Preliminary report / Prethodno priopćenje
Manuscript received: 2018-09-12
Accepted: 2018-09-28
Pages: 1 - 10

Performance Issues And Gains Of Caching The Pathfinding Data

Ivan Porkolab
Visoko učilište Algebra
Zagreb, Croatia
ivan.porkolab@racunarstvo.hr

Goran Djambic
Visoko učilište Algebra
Zagreb, Croatia
goran.djambic@algebra.hr

Danijel Kucak
Visoko učilište Algebra
Zagreb, Croatia
danijel.kucak@algebra.h

Abstract: Using non-cached methods for finding the shortest path between nodes is the most common case when using pathfinding systems. That approach generates a couple of issues. Foremost, it has a significant impact on processing resources as calculations must be done over again for each iteration, even for the repeating events. That's not a big concern if pathfinding is invoked a reasonable number of times or the nodes involved are always different, but if pathfinding occurs many times on the same nodes, then the caching of once calculated path becomes an acceptable course of action. This paper has explored one of such caching algorithms, FAST-N algorithm and compared it with standard non-cached pathfinding. Doing so, it outlined margins of justifiable use of such systems.

On a small number of pathfinding requests or simple node structure, because of increase in memory usage and rather hefty initial calculation processing requirements, it has been concluded that non-cached system makes more sense than cached one. On the other hand, when confronted with a large number of pathfinding requests and more complex node structure, caching can generate significant benefits concerning processing power and speed.

Keywords: Pathfinding; Caching; FAST-N; C#

INTRODUCTION

Pathfinding based on computer science has many uses nowadays. Most common usage is assisted GPS navigation systems, indoor navigation, moving non-playable or playable characters in games, AI-based navigation – namely, self-driving cars, rovers, robots and such. For all those purposes, different systems use different algorithms for finding the shortest or optimal path, and some of them use caching procedures to reduce the impact on processing requirements of those algorithms.

Pathfinding in a standard graph system, where the objective is to find a path from one node to the other has a couple of main problems to solve. As the shortest path is the usual goal, pathfinding algorithms must find all paths that have a potential of becoming the shortest one and then select the one that is the shortest. [10]

The biggest issue with pathfinding algorithms in everyday use, as in getting directions in navigation or moving characters in computer games, is the required computational power to calculate the optimal path. Single computation is usually not an issue but when a large number of such computations must be made, an impact on the system can be significant.[9] In the hypothetical situation where thousands of drivers drive the same route and need directions on it, the system would need to recalculate the same shortest path over and over again. Therefore the need to cache once calculated data.

Mostly used caching algorithms are those based on navmesh, navigational mesh. It converts the geographical data to a number of polygons and then calculates which of those polygons are passable, and how to get from one to other in the shortest manner. [1]

To test the efficiency of the caching pathfinding data, we have developed a simple table referencing caching system called FAST-N which is based on the node graph structure. The reason for developing a node based system versus using some of the predeveloped navigational mesh-based systems is the computational cost of transforming the node based system to a polygonal one. As stated in the literature [9], that conversion can be quite expensive. The cost can be justified in many cases, but in cases when the underlying data is a node-based graph, i.e. a grid of streets and crossroads or some sort of maze in computer games, it makes no sense to transform the graph.

In a series of tests, the FAST-N caching system will be opposed to the standard non-caching algorithm – A* to measure the potential gain of using the cached data. To test the computational efficiency of the opposed approaches, both will solve same Maze problems. Implementations, both of FAST-N and A* algorithms are programmed in C#. All measurement were made on the same computer system under the same conditions.

PATHFINDING

Pathfinding can be done in a number of different ways, using a range of algorithms. Choosing optimal algorithm deeply depends on a structure of the graph, distribution of the nodes and the purpose of the calculations. Some of the methods for finding shortest path are based on algorithms for finding shortest path on graphs, or other methods which are used when we are trying to describe locations of the obstacles. [2]

A couple of the most used algorithms on a graph are Dijkstra's algorithm and more often its more optimized variant - A* (pronounced „A star"). Bout are conducted on prepared graphs node systems, most likely derived from the triangulated poligonal data. [4]

2.1. DIJKSTRA'S ALGORITHM

Dijkstra's algorithm is an algorithm for finding the shortest path between two points in a graph system of a reachable point, resolving so-called Point to Point (P2P) problems. [3] Scientist Edsger Dijkstra had described it in 1959, and since then it has become a base of most modern shortest path algorithms, namely A* which is basically a generalization of Dijkstra's algorithm.

Aldo Dijkstra will find an exact shortest path, so it's not an approximation, it has some issues when dealing with large sets of nodes. By its construct, to reach the desired shortest path, an algorithm will search the whole graph and calculate the distances from each of the nodes. Such approach can make sense on graphs with the smaller number of nodes, but when dealing with more massive sets, the computational power required to achieve that task grows significantly. For instance, a road system of United States has more than 20 million of crossroads so the use of Dijkstra on such a set would be virtually impossible for everyday use in GPS navigation systems. [5] But Dijkstra's is used in navigation, never the less. Not to find the micro route inside the cities, but to find the fastest way from one town to another, where every city is presented as a node in the graph, and with known distances from one city to other. While inside the destination city, different algorithms are used, namely A* as a system to find the shortest path to the exact location. The problem of finding the optimal route through a number of nodes is commonly known as a Traveling Salesman Problem. [6] Also, Dijkstra is commonly used in network routing protocols such as Open Shortest Path First. [5]

2.2. A ALGORITHM*

As a generalization of Dijkstra algorithm, A* has added a heuristic function to the algorithm representing the distance from each node and a destination node. So when moving through the graph, it is not done in the snowballing manner of Dijkstra's, but it can temporarily reject the nodes with increasing distance and select to go through more promising ones first. In real-world applications, the distance function is commonly represented by Euclidian distance. [7] We can look at the Dijkstra's algorithm as a special case of A* algorithm, where the value of heuristic function always equals zero. [8]

2.3. COMPARISON OF PERFORMANCE AND COMMON USE OF A AND DIJKSTRA'S ALGORITHMS*

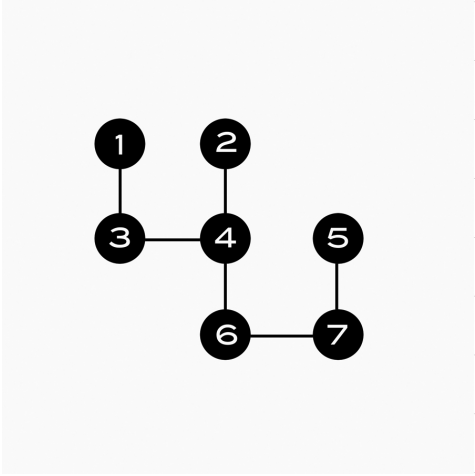
For most of the usage in games, A* proved to be the most adequate, considering ratio of speed and the need for computational power. The same is true for calculating traveling data for GPS systems. Dijkstra's algorithm is the most efficient when there is a need to map the distances from one node to each other nodes in the graph. [7]. That is the reason the Dijkstra's algorithm has been chosen for the precomputation needs of a FAST-N algorithm.

2.4. THE PROBLEM OF SHORTEST PATH CALCULATIONS

When dealing with finding the shortest path, the biggest issue is computer processing power required to find the shortest path when graph consists of a large number of or when there is a significant number of searches required in the short period, especially when the same or similar searches happen a number of times. When confronted with that kind of situation, caching of once calculated search results make sense. [9]

FAST-N ALGORITHM

FAST-N, standing for Fast Asynchronous System for Transient Nodes is an algorithm which precalculates the pathfinding data and exposes results of a search available without the impact on the computational power. It calculates the optimal route from each node to each other node and stores the data in reference table where the optimal path can be found by directly accessing the required node's data. It is done in the manner that the path from one node to the other is „recorded" in a matrix as a series of pointers to the next nodes on the path. On the Figure 1., is an example of such a routing table. If the required path is from node N1 to the node N5, one can quickly get the shortest path without any calculation. In the matrix rows represent initial, start nodes; and the columns represent the final, destination nodes. So, to get the shortest path from nodes N1 to N5, we first take a look at row 1 and read the value in column 5. The value is 3, and that is the first node in our shortest path. Then we take a look at row 3 and the column 5 and get the value 4, which is our second node. Then look at row 4 and column 5 to get 6; then 6 to 5 gives us 7, and finally, 7 to 5 is 5; destination reached. So the shortest path between 1 and 5 is [1, 3, 4, 6, 7, 5]. The path is found with zero runtime calculations, only by referencing the routing table.



	1	2	3	4	5	6	7
1	X	4	1	3	7	4	6
2	3	X	4	2	7	4	6
3	3	4	X	3	7	4	6
4	3	4	4	X	7	4	6
5	3	4	4	6	X	7	5
6	3	4	4	6	7	X	6
7	3	4	4	6	7	7	X

Image 1. Simple node graph and equivalent Reference table

3.1 SETTING UP THE NODES FOR THE PATHFINDING REQUIREMENTS

When setting up nodes in the graph system, the FAST-N algorithm uses different approach then mentioned *navmesh*. It defines the border points for each impassable object, i.e., a building, and uses those points as nodes in pathfinding calculations.

Appropriate points are chosen in a way that each point of a square object has optical visibility of two adjacent nodes. Also, the node is set on each of the possible crossroads when dealing with the finding the rout using the roads or trying to navigate through the labyrinth.

3.2. CALCULATION OF THE SHORTEST PATH

After all of the nodes have been set, calculation of the shortest path from each node to each other node takes place. FAST-N uses Dijkstra‘s algorithm, for it has to calculate the shortest path from each node to each other node. Once all calculations have been done, the result is stored in reference table as shown in Image 1. The result, the reference table is a matrix of N*N in size, where N is the number of nodes. The size of the matrix can present a significant impact on memory when dealing with a large number of nodes.

MAZE PROBLEM

To test benefits of using caching algorithm over standard recalculation approach to the finding of the shortest path it has been decided to use a randomly generated mazes in which is easy to define the overall complexity of the path. Mazes are generated using a couple of rules. They always have exactly one entrance and one exit set diagonally one from other. The complexity of the maze – Cx - is determined as a required number of

turns one must make to get from the entrance to the exit. So if one must pass through 5 nodes to get from start to the end node, a maze has complexity set to Cx = 5, and if the number of necessary nodes is ten, then the complexity is set to Cx = 10.

Complexity determined in such way is used to define baseline conditions for this research. This allows us to generate a number of different mazes used for calculations. When mazes have a similar complexity, we can compare the data gathered from the different layouts. We use a number of different layouts because there was a need to show that results are not dependent on some specific maze layout, but they are rather consistent over a range of completely different, randomly generated layouts. Final results, presented in this paper are average results of all layouts of one complexity level tested.

METHODOLOGY

To achieve as little bias as possible, we used a 100 randomly generated mazes for each complexity level tested. The values on the graph represent an average value of one hundred mazes at the same complexity level. We have tested both cached and non-cached algorithms at the maze complexity levels Cx = 10, 100, and 1000. The FAST-N algorithm was used as cached, and A* algorithm was used as standard, non-caching algorithm. As both algorithms are asynchronous, usage of CPU was maximum at the time of the test. Therefore, to get a comparison, the total time needed to finish the task was measured. The task was to find the shortest path from the entrance to the exit for a number of times: Number of searches Ns = 1, 100 and 1000 times. Once the reference table was filled with cached data, the time needed to get the shortest path using the caching systems like FAST-N is next to zero, so the time needed to create reference table was added to the results of the FAST-N algorithm to be able to compare two opposing systems.

RESULTS

Images 2., 3. and 4. shows the result of direct comparison of non-cached A* and cached FAST-N algorithms. Calculated time needed for a FAST-N algorithm for the number of searches Ns = 1 is set as a baseline of 100 on each graph. All other values are scaled accordingly. The lower the values, less time it took for an algorithm to finish the test.

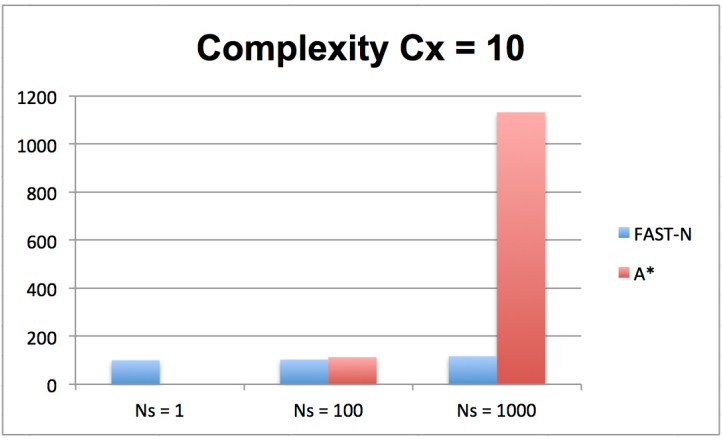


Image 2: Ns = 1, 100, 1000, Cx = 10

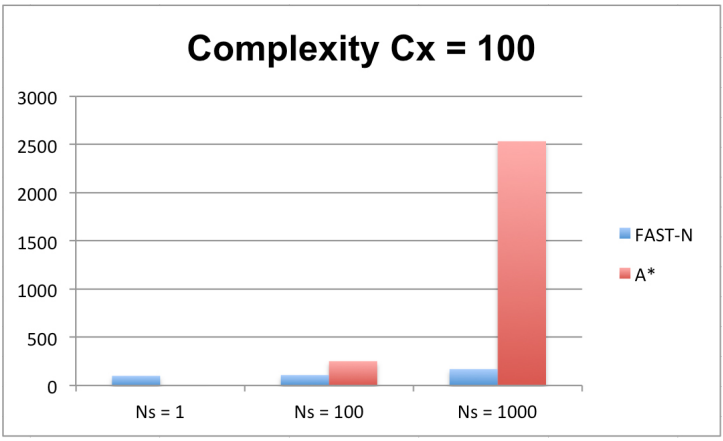


Image 3: Ns = 1, 100, 1000, Cx = 100

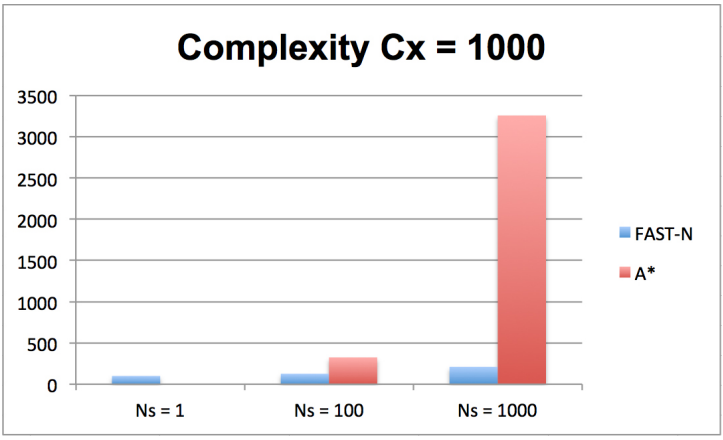


Image 4: Ns = 1, 100, 1000, Cx = 1000

On complexity Cx = 10, as seen on Image 2, A* scored 1,13 for one search, which means it was 88,5 times faster then FAST-N at performing one search. When comparing a number of searches Ns = 100, A* scored 113, and FAST-N 103, which makes it marginally slower at the rate of 9.7%. The difference becomes more obvious on Ns = 1000 where FAST-N scored 117, and A* scored 1132. So on 1000 searches, caching proved to be faster 9,68 times than non-caching approach.

On complexity Cx = 100, seen on Image 3, A* scored 2,50 for Ns = 1 making it 40 times faster than FAST-N. On Ns = 100, FAST-N had score 108, and A* had 252, making it 2,33 times slower. On Ns = 1000, FAST-N had 170 and A* 2533, making the non-cached system slower by the rate of 14,9 times.

On last tested complexity of Cx = 1000, visible on Image 4, on Ns = 1 A* scored 3,21 making it 31,15 times faster than FAST-N. On Ns = 100, A* had a score of 325 and FAST-N of 127, making the A* slower by 2,56 times. On Ns = 1000, FAST-N scored 211 and A* 3257, making it slower by 15,4 times.

6.1. REMARKS

It should be mentioned that reference table generated by FAST-N algorithm allows fast pathfinding from any node to any other node. There is virtually no difference which two nodes are selected. On the other hand, A* makes a major difference in performance depending on which two nodes are selected due to the nature of the algorithm. To minimize that discrepancy, we only tested diagonal positions of the start/end nodes. That configuration is the most favorable for the algorithm as A* trying to find the closest node by measuring its Euclid distances from the destination node.

Also, the *Maze problem* is not a real world problem. Research should be made to see how does caching improves pathfinding on some real world situations, like getting driving direction from navigation in a real city street layout.

FURDER RESEARCH

As shown in this paper, caching pathfinding data is a fairly efficient way to get optimal performance when applied to a more complex Maze problem. Other possible uses of a FAST-N algorithm can be found in caching pathfinding data in computer games, and in finding the fastest route using GPS systems. The FAST-N algorithm itself allows us to make enhancements to the caching system to add a quick and efficient rerouting, i.e., when a certain node becomes unavailable. That can be proved to be beneficial in many real-life situations, like in GPS routing systems, when a certain road becomes blocked or is in a traffic jam. It should be explored how much can FAST-N algorithm help to optimize the performance hit in such situations.

Another possible use is a specific situation when a large number of shortest path calculation must be done to achieve some effect, i.e., particle simulations when possibly tens

of thousands of particles represent a fluid (or gas). The calculation time of such a large number of shortest paths would be significantly reduced by use of a cached system.

CONCLUSION

As expected, results of the test show how caching pathfinding data can make a significant difference in computational power required for the task. Depending on the complexity of the node system and on the number of pathfinding computations on it, we can conclude: (1) in the situations where node system is simple (with number of required turns less or equal to ten), or a problem needs small number of searches through the graph (number of searches less than 90) non-cached approach generates best results; and (2) in the situations where node system is more complex, and there is need to conduct a larger number of searches on the same graph, the cached approach can generate a significant reduction of required computational power.

The trend shows how computational power need rises proportionally with the rise of conducted searches for A* algorithm. On the other side cached FAST-N algorithm has hefty initial requirements, but afterward, it shows an only slight increase in required computational power with the rise of a number of conducted searches.

So as a rule, we can conclude, on a small number of searches, standard non-cached algorithms are far superior to cached ones. On a more substantial number of searches, caching is hugely beneficial, especially on more complex node systems. Caching requires a lot of memory to store cached data, and almost no processing power, while non-cached approach has small memory imprint and large processing power requirement. That should be taken into consideration when deciding which system one should use.

REFERENCES

[1] Gravot, F., Yokoyama, T. and Miyake Y. (2015). Precomputed Pathfinding for Large and Detailed Worlds on MMO Servers, Available from: https://pdfs.semanticscholar.org/aa57/155e58202a5d4b67d05d6bc96cd4b0dce9c7.pdf. Accessed: (2018-09-08).

[2] Yukhimets, D., Zuev, A. and Gubankov, A. (2017). Method of Spatial Path Planning for Mobile Robot in Unknown Environment, Proceedings of the 28th DAAAM International Symposium, pp.0258-0267.

[3] Li, X. H., Hong, S. H. and Fang, K. L. (2011). WSNHA-GAHR: A greedy and A* heuristic routing algorithm for wireless sensor networks in home automation, IET Communications, vol. 5, no. 13, pp. 1797–1805.

[4] Demyen. D. and Buro M. (2006). Efficient Triangulation-Based Pathfinding. Available from: https://skatgame.net/mburo/ps/tra.pdf. Accessed: (2018-09-07).

[5] Goldberg A.V. and Harrelson C. (2005). Computing the Shortest Path: A* Search Meets Graph Theory, Proc. 16th Ann. ACM-SIAM Symp. Discrete Algorithms (SODA ’05), pp. 156-165.

[6] Johnson D. and McGeoch L. (1997). The Traveling Salesman Problem: A Case Study in Local Optimization. Local Search in Combinatorial Optimization, pp. 215-310.

[7] Talan K. and Bamnote G. R. (2015). Shortest Path Finding Using a Star Algorithm and Minimum weight Node First Principle, Available from: http://www.rroij.com/open-access/shortest-path-finding-using-a-star-algorithmand-minimum-weight-node-first-principle.php?aid=56299. Accessed: (2018-05-05).

[8] Goldberg A.V., Kaplan H., Werneck R.F. (2006). Reach for A*: Efficient Point-to-Point Shortest Path Algorithms. Proc. SIAM Workshop Algorithms Eng. and Experimentation, pp. 129-143.

[9] Dickheiser, M. (2003). “Inexpensive precomputed pathfinding using a navigation set hierarchy.” In AI Game Programming Wisdom 2, MA: Charles River Media, 2003, pp. 103–113.

[10] 10] [4] Björnsson, Y., Bulitko, V. and Sturtevant N. (2009). Time-Bounded A*. Twenty-first International Joint Conference on Artificial Intelligence (IJCAI-09). pp. 431-436.