Dynamic Pathfinding in Video Games

Eric Shi

Supervisors: Daniel Harabor, Arthur Mahéo

ID: 31531857
Course: FIT5126

1 Introduction

Artificial intelligence is a main component in video games, with pathfinding being a sub-field. Pathfinding is the problem of finding a path from a start state to a goal state and has applications in many genres of video games.

In role-playing games and real-time strategy games, the player can route units to different locations on the map. Each of these routes would require a pathfinding query. In first-person shooters, non-playable characters would need to be controlled by artificial intelligence and will need to be redirected to different locations depending on the action of the player.

An example of a game that utilises pathfinding is the real-time strategy game Starcraft. In Starcraft, players create units and buildings to destroy the other player's buildings. Players often redirect many units to different locations on the map to do tasks or engage in combat, resulting in many pathfinding queries. For the game to run smoothly, pathfinding algorithms need to be fast with limited CPU and memory resources [3]. However, paths that can be found quickly have issues. The paths can take very long detours, or when viewed by a human the path looks terrible. These paths would lead to a poor player experience, as games of Starcraft can be won or lost if important units take too long to move between tasks or move too late to a close battle.

In video games, the problem with efficient pathfinding is that the environment is often dynamic. In dynamic environments, the world around the agent can change. In video games, this can happen through buildings being constructed or destroyed, or doors on the map opening and closing at varying time intervals. Current methods to attempt these problems make false assumptions that the world is static and can weaken the information that the pathfinding AI agent is given. This results in search being slower and unable to prove if a path is optimal.

In this paper, we explore current methods of pathfinding and improvements to these methods. We also propose an algorithm faster than the state of the art and producing higher quality paths.

2 Literature Review

2.1 The pathfinding problem

The pathfinding problem is finding an optimal path on a graph G from a start node s to a goal node t. A graph G is defined as a set of nodes n_i and edges e_{ij} connecting adjacent nodes n_i and n_j. Each edge has associated cost c_{ij}. A path π from n_1 to n_k is defined as an ordered set of nodes $< n_1, n_2, ..., n_k >$, with each adjacent pair of nodes having an edge between them. The optimal path from s to t is defined as the path which has the smallest cost from the set of all paths from s to t. [9]

2.2 Abstraction and representation of video game maps

2.2.1 Grid representation

Usually, maps in video games are represented as grids. A grid can be viewed as a graph where nodes are tiles and edges are the possible moves between the tiles. Straight line moves are allowed as long as the tile being moved to is empty. Diagonal moves are only allowed if both straight tiles are free in the diagonal direction, as we assume agents have size. This is known as the corner cutting constraint. [11]

Figure 1: Possible grid expansions from a node

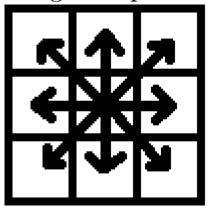
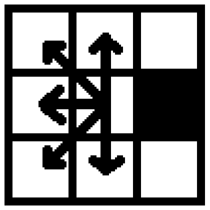


Figure 2: Possible grid expansions from a node when obstacles are present



Edge costs are 1 for straight moves, and $\sqrt{2}$ for traversing diagonally. In video games where each tile has different terrain, the edge cost of traveling to a different node can be modeled as the terrain factor multiplied by the distance of traversal. [13]

Grid representations are beneficial as they are a simple representation of a gaming map. Changes in the map are also easy to model. If there is an obstacle that appears on the map, the corresponding nodes and edges in the grid representation can be queries and changed in constant time. [8]

However grid representations have flaws. The grid representation only allows discrete actions, being only diagonal at 45 degrees, vertical or horizontal. The grid representation does not allow actions at different angles. This can lead to a better path being missed.

The grid representation also does not allow the representation of obstacles that do not conform to the grid shape. For example, circular obstacles cannot be accurately represented by grids. The quality of representing the obstacles can improve by increasing the resolution of the grid, but this leads to an increase in memory usage and an increase in difficulty of finding path as there are more nodes. [8]

2.2.2 Navigation meshes

Another method is navigation meshes. Navigation meshes are a representation of a game map where the map is divided into convex polygons along with a graph that describes how polygons are connected. [14]

Meshes are beneficial as they are more flexible and memory efficient compared to grid maps. However one disadvantage is that navigation meshes are harder to change if the environment changes as the shapes may need to be recomputed. [8]

Recent work has been undertaken to construct navigation meshes for dynamic environments. [14] When the environment changes, only the affected areas are repaired instead of all navigation mesh being recomputed. This works quickly and is able to update the navigation mesh in real time which is essential for video games.

2.3 A* algorithm

In video games and road routing and other applications, A* is the most common algorithm used for path-finding. In the A* algorithm [9], nodes are expanded in priority of lowest $f(n)$ where $f(n) = g(n) + h(n)$. $g(n)$ is the current cost of a node and $h(n)$ is the heuristic estimate of the cost from n to the goal. A* utilises an open list and closed list to store the nodes expanded and the nodes that are yet to be expanded respectively.

The pseudo-code of the algorithm follows:

```
OPEN = {s}
CLOSED = {}
while OPEN $\neq \emptyset$ do
    $n \leftarrow$ head(OPEN)
    if isGoal(n) then
        Return the path
    else
        OPEN $\leftarrow$ OPEN \ {n}
        CLOSED $\leftarrow$ CLOSED $\cup$ {n}
        for each $n'$ $\in$ succ(n) do
            if $n'$ $\notin$ OPEN and $n'$ $\notin$ CLOSED then
                OPEN $\leftarrow$ OPEN $\cup$ {n'}
            end
            if $n'$ $\in$ CLOSED and $f(n')$ decreased then
                CLOSED $\leftarrow$ CLOSED \ {n}
                OPEN $\leftarrow$ OPEN $\cup$ {n}
            end
        end
    end
end
```

Algorithm 1: A* algorithm

As A* expands the node with lowest $f(n)$, the search expands nodes that are the most promising and avoids expanding along sub-optimal paths. A* also has other good properties. A* is complete as A* can find an optimal solution if there is one, and is optimal given an admissible and consistent heuristic. A* is also optimally efficient, no other algorithm which has access to the same information A* can possibly expand fewer nodes. [5]

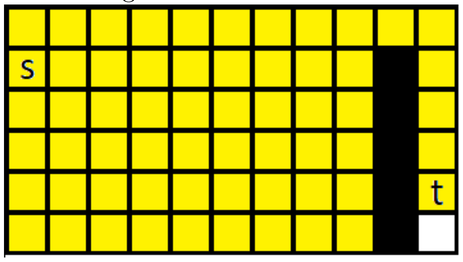
A heuristic is admissible if: $h(n) \leq h^*(n)$ where $h^*(n)$ is the exact cost from n to the goal node t. The heuristic must be less than exact cost for the heuristic to be admissible.

For a heuristic to be consistent, $h(n) \leq h(n,m) + h(m)$ where $h(n,m)$ is a function that returns the exact cost from the pair of nodes n and m. The difference in heuristic values between a pair of nodes must be less than the exact cost between them for a heuristic to be consistent.

Even though A* has nice properties and is an effective algorithm, its performance is very dependent on the heuristic. In A* search, the time complexity is given by $O(b^d)$, where b is the number of branches expanded on and d is the depth of the deepest node of the graph. b is highly dependent on the heuristic. In the case where $h(n) = h^*(n)$, $b = 1$ and A* only expands along the optimal path and has a complexity of $O(d)$. In the opposite case where $h(n) = 0$, A* has to expand along all branches resulting in complexity $O(8^d)$ for grid maps. As can be seen, A* performs better when $h(n)$ approaches $h^*(n)$. With a weaker heuristic more nodes will have similar f-values, forcing them all to be expanded resulting in the high branching factor.

As an example, the octile heuristic is the distance between two locations on a grid undertaking only diagonal and straight moves. When there are no obstacles present on the grid, the heuristic is perfect as $h(n) = h^*(n)$. The search expands straight towards the goal. However when a wall is added between the start and goal locations, search becomes more difficult as the heuristic becomes weaker.

Figure 3: Using the octile heuristic for this pathfinding problem expands every node on the grid before finding a solution because of the wall obstacle



Therefore we would prefer heuristics that are closer to the actual cost. Even though octile distance is a good heuristic for gridmaps with few to no obstacles, video games usually have a large number of obstacles and therefore the search will be slow.

2.4 Differential heuristics

Differential heuristics [12] is an improvement on naive heuristics like octile distance. Unlike these naive heuristics that do not infer about obstacles, differential heuristics provide more information on obstacles and are a stronger heuristic.

Differential heuristics use the triangle inequality[6]. The triangle inequality states that for any triangle, the sum of the lengths of any two sides must be greater or equal to the length of the remaining side. This same reasoning applies to path costs on a graph. We can rearrange the triangle inequality. Let p be a node on the graph that we will call the pivot, s be the start node and t be the goal node.

$d(s,p) \leq d(s,t) + d(t,p)$

$|d(s,p) - d(t,p)| \leq d(s,t)$

As $d(s,t)$ is the actual distance/cost from s to t, this is the same as h^*. As $|d(s,p) - d(t,p)|$ is always less than the true distance/cost, we can use it as a heuristic for planning a path between s and t. This forms the basis of differential heuristics.

To be able to use this idea, the values of $d(s,p)$ and $d(t,p)$ are needed. In a pre-computation step before search, Dijkstra's search is performed from the pivot node p to all nodes on the graph. From Dijkstra's search, $d(n,p)$ will be obtained for all nodes n on the graph. These values are stored in a full all-pairs database where the costs can be retrieved during search.

For differential heuristics, the choice of pivot helps a lot with the strength of the heuristic. The differential heuristic gives the actual cost between two points, if one of the points is on the optimal path between the second point and the pivot. [12] For this to occur, effective pivots should be placed on the outer regions of the graph.

Differential heuristics can also be improved by adding more pivots. By adding more pivots, we have access to more admissible heuristics. As larger admissible heuristics are preferred to make A* more efficient, the maximum of all differential heuristics is used which makes search more efficient.

Even though differential heuristics are stronger than naive heuristics, they require pre-computation and more memory usage. For differential heuristics, the memory requirement is $O(kN)$ and the time to build the table is $O(kN)$, where k is the number of pivots and N is the number of nodes in the graph.

Differential heuristics also make assumptions that the map is static. If obstacles are removed from the map, the differential heuristic can become inadmissible. This is because h^* can decrease as goal nodes can be more easily accessible as obstacles are not present to block a more direct path. If more obstacles are added to the path, differential heuristics weaken in power as h^* could increase as the previous optimal path could now be blocked. However in cases where this doesn't happen, differential heuristics offer large speedups in search.

2.5 Bounded sub-optimal search

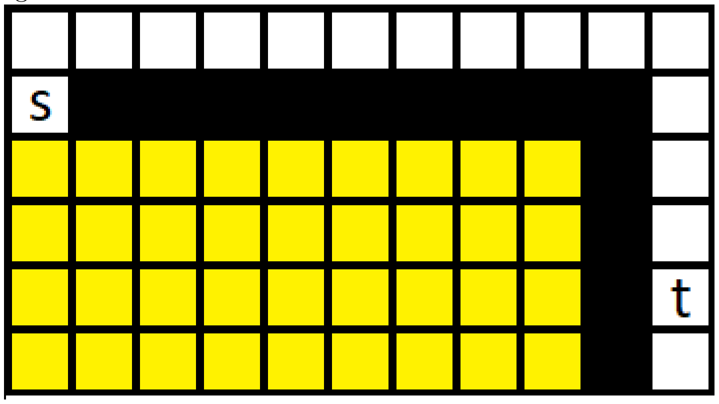
The heuristics discussed can still be too weak for A* to quickly find an optimal solution. However we can loosen the requirement of obtaining an optimal solution. We can approximate a solution where the approximate solution is guaranteed to be within a certain percentage of the optimal solution.

The variant of A* to achieve this is called Weighted A* [10]. In Weighted A*, $f(n) = g(n) + w \times h(n)$ instead, where $w > 1$ is a parameter decided by the user. If $h(n)$ is admissible, the search is bounded and the solution cost cannot be greater than $w\times$optimal cost.

With an inflated $h(n)$, A* search expands nodes more greedily, choosing to expand nodes that it thinks are closest to the goal with less regard for the current cost undertaken. With increasing weight, A* search expands even more greedily, and approaches greedy best-first search when $w \to \infty$. Weighted A* performs best when search problems have near optimal or approximate solutions that can be found very quickly compared to the optimal solution. [7]

However, Weighted A* performs poorly when this isn't the case. Weighted A* can still expand nodes close to the start and not close to nodes that are needed to be bypassed to able to reach the goal.

Figure 4: Weighted A* expansions with an octile heuristic. It expands all the nodes in yellow that are clearly not on any path towards the goal, instead of trying to go around the obstacle



Weighted A* can be altered to be an anytime algorithm, where search continues by expanding the lowest $f(n) = g(n) + w \times h(n)$ after a solution is found, until timeout or the optimal solution is found. [7] Weighted A* is useful if remaining time is available, to continue to improve the current solution.

To be able to identify when the optimal solution is found, anytime Weighted A* keeps track of the upper bound and lower bound of the cost of the optimal solution. The upper bound of the cost of the optimal solution, is the cost of the best path found so far. This is because the cost of the optimal path can't be higher. The lower bound is the unweighted f-value of the best node in open. This is because as $h(n)$ is admissible, when the unweighted f-value increases, we can prove all solutions lower than it can't exist.

2.6 CPD heuristics

Compressed databases were first introduced [2] to speed up search by preprocessing the paths. Future work [1] used CPDs as heuristics for dynamic problems.

Unlike differential heuristics that store heuristic values from k pivots to all the other nodes on the graph, CPDs store information of an optimal move for all nodes and for all start and target pairs.

To create the CPD, there is initially an offline procedure for each node of the graph. For each node, a complete Dijkstra search is done to a goal from the start node. This forms a first-move table, where all nodes are assigned a label that identifies which edge leaving the start node is on the shortest path. This process can be done in parallel. The heuristic value can easily be obtained from the original weights of the graph and the path obtained.

CPD heuristics are stronger than differential heuristics as CPD heuristics return the exact cost on an unchanged map. For differential heuristics to guarantee the same, every node on the graph would need to be a pivot. CPD heuristics are also admissible as long as obstacles on the map are not removed.

For video games, CPD heuristics are beneficial as most of the pathfinding is done in the offline phase and not during the time when the game is running.

However CPD heuristics have issues. Similarly to differential heuristics, CPD heuristics become inadmissible when obstacles are removed. When obstacles are removed, the optimal path cost can decrease. Therefore $h(n) > h^*(n)$ resulting in inadmissibility. When more obstacles are added to the map, similarly to differential heuristics, the CPD heuristic decreases in power.

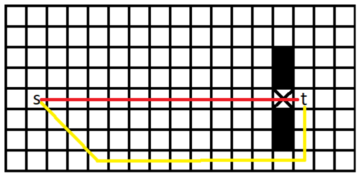


Figure 5: Inadmissibility of CPD heuristics when an obstacle is removed. The red path shows the new optimal path, whereas the yellow path is the CPD path. The CPD's heuristic is inadmissible as it now does not return the optimal solution

CPD heuristics are also memory intensive, taking $O(N^2)$ space, compared to differential heuristics $O(kN)$. However compression helps decrease the size by 99% making CPDs viable [2].

In combination with Weighted A*, CPD heuristics have been used in experiments where edges of the map are changed with faster results[1] compared to Landmarks [6] which uses differential heuristics.

2.7 Focal search

Focal search[4] is a bounded sub-optimal search algorithm based on Weighted A*. The idea of focal search is to use two priority queues for open and focal. Focal is a subset of open, only storing nodes from f_{min} to $w \times f_{min}$, where f_{min} is the lowest f cost in open. Instead of picking nodes from open to expand, we expand the best node from focal according to a user-defined priority function. As the f_{min} increases, new nodes are added into focal from open that are within the new f_{min} to $w \times f_{min}$ bound.

An example of focal search is Weighted A*. Weighted A* is effectively focal search with a user-defined priority function being the lowest $f(n) = g(n) + w \times h(n)$. However the benefit of focal search is that it is not restricted to only expand in this way to find a solution fast.

Similarly to Weighted A*, focal search can also be transformed to be an anytime algorithm, using the same transformation steps as Weighted A*.

Focal search performs well on problems where correlation between the heuristic and minimum amount of actions to goal is low [4]. This is a problem of Weighted A*, as Weighted A* assumes that decreasing the heuristic value will quickly lead to a solution. With a good specialised user defined priority function for a specific pathfinding problem, focal search can quickly find a solution.

```
OPEN = FOCAL = {s}
CLOSED = {}
while $FOCAL \neq \emptyset$ do
    $n \leftarrow$ head(OPEN)
    $f_{min} \leftarrow f$(head(OPEN))
    if isGoal(n) then
        Return the path
    else
        OPEN ← OPEN \ {n}
        FOCAL ← FOCAL \ {n}
        CLOSED ← CLOSED ∪ {n}
        for each $n'$ ∈ succ(n) do
            if $n'$ ∉ OPEN or CLOSED then
                OPEN ← OPEN ∪ {n'}
            end
            if $n'$ ∈ CLOSED and $f(n')$ decreased then
                CLOSED ← CLOSED \ {n}
                OPEN ← OPEN ∪ {n}
            end
            if $f(n') \leq wf_{min}$ and $n'$ ∈ CLOSED then
                FOCAL ← FOCAL ∪ {n}
            end
        end
        if $OPEN \neq \emptyset$ and $f_{min} < f$(head(OPEN)) then
            UpdateLowerBound($wf_{min}$, $wf$(head(OPEN)))
        end
    end
end
```

Algorithm 2: Focal Search

3 Summary of State of the Art

As of current, pathfinding algorithms in static environments are a well explored area. In static environments, current literature have expanded on the A* algorithm with heuristics. Basic heuristics in grids use octile distance. However simple heuristics prove poor, as they don't take into account obstacles on the map. This led to stronger heuristics that provide more information on obstacles, CPDs and differential heuristics. However, both heuristics assume the map is static. If more obstacles appear, CPD and differential heuristics become weaker and if obstacles can disappear, CPD and differential heuristics are invalidated.

In dynamic problems such as video games, the map is constantly changing. Obstacles are moving, appearing and disappearing. This makes the heuristics weaker, which causes A* to be slow. Weighted A* tries to solve this by weighing the heuristic more heavily compared to the cost of a node. In this way, weighted A* could find a feasible path faster by expanding more greedily. Unfortunately this suffers the same problem as A*. Weighted A* can also expand a lot of nodes which is a problem if a decreasing heuristic value does not correlate well with finding a solution. [15]

Pathfinding in video games needs to be responsive and fast, as well as generate solutions that look realistic. With current methods in literature, trying to find a realistic solution is too slow.

Focal search deals with these issues, by expanding all nodes in the focal list within the bound denoted in weighted A*. Nodes that are more likely to generate the solution can be generated first, and the sub-optimality bound will still hold. However, focal search has not been used for dynamic pathfinding applications.

We propose an algorithm to make better use of the bound compared to Weighted A*, by using focal search to expand around obstacles quicker instead of only following a weighted heuristic.

4 Plan for research project

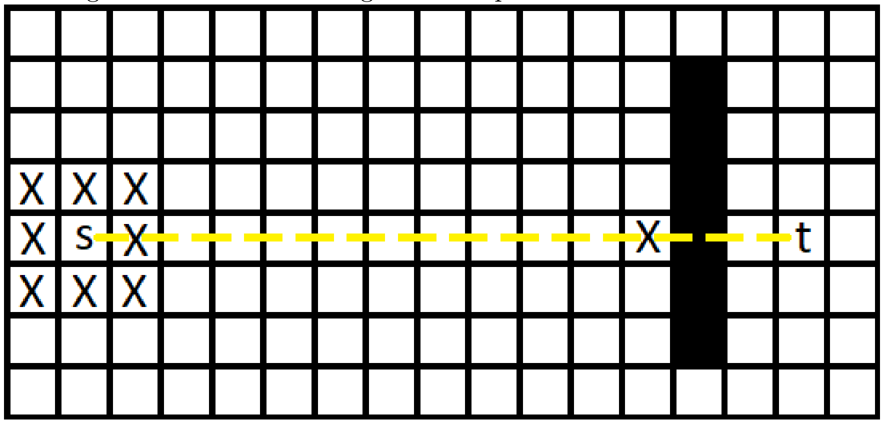
The plan for the research project is to create an algorithm that will quickly find a feasible solution for pathfinding in a dynamic environment. To be able to do so, we would like to quickly explore around obstacles using focal search.

4.1 Choice of Heuristic and Expander Modification

For the algorithm, we're going to be combining ideas from focal search as well as CPD heuristics. The algorithm will first involve creating a CPD for the static map. We then use the CPD as a heuristic in focal search.

In grid maps, nodes that are expanded are neighbours of the current node in 4 cardinal directions as well as 4 diagonal directions. We'll also be using the CPD go along the CPD path until a change in edge cost. A change in edge cost can occur either by a obstacle or the terrain on a tile changing. By going along the CPD path, we can quickly get to the goal if there are no additional obstacles between the current location to the goal.

Figure 6: An example where the CPD is constructed on an empty map with no obstacle. The map is changed with a new obstacle put in place. The nodes that are generated from expanding the start node is shown by X with the X next to the wall generated from following the CPD path.



4.2 Variants of Focal Search

For focal search, we have two different ideas to implementing focal search.

4.2.1 Normal Focal Search

An idea is to experiment with different priority functions for focal. An idea for an heuristic is to trace around obstacles. By expanding nodes within focal that meet that criterion, we can attempt to go around obstacles sooner compared to Weighted A*.

4.2.2 Exhausting the whole list

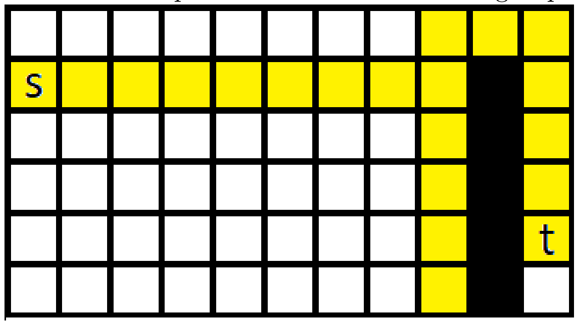
In this approach, we first expand the best node in open. We then add the best node from open into focal and perform focal search. Unlike in the sliding window approach where nodes are added into focal from open when the lower bound increases, we fix the bounds of focal. Eventually, the focal list will not have any nodes when all nodes generated have f-values greater than $w \times f_{min}$. Once this happens, the best node from open is expanded again and a new f_{min} is brought from open to focal and the bounds are updated. With this method, we commit to one node closest to the goal by following the CPD path. From this node, we can try to bypass the obstacle more quickly to get back onto the CPD path.

```
OPEN = {s}
CLOSED = {}
while OPEN $\neq \emptyset$ do
    OPEN ← OPEN \ {n}
    CLOSED ← CLOSED ∪ {n}
    for each $n'$ ∈ succ(n) do
        if $n' \notin$ OPEN and $n' \notin$ CLOSED then
            OPEN ← OPEN ∪ {n'}
        end
        if $n' \in$ CLOSED and $f(n')$ decreased then
            CLOSED ← CLOSED \ {n}
            OPEN ← OPEN ∪ {n}
        end
    end
    $n$ ← head(OPEN)
    $f_{min} \leftarrow f$(head(OPEN))
    Follow the CPD path from $n$ until a node $n'$ before a change in edge
     cost
    n ← n'
    FOCAL = {n}
    while FOCAL $\neq \emptyset$ do
        $n$ ← head(FOCAL)
        if isGoal(n) then
            Return the path
        else
            for each $n'$ ∈ succ(n) do
                if $n' \notin$ OPEN and $n' \notin$ CLOSED then
                    OPEN ← OPEN ∪ {n'}
                end
                if $n' \in$ CLOSED and $f(n')$ decreased then
                    CLOSED ← CLOSED \ {n}
                    OPEN ← OPEN ∪ {n}
                end
                if $f(n') \leq wf_{min}$ and $n' \notin$ CLOSED then
                    FOCAL ← FOCAL ∪ {n}
                end
            end
        end
        if OPEN $\neq \emptyset$ and $f_{min} < f$(head(OPEN)) then
            UpdateLowerBound($wf_{min}$, $wf$(head(OPEN))
        end
    end
end
```

Algorithm 3: Focal Search: Exhausting the whole list

Another idea for exhausting the list is instead of expanding the node completely in focal search, we instead implement wall following. With wall following we only look at node neighbours that are along the obstacle and add them to focal as long as they are within the bound.

Figure 7: Nodes to be expanded with a wall following implementation



4.3 Benchmarking

The algorithm will be tested against benchmarks of CPD search, differential heuristics with varying number of pivots, and bounded sub-optimal searches of weighted A* and focal searches. The maps that will be used will be from Dragon Age Origins, Warcraft III and Rooms. These maps will be sourced from the MovingAI repository [11]. In each benchmark, time taken and number of nodes expanded will be recorded.

We will use Dragon Age Origins maps to represent uniform cost problems, where each tile has same cost of traversal. Warcraft III maps have different terrain, with trees and water that have different traversal costs. Rooms maps are constructed by multiple equal size square rooms put together. The path between rooms only have a single tile entrance. With these different maps, we'll test varying situations that can occur in a video game setting.

For an initial benchmark, we first simulate the AREA experiments [1] to perturb edges on the map. To do this, the optimal path on the original map is found. After, we pick a random node in the path to perturb. From the perturbed node, edges up to 15 traversals away are updated with formula $w'(e) = w(e) \times (3e^{\frac{-x^2}{45}} + 1)$. Edges further away from the perturbed node have lessened increase on cost, while edges near the perturbed node have larger increases in cost. This simulates a problem in video games where an area can have more congestion around a point resulting in slower travel.

4.4 Future obstacles

Later, we'll implement obstacles along the precomputed paths of the maps to benchmark the algorithm. Obstacles will include moving obstacles, as well as obstacles that appear and disappear at different time intervals. For example, doors in a video game can open up permitting a path or can close which will require us to find a different path around the door instead.

5 Conclusion

This paper describes the dynamic pathfinding problem in video games and analyses current approaches to the problem. Current search algorithms of A* search and Weighted A* with precomputed heuristics are slow to search when obstacles appear and move around. We propose a plan to develop an algorithm that uses the strength of compressed path databases (CPDs) heuristics in combination with focal search to be able to quickly find paths around obstacles as well as be within a sub-optimality bound.

References

[1] Massimo Bono et al. "Path Planning with CPD Heuristics". In: *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence* (2019). DOI: `10.24963/ijcai.2019/167`.

[2] Adi Botea. "Ultra-fast optimal pathfinding without runtime search". In: *Proceedings of the Seventh AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE'11)* (2011), pp. 122–127.

[3] Adi Botea et al. *Pathfinding in Games*. 2020. URL: `http://dx.doi.org/10.4230/DFU.Vol6.12191.21`.

[4] Liron Cohen et al. "Anytime Focal Search with Applications". In: *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence* (2018). DOI: `10.24963/ijcai.2018/199`.

[5] Rina Dechter and Judea Pearl. "Generalized best-first search strategies and the optimality of A*". In: *Journal of the ACM* 32.3 (1985), pp. 505–536. DOI: `10.1145/3828.3830`.

[6] Andrew V. Goldberg and Chris Harrelson. "Computing the Shortest Path: A* Search Meets Graph Theory." In: *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms* (2005), pp. 156–165.

[7] Eric A. Hansen and Rong Zhou. "Anytime Heuristic Search". In: *Journal of Artificial Intelligence Research* 28 (2007), pp. 267–297. DOI: `10.1613/jair.2096`.

[8] Daniel Harabor. “Fast and Optimal Pathfinding”. PhD thesis. The Australian National University, 2014.

[9] Peter Hart, Nils Nilsson, and Bertram Raphael. “A Formal Basis for the Heuristic Determination of Minimum Cost Paths”. In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107. DOI: 10.1109/tssc.1968.300136.

[10] Ira Pohl. “Heuristic search viewed as path finding in a graph”. In: *Artificial Intelligence* 1.3-4 (1970), pp. 193–204. DOI: 10.1016/0004-3702(70)90007-x.

[11] Nathan R. Sturtevant. “Benchmarks for Grid-Based Pathfinding”. In: *IEEE Transactions on Computational Intelligence and AI in Games* 4.2 (2012), pp. 144–148. DOI: 10.1109/tciaig.2012.2197681.

[12] Nathan R. Sturtevant et al. “Memory-based heuristics for explicit state spaces”. In: *Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence (IJCAI-09)* (2020), pp. 609–614.

[13] Nathan R. Sturtevant et al. “Pathfinding and Abstraction with Dynamic Terrain Costs”. In: *Proceedings of the Fifteenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE-19)* (2019), pp. 80–86.

[14] Wouter G. van Toll, Atlas F. Cook, and Roland Geraerts. “A navigation mesh for dynamic environments”. In: *Computer Animation and Virtual Worlds* 23.6 (2012), pp. 535–546. DOI: 10.1002/cav.1468.

[15] Christopher Wilt and Wheeler Ruml. “When does weighted A* fail?” In: *Proceedings of the 5th Annual Symposium on Combinatorial Search, SoCS 2012* (Jan. 2012), pp. 137–144.