Experimental Evaluation of Multi-Agent Pathfinding Problems using Answer Set Programming

Esra Erdem, Doga G. Kisa, Umut Oztok, and Peter Schüller

Faculty of Engineering and Natural Sciences
Sabancı University, İstanbul, Turkey

Abstract. Pathfinding for a single agent is the problem of planning a route from an initial location to a goal location in an environment, going around obstacles. Pathfinding for multiple agents also aims to plan such routes for each agent, subject to different constraints, such as restrictions on the length of each path, no intersection of paths/plans, no crossing/meeting each other. It also has variations for finding optimal solutions with respect to the maximum path length or the sum of plan lengths. These problems are important for many real-life applications, such as motion planning, vehicle routing, environmental monitoring, patrolling, computer games. We evaluate the applicability of Answer Set Programming to solve variations of multi-agent pathfinding problems, by experiments with randomly generated problem instances on a grid, on a real-world road network, and on a real computer game terrain.

1 Introduction

Pathfinding for a single agent is the problem of planning a route from an initial location to a target location in an environment, going around obstacles. Pathfinding for multiple agents (**PF**) also aims to plan such routes for each agent, but subject to various constraints, such as no self-intersecting paths, no intersection of paths/plans, no crossing/meeting each other, no waiting idle, restrictions on the length of each path/plan and on the total length of paths/plans, and requirements on visiting multiple target locations. **PF** also has variations for finding optimal solutions where the goal is to minimize the maximum path/plan length, the sum of path/plan length, the number of target locations visited, etc. Some of these **PF** problems have been studied in the literature, but under different names. For instance, if it is required that the plans of the agents do not interfere with each other then **PF** is called multi-agent pathfinding problem. Assuming that the agents are homogenous and move one unit at a time, deciding whether there is a solution of at most k moves to multi-agent pathfinding problem is NP-complete [19]; its optimization variant is NP-hard [30].

Proceedings of the 20$^{\text{th}}$ RCRA workshop on *Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion* (RCRA 2013).
Rome, Italy, June 14–15, 2013.

Despite their difficulty, **PF** problems have played a significant role in different applications, such as motion planning [1], vehicle routing and traffic management [31, 4, 18], air traffic control [32], computer games [23], disaster rescue [13, 11], formation generation for multi-agent networks [24], patrolling and surveillance of environment [16, 35].

In these applications, **PF** is solved subject to various relevant constraints. For instance, in computer games such as Warcraft III, the routes of agents may intersect with each other as long as the agent's plans do not interfere with each other (i.e., the agents can be at the same location in different time steps, but not at the same time). On the other hand, in environmental coverage or surveillance, it may be required that the routes of agents do not intersect with each other at all so that more areas are covered in a shorter amount of time (note also the limited power supplied by batteries of robots). In disaster rescue, it may be required that certain parts of the world are checked by some agent (i.e., they should be covered by some route) since it is more probable for some people to be there. To prevent an agent to do all the work, a restriction may be specified on the length of the route or the duration of the plan for each agent. Furthermore, the maximum plan length or the sum of plan lengths may be minimized to save some battery power.

We introduce a formal framework that is general enough to solve many variations of **PF** (including the ones mentioned above) declaratively, with the possibility of guaranteed optimality with respect to some criteria based on plan lengths, and with the possibility of embedding heuristics. This framework is based on Answer Set Programming (ASP) [14, 2]—a knowledge representation and reasoning paradigm with an expressive formalism and efficient solvers. The idea of ASP is to formalize a given problem as a "program" and to solve the problem by computing models (called "answer sets" [9]) of the program using "ASP solvers", such as CLASP [8]. The expressive formalism of ASP allows us to easily represent variations of **PF**, as well as sophisticated heuristics to improve the computational efficiency and/or quality of solutions. Such a flexible elaboration tolerant [17] general framework is useful in studying and understanding variations of **PF** and different heuristics in different applications. Deciding whether an ASP has an answer set for nondisjunctive programs is NP-complete [3]; therefore, ASP is suitable for solving many **PF** problems.

We investigate the applicability and effectiveness of our ASP-based framework (from the point of view of computational efficiency and quality of solutions) by experiments with randomly generated problem instances on a grid, on a real-world road network, and with a computer game terrain.

The rest of the paper is organized as follows. After providing the formal definition of multi-agent pathfinding problem and its variants (Section 2), we describe how these problems can be encoded in ASP (Sections 3 and 4). Experimental results are presented in Section 5. We review the related work in Section 6, and conclude by providing a discussion of how our framework can further be improved in Section 7.

Parts of this paper are discussed in [5].

2 Pathfinding for Multiple Agents

We consider a general pathfinding problem (i.e., **PF**) where multiple agents need to find paths from their respective starting locations to their goal locations, ensuring that the paths do not collide with static obstacles and that no two agents collide with each other, and view **PF** as a graph problem as follows.

Input:

- a graph $G = (V, E)$,
- a positive integer k,
- a function h that maps every positive integer $i \leq k$ to a pair (v_i, u_i) of vertices in V,
- a set $O \subseteq V$, and
- a function g that maps every positive integer $i \leq k$ to a positive integer l_i.

Output: For every positive integer $i \leq k$ with $h(i) = (v_i, u_i)$ and $g(i) = l_i$,

- a path $P_i = \langle w_{i,1}, \ldots, w_{i,n_i} \rangle$ for some $n_i \leq l_i$ from $w_{i,1} = v_i$ to $w_{i,n_i} = u_i$ in G where each $w_{i,j} \in V \setminus O$, and
- a function f_i that maps every nonnegative integer less than or equal to l_i to a vertex in P_i such that,
  - (i) for every $w_{i,j}$, $w_{i,j'}$ in P_i and for every nonnegative integer $t < l_i$, if $f_i(t) = w_{i,j}$ and $f_i(t+1) = w_{i,j'}$ then $w_{i,j'} = w_{i,j}$ or $w_{i,j'} = w_{i,j+1}$; and
  - (ii) for different paths P_i and P_j and positive integers $t_i \leq l_i$, $t_j \leq l_j$, if $f_i(t_i) = f_j(t_j)$ then $t_i \neq t_j$.

Intuitively, graph G characterizes the environment (e.g., a game terrain) where the agents move around, positive integer k denotes the number of agents, function h describes the initial and goal locations of agents, set O denotes the parts of the environment covered by the static obstacles, and function g specifies the maximum plan length l_i for each agent i. A path P_i in G from an initial location v_i to a goal location u_i characterizes the path that the agent i plans to traverse. The accompanying function f_i denotes which vertices in the path are visited when; the conditions on f_i makes sure that (i) the agent visits consecutive vertices in P_i at consecutive time steps, or waits at a vertex (e.g., to give way to other agents), and that (ii) no two agents meet at the same place (e.g., to avoid collisions).

We define variations of **PF** by restricting solutions further using the following *constraints* (ASP can handle these and more constraints, which we omit for space reasons):

C No path P_i has a cycle.
This constraint can be useful in cases where robotic agents are not preferred to visit the same part of the environment many times, for a more efficient use of their batteries.

I No two different paths P_i and P_j $(i < j)$ intersect with each other.

This constraint can be useful in cases where it is sufficient if only one robotic agent visits each part of the environment.

W Waiting of agents is not allowed (e.g., to minimize idle time): for every path P_i and for every nonnegative integer $t_i < l_i$, if $f_i(t_i) \neq u_i$, i.e., agent i has not yet reached its goal u_i, then $f_i(t_i) \neq f_i(t_i{+}1)$.

X Agents cannot "pass through" each other (switch place): for every two different paths P_i and P_j and for every nonnegative time step $t < \min(l_i, l_j)$, if $f_i(t) = w_x$, $f_i(t{+}1) = w_{x'}$, and $f_j(t) = w_{x'}$ then $f_j(t{+}1) \neq w_x$.

L The sum of the plan lengths is less than or equal to a given positive integer: for each agent i, the smallest time step t_i such that $f(t_i) = u_i$ denotes the length of its plan. Formally the sum of plan lengths is $\sum_{i=1}^{k} \min\{t_i : f(t_i) = u_i\}$. This constraint can be useful in cases where we want to minimize the total time (and energy) spent by agents.

Arbitrary combinations of these constraints can be considered to solve variations of **PF**. For instance, multi-agent pathfinding problems consider **X**; some of them, e.g., [25, 26, 22, 36] focus on finding solutions where the sum of plan lengths is as small as possible (as suggested by **L**). Problems studied by patrolling/surveillance applications [16] do not consider **X**, but since they focus on finding plans that ensure some parts of the environment are visited by some agent, they sometimes consider **I** [35].

An interesting variation of **PF** aims to reach all goals as soon as possible. We call this problem **TPF**, it minimizes maximum plan length over all agents, formally it minimizes $\max_{i=1}^{k} \min\{t_i : f(t_i) = u_i\}$.

3 Solving PF in ASP

We represent a **PF** problem as a program P in ASP, whose answer sets correspond to solutions of the problem. We refer readers to [14, 2], for syntax and semantics of programs.

We describe the input $I = (G, k, h, O, g)$ of a **PF** instance by a set F_I of facts: $\mathit{edge}(v, u)$ represents edges $(v, u) \in E$; $\mathit{start}(i, v)$ and $\mathit{goal}(i, u)$ represent start and goal vertices of each agent $i \leq k$ (i.e., $h(i) = (v, u)$); $\mathit{limit}(i, l_i)$ represents that $g(i) = l_i$; finally $\mathit{clear}(v)$ represents that $v \in V \setminus O$.

The output (P_i, f_i) of **PF** characterizes for each agent i a path plan such that the agent reaches the goal location from its initial location and avoids obstacles. We represent path plans by atoms of the form $\mathit{path}(i, t, v)$ which specify that at time step t, agent i is at vertex v, formally $f_i(t) = v$.

The ASP program P defines path plans recursively. The first vertex visited by agent i at step 0 is its initial location v:

$$\mathit{path}(i, 0, v) \leftarrow \mathit{start}(i, v).$$

If a vertex v is visited by the agent i at step $0 \leq t < l_i$, then either the agent waits at v or it moves along an edge (v, u) to adjacent vertex u:

$$1\{\ \mathit{path}(i, t{+}1, v), \mathit{path}(i, t{+}1, u) : \mathit{edge}(v, u)\ \}1 \leftarrow \mathit{path}(i, t, v), t < l_i, \mathit{limit}(i, l_i).$$

where $0 \leq t < l$ and $l = \max_{1 \leq i \leq k} l_i$ denotes the overall maximum plan length.

We forbid that an agent uses an obstacle vertex by rule

$$\leftarrow \mathit{path}(i,t,v), \mathit{not}\ \mathit{clear}(v).$$

We ensure that each agent i reaches its goal v by the rules

$$\begin{aligned} &\leftarrow \mathit{goal}(i,v), \mathit{not}\ \mathit{visit}(i,v) \\ \mathit{visit}(i,v) &\leftarrow \mathit{path}(i,t,v) \end{aligned}$$

where $\mathit{visit}(i,v)$ describes that the path of agent i contains v.

Finally we enforce that agents do not meet each other by constraints

$$\leftarrow \mathit{path}(i,t,v), \mathit{path}(i',t,v) \qquad (v \in V, 1 \leq i < i' \leq k, 0 \leq t \leq l_i, l_j)$$

The ASP program P is sound and complete.

Theorem 1. *Given a problem instance $I = (G,k,h,O,g)$, for each answer set S of $P \cup F_I$, the set of atoms of the form $\mathit{path}(i,t,v)$ in S encodes a solution (P_i, f_i) $(1 \leq i \leq k)$ to the* **PF** *problem. Conversely each solution to the* **PF** *problem corresponds to a single answer set of $P \cup F_I$.*

The atoms in P that include time steps only depend on atoms of previous time steps. So we can use the splitting set theorem and the method proposed in [7] iteratively at each time step to eventually show (by induction) that the answer sets for $P \cup F_I$ characterize obstacle-free paths for each agent that also avoid collisions of agents.

4 Solving Variations of PF in ASP

To solve variations of **PF** in ASP, we simply add to the main program P described above, a set of ASP rules which encode the relevant constraints. For instance, to solve multi-agent pathfinding, we add to P the ASP constraints describing Constraints $\underline{\mathbf{X}}$. It is important to emphasize here that, we do not modify the rules of the program P; in that sense, our formulation of **PF** in ASP is elaboration tolerant [17]. Let us see how the constraints are formulated.

Constraint $\underline{\mathbf{C}}$ (i.e., no cycles in a path) can be expressed in ASP by the following constraint for each agent $i \in 1 \ldots k$:

$$\leftarrow 2\{\mathit{path}(i,0,v), \ldots, \mathit{path}(i,l_i,v)\}, \mathit{not}\ \mathit{goal}(v). \quad (v \in V)$$

These constraints ensure that, for every agent i, no non-goal vertex in the path P_i is visited twice or more by the agent i.

Constraint $\underline{\mathbf{I}}$ (i.e., no intersection of paths) is encoded as

$$\leftarrow \mathit{visit}(i,v), \mathit{visit}(i',v) \quad (v \in V, 1 \leq i < i' \leq k)$$

which ensures that no two agents i, i' visit the same vertex v.

Constraint **W** (i.e., no waiting) can be formalized as

$$\leftarrow path(i,t,v), path(i,t{+}1,v), not\ goal(i,v) \quad (v \in V, 1 \leq i \leq k, 0 \leq t < l_i).$$

Constraint **X** (i.e., no swapping places) can be represented by the following constraints for pairs of distinct agents i, j:

$$\leftarrow path(i,t,v), path(i,t{+}1,w), path(j,t,w), path(j,t{+}1,v) \quad (i \neq j, (v,w) \in E,\ 0 \leq t < l_i, l_j).$$

Constraint **L** (i.e., the total plan length is restricted by a positive integer z) can be formalized in ASP as follows

$$\leftarrow totalPlanLength(t) \quad (t > z)$$

where $totalPlanLength(t)$ ("the sum of all plan lengths is t") is defined as follows:

$$totalPlanLength(t) \leftarrow sum\langle\{x : planLength(i,x)\}\rangle = t$$
$$planLength(i,t) \leftarrow path(i,t,v), goal(i,v), path(i,t{-}1,v'), not\ goal(i,v') \quad (0 < t \leq l_i, v, v' \in V, 1 \leq i \leq k).$$

Here the aggregate *sum* is used to find the total plan lengths; and $planLength(i,t)$ describes the plan length for agent i.

We can formalize **TPF** in ASP, by simply adding to the formulation of **PF** in ASP (i.e., program P), the rules above that define $planLength(i,x)$ and the following rules:

$$maxPlanLength(t) \leftarrow max\langle\{x : planLength(i,x)\}\rangle = t$$
$$\#minimize\ [\ maxPlanLength(t) = t\ ].$$

The optimization variant of multi-agent pathfinding, which minimizes the maximum plan length, can be represented by adding Constraint **X** to the formulation of **TPF** in ASP.

5 Experimental Results

We performed experiments with various randomly generated instances of **PF** to be able to understand

- how the input parameters affect the computation time and solution quality (i.e., average path plan length);
- how adding constraints to the main problem formulation affects the computation time and solution quality;
- how various heuristics affect the computation time and solution quality.

We randomly generated problem instances of **PF** on three sorts of graphs: random grid graphs, a real road network, and a computer game terrain.

- The grid graphs are of size $25{\times}25$, we vary the number of agents $k = 5, 10, 15, 20$ and the percentage of grid points covered by obstacles $o = 10, 20, 40$.

Table 1. PF and **TPF** on random 25×25 grid graphs with a variable number of obstacles o and agents k.

o %	k	Grounding sec	First Solution sec (plan length)	Optimal Solution sec (plan length)
10	5	0.70	0.27 (30.4)	7.60 (27.4)
	10	1.88	0.84 (31.2)	13.39 (29.7)
	15	3.43	1.39 (32.9)	18.97 (31.6)
	20	6.27	3.66 (33.4)	30.78 (32.6)
20	5	0.39	0.16 (28.0)	4.21 (26.0)
	10	1.59	0.68 (31.5)	16.12 (29.6)
	15	3.51	1.88 (38.0)	20.71 (36.0)
	20	5.90	4.13 (35.8)	31.35 (33.8)
40	5	0.67	0.28 (58.8)	7.12 (54.5)
	10	2.10	1.19 (63.8)	18.07 (59.6)
	15	3.78	4.99 (62.0)	29.19 (57.8)
	20	7.28	10.58 (69.7)	45.30 (66.6)

- The road network dataset [35] consists of 769 vertices and 1130 edges. In our experiments with this road network, $k = 5, 10, 15, 20$.
- The computer game terrain dataset [27] is a map (called battleground.map) of a computer game Warcraft III. It is a 512×512 grid-based graph which consists of 262144 vertices and 523264 edges. Among those vertices, 92268 of them are not covered by obstacles. We only considered those vertices in our experiments by performing a preprocessing step. In our experiments with this game terrain, $k = 5, 10, 15, 20, 25$, we sample k start and goal configurations with Euclidian distance ≤ 25.

For grid graphs and the road network, we used a timeout of 1000 CPU seconds and limited the memory usage to 4GB. For the larger computer game dataset, we limited the memory usage to 10GB.

We used the ASP solver Clasp (Version 2.1.1) with the grounder Gringo (Version 3.0.5) on a machine with four 1.2GHz Intel Xeon E5-2665 8-Core Processors and 64GB RAM. We used Clasp in single-threaded mode with the command line `--configuration=handy` as this configuration performs best in the majority of cases. The reported CPU times are in seconds. Every figure is an average over 10 randomized instances.

We assumed that the maximum plan length l is identical for every agent. We started with $l = 30$ and increased it by 10 if the solver proves that there exists no plan for that plan length. We repeated this until $l = 80$ and above that we considered the problem unsolved.

5.1 Experiments on Artificial Grid Graphs

The goal of these experiments is to understand how the parameters of instances ($n \times n$: grid size, k: number of agents, o: percentage of obstacles) affect the computation time.

Table 1 shows results of our experiments, where we vary the number of agents and the percentage of blocked vertices in the grid graph. Every row in the table

Table 2. PF and **TPF** with combinations of **C**,**W**,**X** on random 25×25 grids with $k = 15$ agents and $o = 20\%$ obstacles. (Sorted by the last column.)

C	W	X	Grounding Time sec	Solver Time First Solution sec (length)	Solver Time Optimal Solution sec (length)	Overall Time to Optimal Solution sec
			3.30	2.56 (34.8)	19.09 (34.4)	22.39
	x		3.71	12.98 (38.1)	37.52 (34.4)	41.33
		x	11.24	3.35 (36.7)	32.19 (34.4)	43.43
x			3.64	18.74 (39.0)	48.96 (34.4)	52.60
x	x		3.85	27.82 (39.0)	66.31 (34.4)	70.16
	x	x	11.62	18.30 (39.0)	70.02 (34.4)	81.64
x		x	11.61	34.33 (38.6)	103.28 (34.4)	114.89
x	x	x	11.59	33.32 (39.0)	116.13 (34.4)	127.72

presents three CPU times in seconds, each averaged over 10 random instances: 1) for grounding, 2) for finding some solution (possibly non-optimal, but less than the given upper bound l on the plan length), and 3) for finding an optimal solution. The second CPU time (for finding the first solution) is included in the third CPU time (for an optimal solution). Averages of plan lengths (solution quality) are presented in parentheses. For instance with $o = 40\%$ of obstacles and $k = 20$ agents, GRINGO grounds the instances in 7.28 seconds, CLASP computes some solution to **PF** in 10.58 seconds with a plan length of 69.7; finding an optimal solution (of length 66.6) takes 45.30 seconds (all numbers are averages over 10 instances).

We can observe from Table 1 that increasing the number of agents increases both the grounding time and solving time (first and optimal solution). The number of obstacles on the other hand primarily influences the plan length of an optimal solution, e.g., with 20 agents the average optimal plan length is 33.8 steps for 20% obstacles, compared to 66.6 steps for 40% obstacles. While the plan length of an optimal solution increases with more obstacles, the time spent to find such a solution stays fairly stable between 10% and 20% obstacles and increases to less than twice with 50% obstacles. We conclude that solve time is primarily determined by the number of agents. This can be explained by the algorithm of the CLASP solver engine which performs very well on constrained search spaces.

5.2 Experiments with Constraints

The goal of these experiments is to understand the effect of constraints on the computation time and the solution quality (average plan length). We considered variations of **PF** with all combinations of **C**, **W**, and **X**, according to which paths must not have cycles, agents are forbidden to wait idle, and head-on collisions of agents are forbidden, respectively.

Table 2 shows the results of these experiments, sorted by the overall time to compute an optimal solution (i.e., by the sum of Grounding and Optimal Solution). **C** and **W** marginally increase grounding time. Contrary to what we could assume, **W** and **C** do not improve initial solution quality or reduce

Table 3. PF and **TPF** with variations of redundancy elimination (E) and circle heuristics (R) on random 25×25 grid graphs with $k = 15$ agents and $o = 20\%$ obstacles.

E	R	Grounding Time sec	Solver Time First Solution sec (length)	Solver Time Optimal Solution sec (length)	Unsolved Instances # (reason)
		3.48	2.44 (34.8)	19.32 (34.4)	–
x		16.71	5.23 (36.9)	56.24 (34.4)	–
	x	1.87	0.95 (37.1)	14.47 (34.5)	2 (no solution)
x	x	17.18	2.11 (38.4)	30.33 (34.5)	2 (out of memory)

search time by constraining the problem more, quite the contrary is the case. **X** significantly increases grounding time, however it has the least effect of all constraints on the time of finding initial solutions. All constraints increase the time to find optimal solutions significantly, their combinations increase that time even more. This can be explained by a more complex search space; our constraints do not seem to cut away significant portions of the search space but the result shows that proving optimality becomes harder.

In summary, adding constraints to **PF** will increase the time for finding solutions, for some constraints times increase more, for others they increase less.

Which constraints to add to the main formulation should be decided on a case-by-case basis depending on the actual application. For instance, since graph representations are discrete abstractions of an environment, and the computed discrete paths characterize the continuous trajectories followed by the agents, **X** may be ignored in some applications where the agents do not necessarily follow straight paths. It may not be possible for two agents to move in opposite directions on an edge (v, u) in the given graph, but it may be possible in the environment for one agent to move from v to u and the other agent to move from u to v following different continuous trajectories. On the other hand, for some other applications where robots move via narrow roads, **X** may be required.

We also experimented with other constraints: Adding **I** to the ASP formulation of **PF** increases computation time in many problems. Note that since **I** ignores the time of an agent visiting a vertex, it may be too strong for many **PF** applications. Adding **L** to the ASP formulation of **PF** similarly increases computation time in many problems.

5.3 Experiments with Heuristics

We utilized "circle heuristics" [6] to improve the computational efficiency in terms of computation time and consumed memory. Circle heuristics identifies for each agent a subgraph of the given graph that is more "relevant" for that agent to search for a path: we introduce two "circles" with a given radius around start and goal positions of the agent, and require that the path connecting start and goal is contained in the union of these circles. The radius can be defined as a constant or a function of some distance between start and goal positions. By preprocessing we identify relevant edges (v, u) of the graph for each agent i as facts of the form $\mathit{relevantEdge}(i, v, u)$, and replace in P all atoms of the form $\mathit{edge}(v, u)$ by $\mathit{relevantEdge}(i, v, u)$.

We also experimented with a "redundancy elimination" heuristics to eliminate certain redundant moves of agents; moving from vertex u to v via other vertices is not allowed if an edge (u, v) exists, except if the agent is waiting at u or v:

$$\leftarrow path(i,t,u), path(i,t',v), edge(u,v), not\ path(i,t{+}1,u), not\ path(i,t'{-}1,v).$$
$$(0 \leq t < t' < l_i, t{+}1 < t', 1 \leq i \leq k, v, u \in V, v \neq u)$$

This heuristics intuitively remove redundancies in paths, and improve the quality of solutions by making average plan lengths smaller. Note that there may still be some redundancies, if the specified maximum plan length is not small enough.

To analyze the effect of adding these heuristics, we considered randomly generated instances of **PF**, over 25×25 grid graphs with $o = 20\%$ obstacles and $k = 15$ agents. With the circle heuristics, for each agent i, we considered a radius of $\lceil \frac{ED_i}{2} \rceil + 3$ where ED_i is the Euclidean distance between start and goal locations of agent i.

Table 3 shows the results where each row shows average CPU time (and plan length) over the same set of 10 random instances that was used in experiments about constraints (Table 2). Here E and R denote redundancy elimination and circle heuristics. We can observe that, since the redundancy elimination heuristic adds further constraints to the problem, the grounding time increases. These constraints, furthermore, do not constrain the search space enough to allow the solver engine to find solutions faster; therefore, the solution time also increases. Contrary to what we expected, solution quality also becomes worse with redundancy elimination. The additional constraints added seem to be misleading for the solver, resulting in worse solution quality and worse efficiency.

With the circle heuristics, only some parts of the graph are considered while computing a solution; therefore, this heuristics significantly reduces both the grounding time and the time to find an optimal solution. Recall, however, that with a small value of the radius, the circle heuristics is neither sound nor complete: the optimal solution found for **PF** with circle heuristics may not be an optimal solution for **PF**; also there may be instances of **PF** that have some solutions, but using the circle heuristics eliminates all solutions (as observed in two instances in Table 3).

In summary, it seems to be a good idea to use the circle heuristics, even though it is not complete (the solution quality is only marginally different); on the other hand, redundancy elimination does not help for improving computation time or solution quality.

5.4 Experiments on a Real Road Network

The results of our experiments with randomly generated instances of **PF** on the road network are presented in Table 4. We performed experiments with and without circle heuristics.

With an increasing number of agents, grounding time does not increase as fast as the time to prove optimality of the solution. Finding an initial solution is always very fast in these instances, and the plan length averages indicate that these initial solutions are often already optimal.

Table 4. PF and **TPF** with and without circle heuristics (R) on a road network graph.

k	R	Grounding sec	First Solution sec (length)	Optimal Solution sec (length)
5		1.01	0.30 (27.7)	9.47 (24.4)
10		2.36	0.90 (30.7)	17.43 (29.4)
15		6.62	2.12 (33.7)	27.86 (32.5)
20		10.95	3.50 (33.2)	41.36 (32.7)
25		19.13	7.04 (37.0)	67.09 (34.6)
5	x	1.02	0.16 (29.5)	4.97 (24.4)
10	x	1.39	0.59 (32.0)	11.65 (29.4)
15	x	3.30	1.14 (35.6)	18.16 (32.5)
20	x	4.42	1.68 (34.9)	22.35 (32.7)
25	x	7.60	4.13 (39.2)	37.73 (34.6)

The time to prove optimality is much greater than the time for finding the initial solution, and the quality of the initial and the optimal solution are very close to each other. Therefore, on a road network, it might be advantageous to try to find some solution (rather than an optimal one).

Circle heuristics performs very well in this benchmark: it significantly reduces grounding time and time to find the first and the optimal solution, while it does not reduce the solution quality: optimal solution lengths are the same with and without usage of (R).

5.5 Experiments on a Real Game Terrain

The results of our experiments with randomly generated instances of **PF** on a game terrain are presented in Table 5. Again we use consider the usage of circle heuristics.

Due to the larger grid size, the problem instances are also bigger in terms of number of facts that describe the map. Therefore grounding consumes a lot of time for this setting. On the other hand, for applications where the environment (e.g., game terrain map) does not change, we can do grounding only once and reuse the ground ASP program for different problem instances in the same environment.

Due to larger instance sizes, the memory requirements are high: for other benchmarks memory stays below 4GB, whereas the largest computer game instance requires 13.4GB without circle heuristics (3.1GB with circle heuristics).

In this domain, an initial solution is found in around a second, whereas finding an optimal solution and proving its optimality take a significant amount of time. (Nevertheless, the time to find an optimal solution is still dominated by grounding time.) Fortunately, the average plan length of initial solutions does not differ much from the average plan length of optimal solutions. Therefore, as noted with the road network benchmark, computing only the initial solution might be sufficient in practice.

Note that instances with 20 agents appear to be easier than those with 15 agents; this is an effect of random generation of instances which created more instances with long solution paths for 15 agents than for 20 agents.

Table 5. PF and **TPF** with and without circle heuristics (R) on game map.

k	R	Grounding sec	First Solution sec (length)	Optimal Solution sec (length)	Memory MB
5		4.41	0.88 (34.8)	24.68 (29.0)	919.4
10		10.02	2.32 (33.0)	46.11 (33.0)	2168.5
15		25.82	6.16 (45.7)	137.64 (45.7)	5843.7
20		17.05	3.87 (35.3)	85.05 (35.3)	4416.4
25		30.45	7.19 (40.3)	161.21 (40.3)	7728.8
5	x	22.94	0.29 (34.8)	9.07 (29.0)	449.8
10	x	56.48	0.81 (34.7)	23.14 (33.0)	975.3
15	x	109.09	1.25 (45.7)	29.45 (45.7)	1763.7
20	x	80.76	1.14 (35.3)	26.71 (35.3)	1588.6
25	x	119.90	1.82 (40.3)	40.48 (40.3)	2365.3

The usage of circle heuristics is clearly advantageous in this setting; it does not reduce solution quality but it reduces memory and time usage.

6 Related Work

Our formal framework for **PF** is general enough to solve variations of pathfinding problems with multiple agents, including multi-agent pathfinding (MAPF), multi-robot routing, etc. Many of these variants have been studied in the literature; thus a comprehensive comparison with the existing approaches is not possible within limited space. Therefore, we briefly discuss related work on MAPF, and report some preliminary experimental results.

Most of the existing solutions to MAPF apply some sort of A* search algorithm, with *decoupled pathfinding* or *centralized pathfinding* approach. In the former approach [23, 4, 33, 10], a path is computed for each agent independently; in case a conflict occurs (e.g., two agents attempt to move to the same location), it is resolved by replanning one of the conflicting agents' route. Although this approach could be used to solve large MAPF instances quickly, it lacks the optimality and completeness guarantees. The latter approach [20, 29, 25] considers the multi-agent system as a single-agent system by combining state spaces of each agent into one state space and then use a search algorithm to find paths for all agents simultaneously. Although the centralized approach can guarantee optimality and completeness, it is not as efficient (in terms of computation time) as the decoupled approach for large problems. More recently, some decoupled pathfinding algorithms [15, 34] are introduced to guarantee completeness for some graphs; and some [26] optimality. Some centralized planners [21, 12] use heuristics to find suboptimal solutions to improve computational efficiency. Our approach to MAPF is centralized, guarantees various sorts of optimality (thanks to elaboration tolerant representation of **PF** and various constraints), soundness and completeness (Theorem 1). As observed from experiments, with some heuristics, the computational efficiency can be improved also.

Some of the related work on MAPF considers grid-based graphs [23, 4, 25], and some consider trees [12]. Our approach is applicable to arbitrary graphs.

It is important to emphasize here that, unlike our ASP-based approach, the search-based methods above do not provide a formal framework; and thus it is hard to ensure and verify various properties (e.g., constraints mentioned above) over paths unless the modeling of the problem and the implementation of the algorithm are modified for each case.

One of the closest related work to ours is [36]: like our approach, the authors introduce a formal framework for solving MAPF to minimize overall plan length on arbitrary graphs with a centralized approach, but using integer linear programming (ILP) instead of ASP. Differently to Yu and Lavalle, our approach is general enough to solve variations of **PF**, some of which are not (or not easily) representable in ILP (e.g., acyclicity constraints). We compared the ILP approach to ours using 180 randomly generated **TPF** instances of 25x25 grid graphs with 0-40% obstacles, and with either 10 or 20 agents (10 instances for each configuration). We used 1000 seconds timeout. We observed that memory usage was higher for ILP (<10GB) than for ASP (<4GB). With 10 agents, ILP found optimal solutions faster (3 seconds) than ASP (11 seconds); average plan length was 27 steps. With 20 agents, ILP did not return a solution for 7 of the 180 instances and solved the remaining instances in 42 seconds on average (deviation 74 seconds), while ASP timed out only for 2 of 180 instances and found optimal solutions for the other instances in 50 seconds on average (deviation 36 seconds); average plan length was 30 steps. We observed that an increased amount of obstacles degrades both ILP and ASP performance, however ILP performance degrades much stronger. This is because ILP is based on linear optimization with additional support for boolean variables, while ASP is well-suited for finding solutions to highly constrained boolean search spaces.

Although the approaches are quite different, we also compared our (centralized, complete, optimal) ASP-based approach with the state-of-the-art (decoupled, incomplete, nonoptimal) MAPF solver MAPP [34] with some randomly generated instances of the game Baldur's Gate, described in [34].[1] Preliminary results conform with our expectations (as also observed in previous studies comparing decoupled and centralized approaches): in terms of computation time, MAPP performs better than our approach as the number of agents and the size of the grid increase; on the other hand, some problems with multiple conflicts cannot be solved by MAPP, while they can be solved by our approach. A more detailed comparison is a part of our ongoing work.

It is important to pinpoint here that, since our ASP-based formal framework is general enough to solve many variations of the multi-agent pathfinding problem, thanks to the high-level representation language of ASP, it can be viewed as complementary to the existing approaches.

[1] Currently, MAPP is the only operational search-based state-of-the-art MAPF solver made available to us by its authors (December 2012). PUSH_AND_SWAP [15] is not available since it is being revised by the authors (personal communication, December 2012). WHCA*(w,a) [28] (an enhancement of WHCA* [23]) provided to us by its authors is not operational since it has not been maintained for a while (personal communication, December 2012).

7 Discussion and Conclusion

We have introduced a general formal framework to solve various pathfinding problems with multiple agents (**PF**), using ASP. We have shown that, due to the expressive formalism of ASP, we can easily represent **PF** and its variations subject to different constraints on the paths, and heuristics to improve computational efficiency and quality of solutions. Such a flexible elaboration tolerant framework is important in studying and understanding **PF** and its applications in different domains (e.g., motion planning, vehicle routing, environmental monitoring, patrolling/surveillance, computer games). In particular, that our framework can be applied to any sort of graphs (e.g., not necessarily grid graphs or trees) is advantageous for various robotic applications.

Acknowledgments

Thanks to anonymous reviewers for their useful suggestions on improving the presentation of the paper. This work is partially supported by TUBITAK Grant 111E116. Peter Schüller is supported by TUBITAK 2216 Fellowship.

References

1. M. Bennewitz, W. Burgard, and S. Thrun. Finding and optimizing solvable priority schemes for decoupled path planning techniques for teams of mobile robots. *Robotics and Autonomous Systems*, 41(2–3):89–99, 2002.
2. G. Brewka, T. Eiter, and M. Truszczynski. Answer set programming at a glance. *Commun. ACM*, 54(12):92–103, 2011.
3. E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. *ACM Computing Surveys*, 33(3):374–425, 2001.
4. K. M. Dresner and P. Stone. A multiagent approach to autonomous intersection management. *J. Artif. Intell. Res. (JAIR)*, 31:591–695, 2008.
5. E. Erdem, D. G. Kisa, U. Oztok, and P. Schueller. A general formal framework for pathfinding problems with multiple agents. In *Proc. of AAAI*, 2013.
6. E. Erdem and M. D. F. Wong. Rectilinear steiner tree construction using answer set programming. In *Proc. of ICLP*, pages 386–399, 2004.
7. S. T. Erdogan and V. Lifschitz. Definitions in answer set programming. In *Proc. of LPNMR*, pages 114–126, 2004.
8. M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. clasp: A conflict-driven answer set solver. In *Proc. of LPNMR*, pages 260–265, 2007.
9. M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.
10. R. Jansen and N. Sturtevant. A new approach to cooperative pathfinding. In *Proc. of AAMAS*, pages 1401–1404, 2008.
11. J. Jennings, G. Whelan, and W. Evans. Cooperative search and rescue with a team of mobile robots. In *Proc. of ICAR*, pages 193–200, 1997.
12. M. M. Khorshid, R. C. Holte, and N. R. Sturtevant. A polynomial-time algorithm for non-optimal multi-agent pathfinding. In *Proc. of SOCS*, 2011.
13. H. Kitano, S. Tadokoro, I. Noda, H. Matsubara, T. Takahashi, A. Shinjou, and S. Shimada. Robocup rescue: Search and rescue in large-scale disasters as a domain for autonomous agents research. In *Proc. of IEEE SMC*, pages 739–746. IEEE Computer Society, 1999.

14. V. Lifschitz. What is answer set programming? In *Proc. of AAAI*, pages 1594–1597. MIT Press, 2008.
15. R. Luna and K. E. Bekris. Efficient and complete centralized multi-robot path planning. In *Proc. of IROS*, pages 3268–3275, 2011.
16. A. Machado, G. Ramalho, J.-D. Zucker, and A. Drogoul. Multi-agent patrolling: An empirical analysis of alternative architectures. In *Proc. of MABS*, pages 155–170, 2002.
17. J. McCarthy. Elaboration tolerance. In *Proc. of Commonsense*, 1998.
18. L. Pallottino, V. G. Scordio, A. Bicchi, and E. Frazzoli. Decentralized cooperative policy for conflict resolution in multivehicle systems. *IEEE Transactions on Robotics*, 23(6):1170–1183, 2007.
19. D. Ratner and M. K. Warmuth. Finding a shortest solution for the n × n extension of the 15-puzzle is intractable. In *Proc. of AAAI*, pages 168–172, 1986.
20. M. Ryan. Exploiting subgraph structure in multi-robot path planning. *JAIR*, 31:497–542, 2008.
21. M. Ryan. Constraint-based multi-robot path planning. In *Proc. of ICRA*, pages 922–928, 2010.
22. G. Sharon, R. Stern, M. Goldenberg, and A. Felner. The increasing cost tree search for optimal multi-agent pathfinding. In *Proc. of IJCAI*, pages 662–667, 2011.
23. D. Silver. Cooperative pathfinding. In *Proc. of AIIDE*, pages 117–122, 2005.
24. B. S. Smith, M. Egerstedt, and A. Howard. Automatic generation of persistent formations for multi-agent networks under range constraints. *Mob. Netw. Appl.*, 14(3):322–335, 2009.
25. T. S. Standley. Finding optimal solutions to cooperative pathfinding problems. In *Proc. of AAAI*, 2010.
26. T. S. Standley and R. E. Korf. Complete algorithms for cooperative pathfinding problems. In *Proc. of IJCAI*, pages 668–673, 2011.
27. N. Sturtevant. Benchmarks for grid-based pathfinding. *Transactions on Computational Intelligence and AI in Games*, 2012.
28. N. R. Sturtevant and M. Buro. Improving collaborative pathfinding using map abstraction. In *Proc. of AIIDE*, pages 80–85, 2006.
29. P. Surynek. An application of pebble motion on graphs to abstract multi-robot path planning. In *Proc. of ICTAI*, pages 151–158, 2009.
30. P. Surynek. An optimization variant of multi-robot path planning is intractable. In *Proc. of AAAI*, 2010.
31. P. Svestka and M. H. Overmars. Coordinated path planning for multiple robots. *Robotics and Autonomous Systems*, 23(3):125–152, 1998.
32. C. Tomlin, G. J. Pappas, S. Member, S. Member, and S. Sastry. Conflict resolution for air traffic management: A study in multiagent hybrid systems. *IEEE Transactions on Automatic Control*, 43:509–521, 1998.
33. K.-H. C. Wang and A. Botea. Fast and memory-efficient multi-agent pathfinding. In *Proc. of ICAPS*, pages 380–387, 2008.
34. K.-H. C. Wang and A. Botea. MAPP: a scalable multi-agent path planning algorithm with tractability and completeness guarantees. *J. Artif. Intell. Res. (JAIR)*, 42:55–90, 2011.
35. L. Xu and A. Stentz. An efficient algorithm for environmental coverage with multiple robots. In *Proc. of ICRA*, pages 4950–4955, 2011.
36. J. Yu and S. M. LaValle. Planning optimal paths for multiple robots on graphs. In *Proc. of ICRA*, 2013.