ISSN 1691-5402
Environment. Technology. Resources
Proceedings of the 9th International Scientific and Practical Conference. Volume 1I

Pathfinding Algorithm Efficiency Analysis in 2D Grid

Imants Zarembo, Sergejs Kodors
Rezekne Higher Education Institution

Abstract. **The main goal of this paper is to collect information about pathfinding algorithms A*, BFS, Dijkstra's algorithm, HPA* and LPA*, and compare them on different criteria, including execution time and memory requirements. Work has two parts, the first being theoretical and the second practical. The theoretical part details the comparison of pathfinding algorithms. The practical part includes implementation of specific algorithms and series of experiments using algorithms implemented.**

Such factors as various size two dimensional grids and choice of heuristics were taken into account while conducting experiments.

Keywords **– A*, BFS, Dijkstra's algorithm, HPA*, LPA*, shortest path problem.**

I Introduction

Pathfinding theory describes a process of finding a path between two points in a certain environment. In the most cases the goal is to find the specific shortest path, which would be optimal, i.e., the shortest, the cheapest or the simplest. Criteria such as, a path, which imitates path chosen by a person, a path, that requires the lowest amount of fuel, or a path from point A to point B through point C is often found relevant in many pathfinding tasks.

The shortest path problem is a pressing issue in many fields, starting with navigation systems, artificial intelligence and ending with computer simulations and games. Although all of these fields have their own specific algorithms, there are many general purpose pathfinding algorithms which can be successfully applied. But it is not always clear what advantages certain algorithm has in comparison to its alternatives.

As a part of this paper pathfinding algorithms A*, BFS, Dijkstra's algorithm, HPA* and LPA* were implemented to analyze their efficiency in an environment based on two dimensional grid. Such factors as grid size, traversed node count and execution time were taken into consideration conducting series of experiments.

II Materials and methods

To assess algorithm efficiency in two dimensional grids experiments were conducted using A*, BFS, Dijkstra's algorithm, HPA* and LPA* to find the shortest path between two randomly placed nodes. Algorithm execution times and traversed node count were measured.

Two dimensional grids used in experiments contained two types of nodes: passable and blocked. 20% of grid was randomly filled with blocked nodes. To assess algorithm efficiency experiments were conducted on various size grids: 64x64, 128x128, 256x256, 512x512 and 1024x1024 nodes.

In case of HPA* three level hierarchy was chosen and initial grid was divided into 4x4 clusters. These parameters were chosen because any smaller division of base grid (64x64 in this case) would lead to incorrect search results while executing preprocessing phase.

Manhattan distance was chosen as heuristic function, because it is strictly grid based distance:

$$H = |x_1 - x_2| + |y_1 - y_2|. \quad (1)$$

Every experiment was repeated 100 times to reduce the amount of random errors. Algorithms were implemented assuming that pathfinding may only occur horizontally or vertically, with no diagonal movement. Every transition between two neighboring nodes costs 1.

All experiments were conducted on the computer with CPU running at a frequency of 2.8 GHz.

III Algorithm A*

A* is a pathfinding algorithm used for finding optimal path between two points called nodes. Algorithm A* uses best-first search to find the lowest cost path between start and goal nodes. Algorithm uses heuristic function, to determine the order in which to traverse nodes. This heuristic is sum of two functions:

G — exact cost of the path from initial node to the current node;

H — admissible (not overestimated) cost of reaching the goal from current node;

$F = G + H$ — cost to reach goal, if the current node is chosen as next node in the path.

Estimated heuristic cost is considered admissible, if it does not overestimate the cost to reach the goal [3].

Selection of heuristic function is an important part of ensuring the best A* performance. Ideally H is equal to the cost necessary to reach the goal node. In this case A* would always follow perfect path, and would not waste time traversing unnecessary nodes. If

overestimated value of H is chosen, the goal node is found faster, but at a cost of optimality. In some cases that may lead to situations where the algorithm fails to find path at all, despite the fact, that path exists. If underestimated value of H is chosen, A* will always find the best possible path. The smaller H is chosen, the longer it will take for algorithm to find path. In the worst-case scenario, $H = 0$, A* provides the same performance as Dijkstra's algorithm [2].

A* starts its work by creating two node lists: a closed list containing all traversed nodes and an open list of nodes that are being considered for inclusion in the path. Every node contains three values: F, G and H. In addition to these three values every node needs to contain information about which node precedes it to determine path by which this node can be reached.

IV Algorithm breadth-first search

Breadth-first search (BFS) is one of the simplest graph search algorithms and is a prototype for several more advanced algorithms. Prim's minimal spanning tree algorithm and Dijkstra's single-source graph search algorithm uses principles similar to BFS [1].

Given a graph $G = (V, E)$ and the starting node s, BFS will systematically traverse edges of G, to find all nodes, that are reachable from node s. It calculates distance (the smallest number of edges) from node s to every reachable node and creates breadth-first tree, which contains all reachable nodes. The root of this tree is node s. Every node v reachable from node s in breadth-first tree makes the shortest path from s to v in graph G, i.e., path which contains the smallest number of edges. The algorithm is applicable to directed and undirected graphs.

To follow search progress, breadth-first search algorithm marks all nodes in white, gray or black. All nodes are white in the beginning. When during the search node is encountered for the first time it becomes gray or black. Gray and black nodes are considered visited. BFS sorts these nodes to ensure that search is progressing breadth-first. If $(u, v) \in E$ and node u is black, then node v is gray or black i.e. all black node neighbors have been visited. Gray nodes can have white neighbors, they represent border between visited and not visited nodes.

The algorithms complexity in time can be expressed as $O(|E| + |V|)$, in the worst case scenario every edge and every node is visited. $O(|E| + |V|)$ can fluctuate between $O(|V|)$ and $O(|V|^2)$ depending on graph edge evaluation.

V Dijkstra's algorithm

Dijkstra's algorithm deals with single-source the shortest path problems in directed, weighted graphs $G = (V, E)$ with non-negative edge costs ($w(u, v) \geq 0$ for every edge $(u, v) \in E$) [2]. Dijkstra's algorithm maintains set of nodes S, whose final shortest-path weights from source s have already been determined. The algorithm repeatedly selects nodes $u \in V - S$ with the minimum shortest-path estimate, sums u and S, and relaxes all edges leaving u.

Dijkstra's algorithm is called "greedy" algorithm, because it always chooses "the lightest" and "the nearest" node $V - S$ to add to the set S.

The simplest implementation of Dijkstra's algorithm holds the set of nodes Q in simple linked list and finding node with minimal weight is linear search in set Q. In this case algorithm execution time is $O(|E| + |V|^2)$. The algorithm worst case performance can be expressed as $O(|E| + |V| \log |V|)$ [5].

VI Algorithm HPA*

Hierarchical pathfinding A* was developed by Adi Botea and his colleagues in 2004. HPA* is a near-optimal pathfinding algorithm, it finds paths which are within 1% of optimal [7]. It is combination of pathfinding and clusterization algorithms, which works by creating an abstract graph on the basis of two dimensional grids. The main HPA* principle is based on dividing search problem into several smaller sub problems, and caching results for every path segment [8].

Clusterization, used in this algorithm, is relatively simple: a low resolution two dimensional grid $s \times s$ is created, where s is a size of new grid. New grid is placed directly above the initial grid. Every node in new grid becomes a cluster. All initial grid nodes that are located under according cluster are considered members of that cluster. Each cluster is considered separate graph. The abstract graph is then created to connect all separate graphs. To achieve that, border nodes needs to be found between neighboring clusters - nodes that are on cluster outer sides are checked. If node has a passable neighbor in an adjacent cluster, it is considered connected, and connection between two graphs representing clusters are added to abstract graph. In cases where there are many adjacent connections between two clusters, they are combined into one entrance. Then entrances are added to abstract graph and connected. Abstract graph still lacks internal edges (paths between entrances inside one cluster). These edges are created by running A* algorithm through every node in each separate cluster. If A* finds path, its cost becomes costs of found abstract edge, else edge is not added to abstract graph. Inter-cluster edges inherit their costs from initial graph edge cost. Finally abstract graph is ready for pathfinding using A* [9].

HPA* pathfinding phase consists of two parts called preprocessing and online search. During preprocessing start and goal nodes are inserted into abstract graph, and inter-cluster edges are added. Then A* is used on abstract graph to find the shortest route. During online search the shortest route found in abstract graph is refined to full path in initial graph using A*. To find full path from start to goal node A* is used in every cluster on nodes that connect clusters. Finally partial results from every cluster are combined into full path [10].

VII ALGORITHM LIFELONG PLANNING A*

Lifelong Planning A* (LPA*) is an algorithm intended for solving the shortest-path problems on known finite graphs whose edge cost increase or decrease over time [5]. S denotes the finite set of nodes of the graph. $succ(s) \subseteq S$ denotes the set of successors of node $s \in S$. Similarly, $pred(s) \subseteq S$ denotes the set of predecessors of node $s \in S$. $0 < c(s, s') \leq \infty$ denotes the cost of moving from node s to node $s' \in succ(s)$. LPA* always determines the shortest path from a given start node s_{start} to a given goal node $s_{goal} \in S$, knowing both the topology of the graph and the current edge costs. The start distances satisfy the following relationship:

$$g^*(s) = \begin{cases} 0 & if\ s = s_{start} \\ min_{s' \in pred(s)}\left(g^*(s') + c(s', s)\right) & otherwise. \end{cases} \quad (2)$$

$g^*(s)$ denotes the start distance to node $s \in S$, i.e., the cost of the shortest path from s_{start} to s.

LPA* is an incremental version of A* that applies to the same finite path-planning problems as A*. It shares with A* the fact that it uses nonnegative and consistent heuristics $h(s)$ that approximate the goal distance of the node s to focus its search. Consistent heuristics obey the triangle inequality $h(s_{goal}) = 0$ and $h(s) \leq c(s, s') + h(s')$ for all nodes $s \in S$ and $s' \in succ(s)$ with$s \neq s_{goal}$. LPA* reduces to a version of A* that breaks ties among vertices with the same F value in favor of smaller G values when LPA* is used to search from scratch and to a version of DynamicsSWSF-FP that applies to path-planning problems and terminates earlier than the original version of DynamicsSWSF-FP when LPA* is used with uninformed heuristics [6].

VIII RESULTS AND DISCUSSION

Algorithm execution time

Breadth-first search is brute-force search algorithm; its results differ noticeably in comparison with informed search algorithms. Table I shows, that the algorithm execution time increases exponentially with search area size increase.

TABLE I
ALGORITHM BFS EXECUTION TIME

Grid size (nodes)	Execution time (ms)
64x64	150
128x128	2803
256x256	48313
512x512	821598
1024x1024	13962457

To find the shortest path in 512x512 node grid, the algorithm took 821 seconds and in 1024x1024 node grid — 13962 seconds. This considerable execution time shows that the algorithm is the most likely not applicable to real-time search problems in large grids.

Increasing the search problem size Dijkstra's algorithm execution time increases linearly. On average in 1024x1024 grid the algorithm finds the shortest path in 2,3 seconds. Table II shows the algorithm execution times for different grid sizes.

TABLE II
DIJKSTRA'S ALGORITHM EXECUTION TIME

Grid size (nodes)	Execution time (ms)
64x64	6
128x128	25
256x256	120
512x512	515
1024x1024	2362

Algorithm A* performance was greater in comparison with Dijkstra's algorithm in every grid size selected for experiments. The algorithms execution time increases linearly with grid size. Table III shows the algorithm execution times for different grid sizes.

TABLE III
ALGORITHM A* EXECUTION TIME

Grid size (nodes)	Execution time (ms)
64x64	4
128x128	16
256x256	77
512x512	265
1024x1024	1148

Lifelong Planning A* performance is higher than Dijkstra's algorithms in all grid sizes, but it is lower than A* performance in 512x512 and 1024x1024 node grids. The algorithms execution time increases linearly with grid size. Table IV shows the algorithm execution times for different grid sizes.

TABLE IV
ALGORITHM LPA* EXECUTION TIME

Grid size (nodes)	Execution time (ms)
64x64	4
128x128	11
256x256	57
512x512	319
1024x1024	1490

Algorithm HPA* execution time, using 4x4 clusters and 3 level hierarchy, is lower than any other algorithm in this experiment. The algorithms execution time increases linearly with grid size. Table V shows the algorithm execution times for different grid sizes.

TABLE V

ALGORITHM HPA* EXECUTION TIME

Grid size (nodes)	Execution time (ms)
64x64	3
128x128	14
256x256	52
512x512	190
1024x1024	775

The experimental data shows, that the fastest execution times belong to HPA* in almost all grid sizes, only dropping behind LPA* in 128x128 grid by 3 ms. The slowest execution times were shown by BFS, which was considerably slower than the second slowest algorithm — Dijkstra's. All algorithm execution times are shown in Table VI and graphically in Fig. 1.

TABLE VI

ALGORITHM EXECUTION TIME

Algorithm	Grid size (nodes)				
	64x64	128x128	256x256	512x512	1024x1024
BFS	150	2803	48313	821598	13962457
Dijkstra	6	25	120	515	2362
A*	4	16	77	265	1148
LPA*	4	11	57	319	1490
HPA*	3	14	52	190	775

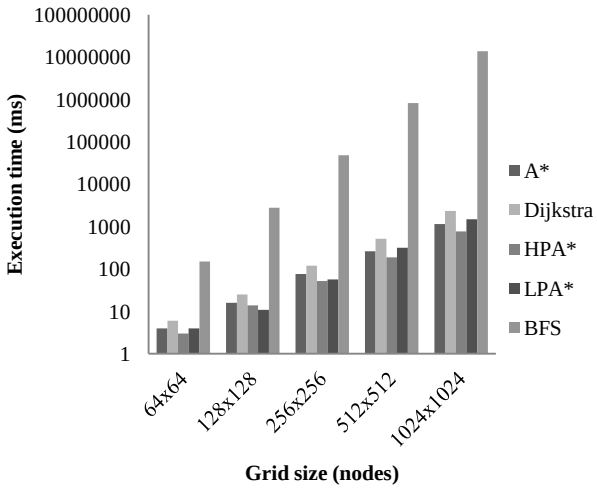

Fig. 1. Algorithm execution time

Traversed nodes

Breadth-first search traverses the most nodes from all the algorithms discussed. Table VII shows the algorithm traversed node count for the different grid sizes.

TABLE VII

ALGORITHM BREADTH-FIRST SEARCH TRAVERSED NODES

Grid size (nodes)	Traversed nodes
64x64	3155
128x128	12887
256x256	52367
512x512	213648
1024x1024	1159255

While searching for a path Dijkstra's algorithm traversed slightly less nodes than BFS. Similar amount of visited nodes for Dijkstra's algorithm and BFS can be explained by the fact, that both algorithms use similar node traversal principles. Table VIII shows the algorithm traversed node count for different grid sizes.

TABLE VIII

DIJKSTRA'S ALGORITHM TRAVERSED NODES

Grid size (nodes)	Traversed nodes
64x64	3173
128x128	13058
256x256	52068
512x512	209251
1024x1024	836977

Algorithm A* traversed less nodes than BFS, Dijkstra's algorithm or LPA* while searching for the shortest path. The algorithm uses heuristics to expand nodes in the direction of the goal thus minimizing traversed node count. Table IX shows the algorithm traversed node count for different grid sizes.

TABLE IX

ALGORITHM A* TRAVERSED NODES

Grid size (nodes)	Traversed nodes
64x64	623
128x128	1576
256x256	8071
512x512	40333
1024x1024	104109

Lifelong Planning A* traversed node count increases linearly with grid size increase. On average LPA* traverses half as much nodes as Dijkstra's algorithm. Table X shows LPA* algorithm traversed node count for different grid sizes.

TABLE X

ALGORITHM LPA* TRAVERSED NODES

Grid size (nodes)	Traversed nodes
64x64	994
128x128	6163
256x256	25004
512x512	115973
1024x1024	460318

Hierarchical Pathfinding A* traversed the least nodes from all selected algorithms in all grid sizes. This can be explained by the fact, HPA* only searches paths within selected clusters, which were chosen in preprocessing phase. Table XI shows the algorithm traversed node count for different grid sizes.

TABLE XI
ALGORITHM HPA* TRAVERSED NODES

Grid size (nodes)	Traversed nodes
64x64	454
128x128	1334
256x256	3551
512x512	10629
1024x1024	41491

Comparing algorithms by nodes traversed, Breadth-first has traversed the most nodes and Hierarchical Pathfinding A* - the least nodes. All algorithms traversed node counts are shown in Table XII and graphically in Fig. 2.

TABLE XII
ALGORITHM TRAVERSED NODES

Algorithm	Grid size (nodes)				
	64x64	128x128	256x256	512x512	1024x1024
A*	623	1576	8071	40333	104109
Dijkstra	3173	13058	52068	209251	836977
HPA*	454	1334	3551	10629	41491
LPA*	994	6163	25004	115973	460318
BFS	3155	12887	52367	213648	1159255

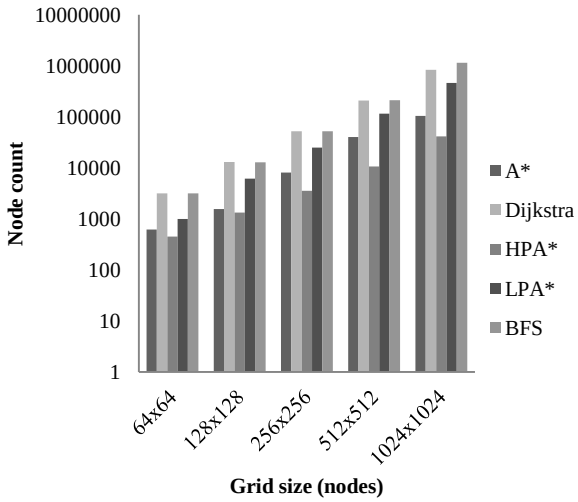

Fig. 2. Algorithm traversed nodes

Breadth-first search results overall are similar to Dijkstra's in 64, 128, 256 and 512 nodes grids, but falls behind in 1024 node grid.

IX CONCLUSIONS

Comparing A*, Breadth-first search, Dijkstra, HPA* and LPA* algorithms execution times in different size two dimensional grids, the slowest was BFS. This result can be explained by the fact, that the algorithm operation principle is very simple and it does not use any heuristics. Dijkstra's algorithm was faster than BFS, but slower than all other algorithms. A* and LPA* performance was similar: LPA* was faster in smaller grids (64, 128, 256), but A* in larger (512, 1024). Which leads to conclusion, that LPA* is better suitable for smaller pathfinding problems, while A* is better used for solving larger problems. Algorithm HPA* was the fastest in searching path between 2 points, primarily because of hierarchical problem division into smaller parts.

Breadth-first search traversed the most nodes, closely followed by Dijkstra's algorithm. LPA* traverses node count was larger than A* in all grid sizes. A* traversed node count was the second best amon discussed algorithms. HPA* traversed the least nodes.

X REFERENCES

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest and S. Clifford, *Introduction to Algorithms (3rd ed.).* MIT Press and McGraw-Hill, 2009. pp. 658.

[2] E. W. Dijkstra, "A note on two problems in connexion with graphs," Numerische Mathematik, vol. 1, 1959, pp. 269-271.

[3] P. E. Hart, N. J. Nilsson and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," in *IEEE Transactions of Systems Science and Cybernetics*, vol. 4, 1968, pp. 100-107.

[4] D. Wagner, T. Willhalm, "Geometric Speed-Up Techniques for Finding Shortest Paths in Large Sparse Graphs". 2003.

[5] S. Koenig, M. Likhachev, D. Furcy. "Lifelong Planning A*". Artificial Intelligence, Vol. 155 Issue 1-2, 2004, pp. 93 - 146.

[6] L. Sangeorzan, K. Kiss-Iakab, M. Sirbu, "Comparison of 3 implementations of the A* algorithm". North University of Baia Mare. 2007.

[7] A. Botea, M. Muller, J. Schaeffer, "Near, Optimal Hierarchical Path-Finding," Journal of Game Development, 1(1): 2004, pp. 7-28.

[8] M. R. Jansen, M. Buro, *HPA* Enhancements. Proceedings of the Third Artificial Intelligence and Interactive Digital Entertainment Conference*, Stanford, California, USA. 2007.

[9] N. Sturtevant, M. Buro, *Partial Pathfinding Using Map Abstraction and Refinement. AAAI 05 Proceedings of the 20th national conference on Artificial intelligence*, vol. 3. 2005. pp. 1392 – 1397.

[10] R. C. Holte, M. B. Holte, R. M. Zimmer, A. J. MacDonald, *Hierarchical A*: Searching Abstraction Hierarchies Efficiently. AAAI 96 Proceedings of the thirteenth national conference on Artificial intelligence*, vol. 1, 1996. pp. 530 – 535.