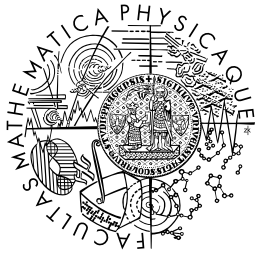


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Jan Koblížek

Group navigation in RTS games using flow networks over flow field regions

Department of Software and Computer Science Education

Supervisor of the master thesis: Mgr. Jakub Gemrot, Ph.D.

Study programme: Computer Science

Study branch: Visual Computing and Game Development

Prague 2023

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

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

In ..... date ....

Author's signature

I would like to thank Mgr. Jakub Gemrot, Ph.D. for his advice and help with my thesis. I would also like to thank my family for their support during my studies.

Title: Group navigation in RTS games using flow networks over flow field regions

Author: Jan Koblížek

Department: Department of Software and Computer Science Education

Supervisor: Mgr. Jakub Gemrot, Ph.D., Department of Software and Computer Science Education

Abstract: This thesis explores the challenges of implementing effective navigation for groups of units in real-time strategy computer games, specifically focusing on the movement of large numbers of homogeneous units across a two-dimensional grid-based map.

The thesis presents a pathfinding algorithm that could be used in RTS games. The algorithm enables the units to utilize multiple paths effectively by modeling the unit navigation as a flow network problem. The units use precomputed flow fields during the navigation. This allows for faster pathfinding times by offloading part of the computation to the preprocessing.

The algorithm's performance is evaluated against a baseline solution using the A* algorithm and other existing solutions. Comparative analysis will be conducted utilizing maps from the Moving AI 2D Pathfinding Benchmark dataset to assess the efficacy of the proposed solution.

Keywords: pathfinding, group navigation, flow networks, RTS games, flow fields

Contents

Introduction

This thesis deals with the topic of navigating groups of units in computer games, specifically in the real-time strategy game genre (RTS): a genre of strategy games where the state of the game changes in real-time (this differentiates it from turn-based games). Popular games in this genre include Starcraft, Warcraft, Age of Empires, and the Command and Conquer series.

In RTS games, the player can command their units to move to a certain goal location. The units then have to navigate to the goal autonomously. Navigation in this type of games faces several problems that don't allow the use of the basic versions of the common pathfinding algorithms. The units move over a map that can change during the gameplay. The players can usually construct buildings obstructing parts of the map or alter the terrain itself (for example, by cutting down the trees in the Age of Empires series). The units have to be able to react to these changes, even if the map changes during their movement, and alter their plans accordingly. The player also expects the units to be responsive to their commands. Therefore, The pathfinding has to finish quickly or provide the units with an approximate movement plan, which will be further refined when the units are already moving.

RTS games are doing pathfinding not only for single units. The player gives movement orders to armies that can consist of hundreds or even thousands of units. The best path for a single unit will often not be optimal for a large group of units, and finding a path for each of the units individually will take too much time and computing power. The units also collide with one another and may block each other's paths. Ideally, the units need to be able to react to such situations without having to recompute their path.

The games often organize units into groups based on their common movement targets and proximity to each other. The group of units can then be assigned a single path. This is often not optimal because the units aren't allowed to occupy the same position at a single point in time. This is best visible on maps with a lot of narrow passages. The units collide with each other and block each other's paths. The units often have to wait a long time before their path is clear and they can pass through the chokepoint. It would often be quicker for a part of the units to choose a different longer path, where they wouldn't have to spend as much time waiting for the other units (Figure 1).

Games attempt to solve this problem in various ways. Age of Empires II, for example, temporarily decreases the size of the unit colliders so the whole group can pass through the narrow section without the units blocking each other. The older Age of Empires I forces the units to recalculate their path if the original path is blocked by other units (if the other units aren't moving).

An interesting method to solve this problem was proposed by Jan Pacovský[1]. The proposed algorithm based on flow networks was called a "Flow Graph." The map is transformed into a flow network with the nodes representing the chokepoints. The network is then used to find the quickest way of transporting all the units to the target, utilizing multiple paths. In this thesis, we will build on Pacovský's work and improve the Flow Graph algorithm, using flowfields precomputed for different regions of the map. This algorithm, called "Regional Flow

Units are trying to move through a chokepoint between two buildings. The units accumulate at the entrance and have to wait until they can pass. It would be quicker for a part of the units to choose a different path.

Figure 1: Starcraft 2 Narrow Passage Problem (Image Source: Akker et al.[2])

Graph," will be evaluated against common pathfinding methods and the basic Flow Graph method.

Goals

The main goal of the thesis is to propose an algorithm that will make the units effectively utilize multiple paths. This should help us avoid the problem we illustrated for Starcraft 2.

Our algorithm, called Regional Flow Graph (RFG), should be able to solve or at least reduce the severity of the common problems from the original algorithm, namely the units getting carried off their path or a wrong separation of unit groups. The unit groups using different paths should be able to separate from each other efficiently, and there shouldn't be many conflicts between the unit paths.

The algorithm should compare well to the standard navigation methods and to the original flow graph algorithm. The most important points of comparison are the time it takes the units to reach the goal (in-game time) and the time it takes the program to do the pathfinding.

We will compare the RFG algorithm's performance to other algorithms (A*, flow field, flow graph) on various metrics. We will also try to measure various characteristics of the method. The real performance of the method will be compared to the one predicted by our model.

We will discuss the problems and inaccuracies of our methods. These will

point us to future research or directly to the possible improvements to our algorithm.

Thesis Structure

The thesis is separated into several chapters. This section will briefly introduce the contents and the purpose of these chapters.

1. First, we will introduce the problem of pathfinding in RTS games. We will define an abstract environment that will be used to run pathfinding requests. We will also define the objectives that a good pathfinding algorithm should achieve.

2. The second chapter will go through the literature related to pathfinding with a focus on pathfinding in RTS games.

3. In the third chapter, we will analyze the problem of navigating a large number of units across the map. Based on our understanding of the problem, we will introduce a theoretical model describing the pathfinding problem and its solutions.

4. The fourth chapter will introduce the Flow Graph algorithm on which we base our proposed method. We will describe the problems with the original algorithm and propose a new Regional Flow Graph algorithm that will attempt to fix them.

5. The fifth chapter will describe the experiments we will run to evaluate our algorithm. It will also describe the testing environment we created and the metrics gathered in our tests.

6. The sixth chapter presents the results of the evaluation. It compares the RFG algorithm's performance to other methods and describes possible problems with our method.

7. The seventh chapter describes possible improvements to our algorithm. It also discusses possible modifications to the algorithm if we want it to achieve different objectives than those considered in the thesis.

8. The conclusion chapter provides a short discussion of the strengths and weaknesses of our algorithm and its possible use in RTS games.

1. Problem Statement

We want to propose a pathfinding algorithm that could be effectively used in RTS games. The RTS game genre covers a wide range of different games, each with different characteristics and requirements. Popular RTS games are proprietary and not open source. We also don't want to target any game in particular but create a pathfinding algorithm relevant to a wider range of games. For this reason, we will create a testing environment providing an abstraction of an RTS game.

We will select a few objectives that will be used to evaluate the performance of our algorithms. The selected objectives will represent some of the common requirements for a good pathfinding algorithm.

1.1 RTS Game Abstraction

We will create a testing environment (simulator) that will provide an RTS game abstraction. The model of the game provided by the abstraction will be simplified compared to a real game and will deal only with unit navigation ignoring things like combat. We will also simulate only one movement order to a single group of homogeneous units (same size and movement speed).

1.1.1 Map

In our testing environment, we represent the map as a two-dimensional grid of tiles with uniform movement cost. Each tile has a boolean passability value determining whether it can be entered by the units (the impassable tiles are called the walls) and 8 neighbors (4 of them on the diagonal). For every neighbor, we can determine its distance (1 for the straight-direction tiles and 2 for the diagonal tiles) and whether the unit can move there (the tile must be passable, and for the diagonal tile, the two neighbors next to it must be passable as well). The tile with distances to the neighbors can be seen in the Figure 1.1. The passability and distance values do not need to be stored because they can all be easily computed based on passability values.

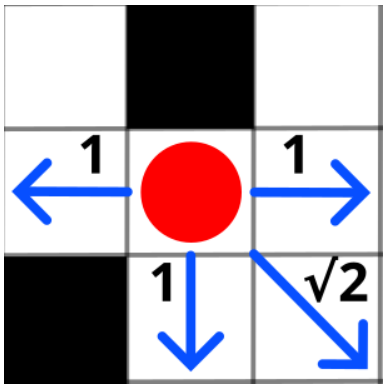
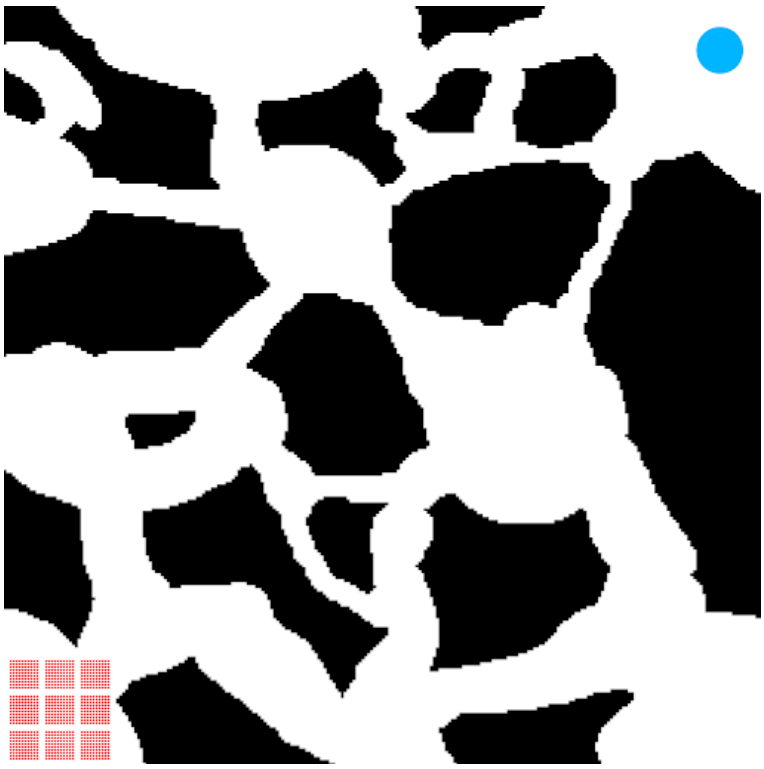


Figure 1.1: Single tile with neighbors and distances to them

1.1.2 Units

The units in RTS computer games can have different collider shapes, with rectangle and circle shapes being the most common. Different units may also have different shapes and sizes. We will use a single type of unit with the shape of a circle.

The unit can occupy any place on the map where it doesn't overlap with any wall tile or other unit. The unit's position can be specified in real numbers by its



The pathfinding problem consists of a map with impassable tiles (black) and passable tiles (white), homogeneous circular units (red) at their starting positions, and a circular target (blue)

Figure 1.2: A single pathfinding problem

x,y coordinates.

The unit navigation in RTS games deals with large groups of units representing whole armies. We will reflect this in our testing environment. An average map used in our simulator will contain between 1000 and 2000 units. All of the units on the map represent a single group of units, and their starting positions are in close proximity to one another.

1.1.3 Pathfinding Problem

In RTS games, the player can select a group of units and order them to move to any navigable position on the map. The game then creates a plan the units can use to navigate to their desired position. The units then autonomously follow the provided plan and are able to react to any dynamic changes that may occur during their navigation.

In real games, multiple groups of units may navigate to different target locations simultaneously. The interactions of different unit groups are outside our work's scope. We will simulate only one movement order given to a single group of units. Our simulator represents this single movement order as a pathfinding problem. The problem is specified by the map, starting positions of the units, and the target the units should navigate toward (Figure 1.2).

The simulator is then provided with a method that will be used to solve the pathfinding problem. The Regional Flow Graph will be one of the simulator's methods, along with methods against which we will compare its performance. The work of the navigation methods can be divided into three phases.

- Preparation phase - This phase can be used to analyze the map or to construct search structures. At this stage, we know neither the units' starting positions nor the movement target. Multiple pathfinding requests will reuse the structures created at this stage.
- Pathfinding phase - The program receives the starting positions of the units and the target they should move to. A plan is created for the units to follow.
- Execution phase - The units are following the plan created in the pathfinding phase and navigating toward the target

The best plan for a single unit may not be the best for hundreds or thousands of units. The paths used by the individual units may come into conflict. The units may collide or block each other's paths. The pathfinding with a large number of units, therefore, calls for different than standard pathfinding methods.

1.2 Objectives

There are many requirements for a good pathfinding algorithm. These requirements may also vary for different games or even different players. One algorithm could, for example, provide a plan which would result in the units reaching their target faster but with higher casualties. Some requirements may also be hard to quantify, e.g., units moving in a natural way.

In our thesis, we will focus on two requirements/metrics that we consider the most important.

- Finishing time - the time it takes until all units reach the target. The player presumably wants the units to execute the movement order quickly.
- Pathfinding time - The time it takes for the algorithm to create the plan the units will follow. Units should react to the player's order almost instantly.

These two requirements go against each other. Longer pathfinding times would allow the algorithm to create a better plan resulting in a faster finishing time. Our pathfinding algorithm should achieve fast times in both of these metrics. We will compare the performance of our algorithm to the original Flow Graph algorithm as well as to the A* and flow field methods that are commonly used in RTS games.

2. Related Work

This chapter will introduce common methods used for navigation in RTS games. First, we will describe the structures used to represent the map. The next three sections will then give an overview of the methods used in the three navigation phases (preprocessing, pathfinding, and execution).

2.1 Representation of the Game World

We need to have a structure representing the game world. The pathfinding will then be performed on this structure. There are many representations possible for a single map. Each of them will, however, have different characteristics. Our choice will influence the quality of the paths generated in the pathfinding phase as well as the time it takes to create a plan.

2.1.1 Grid Representation

The map is seen as a grid of 2D cells. Square cells with 8 possible neighbors (4 diagonal) are the most typical in the RTS genre. Other cell shapes, like hexagons, are used less often. The chosen shape of the cell influences the performance of the pathfinding algorithm[3]. The cells can be marked as passable or not passable, or they can each have a specified movement cost (in our solution, we use the same cost for all of the passable tiles).

This grid is probably the most natural and the simplest way of representing an RTS game world. Many games are played directly on the grid, so this representation comes out of the box. It is also easy to use because of its regular shape of tiles and an unchanging number of neighbors. The grid will, however, typically contain a large number of cells, resulting in a slower pathfinding phase.

2.1.2 Waypoint Graph

In some sense, the grid was a special case of another structure – the waypoint graph. This structure corresponds to the weighted graph structure known from discrete mathematics. The nodes in the graph represent positions on the map, and the nodes with a path between them are connected by an edge with the weight specifying the length of the path.

The number of nodes in this method is typically significantly lower than in the grid representations, but their representation is more complex (the number and the length of the connected edges vary between different nodes). Preprocessing or manual placement by a designer is usually needed to obtain this structure. The paths resulting from this method also typically won't be optimal.

2.1.3 Navigation Mesh

In this method, the walkable areas of the map are covered by a polygon mesh called navigation mesh, or navmesh[4]. The polygons of the mesh have to be convex. This means a direct walkable path exists between any two points inside

a) Waypoint Graph b) Navigation Mesh

Figure 2.1: Same map is represented using the waypoint graph and a navigation mesh (Image Source: Tozour [5])

a polygon. Some versions use only triangles or have a limit on the maximum number of polygon sides.

A single polygon usually covers a large number of tiles making the search space smaller for the pathfinding algorithms using it. Navmesh can also be used to represent non-tile-based game worlds. Unlike the waypoint graph, it contains information about all the walkable parts of the map, and the pathfinding algorithms are able to utilize it (Figure 2.1). This makes the paths found on the navmesh usually shorter.

The navmesh can be placed manually by the designer or automatically generated[6]. Navmesh isn't a natural representation of the game world and therefore has to be recalculated when the map changes.

2.1.4 Hierarchical Representation

Different map representations have different advantages. Less complex ones allow for quicker pathfinding, but the paths may be far from optimal. More detailed ones will provide more accurate results, but the computation will take longer. Hierarchical representations try to combine the advantages of both.

These structures typically consist of several levels, each providing a different accuracy of the world representation. Higher levels provide less detail, while the lower levels represent the world as accurately as possible.

The pathfinding is first done on a higher level. The lower levels can then use this less accurate path during their pathfinding. The path can be used to provide a better heuristic for the algorithm running on the lower level. The lower level pathfinding can also just find the paths between the points of the higher level path.

Our algorithm uses a hierarchical map representation. We use a flow network (a modified version of the waypoint graph) on a higher level and the grid representation on a lower level.

Figure 2.2: Regional decomposition

2.2 Preprocessing

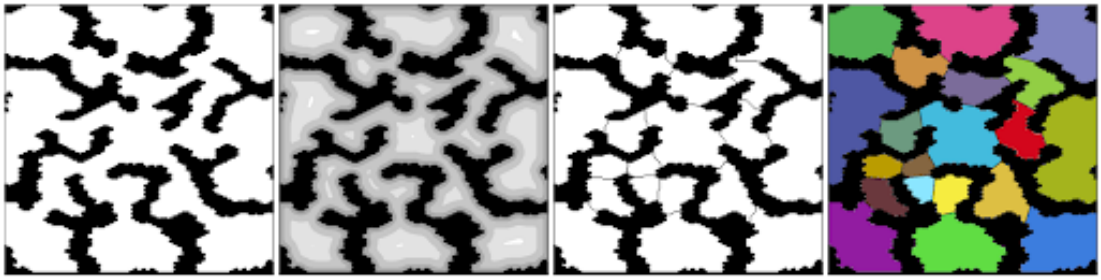
Map preprocessing is performed while the game is loading or during the gameplay in the background over multiple frames. Because of this, the time constraints for the computations aren't too strict. The preprocessing doesn't react to any of the player's orders and lacks some of the information necessary for path creation, like the positions of the units and the goal.

This step, therefore, usually computes the information that multiple pathfinding requests can later reuse. Or simplifies an otherwise complex pathfinding space.

2.2.1 Regional Decomposition

Regional decomposition is used to divide a map into regions (Figure 2.2). The regions should correspond to larger open spaces, separated by narrower sections (chokepoints). The decomposition has various uses. Hierarchical map representations can use the regions as their higher-level map abstraction, which can be used to come up with a heuristic for pathfinding. Pathfinding can also be done directly over the regions.

The borders between the regions correspond to the map's chokepoints. These places can be important for the game's AI. Narrow sections are excellent places to defend against enemy attacks or place defensive structures. Richoux et al.[7] use the chokepoints to find the best places for the wall placement. These typically lie near the chokepoints but not directly at them because they are often on non-buildable (but walkable) terrain. Oliveira et al.[8] use them in a similar way, also taking into account the position of the player base and the resources in the region.



from the left: original obstruction map, height map, regional map with the region borders, map with the gates and regions

Figure 2.3: Water level decomposition process (Image Source: Halldórson[9])

In our algorithm, we use Halldórson's water level decomposition[9], which will be introduced in the first subsection. Other regional decompositions will be introduced in the second subsection.

Water Decomposition

Halldórson's water level decomposition[9] is an efficient and easy-to-understand regional decompositions method inspired by rising water. The algorithm works in several steps (Figure 2.3).

1. First, a height map is created with each grid tile containing a distance to the nearest wall (obstructed tile). Intuitively, the tiles furthest from the wall have the lowest elevation level. The algorithm examines tiles at some radius. If a certain percentage of these tiles is obstructed, the radius is used as a distance to the wall.

   Otherwise, the examined radius gets increased, and the process is repeated until it returns the distance. By working with a percentage of tiles instead of returning a distance of the closest obstructed tile, the algorithm effectively smooths over the obstacle edges and reduces the number of regions generated in later steps. However, it pays for this improvement in an increased computational time.

2. The height map is then used to generate regions and gates separating them. The map is filled with water starting with the tiles at the lowest elevation (the biggest distance from the walls). When a tile is filled with water, there are three possibilities.

   - The tile doesn't have any neighbors filled with water: A new lake is created with a center at this tile. The lakes correspond to the regions of the map and are given unique ids
   - The tile neighbors one existing lake: The tile is added to the lake (the region)
   - The tile neighbors multiple lakes: The tile is added to one of the lakes (certain directions are prefered)

3. The algorithm finds tiles that neighbor a tile with a different region than their own. These tiles are marked as gate tiles. We have generated gate clusters separating the regions.

4. The gate clusters generated in the previous step do not typically form a straight line. The algorithm selects two gate tiles bordering a wall and creates a gate as a straight line between them. The tiles around the gate have their regions reassigned such that the regions are separated by the gate.

The map with regions and gates can then be further processed into a graph that gives a simplified representation of the map and can be used to get estimates for the distances between points on the map.

Other Decompositions

Bidakaew et al.[10] start with the generation of the depth map, just like the water decomposition. Their algorithm then generates a medial axis for the map and marks its important points (junctions, endpoints). The depth values on the paths between two points are analyzed. If there is a local minimum with high enough depth, a gate is created at the corresponding position. The algorithm can mark not just gates but also chokepoint areas (when there is a larger plateau near the local minimum).

Perkins[11] proposed a decomposition algorithm and implemented it as a library called BWTA (Brood War Terrain Analyzer). It first computes a Voronoi diagram of line segments from the edges of the obstacles. The diagram is then pruned to speed up the computation. The algorithm then identifies the region and chokepoint nodes on the diagram (chokepoints have 2 neighbors). The algorithm then merges nearby regions if the radius (distance to the closest wall) of the chokepoint between them is close to the radius of one of the regions. Finally, the narrowest sections of the map near the chokepoints are found and walled off (marked as gates). Uriarte and Ontañón[12] greatly improve the speed of this algorithm and implement it as the BWTA 2 library.

Richoux[13] proposes a decomposition algorithm more specific for Starcraft and implements it in the TAUNT library. His algorithm doesn't just consider the walls when identifying regions but also the elevation, buildability, and presence of resources. This means that two mineral sources should be in separate regions, but two parts of the map separated by a chokepoint could be considered a single region.

Yijun et al.[14] decomposes the map into regions using a quad-tree. Each node of the tree represents a square section of the map and its children four smaller subsections within it. The section is split into subsections when there is a certain percentage of obstacles in the section (completely passable or obstructed sections do not need to be split). The algorithm precomputes distances between the tiles at the edges of the regions and uses them to speed up the pathfinding.

2.2.2 Path Precomputation

Some map preprocessing methods precompute information that can be used to speed up the pathfinding phase of the navigation.

Uras et al.[15] mark the tiles at the wall corners as subgoals. The subgoals are connected to each other if they are mutually visible, and their distance is computed. The path obtained during the planning phase then consists of the start, goal, and sequence of subgoal nodes.

Botea[16] computes for each of the map tiles the first move direction toward every other tile. These directions are then stored as a tree of rectangles, where the rectangles in the leaves encompass only tiles with only one direction.

2.3 Pathfinding

The navigation algorithm has received the unit starting positions and the target they should move to. A plan is created for the units to follow. The structure of the plan will depend on the pathfinding method used but is always some structure describing the path the units should follow toward the target.

Standard pathfinding methods include Dijkstra, A* and its variations, JPS, various hierarchical methods, and others. These algorithms will be described in the following subsections.

2.3.1 Dijkstra

Dijkstra's algorithm[17] is a pathfinding algorithm that used to be popular in the game development industry but was later, in most cases, replaced by other algorithms like A*. Dijkstra is an uninformed search algorithm which means it doesn't utilize any information about the search space beyond the way it can traverse it.

The algorithm can be used in situations when we need to explore the entire search space, such as in the case of the creation of flow fields. The flow field creation is also where we use Dijkstra's algorithm in our program.

```
1 OPEN ← s
2 While !OPEN.Empty:
3     v ← OPEN
4     If v == g:
5         break
6     ForEach neighbor n of node v:
7         If d(n) > d(v) + l(v,n):
8             d(n) = d(v) + l(v,n)
9             OPEN ← n
```

Pseudocode 1: Dijkstra's algorithm

The algorithm (Pseudocode 1) runs on a graph with a starting node s and a goal node g. Each node has a distance h(n), which is set to infinity for all nodes except s at the beginning. There is also a function l(a,b), which returns the length of the edge between the nodes a and b. The algorithm maintains a priority queue OPEN with all the nodes the algorithm encountered but didn't close. The nodes in the queue are sorted based on their distance.

The algorithm selects the first node from OPEN and closes it. If the selected node doesn't belong to the goal, it is closed, and the algorithm inspects all of its neighbors. For each of the neighbors, it adds it to the OPEN (if it isn't already there or isn't closed) and calculates its distance from the start. If the distance of the neighbor decreases, the stored distance value is updated, and the selected node is set as the neighbor node's predecessor.

If the algorithm reaches the goal, a path is retrieved using the chain of node predecessors. The algorithm can also terminate if the OPEN queue contains no more nodes. In such a case, the algorithm determined that the path doesn't exist.

2.3.2 A*

The A* algorithm[18] can be seen as an informed version of the Dijkstra's algorithm. The algorithm tries to first expand the nodes in the direction of the target. This is achieved by using a heuristic function h(v) that estimates the distance from the node v to the target. Heuristic usually leads to the algorithm exploring a lower number of nodes. This, however, isn't always the case and depends on how well the heuristic estimates the distances.

```
1 d(n) = d(v) + l(v,n) + h(n)
```

Pseudocode 2: The modification changing Dijkstra's algorithm to A*

A* performs its search on a graph, and when using an admissible heuristic, it is guaranteed to find an optimal path. The graph (or grid) the A* operates on, however, won't be a perfect representation of the map, and the path found won't typically be optimal.

A* performed directly on the grid is one of the algorithms we compare our Regional Flow Graph method against. The Regional Flow Graph also uses A* in one of its steps.

Heuristics

There are two properties we are interested in when it comes to heuristics - admissibility and consistency. A consistent heuristic estimate is always less than or equal to the estimated distance of any adjacent tile plus the cost of reaching that adjacent tile. Consistency is an important property because an inconsistent heuristic may lead to the opening of already closed nodes. Admissible heuristics always estimate a lower than the real distance. The inadmissible heuristic may lead to suboptimal paths. Despite this, inadmissible heuristics may be useful because they are often able to find the path faster (even if it isn't guaranteed to be optimal).

The most common heuristic used for A* is the direct distance to the goal ignoring the obstacles. This can be computed as the Euclidean distance, manhattan distance, or octile distance. Better heuristics may be obtained from map preprocessing.

Goldberg and Harrelson[19], for example, place a number of points called landmarks on the map. They then precompute the distances from every tile to these points. The distances can then be used as a heuristic for the A* search.

Heuristics can also be obtained by first performing a search on a higher-level abstraction of the map. These will be mentioned in the [link Hierarchical Pathfinding]

A* On Grids

When we perform the A* pathfinding on a grid, we are guaranteed the optimal path. However, in reality, the grid represents an open space for which the path isn't optimal. This is because the pathfinding only considers the movements in 8 directions. The agent in the game can reduce the length of the path by moving at a different angle than one of the 8 allowed directions.

The length of the path can be improved by removing unnecessary points. If two of the points on the path are mutually visible, the points between them can be removed. This can yield a shorter path.

An even shorter path can be obtained by using so-called any-angle pathfinding. These algorithms allow for edges of different lengths and angles. Both nodes at the end of the edge must be mutually visible.

Nash and Koenig[20] propose an any-angle algorithm called Theta*. When the neighbor of the currently checked node v is processed. Its distance is estimated using both the node v as well as parent(v), and both can be set as the neighbor's parent. The resulting paths are, on average, 4% shorter than A* paths.

Harabor et al.[21] proposed an algorithm called Anya, which can find optimal any-angle paths. The algorithm searches over intervals instead of the grid nodes. An interval is a continuous section of one of the grid's rows.

The paths computed on a grid have the form of polygonal chains. Such paths do not look natural, but they can be turned into a smooth spline[22].

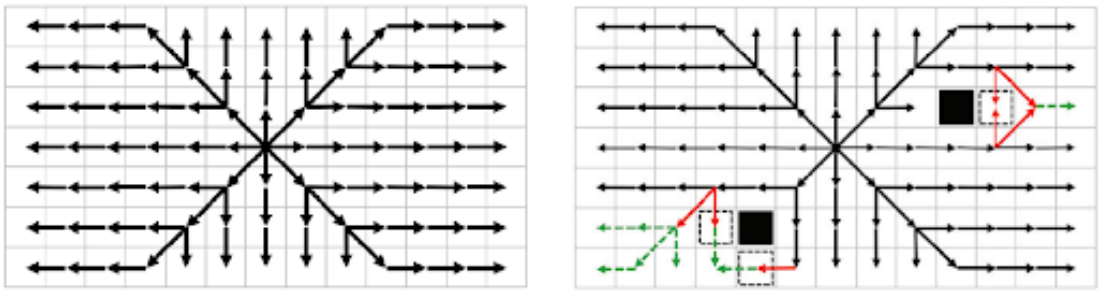
A* On Navigation Meshes

The A* is often a part of the pathfinding on the navigation meshes. The A* works on graphs, so the navmesh must be seen as a graph. This can be done in several ways. The nodes of the graph may be placed at the centers of the polygons, at the centers of the edges, or at the vertices. The resulting graph isn't too different from a waypoint graph A* can easily use to find a path.

The path obtained using A* can be used directly[4], but it produces a jagged path and doesn't have any advantages over the pathfinding on the waypoint graph.

In newer navmesh pathfinding algorithms, the A* only selects the polygons the final path should go through. The final path is then found using the funnel algorithm[23]. This path is optimal within the selected polygons, but there may still exist a better path going through different polygons.

Demyen and Buro[24] propose an algorithm called TA*, which finds an optimal path by iterating over different polygon paths. Each of the paths has a length estimate (smaller than the real length of the path). The real path is then obtained using the funnel algorithm, and its length is measured. The shortest path so far and its length are stored. If the estimate of the polygon path length is lower than the already found path, we know that we have found the optimal path. TRA* algorithm works in a similar way but reduces the size of the navmeshes graph representation by removing unnecessary nodes.



Left: The exploration in the canonical order, Right: algorithm exploring a map (modified version - diagonal moves not allowed at obstacle corners)

Figure 2.4: JPS - canonical exploration and jump points (Image Source: Rabin and Sturtevant[30])

Cui et al.[25] present an algorithm called Polyanyna, a modified version of the Anya algorithm used on grids. Just like its grid version, it performs a search over intervals, not nodes. An interval is a continuous section of one of the polygon edges.

Enemy Units

The length of the path found by the planner often isn't the only criterion for a good path. The game worlds often contain enemies which the agent wants to avoid. Pan[26] estimates the probability of the enemy's presence at a certain tile and appropriately increases its movement cost. Bayili and Polat[27] modify the heuristic function by factoring in the expected damage on the direct path toward the goal. The paths which resulted in the unit's death are also pruned.

IDA*

Iterative deepening A* or IDA*[28] modifies the A* algorithm by setting a depth limit. A node n can only be added to the queue if its distance d(n) is smaller or equal to the limit. If the algorithm doesn't manage to find the path within a specified depth, the limit is increased. The main advantage of IDA* over A* is that it consumes less memory.

2.3.3 JPS

On uniform cost grids, the A* algorithm generates paths that may differ in the move order but are of the same length and have the same ending and starting points. The jump point search algorithm[29] tries to deal with this redundancy and explore as few of these paths as possible.

For this purpose, the algorithm has pruning rules. JPS specifies a canonical movement order for a path between two points on the map (Figure 2.4). First, move diagonally, then straight. When expanding a node, we check for every neighbor if we are reaching it using a canonical order. If this is the case, we

add it to the priority queue. Otherwise, we check if we can reach it using a canonical order from the node's parent. If it can be reached, it is ignored (a different path will be used to add it). When a canonical order cannot reach the neighbor, we mark the node as a primary jump point (a place where the canonical movement order may change). Secondary jump points will be the nodes, where the canonical path changes between the diagonal and cardinal direction while moving to a primary jump point.

The paths found by the algorithm will be a sequence of jump points between the start and the target nodes. Each of them can be reached directly from its predecessor.

JPS+[31] precomputes for every map tile the first jump point node that can be reached by moving in one of the eight directions. This way, we can immediately find out all neighboring jump points neighboring any point on the map.

Rabin and Sturtevant[30] improve the JPS+ search by adding a bounding boxing map preprocessing. For every node and movement direction, it stores a bounding box encapsulating all the tiles for which the optimal path toward them starts with the said direction. During the search, we can ignore jump points in the directions whose bounding boxes don't contain the goal.

2.3.4 Hierarchical Pathfinding

HPA*[32] optimizes pathfinding by constructing an abstract graph over the grid. It divides the world into square sectors of a fixed size. For each of them, the algorithm finds entrances (groups of passable tiles bordering another region) and places a node at their centers. The nodes neighboring the same sector are then connected with an edge of the length equal to the length of the path between them (if such a path exists). During the plan creation, the algorithm uses A* to find a path on the abstract graph. The grid path is then created by connecting the centers of the entrances selected by the pathfinding on the abstract graph.

Li et al.[33] improve HPA* by dividing the world into rectangles instead of squares. Picking them such that their borders contain as many wall tiles as possible.

Factorio[34] divides the world into 32*32 chunks and, for each chunk, places an abstract node for each of the components within it. The path is first found over the nodes and then used as a heuristic for the search directly on the grid.

2.3.5 Flow Field

Flow field is a structure that, for each of the map's tiles, contains a movement direction. It is typically constructed by building a distance field that contains the tile distances to the goal and then retrieving the movement directions as its gradient.

Flow field navigation is one of the methods against which we compare our algorithm. Our algorithm also utilizes flowfields.

The flow fields can contain information other than just the direction toward the goal. Hagelbäck[35] uses different fields describing the target direction, enemy positions, resource positions, and others. The fields are then combined, and the

movement direction is obtained. The combination depends on the unit (workers should avoid enemies, while the soldiers should attack them).

In another paper, Hagelbäck[36] combines the A* and flow field navigation. The flowfields are used near enemy units choosing the best direction to attack. This increases the unit group's battle performance while using less computationally demanding A* to obtain longer paths.

2.3.6 Flow Field Tiles

Supreme Commander 2 uses an interesting algorithm[37] somewhat reminiscent of HPA* but using flow fields. The world is divided into 10x10 regions. For each of them, portals (groups of passable tiles bordering another region) are found, and distances between them are computed.

When a path is requested, it is first computed using the portals. This is done for each unit. The search is optimized by attempting to merge different unit paths together.

For every portal on the path, the region behind it computes a flowfield heading toward it. Many units typically request a single flow field that can be computed only once.

The result can be smoothed over by starting the computation of the regional flowfield not directly from the portal but from another portal further on the path.

The flowfields can be reused from earlier requests or precomputed.

2.3.7 Multi-Agent Pathfinding

The RTS pathfinding, and by extension, our thesis, deals with large numbers of units. We will encounter situations when another unit blocks the unit's path or their two paths cross. There are several multi-agent pathfinding methods that can be used to deal with this kind of problems.

One area of research that may offer a solution is multi-agent navigation. Silver proposes an algorithm called Cooperative A* or CA*[38]. CA* works similarly to the basic A*, but adds another dimension to the search space - the time. The units compute their path and reserve the cell they are passing through for a certain period of time. The cell can not be used by another unit at the same time, and it has to either choose a different route or wait till the cell is freed. The algorithm solves a wide range of pathfinding problems. The units do not have to have the same target, origin, or starting time. The search, however, has to be performed separately for every unit, and adding the time dimension greatly increases the size of the search space.

Silver[38] offers some improvements. The units can find their path, ignoring the time dimension and the reservations. The algorithm then does the pathfinding in the search space with the added time dimension and uses the path that ignores other agents as a heuristic. The algorithm also limits the depth of this search to save computational power and returns only the path prefixes. The article calls this improved algorithm WHCA*.

Erdem et al.[39] define a formal framework that can be used to find a solution to multi-agent pathfinding problems using answer-set programming. The algorithm receives a description of the problem (the map, each agent's starting

position, and target) and a set of constraints, some of them optional (Can agents pass through each other? Can they wait?...). The framework allows us to describe and obtain solutions for a wide range of problems and agent models. However, pathfinding with even 20 agents takes seconds. The Starcraft 2 screenshot used in the introduction (link), for example, contains 60 units.

Alborz et al.[40] introduce an algorithm called biased cost pathfinding. The agents are given a number specifying their priority. Each agent has defined a biased cost function value for every tile of the map, which increases its cost. The algorithm works in iterations. Every unit computes its path using A*. Some of these paths will come into conflict. In such cases, the algorithm increases the biased cost function values for all units involved in the conflict except for the one with the highest priority at the tiles near the conflict location. The process is then repeated again until there are no more conflicts or until a certain time limit is reached.

All of the algorithms mentioned in this section are able to solve problems of the multi-agent pathfinding. This includes problems where there are different goals for different units. Despite this these methods aren't a good fit for our problem mainly because of their long pathfinding times. We are also only solving pathfinding problems with a single goal, which allows us to do optimizations that cannot be used by the algorithms mentioned in this section.

2.3.8 Planning Based On Flow Networks

Many of the conflicts between the unit's paths that occur when doing navigation for multiple units do not have to be resolved in the pathfinding phase. The execution phase will be able to deal with most of the conflicts. The conflicts that can not be resolved during the execution typically occur when a large group of units tries to push through a narrow chokepoint. In such cases, the units have to either wait or choose a path going through a different chokepoint.

For this reason, there are group pathfinding methods that do not try to prevent all conflicts between the unit paths. Instead, they try to limit the number of units passing through some sections of the map and assign alternative paths to parts of the group.

Akker et al. propose a method that sees the map as a flow network[41]. A waypoint graph is constructed over the map, with each edge having an assigned capacity - the number of units that can pass through it per second. The method supports multiple groups of units with different original positions and different navigation targets. The pathfinding problem is solved as a multi-commodity flow problem. The article, however, doesn't specify how the flow network should be created nor how the edge capacities can be computed. It also doesn't evaluate the method's performance against the standard pathfinding algorithms like A*.

Jan Pacovský's Flow Graph method[1], which this thesis is based on, solves the problem in a similar way, but only for a single unit group and target. It specifies a method of constructing the flow network, compares the algorithm's performance to other navigation algorithms, and achieves better planning speeds. The Flow Graph method will be described in detail in the Implementation chapter.

2.4 Execution

In the execution phase, the units follow the plan supplied by the pathfinding phase. The execution is called every frame to calculate the unit's new position. This is done until the unit reaches its goal. During the execution, the units try to follow their pre-computed paths and try to avoid collisions with the walls or other units. The unit movement happens in discrete steps, with a new position in every frame.

Errors may occur during the execution. The units may get stuck, or their pre-computed path may be obstructed (maybe someone built a building that blocks the path). The units have implemented control mechanisms that detect such situations and solve them (for example, by recalculating the unit's path).

2.4.1 Steering Behaviors

The units used in our abstract game model use steering behaviors during their navigation. The steering behaviors[42] direct the unit movement during the execution. They provide reactive steering based on sensory input from the environment. The input from the environment is typically limited to the unit's small neighborhood. The steering behaviors consider each unit independently, and the units do not communicate with each other.

The steering works with a physical of the unit model. The unit has size, position, momentum, and other properties. Different steering behaviors provide forces that influence the unit's momentum. Each force represents some primitive steering behavior, and their combination can create more complicated behaviors. We will introduce a list of some of the commonly used steering behaviors.

Seek - The unit is trying to move to some target. This can be a point on its path, a unit it is pursuing, or a direction specified by a flow field.

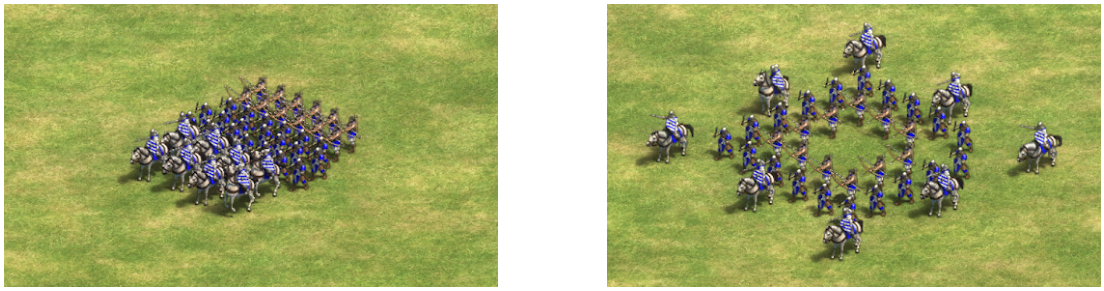
Separation - The unit should keep a certain distance from other units. Nearby units, therefore, exert a repulsive force. The closer they are, the greater the force's magnitude.

Alignment - The unit is trying to copy the behavior of the nearby units. The unit tries to have the same momentum as the nearby units.

Cohesion - The unit tries to reach the center of gravity of the nearby units.

Obstacle Avoidance - The unit tries to avoid a collision with the walls. The force generated depends on the direction toward the wall and its proximity.

Unit Collision Avoidance - The unit tries to avoid collision with other units and adjusts its movement direction accordingly. There are many methods for collision avoidance. The notable ones include RVO, ORCA[43], and Clearpath[44]



a) Row Formation b) Box Formation

Figure 2.5: Age of Empires 2 formations

2.4.2 Formations

The execution doesn't always work with individual units. Some methods coordinate the movement of groups of units. A popular way of implementing a movement of the groups of units is to organize them into formations. The formations aren't used in our abstract game model and are mentioned only as one of the important approaches to group navigation.

The formation is an arrangement of units, where each unit receives a relative position. The units are then trying to maintain this position. The positions may be fixed and unchanging. It can also just be a preferred position toward which the unit will receive a steering force. Formations limit conflicts between the unit paths because the whole formation moves in a single direction. It may also decrease the computation requirement because the whole formation follows a single path. The armies throughout history often fought in rigid formations, and adding them to the game may sometimes add to the game's realism.

Age of Empires 2 was one of the first games to use formations[45] (Figure 2.5). The player could choose from different formation types, which contained up to 40 units. The formations differed in their shape and the positions assigned to different unit types. For example, in the box formation, the ranged units would be surrounded by the melee units.

The formation's movement speed was equal to its slowest unit. When passing through a chokepoint smaller than the formation, the unit collision radius was turned off. This allowed the formation to pass through without problems but somewhat decreased the realism.

3. Problem Analysis

Before introducing our algorithm, we will introduce a theoretical model of group pathfinding. We will define the pathfinding problem for group navigation and analyze its possible solution.

3.1 Map

A map is a grid of squared tiles with a structure as described in the Problem Statement chapter. The tiles have a region assigned indicated by an integer ID (there is a special region for all the walls). The region must be connected (except for the gate region). For any two tiles in the region, there must exist a path between them consisting of only the 4 straight movements (up, down, left, right) that doesn't at any point leave the region.

Definition 1 (neighboring regions)**.** *We define a relation between the regions saying, whether the regions neighbor each other (there exist two tiles of these two regions that neighbor each other):*

$$R_A, R_B \in \mathbf{R} \colon R_A \sim_{N_G} R_B$$

Where $\mathbf{R}$ *is the set of all regions.*

The borders between the regions (tiles that have a neighbor tile of a different region) are marked as gates. There can be multiple regional decompositions of the map, but ideally, the regions should border each other at the map's chokepoints. We call the locations where the regions neighbor each other the gate.

Definition 2 (gate)**.** *The gate is defined by the two regions neighboring it:*

$$G = \{R_A, R_B\}$$

The gate has a length G_l *based on the distance between two tiles on its end.*

Definition 3 (neighboring gates)**.** *We say that the gates neighbor each other when they share the same region:*

$$G_1, G_2 \in \mathbf{G} \colon G_1 \sim_{N_G} G_2 \Leftrightarrow \exists R \in \mathbf{R} \colon G_1 \in R \wedge G_2 \in R$$

3.2 Pathfinding Problem

When a unit moves along a path toward the goal, it passes through a sequence of gates. We call this sequence the unit's gateway path:

Definition 4 (gateway path)**.** *Gateway path is a sequence of gate*

$$P_G = (G_1, G_2, G_3, ...)$$

To find the gateway path, we need to create a graph based on the decomposition, with nodes representing the gates and the edges connecting the gates from the same region.

Definition 5 (pathfinding graph)**.** *We define a graph* $\Gamma = (\mathbf{G}, \mathbf{E})$ *where:*

- $\mathbf{G}$ *is the set of all gates*
- $\mathbf{E} = \{(G_1, G_2) \colon (G_1, G_2) \subseteq \mathbf{G^2} \wedge G_1 \neq G_2 \wedge G_1 \sim_{N_G} G_2\}$ *is the set off all the edges (connections between the gates)*

Definition 6 (valid gateway path)**.** *For a gateway path to be valid within a graph* Γ *there must be an edge between the gates following each other:*

$$Valid(\Gamma, P_G) \Leftrightarrow \forall G_A, G_{A+1} \in P_G \colon (G_A, G_{A+1}) \in \mathbf{E}$$

Definition 7 (region path)**.** *We can also define a region path as the sequence of regions through which the unit travels:*

$$P_R = (R_1, R_2, R_3...) \colon R_1, R_2, R_3... \in (R)$$

Definition 8 (valid region path)**.** *For the region path to be valid there must exist a corresponding valid gateway path:*

$$Valid(\Gamma, P_R) \Leftrightarrow \exists P_G \colon Valid(\Gamma, P_G) \wedge (\forall G_A \in P_G, \exists R_A \in P_R \colon R_A \in G_A \wedge R_{A+1} \in G_A)$$

When the player orders a group of units to move to a certain location a pathfinding request is sent to the pathfinding algorithm.

Definition 9 (pathfinding request)**.** *We define the pathfinding request as a tuple:* $\Pi = (n, o, g, \Gamma)$ *where:*

- n *is the number of units*
- o *is the origin (single 2D point in two-dimensional space approximating the starting position of the group of units)*
- g *is the position of the goal (single 2D point).*

3.3 Flow

We define a metric called the flow that describes the number of units that pass through a gate (or through the goal) per second. The flow fluctuates through time. The flow through the gate G at the time t is written as $\Phi_G(t)$. We can also differentiate an outgoing flow (the flow passing through a gate) and an incoming flow (the flow coming to the gate from its predecessors).

- $\Phi_G^{out}(t)$ - Flow coming out of the gate G at the time t
- $\Phi_G^{in}(t)$ - Flow arriving at the gate G at the time t
- $\Phi_{G_1 \Rightarrow G_2}^{into}(t)$ - Flow arriving at the gate G_2 from the gate G_1 at the time t
- $\Phi_{G_1 \Rightarrow G_2}^{toward}(t)$ - Flow exiting the gate G_1 toward the gate G_2 at the time t
- $\Phi_G^{in}(t) = \sum_{(G_i, G) \in \mathbf{E}} \Phi_{G_i \Rightarrow G}^{into}(t)$

The incoming flow strongly depends on the outgoing flow from the predecessor gate and the movement times between our gate and the predecessor gates.

Definition 10 (unobstructed movement time)**.** *def:unobstructedMovement] Unobstructed movement $t_{G_1 \Rightarrow G_2}$ is the time it would take the units to reach the gate G_2 from the gate G_1 if there were no units blocking the way. In the simulation, there may be units accumulated at the gate blocking the path for the unit. This effect can be seen in Figure*

In our model, we make an assumption that the flow doesn't change when the units are traveling between the gates. This is, however, often not true. Execution problems may occur, which will result in a change in the flow value. If we make this assumption, we can say: $\Phi^{into}_{G_1 \Rightarrow G_2}(t) = \Phi^{toward}_{G_1 \Rightarrow G_2}(t - t_{G_1 \Rightarrow G_2})$ where the $t_{G_1 \Rightarrow G_2}$ is the time of unobstructed movement from G_1 to G_2

If multiple units move toward the goal, we can separate them into groups based on the gate paths they follow. Each group will have specific times the first and the last unit reaches the target, a number of units, and flow values at each gate passed and at the goal. We will call these groups the flow streams.

Definition 11 (flow stream)**.** *Flow stream is a tuple $s = (n, o, g, P_G)$ where*

- *n is the number of units in the stream*
- *o is the origin of the units*
- *g is the goal the units navigate toward*
- *P_G is the gateway path common for all of the units in the stream*

Definition 12 (valid flow stream)**.** *The flow stream s is valid for a pathfinding request Π if the gate path is valid, the origin and the goal neighbor the appropriate gates, and the number of units isn't higher than the total number of units navigating:*

$$\sigma(s, \Pi) \Leftrightarrow ((Region(o) \in G_0) \wedge (Region(g) \in G_{last}) \wedge Valid(\Gamma, P_G) \wedge n \leq \Pi.n)$$

Definition 13 (arrival times)**.** *We define t^{first}_s and t^{last}_s for a flow stream s as the arrival times of the first and last units from the stream.*

Definition 14 (solution set)**.** *We define a set of all the solutions φ to the pathfinding problem Π. The solution is a set of flow streams.*

$$\varphi = \{S \subseteq \mathbb{S} \colon (\forall s_1, s_2 \in S \colon s_1 \neq s_2 \Rightarrow s_1.P_G \neq s_2.P_G) \wedge (\sum_{s \in S} s.n = \Pi.n)\}$$

where $\mathbb{S} = \{s \colon \sigma(s, \Pi)\}$ is a set of all streams valid for a particular pathfinding request and a graph.

Definition 15 (solution time)**.** *The time of the solution t_S is the finishing time of its slowest stream*

$$For S \in \varphi \colon t_S = Max_{s \in S}(t^{last}_s)$$

where $\mathbb{S} = \{s \colon \sigma(s, \Pi)\}$ is a set of all streams valid for a particular pathfinding request and a graph.

Group pathfinding aims to find the quickest way of moving the units from the start to the goal. The quickest solution S_Q can be selected from all the solution sets of flow streams.

Definition 16 (quickest solution)**.** *S_Q is the quickest solution if $S_Q = Min_{S \in \varphi}(t_S)$ where φ is the set of all solutions and t_S is the time it takes the solution S to finish.*

3.4 Flow Graph Algorithm

To find the quickest solution to a pathfinding problem, we use the Flow Graph algorithm. The basic structure of the algorithm is as follows:

1. Construct a flow network where the source represents the units' starting position, the sink represents the target, and the other nodes the gates.
2. Propose several sets of paths that the flow streams could use.
3. Find the best solution time that can be achieved by using each of the path sets. And select the set with the lowest solution time.
4. Assign the units to different paths from the selected set in a way that would result in the quickest finishing time.

In steps 3) and 4), we need to estimate the finish times of different solutions. We know that this time is equal to the slowest stream s from the solution t_s^{last}. We can estimate this time as:

$$t_s^{last} = t_s^{first} + t_s^{\Phi}$$

- t_s^{first} is the arrival of the first unit
- t_s^{Φ} is the time it takes to transport all units from the stream (after the first unit arrived)

The methods that could be used to estimate the arrival time t_s^{first} will be described in the Arrival Times section. The flow time (transport time) t_s^{Φ} using the Equation (3.1) based on the flow. Estimating the flow itself is a difficult task which we will tackle in the Estimating The Flow section.

$$n_s = \int_{t_s^{first}}^{t_s^{first}+t_s^{\Phi}} \Phi_{T,s}^{in}(t)dt \tag{3.1}$$

- n_s is the number of units in the stream
- $\Phi_{T,s}^{in}(t)$ is the flow coming into the target from the stream s at the time t

The steps 2), 3) and 4) also deal with another problem. The set of all the possible solutions is too large. We have a large number of different combinations, each with many possible unit assignments. We would like to limit the number of path combinations proposed in step 2) ideally without losing the combination used by the optimal solution. In steps 3) and 4) we would also like to be able to

determine the best unit assignment and its solution time without going through all of the possible assignments. Ideally, we should be able to determine the unit assignment directly from the path combination and the number of units. The possible ways of reducing the number of examined solutions will be discussed in the "Pruning The Solutions" section.

3.5 Arrival Times

In the Flow Graph algorithm, we often need to calculate the time the units move from point A to point B (unobstructed by other units). This estimate should be performed quickly but be reasonably accurate. The method used for the estimation can greatly influence the characteristics of our algorithm. If we set the A as equal to the origin, we call the movement time the arrival time.

The time of the unobstructed movement will mostly depend on the distance between the two points but also possibly on the distance and shapes of the obstacles. In our estimate, we will only use the distance. And estimate the time as:

$$t_{A \Rightarrow B} = \frac{distance(A, B)}{unitSpeed}$$

To estimate the time we then simply need to estimate the distance traveled. Several methods can be used for this. We will describe the shortest grid path and the gate path methods.

3.5.1 Shortest Path in the Grid

We will find the shortest path on the grid between the two points and return its length. Often the path needs to pass through a specific sequence of gates. The A* pathfinding algorithm can be modified to deal with this requirement.

If we use a path on a grid with eight possible directions, the length of the actual shortest path may be overestimated. This can be prevented by smoothing the path over to get a true shortest path. The true shortest path will underestimate the travel distance taken by the units because not all units will follow the optimal path. The main problem with this method is that the distances can not be easily precomputed, and the pathfinding will take longer than with other methods.

3.5.2 Gate Path

The distance can be estimated using the gateways the path will take. For the flow streams, we even have a gate path already specified. We will sum up the distances between the gates in the path and then add the distance from the origin to the first gate and from the last gate to the goal.

This method will be quicker than the previous method because the distances between the gates can be precomputed. The resulting distance, however, won't equal the shortest distance between the two points.

Ideally, we would like our estimate to match the length of the path used by the units, but different units may take different routes between the two gates. The route depends on the unit's original position, target position, other units,

and various other factors. We, therefore, have to use simpler estimates like the shortest distance between the gates or the distance between the gate centers.

Shortest Distance

The distance between the two closest points on the two gates. The estimate given by this method will always be lower than the distance found by the A* (the A* path must use two points on the gate, and the distance between them must be equal to or higher than the gate distance).

Center Distance

The distance between the gate centers. This estimate will overestimate the shortest distance between the points. There is a path between these points that goals through the gate centers, but it may not be the shortest path.

Given that the shortest path underestimates the distance traveled by the units (not all of them can follow the optimal path), this method is the one used in the implementation.

Weighted Average

This method attempts to combine the two previously mentioned methods (or some others). By averaging their results (which provide an upper and lower bound to the length of the shortest path), it tries to get an estimate closer to the length of the shortest path.

3.6 Estimating The Flow

To get a reasonable estimate of the movement of units, we would like to estimate the flow going through a gate. We don't know all the factors that go into the flow value at a gate, but we can deduce the most important ones.

We can divide these factors into two basic categories

3.6.1 Execution Settings

Execution settings are parameters that are constant in the simulator (speed of the units, unit size, steering force settings. . .). These parameters can be included in the flow computation or factored into the formula as constants (but then the formula would need to be recomputed every time these parameters are changed).

Unit Speed

Slower unit speed will result in lower flow values and vice-versa. But it isn't clear if this relation is linear (it may depend on the steering forces set up and the movement change limitations).

Unit Size

Smaller unit size will result in a higher flow because it is possible to fit a higher number of units into a gate of the same size.

Steering Forces And Turning Speed

Probably very complex and not easily computed dependence. Higher separation forces result in lower flows and higher steering forces in higher flows, but the function describing this property will be complex. The effects of local avoidance or turning and movement change speed will be essentially impossible to predict.

3.6.2 Dynamic Factors

Parameters that either change during the execution or vary between different gates. The impact of these factors will depend on the execution settings chosen in the game, and it will be specific to every game.

Incoming Flow

The incoming flow will set the maximum possible value of the outgoing flow. There may also be other dependencies - a higher flow value may generate more push toward the gate. We consider only the incoming flow at a specific moment. The flow that reached the gate previously will be considered in a different variable.

Overflow

The units tend to accumulate at the gates, and they push the units in front of them forward. A higher amount of accumulated units will generate more push and (in almost all cases) increase the flow. For example, the flow in Figure 3.2 is 10% higher than the flow in Figure 3.1. It may, however, happen that the pushing causes units to get into more collisions and possibly decrease the flow, as observed in the simulation. This can be seen as a problem with the execution. To measure this, we define an overflow function.

Definition 17 (overflow)**.** *We define the overflow as the difference between the cumulative incoming and outgoing flow at the gate G and time t*

$$O_G(t) = \int_0^t \Phi_G^{in}(x)dx - \int_0^t \Phi_G^{out}(x)dx$$

Gate Length

Smaller gates will only allow lower flow values, while the larger gates can be effectively passed even by massive amounts of units. The gate length will almost certainly set an upper limit on the flow that can pass through the gate.

However, even flows smaller than the maximum may be impacted (for example, a lower amount of overflow may be required to push the units through). Moreover, we do not know if the maximum flow linearly depends on the gate length. The units near the walls may move slower than the units at the center of the gate.

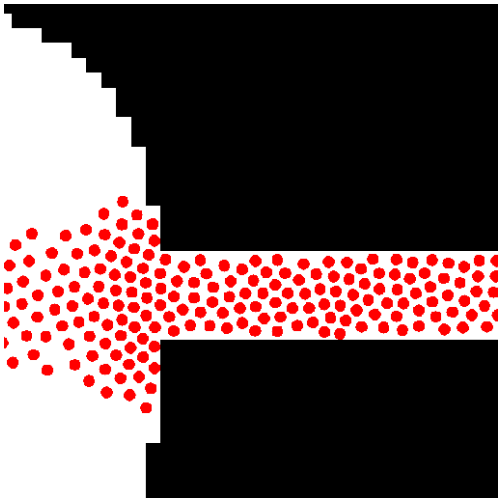


Figure 3.1: Low Overflow

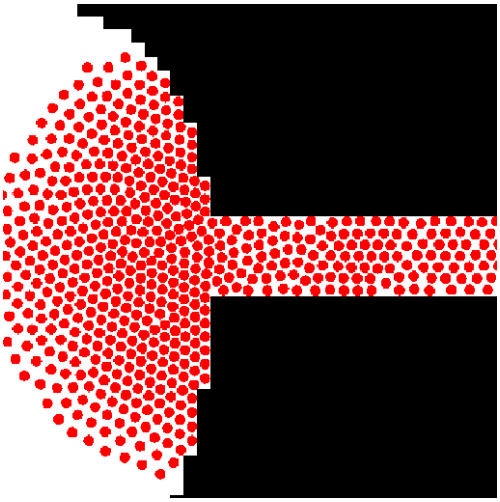


Figure 3.2: High Overflow

Effects Of The Wall Shape

The flow through the gate may be affected by the general shape of the walls near the gate. The walls forming a funnel (Figure 3.5) will have a higher flow than a long tunnel with unchanging width (Figure 3.6). Smoother walls (Figure 3.3) result in a higher flow than rough walls with a lot of bumps and crevices (Figure 3.4). These properties are tough to define and measure, and many of them exist.

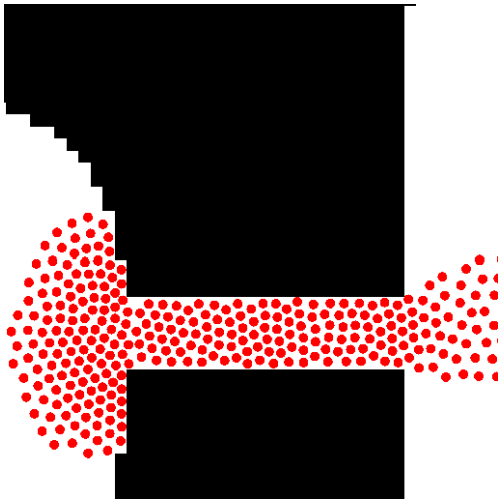


Figure 3.3: Smooth Gate

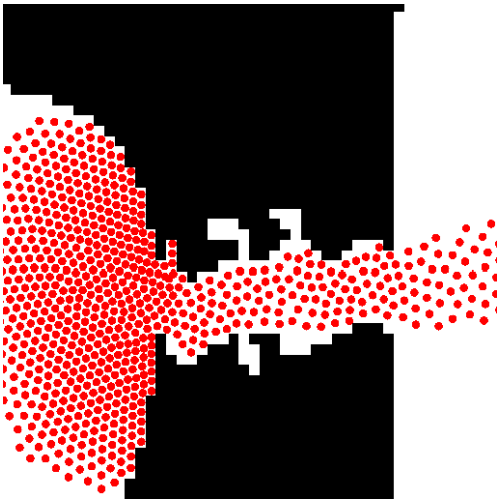


Figure 3.4: Rough Walls

Execution Errors

The final flow can be affected by the errors of the execution. These are hard to predict and may depend on the shape of the map, the choice of execution algorithm, and the assignment to the flow streams.

We will list examples of the navigation errors we encountered when testing the navigation

1. The units at the entrance are pushed close to each other, and some of them collide. This results in the entrance being temporarily blocked (Figure 3.7).

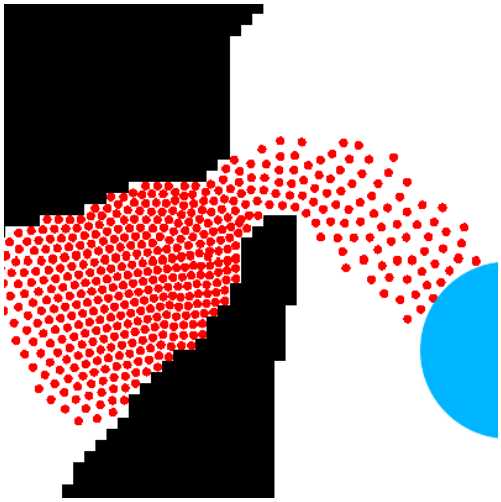


Figure 3.5: Funnel Gate

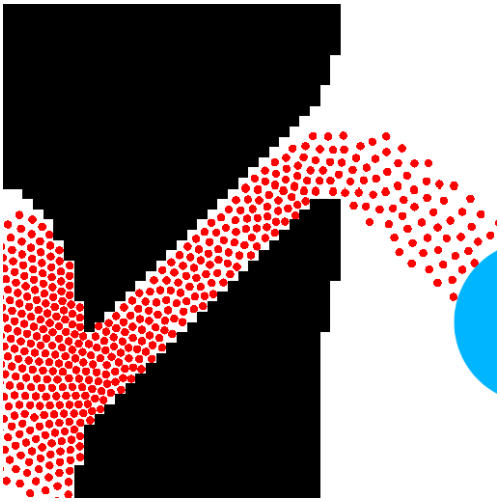


Figure 3.6: Straight Gate

2. In Figure 3.8, we can see the units in the ring having their path blocked by the units above them and being pushed toward the wall.

3. In Figure 3.9, we see two flow streams heading to two different gates. The stream units are, however, mixed up and have trouble separating. Some of the units are getting carried off their path by the stream they do not belong to.

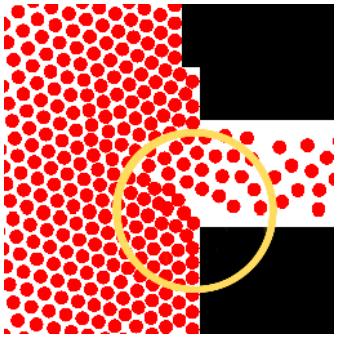


Figure 3.7: Blocked Entrance

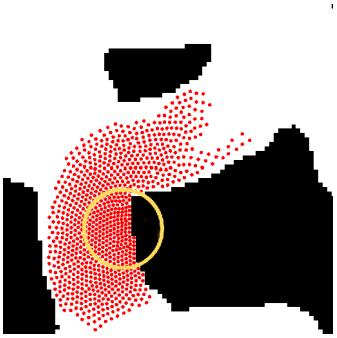


Figure 3.8: Wall Hugging

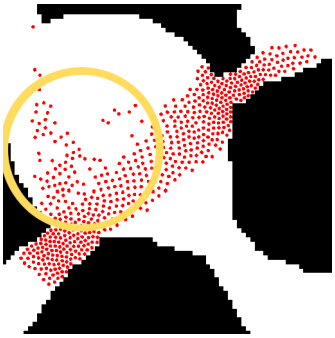


Figure 3.9: Stream Separation Error

Prediction of errors shouldn't be included in the flow computation but should be considered when implementing the execution part.

3.6.3 Flow Function

We would like to define a function that will help us estimate the flow going out of a gate at any given moment in time. This will help us determine which combination of streams will allow us to push the highest flow and achieve the quickest solution.

This function can be created based on the factors mentioned in the previous section. We will, however, consider only some of them. The execution errors and the effects coming from the shape of the walls are too hard to estimate to

be helpful. The unit speed, the steering forces set up, and the movement change speed are static for the whole simulation. Therefore, estimating the flow based on different separation force values is unnecessary. We are left with three variables to compute the flow. We can define a flow function based on them.

Definition 18 (flow function)**.** *Flow function computes flow going out of the gate based on the incoming flow, overflow, and the gate's length*

$$\Phi_G^{out}(t) = F(\Phi_G^{in}(t), O_G(t), G_l)$$

We do not know the interactions between the function variables and the exact function will change with different simulation settings (unit steering forces, speed of movement change, max unit speed...). The function may also ignore some less important variables.

3.6.4 Maximum Possible Flow And No Push Flow

We will define two values important for flow estimation.

Definition 19 (maximum flow)**.** *The maximum flow Φ_G^{max} is the maximum flow that can be pushed through the gate G.*

Definition 20 (no-push flow)**.** *The no-push flow $\Phi_G^{no-push}$ is the maximum flow that can be pushed through the gate G with no push from the overflow.*

These values will not depend on the incoming flow or the momentary overflow and, for given simulation settings, can be estimated using only the gate length.

To illustrate the expected behavior, we will consider three values of a constant incoming flow. In real situations, the flow will not be constant.

1. $\Phi_G^{in} \leq \Phi_G^{no-push}$ The outgoing flow should be equal (or almost equal) to the incoming flow (Figure 3.10).

2. $\Phi_G^{no-push} \geq \Phi_G^{in} \leq \Phi_G^{max}$ The outgoing flow will start at the "no-push level" as the overflow accumulates, the flow will rise and stabilize at Φ_G^{in}. In the end, when there is no more flow incoming, the flow will decrease and then quickly fall off after reaching the "no-push" level (Figure 3.11).

3. $\Phi_G^{max} \leq \Phi_G^{in}$ The outgoing flow will once again start at the "no-push level" and then rise to and stabilize at Φ_G^{max}. When the incoming flow stops, the flow should continue to be at the maximum level while there is still enough overflow and then fall off, similar to the situation in the second case (Figure 3.12).

The time and shape of the transitions are unknown and will depend on the gate length, the incoming flow, and the execution settings. For example, a higher incoming flow will probably build up the push more quickly and shorten the t1 transition periods. This effect may even make it worth pushing in a higher than the maximum flow.

The units do not pass only through one gate, and all of the gates passed will affect the shape of the resulting flow. Moreover, the incoming flow will not be constant in these cases. The basic behavior will still be the same. A higher number of gates that fall into cases 2) and 3) will smooth out and increase the length of the transitions.

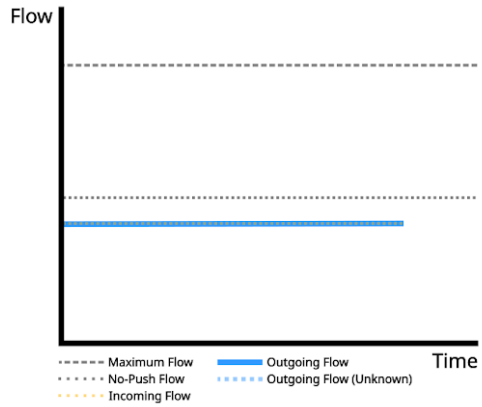


Figure 3.10: Low Incoming Flow

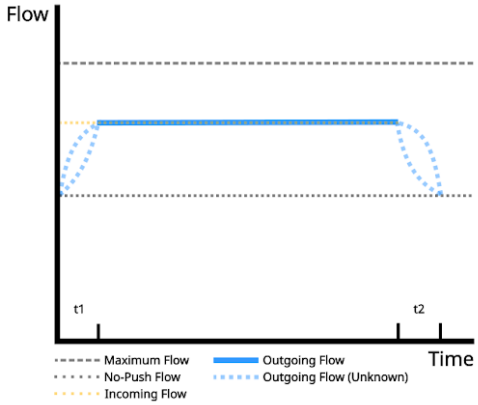


Figure 3.11: Medium Incoming Flow

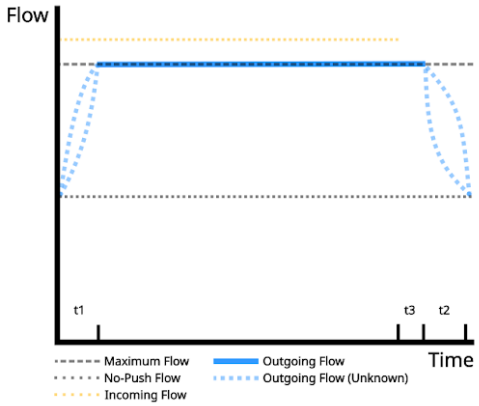


Figure 3.12: High Incoming Flow

3.6.5 Basic Flow Estimate

Our implementation (Flow Graph) uses only a rough estimate of the flow going through an edge. It ignores the overflow effect and assumes a linear relationship between the gate length and the flow.

Definition 21 (basic flow estimate)**.** *We define our basic flow estimate as*

$$\Phi_G^{out}(t) = F\left(\Phi_G^{in}(t), G_l\right) = Min\left(\Phi_G^{in}(t), c \cdot G_l\right)$$

The constant c should be chosen such that the flow capacity ($\Phi_G^{cap} = c \cdot G_l$) is somewhere between the no-push and max flow $\Phi_G^{no-push} \leq \Phi_G^{cap} \leq \Phi_G^{max}$

Ignoring the effects of the overflow will produce inaccuracies. How closely the estimate matches reality will depend on several factors. The closer the $\Phi_G^{no-push}$ and the Φ_G^{max}, are to each other, the more accurate the estimate. Longer transitions between the flow levels and nonlinear effects of the gate length will result in less accurate modeling.

The main advantage of this model is its simplicity, which can be used to make some important assumptions. This model may also be used only for an initial flow estimate, which a more accurate model can further refine.

3.7 Pruning The Solutions

We can use the simplicity of the basic flow estimate to reduce the number of solutions we have to examine. We can show that there exists an optimal solution with certain characteristics. We can make these characteristics a requirement, thus reducing the Flow Graph's search space. We will also be able to compute the unit numbers and the flows for any selected path combination. To do so we will have to prove several theorems.

We can define two special types of gates. The joiner and divider gates. The joiner gates are the gates that are used by two or more streams with different previous gates. The divider gates are the ones used by streams with different following gates.

Theorem 1 (The Quickest Stream Takes The Shortest Path)**.** *Every solution will be a set of flow streams, each with a unique gate path. Choose the shortest of these paths (the quickest one for the units to go through) and call the associated stream the quickest stream. There exists a quickest solution, where the quickest stream takes the shortest path at every divider gate (from the paths taken by the outgoing streams, not all the possible paths):*

Proof. By contradiction. Suppose there is a stream coming from the divider gate that takes a path shorter than the quickest stream. Let's call this stream s_{oth} and the quickest stream s_q, the time it took to navigate the previous part of their path as t^1_{oth} and t^1_q and the times for the rest of the path as t^2_{oth} and t^2_q.

$$t_{oth} = t^1_{oth} + t^2_{oth} \wedge t_q = t^1_q + t^2_q$$
$$t_{oth} \geq t_q \wedge t^2_{oth} \leq t^2_q$$

This means that $t^1_{oth} \geq t^1_q$ we can we can define a path t_{new} combining the first path from the quickest stream s_q and the second one from the other stream s_{oth} where:

$$t^1_{new} = t^1_q \wedge t^2_{new} = t^2_{other} \wedge t_{new} = t^1_{new} + t^2_{new}$$

From This we know $t_{new} \leq t_q \leq t_{oth}$

Given a divider gate G and a previous joiner gate G_J (where the s_{oth} and the s_q were joined or the origin). The characteristics of the original can be easily simulated with the new set of streams.

We can define four new streams s_1, s_2, s_3, s_4 where the s_1 copies the path of the s_q, s_2 copies the s_{oth}, the s_3 copies the first part of s_oth and the second part of s_q, and s_4 uses the two shorter paths and achieving the time t_{new}. We assign the flows:

$$\Phi_{G,s_3}(t) = Min(\Phi_{G,s_{oth}}, \Phi_{G,s_q}(t))$$
$$\Phi_{G,s_4}(t) = \Phi_{G,s_3}(t) + (1 - sgn(\Phi_{G,s_3}(t)) \cdot Max(\Phi_{G,s_{oth}}, \Phi_{G,s_q}(t))$$
$$\Phi_{G,s_1}(t) = \Phi_{G,s_q} - \Phi_{G,s_4}(t)$$
$$\Phi_{G,s_2}(t) = \Phi_{G,s_{oth}} - \Phi_{G,s_4}(t)$$

In this case, s_4 is the quickest stream and it utilizes the quickest path at the gate G and the solution time is equal to the solution with s_q and s_{oth}

□

This doesn't mean that there aren't quickest solutions that do not have this characteristic, but there will always be a solution just as fast, where this is the case. it also doesn't mean that the stream takes the quickest possible path on the map, just that its path is shorter than the paths of the other streams.

Theorem 2 (The Quickest Stream Saturates Its Path)**.** *There exists a quickest solution where the quickest stream saturates its path and has a constant flow:*

Proof. Take the path of the quickest flow and the solution time t_S. The quickest stream s_q can transport at most $n = \Phi^{cap}_{G_{min}} \cdot (t_{S_q} - t_{S_q,o\Rightarrow g})$ where t_{S_q} is the time of the quickest solution, $t_{S_q,o\Rightarrow g}$ the time of the unobstructed movement from the origin to the goal (along the quickest stream path) and $\Phi^{cap}_{G_{min}}$ the minimum gate capacity on the path.

Say that the flow is lower than $\Phi^{cap}_{G_{min}}(t)$ at some time t. Then the flow can be safely increased. If there are no other streams at a gate where this occurs, it is trivial. When there are other streams, some of their units can be assigned to the quickest stream. By doing this, the solution time will either improve or stay the same.

If the flow is higher, the units will accumulate at the lowest capacity gate, and the outgoing flow will still be $\Phi^{cap}_{G_{min}}(t)$. Setting the initial flow already at $\Phi^{cap}_{G_{min}}(t)$ changes nothing.

□

Other Saturating Streams

If the flow in the quickest stream is constant and saturating, we can do the same for the other streams. We can think of the graph with the quickest stream as a graph where the quickest stream's flow reduced the capacities of the gates passed. The next quickest stream will have the same properties on this modified graph. We, therefore, know that all the streams have a constant flow value, and they saturate at least one gate on their path (possibly together with other quicker streams).

We do not have a method of finding the path the quickest stream uses. Setting it simply to the quickest path available isn't valid (we only know that it takes the quickest path of all the streams at every divider gate, not the quickest path, period). Despite this, considering the solutions where the quickest stream uses the quickest path first could be a good heuristic.

In the Flow Graph algorithm, we construct the set of paths used by the streams by adding augmenting paths. This is also a good heuristic for finding reasonable solutions. But the solution found using the method has no guarantee of being optional.

The gate flow for a particular flow stream must be constant between the first unit and the last unit arrival and zero otherwise, with the same flow at every gate. We can define a constant Φ_s and times $t^{start}_{G,s}$ and $t^{end}_{G,s}$ when this flow happens.

The first unit should arrive as quickly as possible. The last unit arrival can be computed from the first unit arrival using the stream's flow:

$$t_s^{first} = \frac{distance(s, o, g)}{unitSpeed}$$
$$t_s^{last} = t_s^{first} + \frac{n}{\Phi_s}$$

All the flow streams should arrive at the same time (if not, it is possible to take a few units from the slowest stream and add them to the quickest stream, this will decrease the flow time). We, therefore, have:

$$\forall s \in S_Q \colon t_{S_Q} = t_s^{last}$$

From this, we can also compute the first and the last unit times not just for the goal, but also for all of the gates. This will assume a constant flow for all the sections of the stream. If the real flow is higher, the result will not worsen, but the last unit time may be quicker than the estimate.

- $t_{G,s}^{first} = \frac{distance(s,o,G)}{unitSpeed}$
- $t_{G,s}^{last} = t_S - \frac{distance(s,G,g)}{unitSpeed}$ For constant flows
- $t_{G,s}^{last} \leq t_S - \frac{distance(s,G,g)}{unitSpeed}$ For non-increasing flows

Theorem 3 (Edges With A Flow In Both Directions)**.** *An optimal solution cannot contain an edge with flow streams heading in both directions. Otherwise, removing these flows could improve the time (or at least not worsen it).*

Proof. Let us have two streams with flows Φ_x, Φ_y along an edge (G_x, G_y) of length l where the Φ_x goes through G_x first and Φ_y in the opposite direction. The lengths of the two streams can be described as $l_x = l_{x1} + l + l_{x2}$ where l_{x1} is the length of the path until the edge and the l_{x2} after the edge, and $l_y = l_{y1}+l+l_{y2}$. Define $\Phi_{min} = Min(\Phi_x, \Phi_y)$. We can assume that $\Phi_x \geq \Phi_y \wedge \Phi_y = \Phi_{min}$ (WLOG).

Now Φ_x can be modified to use its original path until G_x and the Φ_y associated path from there on. The flow value will be set to Φ_{min}. The Φ_y can be modified in a similar way. Another stream copying the path of the original Φ_x with the flow set to $\Phi_x - \Phi_{min}$ is needed.

The lengths of the new streams are $l_1 = l_{x1} + l_{y2}$, $l_2 = l_{y1} + l_{x2}$ and $l_3 = l_x$. The number of units transported using the original streams for a time t can be computed as:

$$n_1 = Max\left(t - \frac{l_{x1} + l + l_{x2}}{unitSpeed}, 0\right) \cdot \Phi_x + Max\left(t - \frac{l_{y1} + l + l_{y2}}{unitSpeed}, 0\right) \cdot \Phi_y$$

And the number of units transported using the new streams n_2:

$$n_2 = Max\left(t - \frac{l_{x1} + l_{y2}}{unitSpeed}, 0\right) \cdot \Phi_y + Max\left(t - \frac{l_{y1} + l_{x2}}{unitSpeed}, 0\right) \cdot \Phi_y$$
$$+ Max\left(t - \frac{l_{x1} + l + l_{x2}}{unitSpeed}, 0\right) \cdot (\Phi_x - \Phi_y)$$

We can rewrite n_1 as

$$n_1 = Max\left(t - \frac{l_{x1} + l + l_{x2}}{unitSpeed}, 0\right) \cdot \Phi_y + Max\left(t - \frac{l_{y1} + l + l_{y2}}{unitSpeed}, 0\right) \cdot \Phi_y + Max\left(t - \frac{l_{x1} + l + l_{x2}}{unitSpeed}, 0\right) \cdot (\Phi_x - \Phi_y)$$

We need to show for $l, l_{x1}, l_{x2}, l_{y1}, l_{y2}, \Phi_x, \Phi_y, unitSpeed \geq 0$ that:

$$n_1 \leq n_2$$

$$Max\left(t - \frac{l_{x1} + l + l_{x2}}{unitSpeed}, 0\right) \cdot \Phi_y + Max\left(t - \frac{l_{y1} + l + l_{y2}}{unitSpeed}, 0\right) \cdot \Phi_y \leq Max\left(t - \frac{l_{x1} + l_{y2}}{unitSpeed}, 0\right) \cdot \Phi_y + Max\left(t - \frac{l_{y1} + l_{x2}}{unitSpeed}, 0\right) \cdot \Phi_y$$

We know that (otherwise the two streams wouldn't exist):

$$Max\left(t - \frac{l_{x1} + l + l_{x2}}{unitSpeed}, 0\right) \cdot \Phi_y + Max\left(t - \frac{l_{y1} + l + l_{y2}}{unitSpeed}, 0\right) \cdot \Phi_y = 2t - \frac{l_{y1} + l_{y1} + l_{x1} + l_{x2} + 2l}{unitSpeed}$$

We also know:

$$2t - \frac{l_{y1} + l_{y1} + l_{x1} + l_{x2} + 2l}{unitSpeed} \leq Max\left(t - \frac{l_{x1} + l_{y2}}{unitSpeed}, 0\right) \cdot \Phi_y + Max\left(t - \frac{l_{y1} + l_{x2}}{unitSpeed}, 0\right)$$

This way we show $n_1 \leq n_2$ and the solution using the new streams is equal or better to the original

□

Additional Flow

Our solution may send higher flow values than specified by the selected solution. This will not worsen the time of the solution (the additional flow will not block the other streams), but it may improve the last unit times for the gates where this flow is present. Based on the basic flow estimate, the solution time shouldn't change, and the flow entering the target should be the same as the one specified by the solution. In reality, however, the higher flow will cause overflow at certain gates, possibly improving the solution time.

For the solution to be correct, the flow should be equal or lower to the one specified by the solution, and the times at the gates should be equal or quicker than the solution.

4. Implementation

Our algorithm is an improved version of the Flow Graph algorithm by Jan Pacovský. We will introduce the original algorithm in the first section of this chapter. The second section will discuss problems with the original algorithm. The third section will describe our algorithm - Regional Flow Graph.

4.1 Existing Solution

Our method is largely based on an existing method proposed by Jan Pacovský called the Flow Graph. Following is a pseudocode of the algorithm (Pseudocode 3). During the preparation phase, the algorithm constructs a flow network representing the map. In the pathfinding phase, it finds a set of paths that can be run concurrently. In the execution phase, the units follow their assigned paths. Each of these steps will be described in detail in the following subsections.

```
Preparation
WaterDecomposition()
CreatePartialFlowGraph()

Pathfinding
FinishFlowGraph(target, units)

solutions
concurrentPaths
While AugmentingPathExists:
    path ← FindTheShortestPath()
    SaturatePath(path)
    If BidirectionalFlow:
        solutions.Add(concurrentPaths)
        concurrentPaths.Add(path)
        concurrentPaths ← MutatePaths(concurrentPaths)
    Else:
        concurrentPaths.Add(path)

solutions.Add(concurrentPaths)

quickestSolution ← SelectQuickestSolution(solution)
unitCounts ← AssignUnitCounts(quickestSolution, units.Count())
assignment ← AssignUnits(unitCounts, units)

Execution
EveryFrame:
    SteerToward(furthestVisiblePathPoint)
```

Pseudocode 3: Flow Graph

4.1.1 Preparation

The method views the map as a flow network, with the unit starting position as a source and the target as a sink. The edges of the network contain two numbers, one indicates the length of the edge, another one its capacity.

In the preparation phase, we can not yet create the whole network because the starting positions of the units and the position of the target aren't yet known. For this reason, we will only create part of the network without the source and sink. We will call this a partial graph. To construct this network, the map first has to be decomposed into a set of regions separated by gates (the gates are placed at the chokepoints). The algorithm uses the water decomposition by Kári Halldórsson (Water Decomposition) for this purpose, but other decomposition algorithms could be used as well.

The nodes of the network are placed at the positions of the gates. Two nodes are then connected with an edge if both of the gates associated with them neighbor a common region. The edge has two gates associated with it, these are the gates belonging to the nodes at both ends of the edge. The capacity of the edge is set to the length of the smaller from the two gates associated with it. The length of the edge is equal to the distance between the centers of the two gates.

4.1.2 Pathfinding

During the pathfinding phase, the Flow Graph algorithm generates a set of paths that can be used simultaneously by the units during the execution. We will call such a set of paths concurrent paths. These will be generated by adding augmenting paths and fixing possible problems with bi-directional flows at the edges along the way. Our method will generate multiple sets of concurrent paths the best set of concurrent paths will then be selected during the unit assignment.

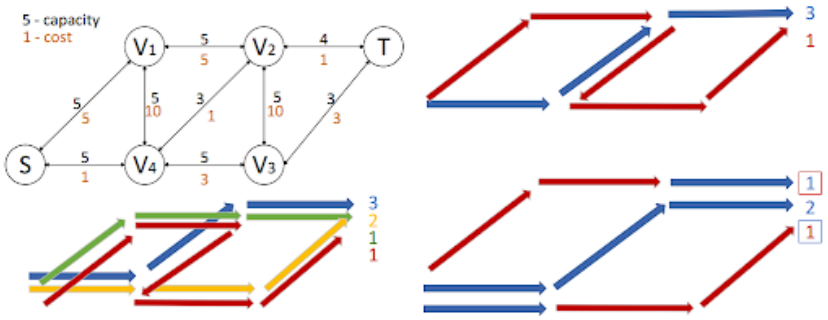
Once we have selected the set of paths to be used, we will assign units to it. First, we will determine the unit counts for individual paths based on their length and flow assigned to them. Then the algorithm assigns units to the paths.

Finishing The Network

The flow network is still missing the source and sink nodes. The sink is added at the place of the target and connected by edges to the gates neighboring the target's region. The edge capacities are set to equal the length of the gates. The edge lengths are the distances between the target and the gate centers. The source is added in a similar way.

Adding Paths

Now that the flow network is finished, we have to find the paths the units will use. We will do this by using the augmenting paths. We will find the shortest path in the network and push a saturating flow through it. We store the saturated path and its flow and continue by adding new augmenting paths. In the end, we won't be able to find any augmenting path and have a set of paths our units can use.



Upper Left: Network with the capacity and length of the edges. Lower Left: Illustration of our method, which repeatedly searches for the shortest path, saturates it. In the case of a bi-directional flow, it mutates the paths. First, it finds the blue, then the yellow, the green, and at last, the red path. Right: Mutation of the paths and removal of the bi-directional flow. Three paths are created instead of two, and the total flow is the same as before the mutation.

Figure 4.1: Path Mutation (Image Source: Jan Pacovský [1])

Bi-Directional Flow

The solution has one problem, the augmenting path may use edges that already have a flow in the opposite direction. We will say that the edge has a bi-directional flow (there are paths that push flow in both directions). Such a situation isn't uncommon in maximum flow problems, but it does alter the paths the flow uses. The stored paths can, therefore, not be used.

Before adding a path that would cause a bi-directional flow, we create a copy of the working set of concurrent paths and add it to the finished sets. We then add the new path to the working set and remove the bi-directional flow by what the algorithm calls path mutations.

Mutations

Mutation is a process used to alter the paths within the working sets in a way that eliminates the bi-directional flow. The process of a path mutation can be seen in Figure 4.1.

We will say that the existing flow is flowing in a positive direction (or positive flow). The flow added by the new path going in the opposite direction will be called negative flow.

We take a look at the newly added path and find its first section with the negative flow. For this section, the algorithm finds the paths with the positive flow using it. From the subsets of these paths with a combined flow higher than the negative flow, we select the one with the minimal flow.

The paths from the selected subset will be mutated. Because the subset has a higher combined flow than the negative flow, one of them is randomly selected only for a partial mutation.

At the end of the process, the newly added path is removed from the set. If any of the new paths added during the mutation resulted in a bi-directional flow,

the mutations would be for them as well.

Full Mutation will create two new paths, both will have the same flow as the mutated path. For both the mutated and newly added path, we will call the part before the bi-directional flow segment the head and the part after it the tail.

The first path will be the concatenation of the mutated path's head and the added path's tail. The second path will be a concatenation of the added path's head and the mutated path's tail. The mutated path is removed from the set.

Partial Mutation will create the same two paths as in the case of the full mutation, but with a reduced flow. The flow of these two paths (Φ_m) will be:

$$\Phi_m = \Phi_p - (\Phi_c - \Phi_n)$$

Where Φ_c is the combined flow of the subset of paths selected for mutation, Φ_p is the flow of the partially mutated path, and Φ_n is the negative flow. The partially mutated path won't be removed. Its flow will only be reduced to $\Phi_p - \Phi_m$

Unit Assignment

From the previous step, we got several sets of concurrent paths. For every set, we will compute the time it will take to transport all of the units to the target. We will then select the set which will have the quickest finishing time.

We will then assign unit counts to the paths from the set. A heuristic function will be used to assign the individual units to the paths based on their position.

Finishing Times of a set of concurrent paths can be estimated by first ordering paths by length. Each of them will have a transit time (the time it takes a unit to move from the start to the target using the path) and a capacity (the number of units it can transport per second). We will define the transit of a path p with a transit time t_p and capacity c at the time t as:

$$T_p(t) = Max((t - t_p) \cdot c, 0)$$

The total transit of the concurrent path set P will be defined as:

$$T_P(t) = \sum_{p \in P} T_p(t)$$

We need to find the time when the total transit is equal to the number units. To do this, we can start adding the based on their transit time. When a new path with transit time t_p is added, we can check the total transit at that time $T_P(t_p)$. If the transit is greater than the number of units, the path isn't used, and the finishing time is lower than t_p otherwise, we continue by adding another path. We remember the transit times of the added paths and their capacities.

When we get all the paths we check the last added path and its transit time t_l. At this point of time, the units weren't all yet transported. The number of remaining units is $n = N - T_P(t_l)$ where N is the total number of units at the start. Let C be the capacity of the selected paths, we can compute the finish time t_f as:

$$t_f = t_l + \frac{n}{C}$$

Unit Numbers Once we have selected a combination path and found out its finishing time t_f we can determine, what number of units should be assigned to each path. The number of units assigned to the path p with the capacity c_p and transit time t_p can be calculated as:

$$n_p = c_p \cdot (t_f - t_p)$$

Assignment Heuristic We have found the paths and determined the number of units for each of them, but we still need to select the individual units to be assigned to the paths. For this purpose, the algorithm uses an assignment heuristic.

The paths differ in at least one gate, and each of the gates has a certain number of units assigned. The algorithm finds the first combination of gates where the paths differ. Each of these gates has a certain number of units it should be assigned. Let's call this number n_G for the gate G. For each gate G, we find the n_G closest units. For each of the units, we find out how many gates have it among their closest units. There are two cases:

1. No gate - the unit isn't assigned to any gate yet.
2. One or more gates - the unit will be assigned to the closest gate from the set

After the assignment, there may still be some unassigned units. In this case, the process is repeated - with just the unassigned units.

4.1.3 Execution

Each of the units has a path assigned to it. The path is a sequence of points on the map leading to the target. The unit receives a steering force in the direction of the furthest visible point on the path.

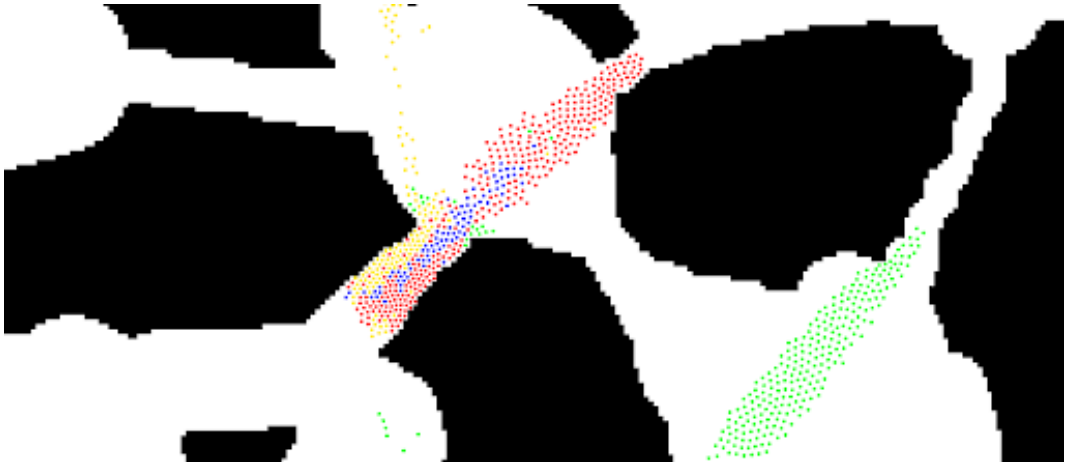
4.2 Observed Problems

When testing the flow graph method, we observed some problems that negatively impact its performance. Some of the encountered problems can be seen in Figure 4.2 and will be further described in the following subsections.

4.2.1 Too Early Path Assignment

The unit assignment to the paths happens during the planning phase when the units are at their starting positions. This is too early. Units could travel a considerable distance before they reach a point where their paths diverge. During this travel, the units from different groups may get mixed up, and they will be harder to separate. Moreover, their position at the start may not be related to their position at the gate.

For these reasons, we think that the assignment should happen during the execution phase and at the last possible moment (at the point where the paths diverge).



Different problems in one image. The green units were trapped by the others and are trying to return to their path using a long route. The yellow units are separating from the red and blue ones, but the separation isn't smooth, and the flow fluctuates. Part of the yellow units is at the end of units going through the chokepoint, which shouldn't happen as their path is longer than the one used by the red units. The end should be made up of the red units.

Figure 4.2: Flow Graph Navigation Problems

4.2.2 Travel Time Consideration

Different streams will start arriving at the target at different times. These travel times depend on the length of the path the stream uses. Flow Graph considers the number of units to be assigned to the individual paths, but not so much which units. There are situations where the units using the longest path are placed at the back of the group and have to wait for the units placed before them. This can increase the finish time. The units with the longest path should be placed in front of the group to allow for a timely finish.

4.2.3 Fluctuating Flow

Our model assigns a flow value to each of the concurrent paths. During the execution, these paths should therefore have at least a similar flow. This, however, doesn't happen, and the flow can fluctuate widely.

4.3 Proposed Solution

We created a new, improved version of the existing Flow Graph algorithm, which we will call the "Regional Flow Graph". Our proposed solution should fix the problems with the flow stream separation, provide an alternative to the path recalculation, better navigate through regions with small obstacles, and reduce both the movement and planning times.

The basic structure of the algorithm is similar to the original, with a few changes that can be seen in Pseudocode 4. The red-colored steps were added to

the original algorithm, and the orange-colored ones were modified.

In the preparation phase, we use our improved version of the water decomposition. We also precompute regional flowfields. For every region and a gate neighboring it, we create a flowfield storing direction toward the said gate.

In the pathfinding phase, we work the same way as the original algorithm up until the point where it assigns the paths to the units. We do not assign units immediately. Instead, the units receive all of the selected paths, and they pick their path at the gates where the paths diverge. Such gates are called divider gates and are also generated in the pathfinding phase. RFG also uses a different path format than the original algorithm, called the regional path. It doesn't store all points on the path, instead storing a gate the units should move to for each region.

During the execution, the units use the regional paths object for their navigation. This allows them to switch between different paths at the divider gates or when they get pushed off their path by other units.

```
Preparation
ImprovedWaterDecomposition()
CreatePartialFlowGraph()
GenerateRegionalFlowFields()

Pathfinding
FinishFlowGraph(target, units)

solutions
concurrentPaths
While AugmentingPathExists:
    path ← FindTheShortestPath()
    SaturatePath(path)
    If BidirectionalFlow:
        solutions.Add(concurrentPaths)
        concurrentPaths.Add(path)
        concurrentPaths = MutatePaths(concurrentPaths)
    Else:
        concurrentPaths.Add(path)

solutions.Add(concurrentPaths)

quickestSolution ← SelectQuickestSolution(solution)
unitCounts ← AssignUnitCounts(quickestSolution, units.Count())
regionalPaths ← CreateRegionalPaths(unitCounts)
regionalPaths.CreateDividerGates()

Execution
EveryFrame:
    regionalPaths.ExecuteRegionalPath()
```

Pseudocode 4: Regional Flow Graph

Figure 4.3: Redundant gate

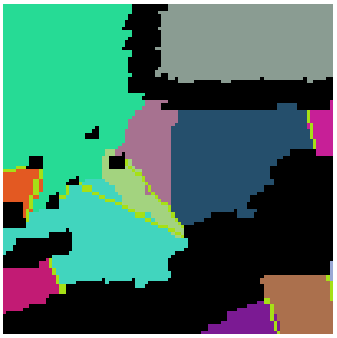


Figure 4.4: Multiple regions enclosed

Figure 4.5: Three point border

4.3.1 Preprocessing

Our method divides the map into a set of regions separated by gates using a modified version of the water decomposition algorithm. A partial Flow Graph is constructed over the map (Flow Graph without a source and sink). We also precompute regional flow fields for every combination of a region and a gate neighboring it. The flow field stores movement directions toward the gate for all of the region's tiles.

Regional decomposition

The regional decomposition should divide the map into a set of regions divided by gates. The original algorithm used the water decomposition by Halldórson for this purpose. This decomposition algorithm has several problems, which will be described in the first subsection. Our improvements to the algorithm will be introduced in the second subsection.

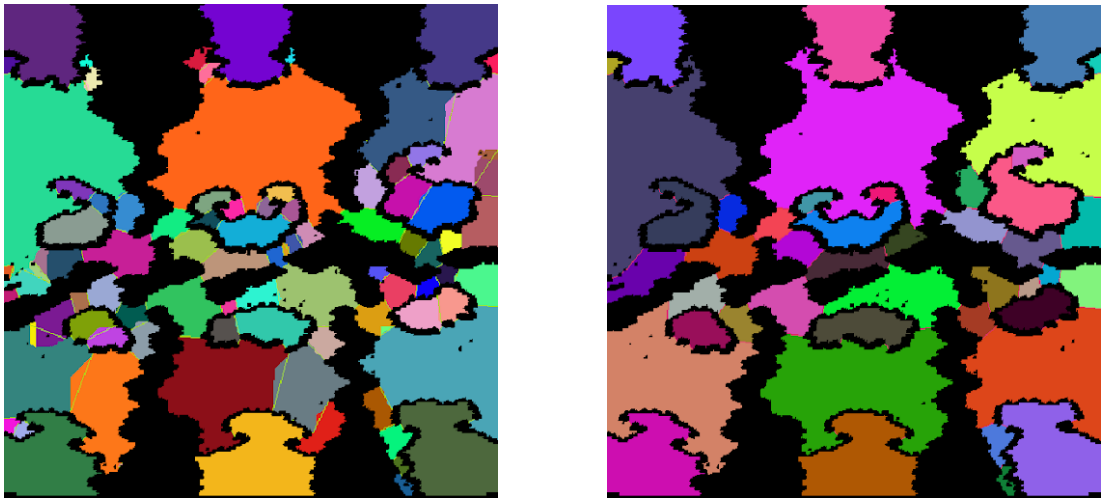
Water Decomposition Problems The water decomposition uses a depth map specifying the distance to the nearest wall. A parameter has to be supplied to the algorithm specifying the intended sensitivity.

A low value of the parameter will mean that a huge number of regions will be created because of small differences in the shape of the walls. While a high value may cause the algorithm to ignore some obstacles altogether.

Different maps require different parameter values for a good decomposition, but the value can not be easily determined automatically. For this reason, the parameter had to be set manually in the experiment run by the original algorithm's author. A good decomposition algorithm wouldn't need such changes, or they would be done automatically because the characteristics of the map may change during the gameplay.

Even with a lower sensitivity to the obstacles, the algorithm may create gates in places where they aren't necessary because it was a meeting point of two "lakes". this effect can be seen in Figure 4.3. When a depth at the gate isn't significantly lower than at the centers of one of the nearby regions, they probably shouldn't be separated.

When the wall sensitivity parameter is set to a higher value than 1, the region borders may appear in unexpected places because the algorithm ignores smaller



a) Halldórson b) Our Modification

Figure 4.6: Water decomposition comparison

obstacles. Triple borders (or even higher) may appear in the open space as seen in Figure 4.5. The algorithm creates gates between two points near the walls, but in this case, they wouldn't exist.

Sometimes the algorithm creates regions not separated by gates. This can create situations when one area surrounded by gates contains multiple regions, as can be seen in Figure 4.4.

Our Modification We created an improved version of the water decomposition algorithm (Figure 4.6) for the purpose of our navigation algorithm. The changes made to the algorithm can be seen in Pseudocode 5. The red-colored steps were added to the original algorithm, and the orange-colored ones were modified.

```
1 CreateDepthMap()
2 BuildRegionsMap()
3 MergeNearbyRegions()
4 RemoveTripleBorderPoints()
5 DetectGateTiles()
6 CreateGates()
7 CleanAroundTheGates()
```

Pseudocode 5: Improved Water Decomposition

1. **Depth Map** - Our version of the algorithm uses the Euclidean distance instead of the Manhattan distance when measuring the distance to the walls.

2. **Merging Regions** - We merge nearby regions if the depth at their border doesn't differ from the depth at the region centers significantly enough. The merging is done until there are no regions satisfying the merging criteria. The merging allows us to set a wall sensitivity parameter to a lower value. There are now two criteria required for a region separation, and the regions created because of smaller irregularities in the obstacle shape will

be merged. Chokepoints with thinner walls around them will, however, still be detected. The merging also often solves the problem of the borders not being separated by gates or triple borders in open space.

The improvement caused by the merging also effectively removes the need to manually set the sensitivity parameter. Manual tweaking may produce better decomposition. The decomposition with the default parameter is, however, typically good enough.

3. **Triple Border Points** - Sometimes, the regional decomposition creates triple border points in the open space. This is a problem for the gate creation step (as the endpoints aren't near the walls). These points are snapped to the nearest wall, which enables the algorithm to correctly create the gates.

4. **Cleaning Around the Gates** - The algorithm also cleans the regions around the gates such that one side of the gate contains only the tiles of a single region and the other the tiles of the other region.

Regional Flow Fields

We precompute a regional flow field for every combination of a region and a gate neighboring it.

1. We create a distance field with distances toward the gate for all of the tiles in the region. The distance field used for the flowfield computation is initialized with 0 at the gate center and progressively increasing values towards both ends of the gate. The distance increase helps to guide the units toward the center of the gate. The higher the increase, the stronger the push.

2. The flow field is then computed as a gradient of the distance field. The flow field stores the movement direction that would result in the shortest path toward the gate.

When we have constructed all of the regional flow fields, we can obtain a movement direction by specifying the unit's positions, the region it is in, and the target gate.

4.3.2 Pathfinding

The plan is created in the same way as in the original Flow Graph algorithm up until the point where the paths are created and assigned to the units.

Our algorithm doesn't use the simple paths from the A* algorithm but a structure called the regional path. The units also aren't assigned their path during the planning. Instead, a divider gate object is created for every gate where different flow streams separate, which will separate the units during the execution.

Divider Gates

A divider gate is an object placed at a gate that divides its flow between several goal gates. The division happens directly at the gate, which removes the problems with the mixups of different streams during the execution and reduces the flow fluctuations.

The divider gate needs to have information about the flow streams that use it, the number of units assigned, their flow, and the goal gate. The units are assigned to the streams proportionally based on the flow of the stream until the target number of units is reached. The gate is divided into sections, each corresponding to one of the streams. The size of these sections at the beginning is proportional to their share of the total flow going through the gate and is adjusted during the execution. The sections are ordered based on the dot product between the direction of the gate and the direction toward the goal gate.

Regional Paths

A regional path is a structure storing a sequence of gates leading to the region with the target. For every region on the path except the final one, it also specifies a gate the units should navigate towards. For the final region, the regional path contains a path from the final gate to the target (in a standard A* format).

Because the units do not know which path they will use during the navigation, they have a list of all the regional paths created during the pathfinding phase. To use the paths, they also need the regional flow fields. And also a dictionary of the divider gates to enable the assignment to different paths.

4.3.3 Execution

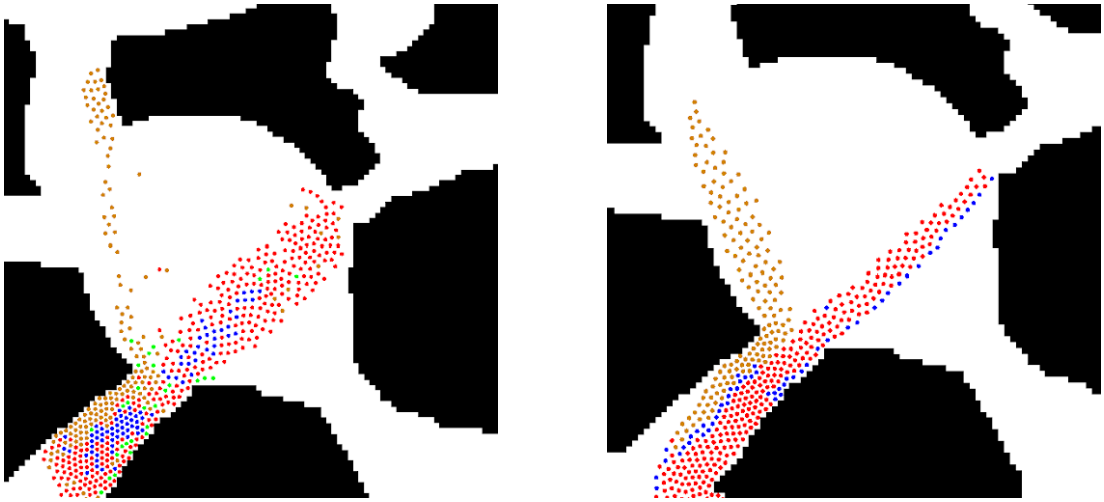
The units get their steering direction from the regional paths object. The object also manages the unit's interaction with the divider gates.

Regional Paths

The process of obtaining a steering force from a regional path is slightly more complicated than obtaining it from a regular path. The process of doing so is shown in Pseudocode 6.

```
1 If CurrentRegionOnthePath:
2     UseRegionalFlowFields
3 Else If InFinalRegion:
4     FollowFinalPath
5 Else:
6     If OtherPathContainsCurrentRegion:
7         SwitchPaths
8     Else:
9         Repath
```

Pseudocode 6: Regional Path Navigation



Flow Graph Regional Flow Graph

Figure 4.7: Flow Stream Separation Comparison

The unit has one of the paths assigned to it, which it uses during navigation. To obtain the steering force, it checks its current region. If its path contains this region, it gets the region's target gate. The steering direction is then obtained from the appropriate regional flow field.

When navigating in the final region, the standard path-following navigation is used to execute the final path.

It may happen that the unit enters a region not used by the path. In such a case, it checks if there exists another path containing this region. If this is the case, it switches to the other path. We call this the soft re-path. If no such path exists, a standard hard re-path is done - the whole path to the target is recalculated.

Soft re-paths are preferred over hard re-paths because there is no need for recalculation. The soft re-paths also reduce the problem of the units getting carried away by another flow stream. When they enter another region, they automatically switch to the other stream, while in the original solution, they often travel great lengths to return to their original stream, which can greatly increase the finishing times.

Divider Gate

The divider gate is split into sections, each corresponding to one of the streams. When a unit arrives at the gate, it enters one of these sections. The unit is then assigned to the corresponding stream. This ensures a cleaner separation performed directly at the gate. The resulting improvement can be seen in Figure 4.7

The sizes of the sections are updated when the unit passes through. The size of the section used is decreased while the size of the other sections increases. This modification is done so that the streams are always assigned the fraction of units proportional to their share of the flow. The size of the section used by the unit is modified as:

$$s = s - \frac{\Phi_t - \Phi_s}{\Phi_t} \cdot Min(Max(\frac{1}{n+10}, 0.01), s \cdot 0.25)$$

- s is the relative size of the section (1 being the size of the gate)
- Φ_t the total flow through the gate
- Φs flow through the section
- n the number of units that passed through the section so far.

The flow values used in the computation are the ones assigned to the different streams passing through the gate modified by a factor, which decreases their value when they have only a few units left. The flow value reaches zero when the stream has no units.

5. Evaluation

This chapter will describe the experiments we ran using our simulator and their results. We need to evaluate the performance of our algorithm (Regional Flow Graph). Our algorithm will be compared against the standard pathfinding algorithms (A*, flow field) as well as against the original Flow Graph algorithm. We will also try to evaluate how closely our algorithm adheres to its predicted behavior and possibly improve our predictions to match reality more closely. The chapter will also describe the problems the algorithm encountered on some of the maps.

5.1 Goals

The experiments are run with several goals in mind. These goals will be introduced in this section.

5.1.1 Comparison To Other Methods

We would like to compare the performance of the proposed method against other pathfinding methods. Their quality will be measured on different metrics. The most important ones are:

1. **Finishing Time** - the time it took all of the units to reach the target (measured in the in-game time)
2. **Pathfinding Time** - the time it took to compute the plan for the units to follow.

The algorithms will be compared on a set of maps with a wide range of characteristics to find out their performance in different circumstances.

On some maps, the methods will also be compared manually in the graphical mode because the navigation time isn't the only thing important for an algorithm. In a real application, it is often not a problem if the units navigate slightly longer, but a chaotic or otherwise unnatural behavior of the units can be a bigger issue.

5.1.2 Finding Errors

The theoretical analysis of the algorithm isn't typically enough to predict the algorithm's behavior. Some problems and errors in the algorithm can only be discovered by thorough testing. The pathfinding experiments, where the algorithm won't perform well, will be looked at in graphic mode. We will document the problem, describe its causes, and, if possible, come up with solutions.

5.1.3 Measuring The Prediction Accuracy

Our method allows us to get an estimate of the times of the unit arrivals to the goal and the gates on the paths. We would like to compare the predicted times to the real times measured during the execution. If the predicted times are

inaccurate, we will investigate the reasons for these errors and try to correct the algorithm.

5.2 Experiment Setup

This section will describe the specifics of the experiments we are running. The pathfinding methods used, the pathfinding problems used, and the metrics measured.

5.2.1 Compared Methods

We will measure the performance of four pathfinding methods - A*, flow field, Jan Pacovský's Flow Graph, and our method, the Regional Flow Graph. Some of these methods can have different variations and run with different settings. We will have to describe the specifics of the algorithms used in this thesis.

A*

The path is computed as described in the Related Work chapter, but in our implementation, we increase the cost of the tiles next to the walls. This helps to reduce the wall hugging the A* is often prone to. The result is a sequence of tiles (or their positions) the unit should pass on its way toward the target.

When following the path during the execution, the unit navigates to the furthest point on the path visible from its position. Finding the furthest point on the path visible from the position of the unit can be a time-consuming task. For this reason, the furthest visible point isn't updated every frame but is recomputed only after a certain period of time. In our implementation, we set this period to 5 seconds. The length of this period is calculated based on the distance from the current position of the unit and the previously computed point. The length then receives a random modification (to avoid a large number of recomputations at a similar time).

Computing a path for each unit will be very time-consuming if we are trying to find a path for a large number of units (the unit numbers in the experiments are typically between 1000 and 2000). We should reuse the paths if possible. The path for a unit is only computed if there already isn't another path visible from the unit starting position. Otherwise, another unit's path is used. The path can be used because the unit steers to the furthest visible point on the path, and we are guaranteed that the first point is visible.

Flow Field

Standard flow field navigation method as described in the Related Work section. A distance field is built by performing a breadth-first search from the target position. This field is then used to get a direction vector toward the goal. The direction is a gradient of the distance field (in some other implementations, it is the direction to the tile with the lowest distance). The flow field is the same for all of the units and is therefore constructed only once.

The execution phase uses the constructed flow field to find the movement direction. For each of the tiles we can retrieve a movement vector from the corresponding flow field cell. The unit isn't typically positioned exactly at the center of the tile. For this reason, the resulting movement vector isn't simply the direction stored in the unit's tile but a combination of the directions stored in the nearest 4 tiles. The directions are weighted based on the distance between the unit and the corresponding tile.

Flow Graph Methods

The Flow Graph and Regional Flow Graph methods work as described in the Implementation chapter. Both of these methods use simple paths that work exactly the same way as in the A* method. RFG also uses flow fields. Like the flow field method, the units obtain their movement direction as a weighted average of nearby tiles.

5.2.2 The Problem Set

The algorithms will be evaluated on a set of 200 pathfinding problems, each of them specified by a map, starting positions of the units, and the position and size of the target. 100 of these pathfinding problems were run on custom-created maps, the other 100 on maps from existing RTS games.

Each of these 200 experiments will be run using all of the 4 pathfinding methods. There will be 10 runs for each of the methods, and the results will be averaged. This is because the navigation can be somewhat non-deterministic, and we should try to minimize the effect of luck in the runs. The slight non-determinism comes from the unit model we are using, and more information can be found in the Unit Movement Randomness section.

Custom Maps

We created 23 custom maps and specified 100 pathfinding problems on them.

Part of the maps are designed to be simple to test some of the algorithm's characteristics while being able to ignore other aspects of pathfinding (Figure 5.1). These include maps with a single path with different gate sizes and shapes (constant size, decreasing size, rough walls, funnel shape, etc.). Another portion of the simple maps are those with different alternative paths, where the division should be simple.

Other custom maps are designed to test some potentially problematic scenarios (Figure 5.2). Weird region shapes like regular rectangles or concentric circles could throw off the regional decomposition. Large gate sizes could throw off the distance estimation (as it always uses gate centers). Some maps have many flow streams passing through a single central region.

RTS Maps

The other half of the experiments use maps selected from the "Moving AI" repository. We selected only RTS game maps (Figure 5.3). Artificially created maps, or those from non-RTS games, weren't used. The purpose of this selection is to test

the algorithm on maps from real RTS games and hopefully find problems that wouldn't occur on simpler custom-created maps. We selected 33 maps from Starcraft and 21 maps from Warcraft 3 and created 61 and 39 pathfinding problems for them, respectively.

Figure 5.1: Simple Custom Map

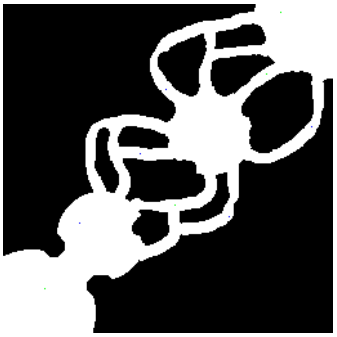


Figure 5.2: Problematic Custom Map

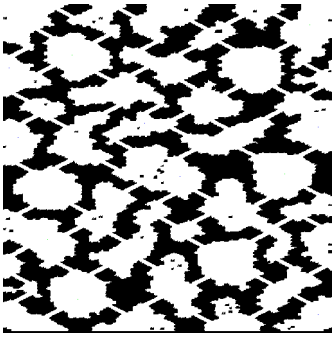


Figure 5.3: Starcraft Map

5.3 Testing Environment

We created a pathfinding simulator in the Unity game engine to allow us to run the experiments. The simulator is capable of running a large number of pathfinding requests on various maps using different navigation methods. This chapter will describe its internal functions and the navigation methods it implements.

The main purpose of the simulator is to simulate the unit navigation. For this purpose, it provides a model of a unit common for all of the pathfinding algorithms (the algorithm can only change the direction of the seek force). This allows for a fair comparison between the algorithms.

The simulator manages the map and stores information about the passability, position of the target, and units. It can also store a decomposition of the map into disjunct regions with gates in between them.

The program allows us to load different experiments. Each of the experiments has an obstruction map, unit start positions, target position, and a target size specified. A list of experiments can be run automatically, each of them multiple times with different pathfinding algorithms.

5.3.1 Simulation Setup

To test the navigation algorithms thoroughly, we need to be able to easily create, load, and run a large number of pathfinding problems/experiments. A single pathfinding problem is specified by two textures, the map texture and the unit start texture, and an integer describing the size of the pathfinding target. These can be easily created and combined (different unit start textures for the same map texture).

The map texture (Figure 5.4) describes which map tiles are obstructed and which are freely passable. Each pixel corresponds to a single tile; a black pixel means that the tile is obstructed, while passable tiles have a different color.

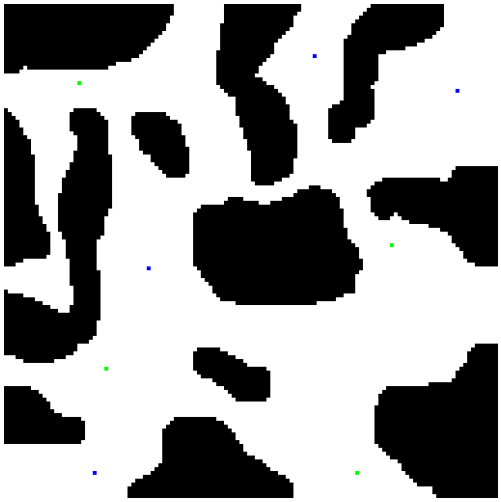


Figure 5.4: Map Texture

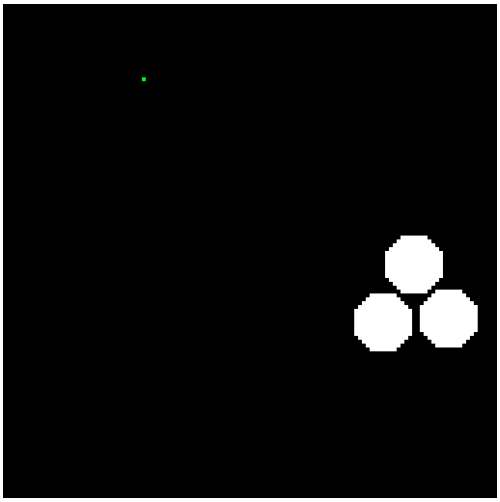


Figure 5.5: Unit Start Texture

The texture also specifies information for the warmup. The warmup is a series of pathfinding requests performed before the "real request." A start point and an endpoint are specified, and a plan is created for them using a specified algorithm, but no unit movement is performed. The purpose of this computation is to simulate the algorithm in the actual game, where the pathfinding request is typically performed after some other request was already executed before it. The starting positions are marked as green, and the target positions as blue pixels, and the pathfinding is done for every pair.

The unit start texture (Figure 5.5) describes the starting positions of the units. A white pixel in the texture means that the corresponding tile contains a unit at the start of the experiment. This allows the user to specify starting positions for a large number of units very quickly. The texture also contains the position of the target (the position the units should move towards) marked by a green pixel.

A single map texture is often reused for different simulation setups with different unit and target positions.

5.3.2 Automatic Simulation

The simulator can automatically run a large number of experiments. The experiments can be specified by a list of simulation setups, a list of pathfinding methods, and a number of simulations. For every combination of a pathfinding method and a simulation setup, a number of simulations specified is performed.

The results of the simulation can be saved into a text file. The name of the simulation setup specifies the folder, and the name and the number of the run are combined into the name of the resulting text file.

5.3.3 Unit Model

The simulator provides a common model of a unit for all of the pathfinding methods. The purpose of this model is to make the method comparison more fair.

Each unit is moved across the map by a number of steering forces. The unit separation and the unit avoidance forces help it to avoid other units. The wall repulsion and turning force move it from the walls, and the alignment force makes it mimic the movement of nearby units. These forces are independent of the pathfinding method used. The only force that is influenced by the method is the seek force, which forces the unit to move toward the goal.

The combination of the forces influences the movement direction and speed of the unit. The speed has a maximum value (common for all of the units), and if it is surpassed, it is scaled down. The new position of the unit is then computed using the movement direction, speed, and time from the previous update. The new position of the unit can not make the unit share space with a wall or another unit. If such a problem occurs, the unit is forced to stop its movement.

Unit Movement Randomness

The movement of the units may vary between different runs of the experiments, resulting in different finishing times. This is because the unit's movement isn't deterministic. The effect of randomness on the run performance is typically small. for example, in some of the runs, the flow stream separation may work slightly better than in others. The unit model has two main sources of randomness:

Wall Repulsion Force - the unit casts rays around itself. When a ray hits a wall, the unit receives a repulsion force based on the distance to the wall and the direction of the ray. The directions of the rays the unit cast receive a random offset every frame. Randomness helps resolve some situations where the unit may get stuck but make the unit's movement non-deterministic.

Path Following - When the units follow a path (normal path, sequence of positions), they try to navigate to the furthest visible point on the path. The unit doesn't update the seek target every frame but only after a certain period. This period receives a random offset to make the units do these calculations in different frames.

5.3.4 Resolving Errors

When moving through the map, the units may get stuck or be unable to see any point on their path. The simulator provides methods of dealing with the most common problems.

Repath

If a unit isn't able to move according to the plan computed in the pathfinding phase, it needs a way of finding a path to the target. We will call the units which are not able to follow the plan lost. If any unit becomes lost, a new path is computed between its current position and the target using an A* algorithm. This is called a repath.

Repathing can only occur with certain algorithms. As the flow field has a movement direction for any position on the map (if there is a path between it and the target), it can not trigger a repath.

Failure

There are situations in which units may get stuck and unable to move to the goal. The simulator has a time limit for the execution (in the game time), which is currently set at 5000s. If this limit is surpassed, the experiment run fails, and the information about the failure is written into the file with the results.

5.4 Metrics

For each algorithm, a number of metrics will be collected and analyzed. One metric can be used to measure different aspects of the algorithm. Despite this, we can still divide the metrics into several categories by their purpose or the thing they are measuring.

Performance Evaluation

These metrics are used to evaluate the performance of the algorithm. Both in terms of the speed of the plan creation as well as the time it takes the units to reach their target.

- Preparation time (time spent preprocessing the map without knowing the target or unit position). The preparation is done before any movement request, and its length isn't too important. A long preparation could, however, be problematic for maps that often change during the game because this would force a new map analysis.
- Pathfinding time (time spent finding the path for the units). The creation of the plan for the units to follow shouldn't take too long. The units commanded by the player should be able to react to a request to move to a certain position almost instantaneously.
- Movement time (the time it takes all the units to reach the target). The most important metric for the algorithm's performance is measured using the in-game time.
- 90% movement time (the time it takes for 90% of the units to reach the target). The movement time can be influenced by a single unit getting lost, stuck, or carried away from its path. This metric reduces the influence of such events.

Moreover, we would also like to know how many times the plan failed and the unit had to readjust. For this, we have two metrics:

- The number of hard re-paths (unit had to recompute its path to the goal). A high number of re-paths can be a problem as a new path has to be created for each of the units doing a re-path. This can significantly slow down the execution.
- The number of soft re-paths (unit had to use an existing path intended for different units). A higher number of soft re-paths isn't as problematic

because the unit simply switches to another already created plan. It can, however, lead to a wrong number of units being assigned to a flow stream or be a signal of other potential problems.

Arrival Times

We would like to gather the times (from the start of the experiment) each unit arrived at certain important milestones on their path. These include the target and each of the gates the unit passes through. We will not just gather the times but also the ID of the unit, the object (gate or target), and the ID of the flow stream of the unit.

The arrival times aren't that useful on their own, but they can be used for further analysis and to compute other metrics. Following is a list of metrics that can be derived from the arrival times.

- Movement times from predecessor gates are based on the first few units from the stream. Are we predicting the time it takes to move between the two gates correctly?
- First unit arrival times for each of the incoming streams. These can be compared to the predicted arrival times.
- The total number of units passing through the gate, both for the whole gate and individual streams. The streams have a certain number of units assigned, but this can change in the case of re-pathing.
- Outgoing and incoming flow. This is the number of units arriving each second in the case of the incoming flow. The incoming flow can be computed from the outgoing flow heading to the gate from the predecessor gates offset by the movement time from the said predecessor gate. The outgoing and incoming flows can also be computed for individual streams and, in the case of the outgoing flow, used to check the correct functioning of the divider gates.
- Overflow. The number of units accumulated at the gate at any point in time. It can be computed from the incoming and outgoing estimates by the map analysis.

Travel Distances

The travel distances are recorded any time a unit passes through a gate or arrives at the target. The distance is the total distance the unit traveled from the start (not just from the previous gate). And just as with the arrival times, we don't gather only the distances but also the unit ID, the gate, and the ID of the flow stream the unit belongs to.

From these distances, we can also get the length of the path units used between two neighboring gates. These can be compared to those predicted by our algorithm.

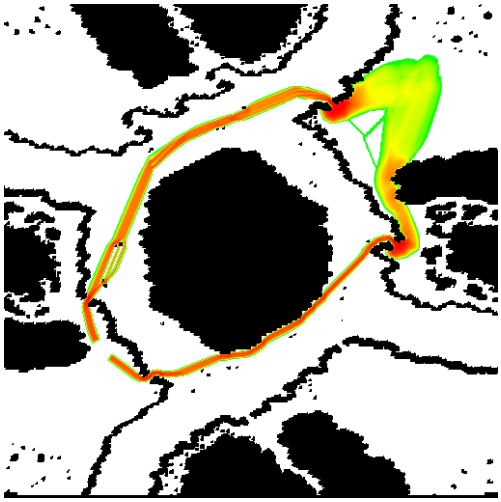


Figure 5.6: Movement Map

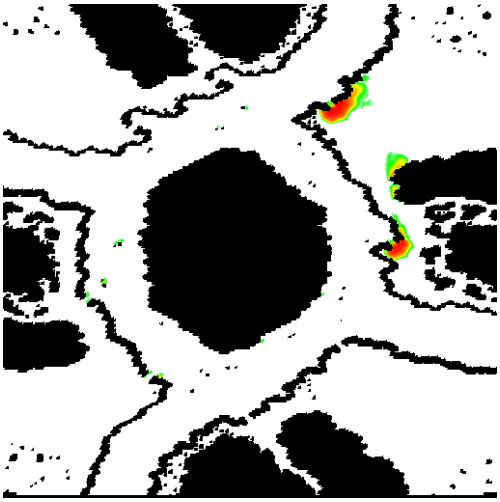


Figure 5.7: Stuck Map

Unit Paths

We would like to visualize the paths the units use during their movement. This would allow us to quickly gauge the different paths used by the units, what portion of the units are using the path, and also, to some extent, the places where the units accumulate.

For each of the tiles, a float number is stored, measuring the usage of the tile by the units. Every second, the units record their positions. Each of these positions has 4 nearby tiles. The number stored in the tile is increased based on the distance between the tile center and the recorded position. The numeral values can be visualized using a logarithmic color scale and displayed as an image called the movement map (Figure 5.6). The movement maps are generated for all pathfinding problems and methods (single map for all the runs) and are available as a part of the Digital Attachment.

Units Getting Stuck

It could be important to identify the places the units find problematic. We identify these places by the units getting stuck there. We say that the unit is stuck if, during the last second, it moves slower than 25

For each of the units, we will record the time it spent stuck. We will also visualize the places where the units are getting stuck with the same method used to visualize the paths producing a stuck map (Figure 5.7). This time, units increase the values stored in the tile only if they are stuck. The stuck maps are generated for all pathfinding problems and methods (single map for all the runs) and are available as a part of the Digital Attachment.

Some of the places where the units are getting stuck are predictable (like the chokepoint entrances) and don't represent a problem or error. In some cases, however, a real problem could be identified.

6. Results

Our algorithm was compared to three alternative pathfinding algorithms using a number of different metrics. We also tried to investigate the experiments where the algorithm didn't perform well and describe the problems and their causes, and possibly suggest solutions.

Overall the algorithm performed better in terms of the time it took for it to find a plan as well as in terms of the overall movement times. This, however, required a longer map preparation. A more detailed comparison of our algorithm to the other methods will be made in the first section of this chapter.

The second chapter will focus on the arrival times and the reasons for the differences between the real arrival times and predicted ones. The third chapter will discuss the reasons for the performance differences and the problems our method faced.

6.1 Comparison To Other Algorithms

There isn't any singular measure of the quality of an algorithm. The results of the comparison in different metrics will be described in the subsections of this section.

6.1.1 Movement Times

The most important metric for the pathfinding algorithm's performance is the time it took before all of the units reached the goal. Each of the experiments had 10 runs for each of the methods. We will get the median from these runs, which will eliminate the lucky and unlucky runs and give us a better understanding of the algorithm's performance.

To easily compare the different algorithms, we computed the relative performance of the algorithms - a ratio between the time of the algorithm and the time of the best method. These ratios can be seen in Table A.1 located in the Attachments. The relative performances are highlighted using a color scale, with the red color being the worst and the green being the best. Table 6.1 displays the average relative ratios for different groups of experiments (based on the source of their map).

We can see that, in general, our method achieves the best finishing times. In fact, it is the best method (sometimes jointly) 121 times out of 200, and its average relative performance is 1.03. The flow field is the second-best method

	Flow Field	**A***	**Flow Graph**	**RFG**
Custom Maps	1.0864	1.168	1.168	1.030
Starcraft Maps	1.212	1.356	1.104	1.029
Warcraft 3 Maps	1.095	1.254	1.189	1.046
All Maps	1.1265	1.242	1.152	1.030

Table 6.1: Relative finish times (mean)

	Flow Fields	A*	Flow Graph	RFG
Custom Maps	1.098	1.181	1.104	1.014
Starcraft Maps	1.231	1.376	1.05	1.014
Warcraft 3 Maps	1.11	1.268	1.07	1.017
All Maps	1.141	1.258	1.081	1.014

Table 6.2: Relative 90% finish times (mean)

achieving an average of 1.126. The original Flow Graph method is third, with performance only slightly worse than the flow field, but in one experiment, it isn't able to finish at all. The A* achieves by far the worst time and is, on average, 25% slower than the best method.

The origin of the map didn't seem to have had a significant impact on our algorithm's performance. Despite a relatively good performance on average, there were experiments where our method performed significantly worse than the others. The reasons for it vary, and they will be described in the problems section of this chapter.

We also measured the times it took for 90% of the units to reach the target. This significantly reduced the impact of a few units getting lost, carried off their path, or stuck along the way. The relative times can be seen in Table A.2 in the Attachments. The mean relative performance in different groups of pathfinding problems is displayed in Table 6.2. We can see that the performance of the Flow Graph methods significantly improved. The average relative performance of the original method was 1.08, and our method achieved 1.014. The performance of the other two methods didn't change much. This means that both of the Flow Graph methods have a problem with single units getting stuck or carried off their path. The problem was, however, significantly reduced in the regional Flow Graph method (though not completely so).

6.1.2 Computation Times

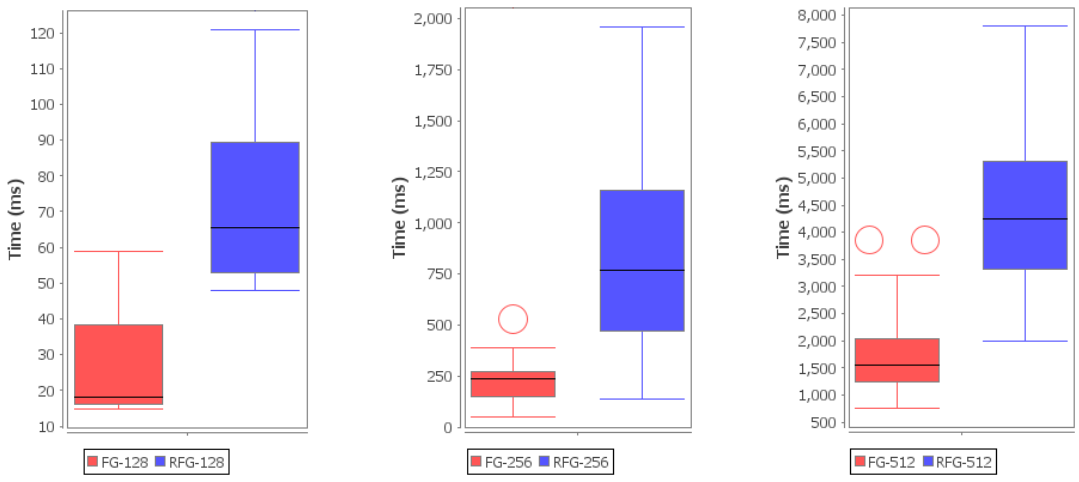
The navigation is performed in three stages, and for each, we can measure the time it takes to calculate.

Preparation

First is the preparation phase, where the analysis of the map is performed. The A* and flow field methods do not perform any analysis. The Flow Graph decomposes the map and computes the distances between the gates. The Regional Flow Graph does the same, but on top of that creates flow fields for each region and a gate neighboring it.

Table 6.3 shows us that the preparation times of the Flow Graph are, on average, about three times quicker than those of the Regional Flow Graph. The preparation times and their ratio, however, very much depend on the map. This can be seen in Figure 6.1. If the map contains big regions with a lot of gates, the Regional Flow Graph is slower. This is because it needs to compute a large number of big flow fields.

The preparation for Regional Flow Graph on 512x512 maps usually takes 3 to 5 seconds. But in some extreme cases may reach almost 9 seconds. This speed



128x128 Maps 256x256 Maps 512x512 Maps

Figure 6.1: Preparation times on different map sizes

	Flow Fields	A*	Flow Graph	RFG
All Maps	0ms	0ms	794ms	2150ms
128x128	0ms	0ms	31ms	90ms
256x256	0ms	0ms	219ms	807ms
512x512	0ms	0ms	1690ms	4344ms

Table 6.3: Average Preparation Times

probably isn't efficient enough for a frequently changing map and can only be used for a less frequent map analysis. However, our implementation of the preparation isn't optimized with ECS, so the algorithm could probably be modified for better performance.

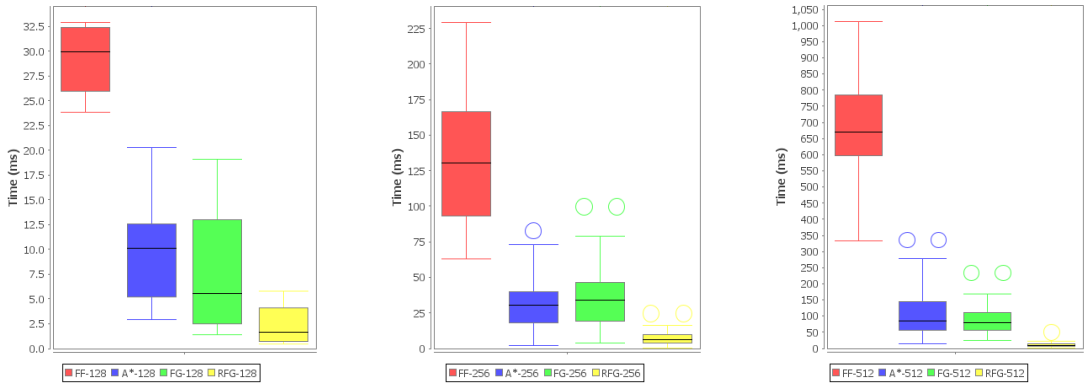
Pathfinding

When the units receive a movement request, they should be able to react quickly and start moving. This is why the navigation methods should have quick pathfinding. Table 6.4 displays the average pathfinding times for different map sizes.

We can see that our method performed by far the best. Creating the plan took, on average, just 8.3ms. This fast pathfinding was enabled by the map analysis done in the preparation phase. The pathfinding times also didn't increase too much with the map size. This is because pathfinding is not performed over the tiles but the regions. Therefore, the pathfinding times depend not on the map

	Flow Fields	A*	Flow Graph	RFG
All Maps	345ms	63.8	71ms	8.4ms
128x128	23.8ms	7.7ms	11.7ms	2.3ms
256x256	137ms	35ms	31.2ms	7.1ms
512x512	683ms	131ms	117ms	11.5ms

Table 6.4: Average Pathfinding Times



128x128 Maps 256x256 Maps 512x512 Maps

Figure 6.2: Pathfinding times on different map sizes

	Flow Fields	A*	Flow Graph	RFG
More Than 10 Repaths	0	43	50	20
More Than 100 Repaths	0	1	6	5

Table 6.5: Maps By Repath Numbers

size but on the map complexity.

The original Flow Graph method, which on average took 71ms to create a plan, could be modified to achieve similar speed, but it would make its preparation just as slow as the one used by our method. Out of the two standard methods, the A* was significantly faster, completing the pathfinding in 64ms on average. The flow field took significantly longer because it needed to go through all of the tiles. On average, the plan creation took 345ms.

Execution

A good navigation method should be able to run smoothly during its execution phase. All of our methods were able to achieve this for the most part.

The only problems happened when a large number of forced repaths were triggered in a short period of time. This resulted in a drop in the FPS. These drops happened for all of the methods except for the flow field, which doesn't do repaths.

6.1.3 Repathing

The repathing is used by 3 methods - A*, Flow Graph, and Regional Flow Graph. In our method, we tried to decrease the number of repaths by changing them into soft repaths, which don't require path recalculation. We succeeded in doing so, but at the same time introduced a new situation where the repathing is triggered. This problem can be seen in Figure 6.10. Because of this problem, the number of maps with a high number of repaths didn't decrease much compared to the original Flow Graph algorithm, as can be seen in Table 6.5. Because the reason for the repaths is different, they occurred in different pathfinding problems. In the attachments, we provide a table with the number of repaths for all pathfinding

	Q0	Q1	Q2	Q3	Q4
Relative Difference	-37%	-9.8%	-5.3%	-1.9%	+25.4%
Absolute Difference	Q0	Q1	Q2	Q3	Q4

Table 6.6: Arrival Prediction Errors

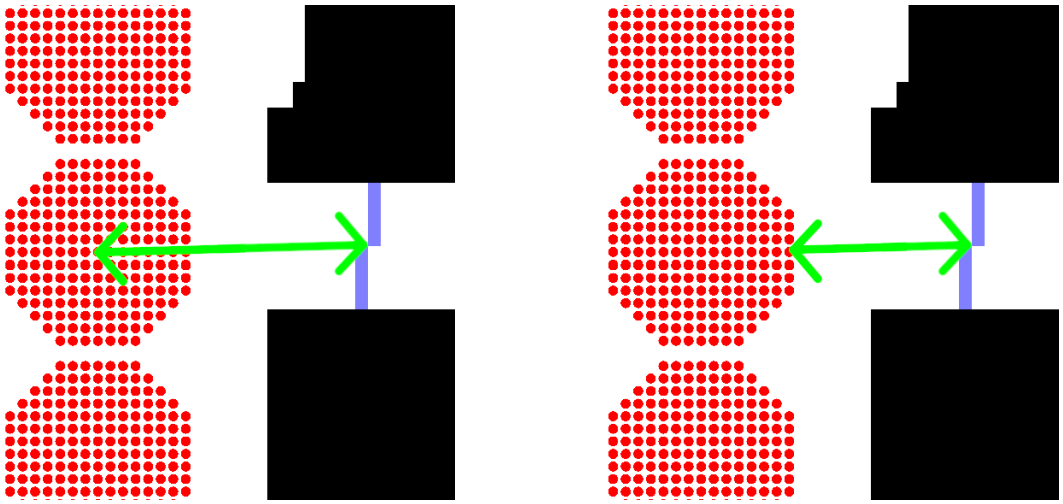


Figure 6.3: Distance to the first gate using: the source node (left), the closest unit (right)

problems (Table A.3). This table also includes the number of soft repaths in a separate column.

6.2 Arrival Times

Our algorithm needs to estimate the time the units reach the target using the RFG method. We would like our estimate to be as close to reality as possible. This is the reason why we measured the arrival times for all of the experiments. We compared them to the predicted arrival times. The comparison can be seen in Table 6.6. It displays the absolute as well as a percentage difference between the predicted and real times. The negative values mean that the units were faster than predicted. The table displayed quartile values. The values for individual units are in the digital attachment. The arrival times of RFG were usually quicker than the predicted times, except for the experiments where we ran into problems.

We also evaluated the estimates for the arrival times at the gates to get a better picture of the reasons for the prediction errors. The median difference between the real and predicted times was 8.3s or 9.7%, with the real arrivals being quicker. The main reason for this difference was a wrong estimate of the arrival times at the first gate on the path. In our method, we represent the units as a single source node placed units' center of gravity. Many units will, however, be significantly closer to their first gate than the source point, resulting in a quicker arrival time (Figure 6.3). The median difference between the real and predicted first gate arrival times was 10.7s.

The arrival estimates improved when we adjusted for the wrong prediction at the first gates. The median difference was 2.7s or 3%, and the real arrival times

were slower. The average deviation was 6.2s. We are still determining the reason why the real arrival times were slower than the predicted ones. It may have been caused by execution errors, units near the walls not moving at full speed, or some other factors.

6.3 Discussion

In this section, we will describe the reasons for the performance differences between the algorithms. We will also describe, analyze and try to propose solutions for the errors that occur in our algorithm.

6.3.1 Performance Differences

We should describe the reasons for the differences in the finish times of the different methods.

The most important reason for the better performance of our method is the separation of the units into different unit streams and, thus, better utilization of different paths. The differences between the methods in this regard can be seen in the movement maps (Figure 6.4).

The A* uses only a single path, and we can see the units gathering at the entrances to the chokepoints. There they have to wait for the units in front of them to pass through, slowing down the navigation. The flow field is somewhat better as it can sometimes utilize multiple paths. This is because the units always use the shortest path to the target. When they gather up at a chokepoint, for some of the units at the side of the group, an alternative path may become shorter, and they will use it. The flow field will therefore be usually equal to or faster than the A*.

Both the Flow Graph and the Regional Flow Graph are capable of effectively using the alternative paths. In this regard, there isn't much difference between them. The Regional Flow Graph should, however, be better at sending the appropriate flows at the right times.

The next important factor influencing the speed of the methods is the navigation errors. The different groups of units may have a hard time separating from each other, blocking each other's way and causing delays. Some units may also be carried off their path by the units belonging to a different group. They may then have to travel long distances to get back to the path assigned to them by the planning.

These navigation errors don't happen with the A* and flow field methods because they do not separate the units into different groups. We can see them visualized on the movement maps of the Flow Graph and Regional Flow Graph methods. The main paths used by the units are seen in orange and yellow. Between them (and sometimes also in other places) are the areas painted green. These are the travel paths of the units that weren't separating effectively or are trying to return to their path using sometimes unorthodox routes.

If we compare the movement graphs of the Flow Graph and Regional Flow Graph method, we can clearly see that these navigation eros occur more frequently when the Flow Graph is used. They still occur when using the Regional Flow

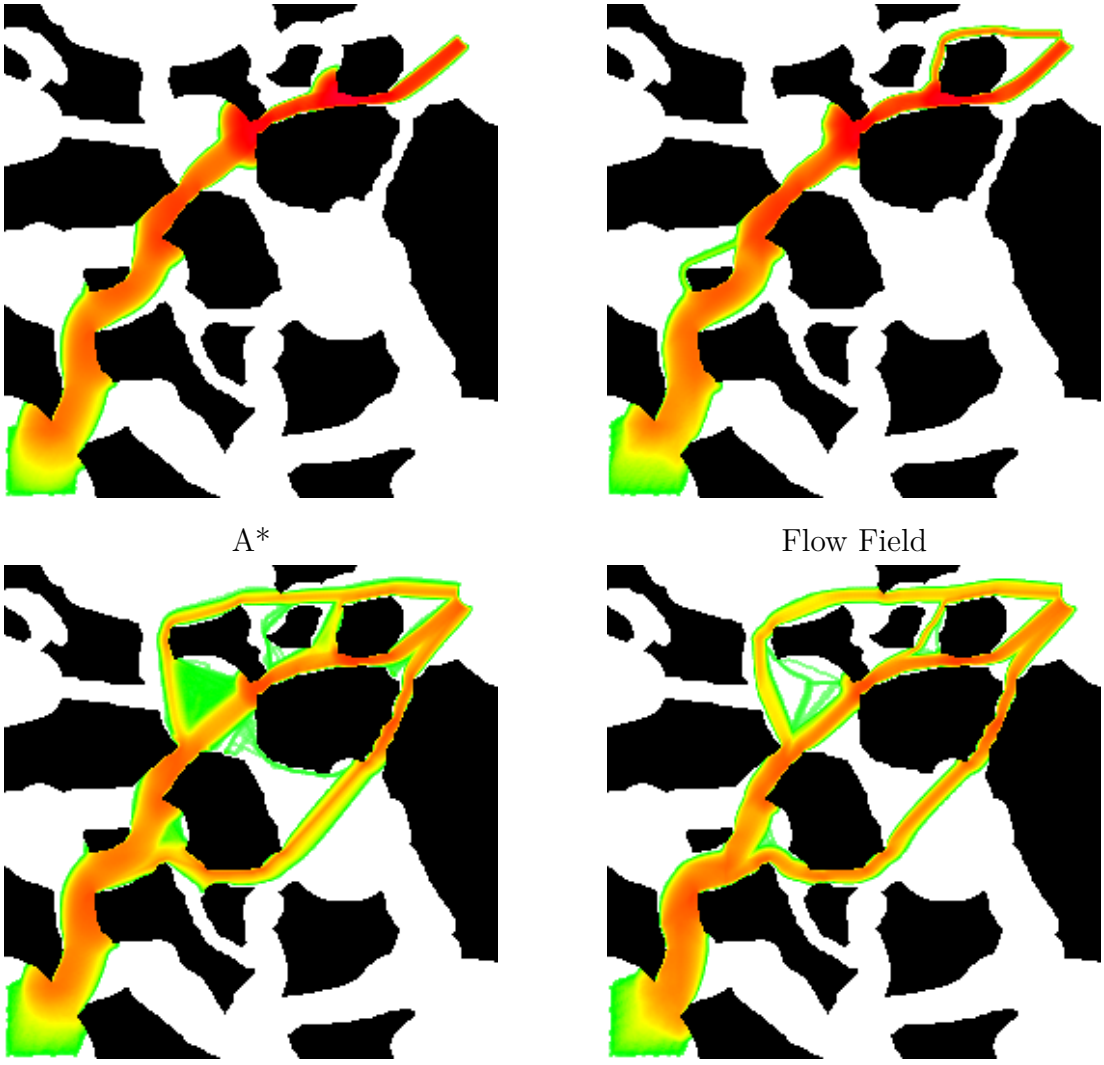


Figure 6.4: Comparison Of Movement Maps From Different Methods

Graph, just less frequently. We can see this from the cleaner separation between the different paths.

6.3.2 Problems

When running the Regional Flow Graph algorithm, we saw that in some experiments, its performance was significantly worse than in others. We ran these experiments in the visual mode and investigated why this happened.

We found some problems and inadequacies in our algorithm. This section will try to describe, explain, and, when possible, propose solutions to these problems.

Streams Crossing

In Figure 6.5, we can see the yellow units trying to reach gate A, while the red units are heading towards gate B. The units from these two streams mix up with each other, some may get carried away by a different stream, or their path may

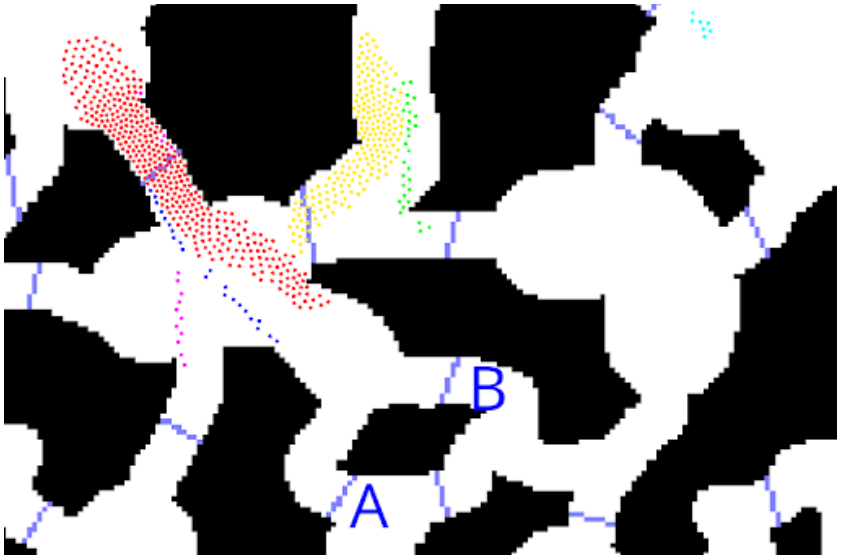


Figure 6.5: Streams Crossing

be blocked for a significant amount of time. This situation increases the finishing time significantly.

The problem occurs because the method doesn't check if the unit paths cross each other. It also doesn't generate all the possible path combinations (because the number of these combinations is very high), which would contain a non-crossing set of paths faster or equal to those used.

Possible Solution - When we find the combination of paths to use, we can check if they cross each other in any of the regions. If this is the case, the paths can be modified to remove the crossing. In this example, the yellow units would go through gate B and the red units through gate A.

Divider Gate Stream Assignment

In Figure 6.7, the green units are heading to the right gate, and the blue ones are heading to the left. Their paths cross, which causes problems and increases the length of the simulation. The assignment of the units at the gate needs to be corrected. The streams should be switched.

The problem occurs when the target gates are located "behind" the divider gate. This is because the segment of the gate the stream received is based on a dot product between the gate direction and a direction to the target gate. The gate direction is the direction between the start and end of the gate. The dot product starts decreasing when the target gate is located "behind" the divider gate, which results in an incorrect segment assignment.

Possible Solution - The divider gate stream assignment shouldn't be based on a dot product. The angle between the direction toward the target gate and the divider gate should be considered instead.

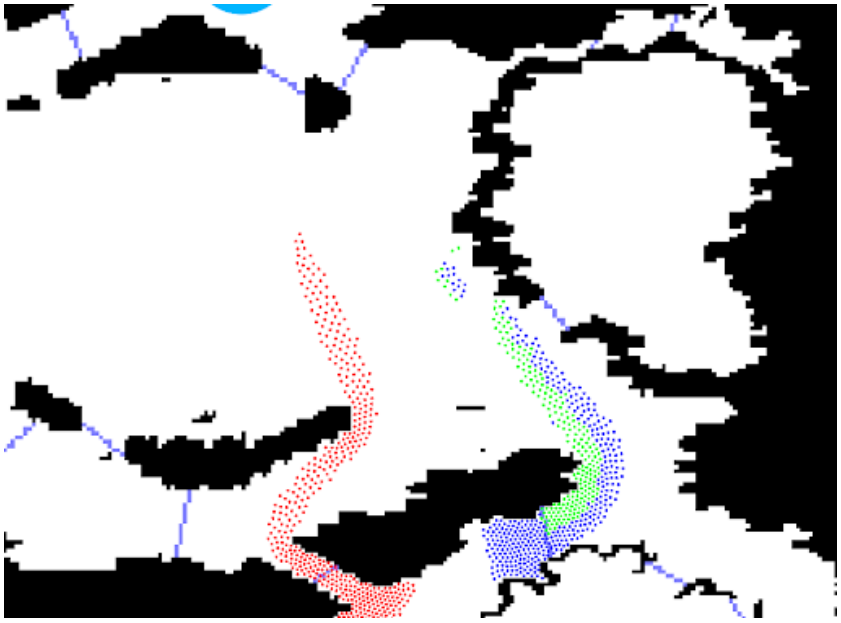


Figure 6.6: Divider Gate - Wrong Stream Assignment

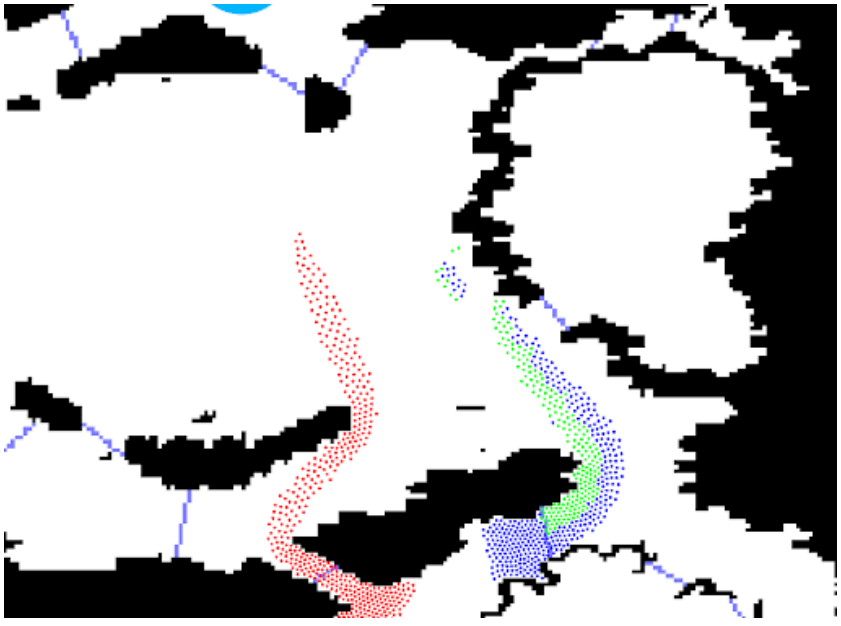


Figure 6.7: Divider Gate - Wrong Stream Assignment

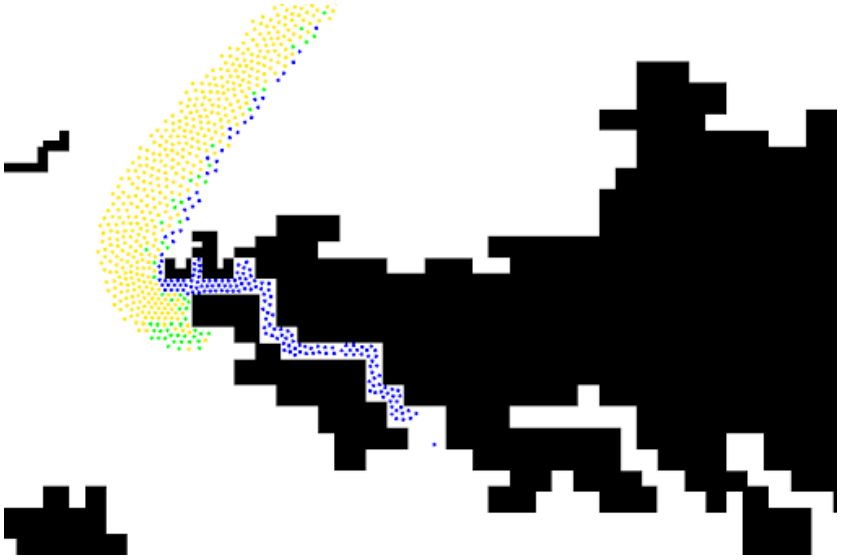


Figure 6.8: Exit Blocked By A Different Flow Stream

Wrong Creation Of Gates

The gates at the region the green units pass through weren't created at the right places. They weren't placed in the narrowest section, which led to a wrong estimation of the flow. At the location the red arrow is pointing to, a gate wasn't created at all (Figure 6.7).

The problem typically occurs in places with thin or small obstacles, which may be ignored by the decomposition.

Possible Solution - The decomposition can be modified to be more sensitive to small obstacles. This will, however, result in a creation of a large number of gates, often created because of a small irregularity in the shape of the walls. We have to essentially weigh between specificity and sensitivity. In our case, we got some false negatives. Alternatively, the decomposition can be created manually by a designer.

Units Blocking An Exit

The blue units are trying to leave a narrow passage, but the exit is blocked by the yellow and green units (Figure 6.8). This gets most of the units stuck, delaying the time of their arrival at the target and making the estimates of our algorithm wrong.

Possible Solution - Remains an open problem.

Nearby Entrances

In Figure 6.9, we can see that the red units were pushed off their path by the green units, and now they are blocking each other's path. This leads to significant delays, chaos, weird movements, and isolated units.

The entrances to the passages used by these two unit groups were close to each other, which led to them getting in each other's way.

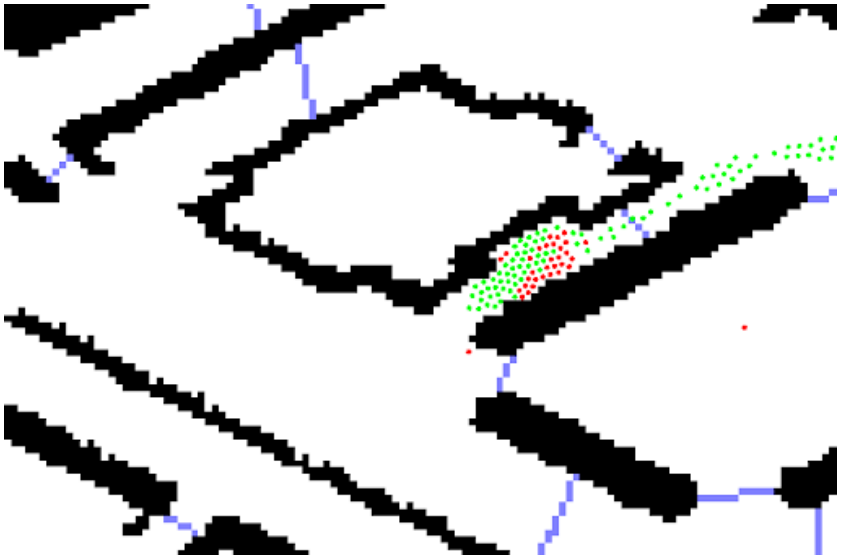


Figure 6.9: Units Pushed Off Their Path

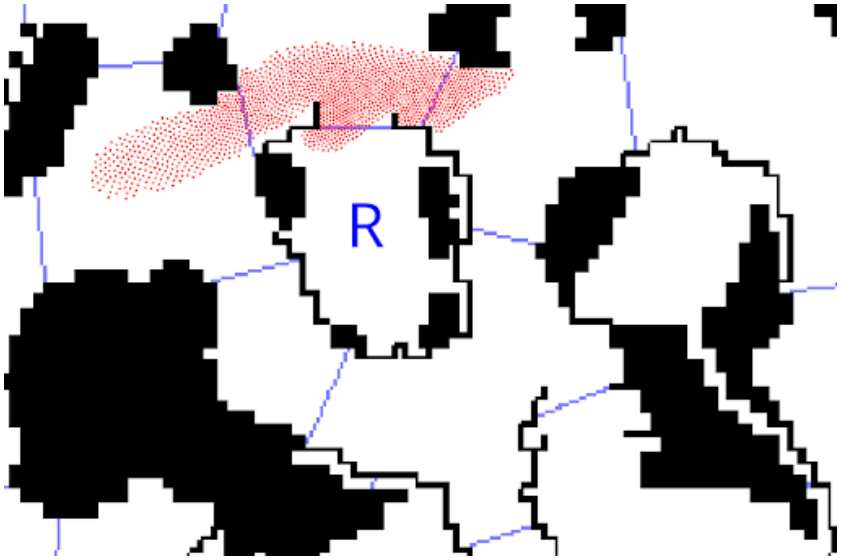


Figure 6.10: Unnecessary Repaths

The problem of the units getting caught up in a different stream has been improved compared to the original method, but it still occurs.

Possible Solution - We couldn't find a good solution. The problem occurs when the incoming flow is significantly greater than the gate capacity. The units then accumulate at the entrance and may block the path to another group of units (which then pushes some of them off the path).

Unnecessary Repaths

Part of the units is pushed to region R (Figure 6.10), which doesn't lie on their regional path. A repath is triggered for every unit that enters the region R. This slows the algorithm down significantly (hundreds of paths may have to be calculated in a short period of time).

Possible Solution - When units enter a region that doesn't lie on any of the regional paths, the algorithm should check if it neighbors any of the regions on its path. If this is the case, a direction should be assigned to the region leading towards the region the units were pushed from.

7. Future Work

The Regional Flow Graph method runs into the problems mentioned in the Results chapter, which may negatively impact its performance. These should be fixed if possible. The results chapter already suggested some solutions When we are selecting the best solution, we need to be able to estimate the finishing time and be able to assign the units in the best way possible. To do this, we need to accurately estimate the flow going through the gates. Our method uses the basic flow estimate. A better estimate could, however, be created based on the flow data. RFG is optimized for the finishing and pathfinding times, but this may result in a subpar performance in other areas. A good algorithm should be also optimized for other objectives and will be often made to fit a particular game. We will discuss possible modifications for modified or additional objectives.

7.1 Better Flow Estimation

In our method, we estimate the capacity c for each gate, which depends linearly on the gate's width. At any point in time, we can have n units trying to enter the gate. The flow going through the gate will be $Min(c, n)$. In the "Problem Analysis" section, we discussed that estimating the flow may be more complicated. We would like to find out the quality of our estimate and inspect other factors influencing the flow values.

We will observe the flow's behavior on a few simple maps. This is done because we would like to limit the factors that could influence the flow. We also won't be able to obtain correct measurements of some metrics on the real game maps. This is because we use an assumption that $\Phi^{into}_{G_1 \Rightarrow G_2}(t) = \Phi G_1 \Rightarrow {G_2}^{toward}(t - t_{G_1 \Rightarrow G_2})$ where the $t_{G_1 \Rightarrow G_2}$ is the time of unobstructed movement from G_1 to G_2. The assumption is, however, broken even by the smallest pathfinding problems (like wall hugging).

7.1.1 Overflow

Overflow measures the number of units accumulated at the gate's entrance. The accumulated units could be pushing the units in front of them into the gate. This could decrease the spaces between the units and thus increase the flow.

To investigate the influence of the overflow, we used the "Decreasing Gate Size Map". This map contains four gates, each with a lower width than the previous one. These decreases allow us to simulate an overflow increasing at a roughly linear rate.

We will measure the flow and overflow values at the third gate. This gate has a width of 11 and an estimated capacity of 13.3 units/s. If the estimate used by our method is correct, we should see a constant flow independent of the overflow values. The measured values can be seen in Image 7.1.

We see that the overflow builds up steadily, then reaches a peak and decreases. The peak corresponds to the time when there are no incoming units from the previous gate.

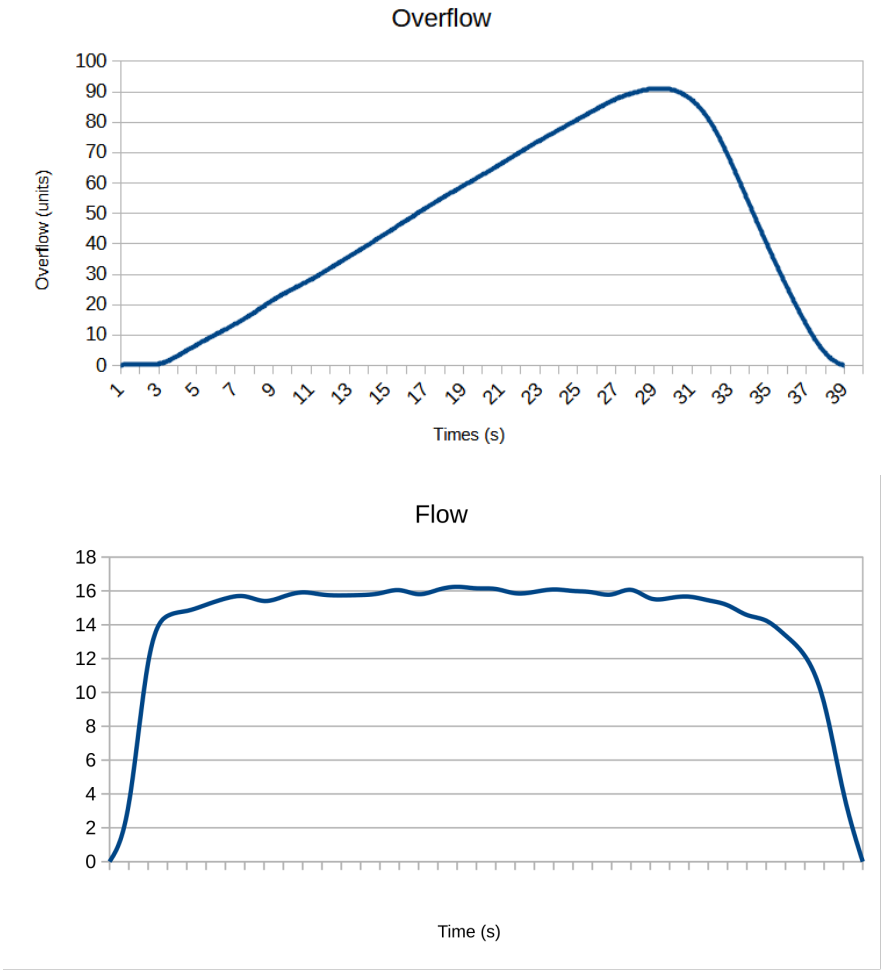


Figure 7.1: Flow And Overflow Relation

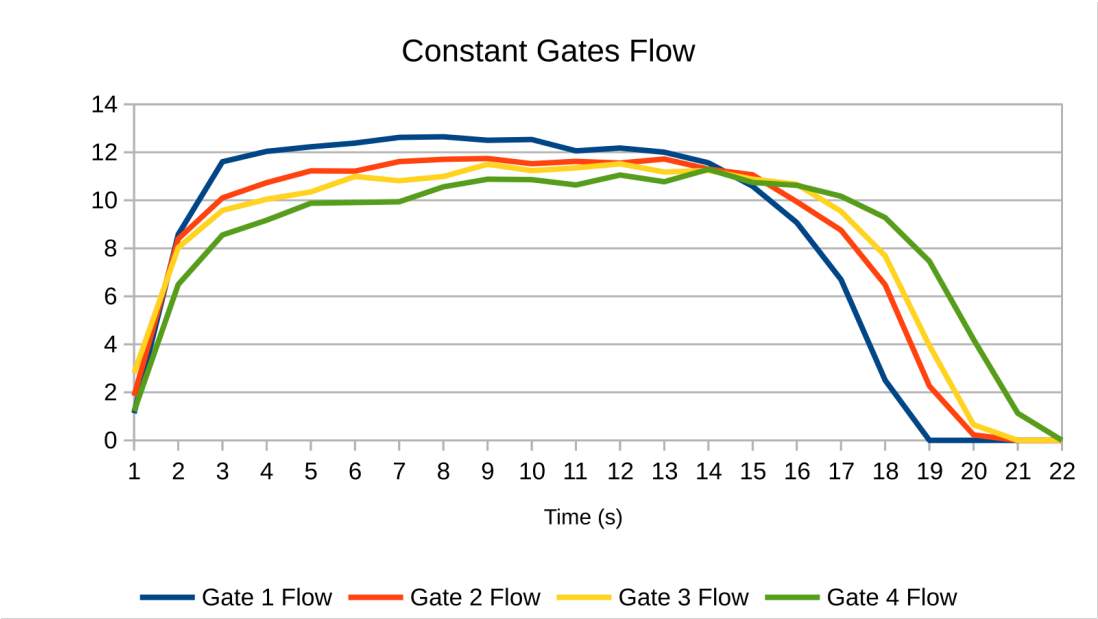


Figure 7.2: Influence Of Previous Gates

The flow quickly builds without any overflow. The first overflow occurs at the 4 seconds where the flow reaches 14.5 units/s. The overflow seems to increase the flow, but each increase of overflow offers diminishing returns. The flow eventually plateaus at 16 units/s. The flow falls off when the overflow starts decreasing. The decrease from a reduced overflow seems to be more significant during the falloff.

Our capacity estimate seems lower than it should be, and the real flow exceeds it by about 20%. The rectangular shape, the predicted flow would have, seems to match the real flow reasonably well but ignores the transitions at the beginning and the end.

7.1.2 Previous Gate Effects

The flow values going through a gate may be influenced by the gates the units used before. We will investigate this effect on the "Constant Gates" map. The map consists of four gates of the same size placed right after each other. The gates have a width of 9 and an estimated capacity of 10.9 units/s. Our estimate would assume the same flow constant flows through all of the gates.

The flows measured at each of the gates can be seen in Image 7.2. The previous gates really seem to influence the flow. Units traverse the first gate roughly 3s faster than the fourth gate. The flow builds up slower at the later gates and reaches lower peak values. The differences are caused by different incoming flows, which then also influence the overflows at the individual gates.

Our estimate doesn't deal with these situations too well. Especially if we have short flows with longer transitions.

7.1.3 Flow Function

We would like to estimate the flow function (Definition 18) so we could get better estimates of the flow going through the gates and the finishing times of the

solution.

We obtained combinations of four metrics - flow, incoming flow, overflow, and gate width. The values of the metrics were gathered only on a few selected maps, where we could guarantee them being reasonably accurate. We then tried to find the function that would predict the outgoing flow based on the other 3 metrics using curve fitting. The flow function was then estimated as:

$$\Phi_G^{out}(t) = \left(G_l - \sqrt{G_l}\right)\left(a + b * S\left(\frac{O_G(t)}{G_l}\right)\right)$$

Where S is the sigmoid function, and parameters $a = 1.35, b = 0.7$

The function is specific to our unit model and couldn't be used for other models. The estimate is based only on a few maps and probably isn't too accurate.

We can reasonably assume that the flow doesn't depend on the gate's width linearly and that the overflow increases the flow but with diminishing returns. And investigating the characteristics of the flow further could be worthwhile.

7.2 Modified Objectives

We identified two main modified requirements that may be useful for different games. One wants to improve the finishing times at the cost of the pathfinding time. The other one wants to increase the cohesion of the unit groups.

7.2.1 Possible Missing Solutions

The algorithm doesn't generate all the sets of concurrent paths. The algorithm generates these sets in a way that is likely to produce the optimal or nearly optimal set. However, this isn't always the case, and the best set of paths can be missing. Finding the flow by adding augmenting paths is only a heuristic. Another solution could examine a higher number of concurrent paths to get to more optimal solutions. However, this would probably require a longer pathfinding phase.

7.2.2 Increased Cohesion

Our algorithm optimizes for a quick finishing time. To achieve it, we often divide the units into small groups, often numbering in single digits. This doesn't look natural. Moreover, small groups of units are vulnerable to possible enemy attacks, which would make our method unusable in real RTS games. The algorithm could be improved by specifying a minimum number of units assigned to a flow stream or a maximum number of streams the solution can use.

Conclusion

The goal of the thesis was to create a pathfinding algorithm that could be effectively used in RTS games. We proposed a pathfinding method called the Regional Flow Graph. We implemented the method in the Unity game engine and evaluated its performance against existing pathfinding algorithms in our custom testing environment.

Our method achieved good results in terms of the metrics we deemed the most important. We achieved a significant improvement compared to existing methods in terms of the pathfinding speed and the time the units spent navigating to their target. These improvements were, however, paid for by a longer preprocessing time. This would make it a good fit for games where the environment doesn't change frequently but a worse one for highly dynamic changes.

Despite achieving quicker finishing times, our method still has several problems which wouldn't make it a good fit for a real-life application. The units often behave in strange ways. Their paths get into conflicts which could break the player's immersion. The algorithm also doesn't consider the potential enemy presence on the map and often separates the units into small groups. This would make the units vulnerable during their navigation.

Overall, our method shows some potential for use in RTS games but still requires modifications and improvements to be truly viable.

Bibliography

[1] Jan Pacovský. Navigation of units in video games using flow networks. Master's thesis, Charles University, Faculty of Mathematics and Physics, Prague, 2019.

[2] Marjan van den Akker, Roland Geraerts, Han Hoogeveen, and Corien Prins. Pathfinding challenges with large groups, Jan 2011.

[3] Yngvi Björnsson, Markus Enzenberger, Robert Holte, Jonathan Schaeffer, and Peter Yap. Comparison of different grid abstractions for pathfinding on maps. *IJCAI*, 2003.

[4] Greg Snook. Simplified 3D movement and pathfinding using navigation meshes. *Game programming gems*, 1(1):288–304, 2000.

[5] Paul Tozour. Fixing pathfinding once and for all. *Retrieved September*, 10:2010, 2008.

[6] Paul Tozour and I S Austin. Building a near-optimal navigation mesh. *AI game programming wisdom*, 1:298–304, 2002.

[7] Florian Richoux, Alberto Uriarte, and Santiago Ontanón. Walling in strategy games via constraint optimization. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 10, pages 52–58, 2014.

[8] Caio Freitas de Oliveira and Charles Andrye Galvao Madeira. Creating efficient walls using potential fields in real-time strategy games. In *2015 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 138–145, 2015.

[9] Kári Halldórsson and Yngvi Björnsson. Automated decomposition of game maps. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 11, pages 122–127, 2015.

[10] Wittaya Bidakaew, Pavadee Sompagdee, Sukanya Ratanotayanon, and Pongsagon Vichitvejpaisal. RTS terrain analysis: An axial-based approach for improving chokepoint detection method. In *2016 8th International Conference on Knowledge and Smart Technology (KST)*, pages 228–233, 2016.

[11] Luke Perkins. Terrain analysis in real-time strategy games: An integrated approach to choke point detection and region decomposition. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 6, pages 168–173, 2010.

[12] Alberto Uriarte and Santiago Ontanón. Improving terrain analysis and applications to rts game ai. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 12, pages 15–20, 2016.

[13] Florian Richoux. Terrain analysis in StarCraft 1 and 2 as combinatorial optimization. In *2022 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8, 2022.

[14] Zhou Yijun, Xi Jiadong, and Luo Chen. A fast bi-directional a* algorithm based on quad-tree decomposition and hierarchical map. *IEEE Access*, 9:102877–102885, 2021.

[15] Tansel Uras, Sven Koenig, and Carlos Hernández. Subgoal graphs for optimal pathfinding in eight-neighbor grids. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 23, pages 224–232, 2013.

[16] Adi Botea. Ultra-fast optimal pathfinding without runtime search. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 7, pages 122–127, 2011.

[17] Edsger W Dijkstra. A note on two problems in connexion with graphs. In *Edsger Wybe Dijkstra: His Life, Work, and Legacy*, pages 287–290. 2022.

[18] 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.

[19] Andrew V Goldberg and Chris Harrelson. Computing the shortest path: A search meets graph theory. In *SODA*, volume 5, pages 156–165, 2005.

[20] Alex Nash and Sven Koenig. Theta* for any-angle pathfinding. *Game AI Pro*, 2:161–171, 2015.

[21] Daniel Harabor and Alban Grastien. An optimal any-angle pathfinding algorithm. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 23, pages 308–311, 2013.

[22] Xiang Xu and Kun Zou. Smooth path algorithm based on a* in games. In *Advances in Computer Science, Environment, Ecoinformatics, and Education: International Conference, CSEE 2011, Wuhan, China, August 21-22, 2011. Proceedings, Part I*, pages 15–21, 2011.

[23] Xiao Cui and Hao Shi. An overview of pathfinding in navigation mesh. *International Journal of Computer Science and Network Security*, 12(12):48–51, December 2012.

[24] Douglas Demyen and Michael Buro. Efficient triangulation-based pathfinding. In *Aaai*, volume 6, pages 942–947, 2006.

[25] Michael Cui, Daniel Damir Harabor, Alban Grastien, and Canberra Data61. Compromise-free pathfinding on a navigation mesh. In *IJCAI*, pages 496–502, 2017.

[26] Hao Pan. Pathfinding and map feature learning in RTS games with partial observability. 2021.

[27] Serhat Bayili and Faruk Polat. Limited-Damage a*: A path search algorithm that considers damage as a feasibility criterion. *Knowledge-Based Systems*, 24(4):501–512, 2011.

[28] Richard E Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artif. Intell.*, 27(1):97–109, 1985.

[29] Daniel Harabor and Alban Grastien. Online graph pruning for pathfinding on grid maps. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 25, pages 1114–1119, 2011.

[30] Steve Rabin and Nathan Sturtevant. Combining bounding boxes and JPS to prune grid pathfinding. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016.

[31] Daniel Harabor and Alban Grastien. Improving jump point search. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 24, pages 128–135, 2014.

[32] Adi Botea, Martin Müller, and Jonathan Schaeffer. Near optimal hierarchical path-finding. *J. Game Dev.*, 1(1):1–30, 2004.

[33] Yan Li, Lan-Ming Su, and Wen-Liang Li. Hierarchical path-finding based on decision tree. In *Rough Sets and Knowledge Technology: 7th International Conference, RSKT 2012, Chengdu, China, August 17-20, 2012. Proceedings 7*, pages 248–256, 2012.

[34] Factorio - new pathfinding algorithm.

[35] Johan Hagelbäck. Potential-field based navigation in starcraft. In *2012 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 388–393, 2012.

[36] Johan Hagelbäck. Hybrid pathfinding in StarCraft. *IEEE Trans. Comput. Intell. AI Games*, 8(4):319–324, 2015.

[37] Elijah Emerson. Crowd pathfinding and steering using flow field tiles. In *Game AI Pro 360: Guide to Movement and Pathfinding*, pages 67–76. CRC Press, 2019.

[38] David Silver. Cooperative pathfinding. In *Proceedings of the aaai conference on artificial intelligence and interactive digital entertainment*, volume 1, pages 117–122, 2005.

[39] Esra Erdem, Doga Kisa, Umut Oztok, and Peter Schüller. A general formal framework for pathfinding problems with multiple agents. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 27, 2013.

[40] Alborz Geramifard, Pirooz Chubak, and Vadim Bulitko. Biased cost pathfinding. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 2, pages 112–114, 2006.

[41] J M Akker, Roland Geraerts, Han Hoogeveen, and Corien Prins. Path planning for groups using column generation. In *International Conference on Motion in Games*, pages 94–105. Springer, 2010.

[42] Craig W Reynolds and Others. Steering behaviors for autonomous characters. In *Game developers conference*, volume 1999, pages 763–782, 1999.

[43] Ben Sunshine-Hill. RVO and ORCA: How they really work. In *Game AI Pro 360: Guide to Movement and Pathfinding*, pages 245–256. CRC Press, 2019.

[44] 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, 2009.

[45] Austin Grossman and Matt Pritchard. Ensemble studios': Age of empires II: The age of kings. In *Postmortems from game developer*, pages 115–126. Focal Press, 2013.

List of Figures

List of Tables

A. Attachments

A.1 Digital Attachment

The electronic attachment contains the Unity project implementing the simulator and different pathfinding methods. It also contains the results of the experiments.

A.1.1 Program

- Unity project with all the source codes and the pathfinding problems (maps, unit positions...)
- User manual with instructions for a simulation setup
- Auto-generated documentation

A.1.2 Pathfinding Data

- Movement maps and stuck maps for all the problems and methods (a single map for all the runs with averaged values)
- Performance tables with information about the finishing, pathfinding, and preparation times. Also contains a table with the data about the repaths.
- The offsets of the real gate arrival compared to the predicted ones
- Data used to estimate the flow function
- Flows - graphs of the flow coming to the target for different methods - comparison between the predicted values and the real ones
- Runs - data about the results of the individual runs
- Target Arrival Times - table comparing the real arrival times at the target and predicted ones.

A.2 Performance Tables

ID	FF	FG	A*	RFG
1	1	1.02	1.02	1.05
2	1.04	1.06	1.06	1
3	1.14	1.12	1.13	1
4	1	1.22	1.5	1.23
5	1	1.09	1.09	1.03
6	1.06	1.27	1.15	1
7	1	1.03	1.03	1.01
8	1	1.3	1.06	1.15
9	1.22	1.69	1.25	1
10	1.1	1.4	1.14	1
11	1.24	1.15	1.26	1
12	1	1.21	1.26	1.44
13	1	1.15	1.15	1.26
14	1	1.16	1.15	1.23
15	1.07	1.1	1.2	1
16	1	1.78	1.16	1.3
17	1.13	1	1.21	1
18	1	1.08	1.12	1
19	1	1.13	1.15	1.05
20	1.14	1.35	1.21	1
21	1.1	1	1.2	1.01
22	1.01	1.02	1.02	1
23	1.01	1.02	1.02	1
24	1.01	1.02	1.04	1
25	1	1.01	1.01	1
26	1.01	1.01	1.01	1
27	1	1.5	1.18	1.06
28	1.07	1.13	1.29	1
29	1	1.46	1.68	1.17
30	1	1.97	1.44	1.17
31	1	1.01	1.01	1
32	1	1.01	1.01	1
33	1	1.01	1.03	1
34	1	1	1	1.01
35	1	1	1	1.01
36	1.28	1.27	1.41	1
37	1.11	1.23	1.12	1
38	1.12	1.13	1.45	1
39	1.08	1.15	1.14	1
40	1	1.7	1.09	1.14
41	1	1.33	1.04	1.06
42	1.03	1.01	1	1.02
43	1.01	1.01	1	1
44	1.01	1	1	1
45	1.17	1.77	1.21	1
46	1.06	1.86	1.08	1
47	1.07	1.23	1.18	1
48	1	1.92	1.06	1
49	1.19	1.11	1.2	1
50	1	1.01	1.02	1.02
51	1	1.07	1	1.02
52	1.11	1.12	1.1	1
53	1.13	1.13	1.13	1
54	1	1.11	1.21	1.06
55	1.06	1	1.09	1.01
56	1	1.02	1.06	1.07
57	1	1.17	1.21	1
58	1.09	1.04	1.18	1
59	1.06	1.02	1.3	1
60	1	1.08	1.04	1.08
61	1	1.03	1.1	1.06
62	1.08	1	1.12	1.01
63	1.1	1.36	1.1	1
64	1	1.02	1.03	1.01
65	1	1.03	1.02	1.01
66	1.01	1.02	1.04	1
67	1.01	1.01	1.01	1
68	1.15	1.05	1.14	1
69	1.39	1.33	1.4	1
70	1.59	1.51	1.66	1
71	1.78	F	1.99	1
72	1.24	1	1.25	1.01
73	1.01	1.01	1.02	1
74	1	1.35	1.28	1
75	1.16	1.17	1.16	1
76	1.42	1.06	1.44	1
77	1.28	1	1.33	1.07
78	1.24	1	1.4	1
79	1.01	1	1.02	1
80	1.01	1	1.02	1.01
81	1	1.01	1.01	1.02
82	1	1	1	1
83	1	1	1.01	1
84	1.19	1.31	1.33	1
85	1.16	1.14	1.24	1
86	1.03	1	1.04	1.01
87	1.05	1.01	1.38	1
88	1.04	1.04	1.09	1
89	1	1	1	1.01
90	1.04	1.07	1.04	1
91	1.27	1.3	1.29	1
92	1.02	1	1	1.02
93	1.01	1.04	1	1.01
94	1.1	1.3	1.17	1
95	1.94	1.03	2.09	1
96	1.24	1.08	1.26	1
97	1	1	1.06	1.01
98	1	1.09	1.23	1.03
99	1	1.17	1.23	1
100	1.14	2.08	1.25	1
101	1.05	1.14	1.09	1
102	1.21	1	1.4	1.02
103	1.11	1.07	1.47	1
104	1.04	1.2	1.07	1
105	1	1.76	1.65	1.13
106	1.12	1.01	1.15	1
107	1.09	1.12	1.23	1
108	1	1.01	1.04	1.01
109	1.03	1.07	1.04	1
110	1.19	1	1.23	1
111	1.03	1.09	1.03	1
112	1.3	1.01	1.42	1
113	1.23	1	1.27	1
114	1.17	1	1.19	1
115	1	1.06	1.25	1.06
116	1	1.01	1.02	1
117	1.21	1.16	1.62	1
118	1.6	1.02	1.51	1
119	1.16	1	1.64	1.01
120	1.37	1	1.6	1.42
121	1.07	1.03	1.37	1
122	1.15	1	1.17	1.01
123	1.19	1	1.23	1.01
124	1.3	1.22	1.55	1
125	1.02	1	1.1	1.03
126	1	1.02	1.02	1.02
127	1.11	1.05	1.2	1
128	1	1.14	1.05	1
129	1.03	1.06	1.05	1
130	1	1.03	1.05	1.05
131	1.14	1.37	1.36	1
132	2.15	1.09	2.34	1
133	1	1.08	1.02	1.01
134	1.32	1.16	1.44	1
135	1.96	1.02	1.97	1
136	1	1.82	2.66	1.18
137	1.26	1.26	1.34	1
138	1	1.03	1.49	1.04
139	1	1.08	1.41	1.01
140	1.01	1.1	1.2	1
141	1.24	1.02	1.3	1
142	1.11	1	1.32	1.01
143	1.54	1.32	1.55	1
144	1.48	1.01	1.57	1
145	1.14	1.01	1.3	1
146	2.05	1.03	1.01	1
147	1.49	1	1.41	1.02
148	1.08	1.13	1.21	1
149	1.12	1.2	1.21	1
150	1	1.4	1.25	1
151	1.69	1.03	2.59	1
152	1.94	1.16	2.02	1
153	1	1.28	1.02	1.08
154	1.12	1	1.45	1
155	1	1.12	1	1.15
156	1	1.01	1.02	1.01
157	1	1.17	1.03	1
158	2.01	1.06	2.03	1
159	1.05	1.03	1.14	1
160	1.13	1.1	1.2	1
161	1.13	1	1.14	1.01
162	1.08	1.33	1.17	1
163	1	1.09	1.02	1.02
164	1.05	1.17	1.56	1
165	1.06	1.09	1.21	1
166	1	1.19	1.04	1.1
167	1	1.23	1.88	1.23
168	1.12	1.11	1.76	1
169	1.01	1	1.04	1
170	1	1.21	1	1.12
171	1	1.24	1.07	1.01
172	1.03	1.08	1.03	1
173	1	1.22	1.01	1.01
174	1.06	1.13	1.08	1
175	1.08	1.12	1.1	1
176	1.09	1.04	1.1	1
177	1.84	1	1.83	1.02
178	1	1.16	1.18	1.11
178	1	1.04	1.07	1.04
180	1	1.17	1.09	1.05
181	1	1.2	1.03	1
182	1	1.15	1.13	1.18
183	1.02	1.02	1.06	1
184	1.07	1.3	1.05	1
185	1	1.14	1.07	1.14
186	1.01	1.01	1	1.05
187	1	1.79	1.13	1.51
188	1.23	1.09	1.33	1
189	1	1.63	1.26	1.04
190	1	1.24	1.21	1.12
191	1	1.04	1.05	1.03
192	1.01	2.01	1.04	1
193	1.06	1.12	1.11	1
194	1.07	1	1.35	1.02
195	1.37	1.33	2.17	1
196	1.09	1.06	1.19	1
197	1.43	1.11	1.63	1
198	1.09	1.15	1.12	1
199	1	1.26	1.4	1.01
200	1.85	1.1	2.34	1

Pathfinding methods : Flow Field, Flow Graph, A*, Regional Flow Graph
Map sources : blue = Custom, white = Starcraft, green = Warcraft 3

Table A.1: Relative finish times

ID	FF	FG	A*	RFG
1	1	1.02	1.01	1.05
2	1.04	1.05	1.05	1
3	1.12	1.09	1.1	1
4	1	1.08	1.47	1.04
5	1	1.05	1.09	1.03
6	1.07	1.07	1.17	1
7	1.02	1	1.06	1.02
8	1.03	1.21	1.1	1
9	1.35	1.64	1.39	1
10	1.12	1.23	1.16	1
11	1.36	1.05	1.39	1
12	1.07	1.16	1.35	1
13	1.02	1.05	1.16	1
14	1	1.1	1.09	1.06
15	1.07	1	1.2	1
16	1	1.57	1.16	1.26
17	1.17	1	1.26	1
18	1.05	1	1.17	1.02
19	1	1.01	1.17	1
20	1.14	1.01	1.22	1
21	1.09	1	1.2	1
22	1	1.02	1.02	1
23	1.01	1.02	1.02	1
24	1.01	1.02	1.04	1
25	1	1.01	1.01	1
26	1.01	1.01	1.01	1
27	1	1.36	1.18	1.05
28	1.08	1.03	1.3	1
29	1	1.26	1.5	1.08
30	1	1.92	1.46	1.17
31	1	1.01	1.01	1
32	1	1.01	1.01	1
33	1	1.01	1.03	1
34	1	1	1	1
35	1	1	1	1
36	1.26	1.08	1.4	1
37	1.12	1.04	1.14	1
38	1.11	1.03	1.45	1
39	1.08	1.03	1.14	1
40	1	1.59	1.1	1.06
41	1	1.05	1.05	1
42	1.02	1.01	1	1.02
43	1.01	1	1	1.01
44	1.01	1.01	1	1
45	1.28	1.8	1.33	1
46	1.19	1.9	1.22	1
47	1.09	1.08	1.2	1
48	1.1	2	1.16	1
49	1.19	1.04	1.21	1
50	1	1.01	1.02	1.01
51	1	1.01	1	1.02
52	1.1	1.03	1.09	1
53	1.12	1.07	1.13	1
54	1.03	1	1.23	1.04
55	1.05	1	1.09	1.01
56	1.01	1	1.07	1
57	1	1.02	1.22	1.01
58	1.08	1.05	1.19	1
59	1.09	1.03	1.34	1
60	1	1.1	1.04	1.09
61	1	1.02	1.08	1.04
62	1.09	1	1.15	1.01
63	1.13	1.02	1.13	1
64	1	1.02	1.02	1.01
65	1	1.02	1.01	1
66	1.01	1.02	1.04	1
67	1.01	1	1	1
68	1.14	1.03	1.13	1
69	1.35	1.1	1.37	1
70	1.57	1.32	1.63	1
71	1.72	F	1.95	1
72	1.38	1	1.38	1.01
73	1.01	1	1.02	1
74	1	1.14	1.31	1.03
75	1.22	1.07	1.23	1
76	1.45	1.09	1.47	1
77	1.29	1	1.33	1.04
78	1.23	1.01	1.4	1
79	1.01	1	1.02	1
80	1	1	1.02	1.01
81	1	1.01	1.01	1.01
82	1	1	1	1
83	1	1	1.02	1
84	1.18	1	1.34	1
85	1.17	1	1.26	1
86	1.02	1	1.03	1
87	1.02	1	1.37	1.01
88	1.2	1	1.27	1.01
89	1	1	1	1.01
90	1.03	1.07	1.04	1
91	1.26	1.29	1.29	1
92	1.01	1	1	1.02
93	1.01	1.04	1	1.01
94	1.11	1	1.19	1
95	1.91	1.01	2.06	1
96	1.24	1	1.27	1
97	1.07	1.01	1.13	1
98	1.08	1	1.31	1.09
99	1.01	1.07	1.23	1
100	1.13	1.85	1.21	1
101	1.05	1.12	1.09	1
102	1.18	1	1.38	1.01
103	1.21	1.07	1.5	1
104	1.05	1.04	1.05	1
105	1	1.62	1.68	1.04
106	1.15	1	1.17	1.01
107	1.11	1.02	1.25	1
108	1	1.01	1.04	1
109	1.05	1.01	1.06	1
110	1.2	1	1.24	1.01
111	1.03	1.08	1.04	1
112	1.28	1	1.4	1
113	1.25	1	1.28	1
114	1.17	1	1.2	1
115	1.01	1.01	1.26	1
116	1	1.01	1.03	1
117	1.21	1.04	1.65	1
118	1.65	1.01	1.56	1
119	1.17	1	1.64	1.01
120	1.42	1	1.66	1.38
121	1.05	1	1.35	1
122	1.17	1	1.18	1.01
123	1.18	1	1.22	1
124	1.43	1.23	1.66	1
125	1.03	1	1.11	1.03
126	1	1.02	1.02	1.02
127	1.17	1	1.24	1.01
128	1	1.04	1.04	1
129	1.03	1.02	1.05	1
130	1	1.03	1.05	1.04
131	1.12	1.05	1.2	1
132	2.21	1.1	2.45	1
133	1	1.01	1.02	1.01
134	1.34	1.08	1.47	1
135	2.14	1.04	2.14	1
136	1	1.68	2.59	1.12
137	1.26	1.01	1.33	1
138	1	1.03	1.52	1.03
139	1	1	1.41	1.01
140	1.04	1.03	1.22	1
141	1.24	1.01	1.29	1
142	1.29	1.01	1.56	1
143	1.51	1.05	1.52	1
144	1.46	1	1.54	1
145	1.14	1.01	1.32	1
146	2.02	1.07	1.18	1
147	1.42	1	1.35	1.01
148	1.16	1.06	1.27	1
149	1.18	1.05	1.27	1
150	1.07	1.03	1.32	1
151	1.6	1	2.56	1
152	2.02	1.08	2.11	1
153	1.03	1	1.04	1.09
154	1.14	1	1.51	1
155	1.03	1.07	1.02	1
156	1	1.01	1.02	1.01
157	1	1.05	1.03	1
158	2.01	1.03	2.03	1
159	1.06	1.04	1.16	1
160	1.14	1.02	1.2	1
161	1.2	1.04	1.21	1
162	1.09	1.02	1.18	1
163	1.04	1.01	1.05	1
164	1	1.03	1.51	1
165	1.07	1.02	1.21	1
166	1.03	1.12	1.06	1
167	1	1.03	1.83	1.14
168	1.07	1	1.68	1
169	1.03	1	1.06	1.02
170	1.03	1	1.03	1.06
171	1	1.03	1.06	1.01
172	1.04	1.01	1.03	1
173	1.01	1.01	1.01	1
174	1.07	1	1.09	1
175	1.1	1.02	1.11	1
176	1.08	1.05	1.09	1
177	1.88	1.01	1.89	1
178	1	1.05	1.07	1.05
178	1	1.01	1.15	1.01
180	1	1.04	1.09	1.04
181	1.02	1.03	1.04	1
182	1.02	1	1.16	1.08
183	1.13	1	1.17	1.1
184	1.07	1.06	1.06	1
185	1	1.04	1.06	1.03
186	1	1.01	1.01	1.04
187	1	1.4	1.15	1
188	1.32	1.03	1.43	1
189	1	1.2	1.27	1.07
190	1.09	1.18	1.33	1
191	1	1.03	1.04	1.01
192	1.08	1.81	1.1	1
193	1.07	1.03	1.12	1
194	1.04	1.01	1.35	1
195	1.31	1.17	2.19	1
196	1.09	1.01	1.18	1
197	1.58	1.12	1.83	1
198	1.12	1	1.16	1
199	1	1.08	1.32	1.02
200	1.82	1.07	2.3	1

Pathfinding methods : Flow Field, Flow Graph, A*, Regional Flow Graph
Map sources : blue = Custom, white = Starcraft, green = Warcraft 3

Table A.2: Relative 90% finish times

ID	FG	A*	RFG	RFGS
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0
4	0.8	0	0	19.7
5	5.1	0	0	0
6	4.8	0	74.2	0
7	0	0	16.8	0
8	4.1	0	0	154
9	96	0.2	0	38.2
10	2.5	0	0	3.4
11	30.2	65.2	0	31.6
12	13.8	0	0	112.2
13	7.3	0	0	17.1
14	1.3	1.6	0	3.5
15	1.9	17.6	1.1	0.8
16	19.5	1.6	0	88.5
17	3.8	0	0	2.2
18	2.7	1.4	0	23.4
19	7.8	1.7	0	6.5
20	1.7	1	0	30.4
21	0	8.9	0	0
22	0	0	0	0
23	0	0	0	0
24	0	0	0	0
25	0	0	0	0
26	0	1	0	0
27	108.1	1.7	0	1802.1
28	8.5	2.8	1.8	59.6
29	100.3	19.3	206.2	1241.3
30	388.9	83.3	0	432.2
31	0	0	0	0
32	0	0	0	0
33	0	0	0	0
34	0	0	0	0
35	0	0	0	0
36	14.7	0	0	2.1
37	1.9	0	0	1.1
38	6.3	1	0	0
39	1.7	0	0	0
40	18.4	0	0	46
41	6	0	0	0.9
42	0	0	0	0
43	0.4	0.9	0	0
44	2.7	10.2	0	0
45	399.5	0	0	7.2
46	211.2	0	0	2.6
47	17	0.1	0	0.8
48	616.5	5.9	0.9	499.7
49	3.2	0	0	0
50	0	0	0	0
51	0.6	0	0	0
52	4.4	11.4	0	0
53	7.6	0.9	0	0
54	0	0	0	0
55	0	0	0	0
56	0	0	0	0
57	1	8.1	0	1.4
58	0	0	0	0
59	0	1.1	0	0
60	0	0	0	0
61	0	0	0	0
62	0	0.4	31.3	0
63	0.2	19.1	0	0
64	0	0	0	0
65	0.8	3.7	0	0
66	3.2	11.3	0	0
67	0	0	0	0
68	0.2	0	0	0
69	9.1	0	0	0
70	23	0	0	0
71	F	0.1	0	0.1
72	2.5	0	0	2.3
73	0	0	0	0
74	3	0	0	0.1
75	0.5	1.4	2.7	0
76	0.4	0.2	0	0
77	0	0	0	0
78	0	0.6	0	0.1
79	0	0	0	0
80	0	0	0	0
81	0	0	0	0
82	0	0	0	0
83	0.5	1	0	0
84	0	0.2	0	0
85	0	0.3	0	0
86	0	0	0	0.7
87	0	0	0	0
88	0	0.1	0	29
89	0	0	0	0
90	0	0	0	1.8
91	0	0	0	1
92	0	0	0	0
93	0	0.5	0	1.4
94	0.8	0	0	2.2
95	0.1	0	0.2	0
96	0	0	0	0
97	0.1	0.3	26.7	15.8
98	11.2	1.2	402.7	266
99	14.2	9.4	41.8	46.7
100	26.3	14.7	278.9	89.7
101	12.8	12.3	0	0
102	6.5	12.1	0	8.4
103	9	10.7	23.8	1.1
104	8.9	6.7	0	0
105	26.6	24	0	61.2
106	6	7.9	0	0
107	6.8	2.2	0	0
108	0.2	1.6	0	0
109	4.1	22.3	0	5.7
110	24.1	33.9	7.1	0
111	9.4	11.8	0	1.6
112	15.1	15	0	0
113	6.9	2.2	0	1.6
114	0.3	0.1	0	1.2
115	5.3	4	0	9.7
116	6.7	2.2	0.3	0
117	2.7	10.6	0	76.1
118	12.1	10.4	1.1	0
119	52.3	39.4	0	56.3
120	0.5	0	0	129.1
121	13	31.2	0	0
122	30.4	30	0	0
123	1	1	94.3	0
124	4.9	49.1	0	16.2
125	3.7	0.1	0	205.5
126	1.7	3.1	0	0
127	13.9	26.7	88.3	82.6
128	22.4	30.4	0	0
129	32	0	0	0
130	0	0.1	0	0
131	2.6	2.1	0	0.8
132	12.7	9.8	0	4.3
133	0	0.1	0	0
134	34.3	7.9	0	5.8
135	28.1	7.8	0.1	230.8
136	30	62.6	39.8	106.6
137	9.3	0.6	0	0
138	0.3	0.9	0	1.1
139	4.2	34.1	0	0.1
140	0.6	12.8	0	0.4
141	0.8	4.4	0	5.6
142	7.1	24.2	0	183.4
143	11.7	13.1	0	57.1
144	0	3.5	0	0.3
145	2.7	0	0.1	0
146	0.7	0.1	0	0.2
147	0	0	0	0
148	3	2.7	36.2	3.1
149	11.8	3	0	6.6
150	37.2	3	1.2	214.2
151	1.2	5	0	0
152	1.1	8.1	6.3	21
153	30.9	15	0	193.1
154	0.1	2.7	1.2	0
155	5.5	10.3	0	55.6
156	0.7	0.2	0	0
157	36.4	1	0	22.2
158	5.5	3.2	17.9	0
159	1.6	3.9	0	0.6
160	5.3	9.2	0	0
161	16.3	19.2	0	92.8
162	4	0.2	0	1.6
163	1	0	0	2.5
164	1	8.9	0	0
165	0.4	32.7	0	0
166	2.2	0	0	0
167	44.3	29.4	83.5	213.7
168	3.6	20.3	8.1	0.7
169	1.5	17.6	0	0.5
170	2.8	2.1	231.1	412.1
171	6.4	0	0	0
172	16.9	1.8	62.9	0
173	2.5	4.6	143.5	0
174	0.6	8.9	56.1	0
175	2.4	2.3	0	0.1
176	0.6	0.4	0	0
177	1.6	2.2	0	0.2
178	2.6	15.1	0	0
178	0	11.5	0	0
180	7.3	0.3	0	0
181	14.1	0	25.7	190.2
182	9.3	2.3	0	105.2
183	1.3	1.4	0	0
184	5	0.3	0	0
185	25.8	0.6	0	215.5
186	27.5	24.8	0	0
187	39.1	0.1	0	0.2
188	9.9	7.4	0	12.8
189	25.2	9.9	0	122.7
190	13.4	97.2	0	14.5
191	0	0	0	0
192	66.4	0.3	0	0.4
193	0.9	3.3	0	45
194	0	0.6	0	0
195	85.2	26.8	0	0
196	5.2	0	0	0
197	66.8	9	0	74
198	2	0	0	3.5
199	16.2	6.2	0	82.8
200	4.4	124	0	0

Flow Graph Repaths, A* Repaths, Regional Flow Graph Repaths, Regional Flow Graph Soft Repaths

Table A.3: Repaths with different methods