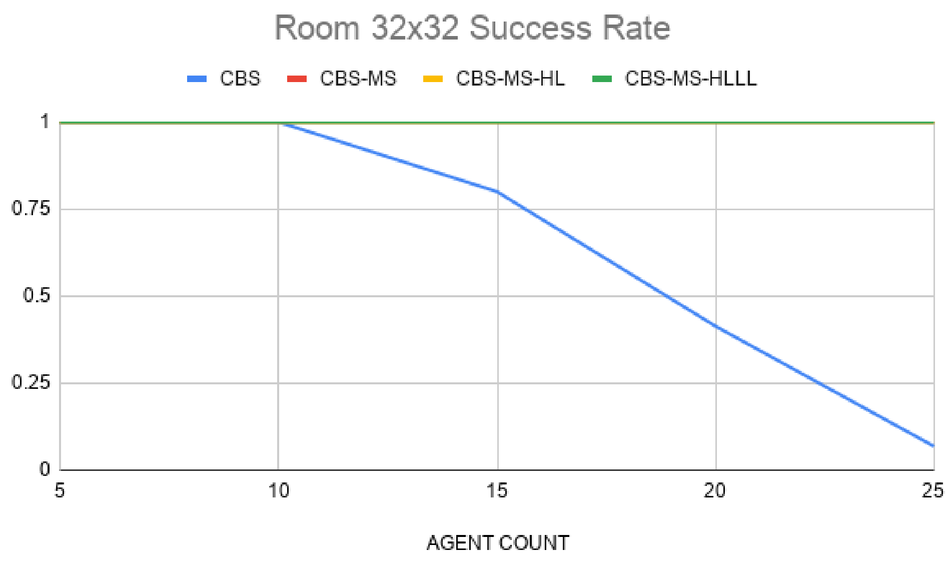


Figure 5.5: Success rate in 32x32 room grid.

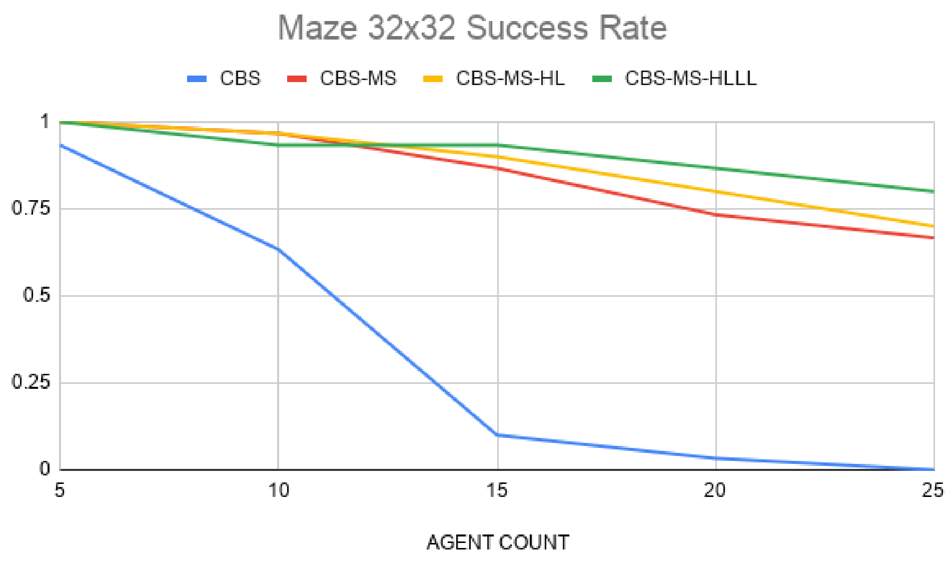


Figure 5.6: Success rate in 32x32 maze grid.

which algorithm performs in difficult scenarios better. Looking at the suc-

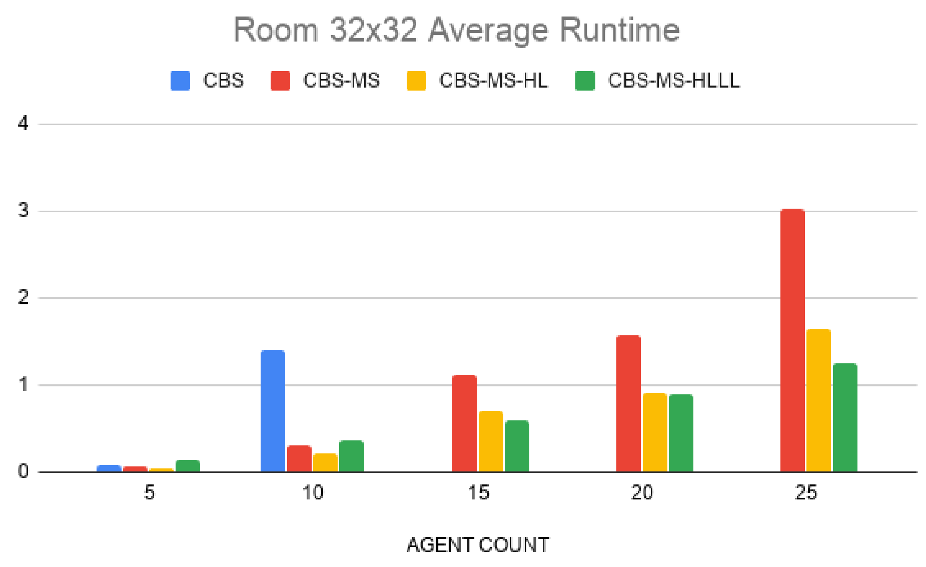


Figure 5.7: Average runtime (sec) in 32x32 room grid.

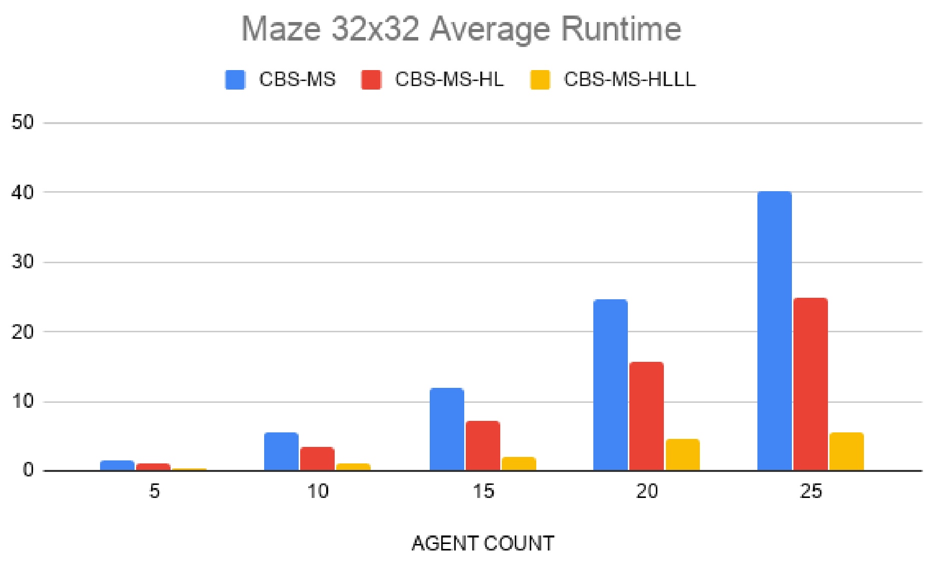


Figure 5.8: Average runtime (sec) in 32x32 maze grid.

cess rate figure 5.9, we can observe CBS has the worst performance among all other algorithms. CBS-MS-HLLL on the other hand is performing as the best algorithm overall. It can be observed that for some situations it does not perform best, such as for 15 and 20 agents in this case, but the trend of indicates it will perform better for higher number of agents. Finally, comparison between CBS-MS and CBS-MS-HL shows that CBS-MS-HL strictly dominates CBS-MS in this map.

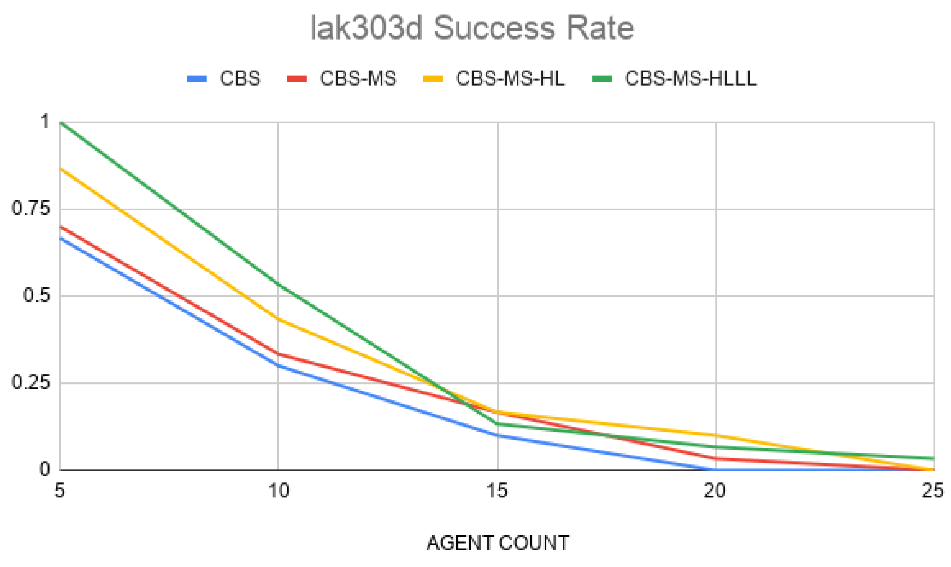


Figure 5.9: Success rate in 194x194 DAO-lak303d map.

Conclusion

In this paper we analyzed the adaptation of CBS for makespan objective function. We purposed two improvements for the algorithm, one addressing high-level search and one addressing low-level search. Although it is not possible to talk about domination of one version of the algorithm over others, we can conclude that CBS is a very flexible algorithm and it can be adjusted and optimized for different kind of MAPF problems. Empirical comparisons show that, these adjustments can lower the amount of nodes during search up to a magnitude of 10.

For future work adaptation of pruning techniques for makespan objective function can be studied to prune non-goal CT nodes without the need to invoke the low-level search of CBS [14]. Also introducing heuristics for high level search [12] for makespan can be studied to increase the performance of CBS by changing the node expansion order in high-level search. Also combination of different improvements, their compatibility and performance can be investigated.

Bibliography

[1] Stern, R.; Sturtevant, N.; Felner, A.; et al. Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks. *arXiv preprint arXiv:1906.08291*, 2019.

[2] Bennewitz, M.; Burgard, W.; Thrun, S. Finding and optimizing solvable priority schemes for decoupled path planning techniques for teams of mobile robots. *Robotics and autonomous systems*, volume 41, no. 2-3, 2002: pp. 89–99.

[3] Dresner, K.; Stone, P. A multiagent approach to autonomous intersection management. *Journal of artificial intelligence research*, volume 31, 2008: pp. 591–656.

[4] Pallottino, L.; Scordio, V. G.; Bicchi, A.; et al. Decentralized cooperative policy for conflict resolution in multivehicle systems. *IEEE Transactions on Robotics*, volume 23, no. 6, 2007: pp. 1170–1183.

[5] Dechter, R.; Pearl, J. Generalized best-first search strategies and the optimality of A. *Journal of the ACM (JACM)*, volume 32, no. 3, 1985: pp. 505–536.

[6] Ratner, D.; Warrnuth, M. Computer & Information Sciences University of California Santa Cruz Santa Cruz, CA 95064.

[7] Silver, D. Cooperative Pathfinding. *AIIDE*, volume 1, 2005: pp. 117–122.

[8] Hönig, W.; Kumar, T. S.; Cohen, L.; et al. Multi-agent path finding with kinematic constraints. In *Twenty-Sixth International Conference on Automated Planning and Scheduling*, 2016.

[9] Standley, T. S.; Korf, R. Complete Algorithms for Cooperative Pathfinding Problems. In *Twenty-Second International Joint Conference on Artificial Intelligence*, 2011.

[10] Sharon, G.; Stern, R.; Felner, A.; et al. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence*, volume 219, 2015: pp. 40–66.

[11] Surynek, P.; Felner, A.; Stern, R.; et al. An empirical comparison of the hardness of multi-agent path finding under the makespan and the sum of costs objectives. In *Ninth Annual Symposium on Combinatorial Search*, 2016.

[12] Li, J.; Felner, A.; Boyarski, E.; et al. Improved heuristics for multi-agent path finding with conflict-based search. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence*, AAAI Press, 2019, pp. 442–449.

[13] Sturtevant, N. R. Benchmarks for grid-based pathfinding. *IEEE Transactions on Computational Intelligence and AI in Games*, volume 4, no. 2, 2012: pp. 144–148.

[14] Sharon, G.; Stern, R. T.; Goldenberg, M.; et al. Pruning techniques for the increasing cost tree search for optimal multi-agent pathfinding. In *Fourth Annual Symposium on Combinatorial Search*, 2011.

Appendix **A**

Acronyms

GBFS Greedy Best-First Search

MAPF Multi Agent Path Finding

BFS Breath First Search

DFS Depth First Search

HCA* Hierarchical Cooperative A*

CBS Conflict Based Search

CT Constraint Tree

MDD Multi-value Decision Diagram

IDE Integrated Development Environment

Appendix B

Contents of enclosed USB

```
 readme.txt ....................... the file with CD contents description
 src ...................................... the directory of source codes
   mapf .................................... implementation sources
   thesis .............. the directory of LaTeX source codes of the thesis
 text ........................................ the thesis text directory
   thesis.pdf ......................... the thesis text in PDF format
```