Proceedings of the Twentieth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE 2024)

Abstraction and Path Computation for Video Game Path Finding with Changing Maps

Teresa Saller[1], **Ramon Lawrence**[1], **Vadim Bulitko**[2]
[1] University of British Columbia
[2] University of Alberta
tsaller@alum.ubc.ca, ramon.lawrence@ubc.ca, bulitko@ualberta.ca

Abstract

Efficient grid-based path finding is important in video games especially for larger maps and when moving many agents. Algorithms based on abstraction have an order of magnitude faster search time performance than A* at the cost of a small amount of memory and increased suboptimality. Some approaches also compute and store paths to improve search performance. This paper evaluates new and existing algorithm variants for abstraction and path computation and investigates their performance for video game path finding with map changes. The results show that abstraction has significant advantages over A* and can be implemented efficiently for changing maps. Computing, storing, and reusing paths also has benefits especially when several searches can be performed before the map changes.

Introduction

Supporting games with larger worlds and more interacting agents requires efficient path finding approaches. Although A* implementations guarantee optimal paths and can be highly optimized, approaches that use abstraction and precomputation can be faster than A* by an order of magnitude. PRA* (Sturtevant 2007) uses state abstraction and planning in the abstract space to speed up searches. Other approaches such as HCDPS (Lawrence and Bulitko 2012) and DBA* (Lee and Lawrence 2013) combine abstraction with path precomputation to further improve online search performance. It is an open question if path precomputation has significant benefits compared to using only abstraction.

The first contribution of this paper is quantifying the benefit of computing and reusing paths for video game path finding in an abstract space (Sturtevant 2007; Bulitko et al. 2007). Prior work (Lee and Lawrence 2013; Lawrence and Bulitko 2012) has shown benefits as the precomputed paths can be reused many times without requiring recomputation. However, there has not been a detailed analysis and experimentation on these benefits and tradeoffs.

The second contribution is an evaluation of several approaches for efficient recomputation of the abstraction and paths in the presence of map changes during gameplay. Approaches are compared to determine when path precomputation is beneficial based on the frequency of map changes.

Problem Formulation

The problem studied is grid-based path finding for games with a square grid and diagonal movement, which is a graph search problem $(G, s_{\text{start}}, s_{\text{goal}})$ where search graph $G = (S, E, c)$ consists of a set of states S, a set of edges E, and a cost function c. Each state s_i is connected to up to eight neighbour states with an edge $(s_i, s_j) \in E$ and a *cost* $c(s_i, s_j) > 0$. The edge costs are 1 for cardinal moves and 1.4 for diagonal moves. States can be represented as a two-dimensional matrix M. State s in matrix location $M[i][j]$ is at row i and column j in the 2D grid. If $M[i][j] = 0$, s is an open state and can be navigated to, otherwise $M[i][j] = 1$ indicating the state is a wall that cannot be navigated to.

The *neighbourhood* of s is $N(s) = \{s' | (s, s') \in E\}$, and state s is *expanded* when an algorithm computes $N(s)$. A solution to a search problem from start state s_{start} to goal state s_{goal} is a *path* $p = (s_{\text{start}} = s_0, s_1, s_2, ..., s_n = s_{\text{goal}})$ such that any pair of states $(s_i, s_{i+1}) \in E$. The *path cost* is the sum of all edges on the solution path.

Abstraction algorithms take the graph G and abstract it into $G' = (S', E', c')$ to reduce the search space size. A state $s' \in S'$ is referred to as a *region representative* for a set of states in S. An edge $e' = (s'_i, s'_j) \in E'$ with cost $c'(s'_i, s'_j) > 0$ is defined when building the abstraction and connects the two neighbor regions with representatives s'_i and s'_j. Edge e' is expanded as a path in G connecting s'_i and s'_j. A path may be precomputed and cached to solve multiple search problems.

Algorithms are evaluated on the *abstraction time* and the number of *states expanded* to produce G' from G and the *memory* to store any precomputed paths between region representatives. When a map changes due to a wall addition or removal, time and states expanded are measured.

Search performance is measured using the number of states expanded, time, and path cost. *Suboptimality* is defined as the ratio of the path cost found by the algorithm to the optimal solution cost minus one and times 100%. An optimal path has 0% suboptimality, and a path with twice the cost as the optimal path has suboptimality of 100%. For changing maps, algorithms are evaluated on *overall time*, which is the sum of all path search times and the time to update the abstraction and precomputed paths.

Background

Approaches using abstraction require additional time and memory to compute and store the abstraction. Abstraction may be based on cliques (Bulitko, Sturtevant, and Kazakevich 2005; Bulitko et al. 2007), sectors (Sturtevant 2007), or other reachability metrics such as all states that are reachable via hill-climbing from a region representative (Lawrence and Bulitko 2012). Algorithms may also use precomputation and storage of paths in an offline fashion that can be used for online search, which trades space for time.

Abstraction Approaches

PRA* (Sturtevant 2007) improves on A*'s (Hart, Nilsson, and Raphael 1968) search time and states expanded by abstracting the search space into sectors and first computing a complete solution in the abstract space. The map is divided into sectors which are squares in the map of a given size (such as 16×16). All states that are reachable without leaving the sector are in a region. A sector may have multiple regions. Each region has a region representative.

Once a path is computed in the abstract space, a path in the search space is built using A* searches between region representatives. Combining these paths results in a path from start to goal. Due to this connecting of paths between region representatives, the paths found can be suboptimal. Some optimizations are possible to reduce this suboptimality. PRA* was applied to maps with dynamic terrain (Sturtevant et al. 2020) with superior performance compared to using weighted A*.

Path Precomputation Approaches

A variety of approaches have used path precomputation and storage in addition to abstraction. By precomputing paths and reusing them during search, it is possible to further reduce the time and suboptimality. HCDPS (Hill Climbing and Dynamic Programming Search) (Lawrence and Bulitko 2012) defines abstract regions where all states are bidirectionally hill-climbable with the region representative. It then built upon this abstraction a compressed path database between all region representatives. Compared to PRA*, this avoids A* searches between region representatives as all path finding is done using hill-climbing, which reduces time and memory use. HCDPS is faster than PRA* with improved suboptimality but its stored paths consume more memory.

DBA* (Lee and Lawrence 2013) adopted the PRA* sector regioning and combined it with the path computation between region representatives used by HCDPS. When evaluated on Dragon Age maps (Sturtevant 2012) the average suboptimality was about 3% while using less than 200 KB of memory. DBA* and HCDPS used path precomputation and were not evaluated on changing maps.

The technique of path compression using hill-climbable subgoals originated in kNN LRTA* (Bulitko, Björnsson, and Lawrence 2010) and was utilized in other work (Lawrence and Bulitko 2012; Lee and Lawrence 2013). Given a path, it is produced by creating subgoals along the path that guarantee a greedy algorithm will reconstruct the path by navigating to each subgoal. The storage size depends on the number of subgoals and is often an order of magnitude reduction compared to the number of states on the path.

LRTA* with subgoals (Hernández and Baier 2011) precomputes a subgoal tree from each goal state where a subgoal is the next target state to exit a heuristic depression. Online, LRTA* will be able to use the subgoal tree to escape a heuristic depression.

Algorithm Variants

Several abstraction and path computation algorithms are empirically compared in this paper.

PRA* (Sturtevant 2007) performs abstraction using sectors and performs no path precomputation. On map changes (adding or removing walls), it recomputes the abstraction only within the affected sector. This requires one or more breadth-first searches within the sector. The states expanded are bounded by the sector size.

PRA* with path caching (PRA*-P) (Li 2016) adds caching of paths to PRA*. Path finding in the abstract space is still performed as previously, but before paths between region representatives are refined, the cache is queried to check whether the refined path has been previously computed and stored. If so, it can be reused. Otherwise, it will be computed and stored.

PRA* with hill-climbing compressed path caching (PRA*-C) is a new variant presented in this paper that instead of caching full paths, caches the compressed paths as a series of hill-climbable subgoals. Storing the subgoals between two region representatives reduces the memory for storing a complete path while allowing hill-climbing rather than A* to refine the path.

DBA* (Lee and Lawrence 2013) is modified to support efficient path recomputation. Its path database is precomputed offline as before, but the algorithm is extended to recompute only the paths connecting neighbouring region representatives of a region where a map change occurred.

Online path finding is similar for all algorithm variants. Given a start cell s and goal cell g, the region representatives for the start, R_s, and for the goal, R_g, are determined as each map cell stores its region number. Each algorithm computes a path in the abstract space between the region representatives R_s and R_g. This produces a path connecting a series of region representatives $R_s, R_1, R_2, ..., R_n, R_g$. The only difference between the algorithms in the abstract space search is that DBA* has exact path (edge) costs between region representatives while PRA* uses a heuristic.

In the map space, A* is used to find the path between s and R_s and R_g and g. Algorithms vary on how paths are found between each pair of region representatives (R_i, R_{i+1}). PRA* uses A* and the variants using caching may have a cached path to use to avoid this search. DBA* always has a hill-climbing compressed path to perform greedy search to navigate. The complete path is found by concatenating the path between s and R_s, the individual paths between region representatives, and R_g and g.

Optimizations can be applied to reduce suboptimality and smooth paths when concatenating smaller paths to produce the overall path. Path trimming (Sturtevant 2007) removes

the last 10% of states in each concatenated path segment and then plans from the end. Increasing the neighbourhood depth (Lee and Lawrence 2013) allows for computing paths that are more than one step away, which reduces the number of concatenated paths. Start and end path optimizations (Lawrence and Bulitko 2012) reduce the path length at the start and end of the path by planning towards a more distant subgoal rather than the start or end region representative. As these optimizations can be consistently applied to any algorithm variant, they are not explored in this work.

Optimizing Recomputation

Abstracting the search space into sectors is performed by one or more breadth-first searches (BFS). Given the fixed-sized sectors (e.g. 16×16 or 32×32), the states expanded to compute an abstraction is fixed based on the sector size. Each state is expanded only once by a single BFS. Larger sectors reduce the amount of memory for the abstraction, but result in paths with more suboptimality and take longer to compute, which is important for map changes. An example map abstraction with 16×16 sector size is in Figure 1. The cells in white are region representatives.

Figure 1: Region abstraction with 16 x 16 sector size.

If there is a wall addition or removal, only the sector where this change occurred potentially needs to be rebuilt. Any optimization must be very efficient as recomputing the abstraction has a bounded cost. Figure 3 contains pseudocode with our check to avoid recomputing a sector abstraction on wall addition. The challenge is efficiently detecting if a wall addition will partition a sector causing the creation of a new region. Our approach determines the number of walls in the eight neighbour states around the added wall and will skip the abstraction if a partition is not possible. If any neighbour state is in another region, it is considered a wall. If a wall is added on a region representative, the abstraction may not need to be performed but a new region representative must be selected.

When removing a wall, the abstraction does not need to be recomputed unless the removal may either add a new neighbour region or allow two regions in a sector to be merged.

Figure 2 shows using a color heat map the impact of placing a wall at that location in terms of the amount of recomputation required. Highly impacted cells include region representatives and areas where sector partitioning occurs.

Figure 2: Heat map showing impact of wall addition with darker areas requiring more recomputation.

```
1  // Added a wall by setting M[i][j] = 1
2  R1 = M[i-1][j-1]+M[i-1][j]+M[i-1][j+1]
3  R2 = M[i][j-1]+M[i][j]+M[i][j+1]
4  R3 = M[i+1][j-1]+M[i+1][j]+M[i+1][j+1]
5
6  C1 = M[i-1][j-1]+M[i][j-1]+M[i+1][j-1]
7  C2 = M[i-1][j]+M[i][j]+M[i+1][j]
8  C3 = M[i-1][j+1]+M[i][j+1]+M[i+1][j+1]
9
10 // Check if partition
11 if ( (R2 < 2 OR C2 < 2) AND
12       (R1 == 0 OR R2 == 0 OR R3 == 0) AND
13       (C1 == 0 OR C2 == 0 OR C3 == 0))
             THEN
14     No partition so no recomputation
15 else
16     Perform abstraction in sector
17     Recalculate neighbours
```

Figure 3: Check to determine if recomputing the abstraction in a sector is needed when adding a wall in that sector.

If the abstraction within a sector is rebuilt or the region representative is replaced with a wall, then the paths to the neighbouring regions are invalidated and need to be rebuilt. It is unclear how to devise generally applicable tests to determine if a path is still valid that is more efficient than recomputing the path using A*. However, for wall addition, it

is possible to perform a hill-climbing check to see if the path is still valid rather than running A*.

An algorithm marks the path as invalid in its database or cache. Rebuilding the invalid path can be done in an *eager* fashion where the path is rebuilt at the same time as the abstraction is recomputed or in a *lazy* manner where the path is only rebuilt when it is needed for path finding between the two region representatives. The DBA* variant uses eager path rebuilding, while the PRA* caching approaches use the lazy approach.

Experimental Results

The four algorithm variants were evaluated on four *Baldur's Gate II* maps (012, 516, 601, 703) and ten maps (hrt000d, orz100d, orz103d, orz300d, orz700d, orz702d, orz900d, ost000a, ost000t, ost100) from *Dragon Age: Origin* available at http://movingai.com (Sturtevant 2012). The Dragon Age maps have an average number of open states of 96,739 and total cells of 574,132.

Algorithms were implemented using Java 21 and executed on a workstation with an Intel Core i9 5.8 GHz processor with 32 GB of memory. The experimental code and data is at https://github.com/ubco-db/database-pathfinding.

Abstraction and Path Precomputation

The states expanded to compute the abstraction and paths between region representatives for varying sector sizes are shown in Figures 4 and 5. The cost to perform the abstraction using breadth-first search (BFS) is the same for all algorithm variants. DBA* also precomputes paths between region representatives and compresses them into hill-climbable subgoals. The result is that the overall time for precomputation is generally between three to eight times higher for DBA* compared to PRA* and the memory consumed for the paths is between 2 to 30 KB. The abstraction times are shown in Figures 6 and 7.

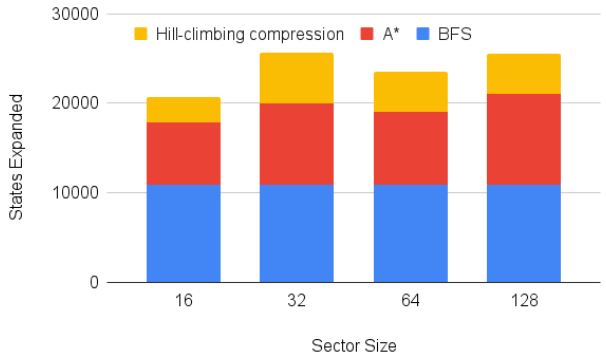

Figure 4: States expanded for abstraction using BFS (all algorithm variants) and precomputing A* paths between region representatives and using hill-climbing compression (DBA* only) for Baldur's Gate II maps.

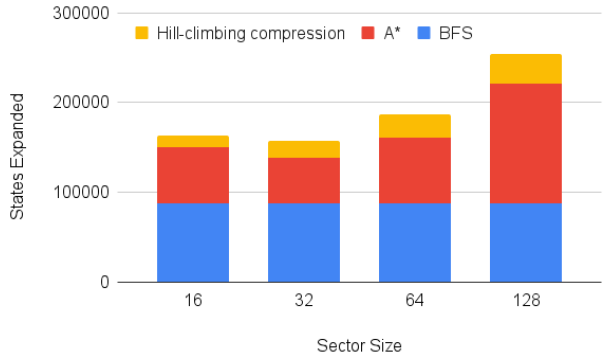

Figure 5: States expanded for abstraction using BFS (all algorithm variants) and precomputing A* paths between region representatives and using hill-climbing compression (DBA* only) for Dragon Age maps.

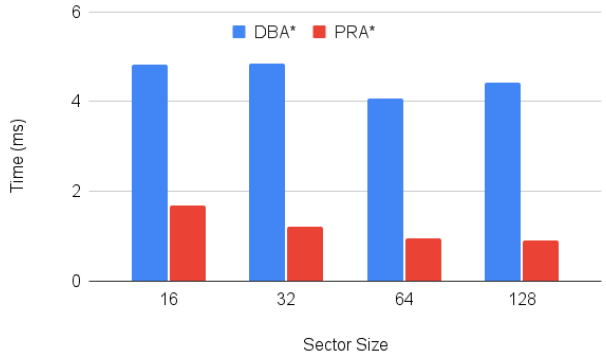

Figure 6: Overall Time to Abstract and Precompute Paths for Baldur's Gate II maps.

Path Reuse

A fundamental question is to determine the potential benefit for storing paths between region representatives when used for path finding. The largest benefit occurs when A* expands many states to find the path. The tradeoff is additional memory usage. Storing the complete paths eliminates any search but consumes more memory while storing hill-climbable compressed paths uses less memory but requires greedy search to rebuild the path.

For both sets of maps, the states expanded and time performed by A* and greedy search between all pairs of neighbouring region representatives was determined. The ratio of dividing A* time (states expanded) by hill-climbing time (states expanded) is in Figures 8 and 9.

On average across all maps, there is a noticeable benefit for having a (compressed) path rather than computing it. This increases with sector size, as the paths between region representatives are longer and more complex. There is a larger benefit in terms of time compared to states expanded as greedy search has a lower overhead than A* for the same

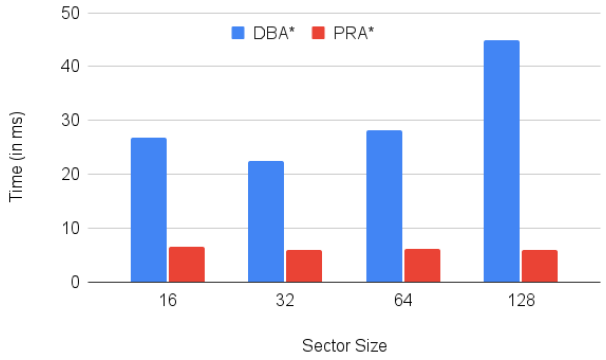

Figure 7: Overall Time to Abstract and Precompute Paths for Dragon Age maps.

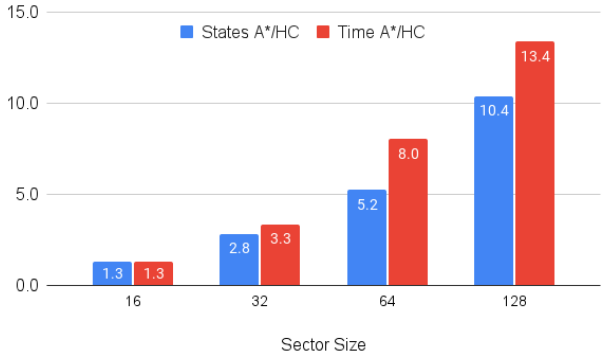

Figure 8: Ratio of A* time (states expanded) divided by hill-climbing time (states expanded) for Baldur's Gate II maps.

number of states expanded.

There is variation in states expanded and time for different maps. For example, on Baldur's Gate maps for sector size 32, the states expanded ratio ranges from 1.9 to 4.8, and the time ratio from 2 to 4.1. There was less variance for Dragon Age maps with the states expanded ratio ranging from 1.5 to 2.1 and the time ratio from 3 to 4. There is slightly more variation between maps as the sector size increases.

Table 1 shows memory used for paths with and without compression between all pairs of neighboring region representatives. Two regions are neighbors if there are two adjacent open cells with one cell in one region and the other cell in the other. For the memory calculations, each state on the path occupies 4 bytes.

For small sector sizes, the majority of the paths between neighboring region representatives require no subgoals and no memory. In comparison, storing the full path occupies 30 to 80 times more space. As an example, for Dragon Age maps sector size 32, of the 1,088 paths between neighbouring region representatives only 22% require subgoals, and the total number of subgoals for all paths is 344. The overall length of all these paths is 28,803 resulting in a factor of 84

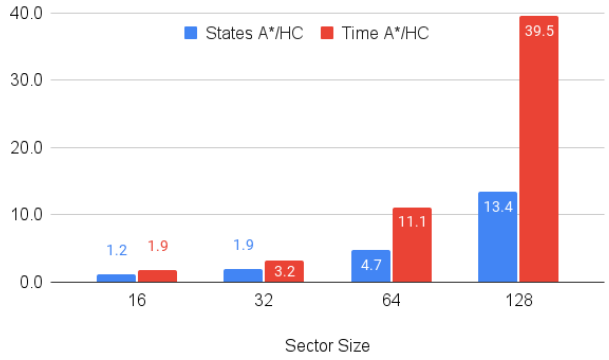

Figure 9: Ratio of A* time (states expanded) divided by hill-climbing time (states expanded) for Dragon Age maps

times savings when compressing the paths.

It is the paths with subgoals that have the most benefit for reuse as those paths will require A* to expand more states. The other paths require no additional space, but may improve time due to using the simpler greedy algorithm. With sector size 32, in the 22% of paths with subgoals, A* expands 2.5 times more states and takes 9 times more time than the greedy algorithm. For the other 78% of the paths, the states expanded is similar but the greedy algorithm is 4 times faster. This insight motivates dynamically balancing between using no precomputed paths as in PRA* and computing all paths as in DBA*

The four algorithm variants were tested on the Dragon Age maps with 10,000 paths between random starts and goals (where the start and goal are not in the same sector) for sector sizes 16, 32, 64, and 128. Figure 10 shows the average time per search. PRA* uses zero memory for paths. PRA* with path caching (PRA*-P) and compressed caching (PRA*-C) have almost the same search performance with the compressed path caching using an order of magnitude less space (Table 1). DBA* has the fastest search performance but may have a slightly higher memory usage than PRA*-C as PRA*-C may not store all possible neighbour paths if they were not used in any search. The suboptimality for a given sector size is within 1% for all algorithms and varies from about 7% for sector size 16 to 31% for sector size 128. Note that no path optimizations were applied to reduce suboptimality. The algorithm variants provide fundamental tradeoffs of memory versus time. All algorithms are faster than A*'s 6 ms average search time.

Changing Maps

Map changes during gameplay that add or remove walls require recomputing abstractions and paths that may be affected. The performance of the algorithm variants was tested on the two sets of maps with multiple sector sizes averaged over five independent trials. Each algorithm precomputed its abstraction and path database. A random open cell is selected to add a wall then that same wall is removed. This forces two abstraction maintenance operations without

Baldur's Gate II Maps							
Sector	**#Paths**	**Length**	**Subgoals**	**%PathsWithSubgoals**	**CompressedSize (KB)**	**FullSize (KB)**	**Ratio**
16	495	6473	78	12%	0.3	25	84
32	164	3874	105	38%	0.4	15	37
64	44	1896	52	63%	0.2	7	36
128	15	1177	48	78%	0.2	5	25
DragonAge Maps							
Sector	**#Paths**	**Length**	**Subgoals**	**%PathsWithSubgoals**	**CompressedSize (KB)**	**FullSize (KB)**	**Ratio**
16	3666	55095	385	8%	1.5	215	143
32	1088	28803	344	22%	1.3	113	84
64	355	16595	306	45%	1.2	65	54
128	121	10719	276	68%	1.1	42	39

Table 1: Path statistics between all neighboring region representative pairs for Dragon Age and Baldur's Gate maps with different sector sizes. Length is the sum of the length of all these paths. Subgoals is the total number of subgoals used to compress all the paths.

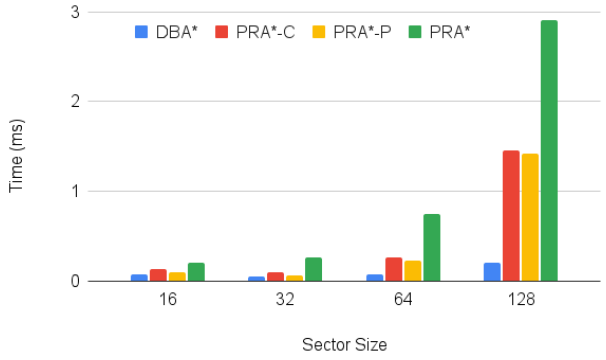

Figure 10: Average search time for random starts and goals on Dragon Age maps for varying sector sizes.

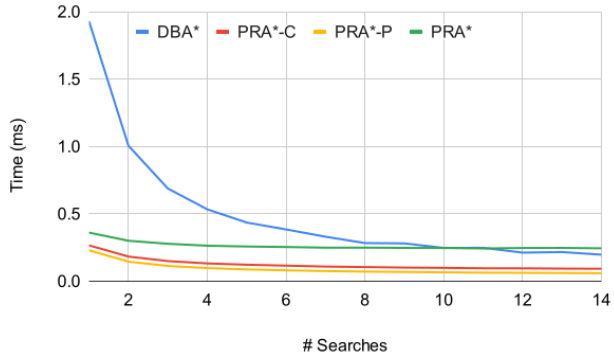

Figure 11: Average time per search when performing two map changes, the associated recomputation of the abstraction and stored paths, and a given number of searches.

changing the map. Then, a varying number of searches are performed on the map with the searches using random start states and having the goal state being the randomly selected cell. This is repeated for 1000 random open cells. The experiment measures how efficiently the algorithms maintain their abstraction and path database. The results for Dragon Age maps for sector size 32 are in Figure 11.

The results demonstrate that all algorithm variants are faster than A* even with frequent map changes. Even with two map changes per search, all algorithms except DBA* complete the maintenance and search in less than 0.5 ms compared to A*'s 6 ms per search. As the number of searches increases between map changes, the algorithms that store paths improve their relative performance. The crossover point is about 11 searches between map changes for DBA* to have faster time than PRA* for sector size 32 and about 15 searches for sector size 64. The two PRA* caching variants have near identical time performance with the compressed path version (PRA*-C) requiring significantly less space as shown in Table 1. PRA*-C represents a good tradeoff between additional memory usage (less than 1 KB) and increased search performance. The suboptimality for all algorithms is similar, and for sector size 32 was approximately 8% suboptimal without any path optimizations.

Discussion

Abstraction and path precomputation algorithms are significantly faster than A* for static and dynamic maps. The algorithm variants have different tradeoffs for memory usage versus time with the algorithms that compute and store more paths (DBA*, PRA* caching) utilizing memory for faster search performance. All algorithms maintain their abstraction and path databases efficiently to outperform A*.

A challenge with investigating performance with map changes is the lack of real-world game data to inform the experiments. It would be valuable to have a standardized data set of map changes from game play. The experiments modified random walls and then performed random searches after the modifications. By adding and removing a wall at a random state which is the goal state of the next search, the experiment guarantees that the map changes cover different areas in the map and that the path would need to be re-computed and not rely on the cache. This accurately re-

flects the time to maintain the abstraction and paths and perform a search. However, real game play may result in clustered map changes and clustered searches where some map changes and searches are far more common than others. In those scenarios, the dynamic path caching of the PRA* variants may be even more valuable as PRA*-C will only cache a path once it is needed for a search. DBA* may compute a path due to a map change that is not used in searches. Precomputing and compressing all paths in the initial map as done by DBA* is useful as this can be done offline and the memory usage is small.

Map modifications beyond single wall additions and removals would not significantly change the results as long as the algorithms are implemented to process all changes in batch rather than individually. Batch modifications would be more efficient than the experiment performed as adding one wall to a sector or multiple walls has no time difference if the abstraction must be recomputed for that sector regardless.

Conclusions and Future Work

This paper investigated the tradeoffs of using abstraction and path computation for video game path finding in 2D grid maps. These techniques significantly speed up search compared to A* with only a small amount of memory usage. The maximum speedup possible was determined by analyzing path finding between region representatives, and varied between 1.2 and 13 times depending on the abstraction sector size. A small number of paths represent the majority of this speedup. For changing maps, optimizations to efficiently maintain the abstraction were used, and the algorithms continue to outperform A* even with frequent map changes. Future work will explore if map change data from different video games affects which algorithm variants have the best overall performance profile. It would also be interesting to compare with other search algorithms such as Jump Point Search (Harabor and Grastien 2014) and investigate abstraction and path precomputation approaches on navigation meshes.

Acknowledgements

The authors acknowledge the funding support of the Natural Sciences and Engineering Research Council of Canada.

References

Bulitko, V.; Björnsson, Y.; and Lawrence, R. 2010. Case-Based Subgoaling in Real-Time Heuristic Search for Video Game Pathfinding. *Journal of Artificial Intelligence Research*, 39: 269–300.

Bulitko, V.; Sturtevant, N.; and Kazakevich, M. 2005. Speeding up learning in real-time search via automatic state abstraction. In *AAAI*, volume 214, 1349–1354.

Bulitko, V.; Sturtevant, N.; Lu, J.; and Yau, T. 2007. Graph abstraction in real-time heuristic search. *Journal of Artificial Intelligence Research*, 30(1): 51–100.

Harabor, D.; and Grastien, A. 2014. Improving jump point search. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 24, 128–135.

Hart, P.; Nilsson, N.; and 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.

Hernández, C.; and Baier, J. A. 2011. Fast Subgoaling for Pathfinding via Real-Time Search. In *Proceedings of the International Conference on Artificial Intelligence Planning Systems (ICAPS)*, 327–330. AAAI.

Lawrence, R.; and Bulitko, V. 2012. Database-driven real-time heuristic search in video-game pathfinding. *IEEE Transactions on Computational Intelligence and AI in Games*, 5(3): 227–241.

Lee, W.; and Lawrence, R. 2013. Fast Grid-Based Path Finding for Video Games. In *Advances in Artificial Intelligence, 26th Canadian Conference on Artificial Intelligence*, volume 7884 of *Lecture Notes in Computer Science*, 100–111. Springer.

Li, X. 2016. *Enhancements to Hierarchical Pathfinding Algorithms*. Master's thesis, University of Denver.

Sturtevant, N. 2007. Memory-Efficient Abstractions for Pathfinding. In *Proceedings of Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 31–36.

Sturtevant, N. R. 2012. Benchmarks for Grid-Based Pathfinding. *IEEE Transactions on Computer Intelligence and AI in Games*, 4(2): 144–148.

Sturtevant, N. R.; Sigurdson, D.; Taylor, B.; and Gibson, T. 2020. Abstraction and refinement in games with dynamic weighted terrain. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 09, 13697–13699.