Hierarchical Path-Finding for Navigation Meshes (*HNA**)

Nuria Pelechano, Carlos Fuentes

Universitat Politècnica de Catalunya

Abstract

Path-finding can become an important bottleneck as both the size of the virtual environments and the number of agents navigating them increase. It is important to develop techniques that can be efficiently applied to any environment independently of its abstract representation. In this paper we present a hierarchical NavMesh representation to speed up path-finding. Hierarchical path-finding (*HPA**) has been successfully applied to regular grids, but there is a need to extend the benefits of this method to polygonal navigation meshes. As opposed to regular grids, navigation meshes offer representations with higher accuracy regarding the underlying geometry, while containing a smaller number of cells. Therefore, we present a bottom-up method to create a hierarchical representation based on a multilevel k-way partitioning algorithm (*MLkP*), annotated with sub-paths that can be accessed online by our Hierarchical NavMesh Path-finding algorithm (*HNA**). The algorithm benefits from searching in graphs with a much smaller number of cells, thus performing up to 7.7 times faster than traditional A* over the initial NavMesh. We present results of *HNA** over a variety of scenarios and discuss the benefits of the algorithm together with areas for improvement.

Keywords: path-finding, hierarchical representations, navigation meshes

---

1. Introduction

Most video games are required to simulate thousands or millions of agents who interact and navigate in a 3D world and show capabilities such as chasing, seeking or intercepting other agents. Path-finding provides characters with the ability to navigate autonomously in a virtual environment. The most well known path-finding algorithm is A*, which explores the nodes of a graph while balancing the accumulated cost with a heuristic to find an optimal path quickly. Throughout the years many algorithms have been proposed to further speed up the basic A* algorithm, but the cost of these algorithms is still strongly dependent on the size of the graph. Hierarchical path-finding aims to reduce the number of nodes that need to be explored when computing paths in large terrains. The reduction in the number of nodes for higher levels of the hierarchy significantly decreases the execution time and memory footprint when calculating paths.

Current hierarchical techniques may result in unbalanced abstractions. For example, top-down hierarchies are created by splitting the environment into large square clusters, where all the clusters contain the exact same number of lower level grid cells. The main disadvantages of such constructions are that the resulting higher level of the hierarchy may have an uneven number of edges between nodes and also an uneven number of walkable cells (since there may be some clusters with a large percentage of the grid cells being occupied by obstacles).

Navigation meshes represented by polygons provide closer representation of the geometry with a lower number of cells than regular grids. Since having a smaller number of cells can greatly accelerate path-finding, it is therefore necessary to extend the concept of hierarchical path-finding to a more general representation of navigation meshes with polygon based cells. Moreover it would also be beneficial to have a hierarchical representation with a balanced number of polygons per node and portals between nodes.

In this paper we present a new hierarchical path-finding solution for large 3D environments represented with polygonal navigation meshes. The presented solution works with navigation meshes where cells are convex polygons, and thus it also includes triangular representations. Our hierarchical graph representation is based on a multilevel k-way partitioning algorithm annotated with sub-path information. Our method presents a flexible approach in terms of both the number of levels used in the hierarchy and the number of polygons to merge between levels of the hierarchy. We evaluate the gains in performance when using our hierarchical path-finding, and discuss the tradeoffs between the number of merged polygons and the number of levels employed for the search. We present a number of benchmarks that can help during the parameter fitting process to achieve the best speedups, as well as a quantitative analysis of the bounds on sub-optimality of the paths found with *HNA**. We also present an evaluation of the bottleneck that appears for certain configurations when inserting the start and goal positions in the hierarchical representation.

2. Related Work

A large amount of work to speed up path-finding focuses on enhancing the A* algorithm to reduce the computational time needed to calculate a path. This comes at the cost of finding sub-optimal paths or allowing a certain degree of error when searching for the optimal path and then allows the algorithm to

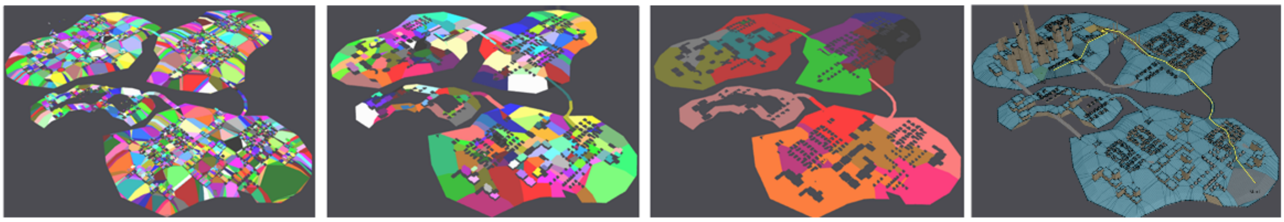

L0: 5151 nodes
L2: 316 nodes
L4: 17 nodes
Path in NavMesh

Figure 1: Hierarchical partition of a polygonal navigation mesh of over 5000 nodes at level 0 (each color identifies a node in the graph), 316 at level 2 and 17 at level 4, and the final path calculated with *HNA**.

repair those errors in future searches that are interleaved with the execution.

The well known A* algorithm [1] is a robust and simple to implement method with strict guarantees on optimality and completeness of solution. The A* algorithm uses a heuristic to restrict the number of states that must be evaluated before finding the true optimal path and it guarantees to expand an equal number or fewer states than any other algorithm using the same heuristic. However A* can be very time consuming for large scenarios. Anytime Planning algorithms find the best suboptimal plan and iteratively improve this plan while reusing previous plan efforts. One of the most popular A$*$ is called Anytime Repairing A* (ARA*) [2]. It performs a series of repeated weighted A* searches while iteratively decreasing a loose bound (ε). It iteratively improves the solution by reducing ε and reusing previous plan efforts to accelerate subsequent searches. However ARA* solutions are no longer guaranteed to be optimal.

D* Lite [3] performs A* to generate an initial solution and repairs its previous solution to accommodate world changes by reusing as much of its previous search efforts as possible. D* can correct ”mistakes” without re-planning from scratch, but requires more memory. Anytime Dynamic A* (AD*) [4] combines the properties of D* and ARA* to provide a planning solution that meets strict time constraints. It efficiently updates its solutions to accommodate dynamic changes in the environment.

DBA* algorithm [5] combines the memory-efficient sector abstraction developed for [6] and the path database used by [7] in order to improve space complexity and optimality. Huang [8] presented a path planning method for coherent and persistent groups in arbitrarily complex navigation mesh environments. The group is modeled as a deformable and splittable area preserving shape. The efficiency of the group search is determined by three factors: path length, deformation minimization, and spitting minimization.

Hierarchical graph representations have also been used for visualization purposes of large data sets [9] [10]. The goal in these applications is to offer an overview first, and then be able to zoom and filter to offer details on demand.

Planning via hierarchical representation has been used to improve performance in problem solving for a long time [11]. Holte et. al. [12] introduced hierachical A* to search in an abstract space and use the solution to guide search in the original space. There has also been work on abstraction based on bottom-up approaches for general graphs [13][14] but without considering balancing the number of nodes or minimizing the edge-cut. Sturtevant and Jansen [15] extended the theoretical work slightly and provided examples of a number of different abstraction types over graphs. In this work graphs are created from 2D grid-like structures by setting a node for each walkable cell . Bulitko et al [16] showed that the quality of paths can decrease exponentially with each level of abstraction. Sturtevant and Geisberger [17] studied the combination of abstraction and contraction hierarchies to speed up path-finding. Abstraction uses a top-down approach creating a 16x16 overlay across the lower level regular grid. Contraction builds a higher level graph using the concept of importance of nodes, which requires priorities for the nodes to be set correctly as they will affect the contraction algorithm.

Hierarchical representations have been used over 2D grid representations [18]. In [19] an adaptive subdivision of the environment is proposed with efficient indexing, updating, and neighbor-finding operations on the GPU which reduces the memory requirements. Another similar method based on *HPA**, but taking into account the size of the agents and terrain traversal capabilities, is Hierarchical Annotated A* (HAA*) [20]. It presents an extension of *HPA** which allow multi-size agents to efficiently plan high quality paths in heterogeneous-terrain environments. Another interesting implementation is *DT-HPA** [21] which uses a decision tree to create a hierarchical subdivision.

Jorgensen presented an automatic structuring method based on a hierarchy that separated buildings into floors linked by stairs and represents floors as rooms linked by doorsteps [22]. This method has a strict hierarchy and does not scale to large outdoors environments such as the ones often presented in video games. Zlatanova [23] presented a framework of space subdivision exclusively for indoor navigation, by identifying rooms and corridors and including semantical information.

There are other approaches that focus on allowing agents to be more environment-aware [24]. In this work planning is based on an Anytime Dynamic A*, and it is carried out satisfying multiple special constraints imposed on the path, such as: stay behind a building, walk along walls or avoid the line of sight of other agents. In [25] a multi-domain anytime dynamic planning framework is presented which can efficiently work across multiple domains by using plans in one domain to accelerate and focus searches in more complex domains. It explores different domain relationships including the use of waypoints and tunnels. The different domains use only two representations in terms of spacial subdivision, a 2D grid, and a triangular mesh.

Hierarchical representations have been used to calculate agents moving between two points at different levels of complexity [26] [27]; from finding a route to animating 3D characters. They have also been used to combine high level path-finding with low level local motion [28]. When using triangular representations, it is possible to optimize the data structures and built in features such as clearance that can greatly improve performance during path-finding [29] [30]. But it is not straight forward to extend this implementation to polygonal meshes (i.e. it would not be enough with a simple triangulation of the polygons). There has been a recent technical report extending *HPA** to triangular representations [31].

As most of the abstract representations for large 3D complex environments employ polygon based representations (e.g: NEOGEN [32], Recast [33], or navmeshes built from the medial axis [34]), it is thus necessary to extend the concept of hierarchical path-finding for general representations of navigation meshes. Polygonal meshes have certain features and characteristics that must be taken into account when evaluating the most suitable hierarchical abstraction to be used.

3. Framework

Our framework consists of a pre-processing phase where the hierarchy is created, and an adapted version of the basic A* algorithm to perform searches online in this hierarchical representation.

The pre-process phase starts with a polygonal navigation mesh that represents an abstract partition of the 3D world. This first navigation mesh is considered to be the lowest level in a hierarchical tree. The rest of the levels in the hierarchy are created by recursively partitioning a lower level graph into a specific number of nodes. The partition is performed until the graph of the highest level cannot be further subdivided. Thus, a particular path planning search can be executed in any level of this hierarchical tree. The higher the level of the hierarchy, the fewer the number of nodes to search in. This approach allows faster path-finding calculations than using a common A* without any hierarchy. Although we have tested our results using the basic A* algorithm, the method presented is general enough to be used with improved versions of A* such as AD*, DBA*, ARA* or D*.

The classic hierarchical path-finding algorithm (*HPA** [18]) for 2D grids consists of having the 2D grid as low level, and builds a higher level by dividing the environment into squared clusters connected by entrances, where all clusters have the same number of low level grid cells. Clusters are connected with inter-edges with cost 1.0 and the cost of intra-edges are calculated with A* [1] algorithm searches inside each cluster, for all pairs of abstract nodes that shared the same cluster.

Gravot et. at. [35] presented a top-down approach to combine a 2D grid partition of large tiles, with a lower level navigation mesh per tile. So each tile of 32x32 meters has its own navigation mesh, which forces the number of cells to be larger than when the polygon decomposition is generated directly from the original map. This 2-level representation improves performance, but the misalignment between axis aligned tiles and geometry causes inconsistencies in the pre-stored tables that force farther subsplitting of tiles.

In this work we propose a bottom-up approach that starts with the initial navigation and it merges cells to obtain a higher level of abstraction respecting the advantages of polygonal navigation meshes. Grouping low level cells in a general navigation mesh is not as straight forward as deciding to group squares of $n \times n$ cells. The goal is to have a good graph partition with a balanced size of components and a small number of edges running between components, as this will reduce the costs of the hierarchical path-finding algorithm. We use a polygon mesh provided by Recast [33] as our initial navigation graph and the multilevel k-way partitioning algorithm (*MLkP*) [36] to create our hierarchical representation. *MLkP* reduces the size of graph G_i to create G_{i+1} by collapsing vertices and edges. This algorithm has been proven to be faster than other multilevel recursive bisection algorithms, and produces high quality graphs.

3.1. Hierarchical representation

The first step is to build the framework for hierarchical searches that is defined as a tree of graphs. We start to compute the lowest graph of the hierarchy ($G_0 = (V_0, E_0)$) by searching the polygons in the original navigation mesh. Each polygon becomes a node in the G_0 graph and edges are created between polygons that share a border in the original mesh.

We define L_{max} as the maximum number of levels for the hierarchical representation, and η as the number of nodes that will be merged between levels of the hierarchy. Once the lowest level graph G_0 is created, the upper levels of the hierarchy $\{G_1, G_2, ..., G_m\}$ are recursively built by partitioning each level until it reaches the minimum number of the nodes in a graph or $m = L_{max}$.

The *MLkP* algorithm starts with a *coarsening phase*, which consists of creating a series of successively smaller graphs derived from the input graph. Each graph is constructed from the previous graph by collapsing together a maximal size set of adjacent pairs of nodes. After the coarsening phase, a *k*-way partitioning of the smallest graph is computed (*initial partitioning phase*). Next the *uncoarsening phase* begins by projecting the partitioning of the smallest graph into the successively larger graphs, refining the partitioning at each intermediate level. The different phases of the multilevel paradigm are illustrated in Fig. 2.

In order to have a good partition the weight of a new node should be equal to the sum of its previous nodes. In our case we are interested in having a balanced number of polygons, therefore nodes in $G0$ are initialized with weight=1. The new edges created are the union of the edges from the previous nodes to preserve the connectivity information in the coarser graph. The coarsening phase ends either when the coarsest graph has a small number of nodes or when the reduction in the size of successively coarser graphs becomes smaller than a given threshold.

The *initial partitioning phase* is performed using a multilevel bisection algorithm [36]. Each partition contains roughly $|V_0|/k$ nodes' weight of the original graph. The division is done by KernighanLin (KL) partitioning algorithm [37] which finds a partition of a node into two disjoint subsets of equal size such that the sum of the weights of the edges between those subsets is minimized.

The uncorseaning phase initially projects the partition by assigning the same partition to the collapsed nodes. After each projection step, the partitioning is refined using various heuristic methods to iteratively move nodes between partitions as long as such movements improve the quality of the partitioning solution. The uncoarsening phase ends when the partitioning solution has been projected all the way to the original graph.

This multilevel partitioning process provides a hierarchy of graphs, where the lowest graph $G_0 = (V_0, E_0)$ corresponds to the original NavMesh of the environment, V_0 is the set $v_0^1, v_0^2, ..., v_0^n$, where each v_0^j is a node representing a polygon of the NavMesh, and E_0 is the set of edges that correspond to portals between nodes of the original NavMesh. Therefore each graph $G_i = (V_i, E_i)$ consists of a set of nodes V_i where each node v_i^j represents a multinode collapsing several adjacent nodes of the lower graph G_{i-1}, i.e: $v_i^j = \{v_{i-1}^1, v_{i-1}^2, ..., v_{i-1}^\gamma\}$.

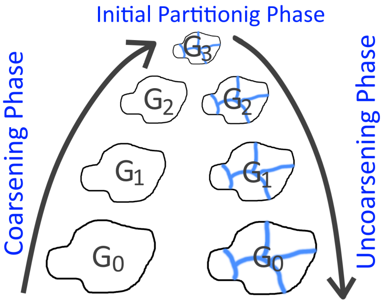


Figure 2: Multilevel k-way partitioning scheme [36].

The procedure allows us to have partitions which ensure high quality edge-cuts, where an edge-cut is defined as the number of edges whose incident nodes belong to different partitions.

The partition is carried out using the *METIS* software package [38], and after the first partition done from G_0 to G_1 all non accessible nodes returned from the NavMesh creation in Recast are eliminated from the hierarchy.

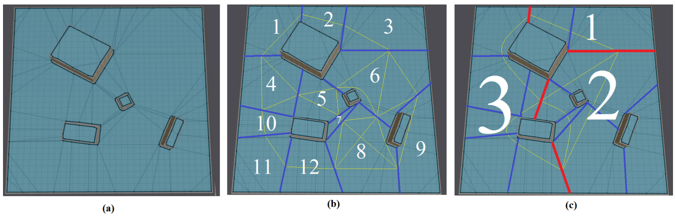
The iteration is done until either it reaches the maximum number of levels in the hierarchy or the graph cannot be further subdivided. The number of merged nodes per level to create a new partition is given by the user defined variable η, where $\eta \approx |V_0|/k$.

Once we have the partitions $\mathcal{P}$, the new nodes and edges between partitions are created. Edges between partitions are the inter-edges of the graph and contain the edges of the lower graph that join different partitions in the higher graph. Therefore, each partition $\mathcal{P}_i$ has a set of inter-edges E_i which depends on the edges of E_{i-1} that connect nodes of V_{i-1} which fall in different partitions of $\mathcal{P}_i$. For each pair of inter-edges in a node v_i of the given partition $\mathcal{P}_i$, A* is applied between them to calculate the cost of the shortest path and store it as an intra-edge for the given node. For all the graphs of the hierarchy, starting from G_1 and moving up to the highest level (note that G_0 does not contain intra-edges), the Hierarchical NavMesh Graph (*HNG*) is created as indicated in the following algorithm:

Algorithm 1 . Build HNG

```
procedure BUILDGRAPH($G_i$)
    for $j \leftarrow 1, |\mathcal{P}_i|$ do
        for $n \leftarrow v_i^1, |V_i^j|$ do
            for $e \leftarrow 1, numEdges(n)$ do
                $m = neighbour(n, e)$
                if $p[n] \neq p[m]$ then
                    $G_{i+1}.addInterEdge(V_{i+1}(n), V_{i+1}(m))$
        for $k \leftarrow 1, v_{i+1}^j.numEdges()$ do
            for $l \leftarrow k + 1, v_{i+1}.numEdges()$ do
                $cost \leftarrow findPath(k, l)$
                $G_i.addIntraEdge(k, l, cost)$
```

Partitions will contain the intra-edges for each pair of edges within a node. Figure 3 shows a simple example with the partitions, inter-edges and intra-edges created. Figure 1 represents $levels = 0, 2, 4$ of the hierarchical partition for a map with 5,515 polygons, with $\mu = 5$ and $L_{max} = 5$.



(a) (b) (c)

Figure 3: Hierarchical subdivision of a simple map, with $\mu = 5$ and $levels = 5$. Red lines in (c) represent inter-edges and yellow lines in (b) and (c) represent intra-edges. Partitions are shown with black (a), blue (b) and red (c) separation lines respectively. Level 0=76 nodes (a), Level 1=12 nodes (b), Level 2=3 nodes (c).

3.2. *Hierarchical path-finding*

Path-finding can be performed at any level of the hierarchy. For given starting and goal positions S and G we need to link this position to the *HNG* and then perform *HNA** in the temporally created graph (note that S and G are linked to the *HNG* and removed once the path is calculated). Note that the algorithm for hierarchical path-finding is conceptually similar to *HPA** [18] but has been adapted to the *HNG* introduced in the previous section. The algorithm proceeds through the following steps:

1. Insert the starting S and goal G positions at the desired level of the hierarchy and connect them to the higher level graph.
2. Search path between S and G at the highest level.
3. Extract intra-edges (optimal sub-paths).

4. Delete temporal nodes.

Algorithm 2 indicates the details of each step of the *HNA** algorithm. Note that currently the function *findPath()* simply calculates A* over the given graph at the level of the hierarchy indicated by the last parameter and heuristic based on Euclidean distance.

Algorithm 2 . Online *HNA**

```
procedure FINDPATHHNA*(S, G, L)
    //step 1. Insert and connect nodes $S$ and $G$ at level $l$
    $n_l^s \leftarrow getNode(S, l)$
    $n_l^g \leftarrow getNode(G, l)$
    if $l = 0$ then
        $path \leftarrow findPath(n_l^s, S, n_l^g, G, 0)$
        return $path$
    $n_{aux}^s \leftarrow linkStartToGraph(S, n_l^s)$
    $n_{aux}^g \leftarrow linkGoalToGraph(G, n_l^g)$
    //step 2. Path-finding between $S$ and $G$ at level $l$:
    $tempPath \leftarrow findPath(n_{aux}^s, S, n_{aux}^g, G, l)$
    //step 3. Extract sub-paths:
    for $subpath \in tempPath$ do
        $path \leftarrow getIntraEdges(subpath, l-1)$
    //step 4. Delete $S$ and $G$:
    $deleteTempNode(n_{aux}^s)$
    $deleteTempNode(n_{aux}^g)$
    return $path$
```

3.2.1. *Inserting S and G and connecting to the graph*

The starting S and goal G positions are inserted in the geometry at level 0 and then recursively looked up the hierarchy for the corresponding nodes at the highest level, L of the hierarchy. S and G are then temporally inserted in the higher level graph G_L as temporal nodes n_{aux}^s and n_{aux}^g respectively.

To connect the temporal node n_{aux}^s with the graph G_L we need to calculate the path from S to each of the inter-edges of higher level node n_L^s containing S. Inter-edges are the union of those edges from G_0 that connect a node n_0^i with a node n_0^j where $p_L[n_0^i] \neq p_L[n_0^j]$, (i.e nodes of level 0 that are neighbors but belong to different partitions of G_L).

The paths between S and each inter-edge, e_L^j, of n_L^s are calculated to create a temporal intra-edge linking n_{aux}^s to the higher level graph G_L. Similarly, temporal intra-edges are calculated linking the goal position G to the graph G_L (see figure 4a for an example of the temporal intra-edges used to connect S and G with the graph at the higher level).

The performance of this step depends on the computational cost of calculating each intra-edge for S and G. In the case of the starting position, it requires calculating paths between S and each edge e_L^j of the node n_L^s. The same applies to connecting the goal position G within its node.

The path-finding algorithm used to connect S and G is independent of the algorithm used at the higher level, since the problem is quite different. In this case we are not finding a path between two points, but finding all the shortest paths between one point (S or G) and many (all edges within the node). We have tested two algorithms, A* and Dijkstra [39].

A* is a faster algorithm than Dijkstra since it uses heuristics to expand less nodes. However in this particular scenario where several A* have to be performed, there will be a number of nodes explored multiple times for each search. Therefore, even though Dijkstra is meant to be slower in finding a single solution, when it comes to finding paths to multiple goals we may benefit from the fact that we only need to run the search once and stop as soon as it finds the last edge of the node.

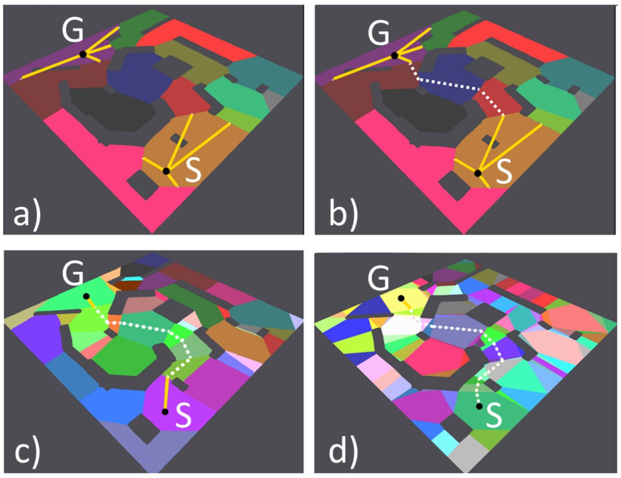


Figure 4: Path-finding computation: S and G are inserted and linked to their partitions at level 2 by calculating shortest paths to each portal in their respective node(a). Paths are calculated at level 2 (b), and then intra-edges are extracted from lower level 1 (c) and the final path is obtained for level 0 (d).

3.2.2. *Search path between S and G at the highest level*

Once the S and G are temporally connected to the higher level graph, path-finding is computed with the A* algorithm in the hierarchical navigation graph (*HNG*) formed by all the nodes in the higher level of the hierarchy and the connection to n_{aux}^s and n_{aux}^g. This path-finding at level i results in the following sequence:

$$ie(n_{aux}^s - v_i^1), v_i^1, v_i^2, ..., v_i^m, ie(v_i^m - n_{aux}^g)$$

Note that $ie(n_{aux}^s - v_i^1)$ contains the sequence of nodes at level 0 that belong to one of the temporal intra-edges added during the connection of S with the first high level node of the path v_i^1, and similarly $ie(v_i^m - n_{aux}^g)$ contains the sequence of nodes at level 0 between the last high level node of the path v_i^m and the goal position G (see figure 4b where the yellow lines indicate the temporal intra-edges created for S and G, and the white dotted lines the intra-edges to go through the highest level nodes of the graph).

The time execution of this path-finding at level i is significantly faster than finding the path at level 0 due to the large reduction in the number of nodes.

3.2.3. *Extract intra-edges*

For the given sequence of high level nodes $\{v_i^1, v_i^2, ..., v_i^m\}$ belonging to the optimal solution for level i, the algorithm re-

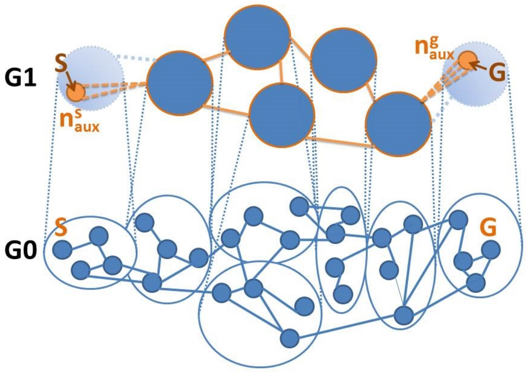


Figure 5: Example of *HNG* with two levels and $\mu = 4$. The orange links and circles represent the edges and nodes that belong the the temporal graph created after linking S and G to the *HNG*. This temporal graph is where the *HNA** is calculated.

cursively extracts the intra-edges for each lower node. The final sequence of intra-edges once level 0 has been reached is the actual path (sequence of polygons in the NavMesh) that the agents need to follow to move from S to G.

3.2.4. Delete temporal nodes

The final and simplest step consists of deleting the temporal nodes n^s_{aux} and n^g_{aux} from the graph, and all their temporal intra-edges. After this step, we recover the original *HNG* to perform future searches.

4. Results

For the evaluation of our method we have used several multilayer 3D scenarios as shown in figure 6 with increasing numbers of cells in the original NavMesh (see table 1 for details on the number of nodes in the map).

Table 1: For each map in figure 6, we show the number of triangles in the original geometry, and the number of nodes in the NavMesh depending on whether we use triangles or polygons.

Map Name	Geometry # Triangles	NavMesh # Triangles	NavMesh # Poly
Serpentine City (a)	135.1K	10,152	3,908
City Islands (b)	110.3K	14,551	5,515
Tropical Islands (c)	239.1K	29,499	12,666

We have calculated a large number of paths over each of these scenarios with increasing values of μ on increasing numbers of levels in the hierarchy to determine which are the best configurations for hierarchical path-finding. Results show that we can achieve significant speedups for certain configurations, while we may get even worse results than A* for other configurations. Therefore in this section we evaluate the overall performance of the algorithm, looking closely at the computational time taken by each step of the *HNA** algorithm (see alg. 2) to determine areas for improvement.

Figure 7 shows the reduction in the number of nodes as we increase the value of μ and the number of levels in the hierarchy. The reduction for the first level is the largest one since we also remove unconnected polygons during the first step of the algorithm. From then on the reduction is due to collapsing nodes based on the value of μ. As we will see when we compute the overall performance of the algorithm, our experimental results show that the most suitable configurations tend to happen when the number of polygons has been reduced around 12% for level 1 (with $\mu \approx 20$), and the second best configuration tends to happen when the number of polygons has been reduced to approximately 2.5% for level 2 (with $\mu \approx 6$).

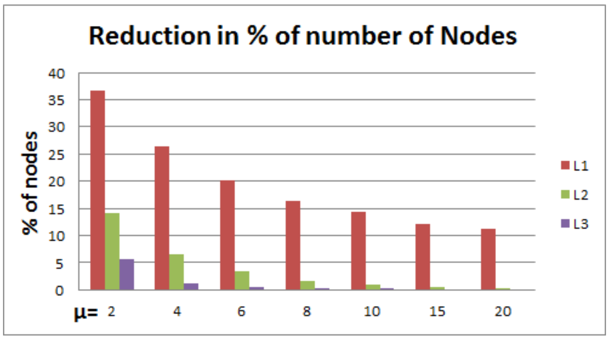


Figure 7: The table shows the percentage of nodes of the original NavMesh for different values of μ and different levels of the hierarchy.

To calculate the overall computational time of *HNA** and compare results, we have computed the average cost of calculating a large number of paths as shown in figures 8, 9 and 10 with an intel core i7-4770 CPU@3.5Gz, 16GB RAM.

For the City island scenario consisting of a NavMesh with 5,515 polygons, we have tested up to 3 levels and increasing values of $\mu = \{2, 4, 6, 8, 10, 15, 20\}$. As we can see in figure 8a, the average cost of performing A* in this scenario is 2.02ms. Using *HNA** we can improve performance with $L1$ and all the values of μ tested ($\mu \in \{2, 20\}$) with the fastest search being 0.51ms for $\mu = 15$. A hierarchy of two levels also improves the computational times for $\mu \in \{2, 20\}$. However for the case of having a hierarchy consisting of 3 levels, we only obtain speedups for $\mu < 7$, since once we collapse 8 or more nodes between levels the total cost is actually worse than simply computing A* at $L0$. To better understand why the computational cost can increase for certain values of μ and levels in the hierarchy, we have displayed the cost of *HNA** at $L1$ and $L2$ in figure 8 (b) and (c) using different colors for each of the significant steps of the algorithm.

The significant steps of the algorithm are: (1) calculating A* at the higher level, (2) extracting intra-edges and (3) connecting S and G within the higher node (Note that the other steps of the algorithm have an insignificant cost below 0.007ms). As we can see in this figure, the cost of computing A* at the highest level decreases since the number of nodes becomes smaller by increasing levels and μ. However the cost of connecting S and G can escalate as the higher level nodes increase in size. This is mainly because as their size gets bigger, the number of inter-edges also becomes bigger, and thus it requires a higher

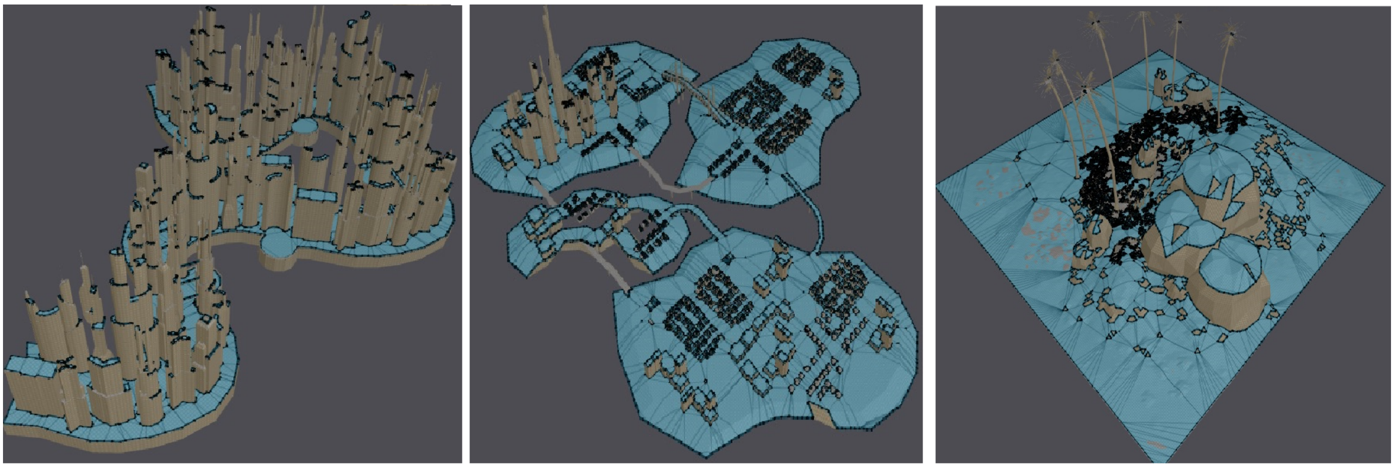

(a) (b) (c)

Figure 6: Scenarios used for evaluation (obtained from http://tf3dm.com/). (a) Serpentine city, (b) City island, (c) Tropical islands

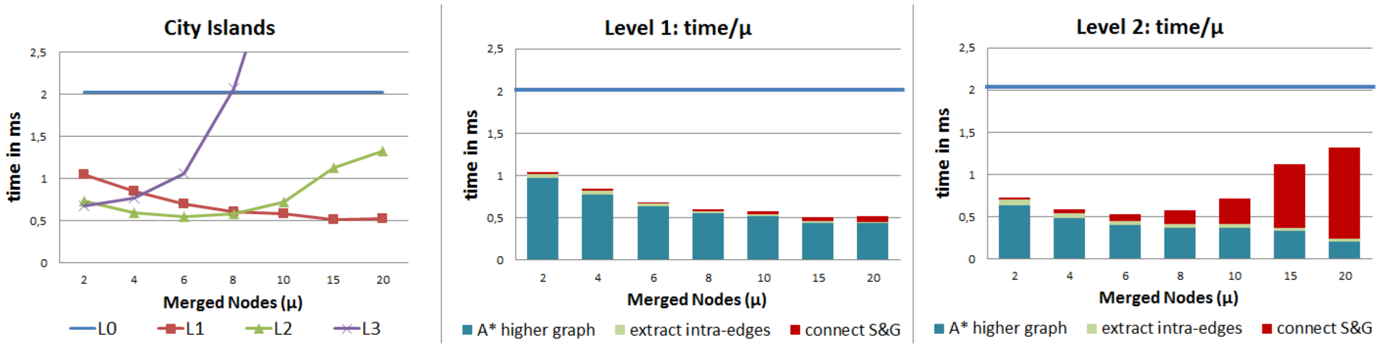


Figure 8: Performance results for the city island scenario (3 levels and $\mu = 2, 4, 6, 8, 10, 15, 20$). (a) show the cost of A* at $L0$, and $HNA*$ at $L1$, $L2$ and $L3$ as μ increases. (b) and (c) show the cost of the different steps of $HNA*$ for L1 and $L2$ respectively.

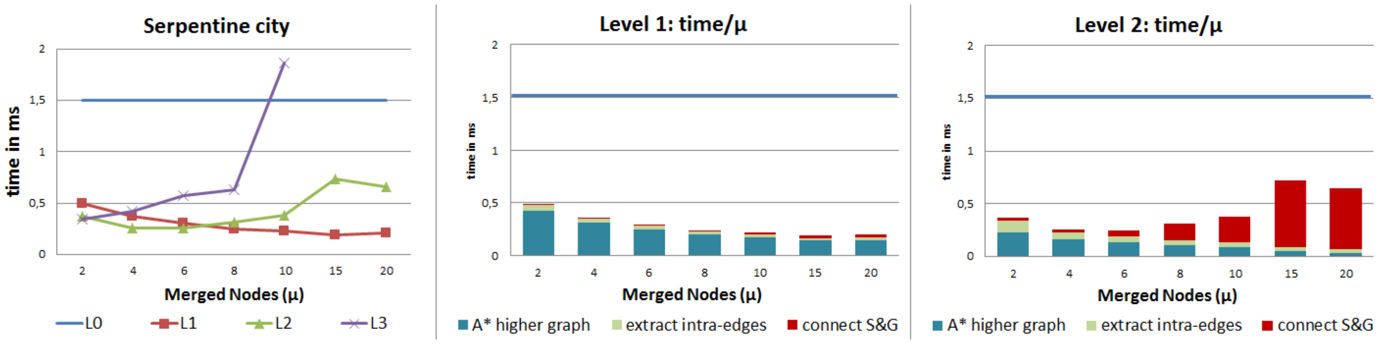


Figure 9: Performance results for the serpentine island scenario (3 levels and $\mu = 2, 4, 6, 8, 10, 15, 20$). (a) show the cost of A* at $L0$, and $HNA*$ at $L1$, $L2$ and $L3$ as μ increases. (b) and (c) show the cost of the different steps of $HNA*$ for L1 and $L2$ respectively.

number of A* searches to compute temporal intra-edges to connect S and G with the HNG.

From our experimental results, we observed that replacing A* by Dijkstra to perform the connection step can improve performance. However the difference is only significant for very large nodes with many inter-edges, while it is almost the same for the configurations where $HNA*$ outperforms A* at L0. Therefore there is still room for improvement in this connection step.

In the serpentine city scenario consisting of a NavMesh with 3,908 polygons, we have tested up to 3 levels and $\mu = \{2, 4, 6, 8, 10, 15, 20\}$. As we can see in figure 9a, the average cost of performing A* in this scenario is 1.5ms. By using $HNA*$ we can improve performance in $L1$ and $L2$ for all the μ values tested, with the fastest search observed for $\mu \in [15, 20]$ and $L1$ when it takes 0.19ms on average to compute a path. This represents a 7.7x speedup over basic A*. As in the previous scenario, for $L3$ we only observe faster searches for small values of μ. In

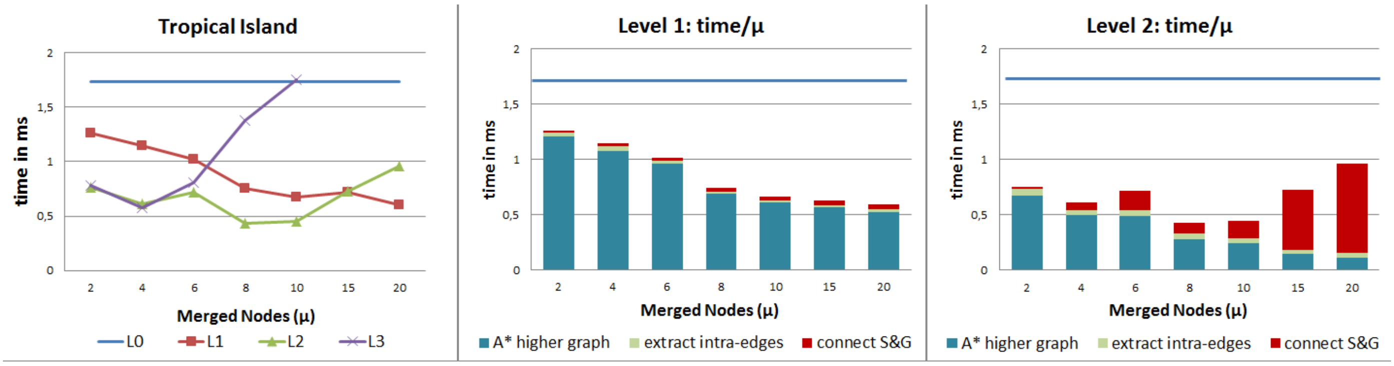


Figure 10: Performance results for the tropical island scenario (3 levels and $\mu = 2, 4, 6, 8, 10, 15, 20$). (a) show the cost of A* at $L0$, and *HNA** at $L1$, $L2$ and $L3$ as μ increases. (b) and (c) show the cost of the different steps of *HNA** for L1 and $L2$ respectively.

figure 9 (b) and (c) we can differentiate the cost for each of the significant steps of the *HNA** algorithm.

In the tropical island scenario with an initial NavMesh of 12.666 polygons, we have also tested 3 levels of the hierarchy and $\mu = \{2, 4, 6, 8, 10, 15, 20\}$. The time taken by each configuration is shown in figure 10. For the combination of levels and values of μ tested in this scenario, the best speedup obtained is 4.0x for L2 and $\mu = 6$.

Therefore, the best speedups achieved by *HNA** have been 7.7x for the serpentine city, 3.9x for the city island, and 4.0x for the tropical city. At $L1$ the cost of the step connecting S and G is almost insignificant compared to the total cost of *HNA**, however from $L2$ onwards this step can become an important bottleneck for larger values of μ. This bottleneck depends largely on the differences in shape and connectivity of the original graph. For example the long structure of the serpentine city makes the edge-cut smaller on average, as merging a larger number of nodes does not increase the number of inter-edges as much as in the city or the tropical island scenarios. Therefore the speedup that can be achieved depends strongly on the configuration of the space and connectivity of the graph G_0.

Figure 11 shows the average number of inter-edges per multinode at levels L2 and L3 in the hierarchy as the value of μ increases. In general the number of inter-edges (i.e. the edge-cut) increases with the value of μ. However we can observe how for the serpentine scenario the number of inter-edges can actually drop significantly above a certain value of μ, as opposed to the other tested scenarios where it increases with μ.

The multilevel k-way partitioning algorithm used to create the *HNG* attempts to reduce the edge-cut while balancing the number of nodes per partition. Reducing the edge-cut will reduce the cost of connecting S and G, but in order to improve the results achieved by our algorithm, it would be necessary to find an alternative method for the step connecting S and G. As we can clearly see in the different results (figures 8-10), increasing both μ and levels always reduces the A* search at the higher level as the search is performed over smaller graphs.

Figure 12 illustrates an example of a worst case scenario for *HNA** where the highest level contains excessively large nodes with many inter-edges. This drastically increases the computational time of inserting and connecting S and G. In this example, the cost of *HNA** would be much higher than simply performing A* in the original NavMesh, since we are now computing 18 paths to connect S, and 10 paths to connect G. One advantage of having a multilevel hierarchy could be to perform the search dynamically at different levels when S and G belong to neighboring nodes of the highest level.

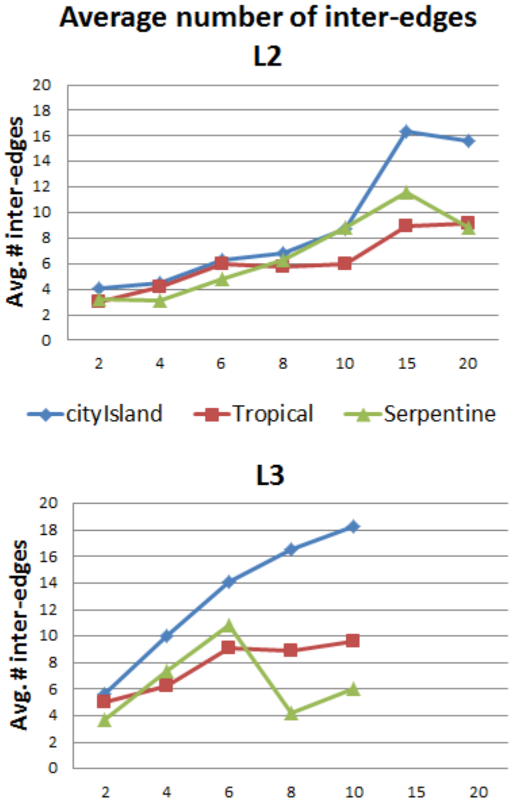


Figure 11: Average number of inter-edges per multinode for levels $L2$ and $L3$ of the hierarchy as the value of μ increases.

In terms of path quality, there are some differences between the paths found with A* over the NavMesh, and the ones obtained when applying *HNA**. These small deviations are due to the fact that intra-edges compute distances between the center points of edges, as opposed to A* that takes into account

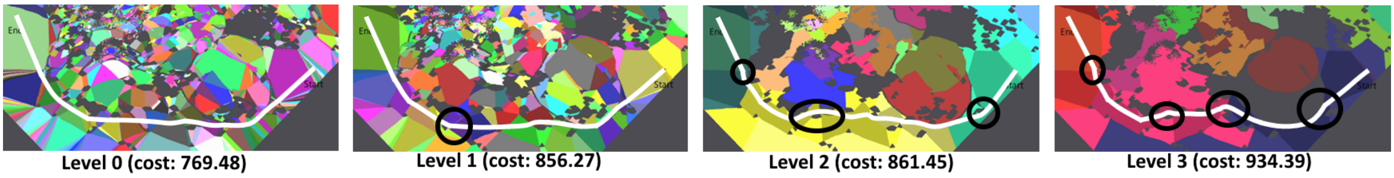


Figure 13: Example of paths calculated at different levels of the hierarchy for the Tropical Island scenario.

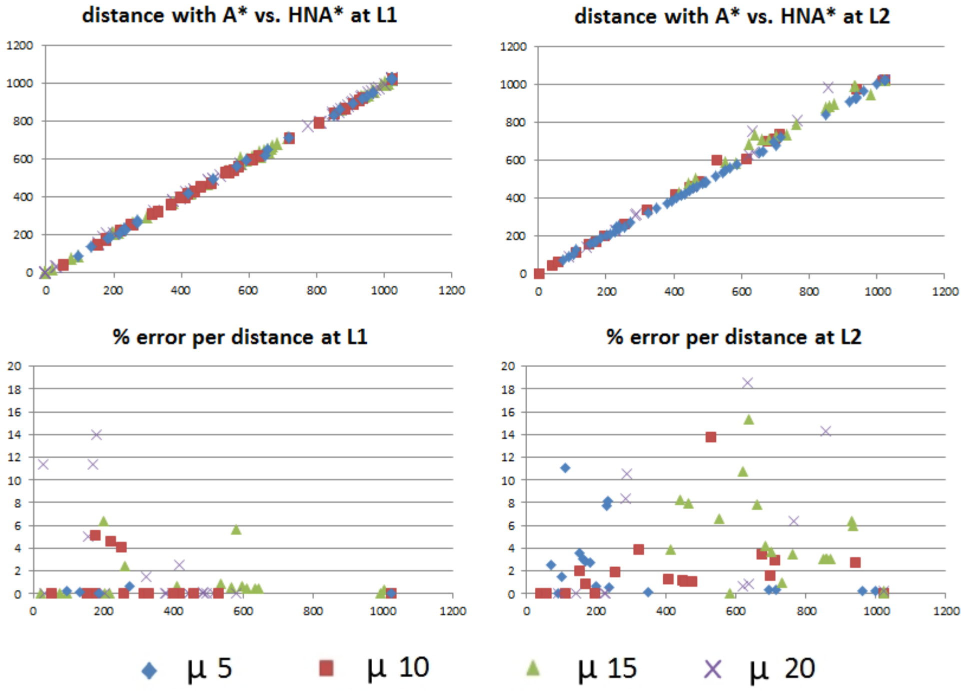


Figure 14: On the left, we show a comparison of path lengths obtained with $A*$ against $HNA*$ for different values of μ and level of search in the hierarchy. On the bottom row we show the percentage of error introduced as the length of the path increases.

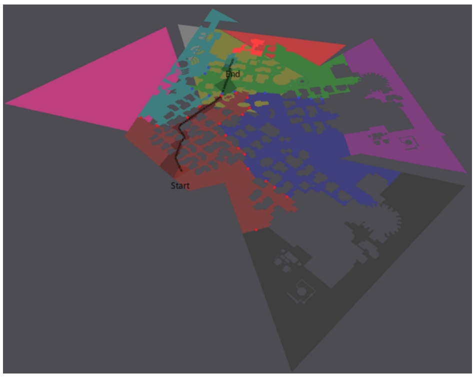


Figure 12: Example of a worst case scenario for connecting S and G. Edges shown as red dots for the cell containing S and blue dots for the cell containing G.

crossing portals through the closest point. In any case, since the paths for intra-edges are always computed using A* over the NavMesh, the impact does not propagate up the hierarchy (i.e. the cost of an intra-edge at level i calculated off-line is not the sum of the costs of the intra-edges at level $i-1$ but it is computed from scratch and stored). Figure 13 shows an example of path quality and cost in meters of the computed path for different levels of the hierarchy. We have chosen an example with a high error to show how as we increase the number of layers we can observe more deviation from the optimal route. In this particular example we observe that for levels 1 and 2 we get a path with an extra cost of around 10% and for level 3 it can add an extra cost of 20%. Note that the path differences happens between nodes of the higher level, or because paths are forced through the selected higher level node, when the optimal may be between two high level nodes.

Figure 14 shows a quantitative evaluation of the path length and percentage of error as the length of the path increases. The four graphs have on the X axis the length of the path between start and goal as computed by A*. The top row shows on the Y axis the length of the path given by the $HNA*$ for searches performed at level 1 (left) and level 2 (right), with $\mu = \{5, 10, 15, 20\}$. All points close to the line $x = y$ indicate that both paths have similar lengths. To highlight the error, we show on the bottom row the percentage of error (Y axis) for different path lengths. As we can observe, the results are on average very similar. The maximum error found for L2 was 18.6% ($\mu = 20$), 15.4%

(μ = 15), 13.8% (μ = 10) and 11.0% (μ = 5), and average errors of 0.39%, 0.17%, 0.11% and 0.05% respectively. The maximum error for L1 was 14% for $\mu = 20$, and approximately 6% for other values of μ.

Navigation meshes can represent a very large number of environments each with its own unique features that will make one configuration better than others. Nevertheless, we wanted to evaluate whether our decision of having a balanced number of cells and minimum edge-cut was in fact the best option towards achieving better speedups. We ran a comparison study using *MLkP* but assigning different weights to either the initial edges or the nodes of G_0. By doing so we obtained a graph partition where the number of nodes of G_{i-1} merged in a node of G_i would be different, and/or the number of inter-edges between nodes of the partition could vary significantly. Note that *MLkP* balances the weight given to the nodes of the G_0 graph. Therefore by randomly assigning different weights, we achieve an unbalanced partition in terms of the number of nodes per cluster. Similarly *MLkP* minimizes the weight of the edge-cut, so if the weights are randomly assigned to the original edges, then the final number of edges contained in each inter-edge will not be minimized. Figure 15 shows the results of this evaluation. We noticed that for small scenarios the differences were not very significant, but as the environment got bigger we could observe that in fact the balanced partition would provide on average slightly better results and also worst case scenarios closer to the average. This can easily be explained by the fact that an unbalanced number of nodes creates higher level graphs with more nodes, which increases the path finding at the higher level. When the smallest edge-cut is not guaranteed, the result may end up with some nodes having a large number of inter-edges which drastically increases the step connecting S or G to the graph. Since this step is the current bottleneck of the algorithm, we can observe in the figure how worst case scenarios can almost triple the total cost of the search.

5. Conclusions and Future Work

In this paper we have presented a novel algorithm to perform hierarchical path-finding over NavMeshes based on a bottom-up approach. Using a multilevel k-way partitioning algorithm, we can create a hierarchy of several levels of complexity with a decreasing number of nodes per level based on a user input variable μ that determines the approximate number of nodes to collapse between consecutive levels of the hierarchy. An advantage of our bottom-up approach as opposed to top-down approaches is that our technique provides a balanced number of both walkable cells and inter-edges between partitions. We have shown how our HNA* algorithm can obtain paths in this representation faster than when applying the basic path-finding directly over the navigation mesh.

A quantitative comparison between HNA* and HPA* would be interesting. However the main difficulty for such comparison is that HPA* is highly sensitive to the granularity of the grid, whereas HNA* does not suffer from this limitation. Therefore it would be hard to find the right parameters for a fair comparison. Nevertheless we expect the benefits of HNA* to become more noticeable as the environment complexity increases, because our bottom-up approach using *MLkP* partitioning provides a good balance of nodes and a minimal edge-cut, whereas this cannot be achieved with an axis aligned regular grid partition. Therefore as the environment increases in size and complexity, we expect HPA* to start suffering from this lack of balance.

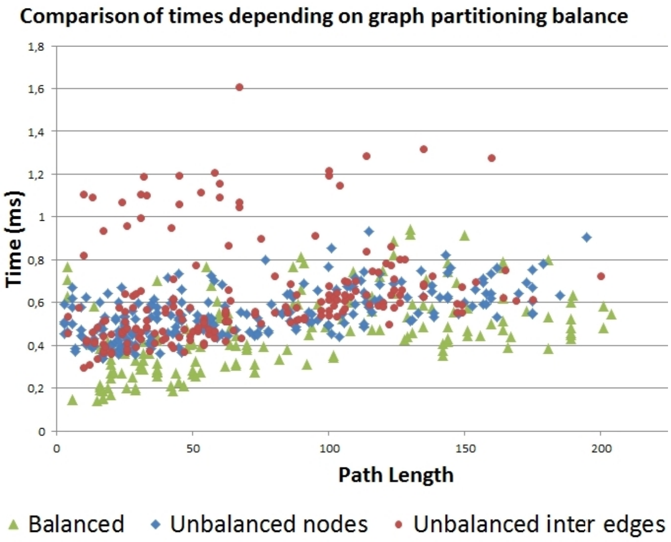


Figure 15: Time taken (in milliseconds) to compute paths with HNA* as the length of the paths increases. Results from the city island scenario, with two levels of hierarchy and $\mu = 6$

.

We have demonstrated results with the A* algorithm, but the architecture presented in this paper could also be used with other variants of A*.

We have shown improvements over a variety of scenarios to demonstrate the potential of the method, but have also evaluated its limitations. Currently the main limitation of this technique is the step that connects the starting and goal position into the hierarchical representation, since its performance drops as the number of level 0 nodes contained in the higher level node (multinode) increases. We have tested and compared two variants for this step, one consisting of calculating A* from S and G to each inter-edge in their respective higher level node, and the second by performing one single Dijkstra search for the node containing S and the node containing G. Despite Dijkstra presenting improvements over A*, it is not fast enough for the critical cases, therefore future work will focus on testing alternatives for this step such as parallel searches, or pre-computing and storing additional data structures to further improve performance. Pre-computing information on a per-cell basis would be more challenging than when working with regular 2D grids since there can be a large variation in shape and size of the initial cells, thus making it difficult to estimate the possible position for S and G.

We have observed that the best speedups can be achieved by having a one level hierarchy with $G1$ containing around 85% less nodes than $G0$, or when having a two level hierarchy where $G1$ has around 70% less nodes than $G0$, and $G2$ has approximately 95% less nodes than $G0$. Even though it may seem that the fastest and simplest option would be to have a one level hierarchy, it is important to emphasize that the comparisons have been done with average costs for a variety of paths in the graph. Therefore, it would be possible to further extend *HNA** to improve performance based on the location of S and G. For instance, the current algorithm checks whether S and G are in the same multinode, and if so it simply performs A* (meaning that this case does not benefit from having a hierarchical representation, but it is also not penalized). Moreover we have also shown that when S and G are in neighbouring nodes of the highest level, then the cost can be high since it is necessary to calculate multiple A* searches to connect S and G, and a negligible cost in finding the high level path. We believe that these two scenarios could benefit from calculating *HNA** in the next level of the multilevel representation. As future work we would also like to consider dynamic updates of the NavMesh and how they could affect the hierarchical representation.

Acknowledgements

This work has been partially by the Spanish Ministry of Science and Innovation and FEDER under grant TIN2014-52211-C2-1-R.

Anonymous.

References

[1] P. E. Hart, N. J. Nilsson, B. Raphael, A formal basis for the heuristic determination of minimum cost paths, IEEE Transactions on Systems, Science, and Cybernetics SSC-4 (2) (1968) 100–107.

[2] M. Likhachev, G. J. Gordon, S. Thrun, Ara* : Anytime a* with provable bounds on sub-optimality, Advances in Neural Information Processing Systems 16 (2004) 767–774.

[3] S. Koenig, M. Likhachev, D*lite, Eighteenth National Conference on Artificial Intelligence (2002) 476–483.

[4] M. Likhachev, D. Ferguson , G. Gordon, A. T. Stentz, S. Thrun, Anytime dynamic a*: An anytime, replanning algorithm, Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS) (2005) 262–271.

[5] W. Lee, R. Lawrence, Trading space for time in grid-based path finding., AAAI.

[6] N. R. Sturtevant, Memory-efficient abstractions for pathfinding, AIIDE (2007) 31–36.

[7] R. Lawrence, V. Bulitko, Database-driven real-time heuristic search in video-game pathfinding., IEEE Trans. Comput. Intellig. and AI in Games 5 (3) (2013) 227–241.

[8] T. Huang, M. Kapadia, N. I. Badler, M. Kallmann, Path planning for coherent and persistent groups, Proceedings of the IEEE International Conference on Robotics and Automation (2014) 1652–1659.

[9] J. Abello, Hierarchical graph maps, Computers & Graphics 28 (3) (2004) 345 – 359.

[10] C. Tominski, J. Abello, H. Schumann, Cgv-an interactive graph visualization system, Computers & Graphics 33 (6) (2009) 660–678.

[11] E. D. Sacerdoti, Planning in a hierarchy of abstraction spaces, Artificial Intelligence 5 (2) (1974) 115 – 135.

[12] R. Holte, R. C. Holte, M. Perez, M. B. Perez, R. M. Zimmer, R. M. Zimmer, A. MacDonald, A. J. Macdonald, Hierarchical a*: Searching abstraction hierarchies efficiently, in: In Proceedings of the National Conference on Artificial Intelligence, 1996, pp. 530–535.

[13] R. C. Holte, C. Drummond, M. B. Perez, R. M. Zimmer, A. J. MacDonald, Searching with abstractions: A unifying framework and new high-performance algorithm, in: Proceedings of the biennial conference-Canadian society for computational studies of intelligence, 1994, pp. 263–270.

[14] R. C. Holte, T. Mkadmi, R. M. Zimmer, A. J. MacDonald, Speeding up problem solving by abstraction: A graph oriented approach, Artificial Intelligence 85 (1) (1996) 321–361.

[15] N. Sturtevant, R. Jansen, An analysis of map-based abstraction and refinement, in: I. Miguel, W. Ruml (Eds.), Abstraction, Reformulation, and Approximation, Vol. 4612 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2007, pp. 344–358.

[16] V. Bulitko, Y. Björnsson, R. Lawrence, Case-Based Subgoaling in Real-Time Heuristic Search for Video Game Pathfinding, Journal of Artificial Intelligence Research (JAIR) 39 (2010) 269–300.

[17] N. R. Sturtevant, R. Geisberger, A Comparison of High-Level Approaches for Speeding Up Pathfinding., in: G. M. Youngblood, V. Bulitko (Eds.), AIIDE, The AAAI Press, 2010.

[18] A. Botea, M. Müller, J. Schaeffer, Near optimal hierarchical path-finding, Journal of Game Development 1 (2004) 7–28.

[19] F. Garcia, M. Kapadia, N. I. Badler, Gpu-based dynamic search on adaptive resolution grids, Proceedings of the IEEE International Conference on Robtics and Automation (2014) 1631–1638.

[20] D. Harabor, A. Botea, Hierarchical path planning for multi-size agents in heterogeneous environments., CIG (2008) 258–265.

[21] Y. Li, L.-M. Su, W.-L. Li, Hierarchical path-finding based on decision tree., RSKT 7414 (2012) 248–256.

[22] C.-J. Jorgensen, F. Lamarche, From geometry to spatial reasoning : automatic structuring of 3D virtual environments, Proceedings of Motion In Games (2011) 353–364.

[23] S. Zlatanova, L. Liu, G. Sithole, A conceptual framework of space subdivision for indoor navigation, in: Proceedings of the Fifth ACM SIGSPATIAL International Workshop on Indoor Spatial Awareness, ISA '13, ACM, New York, NY, USA, 2013, pp. 37–41. doi:10.1145/2533810.2533819.

[24] K. Ninomiya, M. Kapadia, A. Shoulson, F. Garcia, N. Badler, Planning approaches to constraint-aware navigation in dynamic environments, Computer Animation and Virtual Worlds 26 (2) (2015) 119–139.

[25] M. Kapadia, F. Garcia, C. D. Boatright, N. I. Badler, Dynamic search on the gpu, Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (2013) 3332–3337.

[26] W. van Toll, N. Jaklin, R. Geraerts, Towards believable crowds: A generic multi-level framework for agent navigation, in: ASCI.OPEN, 2015.

[27] M. Kallmann, M. Kapadia, Geometric and Discrete Path Planning for Interactive Virtual Worlds, Vol. 8, Morgan & Claypool Publishers, 2016.

[28] N. Pelechano, J. M. Allbeck, N. I. Badler, Controlling individual agents in high-density crowd simulation, in: Proceedings of the 2007 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, SCA '07, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 2007, pp. 99–108.

[29] M. Kallmann, Dynamic and robust local clearance triangulations, ACM Trans. Graph. 33 (5) (2014) 161:1–161:17.

[30] D. J. Demyen, M. Buro, Efficient triangulation-based pathfinding, in: AAAI, Vol. 6, 2006, pp. 942–947.

[31] Z. Bhathena, A. Gheith, D. Fussell, Near optimal hierarchical pathfinding using triangulations (2014).

[32] R. Oliva, N. Pelechano, Neogen: Near optimal generator of navigation meshes for 3d multi-layered environments, Computers & Graphics 37 (5) (2013) 403–412.

[33] M. Mononen, Navigation-mesh toolset for games, GitHub Recast and Detour.
URL `https://github.com/memononen/recastnavigation`

[34] W. van Toll, A. Cook, R. Geraerts, Navigation meshes for realistic multi-layered environments, in: Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on, 2011, pp. 3526–3532.

[35] F. Gravot, T. Yokoyama, Y. Miyake, Precomputed pathfinding for large and detailed worlds on mmo servers. (2014) 269–286.

[36] G. Karypis, V. Kumar, Multilevel k-way partitioning scheme for irregular graphs, journal of Parallel and Distributed Computing 48 (1998) 96–129.

[37] B. W. Kernighan, S. Lin, An efficient heuristic procedure for partitioning graphs, Bell system technical journal 49 (2) (1970) 291–307.

[38] Karypis, Metis - serial graph partitioning and fill-reducing matrix orderings, METIS Software Package.
URL `http://glaros.dtc.umn.edu/gkhome/metis/metis/overview`

[39] E. Dijkstra, A note on two problems in connexion with graphs, Numerische Mathematik 1 (1959) 269–271.