A Complete Subgoal Graph Method for Ultra-Fast Pathfinding

Eric Hall

A thesis submitted for the degree of
Flexible Double Degree in the Bachelor of Advanced Computing (Honours) and the Bachelor of Mathematical Sciences
The Australian National University

November 2021

Except where otherwise indicated, this thesis is my own original work.

Eric Hall
11 November 2021

To everyone
For being human
No more, and no less.

Acknowledgments

I would like to thank my supervisor, Alban Grastien, for meeting with me every week to give me advice and help me decide in what direction to take my project, and for giving me links to lots of useful papers.

I would also like to thank Daniel Harabor, for providing me with the source code for some relevant algorithms, as well as some useful papers.

I would also like to thank everyone at my university, the ANU, for supporting my learning. The layout of this thesis was based on a template provided by the ANU.

Abstract

I introduce a new method for ultra-fast pathfinding on octile grids, called the Complete Subgoal Graph algorithm (CSG). This algorithm has much faster preprocessing times than similar algorithms, and also slightly faster online runtimes. The biggest drawback is that it requires a relatively large amount of memory, although this memory requirement is comparable to that used by certain other algorithms, such as Topping. I propose some ideas for future improvement in terms of reducing this memory requirement.

CSG is specifically designed for pathfinding on octile grids, although extensions to other graphs may be possible in the future. Octile grids are essentially a discretisation of 2-D Euclidean space, and are commonly used as an approximation to 2-D Euclidean space, because they are more convenient to work with. In addition, CSG is mainly focused on applications to the kinds of maps found in video games, as this is a common application for pathfinding algorithms.

The main pieces of work that CSG builds upon are Subgoal Graph algorithms and Compressed Path Databases (CPDs). CSG can essentially be viewed as building a CPD on top of a subgoal graph, in order to remove search from the subgoal graph.

In order to ensure that the online runtime of CSG is competitive with existing algorithms, I needed to make certain improvements. One of the improvements involves taking into account the fact that passing through a subgoal will only be optimal if you are entering from certain directions and leaving in certain directions. I also borrow the idea of bounding boxes and apply them to my algorithm.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Statement

The aim of my work is to improve the performance of ultra-fast pathfinding algorithms on octile grids.

1.2 Definitions for Pathfinding on Octile Grids

Pathfinding is the problem of finding a path from one point to another on some graph. Each edge on the graph has a positive weight, representing the distance between the two nodes it connects. Then given a start node and a goal node, one needs to find a sequence of edges connecting the start node to the goal node. We typically aim to minimize the sum of the distances of the edges in this path.

An **octile grid** (also known as an **8-connected grid**) is a grid of squares, where each square is connected to the horizontally, vertically, and diagonally adjacent squares, and the distance between connected squares is equal to the distance that would be between those squares in Euclidean space. That is, horizontally and vertically adjacent squares have a distance of 1 from each other, whereas diagonally adjacent squares have a distance of $\sqrt{2}$ from each other.

There will typically be **obstacles** on the octile grid, which prevent the agent from passing through certain squares. If there were no obstacles on the octile grid, pathfinding on octile grids would be trivial.

Octile grids arise naturally in that they are a discretisation of Euclidean space. They can be used to approximate and simplify problems that take place in Euclidean space. They are used because it is often much simpler to deal with problems in a discrete space than it is to deal with problems in a continuous space.

Corner cutting is where the agent scrapes alongside an obstacle while moving diagonally between two locations (see Figure 1.1(a)). While this may be acceptable according to the basic definition of an octile grid, it seems very strange for the agent to be able to squeeze between two obstacles that are diagonally adjacent to each other, as illustrated in Figure 1.1(b). In order for the agent to be able to do this, it would essentially need to take up zero volume. For this reason, it is often the case that corner cutting is disallowed in octile grid pathfinding, which adds a small amount

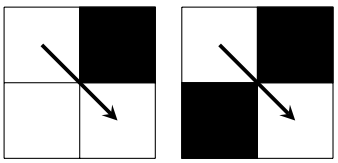


Figure 1.1: Examples of corner cutting

of extra complexity to the problem.

1.3 Online and Offline Computation

A pathfinding algorithm is often split into an offline preprocessing component and an online pathfinding component. Prior to the actual pathfinding, the offline preprocessing component takes the map data and performs some computation on it to output some preprocessing data. Then the online pathfinding component takes the preprocessing data and the map data, and uses this to answer a number of pathfinding queries.

This separation of computation may be useful, because the offline preprocessing needs to be done only once per map, and it can be done well in advance of actually needing to answer any pathfinding queries. For example, in a game, the game designer may be able to perform the preprocessing themselves on their own machine, and then distribute the preprocessing data to their users along with the game. Then the users' machines will not have to perform any preprocessing prior to pathfinding, because they have already been provided with the preprocessing data.

This means that in many applications, it may not be important to have a small preprocessing time, so long as the pathfinding queries themselves are fast.

1.4 Tradeoffs and Ultra-fast Pathfinding

There are a number of different dimensions along which we can compare pathfinding algorithms. The most important dimensions are:

1. Offline Preprocessing Time
2. Online Runtime
3. Space taken by Preprocessing Data

When designing a pathfinding algorithm, we have to make trade-offs between these dimensions. For example, by increasing the amount of preprocessing time, it may be possible to decrease the runtime, but then if we were to want to decrease the preprocessing time, we might have to increase the runtime.

Ultra-fast pathfinding is a term that so far doesn't have a formal definition, but essentially means that the pathfinding algorithm has online pathfinding capabilities with an online runtime very close to the fastest online runtime of any known algorithm, regardless of the amount of offline preprocessing time or memory usage required for the algorithm.

1.5 Motivation

In video games, there is often a need to have a large number of agents that need to move from one location in the map to another. In order to be able to cope with a large number of agents moving from location to location, it is necessary for each of these pathfinding queries to be quick.

Chapter 2

Background and Related Work

Pathfinding problems arise fairly naturally, and octile grids are a fairly natural space to perform pathfinding on. As a result, a lot of research has been done on pathfinding, and a significant amount of it has been done specifically with a focus on octile grids. In this chapter, I explore some of this research.

I will first explain how to find optimal paths when there aren't any obstacles in the grid (this is trivial but of fundamental importance). We will then explore a number of techniques that have been applied to pathfinding on octile grids. These include:

1. Jump Point Search, a symmetry-reducing method that is specifically designed for octile grids.
2. Jump Point Search Plus, an extension to Jump Point Search that precomputes the distances between jump points.
3. Compressed Path Databases, an algorithm that calculates the optimal paths between every pair of points and stores a compressed representation of these paths.
4. Contraction Hierarchies, an algorithm that creates an importance hierarchy upon the nodes so that it is only necessary to search for paths which consist of a sequence of nodes of increasing importance followed by a sequence of nodes of decreasing importance.
5. Subgoal Graphs, an algorithm that identifies important nodes in the graph and searches for a path through these important nodes.
6. Topping, a combination of Compressed Path Databases and Jump Point Search Plus.
7. JPS+BB, an algorithm that associates each move with a box containing all points for which that move is optimal, which can be used to prune moves from the search.
8. JPS+BB+, Topping+, and TOPS, improvements to JPS+BB and Topping in which preprocessing data is only computed for jump points, rather than all points.

9. Brigitte, an algorithm that groups nearby points into regions and focuses on finding bridges between regions, rather than on finding paths between individual points.

2.1 Freespace and Octile Paths

Freespace is an octile grid where all the obstacles have been removed or can be ignored [Harabor et al., 2019]. The **octile distance** between two points is the shortest distance it would take to travel between those points in freespace. An **octile path** between two points is an optimal path in freespace.

It is very easy to find all possible octile paths. This is regularly taken for granted in the literature. I think that due to its fundamental importance, it's worth covering how to find octile paths in more detail than is usually provided. I therefore formulated and proved the following theorem:

Theorem 1. *Suppose we are given a source with coordinates* (x_s, y_s) *and a goal with coordinates* (x_g, y_g)*. Let* $\Delta_x = x_g - x_s$ *and let* $\Delta_y = y_g - y_s$*. Let* d_1 *be the direction* $(\text{sgn}(\Delta_x), \text{sgn}(\Delta_y))$*. If we have* $|\Delta_x| > |\Delta_y|$ *then let* d_2 *be the direction* $(\text{sgn}(\Delta_x), 0)$*, otherwise let* d_2 *be the direction* $(0, \text{sgn}(\Delta_y))$*. Then the octile paths between the source and the goal are precisely those paths that contain exactly* $\min(|\Delta_x|, |\Delta_y|)$ *moves in direction* d_1*, and exactly* $\max(|\Delta_x|, |\Delta_y|) - \min(|\Delta_x|, |\Delta_y|)$ *moves in direction* d_2*. This is illustrated in Figure 2.1.*

Proof. Let n_d denote the number of moves in the direction d contained in a path between the source and the goal. Then the path arrives at the goal if and only if:

$$\begin{cases} x_g &= x_s + n_{(1,1)} + n_{(1,0)} + n_{(1,-1)} - n_{(-1,1)} - n_{(-1,0)} - n_{(-1,-1)} \\ y_g &= y_s + n_{(1,1)} + n_{(0,1)} + n_{(-1,1)} - n_{(1,-1)} - n_{(0,-1)} - n_{(-1,-1)} \end{cases}$$

We can rearrange and use the definitions of Δ_x and Δ_y to get:

$$\begin{cases} \Delta_x &= n_{(1,1)} + n_{(1,0)} + n_{(1,-1)} - n_{(-1,1)} - n_{(-1,0)} - n_{(-1,-1)} \\ \Delta_y &= n_{(1,1)} + n_{(0,1)} + n_{(-1,1)} - n_{(1,-1)} - n_{(0,-1)} - n_{(-1,-1)} \end{cases}$$

Instead of using absolute directions, lets reinterpret this in terms of how the various directions impact Δ_x and Δ_y.

$$\begin{cases} |\Delta_x| &= n_{(\text{sgn}(\Delta_x),\text{sgn}(\Delta_y))} + n_{(\text{sgn}(\Delta_x),0)} + n_{(\text{sgn}(\Delta_x),-\text{sgn}(\Delta_y))} \\ &\quad\; - n_{(-\text{sgn}(\Delta_x),\text{sgn}(\Delta_y))} - n_{(-\text{sgn}(\Delta_x),0)} - n_{(-\text{sgn}(\Delta_x),-\text{sgn}(\Delta_y))} \\ |\Delta_y| &= n_{(\text{sgn}(\Delta_x),\text{sgn}(\Delta_y))} + n_{(0,\text{sgn}(\Delta_y))} + n_{(-\text{sgn}(\Delta_x),\text{sgn}(\Delta_y))} \\ &\quad\; - n_{(\text{sgn}(\Delta_x),-\text{sgn}(\Delta_y))} - n_{(0,-\text{sgn}(\Delta_y))} - n_{(-\text{sgn}(\Delta_x),-\text{sgn}(\Delta_y))} \end{cases}$$

We can cancel opposite moves with each other, reducing the length of the path. Therefore, no path containing opposite moves can be octile. For each pair of opposite directions, lets choose one of them to be the primary direction, and if the path has

n moves in the other direction, let's interpret this as being $-n$ moves in the primary direction. Under this interpretation, we find that the path reaches the goal if and only if:

$$\begin{cases} |\Delta_x| &= n_{(\text{sgn}(\Delta_x),\text{sgn}(\Delta_y))} + n_{(\text{sgn}(\Delta_x),0)} + n_{(\text{sgn}(\Delta_x),-\text{sgn}(\Delta_y))} \\ |\Delta_y| &= n_{(\text{sgn}(\Delta_x),\text{sgn}(\Delta_y))} + n_{(0,\text{sgn}(\Delta_y))} - n_{(\text{sgn}(\Delta_x),-\text{sgn}(\Delta_y))} \end{cases}$$

Suppose $n_{(\text{sgn}(\Delta_x),-\text{sgn}(\Delta_y))} > 0$. Then the second equation tells us that either we have $n_{(\text{sgn}(\Delta_x),\text{sgn}(\Delta_y))} > 0$ or $n_{(0,\text{sgn}(\Delta_y))} > 0$. In the first case, we can remove a move in the direction $(\text{sgn}(\Delta_x), -\text{sgn}(\Delta_y))$ and remove a move in the direction $(\text{sgn}(\Delta_x), \text{sgn}(\Delta_y))$, and we can add two moves in the direction $(\text{sgn}(\Delta_x), 0)$, and these equations will remain satisfied. We have removed two diagonal moves and added two straight moves, so we have reduced the length of the path while still arriving at the goal, so the path cannot have been octile. In the second case, we can remove a move in the direction $(\text{sgn}(\Delta_x), -\text{sgn}(\Delta_y))$ and a move in the direction $(0, \text{sgn}(\Delta_y))$, and we can add a move in the direction $(\text{sgn}(\Delta_x), 0)$. We have removed a diagonal move and a straight move and added a straight move while keeping the equations satisfied, so we have reduced the length of the path while still arriving at the goal, so the path cannot have been octile. Therefore, in any octile path, we cannot have $n_{(\text{sgn}(\Delta_x),-\text{sgn}(\Delta_y))} > 0$.

We can use symmetric logic to prove that we cannot have $n_{(\text{sgn}(\Delta_x),-\text{sgn}(\Delta_y))} < 0$, since in this case we can essentially treat the first equation as if it were the second equation and the second equation as if it were the first equation. Therefore $n_{(\text{sgn}(\Delta_x),-\text{sgn}(\Delta_y))} = 0$ in any octile path between the points. So our equations become:

$$\begin{cases} |\Delta_x| &= n_{(\text{sgn}(\Delta_x),\text{sgn}(\Delta_y))} + n_{(\text{sgn}(\Delta_x),0)} \\ |\Delta_y| &= n_{(\text{sgn}(\Delta_x),\text{sgn}(\Delta_y))} + n_{(0,\text{sgn}(\Delta_y))} \end{cases}$$

Suppose $|n_{(\text{sgn}(\Delta_x),0)}| > 0$ and $|n_{(0,\text{sgn}(\Delta_y))}| > 0$. Then we can reduce the absolute value of each of these numbers by 1, and we can keep the equations satisfied by changing the absolute value of $n_{(\text{sgn}(\Delta_x),\text{sgn}(\Delta_y))}$ or $n_{(\text{sgn}(\Delta_x),-\text{sgn}(\Delta_y))}$ by 1. This will remove two straight moves and add only one diagonal move, reducing the length of the path while still arriving at the goal. Then the path cannot have been octile. Therefore, in any octile path, either $n_{(\text{sgn}(\Delta_x),0)} = 0$ or $n_{(0,\text{sgn}(\Delta_y))} = 0$. Then we have two possible systems of equations:

$$\begin{cases} |\Delta_x| &= n_{(\text{sgn}(\Delta_x),\text{sgn}(\Delta_y))} + n_{(\text{sgn}(\Delta_x),0)} \\ |\Delta_y| &= n_{(\text{sgn}(\Delta_x),\text{sgn}(\Delta_y))} \end{cases}$$

$$\begin{cases} |\Delta_x| &= n_{(\text{sgn}(\Delta_x),\text{sgn}(\Delta_y))} \\ |\Delta_y| &= n_{(\text{sgn}(\Delta_x),\text{sgn}(\Delta_y))} + n_{(0,\text{sgn}(\Delta_y))} \end{cases}$$

These are equivalent to:

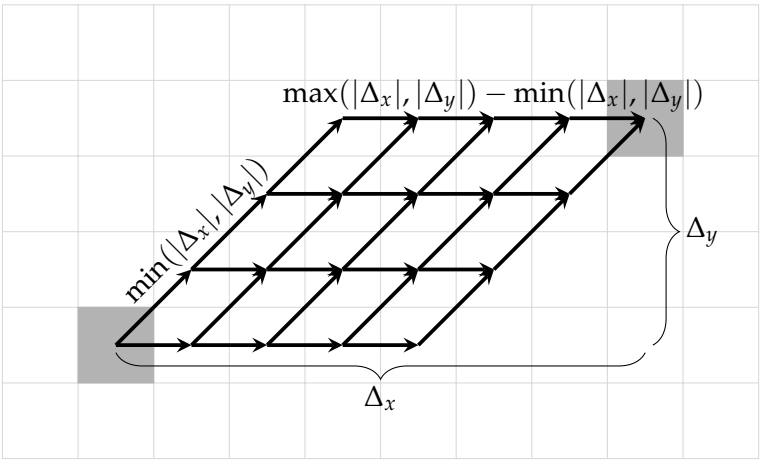


Figure 2.1: All optimal paths in freespace for an example pair of points

$$\begin{cases} n_{(\text{sgn}(\Delta_x),0)} & = |\Delta_x| - |\Delta_y| \\ n_{(\text{sgn}(\Delta_x),\text{sgn}(\Delta_y))} & = |\Delta_y| \end{cases}$$

$$\begin{cases} n_{(\text{sgn}(\Delta_x),\text{sgn}(\Delta_y))} & = |\Delta_x| \\ n_{(0,\text{sgn}(\Delta_y))} & = |\Delta_y| - |\Delta_x| \end{cases}$$

The total length of the first path is $|\Delta_x| + (\sqrt{2}-1)|\Delta_y|$ and the total length of the second path is $|\Delta_y| + (\sqrt{2}-1)|\Delta_x|$. If $|\Delta_x| > |\Delta_y|$, then the second path is shorter, if $|\Delta_y| > |\Delta_x|$, the first path is shorter, otherwise they have the same length. Note that in the case where they have the same length, the paths have the same number of moves in each direction.

We have shown that if $|\Delta_x| > |\Delta_y|$, then the octile paths are the paths with $|\Delta_x|$ moves in the $(\text{sgn}(\Delta_x), \text{sgn}(\Delta - y))$ direction and $|\Delta_y| - |\Delta_x|$ moves in the $(0, \text{sgn}(\Delta_y))$ direction, and that otherwise, the octile paths are the paths with $|\Delta_y|$ moves in the $(\text{sgn}(\Delta_x, \text{sgn}\,\Delta_y))$ direction and $|\Delta_x| - |\Delta_y|$ moves in the $(\text{sgn}(\Delta_x), 0)$ direction.

This is equivalent to the statement that the octile paths are those paths that contain $\min(|\Delta_x|, |\Delta_y|)$ moves in the direction d_1 and $\max(|\Delta_x|, |\Delta_y|) - \min(|\Delta_x|, |\Delta_y|)$ moves in the direction d_2.

So we have proven what we wanted to prove.

□

2.2 Jump Point Search

Jump Point Search (**JPS**) is a technique used to exploit the structure of octile grids, introduced by Harabor and Grastien [2011]. One property of octile grids is that the location you end up in after performing a straight move followed by a diagonal move is the same as the location you end up in after performing the diagonal move

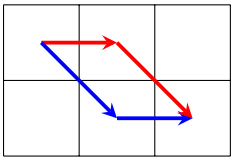


Figure 2.2: Symmetry exploited by jump point search

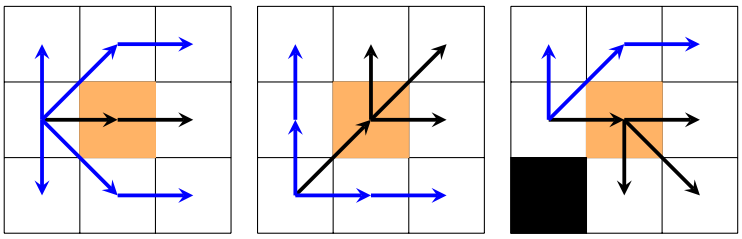


Figure 2.3: Nodes expanded by jump point search

followed by the straight move (see Figure 2.2). The idea of JPS is to exploit this symmetry by always expanding diagonal moves before straight moves, so that the same node isn't considered for expansion multiple times.

Then whenever a node is expanded, only certain neighbours are considered depending on what direction the node was encountered in and what obstacles surround the node. In figure 2.3, the black arrows indicate nodes that are diagonally-first reachable via the orange node, whereas the blue arrows indicate nodes for which the diagonal-first path does not pass through the orange node.

When expanding a node in a straight direction, if there are no obstacles adjacent to that node, the only neighbour JPS would need to consider is the next node in that straight direction, because that's the only node for which it is optimal to travel in a diagonal-first manner from the previous node through the current node.

However, when JPS is expanding a node in a straight direction and has just passed an obstacle, it may need to consider more neighbours, because the diagonal-first path to these neighbours would ordinarily have cut across the obstacle, but this is no longer possible. Points where the search is going straight but then needs to change direction are known as **straight jump points**. These points are the only points where travelling straight gives you a branching factor of more than 1.

When expanding nodes in a diagonal direction, JPS generally needs to consider two nodes in the adjacent straight directions and one node continuing in the diagonal direction. It starts a scan in both of the adjacent straight directions. If each of these scans stops without encountering a straight jump point, this node effectively now has a branching factor of 1 and it can continue in the diagonal direction. Otherwise, this node is a **diagonal jump point**, and the straight jump point(s) encountered by the scans are the successors of this jump point.

So the straight jump points and the diagonal jump points are essentially the places where JPS is forced to branch.

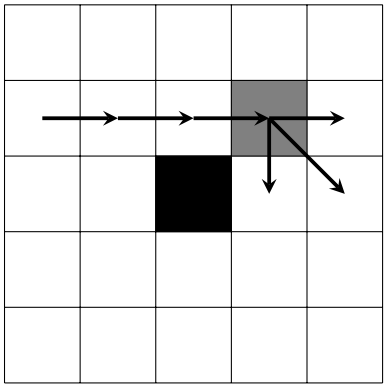


Figure 2.4: A straight jump point

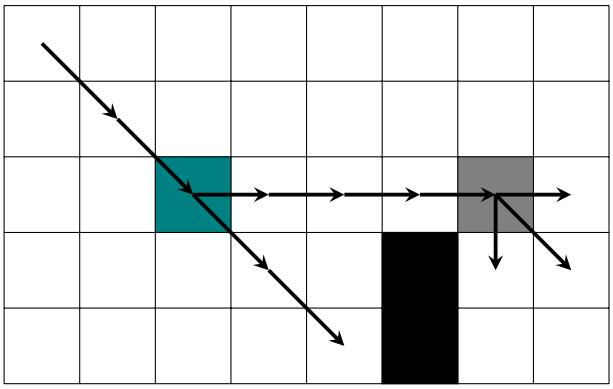


Figure 2.5: A diagonal jump point

Note that the original version of Jump Point Search was designed for when corner cutting is allowed, but it has also been modified to be applicable when corner cutting is not allowed, as was briefly mentioned in Harabor and Grastien [2014]. The version I described is the one where corner cutting is not allowed.

2.2.1 JPS+

Jump Point Search Plus (**JPS+**) is an improvement to JPS where for every point in the grid, the distance to the next jump point in each direction is preprocessed and stored [Harabor and Grastien, 2014]. This means that when finding a path, it is possible to jump directly to the next jump point instead of having to scan in the usual way. This provides a substantial speedup.

This is a relatively light-weight improvement, in the sense that it takes only a small amount of preprocessing and memory. The amount of storage required is only linear in the total number of nodes.

2.3 Compressed Path Databases

Compressed Path Databases (**CPD**s) were first introduced by Botea [2011] and are one of the foundational methods for ultra-fast pathfinding. They are versatile and can be applied to arbitrary graphs. During pre-computation, an instance of the Dijkstra algorithm is run from each point, to compute the distance to each other point. This is used to generate a table which contains the first move on the optimal path from any point to any other point. Then optimal paths can be quickly computed by repeatedly looking up and applying the first move from the first move table.

If there are a large number of points, the first move table may be massive, in which case it may not be possible to apply the method in practice. To remedy this, the table is compressed using a method such as Run Length Encoding [Strasser et al., 2015]. This relies on the fact that optimal paths to nearby points tend to be similar. By ordering the points in such a way that points nearby in the space are nearby in the ordering, runs of the same first move are created in the first move table. Such a run can be compressed to require only two pieces of information: the first move, and the length of the run. For example, the string "nnnnnwwwssswsss" consists of 5 runs, and can be compressed to "n5w3s3w1s3".

It's not necessarily obvious what orderings of the points allow for good compression using this method. Strasser et al. [2015] found that ordering the nodes based on a depth-first preorder was a good heuristic method for ordering the points.

Compression can be improved by exploiting the fact that an optimal path between two points can be computed in either direction: from point A to point B or from point B to point A. This means the first move table only needs to store the first move in one direction, and we can ignore the first move in the other direction [Salvetti et al., 2017]. To determine which of these moves to store, an ordering of nodes is chosen. A first move between two nodes is only stored if the first node is later in this ordering than the second node. It will never be necessary to look up the first move in the opposite direction, so the corresponding first move symbol can be replaced by a wildcard, which can be treated as if it were any symbol. These wildcards can be included in runs of other symbols, which reduces the number of runs and improves compression. One downside of this change is that if you do want to find only a partial path instead of a full path, it may take a much longer time to find it, since you may need to look up several moves in the opposite direction before looking up a move in the forwards direction.

2.4 Contraction Hierarchies

Contraction Hierarchies (**CH**s) are a pathfinding method initially introduced for the purpose of pathfinding on road networks [Geisberger et al., 2008], although they can be applied to all kinds of graphs. They identify a hierarchy of nodes, ordered by importance, such that it is always possible to find an optimal path that passes through a sequence of nodes of increasing importance, followed by a sequence of nodes of decreasing importance. We call such a path an **arching pach** [Uras and

Koenig, 2018]. CHs speed up pathfinding because they mean it is only necessary to search the graph for arching paths, not arbitrary paths.

In Geisberger et al. [2008], these hierarchies are created by contracting nodes one by one. When a node is contracted, it is removed from the graph, and new edges are created between the adjacent nodes to represent travelling between those nodes via the removed node. This ensures that it is not necessary to travel through the removed node to find an optimal path between any of the remaining nodes, so the removed node can be considered "less important" than the remaining nodes, and it can be given a lower place in the contraction hierarchy. Nodes continue being contracted until there is only one node left in the graph. Then the graph with all the added edges satisfies the property that for any two nodes there is always an arching path that is optimal.

Finding the optimal ordering in which to contract the nodes is not an easy problem, but there are heuristics to find a reasonably good ordering of nodes. For example, one good heuristic is to choose to contract the node that minimizes the **edge distance**, defined to be the number of edges that would have to be added when contracting the node, minus the number of edges that are connected to the node [Geisberger et al., 2008].

2.5 Subgoal Graphs

Subgoal Graphs (**SG**s) were first introduced by Uras et al. [2013]. The fundamental idea behind SGs is to reduce the problem of searching for a path over the entire grid to the problem of searching for a path over a subset of relatively important nodes in the grid, known as subgoals. Since there are usually far fewer subgoals than nodes in the grid, this makes it much easier to find a path.

SGs rely on a notion of reachability, and a notion of subgoals. These can be chosen to be many different things depending on the application, but they should satisfy two important properties:

1. Reachability Property: If a node is reachable from another node, it should be easy to compute the optimal path between them.

2. Subgoal Property: If a node is not reachable from another node, there should be an optimal path between them that passes through a subgoal which is not either of the two nodes.

Note that these properties are my own interpretation of some of the work in Uras and Koenig [2017]. In particular, the first property corresponds to the "Refine" step mentioned in that work, and the second property corresponds to the idea of a "Shortest-Path-Cover" mentioned in that work.

If two nodes are reachable from each other, by the Reachability Property, it is easy to compute the optimal path between them. Otherwise, the repeated application of the Subgoal Property shows that it is possible to find a sequence of subgoals between

them, where each subgoal is reachable from the previous subgoal, and by passing through each subgoal in sequence it is possible to optimally find a path from the start to the goal. Then we can use the Reachability Property to find each of these optimal subpaths, and we can concatenate the resulting subpaths to return our optimal path.

The only step missing in the above process is the step where we identify the sequence of subgoals through which we can optimally reach the goal. So SGs have reduced the problem of search over an octile grid to the problem of search over a set of subgoals. This problem is easier to solve quickly, since there are fewer subgoals than non-obstacles.

It may be natural to consider notions of reachability that are symmetric, but this is not necessary. Even in the case of assymetric reachability, as long as it satisfies the properties listed above, it will be possible to reduce the problem of searching through the whole graph to searching through subgoals in the way mentioned above.

The process by which subgoal graphs are used to find paths can be viewed as consisting of the 3 steps "Connect", "Search", and "Refine" [Uras and Koenig, 2017]:

1. Connect the start and goal to the subgoal graph
2. Search the subgoal graph for an optimal path between the start and the goal.
3. Refine the path in the subgoal graph into a path in the original graph.

In precomputation, before these steps, the subgoals are computed, and an edge is added between a pair of subgoals if and only if the second subgoal in the pair is reachable from the first. The distance associated with the edge is set to the optimal distance it would take to travel from one subgoal to the other.

In Step 1, an edge needs to be added to the subgoal graph between the start and each subgoal which is reachable from the start, including the goal if the goal is reachable from the start. Also, an edge needs to be added to the subgoal graph between each subgoal from which the goal is reachable and the goal.

In Step 2, some method is used to search the subgoal graph for an optimal path between the start and the goal. For example, A^* could be used.

In Step 3, we convert the optimal path found in the subgoal graph to an optimal path in the original graph. We do this by exploiting the Reachablity Property. This gives us an optimal sub-path corresponding to each edge, and we can concatenate together these sub-paths to get an optimal path from the start to the goal.

2.5.1 Safe-Reachability and Convex Corners

In the original subgoal graph algorithm by Uras et al. [2013], the notion of reachability used was safe-reachability, and the set of subgoals used was the set of convex corners.

Two points are **safe-reachable** from each other if for every octile path between those points in freespace, the input graph has no obstacles on that path [Uras et al., 2013].

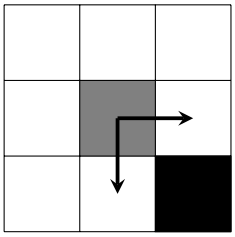


Figure 2.6: Illustration of a convex corner.

A square is a **convex corner** if it does not contain an obstacle, there is an obstacle in a diagonal direction, and there is no obstacle in the adjacent cardinal directions to that diagonal direction [Uras and Koenig, 2018]. This is illustrated in Figure 2.6.

We want to be sure that the notions of safe-reachability and convex corners together satisfy the Reachability Property and the Subgoal Property.

It's clear that if two points are safe-reachable from each other, it is easy to find a path from one to the other. Just choose an arbitrary octile path between the points.

It's not so obvious that if two points are not safe-reachable from each other, there is a convex corner on an optimal path between them. I was not fully convinced of this fact by the proof of Theorem 1 in Uras et al. [2013]. They write "Thus, every shortest path between s and s' needs to circumnavigate at least one obstacle and therefore contains some subgoal s''". While this does make intuitive sense, I'm not entirely sure this qualifies as a proof. Therefore, I provide my own proof of this fact.

In my proof, I use the definition of a diagonal-first path provided in Harabor and Grastien [2011]. That is, a path is diagonal-first if it would not be possible to replace any straight-to-diagonal turning point by a diagonal-to-straight turning point. Note that a turning point is a point where the previous move was performed in a different direction of travel to the next move.

Theorem 2. *If we have two points a and b in the same connected component such that b is not safe-reachable from a, then there is a convex corner on an optimal path from a to b, which is not at a or b.*

Proof. Since a and b are in the same connected component, there is some path between a and b. There are only a finite number of paths starting from a of length less than this path, therefore there must be some path of minimal length between a and b. Therefore some optimal path between a and b exists.

Consider this optimal path. Use Algorithm 3 from Harabor and Grastien [2011] to transform this path into a diagonal-first path with the same length. That is, replace straight-to-diagonal turning points by diagonal-to-straight turning points until this is no longer possible.

Figure 2.7 illustrates all the possible types of turning points. Any other turning point is equivalent to one of these turning points under rotations and flipping. Note that if the second direction is in the same direction as the first direction, it doesn't qualify as a turning point.

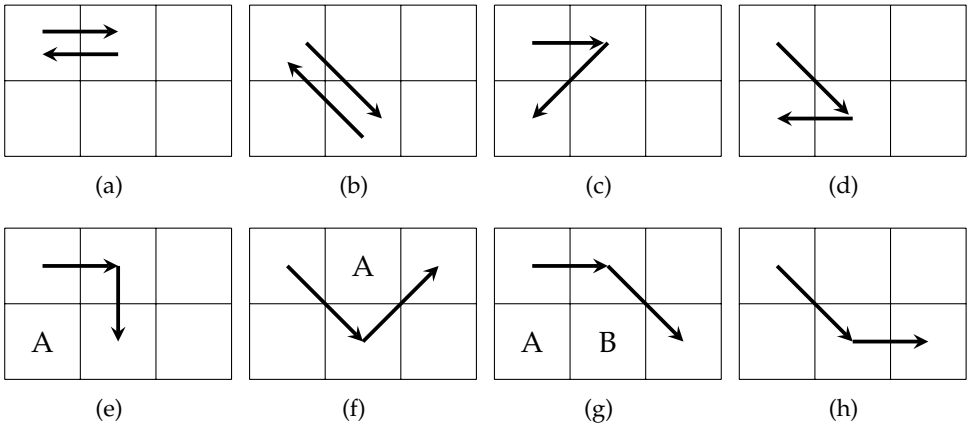


Figure 2.7: Types of turning point

Suppose the path contained any of the turning points in Figures 2.7(a), 2.7(b), 2.7(c), or 2.7(d). Then by replacing these moves with one or zero moves that go directly to the place where these moves end up, we can reduce the length of the path, which contradicts the optimality of our path. Therefore, our optimal path cannot contain any of these turning points.

Suppose the path contained the turning point in Figure 2.7(e). If node A is not an obstacle, we can replace this pair of moves by a single diagonal move, reducing the length of the path, so this is not possible in an optimal path. If instead node A is an obstacle, our turning point is a convex corner. Therefore, if our optimal path contains the turning point in Figure 2.7(e), we have found an optimal path travelling through a convex corner.

Suppose the path contained the turning point in Figure 2.7(f). Since corner cutting is not allowed, node A must not be an obstacle. But this means it is possible to replace this pair of moves by a pair of straight moves, via node A. This reduces the length of the path, so it is not possible to have this kind of turning point if our path is optimal.

Suppose the path contained the turning point in Figure 2.7(g). Since we used Algorithm 3 from Harabor and Grastien [2011] to ensure our path was diagonal-first, it cannot be possible to change this straight-to-diagonal turning point into a diagonal-to-straight turning point. Furthermore, point B cannot be an obstacle, otherwise we would be cutting a corner. Therefore, the reason we cannot change this turning point into a diagonal-to-straight turning point must be because we would be cutting across a corner, in which case point A is an obstacle. Therefore, the turning point must be a convex corner. So if our optimal path contains this turning point, then we have found an optimal path travelling through a convex corner.

So now, the only remaining possibility is that we have an optimal path with only the type of turning point illustrated in Figure 2.7(h). Clearly, our path must be octile, since it consists of a series of diagonal moves followed by a series of straight moves in a direction adjacent to the diagonal direction.

Use an algorithm similar to Algorithm 3 from Harabor and Grastien [2011] to

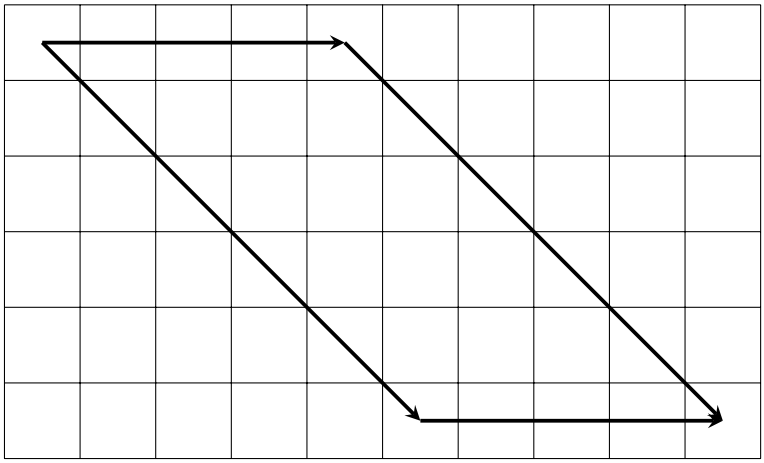


Figure 2.8: Straight-first octile path, diagonal-first octile path, and points in between.

change our diagonal-first path into a straight-first path. That is, replace diagonal-to-straight turning points by straight-to-diagonal turning points until this is no longer possible.

The same logic as before applies to show that if we have any of the turning points in Figures 2.7(a), 2.7(b), 2.7(c), 2.7(d), 2.7(e), or 2.7(f), then we must have an optimal path travelling through a convex corner.

The previous logic does not hold in the case of the turning point from Figure 2.7(g). However, we can apply essentially the same logic to the turning point from Figure 2.7(h), to show that if we have a turning point from Figure 2.7(h), then we have found an optimal path travelling through a convex corner.

We have now transformed our octile path with only diagonal-to-straight turning points into an octile path with only straight-to-diagonal turning points. I make the intuitive leap that when transforming our diagonal-first octile path to a straight-first octile path, the paths in between must contain all points on any octile path between the two points. This is not formally valid proof, but I find it sufficiently intuitively obvious that I am willing to let the reader prove it themselves, if they want. An illustration of this situation is in Figure 2.8. It is intuitively clear that in the process of replacing diagonal-to-straight turning points by straight-to-diagonal turning points one at a time, all points within the parallelogram will be passed through at some point.

Therefore, the two points must be safe-reachable from each other. We have now shown that if two points are not-safe reachable from each other, there must be an optimal path between them that passes through a convex corner.

Also note that in all the cases where we showed that there was a convex corner on the optimal path, this was at a turning point. A turning point cannot be at the start or the end of the path, because it has to have a move before it and a move after it. Therefore we know that the convex corner is not at a or b.

This completes our proof.

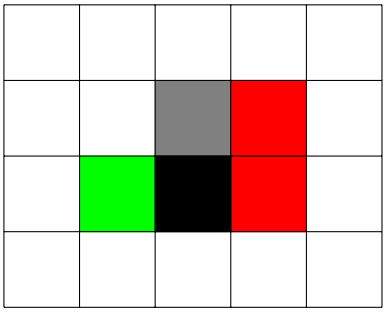


Figure 2.9: Rules for determining if we have a near-convex-corner

□

We have now proven that convex corners and safe-reachability satisfy the requirements desired of reachability and subgoals.

2.5.2 Near-Convex-Corners

When corner cutting is allowed, we can't choose our subgoals to be the set of convex corners. In this case, we use a slightly different set, which I call the **near-convex-corners**. I found the definition of near-convex-corners in the code for Brigitte Grastien [2019].

A point is a near-convex-corner if there is a straight direction in which there is an obstacle, one of the diagonal directions adjacent to that straight direction doesn't contain an obstacle, and it's not the case that the other diagonal direction and the straight direction adjacent to the other diagonal direction both contain obstacles.

This is illustrated in Figure 2.9. If the black square is an obstacle, the green square is not an obstacle, and it isn't the case that both red squares contain obstacles, then the gray square is a near-convex-corner. Note that this check is applied for all rotations and flippings of this template, and the square is a near-convex-corner if it passes any of these checks.

It may not be obvious from this description why I call these "near-convex-corners". This will be clear when you look at Figure 2.10. The near-convex-corners tend to be adjacent to the convex corners. There are some exceptions to this rule, though, as illustrated in Figure 2.11.

Theorem 3. *Suppose corner-cutting is allowed. If we have two points a and b in the same connected component such that b is not safe-reachable from a, then there is a near-convex-corner on an optimal path from a to b, which is not at a or b.*

Proof. The proof of this theorem closely follows the proof of Theorem 2. In this proof, I only describe the necessary changes that need to be made to the proof of Theorem 2 in order to prove this new theorem.

When dealing with the turning point in Figure 2.7(e), we can simply replace this pair of moves with a single move, reducing the length of the path without changing the destination, in which case the path is not optimal.

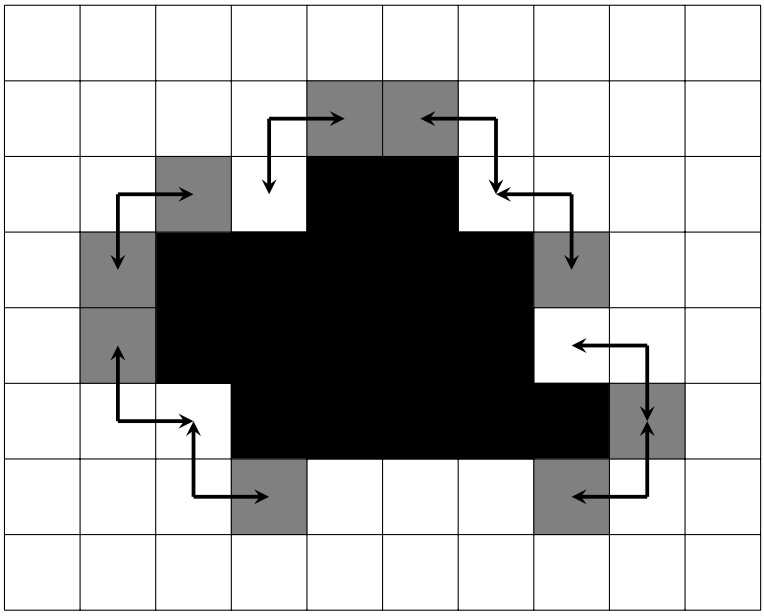


Figure 2.10: Examples of near-convex-corners

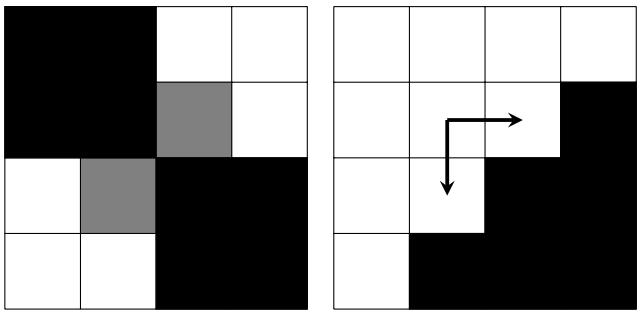


Figure 2.11: Exceptions to near-convex-corners being adjacent to convex corners

When dealing with the turning point in Figure 2.7(f), if point A is not an obstacle, then we can replace the pair of diagonal moves with a pair of straight moves, reducing the length of the path without changing the destination, in which case the path is not optimal. If we instead have that point A is an obstacle, then the turning point must be a near-convex-corner by the definition of a near-convex-corner, using the fact that points where the path passes through cannot be obstacles. In this case, we have a near-convex-corner on an optimal path between a and b.

When dealing with the turning point in Figure 2.7(g), we know that point B must be an obstacle, otherwise we would be able to change the straight-to-diagonal turning point into a diagonal-to-straight turning point. But then the turning point must be a near-convex-corner by the definition of a near-convex-corner, using the fact that points where the path passes through cannot be obstacles. Then we have a near-convex-corner on an optimal path between a and b.

I have described all the necessary changes to the proof of Theorem 2

□

2.5.3 Other Definitions of Reachability

A point is **diagonal-first reachable** from another point if the diagonal-first freespace path between them does not contain any obstacles. This is called "diagonal-first freespace reachability" in Harabor et al. [2019]. One downside of this reachability relation is that it is asymmetrical; if we know whether or not point a is diagonal-first reachable from point b, this does not necessarily tell us whether or not point b is diagonal-first reachable from point a.

In **canonical-freespace-reachablity**, one specific choice of octile path is considered canonical [Uras and Koenig, 2018]. Then a point is canonical-freespace-reachable from another point if the canonical path is free of obstacles.

In Uras and Koenig [2018], they specifically choose the canonical path to be the diagonal-first freespace path if the target has a larger x co-ordinate, and the straight-first freespace path otherwise. This is nice choice of canonical path because it ensures symmetry of the reachability relation. This is effectively a modification of diagonal-first reachability to make it symmetrical.

A point is **freespace reachable** from another point if it is possible to find some octile path between the points that does not contain any obstacles [Uras and Koenig, 2018].

Note that if any of these reachability relations are not satisfied, then safe-reachability is not satisfied. Therefore, we can use the Subgoal Property of safe-reachability with respect to convex corners to deduce the Subgoal Property for any of these reachability relations with respect to convex corners. This also holds with regards to near-convex-corners, in the case where we are allowing corner cutting.

2.5.4 N-level Subgoal Graphs

N-level subgoal graphs define a hierarchy on the subgoals in a subgoal graph such that between any two points in the graph, it is possible to find an optimal path between those points which is an arching path [Uras and Koenig, 2014]. Recall that an "arching path" is a path consisting of subgoals of increasing level in the hierarchy, followed by subgoals of decreasing level in the hierarchy. Thus, N-level subgoal graphs can be viewed as a combination of Subgoal Graphs with Contraction Hierarchies.

Alternatively, as briefly mentioned in Uras and Koenig [2018], N-level Subgoal Graphs can be viewed as the layering of Subgoal Graphs on top of each other. This wasn't really explained in detail, so I now give my own interpretation of this statement, based on their description of subgoal graphs. After generating the initial subgoal graph, we can find a subgoal graph within it, and another subgoal graph within that one, and so on, until this process is no longer possible. Then the "level" of a subgoal is equal to the number of iterations of this process for which this subgoal is included in the graph.

The definition of reachability and subgoals for each of these additional layers would be different to the definition chosen for the first layer. For each layer beyond the first layer, two nodes are defined as reachable if and only if there is an edge between them, and a set of subgoals is chosen such that it is possible to find a path between any two subgoals in the previous layer that only travels over subgoals in the current layer.

The advantage behind creating an N-level subgoal graph is that you know there is always an optimal path between two points which is an arching path, so when searching the graph, you can choose not to explore any nodes which would cause the path to not be an arching path [Uras and Koenig, 2014].

2.6 Topping

Topping is a combination of Compressed Path Databases with JPS+, introduced by Salvetti et al. [2018]. It stands for "Two-Oracle Path PlannING", because it contains a CPD oracle and a JPS+ oracle. The CPD oracle needs to be slightly modified to ensure that it always returns diagonal-first paths, so as to be compatible with the JPS+ oracle.

Then in order to find a path, Topping uses the CPD oracle to work out what the first move should be. Then it uses the JPS+ oracle to work out how many moves in that direction should be made in order to get to the next jump point. This process is repeated until the goal is reached.

By using the JPS+ oracle to work out how far we should go, it is possible to perform many steps at once with only one lookup to the CPD oracle and one lookup to the JPS+ oracle. This is much more efficient than performing many lookups with the CPD oracle, and it still allows for the perfect reduction in branching factor that the CPD allows for.

2.7 JPS+BB

In Rabin and Sturtevant [2016] they apply Geometric Containers to pathfinding on octile grids. Geometric Containers can be viewed as having a similar goal to CPDs in that they aim to compress the table of optimal first moves from a point to each other point. However, Geometric Containers do this in a completely different, more lossy way.

Suppose for a certain point, we are given a table of optimal first moves towards every other point. Then for every first move, a Geometric Container is constructed that contains all points associated with that first move. When answering a query, this Geometric Container can be used to prune the search. If the goal is outside of the Geometric Container, this indicates that a different first move must be optimal to reach the goal, so you can prune the move associated with the Geometric Container from the search.

A Geometric Container is allowed to contain points with different optimal first moves to the move associated with the Geometric Container. Ideally, the number of such points should be minimized, so as to make the Geometric Container more informative, and increase the amount of pruning caused by the Geometric Container. However, if we allow these points to be contained in the Geometric Container, it might allow us to reduce the amount of information required to store the Geometric Container, or reduce the amount of time needed to check whether or not a point is in the Geometric Container, or provide some other advantage.

The type of Geometric Container used by Rabin and Sturtevant [2016] is called a Bounding Box (**BB**). A BB is a rectangle of points; it contains all points with an x-coordinate within a certain range and a y-coordinate within a certain range. This makes it a very simple type of Geometric Container. It is very quick to verify whether or not a point is in a bounding box, and it doesn't take much memory to store a bounding box. The downside is that bounding boxes may contain far more points than necessary, in which case their ability to prune search may be impaired.

Rabin and Sturtevant [2016] combine BBs with JPS+ so as to be able to take advantage of the precomputed distances to the next jump points, and jump multiple squares at every step. The resulting algorithm is called Jump Point Search Plus Bounding Boxes (**JPS+BB**). In order to ensure that the BBs are compatible with JPS, the first-move database used to construct the BBs needs to contain only first moves on diagonal-first paths, rather than general paths. Otherwise, the BB might prune off a diagonal-first optimal path, which is a problem since JPS+ only considers diagonal-first optimal paths.

Furthermore, when combining with JPS+, if there are multiple first moves on optimal diagonal-first paths towards a point, then that node must be contained in the BBs for all directions [Rabin and Sturtevant, 2016]. This is because in JPS+, the moves available out of a node depend on the direction you enter the node in.

2.8 Storing data only for Jump Points

Hu et al. [2021] showed that it is sometimes possible to reduce the amount of preprocessing data that needs to be computed and stored by only storing data for jump points. They applied this idea to JPS+BB, creating JPS+BB+, and they applied this idea to Topping, creating Topping+ and TOPS.

JPS+BB+ is essentially like JPS+BB, except that the bounding boxes are only computed and stored for jump points. Then search can proceed mostly as normal, except that pruning will not be available on the very first jump. The disadvantage of this is that slightly less pruning is possible, which will slightly increase the number of nodes that need to be expanded when searching for the goal. The advantage of this is that far less memory will be required to store the bounding boxes, and far less computation time will be required to precompute the bounding boxes. Furthermore, reducing the amount of memory used can often increase performance, because, for example, it might allow a larger amount of relevant data to fit into a smaller cache with quicker access times, or it might bring relevant pieces of data closer to each other by cutting out swathes of mostly irrelevant data, allowing more pieces of relevant data to fit into a cache line.

TOPS and Topping+ both store CPD data for every jump point. This means that once you have reached a jump point, you immediately know in what direction to jump in order to optimally travel towards any other point in the map. However, the difficulty is in effectively connecting non-jump-points to the graph of jump points. TOPS and Topping+ differ in the way they do this. TOPS is based on search, whereas Topping+ is based on path extraction.

In TOPS, a search is initiated using JPS+, and at every jump point, the CPD data is used to reduce the branching factor to 1. This requires search, which may mean the algorithm wastes time fruitlessly going down the wrong direction, in constrast to Topping, which only extracts a single path. The advantage TOPS has over Topping is that a CPD is only required at each jump point, instead of each point, which reduces the amount of preprocessing time to compute the CPDs and the amount of memory required to store them.

In Topping+, one path is extracted for each jump point successor of the start location. It then chooses the shortest of these paths. This requires the extraction of multiple paths, unlike Topping, which only requires the extraction of a single path. However, like TOPS, Topping+ only requires a CPD at jump points, and so it has the same preprocessing time and memory advantages that TOPS has.

2.9 JPS and Subgoal Graphs

Harabor et al. [2019] showed that the jump points in Jump Point Search can be viewed as subgoals in a Subgoal Graph. To do this, the graph needs to be modified slightly. The nodes in the modified graph are defined to be the set of all node-direction pairs, for every node in the original graph and each of the 8 directions in an octile grid. There is an edge between two node-direction pairs if and only if when travelling

into the first node in the first direction, JPS will decide to expand the second node and travel in the second direction. Then it can be shown that by taking subgoals to be straight jump points and our reachability relation to be diagonal-first freespace reachability, we get a subgoal graph on the modified graph.

2.10 Brigitte

Brigitte is an ultra-fast pathfinding algorithm that aims to compress the optimal paths between all pairs of nodes by grouping nearby nodes into regions, and focussing on finding paths between all points in a region and all points in another region [Grastien, 2019]. It is possible to view it as essentially being like a CPD on the scale of regions, instead of on the scale of nodes.

My project was originally intended to find improvements to the Brigitte pathfinding algorithm. In fact, I had done a previous project finding improvements to the Brigitte pathfinding algorithm, and my honours project was meant to build on top of that work. However, my work on Brigitte lead me to come up with a significantly different algorithm that encapsulates some of the core features that make Brigitte work well. Some of the code in my new algorithm is still based on the code from Brigitte.

Brigitte defines regions in such a way that it is easy to find a path between any two points in a region. Specifically, a **convex region** is defined to be a set of points such that for any two points in the region, it is possible to find an octile path between them which is contained completely within the region.

Suppose we wanted to find an optimal path between two points in a region. Since there is an octile path between the two points, it must be possible to step once towards the destination in either the straight direction or the diagonal direction, while remaining in the region. After taking such a step, we have reduced the distance to the goal, and since we remain in the region, there must again be an octile path to the goal. Then using the same logic, we can take another step, and another, and another, until we reach the goal.

In order to travel between adjacent regions, Brigitte uses the concept of an "abutment". This concept roughly translates to the concept of a "first move" in a CPD.

Brigitte defines an **abutment** to be a structure containing three nodes: a **root node** r, and two **leg nodes** a and b. The root node is the tip of a cone with its edges defined by the two leg nodes. The leg nodes are contained in the region the abutment attaches to, whereas the root node usually shouldn't be.

A **bridge** between two regions contains a pair of abutments, one for each region. It also stores the optimal distance between the roots of each of the abutments, as well as the sub-bridge along which it is possible to travel in order to optimally reach the root of one abutment from the root of the other abutment. At least one of the two abutments of the bridge should have its root node outside of the corresponding region, otherwise when travelling over the bridge, we will remain in the same region, and the same bridge will be chosen repeatedly to find the optimal path between the

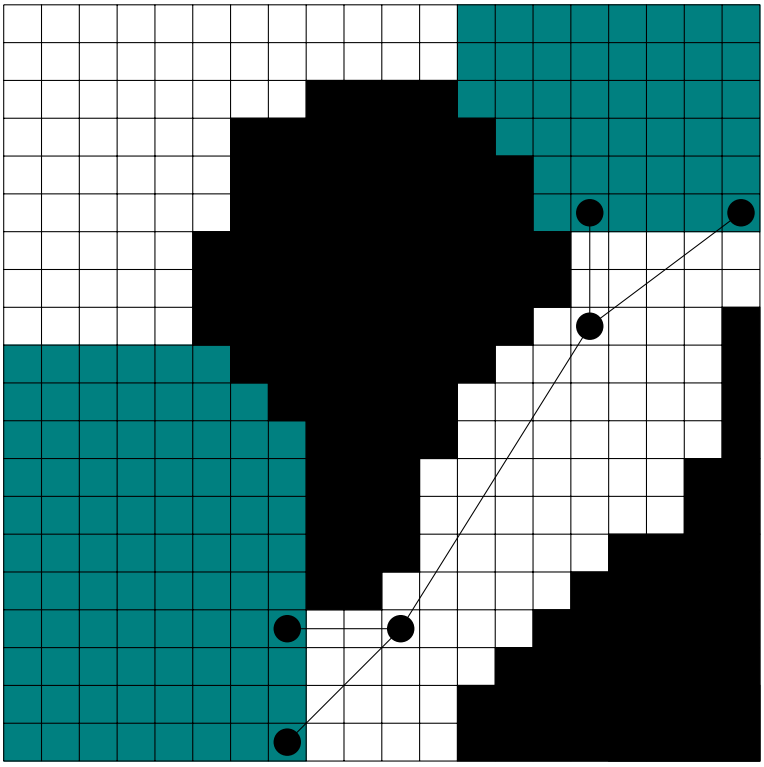


Figure 2.12: Example of a bridge

two points, and we will never make progress.

The general idea is to find a set of bridges between each pair of regions that covers the two regions, in the sense that for any pair of points in the two regions, the optimal path between these two points can be found by travelling over some bridge between the two regions.

To travel over a bridge, you first travel over each abutment. Then you travel over the corresponding sub-bridge. To travel over an abutment, you either travel directly towards the root of the abutment if the point is in the cone of the abutment, or you travel to the nearest leg and then you travel towards the root of the abutment.

To find an optimal path between two points using Brigitte, you work out which region each point belongs to, and consider the set of bridges corresponding to these regions. You then determine the distance that it would take to travel over each of these bridges. The bridge that minimizes this distance is the bridge you travel over.

Chapter 3

The Complete Subgoal Graph Algorithm

The algorithm that I have designed is called the Complete Subgoal Graph Algorithm (**CSG**)[1]. The purpose of CSG is to eliminate almost all search from subgoal graphs, in order to allow for the ultra-fast answering of pathfinding queries. This comes at a cost: significant amounts of memory and preprocessing time are required. Additional modifications can be made to CSG to offset this cost and improve its performance characteristics.

The process of finding paths using subgoal graphs consists of four stages:

1. Check if the goal is reachable from the source, and if so, find the path between them directly.
2. Connect the source and goal to the subgoal graph.
3. Search over the subgoal graph to find a path of subgoals between the source and the goal.
4. Refine the path of subgoals into a path of nodes.

Stage 1 takes relatively little time, so it isn't modified in CSG.

Stage 2 involves search in order to find the subgoals that are reachable from the source and goal. In CSG, this search is removed by precomputing these reachable subgoals for every point and storing them.

Stage 3 is the main component that involves search. CSG removes this search by storing a complete subgoal graph. This complete subgoal graph contains the precomputed optimal distances between every pair of subgoals, and the first subgoal on the optimal path between every pair of subgoals. Then when it comes to searching the subgoal graph, we consider each pair of subgoals connected to the source and the goal. We choose the pair of these subgoals that minimizes the total distance from the source to the goal, using the complete subgoal graph to simplify calculation of this distance. We can then use the complete subgoal graph to repeatedly find the

[1] https://github.com/Eric-C-Hall/CompleteSubgoalGraph

next subgoal on the optimal path between these two subgoals, and this will give us a path between the source and the goal.

Note that this is similar to the way in which CPDs work, and it was inspired by the work of Botea [2011] on CPDs. Unfortunately, unlike in CPDs, run-length encoding is not an efficient method of compressing the table of distances between each pair of subgoals, since it is rare for the optimal distance between a pair of points to be the same as the optimal distance between another pair of points. However, this is somewhat balanced out by the fact that the complete subgoal graph only needs to store the distances between subgoals, whereas a CPD has to store the optimal direction to travel between each pair of non-obstacles, and there are usually far fewer subgoals than non-obstacles.

Stage 4 does not involve search, and is not modified in CSG.

The underlying subgoal graph for CSG needs to have some definition of reachability and some definition of subgoals. In practice, I used the same definitions for subgoals and reachability as were used in Uras et al. [2013]. That is, for reachability, I used safe-reachability, and for subgoals, I used convex corners. See section 2.5.1 for definitions of these terms.

The above summary should be sufficient to provide a good understanding of how the algorithm works. In the rest of this chapter, I will explain the algorithm and its implementation in more detail.

3.1 Data structures

The following data structures are used by the Complete Subgoal Graph Algorithm:

1. Subgoal vector: Vector of all subgoals
2. Safe-reachable subgoal map: Map from each point to subgoals safe-reachable from that point
3. Complete subgoal graph, consisting of the following components:
   (a) Next subgoal table: Map from each goal subgoal to a map from each start subgoal to a subgoal that is safe-reachable from the start and is on an optimal path between the start and goal subgoals
   (b) Distance table: Map from each goal subgoal to a map from each start subgoal to the optimal distance between the start and goal subgoals

3.2 Efficient Implementation of Maps

The algorithm uses several maps. Maps can be implemented in many different ways. The most common ways include using a hash table and using a binary search tree. However, these implementations need to perform a significant amount of calculation in order to read or modify a piece of data in the map. This is a problem, as a

large amount of the time taken by the algorithm involves reading from these maps. For example, finding an optimal path between the first and last subgoals consists of repeatedly looking up and travelling to the next subgoal on the optimal path.

In our specific use cases, we know that each key will be mapped to a piece of data. This allows us to implement maps much more efficiently. Each key can be associated with a specific integer i between 0 (inclusive) and the number of keys (exclusive). Then the associated value can be stored in the ith index of an array. This is similar to a hash table, except that we can ensure a one-to-one correspondance between keys and spaces in the table, so we don't need to worry about hash collisions or creating too many buckets.

When a point (x, y) is used as a key, it is associated with the integer $y \cdot w + x$, where w is the width of the map. When a subgoal is used as a key, it is associated with the index it has in the vector of subgoals.

This implementation of maps is based on the code for Brigitte [Grastien, 2019].

Implementation of Next Subgoal Table When implementing the next subgoal table, it may seem more natural to index the outer vector according to the start subgoal, and to index the inner vectors according to the goal subgoal. However, in practice, it makes more sense to index the outer vector according to the goal subgoal, and to index the inner vector according to the start subgoal. This is because the goal doesn't change as you travel towards it, but the source does. This allows you to dereference the outer vector only once, after which all remaining dereferences occur in the same inner vector. Since all the relevant information is in the same inner vector, it is probably close together in memory, and so it's more likely for relevant pieces of data to be in the same cache lines, which makes reading from memory more efficient. It is unclear whether this causes a significant improvement in the time taken by the algorithm, but it does not have any significant drawbacks, so it may as well be implemented this way.

Reducing Memory Usage in the Distance Table We can exploit the fact that the distance between a and b is the same as the distance between b and a to halve the size of the distance table. We only need to store the distance in one direction, and we also don't need to store the distance between a subgoal and itself. Thus, we only need to store those distances between subgoals where the first subgoal has a smaller index than the second subgoal. This fact is exploited in many algorithms, for example, this is the basis of the work in Salvetti et al. [2017].

3.3 Preprocessing procedure

The preprocessing procedure works as follows:

1. Find all the subgoals
2. Find all safe-reachable subgoals from each point

3. Compute the optimal distances between each pair of subgoals
4. Compute safe-reachable subgoals on an optimal path between each pair of subgoals

Step 1 is completed by looping over each point and checking whether it satisfies the definition of a subgoal.

Step 2 is completed by looping over each subgoal, identifying which points are safe-reachable from that subgoal, and adding the subgoal to the set of safe-reachable corners from each of these points. This is more efficient than directly finding all subgoals safe-reachable from each point, and doesn't add any complexity to the algorithm.

Step 3 is completed by applying Dijkstra's algorithm over the subgoal graph starting from each point in the subgoal graph. Note that this is much more efficient than applying Dijkstra's algorithm over the entire graph including non-subgoals starting from each point in the subgoal graph.

Step 4 is completed using the distances obtained in Step 3. For each pair of start and goal subgoals, these distances can be used to calculate the distance between the start and goal subgoals via any of the subgoals safe-reachable from the start subgoal. The subgoal that minimizes this distance is the one on an optimal path.

3.4 Online Pathfinding procedure

The online pathfinding procedure works as follows:

1. Try the diagonal-first path between the start and the goal.
2. Find the first and last subgoals on the optimal path.
3. Travel to the first subgoal, repeatedly travel to the next subgoal until the last subgoal is reached, and travel to the goal.

If an obstacle-free path is found in Step 1, this path can be returned immediately without running Steps 2 and 3. Otherwise, we know that there is an optimal path through at least one subgoal, since we have shown that the goal is not safe-reachable from the start and we know that safe-reachability and subgoals together satisfy the Subgoal Property.

Step 2 is completed by looping through all pairs of subgoals a and b that are safe-reachable from the start and goal nodes respectively and using the complete subgoal graph to determine which such pair minimizes the distance it takes to travel from the start to the goal via a and b.

In Step 3, "travelling" refers to choosing some arbitrary octile path between the end of the path and the next point, and adding it to the end of the path. The next subgoal is determined using the complete subgoal graph.

Chapter 4

Runtime Improvements

In order to ensure that CSG has an online runtime that is competitive with other similar algorithms, a number of speed improvements need to be made to the algorithm. The majority of these speedups revolve around reducing the amount of time it takes to find the first and last subgoals on the optimal path. This is one of the slowest parts of the algorithm, because all pairs of potential first and last subgoals need to be considered, which means it takes quadratic time in the average number of subgoals safe-reachable from a point.

The first two speedups involve noticing that if we are entering a subgoal from a certain direction, there is only a certain set of nodes towards which travelling via this subgoal would be optimal. This allows us to ignore all subgoals for which this set is empty, and treat subgoals for which this set is nonempty but contains no subgoals in a more efficient manner.

The next speedup involves borrowing the idea of bounding boxes from the work of Rabin and Sturtevant [2016] on JPS+BB, extending it to take into account the previous observation, and applying it when trying to find the first and last subgoals.

The final speedup involves increasing the distance travelled at each step when computing the path between the first and last subgoals, by using a less restrictive notion of reachability than safe-reachability for the next subgoal table.

4.1 Directionally Relevant Subgoals

The points that can be optimally reached via a subgoal may change depending on the direction of travel into the subgoal. That is, a subgoal is only "directionally relevant" for certain points, depending on the direction of travel into the subgoal. By taking the direction of travel into account, it is possible to prune unnecessary successors for subgoals.

This can be viewed as applying some of the characteristics of JPS+ to subgoal graphs. In JPS+, the successors to a given jump point depend on the direction of travel when entering a jump point. It is only necessary to consider a jump point when you are entering and leaving it in certain directions. If you are entering and leaving it from the wrong directions, you don't need to consider the jump point.

In Harabor et al. [2019], they showed that jump points can be viewed as a type

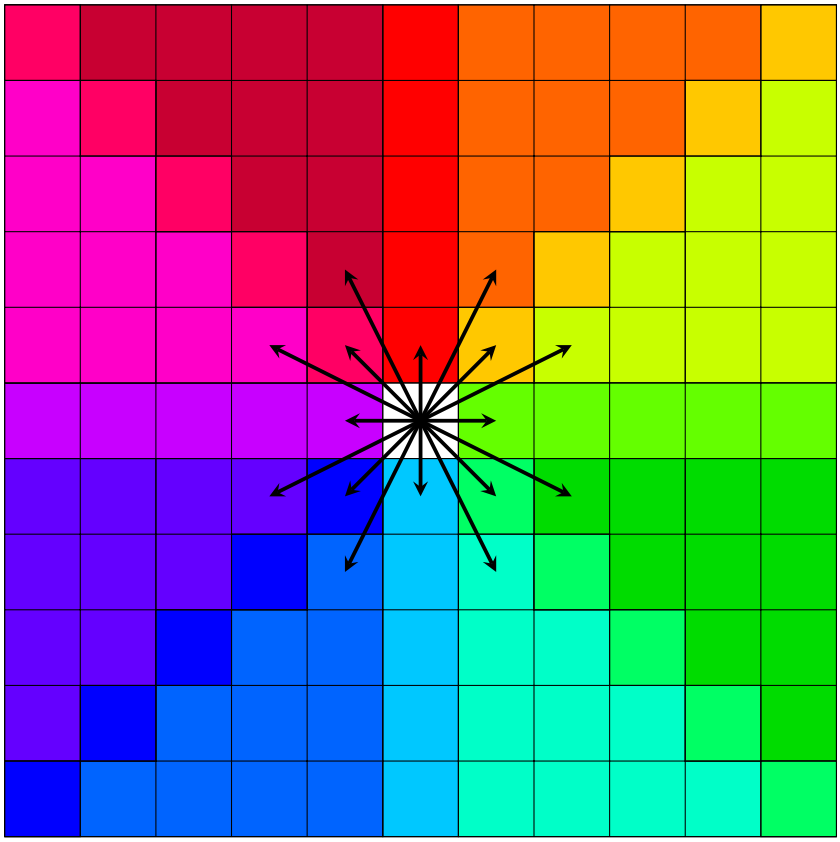


Figure 4.1: The points associated with the 16 cardinal, intercardinal, and secondary intercardinal directions

of subgoal graph, where each node in the graph is split up into 8 different nodes, depending on the direction of travel into the node. I am trying to do the opposite. I am trying to take ordinary subgoal graphs and split their nodes up into directions, in order to reduce the number of successors for each node.

4.1.1 Definition of Directionally Relevant Subgoals

For the purpose of defining directionally relevant subgoals, we consider 16 different directions. We consider the **cardinal directions** (N, S, E, W), the **intercardinal directions** (NE, SE, SW, NW), and the **secondary intercardinal directions** (NNE, ENE, ESE, SSE, SSW, WSW, WNW, NNW).

If a point b can be reached from a point a through a series of steps in a single cardinal/intercardinal direction d, then we say that the **direction between the points** a and b is d. If this is not possible for any single cardinal/intercardinal direction, but it is possible using steps in two adjacent cardinal/intercardinal directions, then we say that the **direction between the points** a and b is the secondary intercardinal direction which lies between these two directions. This is illustrated in Figure 4.1

For each outgoing direction, we have a **representative node** in that direction. This

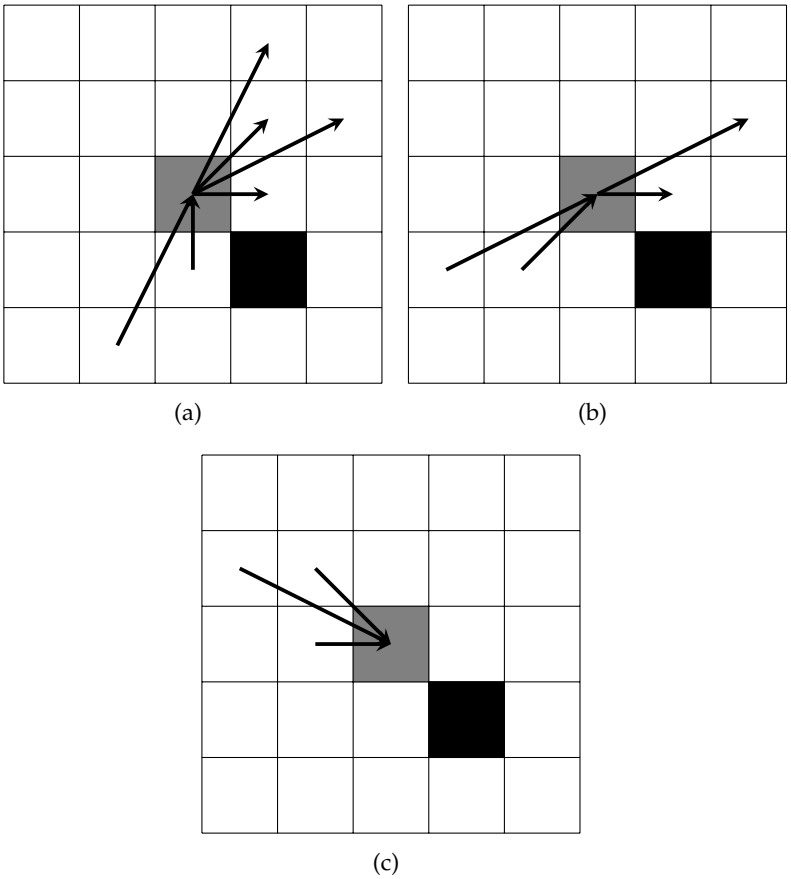


Figure 4.2: The outgoing directions that are directionally relevant to the given incoming directions

is the node pointed to by the corresponding arrow in Figure 4.1. Similarly, for each incoming direction, we have a **representative node** coming in from that direction, which is the same as the node corresponding to the opposite outgoing direction.

A convex corner c is defined to be **directionally relevant** to a given incoming direction and outgoing direction if c is safe-reachable from the incoming representative node, the outgoing representative node is safe-reachable from c, and the outgoing representative node is not safe-reachable from the incoming representative node.

Note that the octile paths from c to any point in a certain outgoing direction will always include octile paths that pass through the representative node for that outgoing direction. This is why the representative node makes a good representative of all the nodes in that direction. The same idea applies in the reverse direction for incoming directions.

Since there are only a small number of possible representative nodes, it is easy to exhaustively show which representative nodes satisfy the relevance criterion. In the case of a single obstacle, the results are shown in Figures 4.2(a), 4.2(b) and 4.2(c).

A convex corner c is defined to be **directionally relevant** to the points a and b if

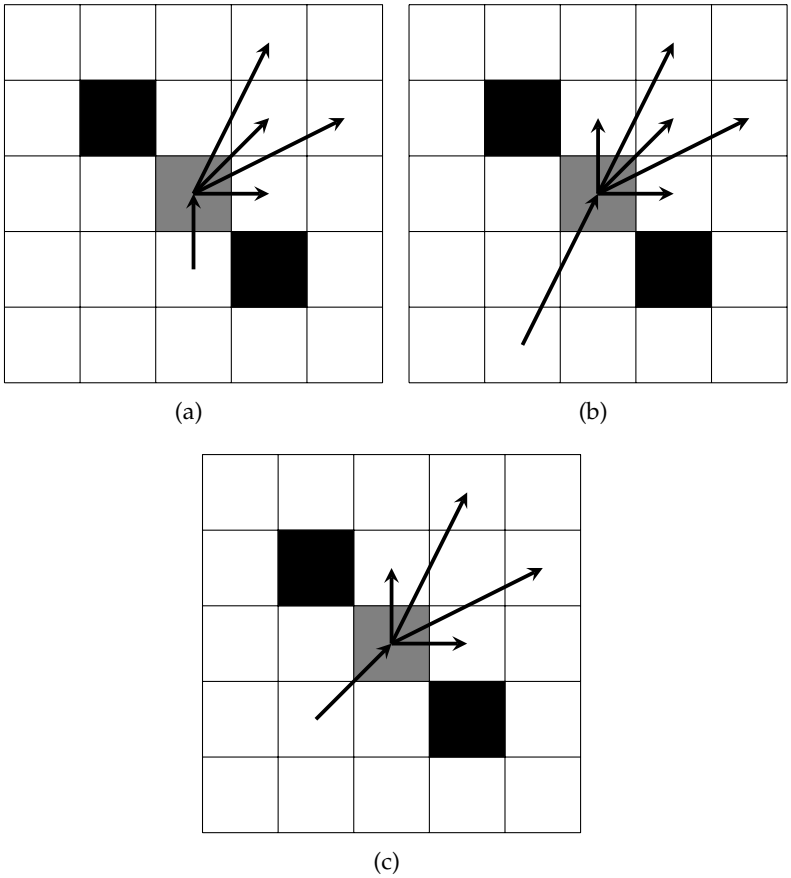


Figure 4.3: Example of when relevant directions to a secondary intercardinal direction are different to the relevant directions of adjacent cardinal/intercardinal directions

the points a and b are safe-reachable from c and the incoming direction from the first point to c is relevant to the outgoing direction from c to the second point.

Lemma 1. *If a point b is not safe-reachable from a point a, then there is a convex corner c on an optimal path between a and b which is directionally relevant to the points before it and after it on the optimal path, which is not at a or b.*

Proof. We can prove this in the same way as Theorem 2 was proved, except that whenever that proof would show that there is a convex corner on an optimal path between a and b, we additionally need to show that this convex corner is relevant to its incoming and outgoing directions. This only occurs in the proof two times: when dealing with the turning point in Figure 2.7(e) where point A is an obstacle, and when dealing with the turning point in Figure 2.7(g) where point B is not an obstacle and point A is an obstacle.

In the case of Figure 2.7(e), where point A is an obstacle, the points before and after the turning point are not safe-reachable from each other due to the presence of the obstacle at point A, but the turning point is safe-reachable from each of these points, therefore the convex corner is directionally relevant to the points before and after it on the path.

In the case of Figure 2.7(g), where point B is not an obstacle and point A is an obstacle, we again see that the points before and after the turning point are not safe-reachable from each other due to the presence of the obstacle at point A, but that the turning point is safe-reachable from each of these points, therefore the convex corner is directionally relevant to the points before and after it on the path.

We have completed the necessary modifications to the proof of Theorem 2.

□

Theorem 4. *If we have two points a and b in the same connected component such that b is not safe-reachable from a, then there is an optimal path betweeen a and b which contains a pair of convex corners c and d such that c is directionally relevant to a and d, where c is not at a or b.*

Proof. Apply Lemma 1 to the points a and b. If the convex corner returned by this lemma is not safe-reachable from a, we can apply the lemma again to a and the convex corner returned by the previous application. Keep applying the lemma in this way as many times as possible. We will eventually be unable to do this any more, because the distance between a and the convex corner returned by the lemma decreases in each iteration. Therefore, the point returned by the lemma will eventually be safe-reachable from a. So we will obtain a convex corner which is safe-reachable from a and is relevant to the previous and next points on the path. Through the same process, we can also obtain a convex corner d on an optimal path between c and b which is safe-reachable from c.

We want to show that c is relevant to a and d. We know that the optimal path between a and c travels through the point p immediately before c on the path. This means that the direction between a and c must either be the direction that p is a representative for, or it must be one of the secondary intercardinal directions adjacent

to that direction. Similarly, if we let q be the point immediately after c on the path, then the direction between c and d must either be the direction that q is a representative for, or it must be one of the secondary intercardinal directions adjacent to that direction.

In the case where the direction between a and c is the same as the direction beween p and c and the direction between c and d is the same as the direction between c and q, we immediately have that c is relevant to a and d.

Otherwise, we know that the direction from c to either a or d is a secondary intercardinal direction. Let r be the representative point for this direction. Also consider the direction from c to the other point out of a or d. Let s be the representative point for this direction.

Without loss of generality, use rotations and flippings to rotate our perspective so that r is to the SSE. For the majority of possible representative points s, if s was safe-reachable from r, it would be possible to reduce the length of the path by travelling directly between r and s instead of travelling between them via c, in which case the path isn't optimal, which is a contradiction. Then these representative points are not safe-reachable from each other, and hence c is relevant to a and d.

The representative points that this doesn't work for are those in the directions NNE, NE, and N. In any of these cases, we know that the previous point is incoming in the NE or N direction, and the next point is outgoing in the NE or N direction. It is not possible for a subgoal to be relevant for NE and NE, and it is not possible for a subgoal to be relevant for N and N, therefore the subgoal must be relevant for either N and NE, or for NE and N.

The case of N and NE is indicated in figure 4.4(a). Point A cannot be an obstacle because q is safe-reachable from c. Then point B must be an obstacle since p and q are not safe-reachable from each other. Then the representatives for the outgoing NE and NNE directions are both not safe-reachable from r. That is, s is not safe-reachable from r, in which case c is relevant to a and d.

A symmetrical argument holds in the case of NE and N.

□

By the repeated application of this theorem, we find that there is an optimal path between any two points that passes through a sequence of subgoals where each subgoal is directionally relevant to the previous and next subgoals.

To be more concrete, we can apply the theorem to points a and b to obtain two convex corners c and d such that c is directionally relevant to a and d. We can then apply the theorem to points c and b. Note that the first convex corner returned by the second application of this theorem will correspond to the second convex corner returned by the first application of this theorem, since in the proof of this theorem, we constructed these convex corners using the same process. So the second application of this theorem will return the convex corners d and e, such that d is directionally relevant to c and e. We can then apply the theorem to the points d and b, and we can continue applying the theorem in this way. We will eventually be unable to apply the theorem, since at every stage the distance from the last convex corner to b will

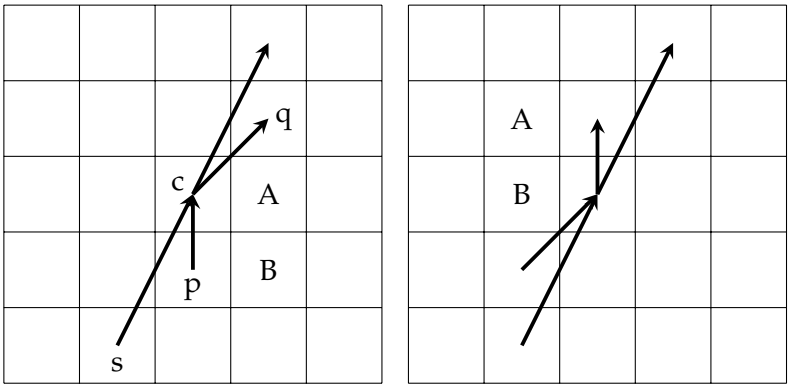


Figure 4.4: Illustrations of the cases in which the previous point is incoming in the NE or N direction and the next point is outgoing in the NE or N direction

decrease. At this point, b must be safe-reachable from the last convex corner, and so we can complete the path.

4.2 Pruning with Directional Relevance

Some subgoals safe-reachable from a given point p are not directionally relevant to any other points when entered from the direction of p. It will never be necessary to travel through a subgoal without points it is directionally relevant to in order to find an optimal path, because there will always be an optimal path that only passes through a sequence of directionally relevant subgoals. This allows us to prune these subgoals from the vector of subgoals safe-reachable from p.

Some subgoals safe-reachable from p may be directionally relevant to certain points when entered from the direction of p, without being directionally relevant to any subgoals when entered from that direction. A subgoal without subgoals it is directionally relevant to only needs to be the first subgoal on a path if it is the only subgoal on the path, because there will always be an optimal path that only passes through a sequence of directionally relevant subgoals. Likewise, if we consider the incoming direction to be coming from the goal instead of from the start, it is only necessary for it to be the last subgoal on a path if it is the only subgoal on the path.

We can exploit this to prune these subgoals when trying to work out which subgoals are the first and last subgoals on the path. Instead of just testing all possible pairs of first and last subgoals in the usual way, we can split this up into two parts: testing distinct first and last subgoals, and testing cases in which the first and last subgoal are the same. Testing cases in which the first and last subgoal are the same can be performed in linear time, whereas testing distinct first and last subgoals has to be performed in quadratic time. During the quadratic test, all subgoals that are not directionally relevant to any subgoals can be pruned, but in the linear test, all safe-reachable subgoals that are directionally relevant to some points must be considered,

regardless of whether or not they are directionally relevant to any subgoals.

It may not be obvious how to test the cases in which the first and last subgoals are the same in linear time. The easiest way to do this is to ensure that the vectors of safe-reachable subgoals near the start and the goal are sorted according to some ordering of positions, and step through the elements in each vector in such a way that a step is taken when the current element in the current vector is less than or equal to the current element in the other vector. Then each vector will be stepped through entirely once, and if a pair of elements has the same place within the ordering, that pair of elements will have been recognised.

We have reduced the amount of subgoals that need to be considered in the quadratic-time stage by adding a linear-time stage to the algorithm, which is a significant improvement.

4.3 Bounding boxes

In order to improve the online runtime further, CSG can be augmented with bounding boxes, similar to those used by Rabin and Sturtevant [2016] in JPS+BB. This change aims to further prune the subgoals we need to consider as first or last subgoals.

For every subgoal and every incoming direction, a bounding box is constructed, that contains all points p such that in order to optimally reach p from the representative for the incoming direction, it is necessary to travel through the subgoal. Then when we are trying to decide which subgoal should be the first subgoal on the optimal path, we can prune those subgoals for which the goal is not contained in this bounding box, since the bounding box was constructed in such a way that it contains all points for which pruning this subgoal is not possible. Similarly, when we are trying to decide which subgoal should be last, we can prune those subgoals for which the start is not contained in the appropriate bounding box.

This process only takes linear time in the number of subgoals safe-reachable from the start/goal. So we have reduced the number of subgoals we need to consider in the quadratic-time stage, at the cost of adding a linear-time stage to the algorithm.

The bounding boxes in CSG+BB can be improved over the bounding boxes in JPS+BB by exploiting directional relevance. While the ones in JPS+BB contain all points that are optimally reachable after performing a certain move, the bounding boxes in CSG+BB contain only those points that are optimally reachable though a path of subgoals where each subgoal is directionally relevant for the previous and next subgoals. This allows us to reduce the size of the bounding boxes, which increases the amount of pruning that is possible using bounding boxes. Figure 4.5 illustrates an example of a bounding box that has been reduced in size using directionally relevant subgoals. In the figure, the bounding box corresponding to the left subgoal would ordinarily have to contain all of the blue nodes above the left subgoal, but it can be empty if we exploit directional relevance. Ordinarily, the bounding box corresponding to the right subgoal would have to contain all the the red and blue

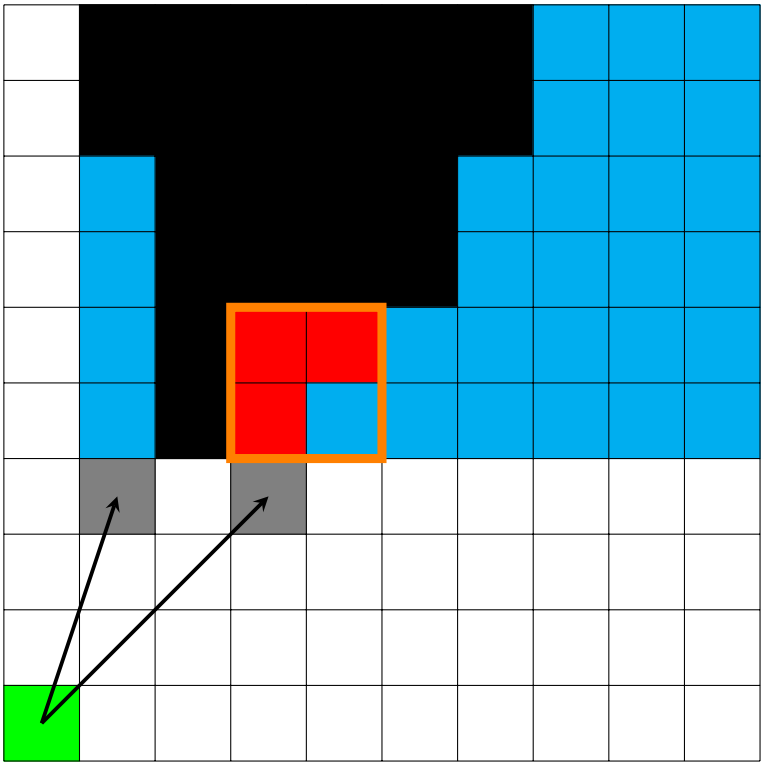


Figure 4.5: Example of bounding boxes simplified using relevant subgoals. Blue nodes are optimally reachable via the subgoal, red nodes are optimally reachable through a path of relevant subgoals.

nodes above and to the right of the right subgoal, but when exploiting directional relevance, it only needs to contain all the red nodes.

Note that the pruning caused by application of bounding boxes using directional relevance subsumes all the pruning caused by directly exploiting subgoals that aren't directionally relevant to any subgoals. However, in practice, pruning using bounding boxes is slightly slower than directly exploiting subgoals that aren't directionally relevant to any subgoals, so it may be better to prune using that method first before pruning using bounding boxes. However, the slight speedup from doing this might not be worth the more complex algorithm it requires. In contrast, pruning the subgoals that are not directionally relevant to any points at all is worth it regardless, since this requires no computation during the online stage at all, and it doesn't add as much complexity to the algorithm.

4.4 Travelling as Far as Possible in the First Move Table

The subgoal graph uses safe-reachability as its reachability relation. Safe-reachability is a much more restrictive notion of reachability than many other notions of reachability, such as diagonal-first reachability. When trying to find the first and last

subgoals on an optimal path, this is a good thing, because it reduces the number of possible candidates for the first and last subgoals. However, when trying to find a path between the first and last subgoals, this means that each step doesn't go as far as would be possible when using another notion of reachability. This increases the number of steps that need to be taken and increases the number of lookups that need to be made to the next subgoal table.

In order to reduce the number of steps that need to be taken, the notion of reachability used in the first move table can be relaxed from safe-reachability to diagonal-first reachability. In order to construct the diagonal-first first move table, for every pair of subgoals, we apply the safe-reachability first subgoal table repeatedly, and we set the subgoal in the diagonal-first first move table to be the last of the returned subgoals which is diagonal-first reachable from the start.

Chapter 5

Performance Analysis and Correctness Testing

5.1 GPPC Code

Ideally, evaluation of methods for path planning should take place in a standardised environment, to enable the comparison of path planning techniques. The main standardised environment used for the comparison of grid-based path planning methods is the Grid-Based Path Planning Competition (the **GPPC**) [1]. It has been announced that the GPPC will be resuming this year, however, at the time of writing this, submissions do not yet seem to be open.

In order to allow for the testing of algorithms on local machines in a way compatible with the GPPC, the organisers of the GPPC have provided code that is similar to that used in the competition. As of writing this, this code was last updated on May 3, 2012; this is the version of the code that we use. The GPPC code is compatible with the MovingAI benchmarks for Grid Based Path Planning Sturtevant [2012].

The GPPC code has a few methods that are used to hook the GPPC code up to the algorithm being evaluated. These are all contained in the file "Entry.cpp". There is a function that returns the name of the algorithm, a function that preprocesses the map and saves the preprocessing data, a function that loads the preprocessing data, and a function that returns a path given a query.

It is best for the Entry.cpp file to be the only file that is modified, because the GPPC code provides a standardised framework to compare different algorithms to each other. If the GPPC code is modified, then this framework is no longer standardised, and it becomes harder to compare algorithms to each other. We cannot be sure that these modifications haven't affected the results in a way which is not reflected by any actual change in the effectiveness of the algorithm.

I followed this principle in implementing CSG, however, the algorithms I compare it to have modified the GPPC code for various reasons.

[1] https://movingai.com/GPPC/

5.2 Benchmarks

The MovingAI benchmarks for Grid Based Path Planning include maps taken from a number of video games, specifically Dragon Age: Origins, Dragon Age 2, Warcraft III, Baldur's Gate II, and Starcraft [Sturtevant, 2012]. Since these maps were taken from video games, they are quite typical maps of the sort that you are likely to encounter in real-world applications of this pathfinding algorithm to games. This is good; when evaluating an algorithm, it's best for the test scenarios to be as close as possible to the actual scenarios the algorithm would be likely to run on in practice. For this reason, these maps, taken from video games, are the maps that I choose to evaluate grid-based path planning methods on.

There are some other maps in the dataset. Some of these maps are quite unnatural and unlike the kinds of maps you are likely to encounter in practice. There are maps that consist of random noise, maps that consist of a number of large, equally sized, perfectly square rooms with holes in the walls, and maps that are mazes. These are unlike the kinds of maps that I would expect to find in real-world applications, so I ignore them in my experiments. Note that if you were to know you were going to be running your algorithm on one of these specific types of problems, it would be possible to design algorithms that behave particularly well on these types of problems, but not so well on the types of problems that are typical in games.

The Baldur's Gate II maps are available in two sizes: their original size, and scaled to 512x512. I chose to use the maps in their original size. Scaling maps changes their properties and creates large, open spaces, so that it no longer reflects what the original map was like. While scaling the maps may make the problem more difficult, it also means that they lose some of their applicability to real world situations, since the modified maps are different in nature to the actual maps. This is why I chose to use the maps in their original size.

The MovingAI benchmarks for Grid Based Path Planning also include scenarios for each of the maps. Each scenario consists of a number of pathfinding queries, and the optimal distance required to solve each query. The GPPC code runs each of these queries and tells you some statistics with regard to the solving of that query. Most importantly, it tells you the total time taken to solve each query, the total length of the returned path, how suboptimal that path is with respect to the optimal distance listed in the scenario file, and whether or not the returned path is valid.

5.3 Testing for Correctness

It is important to test that the implementation correctly returns optimal, valid paths. One method used to test this was to run the test scenarios using the GPPC code, and check that each returned path was "valid", and that the "suboptimality" of each path was 1.

A path is considered **valid** according to the GPPC code if the following conditions hold:

1. The start of the path is the same as the source of the query
2. The end of the path is the same as the goal of the query
3. Each step in the path causes a change in the x and y values by at most 1
4. No part of the path contains an obstacle
5. No corner cutting occurs

A path is also automatically considered valid if it is the empty path.

According to the GPPC code, the **suboptimality** of a path is equal to the length of the path returned by the algorithm divided by the expected length of the optimal path between the source and the goal.

Due to differences in rounding between the GPPC code and the scenario files, the distance calculated for an optimal path by the GPPC code is not exactly the same as the distance stored in the corresponding query in the scenario files. This means that the suboptimality of a path will usually not be exactly 1, even if the path is optimal. In practice, we assume that a query is **optimal** if the suboptimality of the query is between 0.999990 and 1.000001. Note that more leeway is given below the value of 1 than above the value of 1, since the GPPC code rounds the value of $\sqrt{2}$ downwards to 1.4142, which is more rounding than is applied in the scenario files.

For some queries in some scenarios, the source and goal were at the same point. In this case, the GPPC code calculates the suboptimality of the path to be -nan. These queries can be ignored, since it is trivial to find an optimal path when the source and goal are the same point.

We can be reasonably confident that a query has been answered correctly if the returned path is both valid and optimal, according to the definitions given above.

For every query in every scenario file for every map in our dataset, the Complete Corner Graph implementation returned a valid, optimal path.

Further Testing by Comparing to Dijkstra's algorithm Checking that each query in each scenario is solved correctly is not by itself sufficient to be confident in the correctness of the algorithm. The number of queries in the scenario files is limited, and does not necessarily reflect the full range of possible queries that could be given to the algorithm.

In order to have more confidence in the implemenation, I also checked for correctness by randomly generating a large number of queries and comparing the lengths of the paths returned by the Complete Subgoal Graph implementation to the lengths of the paths returned by a simple implementation of Dijkstra's algorithm. Since the implementation of Dijkstra's algorithm is simple, it has less likelihood of bugs, and the Complete Subgoal Graph and implementation of Dijkstra's algorithm were written independently and are unlikely to have the same kinds of bugs. Thus, ensuring that the paths returned by each algorithm have the same length helps us to be more confident that the CSG algorithm was implemented correctly.

5.4 Performance

CSG is designed to improve on the performance of pathfinding algorithms on octile grids. Therefore, it is vitally important to evaluate the performance of CSG, and to compare it to other similar algorithms. In particular, it is designed to be an ultra-fast pathfinding algorithm, so we want to compare it to other algorithms that have extremely fast runtime, but relatively large preprocessing times and space required for preprocessing data.

There are two different ways in which these algorithms can be compared. We can compare them theoretically using asymptotic analysis, or we can compare them practically through experiments. Comparing them theoretically has the advantage of providing more insight to the fundamental reasons why an algorithm behaves quickly or slowly. Comparing them practically has the advantage of comparing them under practical conditions, rather than comparing their performance as the amount of data tends towards infinity.

5.4.1 Asymptotic Analysis in the Special Case of Game Maps

It is possible to perform asymptotic analysis on arbitrary maps. However, this algorithm is designed specifically for the problem of pathfinding in the types of maps typically found in games. It is not useful for it to be able to perform well on strange, pathological maps that the algorithm will never be used on in practice. There are a few things we should take note of when performing asymptotic analysis in the special case of game maps.

In asymptotic analysis, we aim to determine what happens to the performance characteristics as the problem size increases towards infinity. However, there are multiple ways of increasing the size of a map. For example, it is possible to increase the size of a map by simply performing scaling on the map. However, it is highly unlikely that in practice, a larger map will be produced by scaling a smaller map up. In game maps, it is much more likely that a map would be enlargened by adding new rooms and corridors to the map. This has a markedly different impact on the map. Scaling the map up will widen rooms and corridors, without increasing the overall complexity of the map. Adding new rooms and corridors will typically create a winding, complex labyrinth, while keeping the rooms and corridors of a similar size. When performing our asymptotic analysis, we will assume that the way in which the problem size inceases is by adding new rooms and corridors to the map.

Some properties are local to a specific area of the map. That is, a specific area of the map has a certain property, and changing parts of the map far away from that area will not change that property. A very simple example of a local property is the property that a certain point is a convex corner. Changing points next to that point may change whether or not that point is a convex corner, but changing points far away from that point will not.

These properties are interesting with regards to asymptotic performance because as the size of the map increases, these properties remain constant. That is, they are

$O(1)$ in the size of the map. We can use this to help bound the asymptotic time complexity of certain algorithms.

Consider the number of points that are safe-reachable from a certain point. In general, this is not necessarily a local property. It may be the case that a point is safe-reachable from a point on the opposite side of the map. However, in practice, it is very rare that this will occur. Usually, the points safe-reachable from a given point will be the points near that point. Usually, adding more rooms and corridors to the map will not affect the set of points safe-reachable from that point. Therefore, it makes sense to assume that the number of points safe-reachable from a given point is a local property, and that increasing the size of the map by adding rooms and corridors will not increase this number.

5.4.2 Effect of Reachable Subgoal Data on Asymptotic Performance

One of the pieces of data that is computed and stored by CSG is the set of all subgoals safe-reachable from each point. We ask ourselves what impact storing this data has on the performance of the algorithm.

Under the assumption that the number of points safe-reachable from a certain point is a local property, and hence it remains constant as the map is expanded, it follows that the total memory usage and preprocessing time used to compute the reachable subgoals are both linear in the number of nodes. The maps themselves are already typically stored in a form which requires a linear amount of memory, so adding an additional linear memory/preprocessing time requirement isn't a very big deal.

5.4.3 Effect of Complete Subgoal Graph on Asymptotic Performance

The other piece of data that CSG needs to compute is the complete subgoal graph. We ask ourselves what the effect of storing the complete subgoal graph is on performance.

Without compression, the amount of space taken by the distances and first corners in the complete subgoal graph is quadratic in the number of subgoals. It would be possible to compress the table of first corners in a similar manner to what is done for CPDs, however, it is not so easy to compress the table of distances.

At the outset, it is unclear whether this is a good result or bad result, since it is unclear how many subgoals there are in a typical game map. If, for example, the number of subgoals was the square root of the number of nodes in the map, then this would only take linear memory. However, if the number of subgoals is linear in the number of nodes in the map, then this takes quadratic memory in the number of nodes, which is signicantly worse.

Unfortunately, under the assumption that subgoals are defined based on local characteristics, the number of subgoals is linear in the number of nodes in the map, in which case the complete subgoal graph requires quadratic memory in the number of nodes in the map. In general, adding a room or corridor to the map will add a

certain number of nodes to the map and will add a certain number of subgoals to the map. Doing this repeatedly will increase the number of nodes at a linear rate, and will also increase the number of subgoals at a linear rate. As a result, the number of subgoals is linear in the number of nodes.

In order to construct a CSG, Dijkstra's algorithm has to be run over the subgoal graph, starting from every subgoal. In practice, this takes less preprocessing time than other ultra-fast algorithms, such as Topping or Topping+. Topping and Topping+ both need Dijkstra's algorithm to be run over the entire graph, although Topping+ only needs to start the algorithm from every jump point. CSG avoids having to run Dijkstra's algorithm over the entire graph, and instead only needs to run it over the subgoal graph.

5.4.4 Runtime Performance

It's also important to analyse the runtime performance of the algorithm. CSG removes all search from SG, so we expect it to have good runtime performance.

Checking if the goal is reachable from the source takes at most linear time in the size of the map. Under the assumption that the set of points reachable from a given point is a local characteristic, it only takes constant time in the size of the map.

Finding the pair of subgoals to travel through takes quadratic time in the average number of subgoals reachable from a point, and is constant in the size of the map assuming that the number of subgoals reachable from a point is a local characteristic.

Travelling over the subgoal graph takes linear time. Furthermore, it consists entirely of retrieving the next step followed by travelling over the next step. No search is involved, just path retrieval.

5.5 Experimentally Measuring Performance

For the purpose of measuring performance, the maps were grouped according to the game they were taken from. This makes sense because maps from the same game tend to be similar in nature to other maps from the same game, and different in nature to maps from other games. Some amount of grouping is useful, due to the sheer number of maps; the dataset contains 454 different maps, so it would be difficult to read the data if it was taken on a map-by-map basis.

Some basic information about the maps and scenarios associated with each game is displayed in Table 5.1. This includes the number of maps in each dataset, the total number of queries contained in all scenarios for each dataset, the average optimal path length for a query in each dataset, and the average number of nodes in a map. We can see that the Baldur's Gate II maps were the smallest on average, whereas the StarCraft maps were the largest. Dragon Age: Origins had relatively long paths, despite the fact that the maps had only a modest number of nodes.

Table 5.2 tells us the prepocessing and runtime statistics for CSG. In order to obtain this data, the GPPC code was used to preprocess each of the maps and run all the scenarios. For each group, the time taken to preprocess all maps was measured,

Map Type	# Maps	# Queries	Avg. Path Length	Avg. # Nodes
Baldur's Gate II	120	40780	142	4507
Dragon Age: Origins	156	155620	418	21323
Dragon Age 2	67	67200	281	15911
Warcraft III	36	55360	318	112488
StarCraft	75	211390	613	263782

Table 5.1: Characteristics of Scenarios/Maps

Map Type	Runtime	Preprocessing	Memory
Baldur's Gate II	3.3μs	4m35s	457MB
Dragon Age: Origins	5.7μs	21m24s	2000MB
Dragon Age 2	4.2μs	2m26s	303MB
Warcraft III	4.7μs	4m21s	609MB
StarCraft	7.9μs	6h19m18s	19700MB

Table 5.2: Performance Characteristics of CSG

the average time taken to answer a pathfinding query was calculated, and the total filesize of the stored preprocessing data was recorded. CSG did not manage to successfully preprocess the Starcraft map "The Frozen Sea", due to its large size. This map is not included in the statistics for this algorithm.

I computed the same pieces of data for Topping. I used the code written by Salvetti et al. [2018]. The results are in Table 5.3. The Topping implementation had significantly modified the GPPC code. I make the assumption that these modifications do not substantially impact the results. Due to the long preprocessing time for the Starcraft maps, these maps were not all preprocessed at once, rather, they were preprocessed in several different chunks. This involved stopping and restarting computation at several points, which may have caused some computation to be lost. Topping also did not manage to successfully preprocess the Starcraft maps "The Frozen Sea", "Cauldron", "Expedition", or "PrevevalIsles", due to their large size. The results for these maps are not included in the statistics for this algorithm.

I also computed these pieces of data for Topping+. I used the implementation

Map Type	Runtime	Preprocessing	Memory
Baldur's Gate II	4.2μs	14m59s	2MB
Dragon Age: Origins	9.1μs	3h01m47s	2000MB
Dragon Age 2	6.3μs	21m54s	420MB
Warcraft III	Error	10h11m11s	4900MB
StarCraft	17.7μs	6d07h43m40s	24100MB

Table 5.3: Performance Characteristics of Topping

Map Type	Runtime	Preprocessing	Memory
Baldur's Gate II	25.7 μs	8m42s	20MB
Dragon Age: Origins	59.7μs	2h25m32s	175MB
Dragon Age 2	37.0 μs	13m49s	56MB
Warcraft III	52.2 μs	2h36m50s	105MB
StarCraft	109.8 μs	4d07h05m26s	1025MB

Table 5.4: Performance Characteristics of Topping+

written by Hu et al. [2021]. The implementation did not appear to use the GPPC code. I believe that the "C-S-R Time" from this implementation corresponds to the "total-time" from the GPPC code, and that the "C-S-R Time" is measured in microseconds, whereas the "total-time" is measured in seconds. I make the assumption that these pieces of data are comparable after converting them so that the units match.

Unfortunately, when I ran the experiments for Topping+, I was unable to get it to answer queries as fast as expected. This was also the case when I tried using their Topping implementation instead of their Topping+ implementation, so I expect this may be an implementation problem, rather than a problem with the algorithm itself. Nevertheless, the memory improvement that Topping+ provides over Topping is impressive, so I felt it would be worth including Topping+ in my performance comparison anyway.

When running the experiments for Topping+, I had to preprocess the StarCraft maps in chunks, similar to in the case of Topping. This may have caused some computation to be lost.

I also computed the same pieces of data for JPS+BB. I used the implementation by Rabin and Sturtevant [2016]. This was initially written for Windows, and my computer runs Ubuntu. Therefore, in order to run it on the same machine as was used for the rest of the experiments, some modifications to the code were made. When doing so, I took the opportunity to standardise it by making it run with the original GPPC code. It is possible that these modifications had an impact on the algorithm, but it appears to be running as expected.

Unfortunately, I didn't start the experiments early enough to complete the experiments for JPS+BB on Starcraft maps. We would expect this to take approximately 6-7 days of computation. This is because JPS+BB takes approximately the same amount of computation as Topping does, since both algorithms need to find the optimal path from every point in the map to every other point. Notice that the preprocessing times that I did find for JPS+BB were similar to the corresponding times for Topping.

The amount of time taken by these algorithms may vary depending on the machine used. Table 5.6 shows the most important characteristics of my machine.

Map Type	Runtime	Preprocessing	Memory
Baldur's Gate II	4.5μs	11m02s	41MB
Dragon Age: Origins	7.8 μs	4h00m25s	256MB
Dragon Age 2	5.6 μs	34m48s	82MB
Warcraft III	2.9μs	12h17m18s	317MB
StarCraft	Unknown	Unknown	Unknown

Table 5.5: Performance Characteristics of JPS+BB

Operating System	Ubuntu 18.04.2 LTS
Model Name	Duo E8400
Clock Speed	3.003GHz
Cores	2
L1d Cache	32KB
L1i Cache	32KB
L2 Cache	6144KB
L3 Cache	None

Table 5.6: Characteristics of my Machine

5.6 Evaluation of Experimental Results

Figure 5.1 compares the online runtimes of the algorithms I tested. I didn't include Topping+ in the runtime plot because it ran significantly slower than the other algorithms, and so including it in the plot would make it harder to compare the runtimes of the other algorithms.

We can see that CSG has a runtime that is better than or comparable to the other algorithms that were tested. The only instance where another algorithm beat CSG on runtime was in the case of JPS+BB on Warcraft III maps.

Figures 5.2 and 5.3 compare the preprocessing times of the algorithms. I put the data for StarCraft maps in their own plot because StarCraft maps took significantly longer to preprocess than the other maps, and so including them in the same plot would have required scaling that would have made it harder to compare the algorithms on other sets of maps.

We can see that CSG requires much less preprocessing time than the other algorithms. This is especially noticeable in map sets which required a large amount of preprocessing, such as Warcraft III or StarCraft maps.

Figures 5.4 and 5.5 compare the memory requirements of the algorithms. I put the StarCraft maps in their own plot for essentially the same reasons as in the plots of preprocessing times.

Unfortunately, we can see that the memory requirements for CSG are quite high compared to other algorithms. However, they are similar memory requirements to the requirements for Topping, which is an indication that they are within reason.

In summary, compared to the other algorithms that were tested, CSG has a better

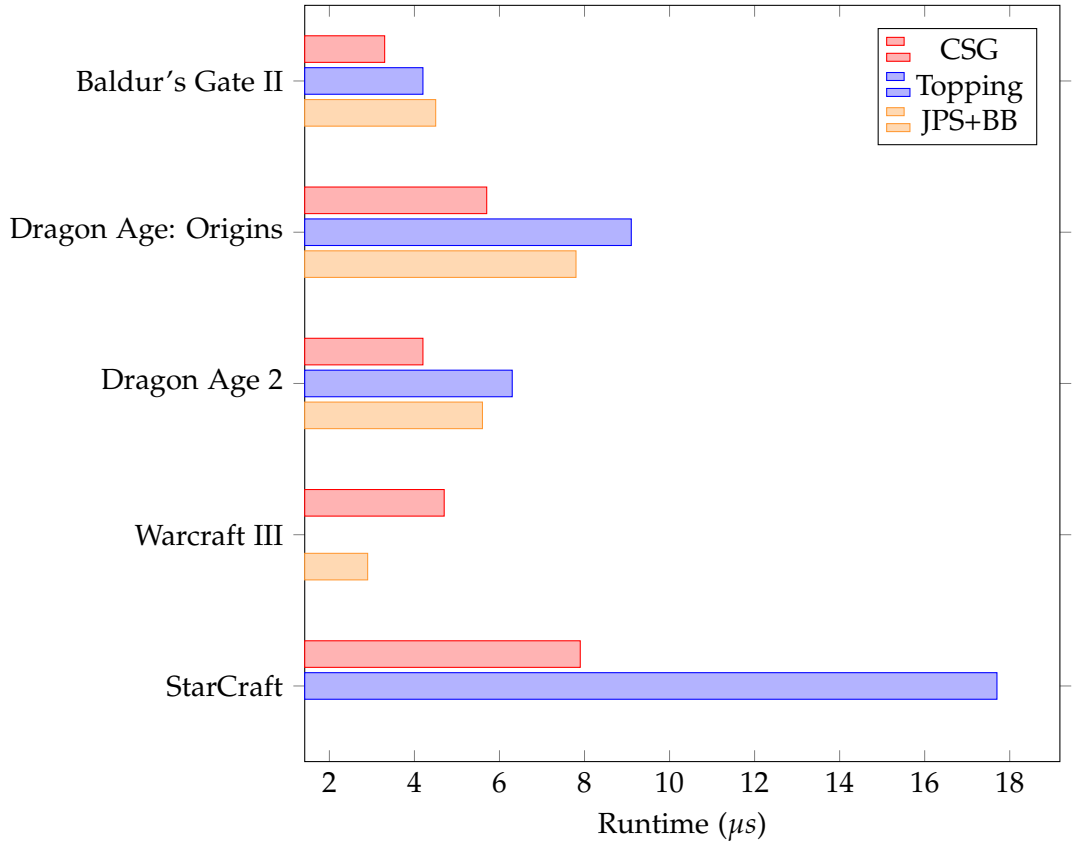


Figure 5.1: Comparison of the online runtimes of different algorithms.

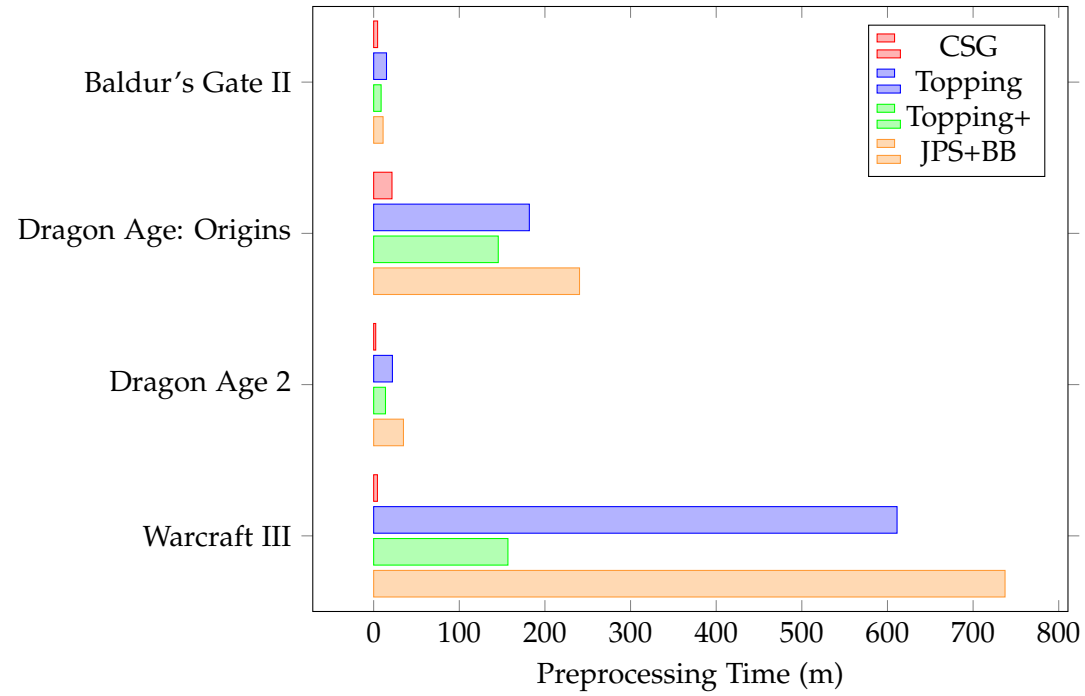


Figure 5.2: Comparison of the preprocessing times of different algorithms on non-StarCraft maps.

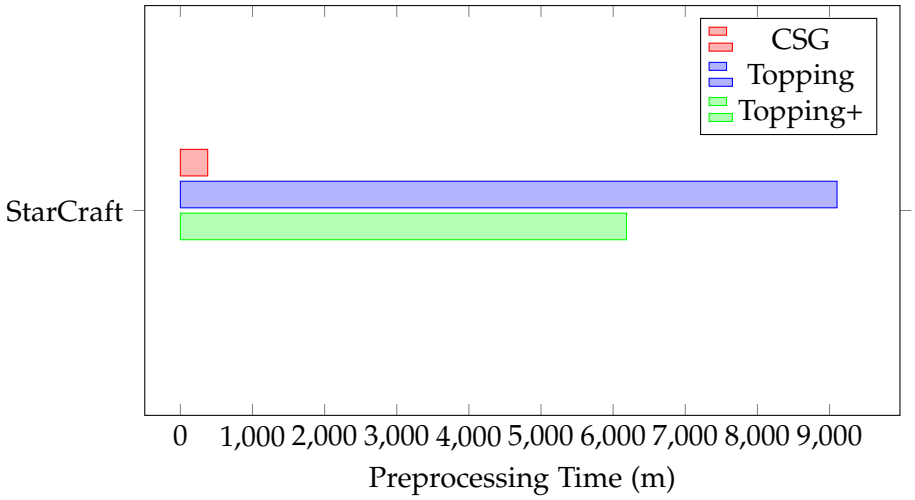


Figure 5.3: Comparison of the preprocessing times of different algorithms on StarCraft maps.

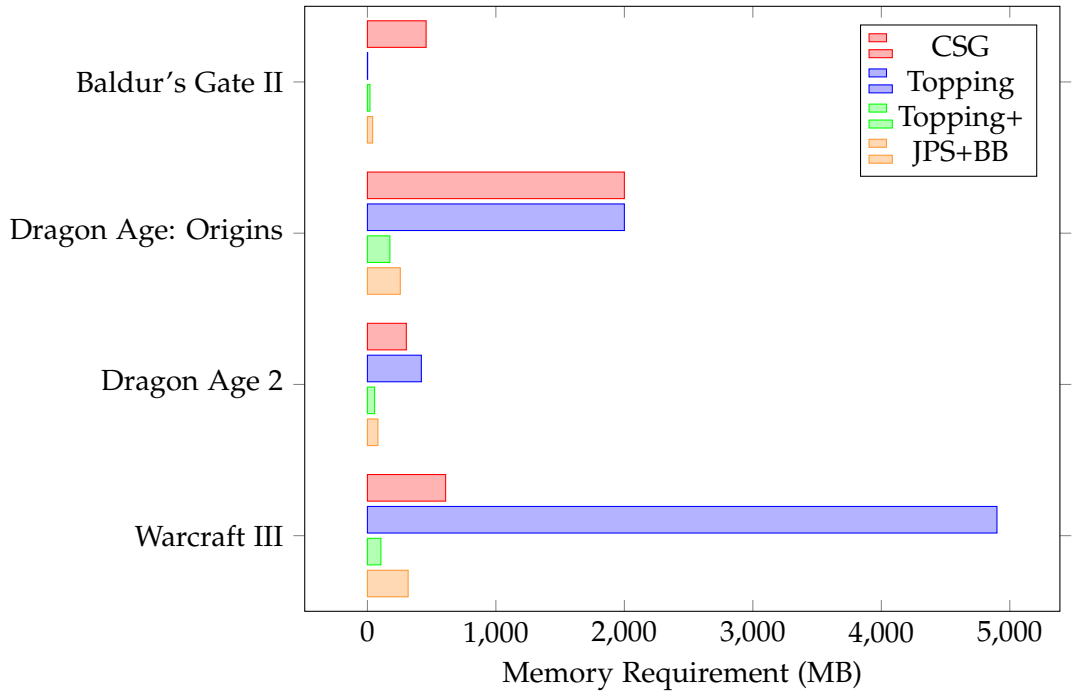


Figure 5.4: Comparison of the memory requirements for different algorithms on non-StarCraft maps.

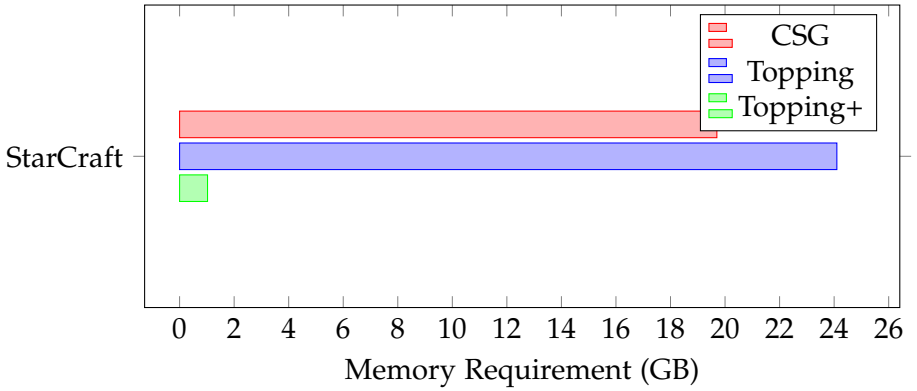


Figure 5.5: Comparison of the memory requirements for different algorithms on StarCraft maps.

Map Type	Avg. # Subgoals	Avg. (# Subgoals)^2	Avg. # Non-obstacles
Baldur's Gate II	449	573219	4507
Dragon Age: Origins	896	1874420	21323
Dragon Age 2	598	537714	7088
Warcraft III	1193	1627820	112488
StarCraft	5453	39385200	263782

Table 5.7: Non-obstacles vs Subgoals

or comparable online runtime, much better preprocessing times, but significantly higher memory requirements.

5.7 Comparison of Number of Subgoals to Number of Non-Obstacles

The size of the complete subgoal graph is quadratic in the number of subgoals. At the outset, it isn't really clear whether this is a good result or a bad result, since it isn't obvious how many subgoals there will usually be in a map. In particular, I wanted to see how algorithms that are quadratic in the number of subgoals compare to algorithms that are linear in the number of non-obstacles in the map.

We have already discussed how theoretically, assuming that subgoals are a local property, the number of subgoals should be linear in the number of non-obstacles, in which case the square of the number of subgoals will be quadratic in the number of non-obstacles. However, I wanted to see how these results held up in practice.

Table 5.7 gives us the information required to make this comparison. It tells us the average number of corners, the average number of corners squared, and the average number of obstacles for each game.

We can see that the number of subgoals squared is significantly higher than the number of non-obstacles. This tells us that we should expect algorithms that take quadratic time/memory in the number of subgoals to take more time/memory than algorithms that take linear time/memory in the number of non-obstacles. Of course, this will depend on what maps specifically we run our algorithms on. In maps with a particularly small number of subgoals, the opposite may be true.

5.8 Identifying Areas for Improvement

It is important to know which parts of the algorithm are the most costly in terms of performance. Table 5.8 compares the components of CSG in terms of the amount of time spent during the online stage, whereas Table 5.9 compares the components of CSG in terms of the amount of memory required to store the preprocessing data, and Table 5.10 compares the components of CSG in terms of the amount of preprocessing time required to preprocess each component.

Map Type	Octile	Double	Single	Compute
Baldur's Gate II	11	36	7	45
Dragon Age: Origins	7	29	6	57
Dragon Age 2	7	24	6	63
Warcraft III	9	34	10	47
StarCraft	Unknown	Unknown	Unknown	Unknown

Table 5.8: Distribution of Time Spent (Percentage)

Map Type	Corners	Safe-Reachable Corners	CSG	BB
Baldur's Gate II	<1	8	90	2
Dragon Age: Origins	<1	13	86	1
Dragon Age 2	<1	27	71	2
Warcraft III	<1	41	58	1
StarCraft	<1	11	89	<1

Table 5.9: Distribution of Memory Usage (Percentage)

Map Type	Corners	Safe-Reachable	CSG	Relevant	BB	Pushing
Baldur's Gate II	<1	<1	26	1	49	24
Dragon Age: Origins	<1	1	28	1	46	24
Dragon Age 2	<1	1	21	3	48	26
Warcraft III	<1	4	18	7	44	28
StarCraft	<1	1	35	1	33	31

Table 5.10: Distribution of Preprocessing Time (Percentage)

In Table 5.8, the "Octile" stage refers to when we check if the goal is immediately reachable from the start. The "Double" stage refers to when we search through pairs of distinct subgoals to find the best choice for the first and last subgoal on the path. The "Single" stage refers to when we search through subgoals to work out if a single subgoal should be both the first and the last subgoal on the path. The "Compute" stage refers to when we construct the path from the start to the goal by travelling over the subgoal graph.

The labels in the other tables are mostly self-explanatory except for the "Pushing" label in Table 5.10, which refers to the amount of time taken to "Push" the subgoals in the first-move-table so that each step travels as far as possible.

The majority of online runtime used by CSG is in the compute stage of the algorithm, with a significant amount of time also being taken in the double stage of the algorithm. This indicates that if we want to reduce the runtime of the algorithm, it might be a good idea to focus on reducing the amount of time taken to construct the path. However, the path construction is extremely basic, consisting purely of looking up the next subgoal on the path and adding all the points on the diagonal-first path between the current subgoal and the next subgoal. This makes it hard to imagine what kinds of improvements can be made here. We might also want to consider improving the "Double" stage, as this also takes up a considerable portion of the time.

We can see from Table 5.9 that the vast majority of memory taken by the algorithm is taken by the complete subgoal graph, although a significant amount of memory is also taken by the map from positions to safe-reachable corners, expecially in the case of Warcraft III maps. This indicates that if we were to want to reduce the memory requirements for CSG, we should probably focus on reducing the size of the complete subgoal graph. In particular, since the complete subgoal graph takes quadratic memory in the number of subgoals, one likely method for achieving such a memory improvement would be by somehow reducing the number of subgoals that need to be stored in the complete subgoal graph.

Most of the preprocessing time is spent calculating the bounding boxes and calculating the complete subgoal graph, although a significant amount of time is also spent pushing the next subgoal table to ensure that each step travels as far as possible. If we did want to reduce the preprocessing time further, it would make sense to focus on making a change to one of these areas. However, the preprocessing time is already a strength of CSG, so it might be a better idea to focus on one of CSG's weaknesses instead.

Chapter 6

Reducing Memory Requirements

One of the main drawbacks of the CSG algorithm is its large memory requirement. Ideally, this is something that should be improved upon in future work. This chapter describes some of the work in progress that I have performed towards achieving this goal.

These methods focus on reducing the number of subgoals in the complete subgoal graph. As we saw in Section 5.8, most of the memory required by CSG is in the complete subgoal graph, which takes quadratic memory in the number of subgoals. Thus, if we could reduce the number of subgoals, we could drastically reduce the memory requirements for this algorithm.

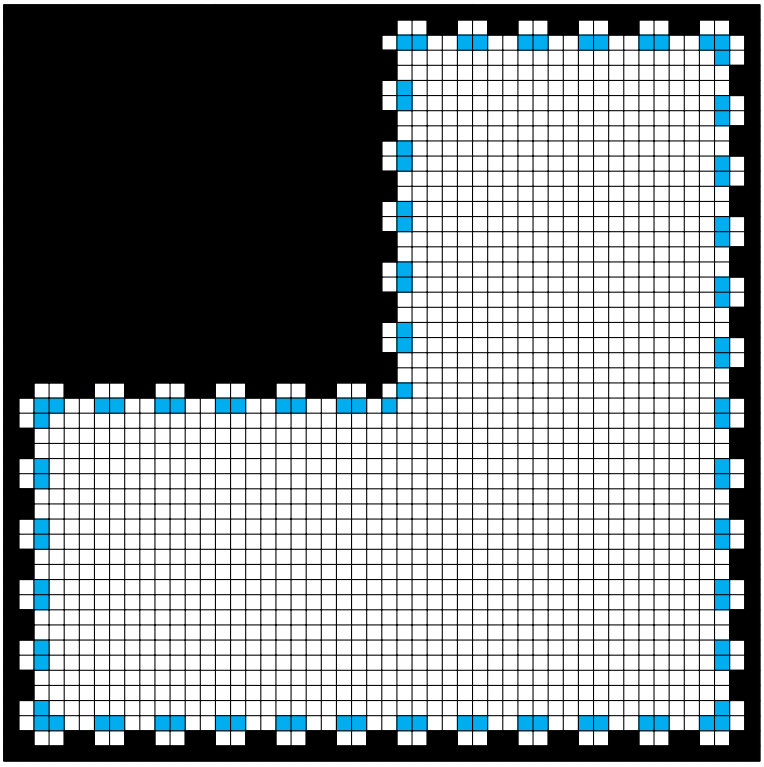
6.1 Smoothing the Graph

Many of the maps in the dataset have quite rough edges. These rough edges create a large number of subgoals. If we could smooth out these edges, it may be possible to reduce the number of subgoals in the graph, and thus reduce the size of the complete corner graph.

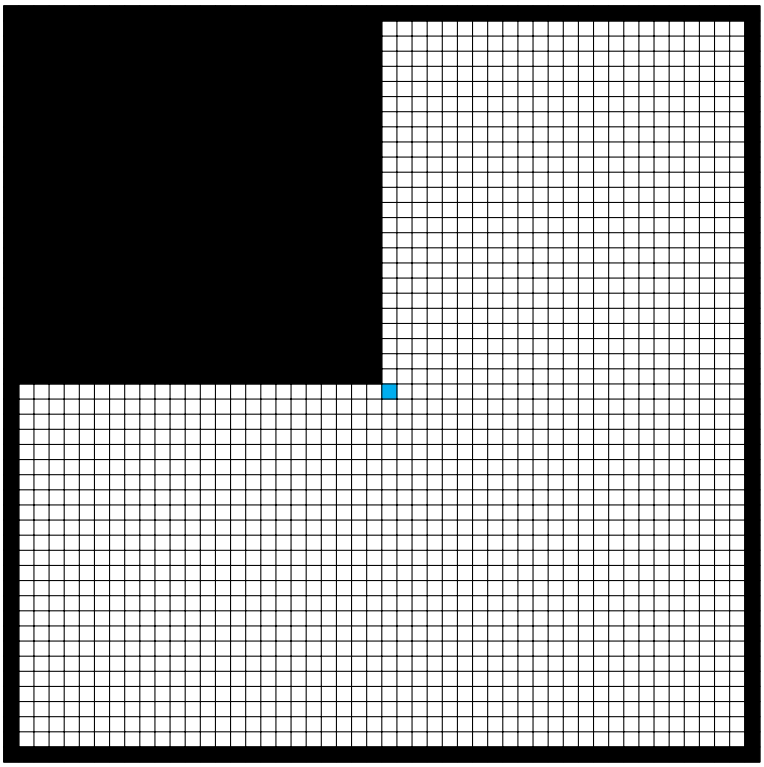
I've created two example maps to illustrate what I mean. Consider Figures 6.1(a) and 6.1(b). Notice how the graphs are extremely similar to each other. However, the number of subgoals in the graph with the jagged border is far, far, higher than the number of subgoals in the graph with the smooth border. This means that if we could somehow transform a map with a jagged border into a map with a smooth border, then the CSG algorithm would require far less memory and preprocessing time to precompute and store the complete subgoal graph.

When we smooth the graph, it will no longer be exactly the same map, so the optimal paths will not necessarily be the same as in the original map. To keep the optimal paths in the two maps as similar as possible, the methods I use to smoothen graphs only add obstacles, without taking away any obstacles, and they make sure that if an obstacle is added, then that point is not on the optimal path between any pair of points that haven't been turned into obstacles. As a result, all optimal paths on the modified map are also optimal paths on the original map, although there are some optimal paths on the original map that are not contained in the modified map.

It will still be necessary to answer queries involving points that were removed



(a)



(b)

Figure 6.1: A comparison between the number of subgoals in a map with a very jagged border to the number of subgoals in a map with a smooth border.

during the smoothing of the graph. I have not yet implemented a method to do this, but I believe this should be possible with more work. I present the current work that I have done, under the understanding that more work may be necessary to make it effective in practice.

One method for smoothing the graph is what I call **straight smoothing**. Straight smoothing consists of the following steps:

1. Choose a convex corner. Consider an intercardinal direction d_0 in which there is an obstacle that is causing this point to be a corner. Choose one of the cardinal directions adjacent to this direction, call this direction d_1. Step in this direction. See Figure 6.2(a) for an illustration of this step.

2. Consider the direction d_2 which is perpendicular to d_1 and facing away from d_0. Step in the direction d_2 until either you hit a wall, or stepping in the opposite direction to d_1 would cause you to hit a wall. Create an obstacle at each location you travelled through during this process. See Figure 6.2(b) for an illustration of this step, and see Figure 6.3 for an example of when the second stopping criterion is met.

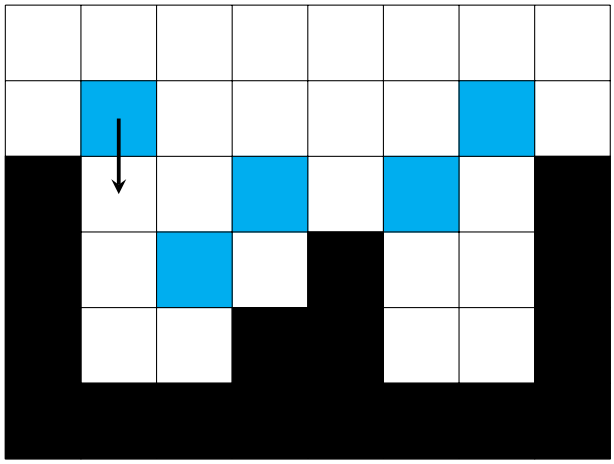
3. Start a flood fill of obstacles in all of the positions in the direction d_1 of each of the obstacles that were just added (see Figure 6.2(c)).

4. For every point added in this process, update whether or not the surrounding points are corners.

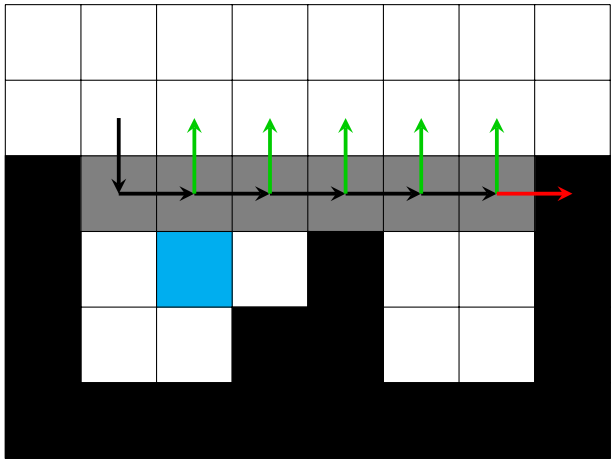
Straight smoothing is not guaranteed to give good results, depending on the corner that was chosen and the direction in which the smoothing was performed. In the example in Figure 6.4, straight smoothing causes every point in the graph to be turned into an obstacle (assuming the graph is connected). This is clearly unhelpful, as there will be no optimal paths in the resulting graph, meaning that it will tell us no information about the optimal paths in the original graph.

Therefore, after performing straight smoothing, the algorithm needs to work out whether the smoothing was beneficial for the graph or not. If the smoothing was beneficial, then the algorithm keeps the modified graph. If the smoothing was not beneficial, then the algorithm reverts the map to how it was before. In general, we want to find a balance between reducing the number of subgoals and hence reducing the memory requirement for the complete subgoal graph, and minimizing the number of obstacles added so that the modified graph is as similar as possible to the original graph.

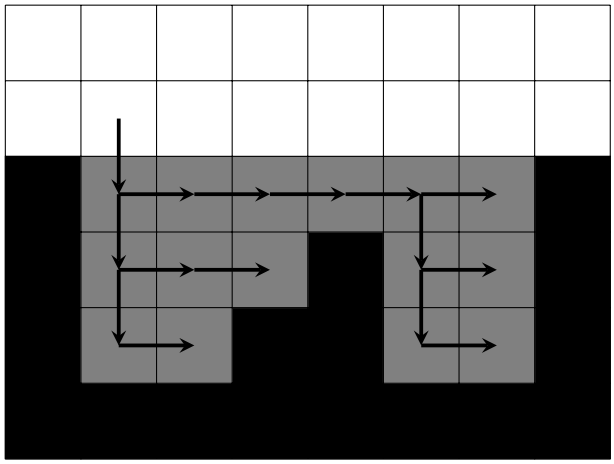
Currently, the method used to determine whether or not the smoothing was beneficial is to check whether or not the ratio between the number of added obstacles and the number of removed corners is greater than some number, n. My algorithm currently has n arbitrarily set to 10. This is still a reasonably rough heuristic for deciding whether or not the straight smoothing was worth it, and I expect better heuristics are possible.



(a)



(b)



(c)

Figure 6.2: The straight smoothing process

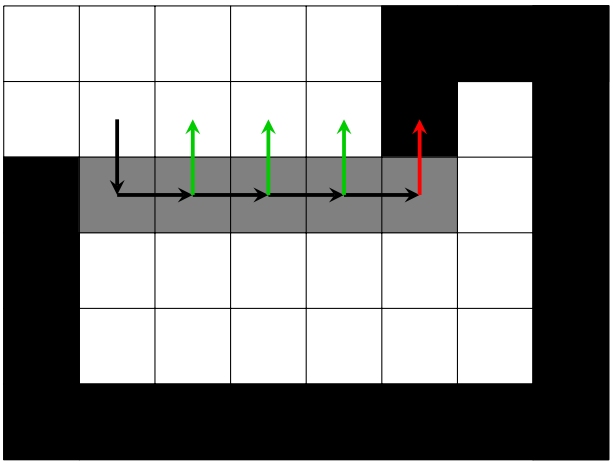


Figure 6.3: Example of the second end condition for step 2 of the straight smoothing process being met.

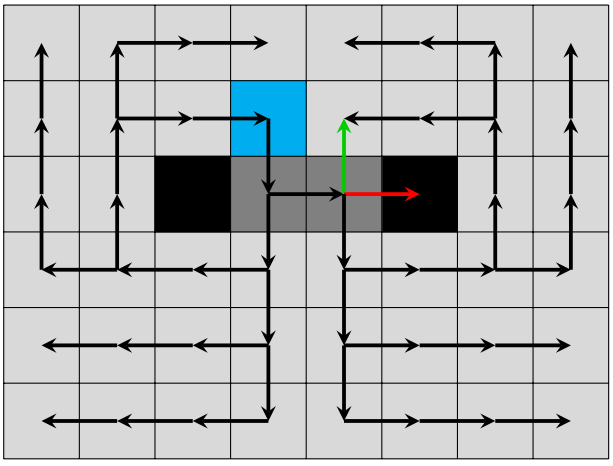


Figure 6.4: Example of a situation in which straight smoothing performs very poorly.

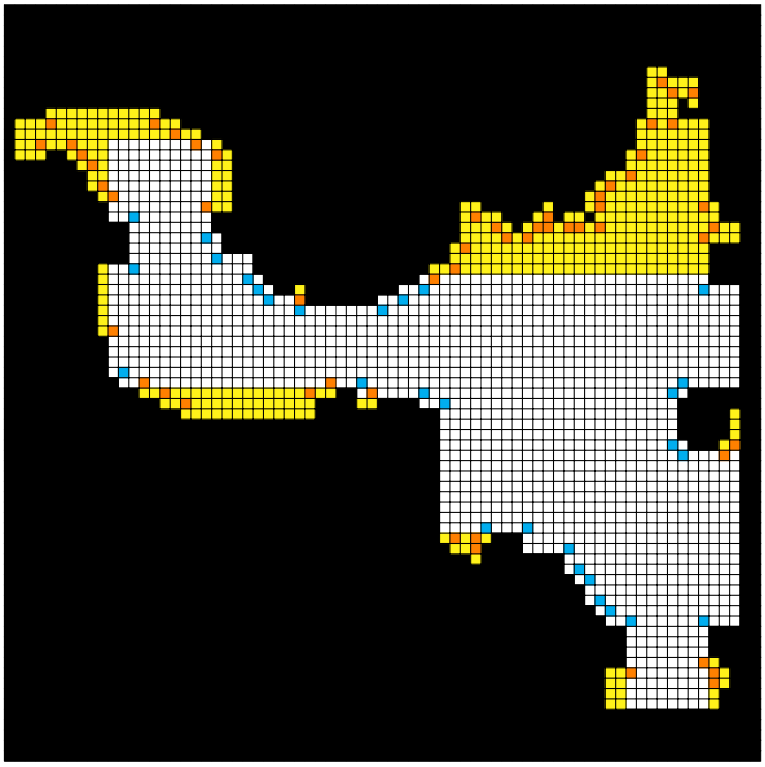


Figure 6.5: Straight smoothing in practice

Straight smoothing is attempted once for every subgoal and every choice of d_0 and d_1, and the algorithm decides whether to keep or revert each of these modifications according to the heuristic described above.

Figure 6.5 indicates how straight smoothing might look when applied to an actual graph.

Another method for this kind of smoothing is what I call **diagonal smoothing**. When corner-cutting is disallowed, diagonal smoothing doesn't work as well, but it is simpler when corner-cutting is allowed, which is the problem I originally used this kind of smoothing on. I'm going to discuss diagonal smoothing in the context of corner cutting being allowed.

The idea is very similar to the idea of straight smoothing: from a subgoal, a diagonal path of nodes is created, and all nodes behind this path of nodes are filled in. This allows us to remove some near-convex-corners from the graph. This is illustrated in Figure 6.6.

Figure 6.7 illustrates why this doesn't work so well in the case where corner cutting is not allowed. In this case, diagonal smoothing will not remove the convex corner, it will create new convex corners along the diagonal path of nodes, and due to the corner cutting, creating this path of nodes will change the optimal distance between some nodes in the modified graph.

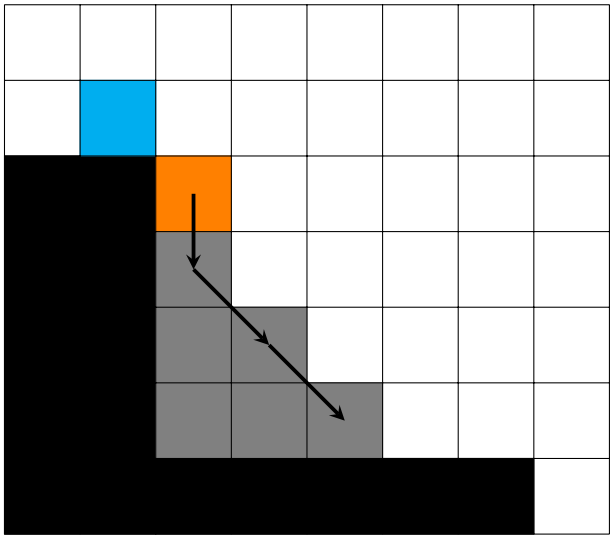


Figure 6.6: Diagonal smoothing when corner cutting is allowed

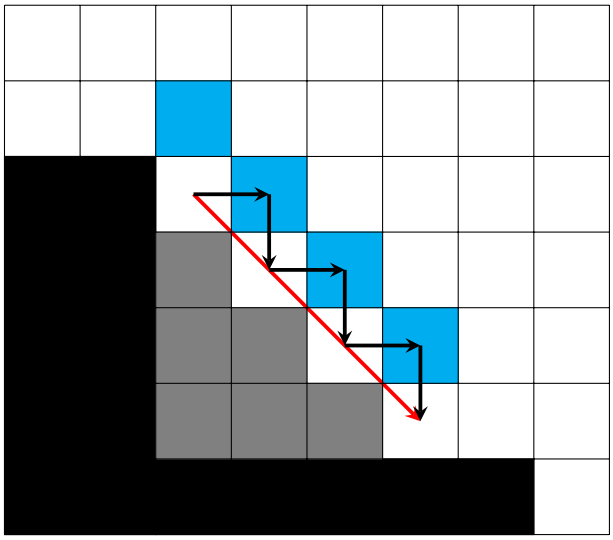


Figure 6.7: Attempted diagonal smoothing when corner cutting is not allowed

6.2 Removing Degree One Subgoals from the CSG

Another potential method that could be used to reduce the amount of memory required by CSG is to borrow the idea of N-level subgoal graphs [Uras and Koenig, 2014], and only include subgoals in the CSG if they have a level of at least k, for some k. In particular, the case $k = 2$ seems interesting.

Travelling over the CSG would work as normal, since it is never necessary to travel over subgoals of a lower level in order to travel between subgoals of a higher level. However, connecting to the CSG would need some modification, since it may be necessary to travel over subgoals of a lower level in order to reach the CSG. It may also be necessary to travel over exclusively subgoals of a level less than k to get from the start to the goal.

I have not implemented this idea, but it seems to have potential.

Chapter 7

Conclusion

I have presented an algorithm for ultra-fast pathfinding on octile grids that performs slightly faster than existing algorithms in terms of online runtime, and drastically cuts down on the time required for preprocessing. The biggest drawback to this method is that it requires memory that is on the upper edge of the memory required by existing algorithms. I have also discussed some potential methods for removing this weakness by reducing the memory requirements for this algorithm.

My method means it is no longer necessary to wait for inordinately long times for preprocessing to finish if you want to be able to answer pathfinding queries with speeds comparable to the fastest currently available. Where existing methods may take hours to finish preprocessing, my algorithm takes minutes, and where existing methods take days, my algorithm takes hours. This may be particularly useful during quick prototyping, where changes are common and you don't want to wait a long time every time you make a change. On the other hand, for a final release, it might make sense to use another algorithm, because if changes to the map aren't likely, it makes sense to invest more time in reducing the amount of memory that is required.

7.1 Future Work

The part of CSG that needs the most improvement at the moment is its memory requirements. Perhaps one of the potential improvements I proposed in Chapter 6 could be ironed out and implemented in CSG, or perhaps some other improvement will prove to be more effective.

CSG could also use some improvement in its online runtime. For example, perhaps it is somehow possible to further reduce the number of pairs of corners we need to check in order to find the first and last corners on the optimal path.

Also, it would be interesting to see if CSG could be applied to problems other than pathfinding on octile grids. In theory, it is reasonable to believe that CSG should be applicable to any graph which can have a subgoal graph constructed on it, although some of the runtime improvements may need modifying.

Bibliography

Botea, A., 2011. Ultra-fast optimal pathfinding without runtime search. In *Proceedings of the Seventh AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. (cited on pages 11 and 26)

Geisberger, R.; Sanders, P.; Schultes, D.; and Delling, D., 2008. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *International Workshop on Experimental and Efficient Algorithms*, 319–333. (cited on pages 11 and 12)

Grastien, A., 2019. Brigitte, a bridge-based grid path-finder. In *Twelfth Annual Symposium on Combinatorial Search*. (cited on pages 17, 23, and 27)

Harabor, D. and Grastien, A., 2011. Online graph pruning for pathfinding on grid maps. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*. (cited on pages 8, 14, and 15)

Harabor, D. and Grastien, A., 2014. Improving jump point search. In *Proceedings of the International Conference on Automated Planning and Scheduling*. (cited on page 10)

Harabor, D.; Uras, T.; Stuckey, P. J.; and Koenig, S., 2019. Regarding jump point search and subgoal graphs. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence (IJCAI-19)*. (cited on pages 6, 19, 22, and 29)

Hu, Y.; Harabor, D.; Qin, L.; and Yin, Q., 2021. Regarding goal bounding and jump point search. *Journal of Artificial Intelligence Research 70*, (2021). (cited on pages 22 and 46)

Rabin, S. and Sturtevant, N., 2016. Combining bounding boxes and jps to prune grid pathfinding. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*. (cited on pages 21, 29, 36, and 46)

Salvetti, M.; Botea, A.; Gerevini, A. E.; Harabor, D.; and Saetti, A., 2018. Two-oracle optimal path planning on grid maps. In *Proceedings of the International Conference on Automated Planning and Scheduling*. (cited on pages 20 and 45)

Salvetti, M.; Botea, A.; Saetti, A.; and Gerevini, A. E., 2017. Compressed path databases with ordered wildcard substitutions. In *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling*. (cited on pages 11 and 27)

Strasser, B.; Botea, A.; and Harabor, D., 2015. Compressing optimal paths with run length encoding. *Journal of Artificial Intelligence Research 54*, (Dec. 2015). (cited on page 11)

Sturtevant, N. R., 2012. Benchmarks for grid-based pathfinding. In *Transactions on Computational Intelligence and AI in Games*. (cited on pages 39 and 40)

Uras, T. and Koenig, S., 2014. Identifying hierarchies for fast optimal search. In *Twenty-Eighth AAAI Conference on Artificial Intelligence*. (cited on pages 20 and 62)

Uras, T. and Koenig, S., 2017. Feasibility study: Subgoal graphs on state lattices. In *Proceedings of the Tenth International Symposium on Combinatorial Search*. (cited on pages 12 and 13)

Uras, T. and Koenig, S., 2018. Understanding subgoal graphs by augmenting contraction hierarchies. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. Code available at: http://idm-lab.org/sg-ch. (cited on pages 11, 14, 19, and 20)

Uras, T.; Koenig, S.; and Hernandez, C., 2013. Subgoal graphs for optimal pathfinding in eight-neighbor grids. In *Proceedings of the International Conference on Automated Planning and Scheduling*. (cited on pages 12, 13, 14, and 26)