Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Cybernetics

DIPLOMA THESIS

Integration of advanced navigation methods and adversarial reasoning techniques for an ORTS player

Prague, 2012 **Author: Viktor Chvátal**

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěřečných prací.

V Praze dne 7.5.2012

podpis

Acknowledgement

I am heartily thankful to my supervisor Mgr. Branislav Bošanský for many valuable suggestions that helped me to make this work better.

Abstract

Real-time strategy games are complex domains that require both fast reactive behavior and reasoninng about strategies. This work describes development of an artificial intelligence player for strategic combat scenario running in Open Real Time Strategy environment that uses two state-of-the-art methods, potential fields for unit movement and Monte-Carlo tree search for reasoning about strategies. Potential fields method is extended with path finding to avoid problems with local minima, Monte-Carlo tree search uses UCT selection strategy and progressive unprunning to limit number of expanded nodes. Resulting player was experimentally compared to other two existing players that are able to solve the same game scenario.

Abstrakt

Real-time strategické hry jsou komplexní domény, které od hráčů vyžadují rychlé reaktivní chovnání i uvažování o strategiích. Tato práce popisuje vývoj hráče s umělou inteligení pro scénář strategického souboje, který běží v prostředí Open Real Time Strategy. Vyvinutý hráč používá dvě pokročilé metody, potenciálová pole pro pohyb jednotek a Monte-Carlo prohledávání pro uvažování o strategiích. Potenciálová pole jsou rozšířena o prohledávací metodu, která řeší problémy s uvíznutím v lokálních minimech. Monte-Carlo prohledávání používá výberovu strategii UCT a metodu progressive unprunning pro expandování menšího množství uzlů. Výsledný hráč byl experimentálně srovnán s dalšími dvěma existujícími hráči, kteří jsou schopni řešit daný scénář.

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics

DIPLOMA THESIS ASSIGNMENT

Student: Bc. Viktor C h v á t a l

Study programme: Open Informatics

Specialisation: Artificial Intelligence

Title of Diploma Thesis: Integration of Advanced Navigation Methods and Adversarial Reasoning Techniques for an ORTS Player

Guidelines:

Developing a competitive real-time strategy player is a challenging problem as there are several complex subtasks that need to be solved. These subtasks include navigation and movement of the units, map exploration, resource collection, and deliberative reasoning about strategies.

The goal of this diploma thesis is to tackle two subtasks and integrate two state-of-the-art techniques for the tactical-combat scenario player in ORTS.

The player will use (1) potential fields for units movement and navigation, and (2) Monte-Carlo Tree Search method for reasoning about strategies. Using a game-tree search method is directly intractable due to a very large state space; hence, an analysis of various levels of abstractions will also be an important part of the work.

Bibliography/Sources:

[1] R.K. Balla, A. Fern: UCT for tactical assault planning in real-time strategy games. In IJCAI'09 Proceedings of the 21st international jont conference on Artifical Intelligence, 2009.
[2] J. Hagelback, S.J. Johansson: Using multi-agent potential fields in real-time strategy games. In AAMAS '08 Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems. 2008.
[3] M. Buro: ORTS: A hack-free RTS game environment. Lecture Notes in Computer Science, pages 280-291, (2003).

Diploma Thesis Supervisor: Mgr. Branislav Bošanský

Valid until: the end of the summer semester of academic year 2012/2013

L.S.

prof. Ing. Vladimír Mařík, DrSc.
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, January 10, 2012

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra kybernetiky

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: Bc. Viktor Chvátal

Studijní program: Otevřená informatika (magisterský)

Obor: Umělá inteligence

Název tématu: Integrace pokročilých metod navigace jednotek a adversariálního uvažování pro hráče v ORTS

Pokyny pro vypracování:

Pro vytvoření úspěšného algoritmu hrajícího real-time strategii je nutno vyřešit sadu těžkých problémů, mezi které patří pohyb a navigace jednotek, prohledání mapy, sbíraní a vyhledávání zdrojů, nebo uvažování o možných strategiích a o oponentovi.

Cílem této diplomové práce je zaměřit se na dva z těchto problémů a zkombinovat dvě techniky, které se pro tyto problémy používají, do jednoho algoritmu pro hráče řešícího scénář taktického boje v prostředí ORTS.

Hráč by měl používat (1) potenciálové pole pro pohyb a navigaci jednotek a (2) Monte-Carlo prohledávání stromu pro uvažování o strategiích a o oponentovi. Vzhledem k velikosti stavového prostoru v real-time strategiích není možné použít metody prohledávající herní strom přímo a proto bude důležitou součástí práce i analýza různých abstrakcí stavového prostoru.

Seznam odborné literatury:

[1] R.K. Balla, A. Fern: UCT for tactical assault planning in real-time strategy games. In IJCAI'09 Proceedings of the 21st international jont conference on Artifical Intelligence, 2009.

[2] J. Hagelback, S.J. Johansson: Using multi-agent potential fields in real-time strategy games. In AAMAS '08 Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems. 2008.

[3] M. Buro: ORTS: A hack-free RTS game environment. Lecture Notes in Computer Science, pages 280-291, (2003).

Vedoucí diplomové práce: Mgr. Branislav Bošanský

Platnost zadání: do konce letního semestru 2012/2013

L.S.

prof. Ing. Vladimír Mařík, DrSc.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 10. 1. 2012

Contents

List of Figures

List of Tables

List of Abbreviations

AI	Artificial Intelligence
BFS	Breath First Search (A tree search method)
FSM	Finite-State Machine
HP	Hit Points (Amount of health of a game object)
MCTS	Monte-Carlo Tree Search
ORTS	Open Real-Time Strategy (An open-source game environment)
RTS	Real-Time Strategy (A genre of a strategy game)
UCT	Upper Confidence Bounds for Trees (A Monte-Carlo tree search method)

Chapter 1

Introduction

A strategy game is a game in which players reasoning skills are very important to the outcome of the game, and the luck and random moves have almost no significance. Strategy games can vary from board games like chess or go to very complex video games simulating military battles or other real-world situations. A strategy game can generally contain one or more players with various goals.

A strategy game includes some type of objects like checkers or chess pieces, workers, buildings or military units situated in an environment, which can be a board or a virtual world simulation map. The game can allow to arrange the elements only to discrete positions, or they can be put to any continuous position within the game map. The map is usually a finite two-dimensional rectangular area.

Players in a strategy game have similar amount of knowledge abount the game, however, the game can be fully observable to all players, or it can reveal only partial information visible to each player. Board games are mostly fully observable, video strategy games often reveal only elements within the sight range of the players objects.

A strategy game can be benign, like single player resource gathering scenario or area scouting simulations, or it can be adversarial, like chess, go or war simulation games. Games can be both collaborative, where at least two players have similar goals, and competitive, where some players have conflicting goals.

Strategy games can be real-time or turn based. Players in turn-based games do their

turns one after another, each performing limited set of actions in his turn in the given time. Strategy games with continuous time are called real-time strategy games. In such games, the time moves continuously ahead and players do their actions at any time they need. There is no guaranteed time to perform reasoning so players have to make decisions fast enough to be able to compete other players in the real time.

Computer implementations of strategy games represent continuous locations or time as discrete variables with very small step between two neighboring values. Time divisions is usually called time frames, spatial variables are divided into elementary positions. Although these variables are discrete in their representation, they make very good approximation to continuous variables, and we need to handle them as they were essentially continuous, because an enumeration of all possible values would be extremely computationally expensive.

Real-time strategy games are complex simulation enviroments usually describing real-world situations. Players playing in these simulations must solve several problems together, like resource gathering, base construction, training war units, attack coordination and object movement in dynamic envoriments. Robust intelligence needs to include both reactive and deliberative processes to deal with all of the described problems. Different methods usually have to be combined together to solve such problems at different levels of abstraction, so that they are able to compute a solution in a time reasonable for the given game.

Real-time strategy games are a significant part of video game industry [4]. Many of these games include artificial intelligence players that are more or less competitive to human players. These AI players often cheat by retrieving more information about the game than real human players just to be more competitive or to make their development easier.

Artificial intelligence methods used in real-time strategy games can be also used for solving real world problems, but in this case AI cannot rely on cheating by retrieving extra information or resources. This is a reason why it is important to search for new AI methods and combinations to deal with problems which developers used to solve by cheating. To involve developers in AI player development, a simulation environment called Open Real-Time Strategy has been developed[3] to provide universal test bed for diferrent real-time strategy game problems. There were also several real-time strategy

game competitions[5] organised to compare new AI players for ORTS environment.

Aim of this work is to develop methods for AI player working in a war simulation, a real-time strategy game that includes an adversarial, highly dynamic environment with game objects situated at continuous positions. Game elements should be able to quickly and autonomously react to environment changes while achieving a goal given by a deliberative reasoning process. Real-time strategy games allow the player to perform so many different actions, so any search of a complete game tree during the deliberative process is computationaly impossible. That is why a game control has to work with abstractions reducing the total number of possible actions, and the reasoning process must use different methods at distinct levels of abstractions.

Chapter 2 of this work introduces war simulations as most common real-time strategy games and defines terminology that is used across this work. After that it describes most important tasks common to real-time strategy war simulations that have to be solved to build a strong artificial intelligence player. It also discusses differences between AI algorithms developed by researchers and video games companies.

Chapter 3 describes various methods and algorithms can can be used for solving tasks in real-time strategy games. It starts with basics of mathematical morphology, follows with a description of path finding and cooperative movement methods, clustering methods and ends with a description of rule-based systems and solving games by searching.

Chapter 4 contains description of Open Real Time Strategy enviroment that is used for simulation of RTS games. It follows with a description of several scenarios used on RTS games competitions.

Design of an AI player is described in the Chapter 5. It starts with a design of potential fields, it folows with a description of path finding extension to potential fields and ends with a description of Monte-Carlo tree search methods.

Chapter 6 contains results of experimental simulations that have been performed against two existing players using various combinations of AI player settings.

Conclusion of this work is described in the Chapter 7.

Chapter 2

Specifics of Real-Time Strategy Games

War simulations are most common adversarial RTS games. Environments in RTS games are highly dynamic and often partially observable, which is very common to real world situations. Algorithms able to reason about situations in RTS games could be used for example for navigating robots in streets full of people, algorithms for area eploration could be used for searching for people in dangerous areas like buildings that are in fire.

This chapter starts with a description of real-time war simulation games in the Section 2.1. List of tasks that need to be solved in war simulations is mentioned in the Section 2.2. Deliberative reasoning about strategies is introduced in the Section 2.3. Differences of AI players in computer games industry and those developed by academical researchers are discussed in the Section 2.4.

2.1 Description of RTS War Simulation Games

War simulations are typically situated in a rectangular game map that is split into small areas called map tiles. Map tiles can be of different type of terrain and height. A terrain type can be a free ground with no movement restriction, a cliff blocking unit movement, a special terrain like water limiting some units to move across, a place containing resources called resource patch or another terrain type according to the type of a game.

Except special scenarios with tasks like resource gathering, war simulations are two

or more player games. Each player takes control of some game objects which have several characteristics. Most common property of a players game object is amount of its maximal and actual health, also known as hit points (HP).

Game objects are usually divided into buildings and units. Buildings are objects that cannot move and have higher amount of maximal HP, units are able to move across the game map. Special units called workers have special abilities to construct new buildings and possibly to repair damaged ones. Each unit can have differrent maximal speed of movement.

Units and buildings can be equipped with a weapon, defined by a shooting range and strength. The bigger the strength of a weapon is, the more it decreases amount of hit points of its target. Definition of the strength parameter varies from game to game, in some games the strength is deterministic, in other games it is a random variable of a known distribution. Weapons can usually shoot repeatedly after certain periods of time called cool down time, a time interval during which the weapon gets ready for another attack.

Important part of the game is performing attacks. Any game objects equipped with a weapon can shoot at any enemy objects within its weapons range, if it is not in the cool down state. A successfull attack leads to decreasing amount of targets hit points. When the hit points drop to zero, the object gets destroyed and disappears from the game. Some buildings are able to produce new units at a cost of spending resources. Newly produced units and constructed buildings start with the maximal amount of HP. Objects lose their HP as they are attacked by enemy units.

Goals of the war simulations can vary. Most common goal is to destroy enemy units or building, or both. Other goal types can involve gathering some amount of resources, exploring the map, constructing a building of some kind or training an army of a given size.

2.2 Tasks in RTS games

A real-time strategy game usually requires a combination of distinct tasks to be solved in order to fulfill its goal. These tasks may include unit movement, resource gathering,

map exploration or team collaboration. Tasks that have to be solved in RTS games are described in this section.

Most of the problems like adversarial planning, resource management, team collaboration and learning from the opponent behavior are briefly discussed in [4]. Unit movement problem is described for instance in [12], more specialized cooperative group movement problem is discussed in [16] and [24].

2.2.1 Movement of the Units

Unit movement has to be solved almost in any RTS game. Fulfilling goals in the game involves moving game units from one location in the game map to another. The units can be workers gathering resources, workers going to build some structures or war units moving to attack enemy objects. Units have to move through the game map together with other game objects that are potentially blocking the way.

Because RTS games are highly dynamic environments, finding a static path from the objects actual position to a new position usually fails because the static path gets blocked by other objects in the game. RTS games need special methods for path finding that are able to respond to fast envirotnemt changes.

It is also useful to move units together in groups to make them more powerful to confront enemy units.

2.2.2 Resource Gathering and Management

In some RTS games players have to collect resources to achieve their goals. In case of single player resource gathering scenarios, players have to explore the map for new resource patches and collect as many resources as possible in a given time.

In more complex RTS game scenarios, players need resources for constructing buildings and producing new units. A strategy of a player in such scenario usually contains two stages. In the first stage, the player needs workers to start gathering resources and to increase resource income. When the income is stabilized, another workers start constructing buildings and offensive structures. Buildings are needed to perform technology

research and for producing war units.

When all needed structures are built and resource production is stabilized, the strategy moves into its second stage, where most of the resources are spent for training war units that perform attacks at the enemy units. Task of a resource management is to balance resource income and outcome, so that there still remains some reserve that can be used in case of serious loss of buildings that are needed for resource or unit production.

Some RTS games include more types of resources that can be situated on distinct places within the game map. Cost of units and buildings in such games is a combination of several resource types. In this case, allocation of workers gathering resources should be balanced to ensure all needed resources are available.

2.2.3 Map Exploration

Real-time strategy games that are partially observable do not show players all information about the state of the game, they send the players only information about parts of the game map that are visible to their own objects. In such games, players need to explore the map so that they are able to locate new resource patches, identify parts of the map that can be used for unit movement and locate units and buildings of the opponent.

The map exploration task requires exploring units to coordinate their movements across the game map in order to explore as much space as possible. This task should also decide how many exploring units are needed, more units means faster exploration, but cost higher amount of resources.

2.2.4 Team Collaboration

In some RTS war games there can be several players that are playing in a team together agains other opponents. Players might be executed on different machines, so they might have a limited channel capacity for exchanging information abount their strategies.

Resource gathering and building constructions usually develop independently with no possibility to cooperate. However, players should negotiate their strategies when performing attacks so that they can join their forces together to produce more intensive

attack. In case of attack of an opponent a player can go to help another player that is being attacked.

Some games also provide functionality to send resources from one player to another. This is a way how one player can help economics of the other players. It also makes possibility to split players tasks, one can gather resources and supply the other that only trains war units and performs attacks.

2.3 Deliberative Reasoning about Strategies

There are many possible actions in RTS games that players may perform, including unit movement, building structures, gathering resources, training units or exploring the map. All these actions can be considered as a part of players strategy, however, including all of these actions in the deliberative reasoning process might lead into incomputability of such process in a time given for the computation, which is very short in real-time games. Other possibility is to control structure building, unit training and resource gathering by separate processes and to perform deliberative reasoning only about war units that perform attacks.

In case of bigger count of units in the game, reasoning about single units should get very complex, it might be easier to group units into unit groups and to reason only about whole groups of units. Resulting strategy of the reasoning process should be destinations of the unit groups and possibly also paths they should move along. The process should also output an estimation of the opponents strategy.

2.4 Commercial and Academic AI Players Differences

Both commercial game developers and academic researchers develop artificial intelligence players for RTS games, however, they have different motivation. Commercial developers try to make the best AI to entertain the player, as it is described in the Section 2.4.1. In the other hand, academic researchers try to develop players that incorporate reusable methods that can be deployed in other projects solving some kind of real world problems.

2.4.1 Commercial AI Players for RTS Games

Main task of commercial AI players is to entertain a human palyer. For some of the commercial games, it is possible to develop custom scripts defining a behavior of a computer AI player, however, for instance in "Age of Empires II", scripts define general player behavior like how many buildings of particular type to build, when to train workers or war units, which resources to gather or when to attack, but they have no control over positions of buildings built or over individual war unit movements. These behaviors are built inside the game engine. Custom AI scripts also allow cheating by increasing player resources.

Connecting a custom player program to the commercial RTS game server is often impossible, games companies try to protect their intellectual property by keeping their communication protocols as secrets.

Commercial RTS game AI players are built mainly for user entertainment, the developers do not care if a player gets information it should not see or gets extra resources just to stay competitive to human players. These players do not seem to be able to solve any real world problems, where such extra information or resources are never available.

2.4.2 Academic AI RTS games Players

Academic researchers have much different goals in AI player development than video games companies. The researchers do not try to entertain human players or to impress them by unexpected strategic behavior, but to develop general, well defined methods reusable across distinct domains of problems. Such domains may include mobile robot navigation in highly dynamic environments, collaborative area exploration, area overseeing, resource management or reasoning about strategies of autonomous war units.

Enviroments used for scientific AI development are made to support no hacks to violate game rules, players in the game receive only information visible to their own objects and the players have to deal with uncertainty and stochasticity of the environment.

Chapter 3

Techniques Used in RTS Games

Developing a RTS game player includes many tasks that have to be solved. These tasks can be divided to single unit actions, called low-level tasks[10], and actions performed by groups of units, called high-level tasks.

This chapter starts with a description of methods for solving low-level tasks. Binary images and binary mathematical morphology methods that can be used for computation of potential field generated by terrain are described in the Section 3.1. The Section 3.2 contains a description of methods for searching shortest paths in weighted graphs. The Section 3.3 explains methods suitable for controlling simultaneous movement of more units at a time. Finally, in the Section 3.4 there is a description of clustering methods that can be used for merging units into groups according to their positions in a game map.

This chapter follows with descriptions of methods for solving high level tasks, rule based systems are described in the Section 3.5, solving games by searching is described in the Section 3.6.

3.1 Binary mathematical morphology methods

Mathematical morphology is a discipline studiyng analysis and processing of geometrical structures, binary mathematical morphology studies structures in binary images.

Mathematical morphology methods can be used in RTS games in two ways, it simplifies computation of potential fields generated by terrain and it can compute distances to nearest cliffs that are useful for path finding.

This section describes only few basic terms and methods of mathematical morphology that seem to be useful in RTS games. This section starts with a description of a binary image and other terminology related to mathematical morphology in the Section 3.1.1. It follows with a description of two morphologival transformations, erosion and dilatation, in the Section 3.1.2. Lastly it describes a fast distance transformation algorithm in the Section 3.1.3.

An introduction to mathematical morphology can be found in [14]. Very detailed explanation of mathematical morphology methods is in a monograph by J. Serra [27]. Fast algorithm for distance transformation is described by Rozenfeld and Pfaltz in their article [25].

3.1.1 Basic terminology in mathematical morphology

An image can be represented in computers using various shapes of grids, some of them are shown in Fig. 3.1. A heaxagonal grid has an advantage that all neighbors of a selected point have same distance to the point, but a representation of such grid in computers is more difficult. For a square grid there exist at least two definitions of neighborhood, a 4-neighborhood containing only points sharing their sides with a selected pixel, and an 8-neighborhood neighboring with a pixel with sides or corners. All pixels in a 8-neighborhood do not have same distances to a selected pixel.

Positions of certain structures in RTS games, for example cliffs, can be represented as structures in a binary image. For path finding purposes, it is reasonable to preprocess such binary images in order to control distance of paths from cliffs using a distance transformation of the binary image or using dilatation with a structuring element.

In RTS games, a game map is often represented as a grid of map tiles of a square shape, a square grid image representation is assumed. For easier computation of a distance transformation, a 4-neighborhood is used.

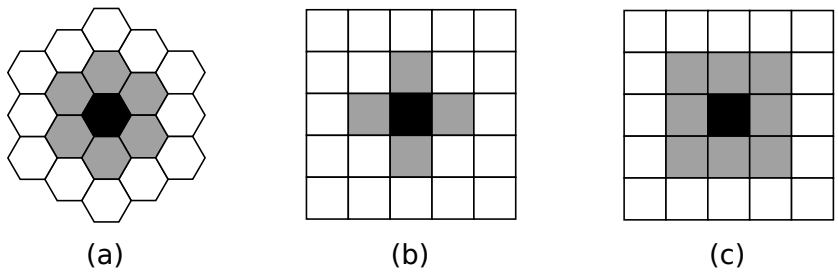


Figure 3.1: Examples of binary image representations and point neighborhood definitions, a selected point is shown in black, its neighbors are colored in gray: (a) hexagonal grid and point neighborhood (b) square grid with 4-neighborhood (c) square grid with 8-neighborhood

A binary image is defined by a function

$$f : \mathbb{Z}^2 \to \{0, 1\} \tag{3.1}$$

For usage in computers, domain of the function f is limited to vectors that lie in a finite rectangle. Every binary image defined by a function f can be also described by a set of points in which the function f has a value equal to 1

$$X_f = \left\{p \in \mathbb{Z}^2 : f(p) = 1\right\} \tag{3.2}$$

A complement to set X is defined as

$$X_f^c = \left\{p \in \mathbb{Z}^2 : f(p) = 0\right\} = \left\{p \in \mathbb{Z}^2 : p \notin X_f\right\} \tag{3.3}$$

A morphological transformation $\Psi(X, B)$ is a relation between an image X and a set B called structuring element. A structuring element is typically smaller than an image X. Positions of points in a structuring element are given according to a point called a representative point. Examples of various structuring elements are shown in the Fig. 3.2.

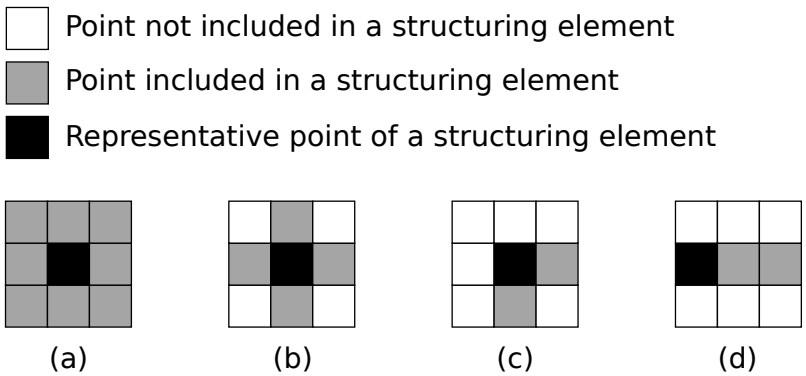


Figure 3.2: Examples of various structuring elements. In these cases, I assume that representative points are also included in structuring elements.

3.1.2 Erosion and dilatation

Erosion is defined as

$$X \ominus B = \left\{ p \in \mathbb{Z}^2 : \forall b \in B : p + b \in X \right\}, \tag{3.4}$$

where X is a binary image, B is a structuring element.

Dilatation $\oplus$ is a morphological transformation defined as

$$X \oplus B = \left\{ p \in \mathbb{Z}^2 : p = x + b, x \in X, b \in B \right\}, \tag{3.5}$$

where X is a binary image, B is a structuring element.

Examples of erosion and dilatation transformations are shown in the Fig. 3.3. Dilatation makes structures thicker, erosion makes structures thinner than before transformation.

3.1.3 Distance transformation

In mathematical morphology, a distance is defined as follows[25]: let P and Q be any two points in a binary image. A distance from P to Q is the smallest positive integer such that there exists a sequence of distinct points $(P_0, P_1, \ldots, P_n), P_0 = P, P_n = Q$, with P_i a neighbor of P_{i-1}, $1 \leq i \leq n$. Note that a distance of two points depends on selection of

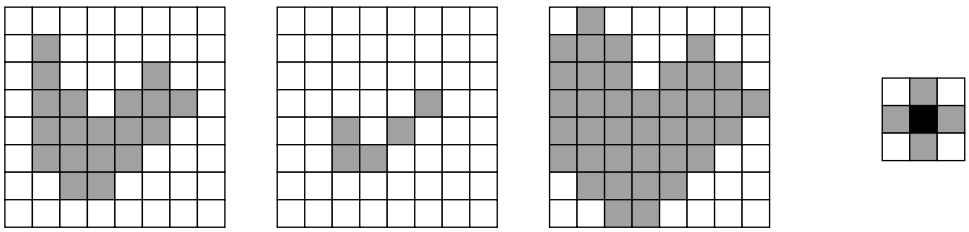


Figure 3.3: Examples of an erosion (b) and a dilatation (c) of an input image (a) with a structuring element (d).

neighborhood type, a 4-neighborhood is assumed in this case.

Given a binary image representation X_f, it is possible to compute a distance from the set X_f for any point in $(i,j) \in \mathbb{Z}^2, 1 \leq i \leq m, 1 \leq j \leq n$ using a two pass algorithm[25]. Let

$$a_{i,j} = \begin{cases} 0 & \text{if}(i,j) \in X_f \\ 1 & \text{otherwise} \end{cases} \tag{3.6}$$

Distance from the set X_f is computed by two functions f_1 and f_2:

$$\begin{aligned} f_1(a_{i,j}) &= \begin{cases} 0 & \text{if } a_{i,j} = 0, \\ \min(a_{i-1,j}+1, a_{i,j-1}+1) & \text{if } (i,j) \neq (1,1) \text{ and } a_{i,j} = 1, \\ m+n & \text{if } (i,j) = (1,1) \text{ and } a_{1,1} = 1, \end{cases} \\ f_2(a_{i,j}) &= \min(a_{i,j}, ai+1, j+1, a_{i,j+1}+1). \end{aligned} \tag{3.7}$$

Result of the algorithm given by functions f_1 and f_2 is in the Fig. 3.4.

6	5	4	5	4	3	4	5
5	4	3	4	3	2	3	4
4	3	2	3	2	1	2	3
3	2	1	2	1	0	1	2
2	1	0	1	0	1	2	3
2	1	0	0	1	2	3	4
3	2	1	1	2	3	4	5
4	3	2	2	3	4	5	6

Figure 3.4: Result of a distance transformation

3.2 Informed graph search methods

In RTS games, it is very often that units have to move from one location of the game map to another. It is useful that units move on a shortest possible path so they save time of the movement. There exist many algorithms for seaching shortest paths in weighted graphs, called informed graph search methods. If we find a graph representation of the game map, we can use such algorithms for finding routes within the game map.

This section describes two methods for finding shortest paths in weighted directed graphs with non negative edge weights. The Dijkstra's algorithm is described in the Section 3.2.1, and the A* algorithm that expands nodes in better order is explained in the Section 3.2.2. Both algorithms are explained in case of directed graphs, but if we need to search in undirected graphs, we can easily transform them to directed graphs by ensuring that for every edge there exists and edge of the opposite direction.

Definitions of weighted directed and undirected graphs and other graph-related terminology is explained for instance in [18], another explanation can be found in [17]. Dijkstra's algorithm is proposed in his original paper [11], the A* algorithm and proof of its optimality for admissible heuristics is explained in [22].

3.2.1 Dijkstra's algorithm

Dijkstra's algorithm is an algorithm that finds shortest paths from a given initial node to all other nodes in a weighted directed graph with non negative weights. It can be modified to stop when a certain destination node is reached.

A pseudo code of the Dijkstra's agorithm is shown in the Fig. 3.5. The algorithm is given a directed weighted graph and an initial node to find paths from. At first it fills the set of unvisited nodes U with all nodes of the input graph. Graph nodes have two properties, a distance that is the shortest known distance from the initial node and a reference to a predecessor, a node that lies on the shortest known path from the initial node. Initially, all nodes distances are set to infinity and predecesor node references to empty values, because distances and predecessors are not known before the search. Distance of the initial node is set to zero. Each node has stored references to its direct successors to quickly enumerate them.

```
function DIJKSTRA_SEARCH(initial_node, graph)
U ← graph
for each node in U
    node.distance ← ∞
    node.predecessor ← ∅
initial_node.distance ← 0
while not EMPTY(U)
    node ← REMOVE_BEST(U, node => node.distance)
    for each successor in node.successors
        new_distance ← node.distance + WEIGHT(node , successor)
        if new_distance < successor.distance
            successor.distance ← new_distance
            successor.predecessor ← node
```

Figure 3.5: Dijkstra's search algorithm. At the end of the algorithm, all nodes reachable from the *initial_node* have computed distances and assigned references to nodes that lie on the shortest path. The function EMPTY returns true if a set is empty, the REMOVE_BEST removes a node from a set that has the smallest value of the given labmda function. The function WEIGHT returns a weight of a graph edge, that means a distance between two neighboring graph nodes.

After the initialization step, the algorithm repeatedly takes graph nodes one by one preffering those with smallest known distance from the starting node. In each algorithm step, a node with the smallest distance is removed from the set U, distances for its successors are computed and if those distances are smaller than previously known distances, values of the distances of the nodes are updated to the new values.

When the algorithm stops, all nodes for those there exists a path from the initial node have computed distance from the initial node and reference to a next node lying in the shortest path from the initial node. If there is no such existing path, a node has initially set distance to infinity and reference to its predecessor has no value set. The shortest path from the initial node can be easily reconstructed by traversing recursively to node predecessor until we end up in the initial node.

```
function A*(initial_node, graph) returns a solution, or failure
closed ← ∅
open ← INSERT(initial_node)
for each node in graph
    node.predecessor ← ∅
initial_node.distance ← 0
loop do
    if EMPTY(open) then return failure.
    node ← REMOVE_BEST(open, node => node.distance + node.heuristics)
    if GOAL(node) then return SOLUTION(node)
    for each successor in node.successors
        new_distance ← node.distance + WEIGHT(node, successor)
        if (successor in open) or (successor in closed)
            if(new_distance < successor.distance)
                successor.distance = new_distance
                successor.predecessor ← node
        else
            successor.distance = new_distance
            successor.predecessor ← node
            open ← INSERT(successor)
```

Figure 3.6: A* search algorithm using the *open* and *closed* list. The REMOVE_BEST function now selects a node with smallest value $f(n) = g(n) + h(n)$.

3.2.2 The A* algorithm

The Dijkstra's algorithm processes graph nodes closest to the initial node first. When searching for the closest path between two nodes, the Dijkstra's algorithm is not so efficient because it has no information how close the nodes might be to the goal. In domains where is an estimation of such information available is often used a search method called A*.

Pseudocode of the A* algorithm is shown in the Fig. 3.6. This version of the algorithm uses two lists, a list called open containing nodes that are going to be expanded, and a list called closed containing already expanded nodes. During the initialization, closed list is left empty and open list is inserted only an initial node.

This algorithm takes nodes one by one from the open list, but it does not prefer

nodes which have the smallest distance from the initial node, but nodes with the smallest distance estimation to the goal. An evaluation function has the form

$$f(n) = g(n) + h(n), \tag{3.8}$$

where $g(n)$ is the smallest known distance of a node from the initial node, $h(n)$ is a heuristic function estimating a distance from the node n to the goal. The algorithm finds the shortest path for any heuristic h that never overestimates the true distance to the goal. Such heuristics are called asmissible.

After removing a node from the open list, the algorithm enumerates its expandants. For every expandant, its new distance from the initial node is computed. If it is already contained in the open or in the closed list, its distance and reference to its predecessor are updated only if a newly computed distance is smaller than previous node value. If it does not belong to the open or the closed list, its distance is set to the computed value.

The A* method usually expands much less nodes than Dijkstra's algorithm to find a goal, because the heuristic function helps the algorithm to expand nodes close to the optimal path, however, in some cases both algorithms may end up with the same number of expanded nodes.

3.3 Cooperative Pathfinding Algorithms

Graph search methods mentioned in the Section 3.2 are suitable for searching for shortest paths in graphs, but they are not able to find collision-free routes for several objects moving simultaneously. In real-time strategy games, such situation with more objects moving at a time is very common. This chapter describes two methods that can be used for avoiding collisions of such objects.

The first method, a space-time reservation table, is described in the Section 3.3.1. This method is suitable for discrete environments and it uses classical search algorithms like A* in a modified structure that represents not only spatial coordinates, but also stores information about movement of multiple objects in time. In the Section 3.3.2, an idea of using potential fields is described. This method is very suitable for higly dynamic environments, but deals with problems like stucking in local extremas.

The space-time reservation table is described for instance in [30]. Potential fields for usage in mobile robotics are described for instance in [31], an implementation for usage in real-time strategy games can be found in [12].

3.3.1 Space-Time Reservation Table

Search algorithms like Dijkstra's algorithm and A* are able to find the shortest path between two points in a graph that represents possible routes in spatial coordinates. In RTS games, we need to find such routes for multiple simultaneously moving objects so that any two objects never get to the same position at the same time, hovewer, there is no problem if they occupy the same location at a different time frame. A possible solution is to extend a two dimensional graph with the third coordinate that represents the time.

The simplest method for searching paths in both spatial and time coordinates is called a space-time reservation table. It represents the coordinates as 3-dimensional matrix of the size $X \times Y \times T$. Spatial coordinates can only take discrete values between $(0, 0)$ and $(X - 1, Y - 1)$, time is limited to take values from 0 to $T - 1$. That means that this method is suitable only for disrete space and discrete time environments.

A picture of a space-time reservation table is shown in the Fig. 3.7. It shows that permanent obstacles occupy certain table cells at all possible time windows, but moving obstacles change their position as they move in the time. Two objects can also occupy the same location as long as it is in distinct time frames.

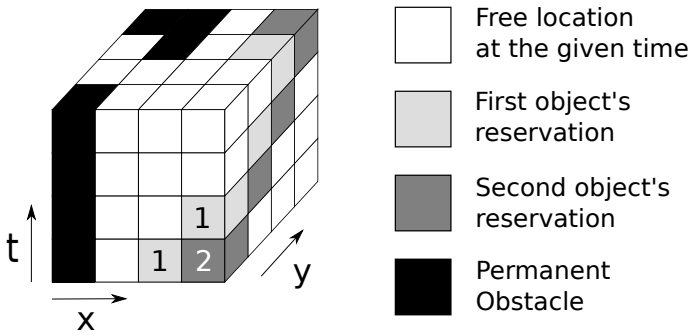


Figure 3.7: A space-time reservation table for two units moving in the space and time along some permanent obstacles

Any of the search algorithms descried in the Section 3.2 can be easily modified to

search paths in the space-time reservation table. A searching graph must be build so that arcs connect only locations in one time frame with same or neighboring locations in the next time frame. Paths for the objects are found one by one, each route is reserved in the table before searching for other one. The searching algorithm checks every node if it is suitable for expansion, nodes that are already reserved are skipped by the search algorithm.

At each movement step, a game object is allowed to increase or decrease one of the coordinates by one or it can wait on a position, however, each action increases the time by one. All possible expandants to a state with position (x, y) and time t are shown in the table 3.1.

State	Action	Next State
(x, y, t)	Move West	$(x-1, y, t+1)$
	Move North	$(x, y-1, t+1)$
	Move East	$(x+1, y, t+1)$
	Move South	$(x, y+1, t+1)$
	Wait	$(x, y, t+1)$

Table 3.1: All possible expandants of a state given by spatial coordinates and a time coordinate in a space-time reservation table

This method is able to find collision-free routes for more objects moving at a time, but has several limitations. Minimum and maximum values of the spatial coordinates and the time are limited by the size of the table, but it could be possibly dynamically extended when needed. Its use is limited to discrete space and discrete time environments, which makes this method very hardly adaptable to continous space and time environments, both space and time have to be discretized and search methods have to be modified to reserve more that one table cell in a time frame, because an object can be situated between two table cells. Other problem is its inability to react to other moving objects that are out of a players control, so there is still a risk of collision and in such case at least a path of the colliding object must be re-planned. Because the space-time reservation table is three-dimensional, number of its cells can grow to very high values that may make the search computationaly expensive.

3.3.2 Potential Fields

Artificial potential fields hadle object movement in a very different way that searching path in a graph. The idea of using potential fields for object navigation comes from an area of mobile robotics where robots need to avoid collisions with other static or moving objects. Mobile robotics engineers got inspired by nature, where physical objects tend to move from places with higher potential to places with low potential. For instance a water flows in a direction with highest decrease of potential. Mountains serve as obstacles to a moving water because they have very high potential. This is why the water flows in valleys that are places with low potential. It moves in opposite direction to potential gradient which determines a direction with the biggest increase of the potential. In local minima of potential fields these is a zero value of potential field gradient. In such places objects cannot move to any direction and stop moving.

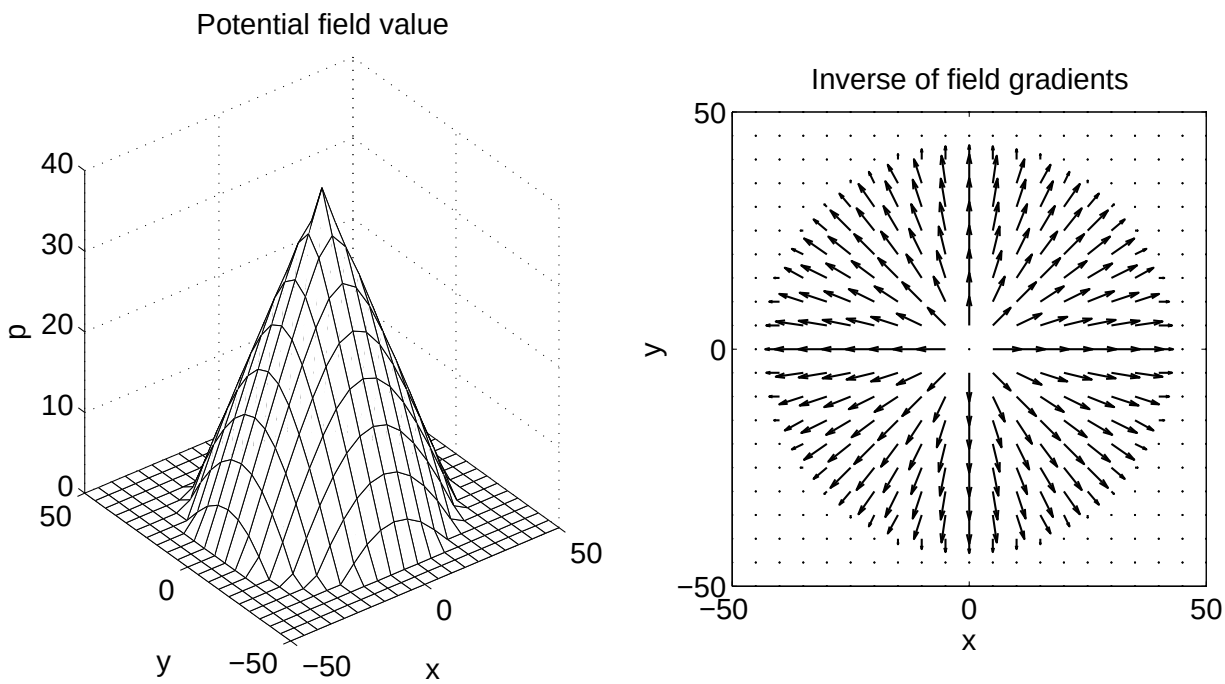


Figure 3.8: An example of a repulsive potential field: $p(r) = max(40-r, 0), r = \sqrt{x^2 + y^2}$

An idea of using potential fields for navigation of an object is very simple: a destination is assigned a field of potential that decreases with decreasing distance to the destination, called an attractive potential field, and obstacles are assigned a potential field that icreases

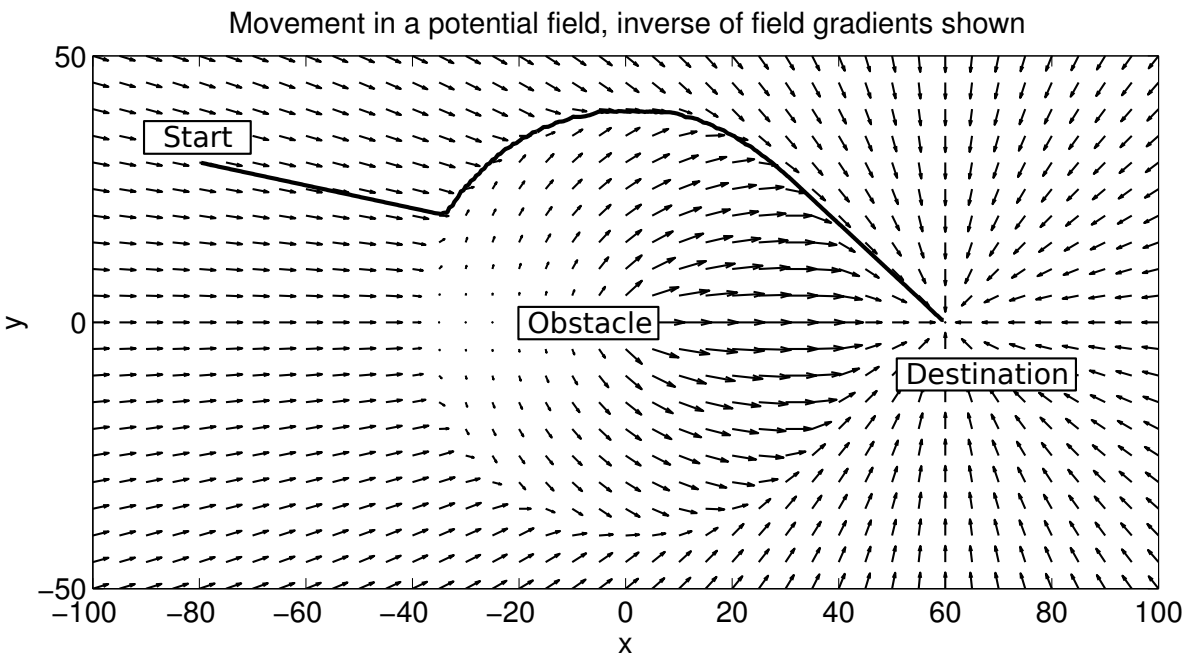


Figure 3.9: An example an object movement in a potential field. There are two sources of potential fields, a repulsive field source in the center of the environment, and an attractive field in the right part of the environment.

its value close to the obstacles, called a repulsive potential field. An example of a repulsive potential field is shown in the Fig. 3.8. In the example, an obstacle generates a repulsive field only up to certain distance from the obstacle, so that objects that move close to the obstacle are forced to move away. Opposite values to the potential field gradients show a direction of the highst decrease of the field gradient, which is the direction of an object's movement. In environments with more than one potential field sources, resulting potential field is computed as a sum of all fields in the environment.

In theory, potential fields are considered to be defined in a continous space and the gradient is computed as

$$\vec{\nabla} p = \left(\frac{\partial p}{\partial x}, \frac{\partial p}{\partial y} \right) \tag{3.9}$$

This gradient can be approximated by a difference

$$\mathrm{d}p_k(x, y) = \left[p(x + k, y) - p(x, y), p(x, y + k) - p(x, y) \right], \tag{3.10}$$

where x, y are discrete coordinates in the potential field and k is a parameter that specifies a distance between a location of gradient computation and location of value comparison. The parameter k influences the resulting value of the difference, but for object movement purposes, potential fields are used only to determine movement direction and not the movement speed, so the size of the difference is not relevant.

A simple scenario with an object moving from a starting position to a destination avoiding an obstacle is shown in the Fig. 3.9. This scenario includes two potential fields, a repulsive potential field of an obstacle and an attractive field of the destination. The object starts moving from the starting position, in every position in the potential field it computes a gradient and moves in the direction opposite to the gradient value. It moves until it ends in a local minimum at the posititon of the destination.

Another example is illustrated in the Fig. 3.10. In this case, there is one attractive field at a destination location and two obstacles generating repulsive fields. In this configuration, an object moves from a starting position towards the destination, but ends up in a local extremum that occured as a result of suming all three potential fields. This is a common problem when using potential fields, this method works very well for single obstacle avoidance, but local extremas are traps for moving objects.

There are solutions to avoid ending up a local extremas. For instance in [12] they use potential fields to model repulsive fields for both war units and map terrain. It is very common that in narrow passages and bays defined by the terrain in the game map there are local extrema. In this article, the authors use two algorithms to search for locations that lead to ending in a local minima in the bays and mark those locations as blocked, so objects do not move to those locations. They also clear narrow passages by setting potential field values in such passages to zero to avoid local extrema. Another approach would be to use both potential fields and pathfinding to find routes avoiding places such as bays with local minimas.

Potential fields are very effective, because there is no planning or searching overhead and value of their gradient can be computed any time an object needs to specify direction of the movement. Approximating the gradient by a difference causes that for each location there are only three potential field value evaluations needed to compute the difference, so this method is very fast even for high number of moving objects in an environment. Potential fields are very suitable for defining an reactive behavior of the object, they can

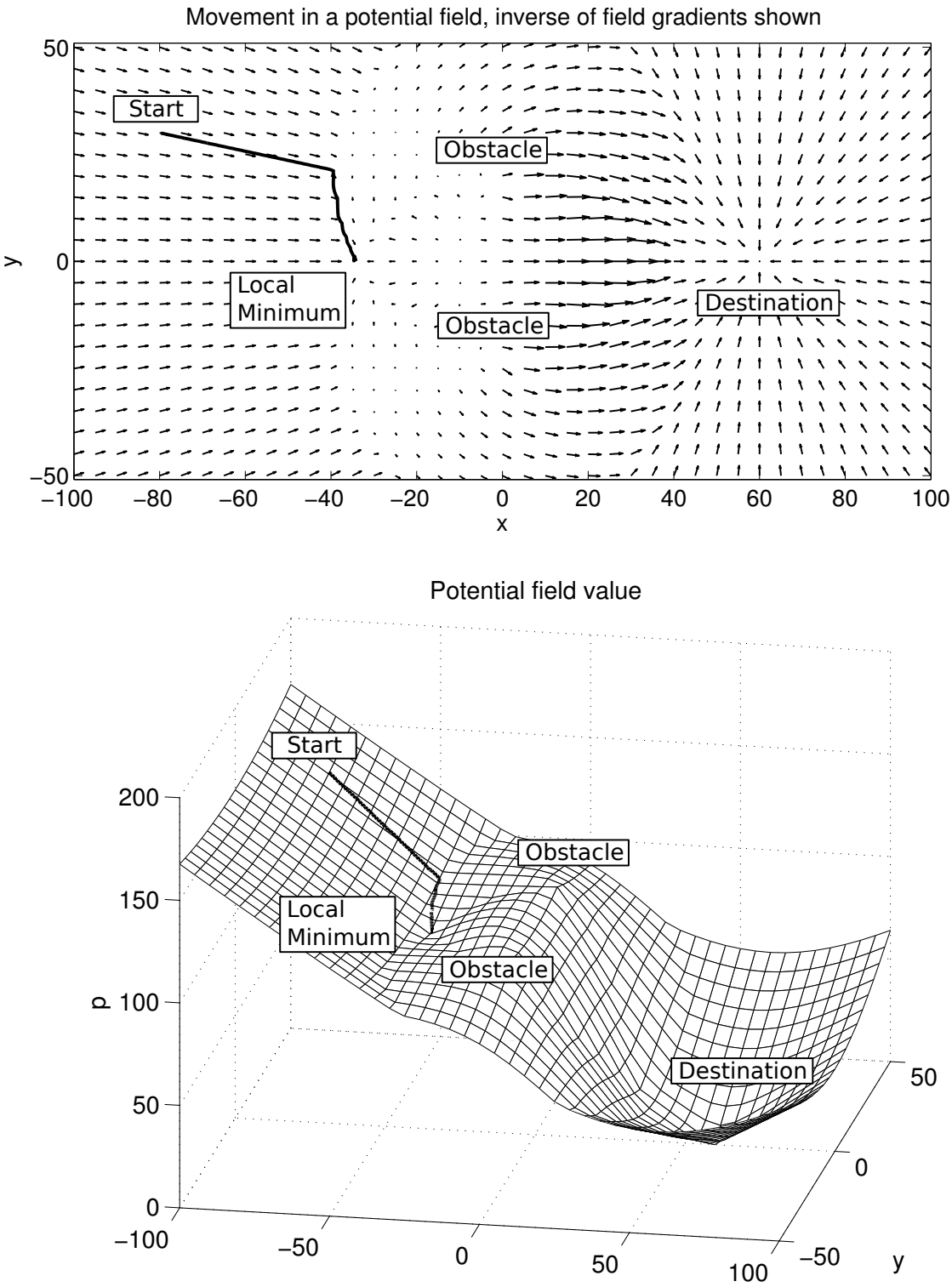


Figure 3.10: An example an object movement in a potential field. There is one source of an attractive field at the destination location and two sources of repulsive fields at positions of obstacles. This configuration causes a moving object end up in a local minima before it reaches the destination.

react to any static or moving obstacles in a real time. However, there are issues like local extrema that have to be solved in order to build a reliable system to control game object navigation. Navigation of objects using potential fields is also very sensitive to selection of functions defining the potential field. Functions can be partially linear, quadratic, exponential, or of any other shape. Idea of potential fields navigation method is very simple, but searching for suitable potential functions can be very challenging.

3.4 Clustering methods

At the beginning of a strategy game, a player usually starts with some units situated in a game map. It is an advantage to join units into groups because it simplifies reasoning and makes unit sronger. Clustering is a method that assigns units into groups so that units within a cluster have similar properties. In RTS games, units are assigned into groups according to their position in a game map.

In this section, two methods of clusterig are described. An iterative algorithm k-means is described in the Section 3.4.1, hierarchical clustering algorithm is explained in the Section 3.4.2.

Many clustering algorithms and related terminology is explained in [15]. Very detailed explanation of the k-means algorithm is in [21].

An input to clustering algorithms is a set of observations $\chi = (\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_m)$. A disjoint partition of the set χ into k sets is a set $\Omega = (C_1, \ldots, C_k)$ so that

$$(\forall i, j \in \mathbb{N})\,(i \leqq k)\,(j \leqq k)\,(i \neq j) : C_i \neq \varnothing, C_i \cap C_j = \varnothing, \bigcup_{i=1\ldots k} C_i = \chi. \tag{3.11}$$

3.4.1 K-means algorithm

The k-means algorithm searches for a disjoint partition Ω^* over all possible partitions Ω of a set of observations χ into k sets so that it minimizes a homogenity W

$$\Omega^* = \operatorname*{argmin}_{\Omega} W, \qquad W = \sum_{i=1}^{k} \sum_{x_j \in C_i} d(\vec{x}_j, \vec{\mu}_i), \tag{3.12}$$

where

$$\vec{\mu}_i = \frac{1}{\|C_i\|} \sum_{\vec{x}_j \in C_i} \vec{x}_j \qquad d(\vec{x}, \vec{y}) = \|\vec{x} - \vec{y}\|^2 \tag{3.13}$$

K-means is an iterative algorithm consisting of the following steps:

1. initialize positions of k clusters μ_i randomly (for instance by selecting k input observations)
2. each observation x_i is assigned a cluster μ_j so that $j = \operatorname*{argmin}_{l=1...k} d(x_i, \mu_l)$.
3. means of clusters are recomputed so that $\mu_i = \frac{1}{\|C_i\|} \sum_{x_j \in C_i} x_j$
4. steps 2 and 3 are repeated until positions of means μ_i do not change

An example of results of the k-means algorithm is shown in the Fig. 3.11. Input data points were generated so they are similar to initial positions of units in the ORTS enviroment. The figure shows results of the k-means algorithm for number of clusters $k = 3$ and $k = 5$.

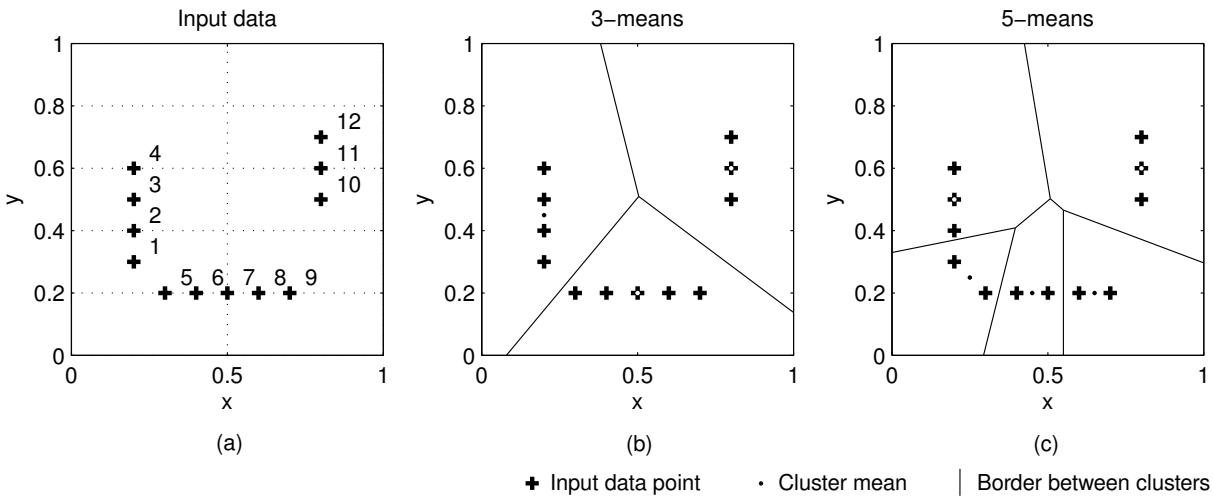


Figure 3.11: Illustration of results of the k-means algorithm: (a) input data points indexed from 1 to 12 (b) result of a 3-means algorithm (c) result of a 5-means algorithm

K-means is a stochastic algorithm due to the random initialization of the cluster means. It is a greedy algorithm, so it is not complete, it may not find an optimal solution to the clustering problem. However, it is easy to implement and very fast, so it can be

used to cluster high number of input points. A disadvantage of this algorithm is that the number of means k must be known prior to the run of the algorithm.

3.4.2 Hierarchical clustering algorithms

Hierarchical clustering algorithms are another approach to clustering. These algorithms build a binary tree called dendrogram, that describes whole clustering process starting from individual points and ending at one cluster containing all data points. There are two types of hierarchical clustering: division algorithms divide bigger clusters into smaller ones. These algorithms are faster when it is not needed to generate whole dendrogram, so they stop when they reach a defined size of cluster. A disadvantage is that division algorithms are more difficult for implementation. In contrast, agglomerative algorithms join smaller clusters into bigger ones. The agglomerative algorithms are easy to implement and their behavior can be parametrized by a distance function. An advantage of hierarchical clustering algorithms is that there is no need to specify number of clusters.

An agglomerative clustering algorithm is defined by following steps:

1. Assign each data point a cluster containing only the single data point
2. Compute distance δ between all pairs of clusters
3. Select two clusters $\operatorname*{argmin}_{i,j} \delta(C_i, C_j)$ and merge them into a single cluster
4. Repeat steps 2 and 3 until there is only a single cluster

Results of the agglomerative clustering algorithm depends on a definition of the distance function δ measuring distances between paris of clusters. A single linkage distance measures distances between two most similar points belonging to distinct clusters

$$\delta_s(C_i, C_j) = \min_{x \in C_i, y \in C_j} d(x, y). \tag{3.14}$$

A complete linkage distance function measures distance between two most distinct points belonging to two clusters

$$\delta_c(C_i, C_j) = \max_{x \in C_i, y \in C_j} d(x, y). \tag{3.15}$$

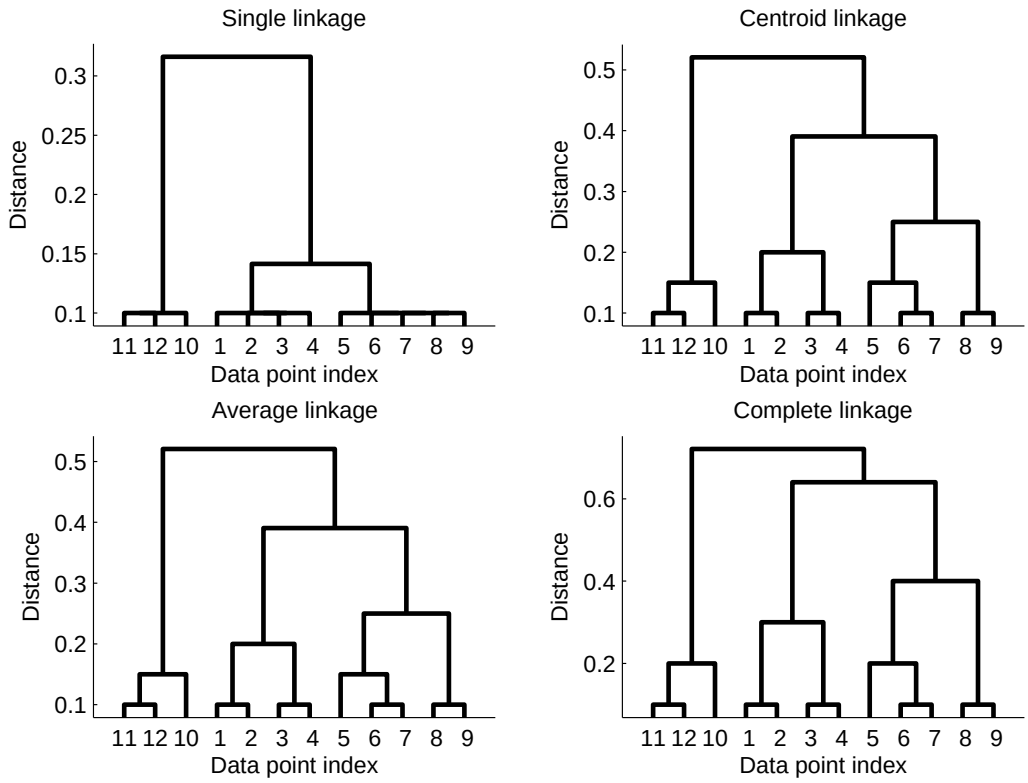


Figure 3.12: Examples of dendrograms generated using different cluster distance functions. Data points are the same as in 3.11 (a).

An average linkage function measures an average distance between points of two distinct clusters

$$\delta_a(C_i, C_j) = \frac{1}{|C_i||C_j|} \sum_{x \in C_i} \sum_{y \in C_j} d(x, y). \tag{3.16}$$

A centroid linkage between two clusters is the same as distance of cluster means

$$\delta_n(C_i, C_j) = d(\mu_i, \mu_j). \tag{3.17}$$

A distance function $d(x, y)$ can be any function satisfying following axioms

$$\begin{aligned} &d(x,y) \geq 0 \\ &d(x,y) = 0 \Leftrightarrow x = y \\ &d(x,y) = d(y,x) \\ &d(x,z) \leq d(x,y) + d(y,z) \end{aligned} \tag{3.18}$$

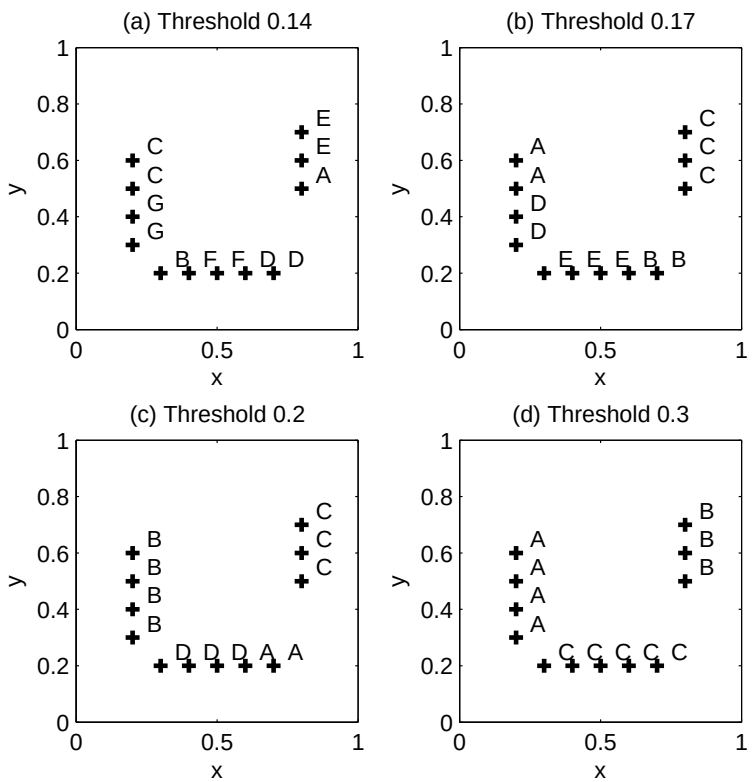


Figure 3.13: Examples of clusters determined by thresholding of a dendrogram computed by centriod linkage clustering, data points are the same as in 3.11 (a). Distinct clusters are labeled with letters A . . . G. (a) threshold 0.14 splits a set into 7 clusters, (b) threshold 0.17 creates 5 clusters, (c) threshold 0.2 creates 4 clusters, (d) threshold 0.3 splits the given set into only three clusters.

Examples of dendrograms generated by hierarchical clustering algorithm using different cluster distance functions is shown in the Fig. 3.12. The dendrograms were generated using the same data as in the Fig. 3.11 (a). Dendrograms can be used for generating clusters of any maximum distance between clusters using thresholding. The single linkage dendrogram is not suitable for thresholding because it allows to create only three possible

arrangement of clusters. Other dendrogram types are better suitable for clustering using thresholds. A centroid linkage agglomerative clustering has an advantage of much faster computation than single, complete or average linkage clustering.

Examples of clusters determined by thresholding a dendrogram are shown in the Fig. 3.13. Used data set is the same as in the Fig. 3.11 (a), the dendrogram was computed using the centroid linkage cluster distance function. Advantage of this approach is that using only one run of the algorithm, it is possible to generate clusters of different distances between their centroids using thresholding, which is very fast. If it is desired to limit maximum size of clusters, every cluster containing more that one point can be split into smaller ones.

3.5 Rule-based systems

This section contains two methods that both include set of rules, where applying these rules changes a state of a system. The first method is a finite state machine described in the Section 3.5.1. A scripting method using a subset of the Jess[13] language is described in the Section 3.5.2.

Explanation of finite automata is in [20]. An expert system language Jess in described in [13].

3.5.1 Finite-state machines

A deterministic finite-state machine is an abstract machine that can be in one of finite set of states. It receives symbols by its input and decides into which state it transits upon the input and the current state. States can have attached entry and exit actions that are executed when entering or exiting a state.

A deterministic finite-state machine is defined as a tuple

$$(\Sigma, \Gamma, S, s_0, \delta, \omega)\,, \tag{3.19}$$

where Σ is a set of input symbols, Γ is a set of output symbols, S is a set of states, $s_0 \in S$

is an initial state, δ is a state transition function $\delta : S \times \Sigma \rightarrow S$, ω is an output function. Sets Σ, Γ, S are non-empty finite sets. There are two common definitions of the output function ω, in a Mealy machine, the output function depends on both current state and input $\omega : S \times \Sigma \rightarrow \Gamma$. In a Moore machine, the output function depends only on a current state $\omega : S \rightarrow \Gamma$.

A finite state machine is very useful for modeling systems that transit between several states and the set of states does not change. A finite-state machine (FSM) is usually connected to other algorithms in an AI player via its input and state entry and exit actions. An example input and state entry actions of a FSM is shown in the table 3.2.

Input	Semantics
a	Enemy units in range
b	No enemy unit in range
c	No enemy units in game
d	Enemy base in range
e	No enemy base in range
f	No enemy bases in game

State	Entry action
A	Approach the closest enemy unit
B	Attack enemy unit
C	Approach the closest enemy base
D	Attack enemy base
E	No action

Table 3.2: An example of a FSM input with its meaning explained and a set of states and their entry action.

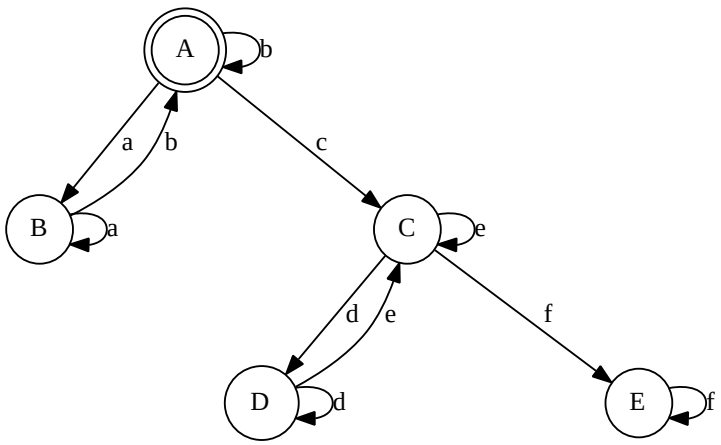


Figure 3.14: An example of a FSM definition using a graph

An example FSM implementing a simple control system for a war unit in a scenario containing enemy war units and enemy bases is shown in the Fig. 3.14. A state stransition table for this FSM is in the Fig. 3.15.

Input ╲ State	A	B	C	D	E
a	B	B			
b	A	A			
c	C				
d			D	D	
e			C	C	
f			E		E

Figure 3.15: An example of a FSM definition using a state transition table.

Finite-state machines are good models for simple behavior that works with limited number of states, because they are easy to design and implement. A disadvantage is that the machine can be only in one state at a time. To implement a system with multiple states, it is still possible to use finite-state machines, but a FSM needs to contain all possible combinations of the states of the system.

3.5.2 Declarative scripting

Declarative scripting systems, also referred as expert systems, consist of a set of variables and a set of rules. Variables define state of a system, application of rules can change the system state. Some RTS games, for example Age of Empires by Ensemble studios, use a subset of the Jess expert system language[13].

A rule in the Jess language consists of a set of preconditions and a set of actions. Rules in a declarative scripting system are applied one by one, if all preconditions of a rule hold, than all actions of the rule are applied. Actions define changes to current values of system variables. An example syntax of a rule in the Jess language is shown in the Fig. 3.16.

In RTS games, preconditions can also refer to the state of a game, for example positions of units or amount of resources. Actions can define for example attacks of war units or construction of a building.

Problems of declarative scripting approaches is that every aspect of players behavior has to be scripted. Creating scripts requires lot of expert knowledge and thus is very time consuming. Other problem is that once a player is scripted, its behavior is easily

```
(defrule
  (precondition 1)
  (precondition 2)
  ...
  (precondition n)
  =>
  (action 1)
  (action 2)
  ...
  (action m)
)
```

Figure 3.16: Syntax of a declarative scripting rule

predictible, because scripts typically do not allow learning through more game simulations.

Both finite-state machines and declarative scripting require that a suitable behavior for a player is known prior to the game is played.

3.6 Solving games by searching

Games where players alternatively decide which action to play can be represented as game trees. According to the definition in [28], a perfect-information game in extensive form can be represented as a tree in which each node represents a choice of one of players, each edge represents an action and leaves represents terminal states.

Examples of game trees is in the Fig. 3.17. In two-player turn-based games, players alternatively select one of possible actions until a terminal node is reached. In real-time strategy games, unit actions are assigned at the beginning of a game and then the game is simulated until there is another decision point or a terminal state is reached.

There is an utility function defined for each terminal state that says how good the state for an appropriate player is. In zero-sum two-player games, an utility of the first player u_1 in a state S is always a negative value of an utility of the second player u_2 in the same state $u_1(S) = -u_2(S)$. In such games, there is usually only one utility function $u(S) = u_1(S) = -u_2(S)$ used.

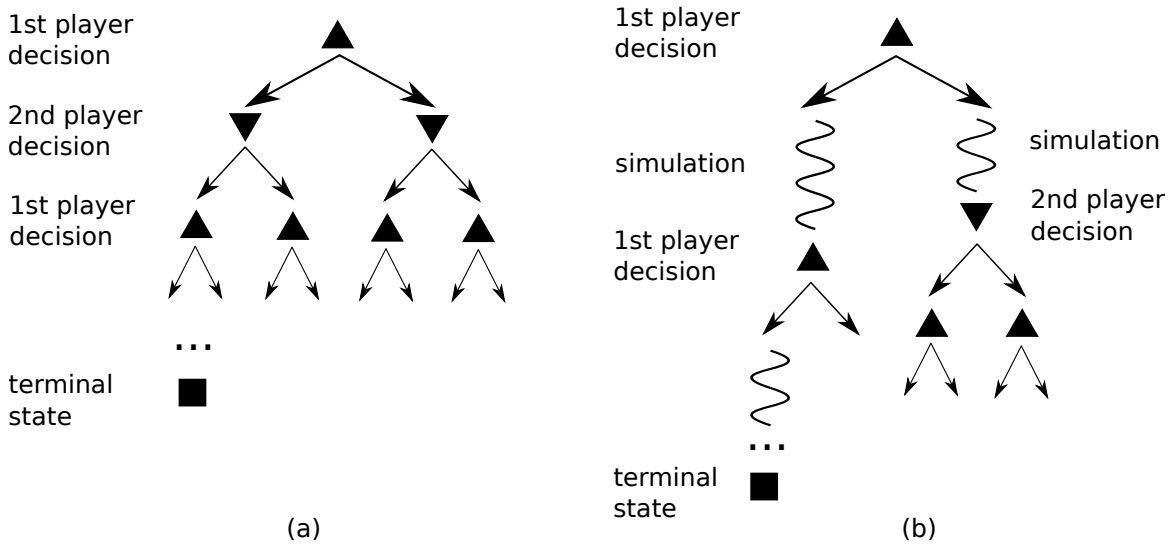


Figure 3.17: Examples of extensive-form games representations for two-player games, players are denoted as ▲ and ▼: (a) turn-based game where players alternatively decide which move to play (b) real-time strategy game tree representation with decision points where players can make decisions. If there is no decision possible, a game simulation is performed until there is another decision point or a terminal state is reached.

One of the simplest utility function is to set $u(S) = 1$ if a player has won in the state S, $u(S) = 0$ if the state is a tie and $U(S) = -1$ if the player has lost (that means an opponent must have won the game in the state S). Generally, an utility function can be any other function assignning each terminal state S a real number.

There are various methods for searching game trees in order to select actions that maximize an outcome of the game for the searching player. The baseline algorithm minimax is described in the Section 3.6.1. Monte-Carlo tree search methods are described in the Section 3.6.2.

Minimax algorithm and its extension alpha-beta prunning are explained in [26] and [28]. Monte-Carlo search methods for strategy games is explained in [8], usage of UCT in RTS games is described in [29] and [2]. Furher improvements called progressive strategies are explained in [6].

```
function MINIMAX_DECISION(state)
    for each operator in state.operators
        operator.value ← MINIMAX_VALUE_MIN(APPLY(operator, state))
    return operator from state.operators with the highest value

function MINIMAX_VALUE_MIN(state)
    if IS_TERMINAL(state) then return UTILITY(state)
    return the lowest MINIMAX_VALUE_MAX of SUCCESSORS(state)

function MINIMAX_VALUE_MAX(state)
    if IS_TERMINAL(state) then return UTILITY(state)
    return the highest MINIMAX_VALUE_MIN of SUCCESSORS(state)
```

Figure 3.18: Pseudocode of the minimax algorithm, the function APPLY(operator, state) applies an operator to a state, the function SUCCESSORS(state) returns all states that are results of the function APPLY using all applicable operators to the state.

3.6.1 Minimax algorithm

In two-player zero sum games with an utility function $u(S)$, the first player wants to maximize the utility $u(S)$, while the second player want to maximize its utility $-u(S)$, which is the same as minimizing the first players utility $u(S)$.

Input to the minimax algorithm is an initial state, actions applicable in states, terminal test function that checks if a state is a terminal state, and an utility function. The algorithm searches all possible states reachable from the initial state by applying actions of the two players and tries to maximize the utility in the decision points of the first player and minimize the utility in the decision points of the second player. A pseoudocode of the minimax algorithm is shown in the Fig. 3.18.

An example of the minimax algorithm result is in the Fig. 3.19. This picture shows a game tree of a modified tic-tac-toe game, situated in a limited board that has only 1×3 fields. Two players alternatively do their moves, the first player can put a cross into an empty field, the second player can put a circle into an empty field. A player that has two own marks neighboring to each other wins the game and gets the utility 1. The other player get the utility value -1. If there are no two marks of the same type next each other, both players get the utility 0. In every decision point, the first player tries

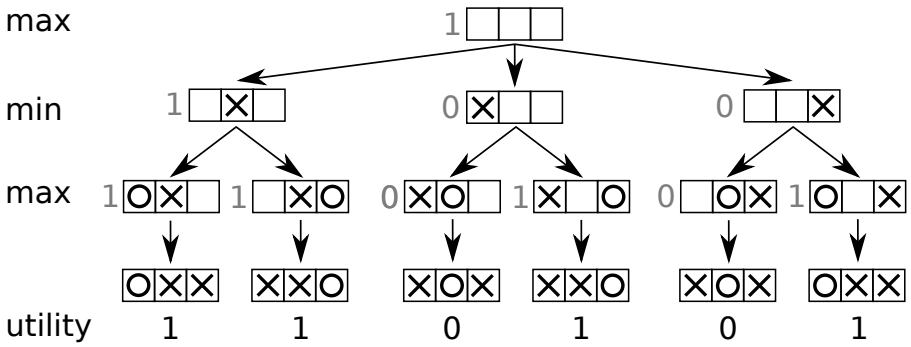


Figure 3.19: Example of a minimax searching algorithm for a modified tic-tac-toe game having only 1×3 fields. Move of the first player is depicted as crosses $\times$, moves of the second player is displayed as circles $\bigcirc$. Numbers under terminal nodes are values of an utility function of the first player, gray numbers are sub-game perfect equilibria for particular decision nodes.

to maximize the minimal value of the second players utility, the second player tries to minimize the maximal value of the first players utility. The first player assumes that the second player will try to choose an action leading to a state with the lowest utility, so the first player maximizes the minimal possible utility to minimize the risk of loosing the game.

3.6.2 Monte-Carlo tree search

Monte-Carlo tree search (MCTS) is another method that can be used for searching the game tree; in comparison to minimax, it does not search the tree completely, it only approximates utility values in tree nodes. It starts with a random exploration and as a number of explored states gets higher, the algorithm is able to estimate more promising moves and explores these moves more frequently.

The MCTS algorithm starts with one node and then it iterates four steps, selection, expansion, simulation and backpropagation as long as there is time available. An illustration of steps in MCTS is shown in the Fig. 3.20. In the selection step, is searches most promising nodes until a leaf node is found, after that it expands one or more new nodes and adds them into the graph, then it plays one game simulation until a terminal node is reached and evaluates its result. At the end of the iteration it backpropagates the result

value into nodes that have been selected in the first step.

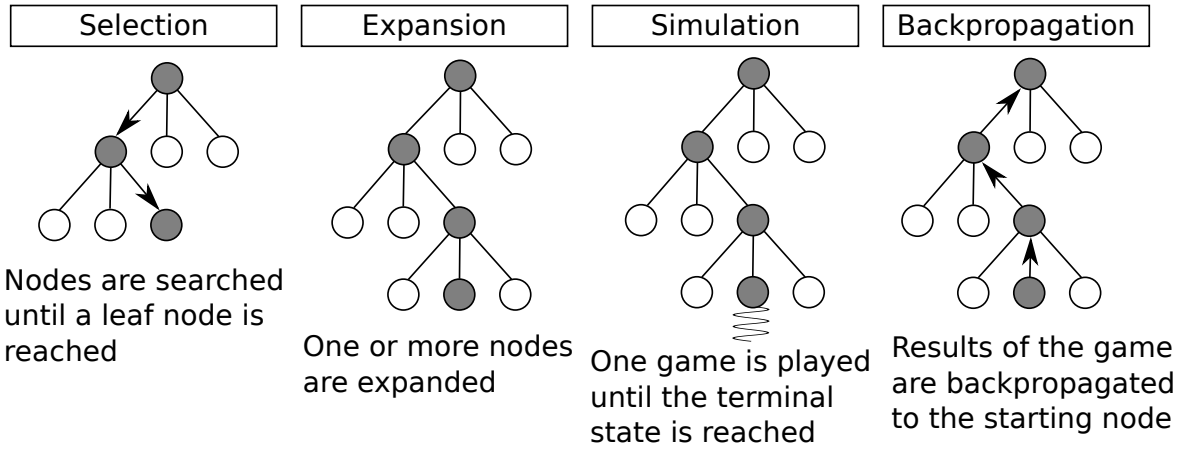


Figure 3.20: One iteration of the MCTS algorithm, scheme is inspired by [7]

The selection task can be handled in several ways, in can prefer expanding states that seem to lead to the best game results (called exploitation), or it can prefer also least promising nodes to be explored (called exploration). An UCT algorithm [19] combines both exploration and exploitation, at each node it select its direct successor with the biggest value

$$v_i + c \times \sqrt{\frac{\ln N}{n_i}}, \tag{3.20}$$

where v_i is a value estimate for a successor node, N is number of times the node was expanded, n_i is number of times a successor was expanded and c is a domain dependent constant. If there are direct successors that have not been yet expanded, one of these nodes is selected randomly.

In the backpropagation step, estimate values v_i are updated as follows

$$v_i \leftarrow v_i + \frac{1}{n_i}\left(R - v_i\right), \tag{3.21}$$

where R is a reward that describes a result of the simulated game.

In case of a big branching factor of a game tree node, a progressive unprunning[7] method can be used. This method starts with a few node successors and increases their number as the node gets visited many times. This approach ensures that most promising nodes are expanded first, but a heuristic knowledge is required.

Chapter 4

Open RTS Game Environment

Testing artificial intelligence players requires a game environment, often referred as game engine, able to run game simulation and connect AI players. Differences of two open source-game engines are discussed in the Section 4.1. An ORTS engine is described in the Section 4.2. Several game scenarios provided within ORTS are described in the Section 4.3. Details of selected strategic combat scenarios is explained in the Section 4.4.

Differences between two RTS game engines, ORTS and Stratagus are discussed in [10]. Stratagus is described in [23]. More information about ORTS can be found in [3], scenarios used in an RTS competition are described in [5] and [1].

4.1 RTS engine selection

Commercial real-time strategy games have often limited possibility to develop custom artificial-intelligence players. Video games companies try to protect their intellectual property so they keep protocols between clients as secret. Some of them, for instance Age of Empires II, include possibility to define a behavior of an AI player using scripts, but these scripts do not include possibility to control unit movements or building positions in a game map. Scripts in Age of Empires II also allow players to cheat by receiving additional resources.

As an opposite to commercial RTS games, there exist open-source engines like ORTS[3]

or Stratagus[23]. Because these engines are open source, anyone can study the source code and modify it according to his needs. A comparison of the two most used engines, ORTS and Wargus, is explained in[10].

Stratagus was built as a Warcraft II clone and later became a generic RTS game engine. It uses an open-source scripting language Lua that provides well documented interface for usage in other products, and it is very often used for scripting game AI. Unit basic behavior like attacking close enemy units and pathfinding is already implemented, but there is no way how to change this low-level behavior.

AI development using ORTS is very different, client code provides access to all low-level actions like unit movement and single attacks, but a user needs to implement pathfinding and attack control before solving his high level problems like reasoning about strategies. Other problem of ORTS is its lack of documentation, users have to study available source codes of existing players in order to learn how to program their own. ORTS is able to define custom game scenarios and types of units using its own scripting language, but a limited documentation makes this task difficult. Fortunately it is possible to use existing scenario definitions used in AIIDE RTS game competition[5][1].

Task of this work was to develop a complex AI player able both to navigate units in a highly dynamic environment and reason about player strategies, so the ORTS was chosen as a test bed for the development because it provides functionality for low-level unit control.

4.2 ORTS architecture

Most commercial RTS game implementations try to save communication resources to achieve better response during the gameplay. These game engines use a server only to negotiate game scenario and number of players, game simulation is performed on every client machine and clients share only actions taken by players. Such implementations spare lot of communication resources and allow player to play using low bandwith networks, but every client is provided information about the whole game, including information a player shoud not get. This allows players to cheat by receiving additional information.

ORTS environment uses a server-client architecture [3]. When the server application starts, it waits for all client connections and creates a game scenario. When all clients connect, it starts a game simulation that is performed in small time frames. In each frame, state of the game is sent to all clients, clients perform their reasoning and send back their actions to the server.

In this architecture, there is no possibility to violate game rules because clients are sent only information they are allowed to get. In order to limit data flow between the server and clients, a server computes only differences between two following game frames and compresses it before sending data to the clients. Difference vectors usually contain only small values, so these vectors can be easily compressed.

4.3 ORTS Competition Scenarios

ORTS provides its custom scripting language for defining custom game scenarios, but due to a poor documentation creation of custom scenarios is quite difficult. Fortunately, ORTS comes with several definitions of game scenarios that were used in RTS game competition[5]. There are also available source codes of AI players that participated a competition in 2009 on a website[1].

This chapter describes four game scenarios used in the competition in 2009: a cooperative path finding scenario in the Section 4.3.1, strategic combat scenario in the Section 4.3.2, full real-time strategy game in the Section 4.3.3 and a tactical combat scenario in the Section 4.3.4.

Example image of an ORTS map divided into map tiles is shown in the Fig. 4.1, an example of the strategic combat scenario is shown in the Fig. 4.1. Scenarios vary in game map size, appearance of distinct types of units and buildings and scenario objectives. Games can be fully observable of partially observable.

4.3.1 Cooperative Path Finding

A goal of a cooperative pathfinding scenario is to gather as much resources as possible in a given time. It is a single player game with perfect information about the game state.

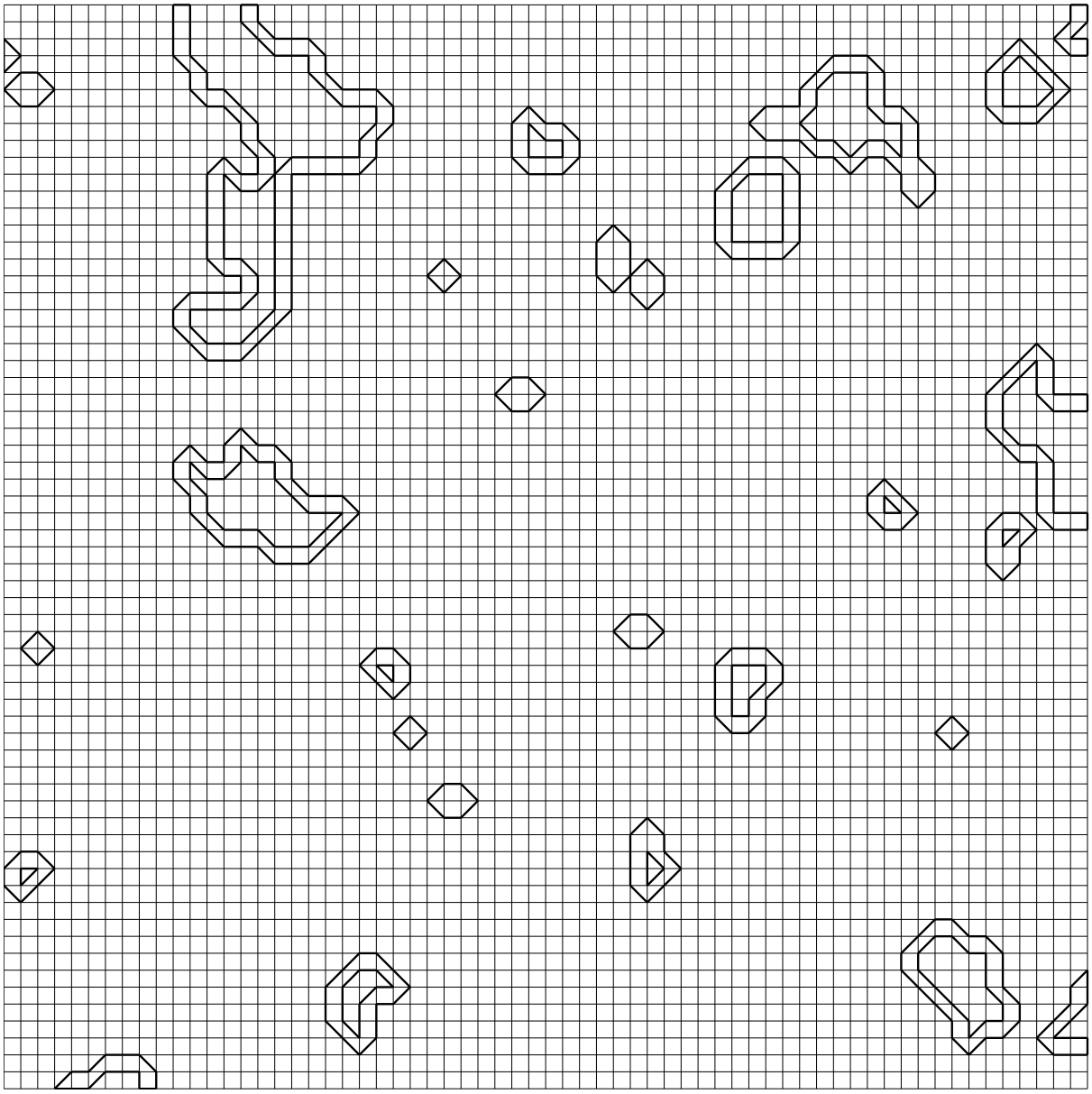


Figure 4.1: Example of a game map in strategic combat scenario. The map containns 64×64 tiles, thick lines depict cliffs that block movement of objects. Units or buildings are not shown in this picture.

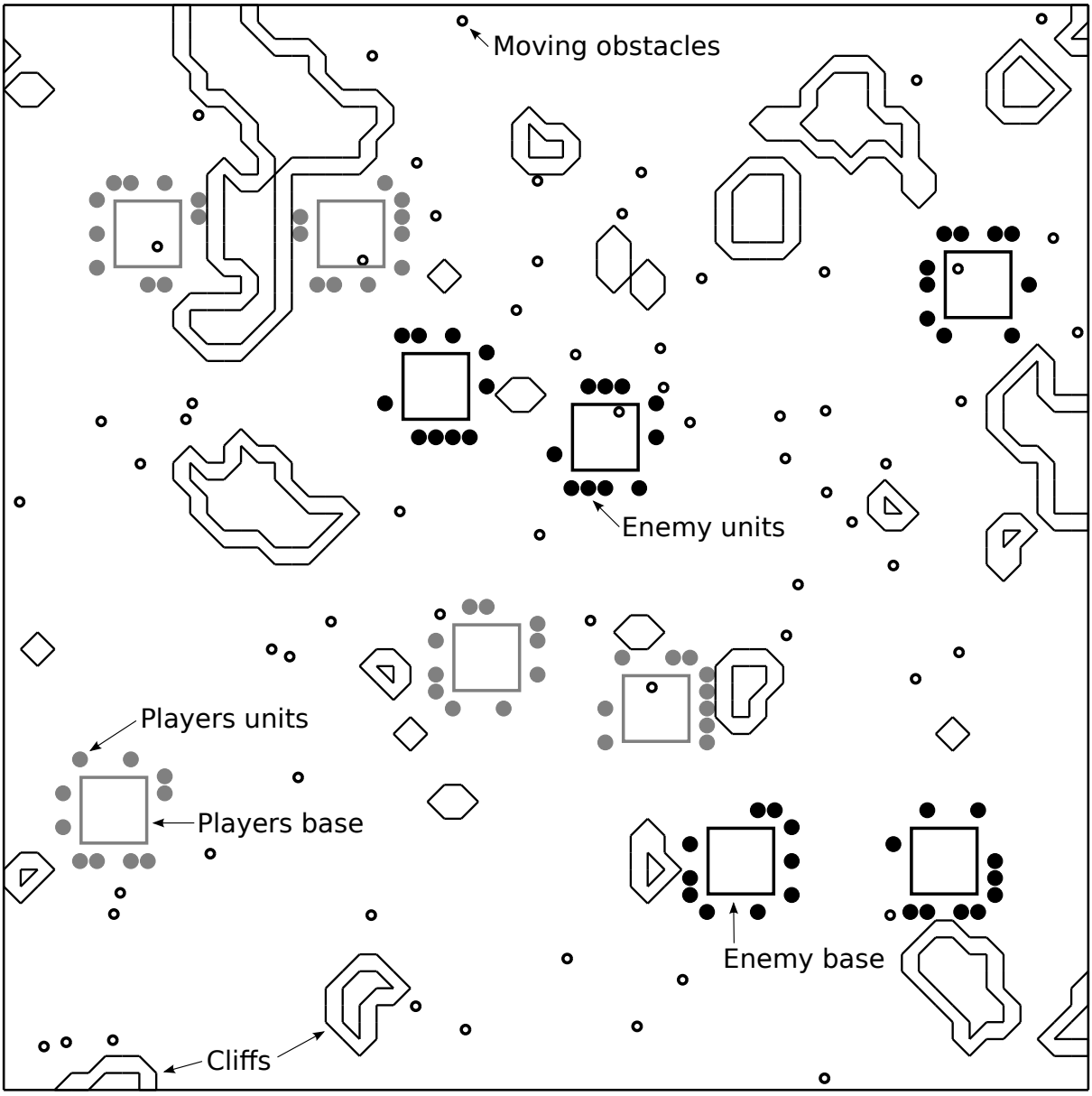


Figure 4.2: Example of a strategic combat scenario. Both players have randomly, but symetrically placed 5 bases (control centers) surrounded by several war units nearby. There are also indestructible moving obstacles performing random moves within the game map.

The scenario is situated on a game map containing 32×32 map tiles, some of those tiles contain resources that can be gathered, called resource patches. Other map tiles can contain cliffs blocking unit movements. Time limit in the competition [1] was set to 10 minutes.

A player starts with a control center building and 20 workers nearby. Workers can move within the game map and gather resources and bring them back to the control center. Number of workers gathering resources from one resource patch is limited to 4. The environment also contains moving obstacles called sheep which do just random moves and make the gameplay more dynamic.

Task of this scenario is to develop an AI player that is able to control workers so that they move between resource patches and control center and they do not collide with each other or moving obstacles. Workers must share resource patches so that there are maximum of 4 workers gathering from one resource patch. Task is to maximize amount of resources gathered in 10 minutes.

4.3.2 Strategic combat

Strategic combat is a two player war simulation game with perfect information where each player has to destroy opponent's buildings. This scenario is situated on a game map containing 64×64 tiles. Some tiles are blocked by cliffs.

Each player is given 5 control center buildings and several units near each building. Units can move within the game map and shoot at enemy units and buildings, control centers have no possibility to shoot. The scenario also includes indestructible moving obstacles. There is no possibility to obtain new units or buildings or to repair or heal units in the game.

Task of this scenario is to control units so they are able to destroy enemy buildings. The task requires players to use cooperative path finding algorithms to move units across the map containing moving obstacles and players have to decide where and when to send their units.

A player that destroys all enemy buildings is declared a winner. If there is no winner within 15 minutes, the game is declared a tie.

4.3.3 Full RTS game

Full real-time strategy game scenario is a two player game with imperfect information where players have to build needed buildings, train units and destroy buildings and units of their opponent.

Game map size is 64×64 tiles. Each player starts with a control center and 6 workers situated nearby. There is also a resource patch in sight so workers can directly start with resource production. Information about map tiles that are out of sight is not provided, players have to scout the map to find new resource patches and locate enemy units and buildings.

A player needs to gather resources in order to create more workers and war units. Workers can build control centers and two types of miliraty buildings, barracks and factories. New workers are produced in control centers, units called marines are trained in barracks and tanks are constructed in factories. Each building or unit costs resources.

This scenario requires players to be able to explore the map, gather resources, decide when and where to build buildings, produce marines and tanks and decide when to send units to fight the enemy. The goal is to destroy all opponent's buildings in 20 minutes.

4.3.4 Tactical combat

Tactical combat is a two player game with perfect information containing only units and moving obstacles, there is no terrain or buildings in a game map. The task is to destroy opponent's units within 5 minutes.

The scenario is situated on a rectangular game map having 64×48 tiles. Each player starts with 20 tanks and 50 marines randomly situated in one half of the map. Each type of unit, tank or marine, has different speed of movement and weapon range and damage.

4.4 Strategic combat in detail

Strategic combat scenario has been chosen as a testing scenario for AI player development. A map of this scenario is divided into 64×64 tiles. An example of a game map in this type of scenario is shown in the Fig. 4.1. Example of initial positions of bases and units is shown in the Fig. 4.2.

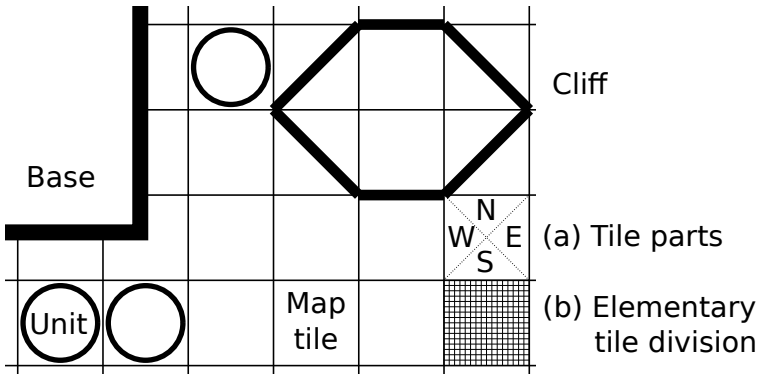


Figure 4.3: Detail of a strategic combat scenario map: (a) each tile is divided into north, east, south, and west part and each part of a tile can have different terrain (b) any object can be situated on any of the 16×16 elementary positions within the game tile. This means that positions of game objects are discrete, but the elementary unit is very small, so it is needed to treat object positions as continous.

A small detail of the map shown in the Figures 4.1 and 4.2 is shown in the Fig. 4.3. It shows that every tile is divided into four parts where each part can have different terrain. Borders of different terrain types can be not only vertical, horizontal, but also diagonal. Positions of object in the game are not aligned to map tiles, but to much smaller division referred as elementary units. Each map tile is divided into 16×16 elementary positions. Every map object, bulding or unit, has to be aligned into one of elementary positions.

Players in the strategic combat start with 5 bases and 50 tanks each. There is no possibility to produce new units or build new buildings or repair them. Bases have no possible actions at all, units can perform two distinct actions, move to a new position and attack an enemy unit within a shooting range. Both actions can be performed simultaneously. Units have limited maximal speed. When a move action is applied, a unit starts moving in the desired direction until it reaches a destination or until it collides with a terrain, another unit or a moving obstacle.

Chapter 5

AI Player Design

This chapter describes the design of an AI player for the ORTS Strategic combat scenario, which is explained in Sections 4.3.2 and 4.4. In this scenario, each of two players controls 50 units in an environment where parts of the map are blocked by cliffs and moving units have to avoid collisions with moving obstacles. Units can shoot at opponents units and buildings and the goal is to destroy all of opponents buildings.

This requiers a navigation method able to control simultaneous movements of many units that works in highly dynamic environment combined with reasoning about strategies.

Navigation is achieved using potential fields that are introduces in the Section 3.3.2. Assignment of potential fields for objects in the Strategic combat scenario is described in the section 5.1. Joining individual war units into groups is explained in the Section 5.2. Extension of potential fields with pathfinding is described in the Section 5.3. Reasoning about strategies using Monte-Carlo tree search is explained in the Section 5.4.

5.1 Units movement using potential fields

When designing a unit navigation method using potential fields, at first it is needed to identify fields acting in an environment. Objects that need to be avoided are assigned repelling fields, destinations that need to be reached are assigned attractive fields. Unit

movement is then controlled by a gradient of the sum of all potential fields acting in an environment.

attractive fields	target enemy objects	target enemy units
		target enemy buildings
repelling fields	environment objects	cliffs
		moving obstacles
	players objects	players units
		players buildings
	neutral enemy objects	enemy units
		enemy buildings

Table 5.1: Types of potential fields in the Strategic combat scenario

Identification of potential fields in the Strategic combat scenario is shown in the table 5.1. Objects in the scenario have been divided into three groups, objects that do not belong to any of the players are referred as environment objects, object under control of an opponent player are called enemy objects and objects under our control are referred as players objects or own objects.

Potential fields of players and environment objects have to be assigned repelling fields because it is needed to avoid them during an unit movement. Enemy units can generate both repelling and attactive fields, that depends on a decision making algorithm that decides which enemy units to avoid and which to attack.

Values of potential fields have been selected based on experiments with real-time simulations using ORTS. Potential fields definitions assume that an object is situated in a space with x and y coordinates and the object is at the zero pozition $(x_0, y_0) = (0, 0)$.

Assigned potential fields for players objects are described in the Section 5.1.1, potentials for cliffs and moving obstacles are explained in the Section 5.1.2 and potential fields for enemy objects are defined in the Section 5.1.3.

5.1.1 Potential fields of players objects

Players objects include buildings that cannot move and units that move in an environment. Players units have to avoid collisions with other players units and buildings,

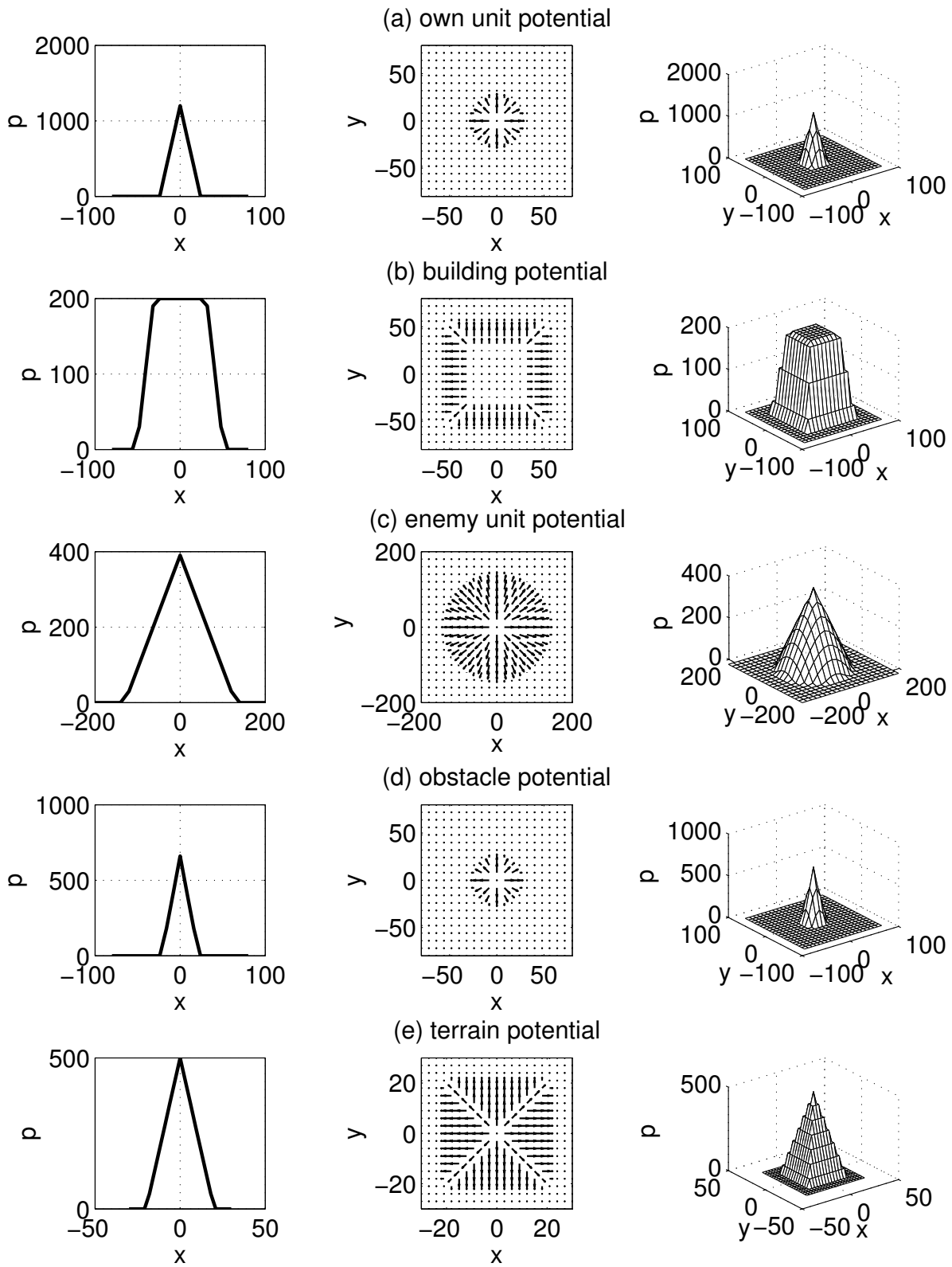


Figure 5.1: Repelling potential fieds for obstacle avoidance: (a), (b) players objects potentntials, (b), (c) enemy objects potentntials, (d),(e) environemnt objects potentials

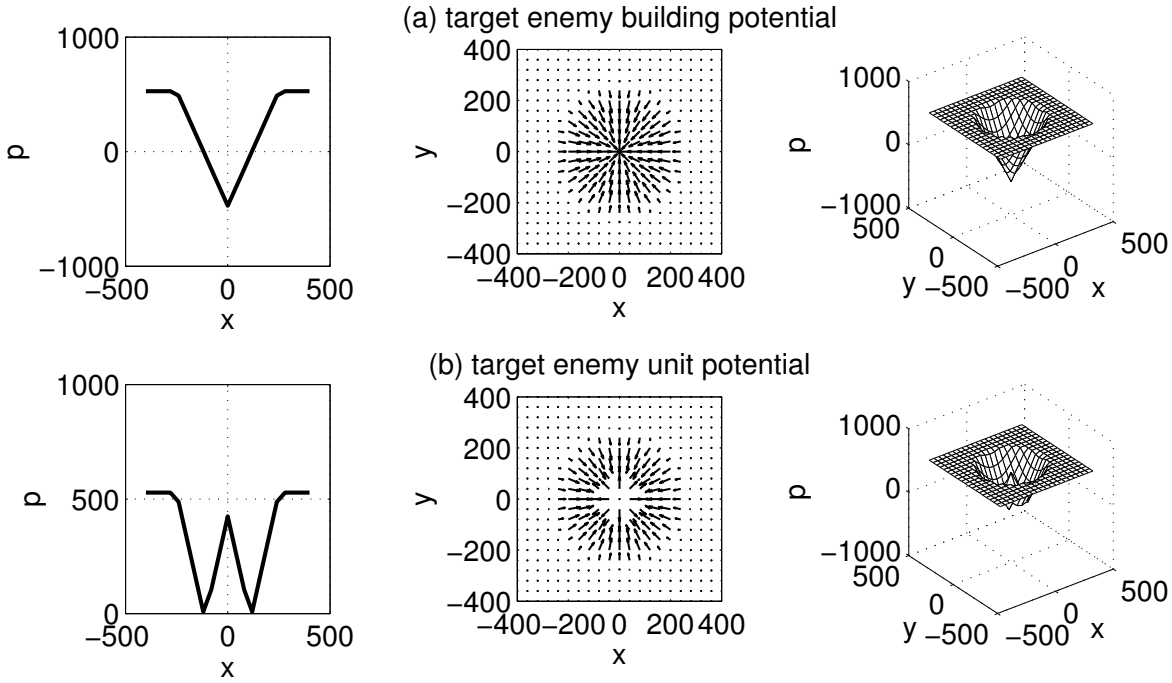


Figure 5.2: Attractive potential fields to reach a destination

because if two players objects collide, ORTS makes them stop until they move away from each other. This is a reason why all players objects have assigned repelling fields.

Potential of players units has been set to

$$p_{\text{ownunit}}(x,y) = \begin{cases} (24-d)\cdot 50 & \text{if } d \leq 24 \\ 0 & \text{otherwise} \end{cases} \qquad d = \sqrt{x^2+y^2}, \tag{5.1}$$

value of potential decreases linearly with eucleidian distance to an object, positions further that 24 are not affected by this field. Illustration of this field is in the fig. 5.1 (a).

Potential of players buildings set to a high value on places occupied by the building and it decreases linearly with the distance:

$$p_{\text{ownbuilding}}(x,y) = \begin{cases} 200 & \text{if } d \leq 31 \\ (51-d)\cdot 10 & \text{if } 31 < d < 51 \\ 0 & \text{otherwise} \end{cases} \qquad d = \max\{|x|,|y|\}, \tag{5.2}$$

in this case, the distance d is computed as Manhattan distance because buildings are square objects. Illustration of the potential field is in the fig. 5.1 (b).

5.1.2 Potential fields of environment objects

Environment objects include cliffs and moving obstacles that both block players unit movements. Computation of a potential field of a moving obstacle is similar to fields of players units, but because moving obstacles are smaller, potential fields affects closer area:

$$p_{\text{obstacle}}(x, y) = \begin{cases} (22 - d) \cdot 30 & \text{if } d \leq 22 \\ 0 & \text{otherwise} \end{cases} \qquad d = \sqrt{x^2 + y^2}. \tag{5.3}$$

Illustration of the obstacle potential field is in the fig. 5.1 (d).

Computation of potential field generated by cliffs is based on the distance transformation (Section 3.1.3) that computes the Manhattan distance to the closest cliff. Positions close to an edge of the map are added a penalty that avoids units to move out of the game map. Potential of game map terrain is computed as a sum of potential of the closest cliff and a penalty if there is a map edge nearby:

$$p_{\text{terrain}}(x, y) = p_{\text{cliffs}}(x, y) + p_{\text{penalty}}(x, y) \tag{5.4}$$

$$p_{\text{cliffs}}(x, y) = \begin{cases} (20 - d_t) \cdot 25 & \text{if } d_t \leq 20 \\ 0 & \text{otherwise} \end{cases}, \tag{5.5}$$

where d_t is a distance transformation that computes a distance to the closest cliff.

$$p_{\text{penalty}}(x, y) = \begin{cases} (30 - d_e) \cdot 10 & \text{if } d_e \leq 30 \\ 0 & \text{otherwise} \end{cases}, \tag{5.6}$$

where d_e is a distance to the closest edge of the map. Illustration of a potential fields generated by a cliff is in the fig. 5.1 (e).

5.1.3 Potential fields of enemy objects

Assignment of potential fields of enemy units depends on a decision-making algorithm that works on a higher level above the potential fields algorithm. The decision-making algorithm defines which enemy objects have to be avoided, these are called neutral enemy objects, and which enemy objects have to be reached and attacked, these are referred as target enemy objects.

Potential field of neutral enemy buildings is defined excactly the same way as for players buildings (eq. 5.2). Neutral enemy units generate repelling field that forces players units to avoid the whole shooting range of the enemy units:

$$p_{\mathrm{neutralenemyunit}}(x,y) = \begin{cases} (130-d)\cdot 3 & \text{if } d \leq 130 \\ 0 & \text{otherwise} \end{cases} \qquad d = \sqrt{x^2+y^2}, \tag{5.7}$$

Target enemy objects generate attractive fields. Target enemy buildings generate field that forces players units to move as close as possible to the building

$$p_{\mathrm{targetenemybuilding}}(x,y) = \begin{cases} (d-118)\cdot 4 & \text{if } d \leq 250 \\ 1056 & \text{otherwise} \end{cases} \qquad d = \sqrt{x^2+y^2}, \tag{5.8}$$

illustration of the target enemy building potential field in the fig. 5.2 (a).

Target enemy units generate a field that is attractive at longer distances, repulsive on shorter distances and has a minimum exactly at the shooting range of the units:

$$p_{\mathrm{targetenemyunit}}(x,y) = \begin{cases} (106-d)\cdot 4 & \text{if } d \leq 106 \\ (106-d)\cdot 8 & \text{if } 106 < d \leq 112 \\ (d-118)\cdot 8 & \text{if } 112 < d \leq 118 \\ (d-118)\cdot 4 & \text{if } 118 < d \leq 250 \\ 528 & \text{otherwise} \end{cases} \qquad d = \sqrt{x^2+y^2}, \tag{5.9}$$

illustration of the target enemy unit potential field in the fig. 5.2 (b).

Example of a combination of all types of potential fields in a strategic combat scenario

is shown in the fig. 5.3.

5.2 Game state abstraction

Potential fields work very well with large number of individual units, but decision-making algorithms may have trouble to reason about large counts of units. To make decision-making algorithms work, it is reasonable to join units into groups that move and attack together. Other reason to join units into groups is that it makes them more powerful when attacking together.

This chapter illustrates usage of hierarchical clustering algorithm (Section 3.4.2) for searching for groups of units. A centroid linkage metric (Eq. 3.17) was used because it gives suitable results and is easy to compute.

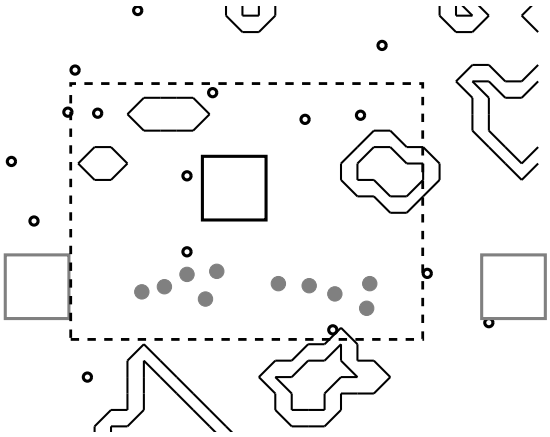
Results of the hierarchical clustering algorithm with centroid linkage metric is shown in the Fig. 5.4. Size of clusters can be easily controlled by thresholding.

5.3 Path finding extension to potential fields

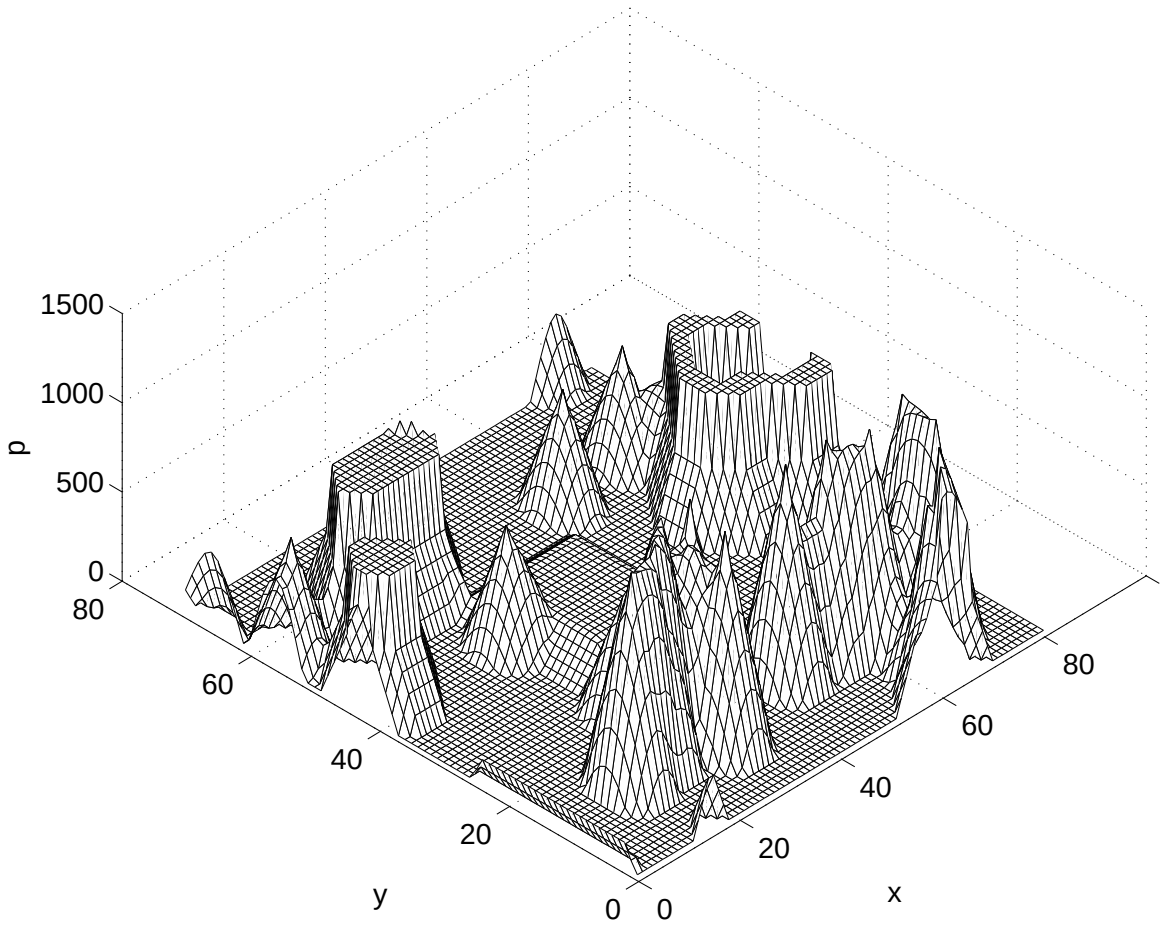
Potential fields are very good method for avoiding cliffs and moving obstacles, but suffer from stucking in local minima. Even in a very simple scenario (Fig. 5.5) they are not able to navigate a unit into its destination.

This section describes an idea of extending potential fields method with path finding, where path finding algorithms find a path from a starting position to a destination that avoids cliffs and buildings, and than lets potential fields navigate units along the found path to the destination.

Because path finding is more computationally intensive than using potential fields, it is practical to mimimize number of runs of a path finding algorithm. Because decision-making algorithms reason about whole groups of units rather than about individuals, it is desired to find paths for whole groups of units and let potential fields to drive individuals along the found paths.



(a)



(b)

Figure 5.3: Example of sum of all potential fields acting in a game scenario: (a) part of a game map has been selected, the selection is marked with the dashed line (b) potential fields of objects in the selected part of the map

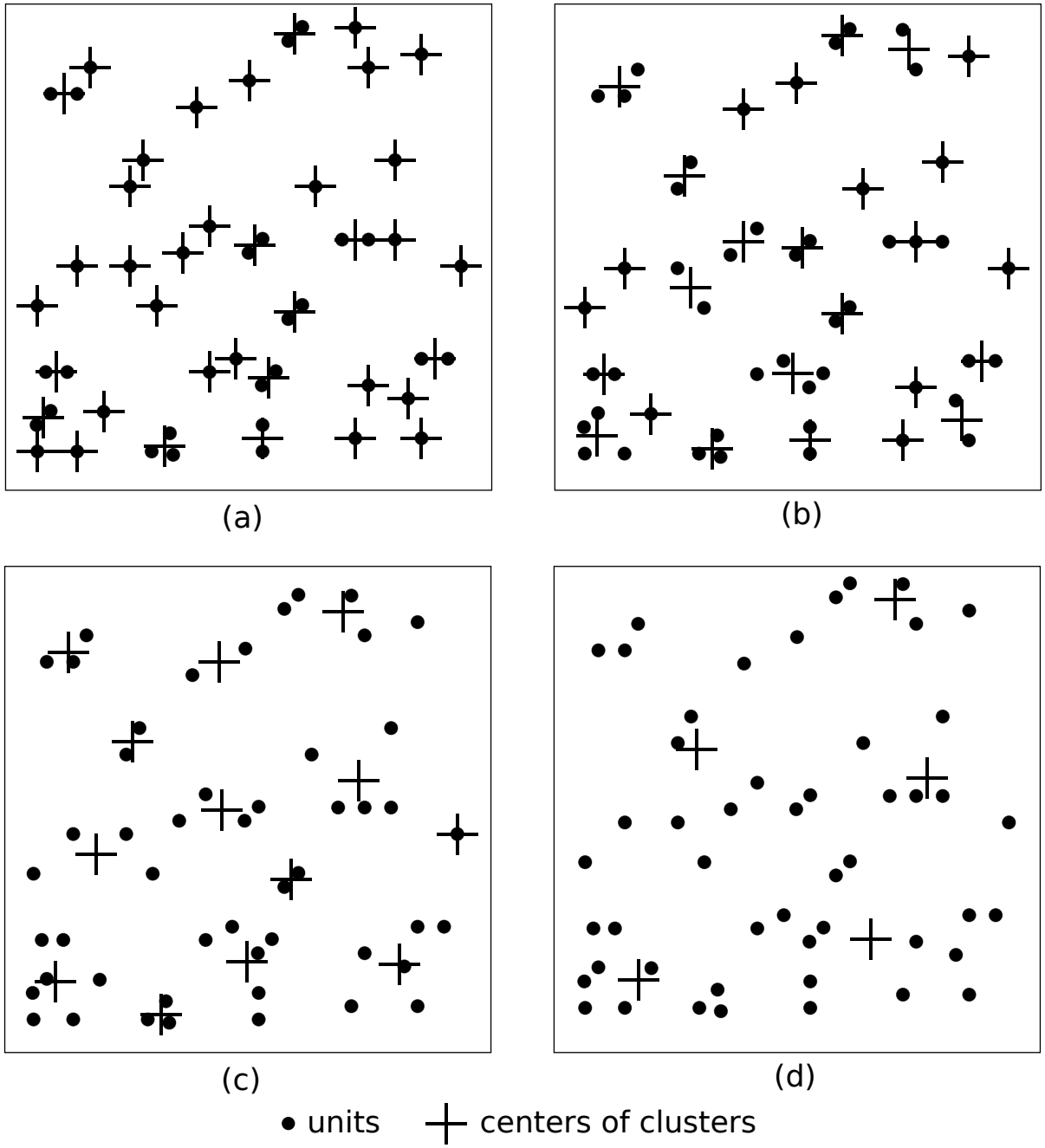
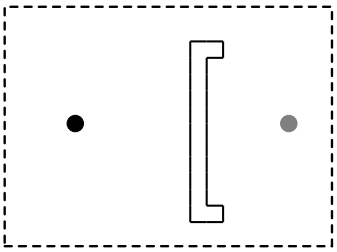
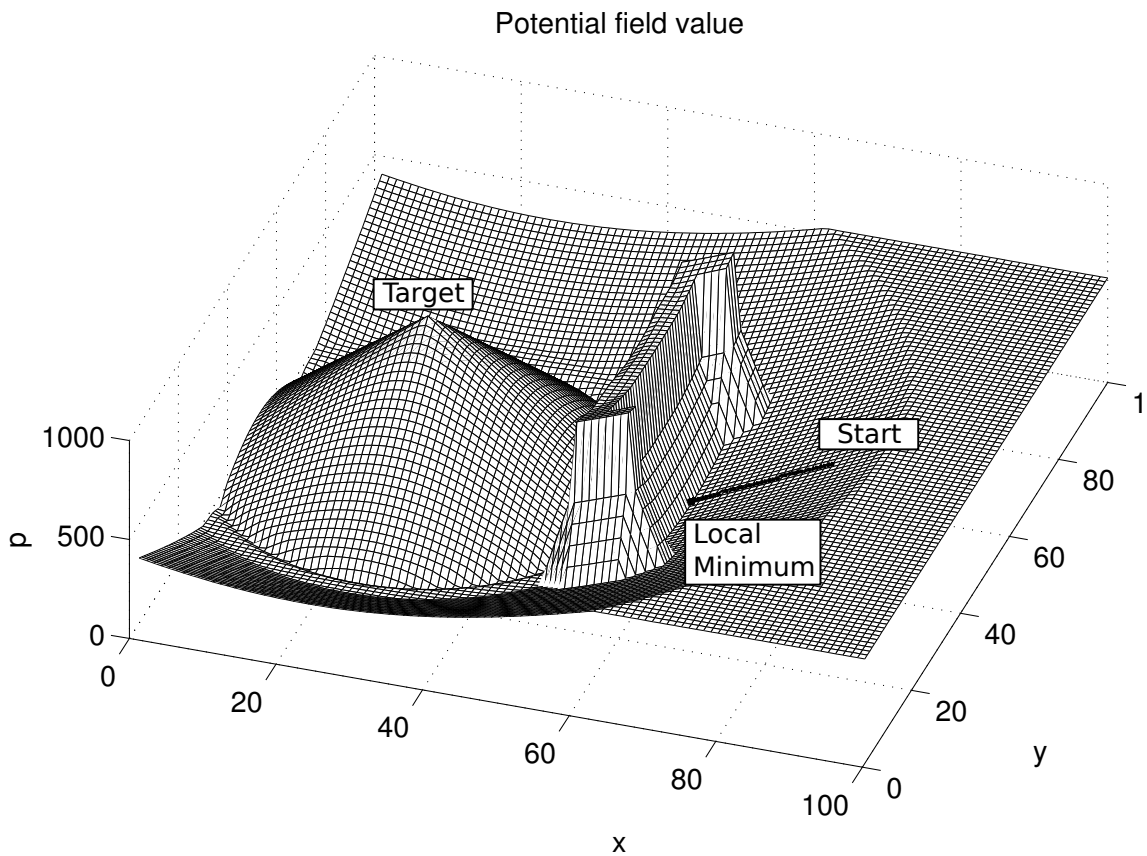


Figure 5.4: Example of clusters found from random unit positions using hierarchical clustering algorithm with centroid linkage metric with different maximal cluster distance thresholds: (a) threshold 32 (b) threshold 64 (c) threshold 112 (d) threshold 200



(a)



(b)

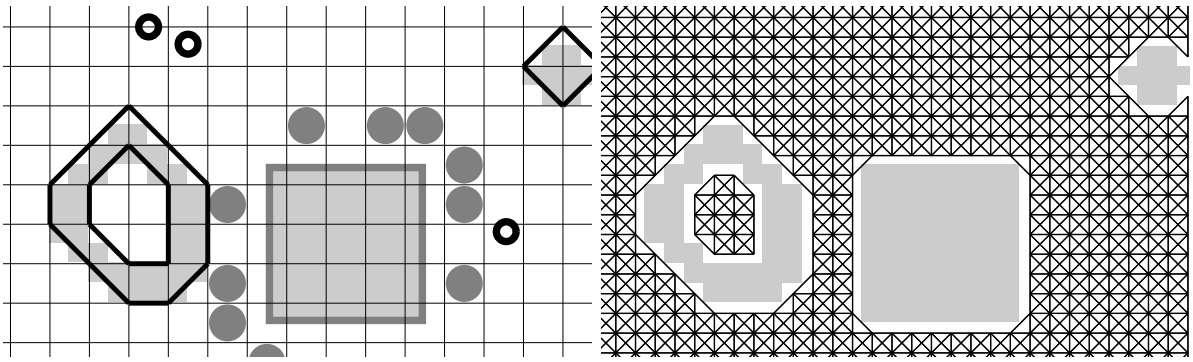
Figure 5.5: Example of a problem of unit navigation using potential fields: potential fields are not able to avoid local minimum. (a) simple game scenario with one players unit, one enemy unit and a cliff between them (b) values of potential field and a path of players object determined by potential field navigation.

Prior to running pathfinding algorithm, is it needed to contruct a graph that can be searched. Construction of a graph for pathfinding is described in the Section 5.3.1. Combination of pathfinding and potential fields is shown in the Section 5.3.2.

5.3.1 Path finding graph construction

Construction of a grah for path finding is divided into two phases. In the first phase, a binary image called passage map is created, it stores information which part of a game map is free for movement and which is blocked. When the passage map is created, its information is used to create the graph for pathfinding.

An example of a passage map is in the fig. 5.6 (a). Positions on the map blocked by cliffs and players bases are declared as impassable and path finding algorithms have to avoid such positions. Other positions are declared passable. Positions blocked by enemy bases do not need to be declared as impassable because enemy bases get destroyed when players units get nearby. Positions blocked by war units or moving obstacles are not taken into account for path finding purposes, these obstacles can be avoided using potential fields navigation algorithm. Granularity of the passage map is set to 2×2 points of passage map in one map tile, it seems to be enough accurate, paths found using passage serve only as rough approximations of shortest paths.



(a) (b)

Figure 5.6: Examples of a passage map and searching graph: (a) passage map, gray fields are marked as impassable, that means that those fields block movement of units. Game map tiles are shown (b) graph used for path finding

A passage map serves as an input into a graph construction algorithms. The algorithm

is very simple, for every passable point in a passage map it creates edges to all passable points in its 8-neighborhood. An example of a path finding graph is shown in the fig. 5.6 (b). Example of a path found using a graph build from the passage map is shown in the Fig. 5.7 (a).

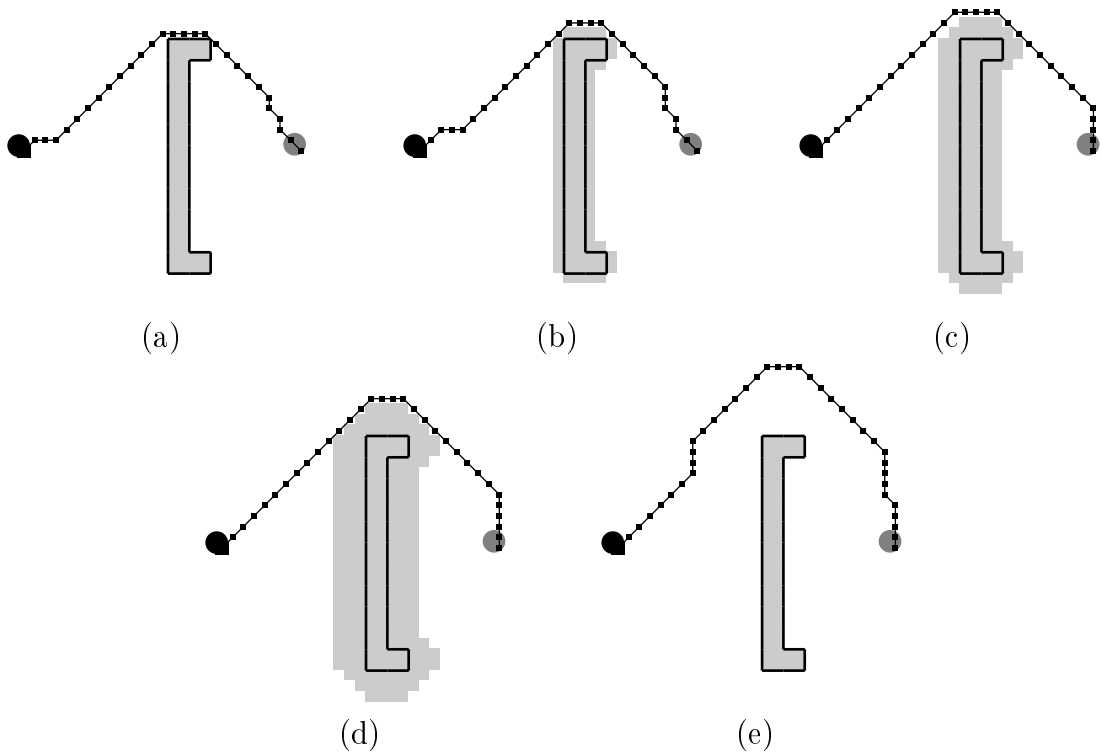


Figure 5.7: Examples of paths found in different passage map transformations in a simple scenario containing only one players unit, one enemy unit and a cliff in the middle: (a) no passage map transformation (b) dilatation of passage map with a structuring element shown in the Fig. 3.3 (c) passage map transformed by dilatation two times (d) passage map transformed by dilatation three times (e) graph edge costs weighted by their distance to the nearest terrain

A path found in a graph created using a passage map with the Dijkstra's algorithm (Section 3.2.1) is the shortest path possible, as shown in the Fig. 5.7 (a), but it forces groups of units to move too close to cliffs, that slows unit group movement. A straightforward solution is to use a dilatation (Section 3.1.2) to mark location near cliffs as impassable. An example of a dilatation is in the Fig. 5.7 (b), example of using the dilatation multiple times is in the Fig. 5.7 (c, d). Dilatation is a fast algorithm that is able to ensure that found paths keep distance to cliffs, but this approach fails when only

possible paths goes through a narrow passage, as shown in the Fig. 5.8 (a).

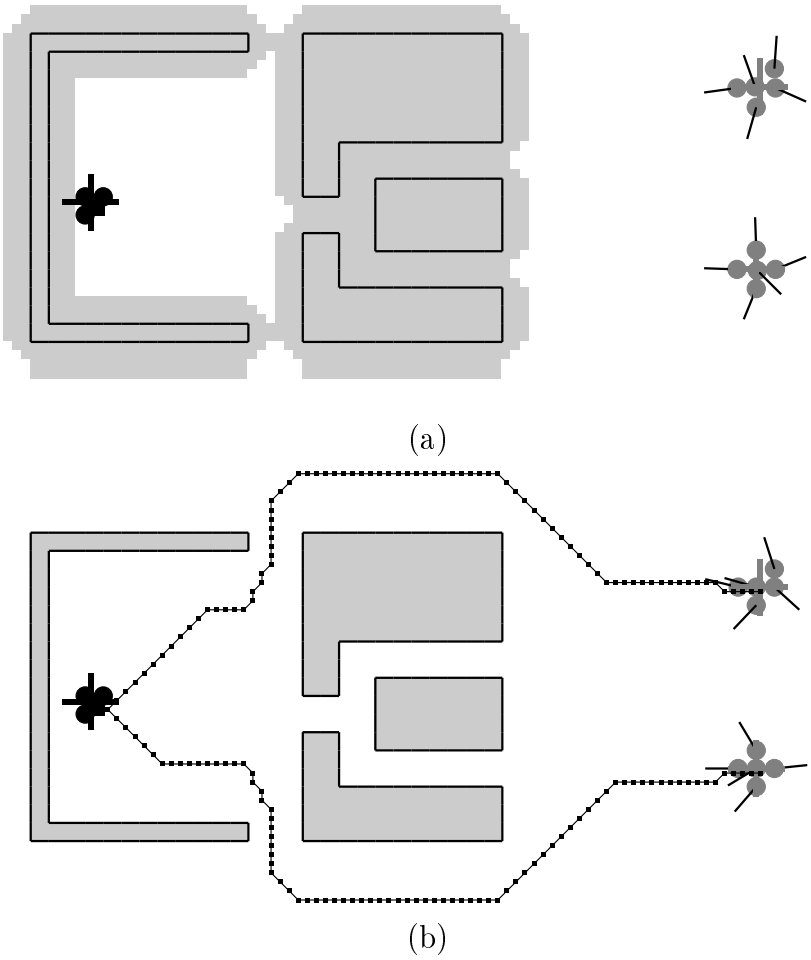


Figure 5.8: Examples of results of path finding: (a) dilatation of the passage map blocks narrow passages and no paths are found (b) using graph with edges weighted by distance to cliffs a path tries to avoid the cliffs, but it finds a way through the passage

Another possibility to make path finding avoid cliffs is to penalize paths that are close to cliffs. The distance transformation algorithm (Section 3.1.3) is used to compute distance to the nearest cliff for every edge of the graph. Cost of the edges is computed as folows:

$$c = e^{\max\{10-d,1\}}, \tag{5.10}$$

where d is a distance to the nearest cliff measured in number of tiles in a passage map. Results of a path finding algorithm is in the Fig. 5.7 (e). This approach works well also for narrow passages as it is shown in the Fig. 5.8 (b).

5.3.2 Combining pathfinding with potential fields

Combination of pathfinding and potential field is performed by generating a descending valley along the found path, where every path vertex v_i in a path $v_0, v_1, \ldots v_n$ generates a potential

$$p_{\text{path}}(x, y, i) = -400e^{\frac{-d^2}{4000}} - 20i, \qquad d = \sqrt{x^2 + y^2}. \tag{5.11}$$

Example of a valley generated by this function is in the Fig. 5.9.

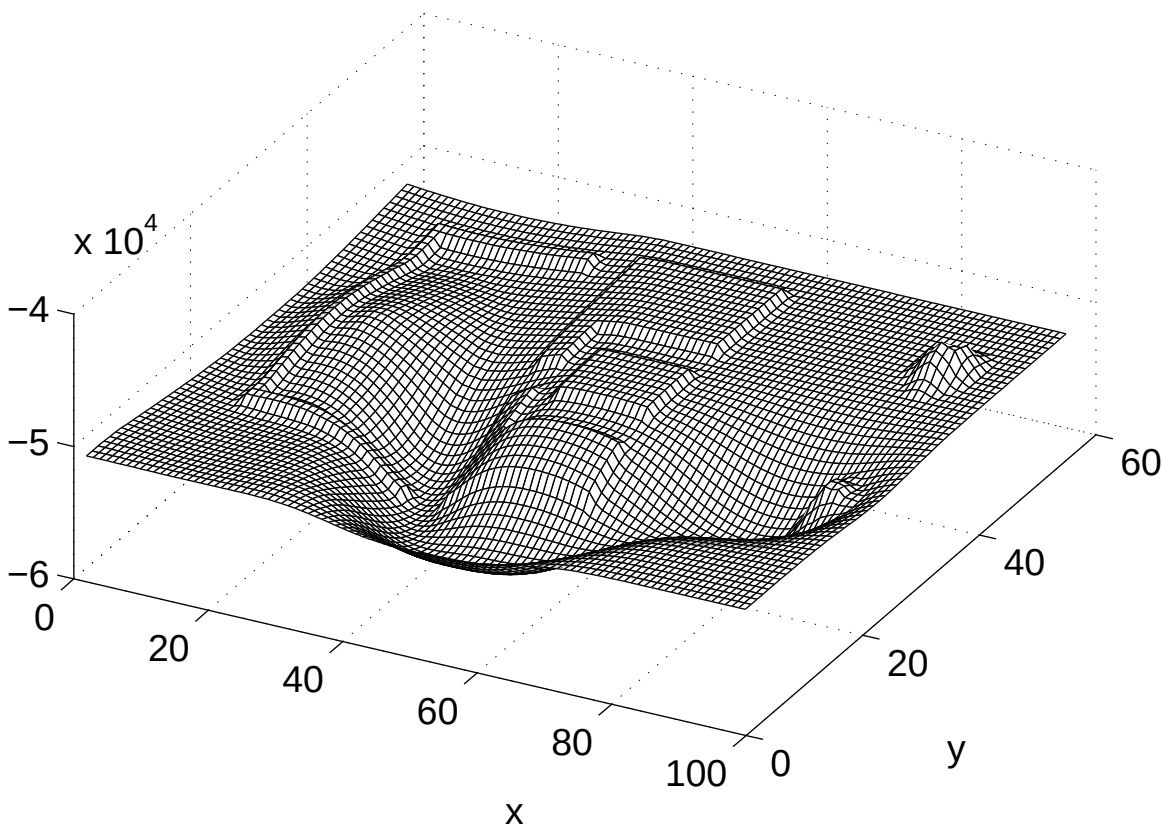


Figure 5.9: Example of a valley with low potential generated by a path, the figure illustrates potential field of the same scenario as in the Fig. 5.8, only the potential of the lower path is shown.

5.4 Monte-Carlo tree search implementation

Implementation of the MCTS algorithm is based on the method described in the Section 3.6.2, but it includes several modifications that make it work in real-time strategy games.

Reasoning about individual units would be impossible due to high branching factor of the game tree, so it is needed to find an abstraction that describes a situation of the game in a simpler way. A description of a game state assumes that units have been joined into groups and contains only positions of the whole groups, number of units in the groups and a total amount of hit points of containing units.

Our version of a game tree is shown in the Fig. 5.10. Starting from the initial node, both players try to make groups of units using the hierarchical clustering method (Section 3.4.2) with five different thresholds, 120, 100, 80, 64 and 48. After that, both players assign every group of units an action that can be of two types, an attack action causes a unit group to reach and attack a group of units of an opponent, and a join action causes a group of units to meet another players group of units and combine them into a larger group. All of these actions do not increment the time in the game.

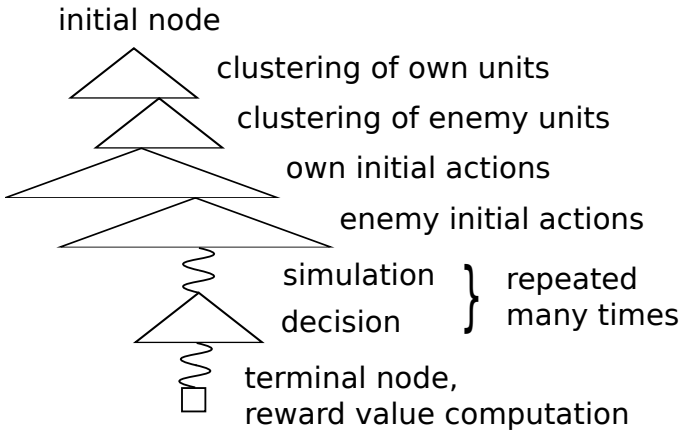


Figure 5.10: Structure of a game tree that is searched using the MCTS algorithm

When there is no idle group (a group with no action assigned), a state is expanded by simulating movements and attacks of groups of units. When a group of units becomes idle, the simulation is stopped and possible resulting states are expanded.

Because the branching factor of a game tree in RTS games is very high even when using reasoning only about whole groups of units, two techniques are used to cut off states that do not seem to lead to good solutions. The first is using a heuristic function that assumes that it is reasonable to attack or join only near groups of units. This limits the branching factor of initial nodes to several thousands of resulting states. The second approach is using a progressive unprunning method, each of the initial nodes expands

only n successors, other n successors are expanded after every v visits of the node, where n and v are parameters.

Selection and value backpropagation is implemented as an UCT algorithm (Section 3.4.2), where reward of a terminal node for players player is computed as

$$R = \begin{cases} \frac{1}{2} & \text{if } HP_e = HP_p \\ 0 + \frac{1}{2} \cdot \frac{HP_i - HP_e}{HP_i} & \text{if } HP_e > HP_p \\ 1 - \frac{1}{2} \cdot \frac{HP_i - HP_p}{HP_i} & \text{if } HP_e < HP_p, \end{cases} \tag{5.12}$$

where HP_e is a sum of hit points of opponent's units, HP_p is a sum of HP of players units and HP_i is a sum of hit points of all units of a player at the beginning of the game. This function takes values in the interval $[0, 1]$, if there is a small difference between the two players, the function returns a value near $\frac{1}{2}$, if one player has much more units that the other, the value get close 0 or 1, depending of which player wins the game. During the backpropagation, nodes under opponent's control are assigned a reward $1 - R$.

5.5 Player control

AI player application uses potential fields for unit movement, abstractions to simplify reasoning, path finding for finding routes for groups of units and Monte-Carlo tree search for reasoning about clustering and unit group actions.

Player behavior si defined by two processes that work at different levels of abstraction. A process called player control system reasons about actions of whole groups of units, unit control system controls actions of single units in the game.

The player control system flow chart is shown in the Fig. 5.11 (a). At the beginning of the game, is assigns groups of units targets that have the closest path distance to the group. After that is runs MCTS simulation, extract the best plan and assigns unit targets according to the found plan. Monte-Carlo simulations are repeated as long as there are enough units in the game. If any of the players in the game has less that 6 units, MCTS search is deactivated and units start to attack closest targets.

The unit control system flow chart is shown in the Fig. 5.11 (b). For each unit, a potential field is computed and the units moves in the direction of the biggest decrease of the potential field gradient. If there are enemy units in range, it founds the weakest unit and attacks it.

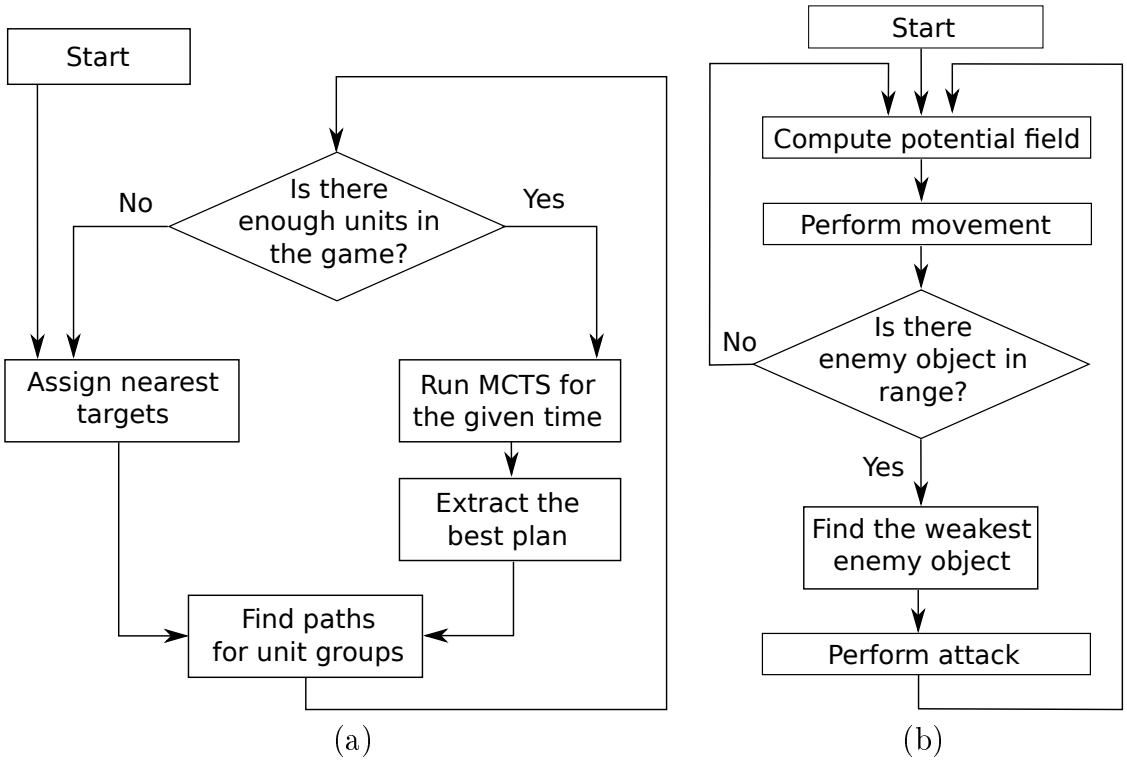


Figure 5.11: Subsystems of AI player: (a) player control system (b) unit control system

An example state of the ORTS strategic combat scenario is shown in the Fig. 5.12. Crosses depict centers of groups that have been extracted from the best plan of the MCTS algorithm. Paths show found routes to assigned enemy groups of units. Each players unit shows a direction of a movement using a black line.

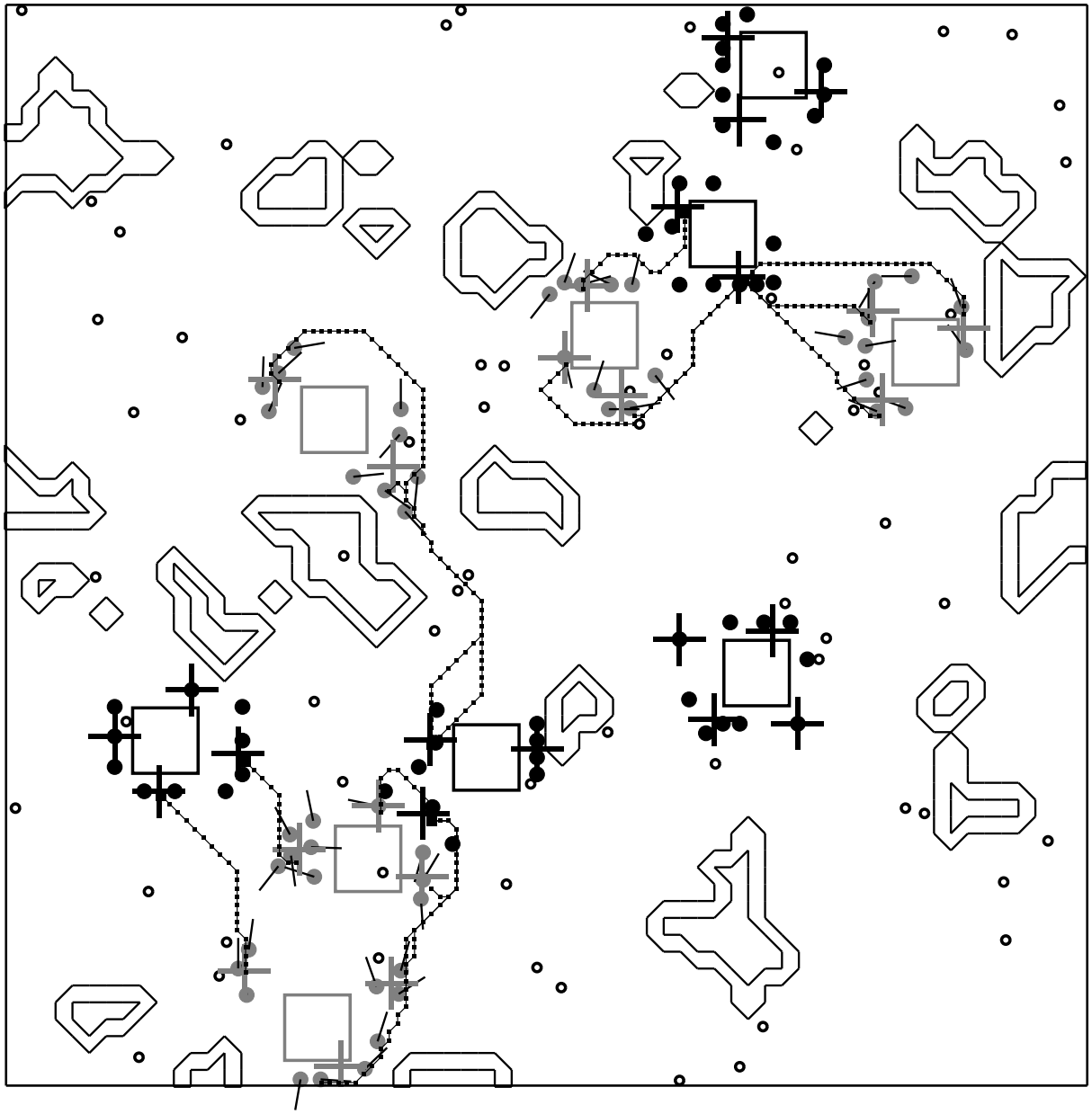


Figure 5.12: A player solving ORTS strategic combat scenario: players objects are shown gray, enemy object are depicted black. Squares are bases that have to be destroyed, round filled objects are units, small round objects are moving obstacles. Black lines show routes found by path finding algorithm, units show a direction of movement using short black lines, the direction has been computed using potential fields method.

Chapter 6

Practical experiments

Developed AI player for the strategic combat scenario, referred as *AdvancedAI*, has been tested agains two existing players. The first tested player was developed as a part of previous work [9] and is referred as *SimpleAI*. The second tested player *Bleckinge*, developed by the Bleckinge team [12], was a winner of the 2009 AIIDE competition [1] and thus is one of the best existing AI players for the given scenario.

Developed Monte-Carlo algorithm has four parameters that can changed to tune the alogrithm behavior:

- time of MCTS decision epoch t, a time limit for MCTS sampling
- maximal depth of the game tree d
- number of unprunned nodes n when an unprunning condition occurs
- number of node visits to fulfill the unprunning condition v

Testing of all combinations of changes of parameters t, d, n, v would be extremely time consuming. To make testing possible, impact of every parameter was tested separately according to the testing plan in the Table 6.1. Each row of the testing plan defines a combination of MCTS parameters. Each combination in the table has been simulated in 10 games agains each opponent, *SimpleAI* and *Bleckinge*.

Results of simulations of strategic combat scenario againts the simpleAI player is shown in the Fig. 6.1. Each value shown in the plot is a mean value of 10 game simu-

Experiment	t	d	n	v
Change of decision epoch length	**500**	15	10	500
	1000	15	10	500
	2000	15	10	500
	5000	15	10	500
Change of maximal game tree depth	1000	**10**	10	500
	1000	**15**	10	500
	1000	**20**	10	500
	1000	**50**	10	500
Change of number of unprunned nodes	1000	15	**5**	500
	1000	15	**10**	500
	1000	15	**20**	500
	1000	15	**40**	500
Change of number of node visits for unprunning	1000	15	10	**200**
	1000	15	10	**500**
	1000	15	10	**1000**
	1000	15	10	**2000**

Table 6.1: AI player testing plan, each row describes one configuration of MCTS parameters: t is a length of the MCTS decision epoch, d is a maximal of the game tree, n is a number of unprunned nodes and v is a number of node visits to perform unprunning

lations, the plots show number of units that survived after one of two players won the game. Results of game simulation agains the Bleckinge player is in the Fig. 6.2.

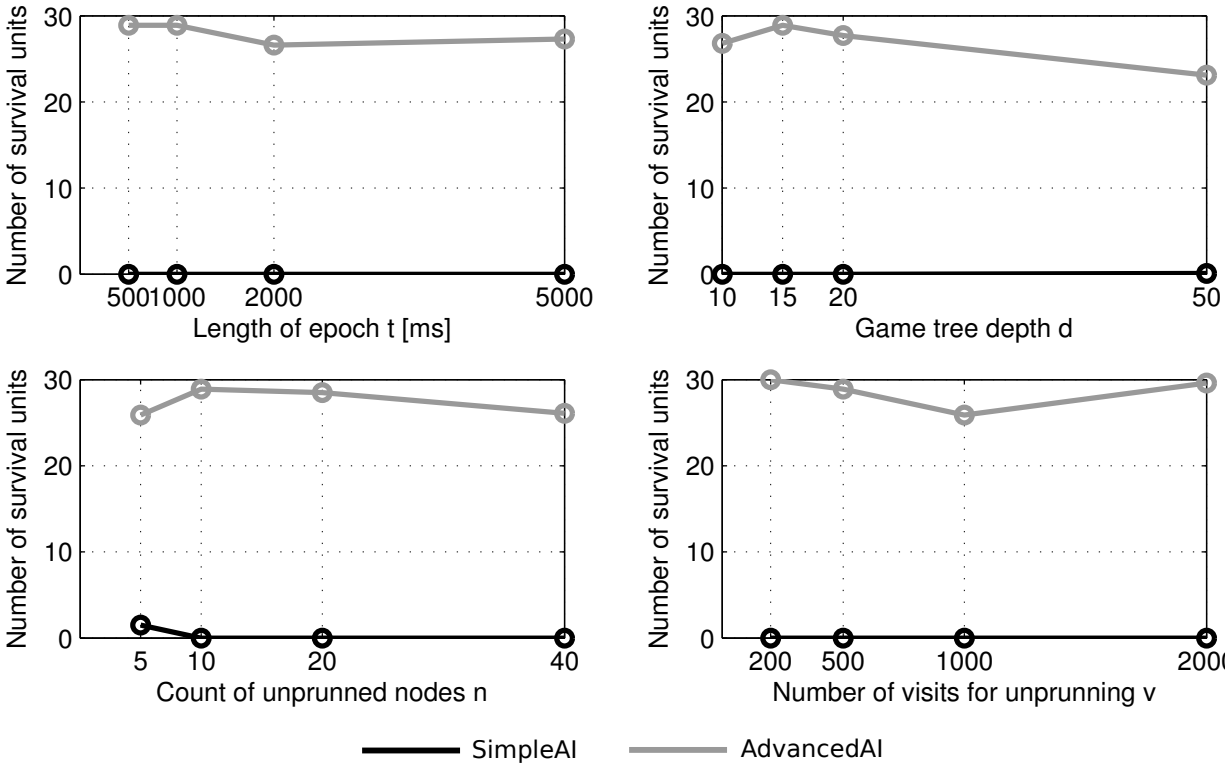


Figure 6.1: Testing of different parameters of MCTS algorithm against the simpleAI player

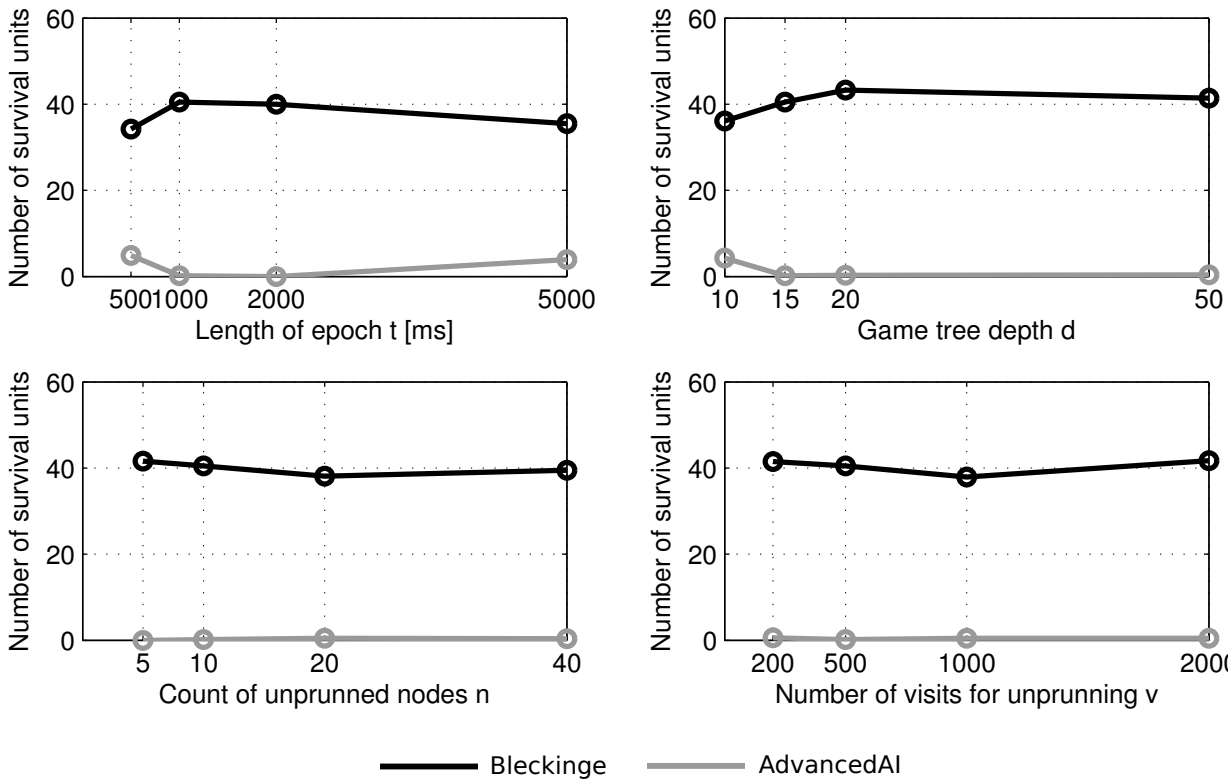


Figure 6.2: Testing of different parameters of MCTS algorithm against the Bleckinge player

Chapter 7

Conclusion

Developing players for RTS games requires dealing with several problems like movement of the units, resource gathering, map exploration and reasoning about players strategies. Task of this work was to develop an AI player for the strategic combat scenario using potential fields and Monte-Carlo tree search.

Potential fields have limitations like stucking in local minima, so a path finding extension to potential fields was developed and implemented. An UCT Monte-Carlo tree search node selection strategy was used to select more promissing nodes in the game tree. To decrease the branching factor of the game tree, a progressive unprunning method was used.

Resulting player was tested agains two existing players, the *SimpleAI* player that has been developed as a part of previous work[9], and the *Bleckinge* player, s winner of the AIIDE 2009 RTS game competition.

Resulting Monte-Carlo tree search algorithm has several parameters, line time length of decision epoch, maximal depth of the game tree, number of unprunned nodes and number of node visits to execute unprunning. Many combinations of these parameters has been tested using ORTS simulations, but changing MCTS parameters had very low impact on players performance. Reason to this low impact could be very small scenario map, where lots of units get destroyed before they can form strategy unit groups move to other locations of the game map. Significant improvement to the *SimpleAI* player was using potential fields for movement where units keep distance to enemies exactly at their

shooting range.

The new resulting algorithm significantly outperforms the *SimpleAI* player, but is outperformed by the *Bleckinge* player. The resulting algorithm usually wins the scenario over the *SimpleAI* player while saving average number of 27.6 of 50 units. The *Bleckinge* player usually wins over the resulting player while saving average count of 39.5 of 50 units.

Bibliography

[1] 2009 ORTS RTS Game AI Competition. Web Page, April 2012. [online] `http://web.cs.ualberta.ca/~mburo/orts/AIIDE09/index.html`.

[2] R.K. Balla and A. Fern. UCT for Tactical Assault Planning in Real-Time Strategy Games. In *Proceedings of the International Joint Conference on Artificial Intelligence. Pasadena, California: Morgan Kaufmann*, pages 40–45, 2009.

[3] Michael Buro. ORTS: A hack-free RTS game environment. *Lecture notes in computer science*, pages 280–291, 2003.

[4] Michael Buro. Real-time strategy games: A new AI research challenge. In *International joint conference on artificial intelligence*, volume 18, pages 1534–1535. Lawrence Erlbaum associates LTD, 2003.

[5] Michael Buro, James Bergsma, David Deutscher, Timothy Furtak, Frantisek Sailer, David Tom, and Nick Wiebe. AI System Designs for the First RTS-Game AI Competition. 2008.

[6] G. M. J. B. Chaslot, M. H. M. Win, and B. Bouzy. Progressive strategies for monte-carlo tree search.

[7] Guillaume Chaslot, Mark Winands, Jaap H. van den Herik, Jos Uiterwijk, and Bruno Bouzy. Progressive strategies for Monte-Carlo tree search. In *Joint Conference on Information Sciences, Salt Lake City 2007, Heuristic Search and Computer Game Playing Session*, 2007.

[8] Michael Chung, Michael Buro, and Jonathan Schaeffer. Monte Carlo Planning in RTS Games. In *CIG*. IEEE, 2005.

[9] Viktor Chvatal. *Practical application of AI methods in RTS games.* Bachelor's thesis, Czech Technical University, Faculty of Electrical Engineering, Department of Cybernetics, 2010.

[10] Renato Luiz de Freitas Cunha and Luiz Chaimowicz. An artificial intelligence system to help the player of real-time strategy games. In *Proceedings of the 2010 Brazilian Symposium on Games and Digital Entertainment*, SBGAMES '10, pages 71–81, Washington, DC, USA, 2010. IEEE Computer Society.

[11] Edsger. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[12] Johan Hagelbäck and Stefan J. Johansson. Using multi-agent potential fields in real-time strategy games. In *AAMAS '08: Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems*, pages 631–638, Richland, SC, 2008. International Foundation for Autonomous Agents and Multiagent Systems.

[13] E. J. Friedman Hill. Jess, the Java Expert System Shell. Technical report, SANDIA National Laboratories, 2000.

[14] Václav Hlaváč and Miloš Sedláček. *Zpracování signálů a obrazů.* České vysoké učení technické v Praze, 2009.

[15] A K Jain, M N Murty, and P. J. Flynn. Data clustering: A review, 1999.

[16] Renee Jansen and Nathan Sturtevant. A new approach to cooperative pathfinding. In *AAMAS '08: Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems*, pages 1401–1404, Richland, SC, 2008. International Foundation for Autonomous Agents and Multiagent Systems.

[17] Demel Jiří. *Grafy a jejich aplikace.* Academia, 2002.

[18] David Joyner, Minh Van Nguyen, and Nathann Cohen. *Algorithmic Graph Theory.* March 2012. [online, version 0.7-r1984] `http://code.google.com/p/graphbook/`.

[19] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou, editors, *Proceedings of the Seventeenth European Conference on Machine Learning (ECML 2006)*, volume

4212 of *Lecture Notes in Computer Science*, pages 282–293, Berlin/Heidelberg, Germany, 2006. Springer.

[20] Mark V. Lawson. Finite automata. In *Handbook of Networked and Embedded Control Systems*, pages 117–144. 2005.

[21] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In L. M. Le Cam and J. Neyman, editors, *Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297. University of California Press, 1967.

[22] N. J. Nilsson P. E. Hart and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems, Science, and Cybernetics*, SSC-4(2):100–107, 1968.

[23] Marc J. V. Ponsen, Stephen Lee-Urban, Hector Munoz-Avila, David W. Aha, and Matthew Molineaux. Stratagus: An open-source game engine for research in real-time strategy games. In D. W. Aha, H. Munoz-Avila, and M. van Lent, editors, *Papers from the IJCAI Workshop on Reasoning Representation and Learning in Computer Games*, 2005.

[24] Mike Preuss, Nicola Beume, Holger Danielsiek, Tobias Hein, Boris Naujoks, Nico Piatkowski, Raphael Stüer, Andreas Thom, and Simon Wessing. Intelligent group movement and selection in realtime strategy games. 2008.

[25] Azriel Rosenfeld and John L. Pfaltz. Sequential operations in digital picture processing. *J. ACM*, 13(4):471–494, October 1966.

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

[27] Jean Serra. *Image Analysis and Mathematical Morphology*. Academic Press, Inc., Orlando, FL, USA, 1983.

[28] Y. Shoham and K. Leyton-Brown. *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations*. Cambridge University Press, 2009.

[29] Nathan R. Sturtevant. An analysis of uct in multi-player games. In *CG '08: Proceedings of the 6th international conference on Computers and Games*, pages 37–49, Berlin, Heidelberg, 2008. Springer-Verlag.

[30] Nathan R. Sturtevant and Michael Buro. Improving collaborative pathfinding using map abstraction. In *AIIDE*, pages 80–85, 2006.

[31] C.W. Warren. Global path planning using artificial potential fields. In *Robotics and Automation, 1989. Proceedings., 1989 IEEE International Conference on*, pages 316 –321 vol.1, may 1989.

Appendix A

Attached CD Content

The CD attached to this work includes:

- Electronic version of this document
  - `/doc/dp.pdf`
- Information how to run ORTS simualtion and players
  - `/doc/readme.pdf`
- JavaDoc documentation for developed player
  - `/doc/RealTimeStrategy-javadoc.tar.bz2`
- Archive with source codes of the ORTS engine, *SimpleAI* and *Bleckinge* players
  - `/src/orts.tar.bz2`
    - The orts server is located in `/orts/apps/orts`
    - The simpleai player can be found in `/orts/apps/simpleai`
    - The Bleckinge player is in `/orts/apps/tankbattle5`
- C++ client library
  - `/src/ortsclient.tar.bz2`
- Developed player (java project)
  - `/src/RealTimeStrategy.tar.bz2`