University of West Bohemia in Pilsen
Department of Computer Science and Engineering
Univerzitni 8
30614 Pilsen
Czech Republic

Motion planning for geometric models in data visualization

The State of the Art and Concept of Ph.D. Thesis

Jakub Szkandera

Technical Report No. DCSE/TR-2019-01
March, 2019

Abstract

A finding of path is an important task in many research areas and it is a common problem solved in a wide range of applications. New problems of finding path appear and complex problems persist, such as a real-time planning of paths for huge crowds in dynamic environments, where the properties according to which the cost of a path is evaluated as well as the topology of paths may change. The task of finding a path can be divided into path planning and motion planning, which implicitly respects the collision with surroundings in the environment.

Within the first group this thesis focuses on path planning on graphs for crowds. The main idea is to group members of the crowd by their common initial and target positions and then plan the path for one representative member of each group. These representative members can be navigated by classic approaches and the rest of the group will follow them. If the crowd can be divided into a few groups this way, the proposed approach will save a huge amount of computational and memory demands in dynamic environments.

In the second area, motion planning, we are dealing with another problem. The task is to navigate the ligand through the protein or into the protein, which turns out to be a challenging problem because it needs to be solved in 3D with the collision detection.

This work was supported by the Ministry of Education, Youth and Sports, project LH11006 Interactive Geometrical Models for Simulation of Natural Phenomena and Crowds, by the Czech Science Foundation, project 17-07690S, and by the project SGS-2016-013 - Advanced Graphical and Computing Systems.

Copies of this report are available on
http://www.kiv.zcu.cz/publications/
or by surface mail on request sent to the following address:

University of West Bohemia in Pilsen
Department of Computer Science and Engineering
Univerzitni 8
30614 Pilsen
Czech Republic

Acknowledgements

I wish to express my gratitude to all the people who supported me during the realization of this thesis. First and foremost, I would like to thank to my supervisor Prof. Dr. Ing. Ivana Kolingerová for her never-ending patience and valued advice. My thanks also go to my family - especially to my grandmother and my girlfriend whose support was very important during my studies.

Contents

Chapter 1

Introduction

The task of finding a path between at least two spots in some environment finds its use in many areas of research and applications, e.g., in the simulations of crowds, in molecular biology or robotics. In the most general form the task is to transfer the moving object from its start position to the goal position in a predefined environment. As an example, we can mention any strategic game, such as Heroes of Might and Magic. The selected hero will automatically move to the goal position (a user-marked position) depending on the virtual environment. The hero avoids obstacles (mountains, rivers, buildings, etc.) and prefers to move along marked paths in front of other kinds of environment (marsh, snowy plains, etc.). A similar principle is also used in research areas, for example when navigating a robot.

The finding path task can be currently split into two major research areas. The first one is a path planning which suits to find the path of the geometrically simple object from the start position to the end position. Most of path planning tasks also include avoiding collisions, but this is not a prerequisite. The second area is motion planning. Unlike path planning, it often uses more complicated objects to navigate through environment. In addition, the rotation of the navigated object, which can even change its shape in more difficult tasks, is allowed.

Although it might seem that the path planning nowadays is an exhausted problem and not very complex matter, the opposite is true. Firstly, there are areas of the path planning that explore and improve their methods, and secondly, in large projects, path planning is often a major problem. In the first group, we can include, for example, the movement of a robot exploring the environment, as it gradually discovers its surroundings and tries to recalculate the path based on updated data. The next example of the problems being investigated are the crisis situations inside the buildings, such as the escape of people from the burning building. In the second group we can

include, e.g., game industrial or car navigation.

The motion planning is very often used in robotics to navigate the robot through an unknown environment, which is gradually revealed by its sensors. The autonomous vehicle navigation is also solved by this discipline just as the navigation of a drone, which is currently one of the most often problems. Another very important and relatively new problem is motion planning in a protein. Proteins are large macromolecules that consist of long atomic chains, called amino acids, and numbers may range in the order of tens to hundreds of thousands. The problem of motion planning through protein consists of a tunnel, cavity and collision detection. The idea is simple, for example to navigate probe or ligand (molecule) into the protein cavity. The protein itself is described by multiple atoms, which are represented as spheres in 3D. The navigated ligand can also be a sphere but it is already a solved problem. More difficult are complex rigid ligands, which can only rotate and move in space, and the most complicated to navigate are non-rigid ligands, which can additionally change their own structure (for example atoms bonds may bend). These problems are widely studied in multiple disciplines, for example biochemistry or computer science. New and better solutions find its application in pharmaceutical design and protein engineering.

The environment, where navigated objects are (generally known as agents) moving, can be divided into two groups – a static environment and a dynamic environment. While the planning for one agent in the static environment, who does not change during the whole simulation, can be considered as a solved problem, the situation is different for more agents or even crowds in a dynamic environment: a repeated recomputation of the path for many changes and many agents may be too slow. For a huge amount of agents the optimality of paths is less important than the speed of computation. The application of this path planning problem usually rather needs the crowd to look well and realistic than to compute and use optimal paths. The environment can also be divided into known, partially know and unknown. Both divisions suit for the path planning as well as for the motion planning. The known environment is known by all agents, unknown environment has to be gradually discovered by each agent and the partially known is somewhere in the middle of these two. Therefore, there is a research space for algorithms, sacrificing optimality for speed.

There are several ways how to represent the planning environment. For example a grid, a mesh, a graph or a configuration space. Each environment representation has its advantages and disadvantages but all of them can be used to represent any kind of the environment (static, dynamic, known, etc.). The difference is only that the path planning mostly uses the grid, mesh or graph representation, and the motion planning, which currently

uses mostly the configuration space representaion. Each representation is suitable for the path planning problems but the motion planning presumes only the configuration space or the graph representation.

There are currently a large number of path planning algorithms but they all have the common problem - they are not real-time for large crowds. We aimed for the real-time computation in a project LH11006 (Interactive Geometrical Models for Simulation of Natural Phenomena and Crowds) we focused mainly on the fast path recomputation in the dynamic city environment and how to accelerate the computation in the case of the huge amount of agents. If hundreds of agents have the same path, then we can calculate this path only once and not a hundred times to speed up the program. In the case that these hundreds of pedestrians will have a similar path, we will compute only one path that will be used for all other agents. It means that all the start positions will be in a circle with a radius ϵ, analogous to the goal positions. This will give us a faster computation, but at a price of some inaccuracy. We call this a group approach.

The group approach has been modified to be faster and more precise. The research presented in this thesis can reduce the high memory and computational demands. The idea is to take the advantage of path similarities. If some agents have a similar start and destination, they can be handled as a group. We also proposed to incorporate a more sophisticated approach of groups formation based on clustering. In this way it is possible to increase the speed-up without fatal consequences on the relative error of the produced paths.

In the course of the research, we obtained a project 17-07690S of the Czech Science Foundation that focuses on motion planning of a agent (ligand) through a protein. That is the reason why this work contains two, at a glance, unrelated parts – path planning and motion planning. However, it is still a finding a path. The group approach has been also used on the protein to navigate huge amount of spherical ligands. At present we are testing a lot of different metrics to find the optimal one which speeds up the computation and/or finds a better solution.

This thesis is organized as follows. The existing environment representations are described in Chapter 2. Chapter 3 outlines existing path planning methods, which are suitable for the simulation of crowds on the graphs. The motion planning is described in Chapter 4. Chapter 5 describes our research of the group approach. Chapter 6 contains our cluster approach and its the results compared with the group approach. Future work is in Chapter 7 and Chapter 8 concludes the thesis.

Chapter 2

Environment

One of the most important parts of the search problem is to represent the environment in which the navigated objects should move. There are a number of ways to describe an environment and each of these representations has its advantages and disadvantages. However, each representation of the environment can be converted to the graph representation.

2.1 Graph

The best data for the graph representation are the road network data, where the intersections are often represented as vertices of the graph G, and the roads that connect them as edges of the graph G. Each road has its length. This length can be attributed to each edge, giving a weighted graph G.

Definition 2.1 (Graph)**.** Graph or undirected graph G is a pair $G = (V, E)$, where V is a finite, non-empty set of objects called vertices, and E is an unordered pair of vertices ($E \subset \binom{V}{2}$). The elements of E are called edges.

Definition 2.2 (Weighted graph)**.** A weighted undirected graph $G = (V, E)$ is an undirected graph with the real function $w : E(G) \to (0, +\infty)$. The real number $w(e)$, called the weight of the edge e, is associated with each edge e.

The real urban data of the graph representation are available on the Open Street Map server [60]. The advantage of this data is that we do not have to manually assign the map vertices to the intersections and graph edges. As the name suggests, it is a database of roads, buildings, rivers and other outdoor spaces that are accessible to everyone. The example of the Open street map is shown in Figure 2.1. Its biggest advantage is at the same time the biggest disadvantage, because anyone can add or change the data. It may happen, for example, that two intersections are not interconnected by roads

Figure 2.1: The example of the Open Street Map - Pilsen. [60]

in the model, although they are interconnected in real world. The missing basic parameters such as road width or altitude may be also a disadvantage.

The advantage is that road network data can also represent underpasses or bridges in 2D environment. The graph will not be planar but it is not a problem at all. On the other hand, parks and squares cannot be well represented because the road network is unable to maintain their shape or size. The next advantage of this type of data is the found path. We can see that the path described by the graph very accurately describes the real path. In addition, we do not have to deal with collisions with surrounding objects (such as buildings, rivers, etc.) in this representation.

2.2 Mesh

The mesh representation, which is primarily suitable for the computer graphics, separates the passages from impenetrable parts. The input data can look like a sequence of points that define polygons. The polygons represent impenetrable obstacles, giving us the structure of a virtual city, see example in Figure 2.2 with white roads and gray impenetrable blocks of buildings. Subsequently, the constrained Delaunay triangulation (CDT) for all supporting points (points forming a polygon) is computed. Each triangle is then represented as a graph vertex and two adjacent triangles are connected by graph edges, see Figure 2.3 (a). This graph can still be modified using a suitable data structure. This can be a cell and portal graph [61] or a navigation graph [61], thus reducing the size of the graph while speeding up the calculation of the graph path planning methods.

Another problem with the mesh representation is the found path (Fig-

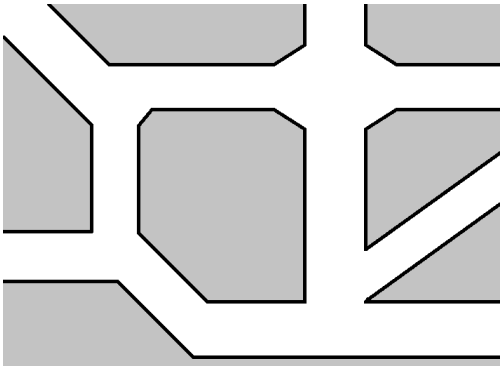
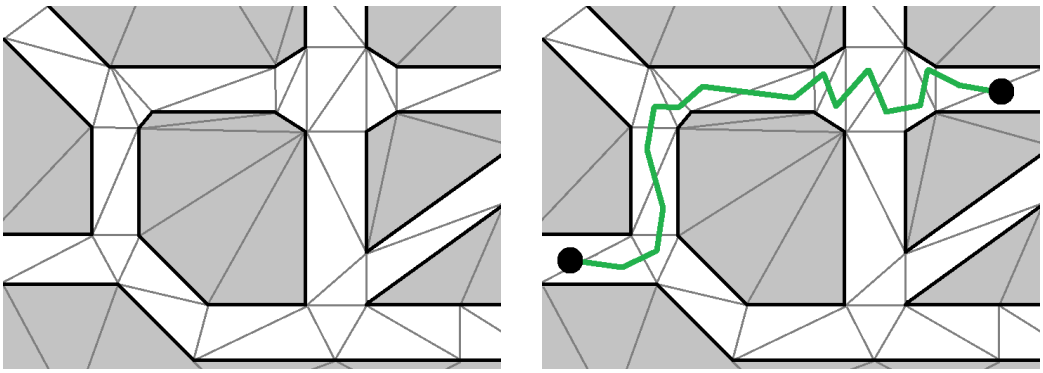


Figure 2.2: The input data for the mesh environment.

ure 2.3 (b)). Obviously, it is clear that this path is very different from the real agent movement if we do not try to simulate the movement of a drunken individual. The final path on the mesh can be approximated to make it more straight and intuitive but it will cost more computing time.

The 2D mesh does not allow us to represent subways and bridges because it allows to create only a planar graph G. On the other hand, a mesh very well represents parks and squares (e.g. shape and size). We consider these limitations only in 2D environment and do not necessarily work for a higher dimension. In 3D, bridges and subways can be included in the mesh representation.



(a) (b)

Figure 2.3: (a) Computed CDT on the example of the input data, (b) Found path on the mesh representation

2.3 Grid

The next possible environment representation is to cover the environment with a uniform grid, usually 2D. Suppose that an agent moves on a grid which can be represented with a matrix. There are two possible approaches how to use the grid for the path planning problem. Firstly, each grid vertex has integer coordinates and can be considered as a possible agent position. The movement is allowed along the grid edges. The second approach expects that every possible position of the agent is in the middle of each cell and the movement is allowed between two adjacent cells. Both these approaches take discrete steps in one of four directions (up, down, left, right), each of which increments or decrements one coordinate. There is also the option to use eight directions (four mentioned above and the diagonal directions) to move through grid environment.

The agent can move in all directions unless there is an obstacle in his surroundings. Then the problem can become more complex and interesting when some of the square tiles to represent obstacles will be shaded. The shaded tiles represents impassable obstacles that the agent has to avoid (Figure 2.4). In this case, every shaded cell has its corresponding vertex and associated edges were deleted from the state transition graph G. Very complicated labyrinths or maps can be constructed this way.

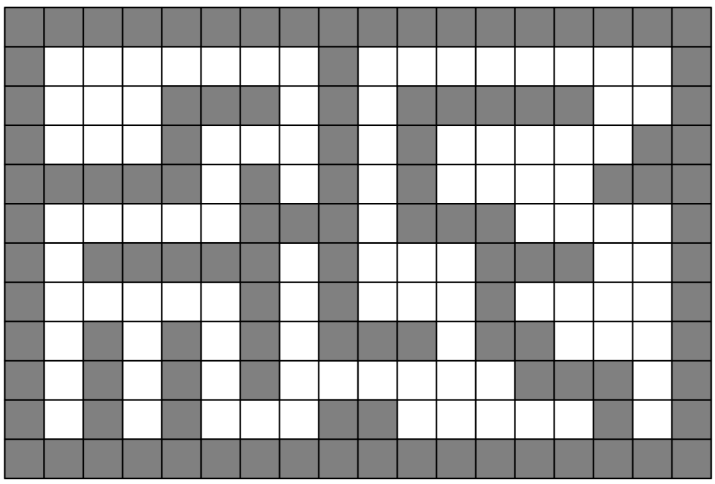


Figure 2.4: The example of the input data for the grid environment.

The grid environment can be easily converted to the graph environment for both mentioned approaches. For example, each cell represents a graph vertex and the adjacent cells are connected by edges of the graph G. Obviously, we get a planar graph G, where each vertex has the degree 4 (four

directions) or 8 (eight directions).

2.4 Configuration space

Path planning models may consider not only the position of the agent but also, for example, its orientation (if it makes sense) or other properties. The configurations are then described by vectors keeping the position of the agent in the space of these models. The simplest problem is to navigate a point agent in 2D because we have only a 2D vector that only describes its position. With any further information, the vector that represents the properties of the agent increases its number of components and the problem is more complicated. For example, for the agent in 3D space, the configuration could be, for example, a six dimensional vector describing its transformation consisting of vectors of the position and the orientation in the space. The term 'state' is also used for vectors that describe the configuration of the agents in the dynamic constraint models.

Chapter 3

Path Planning

First let us introduce necessary definitions.

Definition 3.1 (Path)**.** Path Q from the vertex s to the vertex g in the graph $G = (V, E)$ is every vertex sequence $(s = u_0, u_1, ..., u_n = g)$, where $u_i \in V$ for $i = 0, 1, ...n$ and $u_j u_{j+1} \in E$ for $j = 0, 1, ..., n-1$.

Definition 3.2 (Path weight)**.** The weight $w(Q)$ of the path Q is defined as $\sum_{i=0}^{n-1} w(e_i)$, where $e_i = u_i u_{i+1}$ and n is the number of edges of Q.

Definition 3.3 (Minimal path)**.** Minimal path Q_{min} from the vertex s to the vertex g in the graph $G = (V, E)$ is every path Q such that its weight $w(Q)$ is minimal.

The basic problem of path planning is to find the minimal path in a static known environment (i.e. the geometry of the environment is known and does not change during the calculation). This problem has been already satisfactorily algorithmically solved and therefore we focus on a more interesting issue where the environment can change dynamically (e.g. the formation of a new barrier or a new and cheaper path). This environment can be known in advance, that is, whenever there is a change of the edge weight in the graph G, the method will respond appropriately to each change. Another type of the environment may be the search in an unknown environment where the method responds to each change of the edge rating of the graph G only if it has already explored that part of the graph G.

The path planning algorithms in the static or dynamic environment can be divided into three groups. The first one are single source algorithms, which find all paths from the start vertex s_{start} to every other vertex of the graph G. The next single target algorithms find path from the start vertex s_{start} to the goal vertex s_{goal}. The last group are all-pair algorithms, which find path from each vertex to all other vertices.

There are currently lots of methods to find the minimal path, so we will only pick those that suit to our problem. We are looking for methods that are capable of searching for a path between two vertices in a dynamically changing environment that may be known, partially known or unknown. First, we only need methods that can be used to path planning in a dynamic environment. Therefore, the methods adapted to the edge weight changes in the graph G.

The important algorithms for our purpose are algorithms suitable for finding paths of agents as individuals. Each agent has its own independent (on other agents) path independently, thus we are avoiding algorithms specifically designed for the crowd modeling of agents. We could also demand methods that solve agents collisions with the environment, or methods to solve pedestrian collisions with other pedestrians but it is another path planning problem. In this thesis, however, we only deal with path planning on the graph G. Thus, we have selected almost every general algorithm for a global path planning on the graph environment suitable for the agent-based path planning without collisions. Although these are algorithms primarily developed for robot navigation, it can also suit for the crowd path planning. Therefore, we do not deal with other known algorithms such as search algorithms based on Voronoi diagrams [3] or probability planning [3] or even using Laplace equation to path planning [4].

3.1 Crowd models

At present, there are basically two types of crowd models, a continuous model and an agent-based model. Both models differ in their concept and design. At the same time, it is also possible to combine these two models together, although this is not a trivial matter.

Both models are also used in different path planning problems than agent navigation. They can also simulate not only the movement of humans, but also animals or microscopic organisms.

3.1.1 Continuum model

The continuum path planning model is suitable for dense crowds. It has been observed that the movement in a dense crowd is similar to the fluid flow [16], which can be described by Navier-Stokes equations [25]. An application to car traffic has also been reported [8]. Hughes [33, 34] developed a continuum model that describes a crowd as a continuous density field and introduced an evolving dynamic potential function. These partial differential equations

describe the crowd dynamics to guide the field optimally toward its goal. The continuous density field can be changed into a particle description of the crowd. The particle description with an added velocity-dependent term forms a model [81] which reproduces emerging phenomena of real crowds. The sophisticated continuum model [37] naturally solves inherent collision, produces smooth movement of the agents in a complex environments and the model was expanded from [81]. Moreover, the continuum model problems can be solved in parallel [52]. Although the continuum model is suitable for the dense crowds, the disadvantage is that the fluid simulation is ruled by the laws of physics. Therefore individual requirements of crowd members are neglected.

3.1.2 Agent-based model

The more natural for human beings is the second mentioned model where a path is planned for each agent separately. Every agent can have its individual requirements and the biggest advantage is that the agent-based model respects them. On the other hand the path planning for this model may be quite challenging in terms of time and memory complexity. Moreover, with an increasing number of agents this approach stops to be a real time and may become unsuitable.

Agent-based methods can be divided into local and global methods. The local methods are less diverse because they are mostly focused on the collision detection between agents themselves or between agents and obstacles. Methods of collision avoidance have been developed including grid-based rules [49] or Bayesian decision processes [54]. The parallel computation of the collision avoidance has also been proposed [26]. The smoothing of the planned path for autonomous vehicles [11] or robots [82] ranks among local approaches.

The global methods which help to locate possible evacuation-critical spots in buildings and simulate an emergency scenario belong to fire-escape algorithms [59, 19]. Algorithms [5, 67] similar to the fire-escape algorithms have been proposed but with a focus on the bottlenecks. Also human behaviour have been examined by many interesting proposed approaches, e.g. [62, 27].

The graph representation of the environment is often used for the global agent-based path planning. The most widespread A* algorithm [29], which uses a heuristics to speed up the planning, is proposed for a static environment as well as the basic algorithms – Breadth First Search (BFS), Depth First Search (DFS), Dijkstra's algorithm or Floyd-Warshall algorithm [23, 84], which rank among all-pair algorithms. The minimal cost path for all pairs of vertices is found in the memory complexity $O(|V|^2)$ and time complexity $O(|V|^3)$ in the worst case for a graph with $|V|$ vertices. In the

case of dynamic graphs, where weights of vertices and edges may change over time, the D* algorithm [71] and its improvement D* Focused [72] are more suitable. The D* Lite [43], which modifies the backwards algorithm LPA* [44], may yield even better results than D* Focused. The memory complexity $O(k(|V| + |E|))$ for k agents, $|V|$ vertices and $|E|$ edges can be easily reached by these algorithms because each agent needs its own graph ranking. Hierarchical Annotated A* [28] creates a hierarchical graph to find an almost optimal path and Anytime D* [48] finds a sub-optimal path in a limited time. The path planning for a moving target solves Moving Target D*-Lite [76] or Generalized Fringe-Retrieving A* [75].

In addition, we chose agent-based model for our research. Although this approach has higher computational demands, every individual in the crowd has their own path as was mentioned earlier. We try to reduce these demands by using the similarity of some paths and we use the graph representation as the environment, because this representation is the most frequent and many real data are freely available [60]. In addition, as mentioned in Section 2.1, all possible types of representations can be represented as a graph. The advantage of graph algorithms for path planning is that the algorithm with minor modifications can be used for all other environment representations. We can define the problem as follows

Input 1. *A set of agents $P = \{p_1, p_2, \ldots, p_c\}$, where every agent $p_i \in P$ has a start position $s(p_i) \in \mathbb{R}^2$ and a goal position $e(p_i) \in \mathbb{R}^2$.*

Output 1. *A set of paths $W = \{w_1, w_2, \ldots, w_c\}$, where every path w_i belongs to the agent p_i for each $i = 1, 2, \ldots, c$.*

Obviously $|P| \geq |W|$ because each agent has only one path or none. Path does not exist when the graph G has two or more components and the start position $s(p_i) \in \mathbb{R}^2$ and the goal position $e(p_i) \in \mathbb{R}^2$ do not belong to the same graph component.

3.2 Static graph algorithms

The static path planning graph algorithms are not necessary to introduce because they are already widespread. On the other hand, their ideas are constantly used as the basis for the newer and more complex algorithms. Therefore, we recall the most important path planning graph algorithms for the static environment.

3.2.1 Breadth first search

The breath-first search (BFS) is one of the basic path planning strategies, a basis of other and more complex path planning algorithms. The main idea of this algorithm is as follows: First it finds all vertices reachable from the start vertex s_{start} through one edge $e(s_{start}, s)$. Next it finds all vertices that are reachable from the vertex s_{start} through two edges. Generally all vertices in the k distance (k edges far) from the start vertex s_{start} are discovered sooner than the vertices in $k + 1$ distance.

The asymptotic complexity of BFS is $\mathcal{O}(|V| + |E|)$, where $|V|$ is the number of discovered vertices and $|E|$ is the number of discovered edges.

3.2.2 Depth first search

The second basic path planning strategy, the depth first search (DFS), uses backtracking to search through the graph G. Each step DFS expands the first following vertex, if it has not been visited yet. The order of selection of the next vertex depends on the graph representation and/or on the implementation. When the algorithm reaches the vertex, which has no following vertices or all of them were already visited, it goes back by backtracking. Depth first search uses the stack abstract data type and it has the same asymptotic complexity as BFS, i.e. $\mathcal{O}(|V| + |E|)$, where $|V|$ is the number of discovered vertices and $|E|$ is the number of discovered edges.

3.2.3 Dijkstra

Dijkstra's algorithm, proposed in 1959 by E.W.Dijkstra [17], is one of the most widely used path planning algorithms for the graph representation because of its simplicity and reliability. In fact, this algorithm is a generalization of the BFS algorithm, which does not use the number of edges but the distance (e.g. the Euclidean distance) from the start vertex s_{start}. Let us assume that each vertex of the graph is a city and its edges represent the distances between the cities, which are connected by a road. The Dijkstra's algorithm finds the minimal path from one city to another.

The asymptotic complexity for the easiest implementation solution is $\mathcal{O}(|V|^2)$, where $|V|$ is the number of discovered vertices. The use of an appropriate queue representation for the sparse graphs, where $|E| \ll |V|^2$, will improve the complexity to $\mathcal{O}(|E| + |V| \log |V|)$.

3.2.4 Floyd-Warshall

Floyd-Warshall algorithm (FW) is an algorithm to find the minimal path in the weighted graph G and it ranks among into the all-pair minimal problem algorithms. FW also accepts the negative edge rating and it detects a negatively weighted cycle (if there is some). If there is no negatively weighted cycle, the Floyd-Warshall algorithm finds the minimal path between each pair of vertices.

The asymptotic complexity of the FW algorithm is $\mathcal{O}(|V|^3)$, where $|V|$ is the number of vetrices.

3.2.5 A* search

The A* search algorithm (pronounced as "A star") is widely used in path planning to find the minimal path between multiple vertices of the graph G with time independent edge weights. The algorithm was firstly introduced by Peter Hart, Nils Nilsson and Bertram Raphael in 1968 [29]. Currently it is widely used due to its performance and accuracy. The A* search, from the computation time view, performs better than Dijkstra's algorithm.

As the A* search algorithm (see Alg.1) goes through the graph G, it focuses on the path with the lowest known edge weights (line 4), while the alternative parts of this path discovered during the calculation are stored in the priority queue. If the currently searched path has a higher cost at any time during the computation than another alternative path in the priority queue *open*, the method leaves the higher cost path and returns to the alternative path with the lowest cost. This process is repeated until the goal is reached.

The A* search uses a heuristic function $f(s)$ to determine the order in which the vertices of the graph G will be visited (line 10). This heuristics is a sum of two functions. The first function is the cost of the path $g(s)$ from the start vertex $s_{start} \in G$ to the currently visited vertex $s_{current} \in G$. The second function is the so-called heuristic estimation $h(s)$ from any vertex $s \in G$ to the goal vertex $s_{goal} \in G$. Additionally, it has to be an admissible heuristic estimation, so it must not overestimate the distance between any vertex $s \in G$ and the goal vertex $s_{goal} \in G$.

If heuristics for all vertices $s, u \in G$, which are connected with the edge $e(s,u) \in G$, satisfy the additional condition $h(s) \leq d(s,u) + h(u)$, then the function $h(s)$ is consistent. The function $d(s,u)$ is a metric function. When the additional condition is satisfied, the A* search algorithm can be implemented more efficiently. More specifically, the method can be modified so that no vertex of the graph G will be processed more than once.

Algorithm 1: A* search algorithm

Input : Vertex s_{start}, vertex s_{goal}
Output: A path T from vertex s_{start} to vertex *end*

```
1  open ← s_start ;                    // insert s_start into priority queue
2  closed ← ∅ ;                  // initialize list of processed vertices
3  while open is not empty do
4  |  s_current ← open.top() ; // pop vertex from the priority queue
5  |  if s_current = s_goal then reconstruct T ;
6  |  add s_current to closed from open;
7  |  foreach neighbor s_neigh of current do
8  |  |  if s_neigh is not in closed then
9  |  |  |  temp ← f(s_neigh) ;        // compute heuristic function
10 |  |  |  if s_neigh is not in open then
11 |  |  |  |  s_neigh.f ← temp ;                  // change the value
12 |  |  |  |  add s_neigh to T and to open;
13 |  |  |  else
14 |  |  |  |  if temp < s_neigh.f then
15 |  |  |  |  |  s_neigh.f ← temp;
16 |  |  |  |  |  change T;
17 reconstruct T;
18 return T;
```

The time complexity of the A* algorithm depends on the used heuristic $h(s)$. In the worst case it is exponential, i.e. the number of visited vertices grows exponentially depending on the size of the found solution.

Suppose we look for a path between the initial vertex $s_{start} \in G$ and a goal vertex $s_{goal} \in G$, where the searched graph G is a tree. In this case, polynomial complexity can be achieved if a heuristic function returning the exact value of the distance between any vertex $s \in G$ and the goal vertex $s_{goal} \in G$ meets the condition

$$|h(s) - h\star(s)| = O(\log h\star(s)) \tag{3.1}$$

where $h\star(s)$ is the optimal heuristics. In other words, the relation 3.1 states that the error of the heuristics $h(s)$ will not grow faster than the logarithm of the "perfect heuristics" $h\star(s)$, which returns the exact and minimal possible distance from the vertex $s \in G$ to the vertex $s_{goal} \in G$.

3.2.6 Best-first search

The Best-first search algorithm uses only heuristics to select other vertices to be processed. The natural implementation uses the priority queue to store the most promising vertices for the computation. In essence, this is an A* algorithm that does not use the current cost $g(s)$ of the path at all.

3.3 Dynamic graph algorithms

The group of dynamic algorithms is slowly getting into the greater awareness of the scientific community. The simplest and fastest (from the implementing point of view) solution can be to use the static algorithm for the path recomputation. If a very small number of graph changes appear in the graph G, it can be considered as the correct approach. However, with a large number of graph changes, it is better to use a specialized algorithm. These algorithms are more memory-intensive, but computationally faster than static path planning algorithms.

3.3.1 Original D* (Dynamic A*)(1994)

The first and original D* algorithm [71] was introduced by Anthony Stentz in 1994. Its name (pronounced as "D star") comes from the term "Dynamic A*". This term was used because the D* algorithm is a generalization of the static A* algorithm for the dynamic environment (change of the edge weight, insertion of the new vertex, etc.).

Similar to the A* algorithm the D* algorithm also uses a priority queue Q, to which we insert all vertices, which have to be processed or which have been changed. Each vertex $s \in G$ might have several states $\tau(s)$. The first state is $\tau(s) = NEW$. The meaning of this state is that the vertex s has been never processed in the priority queue. The second state $\tau(s) = OPEN$ informs that the vertex s is still in the priority queue. The last state is marked as $\tau(s) = CLOSED$ and it means that the vertex has been already processed (removed from the priority queue).

The D* algorithm in its initial computation on the graph G works in the same way as the A* algorithm, i.e. it finds the minimal path in the graph G from the start vertex s_{start} to the goal vertex s_{goal} quickly. When a graph change occurs, D* is able to find a new minimal path from the current position of $s_{current}$ to the goal vertex s_{goal} much faster than the A* algorithm that would run repeatedly. However, the D* algorithm has the reputation of a complex algorithm and was pushed out by a much simpler D* Lite algorithm.

3.3.2 Focused D* (1995)

The Focused D* algorithm [72], as the name suggests, is an extension of the original D* algorithm that was introduced in 1995 by the same author as D* algorithm, Anthony Stentz. In contrast to the original version, heuristics are used to focus the algorithm on the most important vertices. This reduces the total number of processed tops of the graph G and accelerates the calculation of the minimal path.

Similarly to the original D* algorithm, the Focused D* uses a priority queue Q, to which vertices for processing are added. All of the graph vertices can come to the same states $\tau(s)$ as in the D* algorithm - NEW, $OPEN$, $CLOSED$. Moreover, the two other states $LOWER$ and $RAISE$ can be assigned to the elements in the priority queue Q.

The basic structure of the algorithm is identical to the original D* algorithm, so we focus only on the spot where these two methods differ. The difference is the key by which the elements in the priority queue are ranked. Previously, it was enough to sort the elements according to their minimal rank for which they can be accessed. We now use the key $f(s_{current}, s)$, which is the sum of two functions

$$f(s_{current}, s) = c(s_{goal}, s) + h(s_{current}, s),$$

where $c(s_{goal}, s)$ is the path cost from the goal vertex s_{goal} to the vertex s and the function $h(s_{current}, s)$ is a heuristic estimate of the distance between the current robot position $s_{current}$ and the vertex s. Thus, the function $f(s_{current}, s)$ is no more than an estimate of the distance between the current vertex $s_{current}$ and the goal vertex s_{goal}.

The addition of the heuristics to the D* algorithm is the main difference between the original D* algorithm and the Focused D*. A newer version of the D* algorithm goes, thanks to the heuristics, through a much smaller number of nodes. The Focused D* still contains minor differences from the original D* algorithm, which are not so significant, and can be found in the article [72].

3.3.3 DynamicSWSF-FP (1996)

DynamicSWSF-FP [63], as originally proposed, finds a path from the goal vertex $s_{goal} \in G$ to the start vertex $s_{start} \in G$ and thus maintains estimates of the goal distances rather than the start distances. The algorithm stores the distance between each vertex $s \in G$ and the goal vertex $s_{goal} \in G$. On the other hand, it is simple to switch the path planning direction to find the path

from the start vertex $s_{start} \in G$ to the goal vertex $s_{goal} \in G$. Furthermore, DynamicSWSFFP recomputes all goal distances that have changed.

The DynamicSWSF-FP uses the breadth-first search (BFS) as its basis, thus both algorithms compute the same start distances during the path planning. Unlike BFS, the DynamicSWSF-FP algorithm is much more complex and reduces the number of searched vertices. Moreover it is able to quickly react to the graph change and recompute the found path. The incremental search algorithm (DynamicSWSF-FP) is not very widespread because it was overshadowed by another algorithm (Section 3.3.4) and its follower (Section 3.3.5). However, it is still a useful algorithm when we need to find the path from several vertices to one goal vertex after each graph change.

3.3.4 LPA*/Incremental A* (2001)

The Lifelong Planning A* algorithm (LPA*) [44], is an incremental search method with a heuristics to find a path between the two specified vertices of the graph G whose edge rating changes in time t. The incremental search consists of recalculating a so-called initial distance (i.e., the distance between the start vertex $s_{start} \in G$ and any vertex $s \in G$) for each vertex that has changed or has not been calculated. Heuristic searches are only used to recalculate the initial distances that are important for the recomputation of the minimal path from the start vertex $s_{start} \in G$ to the goal vertex $s_{goal} \in G$. Thanks to incremental heuristics, the LPA* algorithm recalculates a very small percentage of the initial distances.

Lifelong Planning A* can be used to path planning in the finite and known graph G, whose edge rating increases or decreases over time t. In this case, the algorithm is always able to find the minimal path between the start vertex $s_{start} \in G$ and the goal vertex $s_{goal} \in G$.

LPA* uses two variables in each vertex $s \in G$ to calculate the initial distance. The first one is the so-called g-value $g(s)$ and the second is the so-called rhs-value $rhs(s)$. The rhs-value of the vertex $s \in G$ is based on the g-values of the neigboring vertices of the vertex s. The vertex $s \in G$ is called locally consistent if $g(s) = rhs(s)$. Otherwise it is called locally inconsistent. If all nodes $s \in G$ are locally consistent, then their g-values are equal to the initial distances. Since it is not necessary for all the vertices of the graph G to be locally consistent to find the minimal path, heuristic $h(s, s_{goal})$ is introduced. This helps the LPA* algorithm focus only on the relevant g-values that are important for finding the minimal path between the start vertex $s_{start} \in G$ and the goal vertex $s_{goal} \in G$. The heuristics are identical to the heuristics used in the A* search algorithm (recall Section 3.2.5).

The priority queue Q contains only the vertices of he graph G that are

locally inconsistent, and the method tries to fix them. Inconsistent vertices $s \in Q$ in the priority queue Q are sorted by the special key $k(s)$. This key is nothing but a two-component vector. The elements of the priority queue Q are lexicographically sorted. This means that we first compare the first component of the vectors $k(s), k(u)$ for $s, u \in Q, s \neq u$. If the first component is identical, we compare the second component of the vectors $k(s), k(u)$ for $s, u \in Q, s \neq u$.

3.3.5 D* Lite (2002)

The D* Lite [43] algorithm uses the LPA* algorithm (Section 3.3.4) to mimic the behavior of the original D* algorithm (Section 3.3.1), which is based on the A* algorithm. It uses LPA* to find the minimal path in the dynamic graph G for an object that moves along the initial minimal path. D* Lite is considered to be a simpler algorithm than D*. D* Lite is at worst as fast as D* algorithm.

In Section 3.3.4, an LPA * algorithm has been described that repeatedly finds the minimal path between the start vertex $s_{start} \in G$ and the goal vertex $s_{goal} \in G$ in case of any graph change (edge change, deletion of the vertex $s \in G$, etc.). The difference against D* is that D* Lite repeatedly finds a path from the current vertex $s_{current} \in G$ to the goal vertex $s_{goal} \in G$ each time the graph G changes while the agent moves along the found path toward its goal vertex $s_{goal} \in G$. The method does not need any prerequisites for the change rate of the edge of the graph G, its size, or how far away from the current vertex $s_{current} \in G$ the change appears.

The LPA* algorithm can be used to navigate the agent into the goal position in an unknown environment. As already mentioned, LPA* (Section 3.3.4) searches for the path between the start vertex $s_{start} \in G$ and the goal vertex $s_{goal} \in G$. Their g-values are calculated as the distance from the start vertex $s_{start} \in G$, but for D* Lite we need to change the search direction so that the g-values are counted from the goal vertex $s_{goal} \in G$. In a non-oriented graph, it is only necessary to swap the start vertex with the goal. In addition, the direction of the oriented edges in the oriented graph is to be changed.

The demands on the heuristics are the same as in the A* and LPA* algorithms. If the initial computation of the algorithm returns g-value in the start vertex as $g(s_{start}) = \infty$, the path from the vertex s_{start} to the vertex s_{goal} does not exist. Otherwise, it is possible to track the minimal path from the start vertex s_{start} to the goal vertex s_{goal}. We reach the goal by moving from the current vertex $s_{current}$ to the neighboring node s, which minimizes $d(s_{current}, s) + g(s)$. We repeat this process until we reach the goal s_{goal}.

To solve the problem of reaching the goal vertex in the unknown environment, it is necessary to modify the LPA* algorithm, although most of its functions and methods do not change. The main method have to be modified to correctly calculate all elements in the priority queue Q whenever the agent moves. The heuristics changes with the agent movement because it is always counted with respect to the current position of the agent $s_{current}$, and this is why this part is very important. This modification only changes the keys of all vertices $s \in Q$, but does not affect the local consistency of the vertices and thus does not increase (or reduce) the size of the priority queue Q.

The algorithm can also be used for the path planning in the unknown environment where the graph G would have the shape of a network. Every vertex $s \in G$ would have just eight adjacent vertices. The price of edges $(s, s_{neigh}) \in G$ would be initially set to the value one, but if the agent found an inaccessible spot, then the edge price would change to infinity.

3.4 Any-angle movement

The next group of algorithms removes the constraints of movement directions in the selected environment representation. The following algorithms use the grid representation, which is most suitable for this problem and allows us to move standardly in 4-directions or 8-directions. This representation may seem to be strict, but the opposite is true. Algorithms do not need only to consider direct neighbor cells, but the cells of adjacent cells can be used or any other cell in the whole environment representation. Note that the grid can be easily converted to the graph representation, and these algorithms can also be used in graph environments. On the other hand, algorithms conversion to the graph representation would be a little more complicated.

3.4.1 Field D* (2007)

The Field D* algorithm [21] is a method based on an interpolation path search and its interpolation recomputation. This method extends D* and D* Lite algorithms by using linear interpolation to efficiently calculate low cost paths. Due to linear interpolation, these paths are optimal and very effective in practice. At present, the Field D* algorithm is widely used in robotics for large environments.

Definition 3.4 (Planar graph)**.** A graph is planar if it can be drawn on the plane such that no two edges cross.

The problem solved by the Field D* algorithm can be formulated as follows. Given a region in the plane partitioned into a uniform grid of square

cells T, an assignment of traversal costs $c : T \to (0, \infty]$ to each cell, and two points s_{start} and s_{goal} within the grid, find the path within the grid from s_{start} to s_{goal} with minimum cost.

Such a problem is important to approximate appropriately. A very often used approach is to represent the uniform grid as an undirected planar graph G, where each vertex $s \in G$ has exactly eight neighboring vertices. The path through the graph G is then found. Generally, the vertex is assigned to each cell center, and the edge between two vertices exists only when the two cells are adjacent. The cost of such an edge corresponds to the metric between the two adjacent vertices.

The current methods, which are looking for the minimal path for this problem, do not provide exact solutions, as can be seen in Figure 3.1. The methods for this problem without obstacles provide the green path from the start vertex (green) to the goal vertex (black). On the other hand the Field D* method uses better approach, which finds a shorter path (Figure 3.1, blue).

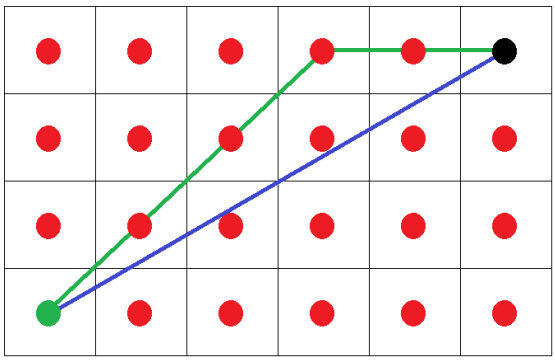


Figure 3.1: The difference in the way found and the minimal way.

The Field D* algorithm is an extension of already described D* algorithms. It uses linear interpolation to calculate optimal paths. This method finds more direct and cheaper paths than traditional cellular search algorithms without loosing a real-time computation.

The algorithm, Field D*, according to the performed tests [21], provides better (cheaper) paths than the D* Lite algorithm in the cell planning environment. On the other hand, the cheaper found path is balanced by longer calculation times. However, the authors suggest that Field D* may not always find a cheaper path than other cellular planning algorithms, although this is a very rare case.

Let us note that the idea that Field D* comes up with can be easily added to all cell-based dynamic D * algorithms. Currently there is a modified

version of this algorithm that improves this deficiency in the form of a longer time calculation.

3.4.2 Theta* (2007)

Theta* [15] is a modified A* algorithm, which produces better paths than the Field D* algorithm (Section 3.4.1). The path planning method Theta* propagates information along grid edges without constraining the paths to grid edges. It is fast, simple and finds short and realistic looking paths in the grid environment. Theta* considers paths that are not constrained to grid edges during the path planning and can thus make more informed decisions during the planning. Theta* finds shorter paths for the path planning problems faster than the Field D* algorithm, it is more general than Theta*. The Field D* can be applied to grids whose cells have different sizes and traversal costs.

The Theta* is basically a modified A* algorithm where the main difference between Theta* and A* is that Theta* allows the parent of a vertex to be any vertex, unlike A* where the parent must be a successor. Its biggest disadvantage is that it is based on A* algorithm, which results in the slow recomputation of the path. However, authors have modified and improved this algorithm to the algorithm Lazy Theta* [57].

What we did not mention above is that the Theta* combines the grid representation of the environment and the visibility graph. The algorithm Theta* searches for the path on the grid, but at the same time it tries to create and move along the visible graph. For this reason, a line-of-sight check between a vertex $s \in G$ and its parent has to be processed. When the vertex $s \in G$ is never expanded, it is wasted computation. A large number of these line-of-sight checks is just the deficiency that is subsequently solved by the algorithm Lazy Theta*.

3.4.3 Incremental Phi* (2009)

Incremental Phi* algorithm [56] combines the advantages of the previous two algorithms. This is the version of the Theta* algorithm (Section 3.4.2), which is incremental. This allows a quick recomputation of the path. The authors created Phi* and Incremental Phi* algorithms and both of them are complete and correct. The Incremental Phi* is simple and it finds paths of essentially the same length as the Theta*. In comparison with basic Theta* algorithm it can also provide a speedup of approximately one order of magnitude for the path planning with the freespace assumption. The path planning with the

freespace assumption is when the agent can move from a given start position to a given goal position without knowing the blockage status a priori.

3.5 Moving goal position

This section summarizes the algorithms used to find paths in an environment where the end position is not static but it moves. These algorithms can be used for any representation of the environment because all of them are based on the algorithms mentioned above, namely the A* search algorithm and D* Lite algorithm.

GAA* (2008)

GAA* (Generalized Adaptive A*) [74] is a modified A* algorithm (Section 3.2.5) that finds the shortest paths in state spaces where the action costs can increase or decrease over time. It handles the path planning of the moving goal s_{goal}. This is a generalization of an older algorithm called "Adaptive A*". Adaptive A* is an incremental heuristic search algorithm that solves series of similar search problems faster than A* because it updates the h-values (values of heuristic) using information from previous searches. It is not guaranteed to find shortest paths in state spaces with the older Adaptive A*. However, GAA* finds shortest paths in state spaces where the action costs can increase or decrease over time and it also outperforms both breadth-first search, A* and D* Lite for moving-target search and works only on the graphs in two dimensions.

G-RFA* (2010)

G-FRA* (Generalized Fringe-Retrieving A*) [75] is a generalization of the previous GAA* algorithm (Section 3.5) for any graphs that are not limited only to 2D. The algorithm uses a technique from another incremental search algorithm FRA* (Fringe-Retrieving A*), which is the fastest algorithm to solve moving target search problems only on two dimensional grids. G-FRA* can also be up to one order of magnitude faster than GAA*. Moreover G-FRA* reuses search trees from previous searches to speed up the current search and thus often find cost-minimal paths for series of similar search problems faster than by solving each search problem from scratch.

MTD*-Lite (2010)

MTD*-Lite (Moving Target D*-Lite) [76] is basically the extension of the D* Lite (Section 3.3.5) algorithm that uses techniques of the G-RFA* algorithm (Section 3.5) to recompute path for the moving goal s_{goal}. Although the D*Lite algorithm is very slow on moving target search problems, where both the start s_{start} and goal s_{goal} vertices can change over time, Moving Target D* Lite is four to five times faster than Generalized Adaptive A*.

Tree-AA* (2011)

Path planning algorithm [31] for an unknown environment based on the A* algorithm (Section 3.2.5). Tree-AA* is an incremental heuristic search algorithm with the freespace assumption and use A* searches to find a minimum-cost path from the current state of the agent to the goal state. Tree-AA* generalizes Path-Adaptive A* (Path-AA*) [30] to reuse the minimal paths of the current and all previous A* searches and thus creates a reusable tree.

Path-Adaptive A* [30] solves navigation problems in initially unknown terrain using planning with the freespace assumption. The agent repeatedly finds and then follows a cost-minimal path from its current state to the goal state. If the agent senses that the cost of at least one action on the path increased while it follows the path, then it repeats the process.

3.6 Fast suboptimal algorithms

As the name itself suggests, fast suboptimal algorithms are looking for a path faster than the other algorithms mentioned above. This is possible after accepting some limitations (such as finding a path within 5 seconds) or rather simplifying the problem. All the algorithms in this section will quickly find some path they claim to be suboptimal and then try to improve it.

Anytime D* (2005)

This is the "Anytime" variant [48] of the D* Lite algorithm (Section 3.3.5), which was created by combining the D* Lite algorithm with the Anytime Repairing A* algorithm. The "Anytime" algorithm is an algorithm that can run under any time constraint. It finds very quickly a suboptimal (in the sense of a short time calculation) path with which it continues to work. This path then improves with time. The more time the algorithm has, the better the found path will be.

HPA* (2004)

HPA* (Hierarchical Path-Finding A*) [7] is an algorithm for the path planning of the large number of objects to move along the large graph. For example, navigating armies in RTS computer games. Objects may have different start positions s_{starts} and potentially different goal positions s_{goals}. HPA* creates a graph hierarchy to quickly find an almost optimal path. The algorithm should find the way for all objects faster than A* for each of them separately.

PRA* (2005)

PRA* (Partial Refinement A*) [73] should address the same problem as the HPA* algorithm (Section 3.6). The difference is in the implementation. Both algorithms should have similar computational requirements. It is not only able to cleanly interleave planning and execution, but it is also able to do so with only minimal losses of optimality.

HAA* (2008)

HAA* (Hierarchical Annotated A*) [28] is a generalization of the HPA* algorithm (Section 3.6), which allows for varying movement of objects around the terrain (e.g. narrow paths where only small objects can pass). HAA* is able to find near-optimal solutions to problems in a wide range of environments yet still maintain exponentially lower search effort over standard A*.

3.7 Others

The last section contains the remaining algorithms that are used for the path planning in the graph G but cannot be classified into the above sections.

SetA* (2002)

This is a next variant of the A* algorithm [36] that searches the binary decision diagram, which is the model of the graph. The algorithm should be faster than A* in some cases. However, it appears that SetA* is only faster in the case of a high degree of graph vertices (graphs similar to full graphs).

LEARCH (2009)

LEARCH [65] (LEArning to seaRCH) is a combination of machine learning algorithms that are used to teach the robot to look for an almost optimal path itself. Combined with the Field D* algorithm (Section 3.4.1), it provides better results.

Chapter 4

Motion Planning

In the previous chapter, we discussed a great part of existing methods for path planning mainly in the graph environment. In parallel with path planning there is another important and very complex problem of finding paths – motion planning. In this case, we only have some space that describes a collision free area and the area with the obstacles description. This problem is more complex and desirable, because it is necessary to process this free space in an appropriate way so that the agent can move though the environment without any collision.

This problem gained attention with robots and their control. The robot should get from the start position s_{start} to the goal position s_{goal} without collision with any obstacle. Usually it does not know the environment in advance, so it had to scan its surroundings and make its decision. It is obvious, therefore, that this is not a simple problem and it brings a number of pitfalls. After working with the robots, autonomous vehicles have been added to research, and nowadays motion planning of drones has the greatest attention.

However, the motion planning solution can also be applied to other areas than to the navigation of the mechanical objects. A very important problem is, for example, navigating the ligand (or probe) into the protein or out of the protein. This problem is in the simplest case representable by 3D c-space if the ligand has a zero size or looks like sphere only. In the case of a more complex ligand, space dimension increases, as both 3D position and 3D orientation need to be considered. Flexible ligands can be represented as articulated mechanisms with relative motion of atoms. The atoms may be constrained due to bond torsions [13] [80] which leads to search in the high-dimensional configuration space.

The representation of the high-dimensional configuration space using a regular grid is not computationally viable, although the methods for the

grid motion planning exist. The solution used in most cases are randomized sampling-based planners [39] [47] which (probably) compute at least one result very quickly with high probability. The idea is to sample the configuration space randomly and to classify the samples as free (without collision) or non-free (collision). The non-free samples are discarded and the free samples are stored in the graph (roadmap). The graph approximates the free regions of the configuration space and enables to search a path with graph-based path planning methods. The tunnel detection of flexible ligands can be processed with the above mentioned sampling-based methods. Moreover, in dynamic scenes time can be taken as another dimension [32].

The widely used sampling-based methods are Probabilistic Roadmaps (PRM) [39] and Rapidly Exploring Random Tree (RRT) [47].

4.1 Protein models

Biomolecules (proteins) [53] are chemical compounds found in living organisms. Proteins consist of the basic building blocks such as proteinogenic amino acids, molecules consisting of few atoms (t̃wenty species based on the definition). Each amino acid contains a portion of the peptide bond that makes it possible to form an amino acid sequence. Proteins consist of one or more of these chains. The sequence in the chain is called the primary structure of the protein. This sequence is encoded in DNA and amino acid chains are synthesized by ribosomes in transcription and translation.

In the areas of chemistry that deal with drug development and molecular enzymes, it is important to analyze these biomolecules. During the analysis of biomolecules, it is necessary to describe the spatial relationships between individual atoms to accelerate molecular surface computation, volume computation, internal cavity analysis, tunnel search in a molecule, etc. Tunnels in molecules of the protein determine their essential properties, such as the induction of a chemical reaction or the complexity of the modification of the molecule. The tunnel analysis is used in the protein engineering, where it helps to better understand specific chemical reactions. These reactions are affected by modifying the structure of the protein by expanding or closing the tunnel at a particular site, thereby facilitating or disabling the reaction.

The Van der Waals model [6] captures the structure of proteins along with interacting forces between atoms and allows the study of the protein behavior. In this model, atoms are represented by spheres. If there is a strong bond between the atoms, their spheres overlap. In other cases, the spheres should not overlap because of the repulsive forces. The Van der Waals model example is illustrated in Figure 4.1. The spherical radii, the Van

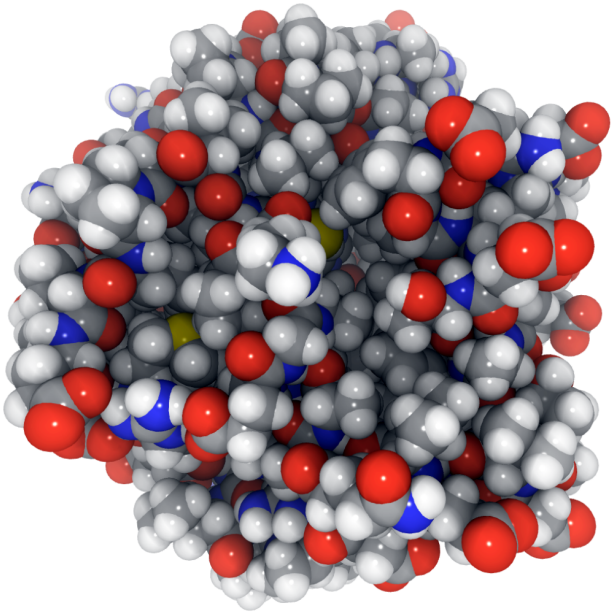


Figure 4.1: An example of the Van Der Waals representation of the molecule [53]

der Waals model constants, are determined by experimental measurement. Model spheres are also usually dyed according to the chemical element to which they belong.

4.2 Probabilistic Roadmaps

The PRM algorithm [24] [41] builds the graph over the explored parts of the configuration space. This approach has two phases. First the random samples are generated and tested for collisions. The second phase tries to connect the close samples with the edge if possible. A precise implementation of random sampling, as well as candidate selection procedures for interconnection and the configuration of the local planning, may vary and is therefore left to the programmer. The possibilities of implementing these procedures are stated in [24] which also compares these procedures in detail.

A sufficient input for the PRM algorithm is a set of obstacles. The knowledge of the start and the goal configuration is not required by the algorithm itself, but their knowledge can be used in some sampling heuristics. The algorithm output is an oriented graph mapping the examined configuration space. The algorithm works with the time complexity $\mathcal{O}(n \log n)$ [40], where n is the number of samples. The graph path planning strategy (Chapter 3)

is always used on this output graph G.

There are two problems in the PRM algorithm. The first one is called boundary value problem. It rises up when connecting the two given configurations. This problem can be difficult to solve under motion constraints, so PRM is primarily used in motion planning without motion constraints. The second problem is called the narrow passage problem. Narrow passages are small regions of configuration space whose removal changes the connectivity of the space. These small regions have a very low probability of coverage by sampling methods. Many iterations are needed to sample them densely enough (or sample them at least once) because of their small volumes. This problem can be solved (or at least approximated) by generating random samples close to obstacles or around medial axis of the environment [45]. However, this is particularly suitable only for low-dimensional configuration spaces [45].

Probabilistic Roadmaps can be used in the computational biology to sample the conformational space in protein folding [2] [58] [80] in order to speed up molecular dynamics simulations.

4.2.1 sPRM

sPRM [40] is a simplified version of the Probabilistic Roadmaps algorithm. Rather than for practical use it is used for the analysis of follow-up algorithms. The time complexity of this algorithm, which is worse than in the case of the standard PRM algorithm, is $\mathcal{O}(n^2)$, where n is the number of samples. On the other hand, unlike previous methods the sPRM finds the path asymptotically optimal [38].

In this algorithm, all generated samples are considered as useful, so they are all inserted into a set of samples. Subsequently, every sample is processed. The algorithm tries to connect each sample with each sample, thus the $\mathcal{O}(n^2)$ complexity. When collision-free connection between two samples is found, their interconnection is inserted into the graph as the graph edge.

4.2.2 PRM*

PRM* [40] is another possible variant of the group of the PRM algorithms. This is an algorithm based on sPRM with the only difference that potential samples for interconnection are selected from the neighborhood with radii $r > 0$.

The goal of this constraint is to reduce the number of attempts to connect two samples to the average $\mathcal{O}(\log n)$. This leads to the time complexity

$\mathcal{O}(n \log n)$, where n is the number of samples. Similarly to sPRM, this algorithm finds the asymptotically optimal path.

4.3 Rapidly-exploring Random Tree

The Rapidly-exploring Random Tree (RRT) is the second and newer of the described probability algorithms. RRT has been designed for use in models with a number of complex physical constraints. Emphasis on the complex physical constraints is enhanced by the use of the agent control inputs as the ranking of the tree edges. Compared to the PRM, the basic version of the algorithm is easier in a few ways. RRT generates a tree instead of the graph, which simplifies the path planning part. In addition, instead of the nearest n vertices, a single candidate for interconnection is found.

The RRT planner represents the graph (roadmap) as a tree rooted at the starting configuration. Next it incrementally grows towards unexplored regions of the configuration space. Similarly to PRM, a sufficient input for the RRT algorithm is a set of obstacles. In addition, it also needs the start configuration. The knowledge of the goal configuration is not required by the algorithm itself, but it may be used in some variants of the RRT algorithm, e.g. the RRT with heuristics. The algorithm output is a tree mapping the examined configuration space.

The basic idea of the algorithm is to link the new randomly generated configurations to the nearest vertex of the tree. The motion constraints are checked during the tree expansion and new configurations are added to the tree if they satisfy the constraints. Therefore, the RRT algorithm is an appropriate planner for flexible ligand [14]. The ability to generate new configurations greatly affects the performance of the RRT. The high-dimensional space problem can be time consuming and the ML-RRT copes with this problem [22]. The method was further extended for flexible ligands [14] and used in several studies [46]. Moreover, the resulting tree can be projected back to 3D space from the high-dimensional space [12]. This projection allows us to visualize and further explore the problem.

The RRT algorithm time complexity is the same as the time complexity of the PRM algorithm – $\mathcal{O}(n \log n)$, where n is the number of samples. It also suffers from the narrow passage problem as the PRM algorithms. Biased sampling, e.g., [45], that work for the PRM, is not suitable for RRT-based planners, as the tree can be stuck due to obstacles. Guiding the tree along a precomputed path by geometry-based methods in the protein [83] is a possible solution for the narrow passage problem.

4.3.1 RRT*

This algorithm is based on a standard RRT algorithm. Like in PRM*, with the increasing number of samples it finds the optimal path and its time complexity is the same - $\mathcal{O}(n \log n)$, where n is the number of samples. The potential samples for interconnection are also selected from the neighborhood with radii $r > 0$ as in the PRM*. The difference is that the created structure remains a tree because the new sample is linked only to the one tree vertex. The chosen tree vertex minimizes the weight of the path from the tree root to the new vertex. Once a new vertex has been added, parental vertex values are adjusted if there is path weight improvement (the path is cheaper).

The input of the algorithm is the start configuration, the set of obstacles, the radius r of the candidate's surroundings to the connection and the number of iterations. The output is the tree mapping the configuration space. The tree is optimal with respect to the cost of the path from its vertices to the root of the tree.

Chapter 5

Group Approach

New path planning problems appear and complex problems persist, such as a real-time planning of paths for huge crowds in dynamic environments, where the properties according to which the cost of a path is evaluated as well the topology of paths may change. We have invented a new approach, called group approach [79], of planning paths for crowds, applicable to any environment representation (e.g. graph, mesh,..). The main idea is to group members of the crowd by their common initial and target positions and then plan the path for one representative member of each group. These representative members can be navigated by classic approaches such as A*, D* Lite, etc., and the rest of the group will follow them. If the crowd can be divided into a few groups this way, the proposed approach will save a huge amount of computational and memory demands in static as well as dynamic environments.

This section describes the path planning approach for many agents in an environment represented by an undirected graph $G = (V, E)$, where V is a finite, non-empty set of vertices, and E is a set of unordered pair of vertices ($E \subset \binom{V}{2}$) called edges. First, we will describe how to plan paths for agents in a group. Then we will discuss an approach for creating the groups and finally we present the experiments.

Let $P = \{p_1, p_2, \ldots, p_c\}$ be a set of agents. Each $p_i \in P$ needs to individually rate vertices of an undirected graph. The graph represents the dynamic environment which can be interpreted as a set of pairs $D = \{(d_1, t_1), (d_2, t_2), \ldots, (d_r, t_r)\}$, where d_i is a graph change and t_i is a simulation time. Each vertex of the graph describes a point in $\mathbb{R}^2$. Moreover, every agent p_i has a starting vertex with the position $s(p_i) \in \mathbb{R}^2$ and a destination vertex with $e(p_i) \in \mathbb{R}^2$.

5.1 Path Planning for Groups

Let $g \subseteq P$ be a group of agents. The group has one leader $p_m \in g$, the main agent that will be followed by others. The criteria for creating groups and choosing a leader will be discussed later, now let us focus on planning paths for the members of g. Figure 5.1 illustrates the idea. First, the path for the leader p_m is computed by any standard algorithm for path planning. The path starts in $s(p_m)$ and ends in $e(p_m)$. After that, the path for each agent $p_i \in g \setminus \{p_m\}$ is computed: First from $s(p_i)$ to $s(p_m)$, then the part of the path from $s(p_m)$ to $e(p_m)$ is reused, and finally the part from $e(p_m)$ to $e(p_i)$ is computed. The path is not necessarily optimal and may require further optimizations, which will be discussed next.

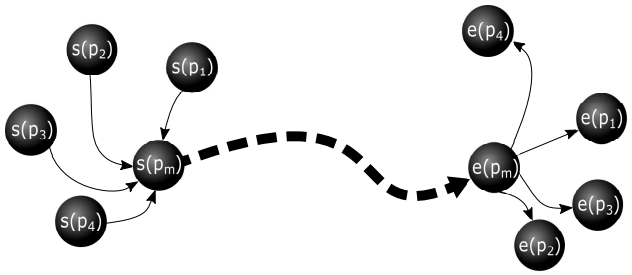


Figure 5.1: The idea of path planning for a group of agents. The path of each agent p_i starts in $s(p_i)$ and ends in $e(p_i)$, p_m is the leader. The others can reuse leader's path.

The agent p_i can find the path of the leading agent p_m (Figure 5.2a) before p_i reaches $s(p_m)$. The path of the agent p_i will be longer because p_i would have to visit $s(p_m)$ and return back to the position where the path of p_m was discovered. Therefore, the path planning is modified in the following way to handle such situations. When the agent p_i reaches the path of the leader p_m, the path planning to the start position $s(p_m)$ is stopped and the essential part of the path of the leader p_m is used instead (Figure 5.2b). Once the agent p_i reaches the destination position $e(p_m)$, path planning algorithm from the position $e(p_m)$ to $e(p_i)$ should be started. However, Figure 5.2c illustrates that a situation similar to Figure 5.2a may happen. The change of the path planning direction transforms this problem to the already solved problem (Figure 5.2b).

5.1.1 Creating Groups

Let us describe how to create groups of agents in such a way that all agents in a group have similar starting positions and similar destinations.

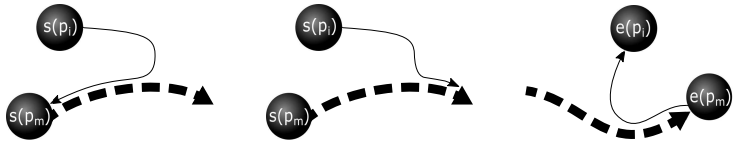


Figure 5.2: Group path planning problems and solutions (a) Problem near the start position, (b) Solution for the start position (c) Problem near the destination

The approach for creating groups of agents is shown in Algorithm 2. It takes a list P of agents and a grouping parameter τ (threshold) as the input and produces a set $Gr = \{g_1, g_2, \ldots, g_q\}$ of non-empty distinct groups of agents. The algorithm also needs a path planning strategy implementing some of the classic path planning algorithms, e.g., A*, D* or D* Lite, because the computation of agent groups also depends on the paths of leading agents.

Algorithm 2 just iterates over all agents and compares them to all groups created so far. There is always the main agent p_m for each group. In our case, it is the first agent which was added to the group. The start position $s(p_i)$ and the destination $e(p_i)$ of each agent p_i is compared with the start and the destination of the main agent $p_m \in g_j$. If both distances are less than some ϵ value computed from the threshold τ, the agent p_i is added to the group g_j and the cycle over groups is terminated. If the agent p_i is not added to any existing group, a new group is created for the agent and added to the set of groups Gr.

The worst-case time complexity is $O(|P||Gr|)$ or $O(|P|^2)$ if each group contains only one agent. The algorithm is output-sensitive, its performance and quality of results depend on the value of ϵ. In the rest of this section we will discuss two strategies for choosing this parameter.

The easiest way of using ϵ constant for all agents might produce bad results: Let us consider a hypothetical example, where $\epsilon = 200$m and the final length of the path of the main agent p_{m1} is 20km. Some agents added to the group of the agent p_{m1} may have their paths longer about approximately 2ϵ. The agent p_i added to the group with the agent p_{m1} will have the path of the length somewhere between 20km and 20km+2ϵ. At this moment the error is relatively small but what if there is the main agent p_{m2} with the path 100km long? The path length of p_i added to the group with p_{m2} is then between 100km and 100km+2ϵ. The path might be even 5 times longer than the minimal one and that is a problem. Small ϵ produces a small number of groups and with growing ϵ the error with the short paths grows.

To fix the shortcomings of the static threshold, we use a dynamic ϵ specific

Algorithm 2: Creating agent groups

Data: A list P of agents, a threshold $\tau \in [0, 1]$, a path planning strategy `find_path(...)`
Result: A set Gr of agent groups

```
1  Gr ← ∅;
2  foreach p_i ∈ P do
3      foreach g_j ∈ Gr do
4          p_m ← the first member of g_j; // the main agent of the
               group (leader)
5          ε ← p_m.path_length * τ/2;
6          if ‖s(p_i) − s(p_m)‖ < ε and ‖e(p_i) − e(p_m)‖ < ε then
7              g_j ← g_j ∪ {p_i};
8              break
9      if p_i was not added to any group then
10         Gr ← Gr ∪ { {p_i} };
11         path ← find_path(s(p_i), e(p_i));
12         p_i.path_length ← Σ_{edge∈path} length(edge);
13 return Gr;
```

to each group, which approximates the maximal allowed error in per cents of the path length of the main agent. For instance if $\epsilon = 10\%$, the final path is allowed to be approximately 1.1-times longer than the path of the main agent. When a new group is created in Algorithm 2, the path length of the main agent is computed, multiplied by half of the grouping parameter τ and used as the group-specific ϵ. We also use this approach to bound the relative error of the found path.

5.2 Experiments and Results

The proposed method was mainly tested on real data – the Open Street Map of the City of Pilsen, Czech Republic, but the agents (pedestrians) were generated in random positions. The only exception is the relative error dependency on the common path which has been tested on artificial data and agents. The solution was implemented in C++ and all experiments were performed on a computer with the CPU Intel® Core™ i7-950 (8MB Cache, 3.07GHz) and 12GB 668MHz RAM.

Although the computational time and space are the most important characteristics to measure, we are not able to determine them generally because the characteristics highly depend on the chosen path planning method and

the distribution of the agents. For example the proposed solution with A* as the path planning algorithm will not save any computational space. On the other hand with the D* Lite algorithm this approach is able to save up to $O(kn)$ space, where k is the number of the similar paths and n is the number of vertices. Therefore, the experiments compare the computational time of the Algorithm 2 and especially the path correctness.

5.2.1 Computational Time

Figure 5.3 shows the computational time of the A* algorithm and the A* with the proposed solution for the cases where $|Gr| = |P|$, $|Gr| = \frac{|P|}{2}$ and $|Gr| = 1$. The computational time of the A* algorithm in this graph problem is $O(n)$ as the experiments prove. The proposed method is completely inappropriate for the case $|Gr| = |P|$ where the agents do not create the common groups at all. When no agent p_i has similar path with agent p_j, every path has to be computed and the proposed method only slows down the computation (the top curve in Figure 5.3). The computational time of the second case $|Gr| = \frac{|P|}{2}$ is slower than the computational time of the single A* algorithm. The computational time of the proposed method and the single standard path planning algorithm is similar when 35-40% of all agents belong to the group g_i with another agent. The situation with $|Gr| = 1$ is the fastest, as was expected.

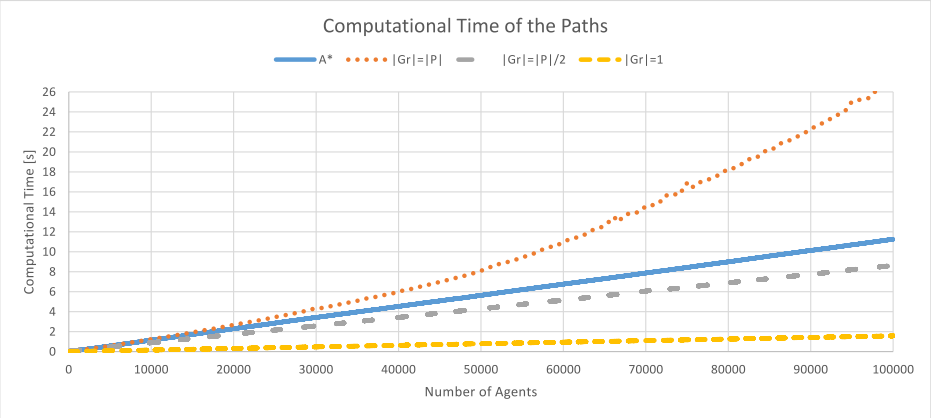


Figure 5.3: Computational time of single A* and A* with groups approach

Figure 5.4 shows how the computational time of A* algorithm with the proposed method changes with different allowed relative error bound ϵ. When the allowed relative error is chosen as 10% of the leader agent path, 74524 groups are created from 100k of agents. The 20% allowed error bound produced 59478 groups, 30% 48624 groups, 40% 41471 groups and 50% 32678

groups. Obviously the growing relative error bounds produces a smaller number of the groups, because the groups have greater radius. The more important the computational time for the 40% and higher bound the computational time grows. The 35% bound is the milestone relative error δ_m where the paths to and from the centre of the group start to be longer than the common path. Therefore, the proposed solution starts to slow down when δ_m is reached. The better agents data (a lot of intended groups) provides higher δ_m.

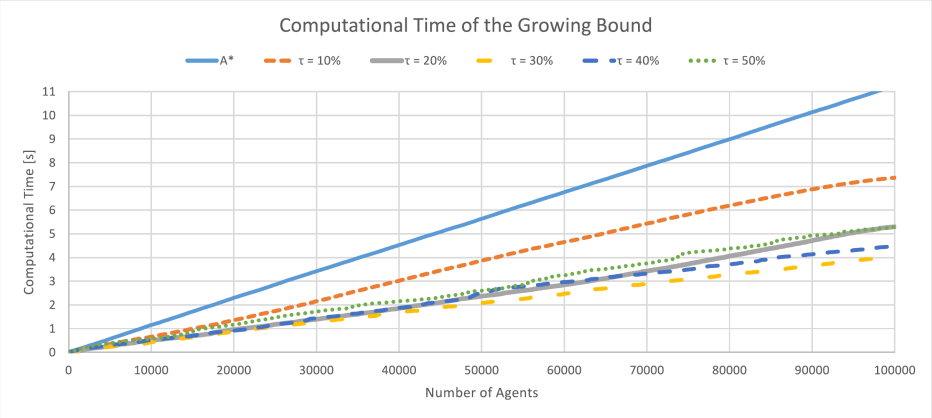


Figure 5.4: Computational time of A* alone versus our approach for different values of the grouping parameter τ

Although Figure 5.4 shows that with less groups the path computation is faster, it is not always true. The proposed solution is faster until the milestone relative error δ_m is reached. When the value δ_m is crossed, the approach starts to slow down, because the paths to and from the centre of the group start to be longer than the common path.

5.2.2 Path Correctness

The correctness of the found path shows how large is the difference between the minimal path and the path found by the proposed solution. The path correctness is measured by a relative error (5.1).

$$\delta = \frac{length(group.path) - length(minimal.path)}{length(minimal.path)} \tag{5.1}$$

where *group.path* is the path found with the group approach including existing path planning method and *minimal.path* is the path found with the same path planning algorithm without the proposed solution. Note that $\delta \geq 0$.

The result in Figure 5.5 shows the dependency of the relative error on the common path. The experiments have been performed on artificially generated data of 1000 agents for the 10% error bound, where all of them were in the same group g_j. The relative error decreases with the growing common path, as was expected. Although the maximal relative error may be large, the proposed solution should be used when the common path is over 80% of the found path.

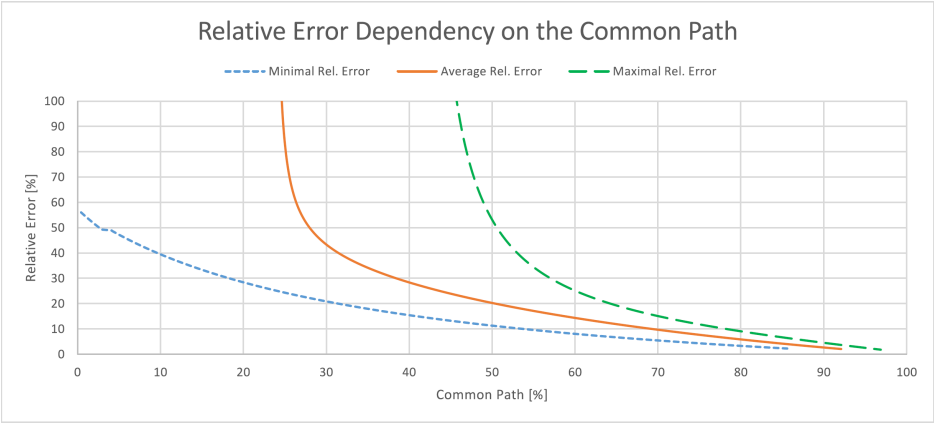


Figure 5.5: The relative error dependency on the length of the common path

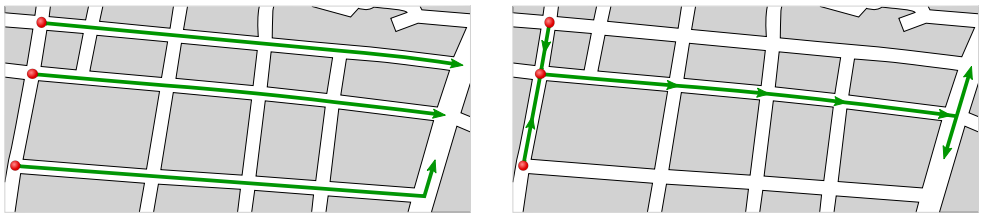
	Relative error per cent					Relative error per cent			
τ [%]	Min	Mean	Max	Groups	τ [%]	Min	Mean	Max	Groups
1	0.000	0.000	0.000	99700	11	0.470	4.610	13.193	93500
2	0.913	0.913	0.913	99500	12	0.470	4.934	17.916	92100
3	0.913	1.104	1.296	99400	13	0.470	5.345	17.916	90700
4	0.913	1.634	2.694	99000	14	0.413	5.602	24.534	89900
5	0.913	2.150	3.358	98900	15	0.161	6.566	24.534	87700
6	0.470	2.104	3.822	98300	16	0.161	7.192	24.534	86600
7	0.470	2.587	5.801	97500	17	0.161	7.316	24.534	84600
8	0.248	2.837	9.092	96800	18	0.261	7.779	26.667	82100
9	0.248	3.999	13.193	95600	19	0.261	8.463	33.868	80200
10	0.248	4.352	13.193	94500	20	0.044	8.924	41.353	78200

Table 5.1: The final path inaccuracy with the proposed solution (Algorithm 2) for increasing values of the grouping parameters τ

The found paths of the proposed solution were compared with the exact minimal paths of the randomly generated data of the agents. The solution has been tested for the allowed relative error from 1% to 20% and its results

are listed in Table 5.1. The Table 5.1 contains the number of groups and the minimal, average and maximal relative error for the allowed relative error. The minimal relative error is the smallest relative error higher than zero. It is obvious that with the decreasing number of groups, the average and maximal relative error grows. Although the maximal relative error grows over the allowed boundary from the value 0.08, the overall average relative error is still within the expected range. Moreover, the average error is smaller than the half of the allowed relative error in every tested case.

Depending on the data there may occur three extreme cases of the proposed solution. In the first case every agent starts and ends his path at the same positions. Then the approach saves a huge amount of the computational time and space. The second case is the worst possible situation, where none agent has a similar path with another agent. This kind of data is unsuitable for the proposed solution because the group computation not only does not save computational time but even slows down the computation. The expected result is shown in Figure 5.6, where the minimal path is found for every agent (red point) without using the proposed solution (Figure 5.6a). Figure 5.6b illustrates the same case with the proposed solution as Figure 5.6a.



(a) (b)

Figure 5.6: (a) Minimal found path of each agent, (b) Found path with the proposed solution

Chapter 6

Cluster Approach

This section describes our path planning approach for many agents in an environment represented by an undirected graph $G = (V, E)$, where V is a finite, non-empty set of vertices, and E is a set of unordered pair of vertices ($E \subset \binom{V}{2}$) called edges. A clustering approach [77] is based on the group approach [79], and, therefore, it uses the same assumptions and sets. The cluster approach has exactly the same idea as the group approach. The only difference is in the way how to create the groups. As the name of the newer approach suggests, we will use clustering to create groups of agents.

First, we will make a clustering background to find the most suitable candidate (Section 6.1). Then we describe a better and faster solution with the use of clusters in Section 6.2 and finally we present experiments and results which will compare the group approach with the clustering approach (Section 6.3).

6.1 Clustering background

The clustering algorithms group similar elements (called *clients*) together [1, 35] into groups (called *clusters*) represented by one centroid (called a *cluster center* or a *facility*). The nature of the elements are based on the application and in a general case they could be any objects, which can be described by a characteristic N-dimensional vector. Depending on the area of application, a transformation into the N-dimensional space may be needed to express elements. For example, for processing a point in a N-dimensional space its N spatial coordinates are used. A similarity of elements is measured by a distance function. The function can be modified based on application requirements and basically might not fulfill the properties of the metrics such as non-negativity, symmetry or triangle inequality. However, the Euclidean

distance is used most often.

The literature shows many clustering methods such as single-link [70, 66], complete-link [42], sweep-line [85], k-means [50] and facility location [10].

Now we will define the clustering task more precisely. As we have chosen the facility location algorithm for our purpose, we will present the definition suitable for this type of algorithm.

A given data set of N input data points $x_1, x_2, \ldots, x_N$ is subdivided into k disjoint subsets $F_i, i = 1, \ldots, k$ each containing n_i data points, $0 < n_i < N$ by minimizing the following mean-square-error (MSE) clustering cost:

$$J_{MSE} = \sum_{i=1}^{k} \sum_{x_j \in F_i} \|x_j - f_i\|^2 \tag{6.1}$$

where x_j is a vector representing the j-th data point in the cluster F_i and f_i is the cluster centre of the cluster F_i. The J_{MSE} function in Eq. (6.1) represents the distance between the data point x_j and the cluster center f_i.

As we can see, the number of clusters has to be determined before clustering, which is not suitable in all scenarios. Moreover, if k equals to N, J_{MSE} loses its measuring property and the result of clustering is not correct. The facility location algorithm keeps a reasonable amount of clusters by minimization of the following clustering cost:

$$J_{FL} = \sum_{f_i \in F} fc + \sum_{j \in C} c_{f_i j} \tag{6.2}$$

where $c_{f_i j}$ is the distance between a data point $j \in C$ and its facility $f_i \in F$. The set C contains all data points. To open a new cluster center, a cost fc must be paid, this way a quantity of cluster centers can be controlled.

The clustering task is an NP-hard problem, so the most algorithms produce only approximate results or have some restrictions. On the other hand, in many scenarios the approximate solution suffices.

Clustering methods are used in technical as well as non-technical disciplines, such as data analysis [18, 3], data mining [20], image segmentation [35], pattern recognition [4], information retrieval [64].

There are several ways how to categorize the clustering algorithms [68]. One of possible subdivisions is into *partitional* and *hierarchical* methods. The partitional ones divide the data into an exact number of clusters (partitions), the hierarchical ones create a hierarchy of small clusters grouped into larger clusters forming a tree structure. Another possible subdivision is into *agglomerative* and *divisive* (or *partitional*) approaches. The agglomerative ones start with each element in a single cluster. The final result is formed by

successively merging these clusters according to a similarity measure until a stopping condition is met. The divisive approaches start with one large cluster which contains all the data. By repeatedly splitting clusters according to a dissimilarity measure smaller clusters are created. Both agglomerative and divisive algorithms stop when there is a predetermined number of clusters or when existing clusters are homogeneous enough so that no further iteration is needed.

The clustering is called hard if each element is assigned into exactly one cluster. Fuzzy clustering determines for each element a degree of membership in several clusters.

Clustering algorithms can be *deterministic* or *stochastic*. Stochastic techniques usually use randomized algorithms which are more suitable for processing large amounts of data due to their smaller time complexity.

6.2 Clustering approach

The output groups of the group approach are sensitive to the order of the input data because the algorithm has a greedy character - the first possible solution is accepted and never reconsidered. An additional optimization of the found groups would improve the final paths but deteriorate the complexity which is already $O(n^2)$. What is more, although the group approach speeds up the path computation, there is still room for acceleration. Therefore, we incorporate to the group creation a clustering by a non-modified local search algorithm [69, 55] with relevant parameters setup discussed in 5.2. The algorithm implicitly optimizes the agent groups and their found paths. Moreover, the clustering even speeds up the computation because the clustering can be done in $O(n \log n)$.

For clustering purposes the agent p in the input set $P = \{p_1, p_2, \ldots, p_N\}$ is described as the following 4-dimensional vector:

$$v_i = \big(s(p_i), e(p_i)\big)^T, v_i \in \mathbb{R}^4 \tag{6.3}$$

where $s(p_i) \in \mathbb{R}^2$ is the starting and $e(p_i) \in \mathbb{R}^2$ the destination position of the agent.

The main idea of the proposed improvement is to incorporate Eq. (6.2) as a heuristic to avoid checking so many possible cases compared to the original group approach at a price of a lowered accuracy of the clustering result.

Let us describe the used *local search* clustering algorithm steps in detail. At first, a coarse initial solution is generated. A cluster center is always created at the first point and further points are then taken in the random order. A point v_i is connected to the closest already open cluster center based

on the measured distance d between the v_j point and the open cluster center. A new cluster center is opened at the point v_j with probability d/fc (or one if $d > fc$). This initial coarse solution is improved by $O(n \log n)$ iterations of the following local search step. The explanation for $O(n \log n)$ steps of iterations is given in [69].

A single local search step can be described as a random selection of a point $v_i \in C \bigcup F$ (it does not matter whether it is a cluster center or not) and it is computed whether the agent p_i can improve the current solution (if v_i is not already a cluster center, the facility cost would have to be paid for its opening). Some clients (points) may be closer to the investigated (new) cluster center f_{v_i} than to their current facility. All such clients can be re-assigned to f_{v_i}, it decreases the connection cost. If these changes result in some cluster center having only a few clients remaining, the cluster center could be closed and its facility cost spared.

A possible improvement of the current solution by declaring the point v_i a new cluster center f_{v_i} and reassigning all near clients from their cluster centers to f_{v_i} is determined by a gain function according to the following relation:

$$gain(v_i) = -fc + \sum_{c_i \subseteq C} ds_i + \sum_{f_j \subseteq F} cs_j \tag{6.4}$$

where fc is the facility cost, or zero if the cluster center f_{v_i} is already open, ds_i is the distance spared by reassigning the client c_i from its current cluster center to the cluster center candidate f_{v_i} and cs_j (close spare) is the facility cost minus expenses for reassigning all remaining clients from their current cluster center f_j to f_{v_i}. If the current cluster center f_j lies closer to v_i than f_{v_i} then $ds_i < 0$ and ds_i needs to be set to 0. Again, if $cs_j < 0$ (cluster center f_j has enough clients, so no spare can be achieved by closing the cluster center and reassigning all their clients to the new cluster center f_{v_i}) then cs_j is set to 0. If $gain(v_i) > 0$, the cluster center at point v_i is opened (if not already open) and reassignments and closures are performed.

The algorithm of the group approach with the facility location clustering is summed up as Algorithm 3. When the clusters, which represent the groups of agents, are computed (Alg. 3, line 1) from the input set of agents P, the path of each agent is planned by any standard path planning strategy, e.g., Dijkstra, A* or D*. First the path of the leader p_m of the group g_i is found (Alg. 3, line 4). Then two paths are computed each other member of the group g_i - first between vertices $s(p_j)$ and $s(p_m)$ and second between $e(p_j)$ and $e(p_m)$ (Alg. 3, lines 6-7). Finally, this paths and path of the leader are joined (Alg. 3, lines 8). This process goes in cycle for each group $g_i \in Gr$. The

procedure $cluster_agents$ generates first a coarse solution of agent groups, then it repeatedly tries to improve it by reassignment of a randomly chosen point to another group. If the new connection improves the overall clustering cost J_{FL}, then it is preserved and another randomly chosen point is investigated. The improvement phase is repeated $O(n \log n)$ times.

Algorithm 3: The clustering path planning approach

```
   Data: A list P of agents, a path planning strategy find_path(...)
   Result: The list P of agents with computed paths
 1 Algorithm cluster_approach
 3     Gr ← cluster_agents(P); // groups of agents
 5     foreach g_i ∈ Gr do
 7         p_m ← the first member of g_j; // the leader of the group
 9         p_m.path ← find_path(s(p_m), e(p_m));
11         foreach p_j ∈ {g_i \ {p_m}} do
13             path_start ← find_path(s(p_i), s(p_m));
15             path_end ← find_path(e(p_i), e(p_m));
17             p_j.path ← Join_paths(path_start, p_m.path, path_end)
18     return P;
 1 Procedure cluster_agents(the list of agents P)
 3     Generate a coarse solution of groups.
 5     repeat O(n log n) times
 7     Pick v_i ∈ P at random;
 9     if gain(v_i) > 0 then
11         Perform reassignments and closures.
12     return groups;
```

6.3 Experiments and Results

The proposed method was implemented in C# and all experiments were performed on a computer with the CPU Intel® Core™ i7-950 (8MB Cache, 3.07GHz) and 12GB 668MHz RAM. The proposed solution was tested on two types of the testing data. First type ('unsuitable data') contains agents generated at random positions and the second type ('suitable data') contains groups of agents (many agents with similar paths). The environment is represented by real data – the Open Street Map of the City of Pilsen, Czech Republic.

We were unable to determine the computational time and space generally because they highly depend on the distribution of the agents and the chosen

path planning algorithm. For instance the A* algorithm as the path planning method will not save computational space because it does not need to store a unique graph rating for each agent. However, dynamic algorithms such as D* Lite algorithm are able to save up to $O(kn)$ space for k similar paths and n vertices. Therefore, the experiments focus on the computational time and especially on the path correctness of the proposed solution.

6.3.1 Computational Time

The computational time of the clustering method without the path planning is shown in Figure 6.1, where the time dependencies on the facility cost fc are depicted. The higher value of fc produces bigger clusters and speeds up the computation of the clusters. The reasonable facility cost is for $fc > 0.1$, where the computational time becomes relatively low (Figure 6.1) and with the growing fc the time still descents.

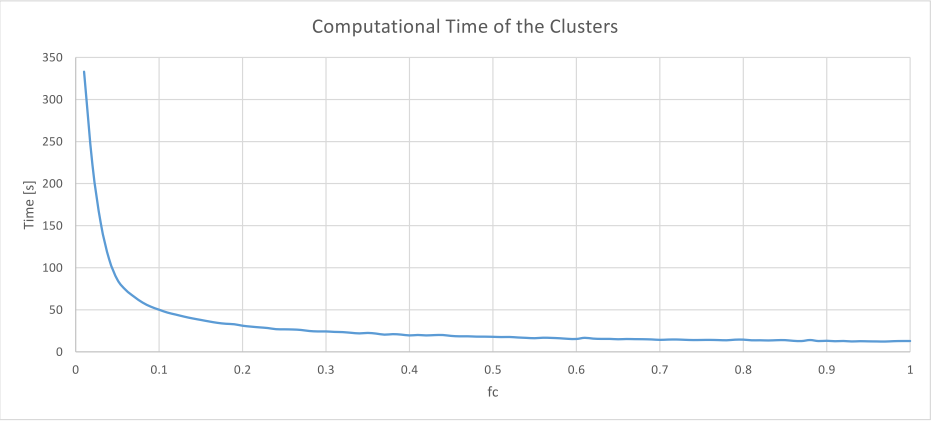


Figure 6.1: Computational time of the clustering method for 100k agents

Figure 6.2 shows the computational time of the standard A* algorithm, the A* with the group approach and with the proposed clustering approach (Alg. 3). The experiments were made for $100k$ randomly generated agents and the measured computational time of both approaches includes the groups creation and path planning strategy. The group approach depicted in Figure 6.2 uses $\tau = 10\%$ and the clustering approach is showed with the facility cost $fc = 0.5$. The proposed clustering approach is the fastest. Small fluctuations on the time curve are caused by the random character of the clustering algorithm. The curve of the clustering approach is $O(n \log n)$, the group approach curve is $O(n^2)$ and the A* curve is $O(n)$. Although the A* is in the graph the slowest, obviously it thanks to its better algorithmic complexity

for some number of agents overruns both the group and the clustering approaches. For the given size of the environment it will be $\approx 300k$ agents for the group approach and over millions of agents for the clustering approach. This number is relatively high and so it is not such a big problem as it might seem.

However, the data of randomly generated agents are the worst possible for the proposed approach. The clustering approach is most suitable for the groups of agents where the clustering approach is a clear choice because it is able to reuse many paths and save a huge amount of the computational time (Figure 6.3). The same computational time of the A* algorithm and the clustering approach with A* algorithm is for billions of agents for the data of the groups of agents.

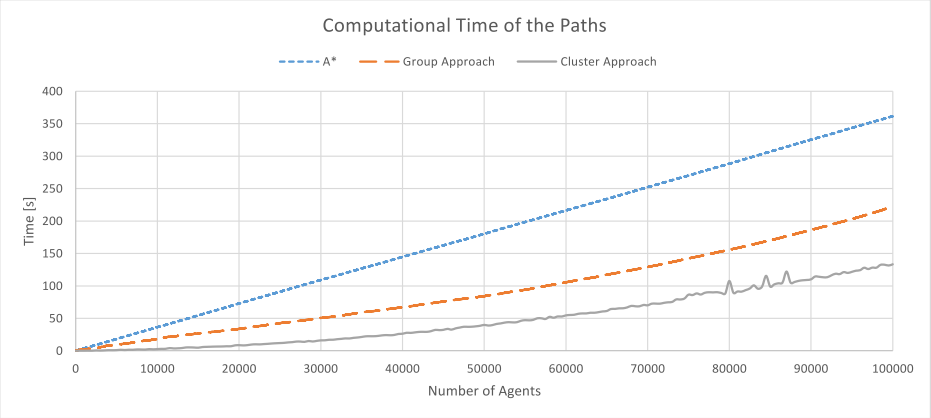


Figure 6.2: Computational time of single A* and A* with group and clustering approaches for the data of randomly generated agents.

6.3.2 Path Correctness

The correctness of the found path shows how large is the difference between the minimal path and the path found by the proposed solution. The path correctness is measured by a relative error (6.5).

$$\delta = \frac{length(group.path) - length(minimal.path)}{length(minimal.path)} \tag{6.5}$$

where *group.path* is the path found with the group or the clustering approach including existing path planning method and *minimal.path* is the path found with the same path planning algorithm without the proposed solution. Note that $\delta \geq 0$.

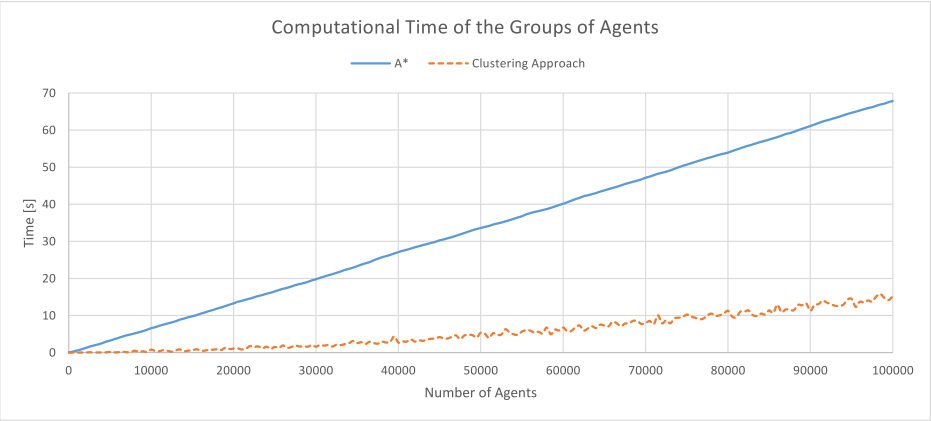


Figure 6.3: Computational time of single A* and A* with the clustering approach for the data of the groups of agents.

The average relative error dependency on the chosen facility cost is shown in Figure 6.4 for two above mentioned types of the testing data. The upper curve, which reaches the relative error up to 15%, represents $100k$ randomly generated agents positions and the lower curve represents data with generated groups of agents (a lot of similar paths). The average relative error of the groups does not exceed 2%. This is an expected result because this type of data are most suitable for the proposed clustering approach. Moreover, the result of the worst case (randomly positioned agents) is acceptable and for the facility cost up to the value 0.1 is comparatively very good. It can also be seen that the relative error does not change too much if $fc > 0.5$.

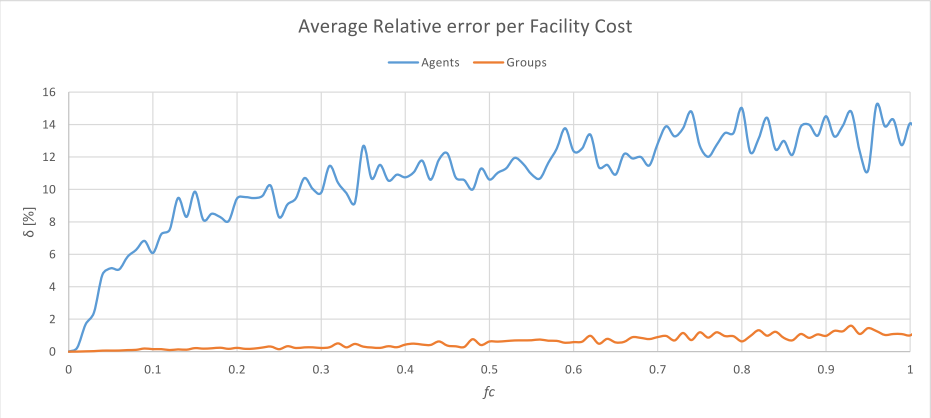


Figure 6.4: The average relative error δ for $100k$ generated agents, either random or in groups, in dependence on facility cost fc.

It means that if a higher relative error is acceptable, number of clusters can be kept relatively small. The parameter fc can be understood as percentage expression number of clusters, where approximately $100(1 - fc)$ per cent of agents from the input data become cluster centers.

Figure 6.5 shows the dependency of the relative error on the relative number of the created groups and the difference between the original group approach (without clustering) and the group approach with clustering (Alg. 3). The x-axis describes the relative number of groups $\lambda = \frac{|Gr|}{|P|}$, where $|Gr|$ is the number of created groups and $|P|$ the number of tested agents. The experiments have been performed on the randomly generated data of $100k$ agents. The higher the number of groups is, the fewer agents each group contains. For example, when the proposed solution creates $100k$ groups (clusters) from $100k$ agents, the average relative error will be zero because every agent is a leader of a group. The other extreme is only one group (a cluster) created from $100k$ agents. In that case, relative error will be enormous because everyone has to follow the leader. The average relative error of both approaches grows with the decreasing number of groups, as can be expected. Although both approaches are almost the same for a high number of groups, the clustering approach gives better results for a smaller number of groups, which contain a higher number of agents. The average relative error of the clustering approach oscillates for the small number of groups because it contains randomized operations which are more visible for greater groups.

In the graphs, attention was drawn to the average relative error. The maximal relative error is influenced by the setting of the parameters - how big attraction of the group leaders for other agents is allowed. If big, then

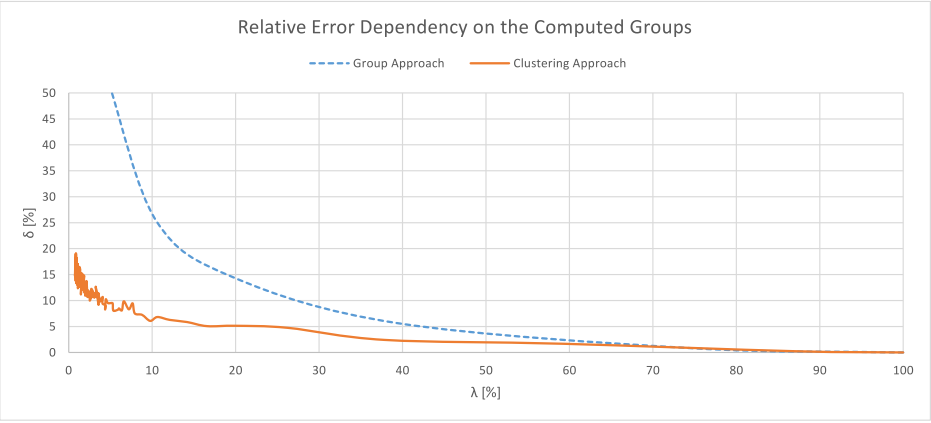


Figure 6.5: The average relative error δ of the group and clustering approaches in dependence on the relative number of groups λ.

even more distant agents are pushed to group with a leader which may result in a high suboptimality of the path for the given agent. Such a path then has a high relative error and so contributes to a high maximal error. Fortunately, maximal errors concern individuals while the approach is aimed at groups or even crowds, so one strongly non-optimal path is not important in the intended applications and will be amortized by good behavior of the whole crowd.

Chapter 7

Future Work

At present, we focus mainly on the motion planning in the biochemical field of proteins. In Chapter 4, we have introduced some possible algorithms that are able to find the path through environment represented by the configuration space, which takes collisions with the environment into account. We decided to focus primarily on the Rapidly-exploring Random Trees (RRT) group, which is, in our opinion, more suitable for this issue.

We can already find a path, through which the ligand will move without collisions. We can compute it for any rigid ligand and any static protein. In the near future, our goal is to test various metrics to help us find the widest path faster or find higher number of paths. The question is whether it is appropriate to use only the Euclidean distance, angular rotation, a combination of both, or incorporate some information about the surroundings or the current state of the ligand into a metric. These metrics should be thoroughly compared between all RRT algorithms and we will select the most suitable candidate, whether in terms of speed or accuracy.

Another challenge lies in the transition from the static protein to the dynamic protein. Individual parts of the protein can move, which affects the length and shape of the resulting tunnel at each step. It may even happen that the tunnel closes itself and becomes impassable.

One of the problems of dynamic data is data itself - specifically their size on the hard drive. There is no analytical equation describing the movement of the protein, and therefore the dynamic behavior of proteins is described as hundreds or thousands of static proteins (frames). All these data describes the same protein in different simulation time t_i, where $i = 0 \dots n$ and $t_0 < t_1 < \dots < t_n$. Another problem will be to find a path or paths inside the dynamic protein. The first possible approach is to find whole paths in each frame from scratch. On the one hand, such an approach will be time consuming, but on the other hand, a large number of paths will help us to

analyze the protein behaviour more closely to speed up computation or to find more accurate paths. The second approach can be to find paths only in the first frame, and then transfer the found paths to the second frame. It would be necessary to check that there is no collision on paths and possibly modify or recalculate the path in collision with the protein. The pitfalls that may arise with this approach are that the PRM (Probabilistic Roadmaps) algorithms will be more suitable for this problem (Section 4.2) than RRT (Rapidly-exploring Random Tree) algorithms, because they sample the whole space and create the graph G. With this information, we could then move individual vertices to the correct positions or change the topology of the resulting graph G.

It is also possible to use the technique of starting points and exit points. The starting points represent an active site in the protein - the site where the chemist wants to bring a ligand, because it has the greatest effect here. The most problematic places (bottlenecks) around the active site are called exit points. In the case of denser sampling of these sites, we could achieve better results. To calculate starting and exit points, Voronoi diagrams are used, which could also be very useful for us to find paths through protein. We do not have to sample randomly, but we can follow the edges of the Voronoi diagram.

A little less difficult, but also substantial, may be the incorporation of nonrigid flexible ligands. The ligand has not just the same shape, but it can vary, which can, of course, make it easier for us to pass through the protein but it can also sometimes make it difficult to pass the ligand through the protein.

Last but not least, there is the possibility to publish our research in the field of path planning. We can speed up some path computation in a partially-known or unknown environment. Acceleration can be achieved by using some local heuristics, which can tell us the real pedestrian behavior on the street.

Chapter 8

Conclusion

At the beginning of this thesis we introduced the possible representations of the path planning environment. Subsequently, we focused on a thorough study and description of static and dynamic algorithms for path planning in known, partially known and unknown environments. The second theoretical part focused on the description of motion planning and its two basic approaches - Rapidly-exploring Random Tree (RRT) and Probabilistic Roadmaps (PRM).

We also introduced our two path planning approaches for many agents in an environment which can be represented by a graph of vertices and edges. Both of the approaches can also be used for other types of environment, e.g., grids or polygonal meshes and both of the proposed methods belong to the class of global path planning methods, which do not handle collisions detection and avoidance.

The first approach, the group approach, creates groups of agents based on their start and goal positions and then it uses any standard graph-based path planning algorithm, e.g., Dijkstra, A*, D* or D* Lite, to plan the paths of group leaders and also partially paths of other group members. The major advantage is that group members share a substantial part of the leader's path, but their paths are not necessarily optimal. Currently, we use the A* algorithm for static graphs in our path planning framework, but it could be easily extended to employ the D* Lite algorithm for dynamic graphs.

We have measured the influence of the grouping parameter on the quality of the paths and the running time. Our experiments indicate that the proposed approach is perfectly tailored for the data with a significant number of potentially similar paths, however, it is output-sensitive and the quality of results as well as the running time depends on the grouping parameter.

The second approach, the clustering approach, creates groups of agents based on their start and target positions and optimizes the size of the created

groups by clustering. Any standard graph-based path planning algorithm, e.g., A*, D* or D* Lite is then used to plan the paths. The substantial part of the computed path of group leaders is shared with the rest of the group. In this way, computation time and memory can be saved at a price of non-optimality of paths. The independence of the proposed approach of the dimension and the data types, e.g. data of cities or proteins, has been confirmed by extensive testing. The running time and the influence of the clustering on the quality of the paths has been measured. The clustering approach proved to be suitable for the data with a significant number of potentially similar paths. Moreover, the experiments proved that it is possible to use the online computation with a relatively small inaccuracy for this type of data.

Although the clustering is the NP-complete problem, the clustering approach has proven to be faster and more accurate. We have achieved a noticeable reduction of the relative error for a small number of groups (the smaller number of groups, the higher relative error). The computation of the clustering approach is nearly a third faster than in the case of the group approach.

Thesis is concluded with a chapter that contains our vision and the potential future research in the field of path and motion planning.

Appendix A

Activities

A.1 Publications on International Conferences

- [**79**] Jakub Szkandera, Ivana Kolingerová, and Martin Maňák. Path planning for groups on graphs. *Procedia Computer Science*, 108:2338–2342, 2017

- [**77**] Jakub Szkandera, Ondřej Kaas, and Ivana Kolingerová. A clustering approach to path planning for groups. In *International Conference on Computational Science and Its Applications*, pages 465–479. Springer, 2017

A.2 Publications in Impacted Journals

- [**51**] Martin Manak, Michal Zemek, Jakub Szkandera, Ivana Kolingerova, Elena Papaleo, and Matteo Lambrughi. *Hybrid Voronoi diagrams, their computation and reduction for applications in computational biochemistry*, volume 74. Elsevier, 2017

- [**78**] Jakub Szkandera, Ondřej Kaas, and Ivana Kolingerová. A clustering approach to path planning for big groups. *International Journal of Data Warehousing and Mining (IJDWM)*, 15(2):42–61, 2019

A.3 Publications in Non Impacted Journals

- [**9**] Petr Broz, Michal Zemek, Ivana Kolingerová, and Jakub Szkandera. Dynamic path planning with regular triangulations. *Machine Graphics & Vision*, 24(3/4):119–142, 2014

A.4 Participation in Scientific Projects

- Interactive Geometrical Models for Simulation of Natural Phenomena and Crowds. Project leader Ivana Kolingerová. Funded by The Ministry of Education, Youth and Sports, project code LH11006
- Advanced Graphical and Computing Systems. Funded by The Ministry of Education, Youth and Sports, project code SGS-2016-013
- Methods of Identification and Visualization of Tunnels for Flexible Ligands in Dynamic Proteins. Project leader Ivana Kolingerová. Funded by Czech Science Foundation, project No. 17-07690S

Bibliography

[1] R.C. Dubes A.K. Jain. Algorithms for clustering data.

[2] Nancy M Amato, Ken A Dill, and Guang Song. Using motion planning to map protein folding landscapes and analyze folding kinetics of known native structures. *Journal of Computational Biology*, 10(3-4):239–255, 2003.

[3] Geoffrey H Ball and David J Hall. Isodata, a novel method of data analysis and pattern classification. Technical report, DTIC Document, 1965.

[4] Andrea Baraldi and Palma Blonda. A survey of fuzzy clustering algorithms for pattern recognition. i. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 29(6):778–785, 1999.

[5] Eric Bonabeau. Agent-based modeling: Methods and techniques for simulating human systems. *Proceedings of the National Academy of Sciences*, 99(suppl 3):7280–7287, 2002.

[6] A_ Bondi. van der waals volumes and radii. *The Journal of physical chemistry*, 68(3):441–451, 1964.

[7] Adi Botea, Martin Müller, and Jonathan Schaeffer. Near optimal hierarchical path-finding. *Journal of game development*, 1(1):7–28, 2004.

[8] Gabriella Bretti, Roberto Natalini, and Benedetto Piccoli. A fluid-dynamic traffic model on road networks. *Archives of Computational Methods in Engineering*, 14(2):139–172, 2007.

[9] Petr Broz, Michal Zemek, Ivana Kolingerová, and Jakub Szkandera. Dynamic path planning with regular triangulations. *Machine Graphics & Vision*, 24(3/4):119–142, 2014.

[10] Moses Charikar and Sudipto Guha. Improved combinatorial algorithms for the facility location and k-median problems. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 378–388. IEEE, 1999.

[11] Keonyup Chu, Minchae Lee, and Myoungho Sunwoo. Local path planning for off-road autonomous driving with avoidance of static obstacles. *IEEE Transactions on Intelligent Transportation Systems*, 13(4):1599–1616, 2012.

[12] Juan Cortés, Sophie Barbe, Monique Erard, and Thierry Siméon. Encoding molecular motions in voxel maps. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, 8(2):557–563, 2011.

[13] Juan Cortés, Léonard Jaillet, and Thierry Siméon. Molecular disassembly with rrt-like algorithms. In *Robotics and Automation, 2007 IEEE International Conference on*, pages 3301–3306. IEEE, 2007.

[14] Juan Cortés, Duc Thanh Le, Romain Iehl, and Thierry Siméon. Simulating ligand-induced conformational changes in proteins using a mechanical disassembly method. *Physical Chemistry Chemical Physics*, 12(29):8268–8276, 2010.

[15] Kenny Daniel, Alex Nash, Sven Koenig, and Ariel Felner. Theta*: Any-angle path planning on grids. *Journal of Artificial Intelligence Research*, 39:533–579, 2010.

[16] Christian J Darken and Rene G Burgess. Realistic human path planning using fluid simulation. 2004.

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

[18] Richard Dubes and A.K. Jain. Clustering methodologies in exploratory data analysis. volume 19 of *Advances in Computers*, pages 113 – 228. Elsevier, 1980.

[19] Zhixiang Fang, Xinlu Zong, Qingquan Li, Qiuping Li, and Shengwu Xiong. Hierarchical multi-objective evacuation routing in stadium using ant colony optimization approach. *Journal of Transport Geography*, 19(3):443–451, 2011.

[20] Usama Fayyad, Gregory Piatetsky-Shapiro, and Padhraic Smyth. From data mining to knowledge discovery in databases. *AI magazine*, 17(3):37, 1996.

[21] Dave Ferguson and Anthony Stentz. Field d*: An interpolation-based path planner and replanner. In *Robotics research*, pages 239–253. Springer, 2007.

[22] Etienne Ferre and J-P Laumond. An iterative diffusion algorithm for part disassembly. In *Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 IEEE International Conference on*, volume 3, pages 3149–3154. IEEE, 2004.

[23] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345–, 1962.

[24] Roland Geraerts and Mark H Overmars. A comparative study of probabilistic roadmap planners. In *Algorithmic Foundations of Robotics V*, pages 43–57. Springer, 2004.

[25] R Glowinski, PG Ciarlet, and JL Lions. Handbook of numerical analysis: Numerical methods for fluids. 2003.

[26] Stephen J Guy, Jatin Chhugani, Changkyu Kim, Nadathur Satish, Ming Lin, Dinesh Manocha, and Pradeep Dubey. Clearpath: highly parallel collision avoidance for multi-agent simulation. In *Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 177–187. ACM, 2009.

[27] Stephen J Guy, Sujeong Kim, Ming C Lin, and Dinesh Manocha. Simulating heterogeneous crowd behaviors using personality trait theory. In *Proceedings of the 2011 ACM SIGGRAPH/Eurographics symposium on computer animation*, pages 43–52. ACM, 2011.

[28] Daniel Harabor and Adi Botea. Hierarchical path planning for multi-size agents in heterogeneous environments. In *2008 IEEE Symposium On Computational Intelligence and Games*, pages 258–265. IEEE, 2008.

[29] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[30] Carlos Hernández, Pedro Meseguer, Xiaoxun Sun, and Sven Koenig. Path-adaptive a* for incremental heuristic search in unknown terrain. In *ICAPS*, 2009.

[31] Carlos Hernández, Xiaoxun Sun, Sven Koenig, and Pedro Meseguer. Tree adaptive a. In *The 10th International Conference on Autonomous Agents and Multiagent Systems-Volume 1*, pages 123–130. International Foundation for Autonomous Agents and Multiagent Systems, 2011.

[32] David Hsu, Robert Kindel, Jean-Claude Latombe, and Stephen Rock. Randomized kinodynamic motion planning with moving obstacles. *The International Journal of Robotics Research*, 21(3):233–255, 2002.

[33] Roger L Hughes. A continuum theory for the flow of pedestrians. *Transportation Research Part B: Methodological*, 36(6):507–535, 2002.

[34] Roger L Hughes. The flow of human crowds. *Annual review of fluid mechanics*, 35(1):169–182, 2003.

[35] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: A review. *ACM Comput. Surv.*, 31(3):264–323, September 1999.

[36] Rune M Jensen, Randal E Bryant, and Manuela M Veloso. Seta*: An efficient bdd-based heuristic search algorithm. In *AAAI/IAAI*, pages 668–673, 2002.

[37] Hao Jiang, Wenbin Xu, Tianlu Mao, Chunpeng Li, Shihong Xia, and Zhaoqi Wang. Continuum crowd simulation in complex environments. *Computers & Graphics*, 34(5):537–544, 2010.

[38] Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning. *The international journal of robotics research*, 30(7):846–894, 2011.

[39] Lydia E Kavraki, Mihail N Kolountzakis, and J-C Latombe. Analysis of probabilistic roadmaps for path planning. *IEEE Transactions on Robotics and Automation*, 14(1):166–171, 1998.

[40] Lydia E Kavraki, Mihail N Kolountzakis, and J-C Latombe. Analysis of probabilistic roadmaps for path planning. *IEEE Transactions on Robotics and Automation*, 14(1):166–171, 1998.

[41] Lydia E Kavraki, Petr Svestka, J-C Latombe, and Mark H Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE transactions on Robotics and Automation*, 12(4):566–580, 1996.

[42] Benjamin King. Step-wise clustering procedures. *Journal of the American Statistical Association*, 62(317):86–101, 1967.

[43] Sven Koenig and Maxim Likhachev. D* lite. In *AAAI/IAAI*, pages 476–483, 2002.

[44] Sven Koenig, Maxim Likhachev, and David Furcy. Lifelong planning A*. *Artificial Intelligence*, 155(1-2):93–146, 2004.

[45] Hanna Kurniawati and David Hsu. Workspace-based connectivity oracle: An adaptive sampling strategy for prm planning. In *Algorithmic Foundation of Robotics VII*, pages 35–51. Springer, 2008.

[46] Vincent Lafaquière, Sophie Barbe, Sophie Puech-Guenot, David Guieysse, Juan Cortés, Pierre Monsan, Thierry Siméon, Isabelle André, and Magali Remaud-Siméon. Control of lipase enantioselectivity by engineering the substrate binding site and access channel. *ChemBioChem*, 10(17):2760–2771, 2009.

[47] Steven M LaValle. *Planning algorithms*. Cambridge university press, 2006.

[48] Maxim Likhachev, David I Ferguson, Geoffrey J Gordon, Anthony Stentz, and Sebastian Thrun. Anytime dynamic A*: An anytime, replanning algorithm. In *ICAPS*, pages 262–271, 2005.

[49] Celine Loscos, David Marchal, and Alexandre Meyer. Intuitive crowd behavior in dense urban environments using local laws. In *Theory and Practice of Computer Graphics, 2003. Proceedings*, pages 122–129. IEEE, 2003.

[50] James MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. Oakland, CA, USA., 1967.

[51] Martin Manak, Michal Zemek, Jakub Szkandera, Ivana Kolingerova, Elena Papaleo, and Matteo Lambrughi. *Hybrid Voronoi diagrams, their computation and reduction for applications in computational biochemistry*, volume 74. Elsevier, 2017.

[52] Tianlu Mao, Hao Jiang, Jian Li, Yanfeng Zhang, Shihong Xia, and Zhaoqi Wang. Parallelizing continuum crowds. In *Proceedings of the 17th ACM Symposium on Virtual Reality Software and Technology*, pages 231–234. ACM, 2010.

[53] Martin Maňák. *Application of Computational Geometry to Modeling and Visualization of Proteins.* PhD dissertation, University of West Bohemia, 2016.

[54] Ronald A Metoyer and Jessica K Hodgins. Reactive pedestrian path following from examples. *The Visual Computer*, 20(10):635–649, 2004.

[55] Adam Meyerson. Online facility location. In *Foundations of Computer Science, 2001. Proceedings. 42nd IEEE Symposium on*, pages 426–431. IEEE, 2001.

[56] Alex Nash, Sven Koenig, and Maxim Likhachev. Incremental phi*: Incremental any-angle path planning on grids. In *IJCAI*, pages 1824–1830, 2009.

[57] Alex Nash, Sven Koenig, and Craig Tovey. Lazy theta*: Any-angle path planning and path length analysis in 3d. In *Third Annual Symposium on Combinatorial Search*, 2010.

[58] Shuvra Kanti Nath, Shawna Thomas, Chinwe Ekenna, and Nancy M Amato. A multi-directional rapidly exploring random graph (mrrg) for protein folding. In *Proceedings of the ACM Conference on Bioinformatics, Computational Biology and Biomedicine*, pages 44–51. ACM, 2012.

[59] Shigeyuki Okazaki and Satoshi Matsushita. A study of simulation model for pedestrian movement with evacuation and queuing. In *International Conference on Engineering for Crowd Safety*, volume 271, 1993.

[60] OpenStreetMap contributors. Planet dump retrieved from https://planet.osm.org . `https://www.openstreetmap.org`, 2017.

[61] Nuria Pelechano, Jan M Allbeck, and Norman I Badler. Virtual crowds: Methods, simulation, and control. *Synthesis Lectures on Computer Graphics and Animation*, 3(1):1–176, 2008.

[62] Stefano Pellegrini, Andreas Ess, Konrad Schindler, and Luc Van Gool. You'll never walk alone: Modeling social behavior for multi-target tracking. In *2009 IEEE 12th International Conference on Computer Vision*, pages 261–268. IEEE, 2009.

[63] Ganesan Ramalingam and Thomas Reps. An incremental algorithm for a generalization of the shortest-path problem. *Journal of Algorithms*, 21(2):267–305, 1996.

[64] Edie M Rasmussen. Clustering algorithms. *Information retrieval: data structures & algorithms*, 419:442, 1992.

[65] Nathan D Ratliff, David Silver, and J Andrew Bagnell. Learning to search: Functional gradient techniques for imitation learning. *Autonomous Robots*, 27(1):25–53, 2009.

[66] F. James Rohlf. 12 single-link clustering algorithms. In *Classification Pattern Recognition and Reduction of Dimensionality*, volume 2 of *Handbook of Statistics*, pages 267 – 284. Elsevier, 1982.

[67] Shawn Singh, Mubbasir Kapadia, Billy Hewlett, Glenn Reinman, and Petros Faloutsos. A modular framework for adaptive agent-based steering. In *Symposium on Interactive 3D Graphics and Games*, pages PAGE–9. ACM, 2011.

[68] Jirı Skála. Algorithms for manipulation with large geometric and graphic data. Technical report, Technical Report DCSE/TR-2009-02, Dept. of Computer Science and Engineering, University of West Bohemia, 2009.

[69] Jiří Skála and Ivana Kolingerová. Accelerating the local search algorithm for the facility location. In *Proceedings of the 12th WSEAS international conference on Mathematical and computational methods in science and engineering*, pages 98–103. World Scientific and Engineering Academy and Society (WSEAS), 2010.

[70] RR Sokal. The principles of numerical taxonomy: twenty-five years later. *Computer-assisted bacterial systematics*, (15):1, 1985.

[71] Anthony Stentz. Optimal and efficient path planning for partially-known environments. In *Robotics and Automation, 1994. Proceedings., 1994 IEEE International Conference on*, pages 3310–3317. IEEE, 1994.

[72] Anthony Stentz et al. The focussed D* algorithm for real-time replanning. In *IJCAI*, volume 95, pages 1652–1659, 1995.

[73] Nathan Sturtevant and Michael Buro. Partial pathfinding using map abstraction and refinement. In *AAAI*, volume 5, pages 1392–1397, 2005.

[74] Xiaoxun Sun, Sven Koenig, and William Yeoh. Generalized adaptive a. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems-Volume 1*, pages 469–476. International Foundation for Autonomous Agents and Multiagent Systems, 2008.

[75] Xiaoxun Sun, William Yeoh, and Sven Koenig. Generalized fringe-retrieving A*: faster moving target search on state lattices. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1-Volume 1*, pages 1081–1088. International Foundation for Autonomous Agents and Multiagent Systems, 2010.

[76] Xiaoxun Sun, William Yeoh, and Sven Koenig. Moving target D* lite. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1-Volume 1*, pages 67–74. International Foundation for Autonomous Agents and Multiagent Systems, 2010.

[77] Jakub Szkandera, Ondřej Kaas, and Ivana Kolingerová. A clustering approach to path planning for groups. In *International Conference on Computational Science and Its Applications*, pages 465–479. Springer, 2017.

[78] Jakub Szkandera, Ondřej Kaas, and Ivana Kolingerová. A clustering approach to path planning for big groups. *International Journal of Data Warehousing and Mining (IJDWM)*, 15(2):42–61, 2019.

[79] Jakub Szkandera, Ivana Kolingerová, and Martin Maňák. Path planning for groups on graphs. *Procedia Computer Science*, 108:2338–2342, 2017.

[80] Xinyu Tang, Bonnie Kirkpatrick, Shawna Thomas, Guang Song, and Nancy M Amato. Using motion planning to study rna folding kinetics. *Journal of Computational Biology*, 12(6):862–881, 2005.

[81] Adrien Treuille, Seth Cooper, and Zoran Popović. Continuum crowds. In *ACM Transactions on Graphics (TOG)*, volume 25, pages 1160–1168. ACM, 2006.

[82] Prahlad Vadakkepat, Kay Chen Tan, and Wang Ming-Liang. Evolutionary artificial potential fields and their application in real time robot path planning. In *Evolutionary Computation, 2000. Proceedings of the 2000 Congress on*, volume 1, pages 256–263. IEEE, 2000.

[83] Vojtech Vonásek, Jan Faigl, Tomás Krajník, and Libor Preucil. A sampling schema for rapidly exploring random trees using a guiding path. In *ECMR*, pages 201–206, 2011.

[84] Stephen Warshall. A theorem on boolean matrices. *J. ACM*, 9(1):11–12, 1962.

[85] Krista Rizman Žalik and Borut Žalik. A sweep-line algorithm for spatial clustering. *Advances in Engineering Software*, 40(6):445 – 451, 2009.