MAPDC: MULTI-AGENT PICK AND DELIVERY WITH CAPACITIES

by
ÇAĞRI ULUÇ YILDIRIMOĞLU

Submitted to the Graduate School of Natural Sciences
in partial fulfilment of
the requirements for the degree of Master of Science

Sabancı University
May 2022

ABSTRACT

MAPDC: MULTI-AGENT PICK AND DELIVERY WITH CAPACITIES PROBLEM

ÇAĞRI ULUÇ YILDIRIMOĞLU

COMPUTER SCIENCE M.A. THESIS, JULY 2022

Thesis Supervisor: Prof. Esra Erdem

Keywords: multi-agent path finding, answer set programming, multi agent pick and delivery with capacities, warehouse automation

Motivated by autonomous warehouse applications in the real world, we study a variant of Multi-Agent Path Finding (**MAPF**) problem where robots also need to pick and deliver some items on their way to their destination. We call this variant the Multi-Agent Pick and Delivery with Capacities (**MAPDC**) problem. In addition to the challenges of **MAPF** (i.e., finding collision-free plans for each robot from an initial location to a destination while minimizing the maximum makespan), **MAPDC** asks also for the allocation of the pick and deliver tasks among robots while taking into account their capacities (i.e., the maximum number of items one robot can carry at a time). We mathematically model this problem as a graph problem, and introduce novel methods using Answer Set Programming with different computation modes: single-shot, anytime, incremental, and hierarchical. We compare these methods empirically with randomly generated instances over various sizes and types of environments.

ÖZET

KAPASİTELİ ÇOK ETMENLİ TOPLAMA VE DAĞITMA PROBLEMİ

ÇAĞRI ULUÇ YILDIRIMOĞLU

BİLGİSAYAR BİLİMİ YÜKSEK LİSANS TEZİ, TEMMUZ 2022

Tez Danışmanı: Prof. Dr. Esra Erdem

Anahtar Kelimeler: çok etmenli yörünge bulma, çözüm kümesi programlama, kapasiteli çok etmenli toplama ve dağıtma, otonom depolar

Gerçek hayattaki otonom depo uygulamalarından ilham alarak, çok etmenli yörünge bulma probleminin toplama ve dağıtma operasyonlarını dahil eden bir versiyonu olan kapasiteli çok etmenli toplama ve dağıtma problemine (**MAPDC**) yenilikçi bir çözüm öneriyoruz. Çok etmenli yörünge bulma probleminin zorluklarına (örneğin, etmenlerin birbirleriyle çarpışmayacak şekilde en kısa yörüngeleri hesaplaması) ek olarak , **MAPDC** probleminin verilen toplama-dağıtma işlerinin etmenlerin kapasiteleri dahilinde en makul şekilde dağıtılması gibi, kendine has zorlukları bulunmaktadır. Bu tezde **MAPDC** problemini matematiksel olarak bir çizge problemi olarak modelleyip, çözüm kümesi programlama teknikleriyle çözüm yöntemleri sunuyoruz. Tekli-deneme, çoklu-deneme, herhangi-zaman, artımlı veya hiyerarşik olan bu yöntemleri rastgele yarattığımız örnekler üzerinde deneysel olarak test edip karşılaştırıyoruz.

ACKNOWLEDGEMENTS

I would like to thank my family, my mother Şebnem, my father Murat, and my Comrade Brother Tolga, and my small girl Boncuk for their continuous support throughout the years.

I also thank my dear girlfriend Şimal for being a part of my life and being a trusted companion to me.

To my jury members, I would like to express my gratitude for their time and effort. I would like to thank my thesis advisor Prof. Esra Erdem for giving me directions and teaching me many things.

I thank all the friends I made along the way. I am thankful to my fellow lab members Cemal, Müge, Aysu and Selin. Aysu and Selin also shared material and knowledge with me, which helped a lot. I also thank my friends Ali and Berker for their friendship.

Lastly, I would like to acknowledge the Technological Research Council of Turkey (TUBİTAK) for partially supporting this work under Grant 118E431.

To my undeniable luck that let me finish this work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. Introduction

Multi-agent path finding (**MAPF**) is the problem of finding paths for agents starting from their given initial locations on a graph to their goal destinations such that the agents do not collide with each other. Although single-agent path finding is a polynomial problem that can be solved by Dijsktra's pathfinding algorithm, makespan optimization variant of **MAPF** is NP-hard (Surynek, 2010). Makespan of a **MAPF** solution is the maximum of path lengths of all agents, and the variant is concerned with finding the minimum makespan that a solution exists.

In the last decade, there has been an increased interest in variants of **MAPF** for different settings. MAPD (Liu et al., 2019), TAPF (Ma & Koenig, 2016), GTAPF (Nguyen et al., 2019) are such variants that include tasks that must be completed by agents, in order to simulate environments such as warehouses. The nature of tasks in these problems usually involve moving through pick and delivery locations for items, or additionally visiting some processing stations along the way.

There are good reasons to be interested in warehouse environments. The room for automation, especially for parts of the material flow that require hauling items from place to place, is considerable. Picking and shipping items account for 55% of operating costs for warehouses. The global pandemic increased the demand for e-commerce and this further emphasizes the importance of order-picking (Guthrie et al., 2021). Traveling between points constitute the greatest expense for order-picking operations, creating incentives to automate travelling between pick and delivery locations.

This interest in warehouse environments with robotic agents motivated us to create a similar variant to the mentioned ones, which we call the Multi-agent Pick and Delivery with Capacities (**MAPDC**) problem. In this setting, agents have their initial and goal locations, but they also need to collectively complete given tasks. Tasks in our case are pick-and-delivery tasks, where an item needs to be picked from a node in the graph and delivered to another. In real world scenarios, agents could be expected to be able to carry more than a single item, so we include capacities into consideration in our formulations.

In this thesis, we study **MAPDC** and its variants, and introduce novel methods to solve them. Our contributions can be summarized as follows.

- We mathematically model **MAPDC** as a graph problem and define it as a computational problem.
- We introduce two novel exact methods to solve **MAPDC** using Answer Set Programming (ASP) (Marek & Truszczyński, 1999; Lifschitz, 2019; Brewka et al., 2016; Gebser et al., 2019; Lifschitz, 2002), based on two types of computation: single-shot ASP and multi-shot ASP (based on incremental computation). In each method, we formally represent **MAPDC** in an expressive language of ASP, and compute solutions using the ASP solver clingo (Gebser et al., 2019).
- To improve the computational performance, we introduce a novel algorithm that computes a solution to **MAPDC** hierarchically using our multi-shot ASP method and in connection with a suite of heuristics.
- We experimentally evaluate our ASP based methods over randomly generated instances, with respect to scalability and quality measures, considering different computation modes: single-shot, anytime, multi-shot, hierarchical.

The rest of thesis is structured as follows. Section 2 provides preliminaries. We define the **MAPDC** in Section 3 and present solutions to the problem in Sections 4 and 5. Then we explain our experimental setup in Section 6, report on results in Section 7, make some discussions on the results and future work in Section 8, and conclude this dissertation in Section 10.

2. Preliminaries

We define some preliminary concepts that are relevant to our problem and how we solve it. First, we describe what Answer Set Programming is, which is the basis for our methodology to solve the **MAPDC** problem. Then, we formally define the **MAPF** problem, which we derive **MAPDC** from.

2.1 Answer Set Programming

Based on the stable model semantics (Gelfond & Lifschitz, 2000), Answer Set Programming is a declarative programming method. ASP programs are a set of rules of the form

$$A \leftarrow L_1, \dots, L_n$$

Where L_1 to L_n are literals, each of them is either an atom or a negated atom. A is an atom or $\perp$. L_1 to L_n is called the body and A is called the head.

ASP rules correspond to logic statements *body* $\implies$ *head* where we have $\wedge$ between all the literals in the body. If the literals in the body are interpreted as $\top$, the atom in the head must also be interpreted as such. *Answer sets* to an ASP program are stable models of the resulting logical system.

We can also have *cardinality expressions* in the place of atoms. Cardinality expressions take the form $l\{A_1, \dots, A_m\}u$ where A_1 to A_m are atoms, and l and u are lower and upper bounds respectively. Such an expression is interpreted as $\top$ if and only if at least l and at most u atoms in the expression are interpreted as $\top$. If l and u are omitted, it is shorthand for $l = 0$ and $u = 1$. Cardinality expressions can also be in the head of rules.

When the body of a rule is $\top$, we call the rule a *fact*. We use facts to represent our knowledge of the system we are trying to solve. For example, we can represent instances of problems as facts.

If the head of a rule is $\bot$, then we call that rule a *constraint*. In such a rule if the body is interpreted as $\top$, the rule would imply $\bot$. This would make the interpretation unfit to be a stable model.

If we have a cardinality expression in the head of a rule, we will call this rule a *choice rule*. For example, if we have a rule of the form

$$\{p, q\}$$

then we must have either p or q in our answer set to respect the cardinality.

To solve problems using ASP, we can follow the paradigm of *generate-and-test* (Lifschitz, 2019). With this method, we *generate* possible solutions using choice rules and then *test* these possible solutions via constraints.

In order to write ASP programs that are at least marginally complex, we need to utilize *schematic variables*. Consider the following program

$$q(1)$$

$$q(2)$$

$$p(A) \leftarrow q(A)$$

In this program, A is a schematic variable and $q(1)$, $q(2)$ are given facts. Since we can have a variable, we do not need to write a copy of the last rule for each i where $q(i)$ might be $\top$. The program still needs all possible rules that can be derived from the last one. This process is called *grounding* and there are readily available *grounders* such as gringo (Gebser et al., 2019).

The ASP solver clingo (Gebser et al., 2019) uses gringo to ground an input program, and then solves the grounded problem.

2.2 MAPF: Multi Agent Path Finding

MAPF problem is a popular problem with many versions, as can be seen in survey studies (Stern et al., 2019). When we look at the reason for interest, some studies cite autonomous warehouse applications of **MAPF** (Tajelipirbazari et al., 2022; Liu et al., 2019; Ma & Koenig, 2016).

In **MAPF**, we are given a graph and a set of agents with their initial and goal locations. The problem is to find a traversal for each agent such that the agents respect some chosen constraints such as collision constraint and edge constraint, subject to some decision or optimization criteria such as finding the minimum makespan that a solution to the given instance exists. This makespan optimization is our main concern for this study since it is both interesting and arguably suitable for ASP applications.

<u>**MAPF**</u>

Input:

- A nonempty set $A = \{a_1, \ldots, a_n\}$ of agents ($n > 0$).
- A graph $G = (V, E)$ (to describe an environment where the agents move around).
- A function $\mathit{init} : A \mapsto V$ (to describe the initial locations of agents).
- A function $\mathit{goal} : A \mapsto V$ (to describe the goal locations of agents).
- A set $O \subseteq V$ (to denote the obstacles in the environment).
- A positive integer t (to specify the maximum makespan—plan length).

Output: For every agent $a_i \in A$, for some positive integer $u \leq t$,

- a path $P_i = \langle w_{i,1}, \ldots, w_{i,n_i} \rangle$ of length n_i ($n_i \leq u$)
  - that the agent a_i will follow to reach its goal location from its initial location (i.e., $w_{i,1} = \mathit{init}(a_i)$ and $w_{i,n_i} = \mathit{goal}(a_i)$),
  - without colliding with any obstacles (i.e., $w_{i,j} \in V \setminus O$), and
- a traversal f_i of the path P_i within time u, such that
  - for every other agent $a_j \in A$ with a path P_j and its traversal f_j within u, $f_i(P_i)$ and $f_j(P_j)$ do not collide with each other.

Figure 2.1 A decision version of **MAPF**.

3. MAPDC: Multi Agent Pick and Delivery with Capacities

Since warehouse environments are among the **MAPF** settings that see reasonable interest, it is fitting to formulate a problem that can better accommodate to this specific setting. In addition to the input graph, agents, their initial and goal locations, we are also given a set of tasks. These tasks are two-step hauling operations where an item needs to be picked up from somewhere and to be delivered to another location. Thus, each task has a pick and deliver node on the graph. Since we introduced the notion of carrying items, we also introduce capacities for each agent. Our formulations are for homogeneous agents, thus the agents have the same capacity.

We proceed with formally defining the **MAPDC** problem, determining its complexity class, and provide an example of a **MAPDC** instance and its solution.

3.1 Problem Definition

We view the environment as a graph $G = (V, E)$. A path P_i that an agent $i \in A$ traverses in this graph is characterized by a sequence $\langle w_{i,1}, w_{i,2}, \ldots, w_{i,n_i} \rangle$ of vertices such that $\{w_{i,j}, w_{i,j+1}\} \in E$ for all $j < n$. A *traversal* $traversal_i$ of a path P_i by an agent i within some time u_i ($u_i \in \mathbb{Z}^+$) is a function that maps each time step $x \leq u_i$, to a vertex in P_i describing the location of agent i at time x.

For two agents $i, j \in A$, they collide with each other at time step x if they are at the same location (i.e., $traversal_i(x) = traversal_j(x)$) or when they are swapping their locations (i.e., $traversal_i(x) = traversal_j(x-1)$ and $traversal_i(x-1) = traversal_j(x)$).

Each given task (id, p, d) is associated by a product id that needs to be picked up at some location $p \in V$ and delivered at some other location $d \in V$. We assume pickup and delivery takes no time. Each agent has a limited capacity to carry at most c number of tasks. We describe the *bag of an agent* $i \in A$ by a function $carry_i$ that maps every time step $x \leq u_i$ to a set of tasks (i.e., products) that the agent is carrying at that time step.

As the tasks are completed during the traversal $\mathit{traversal}_i$ of P_i, we need to pay attention to that the tasks are picked up before they are delivered, and the number of items carried by agent i is not more than its capacity c. We say that an agent *i completes a task* $(id,p,d) \in T_i$ within time u_i if there exist a pickup time x and a delivery time y $(0 \leq x < y \leq u_i)$ such that $\mathit{traversal}_i(x) = p$, $\mathit{traversal}_i(y) = d$ and $(id,p,d) \in \mathit{carry}_i(z)$ for every time step z between x and y only.

An agent i finishes its traversal in $u_i \leq t$ time steps. After time step u_i, the agent stays at its goal location. We define $\mathit{traversal}_i$ for time steps greater than u_i as a constant function: $\mathit{traversal}_i(x) = \mathit{goal}(i), u_i \leq x \leq t$.

Let $\mathit{traversal}_i$ and $\mathit{traversal}_j$ be traversals of two different paths P_i and P_j, respectively, in a graph G within some time t. We say that the traversals $\mathit{traversal}_i$ and *$\mathit{traversal}_j$ do not collide with each other* within time t if,

- for every time step $x, x' \leq t$, if $\mathit{traversal}_i(x) = \mathit{traversal}_j(x')$ then $x \neq x'$ (i.e., if the same vertex is visited by paths P_i and P_j, then it should be visited at different times — no two agents can be at the same location at the same time);
- for every time $x < t$, if $\mathit{traversal}_i(x) = \mathit{traversal}_j(x+1)$ then $\mathit{traversal}_i(x+1) \neq \mathit{traversal}_j(x)$ (i.e., an edge cannot be visited by paths P_i and P_j in reverse directions at the same time — no two agents can swap their locations at the same time).

Based on these notations, **MAPDC** problem can be defined as a graph problem as illustrated in Figure 3.1. Given the environment G whose some parts are occupied by obstacles O, the initial and goal locations for each agent, a set T of all tasks to be handled by the agents, the goal is to allocate the tasks in T to all agents, and, for each agent i, to find a path P_i and a collision-free traversal $\mathit{traversal}_i$ of P_i ensuring that agent i completes the allocated set T_i of tasks by a time step $u_i \leq t$.

3.2 Complexity of MAPDC

We can use the fact that the makespan optimization variant of **MAPF** is NP-hard to conclude that **MAPDC** is NP-hard. Since **MAPF** is NP-hard, and is a special case of **MAPDC** where $T = \emptyset$, **MAPDC** is NP-hard.

Theorem 1. *Makespan optimization variant of* **MAPDC** *is NP-Hard.*

MAPDC

Input:

- A graph $G=(V,E)$ (to describe the environment).
- A set $O \subseteq V$ (to denote the obstacles in the environment).
- A set A of agents.
- A function $\mathit{init}: A \mapsto V$ (to describe the initial locations of agents).
- A function $\mathit{goal}: A \mapsto V$ (to describe the goal locations of agents).
- A set T of tasks (id,p,d) (with unique identifier id, and pick-up and delivery locations $p,d \in V$).
- A positive integer t (to specify the maximum makespan).
- A positive integer c (to specify the capacity of each agent).

Output: For every agent $i \in A$, for some positive integer $u_i \leq t$,

- a set T_i of tasks allocated to the agent i where $\bigcup_{j\in A} T_j = T$ and
  - for every $j \in A$, $i \neq j$ implies $T_i \cap T_j = \emptyset$;
- a path $P_i = \langle w_{i,1}, \ldots, w_{i,n_i}\rangle$ of length n_i $(n_i \leq u_i)$ that the agent i can traverse
  - to reach its goal location $w_{i,n_i} = \mathit{goal}(i)$ from its initial location $w_{i,1} = \mathit{init}(i)$,
  - to complete the allocated tasks T_i (i.e., for every $(\mathit{id},p,d) \in T_i$, there exists $w_{i,j}, w_{i,k} \in P_i$ where $j \leq k$, $w_{i,j} = p$ and $w_{i,k} = d$,
  - without colliding with any obstacles (i.e., $w_{i,j} \in V \setminus O$); and
- a collision-free traversal $\mathit{traversal}_i$ of the path P_i within time u_i and how the bag of the agent i changes during this traversal (i.e., carry_i) such that
  - every task in T_i is completed by the agent i with respect to its traversal, and
  - for every $x \leq u_i$, the agent i can carry as many tasks as its capacity: $|\mathit{carry}_i(x)| < c$.

Figure 3.1 **MAPDC** problem definition

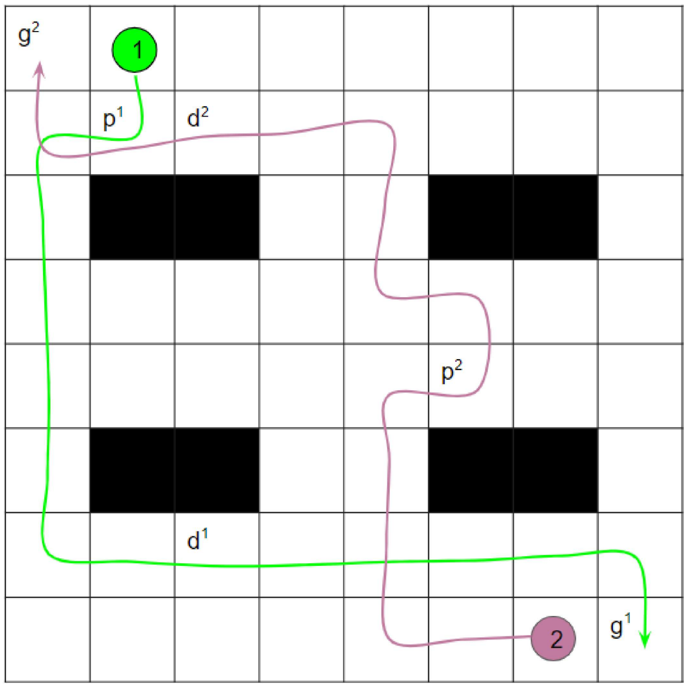
Proof. One can reduce any **MAPF** instance to a **MAPDC** instance, by keeping all inputs the same and choosing the set of tasks for **MAPDC** as $T = \emptyset$. Clearly, the resulting **MAPDC** instance has a solution iff the original **MAPF** has a solution. □

Furthermore, the decision version of the optimization variant of **MAPF** is NP-Complete (Surynek, 2010). Likewise, the decision version of **MAPDC** is NP-complete.

Theorem 2. **MAPDC** *is NP-complete.*

Proof. By Theorem 1 we know that **MAPDC** is NP-hard. We proceed to show that **MAPDC** $\in NP$. **MAPDC** has task assignments, paths, traversals of their paths and a bag description. Given that the makespan of **MAPDC** is a polynomially bounded, we can trace the traversals and bag descriptions of agents for all time steps to check whether: i) they only move through edges, ii) they start from their initial locations and end up at their goal locations, iii) they complete all their assigned

Figure 3.2 An example **MAPDC** instance and its solution



tasks, iv) they stay within their capacities. All this can be done in linear time on the length of the traversals. Thus, we can confirm whether a given solution candidate is a solution to the **MAPDC** in polynomial time and **MAPDC** $\in NP$. □

3.3 An Example for MAPDC

Consider the instance in Figure 3.2. There are two agents which are colored green and magenta. g^1 and g^2 are the goal locations. We also have two tasks, with pick and deliver locations p^1, d^1 and p^2, d^2. Black cells are obstacles and cannot be traversed by any agent.

A solution to this instance is given in the same figure. Ideally, the tasks are assigned in a way that minimizes the makespan of the solution. One could also trace the traversals of agents and observe that they do not collide neither on a vertex nor on an edge in the graph.

Here, the tasks assignment for agents are $T_1 = \{1\}$ and $T_2 = \{2\}$. The paths of agents, p_1 and p_2 are given in the figure. There are many traversals that would make this a solution for the given instance, the most obvious of these would be traversals that are the same length as the given paths, i.e agents do not wait at any point in their path.

4. Solving MAPDC using ASP

We are mainly concerned with solving the **MAPDC** problem using ASP. This amounts to reducing **MAPDC** to finding an answer set to an ASP program. In this section we provide two ways to solve the problem: one single-shot and one multi-shot method.

4.1 MAPDC-ASP: A single-shot ASP solution for MAPDC

MAPDC as defined in 3.1 can be solved by a single-shot ASP encoding. We call this encoding $\mathcal{P}_S$. We will present the problem instance as facts, generate possible traversals for agents and subject them to **MAPF** constraints, generate possible task assignments and add more constraints so that every agents performs its task.

Given a **MAPDC** instance $I = (G = (V,E), O, A, init, goal, T, t, c)$, F^I is the set of facts representing this instance. It is the union of following facts:

`agent(i).` for $i \in A$

`node(v).` for $v \in V$

`edge(v,k).` $(v,k) \in E$

`obs(v).` for $v \in O$

`init(i,v).` for $i \in A$, $init(i) = v$

`goal(i,v).` for $i \in A$, $goal(i) = v$

`task(i,p,d).` for $(i,p,d) \in T$

`capacity(c).`

`time(0..t).`

The input graph G of the problem is described by predicates `node/1` and `edge/2`; the obstacles, agents, and tasks are described by the predicates `obs/1, agent/1, task/3`, respectively; and the initial and goal locations of agents are described by predicates `init/2` and `goal/2`.

The traversal of a path, and the bag of an agent are described by predicates `traversal/3` and `carry/3`. Here, `traversal(I,T,N)` expresses that the agent `I` is at the location `N` at time step `T`, whereas `carry(I,T,ID)` expresses that the agent `I` carries the product specified by the task `ID` at time step `T`.

We now present the rules that form $\mathcal{P}_S$.

Firstly, we recursively generate possible traversals for our agents with the first two rules. The last rule is a constraint that ensures that an agent cannot be at two different places at the same time-step. This is redundant since the rule that generates the traversals already guarantees that, but this redundancy helps with solving time immensely.

```
traversal(A,0,S):- agent(A), init(A,S).
1{traversal(A, I, K); traversal(A, I, V): edge(K,V)}1:-
        traversal(A, I - 1, K), time(I).
:- traversal(A,I,X), traversal(A,I,Y), agent(A), node(X), node(Y), Y < X.
```

Following rules enforce **MAPF** constraints. First one states that agents cannot be traversing nodes that have obstacles on them. Second and third rules enforce vertex and swapping constraints respectively.

```
:- traversal(A,T,V), obs(V).
:- traversal(A,I,N), traversal(B,I,N), time(I), node(N),
        agent(A), agent(B), A < B.
:- traversal(A,I,X), traversal(A,I + 1,Y), traversal(B,I,Y),
        traversal(B,I + 1,X), agent(A), agent(B), edge(X,Y), A < B.
```

We then introduce the rules for the tasks, their completion and the capacity constraint. First two rules state that an agent can choose to start or finish its task if it is in the correct location. Note that an agent can pass through a pick or deliver location of one of its tasks, since it may already be carrying at its maximum capacity. Next three rules ensure that an agent starts its task before finishing the task, and ensure that an agent can only start and finish a task once. Next two rules first define the `carry` atoms that represent the bag of the agent, and ensure that the capacity is not violated.

```
{taskStart(A,T,S)}1:- time(S), agent(A), task(T,P,D),
        assignment(A,T), traversal(A,S,P).
{taskFinish(A,T,F)}1:- time(F), agent(A), task(T,P,D),
        assignment(A,T), traversal(A,F,D).
:- taskStart(A,T,S), taskFinish(A,T,F), S >= F.
:- {taskStart(A,T,S):time(S)} != 1, agent(A), assignment(A,T).
:- {taskFinish(A,T,F):time(F)} != 1, agent(A), assignment(A,T).
```

```
carry(A,I,T):- taskStart(A,T,S), taskFinish(A,T,F), time(I),
        I <= F, I >= S.
:- time(I), agent(A), {carry(A,I,T): task(T,P,D)} > c.
```

With the following rules, we ensure that agents end up at their goal locations at the last time step. We further define the time-steps that agents finish their traversals with the predicates `finish/2`. This would be the first time step that agent arrives at its goal location and does not move for the rest of the time steps. We then define the maximum time step that an agent finishes its traversal, and proceed to minimize it. This way we can give a maximum makespan t to the program and optimize for the makespan.

```
:- agent(A), goal(A,G), not traversal(A, t, G).
finish(A,I):- agent(A), traversal(A,I-1,V), traversal(A,I,G),
        goal(A,G), V != G, {traversal(A,K,V2): node(V2),
        time(K), K>I, V2 != G } = 0.
lastFinish(I):- #max{K : finish(A,K)} = I.
#minimize{I@1,I: lastFinish(I)}.
```

This encoding solves the **MAPDC** problem, but as will be apparent from the results of our experiments, the size of instances that this encoding can solve is fairly limited. Also, since we give a maximum makespan and ground the program for all the time steps until the maximum makespan, we unnecessarily pay a significant price in terms of grounding. We adress this problem with our next encoding, which is a multi-shot encoding that allows us to ground for each time step separately until we come to a time step that enables a solution to exist for the given instance. We not only save up on grounding time this way, but also have a complete method to find the minimum makespan for which a solution exists for a given instance.

4.2 MAPDC-M: A multi-shot ASP solution for MAPDC

We present a multi-shot formulation of **MAPDC** in ASP, which we call $\mathcal{P}_M$.

Multi-shot solving with ASP requires us to use a control mechanism. Instead of having a single program trying to solve it directly, we have sub-programs in multi-shot solving. The order in which we ground these sub-programs, as well as when to try to solve the grounded part, will be specified in this control mechanism.

Gebser et al. (2019) gives a template control mechanism for problems similar to **MAPDC** where we can iteratively expand our horizon, the maximum makespan t.

We ground the sub-programs for increasing values of t, try to solve the grounded portion, and continue increasing t, and grounding more sub-programs, if we cannot find a solution.

For a **MAPDC** instance I, the set of facts corresponding to the instance F^I is formed the same way as the single-shot solution, the only difference is that we do not give t as part of the instance and ground new `time(t)` atoms as we increment over t as will be shown.

The ASP program $\mathcal{P}_M$ consists of three sub-programs: `base, step, check`. Note that Listing 4.1, as taken and slightly modified from Gebser et al. (2019) should be included at the beginning of the formulation. This is the control unit that determines which sub-programs should be grounded with which parameters, and when the solver should attempt to solve the program.

Listing 4.1 Control code for multi-shot solving

```
#script (python)

from clingo import Number, String, Function

def get(val, default):
    return val if val != None else default

def main(prg):
    imin   = get(prg.get_const("imin"), Number(0))
    imax   = prg.get_const("imax")
    istop  = get(prg.get_const("istop"), String("SAT"))
    max_makespan = 500
    step, ret = 0, None
    while ((step < max_makespan) and
           (step == 0 or step < imin.number or (
              (istop.string == "SAT"     and not ret.satisfiable) or
              (istop.string == "UNSAT"   and not ret.unsatisfiable) or
              (istop.string == "UNKNOWN" and //
                    not ret.unknown)))):
        parts = []
        parts.append(("check", [Number(step)]))
        if step > 0:
            prg.release_external(Function("query", //
                    [Number(step-1)]))

            parts.append(("step", [Number(step)]))
            for i in range(step):
                print("t:␣", i, "tmax:␣", step)
                prg.ground([("generation", //
                [Number(i),Number(step)])])
        else:
            parts.append(("base", []))
        prg.ground(parts)
        prg.assign_external(Function("query", //
                [Number(step)]), True)
        print("t:␣", step)
        ret, step = prg.solve(), step+1
#end.
```

The `base` program is grounded only once. It consists of the following rules expressing that the traversals start at the initial locations of agents, and each task is assigned to one agent. Furthermore, when we are solving an instance I, we add the corresponding facts F^I to the base program.

```
traversal(I,0,S):- agent(I), init(I,S).
1{assignment(I,ID): agent(I)}1 :- task(ID,P,D).
```

The `step(t)` program is grounded incrementally for `t=1,2,3,...`. For each agent, `traversal/3` atoms representing the possible nodes an agent can at time step t are generated recursively. The next rule ensures that the agent cannot be at two different locations, which is again a redundant constraint that helps with performance as in $\mathcal{P}_S$.

```
1{traversal(I,t,X); traversal(I,t,Y): edge(X,Y)}1 :- traversal(I,t-1,X).
:- traversal(I,t,X), traversal(I,t,Y), agent(I), node(X), node(Y), Y<X.
```

Then the following constraints are used to enforce vertex and swapping constraints, and that agents do not collide with obstacles. Recall that we call these constraints the *collision constraints* or **MAPF** constraints.

```
:- traversal(I,t,X), traversal(J,t,X), node(X), agent(I), agent(J), I<J.
:- traversal(I,t-1,X), traversal(I,t,Y), traversal(J,t-1,Y),
   traversal(J,t,X), agent(I), agent(J), edge(X,Y), I<J.
:- traversal(I,t,X), obs(X).
```

For the scheduling of the tasks, we use the predicates `taskStart/3` and `taskFinish/3`. An agent can start a task if the agent is at the picking location of the task. An agent can finish a task if the agent is at the delivery location of the task, provided that the task is already started at a previous time-step.

```
{taskStart(I,ID,t)}1 :- agent(I), task(ID,P,D), assignment(I,ID),
  traversal(I,t,P).
{taskFinish(I,ID,t)}1 :- agent(I), task(ID,P,D), assignment(I,ID),
  traversal(I,t,D), taskStart(I,ID,T), time(T), T<t.
```

Following rules enforce the capacity constraint. Agents start carrying the products at the scheduled start times and continue carrying them until the scheduled finish times. Agents cannot carry more than their capacities.

```
carry(I,t,ID) :- taskStart(I,ID,t).
carry(I,t,ID) :- carry(I,t-1,ID), not taskFinish(I,ID,t).
:- agent(I), {carry(I,t,ID): task(ID,P,D)} > c.
```

The `check(t)` program is grounded for each value of `t` until a solution is computed. `query/1` atoms are *external atoms*, meaning their truth values are set in the control unit. They are set to true only for the last time step. This way, the rules that are already grounded for the previous values of t are ignored by the solver. With this sub-program, we ensure that every task is finished , and that agents should end up at their destinations by the last time step.

```
:- {taskFinish(I,ID,T):time(T)} != 1, agent(I), assignment(I,ID), query(t).
:- agent(I), goal(I,X), not traversal(I,t,X), query(t).
```

5. HMAPDC: A Hierarchical ASP Method for MAPDC

Due to the hardness of **MAPDC**, solving time increases rapidly as the size of the input graph grows. This is both due to the increased number of possible paths for agents and a general increase in makespan. We are interested in ways to heuristically improve solving time performance while accepting some sub-optimality on makespan, and incompleteness.

5.1 Overall Idea

One way to achieve this is approaching the problem hierarchically. One can obtain abstracted graphs which have smaller dimensions and solve the problem on these abstracted graphs. The solutions then can be passed to the next hierarchical levels and restrict the paths of agents accordingly. By restricting agents to certain regions, we can decrease the search space of their paths. This way, we refine the paths of agents with each successive level.

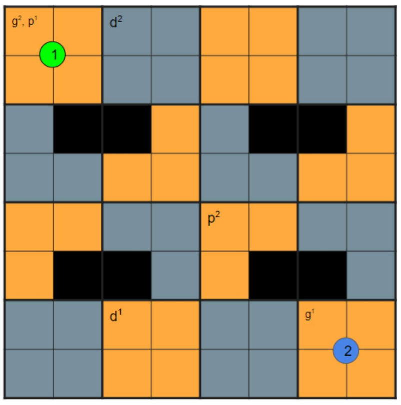
Since we are abstracting the graph, the solutions in higher levels will not be synchronized with respect to time steps. Thus, we waive the collision constraints of **MAPF** for all the hierarchical levels except the lowest one, which takes as input the original graph to the **MAPDC** problem and will yield the actual collision-free solution we are ultimately looking for.

The solutions then can be passed to the next hierarchical levels and restrict the paths of agents accordingly. By restricting agents to certain regions, we can decrease the search space of their paths. This way, we refine the paths of agents with each successive level.

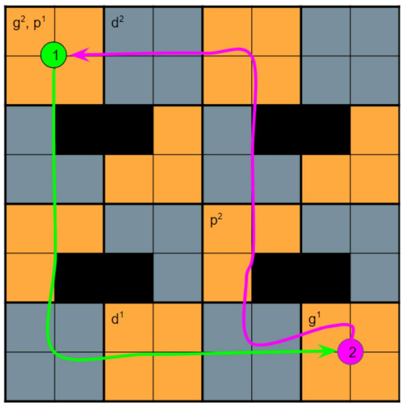
Consider Figure 5.1 where we apply this method to the example given in Figure 3.2. We have two levels. The upper picture in the figure depicts the top abstract level, where the original 8x8 is abstracted into a 4x4 grid with 2x2 regions implied by alternating colors. Instead of moving through nodes in the original graph, agents will move through the regions such that they still complete their tasks, but in an abstract manner. Instead of visiting the exact pick and delivery locations, they just need to visit the regions these are in. Then they move to the region where their goal locations fall into.

Then, in the lower picture, we have the lowest level where we need to find an exact solution to the problem instance in Figure 3.2. We restrict the agents to the nodes that fall into the regions agents move through in the upper level, depicted in the upper picture of Figure 5.1. For agent 1, that includes blue and green nodes; for agent 2, that includes blue and magenta nodes. Thus, for each agent, we save time by only searching through possible paths in the restricted area instead of the whole graph.

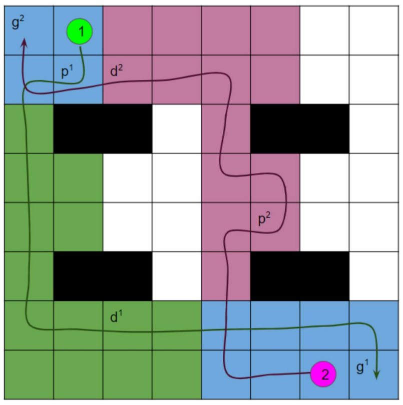
Figure 5.1 Hierarchical representation of the example in Figure 3.2 and its solution



(a) At Level 0, the grid G in Figure 3.2 is viewed as a 4x4 grid G_0. Each region is colored orange or gray, and corresponds to a 4x4 subgrid of G. Agent 1 starts from the upper-left-most region and needs to move to the bottom-right-most one. Agent 2 starts from the bottom-right-most region and needs to move to the upper-left-most one. There are 2 tasks in the environment, their pick and delivery locations are denoted on the upper-left corner of regions. For example, the pick location of the first task (denoted p^1) is somewhere in the upper-left-most region.



(b) A solution for Level 0, where $T_1 = \{1\}$ and $T_2 = \{2\}$. In the solution, the paths of agents are drawn in their respective colors. They complete their tasks by moving through the regions the pick and delivery locations are in.



(c) At Level 1, we work on the original grid G in Figure 3.2. Here, instead of considering all possible solutions to the **MAPDC** instance, we look for solutions that restrict the paths of agents to the nodes that fall into regions that they traverse in the solution given in Figure 5.1b for Level 0. Agent 2 can only move through nodes colored dark pink and blue, while agent 1 can only move through nodes colored dark green and blue. The tasks are assigned from scratch, but this time the assignments are the same as in Figure 5.1b

.

5.2 Revising Definitions for Abstract Levels

The definitions we have made for **MAPDC** also mostly hold for the sub-problems utilized in the **HMAPDC** algorithm. We proceed to define the inputs of **HMAPDC**.

For a non-negative integer l, the lth hierarchical level, or Level l, for a **MAPDC** instance *I=(G=(V,E),O,A,init,goal,T,c,t)* is characterized by a six-tuple $(G_l, init_l, goal_l, pick_l, deliver_l, m_l)$.

- $G_l = (V_l, E_l)$ is the abstract graph of the level. Intuitively, as illustrated in Figures 5.11 and 5.12, the larger (and upper) the Level l, coarser the abstract graph. We can refer to the vertices in V_l as *regions* or simply *vertices of Level l*. Intuitively, these contain a subset of the vertices of G.
- $init_l$ and $goal_l$ are functions $A \mapsto V_l$ that specify the initial and goal locations projected to level l.
- $pick_l$ and $deliver_l$ are functions $T \mapsto V_l$ that specify the pick and deliver locations of the tasks projected to level l.
- and m_l is a many to one function $V_l \mapsto V_{l-1}$ which we call a *level-mapping*, that describes this projection. We read $m_l(v_1) = v_2$ as "vertex v_1 at level l is mapped to vertex v_2 at the next finer level $l-1$". Here are the properties of a level-mapping:
  - vertices that map to the same vertex in the upper level constitute a connected graph: $G_c = (V_c, E_c)$ is fully connected where $V_c \subseteq V_l, |\{v \in V_{l-1} | m_l(v_c) = v, v_c \in V_c\}| = 1$ and $E_c = \{(v_1,v_2) | (v_1,v_2) \in E_l, v_1, v_2 \in V_c\}$,
  - vertices on the same level l has an edge between them iff there exists an edge between vertices that maps to them on the previous level $l+1$: $(v_1,v_2) \in E_l \iff \exists (v_3,v_4) \in E_{l+1}, m_{l+1}(v_3) = v_1 \wedge m_{l+1}(v_4) = v2$,
  - $init_l$ and $goal_l$ are mapped from the corresponding functions $init_{l+1}$ and $goal_{l+1}$: for $i \in A$, $m_{l+1}(init_{l+1}(i)) = init_l$ and $m_{l+1}(goal_{l+1}(i)) = goal_l$,
  - $pick_l$ and $deliver_l$ are mapped the same as above: for $r \in T$, $m_{l+1}(pick_{l+1}(i)) = pick_l$ and $m_{l+1}(deliver_{l+1}(i)) = deliver_l$
  - for the last level $l = numlevels - 1$, all graph dependent entities in the level are the same as in I: $G_{numlevels-1} = G$, $init_{numlevels-1} = init$, $goal_{numlevels-1} = goal$, and for task $r = (id, p, d)$, $pick_{numlevels-1}(r) = p$,

$deliver_{numlevels-1}(r) = d.$

Level 0 characterization of I is the most abstract view of the given **MAPDC** problem. As levels increase, we get a less abstract view of the given **MAPDC** problem. For a hierarchy with a number of levels, $numlevels > 0$, the last level $l = numlevels - 1$ corresponds to the given **MAPDC** instance I.

For an agent i, we denote its traversal at Level l by $traversal_{l,i}$. Note that this traversal need not be collision-free for any Level l, $l < numlevels - 1$. Consider any region in Figure 5.1a, all the regions have more than one node in them, meaning more than one agent can be present in them at the same time step. Also, time steps in abstract levels are not synchronized for different agents. Consider Figure 5.1b. Agent 1 moves to the region below in one time step, and Agent 2 moves to the region to its left in one time step. But also consider Figure 5.1c, Agent 1 takes 3 time steps to move to a node that belongs to the region mentioned, while Agent 2 takes just one time step. Because of these reasons, collisions are not defined for abstract levels.

The definition of completing a task for this hierarchical version of the problem does not fundamentally differ from our previous definition, but we waive the assumption that the pick and deliver location of a task will not be the same. This is because the pick and deliver locations of a task can fall into the same region in an abstracted graph. This also means a task can be both started and finished at the same time step in an abstract level.

Let us denote the bag description of an agent i as $carry_{l,l}$ at Level l. For Level l, functions $pick_l : T \mapsto V_l$ and $deliver_l : T \mapsto V_l$ describe the pick and deliver locations of the tasks at level l. A task r is completed by an agent i within time $u_{l,i}$ at Level l if there exist time steps x and y, $0 \leq x \leq y \leq u_{l,i}$ such that $traversal_{l,i}(x) = pick_l(r)$, $traversal_{l,i}(y) = deliver_l(r)$ and $r \in carry_{l,i}(k)$ only for all time steps $x \leq k \leq y$.

5.3 Input and Output of HMAPDC

In this study, we consider *grid graphs*. We represent them as matrices: the cells in a matrix are the vertices and only adjacent cells can have edges between them. This is because we can algorithmically generate hierarchical levels easily for graphs with such structure. We detail how we obtain the hierarchical levels later in this section.

HMAPDC takes as input a **MAPDC** instance $I = (G, O, A, init, goal, T, c, t)$ and a sequence L of *numlevels*, hierarchical levels L=$[(G_0, init_0, goal_0, pick_0, deliver_0, m_0), \ldots, (G_{numlevels-1}, init_{numlevels-1}, goal_{numlevels-1}, pick_{numlevels-1}, deliver_{numlevels-1}, m_{numlevels-1})]$, and finds a solution to I.

Consider Figure 5.1 where we apply **HMAPDC** on the example from Figure 3.2. The Figure 5.1a denotes Level 0. The graph is abstracted into a 4x4 grid. Each vertex in this abstracted 4x4 grid graph is a region. If there is an edge in the original graph that connects a node from a region r_1 to another node in region r_2, then the regions r_1 and r_2 are connected by an edge in this abstract level. In Figure 5.1a, all the regions have an edge to their neighbours.

There are three different kinds of sub-problems for solving **MAPDC** via the **HMAPDC** algorithm. **MAPDC-Top**, **MAPDC-Middle** and **MAPDC-Bottom**. **MAPDC-Top** is the highest level, which is $l = 0$. **MAPDC-Bottom** is the lowest level with $l = numlevels - 1$. We have one instance each for **MAPDC-Top** and **MAPDC-Bottom** for a given **MAPDC** instance, while there are $numlevels - 2$ instances for **MAPDC-Middle**. We start solving from top to bottom.

For all levels other than the first one, where we solve **MAPDC-Top**, we need to specify for each agent which vertices it can move through at that level. For level l, we do this via the function $valids_l : A \mapsto 2^{V_l}$, which takes as parameter an agent and returns the set of vertices that agent can move through. We obtain this function from the solution of the previously solved level. Specifically, for agent $i \in A$, $valids_l(i) = \{v \in V_l |$ for some $x \leq t, traversal_{l-1,i}(x) = m_l(v)\}$.

We proceed to define these three sub-problems. Figures 5.2, 5.3, and 5.4 show the definitions of **MAPDC-Top**, **MAPDC-Middle**, and **MAPDC-Bottom** respectively.

MAPDC-Top

Input:

- A graph $G = (V, E)$
- A set A of agents.
- A function $\mathit{init} : A \mapsto V$
- A function $\mathit{goal} : A \mapsto V$
- A set T of tasks
- A function $\mathit{pick} : T \mapsto V_0$
- A function $\mathit{deliver} : T \mapsto V$
- A positive integer t (to specify the maximum makespan).
- A positive integer c (to specify the capacity of each agent).

Output: For every agent $i \in A$, for some positive integer $u_i \leq t$,

- a set T_i of tasks allocated to the agent i where $\bigcup_{j \in A} T_j = T$ and
  - for every $j \in A$, $i \neq j$ implies $T_i \cap T_j = \emptyset$;
- a path $P_i = \langle w_{i,1}, \ldots, w_{i,n_i} \rangle$ of length n_i ($n_{0,i} \leq u_i$) that the agent i can traverse
  - to reach its goal location $w_{i,n_i} = \mathit{goal}(i)$ from its initial location $w_{i,1} = \mathit{init}(i)$,
  - to complete the allocated tasks T_i (i.e., for every $r \in T_i$, there exists $w_{i,j}, w_{i,k} \in P_i$ where $j \leq k$, $w_{i,j} = \mathit{pick}(r)$ and $w_{i,k} = \mathit{deliver}(r)$; and
- a traversal $\mathit{traversal}_i$ of the path P_i within time u_i and how the bag of the agent i changes during this traversal (i.e., carry_i) such that
  - every task in T_i is completed by the agent i with respect to its traversal, and
  - for every $x \leq u_i$, the agent i can carry as many tasks as its capacity: $|\mathit{carry}_i(x)| < c$.

Figure 5.2 **MAPDC-Top** problem definition

MAPDC-Middle

Input:

- A graph $G = (V, E)$
- A set A of agents.
- A function $\mathit{init} : A \mapsto V$
- A function $\mathit{goal} : A \mapsto V$
- A function $\mathit{pick} : T \mapsto V$
- A function $\mathit{deliver} : T \mapsto V$
- A function $\mathit{valids} : A \mapsto 2^V$ (to specify which vertices an agent is allowed to traverse)
- A positive integer t (to specify the maximum makespan).
- A positive integer c (to specify the capacity of each agent).

Output: For every agent $i \in A$, for some positive integer $u_i \le t$,

- a set T_i of tasks allocated to the agent i where $\bigcup_{j \in A} T_j = T$ and
  - for every $j \in A$, $i \neq j$ implies $T_i \cap T_j = \emptyset$;
- a path $P_i = \langle w_{i,1}, \ldots, w_{i,n_i} \rangle$ of length n_i ($n_i \le u_i$) that the agent i can traverse
  - to reach its goal location $w_{i,n_i} = \mathit{goal}(i)$ from its initial location $w_{i,1} = \mathit{init}(i)$,
  - to complete the allocated tasks T_i; and
- a traversal $\mathit{traversal}_i$ of the path P_i within time u_i and how the bag of the agent i changes during this traversal (i.e., carry_i) such that
  - every task in T_i is completed by the agent i with respect to its traversal, and
  - for every $x \le u_i$, the agent i can carry as many tasks as its capacity: $|\mathit{carry}_i(x)| < c$.
  - the agent only moves through its allowed vertices (i.e., $traversal_i(x) \in \mathit{valids}(i)$, for every $x = 0, \ldots, u_i$)

Figure 5.3 **MAPDC-Middle** problem definition

<u>**MAPDC-Bottom**</u>

Input:

- A graph $G = (V, E)$
- A set A of agents.
- A function $\mathit{init} : A \mapsto V$
- A function $\mathit{goal} : A \mapsto V$
- A function $\mathit{pick} : T \mapsto V$
- A function $\mathit{deliver} : T \mapsto V$
- A function $\mathit{valids} : A \mapsto 2^V$ (to specify which vertices an agent is allowed to traverse)
- A positive integer t (to specify the maximum makespan).
- A positive integer c (to specify the capacity of each agent).

Output: For every agent $i \in A$, for some positive integer $u_i \leq t$,

- a set T_i of tasks allocated to the agent i where $\bigcup_{j \in A} T_j = T$ and
  - for every $j \in A$, $i \neq j$ implies $T_i \cap T_j = \emptyset$;
- a path $P_i = \langle w_{i,1}, \ldots, w_{i,n_i} \rangle$ of length n_i ($n_i \leq u_i$) that the agent i can traverse
  - to reach its goal location $w_{i,n_i} = \mathit{goal}(i)$ from its initial location $w_{i,1} = \mathit{init}(i)$,
  - to complete the allocated tasks T_i; and
- a **collision free** traversal $\mathit{traversal}_i$ of the path P_i within time u_i and how the bag of the agent i changes during this traversal (i.e., carry_i) such that
  - every task in T_i is completed by the agent i with respect to its traversal, and
  - for every $x \leq u_i$, the agent i can carry as many tasks as its capacity: $|carry_i(x)| < c$.
  - the agent only moves through its allowed vertices (ie. $traversal_i(x) \in \mathit{valids}(i)$, for every $x = 0, \ldots, u_i$)

Figure 5.4 **MAPDC-Bottom** problem definition

$\underline{\mathcal{P}_{top}}$

```
#program base.
traversal(I,0,S):- agent(I), init(I,S).
1{assignment(I,ID): agent(I)}1 :- task(ID,P,D).

#program step(t).
1{traversal(I,t,X); traversal(I,t,Y): edge(X,Y)}1 :-
traversal(I,t-1,X).
:- traversal(I,t,X), traversal(I,t,Y), agent(I), node(X), node(Y), Y<X.

{taskStart(I,ID,t)}1 :- agent(I), task(ID,P,D), assignment(I,ID),
  traversal(I,t,P).
{taskFinish(I,ID,t)}1 :- agent(I), task(ID,P,D), assignment(I,ID),
  traversal(I,t,D), taskStart(I,ID,T), time(T), T<t.

carry(I,t,ID) :- taskStart(I,ID,t).
carry(I,t,ID) :- carry(I,t-1,ID), not taskFinish(I,ID,t).
:- agent(I), {carry(I,t,ID): task(ID,P,D)} > c.

#program check(t).
:- {taskFinish(I,ID,T):time(T)} != 1, agent(I), assignment(I,ID), query(t).
:- agent(I), goal(I,X), not traversal(I,t,X), query(t).
```

Figure 5.5 ASP program for **MAPDC-Top**: $\mathcal{P}_{top}$

$\underline{\mathcal{P}_{mid}}$

```
#program base.
traversal(I,0,S):- agent(I), init(I,S).
1{assignment(I,ID): agent(I)}1 :- task(ID,P,D).

#program step(t).
1{traversal(A, t, K); traversal(A, t, V): edge(K,V), valid(A,V)}1:-
    traversal(A, t - 1, K), agent(A).
:- traversal(A,t,K), not valid(A,K).
:- traversal(I,t,X), traversal(I,t,Y), agent(I), node(X), node(Y), Y<X.

{taskStart(I,ID,t)}1 :- agent(I), task(ID,P,D), assignment(I,ID),
  traversal(I,t,P).
{taskFinish(I,ID,t)}1 :- agent(I), task(ID,P,D), assignment(I,ID),
  traversal(I,t,D), taskStart(I,ID,T), time(T), T<t.
carry(I,t,ID) :- taskStart(I,ID,t).
carry(I,t,ID) :- carry(I,t-1,ID), not taskFinish(I,ID,t).
:- agent(I), {carry(I,t,ID): task(ID,P,D)} > c.

#program check(t).
:- {taskFinish(I,ID,T):time(T)} != 1, agent(I), assignment(I,ID), query(t).
:- agent(I), goal(I,X), not traversal(I,t,X), query(t).
```

Figure 5.6 ASP program for **MAPDC-Middle**: $\mathcal{P}_{mid}$

$\underline{\mathcal{P}_{bot}}$

```
#program base.
traversal(I,0,S):- agent(I), init(I,S).
1{assignment(I,ID): agent(I)}1 :- task(ID,P,D).

#program step(t).
1{traversal(A, t, K); traversal(A, t, V): edge(K,V), valid(A,V)}1:-
    traversal(A, t - 1, K), agent(A).
:- traversal(A,t,K), not valid(A,K).
:- traversal(I,t,X), traversal(I,t,Y), agent(I), node(X), node(Y), Y<X.

{taskStart(I,ID,t)}1 :- agent(I), task(ID,P,D), assignment(I,ID),
  traversal(I,t,P).
{taskFinish(I,ID,t)}1 :- agent(I), task(ID,P,D), assignment(I,ID),
  traversal(I,t,D), taskStart(I,ID,T), time(T), T<t.
carry(I,t,ID) :- taskStart(I,ID,t).
carry(I,t,ID) :- carry(I,t-1,ID), not taskFinish(I,ID,t).
:- agent(I), {carry(I,t,ID): task(ID,P,D)} > c.

:- traversal(I,t,X), traversal(J,t,X), node(X), agent(I), agent(J), I<J.
:- traversal(I,t-1,X), traversal(I,t,Y), traversal(J,t-1,Y),
   traversal(J,t,X), agent(I), agent(J), edge(X,Y), I<J.
:- traversal(I,t,X), obs(X).

#program check(t).
:- {taskFinish(I,ID,T):time(T)} != 1, agent(I), assignment(I,ID), query(t).
:- agent(I), goal(I,X), not traversal(I,t,X), query(t).
```

Figure 5.7 ASP program for **MAPDC-Bottom**: $\mathcal{P}_{bot}$

5.4 ASP Programs for the Sub-problems

ASP programs for all the sub-problems are multi-shot solutions derived from $\mathcal{P}_M$. **MAPDC-Top** waives the collision constraints, thus we omit the rules on collision constraints from $\mathcal{P}_M$ to obtain $\mathcal{P}_{top}$. $\mathcal{P}_{top}$ is given in Figure 5.5.

The sub-problem **MAPDC-Middle** restricts agent movement to the regions implied by previous solutions. Thus, we utilize atoms to indicate where an agent can move to. `valid(a,v)` means that agent a can use vertex v. We obtain $\mathcal{P}_{mid}$ from $\mathcal{P}_{top}$, since both waive the collision constraints. To do that, we modify the choice rule that recursively generates the traversals of agents, and add a redundant constraint to the `step(t)` subprogram to make sure agents do not move into regions that are not allowed. $\mathcal{P}_{mid}$ is given in Figure 5.6

At the last level, **MAPDC-Bottom**, we need to find collision-free traversals, so to obtain $\mathcal{P}_{bot}$, we take $\mathcal{P}_M$ and do the same modifications described for $\mathcal{P}_{mid}$ to the generation part. $\mathcal{P}_{bot}$ is given in Figure 5.7.

5.5 Graph Partitioning

In this study, we consider *grid graphs*. We represent them as matrices: the cells in a matrix are the vertices and only adjacent cells can have edges between them. An example can be seen in Figure 5.8

We are working with grid graphs, so we have a method to partition grids. We give the first level size as input to the partitioning algorithm and divide the grid into regions accordingly. Then, with each level, we halve the dimensions of the regions and obtain new regions. Whenever we create a level in such a manner, we also check for unconnected components in each region. If there are any, we divide these regions into their connected components.

It should be noted that other kinds of partitioning methods can be used. In real-world applications such as warehouse automation, these partitions can be engineered for efficiency.

In Figure 5.1, we have chosen the first level size as 4x4. We can only divide the regions at Level 0 once, so we do not have any mid-levels.

Here, we provide the algorithms we use to create the hierarchical levels. Although the algorithm is not sophisticated, it may still be hard to follow. There are some functions used that are not explicitly stated, we will describe them here.

Firstly, we use the *slicing operator* $[startindex : stopindex]$. This is used to slice a list/sequence, meaning $list[0 : 10]$ is a list that has the first ten elements of $list$, element at the 10th index is excluded. This slicing operator can be used for matrices as well. $M[startrow : stoprow][startcol : stopcol]$ returns a slice of matrix M with respect to the parameters. $M[0 : 10][5 : 10]$ would return the first ten rows of M and these rows would be sliced between 5th and 10th index.

We use this operator on matrices with cells that have binary values. Whenever one of the indices for the operator is out of bounds, we fill the out of bound values with 1. If we do the operation $M[0 : 10][0 : 10]$ when M is 5x5, the operation still returns a 10x10 matrix with 1 padded for the out of bound values.

We use *regions* as data structures that store values that are needed for the graph partioning algorithm. A region r defined over a grid graph G has the following data:

- *r.parent*: Each region has a single parent region, or has no parents. Regions at level 0 do not have parents, while in the consecutive levels all regions have a parent from the upper level.

- $r.ul$: ul stands for upper-left. It is the upper-left-most coordinate of the region and lets us keep track of where the region corresponds to in a grid graph.
- $r.mm$: mm stands for *membership matrix*. It is the rectangular slice of the original graph starting from the coordinates $r.ul$. As in grid graphs, a value of 0 in this matrix means the node is in region r, while a value of 1 indicates the node is not part of the region or is an obstacle. All mms in the same level are the same size, but some of them may have some part of them filled with 1s. For example, when there are unconnected parts in r, we create new regions that correspond to the connected components. These new regions have the same ul as r, but their membership matrices differ.
- $r.id$: This is the id of the region. In hierarchical levels, these are the node ids for the abstracted graphs.
- $r.vertices$: This is the list of vertices in the region. These vertices are the ones in V_G, the vertices on the original graph this region is defined on.

When we want to create a new region, we use the function $newRegion(parent, ul, mm, id)$, and we use the function $findVertices(G, r)$ to find the vertices of region r in G. $r.vertices$ is initialized as $findVertices(G, r)$ after the regions are determined.

Function $connecteds(r)$ takes a region as input a region and returns components, as in connected components. Here, *components* are a list of membership matrices the same size as $r.mm$, and only the vertices that are in the same component have value 0 for each component. We believe finding connected components are fundamental enough that we do not detail its implementation. In our implementation, we use the *scipy* Python library (Virtanen et al., 2020) to find them. One example of dividing a region to its connected components can be seen in Figure 5.9b. In Figure 5.9c, the membership matrices returned for the newly created regions can be seen.

We create all the hierarchical levels we feed into the **HMAPDC** algorithm via function *createLevels*, which is described in Algorithm 1. It takes as input a **MAPDC** instance $I = (G = (V, E), O, A, init, goal, T, t, c)$ and a first level size (X, Y). X is the number of rows and Y is the number of columns we desire in our top level. Algorithm 1 first initializes the $pick : T \mapsto V$ and $deliver : T \mapsto V$ functions of the given tasks T. Then, it utilizes the function $createFirstLevel$(described in Algorithm 2) and calls the function $adoptToLevel$ (described in Algorithm 4) to create the first hierarchical level. Then it sets the mapping m_l of the first level $l = 0$ to empty set, since the top level does not need a mapping. Then it appends the first level into level list ls. It proceeds to create new hierarchical levels until the regions have size 1x1.

It does this in a while loop that utilizes the function *createNewLevel* (described in Algorithm 3) and the same *adoptToLevel* function. Still in the loop, it sets the mapping m_l of the current level it is working on such that every region in level l maps to its parent region in the upper level $l-1$.

The function *adoptToLevel* creates the hierarchical level components $init_l, goal_l, pick_l$ and $deliver_l$. It does this by taking as input the *regions* of level l, agents of the instance A, their initial and goal location functions *init* and *goal*, the set of tasks T and their *pick* and *deliver* functions, and finding out which region they fall into in the level.

Algorithm 1: createLevels: An algorithm to create all hierarchical levels for **HMAPDC**

```
Input: MAPDC instance (G,O,A,init,goal,T,t,c) , a first level size $(X,Y)$
Output: a sequence of hierarchical levels
1  pick(r) ← p for $r \in T$;
2  deliver(r) ← d for $r \in T$;
3  ls ← [];
4  regionctr ← 0;
   /* Call CreateFirstLevel to create the grid $G_0$ for the first level, and the
      regions in it, then call adoptToLevel to get the other components that
      characterize Level 0.                                                    */
5  $G_0$, regions, regionctr ← CreateFirstLevel($G, (X, Y)$);
6  $init_0, goal_0, pick_0, deliver_0$ ← adoptToLevel(regions,A,init,goal,T,pick,deliver);
7  $m_0 \leftarrow \emptyset$;
8  ls.append(($G_0, init_0, goal_0, pick_0, deliver_0, m_0$));
   /* Create new levels with CreateNewLevel and adoptToLevel using the same
      process above, until the membership matrices of the regions become size
      1x1.                                                                     */
9  while rows(regions[0].mm) != 1 and cols(regions[0].mm != 1) do
10     $G_l$, regions, regionctr ← CreateNewLevel(G,regions,regionctr);
11     $init_l, goal_l, pick_l, deliver_l$ ←
         adoptToLevel(regions,A,init,goal,T,pick,deliver);
12     $m_l(r)$ ← r.parent for $r \in regions$;
13     ls.append(($G_l, init_l, goal_l, pick_l, deliver_l, m_l$));
14 return ls;
```

Algorithm 2: CreateFirstLevel: An algorithm to create the abstract grid graph at the top level, Level 0

```
Input: a grid graph $G$, and a first level size $(X,Y)$
Output: abstracted graph for the first level, its regions, and the region counter
regionctr ← 0;
regionsize ← ($\lceil rows(G)/X \rceil$, $\lceil cols(G)/Y \rceil$);
regions ← [];
// Divide $G$ into $X*Y$ subgrids, create regions for them
for i in {0,...,X-1} do
    for j in {0, ..., Y-1} do
        ul ← (i * regionsize[0], j * regionsize[1]);
        mm ← G[ul[0] : ul[0] + regionsize[0]][ul[1] : ul[1] + regionsize[1]];
        r ← newRegion(∅,ul, mm, regionctr);
        regionctr ← regionctr + 1;
        regions.append(r);
/* For every region, first get the connected components (cs) of their membership matrices (mm).
   If there are more than one, create a new region for each and remove the original.  Remove
   empty regions.                                                                          */
for r in regions do
    cs = connecteds(r);
    if |cs| > 1 then
        for c in cs do
            rn ← newRegion(r.parent, r.ul, c, regionctr);
            regionctr ← regionctr + 1; regions.append(rn);
        regions.remove(r);
    else if |cs|== 0 then
        regions.remove(r);
/* Iterate over all pairs of regions, find which vertices of $G$ fall into them, if there is an
   edge in $G$ between the the vertices of regions, add the pair of region ids to first level
   edges $e$                                                                               */
e ← [];
for $r_1$ in regions do
    $r_1$.vertices ← findVertices(G,$r_1$);
    for $r_2$ in regions do
        $vertices_1 \leftarrow r_1.vertices()$;
        $vertices_2 \leftarrow r_2.vertices()$;
        if $\exists v_1 \in vertices_1, v_2 \in vertices_2$ such that $(v_1, v_2) \in g$ then
            e.append(($r_1.id, r_2.id$));
rs ← {r.id for $r \in regions$};
return (rs,e), regions, regionctr;
```

Algorithm 3: CreateNewLevel: An algorithm to create a new hierarchical level graph for **HMAPDC**

Input: a grid graph G, regions rs of the previous level, a region counter rgctr

Output: graph, regions of the new level and the region counter

```
1  regionctr ← rgctr;
2  prevregionsize ← (rows(rs[0].mm), cols(rs[0].mm))
3  regionsize ← ($\lceil prevregionsize[0]/2 \rceil$, $\lceil prevreegionsize[1]/2 \rceil$);
4  regions ← [];
   /* Divide each given region into 4 (2x2) new ones, by dividing their membership matrices into
      subgrids.                                                                                */
5  for r in rs do
6      for i in {0,1} do
7          for j in {0,1} do
8              ul ← (r.ul[0] + i * regionsize[0], r.ul[1] + j * regionsize[1]);
9              mm ← r.mm[ul[0] : ul[0] + regionsize[0]][ul[1] : ul[1] + regionsize[1]];
10             rnew ← newRegion(r.id, ul, mm, regionctr);
11             regionctr ← regionctr + 1;
12             regions.append(rnew);
   /* For every region, first get the connected components (cs) of their membership matrices (mm).
      If there are more than one, create a new region for each and remove the original.       */
13 for r in regions do
14     cs = connecteds(r);
15     if |cs|> 1 then
16         for c in cs do
17             rnew ← newRegion(r.parent, r.ul, c, regionctr);
18             regionctr ← regionctr + 1;
19             regions.append(rnew);
20         regions.remove(r);
21     else if |cs| == 0 then
22         regions.remove(r);
   /* Iterate over all pairs of regions, find which vertices of $G$ fall into them, if there is an
      edge in $G$ between the the vertices of regions, add the pair of region ids to the level
      edges $e$                                                                                */
23 e ← [] ;
24 for $r_1$ in regions do
25     $r_1$.vertices ← findVertices(G,$r_1$);
26     for $r_2$ in regions do
27         $vertices_1 \leftarrow r_1.vertices()$;
28         $vertices_2 \leftarrow r_2.vertices()$;
29         if $\exists v_1 \in vertices_1, v_2 \in vertices_2$ such that $(v_1, v_2) \in E_G$ then
30             e.append(($r_1.id, r_2.id$));
31 rs ← {r.id for $r \in regions$};
32 return (rs,e),regions, regionctr;
```

Algorithm 4: adoptToLevel

Input: *regions*, agents A, *init* and *goal* for A, tasks T, *pick* and *deliver* for T

Output: $init_l, goal_l, pick_l, deliver_l$

```
1  $init_l \leftarrow \emptyset$;
2  $goal_l \leftarrow \emptyset$;
3  $pick_l \leftarrow \emptyset$;
4  $deliver_l \leftarrow \emptyset$;
   /* For every agent, project their initial and goal locations to the current
      level, by finding which region they fall into, by going through the
      vertices that fall into each region.                                    */
5  for $a$ in $A$ do
6      i $\leftarrow init(a)$;
7      g $\leftarrow goal(a)$;
8      for $r$ in $regions$ do
9          if $i$ in $r.vertices$ then
10             $init_l(a) \leftarrow r.id$ ;
11         if $g$ in $r.vertices$ then
12             $goal_l(a) \leftarrow r.id$;
   /* for every task, project their pick and delivery locations to the current
      level, the same way as above.                                           */
13 for $t$ in $T$ do
14     p $\leftarrow pick(t)$;
15     d $\leftarrow deliver(t)$;
16     for $r$ in $regions$ do
17         if $p$ in $r.vertices$ then
18             $pick_l(a) \leftarrow r.id$ ;
19         if $g$ in $r.vertices$ then
20             $deliver_l(a) \leftarrow r.id$;
21 return $init_l$, $goal_l$, $pick_l$, $deliver_l$;
```

Algorithm 5: findVertices

Input: grid graph G, a *region* on this graph

Output: a list of vertices that fall into the *region* with respect to G

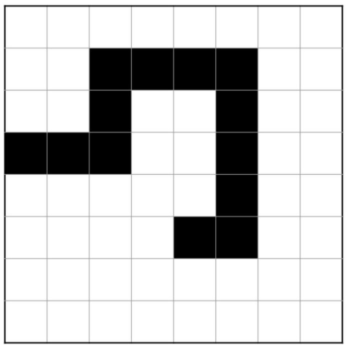
```
1  vs $\leftarrow$ [];
2  ul $\leftarrow$ region.ul;
3  for row in {0, ..., rows(region.mm) - 1} do
4      for col in {0, ..., cols(region.mm) - 1} do
5          if mm[row][col] == 0 then
6              vs.append(toVertex(G,(ul[0] + row, ul[1] + col)));
7  return vs;
```

5.5.1 An example for graph partitioning

An example of our partitioning method can be seen in Figure 5.9. This time, 2x2 is chosen as first level size. The levels are 2,0,1 from top to bottom in the figure respectively. It can be seen that in the first level we divided the regions into their connected components. When moving from Level 0 to 1, we did not further divide the already smaller regions dark blue, brown and dark green. Every level has an expected region size. For level 0 this is 4x4 ($8/2 \times 8/2$) and for Level 1 2x2 ($4/2 \times 4/2$). Since no dimension of the smaller regions is greater than 2, they are of the expected size (or smaller) and are not divided further. The rest of the regions (e.g the red one) is divided into four regions as shown, for $l = 1$. The next level after that is level 2, which is the original graph.

We handle odd grid sizes the same way, only the expected level sizes are up-ceiled whenever necessary. In Figure 5.10, the levels are respectively 2,0,1 as well. The expected level size of level 0 is 3x3 ($\lceil 5/2 \rceil \times \lceil 5/2 \rceil$).

Figure 5.8 An Example grid graph and its matrix representation

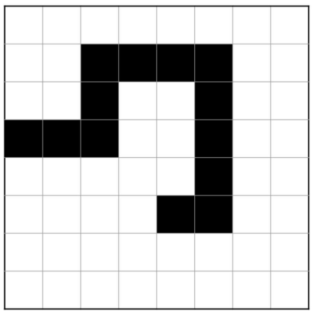


(a) A grid graph G.

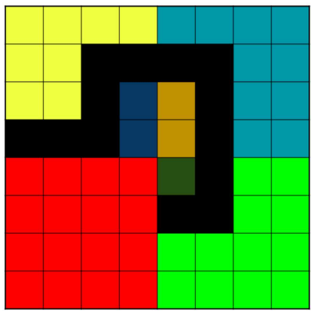
$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

(b) Matrix representation of G.

Figure 5.9 An example of partitioning



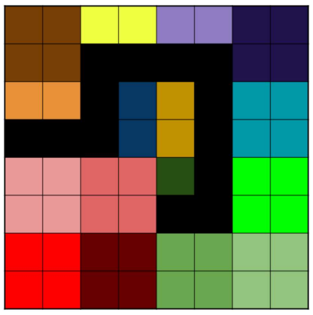
(a) The original grid graph G.



(b) G_0 given by *CreateFirstLevel*$(G,(2x2))$. Regions are colored differently. Obstacles do not belong to any region. Normally, regions in this level are 4x4 grids, but the obstacles required us to divide some regions, e.g., the yellow and dark blue regions r_1 and r_2. r_1 and r_2 still have 4x4 membership matrices. For r_1, only the yellow colored nodes are members in this matrix, i.e., has value 0. After the call for *CreateFirstLevel*, *regions* are the set of regions depicted in G_0, and $rgctr = 9$ as can be traced from Algorithm 2

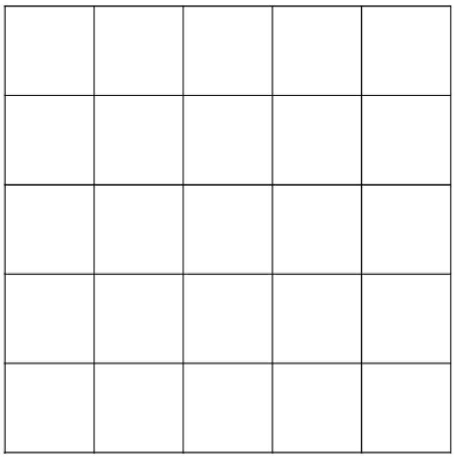
$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

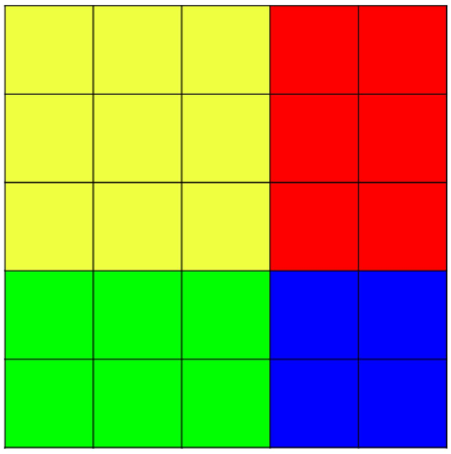
(c) $r_1.mm$ and $r_2.mm$. respectively. Both $r_1.ul$ and $r_2.ul$ are $(0,0)$



(d) G_1 given by $CreateNewLevel(G, regions, rgctr)$

Figure 5.10 An odd example of partitioning


(a) An empty 5x5 grid G.


(b) Level 0 graph G_0 created by calling *CreateFirstLevel(*$G, (2x2)$*)*.

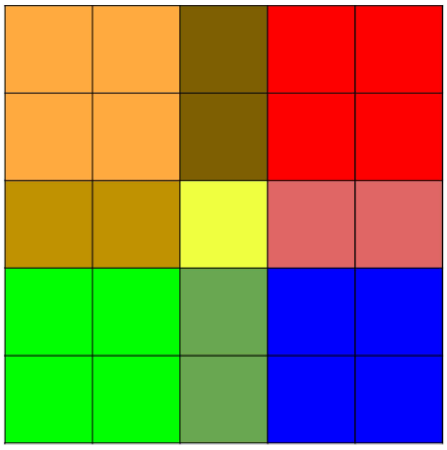
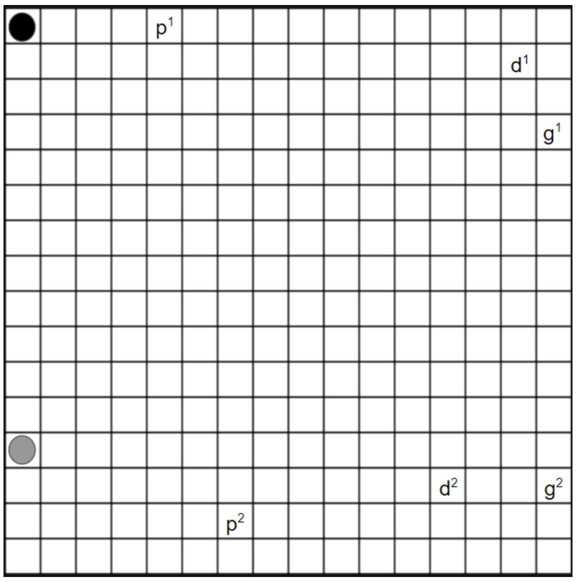
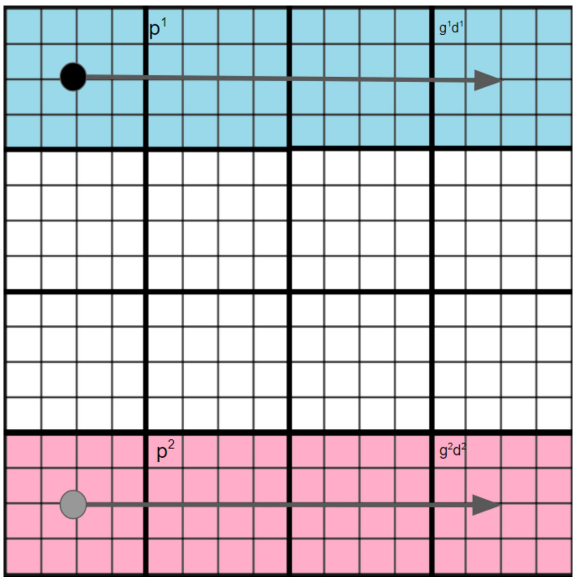

(c) Level 1 graph G_1, created by calling $CreateNewLeveL(G, regions_0, regioncounter)$. $regions_0$ and $regioncounter$ are outputs of in the output of *CreateFirstLevel(*$G, (2x2)$*)*.

Figure 5.11 A 3-level example for **HMAPDC**, first part.

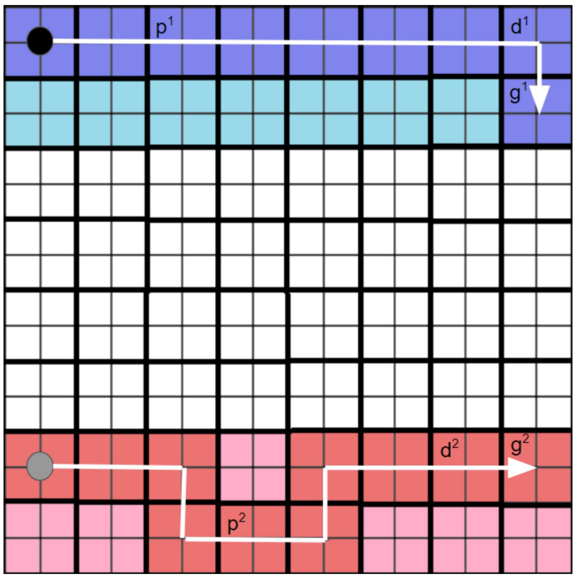


(a) A **MAPDC** instance. There are two agents (with goal locations denoted g^1 and g^2) and two tasks (with pick locations p^1, p^2 and delivery locations d^1, d^2). First agent a_1 is colored black, the other, a_2 gray.

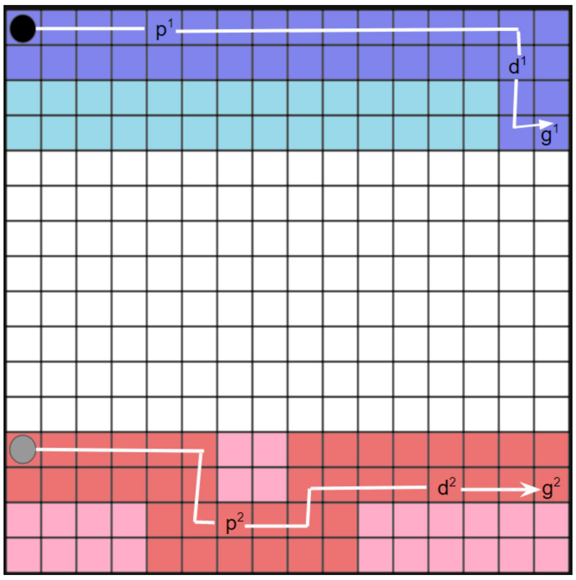


(b) Level 0 and its solution. G_0 is a 4x4 grid, highlighted by thick borders. Here, $T_1 = \{1\}$ and $T_2 = \{2\}$. The regions that agents pass through are colored light blue and pink respectively. These will be the regions that agents will be able to move through in the lower levels.

Figure 5.12 A big example for **HMAPDC**, second part.



(a) Level 1 and its solution. The regions of G_1 are highlighted by thick borders. It is an 8x8 grid in this case. The task assignment is the same as Level 0. The regions agents move through in this level are colored blue and red respectively. Their paths at this level are colored white. Again, the agents will be able to move through only the blue and red regions in the lower level.



(b) Level 2 and its solution. G_2 is the original grid G. In the solution, the agents are assigned the same tasks again. They move through blue (respectively red) colored nodes to complete their tasks and reach their goal destination.

5.6 HMAPDC: Solving MAPDC using ASP Hierarchically

Here, we give a detailed description of our methodology. Algorithm 6 describes the main method.

We proceed to describe how we form the ASP instances for the sub-problems.

We have mentioned that the hierarchical levels are 6-tuples $(G_l = (V_l, E_l), init_l, goal_l, pick_l, deliver_l, m_l)$. For each level, we call F^{I_l} the *level facts*. Level facts of a level has all the level dependent information about a **MAPDC** instance I. F^{I_l} is obtained from a level as follows:

`agent(a).` for $a \in A$

`node(v).` for $v \in V_l$

`edge(v,k).` for $(v,k) \in E_l$

`init(a,v)` for $a \in A, init_l(a) = v$

`goal(a,v).` for $a \in A, goal_l(a) = v$

`task(r,p,d).` for $r \in T, pick_l(r) = p, deliver_l(r) = d$

`capacity(c).` for c

The function $levelFacts(G_l, A, init_l, goal_l, T, pick_l, deliver_l)$ returns the set of facts mentioned above.

In addition to these, for **MAPDC-Middle** and **MAPDC-Bottom**, we need the information on which agent can visit which vertices. We call these facts F^v. The function called *findValids* described in Algorithm 7 takes a solution to the previous level s, level vertices V_l, and m_l, and returns a set of "`valid(a,v).`" facts that describes agent a can move through vertex v.

Algorithm 6 details our method.

After obtaining the facts for the **MAPDC-Top** instance, we solve it via ASP at line 2. In the loop that spans lines 3-7, we use the previous solution s to obtain F^v, then together with other facts that constitute the **MAPDC-Middle** instance, we solve it and store the answer set in s. We do this for all **MAPDC-Middle** instances, and then between lines 8-9 we obtain the facts for the last level, which constitute the **MAPDC-Bottom** instance, and solve it.

Algorithm 6: HMAPDC

```
Input: A MAPDC instance $I = (G, O, A, init, goal, T, c)$, hierarchical levels
       $L = [(G_0, init_0, goal_0, pick_0, deliver_0, m_0), \ldots,$
       $(G_{numlevels-1}, init_{numlevels-1}, goal_{numlevels-1}, pick_{numlevels-1},$
       $deliver_{numlevels-1}, m_{numlevels-1})]$
Output: A solution to $I$
1  $F^{I_0} \leftarrow$ levelFacts($G_0$, $A$, $init_0$, $goal_0$, $T$, $pick_0$, $deliver_0$);
2  s $\leftarrow$ compute answer set for $P_{top} \cup F^{I_0}$;
3  for $l$ in {1,...,numlevels-2} do
4      $F^v \leftarrow$ findValids(s, $V_l$, $m_l$);
5      $F^{I_l} \leftarrow$ levelFacts($G_l$, $A$, $init_l$, $goal_l$, $T$, $pick_l$, $deliver_l$);
6      s $\leftarrow$ compute answer set for $P_{mid} \cup F^{I_l} \cup F^v$ ;
7  $F^v \leftarrow$ findValids(s, $V_{numlevels-1}$, $m_{numlevels-1}$);
8  $F^{I_{numlevels-1}} \leftarrow$
     levelFacts($G_{numlevels-1}$, $A$, $init_{numlevels-1}$, $goal_{numlevels-1}$, $T$, $pick_{numlevels-1}$,
     $deliver_{numlevels-1}$);
9  s $\leftarrow$ compute answer set for $P_{bot} \cup F^{I_{numlevels-1}} \cup F^v$;
10 solution $\leftarrow$ extract answer from s;
11 return solution;
```

Algorithm 7: findValids

```
Input: a solution $s$ as an answer set (to the sub-problem at level $l-1$), $V_l$, $m_l$
Output: a set of facts of valid/2 atoms that represent the regions agents can
        move through in level $l$
1  valids $\leftarrow$ {};
2  for traversal(a,t,v) in $s$ do
3      for $v_l$ in $V_l$ do
4          if $v == m_l(v_l)$ then
5              valids $\leftarrow$ valids $\cup$ "valid(a,v_l).";
6  return valids;
```

5.6.1 An example for HMAPDC

Consider the example in Figure 3.2. Figure 5.1 illustrates how we can apply **HMAPDC** to this instance. There are just two levels in this case. The first level reduces the dimension of the grid to 4x4. The regions are colored orange and gray. We then find paths for agents in this abstracted graph, such that each agent arrives at its goal region and all the tasks are completed.

In the second and last level we solve the original problem on the original graph, but the agents are limited in which nodes they can use to complete their traversals as implied from the upper level. Green nodes are only allowed to the green agent, magenta nodes to the magenta agent, and blue nodes to both. This way, both the grounder and the solver do not need to explore all possible paths for agents but a considerably small subset of them.

An example with 3 levels (one top, one middle, one bottom) can be seen in Figure 5.11 and its continuation (Figure 5.12).

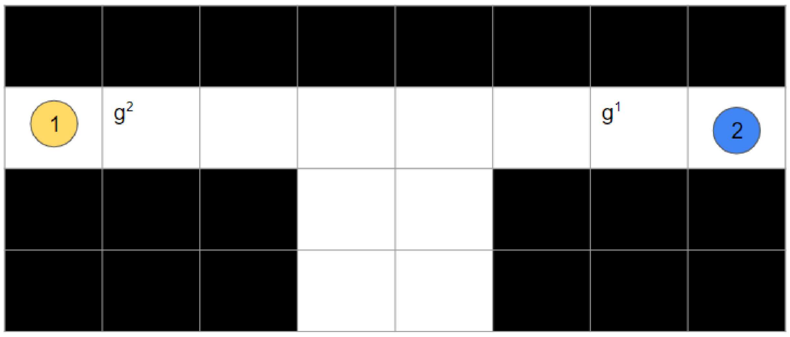
5.7 Limitations of HMAPDC

As will be apparent in Section 7 where we present our results, **HMAPDC** performs much better in terms of computation time. However, this is achieved through sacrifice on both completeness and optimality. We also do not have any guarantees on the level of sub-optimality **HMAPDC** causes.

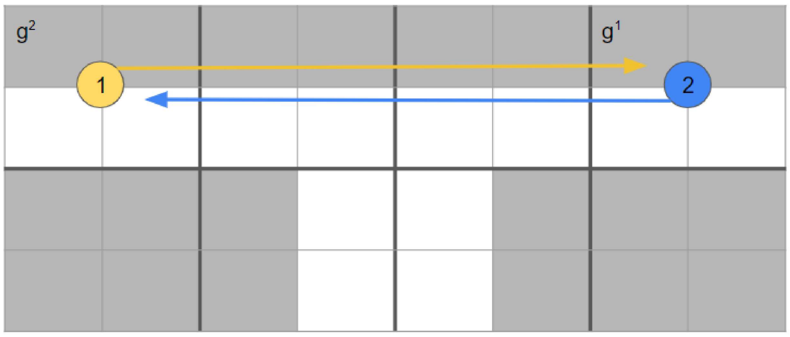
Figure 5.13 shows one case where **HMAPDC** cannot find a solution. First picture is the given instance. Second picture shows the regions created for the top level by bold borders. In order to solve the instance, agents 1 and 2 need to swap places in a corridor, which is normally possible. But the high level solution found in the second picture, which did not take into consideration the collision and swapping constraints, forces the paths of agents to stay in the green zone in the third picture. The dark blue area is out of bounds for both agents in the third picture, thus the agents cannot use those nodes to give way to each other and a solution cannot be found.

Also, the solution we find at the first level should have considerable influence on both the makespan we find and the computation cost. This may be overcomed by producing several solutions at the highest level, which is computationally the

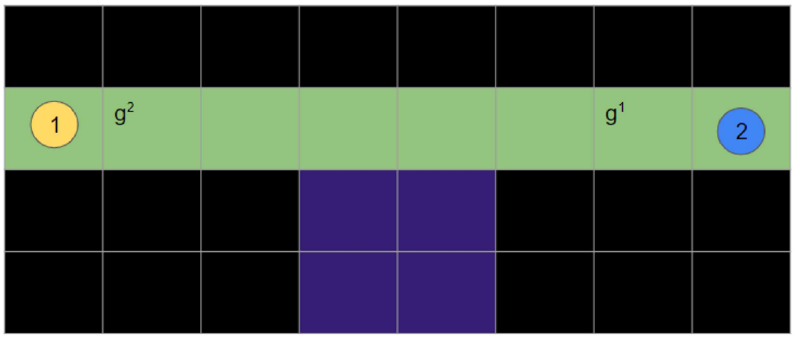
Figure 5.13 A demonstration of the incompleteness of **HMAPDC**



(a) The original **MAPDC** instance.



(b) Level 0 (regions are differentiated by bold borders) and its solution.



(c) At Level 1, both agents are restricted to the green nodes and cannot use dark blue nodes to give way to each other. No solution can be found for this level.

cheapest and has the most influence on the successive levels, and solving the problem with a portfolio of first level solutions.

5.8 Further Augmentations to HMAPDC

We propose two additional and separate augmentations to **HMAPDC**. Firstly, we propose adding an extra **MAPDC-Middle** level before **MAPDC-Bottom**. We call this *additional layer*.

Secondly, we propose the *task assignment* heuristic. We add *al* and/or *h* to **HMAPDC** to denote when we use these augmentations as **HMAPDC+al**, **HMAPDC+h**, **HMAPDC+al+h**.

5.8.1 HMAPDC+al: Adding an extra layer before MAPDC-Bottom

Normally, the last **MAPDC-Middle** level has regions of size 2x2 and we pass its solution directly to **MAPDC-Bottom** which enforces collision constraints. With the additional **MAPDC-Middle** level, we reduce the region sizes further down to 1x1, meaning we solve the problem on the original graph but without the collision constraints. The solution produced restricts **MAPDC-Bottom** to just being able to reschedule the agents on the paths given by the additional layer. This may help with the larger graphs, but also increase the possibility that **MAPDC-Bottom** will not be able to find a solution.

We apply this heuristic by inserting Algorithm 8 to Algorithm 1 just before the return statement, after Line 13. We add a copy of the last level to the sequence of levels we need to solve, only changing the parents of the regions to themselves.

Algorithm 8: additional layer

```
1 $G_{al} = (V_{al}, E_{al}), init_{al}, goal_{al}, pick_{al}, deliver_{al}, m_{al} \leftarrow$ ls[-1];
2 $m_{al}(v) \leftarrow v$ for $v \in V_{al}$;
3 ls.append($(G_{al}, init_{al}, goal_{al}, pick_{al}, deliver_{al}, m_{al})$);
```

5.8.2 Task Assignment Heuristic

Normally with **HMAPDC**, we only restrict the movement of agents into regions of the graph. But with this heuristic, we also enforce the task assignment given by **MAPDC-Top** onto lower levels. Since we decrease the number of choices in lower levels, we expect this to help with solution time.

To implement this heuristic, we need a function called $findAssignments(s)$ which will take as input a solution s, and extract and return a list of "$assignment(a, id).$" facts. This will be done in a very similar way to Algorithm 7, thus we do not detail its implementation. Let us call these facts F^a, and insert "$F^a \leftarrow findAssignments(s);$" into Algorithm 6 after line 2.

Then, we need to add these facts that constitute the top level assignments to the level facts F^{I_l} for every $l > 0$. We do this by inserting "$F^{I_l} \leftarrow F^{I_l} \cup F^a$;" into the same algorithm after line 5.

5.9 HMAPDC-P: Utilizing Diverse Solutions to Improve HMAPDC

One shortcoming of **HMAPDC** is that the solutions found by higher levels can affect performance a lot both in terms of computation time and makespan, and that we do not have the means to tell what kind of top level solution would benefit the lower levels most.

One way to approach this would be to find some diverse solutions on the highest level and have **HMAPDC** try all of them. Morag, Felner, Stern, Atzmon, Boyarski, Louis & Toledano (Morag et al.) shows that having several attempts using fast but sub-optimal solvers can improve performance significantly.

Eiter et al. (2009) provides methods to find similar/diverse solutions for ASP programs. With CLINGO, we can populate a given number of solutions to an ASP program, but we do not have any guarantees on their similarity/diversity.

If we wanted k solutions that are at least d distant to each other by a defined metric, we can utilize online method 2 given in Eiter et al. (2009). This method runs the program for k times, and adds appropriate constraints in each iteration to ensure the solutions found are similar or diverse. Algorithm 9 uses this method.

To utilize this method, we require a similarity measure between two solutions. We also want this similarity measure to be impactful, meaning solutions that are distant to each other with respect to this measure differ sufficiently. One such measure could be the similarity between task assignments of two solutions. Paths of agents depend heavily on which tasks an agent must perform, so this measure should be impactful. Consider two answer sets s_1, s_2, which have `assignment/2` atoms and let T be the set of tasks, A be the set of agents for the instance we are solving. *Task-similarity* of these would be

$$\frac{|\{t \in T | \exists a \in A, \texttt{assignment(a,t)} \in s_1 \land \texttt{assignment(a,t)} \in s_2\}|}{|T|}$$

We propose a Task Assignment Portfolio (TAP) (Algorithm 9) that computes p solutions to **MAPDC-Top**, such that no two solutions are more than r task-similar. As TAP produces solutions, it calls **HMAPDC-P** which is presented in Algorithm

10. **HMAPDC-P** is a slightly modified version of **HMAPDC**. It does not solve the first level and instead takes its solution as an additional input. Since TAP does not depend on any output from **HMAPDC-P**, several **HMAPDC-P** runs can be made in a parallel way as new solutions to the first level are produced.

Below is the ASP program $P_{portfolio}$ which is used in Algorithm 9. This program constrains a new solution to be at most r task-similar to any previous solution.

```
:- L{task(I,P,D): prevAssignment(S,A,I), assignment(A,I)},
    limit(L), prevSolution(S).
```

"`limit(`$\lfloor r * |T| \rfloor$`).`" should be added to the program.

Algorithm 9: TAP (Task Assignment Portfolio)

```
Input: a MAPDC instance $I$, a first level size $(X,Y)$, allowed task-similarity
       $r$, size of portfolio $p$
1 $P_{portfolio} \leftarrow P_{portfolio} \cup \{$limit($\lfloor r * |T| \rfloor$).$\}$;
2 L $\leftarrow$ createLevels($I, (X,Y)$);
3 $G_0, init_0, goal_0, pick_0, deliver_0 \leftarrow L[0]$;
4 $F^{I_0} \leftarrow$ levelFacts($G_0, A, init_0, goal_0, T, pick_0, deliver_0$);
  /* In the loop, find top-level solutions, run HMAPDC-P with the found solutions, and add
     constraints to the portfolio program so that consequent top-level solutions are diverse.
     */
5 for $i$ in $\{1,\ldots,p\}$ do
6     s $\leftarrow$ solve($P_{top} \cup P_{portfolio} \cup F^{I_0}$);
7     HMAPDC-P($I, s, L$);
8     assignments $\leftarrow$ findAssignments(s);
9     $P_{portfolio} \leftarrow P_{portfolio} \cup \{$prevSolution(i).$\} \cup$
       $\{$prevAssignment(i,a,t).$|$assignment(a,t). $\in assignments\}$;
```

Algorithm 10: HMAPDC-P

```
Input: A MAPDC instance $I = (G, O, A, init, goal, T, c)$, a solution to the first
       level $s$, hierarchical levels $L = [(G_0, init_0, goal_0, pick_0, deliver_0, m_0), \dots,$
       $(G_{numlevels-1}, init_{numlevels-1}, goal_{numlevels-1}, pick_{numlevels-1},$
       $deliver_{numlevels-1}, m_{numlevels-1})]$
Output: a solution to $I$
1 for $k$ in $\{1, \dots, numlevels\text{-}2\}$ do
2     $F^v \leftarrow$ findValids(s, $V_l$, $m_l$);
3     $F^{I_l} \leftarrow$ levelFacts($G_l$, $A$, $init_l$, $goal_l$, $T$, $pick_l$, $deliver_l$);
4     s $\leftarrow$ compute answer set for $P_{mid} \cup F^{I_l} \cup F^v$ ;
5 $F^v \leftarrow$ findValids(s, $V_{numlevels-1}$, $m_{numlevels-1}$);
6 $F^{I_{numlevels-1}} \leftarrow$
    levelFacts($G_{numlevels-1}$, $A$, $init_{numlevels-1}$, $goal_{numlevels-1}$, $T$, $pick_{numlevels-1}$,
    $deliver_{numlevels-1}$);
7 s $\leftarrow$ compute answer set for $P_{bot} \cup F^{I_{numlevels-1}} \cup F^v$;
8 solution $\leftarrow$ extract solution from s;
9 return solution;
```

5.10 On the Generalizability of of HMAPDC

Even though **HMAPDC** is designed to solve **MAPDC**, it should be clear that it could be used for **MAPF** as well. But there are other **MAPF** variants with different definitions of tasks, different requirements like group completion (Nguyen et al., 2019), or even versions where agents have batteries and need to visit charging stations when they run out of energy (Bogatarkan et al., 2020).

We believe many of these versions can benefit from a spatial hierarchical scheme like **HMAPDC**.

Our tasks have two steps, but more steps for tasks would be easy to adapt to. In general, variants of **MAPF** that introduce some kind of assigned or to-be-assigned way-points for agents would require little change in the general algorithm of **HMAPDC** which is presented in Algorithm 6.

Also consider the case where agents need to charge their battery after certain number of moves at certain charging locations on the grid. This case may require more changes to **HMAPDC**. One approach could be to include force the higher level solutions to have a battery charging point in the paths of all the agents. Another could be to find the solution to an abstract level and then expanding it such that the path includes the nearest charging station.

6. Experimental Evaluations

We evaluate our methods for **MAPDC** by empirically analyzing their scalability, and compare them with each other.

6.1 Research Questions

We are interested in testing the computational performance of our ASP solutions for **MAPDC**. The approaches differ in their optimality, completeness and intended usage so it would be best to state our research questions for each of them separately.

MAPDC-ASP and **MAPDC-M** are complete and optimal methods. It should be expected that they scale poorly as the problem size grows. Though such ASP-based methods can still be used as sub-solvers for distributed approaches (Zhang et al., 2021).

Here are the questions that we want to investigate for our **MAPDC** methods

- How does **MAPDC-ASP** (respectively **MAPDC-M**) scale as the problem size increases?
- How does the grid structure affect the computational performance?
  - When there are random obstacles?
  - When there is a structure e.g a warehouse?
- How does **MAPDC-ASP**, **MAPDC-M** and **MAPDC-P** (Tajelipirbazari et al., 2022) differ in performance and utility?

Our questions regarding **HMAPDC** are the following.

- How does **HMAPDC** scale?

Table 6.1 Benchmark 1: Warehouse configurations used in our experiments.

Configuration	1	2	3	4	5	6	7
Grid Size	10x10	10x10	10x10	10x10	10x20	10x40	20x20
Shelf Width	1	1	2	2	2	2	2
Vertical Distance	1	1	1	2	2	2	2
Horizontal Distance	1	2	1	2	2	2	2

- How does initial region size affect performance/solution quality?
- How much optimality is compromised?
- How does **HMAPDC** compare to **MAPDC-ASP** and **MAPDC-M**?
- How does the proposed heuristics and augmentations (i.e., adding an additional layer, task assignment heuristic, starting with diverse solutions) improve the performance of **HMAPDC**?

6.2 Benchmarks

We created several benchmarks to investigate these questions.

6.2.1 Benchmark 1

The first benchmark is comprised of configurations described in Table 6.1 and is made to answer the following research questions: how do **MAPDC-ASP** and **MAPDC-M** perform in different warehouse designs? - how do they scale?

We have generated 7 different warehouse environments, varying the grid size, shelf width and horizontal/vertical distances between shelves as described in Table 6.1. Two of these configurations are shown in Figure 6.1 and Figure 6.2. In these configurations, the initial and goal locations of agents are on the last row of the grid, and we leave a margin of 2 cells from the left and the bottom before starting the shelves. We also leave a margin on the right, equal to the vertical distance from the top. If the shelves touch the right wall, we clear the right-most column from shelves as in Figure 6.2. In order to have predictable hardness for tasks, picking and delivery locations of the tasks are scattered along exactly one shelf in the vertical direction.

For each such warehouse configuration, we have created 9 triples by varying the number of agents, the number of tasks, and the capacity of the agents. For every

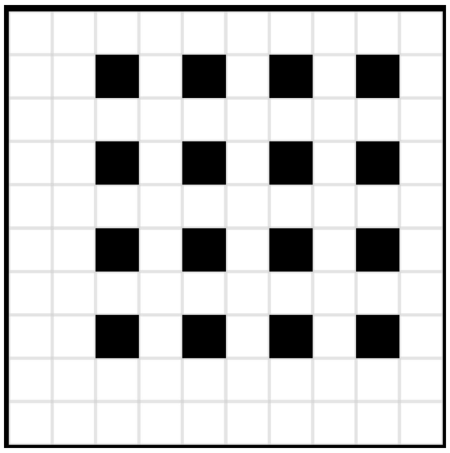


Figure 6.1 Configuration 1 of Table 6.1

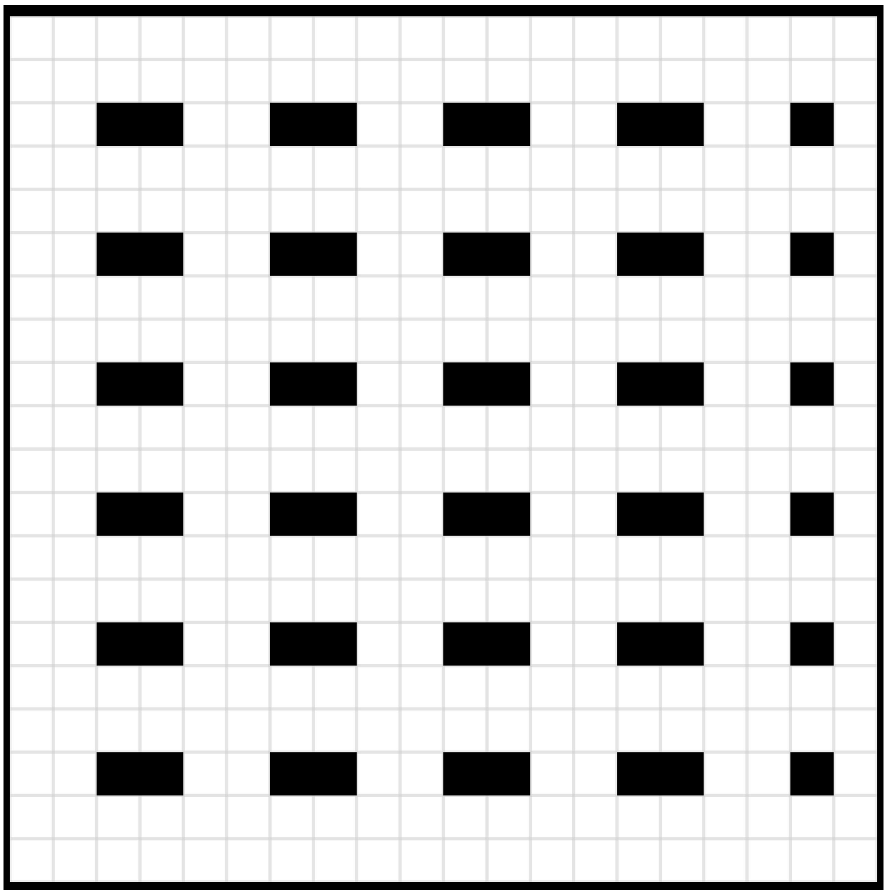


Figure 6.2 Configuration 7 of Table 6.1

combination of configuration, agent number, task number and capacity, we have randomly generated 5 instances and reported the average computation times and makespans.

6.2.2 Benchmark 2

With our second benchmark, we are interested in investigating how **MAPDC** methods and **HMAPDC** compare in terms of the makespans and the computation time. For this, we created instances that push the capabilities of **MAPDC** methods, while being relatively easy to handle for **HMAPDC**. We generate graphs for grid sizes 24x24, 36x36, 48x48 with *no obstacles*, *10% random obstacles*, and *warehouse* structure, and on these grid graphs we consider 5, 10, 20 agents. For the *10% random obstacles* case, we place obstacles on the grid randomly by generating a random number between the vertex values (0-99 for a 10x10 grid), and if the grid is not fully connected after all obstacles are placed, we restart the process. The number of tasks are determined by multiplying the number of agents and the given capacity in the experiment. For each combination of these parameters, we generate 5 **MAPDC** instances.

For warehouse instances, we leave a margin of 1 cell from the top and the right side, then start laying the shelves. We lay 10 cell-long shelves horizontally and then clear the right-most column from shelves. We also clear the last row of the grid. The 24x24 warehouse configuration is given in Figure 6.3.

Unlike Benchmark 1, the start and goal locations of agents, as well as pick and deliver locations of tasks are completely random.

6.2.3 Benchmark 3

In order to test the scalability of **HMAPDC**, and its comparison with other approaches, we generated bigger instances. We use grids of sizes 24x24, 48x48, 72x72, and 96x96. We use the same graph structures as Benchmark 2. We consider 5, 10, 20, 40, 60 agents; and one task for each agent. For each grid and number of agents, we generate 5 **MAPDC** instances. Again, the initial and goal location of agents as well as pick and delivery location of tasks are completely random.

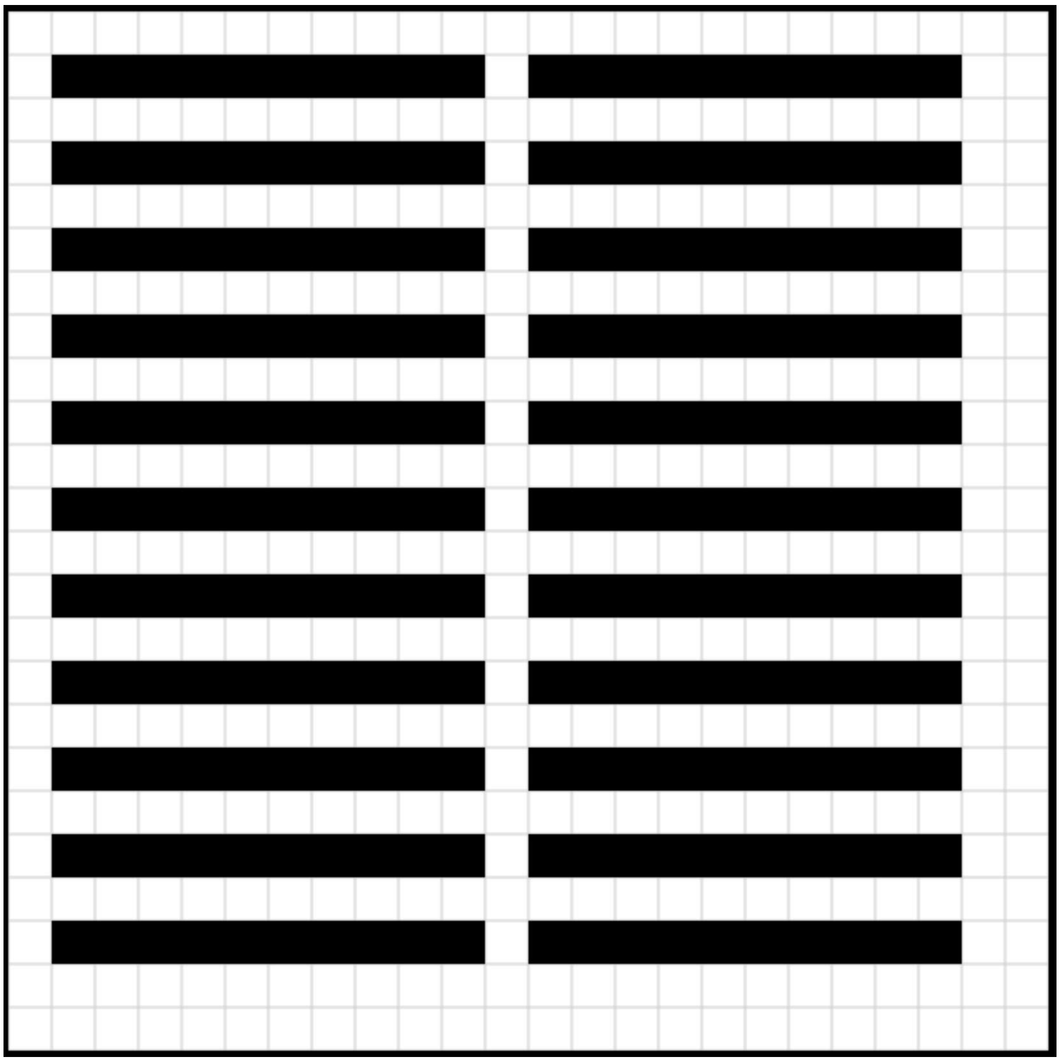


Figure 6.3 Warehouse configuration for grid gize 24x24

6.3 Experiments

We design several experiments to answer our research questions. For all the experiments, we have used the ASP solver Clingo (Version 4.5.4), on a Linux server with dual 2.4 GHz Intel E5-2665 CPUs and 64 GB memory.

6.3.1 Experiment 1

This experiment aims to measure the scalability of our single-shot and multi-shot encodings, and is run on Benchmark 1. It was done in Tajelipirbazari et al. (2022) and with two different encodings. We tested two encodings against each other, one of them is our graph based approach, **MAPDC-G**, which we presented in Section 3 and call **MAPDC-M**. The other is **MAPDC-P**, which is a planning based encoding.

We have experimentally evaluated **MAPDC-P** and **MAPDC-M** as to better understand their scalability in terms of the computation time. We also test the usability of **MAPDC-ASP**.

We also run Clingo with both default and handy settings and report our findings.

In later experiments, especially with ones that involve **HMAPDC**, we desire bigger warehouses. For any instance on a grid bigger than 20x20, we choose shelf width as 10, and vertical and horizontal distances as 1.

6.3.2 Experiment 2

This experiment aims to find what initial size we should choose for **HMAPDC**. We test on empty grids, grids with 10% random obstacles, and grids with corridors in them (called warehouse). We run this experiment on Benchmark 3 with capacity 1.

We have chosen to experiment with the following first level sizes: 6x6, 8x8, 10x10, 12x12. These sizes are managable for our multi-shot encodings, especially for the first level where we waive collision constraints.

6.3.3 Experiment 3

We aim to compare our multi-shot solution and **HMAPDC** with Experiment 3. We compare the makespans and solution times to learn how suboptimal **HMAPDC** is and how much time it saves us.

We run this experiment on Benchmark 2. The instances of Benchmark 2 are designed to be challenging enough for our multi-shot encoding, where using the sub-optimal **HMAPDC** becomes more reasonable.

Task counts for this experiments are given as *number of agents* * *capacity*.

6.3.4 Experiment 4

We test our augmentations to **HMAPDC**. Recall that we proposed two augmentations called *additional layer* and *task assignment* heuristics. We run this experiment on Benchmark 3 and report the time taken and the makespan.

6.3.5 Experiment 5

With this test, we aim to demonstrate how our portfolio approach, TAP, can help with improving performance. We run it on a portion of Benchmark 3, specifically the ones with grid size 72x72. Although TAP can be parallelized, we run TAP to work sequentially and report the first and best solutions both in terms of time and makespan. Benchmark 3 has 5 instances per configuration, but we test our approach on 1 instance for each configuration. We run TAP with first level size 8x8, and try two different portfolio sizes: $r = 0.75, p = 4$, and $r = 0.9, p = 10$.

Furthermore, we choose some configurations that previous experiments could not solve and apply TAP on them to see whether it helps with completeness.

7. Results

We present the results of our experiments in this section.

7.1 Results for Experiment 1: Scalability of MAPDC Methods

We have observed (Figure 7.1) that increasing the number of agents helps both **MAPDC-P** and **MAPDC-G** (identical to **MAPDC-M**) in finding solutions more efficiently. This observation makes sense as increasing the number of agents effectively reduces the number of tasks assigned to an agent, and, in turn, reduces the maximum makespan for the problem instance.

We have observed (Figure 7.2) that increasing the number of tasks increases the number of tasks that needs to be completed by each agent. Hence, the maximum makespan also increases, and this reduces the efficiency of both **MAPDC-P** and **MAPDC-M**. Note that some of the instances could not be solved within the time limit as the number of tasks increases to 32 for 16 agents.

Similar to increasing the number of agents, increasing the capacity of agents (Figure 7.3) leads to a decrease in maximum makespan, and thus reduces the computation time for both **MAPDC-P** and **MAPDC**. It is interesting to compare results in Figures 7.1 and 7.3. Even though increasing the number of agents and the capacity both reduce the maximum makespan and improve runtime performance, improvements in Figure 7.3 are more visible. This showcases the importance of capacity in the **MAPDC** problem. We reason that increase in the number of agents reduces the maximum makespan, but at the same time strains the solving process due to increased number of possible collisions to avoid.

We have also compared the single-shot and multi-shot computations of Clingo over some **MAPDC** instances (Table 7.1). The single-shot ASP formulations of **MAPDC** utilize weak constraints to optimize the maximum makespan, while a sufficiently small upper bound is provided on the makespan for the purpose of grounding. In our experiments with single-shot, we have provided the optimum makespans as upper bounds on makespans. In this way, the single-shot computations do not need to make too many optimizations for large upper bounds but show their best performance alleviating the disadvantage of grounding due to large makespans. It can be seen that even under these ideal conditions, the multi-shot computations perform nearly as good as the single-shot ones for many instance. The run-time of multi-shot computations include the time required to find the optimum makespan, making them more practical to use, in particular, with **MAPDC-P**.

Furthermore, we have evaluated the single-shot computations with anytime search, with time threshold of 1000 seconds, over some **MAPDC** instances with an upper bound of 80 on makespan (Table 7.2). We have observed that anytime search helps with finding a solution for more instances, in particular, for **MAPDC-M**, but at the cost of computation time due to grounding with a large makespan.

Finally, our experiments show that using Clingo with `handy` configuration (as in Erdem et al. (2013)) improves the computational performances of **MAPDC-P** and **MAPDC-M** for hard instances (cf. results for Configurations 6 and 7 at Table 7.4). This suggests the use of Clingo with a portfolio of different configurations.

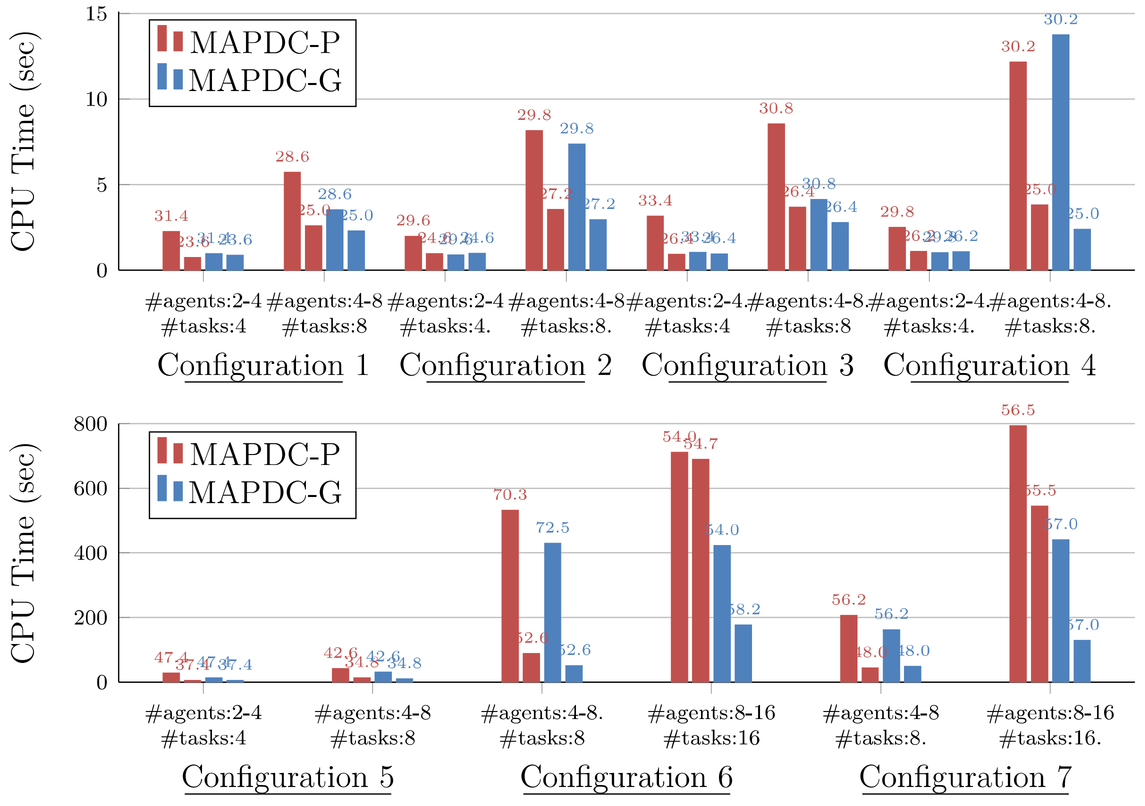


Figure 7.1 Experiment 1: Scalability as the number of agents increases when capacity = 1 (Table 7.3).

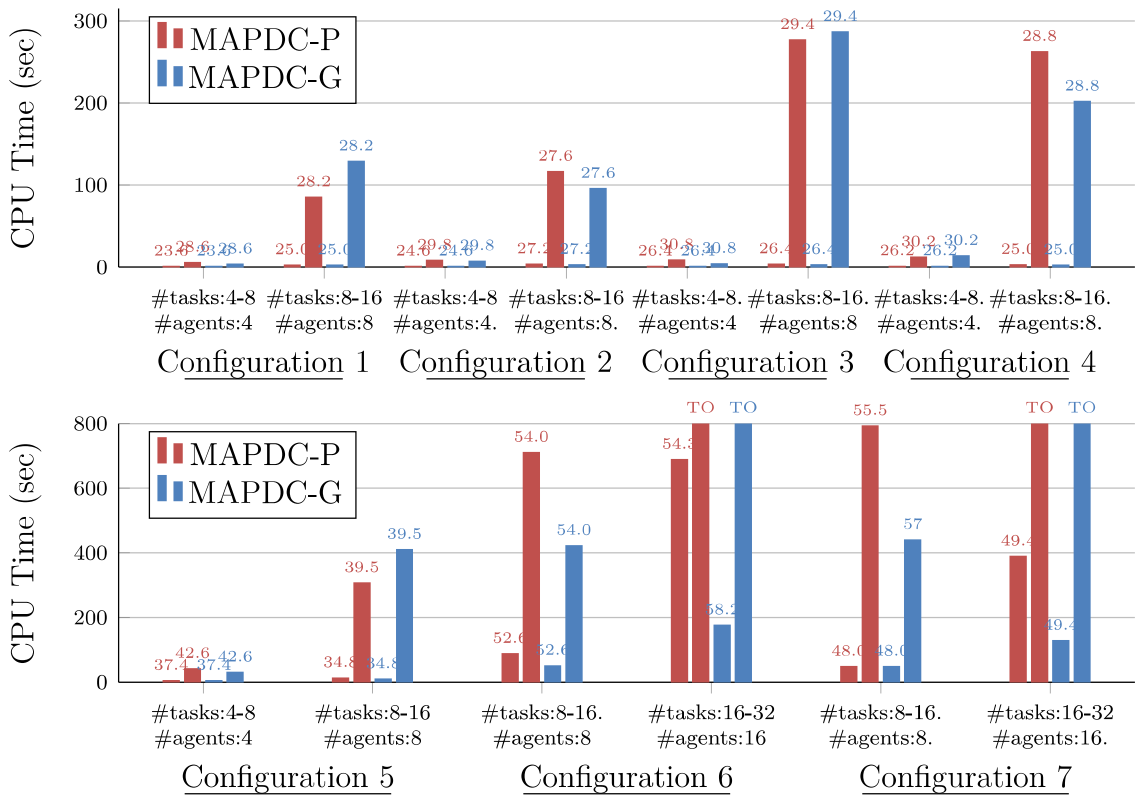


Figure 7.2 Experiment 1: Scalability as the number of tasks increases when capacity = 1 (Table 7.3).

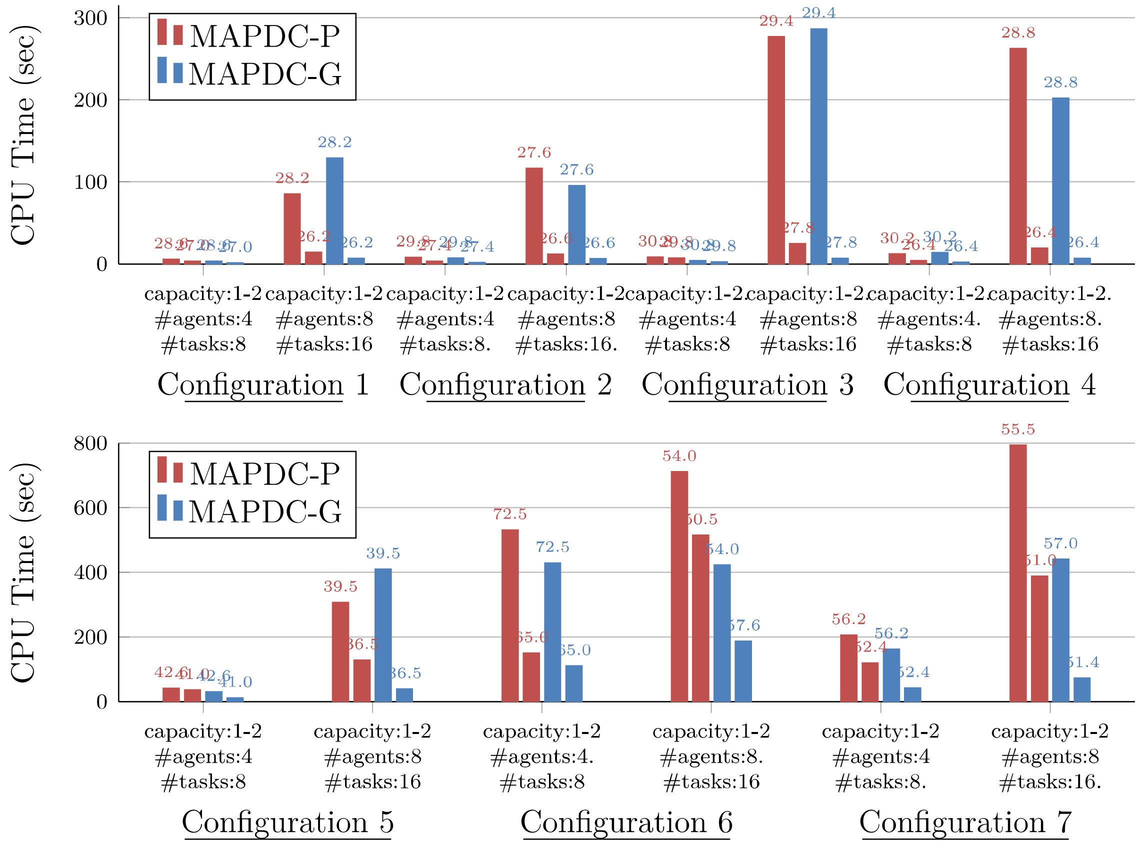


Figure 7.3 Experiment 1: Scalability as the capacity increases (Table 7.3).

Table 7.1 Experiment 1: Comparison of single-shot and multi-shot computations over Configuration 5 instances (agents, tasks, capacity).

	Single-shot				Multi-shot			
	MAPDC-G		**MAPDC-P**		**MAPDC-G**		**MAPDC-P**	
instance	CPU time	makespan (#solved)	CPU time	makespan (#solved)	CPU time	makespan (#solved)	CPU time	makespan (#solved)
(2, 2, 1)	2.98	35.4 (5)	0.99	35.4 (5)	1.92	35.4 (5)	2.16	35.4 (5)
(2, 4, 1)	9.52	47.4 (5)	27.59	47.4 (5)	13.23	47.4 (5)	27.48	47.4 (5)
(2, 4, 2)	4.63	41.6 (5)	8.13	41.6 (5)	3.84	41.6 (5)	14.54	41.6 (5)
(4, 4, 1)	7.51	37.4 (5)	4.49	37.4 (5)	4.66	37.4 (5)	4.79	37.4 (5)
(4, 8, 1)	25.11	42.6 (5)	39.14	42.6 (5)	30.69	42.6 (5)	41.68	42.6 (5)
(4, 8, 2)	12.62	41.0 (5)	40.48	41.0 (5)	11.81	41.0 (5)	36.41	41.0 (5)
(8, 8, 1)	15.49	34.8 (5)	16.63	34.8 (5)	9.73	34.8 (5)	13.15	34.8 (5)
(8, 16, 1)	146.90	**39.5 (4)**	493.02	**39.5 (4)**	410.00	**39.5 (4)**	307.07	**39.5 (4)**
(8, 16, 2)	29.71	36.6 (5)	255.04	36.6 (5)	38.94	36.6 (5)	128.82	36.6 (5)

Table 7.2 Experiment 1: Single-shot computations with anytime search vs multi-shot computations, over Configuration 7 instances (agents, tasks, capacity), with the time threshold of 1000 secs and the upper bound 80 on makespan.

Single-shot anytime

	MAPDC-G		MAPDC-P	
instance	CPU Time	Makespan (#opt/#solved)	CPU Time	Makespan (#opt/#solved)
(4, 4, 1)	95.58	43.0 (5/5)	57.99	43.0 (5/5)
(4,8,1)	201.12	56.2 (5/5)	794.14	**56.2 (4/5)**
(4,8,2)	122.59	52.4 (5/5)	710.64	52.4 (5/5)
(8,8,1)	189.66	48.0 (5/5)	202.67	48.0 (5/5)
(8,16,1)	855.31	**58.2 (2/5)**	timeout	timeout
(8,16,2)	255.07	51.4 (5/5)	997.27	**76.5 (0/2)**
(16,16,1)	457.70	49.4 (5/5)	997.23	**67.0 (0/4)**
(16,32,1)	997.61	**69.4 (0/5)**	timeout	timeout
(16,32,2)	836.89	**54.0 (2/5)**	timeout	timeout

Multi-shot

	MAPDC-G		MAPDC-P	
instance	CPU Time	Makespan (#opt/#solved)	CPU Time	Makespan (#opt/#solved)
(4, 4, 1)	16.99	43.0(5)	16.21	43.0(5)
(4,8,1)	162.02	56.2(5)	233.99	56.2(5)
(4,8,2)	42.35	52.4(5)	127.96	52.4(5)
(8,8,1)	48.76	48.0(5)	64.74	48.0(5)
(8,16,1)	440.30	57.0 (3)	816.92	56.5 (2)
(8,16,2)	72.97	51.4(5)	511.79	49.8(4)
(16,16,1)	128.51	49.4 (5)	544.47	47.0 (4)
(16,32,1)	timeout	timeout	timeout	timeout
(16,32,2)	264.36	52.2 (4)	timeout	timeout

Table 7.3 Results with CLINGO's `default` configuration

				MAPDC-G		MAPDC-P	
configuration	agents	tasks	capacity	CPU time	makespan (#solved)	CPU time	makespan (#solved)
	2	2	1	0.46	24.4 (5)	0.53	24.4 (5)
	2	4	1	0.96	31.4 (5)	2.57	31.4 (5)
	2	4	2	0.69	29.8 (5)	2.19	29.8 (5)
	4	4	1	0.86	23.6 (5)	0.79	23.6 (5)
Configuration 1	4	8	1	3.52	28.6 (5)	4.33	28.6 (5)
	4	8	2	1.45	27.0 (5)	2.69	27.0 (5)
	8	8	1	2.29	25.0 (5)	2.55	25.0 (5)
	8	16	1	128.86	28.2 (5)	71.54	28.2 (5)
	8	16	2	6.72	26.2 (5)	12.27	26.2 (5)
	2	2	1	0.39	21.2 (5)	0.37	21.2 (5)
	2	4	1	0.88	29.6 (5)	2.28	29.6 (5)
	2	4	2	0.81	29.2 (5)	2.34	29.2 (5)
	4	4	1	0.98	24.6 (5)	0.96	24.6 (5)
Configuration 2	4	8	1	7.36	29.8 (5)	6.75	29.8 (5)
	4	8	2	1.66	27.4 (5)	3.10	27.4 (5)
	8	8	1	2.95	27.2 (5)	3.87	27.2 (5)
	8	16	1	95.61	27.6 (5)	59.76	27.6 (5)
	8	16	2	6.53	26.6 (5)	11.92	26.6 (5)
	2	2	1	0.40	22.8 (5)	0.44	22.8 (5)
	2	4	1	1.04	33.4 (5)	4.01	33.4 (5)
	2	4	2	0.76	31.4 (5)	2.68	31.4 (5)
	4	4	1	0.95	26.4 (5)	1.05	26.4 (5)
Configuration 3	4	8	1	4.13	30.8 (5)	7.15	30.8 (5)
	4	8	2	2.66	29.8 (5)	6.28	29.8 (5)
	8	8	1	2.78	26.4 (5)	3.65	26.4 (5)
	8	16	1	286.58	29.4 (5)	165.75	29.4 (5)
	8	16	2	6.95	27.8 (5)	21.45	27.8 (5)
	2	2	1	0.45	23.4 (5)	0.48	23.4 (5)
	2	4	1	1.01	29.8 (5)	2.48	29.8 (5)
	2	4	2	0.74	27.4 (5)	1.69	27.4 (5)
	4	4	1	1.07	26.2 (5)	0.96	26.2 (5)
Configuration 4	4	8	1	13.74	30.2 (5)	8.87	30.2 (5)
	4	8	2	2.31	26.4 (5)	3.49	26.4 (5)
	8	8	1	2.39	25.0 (5)	3.05	25.0 (5)
	8	16	1	201.95	28.8 (5)	107.11	28.8 (5)
	8	16	2	7.02	26.4 (5)	19.34	26.4 (5)
	2	2	1	1.92	35.4 (5)	2.69	35.4 (5)
	2	4	1	13.23	47.4 (5)	44.88	47.4 (5)
	2	4	2	3.84	41.6 (5)	23.37	41.6 (5)
	4	4	1	4.66	37.4 (5)	6.19	37.4 (5)
Configuration 5	4	8	1	30.69	42.6 (5)	38.96	42.6 (5)
	4	8	2	11.81	41.0 (5)	29.34	41.0 (5)
	8	8	1	9.73	34.8 (5)	17.99	34.8 (5)
	8	16	1	410.00	39.5 (4)	429.42	39.5 (4)
	8	16	2	38.94	36.6 (5)	139.65	36.6 (5)
	4	4	1	27.82	56.6 (5)	70.44	56.6 (5)
	4	8	1	429.16	**72.5 (4)**	440.65	**70.3 (3)**
	4	8	2	110.65	65.0 (5)	253.01	65.0 (5)
	8	8	1	50.86	52.6 (5)	212.67	52.6 (5)
Configuration 6	8	16	1	422.37	**54.0 (1)**	timeout	timeout
	8	16	2	187.27	**57.6 (5)**	138.18	**48.0 (1)**
	16	16	1	176.37	**58.2 (5)**	timeout	timeout
	16	32	1	timeout	timeout	timeout	timeout
	16	32	2	440.93	**55.2 (4)**	timeout	timeout
	4	4	1	16.99	43.0 (5)	16.21	43.0(5)
	4	8	1	162.02	56.2 (5)	233.99	56.2 (5)
	4	8	2	42.35	52.4 (5)	127.96	52.4 (5)
	8	8	1	48.76	48.0 (5)	64.74	48.0 (5)
Configuration 7	8	16	1	440.30	**57.0 (3)**	816.92	**56.5 (2)**
	8	16	2	72.97	**51.4 (5)**	511.79	**49.8 (4)**
	16	16	1	128.51	**49.4 (5)**	544.47	**47.0 (4)**
	16	32	1	timeout	timeout	timeout	timeout
	16	32	2	264.36	**52.2 (4)**	timeout	timeout

Table 7.4 Results with CLINGO's `handy` configuration

				MAPDC-G		**MAPDC-P**	
configuration	agents	tasks	capacity	CPU time	makespan (#solved)	CPU time	makespan (#solved)
	2	2	1	0.57	24.4 (5)	0.50	24.4 (5)
	2	4	1	1.00	31.4 (5)	2.30	31.4 (5)
	2	4	2	0.88	29.8 (5)	1.63	29.8 (5)
	4	4	1	1.18	23.6 (5)	0.72	23.6 (5)
Configuration 1	4	8	1	2.82	28.6 (5)	3.93	28.6 (5)
	4	8	2	1.75	27.0 (5)	2.26	27.0 (5)
	8	8	1	3.07	25.0 (5)	2.64	25.0 (5)
	8	16	1	68.69	28.2 (5)	63.33	28.2 (5)
	8	16	2	5.47	26.2 (5)	13.30	26.2 (5)
	2	2	1	0.50	21.2 (5)	0.37	21.2 (5)
	2	4	1	1.06	29.6 (5)	2.10	29.6 (5)
	2	4	2	0.98	29.2 (5)	2.22	29.2 (5)
	4	4	1	1.34	24.6 (5)	1.01	24.6 (5)
Configuration 2	4	8	1	4.48	29.8 (5)	6.52	29.8 (5)
	4	8	2	1.95	27.4 (5)	2.92	27.4 (5)
	8	8	1	3.97	27.2 (5)	3.73	27.2 (5)
	8	16	1	52.22	27.6 (5)	50.02	27.6 (5)
	8	16	2	4.62	26.6 (5)	13.29	26.6 (5)
	2	2	1	0.49	22.8 (5)	0.42	22.8 (5)
	2	4	1	1.08	33.4 (5)	3.44	33.4 (5)
	2	4	2	0.90	31.4 (5)	3.02	31.4 (5)
	4	4	1	1.28	26.4 (5)	1.09	26.4 (5)
Configuration 3	4	8	1	2.99	30.8 (5)	6.49	30.8 (5)
	4	8	2	2.52	29.8 (5)	5.69	29.8 (5)
	8	8	1	3.19	26.4 (5)	3.20	26.4 (5)
	8	16	1	146.62	29.4 (5)	172.00	29.4 (5)
	8	16	2	5.30	27.8 (5)	20.61	27.8 (5)
	2	2	1	0.59	23.4 (5)	0.47	23.4 (5)
	2	4	1	1.14	29.8 (5)	2.16	29.8 (5)
	2	4	2	0.87	27.4 (5)	1.65	27.4 (5)
	4	4	1	1.51	26.2 (5)	0.90	26.2 (5)
Configuration 4	4	8	1	8.58	30.2 (5)	8.64	30.2 (5)
	4	8	2	2.33	26.4 (5)	2.84	26.4 (5)
	8	8	1	3.47	25.0 (5)	3.00	25.0 (5)
	8	16	1	97.73	28.8 (5)	110.21	28.8 (5)
	8	16	2	6.25	26.4 (5)	17.32	26.4 (5)
	2	2	1	3.10	35.4 (5)	2.37	35.4 (5)
	2	4	1	10.83	47.4 (5)	37.53	47.4 (5)
	2	4	2	5.13	41.6 (5)	19.80	41.6 (5)
	4	4	1	7.52	37.4 (5)	5.51	37.4 (5)
Configuration 5	4	8	1	24.77	42.6 (5)	40.43	42.6 (5)
	4	8	2	12.05	41.0 (5)	33.51	41.0 (5)
	8	8	1	15.43	34.8 (5)	15.27	34.8 (5)
	8	16	1	221.68	39.5 (4)	376.10	39.5 (4)
	8	16	2	20.96	36.6 (5)	127.45	36.6 (5)
	4	4	1	57.50	56.6 (5)	44.43	56.6 (5)
	4	8	1	393.59	72.5 (4)	479.13	72.5 (4)
	4	8	2	75.52	65.0 (5)	212.28	65.0 (5)
	8	8	1	107.17	52.6 (5)	94.21	52.6 (5)
Configuration 6	8	16	1	283.07	54.0 (1)	734.57	54.0 (1)
	8	16	2	184.27	**57.6 (5)**	491.01	**50.5 (2)**
	16	16	1	336.67	**58.2 (5)**	801.47	**54.7 (3)**
	16	32	1	timeout	timeout	timeout	timeout
	16	32	2	414.33	**55.2(5)**	timeout	timeout
	4	4	1	38.92	43.0 (5)	12.92	43.0 (5)
	4	8	1	140.77	56.2 (5)	214.01	56.2 (5)
	4	8	2	61.09	52.4 (5)	91.01	52.4 (5)
	8	8	1	101.25	48.0 (5)	48.13	48.0 (5)
Configuration 7	8	16	1	372.09	**54.8 (4)**	818.10	**57.0 (3)**
	8	16	2	129.54	51.4 (5)	451.97	51.4 (5)
	16	16	1	270.70	**49.4 (5)**	378.01	**47.0 (4)**
	16	32	1	timeout	timeout	timeout	timeout
	16	32	2	328.81	**52.3 (4)**	timeout	timeout

7.2 Results For Experiment 2: Initial size for HMAPDC

Tables 7.5, 7.6, 7.7 and 7.8 report our results for **HMAPDC** with initial sizes 6x6, 8x8, 10x10, 12x12 for Level 0. Each table reports results on different grid sizes for the **MAPDC** instance. In the tables, *type* describes the type of obstacles, *none* stands for no obstacles, *random* for 10% random obstacles, *wh* for warehouse obstacles, *a* states the agent count. For this experiment task count is equal to agent count for all instances. Agent capacities are 1.

Figures 7.4, 7.5, 7.6 visualize the mentioned tables for different graph types. The line plots are for run times and the column plots are for makespans. The columns are also labeled with the number of instances solved in the given time of 2000 seconds.

One thing to note would be that the number of instances solved and the performance metrics should be considered simultaneously. For example consider Figure 7.5, grid size 96x96 and agent count 40. At first glance, 6x6 for first level size seems to find the best makespans. But it can only find answers for 3 instances out of 5, it is highly probable that those 3 instances are easier to solve, i.e., their optimal makespans may be low. The other first level sizes solve 4 or 5 instances. The other instances that they are able to solve may have higher optimal makespans, bringing the reported average makespan higher. The same is true for runtime.

For this reason, the most accurate conclusions that can be arrived from the plots presented in figures mentioned above are the ones where the number of solved instances are the same for different first level sizes.

An observation about these figures is that the most successful approach, especially with respect to the makespan, may fluctuate considerably as any parameter of the configurations change. For example, consider Figure 7.5, 72x72 grid. As agent count goes from 10 to 20, first level sizes 8x8 and 10x10 change place in terms of their makespan performance, while the other approaches more or less stay in their previous rank. We speculate this is because of the discrete and complicated nature of partioning a graph.

Having said this, it is possible to come to some conslusions from these figures. For example, first level size 12x12 consistently performs worse than other approaches in terms of runtime, and in many cases it can produce smaller makespan solutions. This is to be expected. Imagine increasing first level size indefinitely. One would get close to using **MAPDC-M**, which would find the optimum makespan but would take much longer to do so.

Another observation is that first level size 6x6 consistently finds worse makespans than other approaches in many cases. This should also be intuitive, since it abstracts the input graph the most, resulting in the biggest region sizes. As the region sizes grow, the solutions to abstract **HMAPDC** levels become more and more detached from the actual graph.

These results also pinpoint when **HMAPDC** fails to find a solution. Consider Table 7.5, type warehouse and agent count 60. **HMAPDC** mostly fails for all of the first level sizes we experimented with. In this configuration, there are both corridors and a large number of agents. Waiving collision constraints on higher levels constrains the bottom level in such a way that the agents cannot easily give way to each other and no solution can be found in the given time of 2000 seconds, or maybe a solution is impossible with the said constraints. We tackle this problem with our portfolio approach with Experiment 5.

Finally, we make the observation that first level sizes 8x8 and 10x10 perform well for many configurations we test on. This means there is a sweet spot for first level size, and that finding exactly which first level size to choose for an instance is not an easy problem for **HMAPDC**. More analysis on this can be done, but we are content with choosing 8x8, a general well-performer in many cases, to be our default first level size for further experiments.

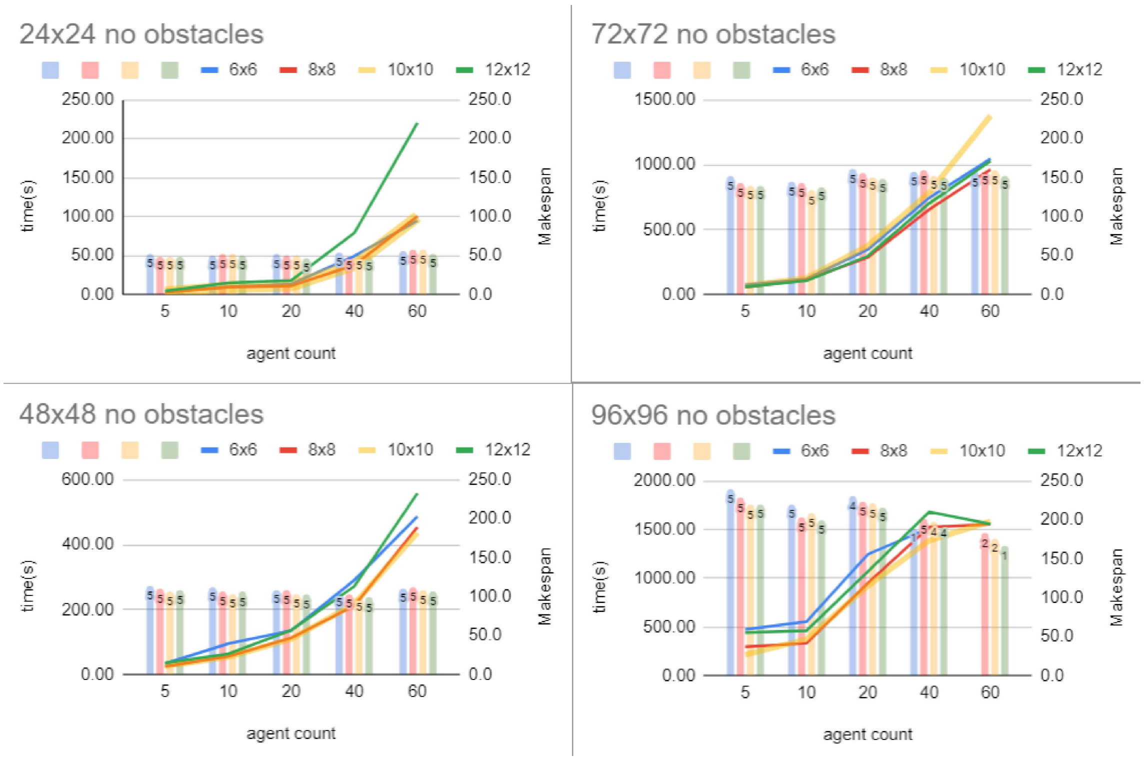


Figure 7.4 Experiment 2: Results for instances with no obstacles

Figure 7.7 shows computation time performance for the 3 graph types we use. It

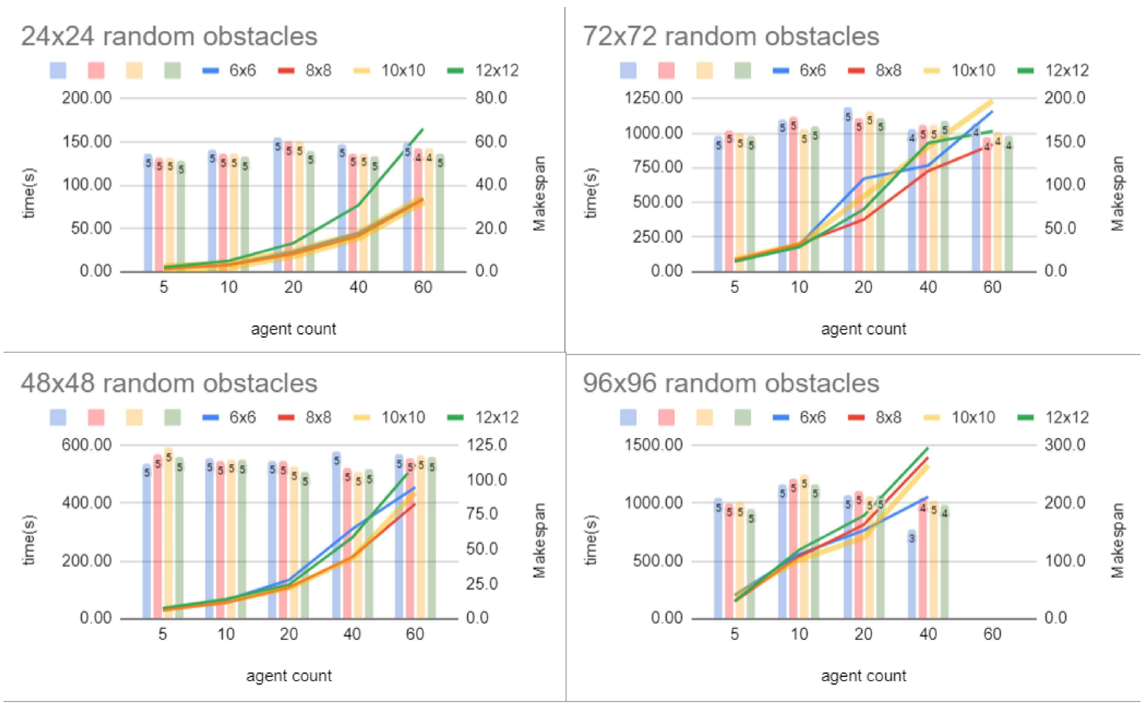


Figure 7.5 Experiment 2: Results for instances with 10% random obstacles

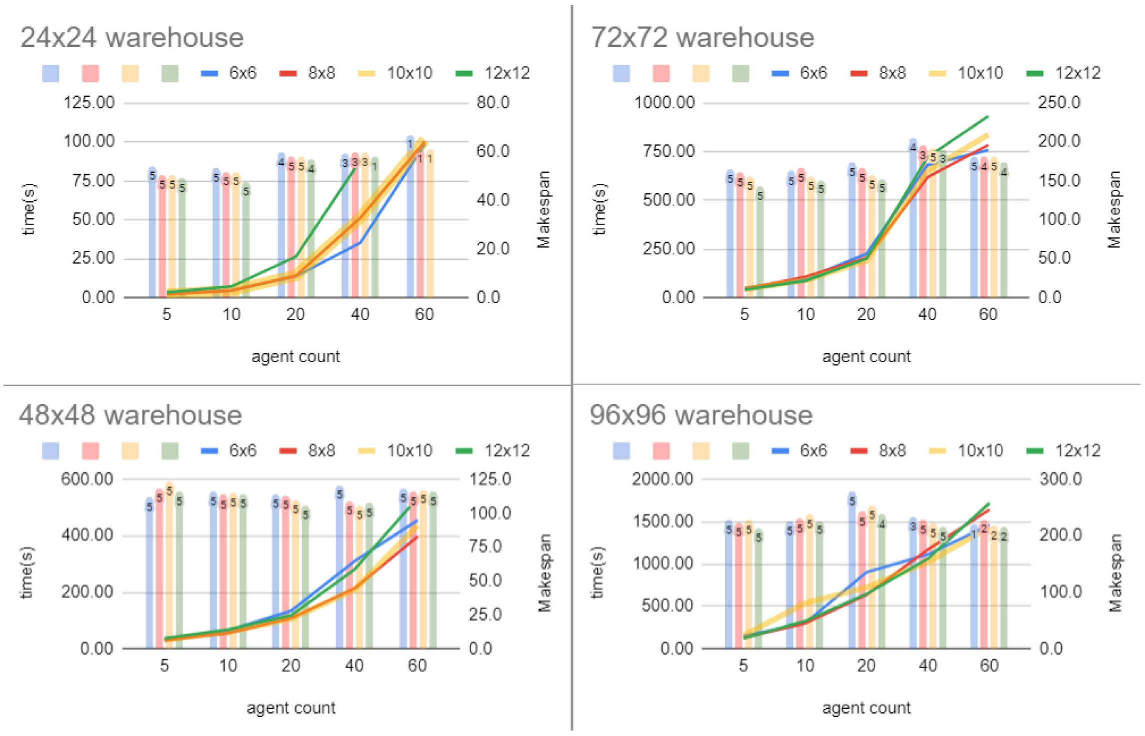


Figure 7.6 Experiment 2: Results for instances with warehouse setting

seems like graph type does not affect scalability.

Warehouse type consistently performs better as grid size grows, this is because there are a lot of obstacles in the grid. This means the graph we are working with is considerably smaller than the other types. Also, the average number of edges a

Table 7.5 Experiment 2: Results for grid size 24x24

		6x6		8x8		10x10		12x12	
type	a	time(s)	makespan	time(s)	makespan	time(s)	makespan	time(s)	makespan
none	5	4.52	52.2(5)	3.66	49.2(5)	3.60	49.2(5)	5.46	49.2(5)
	10	7.65	50.0(5)	7.36	51.8(5)	7.37	51.8(5)	12.21	50.6(5)
	20	18.17	51.4(5)	15.17	50.0(5)	15.22	50.0(5)	25.20	47.0(5)
	40	46.33	54.2(5)	36.69	49.2(5)	36.68	49.2(5)	73.39	48.2(5)
	60	87.14	55.0(5)	93.13	57.4(5)	93.11	57.4(5)	183.04	52.2(5)
random	5	4.04	54.6(5)	3.38	52.8(5)	3.37	52.8(5)	5.09	51.4(5)
	10	8.16	56.4(5)	7.58	54.4(5)	7.51	54.4(5)	12.56	53.2(5)
	20	23.45	62.2(5)	20.66	60.0(5)	20.71	60.0(5)	33.06	56.0(5)
	40	44.70	59.0(5)	41.31	54.8(5)	41.25	54.8(5)	76.85	53.2(5)
	60	84.28	59.4(5)	83.75	57.0(4)	83.69	57.0(4)	165.30	54.6(5)
wh	5	2.52	54.2(5)	2.20	50.6(5)	2.20	50.6(5)	3.39	49.2(5)
	10	4.77	53.4(5)	4.57	51.8(5)	4.64	51.8(5)	7.26	47.8(5)
	20	13.74	59.8(4)	14.15	58.0(5)	14.34	58.0(5)	26.33	57.0(4)
	40	35.22	59.3(3)	51.25	60.0(3)	51.41	60.0(3)	88.07	58.0(1)
	60	98.63	67.0(1)	100.20	61.0(1)	101.44	61.0(1)	0.00	0.0(0)

Table 7.6 Experiment 2: Results for Grid Size 48x48

		6x6		8x8		10x10		12x12	
type	a	time(s)	makespan	time(s)	makespan	time(s)	makespan	time(s)	makespan
none	5	33.87	114.0(5)	25.33	109.6(5)	26.19	107.4(5)	35.57	108.8(5)
	10	95.18	112.8(5)	56.16	106.2(5)	53.68	103.8(5)	63.95	105.8(5)
	20	135.70	109.0(5)	113.25	107.6(5)	109.52	103.2(5)	136.88	102.2(5)
	40	291.01	105.2(5)	212.23	103.4(5)	217.55	99.6(5)	271.88	98.6(5)
	60	487.46	110.4(5)	454.36	113.0(5)	437.35	108.4(5)	558.74	106.6(5)
random	5	32.15	112.2(5)	29.74	118.6(5)	32.32	123.4(5)	36.79	116.4(5)
	10	63.81	116.2(5)	54.83	113.8(5)	60.02	115.0(5)	67.83	114.4(5)
	20	134.56	114.2(5)	107.96	113.4(5)	105.92	109.8(5)	118.15	106.0(5)
	40	310.45	121.0(5)	213.47	109.0(5)	209.04	105.6(5)	280.92	107.6(5)
	60	455.85	118.6(5)	398.17	116.2(5)	437.62	117.8(5)	534.63	116.4(5)
wh	5	15.45	110.0(5)	14.51	107.2(5)	14.82	108.4(5)	16.87	102.4(5)
	10	35.84	115.4(5)	40.73	120.8(5)	40.26	118.4(5)	45.53	112.6(5)
	20	88.98	124.2(5)	67.64	111.4(5)	74.66	113.4(5)	89.72	108.2(4)
	40	219.49	135.3(3)	204.37	128.0(5)	224.16	126.5(4)	226.59	113.0(5)
	60	286.35	124.7(3)	283.37	114.5(4)	373.74	122.2(4)	461.19	115.8(4)

node has is lowest in the warehouse setting, resulting in a smaller search space for the traversals of agents. As graph size grows, random obstacle type again performs better than the no obstacle type, because of the same reasons.

Table 7.7 Experiment 2: Results for Grid Size 72x72

		6x6		8x8		10x10		12x12	
type	a	time(s)	makespan	time(s)	makespan	time(s)	makespan	time(s)	makespan
none	5	73.33	151.8(5)	55.96	143.6(5)	67.53	139.6(5)	58.81	139.6(5)
	10	120.23	145.0(5)	114.23	142.6(5)	126.61	133.4(5)	106.32	138.0(5)
	20	347.69	160.4(5)	287.29	155.0(5)	376.19	151.2(5)	296.72	148.8(5)
	40	745.72	157.8(5)	654.65	158.6(5)	785.64	152.6(5)	702.48	151.2(5)
	60	1045.38	156.8(5)	964.01	158.4(5)	1377.33	158.8(5)	1028.50	152.6(5)
random	5	76.91	156.2(5)	85.60	163.6(5)	87.01	159.6(5)	73.37	156.4(5)
	10	192.99	176.4(5)	200.96	179.6(5)	198.21	164.8(5)	178.37	167.8(5)
	20	671.62	189.4(5)	376.10	178.0(5)	543.68	185.2(5)	451.37	178.2(5)
	40	766.93	165.0(4)	727.63	169.6(5)	901.05	169.8(5)	929.40	174.8(5)
	60	1159.32	171.8(4)	918.77	155.0(4)	1235.43	161.8(4)	1013.92	157.2(4)
wh	5	48.90	165.4(5)	44.82	160.4(5)	44.18	155.8(5)	41.29	143.8(5)
	10	97.82	162.6(5)	108.57	166.8(5)	90.14	156.4(5)	87.29	151.2(5)
	20	226.23	174.0(5)	204.17	166.0(5)	190.33	157.8(5)	200.50	152.6(5)
	40	681.91	205.8(4)	619.28	195.0(3)	652.21	192.2(5)	721.83	191.0(3)
	60	758.36	181.4(5)	784.85	181.2(4)	838.35	181.6(5)	933.62	174.2(4)

Table 7.8 Experiment 2: Results for Grid Size 96x96

		6x6		8x8		10x10		12x12	
type	a	time(s)	makespan	time(s)	makespan	time(s)	makespan	time(s)	makespan
none	5	473.73	238.8(5)	297.62	228.0(5)	215.20	218.8(5)	442.42	219.8(5)
	10	555.21	219.8(5)	333.25	203.0(5)	374.63	208.4(5)	463.57	200.0(5)
	20	1247.28	230.5(4)	951.48	222.8(5)	922.90	220.6(5)	1070.13	215.6(5)
	40	1516.17	189.0(1)	1527.38	200.6(5)	1389.41	197.0(4)	1682.03	194.8(4)
	60	0.00	0.0(0)	1556.26	182.5(2)	1584.51	176.5(2)	1557.08	166.0(1)
random	5	204.09	209.2(5)	149.27	200.0(5)	195.06	201.6(5)	152.09	189.8(5)
	10	559.33	233.2(5)	538.39	242.0(5)	510.15	250.6(5)	595.72	232.6(5)
	20	765.67	214.8(5)	813.37	221.6(5)	701.10	211.4(5)	891.56	214.2(5)
	40	1055.19	154.7(3)	1399.23	209.2(4)	1329.12	204.2(5)	1481.46	198.2(4)
	60	0.00	0.0(0)	0.00	0.0(0)	0.00	0.0(0)	0.00	0.0(0)
wh	5	150.40	228.4(5)	130.16	223.4(5)	156.19	228.0(5)	123.26	213.2(5)
	10	301.55	226.0(5)	298.51	230.2(5)	536.73	238.2(5)	326.85	227.2(5)
	20	904.51	279.0(5)	636.70	243.2(5)	724.43	254.2(5)	647.57	238.2(4)
	40	1116.17	234.7(3)	1173.58	228.6(5)	1028.20	222.6(5)	1066.00	216.4(5)
	60	1461.84	221.0(1)	1645.06	229.5(2)	1435.26	218.5(2)	1723.15	215.5(2)

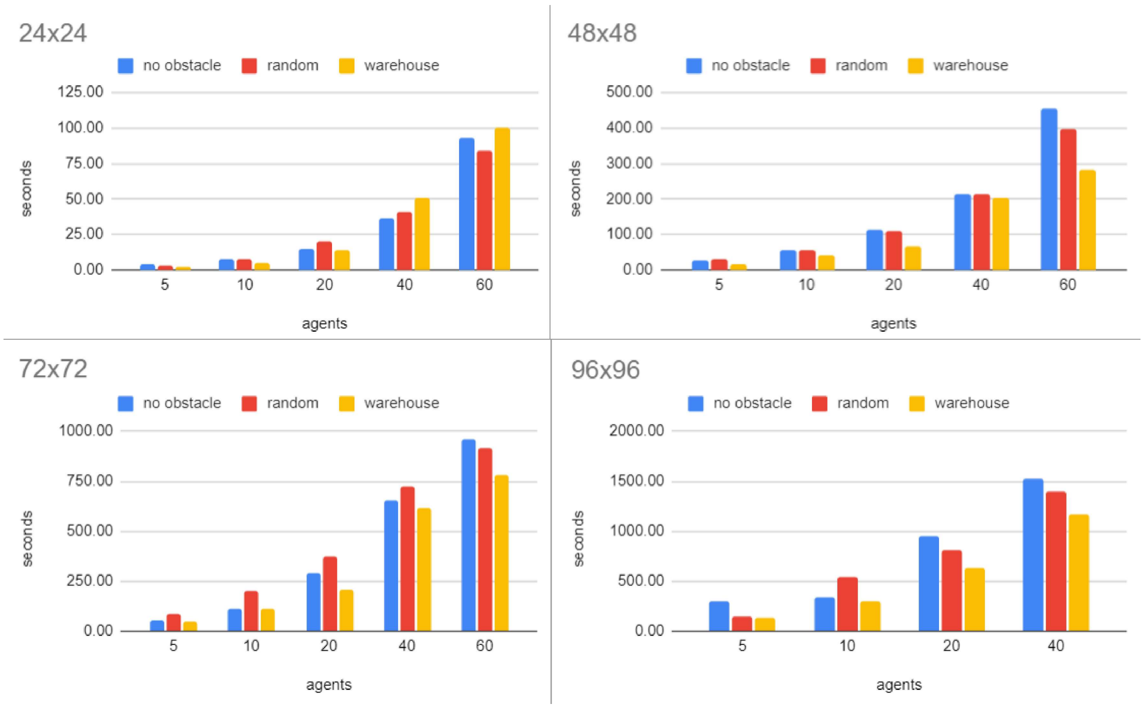


Figure 7.7 Experiment 2: Computation time by graph type for first level size 8x8

7.3 Results for Experiment 3: MAPDC-M vs HMAPDC

Table 7.9 shows the performance of our multi-shot approach vs **HMAPDC** for Benchmark 2 instances. The column *size* shows the grid size (all grids are square). Column *c* is for capacity. Since the metrics we mainly care about in this experiment are the speedup we gain by using **HMAPDC** and the sub-optimality **HMAPDC** causes, we only report results for instances that both of them can solve. Table 7.9 contains values for computation time, number of rules, and makespan, while 7.10 reports the said metrics as a summary of the former table. The metrics on the summary table are calculated as: speedup = multi-shot solving time / hmapdc solving time, suboptimality = **HMAPDC** makespan / multi-shot makespan, rules savings = multi-shot rules / **HMAPDC** rules.

We also do the same experiment for 2 graphs that have a very high percentage of obstacles, obtained from Bogatarkan et al. (2019). Both are 20x20 grids. We randomly generated 3 instances for each configuration in Tables 7.11, 7.12 which contain the actual values and their summary respectively. Figures 7.8 and 7.9 show the grids.

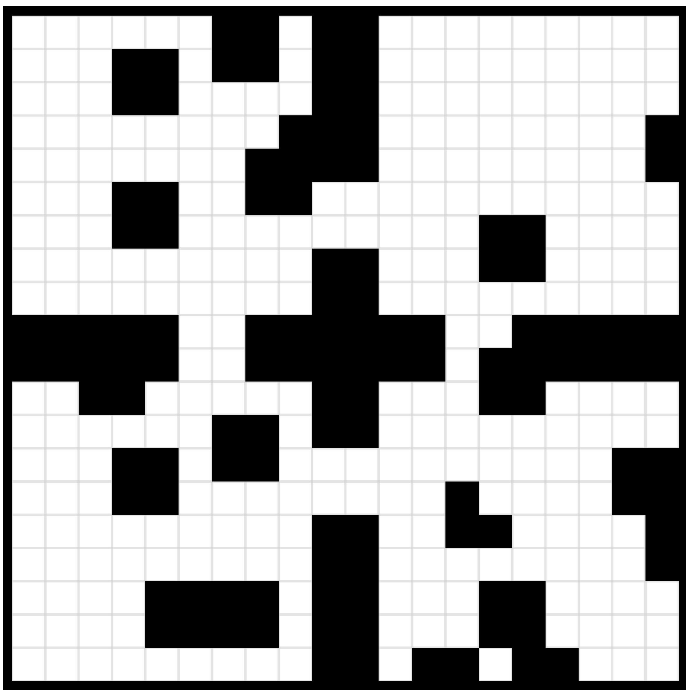


Figure 7.8 Experiment 3: Grid g1 (Bogatarkan et al., 2019)

The configurations that are not present in these tables are ones that the multi-shot approach failed to solve.

As can be seen from Table 7.9, **HMAPDC** is at least an order of magnitude faster,

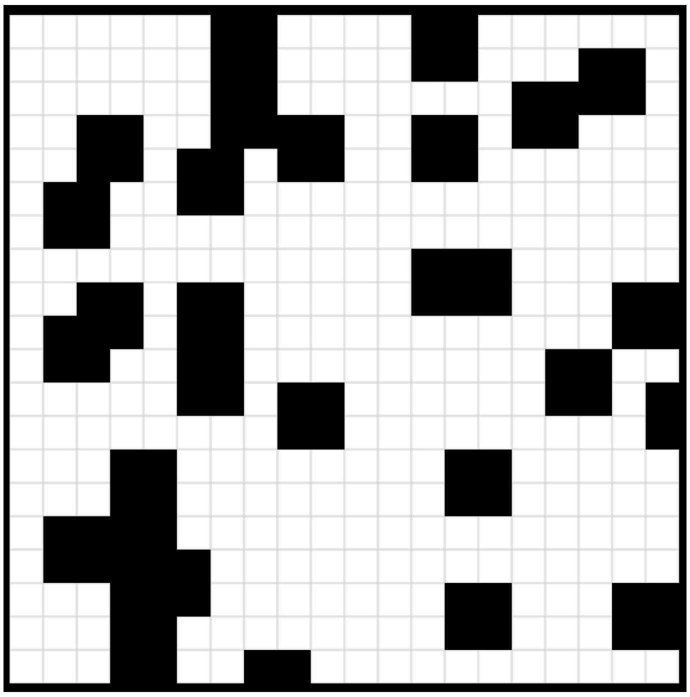


Figure 7.9 Experiment 3: Grid g2 (Bogatarkan et al., 2019)

and sometimes 2. This also holds for the graphs with a high percentage of obstacles as reported in Tables 7.11 and 7.12.

We observe up to 18% sub-optimality in Table 7.10. The sub-optimality is unpredictable and we did not recognize a pattern to it.

Table 7.9 Experiment 3: **MAPDC-M** vs **HMAPDC** on instances both can solve. All grids are square shaped.

grid	grid			Multishot			HMAPDC		
size	type	a	c	time(s)	rules	makespan	time(s)	rules	makespan
24	none	5	1	69.2	2.36E+07	44.63	2.7	7.84E+05	47.50
	none	5	2	170.0	3.09E+07	53.75	5.1	1.38E+06	55.75
	none	10	1	219.2	5.81E+07	50.00	7.3	2.05E+06	51.00
	none	10	2	542.4	7.78E+07	59.33	16.2	3.69E+06	65.00
24	random	5	1	68.1	2.19E+07	48.00	3.1	8.84E+05	50.80
	random	5	2	362.9	2.82E+07	57.00	6.5	1.56E+06	62.00
	random	10	1	199.1	5.84E+07	57.60	10.4	2.62E+06	60.80
	random	10	2	671.1	5.21E+07	53.50	13.7	3.07E+06	61.00
24	wh	5	1	44.8	1.12E+07	54.20	3.2	8.48E+05	58.20
	wh	5	2	744.0	1.62E+07	66.60	9.1	1.61E+06	74.60
	wh	10	1	89.3	2.48E+07	54.80	6.2	1.60E+06	59.00
	wh	10	2	1927.9	2.60E+07	55.00	9.6	1.92E+06	60.00
36	none	5	1	754.1	1.99E+08	73.80	10.6	2.66E+06	78.20
	none	5	2	1179.5	1.84E+08	73.00	14.1	3.46E+06	86.00
	none	10	1	1412.7	3.65E+08	71.00	17.6	4.41E+06	72.50
36	random	5	1	535.7	1.53E+08	74.00	10.4	2.50E+06	81.00
	random	5	2	1395.6	1.94E+08	83.67	21.4	4.37E+06	98.33
	random	10	1	1228.5	3.21E+08	75.33	21.9	5.12E+06	84.33
36	wh	5	1	240.0	7.49E+07	79.20	8.4	1.86E+06	90.00
	wh	5	2	452.3	8.58E+07	84.00	9.8	2.29E+06	86.00
	wh	10	1	957.8	1.74E+08	85.60	26.3	5.06E+06	99.40
48	none	5	1	928.1	3.10E+08	62.00	5.0	1.39E+06	62.00
48	random	5	1	1762.2	5.44E+08	88.50	13.9	3.44E+06	92.50
48	wh	5	1	1176.6	2.72E+08	101.75	16.6	3.74E+06	108.00
	wh	10	1	1750.9	5.41E+08	99.00	35.7	7.88E+06	117.00

Table 7.10 Experiment 3: Summary table for Table 7.9 that shows the advantages (and disadvantages) of **HMAPDC** against **MAPDC-M**

grid size	grid type	a	c	speedup	suboptimality	rules savings
24	none	5	1	25.6	1.064	30.06
	none	5	2	33.4	1.037	22.46
	none	10	1	29.9	1.020	28.36
	none	10	2	33.4	1.096	21.12
24	random	5	1	21.8	1.058	24.76
	random	5	2	56.1	1.088	18.10
	random	10	1	19.1	1.056	22.29
	random	10	2	48.9	1.140	16.95
24	wh	5	1	13.9	1.074	13.24
	wh	5	2	81.7	1.120	10.05
	wh	10	1	14.3	1.077	15.56
	wh	10	2	200.8	1.091	13.55
36	none	5	1	71.5	1.060	74.99
	none	5	2	83.7	1.178	53.13
	none	10	1	80.4	1.021	82.77
36	random	5	1	51.7	1.095	61.09
	random	5	2	65.3	1.175	44.49
	random	10	1	56.0	1.119	62.73
36	wh	5	1	28.6	1.136	40.35
	wh	5	2	46.3	1.024	37.42
	wh	10	1	36.4	1.161	34.46
38	none	5	1	186.0	1.000	223.94
48	random	5	1	126.4	1.045	158.15
48	wh	5	1	70.9	1.061	72.71
	wh	10	1	49.0	1.182	68.62

Table 7.11 **MAPDC-M** vs **HMAPDC** on high obstacle graphs from Bogatarkan et al. (2019)

			MAPDC-M			HMAPDC		
graph	a	c	time(s)	rules	makespan	time(s)	rules	makespan
g1	5	1	41.4	1.45E+07	47.3	2.9	8.05E+05	48.3
	5	2	50.4	1.34E+07	44.3	3.3	8.93E+05	48.3
	10	1	105.1	2.80E+07	44.3	6.3	1.70E+06	48.3
	10	2	144.4	2.74E+07	44.0	7.7	1.94E+06	49.3
	20	1	219.0	5.53E+07	41.7	10.9	2.62E+06	44.0
	20	2	771.7	6.71E+07	47.0	21.6	3.68E+06	49.3
g2	5	1	33.7	1.21E+07	43.3	2.3	6.74E+05	43.3
	5	2	208.3	1.34E+07	44.0	3.5	8.63E+05	47.3
	10	1	60.6	1.99E+07	36.0	3.7	1.05E+06	38.7
	10	2	174.6	2.69E+07	43.7	8.1	2.01E+06	51.7
	20	1	190.4	5.16E+07	40.3	9.1	2.33E+06	41.7
	20	2	470.7	6.02E+07	43.0	15.9	3.39E+06	46.0

Table 7.12 Summary table for Table 7.11 that shows the advantages (and disadvantages) of **HMAPDC** against **MAPDC-M**

graph	a	c	speedup	suboptimality	rules savings
g1	5	1	14.07	1.021	18.00
	5	2	15.44	1.090	15.02
	10	1	16.62	1.090	16.41
	10	2	18.74	1.121	14.11
	20	1	20.16	1.056	21.09
	20	2	35.74	1.050	18.25
g2	5	1	14.35	1.000	18.01
	5	2	60.08	1.076	15.50
	10	1	16.50	1.074	19.03
	10	2	21.49	1.183	13.33
	20	1	20.81	1.033	22.19
	20	2	29.55	1.070	17.75

7.4 Results for Experiment 4: Augmentations and Heuristics for HMAPDC

With this experiment, we aim to test our augmentations to **HMAPDC**. Tables 7.13, 7.14 and 7.15 report our findings for different grid sizes from Benchmark 3. For each configuration, we highlighted the best performing method in terms of computation time.

It can be seen through Tables 7.13, 7.14, 7.15 that usually, without any augmentation, **HMAPDC** performs the best. When another method performs better, it is not by a large margin. Consider Table 7.13, the last row. The biggest performance gain in the table is between h and *no heuristic* in this row, but even then it is only a speedup of around 5%. Also, the row before the last one shows *no heuristic* outperforming h with a better margin.

Our rationale of using al, the additional layer heuristic, was that the last level of **HMAPDC** where we solve **MAPDC-Bottom** is the one that takes the most time. It is no wonder, since it is the only layer with collision constraints. We claimed that if we add an **MAPDC-Middle** level before the last, we could help the last level since all it would need to would be to reschedule the agents on the paths dictated by this added layer. However, although **MAPDC-Middle** levels are easier to solve since there are no collision constraints, it is still computationally non-negligable. So, even though the last level is solved easier, the added layer adds enough computational cost to negate the benefits.

With the task assignment heuristic h, our rationale was that since we would be restricting the search space by enforcing the top level task assignments onto lower levels, in practice this did not help much. We speculate the reason for this is that once the paths of agents are restricted, the tasks they can take on is already restricted sufficiently. For an agent to be assigned a task, it needs to be able to visit both the pick and deliver location of a task. We actually observe that in some cases using h can be a bit detrimental to performance. By restricting the task assignments, we may also be discarding solutions with lower makespans.

Table 7.13 Experiment 4: Results for Grid Size 24x24

		al		h		al+h		no heuristic	
grid type	a	time(s)	makespan	time(s)	makespan	time(s)	makespan	time(s)	makespan
	5	3.80	49.6(5)	**3.64**	49.2(5)	3.74	49.6(5)	3.66	49.2(5)
	10	7.72	52.2(5)	**7.23**	51.8(5)	7.29	52.2(5)	7.36	51.8(5)
none	20	15.85	50.4(5)	18.11	50.0(5)	16.11	50.4(5)	**15.17**	50.0(5)
	40	39.81	49.4(5)	41.84	49.8(5)	40.85	50.2(5)	**36.69**	49.2(5)
	60	102.35	57.4(5)	101.95	57.6(5)	110.53	58.0(5)	**93.13**	57.4(5)
	5	3.51	52.8(5)	3.47	52.8(5)	3.69	53.2(5)	**3.38**	52.8(5)
	10	8.04	54.8(5)	7.77	54.2(5)	8.29	54.8(5)	**7.58**	54.4(5)
random	20	21.03	59.2(5)	21.28	60.0(5)	21.58	59.5(4)	**20.66**	60.0(5)
	40	**39.37**	52.7(3)	41.26	54.4(5)	45.37	54.5(2)	41.31	54.8(5)
	60	90.04	55.7(3)	90.64	58.5(4)	100.34	58.3(3)	**83.75**	57.0(4)
	5	2.67	51.0(5)	**2.19**	50.6(5)	2.62	51.0(5)	2.20	50.6(5)
	10	5.53	52.2(5)	4.69	52.4(5)	5.61	52.8(5)	**4.57**	51.8(5)
wh	20	16.63	58.4(5)	**13.86**	58.0(5)	16.37	58.2(5)	14.15	58.0(5)
	40	57.89	60.7(3)	59.10	63.0(2)	67.73	63.5(2)	**51.25**	60.0(3)
	60	108.74	61.0(1)	**94.05**	61.0(1)	102.98	61.0(1)	100.20	61.0(1)

Table 7.14 Experiment 4: Results for Grid Size 48x48

		al		h		al+h		no heuristic	
grid type	a	time(s)	makespan	time(s)	makespan	time(s)	makespan	time(s)	makespan
	5	25.71	109.8(5)	26.43	109.4(5)	26.98	109.8(5)	**25.33**	109.6(5)
	10	56.47	106.6(5)	57.09	107.6(5)	57.62	108.0(5)	**56.16**	106.2(5)
none	20	**111.74**	108.0(5)	115.24	108.0(5)	116.49	108.4(5)	113.25	107.6(5)
	40	234.33	103.8(5)	235.65	104.0(5)	245.20	104.4(5)	**212.23**	103.4(5)
	60	465.45	114.0(5)	543.92	113.6(5)	544.93	114.0(5)	**454.36**	113.0(5)
	5	**28.98**	118.6(5)	29.43	116.2(5)	29.37	116.6(5)	29.74	118.6(5)
	10	57.98	114.4(5)	55.22	113.0(5)	57.10	113.4(5)	**54.83**	113.8(5)
random	20	120.26	116.2(5)	118.72	113.4(5)	122.29	113.8(5)	**107.96**	113.4(5)
	40	217.54	108.2(5)	242.63	110.2(5)	261.57	110.6(5)	**213.47**	109.0(5)
	60	433.04	117.2(5)	532.46	121.0(5)	576.03	123.0(4)	**398.17**	116.2(5)
	5	16.57	106.8(5)	14.62	106.4(5)	15.96	106.8(5)	**14.51**	107.2(5)
	10	46.24	121.6(5)	**40.15**	121.2(5)	45.83	121.6(5)	40.73	120.8(5)
wh	20	79.66	112.2(5)	73.54	114.2(5)	87.93	114.6(5)	**67.64**	111.4(5)
	40	251.95	128.4(5)	237.81	127.4(5)	286.16	127.8(5)	**204.37**	128.0(5)
	60	334.32	115.0(4)	356.14	117.5(4)	440.72	118.0(4)	**283.37**	114.5(4)

Table 7.15 Experiment 4: Results for Grid Size 72x72

		al		h		al+h		no heuristic	
grid type	a	time(s)	makespan	time(s)	makespan	time(s)	makespan	time(s)	makespan
	5	59.17	144.4(5)	57.83	143.6(5)	58.58	144.0(5)	**55.96**	143.6(5)
	10	116.96	143.0(5)	124.01	145.0(5)	123.09	145.4(5)	**114.23**	142.6(5)
none	20	292.41	155.4(5)	337.37	158.6(5)	340.80	159.0(5)	**287.29**	155.0(5)
	40	703.63	160.0(5)	693.93	159.0(5)	722.17	159.4(5)	**654.65**	158.6(5)
	60	1009.98	158.0(5)	1209.30	156.5(4)	1305.58	163.4(5)	**964.01**	158.4(5)
	5	**83.13**	164.4(5)	85.65	163.8(5)	84.36	164.2(5)	85.60	163.6(5)
	10	205.36	181.0(5)	216.23	184.4(5)	219.79	184.8(5)	**200.96**	179.6(5)
random	20	394.41	179.6(5)	415.63	177.0(5)	419.27	177.4(5)	**376.10**	178.0(5)
	40	774.00	170.8(5)	869.36	172.2(5)	850.26	172.6(5)	**727.63**	169.6(5)
	60	963.49	155.5(4)	1392.62	160.5(4)	1354.59	161.0(4)	**918.77**	155.0(4)
	5	46.60	159.6(5)	**42.00**	159.6(5)	46.25	160.0(5)	44.82	160.4(5)
	10	121.89	167.0(5)	109.41	167.2(5)	120.19	167.6(5)	**108.57**	166.8(5)
wh	20	241.11	167.8(4)	**193.95**	160.2(5)	219.67	160.6(5)	204.17	166.0(5)
	40	740.91	196.7(3)	808.54	204.0(3)	970.74	204.3(3)	**619.28**	195.0(3)
	60	950.94	181.8(4)	1139.31	183.0(3)	1328.63	183.7(3)	**784.85**	181.2(4)

7.5 Results for Experiment 5: Starting with a Portfolio of Task-Diverse Solutions

Table 7.16 shows the results for starting with $p = 4$ solutions (at the top level) where no two of these solutions can be more than $r = 0.75$ task-similar. a time threshold of 1000 seconds, to solve 1 instance of each configuration with grid size 72x72 in Benchmark 3. We report the first solution as well as the ones with the best computation time and with the best makespan.

It can be seen that , when we start with a portfolio of 4 diverse solutions at the top level, we can speed up the computation by up to 30% for the best case. It is also possible to find solutions with meagerly better makespans. It also shows, however, that using a portfolio does not result in consistent performance gains. Only 3 out of 15 configurations resulted in more than 10% speedup, and for many of the configurations the first solution is nearly as good as the best ones.

Table 7.16 Experiment 5: Starting with a portfolio of diverse solutions, for 72x72 instances, $r = 0.75, p = 4$

grid		first		best time			best makespan		
type	a	time(s)	makespan	time(s)	makespan	speedup	time(s)	makespan	makespanup
none	5	51.49	148	51.49	148	1.00	51.49	148.00	1.00
	10	99.08	129	87.09	131	1.14	99.08	129.00	1.00
	20	313.42	158	313.42	158	1.00	313.42	158.00	1.00
	40	524.44	149	498.47	149	1.05	524.44	149.00	1.00
	60	861.25	152	843.39	146	1.02	843.39	146.00	1.04
random	5	67.45	168	65.48	168	1.03	69.06	164.00	1.02
	10	189.03	180	189.03	180	1.00	203.14	173.00	1.04
	20	400.57	183	307.16	164	1.30	307.16	164.00	1.12
	40	966.91	189	948.26	187	1.02	948.26	187.00	1.01
	60	744.48	146	744.48	146	1.00	744.48	146.00	1.00
wh	5	34.22	141	34.22	141	1.00	34.22	141.00	1.00
	10	67.96	148	57.06	140	1.19	57.06	140.00	1.06
	20	225.69	173	219.69	161	1.03	219.69	161.00	1.07
	40	-	-	-	-	-	-	-	-
	60	776.14	176	763.86	177	1.02	776.14	176.00	1.00

To demonstrate how using a portfolio of top level solutions helps with incompleteness, as discussed in Section 5.7, we decided to try this approach on the most problematic configurations from Benchmark 3. Consider Table 7.5, type warehouse and agent count 60. Level 0 sizes 6x6, 8x8, and 10x10 could only solve 1 out of 5 instances, while 12x12 failed in all instances in the given 2000 seconds. But, when an instance of this configuration could be solved, it took around 100 seconds to do so. Consider the result for Level 0 size 8x8, 1 instance was solved in 100 seconds. This result points to the incompleteness of **HMAPDC**, for all 4 other instances of the same configuration, bad first level solutions restricted the lower levels such that a solution was impossible to find (as in Figure 5.13). If the first level solutions were slightly different, this could have been prevented. Thus, we test our portfolio method to see whether we can solve these instances by utilizing a diverse set of first level solutions. We applied our portfolio method with $r = 0.9$ and $p = 10$, first level size 8x8, with a time limit of 250 seconds, to all 5 instances of this configuration. Table 7.17 contains our results. The columns 0-4 are different instances of the same configuration. The rows 0-9 are different runs of the portfolio. It can be seen that, with the portfolio approach, we could solve 4/5 instances compared to 1/5, which was the case in Table 7.5, warehouse, 60 agents, first level size 8x8.

Table 7.17 Experiment 5: Starting with a portfolio of diverse solutions ($r = 0.9, p = 10$), for 24x24, warehouse, 60 agents, Benchmark 3 instances

i	0		1		2		3		4	
p	time(s)	makespan	time(s)	makespan	time(s)	makespan	time(s)	makespan	time(s)	makespan
0	-	-	-	-	-	-	102.52	61	-	-
1	84.53	61	-	-	-	-	-	-	-	-
2	-	-	-	-	-	-	-	-	-	-
3	92.6	64	-	-	-	-	-	-	-	-
4	91.31	58	-	-	-	-	77.07	56	-	-
5	-	-	-	-	-	-	-	-	-	-
6	-	-	-	-	-	-	-	-	-	-
7	-	-	76.65	57	-	-	-	-	-	-
8	-	-	76.68	56	-	-	-	-	69.14	59
9	106.37	64	72.66	56	-	-	-	-	-	-

Consider the instance from Table 7.16 that our portfolio approach could not solve with $r = 0.75, p = 4$. Since we could not find an answer with these settings, we tried to solve it with $r = 0.9, p = 10$. Table 7.18 reports the results for each run in the portfolio. We were able to find a solution when we increased the size of the portfolio. We intuitively increase r when we increase the portfolio size. This is because the difference in task assignments we enforce is between all pairs of solutions we work with, enforcing a high difference ratio (low r), could misguide the algorithm to find bogus task assignments to be able to satisfy the difference constraints.

Table 7.18 Experiment 5: Starting with a portfolio of diverse solutions ($r = 0.9, p = 10$), results for the instance from Table 7.16 with grid type warehouse, a = 40

p	time(s)	makespan
0	-	-
1	-	-
2	-	-
3	-	-
4	-	-
5	762.17	199
6	-	-
7	718.27	203
8	-	-
9	-	-

Since it seems like our portfolio approach is more successful with a higher portfolio size, we re-performed the test for Benchmark 3, 72x72 instances, the same way we described for the results in Table 7.16 with $p = 10$ and $r = 0.9$. Results for this experiment are given in Table 7.19. It can be seen that this time all the instances were solved, and we have a speedup for more of the instances. This shows that our portfolio approach is best used with a big portfolio size.

Table 7.19 Experiment 5: Starting with a portfolio of diverse solutions, for 72x72 instances, $r = 0.9, p = 10$

grid		first		best time			best makespan	
type	a	time	makespan	time	makespan	speedup	time	makespan
n	5	51.67	148	51.67	148	1.00	51.67	148
	10	98.93	129	72.84	129	1.36	98.93	129
	20	311.21	158	263.48	149	1.18	263.48	149
	40	521.4	149	495.87	149	1.05	542.43	148
	60	860.37	152	844.83	146	1.02	844.83	146
r	5	67.29	168	54.98	152	1.22	54.98	152
	10	189.32	180	185.3	180	1.02	203.38	173
	20	402.56	183	274.09	158	1.47	274.09	158
	40	-	-	903.25	183	-	903.25	183
	60	743.39	146	729.9	146	1.02	749.02	142
w	5	33.94	141	33.94	141	1.00	33.94	141
	10	68.13	148	55.16	140	1.24	55.95	138
	20	224.01	173	164.08	153	1.37	164.08	153
	40	-	-	717.63	203	-	762.38	199
	60	772.86	176	730.09	178	1.06	789.46	172

There is one downside to increasing the portfolio size. We say that runs in a portfolio can be easily parallelized, but for a new run of **HMAPDC-P** to start, a new first level solution must be generated. Even though the first level is the easiest one to solve because of its small size and high level of abstraction, it still takes time to compute diverse solutions. This effect is more apparent in instances with high number of agents and tasks. We can calculate how long all the previous first level solutions takes until we come to a specific run, and add that overhead time to the time of each run. This way, we can obtain the time passed from the moment our portfolio approach starts and the moment we obtain the first solution. Table 7.20 reports our results in such a way. Note that some small speed-ups are lost since they were not significant enough to account for the overhead time. For example, consider the last row of Table 7.20. Before adjusting for the mentioned overhead, we have a speedup of 6% for some top-level solution in the portfolio, but after adjusting for the time it takes to start the **HMAPDC-P** run for that solution, the first solution found before diversifying became the best performer and we lost the speedup. It can still be seen that the trend of improvement continues after adjusting for the overhead.

Table 7.20 Experiment 5: Starting with a portfolio of diverse solutions, for Benchmark 3, 72x72 size instances with first level size = 8x8, $p = 10$, $r = 0.9$, adjusted for first level costs

grid		first	best time		adjusted best time	
type	a	time(s)	time(s)	speedup	time(s)	speedup
none	5	51.67	51.67	1.00	51.71	1.00
	10	98.93	72.84	1.36	73.07	1.35
	20	311.21	263.48	1.18	266.38	1.17
	40	521.4	495.87	1.05	505.95	1.03
	60	860.37	844.83	1.02	860.37	1.00
random	5	67.29	54.98	1.22	54.99	1.22
	10	189.32	185.3	1.02	186.21	1.02
	20	402.56	274.09	1.47	277.28	1.45
	40	-	903.25	-	968.2	-
	60	743.39	729.9	1.02	743.39	1.00
wh	5	33.94	33.94	1.00	33.96	1.00
	10	68.13	55.16	1.24	55.42	1.23
	20	224.01	164.08	1.37	176.81	1.27
	40	-	717.63	-	764.6	-
	60	772.86	730.09	1.06	772.86	1.00

8. Discussion

Our results demonstrate the scalability of our approaches. Even though we have two optimal and complete methods, one single-shot **MAPDC-ASP** and one multi-shot **MAPDC-M**, practically we only use the multi-shot one since it is more useful when we are trying to find the optimal makespan for a given **MAPDC** instance. **MAPDC-M** is an optimal and complete method that scales poorly both with increasing number of agents and increasing grid sizes. Our sub-optimal and incomplete approach **HMAPDC** scales much better, and it can be used as a portfolio to partially compensate for its incompleteness and sub-optimality.

HMAPDC is most incomplete when there are narrow corridors in the environment and the number of agents are high. But, if the high level solution allows it, **HMAPDC** can still successfully solve the instance in a reasonable time.

Our proposed augmentations to **HMAPDC** cannot reliably improve performance, and sometimes are detrimental to it.

The portfolio approach sometimes can improve performance, but its usefulness is most apparent in the improvement on the success rate for configurations with a high rate of failure.

We discuss some other possible augmentations to **HMAPDC** that we left for future work.

Even though **HMAPDC** is designed to be fast, and is much faster than our multi-shot method **MAPDC-M**, its unpredictable incompleteness is something we would like to get rid of if possible.

One augmentation could be to add not all but some of the collision constraints we waive for the higher abstract levels. For example, we cannot add the vertex collision constraint since many agents can be present in a region, but we can add the swapping constraint. This way, agents may be inclined to give way to each other in cases with narrow corridors in higher levels, which is one of the reasons our algorithm is incomplete as was demonstrated in Figure 5.13.

This could be possible by adding the following lines to the `step(t)` subprograms of ASP programs $\mathcal{P}_{top}$ and $\mathcal{P}_{mid}$, to forbid agents from swapping their positions in 1 or 2 time steps. Forbidding swapping in two time-steps is also necessary since otherwise agents could abuse the waiving of vertex collision constraint to swap their positions.

```
:- traversal(A1,t,V1), traversal(A2,t,V2), traversal(A1,t-1,V2),
        traversal(A2,t-1,V1), A1 != A2, V1 < V2.
:- traversal(A1,t,V1), traversal(A2,t,V2), traversal(A1,t-2,V2),
        traversal(A2,t-2,V1), A1 != A2, V1 < V2.
```

But consider Table 7.5, warehouse setting with 60 agents, where **HMAPDC** with different first level sizes mostly failed to find solutions in the given time. In practice, **HMAPDC** will lose success rate when there are narrow corridors and high number of agents. But introducing these constraints at higher levels for instances with high number of agents, even when the grid size is managable, will increase solving time in such a drastic way that even the highest abstract levels will not be solved in a timely manner. We tested this for one of the unsolved instances of said configuration with 1000 seconds time limit, first level size 8x8, and even the first level could not be solved in the given time. We believe adding other kinds of constraints would be detrimental, too, when the agent count is high.

Another approach to reduce the incompleteness could be to restrict the number of agents in regions of abstract levels. Technically, since only the paths and not the traversals of agents affect lower level solutions, this could be circumvented by the algorithm by rescheduling the agents on their same paths, resulting in no difference in the solutions. But maybe, in practice, it may result in higher level solutions that have the agents give way to each other. This kind of restriction on the number of agents in regions can be implemented by adding the following rules to the `step(t)` subprogram of $\mathcal{P}_{top}$ and $\mathcal{P}_{mid}$:

```
:- L+1{agent(A):traversal(A,t,V)}, rlim(V,L), node(V).
```

Where "`rlim(V,L).`" should be added to the level facts with respect to the number of vertices in a region ($|region.vertices|$).

We were not able to rigorously test this, but as with the previous suggestion of adding some constraints on higher levels, we tested on the same instance that could not be solved by **HMAPDC** with first level size 8x8. It still could not find a solution in 1000 seconds.

Note that for both of the cases, our portfolio approach TAP was able to produce solutions in around 100 seconds.

We believe this means, together with the relative failure of the augmentation *al* and the heuristic *h* we tested for, that it is hard to improve **HMAPDC**. Our portfolio approach is the only approach that at least showed relative promise of improving the performance of **HMAPDC**, but it can frequently help with its incompleteness for large portfolio sizes.

It is possible to introduce other ways to use a portfolio of top-level solutions. Introducing a new similarity measure would be a possibility. In our experience, measuring the similarity of solutions **MAPDC** instances via agent paths proved to be a challenging problem, but we believe it may be promising.

One other way to expand on **HMAPDC** could be to analyze the high level solutions and pinpoint the regions that could be problematic in the sense that agents would need to swap locations, or there is heavy traffic through the region. Then, neighbouring regions may be added to the *valids* of agents for them to give way to each other more easily.

9. Related Work

There are a plethora of studies on **MAPF**, and its variants that focus on warehouse environments. The problem is approached with a variety of methods, including search based (Hönig et al., 2018), ASP based (Erdem et al., 2013; Gómez et al., 2021), satisfiability based (Surynek, 2012), and integer linear programming (Yu & LaValle, 2016) approaches.

Some studies on **MAPF** utilize spatial hierarchies like we do. Kazemy (2018) was the inspiration behind our **HMAPDC** method. They are concerned with solving a generalized version of **MAPF** which also has waypoints for agents, and take an input grid and create hierarchical levels for the grid. They solve their version of **MAPF** by solving each level and restricting agents into regions implied by previous levels as we do. Their hierarchical method is designed for robotics applications where some cells on the grid may have partial obstacles. They utilize replanning to compansate for infeasible solutions that may arise from an agent trying to pass through a cell with partial obstacles. In general, while utilizing hierarchical levels in a similar way to us, this study is more interested in the robotics applications of their approach, while we are focused on a more abstract computational problem. Also, the problem we work on has an additional combinatorial element to it with the addition of tasks that need to be allocated to agents. We also differ in the manner we create abstract graphs, where we have a top-down approach (start with a desired first level size and continue creating finer levels until original grid is reached), and they have a bottom-up approach (continue abstracting the graph until a level with sufficient coarseness is reached). They also use ASP to solve their hierarchical levels, utilizing both multi-shot and single-shot programs in their algorithm, unlike **HMAPDC** which only uses multi-shot programs.

Husár et al. (2022) studies improvements on reduction based approaches to **MAPF**, like our ASP based methods. As we restrict the traversals of agents with respect to a spatial hierarchy, they restrict the paths of agents in similar ways, e.g., to their shortest path to their goals. Unsurprisingly, complying with our experience, their methods that guarantee optimality and completeness have limited use, while methods that waive these guarantees result in considerable speed-ups.

Morag et al. aims to approach **MAPF** practically, i.e., cleverly using fast, easy to implement, albeit sub-optimal, methods with several attempts to reduce the average sub-optimality and incompleteness. This was the inspiration behind our portfolio method, as it is computationally cheap to generate first level solutions. They use *prioritized planning* as a simple method and modify it slightly to utilize random restarts. They report that their approach is promising when compared to optimal ones such as CBS (Ma & Koenig, 2016).

Like us, a number of studies focus on warehouse environments. TAPF (Ma & Koenig, 2016) and later GTAPF (Nguyen et al., 2019) are introduced as problems that fits this environment via the addition of targets that must be reached by any agent. Ma & Koenig (2016) introduces a conflict based search (CBS) method to optimally solve TAPF, where the number of targets must be equal to the number of agents, and Nguyen et al. (2019) provides an ASP based optimal solution to the more generalized GTAPF, where agents need to complete possibly multi-step tasks.

Gómez et al. (2021) introduces an ASP encoding for **MAPF**, improving on other ASP based methods (such as Erdem et al. (2013)) by utilizing search to reduce the grounding size of the programs and reducing the grounding complexity of the ASP programs. Their search based improvement restrict the generation rule for agent paths so that useless paths are not explored; if the shortest path of an agent from its current location to its goal location is longer than the times-steps it has left to reach its goal, we should not explore the paths that could be generated from there. They also reduce the grounding complexity so that the number of collision constraints generated depends linearly on the number of agents, instead of quadratically in our case. Both improvements can be considered for our encodings in future work, though the implementation may require tinkering since they approach **MAPF** as a planning problem instead of a graph problem like we do.

Hönig et al. (2018) works on an unnamed **MAPF** variant, where shelves need to be moved to goal stations by automated agents in an automated warehouse environment. They present a CBS based optimal method in terms of total plan lengths, and a bounded suboptimal method called ECBS. Since their formulation is made for moving underneath shelves and carrying them to goal stations, they do not consider capacities for agents unlike us. Their suboptimal ECBS method also is bounded suboptimal, unlike our **HMAPDC** method.

Ma et al. (2017) introduces Multi-agent Pick and Delivery (MAPD) problem, which is a similar problem to **MAPDC** but in a setting where pick-and-deliver tasks are introduced to the environment dynamically. They provide token based solutions to the problem which are not optimal but produce solutions near-real-time. It should also be noted that in MAPD, agents do not have goal locations, which is a considerable difference from our approach.

Liu et al. (2019) study MAPD problems by first assigning tasks to agents, and then use a search-based path finding algorithm to compute collision-free paths, while considering the task assignment and minimizing the makespan.

Chen et al. (2021) introduces capacities to MAPD and they propose a coupled method that searches for the best assignment of tasks to agents. Chen et al. (2021) is the only other method with capacities to our knowledge. They also optimize for *total travel delay*, which is the sum of plan lengths for agents, instead of optimizing for makespan like we do.

10. Conclusion

We have mathematically modelled **MAPDC** problem and introduced novel methods to solve it using ASP. These methods are **MAPDC-ASP**, **MAPDC-M**, and **HMAPDC**. We further proposed augmentations to **HMAPDC**, one of which is our portfolio method, which we believe to be promising.

We analyzed the scalability of our methods in a variety of experiments. Because of the theoretical hardness of the problem, our complete and optimal methods **MAPDC-ASP** and **MAPDC-M** scale poorly as the problem size grows. **HMAPDC** scales to much bigger graphs and agent numbers, but is incomplete and suboptimal. Our portfolio approach can be utilized to alleviate the incompleteness to a degree.

HMAPDC depends on a graph partitioning scheme. We detail the implementation of such a scheme for grid graphs, and tested what Level 0 size we should choose. To our understanding, Level 0 sizes 8x8 and 10x10 perform best on most cases, but for even bigger grid sizes than we tested for, 12x12 might be considered as well.

We investigated whether our proposed augmentations to **HMAPDC** were beneficial, and concluded that only the portfolio approach is promising for the most part.

We left the comparison of our methods with other approaches as future work. Although there are no studies that work on the exact same problem as us (**MAPDC**), there are similar problems such as **MAPD** which we can make a comparison with by changing our formulations to not include goal locations for agents.

We also discussed how **HMAPDC** can be improved further and left the suggested improvements as future work.

BIBLIOGRAPHY

Bogatarkan, A., Erdem, E., Kleiner, A., & Patoglu, V. (2020). Multi-modal multi-agent path finding with optimal resource utilization. In *Proceedings of 5th International Conference on the Industry 4.0 Model for Advanced Manufacturing*, (pp. 313–324). Springer.

Bogatarkan, A., Patoglu, V., & Erdem, E. (2019). A declarative method for dynamic multi-agent path finding. In *GCAI*, (pp. 54–67).

Brewka, G., Eiter, T., & Truszczynski, M. (2016). Answer set programming: An introduction to the special issue. *AI Magazine*, *37*(3), 5–6.

Chen, Z., Alonso-Mora, J., Bai, X., Harabor, D. D., & Stuckey, P. J. (2021). Integrated task assignment and path planning for capacitated multi-agent pickup and delivery. *IEEE Robotics and Automation Letters*, *6*(3), 5816–5823.

Eiter, T., Erdem, E., Erdoğan, H., & Fink, M. (2009). Finding similar or diverse solutions in answer set programming. In *International Conference on Logic Programming*, (pp. 342–356). Springer.

Erdem, E., Kisa, D. G., Oztok, U., & Schüller, P. (2013). A general formal framework for pathfinding problems with multiple agents. In *Twenty-Seventh AAAI Conference on Artificial Intelligence*.

Gebser, M., Kaminski, R., Kaufmann, B., & Schaub, T. (2019). Multi-shot asp solving with clingo. *Theory and Practice of Logic Programming*, *19*(1), 27–82.

Gelfond, M. & Lifschitz, V. (2000). The stable model semantics for logic programming. *Logic Programming*, *2*.

Gómez, R. N., Hernández, C., & Baier, J. A. (2021). A compact answer set programming encoding of multi-agent pathfinding. *IEEE Access*, *9*, 26886–26901.

Guthrie, C., Fosso-Wamba, S., & Arnaud, J. B. (2021). Online consumer resilience during a pandemic: An exploratory study of e-commerce behavior before, during and after a covid-19 lockdown. *Journal of Retailing and Consumer Services*, *61*, 102570.

Hönig, W., Kiesel, S., Tinka, A., Durham, J., & Ayanian, N. (2018). Conflict-based search with optimal task assignment. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems*.

Husár, M., Svancara, J., Obermeier, P., Barták, R., & Schaub, T. (2022). Reduction-based solving of multi-agent pathfinding on large maps using graph pruning. In *AAMAS*, (pp. 624–632).

Kazemy, O. K. (2018). *Hybrid generalized multi-agent pathfinding: a hierarchical method using ASP*. PhD thesis.

Lifschitz, V. (2002). Answer set programming and plan generation. *Artificial Intelligence*, *138*(1-2), 39–54.

Lifschitz, V. (2019). *Answer set programming.* Springer Heidelberg.

Liu, M., Ma, H., Li, J., & Koenig, S. (2019). Task and path planning for multi-agent pickup and delivery. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS).*

Ma, H. & Koenig, S. (2016). Optimal target assignment and path finding for teams of agents. *arXiv preprint arXiv:1612.05693.*

Ma, H., Li, J., Kumar, T., & Koenig, S. (2017). Lifelong multi-agent path finding for online pickup and delivery tasks. *arXiv preprint arXiv:1705.10868.*

Marek, V. W. & Truszczyński, M. (1999). Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm* (pp. 375–398). Springer.

Morag, J., Felner, A., Stern, R., Atzmon, D., Boyarski, E., Louis, S., & Toledano, M. A practical approach to multi-agent path finding in robotic warehouses.

Nguyen, V., Obermeier, P., Son, T. C., Schaub, T., & Yeoh, W. (2019). Generalized target assignment and path finding using answer set programming. In *Twelfth Annual Symposium on Combinatorial Search.*

Stern, R., Sturtevant, N. R., Felner, A., Koenig, S., Ma, H., Walker, T. T., Li, J., Atzmon, D., Cohen, L., Kumar, T. S., et al. (2019). Multi-agent pathfinding: Definitions, variants, and benchmarks. In *Twelfth Annual Symposium on Combinatorial Search.*

Surynek, P. (2010). An optimization variant of multi-robot path planning is intractable. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 24, (pp. 1261–1263).

Surynek, P. (2012). On propositional encodings of cooperative path-finding. In *2012 IEEE 24th International Conference on Tools with Artificial Intelligence*, volume 1, (pp. 524–531). IEEE.

Tajelipirbazari, N., Yildirimoglu, C. U., Sabuncu, O., Arici, A. C., Ozen, I. H., Patoglu, V., & Erdem, E. (2022). Multi-agent pick and delivery with capacities: Action planning vs path finding. In *International Symposium on Practical Aspects of Declarative Languages*, (pp. 24–41). Springer.

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., Carey, C. J., Polat, İ., Feng, Y., Moore, E. W., VanderPlas, J., Laxalde, D., Perktold, J., Cimrman, R., Henriksen, I., Quintero, E. A., Harris, C. R., Archibald, A. M., Ribeiro, A. H., Pedregosa, F., van Mulbregt, P., & SciPy 1.0 Contributors (2020). SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, *17*, 261–272.

Yu, J. & LaValle, S. M. (2016). Optimal multirobot path planning on graphs: Complete algorithms and effective heuristics. *IEEE Transactions on Robotics*, *32*(5), 1163–1177.

Zhang, H., Yao, M., Liu, Z., Li, J., Terr, L., Chan, S.-H., Kumar, T. S., & Koenig, S. (2021). A hierarchical approach to multi-agent path finding. In *Proceedings of the International Symposium on Combinatorial Search*, volume 12, (pp. 209–211).