Partially Precomputed A*

William Hewlett

I. Abstract

A* is a commonly used technique for finding shortest paths for navigation in video games. We propose Partially Precomputed A* (PPA*), which is much faster than A* at run-time, and uses much less memory than completely precalculating all shortest paths with an algorithm such as Floyd-Warshall. At runtime, PPA* is very similar to A*, so it is simple and safe to integrate into existing video game code bases.

II. Introduction

One of the challenges that video game developers face is how to program virtual characters that are capable of navigating to their goals. This problem is typically implemented by searching for a shortest path in a graph that represents the reachable terrain of a virtual world. A* [1], the computationally optimal technique for finding a shortest path, is a frequently used method for path planning in the video game industry because of its simplicity and the size of game graphs. However, A* is becoming unmanageable for the extensive levels and large numbers of characters in today's video games. In this paper we introduce *Partially Precomputed A** (PPA*), a practical method for achieving fast in-game search times in large graphs.

The simplest precomputation strategy is to calculate every possible shortest path offline and store them in a large matrix. This technique is known as *All Pairs Shortest Paths* (APSP). If there are n nodes in the graph, then the matrix is n^2 in size, where entry i, j contains a reference to the next node after i in a shortest path from i to j (Figure 1). The time required to calculate the node to travel to next is $O(1)$, but the space required is $O(n^2)$. PPA* uses much less memory than a full APSP matrix, and is faster than A* at run-time.

PPA* consists of both a precomputation technique and a search technique. In the precomputation phase, an input graph is clustered into a hierarchy of many highly connected small subgraphs, and APSP calculations are made upon each subgraph. To perform searches of the input graph after the precomputation phase, PPA* performs an A* search on a related small generated graph, where a shortest path in the generated graph exactly corresponds to a shortest path in the large graph. Like A*, PPA* always returns the shortest path if one exists. Our experiments reveal that PPA* definitively outperforms A* on a test set of real world graphs, and that PPA* has a more pronounced advantage on larger graphs. Furthermore, it is easy to integrate PPA* into modern video game code bases because the in-game search code is simply performing A* on a precalculated graph.

III. Related Literature

A* [1] is provably the optimal algorithm for finding a single path in an unknown graph, but if multiple searches are performed over the same graph then other algorithms are competitive. Floyd-Warshall [2] [3] and Johnson's Algorithm [4] are

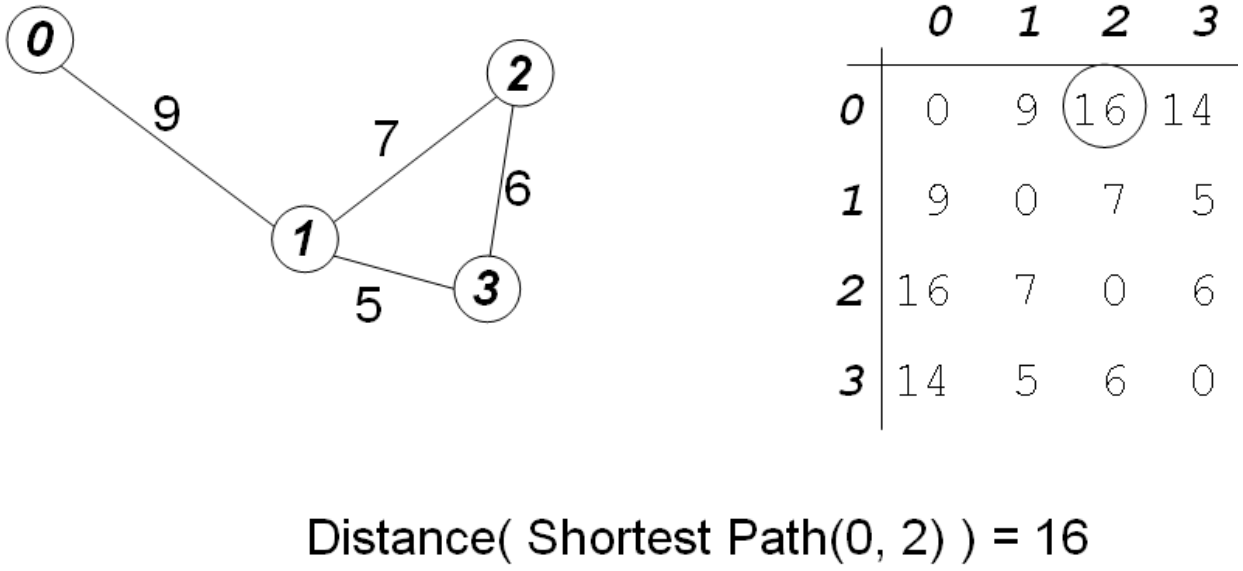


Figure 1. All Pairs Shortest Path (APSP) distance matrix

both techniques that find *every* shortest path in a graph, but are too slow as run-time algorithms for graphs with thousands of nodes. PPA* uses techniques from both A* and Johnson's Algorithm to quickly find single paths using precomputed results.

Several algorithms use a hierarchical approach to perform *approximate* shortest path computation. For example, the HPA* algorithm [5] produces paths that are within 1% of optimal on maps for video games, with searches that are 10 times as fast as A*. Sturtevant [6] implements an alternative approximate search algorithm for the game Dragon Age^{TM} which expands 100 times less nodes than A*, while only using 3% additional memory. Given that game graphs are usually approximations of the space that can be navigated, game developers typically are not concerned with optimality, as long that the quality of the result is close to optimal. Comparisons between PPA* and A* suggest that our work is competitive with these approaches, while always producing the optimal path, and presumably game developers would prefer an optimal path over a nearly optimal path if path generation speeds are similar. It is difficult to do measured comparisons between optimal and near-optimal approaches, since near-optimal approaches can typically trade off optimality for path generation speed. For this reason in this paper we will compare PPA* to techniques that like PPA* always produce the true shortest path.

A number of approaches are orthogonal to PPA* in that they could potentially be applied in combination for additional performance enhancement. In Sturtevant et. al. [7], APSP matrices are partially computed and then used to produce a better heuristic function for A*. In Section V, we find that PPA* searches faster and expands less nodes than this technique while using a comparable amount of memory. Approaches such as that of Bjornsson and Halldorsson [8] and Hierarchical A* [9] also use clustering to create better heuristic functions, but they improve performance over A* by less than a factor of 2. PPA* can use any admissible heuristic, and in our tests we use the straight line distance heuristic function for both A* and PPA*.

Cazenave ([10] and [11]) lists a number of very useful practical optimizations to A* algorithms. Of these, we use lazy cache optimization, preallocation of memory, and maintaining the best path for each visited point in the node in both our PPA* and A* algorithms. Because performing PPA* at runtime involves performing an A* search, and because that A* search comprises the majority of the runtime of PPA*, any technique which improves the speed of A* will improve the speed of PPA* as well.

IDA* [12] trades performance in order to use less memory during a search, so it is unlikely to be useful for games; however, in certain domains, such as grid maps for games, algorithms such as Fringe Search [13] use a combination of A* and IDA*, promising a 10-40% speedup over A*. An open question is whether a technique such as Fringe Search or heuristic methods such as Sturtevant et. al. could be combined with PPA* to achieve further performance gains.

There are some search techniques, such as D* Lite [14], which search at a similar speed to A* but after changes to a graph can replan faster than performing another search. While PPA* offline structures can be dynamically altered if small changes are made to a graph (See Section V), replanning techniques such as D* Lite are faster if a large number of dynamic changes are expected.

In the navigation search domain, there are a number of techniques with very fast search times, but it is difficult to compare them to PPA*. For example, Bast et al. [15] produce shortest paths in an average of 5 milliseconds on a road map graph of the entire US with 24 million nodes. These paths are for the *travel time road map*, where each edge length is the amount of time to traverse that edge at posted speed limits, rather than the *distance graph*, which is the embedded planar graph and is used in our results. The travel time road map is a very different graph topologically than the distance graph, since shortest paths in the travel time road map use freeways almost exclusively, whereas shortest paths on the distance graph are much more varied, often using surface streets. Because of this, techniques that attempt to establish "corridors", certain edges that are always traversed to get to certain regions, are much more effective on travel time graphs. Navigation search techniques are concerned about how long it will take to drive to a location, but in video games travel times are mostly uniform over distance and the distance graph is more relevant. It is unclear exactly how much faster it would be in general to search travel time graphs, but for one subsection of the search in [15] (finding the total "distance" of the shortest path), searching the distance graph is 8 times slower than searching the travel time graph.

The Bast et al. technique[16] uses Dijkstra's single-source shortest path algorithm[17] to determine a *highway hierarchy*, a set of edges that are commonly on shortest paths between nodes far away from each other. Like all precomputed hierarchical approaches, it relies on performing searches on successively larger and larger regions of the graph, but unlike PPA* which uses A* to search a simple generated graph, it relies on a bidirectional Dijkstra's search with lookups into tables which contain the distances of the highway edges. As a speed optimization, Bast et al. uses an APSP matrix for lookups of its highest graph, while PPA* uses APSP matrices at every level of the hierarchy. (Sanders and Schultes[18] is a good introduction to this line of algorithms, based on approaches from the graph theory community.)

Our approach is most similar to the HEPV algorithm [19] which also comes from the Navigation Search literature. Like PPA*, HEPV uses clustering and APSP matrices to produce a precomputed data structure, but the run-time search is different. Instead of searching the reduced graph using A*, HEPV searches it by beginning with the start and goal nodes and recursively expanding nodes connected to them. This effectively expands every node in the entire reduced graph, and in experiments it performs much worse than A* on the original graph.

The problem with high speed search on large graphs appears in many different research communities. This paper bridges the gap between the navigation search and video game search communities by providing a competitive precomputed search algorithm that is easy to integrate into existing game code because it reuses existing components.

Figure 2. Map of Venice

IV. Our Approach

Conceptually, there are three different steps in PPA*.

1) Partition a graph into subgraphs
2) Calculate APSP matrices for the subgraphs
3) Perform searches

In practice, the first two steps are performed offline and repeated at different levels of the search data structure hierarchy. For ease of explanation, we will first describe the full approach with a single hierarchical layer, and then explain how the clustering and finally the search is expanded to multiple layers.

A. Single Layer PPA*

To describe single layer PPA* we will consider an example based on a 1913 map of Venice[20]. We would like to perform fast searches over this map, for example from the triangle (Ex Campo Di Marte) to the star (Giardini Pubblici) in Figure 2. First, each island in the city is marked with a node and each bridge corresponds to an edge as in Figure 3. Next, the nodes are clustered as in Figure 4. The nodes in a cluster and the edges between these nodes form a *subgraph*. The goal of the clustering is to minimize the number of nodes that have edges that cross clusters. These nodes that have edges into different clusters are called *border nodes* and the edges that cross into other clusters are *border edges*. These special nodes and edges can be seen in Figure 5. As a precomputation step, after clustering we calculate the APSP matrices for each *subgraph*, so that at runtime we can lookup the shortest distance from any node in the subgraph to any other node in the *subgraph*.

In addition to the lower level *subgraphs*, there is a higher level *parent subgraph* that consists of all of the *border nodes* of each of its *child subgraphs*. Edges in this higher level subgraph consist of the *border edges* as well as *virtual edges*. *Virtual edges* are edges that we add to the higher level subgraph which represent connectivity in the lower subgraphs. Each *border node* in a lower level subgraph has a *virtual edge* to each other border node in its subgraph which represents the shortest path between the two nodes as seen in Figure 6. This *virtual edge* has a length of the precomputed shortest path. The entire highest level subgraph can be seen in Figure 7. We will also precompute the APSP matrix for this highest level subgraph.

Figure 3. Graph of Venice

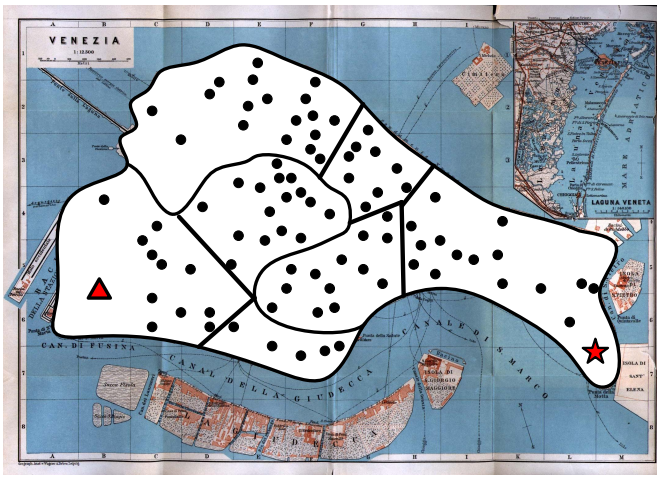


Figure 4. Nodes Clustered

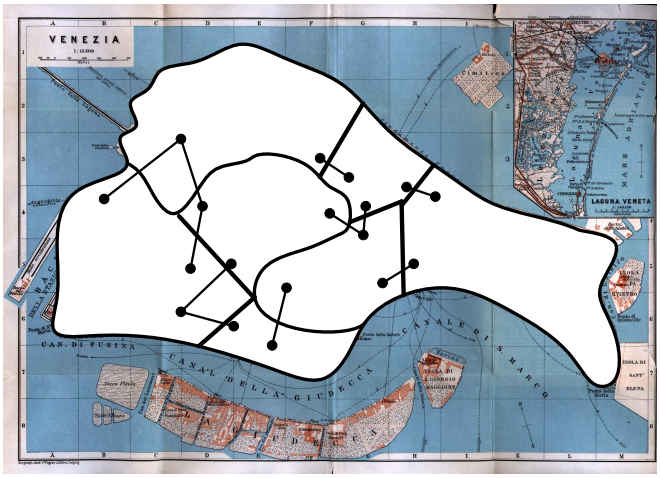


Figure 5. Border Nodes and Border Edges

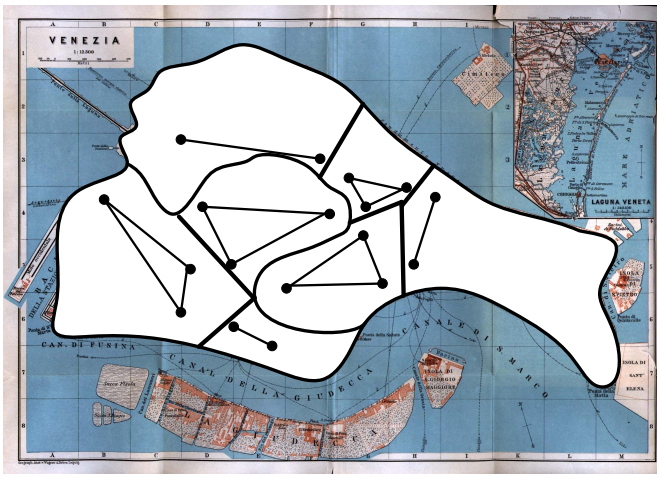


Figure 6. Virtual Edges

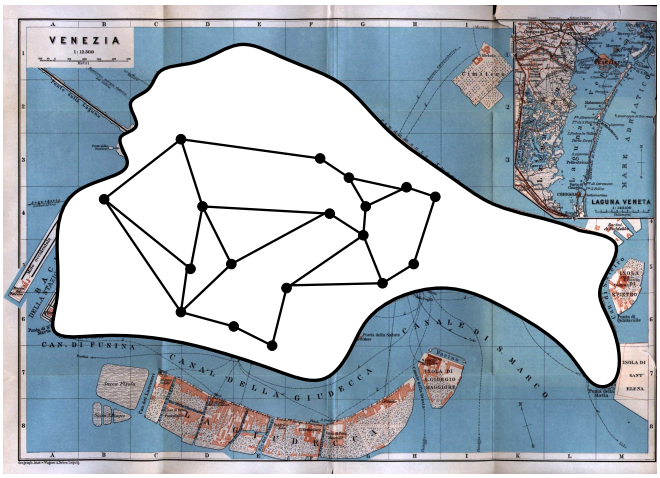


Figure 7. Parent Subgraph

At runtime, we would like to perform a search on the graph of Venice. First we construct virtual edges between the start node and the border nodes of the starting node's subgraph. The lengths of those edges will be the distances recorded in the APSP matrix associated with the starting node's subgraph. We can construct similar virtual edges in the goal node's subgraph as seen in Figure 8.

Finally, we can construct virtual edges between each of the border nodes of the starting node's subgraph and each of the border nodes of the ending node's subgraph. The lengths of these virtual edges can be found in APSP matrix of the higher level subgraph. Putting these virtual edges together with the virtual edges of the start and end subgraphs for this search we get the final search graph, as seen in Figure 9. Instead of performing an A* search of the original graph with almost 100 nodes, we perform a search on this constructed graph with only 7 nodes.

To search a Single Layer PPA* data structure, PPA* search behaves much like A* search except that it searches over a different, smaller graph. The nodes of this smaller graph are a subset of the nodes of the original graph. Like A*, PPA* expands a start node, generates neighbors, and expands nodes until it expands the goal node. Each node expanded by PPA* is a node that will be expanded by A* and the nodes that are expanded by PPA* are expanded in the same order as the nodes

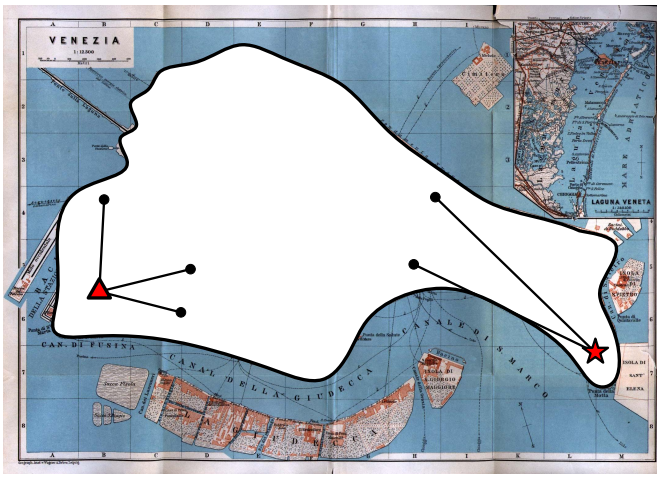


Figure 8. Start and Goal Subgraphs: Virtual Edges

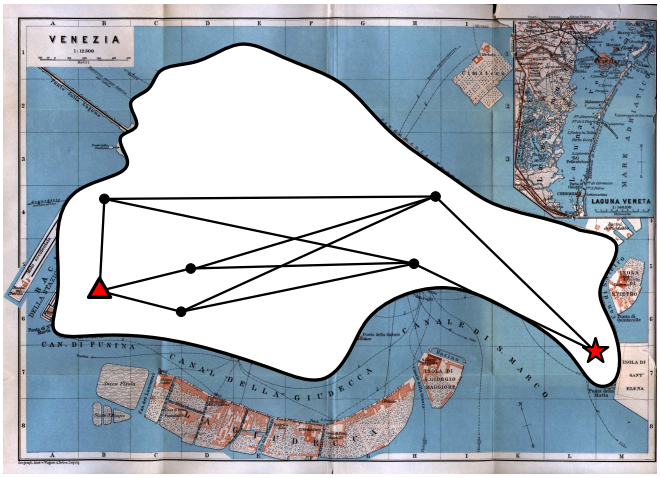


Figure 9. Final Searchable Graph

are expanded by A*. Furthermore, PPA* uses the same $h()$ and $g()$ functions that A* uses, and when it evaluates a node, it obtains the same outputs from the $g()$ and $h()$ functions for that node that A* does. The paths returned by A* and PPA* are exactly the same. To verify the correctness of our implementation, after each search in a testing run we compare the A* calculated path against the PPA* calculated path.

B. Single Layer PPA: Clustering and Analysis*

In the simple single layer case, first a clustering algorithm [21] is used to cluster the original graph into a series of disjoint subgraphs. After clustering, we identify *border nodes*, which are nodes in a subgraph with edges to nodes in other subgraphs. Given a set of subgraphs, the ideal clustering of the original graph will minimize the number of border nodes while allocating similar numbers of nodes to each subgraph. In the planar Venice example (Figure 4), our clustering is simple, but in general a clustering technique for PPA* should not use the physical location of the nodes. Instead, minimizing the number of border nodes is the only important clustering criterion.

Two nodes which are far apart may be in the same cluster, and clusters can overlap in “physical space”, as long as each node is contained by a single cluster and the number of border nodes per cluster are minimized. The PPA* algorithm works on the

graphs that A* works on; it requires a graph with no negative cycles and an admissible heuristic. In many other precomputed techniques (especially non-optimal techniques such as HPA*[9] and Sturtevant[6]), a single node represents a region of space and an entire child subgraph. In the PPA* data structure, a parent subgraph consists of all border nodes of each child subgraph, which is normally several nodes contributed per child subgraph.

Assume that the original graph is clustered into f subgraphs each with k nodes. For each subgraph, an APSP matrix is generated. The precalculated matrices are each of size k^2, so the asymptotic memory of storing these matrices is $O(fk^2)$. We would like to estimate the algorithmic cost of memory in terms of n rather than k and f so that PPA* can be compared to other techniques. First we will estimate the memory costs of Single-Layer PPA*, in Section IV-C we will extend these results to Multi-Layer PPA*.

Suppose there are c border nodes in each subgraph. We will form a new subgraph from these nodes, called the *parent subgraph*. Each of the disjoint subgraphs has the parent graph as a parent, but the parent graph *only* contains information about the border nodes of its child subgraphs (as opposed to all nodes in the original graph). The total number of nodes in the parent graph will be fc. We will construct an APSP matrix for the parent graph as well. So the total space for all of the APSP matrices of this graph will be $O(fk^2 + (fc)^2)$. As f and k approach the square root of the total number of nodes, $\sqrt{n}$ (since $fk = n$), the space cost approaches $O(n\sqrt{n} + nc)$, or $O\left(n\sqrt{n}\right)$ if c is small. This Single-Layer PPA* memory cost is too expensive, so we will reduce it by using multiple levels of hierarchy (see Section IV-C). All results (see Section V) use Multi-Layer PPA*.

C. Multi-Layer PPA*: Clustering and Precalculation

The offline precalculation step of PPA* consists of two steps, partitioning the original graph into a hierarchy of subgraphs and then calculating the APSP matrix for each subgraph. To cluster a graph with a levels of hierarchy, we first cluster the nodes of the original graph to produce several distinct child subgraphs and then cluster each of the child subgraphs recursively. This process is similar to that of a quad tree decomposition. Eventually we reach the lowest level of subgraphs, and the set of the lowest level of subgraphs will contain every node of the original graph, with each node belonging to exactly one lowest level subgraph. For the sake of clarity we will call these lowest level subgraphs *level-0 subgraphs*. Like the parent graph in the single layer case, a parent subgraph of a set of level-0 subgraphs will contain only the *border nodes* of those subgraphs (recall that border nodes are those that have edges to other nodes outside the subgraph). These level-1 parent subgraphs are disjoint from each other but not every node in the original graph is contained by them, since many nodes aren't border nodes. We repeat the process of associating border nodes of level-1 subgraphs with their level-2 subgraph parents, and so forth to the top graph, of level-$(a-1)$. This top graph will only contain nodes which have edges that cross our first clustering of the graph. While the top graph only contains a fraction of the total nodes of the original graph, for any node we can get its level-0 subgraph, find the parent of that subgraph, the parent of that subgraph and so forth up until the highest level subgraph. In effect, each node has a list of "ancestor" subgraphs, but only the lowest, level-0 subgraph in this list is guaranteed to contain that node.

There exist a plethora of graph clustering algorithms. For this work we chose the Metis clustering algorithm [21]. The Metis

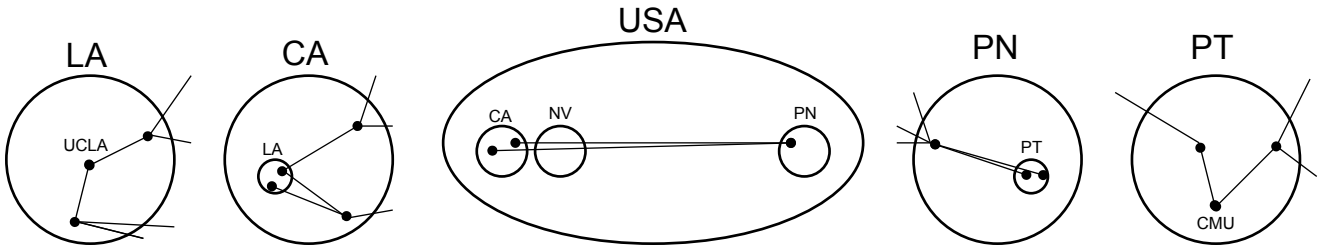
algorithm clusters graphs quickly (an order of magnitude faster than other algorithms we tested) and generally produces very good clusters. The Metis algorithm attempts to divide a graph into f subgraphs where each subgraph has approximately the same number of nodes and where the number of edges that has to be cut to separate the subgraphs (in effect, the border edges) is minimized. Our metric is different; we want to find evenly sized subgraphs with as few *border nodes* as possible; however, in practice the Metis clustering does a fair job of meeting our metric.

Finally, after all subgraphs are determined, APSP matrices are computed intra-subgraph. Lower level subgraph shortest paths are calculated first, so that virtual links can be constructed in higher subgraphs to properly calculate the shortest path distances in those subgraphs. Since graphs in gaming are relatively sparse, we use Johnson's algorithm [4] to calculate the APSP matrices. Johnson's algorithm has a running time of $O\left(V^2 \log V + VE\right)$, which compares favorably to the $O\left(V^3\right)$ running time of Floyd–Warshall [2][3] for graphs where E is closer to V than V^2.

We can model the algorithmic memory usage of PPA* by assuming as before that the graph is broken up into subgraphs each with k nodes. We will assume that we have a perfect hierarchical graph, where the number of border nodes c per subgraph and the number of nodes k in a subgraph is constant at each of the a levels of the hierarchy. To calculate the asymptotic bound we multiply the total number of subgraphs by the memory of each subgraph, which is proportional to k^2. There is one highest level subgraph, and it has k nodes in it, like all subgraphs. The number of subgraphs in the next level down of the hierarchy is k/c, since each lower level subgraph has c nodes that they contribute to the highest level subgraph. Similarly, the level below that will have k/c graphs for every graph in the second highest level, or $(k/c)^2$ total subgraphs. This chain can be extended down until the lowest level, which will have $(k/c)^a$ subgraphs. We can then compute the total number of subgraphs as sum of all of these quantities multiplied by the size of each or $O(k^2((k/c)(a+1)-1)/(k/c-1))$. The total number of nodes in the graph is n, which is also equal to the number of subgraphs at the lowest level of the hierarchical tree, multiplied by k, since every node appears exactly once in the lowest level. Since there are $(k/c)^a$ lowest level subgraphs, $n = k(k/c)^a$. Solving for a we obtain $a = \log(N/k)/\log(k/c)$. We can then plugdouble this back into the original memory equation to solve for the total amount of memory used by this ideal hierarchical approach. Fortunately, it simplifies neatly to $(k(n-c)/(k-c)$. If k/c, or the ratio of border nodes to nodes in a subgraph is constant (which is common in our experiments), we can simplify this to $O(n)$ memory for PPA*.

It is simple to make small dynamic changes to the overall graph at runtime. This is important for video games that might have dynamic events such as a fallen bridge or a traffic jam, which might alter the in-game graph. There are four types of possible dynamic graph changes, node insertion, node deletion, edge insertion, and edge deletion. While each type of change uses a similar amount of computation, our experiments focus on edge deletion because it involves the smallest changes to the data structure of the graph. To perform an edge deletion, first the edge is removed, then the subgraph which contains the edge is recalculated. Each parent subgraph above this subgraph is also recomputed, so deleting a high level edge is faster than deleting a low level edge. For example, if a map has a number of islands connected by bridges, removing a bridge will be faster than removing a street on an island, since when a bridge is deleted only the "bridge-level" subgraph has to be recomputed, whereas when a street is removed both subgraph which contains the island as well as the bridge subgraph have to be recomputed, since the "bridge level" subgraph contains virtual links over the island between bridges which might use the removed street.

Figure 10. Example of PPA* Search



To improve update times, at each level we check the distances between each border node of a subgraph before and after a dynamic change. If the distances between border nodes do not change, we cease recursing and avoid recalculating higher level graphs. While recompute times of about 0.20 seconds (see Section V) are reasonable for some applications, if there are large scale dynamic changes then it is likely that a complete re-clustering will be necessary to preserve fast search speeds.

The precalculation phase is readily amenable to parallization. Precalculation consists of two stages, clustering and APSP calculations. Each clustering operation is independent of every other clustering operation, although top level clustering operations must be performed before lower level clustering operations. The calculation of APSP matrices is embarrassingly parallel; that is, each matrix can be calculated completely independently, although lower level APSP matrices must be calculated before their parents. All of our tests were done with a single processor, but it should be noted that with k processors we expect a precalculation speedup of nearly k.

D. Multi-Layer PPA: Search*

Performing searches on hierarchical graphs with more than one level is similar to the single layer case (Figure 10). An important difference from the single layer case is that subgraphs will have higher level parent super-subgraphs that contain them. Each node has a subgraph that immediately contains it, which in turn is contained by a higher level subgraph, and there is a such a parent for each level of the hierarchy. So if there are four levels of hierarchy then each node has four subgraphs which are associated with it. Higher level subgraphs only contain APSP distance information for border nodes of their immediate child subgraphs, not every node contained by those subgraphs. Given a start node in a lowest level subgraph, PPA* generates the border nodes of that subgraph. The distance to each of the border nodes is the precalculated distance to them in the APSP matrix for the subgraph. To expand one of these border nodes, consider the parent graph containing it. This parent graph will have its own border nodes, which are nodes that connect to border nodes in other parent graphs. The distance to each parent border node is available in the parent graph APSP matrix. This process generates and expands nodes toward the uppermost parent graph. The search also expands nodes downwards towards the goal node. If a node is being expanded and one of its subgraphs is in the set of subgraphs associated with the goal node, PPA* generates border nodes for the child subgraph associated with the goal which is below the common subgraph. Finally, if a node is in the same immediate subgraph as a goal node, PPA* can generate the goal node. (See Pseudo-code 1)

For example, consider a street search from the UCLA campus in Los Angeles, California to the Carnegie Mellon campus in Pittsburg, Pennsylvania (See Figure 10). Suppose that this is a three level hierarchy where each city has its own subgraph,

```
for i=0 to levelsOfHierarchy do
    # Subgraph(n,i) returns ith ancestor subgraph of n   currentSubgraph = Subgraph(currentNode, i)
    startSubgraph = Subgraph(startNode, i)
    goalSubgraph = Subgraph(goalNode, i)
    if (currentSubgraph == startSubgraph) then
        foreach borderNode in currentSubgraph do
            Add borderNode to Open List
        end
    end
    if (currentSubgraph == goalSubgraph) then
        if (i == 0) then
            Add goalNode to Open List
        end
        else
            ls = Subgraph(goalNode, i-1)
            foreach borderNode in ls do
                Add borderNode to Open List
            end
        end
    end
end
```

each state has its own subgraph, and the highest subgraph is the entire United States. The search will first expand the start node, which will generate the border nodes for Los Angeles. These border nodes for Los Angeles are the nodes which have connections outside of Los Angeles. When one of these Los Angeles border nodes is expanded, it will generate the border nodes for California, since California is higher level subgraph for the Los Angeles subgraph. Border nodes of the California subgraph are contained in the United States subgraph, but in this example the United States doesn't have border nodes, since it is the highest subgraph. Since the United States subgraph is an eventual parent subgraph of the Carnegie Mellon goal node, we expand downwards into the child subgraph of the United States containing Carnegie Mellon, which is the Pennsylvania subgraph. When a California border node is expanded, each border node of Pennsylvania is generated. Note that border nodes from Nevada will not be expanded, even though the eventual path might travel from California through Nevada, since Nevada is not an ancestor subgraph of Carnegie Mellon. Since Pennsylvania is an ancestor of the immediate subgraph containing Carnegie Mellon, when expand the border nodes of Pennsylvania we generate the border nodes of the Pittsburg subgraph. Finally, when one of the Pittsburg border nodes is expanded we generate the Carnegie Mellon goal node, and when we expand the Carnegie Mellon goal node we are finished. The path is our UCLA start node, a Los Angeles border node, a California border node, a Pennsylvania border node, a Pittsburg border node, and finally our Carnegie Mellon goal node. The actual path is constructed using the information stored in the APSP matrices.

In order to construct a complete path from a PPA* search, it is necessary to find the nodes that connect each PPA* node together. This path is always contained in a set of existing APSP graphs. There are two possible methods of recomputing this path: store paths in APSP matrices or perform a secondary search. In our experiments we always performed these secondary searches to construct full paths, and our APSP matrices only contain distance information. To perform a secondary search for an APSP graph, one can run A* on subgraph, but one must use the APSP distance values as the heuristic function, since the APSP values are the exact distance to the goal. Because this is an exact heuristic function, the A* algorithm will only expand nodes along the direct path to the goal. Some edges in a subgraph are "virtual" in that they do not exist in that subgraph but are contained in a lower level subgraph and APSP matrix. We recursively perform A* on these links in the lower level

subgraphs.

In this paper we use a simplified version of A* called ResolvePath for this step. (See Pseudo-code 2) Since the heuristic is perfect there is no need to maintain an open list or "expand" nodes, instead for each node we look through each of its neighbors and the correct "next" node is immediately calculated.

```
Procedure ResolvePath(c, n, s, p)
Input: c = The current node
Input: n = The next node
Input: s = The subgraph containing c and n
Input: p = The path between c and n
Output: p = The path between c and n
while c ≠ n do
    edge.cost = edge.length + distance(edge.to, n)  bestEdge = arg min_edge edge.cost; edge ∈ c
    if !bestEdge.virtual then
        Append(p, bestEdge.to)
    end
    else
        ls = LowerSubgraph(s, n)
        # ls is child of s, contains node n ResolvePath(n, bestEdge.to, ls, p)
    end
    c = bestEdge.to
end
```

In our example above (See Figure 10), we had a California border node and a subsequent Pennsylvania border node which were produced by our search. The actual best path might travel from California through the neighboring state Nevada on the way to Pennsylvania. In this case the optimal path in the United States subgraph would travel first to a Nevada border node on the California side of Nevada to a Nevada border node on the other side of Nevada. The edge across Nevada between these border nodes would be virtual, since information about the path would be stored in the Nevada subgraph rather than the United States subgraph. We would then run our ResolvePath function between these nodes in the Nevada subgraph, which might recurse into a Nevada city subgraph. Each node in the overall path is visited only once and in practice ResolvePath consumes less than 20% of the total search time. All reported results include this path resolution step in the time required to generate a path.

There are a few edge cases which are unintuitive. For example, suppose the start node and the goal node are in the same graph (Figure 11). It might appear that PPA* could just use the precomputed path between them. Unfortunately, while PPA* can generate the goal node in this case, it might be that the shortest path between the start and goal nodes travels through border nodes and in different subgraphs. The APSP matrix only stores a shortest path contained in the subgraph and does not consider paths that travel outside the subgraph. In this example, the goal node and border node A are generated from the start node. Assuming an admissible heuristic function, PPA* will next expand border node A. Node A is contained by an ancestor graph of the goal graph, so PPA* looks for members of this super graph which are also members of the lower goal subgraph. The lower subgraph from the super graph on the ancestor list is the goal subgraph, so PPA* generates B, a border node of the goal subgraph. Since $g() + h()$ for node B is less than $g() + h()$ for the goal node through the start node, PPA* expands node B. Finally, PPA* generates the goal node a second time, and once it expands the goal node it will have a shortest path from the start to the goal. To produce the actual set of nodes in the final shortest path, PPA* consults the APSP matrix for each subgraph it traverses. In this example, when resolving the edge from A to B, the shortest path is in the super graph:

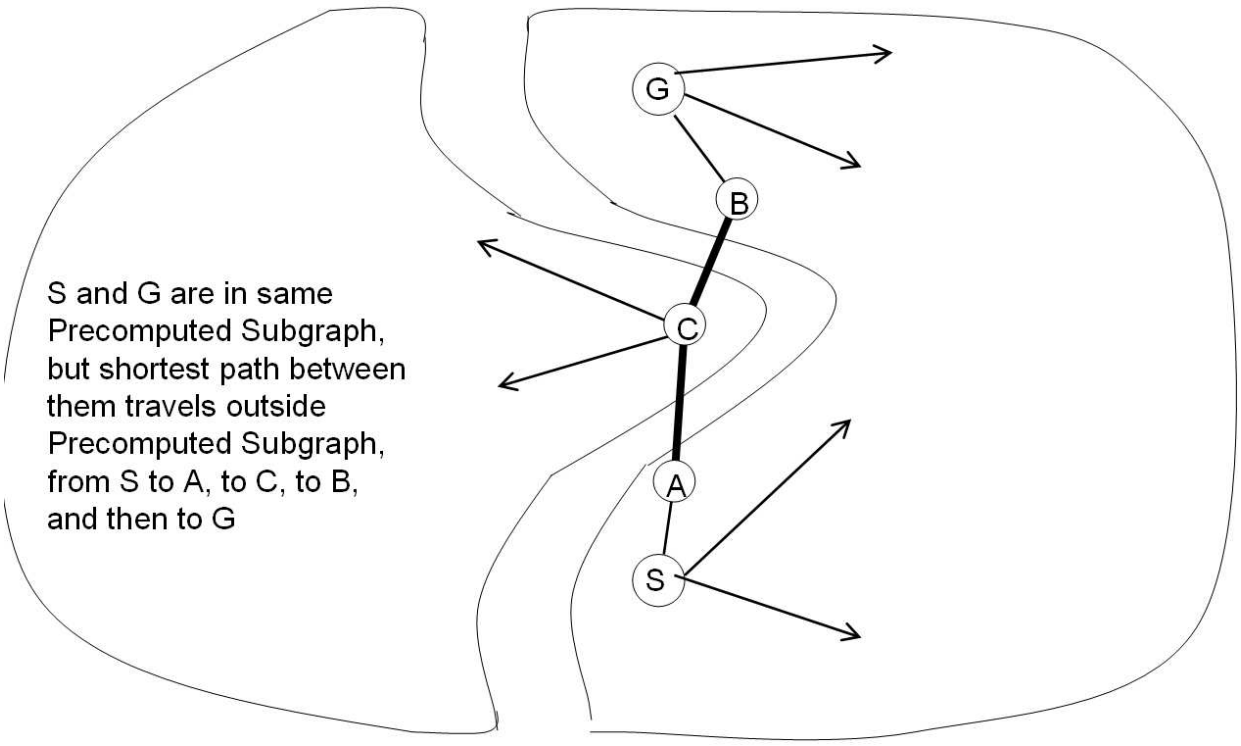


Figure 11. Difficult edge case

$A \Rightarrow C \Rightarrow B$. The final shortest path is $start \Rightarrow A \Rightarrow C \Rightarrow B \Rightarrow goal$. Unlike approximate shortest path algorithms, PPA* always returns the shortest path.

The PPA* search of a graph is an A* search on every border node in the ancestor subgraph lists for the start subgraph and the goal subgraph. The order of expansions is based on the $g()$ and $h()$ functions, so PPA* might expand one node downwards and later expand another node upwards, just as A* will sometimes expand nodes that are two hops away from the start node before expanding all of the nodes that are one hop away. Unfortunately, as the number of border nodes per subgraph increases, the performance of PPA* decreases. If there are c border nodes per subgraph and a levels of hierarchy the number of nodes that PPA* needs to expand in the worse case is c^{2a}. In effect, A* is being performed on a graph with a depth of $2a$ and a branching factor of c. In Section V, we will show experimental results which indicate that PPA* outperforms A* on real world graphs by a wide margin.

One concern that software engineers may have when confronted with a complex algorithm like PPA* is that it is too difficult or dangerous to integrate into an existing codebase. While the offline precomputation step is intensive, the in-game search algorithm can easily be integrated into an existing A* algorithm. The heuristic function and the overall logic of the A* algorithm are exactly the same, but the Add Neighbors to the Open List function is different. In A*, the edge length is the distance of an outgoing edge, but in PPA* the edge length is calculated by consulting the stored APSP matrix. Similarly, the successors to a node in A* are merely the neighbors of a node, but in PPA* they are the border nodes of the appropriate subgraph. To reduce complexity (at the cost of memory), the successor nodes as well as distances can be stored in APSP matrices, otherwise an algorithm such as ResolvePath should be used.

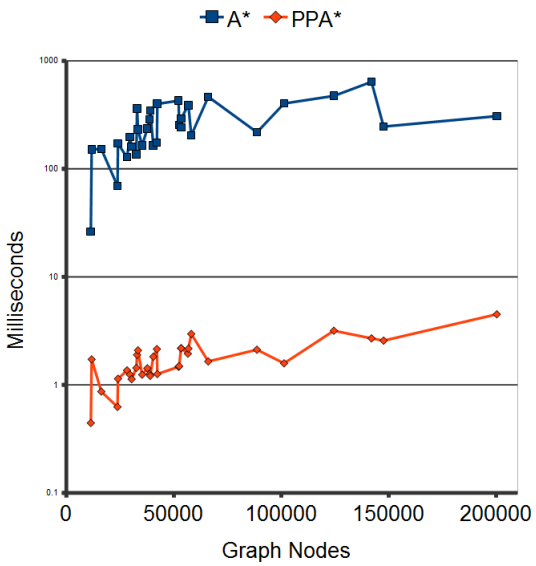


Figure 12. Average search times (logarithmic scale)

V. Results

Our results consist of two sets of experiments. For one set of experiments, we used road maps of California from the US Census Bureau. These maps vary in size from 11383 to 200288 nodes, and our experiments on them show the performance of PPA* on a range of different map sizes with relatively irregular node placement. The second set of experiments are performed on $512x512$ "room" maps of around 200000 nodes that are used by Sturtevant et. al. [7] and are publicly available. This second set of experiments clearly demonstrate the value of PPA* in comparison to a heuristic based search approach.

Each search consisted of picking two vertices at random and then finding the shortest path between them. The same set of vertices were used for both PPA* and A*. For each test, we performed 1000 such searches and averaged the results. The searches were evaluated on a 2.6 GHz Core i7 using a single thread. The average search time for PPA* was significantly less than for A* (Figure 12). Like all of the graphs in the results section, this graph uses a logarithmic scale to allow for comparison. It should be noted that the time listed in this graph is the total time to produce the entire optimal path for each method.

A method for comparing search algorithms that is independent of the hardware is to consider the number of nodes that the search expands or generates. The best way of explaining the difference between expanded nodes and generated nodes is to use A*, since PPA* uses A* to search its smaller graph. In A*, the start node is first placed on the open list, which is a priority list of nodes that need to be expanded. The top node of the open list is then expanded, and its neighbors that haven't been expanded yet are added to the open list. The search ends when the goal node is expanded. The generated nodes are nodes that are added to the open list, while the expanded nodes (a much smaller subset of the generated nodes) are only the nodes that get expanded. Figure 13 shows the large difference in the number of expanded nodes between A* and PPA*. This discrepancy can be explained by the fact that a path in the PPA* graphs is much shorter than a path in the A* graph.

The number of nodes generated by both A* and PPA* is closer (Figure 14), but PPA* still outperforms A* by an order of magnitude. The reason that the generated nodes are closer is that the PPA* graph is highly connected, so each time a node is

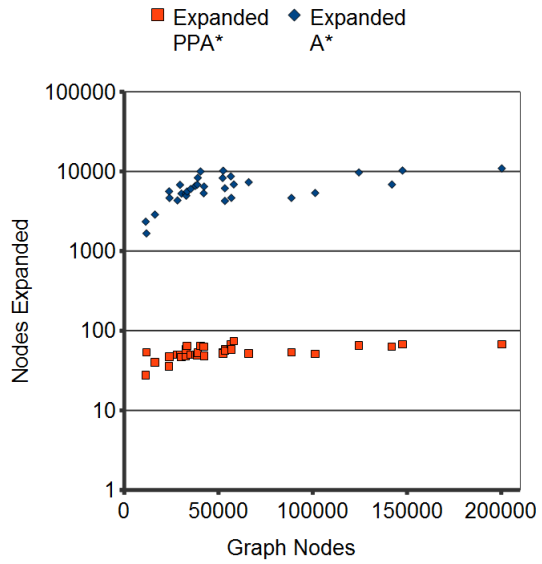


Figure 13. Expanded nodes per search (logarithmic scale)

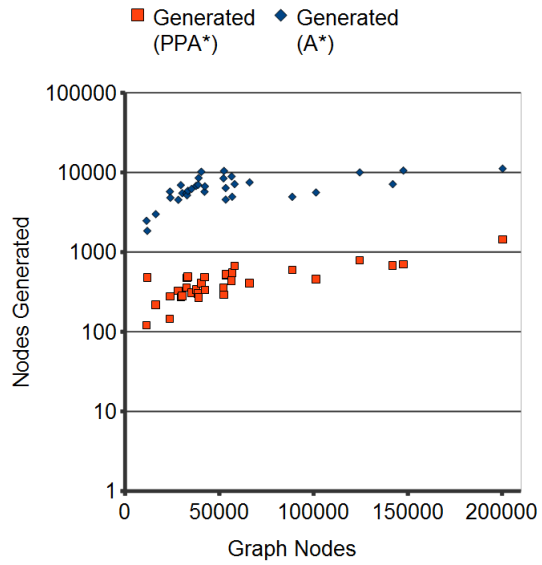


Figure 14. Generated nodes per search (logarithmic scale)

expanded, a large number of nodes are generated. In effect, the branching factor of A* is smaller than PPA*, but the number of hops required to from the start to the goal node is so much lower in PPA* that the number of total nodes generated (and the search time) is much lower in PPA*. Additionally, most of the work done in each search is per expanded node rather than per generated node, and most search algorithms are compared by the number of expanded nodes rather than the number of generated nodes.

The second set of experiments also consisted of 291 random searches performed on a $512x512$ "room" map, the same map which was used in Sturtevant et. al. [7] These searches all had a solution length between 256 and 512 nodes. With an optimized distance heuristic with advanced placement (the most effective search in Sturtevant et. al.), an average of 3479 nodes were expanded and each search took an average of 0.054 seconds. On the same map, PPA* expanded an average of 53.25 nodes

Table I
COMPARISON VS. STURTEVANT ET. AL. ON 200000 NODE ROOM GRAPH

	Average Search Time (milliseconds)	Average Nodes Expanded Per Search	Precomputed Memory	% of APSP memory	Precomputation Time (seconds)
Sturtevant 2009	54	3479	N/A	~0.1%	N/A
PPA*	1.4	53	176 MB	0.1%	213

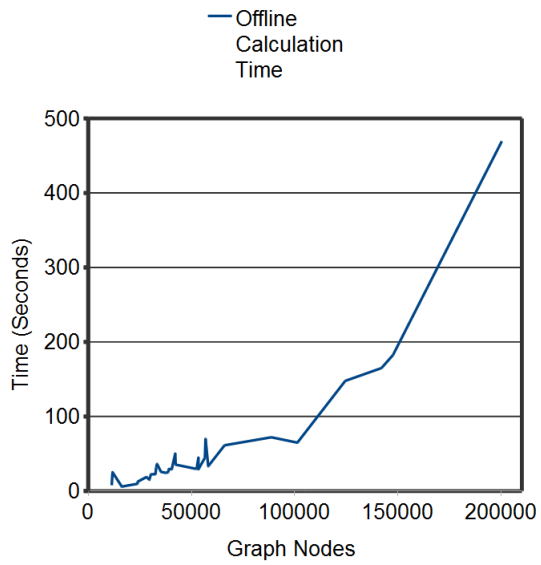


Figure 15. Precalculation Times

and each search took an average of 0.0014 seconds, significantly better in both nodes expanded and execution time (See Table I). It should be noted that the number of nodes expanded by PPA* is usually smaller than the total path length of a search in the original graph because PPA* is effectively searching a much smaller graph. Additionally, in these experiments PPA* used a straight line heuristic, which while correct is less effective than the octile heuristic used by Sturtevant et. al. for this problem.

Precomputation consisted of two stages, clustering and computing APSP matrices. It took 469 seconds to precalculate our largest roadmap graph, which has 200284 nodes (Figure 15). Computing APSP matrices dominated the time spent on precomputation. Because clustering was such a small percentage of overall precomputation time, a more specialized clustering technique might be more effective for PPA* both in precomputation time and for run-time performance. Graphs of more than 50000 nodes had 5 levels of hierarchy with 4 subgraphs per parent subgraph for a total of 1365 subgraphs, of which 1024 were level-0 subgraphs.

PPA* has substantial storage benefits. An alternative to PPA* would be to calculate the APSP matrix for the entire graph during precomputation time. A search at run time of this matrix would be very fast, but the matrix would be unacceptably large. Assuming that each entry used four bytes, the fully precomputed matrix for 205373 nodes would take 169GB of space, which is too large for most practical applications. The same $512x512$ "room" map with 205373 nodes and four levels of hierarchy requires 176MB of space for PPA*. Figure 16 shows the dramatic memory savings of PPA* compared to full APSP matrices. PPA* has a competitive memory footprint with Sturtevant et. al. [7], who reported that their technique used $1/1000$ of the total memory needed for the full APSP matrices (See Table I).

It is relatively inexpensive to recalculate the APSP matrices after small dynamic changes. Using the 200000 node "room" graph of Sturtevant et. al., we built our normal precomputed graph which took 4.99 seconds to cluster and 207.87 seconds to build all 1365 APSP matrices, for a total precomputation time of 212.86 seconds. Then we deleted 100 edges at random, rebuilding the APSP matrices after each edge deletion. It took an average of 0.197 seconds to perform all required APSP matrix rebuildings after each edge was deleted.

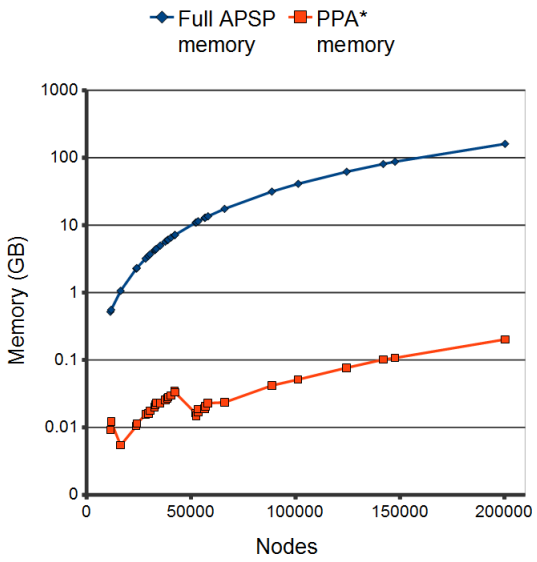


Figure 16. Precomputed memory costs (logarithmic scale)

VI. Conclusion And Future Work

Many video game developers use A* or completely calculated shortest paths to navigate agents in virtual worlds. As the size and number of agents in these worlds increase, these algorithms will be too expensive for performance and memory. We introduced the Partially Precomputed A* (PPA*) algorithm. PPA* makes large performance and memory improvements on these existing algorithms and is simple and safe to integrate into existing game engines.

We would like to make further improvements in our search algorithm. One promising avenue is to use bidirectional search, which begins at both the start and goal node instead of searching from the start node to the goal. Since the middle of the PPA* graph is much more dense than the ends, this technique promises dramatic speed gains.

In the longer term, we would like to extend PPA* to situations where APSP matrices are not precomputed, but are gradually filled in as more searches are completed. This is similar to Lifelong Planning A* [22], which performs successive searches of a graph with different start nodes and a fixed goal node, except we would allow arbitrary start and goal nodes. The intuition is that certain PPA* subgraphs are likely to be traversed by a number of searches, and shortest paths to and from the border nodes of these subgraphs are gradually developed.

All of the experiments done with PPA* use a straight-line heuristic, which is correct but can be improved. Most other techniques for improving A* focus on improved heuristics. Since PPA* is heuristic-independent and accepts any admissible

heuristic, it follows that these other techniques could be used to speed up PPA* just as they speed up A*. It is unclear at this time which heuristics are most effective on the condensed graph that PPA* uses; this is an active area of our research.

We are interested in finding applications where many A* searches need to be done quickly, without a separate precomputation time, which might be good candidates for PPA*. For settings where precomputation is not an issue but run time performance is, we feel precomputed searches like PPA* are clearly useful.

ACKNOWLEDGMENTS

I would like to thank Professor Demetri Terzopoulos, Professor Richard Korf, Professor Adam Meyerson, Gabriele Nataneli, Shawn Singh, and all of the anonymous reviewers. I would also like to thank Intel Corp., Microsoft Corp., and AMD/ATI Corp. for their generous support through equipment and software grants.

REFERENCES

[1] P. Hart, N. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *Systems Science and Cybernetics, IEEE Transactions on*, vol. 4, no. 2, pp. 100–107, July 1968.

[2] R. W. Floyd, "Algorithm 97: Shortest path," *Commun. ACM*, vol. 5, no. 6, p. 345, 1962.

[3] S. Warshall, "A theorem on boolean matrices," *J. ACM*, vol. 9, no. 1, pp. 11–12, 1962.

[4] D. B. Johnson, "Efficient algorithms for shortest paths in sparse networks," *J. ACM*, vol. 24, no. 1, pp. 1–13, 1977.

[5] A. Botea, M. Muller, and J. Schaeffer, "Near optimal hierarchical path-finding," *Journal of Game Development*, vol. 1, pp. 7–28, 2004.

[6] N. R. Sturtevant, "Memory-efficient abstractions for pathfinding," *Proceedings of the Third Artificial Intelligience and Interactive Digital Entertainment Conference*, pp. 31–36, 2006.

[7] S. NR, F. A, B. M, S. J, and B. N, "Memory-based heuristics for explicit state spaces," *International Joint Conference on Artificial Intelligence*, pp. 609–614, 2009.

[8] Y. Bjornsson and K. Halldorsson, "Improved heuristics for optimal path-finding on game maps," *Proceedings of the Second Artificial Intelligience and Interactive Digital Entertainment Conference*, pp. 9–14, 2006.

[9] R. C. Holte, M. B. Perez, R. M. Zimmer, and A. J. MacDonald, "Hierarchical a*: Searching abstraction hierarchies efficiently," in *AAAI/IAAI, Vol. 1*, pp. 530–535, 1996.

[10] T. Cazenave, "Optimizations of data structures, heuristics and algorithms for path-finding on maps," in *Computational Intelligence and Games, 2006 IEEE Symposium on*, pp. 27–33, IEEE, 2007.

[11] S. J. Louis and G. Kendall, eds., *Proceedings of the 2006 IEEE Symposium on Computational Intelligence and Games (CIG06), University of Nevada, Reno, campus in Reno/Lake Tahoe, 22-24 May, 2006*, IEEE, 2006.

[12] R. E. Korf, "Depth-first iterative-deepening: an optimal admissible tree search," *Artif. Intell.*, vol. 27, no. 1, pp. 97–109, 1985.

[13] Y. Bjornsson, M. Enzenberger, R. Holte, and J. Schaeffer, "Fringe search: Beating a* at pathfinding on computer game maps," *Proceedings of the IEEE Symposium on Computational Intelligence in Games*, pp. 125–132, 2005.

[14] S. Koenig and M. Likhachev, "Improved fast replanning for robot navigation in unknown terrain," in *in Proceedings of the International Conference on Robotics and Automation*, pp. 968–975, 2002.

[15] H. Bast, S. Funke, D. Matijevic, P. Sanders, and D. Schultes, "In transit to constant time shortest-path queries in road networks," in *9th Workshop on Algorithm Engineering and Experiments (ALENEX'07)*, (New Orleans, USA).

[16] H. Bast, S. Funke, P. Sanders, and D. Schultes, "Fast routing in road networks with transit nodes," *Science*, vol. 316, no. 5824, p. 566, 2007.

[17] E. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[18] P. Sanders and D. Schultes, "Highway hierarchies hasten exact shortest path queries," *Algorithms–ESA 2005*, pp. 568–579, 2005.

[19] N. Jing, Y.-W. Huang, and E. A. Rundensteiner, "Hierarchical optimization of optimal path finding for transportation applications," in *CIKM*, pp. 261–268, 1996.

[20] K. Baedeker, "Northern italy handbook for travelers," 1913.

[21] G. Karypis and V. Kumar, “A fast and high quality multilevel scheme for partitioning irregular graphs,” Tech. Rep. TR 95-035, 1995.

[22] S. Koenig, M. Likhachev, and D. Furcy, “Lifelong planning a*,” *Artif. Intell.*, vol. 155, no. 1-2, pp. 93–146, 2004.