Benchmarks for Pathfinding Search: Iron Harvest

Daniel D. Harabor,[1] **Ryan Hechenberger,**[1] **Thomas Jahn**[2]

[1]Faculty of Information Technology, Monash University, Australia
[2]KING Art Games, Germany
{daniel.harabor, ryan.hechenberger}@monash.edu
thomas.jahn@kingart-games.com

Abstract

Pathfinding is a central topic in AI for games, with many approaches having been suggested. But comparing different algorithms is tricky, because design choices stem from different practical considerations; e.g., some pathfinding systems are grid-based, others rely on a navigation mesh or visibility graph and so on. Current benchmarks mirror this trend, focusing on one set of assumptions while ignoring the rest. In this work we present a new unified benchmark using data from the game Iron Harvest. For 35 different levels in the game we generate several complementary map representations (grid, mesh and obstacle-set) and we provide a common set of challenging instances. We describe and analyse the new benchmark and then compare several leading pathfinding algorithms that begin from different assumption sets. Our goal is to allow researchers and practitioners to better understand the relative strengths and weakness of competing techniques.

Introduction

Pathfinding search is a fundamental operation for game AI. Most often pathfinding facilitates *movement*, by providing agents with collision-free trajectories through the virtual environment (Stout 1996). However pathfinding also plays other roles, for example in *tactical decision-making*, where agents must decide which resource points to mine or which cover points to seek (Johnson 2019). Similarly in *strategic decision-making*, such as Goal-oriented Action Planning (Orkin 2006), where pathfinding data usually provides a necessary input for higher-level tasks.

Because of its central role, a large variety of pathfinding approaches have been suggested in the research literature. Yet deciding between competing methods can be tricky, because algorithmic design choices often reflect developer tooling and design constraints. For example, some practitioners prefer grids (Sturtevant 2007; Kring, Champandard, and Samarin 2010); others use navigation meshes (Demyen and Buro 2006; Brewer 2019); and still others recommend different kinds of graphs, such as waypoint (Sturtevant 2019) or visibility (Young 2001; Oh and Leong 2017). Experimental evaluations meanwhile can be application-specific, maps may not be publicly available and comparisons can be limited to other similarly compatible solvers.

Figure 1: In-game screenshot of Iron Harvest. From the KING Art Games Press Kit.

For 2D pathfinding there exists a popular set of grid-based benchmarks (Sturtevant 2012) that feature data drawn from real games. These benchmarks help to address some of the mentioned issues. However, recent progress has reduced solve times on many of these grid maps to single-digit micro-seconds per query and sometimes less (Uras and Koenig 2018; Harabor et al. 2019; Hu et al. 2021). Moreover, being grid-based, these maps are of limited utility for practitioners that are not working with grid-based setups. To the best of our knowledge no benchmarks currently exist for mesh-based pathfinding; no current graph-based benchmarks use data from real games and no single benchmark exists which allows developers to measure performance, and analyse tradeoffs, across all three map representation types at the same time. We introduce a new, unified and more challenging pathfinding benchmark to fill this gap.

Using data from Iron Harvest, a recent real-time strategy game, we provide 35 distinct maps and 70,000 associated problem instances. Each map has three different representations: navigation mesh, obstacle map and grid. There are 2,000 co-feasible instances per map which allow comparisons across the different representations. We describe the benchmark, our instance generation approach and provide experimental results for some currently leading grid-based and mesh-based path planners. The data is publicly available for download from https://bitbucket.org/shortestpathlab/benchmarks

(a) Mesh (b) Obstacle (c) Grid (d) Rasterisation effects

Figure 2: We show examples (2a-2c) of different map representations for the level `scene_sp_sax_07`. We also report (2d) the effect that grid resolution has on path length (grid-optimal vs mesh-optimal) for a set of 100 co-solvable instances.

Iron Harvest

Developed by KING Art Games, and set in the alternate history of 1920+, Iron Harvest gives players control over large armies of cooperative agents. The armies compete in a variety of environments, each characterised by different types of static obstacles that the agents must avoid while moving. We give a brief description of the Iron Harvest pathfinding system and then we describe a set of 35 maps from the game which are the the basis for the new benchmark.

During development of the game, human designers create *levels* by placing buildings and other polygonal obstacles on a 2.5D map[1]. When the player enters a new level a large amount of data is loaded from persistent storage into memory. At this time the game generates *navigation meshes* from the level's obstacle set. A mesh is simply a collection of non-overlapping convex polygons, each entirely traversable or non-traversable (Snook 2000; Tozour 2002). Together, these polygons completely describe the navigable and non-navigable areas in a level. There exist many ways to generate a navigation mesh (Mononen 2009; van Toll et al. 2020) but the Iron Harvest system always produces a Constrained Delaunay Triangulation. This approach is well known in the pathfinding literature (Kallmann, Bieri, and Thalmann 2004; Demyen and Buro 2006; Cui, Harabor, and Grastien 2017) and has small overhead costs, typically extending a level's load time by only a few seconds.

Given a pair of traversable points, resp. the *start* and *target* location of an agent, the Polyanya algorithm (Cui, Harabor, and Grastien 2017) searches the mesh to identify a Euclidean-optimal start-target path. Agents follow their assigned paths, as part of a subsequent execution procedure. The complete navigation system has a variety of other components that can be considered pathfinding-adjacent (e.g., for updating the mesh when obstacles are added or removed, for handling agent geometry and for collision avoidance).

[1]The level assets are rendered in 3D but their positions in the world, and agent movement, are all mapped onto a 2D plane.

Map Descriptions

For each of the 35 levels from Iron Harvest we generate three distinct representations: navigation meshes, obstacle maps, and grid maps. Examples of each are shown in Figure 2 and in Table 1 we report a variety of key statistics. Note that in the table, as in the game, the maps are divided into different subsets: one for each single-player campaign and one (`mp`) for the set of competitive multiplayer maps. The file description for each format is included in the repository.

Navigation Meshes

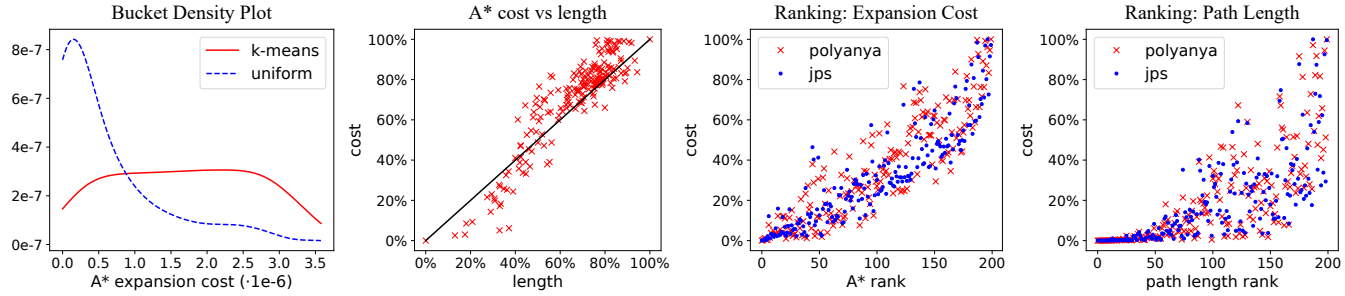
A navigation mesh is a data structure that describes the traversable and non-traversable areas of the map. Constructed online using KING Art's internal tooling, each mesh takes the form of a Constrained Delaunay Triangulation. We give a full decomposition, including non-traversable faces located inside obstacles on the mesh. To store navigation mesh data we define a `mesh` file format which is a small extension of the well known face-vertex standard (our extensions add support for describing non-traversable faces).

Obstacle Maps

Generated from mesh data, an obstacle map describes a level's terrain and play area. An *enclosure* is a polygon that specifies the bounds of a navigable region (i.e., valid paths never leave their enclosure). An *obstacle* is a polygon that cannot be crossed by any valid path. Multiple enclosures can exist per map, each disconnected from others. To store obstacle maps we use a simple text format where each line specifies the vertices of an obstacle polygon, in CW order. To distinguish enclosure polygons, we use a CCW order.

Grid Maps

Generated from mesh data, a grid map describes a level's terrain in terms of traversable and non-traversable cells. We rasterise the (outermost) enclosure of each mesh using a square grid and mark as non-traversable any cell that overlaps with any part of an obstacle (other cells are marked traversable). For a mesh with dimensions $H \times W$, we generate $kH \times kW$ grid cells, where k is the specified resolution.



(a) k-means vs uniform selection (b) path-length vs A* cost (c) ranking cost (d) ranking length

Figure 3: Clustering efficacy for the map `scene_sp_sax_07`. We look at the distribution of selected instances in Figure 3a and the rank of each instance according to A* expansion cost and path length metrics, in Figure 3b. In Figures 3c and 3d, we analyse how well the A* expansion cost and path length ranking compare to the expansion cost rank of different planning algorithms (here, for improved clarity, we only plot every 10th instance).

Looking at Figure 2d, we see that resolutions below 8x produce large deviations, when comparing the length of a Euclidean-optimal mesh path vs. a grid-optimal path. This is due to topology-altering rasterisation effects; e.g., narrow corridors that are traversable on the mesh become blocked on the grid. To mitigate rasterisation effects we select only 8x resolution grid maps. To store the grid data we follow the standard `map` format described in (Sturtevant 2012).

Generating Benchmarks

When evaluating path planning systems, practitioners generate and solve *benchmark problem instances*; so as to understand runtime performance for a range of realistic settings.

Perhaps the simplest approach to generating instances is random sampling. Here the practitioner selects k start-target problems and measures the performance of the planner by solving these instances. With a sample size of 10,000, as suggested in (Sturtevant and Geisberger 2010; Antsfeld et al. 2012), our standard error estimate can be reduced to $\pm 2\,\%$ of the sample mean. The main drawback of the sampling approach is that good performance on average is not sufficient to guarantee real-time performance. In practice, game developers need to understand how the path planner behaves across the entire range of possible queries. Some developers suggest to overcome this problem by recording and testing *all* instances that appear during play (Gillberg 2019). This benchmark is representative but requires extensive telemetry and the resulting set of problems can be prohibitively large.

Our approach to generating benchmark instances strikes a middle ground: we combine large random sampling with a clustering algorithm that produces a much smaller but representative test set. For each map we begin with 50,000 feasible problem instances, selected at random, which gives a standard error estimate $\pm 0.9\,\%$. We then solve each instance using A* (octile heuristic) search. We use the grid version of each map as this guarantees that instances are co-feasible for the obstacle map and navigation mesh representations. Next, we group instances into *buckets*, based on the number of nodes expanded and from those buckes we select a representative test set. Our approach produces 2,000 problem instances per map, which is sufficient to evaluate planner performance across the entire range of the sample set.

Creating Buckets

Let c_i denote the *minimum expansion cost* of an instance i, as measured by grid A* (octile heuristic) while computing an optimal solution for i in both the forward and backward search (start to target and target to start), then choosing the minimum between the two to reduce search direction bias. We sort the instances by c_i and place them into a single bucket B_0. We define $L(B)$ as the minimum cost for any instance in bucket B, and $H(B)$ as the maximum. We now recursively split the buckets as follows:

If $|B_i| < 20$: end recursion.

Else if $H(B_i) = L(B_i)$: we split B_i evenly into new buckets of size 10, extra elements go into the first (new) bucket.

Else: we split B_i into n buckets spanning range s, where $n = \lfloor |B_i| \div 10 \rfloor$ and $s = (H(B_i) - L(B_i)) \div n$. Each new bucket P_j receives items of cost $\big[L(B_i) + js, L(B_i) + (j+1)s\big)$. We drop empty buckets.

After splitting, we loop over buckets in order. If the current bucket $|B_i| < 10$ we merge it with the next bucket (if one exists). Afterward, we repeat this process, merging the last bucket with the previous bucket until the size of the (new) last bucket is at least 10. Finally, for every bucket where $|B_i| \geq 20$, we evenly split it. This whole process will result in a set of ordered buckets of sizes 10-19.

Choosing Instances

To generate our representative instances we first choose the bucket with lowest and highest cost. We then perform a k-means clustering on the remaining buckets ($k = 198$) in order, where each cluster is placed on a 1D line, spanning the locations from $L(B_L)$ (B_L lowest cost bucket in cluster) to $H(B_H)$ (B_H highest bucket cost). We then find the clustering that will give the maximum value of the minimum spanning length. This was done with dynamic programming. We then select the maximum cost bucket in each cluster, bringing us a total of 200 buckets. From each bucket we select the 10 instances with the highest expansion cost and discard all the rest (i.e., 2,000 instances per map).

maps	#	#obst	#face	#trav	trav-D	#vert	vert-D	cells	fit	inc μ	inc σ	inc max
cha	4	371	11 806	5722	2.12	5906	6.00	7 337 056	98.95 %	1.08	0.048	2.51
pol	6	371	15 150	6522	2.09	7578	6.00	6 545 639	98.68 %	1.06	0.038	1.78
rus	7	353	16 378	6946	2.08	8192	6.00	5 530 591	98.25 %	1.14	0.35	13.3
sax	7	282	13 934	5869	2.08	6970	6.00	7 072 777	98.70 %	1.06	0.033	3.07
mp	10	285	12 571	5388	2.08	6288	6.00	5 721 490	98.91 %	1.06	0.027	1.68
end	1	639	23 826	11 332	2.09	11 916	6.00	24 548 064	99.27 %	1.06	0.026	1.56
all	35	332	14 281	6198	2.09	7144	6.00	6 817 389	98.76 %	1.08	0.16	13.3

Table 1: `maps`: the map group; `#obst`: avg no. of obstacles; `#face`: avg no. of mesh faces; `#trav`: avg no. of trav. faces; `trav-D`: avg deg. for trav. faces; `#vert`: avg no. of mesh vertices; `vert-D`: avg deg of mesh vertices; `cells`: avg no. of trav. grid cells; `fit`: % trav. grid area vs. trav. mesh area; `inc`: path length increase (grid ÷ mesh) reported as mean μ, stdev σ and `max`

The k-means clustering was done to provide a more balanced representation of instances based on cost. In Figure 3a we compare our method to a uniform selection strategy, where the sample set is divided into 200 groups of roughly uniform size ($\pm$ 1 instance) and then selecting from each bucket the 10 highest-cost instances. Notice that the uniform distribution strongly favours easy instances as they are much more numerous. Our approach meanwhile selects instances from across the difficulty spectrum. This allows practitioners to better understand performance in a wider range of practical settings. We provide our implementations as part of the benchmark, for a further contribution.

Grouping Metric

When the reference planner is the same as the planner under evaluation the expansion cost metric is a perfect guide for ranking and grouping instances. When the reference planner is different to the planner under evaluation the cost metric may become misleading; e.g., $c_i < c_j$ according to reference planner A but $c_j < c_i$ according to planner-of-interest B.

In Sturtevant's popular benchmark set (Sturtevant 2012) the instances are grouped using a *path length* metric. Being independent of any particular planner, this approach seems advantageous compared to expansion cost. In Figure 3b we show the expansion cost rank and path length rank for one set of benchmark problems. We see that our clustering approach selects instances with a similar rank according to both metrics. Next we report, in Figure 3c, the expansion cost rank of grid A* vs. the expansion cost rank of two optimal planners: grid-based Jump Point Search (Harabor and Grastien 2014) and mesh-based Polyanya (Cui, Harabor, and Grastien 2017). Again, we see a strong correlation, despite differences in map representation and searching and pruning strategies. In Figure 3d we re-cluster the sample using the path length metric and repeat the same experiment. Notice that now instances which have similar rank for path length can have very different rank according to expansion cost. This is the main weakness of the path length metric.

Experiments

We test several currently leading methods and baselines from the academic literature. The idea is to provide performance indicators and examine tradeoffs. The comparisons are apples-to-oranges and there is no winner.

instances	#	Polyanya	A*	JPS
cha	8 k	18.99	5891	53.05
pol	12 k	20.51	4757	47.24
rus	14 k	24.88	6305	62.62
sax	14 k	23.86	7172	56.44
mp	20 k	26.30	6055	50.02
end	2 k	9.195	2948	25.60
all	70 k	123.7	33 130	295.0

Table 2: Runtime results for all instances in each map group. Results are given as cumulative runtimes (in seconds).

- Polyanya (Cui, Harabor, and Grastien 2017), is a leading online mesh-based planner, and also the algorithm used in Iron Harvest. For our experiments we use a C++ implementation due to the original authors (Cui, Harabor, and Grastien 2017). Note that we slightly modify their code, including adding support for our new mesh format.
- JPS (B+P) (Harabor and Grastien 2014). This is a leading planner for online grid search. We use C++ implementations from the Warthog pathfinding library [2].
- Grid A*. This is a standard reference algorithm. Our C++ implementation shares a common base with JPS.

Experiments were conducted on an Intel Core i7-8750H CPU. fixed at 2.2 GHz with boost turned off. Our operating system is Arch Linux (Kernel version 5.17.5) and we compile with G++ 11.2.0. We run one map at a time, single query and in input order. Results are reported in Table 2. The total runtime of A*, over all 70K queries, was $\sim$9 hours, as compared to $\sim$5 minutes for JPS and $\sim$2 minutes for Polyanya. Both Polyanya and JPS are online and optimal algorithms, although they use different representations. An advantage of Polyanya over JPS is that it guarantees to return Euclidean shortest paths, rather than grid-optimal shortest paths. An advantage of JPS is that the grid is faster to update in case of dynamic changes.

[2] https://bitbucket.org/dharabor/pathfinding

Conclusion

We introduce a new unified pathfinding benchmark using data from the game Iron Harvest. Containing larger and more challenging maps and instances than those currently available in the literature, our benchmark is also the first which allows for "fair" comparison of pathfinding algorithms that start from different assumption sets. Future work includes extending the benchmark to other query types; e.g. dynamically changing terrain and multi-target search.

Acknowledgements

We thank KING Art Games, for the release of Iron Harvest game data. Their generous contribution makes this work work possible. We also thank Nathan Sturtevant, for useful comments and discussion on the topic benchmark biases. Nathan pointed out the impact that search direction can have when generating instance data and these comments allowed us to adjust and improve our approach. Research at Monash University is supported in part by the Australian Research Council under grants DP190100013 and DP200100025 and also by a gift from Amazon.

References

Antsfeld, L.; Harabor, D.; Kilby, P.; and Walsh, T. 2012. Transit routing on video game maps. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 8, 2–7.

Brewer, D. 2019. Tactical Pathfinding on a NavMesh. *Game AI Pro 360: Guide to Tactics and Strategy*, 25.

Cui, M. L.; Harabor, D.; and Grastien, A. 2017. Compromise-free Pathfinding on a Navigation Mesh. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, 496–502. AAAI Press.

Demyen, D.; and Buro, M. 2006. Efficient Triangulation-Based Pathfinding. In *AAAI*, 942–947.

Gillberg, J. 2019. AI for Testing: The Development of Bots that Play 'Battlefield V'. Game Developers Conference.

Harabor, D.; and Grastien, A. 2014. Improving Jump Point Search. *Proceedings of the International Conference on Automated Planning and Scheduling*, 24(1).

Harabor, D. D.; Uras, T.; Stuckey, P. J.; and Koenig, S. 2019. Regarding Jump Point Search and Subgoal Graphs. In *IJCAI*, 1241–1248.

Hu, Y.; Harabor, D.; Qin, L.; and Yin, Q. 2021. Regarding Goal Bounding and Jump Point Search. *Journal of Artificial Intelligence Research*, 70: 631–681.

Johnson, E. 2019. Guide to Effective Auto-Generated Spatial Queries. *Game AI Pro 360: Guide to Tactics and Strategy*, 183.

Kallmann, M.; Bieri, H.; and Thalmann, D. 2004. Fully dynamic constrained delaunay triangulations. In *Geometric modeling for scientific visualization*, 241–257. Springer.

Kring, A.; Champandard, A.; and Samarin, N. 2010. DHPA* and SHPA*: Efficient hierarchical pathfinding in dynamic and static game worlds. In *AIIDE*, volume 5.

Mononen, M. 2009. Recast Navigation. https://github.com/recastnavigation/recastnavigation. Accessed: 2022-06-21.

Oh, S.; and Leong, H. W. 2017. Edge N-Level Sparse Visibility Graphs: Fast Optimal Any-Angle Pathfinding Using Hierarchical Taut Paths. In *Symposium on Combinatorial Search (SoCS)*, 64–72.

Orkin, J. 2006. Three States and a Plan: The A.I of F.E.A.R. Game Developers Conference.

Snook, G. 2000. Simplified 3D Movement and. Pathfinding Using Navigation Meshes. In *Game Programming Gems*, 288–304. Charles River Media.

Stout, B. 1996. Smart Moves: Intelligent Pathfinding. *Game Developer Magazine*, October: 28–35.

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

Sturtevant, N.; and Geisberger, R. 2010. A comparison of high-level approaches for speeding up pathfinding. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 6, 76–82.

Sturtevant, N. R. 2007. Memory-Efficient Abstractions for Pathfinding. *AIIDE*, 684: 31–36.

Sturtevant, N. R. 2019. Choosing a search space representation. In *Game AI Pro 360*, 13–18. CRC Press.

Tozour, P. 2002. AI Game Programming Wisdom 2. In Rabin, S., ed., *Search Space Representations*, 85–113. Charles River Media.

Uras, T.; and Koenig, S. 2018. Understanding Subgoal Graphs by Augmenting Contraction Hierarchies. In *IJCAI*, 1506–1513.

van Toll, W.; Triesscheijn, R.; Kallmann, M.; Oliva, R.; Pelechano, N.; Pettré, J.; and Geraerts, R. 2020. Comparing navigation meshes: Theoretical analysis and practical metrics. *Computers & Graphics*, 91: 52–82.

Young, T. 2001. Expanded Geometry for Points-of-Visibility Pathfinding. In *Game Programming Gems 2*, 317–323. Charles River Media.