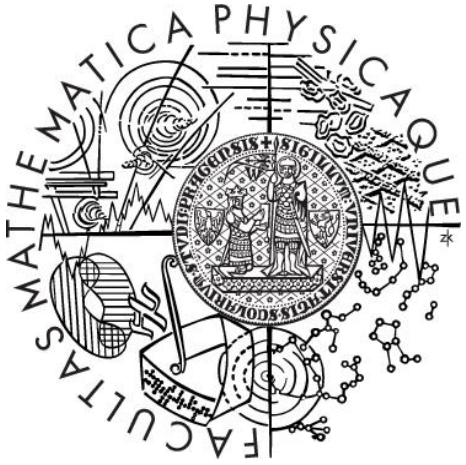
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Marika Ivanová

Adversarial Cooperative Patfinding

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis: RNDr. Pavel Surynek, Ph.D.
Study programme: Informatics
Specialization: Theoretical Computer Science

Prague 2014

Acknowledgements

Presented master thesis is focused on a novel problem based on Cooperative Pathfinding. Work on this thesis started at the beginning of 2013 and so far its results have been published in

- Marika Ivanová and Pavel Surynek. Adversarial cooperative path-finding: A first view. AAAI Late Breaking Track, 2013
- Marika Ivanová and Pavel Surynek. Adversarial Cooperative Path-finding: Complexity and Algorithms. In press, ICTAI 2014.

In the first publication we introduced the problem and described its complexity. The paper was the only contribution from Czech Republic on AAAI 2013. Second paper containing detailed proofs of the problem complexity with all technical details and experimental evaluation were recently accepted as a regular paper on ICTAI 2014 conference, which is going to take place in November 2014. There is a wide range of further research and so more publications can be expected.

I am very grateful to my supervisor RNDr. Pavel Surynek, Ph.D. for his guidance, endless patience and willingness to give me valuable advices regarding all aspects of the thesis. Neither this thesis nor the publications would ever arise.

I would also like to thank prof. Jan Arne Telle from University in Bergen for his suggested books concerning complexity of the studied problem.

I must not forget to thank my family and relatives that supported me all the time and enabled me to study.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... signature of the author

Název práce: Kooperativní hledání cest s protivníkem

Autor: Marika Ivanová

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí diplomové práce: RNDr. Pavel Surynek, Ph.D., Katedra teoretické informatiky a matematické logiky

Abstrakt: Předložená práce definuje a zkoumá problém kooperativního hledání cest s protivníkem (adversarial cooperative path finding - ACPF), který představuje zobecnění známé úlohy kooperativního hledání cest. Oproti standardní kooperativní verzi, v níž je úkolem najít nekolidující cesty pro několik agentů spojující jejich počáteční a cílové pozice, ACPF uvažuje navíc agenty ovládané protivníkem. Práce se zaměřuje jak na teoretické vlastnosti, tak na praktické techniky řešení uvažovaného problému. Úlohu ACPF zavádíme formálně pomocí pojmů z teorie grafů a zkoumáme její výpočetní složitost, kde ukazujeme, že úloha je PSPACE-těžká a patří do třídy EXPTIME. Představujeme a diskutujeme možné metody vhodné pro praktické řešení ACPF. Uvažované řešící postupy zahrnují hladové algoritmy, minimaxové metody, Monte Carlo Tree Search a adaptaci algoritmu pro kooperativní verzi. Z provedeného experimentálního vyhodnocení vyplývá mimo jiné překvapivě častá úspěšnost hladových metod a spíše slabší výsledky u Monte Carlo Tree Search.

Klíčová slova: Kooperativní hledání cest, protivník, Monte Carlo Tree Search

Title: Adversarial Cooperative path-finding

Author: Marika Ivanová

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: RNDr. Pavel Surynek, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: Presented master thesis defines and investigates Adversarial Cooperative Path-finding problem (ACPF), a generalization of standard Cooperative Path-finding. In addition to the Cooperative path-finding where non-colliding paths for multiple agents connecting their initial positions and destinations are searched, consideration of agents controlled by the adversary is included in ACPF. This work is focused on both theoretical properties and practical solving techniques of the considered problem. ACPF is introduced formally using terms from graph theory. We study computational complexity of the problem where we show that the problem is PSPACE-hard and belongs to EXPTIME complexity class. We introduce and discuss possible methods suitable for practical solving of the problem. Considered solving approaches include greedy algorithms, minimax methods, Monte Carlo Tree Search and adaptation of algorithm for the cooperative version of the problem. Surprisingly frequent success rate of greedy methods and rather weaker results of Monte Carlo Tree Search are indicated by the conducted experimental evaluation.

Keywords: Cooperative Path-finding, adversarial, Monte Carlo Tree Search

Contents

Introduction

This master thesis addresses adversarial cooperative path-finding (ACPF), a problem from artificial intelligence field.

ACPF can be regarded as a generalization of cooperative path-finding (CPF) [30] which is extended by adversarial element outside our control. The standard cooperative path-finding is a path planning problem in a fully observable static environment where agents must find non-colliding routes to given separate destinations. All the agents are controlled centrally while agents themselves make no decisions. CPF is motivated by problems arising in both real environments and virtual worlds (multi-robot navigation, container relocation, path-finding in computer games [37]).

The notion of CPF has however limited applicability as not all the environments are fully cooperative. That is, we cannot regard all the agents as controllable and the environment as static any more. Dealing with such adversarial or hostile elements in the environment beyond the standard CPF is thus desirable.

Our suggestion is to introduce two or more teams of agents that compete in finding paths to target destinations to model the adversarial element in CPF. The objective in the ACPF problem is to control agents of one selected team so that its agents reach their destinations before agents of adversarial teams. There is a wide range of possibilities how teams of agents can harm each other (occupying target destination, route blocking, agent blocking). Hence, combinatorial difficulty in ACPF comes not only from the need of avoidance but also from the need to consider possible harmful actions of adversaries.

The aim of presented work is to investigate the ACPF problem from theoretical as well as practical perspective. Initially we will concentrate on formal definition of ACPF. Subsequently we will study theoretical properties of the problem, particularly its computational complexity.

Investigation of possible solving methods and their experimental verification is another objective of this work. As the problem is related to n-player games and cooperative path-finding, we will also focus on methods developed for these similar problems and consider their adaptation and application for ACPF. Conducted experiments shall give us a closer insight into suitability of the suggested methods in various ACPF instances. We would like to find out what approaches are appropriate for different types of instances.

Solving ACPF can be approached from two different directions: combinatorial games and path-finding. In terms of artificial intelligence [23], ACPF is an n-player zero-sum sequential and symmetric game with perfect information. We can also compare ACPF to popular abstract strategy games like chess or checkers.

As a starting point we propose two greedy methods. Natural characteristics of our problem immediately suggests application of existing techniques used in diverse game-like applications. Minimax algorithm with alpha-beta pruning is a traditional algorithm used for developing computer programs playing board games. Monte Carlo methods offer an alternative approach, that was successfully applied to computer go [6] and hex [1].

From the path-finding perspective we can be inspired by algorithms used for multi-agent path-finding in non-adversarial environments. There is a possibility to adjust these techniques and employ them in solving ACPF instances. One such algorithm with few different versions is introduced and tested in this work.

Document Structure

The first chapter gives a detailed description of multi-agent path-finding, in particular cooperative path-finding since these tasks are closely related to the studied ACPF problem.

Beginning of the second chapter is dedicated to introduction of the adversarial element present in the studied problem. We provide a formal definition of ACPF using terms from graph theory. Based on the introduced formalism we study computational complexity of the problem and try to determine its membership in complexity classes. Results of this thesis were partially published at AAAI conference [16].

Remaining two chapters are focused on practical solving methods. The third chapter discusses possible variants of greedy algorithms, describes common techniques known from games and introduces an algorithm inspired by cooperative path-finding. Final chapter consists of description of conducted experiments and presentation of their results.

1. Multi-agent Path-finding

In this chapter we provide a general knowledge of multi-agent path-finding and take a closer look at its special case known as cooperative path-finding. In the end of the chapter we briefly summarize information about related work on similar topics.

Basic path-finding is a well known task in computer science. The objective is to find a route between two selected points. We will also use term *source* and *target* to refer to these two points.

Let us consider an environment with several identical moving entities called *agents*. Source and target locations are determined for each agent. The objective is to find a route for every agent from its initial position to a given target location. Agents must not collide with obstacles and other agents that are also moving along planned routes towards their targets.

In the further text, we will use several abstractions in order to represent the environment and model the movement of the agents. The environment is modeled as an unweighted undirected graph. Agents can move along edges from one vertex to another. Continuous time is divided into discrete time steps, where the relocation from one vertex to its neighbor takes exactly 1 time step.

1.1 Basic Definitions and Properties

We recognize a few similar tasks and variants of path-finding for multiple agents that slightly differs in various aspects from each other.

Muliti-agent path-finding (MPF) [11] supposes a group of agents in a given environment, where initial and target positions are determined for each agent. All individual agents must avoid collisions. Movement of the agents is carried out in discrete time steps. Agent can shift from one vertex to its neighbor on condition that the neighbor is either unoccupied or is being left by other agent in the same time step. At most one agent is allowed to pass an edge within one time step. That is, agents are not allowed to exchange their positions within one time step.

Pebble motion on graphs [19, 36] is a very similar problem as MPF. It can be regarded as a restricted variant of MPF. The difference consists in rules for movement. While MPF enables entering a vertex that is simultaneously being left by other agent, such transfer is not permissible in pebble motion. As an illustration we can mention 15 puzzle also known as *Lloyd's 15*.

Cooperative Path-finding [30] is a special case of MPF where each agent is assumed to have full knowledge of all other agents and their planned routes. Precisely speaking, solving algorithm can take into account paths[1] planned for agents that were processed earlier and adjust paths searched later according to them.

[1]Strictly formally speaking, the term path is not fully accurate in this place. In the graph theory, a term path is used when we want to address a sequence of adjacent vertices without repeating the same vertex. In our context, vertices may repeat in one sequence, nevertheless we will call it path, because it corresponds to usual phrase "path-finding".

Several every-day situations can be modeled by CPF. In traffic control we need every car to safely pass a crossroad in a short time. Another example might be found in logistics, where some objects are being simultaneously relocated. Planning of data transfer between communication nodes can also be expressed as a CPF instance [37]. Military operations can utilize CPF as a simulation of an effective movement. There is also very wide usage of CPF in game industry. Unit movement from one place to another, raw material extraction or transportation in real-time strategical games and many others.

For completeness we provide a formal definition of the multi-agent path-finding problem.

Definition. *Multi-agent path-finding problem* is a quadruple

$$\Sigma = (G, A, \lambda_0, \lambda_+)$$

where the symbols have following meaning:

$G = (V, E)$ is an unweighted undirected graph

$V = \{v_1, v_2, \ldots, v_n\}$ denotes a finite set of vertices

$E \subseteq \binom{V}{2}$ denotes a set of edges

$A = \{a_1, a_2, \ldots, a_k\}$ is a a finite set of agents

$\lambda_0 : A \to V$ is an injective mapping that assigns an initial vertex to each agent

$\lambda_+ : A \to V$ denotes an injective mapping assigning a target vertex to each agent

1.2 Theoretical properties

Unlike single-robot path-finding, where a path from a source to a target exists whenever a graph is connected, existence of a solution in multi-robot path-finding is not granted. In some cases, agents can obstruct each other. Let us imagine a simple instance comprised of a path graph with agents seated at the ends of the path and a target of each agent is located at the other agent's initial vertex. Such situation is clearly insoluble, because agents cannot get round each other and hence will never reach their target destinations.

Regarding the time complexity, it is known that the pebble motion problem and thus multi-robot path-finding can be solved in polynomial time $\mathcal{O}(|V|^3)$ and the solution consists of $\mathcal{O}(|V|^3)$ moves [19, 37].

The objective is sometimes to find an optimal solution for a MPF instance. Let $P = \{p_1 \ldots p_k\}$ be a set of paths found for k agents in an arbitrary solution and let t_i denote an arrival time of agent a_i. There are three common objectives that can be optimized [42]:

- Minimum total arrive time: $\min \sum_{i=1}^{k} t_i$
- Minimum makespan: $\min \max_{1 \leq i \leq k} t_i$

- Minimum total distance: $\min \sum_{i=1}^{k} len\,(p_i)$

These optimization variants can be modified into decision problems. For a given natural number η we ask whether there exists a solution with total arrive time (makespan or total distance) not greater than η. It was showed that the decision problems are NP-hard [42]. A proof of NP-completeness of the decision variant, particularly the case with makespan, were presented in [37].

1.3 Selected Algorithms

Methods for multi-robot path-finding can be divided into coupled (centralized) and decoupled (decentralized) approaches [39]. Coupled methods regard whole group of agents as a single entity and try to plan all their paths at the same time. Although this approach is theoretically optimal, excessive complexity prevents its wide practical usage. On the contrary, decoupled methods decompose the problem into several sub-problems. Their strategy is to find paths for individual agents independently and subsequently resolve possible collisions that might occur. Decoupled approach is neither optimal nor complete. Nevertheless, it significantly reduces the time complexity.

A typical decoupled method employs A* algorithm [12, 34] for finding paths for every agent independently. A* is similar to Dijkstra's algorithm [10] suitable for weighted graphs. These algorithms finds shortest path between two vertices in a given graph. The difference consists in employing heuristic estimation of a distance between source and target vertex in A*. Basic principle of A* is described by the algorithm 1 below.

Algorithm 1 A* algorithm

```
function search(source, target)
  create OPEN list, OPEN ← {source}
  create CLOSED list, CLOSED ← {∅}
  while OPEN ≠ ∅ do
    v ← remove v with the lowest cost from OPEN
    if v is equal to target then
      return reconstruct_path(target)
    end if
    CLOSED ← CLOSED ∪ {v}
    for all s ∈ Successors(v) do
      if s ∉ OPEN then
        if s ∉ CLOSED then
          estimate cost for s
          parent(s) ← v
          OPEN ← OPEN ∪ {s}
        end if
      end if
    end for
  end while
  return failure
```

A* algorithm considers following values for every vertex v:

- $g(v)$, a distance from s to v,
- $h(v)$, heuristic estimation of a distance between v and target t
- $f(v) = g(v) + h(v)$

Heuristic function should be consistent (or monotone) in order to algorithm be optimal. In other words, if vertex u is a successor of vertex v and $dst(u, v)$ is a distance between them, then $h(v) \leq h(u) + dst(u, v)$. Dijkstra's algorithm always considers zero heuristic value, which is indeed consistent, but the search is not directed towards the target vertex and therefore takes longer time.

Solving adversarial extension of CPF were inspired by several cooperative algorithms. Their description is provided here.

1.3.1 Local Repair A*

One of the simplest path-finding algorithm for multiple agents is known as Local repair A* (LRA*) [43]

When paths for all the agents are found using A*, it is necessary to check whether agents will collide in following planned steps. If a collision is encountered, additional calculation have to be conducted in order to repair paths of the colliding agents. Parts of the planned paths are gradually prolonged until the whole pre-calculated movement is legal.

LRA* calculation can take very long time, especially in ragged graphs or with a high density of agents. This algorithm does not expect any shared information between agents, only their current positions.

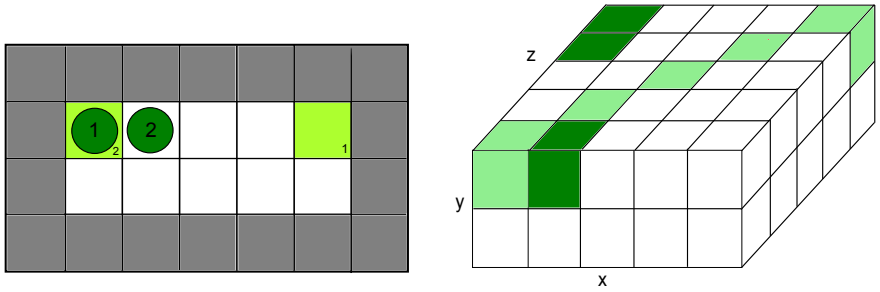
Once we have an information about paths planned for previously processed agents, we can incorporate this knowledge in further calculations. Agents whose paths are calculated later, respect the scheduled movement of the previously considered agents.

1.3.2 Cooperative A*

Cooperative A* (CA*) [30] is a decoupled algorithm, that works with three dimensional space-time (spatial graph). We can imagine a spatial graph as t copies (levels) of the original graph $G_0 = (V_0, E_0)$. These levels are piled up and represent time dimension of the movement of the agents. For this reason the edges between vertices belonging to one level are removed while neighboring levels are connected according to the edges in G_0. Spatial graph is directed, because agents must not be allowed to move back in time. If a vertex in the original graph is a target of some agent, all corresponding vertices in the spatial graph are also considered as the agent's targets, because they represent the same vertex in the original graph.

Suppose $(u_0, v_0) \in E_0$ undirected and u_1 and v_1 are corresponding vertices in the next level. Directed edges in the spatial graph will then be (u_0, v_1), (v_0, u_1), (v_0, v_1) and (u_0, u_1). Corresponding vertices in adjoining levels are always connected by directed edge, which ensures that the agent can remain at a vertex

for several time steps. Figure 1.1a shows an exemplary initial position[2] with two cooperating agents and possible spatial graph with planned paths (1.1b).



(a) Initial position **(b)** Spatial graph

Figure 1.1: An exemplary instance with two cooperating agents (a) and possible spatial graph with reservations

Algorithm planned path for agent 1 first, directly from upper left corner to the upper right corner, ignoring the fact that agent 2 currently occupies neighboring location. When the agent 2 is processed, first agent's planned path has to be taken into account and the solving algorithm must make way for it. Hence second agent will make 3 steps: down, left and up to its target position. Reservations on [2, 1, 2] and [1, 1, 3] are not visible in the figure (b).

Having this structure we can place agents to their initial vertices in the lowest level and run A* algorithm for each agent one by one. Once a path for an agent is found in the spatial graph, vertices on the path are reserved in time steps when the agent should enter these vertices and all other agents of which paths are calculated later will have to avoid them in the specified time steps.

The resulting paths depend on the order in which the solving algorithm calculated movement of the agents. In some cases, this approach will not find all paths, although the solution exists. Consider the situation in the figure 1.2. If the movement of agent a_1 is planned as first, it reaches its target $\lambda^+(a_1)$ after two time steps. However, another agent a_2, that was processed afterwards, cannot get into its destination $\lambda^+(a_2)$, because the first agent standing on its target blocks the bottleneck with the path leading to $\lambda^+(a_2)$. Hence there will be no path leading from second agent's initial position to its target in the corresponding spatial graph. The solution is to process agent 2 before agent 1. In such situation we have to change the agent ordering and start whole calculation again. In the worst case we have to do $k!$ restarts, where k is the number of robots, which is inconceivable. Some approaches how to reduce this complexity were proposed, for example constructing a priority scheme [3], which in practice allows to find the best ordering faster, but does not guarantee finding the most suitable ordering in every case.

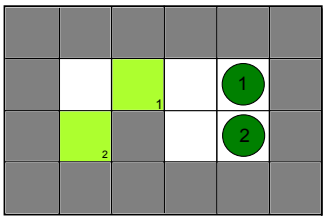


Figure 1.2: Example of blocking agents

[2]Explanation of diagrams used in this text is to be found in the Appendix C 4.2.5

1.3.3 Hierarchical Cooperative A*

As the algorithms above are based on A* algorithm, some admissible heuristic is necessary. The idea of the Hierarchical Cooperative A* (HCA*) [30] is to compute heuristic on demand, which is more appropriate in a dynamic context. HCA* algorithm is a generalization of Hierarchical A* [13] for the cooperative context. The heuristic is calculated using a domain abstraction, where the time dimension and other agents are ignored and heuristic is a distance from source to target in this simplified environment.

1.3.4 Windowed Hierarchical Cooperative A*

There are several difficulties with the algorithms mentioned so far. Besides already discussed problem with agent blocking narrow corridors and sensitivity to the agent ordering, there is also problem with computing whole route in a large spatial graph, moreover, since the environment is dynamic, precomputed paths are rarely used till the end. Usually they have to be re-planned several times during the whole process, especially in crowded environments.

Windowed Hierarchical Cooperative A* (WHCA*) [30] tries to overcame this issue. Paths are not fully calculated, but the computation is limited to a fixed depth of w steps (window size). In other words, we execute classic cooperative search, but only for first w steps. Beyond this limit, all other agents are ignored and only abstract search as used in HCA* is performed. In fact, the path is fully calculated, but interaction with other agents is considered only w steps ahead. In addition, the windowed search can continue once the agent has reached its destination. The agent's goal is no longer to reach the destination, but to complete the window via a terminal edge.

1.3.5 Summary of CPF methods

Besides algorithms described above, approaches for solving MPF were being designed and studied in recent decades and the research still continues. Designed algorithms sometimes require special types of graph or have specific requirements for agents.

Operator Decomposition and Independence Detection (OD+ID) techniques were proposed in [32]. This algorithm is optimal, complete and anytime, but prematurely terminated runs of the algorithm return only conflicting paths. Furthermore its running time is prohibitively expensive for many practical applications. [33] presents an enhancement of this method.

An alternative approach using so called *directional maps* for solving CPF were suggested in [17, 18].

Generally incomplete, algorithm MAPP [40] has favourable polynomial worst-case upper bounds for the running time, the memory requirements and the length of solution. MAPP is complete for some special classes of problems.

Two multiflow models that compute minimum last arrival time and minimum total distance respectively were introduced in [42].

Polynomial algorithm BIBOX [35] was designed for MPF on bi-connected graphs with at least two unoccupied vertices. Time complexity of BIBOX is $\mathcal{O}(|V|^3)$.

Graph decomposition [24] produced another sub-optimal MPF algorithm. Planning then becomes a search in the much smaller space of subgraph configurations.

FAR (Flow Annotation Replanning) [14] is a method designed for grid graphs. FAR solves problems more quickly and requires less memory then WHCA*. As many other approaches, FAR trades the completeness for an improved efficiency.

Push and Swap [22] is another CPF algorithm that claims completeness, however, in [8] were identified instances for that this algorithm fails to find a solution. Algorithm Push and Rotate is then presented as an adaptation of the Push and Swap technique. By fixing the Push and Swap's shortcomings, authors obtained an algorithm that is complete for the class of instances with two unoccupied locations in a connected graph.

[4] is focused on coordinating *self-interested agents*. A self-interested agent in MPF always chooses to follow the path with the best individual social welfare (minimum sum of tax and travel costs).

Alternative approach introduces a two-level Increasing Cost Tree Search (ICTS) algorithm [29]. ICTS is consequently compared with A* based techniques. Another algorithm capable to solve wider range of instances than ICST is called Conflict-based Search (CBS) [28]. All low-level searches in CBS are performed as single-agent searches.

Following table 1.1 summarizes properties of several algorithms mentioned in this section. The second column expresses whether an algorithm is optimal relative to at least one objective (total distance, makespan, or total arrival time).

Algorithm	Optimal	Complete	Coupled (C)/ Decoupled (D)	Requirements on graph
(WH)CA* [30]	✗	✗	D	Arbitrary
OD+ID [33]	✓	✓	D	Arbitrary
BIBOX [35]	✗	✗	C	Biconnected, 2 blanks
MAPP [40]	✗	✗	D	Grid graph
FAR [14]	✗	✗	D	Grid graph
Push&Rotate [8]	✗	✓	C	2 blanks
CBS [28]	✓	✓	C	Arbitrary

Table 1.1: Properties of selected MPF algorithms

Among these algorithms we chose a basic idea of one of the solving algorithms for the adversarial version of CPF. Due to its efficiency and simplicity we selected cooperative A*.

2. Adversaries

As we have seen in section 1, basic path-finding, where the task is to search a path from given source vertex to a certain target vertex, can be generalized by multi-agent path-finding and further by cooperative path-finding. Nevertheless we can generalize further. This time we will divide agents into groups, where members of one group cooperate and try to defeat adversarial groups. Let us use term *teams* for these cooperative groups.

2.1 Basic Attributes and Characteristics

The aim of adversarial cooperative path-finding is to find suitable paths for selected team of agents in a given undirected unweighted graph. Agents are placed at vertices and can move along edges. Unlike CPF, agents are divided into competing teams. Agents that belong to other teams than the selected one are considered as adversaries. Another distinction from CPF is that an agent has not only single target vertex, but generally can have a target set of vertices. If the target set of vertices of an agent is empty, such agent's terminal location can be arbitrary vertex. The goal of one team is to lead all its agents to their targets before opposing teams manage to relocate their agents to the corresponding opponent's targets. Competing teams can potentially harm each other by blocking opponent's targets or paths towards them.

Proposed algorithm searches a solution for a given team, i.e. finds a sequence of moves, that leads all members of the team to their targets. No matter how the opponents act, we want the selected team to accomplish the target positions. In order to succeed this task, one should try to predict the behavior of adversarial teams. Knowledge about opponents' locations is also essential for preserving the legality of a move.

Performance of the agent movement is divided into discrete time steps exactly as it is in CPF. The agents' positions are known for every time step - in other words, there exists a mapping that assigns current vertex to each agent. All the teams have information about target vertices of every agent, even targets of adversarial agents.

The connection of ACPF with n-player games suggests its comparison with popular board games. Unlike classical board games, ACPF do not define any specific opening formation, size or shape of the environment, number of agents, location of target positions. Such freedom of assignment leads to a huge number of different instances of the problem with very diverse characteristic properties. Another difference between considered problem and typical board games consist in moving the agents (term piece is more common in board games). Agents of a team in ACPF move simultaneously within a time-step, while in board games, player usually selects one particular piece and perform a single move with it. This fact makes the state space extremely large, as we shall see in the theoretical part of this work. Game principles are very diverse, some are more resembling our problem than others. ACPF is closer related to abstract strategy games, where pieces can move and interact among each other. Well known examples are *checkers*, *agon*, *chess* and many variants of chess like *shoggi* or *xiangqi*. Other strategy

games where pieces remains on a fixed positions after being placed are rather distantly related to ACPF. *Go*, *hex* or *tic-tac-toe* are some of many examples of such games.

2.1.1 Technical Difficulties

Agents belonging to a particular team has all aspects of cooperative agents described in the section 1.1. Within one move, an agent can vacate its current vertex for its teammate and prevent the opponent to take over the vertex. Nonetheless a team has no information about further moves of any opposing team. Even though our team can plan the approach in advance, it is not always guaranteed that the plan will be feasible. The opponent can make an unexpected move and the planned moves may be suddenly impossible. Solving algorithm must be able to handle such situation and create a new plan.

Since the teams are competing, a situation when two agents from different teams want to access one vertex may occur. It is necessary to decide who will enter and who should not be allowed. Due to this issue we determine an ordering of teams to move, as is commonly known from board or card games. When the simulation is visualized it seems that teams moves parallelly, but in fact they take turns.

Another consequence of unknown opponent's moves is that in most cases achievement of a target vertex is not necessarily possible. Opposing agent might block the way towards some other agent's target vertex. This blocked agent would never reach its target. To overcome this inconvenience we limit the progress to a certain number of time steps. Whenever some team reaches a target positions within the time limit, calculation is over. If no team succeeds in it, we evaluate which team came closer to its goal. This team is then considered as winner.

2.2 Practical Usage

The most obvious application is in game industry. Real-time strategies are one example, where ACPF can be useful. For example raw material extraction units while meeting opponent units, patrolling agents, transporters and surely many other tasks in so diverse world of computer games can be modeled using ACPF.

There are also examples in the real world, where ACPF can be useful. Police actions or military operations can be simulated by ACPF as well. One could object, that behavior of participating groups is far from following rules defined later in this chapter. That is true indeed, nevertheless our simulation could identify weaknesses and bottlenecks in a plan, so the plan can be enhanced soon enough.

Investigated topic may serve as a starting point for studying tactical military manoeuvres. Manoeuvres such as blockade, encirclement or flanking could be studied and modeled using ACPF.

2.3 Related problems with adversarial aspects

Although competitive elements can be found in many different fields of artificial intelligence research, our studied problem is quite specific and mentioned works are rather distantly related.

Capture the flag [15] is a resembling problem: two-sided game played by teams. Each team owns a territory with a flag located there. The flag can be captured by an opposing agent. An agent in opponent's territory can be intercepted by some opposing agent. The objective of the game is to capture opponent's flag and return with it to safety while protecting one's own flag from the enemies with symmetric goal.

Similar work in adversarial search predominantly addresses applications in video games, especially in real-time strategies. This research usually considers more complex agents with wider range of actions, whereas we focus merely on movement.

Pacman is nowadays already rather a classical video game. Its features resemble ACPF: A single agent is supposed to gather up tokens in a maze. Adversaries try to prevent the agent from succeeding this task. Whenever an adversary clashes with the agent, the game is over. [26] investigates application of Monte Carlo tree search methods for Pacman.

Simulation based adversarial planning in real-time strategies is studied in [25]. Heuristics for large multi-agent simulations are investigated in [21]. [7] presents Monte Carlo methods in strategy games, particularly capture the flag.

Another very large category of related topics is comprised of various n-player games. For example games where pieces can move and interact among each other (*chess*, *checkers*) are closer related to our problem than those games considering merely placing pieces on chosen locations (*go*, *hex*, *gomoku*).

2.4 Formal Definition

We are not aware of any other work dealing with ACPF since its first introduction in [16]. This section provides essential formalism and terminology of the studied problem. Definitions are based on usual CPF terminology. Adversarial element present in ACPF requires extension and adaptation of these existing definitions.

2.4.1 ACPF Problem

Definition. *Adversarial cooperative path-finding problem* (ACPF) is a 7-tuple

$$\Sigma = (G, A, \mathcal{T}, t^*, \lambda_0, \lambda_+, \hat{\alpha})$$

where

$G = (V, E)$ is an undirected graph

$V = \{v_1, v_2, \ldots, v_n\}$ denotes a finite set of vertices

$E \subseteq \binom{V}{2}$ denotes a set of edges

$A = \{a_1, a_2, \ldots, a_k\}$ is a a finite set of agents

$\mathcal{T} = \{T_1, T_2, \ldots, T_t\}$ is a finite set of teams, $t \leq n$. Teams are disjunct sets of agents and every agent belongs to exactly 1 team, formally:

$$\bigcup_{i=1}^{t} T_i = A \qquad \forall T_i, T_j \in \mathcal{T}, i \neq j : T_i \cap T_j = \emptyset$$

$t^* \in \{1, 2, \ldots, t\}$ denotes an index of a team, for which the algorithm searches a solution

$\lambda_0 : A \to V$ is an injective mapping that assigns an initial vertex to each agent

$\lambda_+ : A \to \mathcal{P}(V)$ assigns a target set of vertices to each agent

$\hat{\alpha} : \mathcal{L} \to \mathcal{A}^1$ is a partial functional, that from given previous sequence of all the agents' placement determines a subsequent placement of the adversary agents. The domain and range of this functional can be defined as follows:

$\mathcal{L}^l = \{(\lambda_{m_1}, \ldots, \lambda_{m_l}) | \lambda_{m_i} : A \to V,\ 1 \leq i \leq l\}$ a set of placement sequences of length l

$\mathcal{L} = \bigcup_{i \in \mathbb{N}} \mathcal{L}^i$ a set of placement sequences of all possible lengths

$\mathcal{A}^1 = \{\alpha | \alpha : A \setminus T_{t^*} \to V\}$ a set of all adversary agents' placements.

The functional itself is unknown, but it is supposed to be an intelligent black box. It is a partial functional, because not every placement sequence can occur. Section 2.4.2 describes rules for legal movement of the agents. Note that $\hat{\alpha}$ determines the placement only for adversarial agents. The agents belonging to the team T_{t^*} are handled by developed algorithm.
Note: Sometimes we demand additional limitation for mapping λ_+ :

$$\forall T \in \mathcal{T} \forall a_1, a_2 \in T, a_1 \neq a_2 : \lambda_+(a_1) \cap \lambda_+(a_2) = \emptyset$$

if an agent's set of target vertices should be distinct from all targets of other agents belonging to the same team. Or even stricter condition can be applied:

$$\forall a_1, a_2 \in A, a_1 \neq a_2 : \lambda_+(a_1) \cap \lambda_+(a_2) = \emptyset$$

when any two agents must have distinct set of vertices, regardless their teams.

2.4.2 Agent Movement

Agents move along edges or stay at a vertex. The moves are taken as discrete transformations of agents' positions. For this purpose, we divide time into a discrete time steps. Arbitrary number of agents can move at every time step and every vertex can be occupied by at most one agent. In order to describe movement formally, we use a mapping $\lambda_i : A \to V$ for time step i and the generalized inverse $\lambda_i^{-1} : V2 \to A \cup \{\perp\}$ that satisfies:

1. $\forall a \in A : \lambda_i(a) = \lambda_{i+1}(a) \vee \{\lambda_i(a), \lambda_{i+1}(a)\} \in E$
   (Agent moves along an edge or stay at a vertex.)

2. $\forall a, a' \in A\ \forall u, v \in V : (\lambda_i(a) \neq \lambda_{i+1}(a) \Rightarrow \lambda_i^{-1}(\lambda_{i+1}(a)) = \perp) \vee$
   $(\lambda_i(a) = u\ \&\ \lambda_{i+1}(a) = v\ \&\ \lambda_i^{-1}(v) = a' \neq a \Rightarrow \lambda_{i+1}(a') \notin \{u, v\})$
   (Agent can move to unoccupied vertex v or if v was occupied at time i by agent a' different from a, agent a' must move away. Position swapping is not allowed.)

3. $\forall a, b \in A : a \neq b \Rightarrow \lambda_i(a) \neq \lambda_i(b)$
   (λ_i is an injective mapping. Every vertex can be occupied by at most 1 agent.)

4. $\forall a \in A \setminus T_{i+1 \pmod t} : \lambda_i(a) = \lambda_{i+1}(a)$
   (Rotation of a team to move. The only agents that can move between time i and $i+1$ are those from the team $T_{i+1 \pmod t}$. All the others must stay at their earlier vertices.)

2.4.3 Solution

Since our goal is to find a solution for specific team and the domain of λ_i is whole set A of agents, we introduce $\lambda_i^* : T_{t^*} \to V$ as a restriction of λ_i to T_{t^*} for every time step i. We can also write $\lambda_i^*(a) = \lambda_i|_{T_{t^*}}(a)$ for some $a \in T_{t^*}$. The expression on the right side of this equation is just a standard notation for mapping restriction, but for better readability and simplicity we'll prefer the notation on the left side.

Intuitively, a solution is a movement of agents of a specific team that leads them to some of their target positions. Before we define the solution formally, let us consider some situations. The problem definition doesn't guarantee existence of the solution. For instance the graph is not required to be connected and the agents are not necessarily in the same component as their targets. Even some CPF instances does not have any solution. Another example is when an adversarial agent blocks the path to a desired vertex, therefore such vertex can't be reached by other agents.

Aware of these properties we propose two auxiliary definitions before the solution definition itself.

Definition. A *progress* of length m of an ACPF problem Σ is a sequence of mapping

$$\vec{s} = \left[\lambda_0^*, \lambda_1^*, \ldots, \lambda_{m-1}^*\right]$$

The mapping λ_i^* determines position at time step i for every agent $a \in T_{t^*}$. Other agents (those from $A \setminus T_t^*$) are controlled by adversarial functional $\hat{a}$ This mapping must satisfy:

$$\forall a \in T_{t^*} : \lambda_0^*(a) = \lambda_0(a)$$

in other words, the starting position of agents is known from the problem definition. Note that the constraints from section 2.4.2 are also satisfied by these mappings, because it is a restriction to domain T_{t^*}. The length of a progress is regarded as a number of time steps. After the last step, all agents can occupy arbitrary vertex they reached, no matter if it's agent's target or not.

Definition. A *partial solution* of length m of an ACPF problem Σ is a progress of length m, where following holds:

$$\exists a \in T_{t^*} : \lambda^*_{m-1}(a) \in \lambda_+(a)$$

In partial solution, at least one agent of considered team must occupy its target vertex.

Definition. A *solution* of length m of an ACPF problem Σ is a partial solution of length m, where

$$\forall a \in T_{t^*} : \lambda^*_{m-1}(a) \in \lambda_+(a)$$

After carrying out all steps of the solution, there must be no agent located at a vertex different from its targets.

2.4.4 Quality of a Progress

The nature of defined problem admits situations, where solution achievement is impossible, sometimes we cannot even accomplish partial solution. Hence we need to estimate quality of a progress. Two different progresses may lead to the same state. There exists several ways how to evaluate a progress and one of the simplest method counts number of agents that reached their target positions.

Definition. Let $\mathcal{S}$ denote a set of all possible progresses of an ACPF problem Σ. *Goal evaluation* is a function $\gamma_0 : \mathcal{S} \to \mathbb{N}$ determining a number of agents occupying their goal positions after the last step of the progress. Formally

$$\forall \vec{s} \in \mathcal{S}, |\vec{s}| = m : \gamma_0(\vec{s}) = |\{a \in T_{t^*} : \lambda^*_{m-1}(a) \in \lambda_+(a)\}|$$

Such definition estimates the quality of a progress, but doesn't take into account any adversaries, even though they can be added straightforwardly: Function $\gamma_1 : \mathcal{S} \to \mathbb{N}$ determines difference between number of agents from team T_{t^*} occupying their targets and adversary agents occupying their targets:

$$\forall \vec{s} \in \mathcal{S}, |\vec{s}| = m : \gamma_1(\vec{s}) = \gamma_0(\vec{s}) - |\{e \in A \setminus T_{t^*} : \lambda_{m-1}(e) \in \lambda_+(e)\}|$$

Clearly, there can occur many progresses with equal γ_1 value and it would be useful, if we were able to fine the ordering and distinguish the quality of a progress in a better way. Hence we will also consider the distance of agents to their targets, such that the closer is the agent to its target, the better.

Definition. Let $P(s,t)$ be the shortest path from source vertex s to target vertex t and $dst(P(s,t))$ be a length of such path. For a progress $\vec{s}$ of the length i we define *distance evaluation* $\delta_0(\vec{s})$ as a sum of all lengths of the shortest paths from agents' current positions to their closest targets. Formally

$$\delta_D(\vec{s}) = \sum_{a \in T_{t^*}} \min_{t \in \lambda_+(a)} \{dst(P(\lambda_i(a), t))\} - \sum_{a \in A \setminus T_{t^*}} \min_{t \in \lambda_+(a)} \{dst(P(\lambda_i(a), t))\}$$

If we require more realistic estimation of the length from source to target, we can define $P_E(s,t)$ as a shortest path between s and t consisting merely unoccupied vertices. In this case we have *empty distance evaluation* $\delta_E(\vec{s})$

$$\delta_E(\vec{s}) = \sum_{a \in T_{t^*}} \min_{t \in \lambda_+(a)} \{dst(P_E(\lambda_i(a), t))\} - \sum_{a \in A \setminus T_{t^*}} \min_{t \in \lambda_+(a)} \{dst(P_E(\lambda_i(a), t))\}$$

Remark. If the shortest path between two vertices does not exist, we use length of the longest possible path in considered graph.

Observation. $\forall \vec{s} \in \mathcal{S} : \delta_D(\vec{s}) \leq \delta_E(\vec{s})$

Having the definitions above, an ordering on progresses, partial solutions and solutions can be introduced. A total ordering on $\mathcal{S}$ can then then defined as

$$\begin{aligned} \forall \vec{s_1}, \vec{s_2} \in \mathcal{S} : \vec{s_1} \leq \vec{s_2} \Leftrightarrow \\ \gamma_1(\vec{s_2}) \leq \gamma_1(\vec{s_1}) \\ \vee \gamma_1(\vec{s_1}) = \gamma_1(\vec{s_2}) \ \& \ \delta_x(\vec{s_1}) \leq \delta_x(\vec{s_2}) \end{aligned}$$

Which means that first we take interest into fulfilled targets, and if we can't decide by this property we consider how close are remaining agents to their targets. Instead index x in δ_x we can choose D or E depending on whether we want to consider merely distances in graph or whether we also want to consider other agents.

There are more possibilities how to define ordering on $\mathcal{S}$, but these are sufficient for our purposes.

2.5 Problem Complexity

Firstly let us specify what kind of problem are we trying to solve. With respect to complexity we are interested in the decision variant of the problem given a graph and placement of all agents (instance of ACPF problem), we want to decide whether there exists a winning strategy for a selected team.

Before we continue, lets remind needful definitions commonly present in literature [31]:

Definition. *PSPACE* is the class of languages that are decidable in polynomial space on a deterministic Turing machine. In other words,

$$PSPACE = \bigcup_k SPACE(n^k)$$

Definition. A language B is *PSPACE-complete* if it satisfies two conditions:

1. B is in PSPACE

2. every $A \in PSPACE$ is polynomial time reducible to B.

If B merely satisfies condition 2, we say that it is *PSPACE-hard*.

For determining a complexity class of ACPF problem lets look at some similar problems. Since the problem is basically an n-player game, our first attempt will be to show its membership in PSPACE-complete complexity class. According to definition of PSPACE-completeness, to prove this, we must show that the problem is PSPACE and PSPACE-hard.

To show membership in PSPACE is easier on the first sight, but unfortunately our attempts failed so far. By contrast, we already know that the question whether an instance of ACPF problem has a winning strategy for a selected team is PSPACE-hard. [16] offers a sketch of proof, here we will give an alternative and technically easier proof with detailed explanation.

Proof of PSPACE-hardness of ACPF is based on several definitions presented for example in [31] and [38]:

Definition. Boolean formula ϕ is in *conjunctive normal form*, (CNF), if and only if $\phi = D_1 \wedge \cdots \wedge D_m$, where each D_i is a disjunction of literals. Individual D_is are called *clauses*.

In case that every clause of formula ϕ has exactly three literals we say that ϕ is in 3CNF.

Definition. Boolean formula is said to be *fully quantified* when each variable of the formula appears within the scope of some quantifier. Sometimes we use term *sentence*.

Definition. *Game formula* is a fully quantified boolean formula ϕ written in form $Q_1 x_1 Q_2 x_2 \ldots Q_n x_n \psi(x_1, x_2, \ldots, x_n)$, where each Q_{2i} is $\forall$ and each Q_{2i+1} is $\exists$. ψ is a quantifier free boolean formula.

Definition. Further we introduce a language GF:

$$GF = \{\phi : \phi \text{ is a valid game formula}\}$$

Language GF is known to be PSPACE-complete.

Following auxiliary lemmas are useful in the main proof.

Lemma 1. *(**Vertex booking**). Let $\Sigma = (G = (V, E), A, \mathcal{T}, t^*, \lambda_0, \lambda_+, \hat{\alpha})$ be an ACPF instance and $v \in V$ be so called booked vertex. Next, let $b \in \mathbb{N}$ be so called booking time and T_{t_b} be some selected team (not necessarily T_{t^*}) for what the vertex should be booked. Then there exists a modified instance $\Sigma' = \left(G' = (V', E'), A', \mathcal{T}, t^*, \lambda_0', \lambda_+', \hat{\alpha}'\right)$ such that the booked vertex v is not accessible by any other agent $a \in A$ before time-step b.*

Proof. Σ' is derived from Σ by adding an extra vertex v' that is connected by an edge to the vertex v.

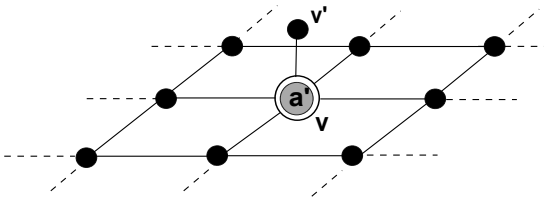


Figure 2.1: Vertex booking

Alongside we put a new agent $a' \in T_{t_b}$ on the vertex v. Controlling functional $\hat{\alpha}'$ shall lead agent a to the vertex v' at time-step b. Hence any teammate located at a vertex adjacent to v can enter v at time step b. If it does not happen, opponent could have the opportunity to occupy v at time step $b+1$.

Formally we have $V' = V \cup \{v'\}$, $E' = E \cup \{(v, v')\}$, $A' = A \cup \{a'\}$, $\lambda_0(a') = v$, $\lambda_+(a') = \emptyset$ □

Lemma 1 addresses the ability of agents to reserve place for their teammates. If a vertex is occupied by an agent from some team, this agent can wait till its teammate reaches an adjacent vertex and hold the ground, which will prevent opponent's agents to enter the vertex.

Next we will formulate a similar lemma that will be useful for forcing agents to enter a particular vertex in desired time step.

Lemma 2. *(**Hazardous vertex**). Let $\Sigma = (G = (V, E), A, \mathcal{T}, t^*, \lambda_0, \lambda_+, \hat{\alpha})$ be an ACPF instance and T_{t_h} be some selected team (not necessarily T_{t^*}) and let $v \in V$ be so called hazardous vertex. Next, let $s \in \mathbb{N}$. Then there exists a modified instance $\Sigma' = \left(G' = (V', E'), A', \mathcal{T}, t^*, \lambda'_0, \lambda'_+, \hat{\alpha}'\right)$ such that the hazardous vertex v can be occupied by an agent $a' \in A \setminus T_{t_h}$ at time step s, if this vertex is not entered by any agent $a \in T_{t_h}$.*

Proof. Σ' is derived from Σ by attaching a path $v'_0 \dots v'_{s-1}$to the vertex v.

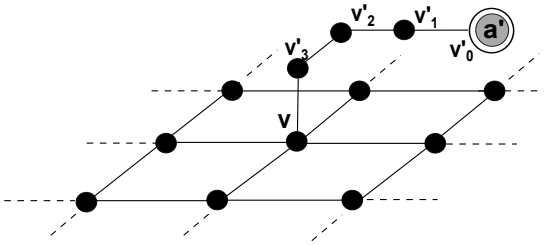


Figure 2.2: Hazardous vertex, $s = 4$

Vertex v'_0 contains an agent $a' \in A \setminus T_{t_h}$. This agent can approach the vertex v every time step and enter it in the s-th step, unless any other agent do it earlier.

Formally we have $V' = V \cup \{v'_0 \dots v'_{s-1}\}$, $E' = E \cup \{(v'_0, v'_1), (v'_1, v'_2) \dots (v'_{s-1}, v)\}$, $A' = A \cup \{a'\}$, $\lambda_0(a') = v'_0$, $\lambda_+(a') = \emptyset$ □

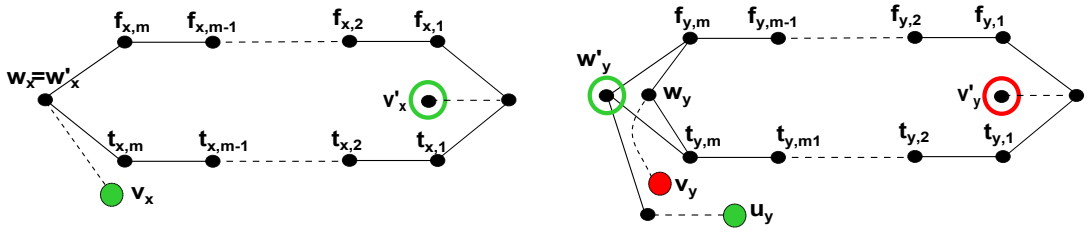
Lemma 2 points out a situation, where some vertex v is approached by an agent and if other agent from different team want to enter it, it should do it before time step s, because at that time it will not be possible. We say that vertex v is *hazardous* from step s.

Lemma 3. *(**Winning strategy**). Let T_s denote a selected team. For each game formula ϕ there exists an ACPF instance such that if there exists a winning strategy for team T_s, then the only winning approach is the fastest possible advancement of agents from T_s towards their targets.*

Remark: By the fastest possible advancement we mean that every agent rushes towards its target using a path with minimal length. An agent is also not allowed to wait at a vertex other than its target. The construction of an ACPF instance is based on the proof of NP-hardness of the optimization variant of multi-robot path-planning [42].

Proof. First we construct an ACPF instance for given formula. Then we show that the fastest possible advancement of team T_s is the only way how to succeed.

Let $\phi : Q_1 x_1 \dots Q_n x_n \psi$ be a game formula. Without loss of generality, we can assume that ψ, the quantifier free part of ϕ, is in 3CNF[1]. Next, let m stands for number of clauses and n denote number of variables in ϕ. Further we assume that every variable is present in both literals. If some variable or its negation does not appear in any clause, we can add an extra clause containing the missing variable and its negation, so the clause would be always satisfied and the truth value of the formula would not change.



(a) Existentially quantified variable gadget

(b) Universally quantified variable gadget

Figure 2.3: Variable gadgets

For every variable x that occurs in ϕ we construct a variable gadget - two vertices w_x and v'_x connected by two paths of length m (number of vertices between w_x and v'_x, see figure 2.3). Vertices of the horizontal gadget paths are numbered from the right-hand side, $t_{x,i}$ for lower path and $f_{x,i}$ for upper path, $i = 1 \dots m$.

There will be an extra path connected to w_x ended by vertex v_x. The length of this path corresponds to the variable order in the quantifier part of ϕ. Paths connected to gadgets of variables x_1 and x_2 are 1 vertex long and are thus identical to the vertex w_x. Paths of every next pair of variable gadgets are one vertex longer then the paths of previous pair. Hence the length of the longest path is $\lceil n/2 \rceil$.

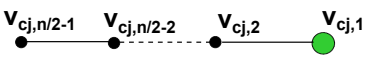


Figure 2.4: Clause gadget

[1]Each formula is logically equivalent to a formula in CNF and every formula in CNF can be transormed into 3CNF [38].

The rightmost vertex, where the upper and lower paths meet, is connected to a path which is ended by vertex v'_x. Length of this path is determined such that all agents have the same distances to their targets.

Vertex v_x of an existentially quantified variable is an initial vertex for agent a_x belonging to the selected team T_s (associated with green color). Vertex v'_x represents a target vertex of the agent a_x.

Universally quantified variable y has an additional vertex w'_y connected to the upper and lower horizontal path. At the vertex v_y, there sits an agent a_y belonging to the adversarial team (associated with red color). Vertex v'_y is the target vertex of the agent a_y, similarly as for the existentially quantified variable gadget. Another path ended by vertex u_y is connected to the vertex w'_y. The length of this path is exactly one vertex longer than the path $w_y \dots v_y$. A green agent $a_{y,u}$ is placed at the vertex u_y. Vertex w'_y is a target of the agent $a_{y,u}$.

Agent a_x entering either upper or lower path of the x variable gadget simulates evaluation of x. Order in which an agent $_x$ is able to enter one of these two paths is exactly the order in which the variable x is evaluated.

Further we construct clause gadgets (fig. 2.4). For the j-th clause there is a path $v_{c_j,1} \dots v_{c_j,\lfloor n/2 \rfloor - 1}$. Vertex $v_{c_j,1}$ contains a green agent a_j, $j = 1 \dots m$.

Let x be the i-th quantified variable and $k = \lfloor i/2 \rfloor + 1$. A clause vertex $v_{c_j,k}$ is connected to vertex $f_{x,j}$ if x appears as a positive literal or to $t_{x,j}$ if x appears as a negative literal in clause c_j. Example of full construction is depicted in the figure 2.5, where the connection between variable and clause gadgets is showed. There are always three edges connecting a clause gadget and the rest of the graph, as there are 3 literals in every clause.

Finally we add target vertices for agents starting from clause gadgets. Let us denote them v'_{c_j}, $j = 1 \dots m$. These vertices are connected to the variable gadgets representing variables that appeared in a clause. For example if a variable x is present in clause c_j, we add a path between v'_{c_j} and w'_x. Number of vertices between v'_{c_j} and w'_x is exactly

$$\left\lfloor \frac{n}{2} \right\rfloor + j - \left\lfloor \frac{i}{2} \right\rfloor - 1$$

where i stands for order of variable x in the quantifier part of a formula. Our intention is that the length of the shortest path between source and target vertex is equal for all clause agents. If agent a_j starting from $v_{c_j,1}$ selects x variable gadget for its advancement towards the target, the total length of the route would then be

$$\underbrace{\left\lfloor \frac{i}{2} \right\rfloor + 1}_{\text{clause gadget}} + \underbrace{m - j + 2}_{\text{variable gadget}} + \underbrace{\left\lfloor \frac{n}{2} \right\rfloor + j - \left\lfloor \frac{i}{2} \right\rfloor}_{\text{rest}} = m + 3 + \left\lfloor \frac{n}{2} \right\rfloor$$

which is a constant.

Target vertices of green agents a_x and a_j and corresponding agents in other gadgets are hazardous from time

$$2(m + 2 + \left\lfloor \frac{n}{2} \right\rfloor)$$

That is right after the agents could arrive there, in case they use the shortest possible paths. (18 in the example in 2.5)

For a better insight into the construction, let us consider following example. For formula

$$\begin{aligned}\phi: \quad & \exists x \forall a \exists y \forall b \exists z \forall c \\ & (b \vee c \vee x) \wedge (\neg a \vee \neg b \vee y) \wedge (a \vee \neg x \vee z) \wedge (\neg c \vee \neg y \vee \neg z)\end{aligned}$$

we get an ACPF instance depicted in the figure 2.5. Red numbers near some vertices indicates that the vertex is hazardous from a particular time, i. e. such vertex can be entered by red agent in a specified time step. Black numbers near some arcs signifies that there is some number of vertices on a particular arc. For clearer arrangement these vertices are not displayed implicitly.

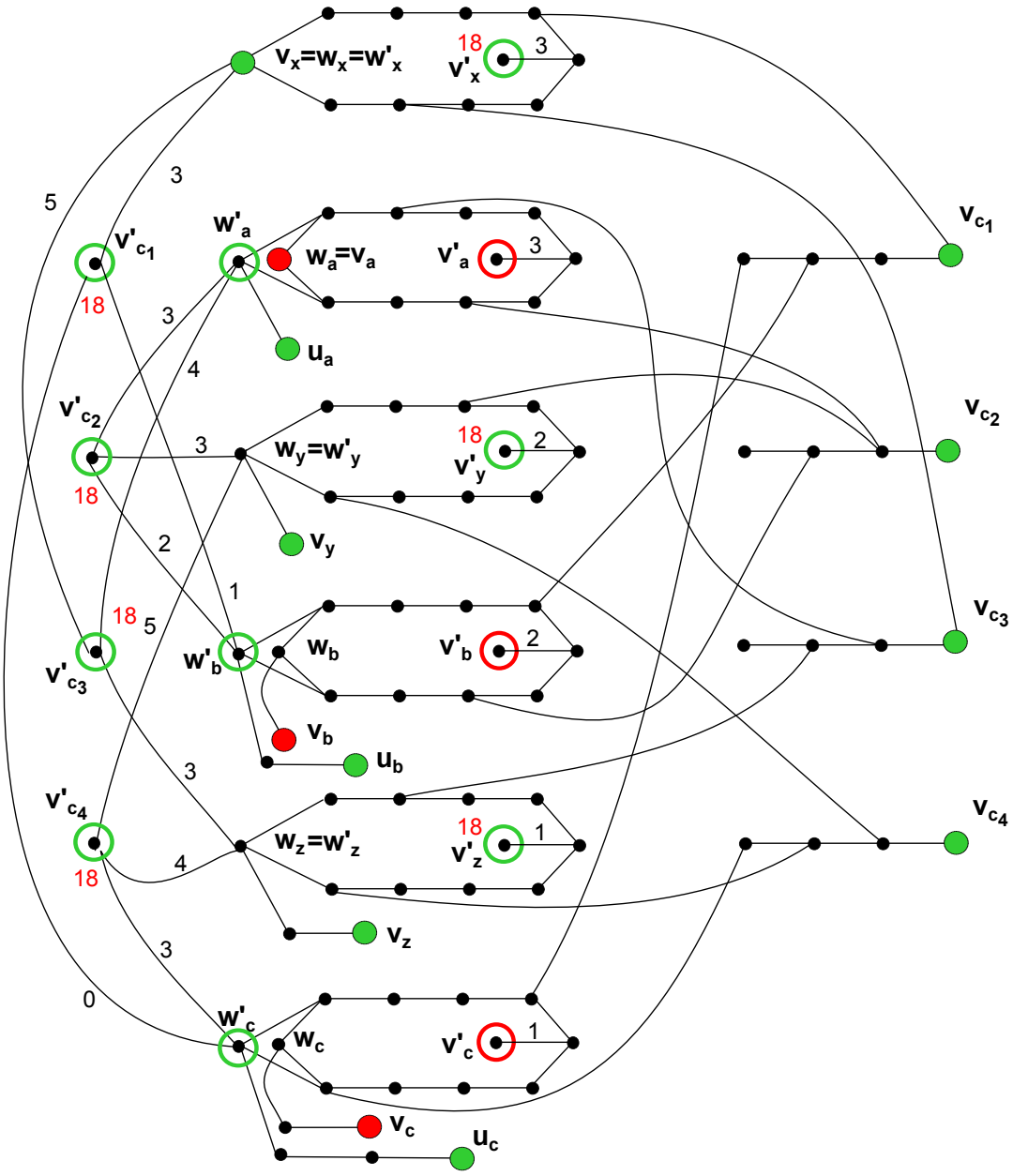


Figure 2.5: Example of reduction

The construction of an ACPF instance for a given formula is now finished. The

claim that if there is a winning strategy for selected team, then it is the fastest possible advancement remains to be proved. We will show that any deviation of an agent from the fastest route leads to failure of the agent's achievement.

If green agents tried to use another path then the shortest one, they would give up their opportunity to win. The same would happen if an agent stayed at a vertex. That is because target vertices of agents starting from existentially quantified variable gadgets and clause gadgets are hazardous exactly one time step after the expected arrival time. If agents held up somewhere, its target would become occupied by an adversarial agent.

□

Now we have necessary apparatus for proving following statement.

Proposition 1. *Decision problem whether there exists a winning strategy for selected team in a given ACPF instance is PSPACE-hard.*

Proof. We will show a polynomial reduction from GF to ACPF.

Formally correct proof requires proving of following equivalence: Boolean formula ϕ is valid if and only if there exists a winning strategy for the selected team in the provided ACPF instance.

Let us start with implication from left to right. Suppose ϕ to be valid. Variables gradually acquire their truth value one by one according to the order in quantifier part of ϕ. In our example all possible sequences of evaluation showing validity (validity certificate) of ϕ are indicated in the evaluation tree depicted in the figure 2.6. Each path from root to leaf in the tree represents one sequence of evaluations. According to each such sequence we can lead the agents so that the selected (green) team wins.

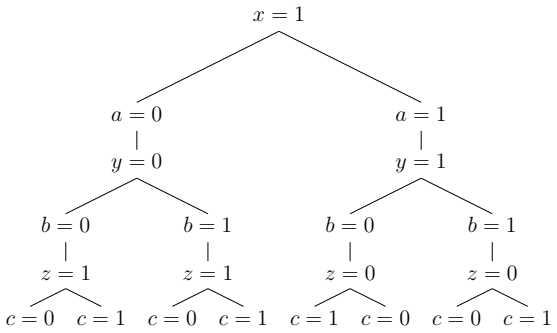


Figure 2.6: Validity certificate (evaluation tree)

Suppose we start in the root. Every time we visit a vertex associated with a variable during descending towards some leaf, another agent in the constructed ACPF instance is ready to enter upper or lower path within the variable gadget. If some variable x is evaluated as true, corresponding agent a_x enters the lower path (vertex $t_{x,m}$), otherwise the agent goes to the upper path (vertex $f_{x,m}$). Since the evaluation satisfies the formula, every clause c_j has at least one variable x that causes satisfaction of c_j. In such case, the clause gadget is connected to the variable gadget through a vertex in the other path than the one through which a_x advances towards its target. Clause agent a_j can then enter the variable gadget without encountering a_x. Both agents can continue undisturbed. Descending

path from root to leaf in the validity certificate can be regarded as a pattern for the advancement of agents in the constructed ACPF instance.

A purpose of green agents $a_{y,u}$ (u-agents) is to prevent red agents from unwanted behavior. Now lets inquire into what happens if the red team behaves unexpectedly:

1. *Waiting:* If a_a tried to be binding at a vertex w_a or before it reaches w_a, u-agent would be able to help a clause agent to reach its target, even if the formula were not valid: u-agent would control vertices w'_a and $f_{a,m}$ or $t_{a,m}$ and red agent would not have any opportunity to impact next progress of any other agent. u-agent acts as the agent a' from the booking lemma 1.

2. *Transfer to another vertex gadget:* Red agent could also try to get into another variable gadget through some clause gadget. Suppose that adversarial agent a_{x_i} wants to get to vertex gadget corresponding to variable x_k through clause gadget of c_j. Number i is even and number k is odd, because adversarial agents are associated with even variables, $i \leq k$. Further assume that clause gadget is connected to vertex $f_{x_k,j}$.

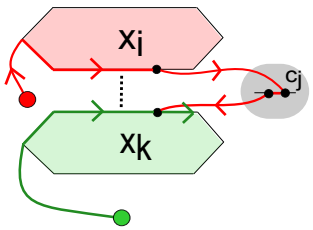


Figure 2.7: Transfer to another variable gadget

Our objective is to calculate that adversarial agent a_{x_i} would arrive at $f_{x_k,j}$ always later than green agent a_{x_k}. Let $dst(a, v)$ denote the shortest distance from an initial vertex of agent a to vertex v. Now we can compare the two distances:

$$dst(a_{x_k}, f_{x_k,j}) \leq dst(a_{x_i}, f_{x_k,j})$$

$$\underbrace{\frac{k}{2}+1}_{\substack{\text{path}\\ \text{to } w_{x_k}}} + \underbrace{m-j+1}_{\text{variable gadget}} \leq \underbrace{\frac{i}{2}}_{\substack{\text{path}\\ \text{to } w_{x_i}}} + \underbrace{m-j+1}_{\text{variable gadget}} + \underbrace{\frac{k-i+1}{2}}_{\text{clause gadget}} + \underbrace{1}_{f_{x_k,j}}$$

$$\frac{k}{2} \leq \frac{k}{2} + \frac{1}{2}$$

which always holds. This inequality ensures that whenever red agent decides to enter another variable gadget, green agents are not affected by this unexpected behavior.

At this point we have showed that any pathological movement of the red team would not affect the winning strategy.

Now we will show the direction from right to left. Whenever there exists a winning strategy for the constructed ACPF instance, variable and clause agents must reach their targets on time. This is possible only in case variable agents and clause agents do not meet on the horizontal path. They must use different paths. The variable agents' selection of paths determines the evaluation of corresponding variables. Whenever an agent and clause pass by each other on the parallel horizontal paths, corresponding variable causes satisfying of the clause.

□

The next attempt will be to show that ACPF belongs to the EXPTIME class. According to [31], we define

$$EXPTIME = \bigcup_{k \in \mathbb{N}} TIME(2^{n^k})$$

For this proof we will provide an algorithm that decides whether there exists a winning strategy. Game tree can be regarded as an AND-OR tree. Although theoretically there can be more teams involved and AND-OR trees are suitable for two player games, simple assertion will prevent complications. For the purposes of the proof we will assume that all adversarial teams are considered as one adversary and the move alternation is going on between the selected team and the adversary containing all other teams.

Proposition 2. *Question whether in a given instance of ACPF exists a winning strategy for a selected player belongs to EXPTIME complexity class.*

Proof. Let n be a number of nodes and k be a number of agents. Size of the state space is then $2\binom{n}{k}k!$ and since there can be up to n agents, we get $2n!$ state space. Because we distinguish which team is to move, we have to multiply by 2. Since

$$\mathcal{O}(n!) \subseteq O(n^n) = \mathcal{O}((n)^n) \subseteq \mathcal{O}((2^n)^n) \subseteq \mathcal{O}(2^{n^2})$$

we have $DTIME(n!) \in EXPTIME$, where $DTIME(n!)$ are languages recognizable by deterministic algorithms running in $\mathcal{O}(n!)$.

Algorithm 2 requires an initial state *root* and a determination of the selected team *me*. It returns *true* when there exists a winning strategy for team *me*, $false$ otherwise. We need to show that it runs in exponential time.

Possible problem could arise with infinite loops: some state is repeatedly visited. To prevent such behavior we create a *visited* list that contains all states that have been already visited. As a consequence of the list usage, we visit every state at most once . Finally, member operation of the list takes time $|visited| \times \mathcal{O}(n)$. Size of the *visited* list is not greater than $\mathcal{O}(n!)$. Hence the time complexity of the deciding algorithm is exponential.

□

Algorithm 2 Decides whether an instance of ACPF problem has a winning strategy for the selected team

```
function ACPF_decider (root, me, first_to_move)
  create global list visited ← {root}
  if first_to_move == me then
    return ACPF_me(root)
  else
    return ACPF_adv(root)
  end if
end function

function ACPF_me(node)
  visited ← visited ∪ {node}
  if is_terminal(node) then
    return false
  end if
  for all s ∈ Successors(node) do
    if s ∉ visited then
      if ACPF_adv(s) then
        return true
      end if
    end if
  end for
  return false
end function

function ACPF_adv(node)
  visited ← visited ∪ {node}
  if is_terminal(node) then
    return true
  end if
  for all s ∈ Successors(node) do
    if s ∉ visited then
      if not ACPF_me(s) then
        return false
      end if
    end if
  end for
  return true
end function
```

3. Solving Adversarial Version

An instance of the ACPF problem can be treated in different ways. Since the problem is a generalization of cooperative path-finding, one option is simply to adapt methods for CPF and use them while solving ACPF. Probably the most promising possibility was mentioned in the introduction. Since the problem has all aspects of n-player game with perfect information, we can try to find solution using existing techniques for such games. Another approach is to extract some information from a particular problem. It is also possible to combine different methods.

This chapter describes suggested methods and discuss their suitability for the solved problem. The biggest emphasis is put on methods that has been tested in the experimental part of the thesis. We focus on 4 basic approaches:

- Greedy
- Cooperative
- Minimax
- Monte Carlo

Greedy and cooperative approaches are introduced in this thesis for the first time. Remaining two are well-known methods that were adapted for our purposes and applied on solving ACPF instances.

3.1 Greedy Methods

Although greedy algorithms are known for their low computational cost and simplicity, they are not expected to represent too powerful solving approaches. Nevertheless we will describe them here as greedy methods are a good starting point. Moreover their combination with other methods might be beneficial. Greedy move selection in ACPF is decided every time a team is to move. We can distinguish two types of greedy methods:

Centralized: From a given position all possible moves of a whole team have to be generated. Moves are then evaluated one by one. The move that leads to a placement with the best quality of state is selected.

Decoupled: Agents are processed one by one and next moves are always generated only for the currently considered agent. A move leading an agent to the best position is remembered and following agents must take into account already reserved vertices. In contrast to the centralized greedy algorithm, decoupled approach does not necessarily lead to the best possible next placement from a given state, because agents whose target is computed earlier may prevent other team members to move on more suitable vertices. Decoupled greedy method is always worse than the centralized one. On the other hand, decoupled algorithm does not need to generate all possible moves, which makes it suitable when we integrate it with other approaches, where efficiency is important.

Decision for the next move is determined by the evaluation function. It is not surprising that two greedy methods with different evaluation functions behave differently. Consider following situation in the figure 3.1 on the right. If the green agent uses distance evaluation, it seems more advantageous to remain in the corner (2 steps from the target) rather then move one cell down (3 steps from the target). But since the red agent 1 will not move anymore, greedy strategy with distance evaluation will not succeed in this instance, because it will not force the green agent to move towards its target and red agent 2 reaches its target after 4 steps. If the greedy method used empty distance evaluation, the right path for the green agent would be found.

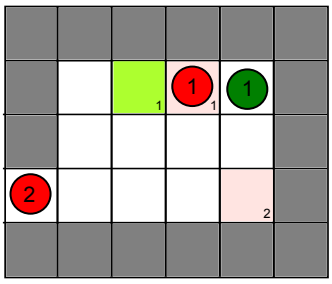


Figure 3.1: Failure of the distance evaluation

It is generally known that greedy methods are myopic and it holds in our case as well. Greedy algorithms are sometimes not able to find paths for agents, even if any interaction with opponent's agents occurs. Figure 3.2 shows a simple situation, where two agents from the same team have to get round each other in a narrow corridor. The optimal movement of the agents is marked with dashed arrows. Agent 2 should reach its target after 5 steps. Unfortunately, greedy methods will never find this solution, because the side steps of agent 2 means temporary deterioration of its position, and greedy method would never let it happen. Cooperative approach described in the following section would discover this solution and lead the agents to their targets.

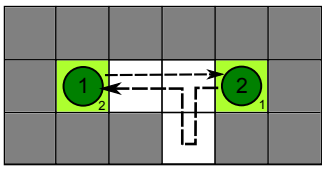


Figure 3.2: Instance that cannot be handled by greedy strategies

3.2 CPF Approach

This approach uses methods developed for CPF problems. Naturally, we cannot simply ignore the opponent's agents. On the other hand, we do not need to treat them as an adversaries, but rather as obstacles in the environment, that are able to change their locations every time step they are to move. We must follow the rules for movement (see section 2.4.2), but it is not necessary to care about our opponent's targets, although it could be also included.

Common foundations of many CPF methods is an adaptation of A* algorithm for multi-agent environment. Similarly works experimentally tested method that is introduced here. In the following text we will call the suggested technique ACA* (Adversarial Cooperative A*).

3.2.1 Adversarial Cooperative A*

Suggested ACA* algorithm is based on CA* technique. We compute the paths for our agents in a spatial graph and every time step during the simulation we select next planned move from pre-calculated paths and check, whether the move

is legal taking into consideration all adversaries, that can unpredictably change their locations. If it is, the move is carried out, otherwise we have to re-plan. Pseudocode 3 shows the ACA* algorithm.

Algorithm 3 Adversarial Cooperative A*

```
paths ← plan paths using CA* for each member of a selected team
while terminal condition not satisfied do
  move ← select next move from paths for all agents
  if move is currently illegal then
    paths ← plan paths using CA* from current position
    move ← select next move from paths for all agents
  end if
  play(move)
  opponent's turn
end while
```

Re-planning

Path plan update is absolutely essential. We can never know the opponent's answer (if there is more than one possible move, which usually is), and hence we cannot get by re-planning. It is not specified how the re-planning phase should be performed. One possibility is throw away all other paths every time a conflict is detected. Advantage of this action consists in finding more suitable global plan for present situation. On the other hand, we often have to spend more time than necessary. Alternatively we can try to repair only conflicting paths and keep those that are still admissible. This approach avoids unnecessary computing, although perhaps neglects more favourable solutions. Potential combination assumes for example detection of a conflicting agent and an attempt to repair its path. If such minor adaptation does not help, algorithm re-calculate all paths again.

A question about how to deal with adversaries during path-planning arises naturally. To be more precise, how to cope with vertices, that are in the current situation occupied by adversarial agents. There are basically two options:

- **Ignore:** paths are calculated as there were only teammates and all adversaries are omitted. The idea is based on assumption, that adversarial agents also changes their positions and one should hope, that the currently occupied vertex will be clear when the agent reaches it.

- **Avoid:** suppose that adversaries are not going to move from their current vertices and thus we need to find more suitable path, possibly longer.

There also exists a compromise between the two cases. We can ignore the locations of the adversaries when they are far away, but try to avoid them when their distance is shorter then some threshold. Another possibility is to avoid only those agents that already accomplished their targets.

It is easy to understand, that the approach that completely ignores adversaries will often fail to lead an agent to its target. When the paths is planned through an occupied vertex, the agent would possibly wait forever near the vertex expecting

that the opponent will eventually leave it. Hence at least some threshold is necessary.

Unfortunately, ignoring opponent's agents in some types of instances is very precarious and may cause inability to achieve target vertex for an agent, who's target is reachable otherwise. Figure 3.3 illustrates a sample situation. Suppose that the threshold is set for 2, so the algorithm will ignore agents that are further than 2 steps from the processed agent. Green agent 1 can easily reach its target by going around the chain of red adversarial agents. However, the variant of ACA* that ignores adversaries would lead it along the right wall through the opponents. When it arrives on the vertex below red agent 1, the re-planning procedure will repair the path plan and provide new one steered through vertex with red agetn 3, which was omitted, because is 3 steps away from the current location of the green agent. When the agent launches out, reaches the location below the red agent 3 and realizes that the next planned step is impossible, re-planning will update again its schedule. This time the red agent 1 will be ignored again and so forth. So the resulting movement of the green agent is patrolling under the chain of adversarial agents.

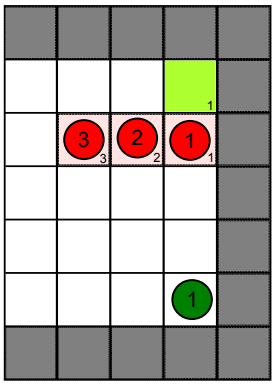


Figure 3.3: Tricky instance for ACA* with opponent omitting

This problem would not occur if at least opponent's agents standing on their targets were not neglected and path plans would be adjusted according to their locations. One should also mention that the situation in the figure is not fully displayed, otherwise it would be a winning placement for the red team. There are other agents that did not arrive on their desired vertices, but they are not relevant for this particular example.

Even though we try to avoid pointless waiting near an opponent who is not moving from its location, excessive circumvention is also unintentional, because it might make place for opponent. A simple heuristic can be suggested: as we have information about opponent's targets, we will try to go around adversarial agent only if it stays on its target, otherwise we will adamantly stay and wait for a reaction of the opponent.

Agent Priorities

As already mentioned in the section 1.3.2, prioritizing of the agents may have an impact on ability to find a solution. Several heuristics for sorting agents can be implemented in order to improve finding paths leading to all agents' targets. Agents can be sorted by different attributes, for example distance to its target, distance to the closest opponent, number of necessary re-plannings so far, etc.

The heuristic employed in experimentally tested algorithm is inspired by 'fail-first' [9], a heuristic known from dynamic variable ordering used in constraint programming. In our case it means, that we compute paths for agents with less unoccupied adjacent vertices earlier then for agents with more unoccupied adjacent vertices. This idea is based on assumption, that those agents with higher

freedom are more likely to find a suitable path, therefore we can process them later and rather focus on more vulnerable teammates.

3.3 Game methods

Traditional approaches for games seem to be most suitable options. However, a closer look at the problem reveal some difficulties. These difficulties are consequent upon the problem complexity, especially its excessive branching factor growing exponentially with the number of agents.

3.3.1 Game tree

Let us imagine some initial position of the ACPF problem as a root of a tree. All possible situations that can be achieved in 1 step are descendants of the root. Their descendants will be constructed in the same way etc. First we focus on ACPF instances with two players called MIN and MAX [1]. A position which satisfies terminal conditions (i. e. all agents or a required number of agents of a particular team reached their target positions or maximal allowed number of steps were accomplished) represents leaves v_i of the tree. Leaves have assigned a number $value(v_i)$ such that

$$value(v_i) = \begin{cases} 1 & \text{if the corresponding state is winning for player MAX} \\ 0 & \text{if the state is a draw} \\ -1 & \text{if the state is winning for player MIN} \end{cases} \tag{3.1}$$

Optimal strategy can be determined from the *minimax value* of each node [23]. Minimax value stands for the utility of one player being in the corresponding state, assuming that both players play optimally to the end of the game. Formally, the minimax value for a state s is

$$minimax(s) = \begin{cases} value(s) & s \text{ is terminal state} \\ \max\limits_{a \in Actions(s)} \{minimax(f(s,a))\} & \text{player MAX to move} \\ \min\limits_{a \in Actions(s)} \{minimax(f(s,a))\} & \text{player MIN to move} \end{cases} \tag{3.2}$$

$f(s, a)$ stand for resulting state if action a (move of entire team) is applied in state s.

For multiplayer games, which includes ACPF, single value of each node is replaced by vector of values, where individual components give utility of the state from each player's viewpoint.

Action selection process requires trying of different sequences of actions and determine a value of situation that has arisen. From the very beginning of the game strategy programming, the selection process had been divided into two distinct approaches [27]: Brute force search (type A) and selective search (type B). This theory dates back to the fifties, when power of computers did not achieve

[1] These terms are prevalent in literature.

the power for effective brute force utilization and hence selective search was rather promoted. The importance of brute force method increased with progressive advancement of computers. In case of ACPF, most instances are too complex for brutal force, even using modern computers.

3.3.2 Minimax approach

Minimax algorithm is a well known example of a brute force algorithm, detailed description is provided for example in [23]. The efficiency of the minimax algorithm can be very easily improved by a slight modification of the algorithm. This enhancement is called alpha-beta pruning. The idea is grounded by observation, that some branches, that are clearly worse than other branches explored so far, can be ignored without any risk of losing the optimal result.

An example of a game tree is depicted in the figure 3.4. Numbers in the nodes stand for their minimax value. Basic minimax algorithm would have to visit every node in the tree, while alpha-beta pruning avoid searching non-promising gray subtrees . No matter whether we use pruning or not, for a certain depth the algorithm always returns the same node.

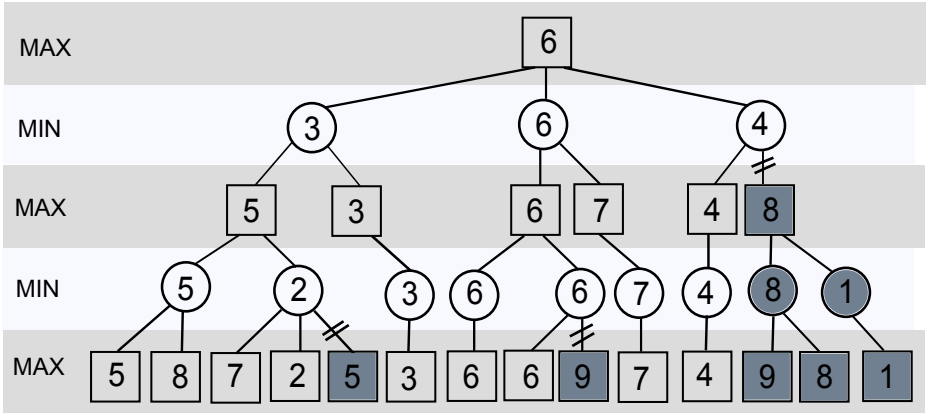


Figure 3.4: Alpha-beta search tree

Let b denote a branching factor and d be the assigned depth. Then we get

$$b + b^2 + ... + b^d = b^{d+1} - 1 \tag{3.3}$$

performed moves, b^d callings of the evaluation function and thus exponential time complexity. Algorithm 4 shows detailed pseudocode of the alpha-beta algorithm. From a current position (root) we call method `AlphaBetaMAX` with the widest possible (α, β) interval. Functions `AlphaBetaMAX` and `AlphaBetaMIN` are defined in terms of each other, or so called mutual recursion.

Algorithm 4 Alpha-beta pruning algorithm

```
function AlphaBeta()                    ▷ call from the root
  score ← AlphaBetaMAX(−∞, +∞, depth)
end function

function AlphaBetaMAX(α, β, depth)
  if depth == 0 then
    return evaluate_state()
  end if
  possible_moves ← generate_moves()
  for move ∈ possible_moves do
    play_move(move)
    score ←AlphaBetaMIN(α, β, depth − 1)
    undo_move(move)
    if score ≥ β then
      return β                          ▷ beta cutoff
    end if
    if score > α then
      α ← score
    end if
  end for
  return α
end function

function AlphaBetaMIN(α, β, depth)
  if depth == 0 then
    return −evaluate_state()
  end if
  possible_moves ← generate_moves()
  for move ∈ possible_moves do
    play_move(move)
    score ← AlphaBetaMAX(α, β, depth − 1)
    undo_move(move)
    if score ≤ α then
      return α                          ▷ alpha cutoff
    end if
    if score < β then
      β ← score
    end if
  end for
  return β
end function
```

Iterative deepening

One possible disadvantage of the alpha-beta algorithm consists in the inability to stop the computation at any time or "on demand". The algorithm has to finish the level of the search tree in the assigned depths, otherwise some important

branches would be neglected, which may lead to an inaccurate result. However, this drawback can be overcome by using so called iterative deepening technique, which combines both BFS (breadth first search) and DFS (depth first search) approach. At the beginning, the iterative deepening runs minimax (or alpha-beta) search with depth 1 and every next iteration increases the depth. Best successor found so far is stored and computation can be stopped in any time, even if the current iteration has not finished. Best successor from the previous iteration would then be returned.

Algorithm 5 Iterative deepening

```
function IterativeDeepening(maxDepth)
  for depth = 0 to maxDepth do
    bestMove ← findBestMove(depth)          ▷ using DFS
  end for
  return bestMove
end function
```

Time complexity of the iterative deepening search is certainly worse, but asymptotically remains unchanged:

$$db^1 + (d-1)b^2 + \cdots + 3b^{d-2} + 2b^{d-1} + b^d \in \mathcal{O}(b^d) \tag{3.4}$$

Performance of the alpha-beta algorithm is affected by move order. If more promising moves are calculated earlier, then bigger branches are pruned which leads to faster algorithm running. Since iterative deepening evaluates the moves during every iteration, we can order them according their value from previous iteration.

Move returned by some minimax algorithm using iterative deepening will be always the same as move returned by minimax without this technique, provided they both searched to the same depth. Only in the first case the calculation would take longer time. So the reason for it is mainly when the computation is limited by computational time instead of search depth, or in some applications, when we need to react on a user input and return the result whenever the user wishes (some user event occurs).

3.3.3 Monte Carlo approach

Monte Carlo methods have wide usage in various applications including numerical algorithms, statistical physics and artificial intelligence.

General characteristics

Monte Carlo Tree Search (MCTS) represents a method for finding optimal decisions that combines tree search and random simulations [5]. In general, MCTS is usually useful when standard methods fails due to excessively large state space, difficult evaluation function or complicated rules. Besides that, it is suitable for games with imperfect information or certain randomness. MCTS were successfully applied in computer Go [6], where surprisingly outperformed alpha-beta algorithm.

Important property of MCTS follows from the fact that it does not need any domain specific knowledge and hence is useful in games where calculation of a fitness function is difficult. Unlike depth limited minimax, The only necessary knowledge is the ability to recognize terminal state. However, additional problem dependent knowledge can achieve improvement.

Basically the algorithm is composed of 4 stages:

- Selection
- Expansion
- Simulation
- Backpropagation

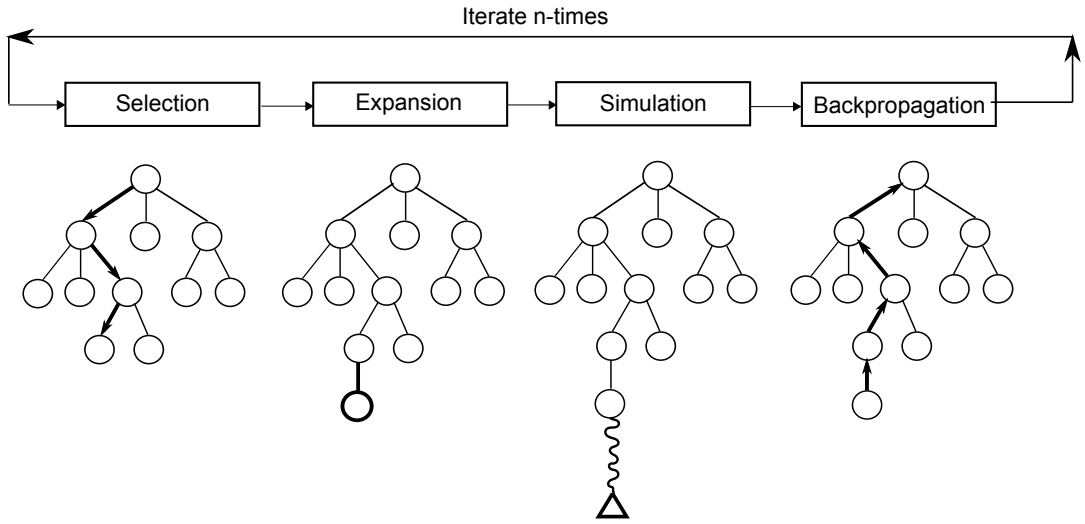


Figure 3.5: Stages of MCTS

Selection phase starts in the root node and recursively selects children until a node representing a non-terminal state or having unvisited children is reached.

Expansion phase adds one or more child nodes corresponding to the actions available from a state represented by node selected in the previous phase. Selection and expansion are often called *tree policy*.

A sequence of actions is run during the *simulation* phase until a terminal conditions are fulfilled, typically a terminal state is reached. This simulation produces a reward value. Term *default policy* is sometimes used for addressing the simulation phase.

Reward value is then during the *backpropagation* stage propagated up and visited nodes update their statistics.

These stages are repeated in certain number of iterations and game tree is gradually built. The more iterations are accomplished, the larger search tree is built and hence the performance of the algorithm increases. The way that determines how are the nodes selected (tree policy) represents crucial characteristics of the MCTS algorithm. More formal description of the algorithm 6 is presented at the end of this section.

Before we give an overview of MCTS variants, lets make a reference to the Bandit problem and exploitation-exploration dilemma, that are important for understanding the algorithm run.

Bandit problem

Suppose we have K arms of multi-armed bandit slot machine and the objective is to sequentially choose among K possible arms in order to maximize the total reward. Reward distribution is unknown at the beginning, therefore the every next optimal action selection must be guessed according to previous observations. This leads to the well known exploitation-exploration dilemma: should we exploit the arm that is currently believed to be optimal, or should we rather explore other possibly suboptimal actions, that could potentially turn out to be advantageous?

A possible approach is called UCB1 (Upper confidence bound) [2]. Selected arm k satisfies

$$k \in \arg\max_{j \in J} \left(\bar{X}_j + \sqrt{\frac{2 \ln n}{n_j}} \right) \tag{3.5}$$

where $\bar{X}_j$ is an average reward from arm j, n is a total number of attempts and n_j denote how many times was the arm j selected.

Upper Confidence Bounds for Trees

Combining of the MCTS algorithm and UCB approach for solving bandit problem leads to the Upper Confidence Bounds for Trees (UCT) algorithm. Individual vertices of the MCTS tree are regarded as individual arms of the bandit problem. During the selection phase, every successor of a vertex is selected in order to meet

$$k \in \arg\max_{j \in J} \left(\bar{X}_j + 2C_p \sqrt{\frac{2 \ln n}{n_j}} \right) \tag{3.6}$$

Constant C_p adjusts exploitation-exploration ratio. If for example $C_p = 0$, the successor with the highest average reward is selected in every step. The higher C_p constant, the more favourable exploration.

In comparison to minimax, tree search carried out in UCT is more resembling human approach People usually do not try to consider exhaustively all possible options, but rather focus on promising ones and examine them more carefully.

Variants

Several variants of MCTS method were proposed in [5]. The method itself does not specify tree policy, i. e. how are the nodes of the tree selected and expanded.

Flat Monte Carlo is a simple Monte Carlo based method that does not build a search tree. It merely runs certain number of random simulations from a current state and returns the successor with the highest average reward.

Flat UCB regards the current state successor selection as a bandit problem. Similarly to the Flat Monte Carlo, Flat UCB does not build a tree.

MCTS builds a tree using a tree policy.

UCT is a MCTS with an arbitrary UCB policy. Every node on the path from root to leaf during the selection phase is treated as a bandit problem.

Greedy move probability

We believe that greedy move probability were firstly introduced in this work. It is a heuristic that can be used during the simulation phase. Moves are not selected only randomly, but with a certain probability p we can use a greedy move that can direct the simulation phase. Random move is then played with probability $1 - p$.

Since the simulation phase has to try large number of moves, it would be very time consuming to evaluate all possible moves and select the best one, especially with high number of agents. Instead of that we use decoupled greedy approach described in the section 3.1.

Algorithm 6 general MCTS

function UCTSearch(s_0)
  create root node v_0 with state s_0
  while within computational budget **do**
    $v_l \leftarrow TreePolicy\,(v_0)$
      ▷ $State\,(v)$ denotes a state corresponding to vertex v
    $value \leftarrow DefaultPolicy\,(State\,(v_l))$
    $Backup\,(v_l, value)$
  end while
  return $Action\,(BestChild\,(v_0, 0))$
end function

function TreePolicy(v)
  while v is nonterminal **do**
    if v is not fully expanded **then**
      return Expand(v)
    else
      $v \leftarrow BestChild(v)$
    end if
  end while
end function

function DefaultPolicy(v)
  while s is nonterminal **do**
    choose random $v \in Successors\,(v)$
    $s \leftarrow f\,(v, a)$ ▷ apply action a in state s
    return reward for state s
  end while
end function

function Backup($v, value$)
  while $v \neq$ null **do**
    $N\,(v) \rightarrow N\,(v) + 1$
    $Q\,(v) \rightarrow Q\,(v) + value$
    $v \rightarrow Parent\,(v)$
  end while
end function

function Expand(v)
    ▷ $Actions\,(s)$ denotes all feasible actions from state s
  choose $a \in$ untried actions from $Actions\,(State\,(v))$
  add a new child v' to v
  return v'
end function

function BestChild(v)
  return $\arg\max_{v \in v'} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2 \ln N(v)}{N(v')}}$
end function

3.4 Exploiting instance properties

Games such as go [6], chess [27] and others were studied in detail and appropriate algorithms were tailored to their needs. Unlike most board games, solving ACPF is not that straightforward, because the graph, numbers of agents and their placements are not fixed. Therefore some methods can be successfully used in some particular situations and fail in others. This fact suggests to decide solving method when the problem instance is known and perhaps combine more approaches or possibly change them during the simulation.

3.4.1 Offensive and defensive tactics

One possible suggestion is motivated by a security operation with a protected person (VIP) and guards who are trying to secure relocation of VIP to its destination as suggested in [16] along with other ideas. Three different roles for agents are proposed, let us call them *VIA* (very important agent), *guard* and *attacker*. Roles merely indicate that agents are treated differently by a solving algorithm. An objective of every VIA is to reach its target as soon as possible, while guards should support them and defend them against opponent. Attackers try to harm adversarial agents for example by preventing opponent's VIA to reach its target.

The definition of ACPF allows instances where several agents do not have any particular target location. An agent without any target can be used as guard or attacker while an agent with a target should be treated as VIA and protected by other agents. Agent's role is not necessarily fixed, but can vary during the simulation.

Game-based methods would naturally use agents without target in order to find the most suitable move. Cooperative and greedy algorithms would normally ignore them, but simple heuristic can assign them new targets. New target could be some adversarial agent's target and the agent would become an attacker.

3.5 Method Summary

All approaches addressed in this chapter have some typical characteristics. They can be distinguished by their basic principle, dealing with adversaries or time complexity.

To sum up this chapter we include following table containing selected attributes of studied methods:

Algorithm	Principle	Coupled (C)/ Decoupled (D)	Call
Greedy 3.1	-	C	every move
ACA* 3.2.1	path-finding	D	when needed
Alpha-Beta [27]	state space search	C	every move
MCTS [5]	state space search	C	every move

Table 3.1: Properties of studied algorithms for ACPF

4. Experiments

In this chapter we describe performed experiments and show their outcomes. We implemented several strategies that were supposed to solve ACPF instances. Strategies were tested and compared in different environments (graphs and agent placements). An application for testing and comparing strategies were developed and its user documentation can be found in the appendix 4.2.5, which was created in order to ensure reproducibility of the experiments.

Some strategies have adjustable parameters and experiments were carried out with different settings. Our objective is to experimentally measure convenient values of the parameters within particular technique and subsequently compare individual algorithms among others. We will also try to identify and categorize types of instances suitable for individual techniques.

4.1 Methodology

Problem definition in chapter 2 is very general and for our experimental reasons we will consider several simplifications, so we can model practical problems in more representative way. Following simplifications were employed:

- Graph is a 4-connected grid with possible obstacles
- Always two adversarial teams
- An agent has either one particular target vertex or no target at all (every vertex can be its terminal position).

4-connected grid graph $G_{m,n}$ is defined as graph Cartesian product $P_m \times P_n$ of path graphs on m and n vertices [41].

4.1.1 Instance pattern generation

For obtaining more reliable data, we need to conduct as many runs on different graphs as possible. Although individual graphs should be various, usually we need graphs fulfilling definite properties like width, height, number of agents etc. Sometimes we also demand agent and target placement limited to certain positions.

For this purpose we use so called *map patterns*, that define properties of graph and agents. Particular instances are generated randomly, but have to fulfill defined pattern.

A pattern defines a graph and 2 subsets of vertices, *initial region* and *terminal region* for each team. Initial region of team t determines all possible starting positions (sources) for members of t, while terminal region dictates possible target vertices of agents from t. In general, vertices belonging to one region are not required to be adjacent and all regions can overlap arbitrarily. Exact positions are generated randomly. There is also information about number of agents and how many of them do have particular target veretx.

4.1.2 Scenarios

Definition of the ACPF problem offers a wide range of possible problem instances. These instances may be very diverse: various size and structure of a graph as well as number of agents might significantly influence a performance of studied algorithms. Furthermore, different algorithms can be more successful in different environments.

In this work we try to capture diverse classes of instances and test the algorithms on them. We will use term *scenario* for a class of instances. All these scenarios can be parametrized by several attributes.

Following paragraphs describes particular scenarios and their properties. Appendix 4.2.5 contains explanation of diagrams present in this chapter.

Exchange

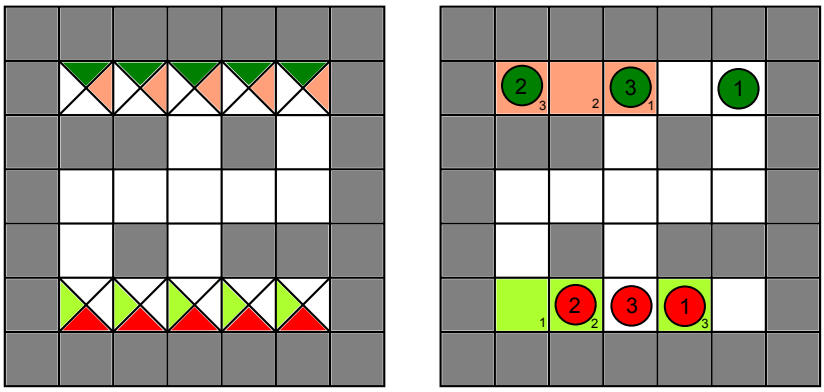
Initial region of team t_1 is identical to the target region of adversarial team t_2 and terminal region of t_1 is identical to the initial region of t_2, in other words, teams try to exchange their positions. Initial and target regions of a team do not overlap. Maximal number of agents in a team t is less or equal than the number of vertices of the t's initial region. Exemplary map is depicted in the figure 4.1a and its possible randomly generated instance is showed in the figure 4.1b.

Race

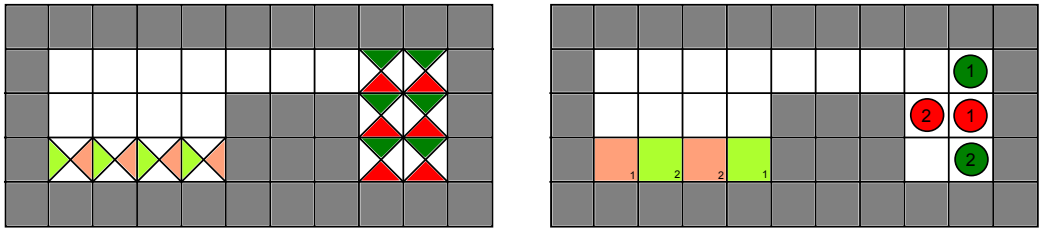
Initial region and terminal region of both teams are identical. Similarly to the previous case, the initial and terminal regions do not have any common vertices. Teams race each other in positioning the agents at their targets. Size of a team must not be greater than half of the initial region, because this region is shared with the opponent's team. Figure 4.1c shows and example of a race map. Figure 4.1d depicts a possible instance generated from the map.

Mingled

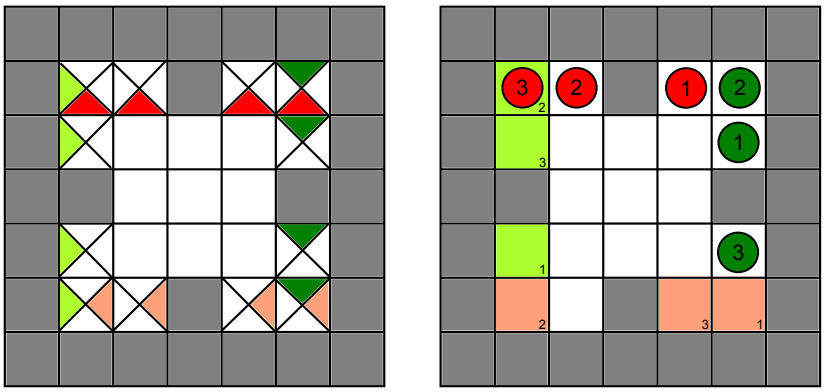
Mingled scenario supposes distinct terminal and initial regions for both teams. No identical regions are allowed. Initial and terminal regions of a team must not have common vertices. Some overlaps between regions belonging to two different teams are allowed. Shortest paths from initial region vertices to target region vertices of one team should cross respective paths of the other team. The idea for this scenario is an imitation of a crossroad. Exemplary map and its instance are showed in the figure 4.1e and 4.1f respectively.



(a) Exchange map **(b)** Exchange instance



(c) Race map **(d)** Race instance



(e) Mingled map **(f)** Mingled instance

Figure 4.1: Exemplary maps of individual scenarios and their potential randomly generated instances

4.2 Results

This section presents conducted experiments. Settings of individual experiments are described and their outcomes are showed. We also try to discuss and explain particular outcomes. In the end we provide a general findings that emerged from our experiments.

4.2.1 Greedy deciding

Greedy strategy is the simplest approach without many adjustable parameters. It is possible to choose between two different evaluation functions according which

the greedy move is selected. The result of following experiment may seem surprising, but closer contemplation suggests an explanation.

Distance vs. empty distance evaluation

In this experiment we examine behavior of distance evaluation (do not consider other agents when computing distances to the targets) and empty distance evaluation (consider agents) defined in section 2.4.4. These two versions of greedy strategy repeatedly plays against one another. Experiments were performed in two different scenarios. Firstly in exchange scenario without obstacles and then in mingled scenario with obstacles. Both scenarios employed a grid graph 7×7 with the same number of agents in both teams, each agent had a single target. Results are presented on graphs 4.2 and 4.3.

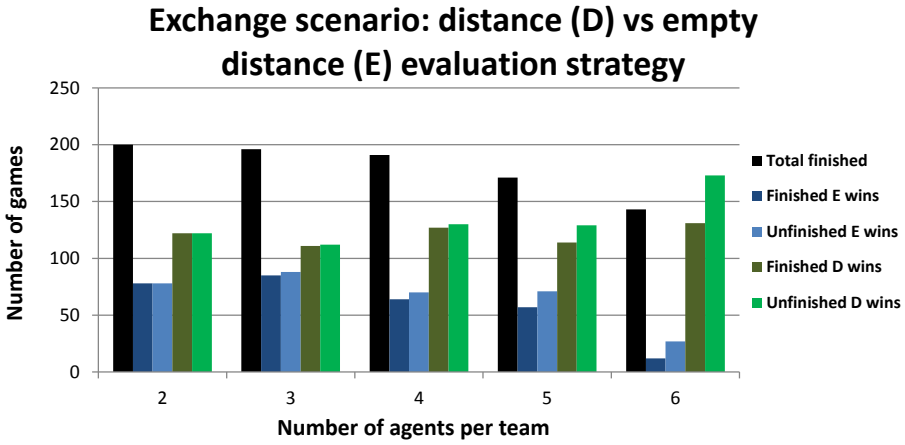


Figure 4.2

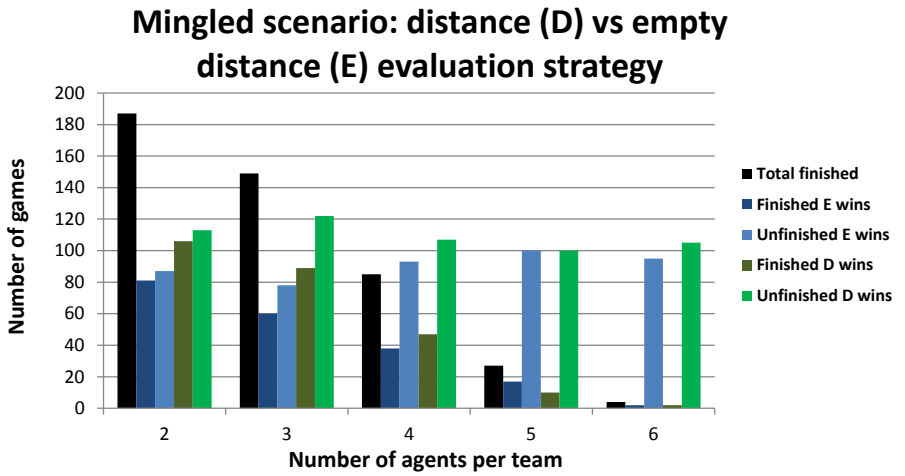


Figure 4.3

Horizontal axis determines number of agents in one team, vertical axes represents number of runs. For every number of agents we see five columns: blue color

denote winning runs of the team that uses empty distance evaluation strategy (team E). Winning runs of team that uses distance evaluation strategy (team D) are represented by green color. Dark blue and dark green columns represent simulations when all agents belonging to a particular team reached their targets (finished runs). Their sum gives the black column, which denotes number of finished runs, no matter which strategy won. Light blue and light green columns stand for the runs when not all agents necessarily reached their targets. Winner was determined after position evaluation when time limit elapsed. Light blue and light green columns sums up to 200, which was the total number of simulations for each number of agents: 100 runs with D team first to move and 100 moves with E team first to move.

As the strategy with empty distance evaluation provides more realistic estimation, it was supposed to outperform the strategy with distance evaluation. Unexpectedly the strategy with just distance evaluation is more successful in our experiment. The explanation of this counterintuitive conclusion is that agents of team D try to go around adversarial agents, which often clear path for the opponent's E team that just try to approach the targets without compromises. Thus the "stubborn" approach seems to be much more successful.

4.2.2 Cooperative path-finding approach

Adaptation of cooperative methods in adversarial environment gave us ACA* algorithm. We will examine the behavior of this algorithm while using different settings.

Agent sorting

We already discussed, that cooperative algorithms based on spatial search may fail to find paths when using certain agent order, while for another order it can successfully find required path plans. Aim of this experiment is to compare performance of two cooperative approaches, when one uses agent sorting from section 3.2.1. We would like to find out whether the different performance will be manifested in our adversarial adaptation.

All simulations of this experiment are conducted in exchange scenario on a symmetric grid map with dimension 10×6 cells and 4 obstacles Agents and targets are placed on the first two, resp. last two columns. Number of agents were gradually increased from 1 up to 10 per each team. 10 simulations were accomplished for every number of agents. Opponent's team were guided by greedy strategy. Every time the search algorithm fails to find paths, we try to re-arrange order of the agents in a team. We observe and compare numbers of necessary rearrangements.

From the graph 4.4 is perceptible that initial sorting of agents influences the number of necessary agent rearrangements. It is more likely to find paths with some heuristic agent sorting.

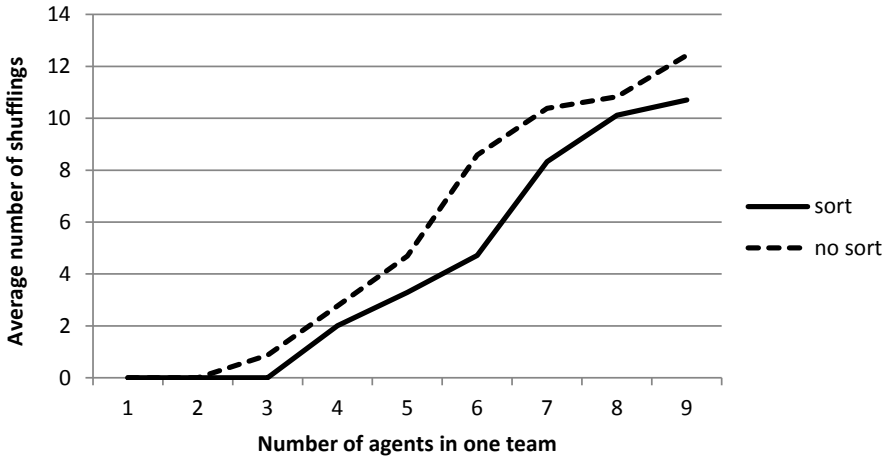


Figure 4.4: Average shuffle count depending on number of agents

Maximal number of rearrangements in our experiment were 30. After exceeding this limit the algorithm concluded that there is no way how to reach targets and let the whole team stay on the current places. Indeed, the algorithm does not try all possible agent ordering, because for k agents there are $k!$ different orderings and to try them all would be too time consuming. Hence we try a limited number of orderings and if we don't succeed to find paths, we do nothing and wait for opponent's response. Search is then started again as soon as our team is to move.

4.2.3 Minimax methods

Minimax algorithm is together with alpha-beta pruning a traditional method for solving game-like problems. By this experiment we gain a better insight into its properties when applied on ACPF.

Number of leaves

Every time minimax or alpha-beta algorithm reach a leaf node, it has to evaluate the position corresponding to the current leaf. Minimax must visit all possible leaves, but pruning can significantly reduce their number. This experiment tries to find out how significant the reduction is in case of ACPF.

We perform two types of experiments concerning number of leaves. First we use exchange scenario on grid graph 7×7. Search depth was set on 2 steps and the number of agents varies from 2 to 6 per each team. Following table shows the results.

Number of agents	no pruning no sorting	pruning no sorting	pruning sorting
2	283	113	52
3	4730	464	392
4	52336	6966	1408
5	1736716	28970	9224
6	-	36096	28432

Table 4.1: Number of leaves for different minimax settings and agent counts

It is clear how the alpha-beta pruning rapidly decreases number of leaves and hence brings a major improvement. In our case the reduction was even 2 orders for 5 agents. Basic minimax used in environment with 6 agents on both sides is not able to work in a meaningful time. Furthermore move sorting at the root node can also reduce the number of leaves, although the difference is not so noticeable.

Second experiment concerning number of leaves were conducted using one particular ACPF instance. Figure 4.5 shows a progress of the leaf count during the run. Left graph depicts all three minimax variants, where the basic minimax without pruning visits significantly more leaves then variants with pruning. Right graph should offer better image of alpha-beta pruning with and without move sorting, because on the left graph it is not easily recognizable.

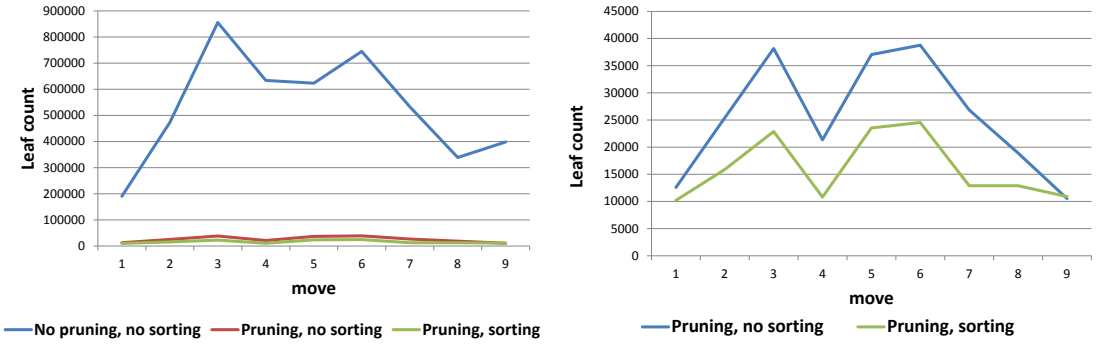


Figure 4.5: Progress of the number of leaves during one run

Again we see that move pre-sorting saves some leaves visits, but we must not forget that the sorting also costs some time.

4.2.4 Monte Carlo methods

Second method for game environments were Monte Carlo tree search, specifically its variant known as UCT. In our experiments we focus on UCT parameter and greedy move probability.

Tuning UCT parameter

As mentioned in the algorithm section, UCT parameter determines the balance between exploration and exploitation of the branches within MCTS tree. The

aim of this test is to find optimal value of the UCT parameter. We try to find the order, where the optimal UCT parameter belongs.

MCTS run against cooperative strategy from the section 3.2. Each tested value of the UCT parameter were used in 100 games.

UCT coefficient	exchange	race
0.001	30	68
0.01	38	56
0.1	30	62
1.0	54	72
10.0	39	61
100.0	29	57
1000	35	63

Table 4.2: Number of won games with different UCT parameter value

Greedy move probability

Greedy moves can be used in the simulation phase of the MCTS algorithm as described in the section 3.3.3. A purpose of this experiment is to estimate the optimal value of the greedy move probability and to see how its increasing value affects success rate of the algorithm.

Experiments were conducted in all three described scenarios: exchange, race and mingled. Width and height of the map was always 7 cells and both sides used 3 agents. First team was led by MCTS strategy, while its opponent used cooperative strategy. 100 matches were conducted for each environment. The result of this experiment is depicted in the figure 4.6

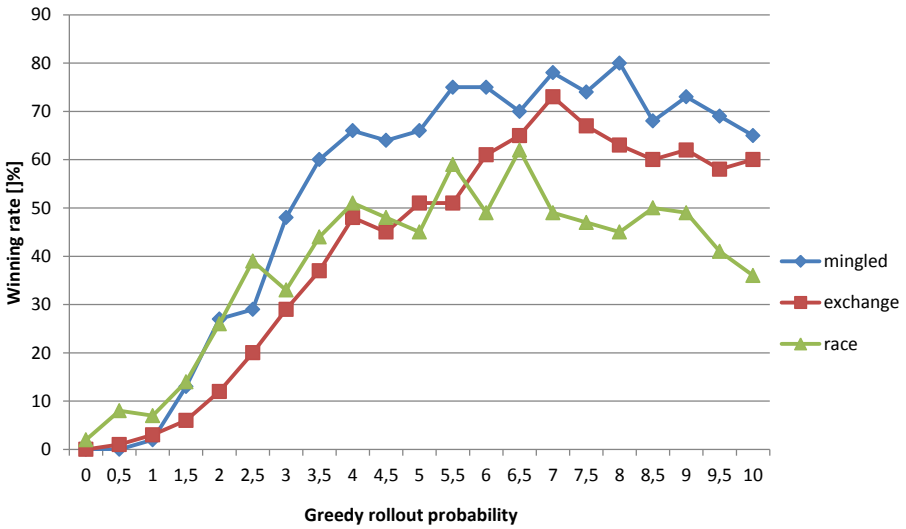


Figure 4.6: Greedy roll-out move probability

From the graph we can clearly see, that the winning rate increases with growing greedy roll-out probability until a certain point. The MCTS with zero greedy

move probability is very weak and almost always loses. The winning rate stagnates around probability 0.7 and rather decreases with higher probabilities.

4.2.5 Strategy tournament

A substantial objective of the experimental section is to provide a summary of performance of studied strategies. We strive to draw a comparison of implemented algorithms. In the previous sections we showed that some algorithms evince different performance with different settings. In this section we try to pick the best settings of each algorithm and conduct a tournament among examined strategies.

We are aware of very different success rates of algorithms in different types of environment. Owing this fact we performed experiments in different types and sizes of scenarios. We inquired into exchange, race and mingled scenario, each with smaller and bigger map.

Certain inequity follows from the advantage of starting player. To overcome this issue we always run 100 matches when strategy A plays in the first step and 100 matches when strategy B begins.

Results of matches are stated in the following tables. Every cell contains two scores. The upper score indicates finished runs, when all agents of one team reached their targets. Lower score is total score, when we consider unfinished runs as well. In such case a winner is determined using empty distance evaluation.

Small instances

Small instances are generated on 6×6 map with 3 agents in both teams. Greedy algorithms uses distance evaluation. Depth of alpha-beta is set on 3, agent sorting is not employed. MCTS expands 4000 nodes and greedy move probability is 0.7. ACA* algorithm uses agent sorting. Heuristic for A* computes distances between vertices ignoring agents.

Large instances

Large instances used map 10×10 with 6 agents at each side. It was necessary to reduce depth of alpha-beta algorithm on 1, because otherwise it did not compute in practical time. MCTS were used in variant Flat UCB, when only first level of the search tree is built. Number of expanded nodes was 14000. Greedy and cooperative algorithms remained unchanged.

Small exchange

Exchange	Greedy	ACA*	Alpha-beta	MCTS
Greedy		34:14 81:19	5:4 44:56	0:12 74:26
ACA*	34:37 43:57		19:45 25:75	34:54 41:59
Alpha-beta	9:6 81:19	42:11 81:19		4:4 88:12
MCTS	20:3 35:65	66:24 70:30	14:8 29:71	

Table 4.3: Strategy tournament with small exchange scenario

Small race

Race	Greedy	ACA*	Alpha-beta	MCTS
Greedy		52:20 58:42	5:26 36:64	25:41 41:59
ACA*	34:40 54:46		10:62 19:81	27:59 38:62
Alpha-beta	27:10 75:25	68:5 84:16		36:3 88:12
MCTS	54:15 71:29	67:20 69:31	9:27 24:76	

Table 4.4: Strategy tournament with small race scenario

Small mingled

Mingled	Greedy	ACA*	Alpha-beta	MCTS
Greedy		47:33 62:38	9:13 37:63	4:27 62:38
ACA*	46:31 54:46		32:34 46:54	21:53 41:59
Alpha-beta	22:4 81:19	60:11 84:16		3:10 83:17
MCTS	36:7 53:47	56:29 65:35	18:6 32:68	

Table 4.5: Strategy tournament with small mingled scenario

Large exchange

Exchange	Greedy	ACA*	Alpha-beta	MCTS
Greedy		27:20 67:33	1:0 52:48	1:0 58:42
ACA*	22:24 42:58		18:33 31:69	8:0 52:48
Alpha-beta	0:0 61:39	37:17 72:28		0:0 53:47
MCTS	1:2 48:52	0:17 49:51	0:3 56:44	

Table 4.6: Strategy tournament with large exchange scenario

Large race

Race	Greedy	ACA*	Alpha-beta	MCTS
Greedy		20:27 52:48	7:7 57:43	8:2 55:45
ACA*	30:13 64:46		38:10 66:34	24:2 59:41
Alpha-beta	11:2 61:39	29:16 61:39		11:1 50:50
MCTS	1:7 50:50	3:18 51:49	1:7 43:57	

Table 4.7: Strategy tournament with large race scenario

Large mingled

Mingled	Greedy	ACA*	Alpha-beta	MCTS
Greedy		45:26 62:38	3:4 61:39	12:0 65:35
ACA*	31:40 45:55		35:28 49:51	11:00 47:53
Alpha-beta	5:0 56:44	32:39 34:26		3:0 49:51
MCTS	0:3 58:42	59:41 49:51	1:3 56:44	

Table 4.8: Strategy tournament large mingled scenario

Findings

From the results of strategy tournament we can infer several conclusions. Game methods on small instances evidently outperform greedy and cooperative methods. This result is caused by the fact, that the state space is not too large and game methods are able to calculate greater part. Greedy method is more successful in exchange scenario, because cooperative algorithm tries to go around, which is what greedy method waits for.

Runs in large scenarios finishes much less often than in small scenarios. Game methods have still quite good results, but they would not be able to compute with higher number of agents. ACA* algorithm is successful in race scenario. Greedy strategy succeeds in exchange scenario again.

Overall result of the strategy tournament is that game methods are good as long as they are able to compute. With higher number of agents we observed that greedy method is surprisingly successful, especially when agents are suppose to go facing each other. Nevertheless a shortcoming of greedy method is that it cannot deal with some situations, that can be easily solved by cooperative strategy. Unfortunately, cooperative strategies in the competitive environment are too loyal towards opponent and hence there are not many types of instances where cooperative algorithm succeeds.

Conclusion

This master thesis addressed Adversarial Cooperative Path-finding, a generalized and extended version of traditional Cooperative Path-finding.

Since the adversarial version of CPF was not introduced in any previous work, the first aim of this thesis was to formally define considered problem. Based on the definitions, we tried to prove its membership in complexity classes. The problem belongs to PSPACE-hard complexity class, which was briefly showed in [16]. An alternative proof of PSPACE-hardness with detailed description and explanation was presented here. Unfortunately our attempts for proving PSPACE-completeness of the problem failed so far and hence it remains an open question. If it were possible to polynomially reduce number of moves, as is possible in CPF, we would be able to prove that ACPF belongs to PSPACE and immediately PSPACE-completeness. Nevertheless at least we have an upper bound, because we proved that the problem is also member of EXPTIME.

Subsequently we described possible techniques for controlling selected team against its opponents. Two main different approaches were discussed: adapted methods known from cooperative path-finding and application of the game methods. The excessive complexity of the problem make game methods impossible when the ACPF instance is big, i. e. when there are too many agents, since the branching factor of the problem grows exponentially with the number of agents. Despite our low expectation, surprisingly successful seem to be greedy methods.

Experimental part provides us a closer insight. We see how does the minimax algorithm perform when different settings are used and how it deals with increasing number of agents. Similar tests were conducted for Monte Carlo methods. The most noticeable conclusion is that game methods can be employed only for instances with limited number of agents. Our hypothesis that MCTS could be a suitable solving technique was disproved. In case of more agents the time spent by computing is excessive and leads us to inference that there has to be another way how to deal with bigger or more crowded instances.

Possible approach is adaptation of algorithms for the cooperative path-finding problems, that were plentifully studied and researched in recent decades. Adversarial Cooperative A*, an adaptation of CA* algorithm, were proposed with different settings. We studied its behavior in various environments. This method is suitable for environments where agents from one team have to cooperate and do not interact with adversarial agents very often. Despite rather weaker performance in the tournament, development of a refined cooperative based algorithm might be interesting task for future research. Investigation of domain dependent heuristics could be a potential starting point.

Bibliography

[1] Broderick Arneson, Ryan B. Hayward, and Philip Henderson. Monte carlo tree search in hex. *IEEE Trans. Comput. Intellig. and AI in Games*, 2(4):251–258, 2010.

[2] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2-3):235–256, 2002.

[3] 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):89–99, 2002.

[4] Zahy Bnaya, Roni Stern, Ariel Felner, Roie Zivan, and Steven Okamoto. Multi-agent path finding for self interested agents. In Malte Helmert and Gabriele Röger, editors, *SOCS*. AAAI Press, 2013.

[5] Cameron Browne, Edward J. Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Trans. Comput. Intellig. and AI in Games*, 4(1):1–43, 2012.

[6] Bernd Brügmann. Monte carlo go, 1993.

[7] Michael Chung, Michael Buro, and Jonathan Schaeffer. Monte carlo planning in rts games. In *IEEE SYMPOSIUM ON COMPUTATIONAL INTELLIGENCE AND GAMES (CIG)*, pages 117–124, 2005.

[8] Boris de Wilde, Adriaan ter Mors, and Cees Witteveen. Push and rotate: cooperative multi-agent path planning. In *AAMAS*, pages 87–94, 2013.

[9] Rina Dechter. *Constraint processing*. Elsevier Morgan Kaufmann, 2003.

[10] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959. 10.1007/BF01386390.

[11] Michael Erdmann and Tomas Lozano-prez. On multiple moving objects. *Algorithmica*, 2:1419–1424, 1987.

[12] P.E. Hart, N.J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4:100–107, 1968.

[13] R.C. Holte, R. C. Holte, M.B. Perez, M. B. Perez, R. M. Zimmer, R. M. Zimmer, A.J. MacDonald, and A. J. Macdonald. Hierarchical a*: Searching abstraction hierarchies efficiently. In *In Proceedings of the National Conference on Artificial Intelligence*, pages 530–535, 1996.

[14] Ko hsin Cindy Wang and Adi Botea. Fast and memory-efficient multi-agent pathfinding. In *In ICAPS*, pages 380–387, 2008.

[15] Haomiao Huang, Jerry Ding, Wei Zhang 0013, and Claire J. Tomlin. A differential game approach to planning in adversarial scenarios: A case study on capture-the-flag. In *ICRA*, pages 1451–1456. IEEE, 2011.

[16] Marika Ivanová and Pavel Surynek. Adversarial cooperative path-finding: A first view. AAAI Late Breaking Track, 2013.

[17] M. Renee Jansen and Nathan R. Sturtevant. Direction maps for cooperative pathfinding, 2008.

[18] M. Renee Jansen and Nathan R. Sturtevant. A new approach to cooperative pathfinding. In Lin Padgham, David C. Parkes, Jörg P. Müller, and Simon Parsons, editors, *AAMAS (3)*, pages 1401–1404. IFAAMAS, 2008.

[19] Daniel Kornhauser, Gary L. Miller, and Paul Spirakis. Coordinating pebble motion on graphs, the diameter of permutation groups, and applications. In *FOCS25*, pages 241–250. IEEE, October 1984.

[20] Petr Koupý. Visualization of problems of motion on a graph, 2010.

[21] Viliam Lisý. Adversarial planning for large multi-agent simulations. In Wiebe van der Hoek, Gal A. Kaminka, Yves Lespérance, Michael Luck, and Sandip Sen, editors, *AAMAS*, pages 1665–1666. IFAAMAS, 2010.

[22] Ryan Luna and Kostas E. Bekris. Push and swap: Fast cooperative path-finding with completeness guarantees. In Toby Walsh, editor, *IJCAI*, pages 294–300. IJCAI/AAAI, 2011.

[23] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (2nd Edition)*. Prentice Hall, December 2002.

[24] Malcolm Ross Kinsella Ryan. Exploiting subgraph structure in multi-robot path planning. *J. Artif. Intell. Res. (JAIR)*, 31:497–542, 2008.

[25] Franisek Sailer, Michael Buro, and Marc Lanctot. Adversarial planning through strategy simulation. In *CIG*, pages 80–87. IEEE, 2007.

[26] Spyridon Samothrakis, David Robles, and Simon M. Lucas. Fast approximate max-n monte carlo tree search for ms pac-man. *IEEE Trans. Comput. Intellig. and AI in Games*, 3(2):142–154, 2011.

[27] Claude Shannon. Programming a computer for playing chess. Technical report, Bell Telephone Laboratories, Inc., Murray Hill, N.J., March 1950.

[28] Guni Sharon, Roni Stern, Ariel Felner, and Nathan Sturtevant. Conflict-based search for optimal multi-agent path finding. In *to appear in AAAI*, 2012.

[29] Guni Sharon, Roni Stern, Meir Goldenberg, and Ariel Felner. The increasing cost tree search for optimal multi-agent pathfinding. In Toby Walsh, editor, *IJCAI*, pages 662–667. IJCAI/AAAI, 2011.

[30] David Silver. Cooperative pathfinding. In *AIIDE*, pages 117–122, 2005.

[31] Michael Sipser. *Introduction to the theory of computation*. PWS Publishing Company, 1997.

[32] Trevor Scott Standley. Finding optimal solutions to cooperative pathfinding problems. In Maria Fox and David Poole, editors, *AAAI*. AAAI Press, 2010.

[33] Trevor Scott Standley and Richard E. Korf. Complete algorithms for cooperative pathfinding problems. In Toby Walsh, editor, *IJCAI*, pages 668–673. IJCAI/AAAI, 2011.

[34] Bryan Stout. *The Basics of A* for Path Planning*. Charles River Media, Inc., Rockland, MA, USA, 2001.

[35] Pavel Surynek. A novel approach to path planning for multiple robots in bi-connected graphs. In *ICRA*, pages 3613–3619. IEEE, 2009.

[36] Pavel Surynek. On pebble motion on graphs and abstract multi-robot path planning. In *Proceedings of the ICAPS 2009 Workshop on Generalized Planning: Macros, Loops, Domain Control. September 20th, 2009, Thessaloniki, Greece*, 2009.

[37] Pavel Surynek. Abstract path planning for multiple robots: A theoretical study, 2010.

[38] M. Walicki. *Introduction to Mathematical Logic*. World Scientific, 2012.

[39] Ko-Hsin Cindy Wang. Bridging the gap between centralised and decentralised multi-agent pathfinding. In *Innovative Applications of Artificial Intelligence*. Fourteenth Annual AAAI/SIGART Doctoral Consortium, 2009.

[40] Ko-Hsin Cindy Wang and Adi Botea. Tractable multi-agent path planning on grid maps. In Craig Boutilier, editor, *IJCAI*, pages 1870–1875, 2009.

[41] Eric W. Weisstein. Grid graph. From MathWorld—A Wolfram Web Resource. Last visited on 03/3/2014.

[42] J. Yu and S. M. LaValle. Structure and intractability of optimal multi-robot path planning on graphs. In *The Twenty-Seventh AAAI Conference on Artificial Intelligence*, 2013.

[43] Alexander Zelinsky. Mobile robot navigation exploration algorithm, 1992.

List of Abbreviations

- **ACPF** Adversarial Cooperative Path-finding
- **BFS** Breadth First Search
- **CA*** Cooperative A*
- **CPF** Cooperative Path-finding
- **DFS** Depth First Search
- **HCA*** Hierarchical Cooperative A*
- **MCTS** Monte Carlo Tree Search
- **MPF** Multi-agent Path-finding
- **UCB** Upper Confidence Bound
- **UCT** Upper Confidence Bound for Trees
- **WHCA*** Windowed Hierarchical Cooperative A*

List of Figures

List of Tables

List of Algorithms

Appendix A

Content of the CD

- **dist:** executable version of the application used for experiments
- **source:** source code of the application (NetBeans project)
- **maps:** files with maps used in the tournament
- **stats:** directory with output files of conducted experiments. Every experiment has its own folder that further contains
  - **graphs** recorded instances, that can be used as inputs again.
  - **runs** recorded progresses of individual matches. These files can be used as an input for the program Graphrec [20].
  - **acpf.properties** property file associated with the experiment
  - **matchinfo.log** file with recorded statistics of the experiment
- **thesis.pdf:** electronic version of this text

Appendix B

SW prototype user documentation

A part of the thesis is also a console application which were used in the experimental part. The application is written in Java. Communication between the application and user is solved by editing the property file of which format is described in the following section.

Property file format

File `acpf.properties` stores the configuration of the application run and is used as an user interface of the program. It defines input file, maximal number of steps before the simulation finishes, number of teams and what strategies do the teams use. Every strategy has settings of its own parameters. Defined items are

Attribute	possible values	description
`input_file`	path to file	A path to the input file
`max_move_count`	$\mathbb{N}$	Maximal number of moves performed
`unsaved_entity_count`	$\mathbb{N} \cup \{0\}$	Number of entities that are not required to reach its target
`simulation_count`	$\mathbb{N}$	How many runs with this settings should be conducted
`<team>.strategy`	$\{mc, cp, ab, gr\}$	Strategy for `team_name`. Abbreviations stands for Monte Carlo Tree Search, Cooperative, Alpha-Beta and Greedy
`<team>.ab.timeLimit`	$\mathbb{N}$	Number of milliseconds that minimax algorithm can use for one move calculation. If 0, then there is no restriction and depth limit is used.
`<team>.ab.depth`	$\mathbb{N}$	Minimax search depth. Value is ignored if the time limit is set.
`<team>.ab.usePruning`	$\{true, false\}$	Should minimax use alpha-beta pruning?
`<team>.mcts.timeLimit`	$\mathbb{N}$	Time limit for MCTS

<team>.mcts.playOuts	$\mathbb{N}$	Number of MCTS iterations
<team>.mcts.uctCoef	$\mathbb{R}_+$	UCT coefficient
<team>.mcts.useFlatUcb	$\{true, false\}$	Should be the MCTS tree constructed?
<team>.mcts.grRollProb	$< 0, 1 >$	Greedy roll-out probability
<team>.mcts.rollCount	$\mathbb{N}$	Length of the MCTS simulation phase
<team>.gr.considerAgents	$\{true, false\}$	Should the greedy strategy consider other agents?
<team>.coop.shuffleCount	$\mathbb{N}$	How many times should the agents be shuffled in order to find paths for agents in different order (current ordering fails)
<team>.coop.sortAgents	$\{true, false\}$	Should the agents be sorted before the first search

Table 4.9: Property file description

Input files

There are two types of input files that will be described in following sections. Name of the input file is specified in the property file.

Graph file is a XML input file that provides unambiguous description of an ACPF instance. File is composed from the XML header and a single root element <graph width="[width]" height="[height]>", where width and height specifies the dimension of the grid graph that will be produced. Root element contains:

- <entity id=[entity_id] startnodeid=[start_node_id] teamid=[team_id] />
- <target entityid=[entity_id] nodeid=[node_id] />
- <obstacle nodeid=[node_id] />

Map file input format Files *.map define exactly how shall the graph look like, but the agent and target placements are ambiguous. Let us look at the following example, that represents small mingled scenario:

```
5 5 3 3 3 3
2_ 0_ 0_ 0_ 01
1_ XX __ XX _1
1_ __ __ __ _1
1_ XX __ XX _1
10 _0 _0 _0 _2
```

First line with numbers describes dimension of the grid graph. In our case the width and height of the graph are 5. Next two digits defines number of agents in selected and adversarial team respectively. Last two numbers indicate how many agents do have a target vertex.

Following lines defines a map pattern. Numbers 0 and 1 respectively denote selected and adversarial team respectively. Number 2 stands for both teams. Character `X` means that there is an obstacle at the represented vertex. Every possible vertex is associated with two characters. Left position defines possible source and right position stands for possible target. So in our example the upper left corner is a vertex, that can be a source vertex of agent belonging to any team. Lower left corner represents a vertex, that can be initial vertex of an agent from the adversarial team, but can be also a target of the selected team.

Output files

Every experiment defines number of runs for a given input file (usually map file). Output files are created into a new folder within `stats` directory. Program creates two types of output files. First are records of the processed instances (graph file from the previous section). In case that the input file was graph file, the output file is actually its copy. Second file defines movement of the agents carried out in the processed instance. These files are inputs for visualization program Graphrec [20].

Appendix C

Several passages of this thesis contain instance and map diagrams to illustrate exemplary situations. This section clarifies and explain the meaning of used symbols. One square in diagrams represents a one vertex of a grid graph.

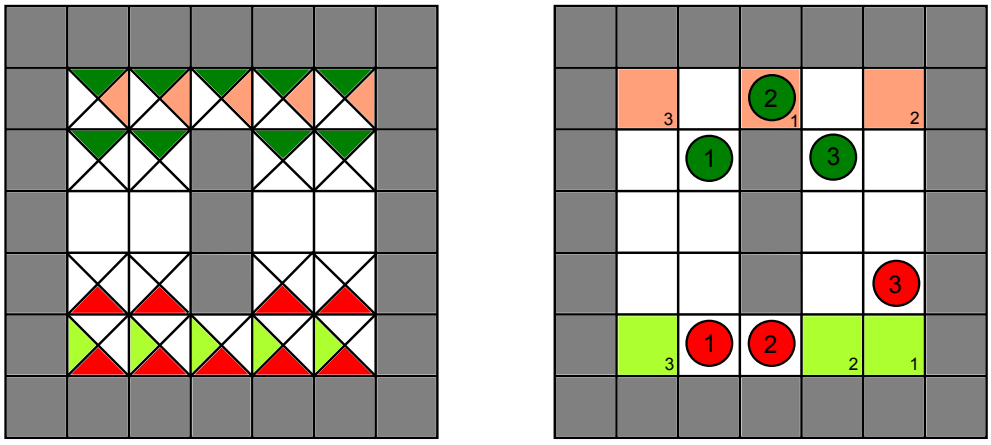
Map patterns

In some cases we would like to describe a whole class of instances satisfying a certain properties. For this reason we introduced so called map patterns. Using map patterns together with given number of players we are able to express where the agents can appear on the map and where are possible locations for their targets. Structure of a grid graph is fixed, while exact agents' and their targets' locations may vary.

Figure 4.7: Meaning of a map cell

The diagram on the right clarifies the meaning of a map cell. There can be up to four types of objects on a cell, each has its position in the square and for better orientation we use a color to distinguish individual objects. Instance generator always assigns at most one agent and at most one target to a cell.

Figure 4.8a depicts an example of a grid map pattern with an obstacle in the center. The map definition also contains information of determined number of agents and targets. The depicted map has 3 agents on both sides, each with a single target. Agents of a player can be placed in the upper two lines, their targets lays on the bottom line. Agents of the adversarial team can initially occupy two bottom lines and their goals lay on the top line.



(a) Map pattern **(b)** Instance diagram

Figure 4.8: Example of a map pattern with 3 agents and possible instance generated from it

Instance diagrams

For describing a particular situation (ACPF instance) we use instance diagrams. These diagrams give information about current position of all agents present in the situation and their targets. Agents belonging to one team are marked off from other teams. We used green circles to characterize the selected team and red circles for the adversarial team.

One possible example that could be generated from previously described map pattern is depicted in the picture 4.8b. Individual agents are marked with numbers. Small numbers in the corner of some cells means, that the vertex is a target location of an agent with the same number. Green color of the cell indicates, that the agent belongs to the selected team. Red color is used for opponent's agents. Possible obstacles and borders are represented as dark gray blocks.