Dynamic Pathfinding in Video Games

Eric Shi

Supervisors: Daniel Harabor, Arthur Mahéo

ID: 31531857
Course: FIT5128
Word Count: 3948

ABSTRACT

Pathfinding is the problem of finding a path from a start to a target location. Pathfinding has many applications, for example in video games and GPS systems. Pathfinding works well on static maps, however when the map changes, pathfinding algorithms weaken. Our approach improves current dynamic pathfinding algorithms in video game settings by exploiting more information from Compressed Path Databases which is the state of the art in dynamic pathfinding.

Contents

1 Introduction

Pathfinding has many applications, especially in video games. For example in real-time strategy games and role playing games, the player directs units on a map to fulfil a task. In Starcraft, players constantly move units to new locations to construct building and engage in combat. Games need to quickly find high quality paths quickly and with limited time and computing resources to maintain a good player experience (Botea et al., 2020).

However, pathfinding in video games is difficult as the map is dynamic. Dynamic maps have constant changes. For example, terrain can change or buildings are constructed. Dynamic maps pose issues as current state of the art algorithms make assumptions that the map is static. When presented with a dynamic map, state of the art algorithms are either too slow to find a good path or generate a poor path quickly.

Bounded sub-optimal searches deal with this problem, by allowing paths to not have optimal cost but be within a factor of the optimal cost. By dropping the optimality constraint but also restricting paths to within a factor of the optimal cost, paths can be found faster and be decent quality.

Search algorithms achieve this by having lower and upper bounds of the path cost. The lower bound is the lowest the path cost can potentially be and the upper bound is the path cost of the best tentative path. Once the gap between the upper bound and lower bound is within a factor, sub-optimality is guaranteed.

Our approach is a build up on Compressed Path Database search, the state of the art algorithm in dynamic pathfinding. CPD search deals with the bounded sub-optimal pathfinding problem by expanding nodes near the start to raise the lower bound. Nodes are expanded until the gap between the lower and upper bound is within a factor ϵ.

We propose that instead of focusing on raising the lower bound through CPD search, decreasing the upper bound by finding a better tentative path can find a path faster. To do so, we propose wall-following. By expanding around the obstacle and getting around it instead of expanding from the start, we can quickly find a better tentative path and lower the upper bound rapidly to close the gap.

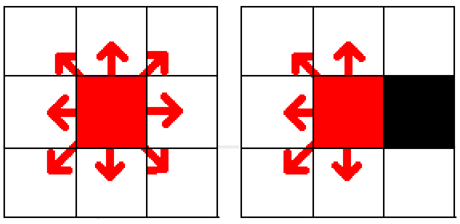
We found that our new search finds paths faster than current state of the art searches: CPD search with early termination and Weighted A* with CPD heuristics.

2 Problem description

The pathfinding problem takes input graph $G = (N, E)$ with nodes N and edges E. Each edge has a weight w, where $w(m, n)$ is the weight cost between node m and node n. Given a start node s and a target node t, the output is a path on G from s to t where a path is defined as a tuple of nodes $(s, \dots, t)$. The optimal path from s to t is the path with the lowest cost from the set of all paths from s to t.

We study path problems on grids. To represent a grid as a graph, nodes are represented as tiles with edges being the path between the tiles. Straight line moves on the grid are allowed if the node being traversed to is unblocked and have base cost of 1. Diagonal moves on the grid are allowed if both tiles in the horizontal and vertical direction are traversable. This is because agents are assumed to have size. This is known as the corner cutting constraint. Diagonal moves have base cost of $\sqrt{2}$.

FIGURE 1: Available traversals from a node on a gridmap)



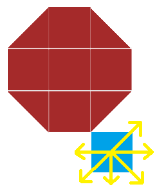
In video games where each tile has different terrain, the edge cost of travelling to a different node can be modelled as the average terrain factor between the two nodes multiplied by the base cost (Sturtevant et al., 2019).

For dynamic pathfinding we assume an offline phase and an online phase. In the offline phase, we have prior knowledge of the graph before receiving path planning queries. In the online phase for each query, a start and target location is given with edges perturbed on the graph.

We focus on edge perturbations where nodes are made untraversable by blocking all incoming and outgoing edges of the node. Corner cutting is allowed on newly perturbed nodes –see fig. 2, however the algorithms being discussed are still applicable on newly perturbed obstacles that do not allow corner cutting.

We assume that the graph does not change during the episode, and only between queries. Each query is distinct and independent from other queries.

FIGURE 2: An example 3x3 perturbed area with a node showing possible traversals with arrows. Note that corner cutting is allowed along perturbed nodes.)



3 Background

3.1 A* ALGORITHM

The A* algorithm (Hart et al., 1968) is the most prevalent pathfinding algorithm used in video games. In A*, each node has two values:

1. a cost from the start $g(n)$; and,
2. an estimated cost to the target from the node $h(n)$.

Nodes are expanded in order of their sum $f(n) = g(n) + h(n)$.

A* is a complete algorithm, it will always find a solution it one exists. A* is also optimal, the solution A* finds is always the shortest path. A* is also optimally efficient up to tie-breaking, no other search can find the optimal path in a lower amount of expansions than A* given the same information. However running vanilla A* is often too slow for video games.

The performance of A* is heavily dependent on the heuristic. To have optimality guarantees the heuristic must be admissible. A heuristic $h(n)$ is admissible if $h(n) \leq h'(n)$, for all nodes and where $h'(n)$ is the actual cost of n to target.

Even if the heuristic is admissible, if there is a large difference between the heuristic and the actual cost the amount of nodes to expand is high resulting in slower search.

A commonly used heuristic is the octile heuristic. The octile heuristic is the distance from start to target given only diagonal and straight moves on a map with no obstacles. The heuristic is perfect and search is fast if there are no obstacles between the start and target. However in video games, the map has many obstacles with terrain changes and buildings. This makes search slow as the gap between the true cost and the octile heuristic widens – see fig. 3.

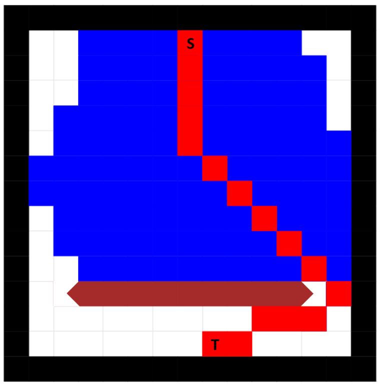
To improve the speed of A*, current state of the art approaches

Algorithm 1: A* algorithm

```
OPEN = {s}
CLOSED = {}
while OPEN ≠ ∅ do
    $n$ ← head(OPEN)
    if isGoal(n) then
        Return the path
    else
        OPEN ← OPEN \ {n}
        CLOSED ← CLOSED ∪ {n}
        for each $n'$ ∈ succ(n) do
            if $n'$ ∉ OPEN and $n'$ ∉ CLOSED then
                OPEN ← OPEN ∪ {n'}
            end
            if $n'$ ∈ CLOSED and $f(n')$ decreased then
                CLOSED ← CLOSED \ {n}
                OPEN ← OPEN ∪ {n}
            end
        end
    end
end
```

involve stronger heuristics as well as modifications to A* to be bounded sub-optimal.

FIGURE 3: Most of the nodes have to be expanded (in blue) by backfilling because of the poor octile heuristic that does not take into account the new obstacle.)



3.2 SUB-OPTIMAL PATHFINDING ALGORITHMS

Using heuristics with assumptions of static maps with A* to find the optimal solution is often too slow because of back-filling. By using bounded sub-optimal pathfinding algorithms, a path can be found that is controllably worse but can be found faster. In the bounded sub-optimal pathfinding problem, a valid path is any path a factor of $\epsilon > 1$ worse than the optimal path.

A common approach to achieve sub-optimal pathfinding is inflating the heuristic in A* search by a factor $\epsilon > 1$. The resulting algorithm is called Weighted A* (Pohl, 1970). In doing so, the search acts greedily and can find a solution faster by back-filling less. Even though the heuristic is now not admissible, the search is bounded and the solution cost cannot be greater than $\epsilon \times f_{min}$. (f_{min} is interchangeably used for the lower bound.) The search is deemed bounded sub-optimal. Bounded sub-optimal searches are good for video games, as a path can be found fast but the quality of the path is guaranteed to be decent.

However this approach does not work well if the heuristic does not correlate well with the actual cost of the solution. (Wilt and Ruml, 2012). For example, hard obstacles added to the map which are not accounted for by the heuristic leads to more expansions through back-fill and slower runtimes.

A solution for this issue is FOCAL search (Cohen et al., 2018; Pearl and Kim, 1982). FOCAL search, instead of expanding nodes in order of f, FOCAL search has a second priority queue FOCAL, sorted on a user defined priority function. FOCAL only stores nodes less than $\epsilon \times f_{min}$ to ensure bounded sub-optimality where the path cost cannot be greater than $\epsilon \times$ optimal cost. When f_{min} raises by expanding from OPEN, FOCAL is updated to include newer nodes that are now within the bound. As the user defined priority function is domain specific, it's more flexible than early termination and weighted searches. This can reduce the number of expansions, decreasing the time of search.

The main drawback to FOCAL search is two priority queues need to be maintained instead of only one. However, FOCAL search still works well in practice.

Algorithm 2: Focal Search

```
OPEN = FOCAL = {s}
CLOSED = {}
while $FOCAL \neq \emptyset$ do
    $n \leftarrow$ head(FOCAL)
    $f_{min} \leftarrow f$(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 $f(n') \leq \epsilon f_{min}$ and $n'$ ∈ CLOSED then
                FOCAL ← FOCAL ∪ {n}
            end
        end
        if $OPEN \neq \emptyset$ and $f_{min} < f$(head(OPEN)) then
            UpdateLowerBound($\epsilon f_{min}$, $\epsilon f$(head(OPEN))
        end
    end
end
```

3.3 COMPRESSED PATH DATABASES

The state of the art for heuristics is Compressed Path Databases. CPDs store optimal paths on the static map for fast retrieval.

To construct a CPD, there is an offline procedure to run a modified Dijkstra's search algorithm: find the all-pairs shortest paths from all start locations. From the Dijkstra searches, a first move table is constructed where each target location is assigned a label to indicate which edge from the start location is on the shortest path.

CPDs are then compressed by run-length encoding. In run-length encoding, a string of symbols in the table can be compressed if they're the same. Compression helps decrease the size of path databases by 99% (Botea, 2011, p. 2), making them viable to be used in video games.

To utilise the CPD as a heuristic (Bono et al., 2019), the CPD moves are followed from start to target. The nodes that are traversed along the way to the target are cached for retrieval in the future, as reading from the CPD directly each time is costly.

The CPD heuristic is perfect on the static map. However when the map changes the CPD information is no longer accurate. CPD heuristic tries to respond to this by exploiting the CPD information. When the CPD is changed CPD search is CPD heuristic in combination with A*. When a start and target pair is queried from the heuristic, it returns two values,

1. The lower bound (lb), the path cost between start and target on the original graph
2. The upper bound (ub), the actual cost between start and target on the changed graph

These values are then stored in the CPD cache for fast retrieval.

In practice CPD heuristics work well, as the heuristic provides strong upper and lower bounds for sub-optimal pathfinding. CPD search can be terminated early in bounded sub-optimal search when $ub(incumbent) + g(incumbent) \leq \epsilon \times f_{min}$, where the incumbent is the node with the best tentative path. However CPD heuristics weaken and have greater back-filling behaviour when more obstacles are added to the map similarly to the octile heuristic. CPD heuristics also become inadmissible if obstacles in the original map are removed losing its sub-optimality bounding guarantees.

4 Methodology

4.1 THE IDEA

The issue with Weighted A* and CPD search with early termination is back-filling when obstacles are added into the map. We introduce *wall following*, to bypass perturbed walls and continue the search closer to the target.

We also exploit the CPD heuristic by finding improvements to the cost of nodes by following CPD paths. However this can be expensive if done constantly to all nodes. Therefore we introduce branching nodes to reduce the amount of times improvements are done.

4.2 TECHNICAL DESCRIPTION

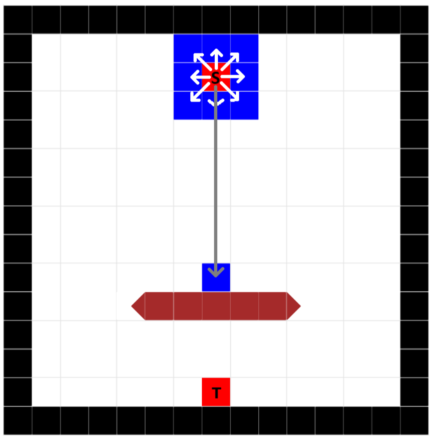
4.2.1 *CPD successor*

CPDs provide perfect heuristic information when the graph is not changed. However in the dynamic setting, the graph is altered. However CPDs can be used to skip large parts of the map which have not been changed, and search closer to the obstacle.

In a normal grid expansion, a node has up to 8 successors. We propose adding an extra successor to a node: the CPD successor. The CPD successor is the node that is generated by following CPD moves from a given node to target until an edge is perturbed – see fig. 4. We interchangeably also use the term perturbed node to refer to the node of the CPD successor. The CPD successor is generated if it is reached with more than one step. This is because a CPD successor generated within one step is generated normally through expansion.

By generating the node next to the perturbation, the CPD can avoid expansions that are not useful leading up to the perturbation. These nodes leading up to the perturbation are only expanded when it is necessary to raise f_{min}.

FIGURE 4: The start node has 8 grid successors in white, but also a CPD successor denoted by the grey arrow.



4.2.2 *Focal Search*

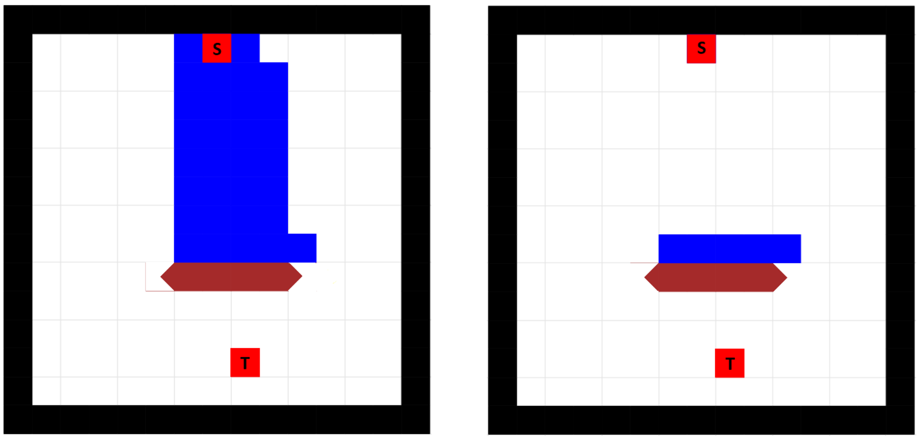
To find paths faster, we utilise an altered FOCAL search. In our version of FOCAL search, FOCAL and OPEN both have priority function $f(n) = g(n) + h(n)$. However, instead of storing nodes less than $f_{min} \times \epsilon$, FOCAL only stores nodes along newly constructed walls, and nodes along in-built walls if one of the node's ancestors was along a newly constructed wall. To ensure solutions are bounded sub-optimal, nodes are only expanded from focal if the lowest f from focal is within the bound of $f_{min} \times \epsilon$. Otherwise, f_{min} is increased by expanding from OPEN.

Algorithm 3: Focal Search: Wall Following

```
OPEN = {s}
FOCAL = {}
incumbent ← s
while OPEN ≠ ∅ do
    $f_{min}$ ← f(head(OPEN))
    if f(head(FOCAL)) ≤ $\epsilon \times f_{min}$ then
        n ← head(FOCAL)
        OPEN ← OPEN \ {n}
        FOCAL ← FOCAL \ {n}
    else
        n ← head(OPEN)
        OPEN ← OPEN \ {n}
        FOCAL ← FOCAL \ {n}
    if isGoal(n) then
        Return the path
    else if $\epsilon \times f_{min} \geq ub$(incumbent) then
        Return the path
    else
        for each n' ∈ succ(n) do
            if ub(incumbent) > ub(n') then
                incumbent ← n'
            OPEN ← OPEN ∪ {n'}
            if n' is next to a wall then
                FOCAL ← FOCAL ∪ {n'}
            end
        end
end
```

The advantage of wall-following is to only explore around the obstacle. From wall-following, we can quickly find nodes that go around the obstacle, compared to Weighted A* and CPD search with early termination – see fig. 5.

FIGURE 5: This shows the nodes expanded during search for focal and CPD search with early termination. The left side shows expansions done by CPD search with early termination. The right shows expansions done by focal search.



4.2.3 *CPD String Pulling*

We also propose CPD string pulling. As the CPD finds optimal paths on an unperturbed map, the CPD can be queried to find if the CPD path is an improvement over the current path found by search. We refer to this as string pulling.

To do so, we introduce *shoot nodes*. A shoot node is any node where there is a CPD successor from the node to the target. If a CPD path from the current node to its shoot node has an improved cost compared to its normal path, the current node's parent is changed to the shoot. Shoot nodes are passed from parent to child.

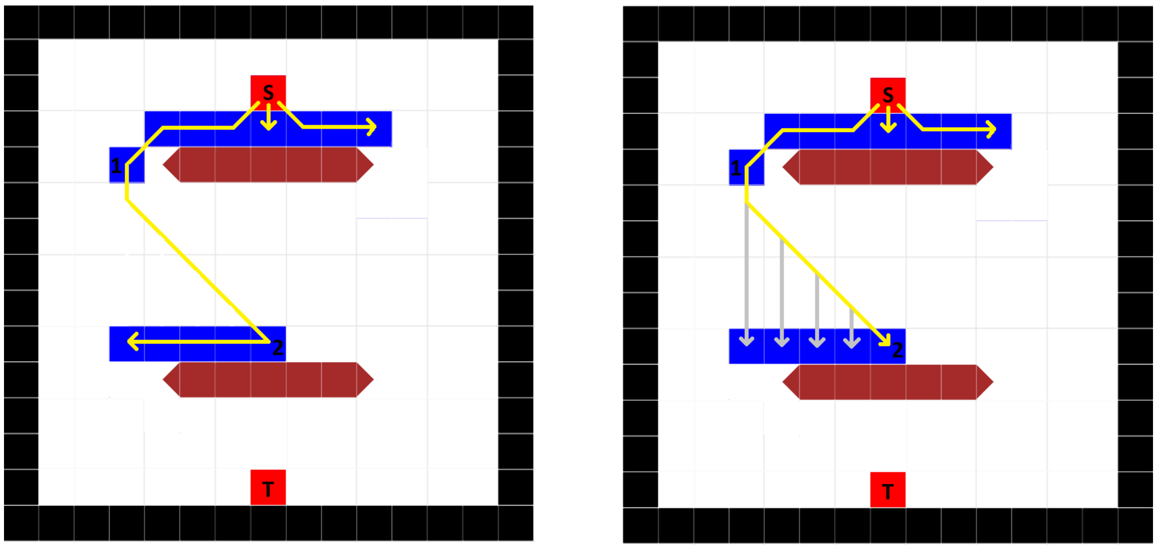
Algorithm 4: Focal Search: String Pull Update Value

```
if $n'$ was generated by being the CPD successor of $n$ then
    shoot(n') ← n
else
    shoot(n') ← shoot(n)
    if $g$(n) > $g$(shoot(n)) + $ub$(n', shoot(n)) then
        $g$(n) ← $g$(shoot(n)) + $ub$(n', shoot(n))
        parent(n') ← shoot(n);
```

By doing this, nodes that are not obstructed by newly perturbed obstacles have drastically improved f values and a good path can be found faster.

The advantages of string pulling is that paths found are more natural. With a wall-following search, a path found within the bound will suffer from wall hugging without string pulling. Wall hugging paths looks unnatural to a player of a game compared to diagonal paths that skip the wall. – see fig. 6.

FIGURE 6: This diagram shows paths found by wall-following without string pulling (left) and wall-following with string pulling (right). The shoot node is (1) with CPD successor node (2).

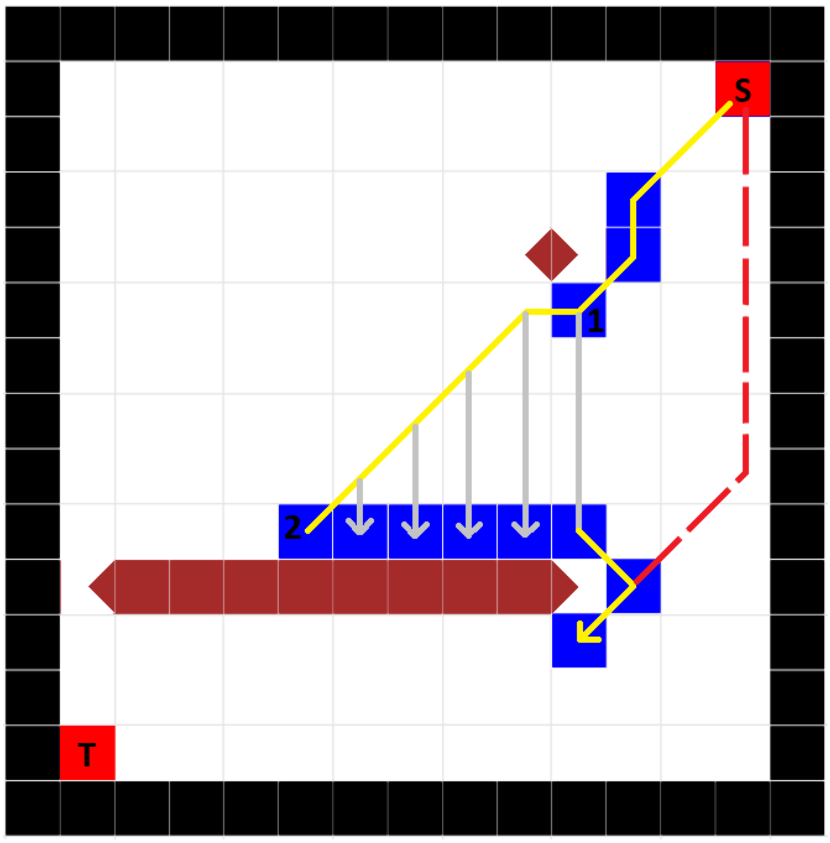


Another advantage is to reduce re-expansions. As nodes have their g-values improved with string pulling, the node has a less chance of being improved and re-expanded. This leads to less work for the search.

However string pulling has disadvantages. For string pulling to work efficiently assumes that backwards and forwards traversals are equivalent. String pulling is done backwards for speed, as the CPD cache can be relied on. This holds for grid maps, as backward and forward edge costs are equivalent. However this does not hold on road networks, as one side of the road can be more costly to traverse compared to the other.

String pulling is also not optimal. With additional wall following, previous nodes can be visible again – see fig. 7.

FIGURE 7: This diagram shows nodes in wall-follow search with string pull and the paths in yellow with string pulls in grey to reach each expanded node found by search. The shoot node is node (1), which has a CPD successor (2). The optimal path is shown in dotted red. When the search reaches the end of the perturbed obstacle, instead of string pulling to the start, it string pulls to its shoot which is more expensive.



4.2.4 *Branching Nodes (Wall expansion)*

We introduce branching nodes to make search faster by not string pulling as often and to skip over large amounts of walls which do not lead closer to the target. A branching node is a node along a wall where:

1. wall-following from the node lead to multiple possible wall successors that do not backtrack – see fig. 8; or,
2. the node and its parent do not have the same perturbed node and the node has a CPD successor – see fig. 9.

Similarly to the CPD expansion, we introduce wall-following expansions to find branching nodes. In a wall following expansion if the node is expanded from FOCAL, there is an additional successor for each edge of the node if that edge leads to a node along a wall. The edge is wall-followed until a branching node. These wall-followed nodes are added onto FOCAL, instead of all wall nodes being added to FOCAL. Nodes generated by being CPD successors are still added into FOCAL.

String pulls are only done on the branching nodes, reducing the amount of times the CPD is queried to retrieve an optimal path. This should speed up the algorithm.

If perturbed nodes do not allow corner cutting, a solution is to identify if the node is next to the corner of the perturbation. If so, the node is a branching node and put on to FOCAL.

FIGURE 8: The wall-following successors generated after expanding the centre blue node. These wall-following successors are branching as they have multiple successors next to walls that do not backtrack. These successors are shown by the green arrows.

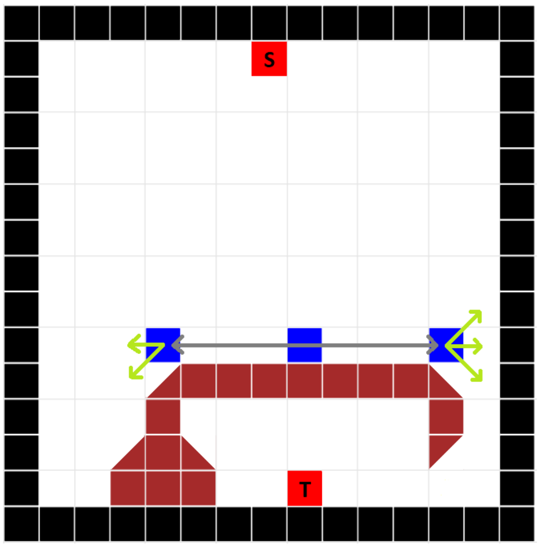
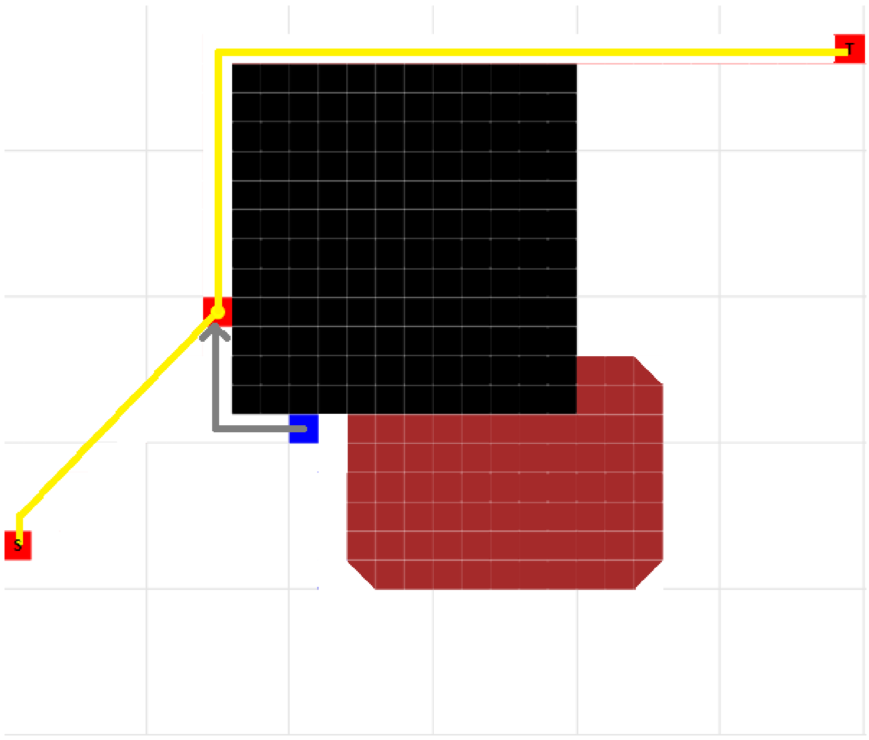


FIGURE 9: The wall-following grey arrow generates the red node from the bottom blue node. This is because all the nodes leading up to the red have CPD paths leading back into the same perturbed node as the blue node. The red node is generated as it's CPD path does not lead to the same perturbed node as the blue node. The red node also now has a CPD path towards the target, it is string pulled from the start and a path can be returned.



5 Results

5.1 MAPS

We evaluate different types of searches with various maps from the MovingAI Database (Sturtevant, 2012). Each map consists of a grid and start and target pair instances to solve. The maps used are:

DRAGON AGE ORIGINS, DRAGON AGE 2: These benchmarks represent uniform-cost maps in video games.

WARCRAFT 3: These maps are representative of weighted cost game maps. In Warcraft 3 there are varying types of terrain given different traversal multipliers: Ground 1.0, Trees 1.5, Shallow water 2.0, and Water 2.5. When traversing between one terrain type to another terrain type, the edge is weighted by the average of the multipliers between the two terrains.

ROOMS: The Rooms benchmark are 512x512 maps having square rooms of size 8x8, 16x16, 32x32 and 64x64. With a probability of 0.8, a random cell between each room is unblocked.

MAZES: The Mazes benchmark are 512x512 maps. Each map is constructed by extending from a unblocked centre cell and branching out by extending corridors. The width of a corridor on each map can be 1, 2, 4, 8, 16 and 32.

5.2 ALGORITHMS

We undertake two experiments between these search algorithms.

OPTIMAL: CPD search with early termination.

WEIGHTED: Weighted CPD search

FOCAL: Focal search with wall following and string pulling at all nodes

NO SP: Focal search with wall following and no string pulling

BRANCHING: Focal search with focal only containing branching nodes and CPD successors

These pathfinding algorithms were developed from C++, and the base codes are available from https://bitbucket.org/dharabor/pathfinding/src/master/.

5.3 EXPERIMENTS

We undertake two experiments between these search algorithms using the hard perturbation policy. The test machine had an i5-7400 with 16GB memory.

HARD PERTURBATION POLICY: For a hard perturbation, the node being perturbed has its edges become not traversable. For the hard obstacle perturbation, a central node for the hard perturbation is picked 3/4 along the unperturbed path. From the central node, all nodes are hard perturbed within a radius of min(15, path length/16).

EXPERIMENT 1: All symmetrical optimal paths on the unperturbed path are blocked using the hard obstacle perturbation policy. The experiments are done with $\epsilon = 1.10$, solutions are accepted if the solution is within 10% of the optimal cost.

EXPERIMENT 2: The hard obstacle perturbation policy is used until the optimal path cost on the perturbed map is 15% greater than the optimal path cost on the unperturbed map. The experiments are done with $\epsilon = 1.10$.

For both experiments, hard perturbations that cause an instance to not have a solution are removed. We assume that a connectivity test is done first before undertaking search for all algorithms.

5.4 RESULT TABLES

Benchmark	Instances	Algorithm	Time (ms)				Nodes Expanded			
			Q1	Median	Q3	Mean	Q1	Median	Q3	Mean
DAO	86168	Opt	0.1444	1.0284	4.4480	3.6352	224	1716	6933	5656.0434
		Weighted	0.0883	0.6045	2.5364	2.7777	82	674	3637	4873.9047
		No SP	**0.0515**	**0.2677**	1.0155	2.1336	10	32	296	3216.0227
		Focal	0.0684	0.3565	1.1968	1.9807	16	52	238	2322.7684
		Branching	0.0648	0.2891	**0.9688**	**1.0916**	**5**	**10**	**48**	**782.5864**
WC3	55256	Opt	0.7952	2.8244	5.9796	4.0559	814	3261.5	7062.25	4652.0796
		Weighted	0.4336	1.231	2.250	1.7058	210	855	2317	2046.64
		No SP	**0.2151**	**0.4946**	**0.7640**	0.6000	15	30	50	148.8786
		Focal	0.2981	0.6344	0.9538	0.7101	21	38	53	71.3175
		Branching	0.2284	0.5081	0.7701	**0.5620**	**4**	**5**	**6**	**36.7413**
DA2	34154	Opt	0.1071	0.5411	2.0530	1.4639	173	1031	3755	2651.0535
		Weighted	0.0693	0.3638	1.4655	1.3507	73	553	2844	2659.4576
		No SP	0.0421	0.1711	1.0373	1.1076	10	35	1857.75	1872.5753
		Focal	0.05637	0.2152	0.8537	**1.0617**	16	60	665	1426.0440
		Branching	**0.0419**	**0.1294**	**0.6514**	1.0765	**5**	**15**	**128**	**1197.9411**
Rooms	72176	Opt	1.1908	3.5281	8.0156	6.5631	1182	4220	10063.25	8239.8466
		Weighted	0.8667	1.7838	3.9868	4.4759	536	1622	5255	5992.7210
		No SP	**0.5031**	**0.8632**	**2.7548**	**3.9119**	45	179	3434	4752.4454
		Focal	0.6733	1.5899	4.8110	4.7323	88	487	1880	2754.9014
		Branching	0.5756	1.0969	3.1325	4.2567	**9**	**44**	**326.5**	**2423.3055**
Mazes	59041	Opt	2.2209	13.4103	29.5315	18.4086	3406	19442	39507	24356.0220
		Weighted	1.2948	4.0600	6.8108	4.6814	1080	3543	8406	6230.3124
		No SP	**0.4349**	**1.0649**	**1.5982**	**1.0756**	27	52	70	136.4209
		Focal	0.5393	1.2880	1.8858	1.3110	41	73	120	151.0278
		Branching	0.6374	1.1552	1.6056	1.1648	**6**	**8**	**10**	**61.0495**

TABLE 1: Results for Experiment 1

Benchmark	Instances	Algorithm	Time (ms)				Nodes Expanded			
			Q1	Median	Q3	Mean	Q1	Median	Q3	Mean
DAO	52994	Opt	0.2257	1.5472	**6.3717**	**5.1855**	440	2890	11292.75	8955.6108
		Weighted	0.2064	1.5988	6.9065	5.5735	416	3230	13804	10923.2367
		No SP	**0.2016**	**1.4428**	6.4694	5.4374	346	2482.5	10782.5	8859.0063
		Focal	0.2288	1.6594	7.2204	5.9881	297	2058	9009	7362.8974
		Branching	0.2900	1.8802	7.8538	6.4591	**290**	**2036**	**8949**	**7321.8748**
WC3	54930	Opt	4.5235	17.4584	37.9798	24.7890	6098.5	22887.5	48050.75	31205.7816
		Weighted	4.5100	18.7757	42.9921	29.3423	7459.5	33034	78956.75	56318.98
		No SP	**3.2454**	**12.2471**	**26.8529**	**18.0109**	4193	15643	32799	21528.5831
		Focal	3.8315	14.1875	30.5568	20.7191	3364.25	12477.5	26143	17384.1859
		Branching	3.6523	13.0869	27.4052	18.5209	**3312.25**	**12277**	**25626.75**	**17042.6905**
DA2	20293	Opt	0.1659	0.7730	**2.7495**	**1.7637**	321	1611	5398	3499.3343
		Weighted	**0.1478**	**0.7508**	2.8225	1.9345	302	1715	5970	4192.1684
		No SP	0.1511	0.7721	2.9311	1.9159	250	1512	5393	3548.0450
		Focal	0.1671	0.8662	3.3205	2.1486	218	**1282**	4578	3064.7706
		Branching	0.2152	1.0845	3.9170	2.4945	**212**	1283	**4548**	**3051.4203**
Rooms	64020	Opt	5.8221	22.9227	52.6085	32.6960	9185.75	33837	73443	45571.0538
		Weighted	**5.3547**	**20.8479**	**48.4040**	**30.6577**	8896.5	33521	75906	48484.0258
		No SP	5.5204	21.8831	53.3117	38.3008	8238.75	31595.5	73989.5	60854.28
		Focal	7.4893	30.9623	72.2652	45.9588	6706	22787	48361.5	31186.7727
		Branching	8.0138	30.9655	70.7227	45.9739	**6061**	**20656**	**43900.75**	**28403.4312**
Mazes	8154	Opt	0.0864	0.4591	1.7051	1.4543	144	875	3104.75	2510.8889
		Weighted	**0.0772**	0.4367	1.7110	1.5349	130.25	857	3264.5	2929.3164
		No SP	0.0792	**0.3862**	**1.3670**	**1.1956**	109	634	2195	1818.6837
		Focal	0.0884	0.4383	1.5742	1.3960	97	554	1957.75	1644.9327
		Branching	0.1152	0.5105	1.6654	1.4536	**94**	**539**	**1927**	**1621.9207**

TABLE 2: Results for Experiment 2

6 Discussion

In the results, we observed the time and nodes expanded of different searches for the experiments. The experiments were set up to show advantages of wall-follow searches to avoid the back-filling behaviour which happens with Weighted CPD Search and CPD Search with early termination.

For experiment 1, we see that no string-pull search performs the best across most benchmarks. However, branching search often leads to lower expansions for harder test cases, as string pulling helps reduce the upper bound and going straight to the branching nodes helps the search skip over walls and resume along the CPD path. The lower expansions for harder cases is reflected in lower search times for branching search and focal search for Dragon Age 2 benchmarks and Dragon Age Origins benchmarks. In terms of expansions, branching search performs the best, however it does not always performs the best on time. This is because branching search does more work through wall-following to generate successors. Optimal search with early termination and weighted do not perform as well, as they suffer from back-filling on most instances. –see fig. 10.

The cases that wall following searches perform evenly or worse can be seen in experiment 2. In experiment 2, the optimal paths are greater than the initial bound of 10%. This means that it is necessary to back-fill by expanding from OPEN. This leads to all searches doing a closer amount of expansions compared to experiment 1. –see fig. 11 However branching still expands the least amount of nodes across all instances, but the gain in performance from wall-following expansions drops dramatically. Optimal and weighted perform the best, with no string pulling doing better only for Warcraft 3 and Mazes.

An interesting finding is there are exceptions where string pulling leads to more expansions than no string pulling. This is because when string pulling, the f of nodes is decreased. This leads to more back-fill wall expansions from FOCAL until the bound is exhausted. With no string pulling, the f of nodes is not decreased by string pulling and thus no back-filling occurs. However in majority of cases, FOCAL expands less nodes compared to no string pulling because focal nodes have lower cost so less expansions are needed before a suitable path can be found.

FIGURE 10: Search on arena.map: Optimal search on the left expands a lot of nodes to find a solution, whereas branching search on the right only expands a few nodes to find a solution. (Expansions are in blue, the start is on the bottom left).

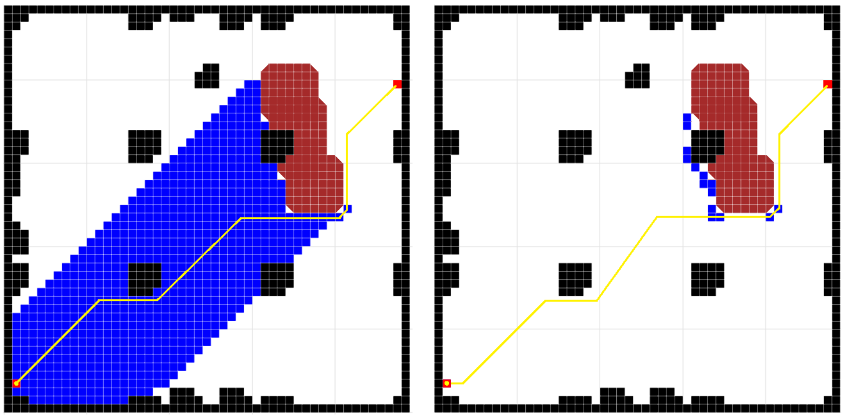
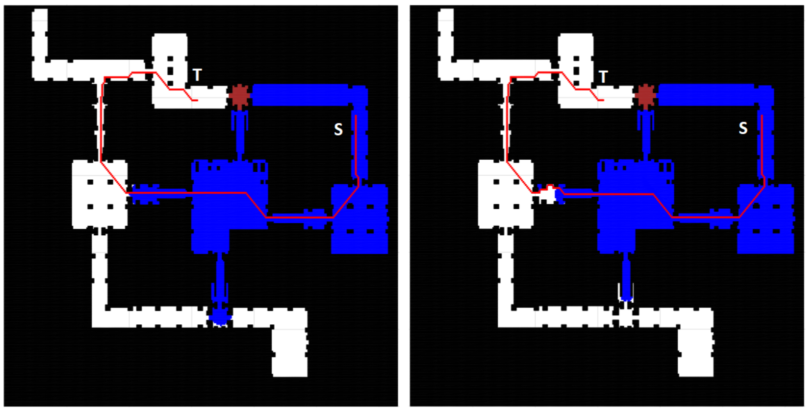


FIGURE 11: Search on dr_slavers.map: Optimal search on the left and branching search on the right both expand a similar amount of nodes as the corridor that the optimal path on the unperturbed map could go through is completely blocked. (The optimal path on the unperturbed map goes up and to the left to the target). The optimal cost on the newly perturbed map is a lot greater than 10% on the base map and thus a lot of expansions from OPEN are necessary to raise the lower bound.

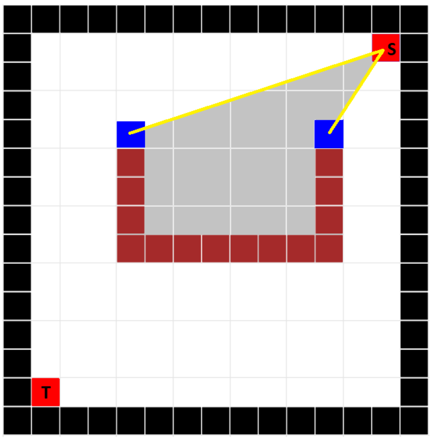


6.1 FUTURE WORK

6.1.1 *Pruning nodes*

An idea is to utilise wall following to prune nodes with triangle reasoning.

FIGURE 12: With triangular reasoning, grey nodes can be pruned as they cannot lie on the optimal path. This rapidly raises f_{min}.

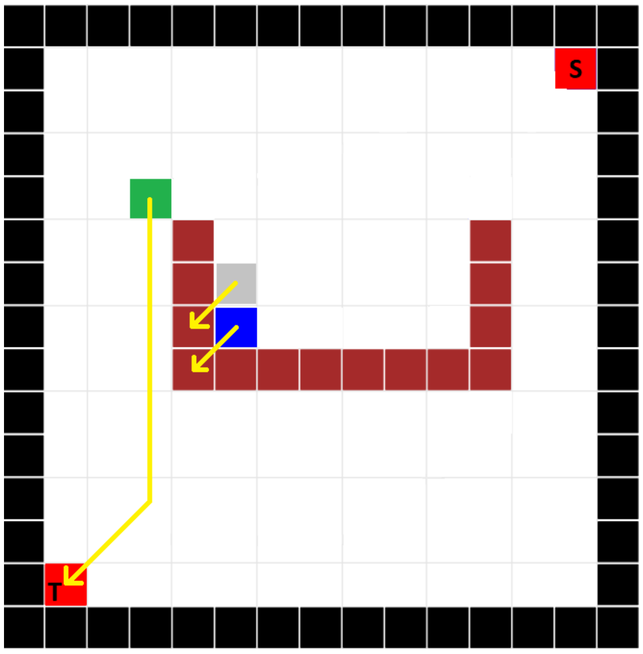


By using triangle reasoning, expansions can be reduced as we can attempt to reason that the area in the triangle is unfeasible and cannot lie on an optimal path. This can help with getting out of concave obstacles which is a big weakness for searches.

6.1.2 *Improved String Pulling and Identification of Branching Nodes*

In concave obstacles, string pulling is done constantly as the CPD path can be followed. However the CPD path leads straight back into the perturbation and therefore string pulling was unnecessary from that node. The node also did not needed to be added into FOCAL. For better string pulling and search, string pulls and adding the node to FOCAL should only occur when a node's CPD path leads into a perturbed area not seen before in the search.

FIGURE 13: In current implementation of branching search, the node in grey is the wall following successor, however its CPD path in yellow leads straight into the same perturbation. Wall following should continue upwards until the green node is reached where the CPD path does not lead into the same perturbation.



6.1.3 *Soft Obstacles*

Another problem that still needs to be approached is dealing with terrain changes. In video games, part of the map can change to swamp or water and become harder to traverse. An idea to attempt the problem is to use FOCAL search and use the maximum edge multiplier as the priority function. The maximum edge multiplier for a node is the maximum multiplier between all of the nodes' unperturbed edge cost and perturbed edge cost. In this way, the search will be led into nodes that are less perturbed before going to nodes that are heavily perturbed.

7 Conclusion

We introduce improvements to CPD Search to deal with the dynamic pathfinding problem with hard obstacles. We use new bounded sub-optimal methods to lower upper bounds instead of raising lower bounds by trying to wall-follow around the obstacle.

These methods work well for test instances where there exists a solution within the initial sub-optimality bound. However we need to do more work in the case where we have to raise the lower bound to find a valid sub-optimal solution.

Future work lies in using triangular reasoning to rapidly raise the lower bound and prune nodes that can't lie on the optimal path to be able to do better than early termination and weighted CPD searches. The definition of branching node can also be improved to string pull less, by not having a branching node if it's CPD move leads into the same perturbed area.

References

Bono, M., Gerevini, A. E., Harabor, D. D., & Stuckey, P. J. (2019). Path planning with CPD heuristics. *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*. https://doi.org/10.24963/ijcai.2019/167

Botea, A. (2011). Ultra-fast optimal pathfinding without runtime search. *Proceedings of the Seventh AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE'11)*, 122–127

Botea, A., Bouzy, B., Buro, M., Bauckhage, C., & Nau, D. (2020). Pathfinding in games. http://dx.doi.org/10.4230/DFU.Vol6.12191.21

Cohen, L., Greco, M., Ma, H., Hernandez, C., Felner, A., Kumar, T. K. S., & Koenig, S. (2018). Anytime focal search with applications. *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence*. https://doi.org/10.24963/ijcai.2018/199

Dechter, R., & Pearl, J. (1985). Generalized best-first search strategies and the optimality of A*. *Journal of the ACM*, *32*(3), 505–536. https://doi.org/10.1145/3828.3830

Hansen, E. A., & Zhou, R. (2007). Anytime heuristic search. *Journal of Artificial Intelligence Research*, *28*, 267–297. https://doi.org/10.1613/jair.2096

Harabor, D. (2014). *Fast and optimal pathfinding* (Doctoral dissertation). The Australian National University

Hart, P., Nilsson, N., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, *4*(2), 100–107. https://doi.org/10.1109/tssc.1968.300136

Pearl, J., & Kim, J. H. (1982). Studies in semi-admissible heuristics. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *PAMI-4*(4), 392–399. https://doi.org/10.1109/TPAMI.1982.4767270

Pohl, I. (1970). Heuristic search viewed as path finding in a graph. *Artificial Intelligence*, *1*(3-4), 193–204. https://doi.org/10.1016/0004-3702(70)90007-x

Sturtevant, N. R. (2012). Benchmarks for grid-based pathfinding. *IEEE Transactions on Computational Intelligence and AI in Games*, *4*(2), 144–148. https://doi.org/10.1109/tciaig.2012.2197681

Sturtevant, N. R., Sigurdson, D., Taylor, B., & Gibson, T. (2019). Pathfinding and abstraction with dynamic terrain costs. *Proceedings of the Fifteenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE-19)*, 80–86

Wilt, C., & Ruml, W. (2012). When does weighted A* fail? *Proceedings of the 5th Annual Symposium on Combinatorial Search, SoCS 2012*, 137–144