A Comparison of High-Level Approaches for Speeding Up Pathfinding

Nathan R. Sturtevant[1]
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada
nathanst@cs.ualberta.ca

Robert Geisberger
Faculty of Informatics
Karlsruhe Institute of Technology
Karlsruhe, Germany
geisberger@kit.edu

Abstract

Most games being shipped today use some form of high-level abstraction such as a navmesh or waypoint graph for path planning. These structures can generally be represented in a form which is compact enough to meet the tight memory constraints in a game. But, when such a graph grows too large, finding paths can still be a complex task. This challenge was faced in Dragon Age: Origins and solved by adding an additional level of abstraction. In the last few years a variety of novel approaches have been developed for finding optimal paths through graphs with specific design applications for road networks. Currently these techniques cannot be feasibly applied to the lowest detail of movement possible in a game map, but can be applied to the high-level abstractions which are commonly found in games. In this paper we describe the pathfinding challenge faced before shipping the title Dragon Age: Origins and perform a postmortem analysis on the extended abstraction that was used in comparison to building more advanced heuristics or the use of contraction hierarchies. We show that contraction hierarchies and abstractions have similar overhead and performance and are both useful approaches for high-level planning in games.

Introduction and Background

The problem of finding paths in games is, on one level, well understood. In particular, while players in the game may traverse terrain with a fine-grained or even real-value representation, planning for movement occurs at a higher level of abstraction, generally represented by a graph. Such representations include navmeshes (Tozour 2002) or waypoint graphs (Lidén and Valve-Software 2002), but have also been described algorithmically, such as HPA* (Botea, Müller, and Schaeffer 2004). These representations may not necessarily solve underlying issues such as turning constraints (Pinter 2001; Sturtevant 2009), an issue we will not address here.

These representations work well, but there are two possible shortcomings. First, they can either become so large that pathfinding within the abstract representation may still take longer than per-frame constraints. Second, if the abstract representation is too coarse, it may result in poor pathfinding behavior, as low-level obstacles may not be adequately represented in the abstraction. These two issues are linked, as creating a finer grained abstraction will result in a larger abstract graph.

This issue was faced in the pathfinding system for the game Dragon Age: Origins (DAO) and was solved by adding a second level of abstraction on top of the existing abstraction previously described (Sturtevant 2007). But, no scientific analysis was performed to measure the effect of this approach, or to compare it with other possible approaches. The goal of this paper is to do both. In addition to analyzing the parameters that could have been used to tune the second level of abstraction, we look at alternate approaches to address this problem, a special abstraction called a contraction hierarchy, and better heuristics.

In the last few years a variety of novel approaches have been designed for finding optimal paths through graphs with specific applications to road networks; see (Delling et al. 2009) for an overview. Currently these techniques cannot be feasibly applied to the lowest detail of movement possible in a game map, but can be applied to the high-level abstractions which are commonly found in games. These approaches include *contraction hierarchies* and ALT (A*, landmarks and triangle inequality) (Goldberg and Harrelson 2005), also called differential heuristics (Sturtevant et al. 2009).

Our contributions are as follows. We publish for the first time the two-level approach used for pathfinding in Dragon Age: Origins and describe in detail why this approach was necessary. We adapt approaches from road networks to game constraints, and provide important comparisons for practically applying these approaches, showing the strength and weakness of each. We show that abstraction and contraction hierarchies can both be implemented efficiently and have similar performance, however the cases that are hard for each approach differ considerably. The heuristic approach we compare has reasonable performance, however the memory overhead and lack of incremental computation make it less desirable in practice.

Problem Definition

Assume an input graph G which represents, at some level of abstraction, the possible paths which can be followed in the

[1]Supported by Alberta iCore. Current address: Department of Computer Science, University of Denver, sturtevant@cs.du.edu

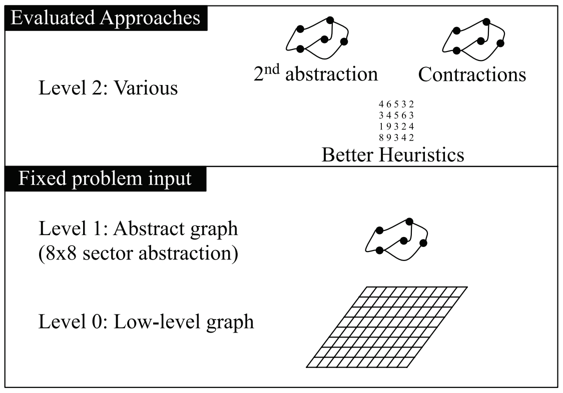


Figure 1: Overview of approaches.

world. G can be created from a navmesh, waypoint graph, HPA*, or other approaches. The task is to find paths between arbitrary nodes in G as quickly as possible while minimizing the memory overhead, minimizing work overhead, and maximizing the path quality. Memory and work overhead can be measured directly. The path quality can be measured by sub-optimality, however sub-optimality is not guaranteed to be a useful measure of the visual quality of a path.

There are a number of ways by which a path p in the abstract graph G can be turned into an followable path in the world. However, given identical paths, the process will be the same regardless of what mechanism was used to create the path. Therefore, we ignore the final refinement step as it will be the same across all approaches.

Approaches

This paper is built around the comparison between a number of existing technologies that have been developed within differing communities. We attempt to provide a unified understanding of these methods and how they can be applied to real-time video games. An overview of these approaches is given in Figure 1. We assume the existence of a low-level graph and an abstract graph which represents that. We experiment with approaches which can be placed on top of the level 1 graph to improve the performance of finding paths in the level 1 abstract graph. In the next three sections we detail the three approaches.

Abstraction Hierarchies

The Dragon Age: Origins (DAO) pathfinding abstraction was designed to use as little memory as possible so as to augment the low-level memory structures which used most of the memory allocated to pathfinding. In the original schema a map is divided with a 16×16 grid that overlays the maps creating a series of *sectors* which can be implicitly indexed using a low-level x/y coordinate. Each sector is then subdivided into *regions* where two low-level locations are in the same region if they are in the same sector and there is a path between them that does not leave the sector. An example is shown in Figure 2. There are four sectors which are divided into a graph with 9 regions and 10 edges. This

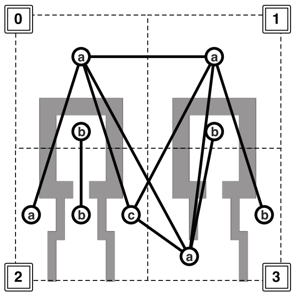


Figure 2: Abstract graph created with a 16×16 overlay for Dragon Age: Origins.

abstraction is a one-layer version of the *sector* abstraction described in (Sturtevant and Jansen 2007).

Each region is represented by a single point which can be located anywhere within the region, or moved around to improve the quality of paths. These points form nodes in an abstract graph, while edges are found by comparing which borders can be crossed into neighboring regions.

In order to find a path between an arbitrary start and goal location, the sector and region of the start and goal are first computed. The region center is either found through a small breadth-first search or implicitly when there is only one region in a sector. Given a sector and region, a search can take place in the abstract graph, and the resulting abstract path can be refined into a low-level path. In the game, these planning steps are interleaved with smoothing, resulting in responsiveness even with many agents and degradable path quality as less time is available.

The sector abstraction is memory efficient, generally using less than 100k of storage for a map. Full details and experimental results can be found in (Sturtevant 2007). Edge costs in the game are dynamically weighted both in the low-level grid and in the abstraction as a result of occupancy information and high-level events like area effects.

An Additional Level of Abstraction

Before DAO shipped, it was determined that the sector abstraction used for the original game needed to be enhanced to handle a number of situations. On the largest maps pathfinding requests could run out of time when planning in the abstraction while on smaller indoor maps the large abstraction was causing issues because it did not adequately represent the underlying terrain. If the abstraction were to be built at a higher granularity to fix these issues, the map would again become too large to solve hard problems.

As a result, another layer of abstraction was built on top of the existing abstraction. We demonstrate this in Figure 3. Sector 0 in this figure is the same as Figure 2. However, that entire figure is now contained into a single high-level sector. This extra layer was built using a sector size twice the size that in the original sector abstraction. In the final game the original abstraction was based on sectors of size 8×8 to 16×16, depending on the original size of the map. The high-level abstraction was then built based on sector sizes of 16×16 to 32×32. Extra space in the regular abstraction was used to map nodes in the regular abstraction into nodes

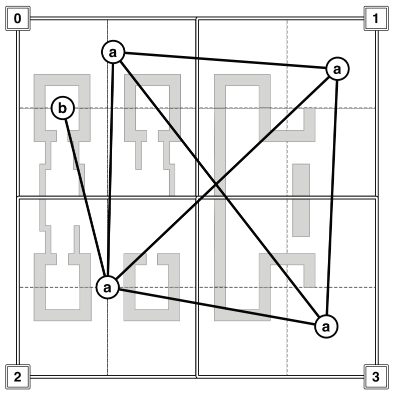


Figure 3: Representing an additional level of abstraction.

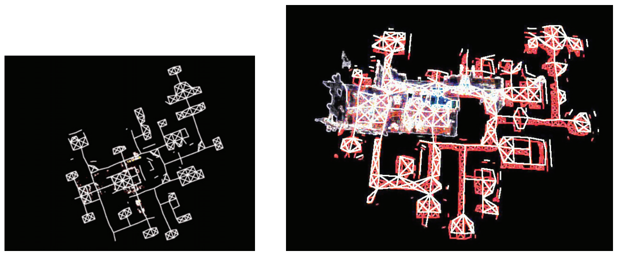


Figure 4: Comparing map representation with 1 and 2 levels of abstraction.

in the high-level abstraction. Abstractions for sample maps before and after the second level was added are shown in Figure 4.

With the new abstraction, path planning occurs as follows. Given a start and goal location, the corresponding nodes in the high-level abstraction are located, and a high-level path is found. The high level path is used for two purposes. First, planning in the regular abstraction is restricted to take place in the sectors that the abstract path passes through. This greatly restricts the number of nodes expanded when planning in the regular abstraction. Furthermore, if the high-level path is long enough it is broken into pieces, and the refinement for each piece occurs independently. In this way much of the planning does not have to be immediately computed, but can be delayed until needed.

Enhanced Heuristics

The standard approach in the heuristic search community to improved heuristics has been pattern databases (PDB) (Culberson and Schaeffer 1998). PDBs have very successfully reduced the search overhead in combinatorial puzzles, but are inadequate for domains which grow polynomially. Effective heuristics for these domains are based on exact distances in the search space, as opposed to the abstract distances used for combinatorial puzzles. There is a broad class of True-Distance Heuristics (Sturtevant et al. 2009) that work in a variety of domains.

The algorithm engineering community developed a number of true-distance heuristics for application to road networks. The simplest of these was ALT (Goldberg and Harrelson 2005), also called a differential heuristic. A precomputed single-source shortest path (SSSP) solution from any single point in a map, called a *landmark*, can be used to provide a heuristic for any two other points in the map using the triangle inequality. Specifically, given a distance function $d(a, C)$ where C is a state where the SSSP information is available, $h(a, b) = |d(a, C) - d(b, C)|$. Using the max of multiple landmarks increases both accuracy and memory usage; each landmark has memory cost $O(N)$ where N is the number of nodes in the graph.

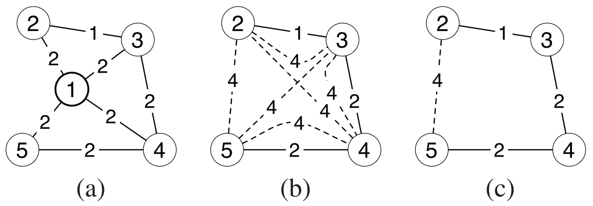


Figure 5: Contracting a node in a graph. Dashed edges represent shortcuts.

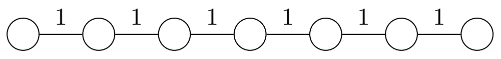
Contraction Hierarchies

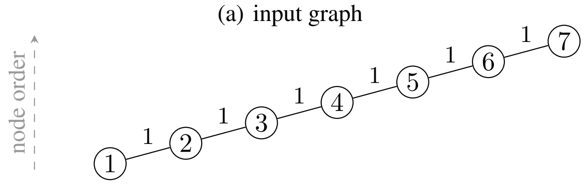
Contraction hierarchies (CH) (Geisberger et al. 2008) take a different approach to abstraction. Instead of abstracting groups of nodes together in a single step, single nodes are abstracted one at a time through a processes called *contraction*. Each node in the graph is assigned an importance level, i.e. $I(v) = 1..N$ and contracted in order from lowest importance to highest. Contracting a node v means removing v from the graph without changing shortest path distances between the remaining (more important) nodes. A trivial way to contract a node v is to introduce a shortcut edge (u, w) with cost $c(u, v) + c(v, w)$ for every path $u \rightarrow v \rightarrow w$ with $I(v) < I(u), I(w)$. But in order to keep the graph sparse, we can try to avoid a shortcut (u, w) by finding a *witness* – another path p from u to w fulfilling $c(p) \leq c(u, v) + c(v, w)$. Such a witness proves that the shortcut is never needed.

This is demonstrated in Figure 5. In part (a) of this figure we show the original graph, where the intent is to contract node 1. The naive contraction is in part (b), where a dashed edge is added for each possible path through node 1 in the original graph. In part (c) we show the sparse contraction where only a single edge is added. It can be easily shown that there are witnesses for all other dotted edges in part (b).

To compute a shortest path between a source s and target t, we perform a bidirectional Dijkstra search in the CH. The special restriction on this CH search is that it only goes *upward*, i.e. we only expand a node if that node is more important than the current node being expanded. Both search scopes eventually meet at the most important node of a shortest path. The search is halted when the minimum distance in the priority queue is not smaller than the tentative shortest path length. The search can also benefit from using A* and heuristics which we cannot describe in detail here.

In practice this takes place on a fully contracted graph,


(a) input graph


(b) CH: no shortcuts needed

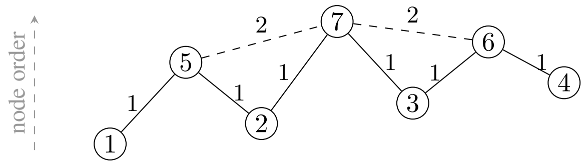

(c) CH: small search space

Figure 6: Two possible CHs from the input graph. The nodes are labeled with their importance and accordingly vertically aligned.

but for simplicity we demonstrate this process in the contracted graph in Figure 5(c), where we find the shortest path between nodes 3 and 5. Node 5 has no neighbors that are more important, so no neighbors will be expanded in the bidirectional search. Node 3 has two neighbors, but only node 4 is more important, so only that node will be expanded. Node 4's only unexplored neighbor is node 5, so the shortest path from node 3 to node 5 passes through node 4.

The shortest path from node 2 to node 5 uses the shortcut edge. Along with this edge is information which allows it to be unpacked into a full path between nodes 2, 1, and 5. This means that it only takes a single expansion to find the length of the shortest path (4), but generating the actual path takes longer, as shortcut edges must be unpacked.

Node ordering

The node ordering is important for the efficiency of the CH and allows different tradeoffs between time and space. Shortcuts increase the required space but can reduce the search space. In Figure 6, we show an example of the contraction of a line graph. Contracting the nodes in the order of the line, Figure 6(b), requires no shortcuts, but does not reduce the search space. Consider the query between the leftmost and rightmost node. The CH query will visit nodes $\{1, \dots, 7\}$ in the forward search and node 7 in the backward search. So all nodes of the graph are visited. Figure 6(c), on the other hand, requires 2 shortcuts. But the same query will only visit nodes $\{1, 5, 7\}$ in the forward search and $\{4, 6, 7\}$ in the backward search. For general line graphs, iteratively contracting every other node requires $O(n)$ shortcuts, resulting in a search space of size $O(\log n)$.

A heuristic ordering is used to compute the importance of each node, because the computation of an optimal ordering (i.e. shortcut minimal or search space minimal) is NP-hard. We use a priority queue whose minimum element contains the best node to contract next, and then successively contract nodes according to the priority of the remaining nodes.

Before defining the heuristic terms, we introduce the edge property $r(u, w)$ as the number of *original edges* contracted below an edge in the graph. It is initialized with 1 for each edge. If a new shortcut (u, w) is added from the edges (u, v) and (v, w), we set $r(u, w) := r(u, v) + r(v, w)$. Additionally, let $\phi(v)$, be the set of shortcuts that would be added if node v would be contracted next.

The priority is the linear combination of four terms:

- The first importance term for contraction is $\delta(v)$ or the *edge difference*. This is the net difference in edges added to the graph by contracting v next. Formally, let $\delta(v) := |\phi(v)| - |\{(u, v) \mid v \text{ uncontracted}\}| - |\{(v, w) \mid v \text{ uncontracted}\}|$. This term works to keep the contracted graph sparse and to improve the distribution of node contractions.
- The next term is $\sigma(v)$, which measures the number of original edges contracted into new shortcut edges introduced when contracting v. We define $\sigma(v) := \sum_{(u,w)\in\phi(v)} r(u, w)$. This term works to keep the contracted graph sparse.
- The third term is $\gamma(v)$, an upper bound on the length of an *upward path*. It is initialized with 0 for each node. After the contraction of node v, we update $\gamma(u) := \max(\gamma(u), \gamma(v) + 1)$ for each uncontracted neighbor u of v. This term is used to improve query performance, but does not have a large impact.
- The final term is $\lambda(v)$, the number of already *contracted neighbors* u of v. This term improves query performance by improving the distribution of node contractions.

We assign each node v a priority $\rho(v)$ based on how attractive it is to contract it next. Initially, this term is computed for every node in the graph and the nodes are placed into a priority queue. For the sake of efficiency, after each new node is contracted, these values are re-computed only for the neighbors of the contracted node, but not for the entire graph. As a result of this lazy updating, the heuristics are then recomputed for a node before contracting it, and if the value increases then the node is re-inserted into the priority queue instead of being contracted.

Given a weighted coefficient vector ζ, we define the priority $\rho(v) := \zeta \cdot (\delta(v), \sigma(v), \gamma(v), \lambda(v))^T$. The choice of ζ is an important tuning parameter which we look at during the experimental results. For our fast version (Large Mem) we use $\zeta = (25, 60, 20, 2)$ and for the space-efficient version (Low Mem) we use $\zeta = (0, 1, 0, 0)$. This version only contracts based on the $\sigma(v)$ term.

Reducing Contraction Hierarchy Overhead

There are a number of observations which reduce the memory required for contraction hierarchies. The first important observation is that we need to store an edge only with its less important endpoint, as the search only progresses upwards in importance. Because shortcuts can span long paths, we need to store them explicitly and cannot use the data structure used for DAO. Instead, we use a block representation

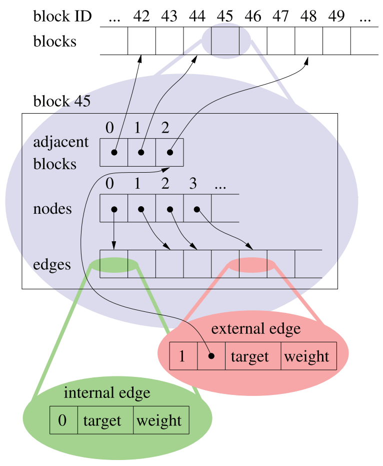


Figure 7: Graph data structure. Each edges stores a flag indicating whether it is an external edge leading to a node in a different block.

developed for mobile systems (Sanders, Schultes, and Vetter 2008) to compress the CH. Each block contains a subset of nodes and all incident edges. The main idea is to distinguish between 'internal' edges within a block and 'external' ones; this is illustrated in Figure 7. For an internal edge, we only need to store the offset to the target node within the block. This saves space as we can arrange nodes so that most edges are internal. Space is further reduced by using the minimum necessary bit encoding length per block. This compresses data by 70% but slows down the query by about 50%.

Experimental Results

We experimented with implementations of the three main approaches and compare the results here. Experiments were performed on a dual core 2GHz Opteron with 8GB RAM and 1MB cache per core, however only one core was used for each of the experiments, and only a small fraction of the available RAM was used. This hardware is faster than a low-end target system for most games.

As the code used to compare these techniques comes from several different code bases, each with different levels of optimization, we will measure both time and work, with work measured in nodes expanded (also called nodes settled in the algorithmics community). The costs measured here are cheaper than in practice, because in practice we re-weight edges based on underlying properties of the map.

The maps we used are taken from Baldur's Gate and are scaled to 512×512. The maps in DAO can be 4 to 16 times larger. We partially account for this by generating graphs only based on 8×8 underlying sectors, while the underlying sectors in DAO are up to 16×16. As a result, memory and speed estimates may be slightly underestimated across the experiments reported here.

We experimented on 120 maps. The average map has 1335 nodes, with total sizes ranging between 316 and 3787 nodes. On each map we construct a set of 10,000 problems chosen at random. It is possible that no path will exist between the points, in which case the cost of discovering this is included in the results. The average length of a path is 228 with edge weights between 2 and 16.

Table 1: Memory Usage

	Avg	Min	Max
Baseline (uncompressed)	28k	16k	52k
Differential	55k	22k	126k
CH Full Large Mem	111k	24k	462k
CH Full Low Mem	95k	23k	287k
CH Mobile Large Mem	29k	8k	116k
CH Mobile Low Mem	25k	7k	64k
Sector Abs (16×16)	35k	20k	66k
Sector Abs (24×24)	31k	18k	57k
Sector Abs (32×32)	30k	17k	55k

Memory Usage

We begin by measuring the total memory usage of each approach in Table 1. The first line is the memory required to store the (8×8 sector abstraction) graph. Without compression this takes 28k on average. Compressing similar sectors can reduce this to around 20k (Sturtevant 2007). Adding 10 differential heuristics approximately doubles the total memory usage to 55k on average. A full-memory contraction hierarchy takes around 100k of memory on average, depending on the heuristic ordering scheme used. The mobile implementation uses approximately the same amount of memory as the uncompressed baseline graph on average, but in the worst case, uses more. Adding an additional level of the sector abstraction only increases the memory usage slightly.

Planning Costs

We next evaluate the cost of planning with each approach. For ease of comparison, we have put all results in Table 2, but we will cover each result in detail.

Results planning on the initial graph using A* can be found at the top of Table 2. The average problem requires 128 nodes and takes 228μs to solve. This result includes unsolvable problems from the test set. Removing these does not significantly affect the average work. We then report the average of the 10% longest problems on each map, which requires significantly more node expansions (348.6) and time (595μs). The longest 1% of all problems require almost one thousand node expansions and 1.4ms. Finally, the hardest 1% (measured by nodes expanded) requires 2.4ms on average and almost 1400 nodes.

The next result in Table 2 is for A* and differential heuristics. These results are with 10 landmarks (each storing the single-source shortest path to a single point). With the improved heuristics A* expands three times fewer nodes, but is only twice as fast on average. On the longest 10% from each map it expands 4 times fewer nodes and is almost 3 times faster. On the longest 1% of all problems it expands 6 times fewer nodes and is just over 3 times faster. But, on the hardest problems, almost 850 nodes are expanded on average and the time increases to 1.7ms.

The improved heuristics provide a significant reduction

Table 2: Results for path planning.

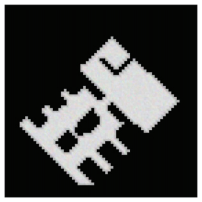
	All		10%		long 1%		hard 1%	
	nodes	[μs]	nodes	[μs]	nodes	[μs]	nodes	[μs]
A*	128.0	228.0	348.6	595.0	937.6	1400	1395.5	2400
A* + 10 heuristics	40.5	110.0	77.2	209.0	146.2	390.0	848.9	1700
Sector Abs (16 $\times$ 16)	44.3	75.8	115.4	183.0	318.4	404.1	452.5	609.8
Sector Abs (24 $\times$ 24)	25.3	43.2	63.7	98.0	183.6	214.5	250.5	302.3
Sector Abs (32 $\times$ 32)	17.2	30.5	42.0	64.3	121.4	137.5	165.5	187.8
CH Large Mem	32.3	36.4	70.3	68.6	61.6	49.4	152.9	144.5
CH Low Mem	59.8	59.3	137.8	150.4	130.3	101.7	374.9	334.3

in nodes expanded with a lower reduction in speed except on the hardest problems, where performance is still poor. The worst-case results would improve with the use of more heuristics, however this would increase the memory and node expansion overhead as well. This approach returns optimal paths through the abstract graph, but the paths are not usable until the entire path is completed, a disadvantage when compared to the other approaches.

The next three lines of Table 2 evaluate the performance of the second level of abstraction as the sector size changes. For both the sector abstraction and CH, we evaluate the cost of finding the initial path here, and then measure the refinement cost separately. The A* performance is based on a 8×8 abstraction, so using a high-level 16×16 abstraction would, in the best case, provide a 4 times reduction in nodes expanded, while we see a 3 times reduction across each set of problems. The 24×24 abstraction results in approximately a 5 times reduction in node expansions, and the 32×32 abstraction results in approximately an 8 times reduction in node expansions, even on the hardest problems. Thus, the abstraction can effectively reduce node expansions. Additional work is required to incrementally refine this result into a followable path.

Finally, we look at contraction hierarchies at the bottom of Table 2. We compare only the mobile implementation, as the other implementation uses too much memory. We compare one contraction ordering which takes more memory and results in more efficient planning to another ordering that uses less memory but has less efficient planning. As with abstraction, the contraction hierarchies return paths that must be refined in order to use them. We report just the time to find the path and not to refine it, as this can be done incrementally.

The low-memory approach is about 2 times slower than the higher memory approach. The performance of the CH is worse than a 24×24 or 32×32 abstraction on average, and for the 10% hardest problems. But, a peculiar things happens when we look at the longest 1% of all problems. Contraction hierarchies here are actually faster than on the 10% longest from each map. This shows the strength of contraction hierarchies and why they have been so successful in road networks. Longer paths are likely to have more shortcuts resulting in very short paths. For these paths the refinement process will be more expensive, but optimal distances can be computed very quickly. If we instead look at the hardest 1% of problems, we see that there are many problems that are hard for contraction hierarchies, with an average solution time of 144μs, about 23% faster than the 32×32 sector abstraction.


(a)

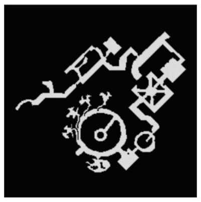

(b)

Figure 8: Map (a) is easy for abstraction but hard for CHs. Map (b) is hard for abstraction but easier for CHs.

We illustrate these differences in Figure 8. We ranked maps by difficulty for abstraction and CHs and looked for the maps with the largest difference in ranking between the two approaches. Map (a) has a difference in ranking of 36 (out of 120) and is easier for abstraction as all paths are short. Map (b) has a difference in ranking of 65 and is hard for abstraction because paths are long, where CHs can reduce the abstract planning cost with many shortcut edges.

Refining Abstract Paths

We report the total cost of refinement for the sector abstraction in node expansions and total suboptimality in Table 3. This refinement is done via an A* search which is constrained to stay within the same sectors as the abstract path. As the cost of refinement is length-dependent, we report the average number of node expansions required for each high-level path node that is refined. The best and worst columns give a general range of values that the refinement cost might fall in between, although these are not absolute worst and best cases for all maps. The average cost of refinement is smaller for smaller sector sizes. We measured several refinement lengths, as we have a choice of how many edges to refine at each step. Shorter refinement lengths decrease the average work but increase suboptimality. In general, this data serves to show that refining the abstract paths is much easier than finding paths in the first place.

We report the cost of path refinement for CHs in Table 4. Refinement in CHs is just a matter of unpacking edges. We report the number of edges which must be refined to generate a complete path, although fewer edges must be refined in practice to actually start moving. None of these numbers is particularly large, meaning that refinement in CHs will be cheaper than in sector abstractions. It is interesting to note

Table 3: Node expansions per each refined node.

Sector size	Refine	Best	Actual	Worst	Subopt
16×16	5	2	2.3	4	1.05
	∞	2	2.6	4	1.05
24×24	5	3	3.7	9	1.09
	∞	3	4.5	9	1.08
32×32	2	4	4.5	16	1.14
	5	4	5.4	16	1.11
	∞	4	6.5	16	1.10

that on the hardest 1% of problems refinement is easy, while on the longest 1% of problems it is hard. This is the opposite of the CHs planning results, and makes sense, as on the long problems there will be more shortcuts to refine.

Discussion and Conclusions

We summarize our results in Table 5. The results show that state abstraction and contraction hierarchies are competitive approaches to finding paths in game maps when the abstract representation grows too large to quickly support the necessary pathfinding operations. Improved heuristics have worse performance than both of these approaches.

Returning to one of our original motivations, we can now ask whether we would make different decisions in the design of the high-level abstraction for Dragon Age: Origins. DAO used a second level of abstraction with a sector size twice the underlying sector size. For the results here, this corresponds to 16×16 sectors. The results of this paper suggest that a larger high-level abstraction, either 24×24 or 32×32, would have improved performance.

The comparison between abstraction and contraction hierarchies is difficult. Abstraction is attractive as it is easy to use and understand, and the high-level planning tasks are recursively similar to the low-level planning tasks. Abstractions can also be computed quickly and can be dynamically changed with the map. The downside of abstraction is that it returns suboptimal paths.

Contraction hierarchies offer the potential of higher performance for long tasks and are guaranteed to return optimal results. Contraction hierarchies can also answer distance queries about a map quickly, something that is often needed for other AI tasks in a game. They also represent a different performance profile with longer paths being easier to compute that shorter paths. Contraction hierarchies rely on a heuristic which ranks the order of contraction, but simple heuristics have suitable performance. Finally, although we have not discussed it in detail here, they are more difficult to adapt when edge costs change drastically, although small changes do not present a large challenge (Schultes and Sanders 2007).

Thus, it is difficult to say that one approach "beats" the other. Instead, they each have their own strengths and weaknesses. Any particular individual would need to balance these to make the final decision of what approach to use. Regardless, these are all techniques that should be understood by experts in the industry, as they are an integral part of the toolbox needed for building a high-quality motion planner, which is an important piece of any high-quality AI system.

Table 4: Average edges unpacked to refine a CH path.

	All	10%	long 1%	hard 1%
Large Mem (nodes)	13.6	29.7	73.5	9.2
Low Mem (nodes)	6.5	14.1	49.8	7.5

Table 5: Advantages and disadvantages of approaches.

Approach	Memory	Optimal	Dynamic	CPU
Baseline	small	yes	yes	high
Heuristics	larger	yes	no	medium
Abstraction	small	no	yes	low
Contraction	small	yes	some	low

References

Botea, A.; Müller, M.; and Schaeffer, J. 2004. Near optimal hierarchical path-finding. *J. of Game Develop.* 1(1):7–28.

Culberson, J. C., and Schaeffer, J. 1998. Pattern databases. *Computational Intelligence* 14(3):318–334.

Delling, D.; Sanders, P.; Schultes, D.; and Wagner, D. 2009. Engineering Route Planning Algorithms. In Lerner, J.; Wagner, D.; and Zweig, K. A., eds., *Algorithmics of Large and Complex Networks*, volume 5515 of *Lecture Notes in Computer Science*. Springer. 117–139.

Geisberger, R.; Sanders, P.; Schultes, D.; and Delling, D. 2008. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In McGeoch, C. C., ed., *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, volume 5038 of *Lecture Notes in Computer Science*, 319–333. Springer.

Goldberg, A. V., and Harrelson, C. 2005. Computing the Shortest Path: A* Search Meets Graph Theory. In *Proceedings of the 16th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'05)*, 156–165.

Lidén, L., and Valve-Software. 2002. Strategic and tactical reasoning with waypoints. *AI Game Programming Wisdom, Charles River Media* 211–220.

Pinter, M. 2001. Toward more realistic pathfinding. In *gamasutra.com.*

Sanders, P.; Schultes, D.; and Vetter, C. 2008. Mobile Route Planning. In *Proceedings of the 16th Annual European Symposium on Algorithms (ESA'08)*, volume 5193 of *Lecture Notes in Computer Science*, 732–743. Springer.

Schultes, D., and Sanders, P. 2007. Dynamic Highway-Node Routing. In Demetrescu, C., ed., *Proceedings of the 6th Workshop on Experimental Algorithms (WEA'07)*, volume 4525 of *Lecture Notes in Computer Science*, 66–79. Springer.

Sturtevant, N. R., and Jansen, M. R. 2007. An analysis of map-based abstraction and refinement. In *SARA*, 344–358.

Sturtevant, N. R.; Felner, A.; Barrer, M.; Schaeffer, J.; and Burch, N. 2009. Memory-based heuristics for explicit state spaces. In *IJCAI*, 609–614.

Sturtevant, N. R. 2007. Memory-efficient abstractions for pathfinding. In *AIIDE*, 31–36.

Sturtevant, N. R. 2009. Optimizing motion-constrained pathfinding. In *AIIDE*.

Tozour, P. 2002. Building a near-optimal navigation mesh. In *AI Game Programming Wisdom. (S. Rabin, ed.)*, 171–185.