Multi-agent Pick and Delivery with Capacities: Action Planning Vs Path Finding

Nima Tajelipirbazari[1], Cagri Uluc Yildirimoglu[2], Orkunt Sabuncu[1], Ali Can Arici[3], Idil Helin Ozen[3], Volkan Patoglu[2], and Esra Erdem[2](✉)

[1] Department of Computer Engineering, TED University, Ankara, Turkey
[2] Faculty of Engineering and Natural Sciences, Sabanci University, Istanbul, Turkey
esra.erdem@sabanciuniv.edu
[3] Ekol Logistics, Istanbul, Turkey

Abstract. 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 study this problem with two different approaches, action planning vs path finding, using Answer Set Programming with two different computation modes, single-shot vs multi-shot.

Keywords: Multi-agent path finding · Pick and delivery with capacities · Action planning · Answer set programming

1 Introduction

Most warehouses share the same general pattern of material flow [1]: they receive bulk shipments, put them away for quick retrieval, pick them in response to customer requests, and then pack and ship them out to customers. The third component of this flow, i.e., order-picking, typically accounts for about 55% of warehouse operating costs. With increasing demand on e-commerce due to the pandemic, the importance and effect of order-picking also increase [9]. Furthermore, among the three main phases of order-picking (i.e., traveling to the location of the product, searching for the product at that location, and grabbing/collecting the product), traveling comprises the greatest part of the expense of order-picking.

With these motivations, we study a variant of Multi Agent Pathfinding (**MAPF**) problem, where the agents need to pick and deliver some items on their way to their destination. We call this problem **MAPDC**. In addition to the challenges of MAPF (i.e., finding collision-free plans for each agent from its initial location to a goal destination

This work has been partially supported by Tubitak Grant 118E431.

while minimizing the maximum makespan), **MAPDC** asks also for the allocation of the pick and deliver tasks among agents while taking into account their capacities (i.e., the maximum number of items one agent can carry at a time).

We study this problem with two different approaches, action planning vs path finding, based on Answer Set Programming (ASP) [2,7,11,15,17].

In the planning approach, we represent the agents' actions (i.e., moving to a location, picking a product, delivering a product), and the change in the warehouse (i.e., over the locations and the bags of agents) by an action domain description in ASP. This description takes into account the collision constraints and the capacity constraints. After that, we model **MAPDC** as a planning problem, with its initial state description (i.e., initial locations of agents and the order-picking tasks) and goal description (i.e., destinations of agents while enforcing all the tasks are completed).

In the path finding approach, we model **MAPDC** as a graph problem. We view the warehouse environment as a graph, allocate tasks to each agent, and compute a path and its traversal recursively for each agent, respecting the collision and capacity constraints and ensuring the completion of all tasks, while minimizing the maximum makespan.

We compare these two approaches empirically with randomly generated instances over various sizes and types of warehouses (e.g., where shelves are shorter/longer, closer/farther to/from each other). We use the ASP solver CLINGO [5] in our experiments, considering single-shot vs multi-shot computations.

2 Related Work

MAPF is a well-studied problem with various optimizations, using different approaches, such as ASP [4], ILP [21], SAT [19], and search-based [18] methods.

Considering pick and delivery tasks assigned to agents, some variants of **MAPF** have been studied. For instance, **TAPF** [13] generalizes **MAPF** by assigning targets to teams of agents, while **G-TAPF** [16] further allows greater number of tasks per agent. **MAPDC** differs from **G-TAPF** in that it does not consider teams of agents or assumes an order of tasks. **MAPDC** considers capacity constraints of the agents and asks for the order of tasks as well.

Liu et al. [12] study **MAPD** problems (i.e., **MAPDC** where the capacity of each agent is 1) by first assigning tasks to agents, and then using a search-based path finding algorithm to compute collision-free paths. The search stage tries to minimize the makespan after committing to the task assignment found in the first stage. Although dividing the overall problem into two stages scales better, it does not guarantee optimal solutions for the overall problem. **MAPDC** considers capacity constraints and guarantees optimal solutions without dividing the problem into two parts.

For the offline setting, Honig et al. [10] propose a Conflict Based Search (CBS) approach to solve **MAPD**. It is complete and optimal for the sum of path lengths, a metric different from the one we consider in our approaches. CBS is known to scale poorly as the number of conflicts among agents increases [16]. Along these lines, authors have also presented a bounded sub-optimal approach (ECBS) to improve scalability.

In online versions of **MAPD** [14], agents have to fulfil a stream of delivery tasks. The tasks are allocated to free agents such that every agent can execute at most one task. Then, a path from the initial location to the pick-up location, and then to the delivery location, and finally to the destination, is computed. The agents cannot rest in their destinations after they finish executing tasks. Grenouilleau et al. [8] also address online **MAPD** problems where new tasks appear whenever they arrive. They first assign tasks to agents using an heuristics approach and use a modified A* search algorithm so that one search call is sufficient, instead of two calls, for finding the two paths of the agent: one from its current location to the pickup location and the other one from the pickup location to the delivery one.

Different from these studies, we focus on an offline method that allocates tasks to compute plans, and that computes plans to pick and deliver the allocated tasks. Therefore, in our approach, task allocation and planning are not decoupled: they are handled at the same time. Furthermore, an agent can handle more than one task at a time, and it is not assumed to disappear when it reaches its destination.

Chen et al. [3] solve **MAPD** problem in both offline and online settings. Similar to **MAPDC** problem, they also consider capacities where agents can carry multiple items at a time. Unlike the previous **MAPD** solution techniques, their search-based algorithm handles task assignment and path finding simultaneously. Even though considering actual path costs while assigning tasks to agents improves the quality of solutions found, in its current form, their algorithm does not guarantee optimality, unlike our proposed solutions for the **MAPDC** problem. Regarding optimality of solutions, they propose a variant of the algorithm where local search techniques are used to further improve the best solution at hand.

Another related study is by Vodrazka et al. [20] since they introduce a planning approach to solve and study **MAPF**. They consider a special case of a planning problem with only two actions, move and wait, subject to the constraint that there is no collision between the agents. They present a method that first computes a sequential plan where at most one agent moves to a free location at a time, and cuts the sequential plan into sub-plans, and parallelize them so that several agents can move at a time without any vertex collision. These authors also present another method that splits the move action into start-move and finish-move actions, and ensures that, for each action, the agents need to have a token as a precondition and pass the token to another agent as an effect. Our planning method for **MAPDC** considers picks and delivers as well. Thanks to the expressive formalism of ASP, concurrent plans can be generated without further need to split sequential plans or actions.

Overall, different from the related work, both of our action planning and path finding based approaches address the offline version of **MAPDC** while considering the capacities of agents, solve the task assignment and path finding problems simultaneously to ensure the optimality of solutions in terms of the overall completion time (the maximum makespan) of tasks.

3 MAPDC-P: Solving MAPDC with a Planning Approach

We model the **MAPDC** as a planning problem by representing actions of agents and changes in the warehouse by an action domain description which also considers all collision and capacity constraints. Agents use these plans to navigate from their starting locations to destinations while picking up and delivering items. Our goal is to pick-up and deliver every item in tasks, while optimizing the overall completion time of tasks.

3.1 MAPDC as a Planning Problem

We consider the environment of agents as a graph $G = (V, E)$ where the set V of locations form a 4-neighbour grid and the edges E are based on the adjacent locations in this grid. Some of the locations may be occupied with shelves and they are treated as static obstacles for the agents.

The functional fluent *position* represents the locations of agents. Specifically, for an agent $i \in A$ the fluent value $pos(i) = l@x$ represents the location of i at time step $x \leq u \in \mathbb{Z}^+$ is $l \in V$. Consecutively, $pos(i) = init(i)@0$ and $pos(i) = goal(i)@u$ hold, where *init* and *goal* functions specify the initial and goal locations of agents, respectively.

Agents move along the edges of the graph. Considering the grid structure of the environment, we model $move(i, dir)$ action of agent i, where dir is among four cardinal directions. Specifically, an action occurrence $move(i, dir)@x$ represents a movement of agent i in direction dir at time step x. Whenever $pos(i) = l_1@x-1$ holds, a $move(i, dir)@x$ action occurrence changes the position of the agent i such that $pos(i) = l_2@x$ holds where $(l_1, l_2) \in E$ and position l_2 is adjacent to l_1 in the direction dir[1].

In a **MAPDC** problem, we consider vertex and edge collisions. A vertex collision occurs whenever $pos(i) = l@x$ and $pos(j) = l@x$ hold for two different agents i and j at time point x. Similarly, an edge collision occurs whenever $pos(i) = l_1@x$, $pos(j) = l_2@x$, $pos(i) = l_2@y$, and $pos(j) = l_1@y$ hold s.t. $i \neq j$, $y = x+1$ and $(l_1, l_2) \in E$.

Each task $t \in T$ of the form (id, p, d) has a unique identifier *id*, a pick up location $p \in V$, and a delivery location $d \in V$. A task must be assigned to a unique agent and the agent fulfils it by picking up a product from the task's pick up location and carrying the product until delivered to the task's delivery location. To this end, an agent may perform *pickup* and *deliver* actions with the condition that the agent must be at the pick up and delivery locations, respectively. Specifically, for a task (id, p, d) action occurrences $pickup(i, id)@x$ and $deliver(i, id)@x$ are possible if $pos(i) = p@x-1$ and $pos(i) = d@x-1$ hold, respectively. Additionally, for the latter action to be possible the agent must be carrying the product respective to the task. The boolean fluent *carrying* is used for representing the products an agent carries. The occurrence $pickup(i, id)@x$ causes $carrying(i, id)@x$ to hold and the fluent keeps holding until $deliver(i, id)@y$ occurs such that $y > x$. Note that a task with identifier *id* is fulfilled whenever a $deliver(i, id)$ action occurs by an agent i at time step x. Moreover, the product related to a task cannot be picked up more than one time by an agent. To this end, for an action $pickup(i, id)$ occurrence, neither the task with id *id* is fulfilled before nor the agent i is carrying the product related to the task.

A plan, which is composed of occurrences of actions *move*, *pickup* and *deliver*, is a solution of a **MAPDC** problem, if and only if there are no vertex or edge collisions considering the *position* fluent values projected by the actions in the plan, all agents are at their goal locations at the last time step of the plan, all tasks are fulfilled, and each agent carries at most c number of products at any time step, where capacity c is given as an input of the problem. The last condition constrains that for any agent i and any

[1] The effects of an action appear at the same time step as the occurrence of the action, instead of the next time step, to comply with the multi-shot ASP formulation of **MAPDC-P** explained in the next section.

time step x, $\left|\{id \mid carrying(i,id)@x\ holds\}\right| \leq c$. Based on these notations, **MAPDC** problem can be defined as a planning problem (i.e., **MAPDC-P**) as illustrated in Fig. 1.

3.2 Solving MAPDC-P Using Multi-shot ASP

We rely on multi-shot ASP [5] for solving the **MAPDC-P** problem. Multi-shot solving is used to encode dynamic domains, such as planning problems, where the logic program changes during the solving process. Given a planning problem, one can encode a multi-shot ASP program by partitioning the whole program into three parts; the static part where knowledge that does not change with time is represented, the cumulative part corresponding to knowledge that accumulates at increasing time steps, and the volatile part where we represent knowledge that is added for a specific time step and retracted when the time step is increased during the multi-shot solving process. As a planning problem, **MAPDC-P** is naturally suitable for encoding via multi-shot ASP.

MAPDC-P

Input:
- A graph $G = (V, E)$ where the set V of locations form a 4-neighbour grid and the edges E are based on adjacent locations in this grid.
- A set $O \subseteq V$ (to denote the parts of the environment occupied by shelves).
- A set A of agents.
- A function $init : A \mapsto V$ (to describe the initial locations of agents).
- A function $goal : A \mapsto V$ (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 some positive integer $u \leq t$, a plan W of length u, i.e., a sequence $\langle A_1, ..., A_u \rangle$ where each A_x, $1 \leq x \leq u$ is a set of action occurrences among the following ones:
- $move(i, dir)@x$ (to describe that agent i moves in direction dir at time step x),
- $pickup(i, id)@x$ (to describe that agent i picks up a product for task id at time step x),
- $deliver(i, id)@x$ (to describe that agent i delivers a product for task id at time step x),

such that the following hold:
- No parallel move actions are allowed for an agent at any time step, i.e., for any A_x, there are no $move(i, dir_1)@x \in A_x$ and $move(i, dir_2)@x \in A_x$ s.t. $dir_1 \neq dir_2$.
- All tasks are complete, i.e., $\forall\, task(id, p, d) \in T$ there exists a time step $x \leq u$ and an agent $i \in A$ s.t. $deliver(i, id)@x \in A_x$.
- Agents do not carry more than their capacity, i.e., $\left|\{id \mid carrying(i,id)@x\ holds\}\right| \leq c$.
- All agents are at their goal locations at the last time step, i.e., $\forall i \in A, pos(i) = goal(i)@u$ holds.
- There are no vertex or edge collisions, i.e., whenever $pos(i) = l_i@x$ and $pos(j) = l_j@x$ hold for $i \neq j$, $l_i \neq l_j$ also holds and whenever $pos(i) = l_i@x$ and $pos(j) = l_j@x$ hold for $i \neq j$, $(l_i, l_j) \in E$, $pos(i) = l_j@x+1$ and $pos(j) = l_i@x+1$ cannot hold together.

Fig. 1. MAPDC-P: **MAPDC** as a planning problem.

During multi-shot solving, CLINGO increases the time step until it finds an answer set, which corresponds to a plan for the problem. This way of solving has the advantage that the computed plan will be optimal in terms of the time steps. Hence, a plan for a **MAPDC-P** problem found by multi-shot solving will be optimal regarding its makespan.

In our encoding unlike the common practice in ASP community where actions are executed at time step x and its effects are seen at time step $x+1$, we encode effects of an action at the same time step as the occurrence of the action to eliminate the extra grounding and solving step that otherwise multi-shot solver would need in order to work with the atoms of the next time step.

Similar to a multi-shot ASP encoding of a general planning problem, our **MAPDC-P** encoding has three parts. The static part is mainly composed of the grid environment and facts from the **MAPDC-P** instance. For this part, we rely on the encoding of M domain in asprilo for the instance format and some domain predicate signatures. Asprilo is a framework for experimenting with logistic domains in ASP [6]. Briefly, we use predicates `isRobot/1`, `task/3`, `shelfPos/3`, and `finalPos/2` for representing agents, tasks, locations of shelves (these will be used as pick up and delivery locations of tasks), and goal locations of agents, respectively. Additionally, we have instances of `pos/1` and `nextto/3` predicates to represent set V of grid locations and subset of edges E, where each edge has no location occupied by an obstacle.

The static part also includes the following rule. This choice rule succinctly assigns each task to a unique agent. Later, the `assign/2` predicate will be used as preconditions for representations of actions *pickup* and *deliver*. Note that the `#program` directive groups a set of rules as a program part. Here the static part is named as `base`.

```
#program base.
1{assign(I,ID): isRobot(I)}1 :- task(ID,P,D).
```

In the cumulative part, we represent dynamic knowledge of **MAPDC-P** accumulating with each increasing time point. Specifically, in this part we represent actions and fluents of the domain. Below, an instance `move(i,dir,x)` of the `move/3` predicate represents the action occurrence $move(i,dir)@x$ and the predicate `direction/1` represents cardinal movement directions. Note that parameter x in all the remaining rules is a program part parameter. It represents time steps in our domain and is controlled by CLINGO during incremental grounding of the program part named as `step` by the `#program` directive. Basically, it is instantiated with value 1 and incremented by 1 in each grounding stage of multi-shot solving. Hence, the choice rule below encodes that each agent can move in any of the directions at a time step. The upper bound in the choice rule head neatly represents the constraint that no parallel movement actions are allowed for an agent. The second rule represents the effect of movement action. An instance `pos(i,l,x)` of the `pos/3` predicate represents the fluent valuation $pos(i) = l@x$ holds. The third rule constrains that no agent moves in a direction where there is no edge from the current position of the agent.

```
#program step(x).
{move(I,DIR,x): direction(DIR)} 1 :- isRobot(I), time(x).
pos(I,L,x) :- move(I,DIR,x), pos(I,L',x-1), nextto(L',DIR,L).
:- move(I,DIR,x), pos(I,L ,x-1), not nextto(L ,DIR,_).
```

Next two rules represent the *position* fluent is inertial and functional, respectively. For any agent, when there is no reason to change its location (whenever it does not move), its location from the previous time step stays the same for this time step. Related to this, an agent must be at exactly one location at each time step.

```
{pos(I,L,x)} :- pos(I,L,x-1), isRobot(I), time(x).
:- {pos(I,L,x)}!=1, isRobot(I), time(x).
```

The following group of rules guarantees that there are no edge or vertex collisions in the plan found as an answer set. The `moveto(l',l,x)` instance in an answer set represents that an agent has moved from `l'` to `l` at time step *x*.

```
moveto(L',L,x) :- nextto(L',DIR,L), pos(I,L',x-1), move(I,DIR,x).
:- moveto(L',L,x), moveto(L,L',x), L < L'.
:- {pos(I,L,x): isRobot(I)}>1, pos(L), time(x).
```

The rules presented upto now in the cumulative part are based on the encoding of asprilo's M domain [6], which basically encodes the **MAPF** problem.

In a **MAPDC-P** problem, agents can perform additional pick up and deliver actions. The following rule represents that an agent *i* can choose to pick up a product for the task *id* at time step *x* if the task *id* has been assigned to *i* (`assign/2` predicate in the body), agent *i* is not already carrying the product for *id* (negative `carrying/3` predicate), and the task *id* has not been fulfilled yet (negative `delivered/3` predicate).

```
{pickup(I,ID,x)} :- pos(I,P,x-1), task(ID,P,D),
  assign(I,ID), not delivered(ID,x), not carrying(I,ID,x-1).
```

Similarly, the following choice rule represents *deliver* actions where `deliver(i,id,x)` basically corresponds to the action occurrence $deliver(i,id)@x$ in a plan. Note that, for a *deliver* action, we need the agent must be carrying the respective product (represented by the `carrying/3` predicate in rule body).

```
{deliver(I,ID,x)} :- carrying(I,ID,x-1), pos(I,D,x-1),
   task(ID,P,D), assign(I,ID).
```

Considering the previous rules defining *pickup* and *deliver* actions, an atom of the form `carrying(i,id,x)` represents that the fluent $carrying(i,id)$ holds at time step *x*. The first rule in the following group states that *carrying* fluent is a direct effect of *pickup* action. Similarly, `delivered(id,x)` represents that task *id* has been fulfilled via *delivered* fluent. This fluent is an effect of *deliver* action (represented by third rule). While the second rule represents that *carrying* fluent does not change its value until the agent delivers the respective package, the last rule represents that the *delivered* fluent persists after becoming true.

```
carrying(I,ID,x) :- pickup(I,ID,x).
carrying(I,ID,x) :- carrying(I,ID,x-1), not deliver(I,ID,x).
delivered(ID,x) :- deliver(I,ID,x).
delivered(ID,x) :- delivered(ID,x -1).
```

Thanks to the powerful construct of aggregates in ASP, the following constraint rule prevents any agent carry more products than the capacity *c* at any time step.

```
:- isRobot(I),time(x),#count{ID : carrying(I,ID,x)} > c.
```

The following rules comprises the volatile part where we represent knowledge that does not accumulate and is related to a specific time step. In a typical multi-shot encoding of a planning problem, this part is used for representing goal conditions of the problem. While the first rule guarantees that all tasks are fulfilled, the second one assures all agents are at their destination locations at the last time step u. Note that the `query/1` predicate is an external one that is controlled by CLINGO. Given an instance of it (e.g., `query(y)`), CLINGO sets its truth value as true when searching for a plan at horizon `y`. In case no plan is found for horizon `y`, the truth value of `query(y)` becomes false and the current horizon is incremented.

```
#program check(x).
:- ordered(ID), not delivered(ID,x), query(x).
:- finalPos(I,L), not pos(I,L,x), query(x).
```

4 MAPDC-G: Solving MAPDC with a Path Finding Approach

We model the **MAPDC** problem as a graph problem. The idea is to view the warehouse environment as a graph, allocate tasks to each agent, and compute a path and its traversal for each agent, respecting the collision constraints and ensuring the completion of all tasks, while minimizing the maximum makespan.

4.1 MAPDC as a Graph Problem

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 $\langle w_{i,1}, w_{i,2}, \dots, w_{i,n_i}\rangle$ 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 suppose that each pickup and delivery takes one unit of 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 $traversal_i$ 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 $traversal_i(x) = p$, $traversal_i(y) = d$ and $(id, p, d) \in carry_i(z)$ for every time step z between x and y only.

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

MAPDC-G

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 $init : A \mapsto V$ (to describe the initial locations of agents).
- A function $goal : A \mapsto V$ (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 $w_{i,n_i} = goal(i)$ from its initial location $w_{i,1} = init(i)$,
  - to complete the allocated tasks T_i (i.e., for every $(id, p, d) \in T_i$, 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 $traversal_i$ 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: $|carry_i(x)| < c$.

Fig. 2. **MAPDC-G**: **MAPDC** as a graph problem.

Based on these notations, **MAPDC** problem can be defined as a graph problem (i.e., **MAPDC-G**) as illustrated in Fig. 2. 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 $traversal_i$ of P_i ensuring that agent i completes the allocated set T_i of tasks by a time step $u_i \leq t$.

4.2 Solving MAPDC-G Using Multi-shot ASP

We present a multi-shot formulation of **MAPDC-G** in ASP, based on the definition in Fig. 2. 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 `traverse/3` and `carry/3`. Here, `traverse(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`.

The ASP formulation for **MAPDC-G** consists of three parts: `base`, `step`, `check`. Note that Listing 7 in [5] should be included at the beginning of the formulation.

The `base` program is grounded only once. It consists of the rules expressing that the traversals start at the initial locations of agents, and each task is assigned to one agent:

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

The `step` program is grounded incrementally for `t=1,2,3,...`. For each agent, a path and its traversal are generated recursively ensuring that the agent cannot be at two different locations.

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

Then the following constraints are used to ensure that agents do not collide with each other at a node or an edge, or with obstacles.

```
:- traverse(I,t,X), traverse(J,t,X), node(X), agent(I), agent(J), I<J.
:- traverse(I,t-1,X), traverse(I,t,Y), traverse(J,t-1,Y),
   traverse(J,t,X), agent(I), agent(J), edge(X,Y), I<J.
:- traverse(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 product of that task. An agent can finish a task if the agent is at the delivery location of the product of that task, provided that the task is already started at a previous time-step.

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

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` program is grounded for each value of `t` until a solution is computed. In particular, we need to ensure that every task is completed, and that agents should end up at their destinations.

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

Table 1. Warehouse configurations used in our experiments.

Configuration	1	2	3	4	5	6	7
Grid size	10×10	10×10	10×10	10×10	10×20	10×40	20×20
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

5 Experimental Evaluations

We have experimentally evaluated **MAPDC-P** and **MAPDC-G** to better understand their scalability in terms of the computation time. To this end, we have generated 7 different warehouse environments, varying the grid size, shelf width and horizontal/vertical distances between shelves as described in Table 1.

For each such warehouse configuration, we have created 9 instances that vary the number of agents, the number of tasks and the capacity of the agents. For every combination of configuration, agent number, task number and capacity, we have randomly generated 5 instances and reported the average computation times and makespans.

In each instance, the initial and goal locations of agents lie at the bottom row of the layout and were chosen randomly. Picking and delivery locations of tasks are selected from a pool of nodes that are vertically adjacent to the shelf nodes. In order to have predictable hardness for tasks, picking and delivery locations of the tasks pass through exactly one shelf in the vertical direction. Combinations that only differ in capacity have the same instances except for the capacity values.

For the experiments, we have used CLINGO (4.5.4) with *default* and *handy* configurations, on a Linux server with dual 2.4 GHz Intel E5-2665 CPUs and 64 GB memory.

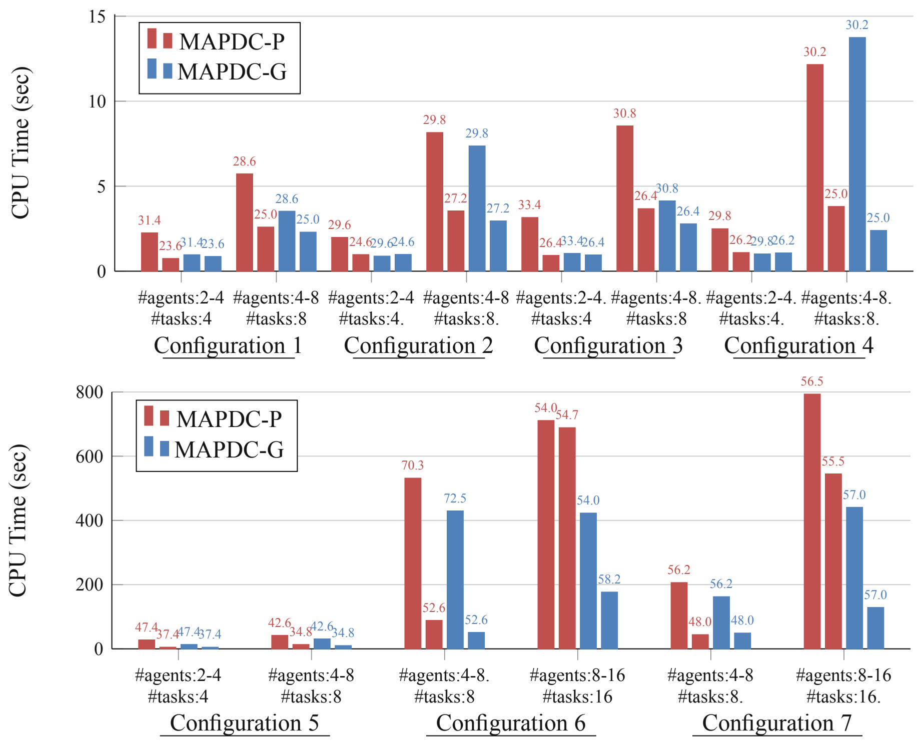

Fig. 3. Scalability as the number of agents increases when capacity = 1 (Table 2).

Table 2. Results with CLINGO's `default` configuration

Configuration	Agents	Tasks	Capacity	MAPDC-G		MAPDC-P	
				CPU time	Makespan (#solved)	CPU time	Makespan (#solved)
Configuration 1	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)
	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)
Configuration 2	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)
	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)
Configuration 3	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)
	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)
Configuration 4	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)
	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)
Configuration 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)
	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)

(*continued*)

Table 2. (*continued*)

Configuration	Agents	Tasks	Capacity	MAPDC-G		MAPDC-P	
				CPU time	Makespan (#solved)	CPU time	Makespan (#solved)
Configuration 6	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)
	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
Configuration 7	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

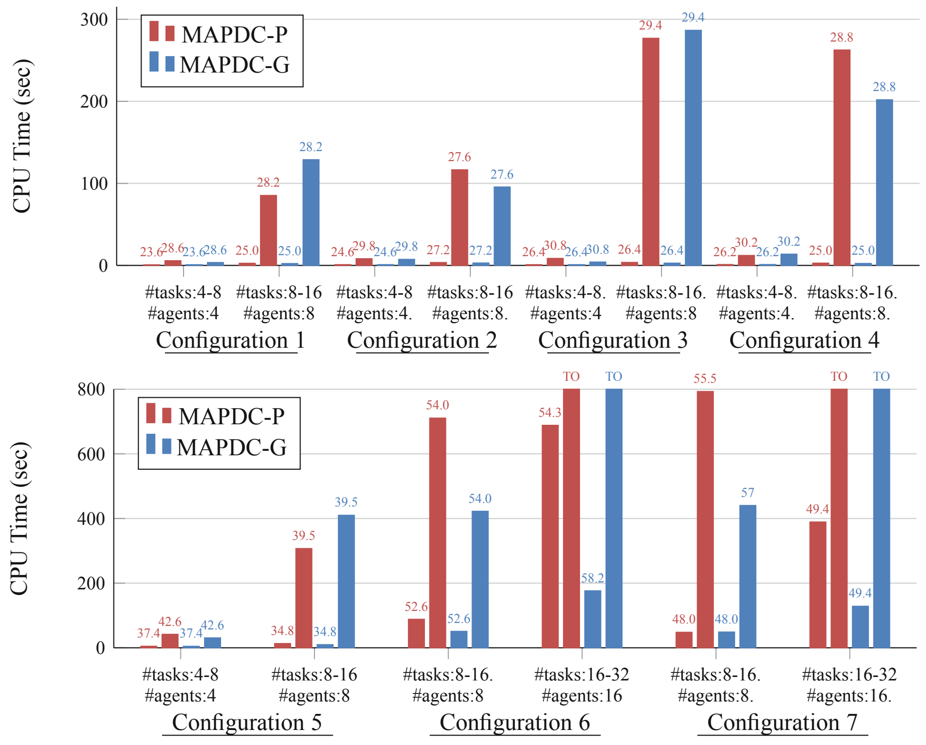

Fig. 4. Scalability as the number of tasks increases when capacity = 1 (Table 2).

For each configuration, we have analysed the effect of changing the number of agents, tasks, and capacity on performances of our solution methods for **MAPDC-P** and **MAPDC-G**. In order to have a controlled comparison, we have doubled a factor while keeping the others fixed, and compared the results for both methods. The results are presented in Table 2.

We have observed (Fig. 3) that increasing the number of agents helps both **MAPDC-P** and **MAPDC-G** 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 (Fig. 4) 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-G**. 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 (Fig. 5) leads to a decrease in maximum makespan, and thus reduces the computation time for both **MAPDC-P** and **MAPDC-G**. It is interesting to compare results in Figs. 3 and 5.

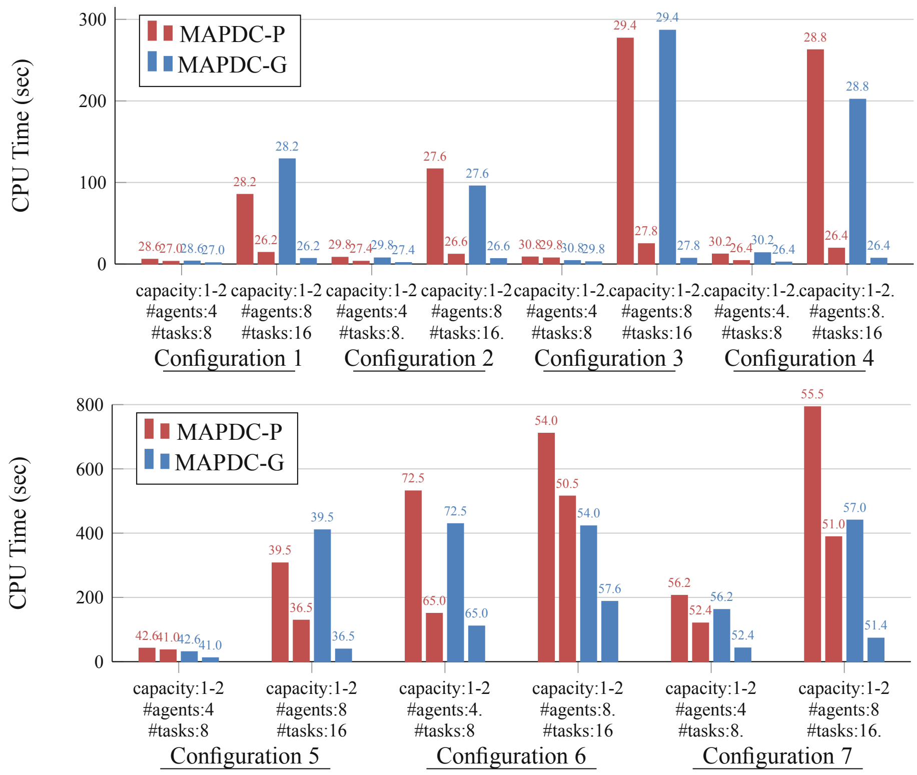

Fig. 5. Scalability as the capacity increases (Table 2).

Even though increasing the number of agents and the capacity both reduce the maximum makespan and improve runtime performance, improvements in Fig. 5 are more visible. This showcases the importance of capacity in the **MAPDC** problem. Increasing 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 3). 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 singleshot ones for many instances. The run-time of multi-shot computations include the time

Table 3. 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	
			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 4. 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.

Instance	Single-shot anytime				Multi-shot			
	MAPDC-G		MAPDC-P		MAPDC-G		MAPDC-P	
	CPU time	Makespan (#opt/#solved)	CPU time	Makespan (#opt/#solved)	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)	16.99	43.0(5)	16.21	43.0(5)
(4,8,1)	201.12	56.2 (5/5)	794.14	**56.2 (4/5)**	162.02	56.2(5)	233.99	56.2(5)
(4,8,2)	122.59	52.4 (5/5)	710.64	52.4 (5/5)	42.35	52.4(5)	127.96	52.4(5)
(8,8,1)	189.66	48.0 (5/5)	202.67	48.0 (5/5)	48.76	48.0(5)	64.74	48.0(5)
(8,16,1)	855.31	**58.2 (2/5)**	timeout	timeout	440.30	57.0 (3)	816.92	56.5 (2)
(8,16,2)	255.07	51.4 (5/5)	997.27	**76.5 (0/2)**	72.97	51.4(5)	511.79	49.8(4)
(16,16,1)	457.70	49.4 (5/5)	997.23	**67.0 (0/4)**	128.51	49.4 (5)	544.47	47.0 (4)
(16,32,1)	997.61	**69.4 (0/5)**	timeout	timeout	timeout	timeout	timeout	timeout
(16,32,2)	836.89	**54.0 (2/5)**	timeout	timeout	264.36	52.2 (4)	timeout	timeout

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 s, over some **MAPDC** instances with an upper bound of 80 on makespan (Table 4). We have observed that anytime search helps with finding a suboptimal solution for all instances, in particular, for **MAPDC-G**, but at the cost of computation time due to grounding with a large makespan. **MAPDC-G** computes optimal solutions for 34 instances (out of 45 instances), using anytime search; and, for some of the remaining instances (e.g., with 8 agents, 16 tasks, 1 capacity), the average suboptimal values are close to the optimal ones.

Finally, our experiments show that using CLINGO with `handy` configuration (as in [4]) improves the computational performances of **MAPDC-P** and **MAPDC-G** for hard instances (cf. results for Configurations 6 and 7 at Table 5). This suggests the use of CLINGO with a portfolio of different configurations.

Table 5. Results with CLINGO's `handy` configuration

Configuration	Agents	Tasks	Capacity	MAPDC-G		MAPDC-P	
				CPU time	Makespan (#solved)	CPU time	Makespan (#solved)
Configuration 6	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)
	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
Configuration 7	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)
	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

6 Conclusions

We have introduced two novel methods, based on action planning (**MAPDC-P**) and path finding (**MAPDC-G**), to find optimal solutions for the Multi-Agent Pick and Delivery with Capacities problem (**MAPDC**). Both methods rely on formulating the problems in Answer Set Programming, and take advantages of multi-shot computation of the ASP solver CLINGO.

We have experimentally evaluated these two methods on a rich set of benchmark instances that are randomly generated with varying numbers of agents, tasks and capacities, over different warehouses that vary in grid size, shelf width, and horizontal/vertical distances between shelves. We have observed that **MAPDC-P** is more efficient in time in small instances while **MAPDC-G** scales better for larger instances. We have also observed that **MAPDC-P** benefits more from the multi-shot computation.

We have also experimentally evaluated these two methods considering two other computation modes of ASP: single-shot and anytime search. We have observed the advantages of multi-shot computation over single-shot computation, in particular for **MAPDC-P**, in terms of computation time, and the advantages of single-shot anytime computation, in particular for **MAPDC-G**, in terms of the number of problems solved.

In this study, we have evaluated two approaches (i.e., action planning and path finding) in the context of declarative programming. Further evaluations considering non-declarative approaches (e.g., based on search) are part of our ongoing work.

References

1. Bartholdi, J.J., III., Hackman, S.T.: Warehouse and distribution science. Supply Chain and Logistics Institute, Georgia Institute of Technology (2019)
2. Brewka, G., Eiter, T., Truszczynski, M.: Answer set programming: an introduction to the special issue. AI Mag. **37**(3), 5–6 (2016). https://doi.org/10.1609/aimag.v37i3.2669
3. Chen, Z., Alonso-Mora, J., Bai, X., Harabor, D.D., Stuckey, P.J.: Integrated task assignment and path planning for capacitated multi-agent pickup and delivery. IEEE RAL **6**(3), 5816–5823 (2021). https://doi.org/10.1109/LRA.2021.3074883
4. Erdem, E., Kisa, D., Oztok, U., Schüller, P.: A general formal framework for pathfinding problems with multiple agents. In: Proceedings of AAAI (2013)
5. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Multi-shot ASP solving with clingo. TPLP **19**(1), 27–82 (2019). https://doi.org/10.1017/S1471068418000054
6. Gebser, M., et al.: Experimenting with robotic intra-logistics domains. TPLP **18**(3–4), 502–519 (2018). https://doi.org/10.1017/S1471068418000200
7. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Gener. Comput. **9**, 365–385 (1991)
8. Grenouilleau, F., van Hoeve, W.J., Hooker, J.N.: A multi-label A* algorithm for multi-agent pathfinding. In: Proceedings of ICAPS, pp. 181–185 (2019)
9. Guthrie, C., Fosso-Wamba, S., Arnaud, J.B.: Online consumer resilience during a pandemic: an exploratory study of e-commerce behavior before, during and after a COVID-19 lockdown. JRCS **61**, 102570 (2021). https://doi.org/10.1016/j.jretconser.2021.102570
10. Hönig, W., Kiesel, S., Tinka, A., Durham, J., Ayanian, N.: Conflict-based search with optimal task assignment. In: Proceedings of AAMAS (2018)
11. Lifschitz, V.: Answer set programming and plan generation. AIJ **138**, 39–54 (2002). https://doi.org/10.1016/S0004-3702(02)00186-8
12. Liu, M., Ma, H., Li, J., Koenig, S.: Task and path planning for multi-agent pickup and delivery. In: Proceedings of AAMAS, pp. 1152–1160 (2019)
13. Ma, H., Koenig, S.: Optimal target assignment and path finding for teams of agents. In: Proceedings of AAMAS, pp. 1144–1152 (2016)

14. Ma, H., Li, J., Kumar, T.K.S., Koenig, S.: Lifelong multi-agent path finding for online pickup and delivery tasks. In: Proceedings of AAMAS, pp. 837–845 (2017)
15. Marek, V.W., Truszczyński, M.: Stable models and an alternative logic programming paradigm. In: Apt, K.R., Marek, V.W., Truszczynski, M., Warren, D.S. (eds.) The Logic Programming Paradigm. Artificial Intelligence. Springer, Heidelberg (1999). https://doi.org/10.1007/978-3-642-60085-2_17
16. Nguyen, V., Obermeier, P., Son, T.C., Schaub, T., Yeoh, W.: Generalized target assignment and path finding using answer set programming. In: Proceedings of IJCAI, pp. 1216–1223 (2017). https://doi.org/10.24963/ijcai.2017/169
17. Niemelä, I.: Logic programs with stable model semantics as a constraint programming paradigm. AMAI **25**, 241–273 (1999)
18. Sharon, G., Stern, R., Felner, A., Sturtevant, N.R.: Conflict-based search for optimal multi-agent pathfinding. AIJ **219**, 40–66 (2015). https://doi.org/10.1016/j.artint.2014.11.006
19. Surynek, P.: On propositional encodings of cooperative path-finding. In: Proceedings of ICTAI, pp. 524–531 (2012). https://doi.org/10.1109/ICTAI.2012.77
20. Vodrázka, J., Barták, R., Svancara, J.: On modelling multi-agent path finding as a classical planning problem. In: Proceedings of ICTAI, pp. 23–28 (2020). https://doi.org/10.1109/ICTAI50040.2020.00014
21. Yu, J., LaValle, S.M.: Optimal multirobot path planning on graphs: complete algorithms and effective heuristics. IEEE TRO **32**(5), 1163–1177 (2016). https://doi.org/10.1109/TRO.2016.2593448