Extended Essay in Computer Science

Comparing the efficiency of different path-finding algorithms to find the shortest path in a square-shaped maze

Candidate name: Jaromír Látal

Candidate number: 000771-0028

Examination session: May 2015

Supervisor: RNDr. Eva Hanulová

School: Spojená škola Novohradská

Word count: 3660

(blank page)

Acknowledgments

I would like to thank Dr. Nathan Sturtevant of University of Denver, Prof. Richard Korf of University of California and David Silver of University College London for answering the numerous questions regarding their papers. I would like to express my gratitude to my Computer Science teacher RNDr. Eva Hanulová for her time, patience and invaluable feedback during work on this Extended Essay.

I would like to express my thanks to Xueqiao Xu for giving me the permission to edit his path-finding library and use it without copyright notices.

Abstract

During an algorithmic challenge I stumbled upon a problem where I could not find the shortest path in a square-sized maze, because of the time-inefficiency of the algorithm. I started to wonder how to solve this problem and found different path-finding algorithms which could be implemented.

In order to have the best execution time, I had to compare their efficiency, regardless of the input supplied. Therefore I have decided to **compare the efficiency of different path-finding algorithms to find the shortest path in a square-shaped maze.**

The three algorithms, A*, Bi-directional Breadth First Search and Dijkstra's algorithm were used in the testing. Regarding A*, the Manhattan, Euclidean and Chebyshev heuristic functions were used.

The application used for testing was coded in scripting-language Python 3.x. It was capable of generating random mazes of size $2^n x 2^n$ / parsing the already existing maps from games to measure the computation time and the number of function calls made by each algorithm on each problem set.

Efficiency of each algorithm was measured in terms of computation time and number of function calls made. The priority of the testing application was set to real-time in order to achieve the most accurate level of measurements.

Despite the different complexity of each algorithm, each of them found the shortest path within reasonable time interval. A* (Manhattan) was the fastest on both datasets, qualifying itself as more suitable for the path-finding process.

The gathered results are usable in real-life situations as long as the path-finding process is bounded within a grid. There are a number of possible improvements – implementation of different path-finding algorithms or usage of a larger-sized problem set.

Word count: 271

Table of Contents

1. Introduction

Since I have discovered passion for programming, I sought to learn as much as possible by not only learning at school but also outside of class – by solving various algorithmic challenges present at online judges. Because there is such a competition present, the only success factor leading to be the winner is the memory efficiency and the speed of the implementation – one hundredth of a second is often enough to be the deciding aspect.

Designing and implementing algorithm for such a competition is rather different, mostly in terms of available computational power and input present. While for most computers it is feasible to calculate and output the result even when using naïve implementations, the online judge evaluates the code at a computer whose computational power is not sufficient and memory not large enough, which results in time limit being exceeded or using more memory than given.

Recently I have stumbled upon the same problem while I was solving another online judge challenge. The task was to find the shortest path from point A to point B in a square-shaped maze, however after I had designed the solution I encountered a problem – the solution did not satisfy the time constraints set by the judge. To improve the execution time, I tried to improve my already existing algorithm and to implement a different algorithm; the result was yet the same.

Given the fact that this topic has as well as real-life significance and has drawn my interest so much, I decided to carry out a further investigation by choosing it as a topic of my Extended Essay. My research question is:

Comparing the efficiency of different path-finding algorithms to find the shortest path in square-shaped maze.

I will study different path-finding algorithms to find the shortest path and then implement three of them. This Extended Essay will describe each of them and then compare their efficiency.

In order to accomplish this, I will develop a testing system, which will be programmed in Python. The efficiency of the algorithms will be calculated according to their computation times and number of function calls made. To ensure the objectivity of the

research and results, I will use two different datasets to test the performance of the algorithms – dataset of random mazes and dataset of game maps.

Lastly, all testing will be done in Python programming language, as I am very familiar with it thanks to the various problems I have tackled before.

2. Path-finding

Path-finding, is by the basic definition, process of finding the shortest route possible from the point A to the point B. This field of research is based heavily on Dijkstra's algorithm for finding the shortest path on a non-negative weighted graph. (Millington & Funge, 2009)

At its kernel, path-finding relies on searching through graph by starting at one vertex and exploring adjacent nodes until the destination node is found, with the intention of finding the shortest route present.

2.1 Real-life application areas of path-finding

Path-finding has many usage areas. Path-finding algorithms are widely employed in the field of robotics, in order to guide a robot around an area evading the obstacles – all without human intervention. (Carsten, Ferguson, Rankin, & Stentz, 2007; Lee & Yu, 2009)

This is useful especially in cases where the human intervention is unable – the distance in-between causes too much delay or the robot should operate underwater, where the remote controlling is not possible.

Another use of path-finding is in computer games (Davis, 2000; Sturtevant, 2007), where an entity has to move between two points, avoiding any walls or obstacles on the way.

Path-finding algorithms can be also used to find the shortest route between two cities (Jacob, Marathe, & Nagel, 1999), the best way to plan an optimized route for couriers & delivery drives or the shortest path when connecting to another server – path in an enormous graph of interconnected hubs.

2.2 Environment representation

In order to be able to find the shortest path, the environment, in which the search will be conducted, has to have its representation. There are multiple possible map representations, such as polygonal maps or navigation meshes (Patel, 2014b), for purposes of this Extended Essay the environment will be represented as a grid of octiles – which is one of the four possible grid topologies. (Björnsson, Enzenberger, Holte,

Schaejfer, & Yap, 2003) Octiles are squares where the movement is allowed as well as orthogonally, not only vertically or horizontally.

2.3 Path-finding Algorithms

There are many path-finding algorithms which are known and used. Ranging from the simplest to the more robust, we know:

2.3.1 Bi-directional breadth-first search (BFS)

Breadth-first search algorithm (Moore, 1959), also known as BFS, begins at the start node and then examines all neighbouring nodes, then all neighbouring nodes two steps away, and continues until the goal node is not found. Even though this algorithm finds its way around obstacles, if there are several paths present, it chooses the shortest one.

BFS algorithm has a couple of caveats – it stretches into all directions equally, instead of directly heading towards the goal node, which means that the finding of the shortest path often takes more time and has large memory requirements.

Bi-directional BFS (Pohl, 1969) is the enhancement of the BFS algorithm. The path-finding starts two simultaneous breadth-first searches, the first one from the start node and the second one from the goal node, stopping when it finds a node which has been already visited by both searches. Even though this enhancement saves substantial amount of work in comparison to BFS (by a factor of 2) (*Figure 1*), it is still considered to be an inefficient path-finding algorithm.

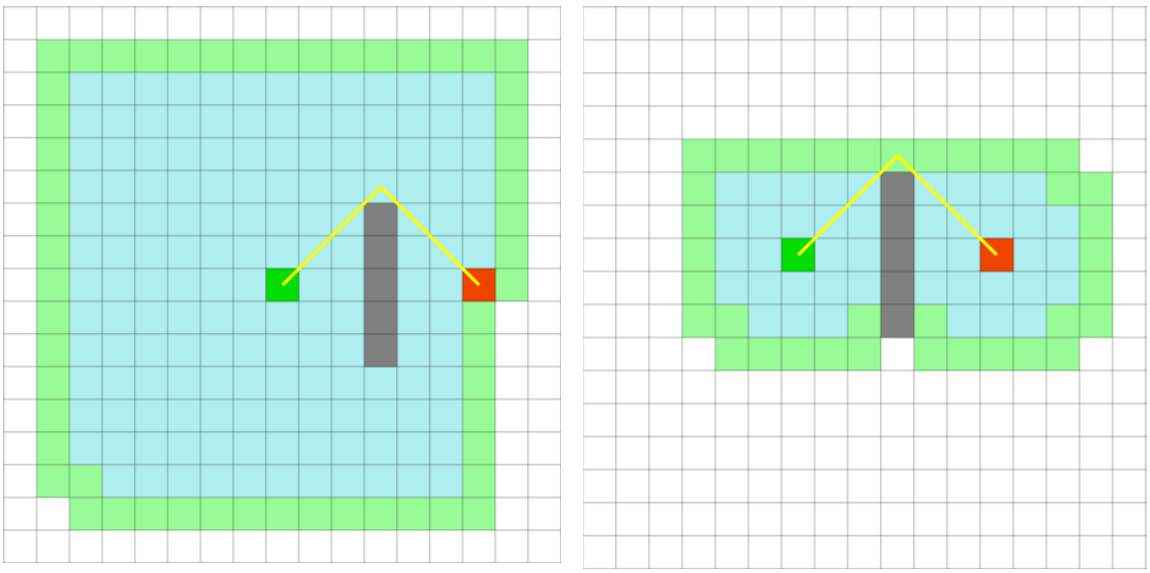


Figure 1[1]. Comparison of efficiency of BFS vs Bi-directional BFS.

[1] Image done using http://qiao.github.io/PathFinding.js/visual/

2.3.2 Dijkstra's algorithm

Dijkstra's algorithm (Dijkstra, 1959) works by expanding outwards from the start node until it does not find the goal node. To ensure expanding by the same amount in all directions, it expands the farthest node from the start node; therefore this algorithm will stumble into the goal node.

Every node is assigned a tentative distance value: starting node has value of 0 and all the others have value of infinity. Afterwards, all nodes are marked as unvisited and the starting node is set as the current node. A list containing all unvisited nodes is created.

We calculate the tentative distances of all unvisited neighbours of the current node and compare them to the currently assigned value and use the smaller one (as finding the shortest path requires that the cost (=distance) of moving from the starting node to the destination node is the lowest one possible).

The already visited node is put the closed list, which is the list of nodes which have been already visited and therefore they should not be visited again. The algorithm selects the node with the lowest tentative distance and repeats the whole process again.

When the destination node was visited, we follow the child-parent relation from the destination node to the starting node, constructing the shortest path from the start node to the destination node.

2.3.3 A*

A* algorithm (Hart, Nilsson, & Raphael, 1968), one of the best established algorithms for searching of the shortest path (Stout, 1999) is known as the star of the search algorithms.

A* combines the core of the Dijkstra's algorithm (expands the farthest nodes from the start node) and Greedy Best-First-Search's algorithm (implements a heuristic to select the node closest to the goal node).

A* uses the starting node and the destination node to find the shortest path. The starting node is marked "O" and the destination node is marked "X". The algorithm executes the search of the shortest path by starting with the starting node and then expanding to the surrounding nodes. (*Figure 2)*

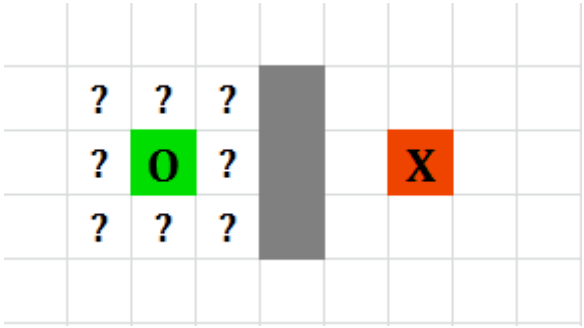


Figure 2. Initial state of A* algorithm – expansion from the starting node.

This operation continues until the destination node is found. A* keeps the track of the nodes which will be used during the search; the algorithm places the nodes to be examined in the list called Open List. At the beginning, it places the starting node to the Open List and examines all its neighbours. Afterwards it moves the starting node from the Open List to the Closed List and the neighbouring nodes to the Open List. (*Figure 3)* Closed List holds the nodes which were already visited and therefore they will not be visited again.

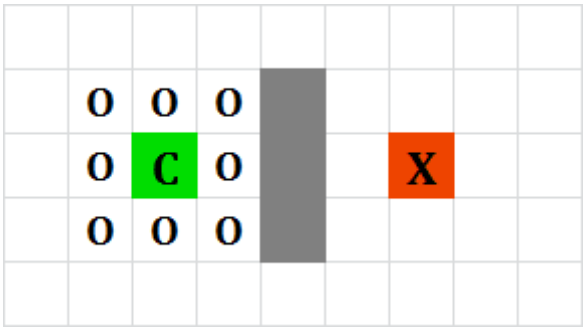


Figure 3. Insertion of the neighbouring nodes to the Open List.

This process outlined is made during one iteration of A*; however we need to know additional information – how the nodes are linked together. Despite the Open List contains the list of adjacent nodes, we need keep track of how the adjacent nodes are linked together as well, therefore the A* tracks the parent node of the each node present in the Open List. A node's parent node is the node from which we got to the current node. During the first iteration each node will have the starting node as its parent. (*Figure 4)*

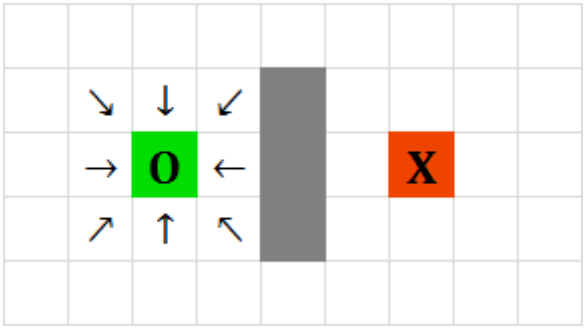


Figure 4. Starting node and its parent relation.

The A* algorithm uses the parent links to establish the path back to the starting node, when the destination node is reached. At this point the process starts over, choosing a new node to be checked from the Open List. During the first operation only one node was in the Open List, now there are eight of them; therefore to choose the next one a cost needs to be assigned to each of the nodes, *f(n)*.

In the standard terminology, cost of each node is defined by equation:

$$f(n) = g(n) + h(n)$$

where:

g(n) is the cost of getting to the node *n* from the starting node

and

h(n) is the heuristic approximation.

In order to calculate the *g(n)*, the *g(n)* of its parent will be taken and increased by either 10 in case of moving horizontally or vertically or 14 in case of moving orthogonally. These values are being used to simplify the calculations as we avoid calculating square roots and using floating point arithmetic – as the actual distance to move orthogonally is $\sqrt{1^2 + 1^2}$ or $\sqrt{2}$, which is roughly 1.414 times the cost of moving horizontally or vertically. The values are scaled by 10 otherwise the cost would remain the same - $\sqrt{2}$ rounded down is 1 – which is the same as the cost of the horizontal or vertical movement.

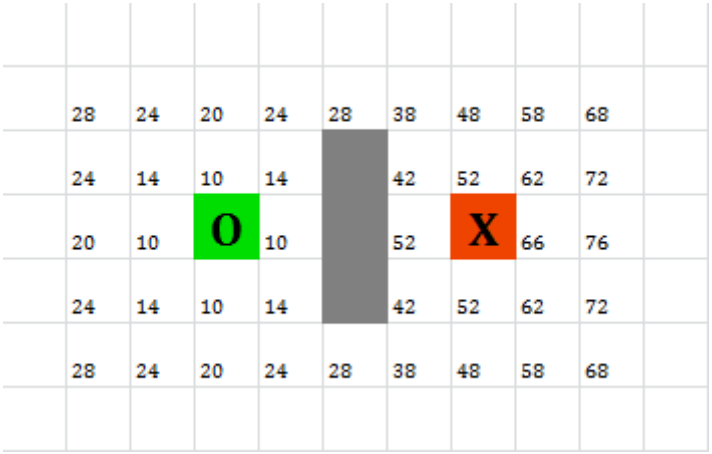


Figure 5. Example calculation of g(n) – the distance from the starting node

The algorithm selects the node with the lowest cost from the Open List, which is a Priority Queue with the nodes being sorted by its cost, meaning when the element is popped from the Queue, it is the node with the lowest cost *f(n)*.

As the search of the nodes towards the destination node continues, A* adds them to the Open List and transfers them to the Closed List when it will not need to manipulate them anymore.

When the destination node is reached, reconstruction of the shortest path is made by back-tracking the parent relations of the nodes in the Closed List. (Figure 5)

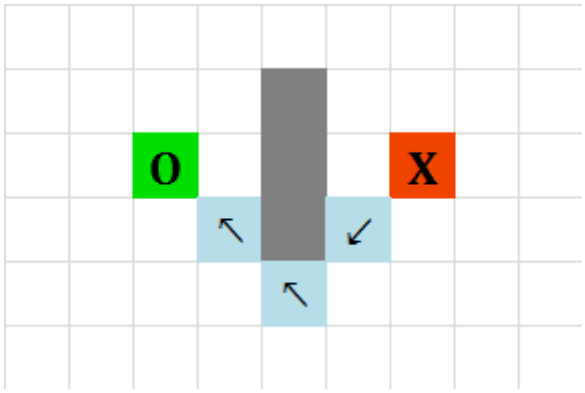


Figure 6. Construction of the shortest path upon reaching the destination following the child-parent relations.

2.3.3.1 A* heuristic

A* uses the heuristic function *h(n)* to estimate the cost from node *n* to the goal node. Because heuristic can be used to control A*'s behaviour (*Figure 6*), it is important that

the heuristic is admissible (or optimistic), meaning it never overestimates or underestimates the cost of reaching the goal. (Anderson, 2014)

If *h(n)* is always equal to the cost of moving from node *n* to the goal node, A* will only expand the best path, resulting in perfect behaviour.

If *h(n)* underestimates the cost of moving, as a result will A* expand more nodes, resulting in worse performance.

If *h(n)* overestimates the cost of moving, A* can run faster but will no longer guarantee finding the shortest path, as it may wrongly estimate not the shortest path to be actually the shortest.

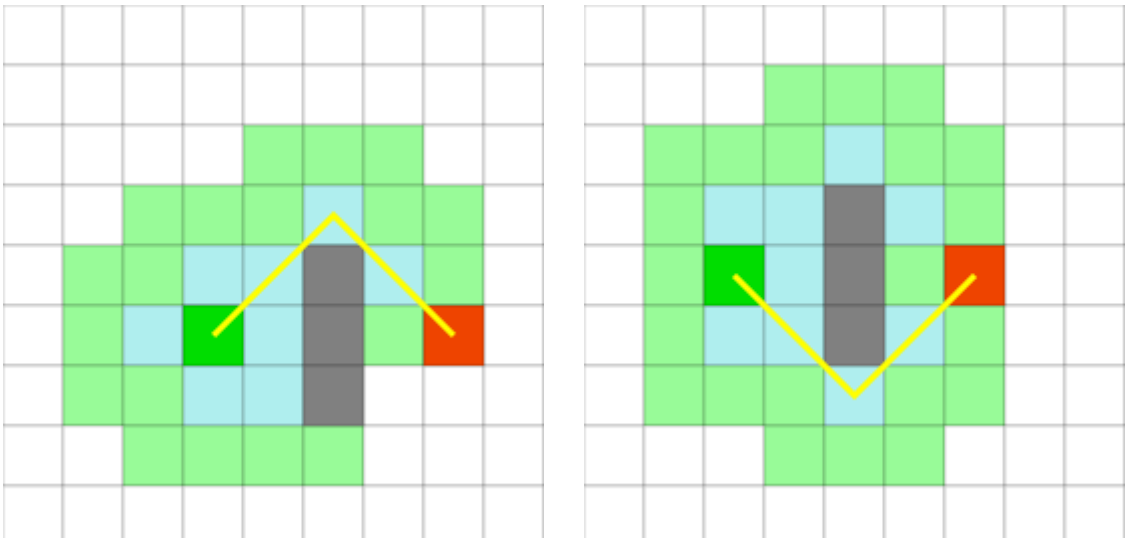


Figure 7[2]. Comparison of A* behaviour using different heuristic. Manhattan (left) vs. Euclidean (right).

2.3.3.1.1 A* heuristic – Manhattan distance

Manhattan distance is the standard heuristic used for square grids. (Patel, 2014a) It allows us to move in four directions.

Manhattan distance is defined by the equation:

$$h(n) = D * (|node.x - goal.x| + |node.y - goal.y|)$$

where:

D is the cost from moving from one node to an adjacent node.

[2] Image done using http://qiao.github.io/PathFinding.js/visual/

The advantage of this heuristic function is that it is not expensive and therefore can run faster than others. The major disadvantage is however that it tends to overestimate the cost to the goal, which results in possibility of not finding the most optimal solution.

2.3.3.1.2 A* heuristic – Euclidean distance

Euclidean distance is used when the movement in every direction is allowed.

Euclidean distance is defined by the equation:

$$h(n) = D * \sqrt{(n.x - goal.x)^2 + (n.y - goal.y)^2}$$

The Euclidean heuristic is admissible; however it usually underestimates the distance, which results in more nodes being visited as needed. Additionally, in comparison to Manhattan distance, it is more computationally expensive.

2.3.3.1.3 A* heuristic – Chebyshev distance

Chebyshev distance is usually used for grids with enabled diagonal movement.

Chebyshev distance is defined by the equation:

$$h(n) = D * max(|node.x - goal.x|, |node.y - goal.y|)$$

Chebyshev distance balances the *g(n)* and *h(n)* in the calculation of the total cost, which causes that it is a slower heuristic method than Manhattan distance.

3. Research

3.1 Research plan

In order to determine which algorithm is the most efficient, they have to be implemented in real-life conditions. As the usage and the implementation of the path-finding algorithms vary, their performance will be tested on different sets of inputs. The research plan is following:

1. Create a Python implementation of three different path-finding algorithms.
2. Record the computation times of the algorithms and the amount of the function calls made on different sets of inputs.
3. Evaluate the data collected during step 2 to determine which of the algorithms the fastest one is.

3.2 Building the application

To test the execution times of these three different algorithms, I will create a Python script, which will be capable of:

- Generating a random square maze / choosing a random map from the data set
- Measuring the computation time of all three algorithms and the number of function calls on the chosen problem set
- Keeping track of the number of computation times and amount of function calls of every algorithm

The generation and the measurement will be performed using this script.

3.2.1 Generating a random square maze

The method of generating a random square maze will be same as used in (Silver, 2005) and subsequently in (Standley, 2010; Standley & Korf, 2011): random grids of size $2^n x 2^n$ (32x32, 64x64 and 128x128) will be generated, in which each cell is an obstacle with 20% probability, as this percentage gives interesting mazes with a high probability that the destination is reachable, as where higher probabilities tend to have many separate unreachable sections (D. Silver, personal communication, October 2, 2014) (see Appendix A) and creates sufficient number of obstacles so that the path-finding is not trivial. (R. Korf, personal communication, October 22, 2014) (see Appendix B)

```
1000000000000000
0010100010010OS0
1000000010011000
00T0001001000000
0100000100000000
0010001000000000
0100000100010000
0000000000000000
1000000000000000
0000110000000010
0010000100001000
1000010100010001
0000010000010000
0100001000100110
0000001100000000
0100001100010100
```

Figure 8. Example of a random generated map (1 – obstacle, 0 – free octile, S – starting node, T – destination node)

3.2.2 Dataset of game maps

Second dataset on which the efficiency of the algorithms will be measured is the set of maps by (Sturtevant, 2012) from the following games:

- Baldur's Gate II
- Dragon Age: Origins
- StarCraft I
- Warcraft III

The maps are square-shaped as well; however their size is no longer $2^n x 2^n$ (except a few exceptions) – the maps included are of one of the following sizes: 37x37, 41x41, 49x49, 66x66, 84x84, 128x128, 194x194, 256x256, 257x257, 258x258 and 384x384.

Since artificial maps created algorithmically have different properties than maps created for particular applications (Sturtevant, 2012), it could affect the performance of the path-finding algorithms. It can be seen on Figure 8 and Figure 9 – while the obstacles in case of random generated map are placed randomly, in case of the game map they are concentrated and create one large obstacle. Therefore the usage of two datasets will compare the performance of each algorithm on a completely different problem set – the differences in the performance will be visible and distinguishable.

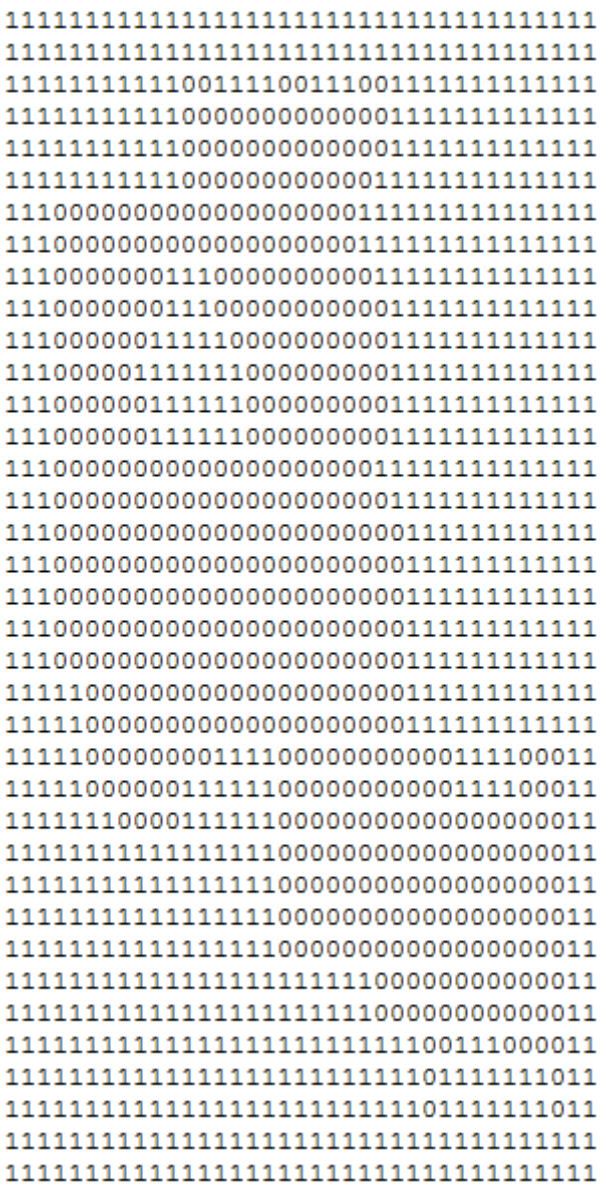


Figure 9. Example of a game map from game Dragon Age: Origins (1 – obstacle, 0 – free octile)

3.2.3 Computation time and number of function calls measurement

In order to evaluate the efficiency of the algorithms the following will be measured:

- Computation time of each algorithm
- Number of function calls of each algorithm

The number of function calls of each algorithm is measured because function call overhead is large compared to other instructions (Python Speed - Python Wiki, 2012), and with algorithms like Dijkstra, which have to expand thousands of cells in order to find the shortest path, it could affect the results. Moreover, the fewer function calls the algorithm makes, the shorter the computation time will be and CPU usage will be lower.

In case if two algorithms will have the same computation time, the amount of function calls made will be the deciding factor for the evaluation of their efficiency.

All measurements will be done using *cProfiler* library from Python's standard library. This library deterministically profiles Python applications, measuring how long and how often are certain parts of the application executed. (van Rossum, 2008)

4. Data collection and results

4.1 Computation times and number of function calls

4.1.1 Dataset of random mazes

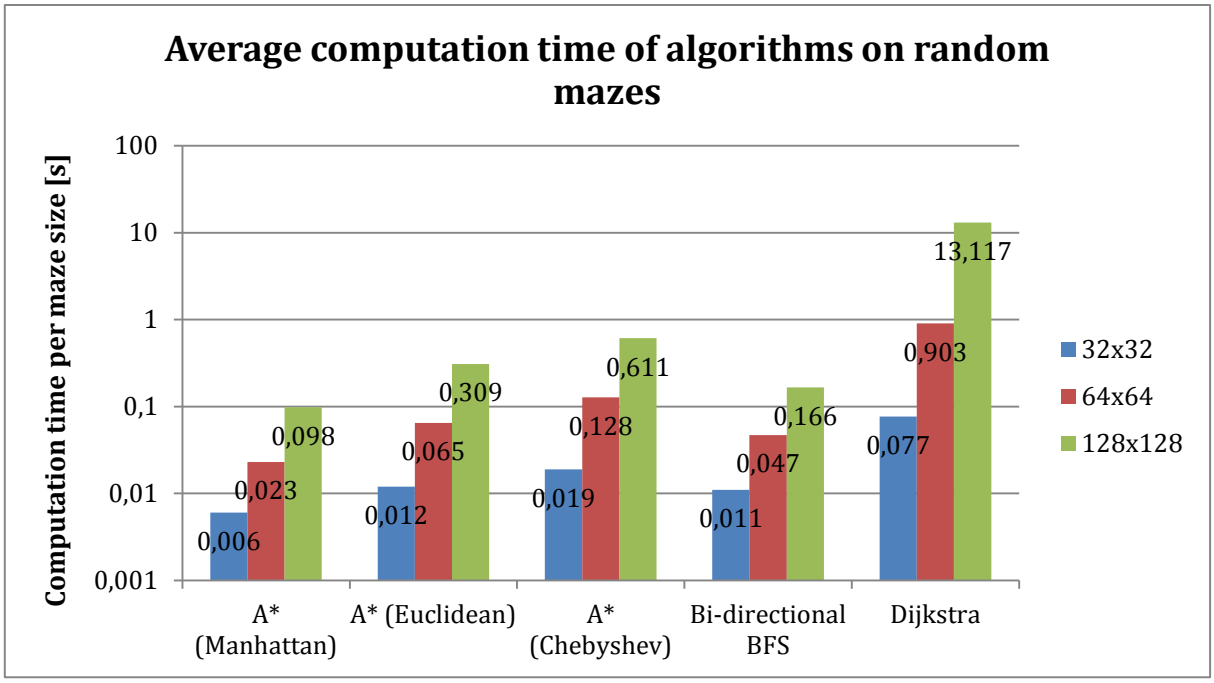
Computation times and number of function calls were measured 120 times per each maze size (32x32, 64x64 and 128x128), together for 360 times. Before computation the priority of the application was set to real-time, as a real-time priority thread or process can never be pre-empted by timer interrupts and runs at a higher priority than any other thread in the system (Osteman, 2009), which results in that the measurement is as accurate as possible – there are no disruptions.

Average computation times of algorithms on random mazes			
Algorithm	**32x32 [s]**	**64x64 [s]**	**128x128 [s]**
A* (Manhattan)	0.006	0.023	0.098
A* (Euclidean)	0.012	0.065	0.309
A* (Chebyshev)	0.019	0.128	0.611
Bi-directional BFS	0.011	0.047	0.166
Dijkstra	0.077	0.903	13.117

Table 1. Average computation times of algorithms on random mazes

Average number of function calls of algorithms on random mazes			
Algorithm	**32x32**	**64x64**	**128x128**
A* (Manhattan)	3557	13617	58004
A* (Euclidean)	6364	35245	167525
A* (Chebyshev)	10342	69834	331793
Bi-directional BFS	5394	24278	83120
Dijkstra	12909	52090	201258

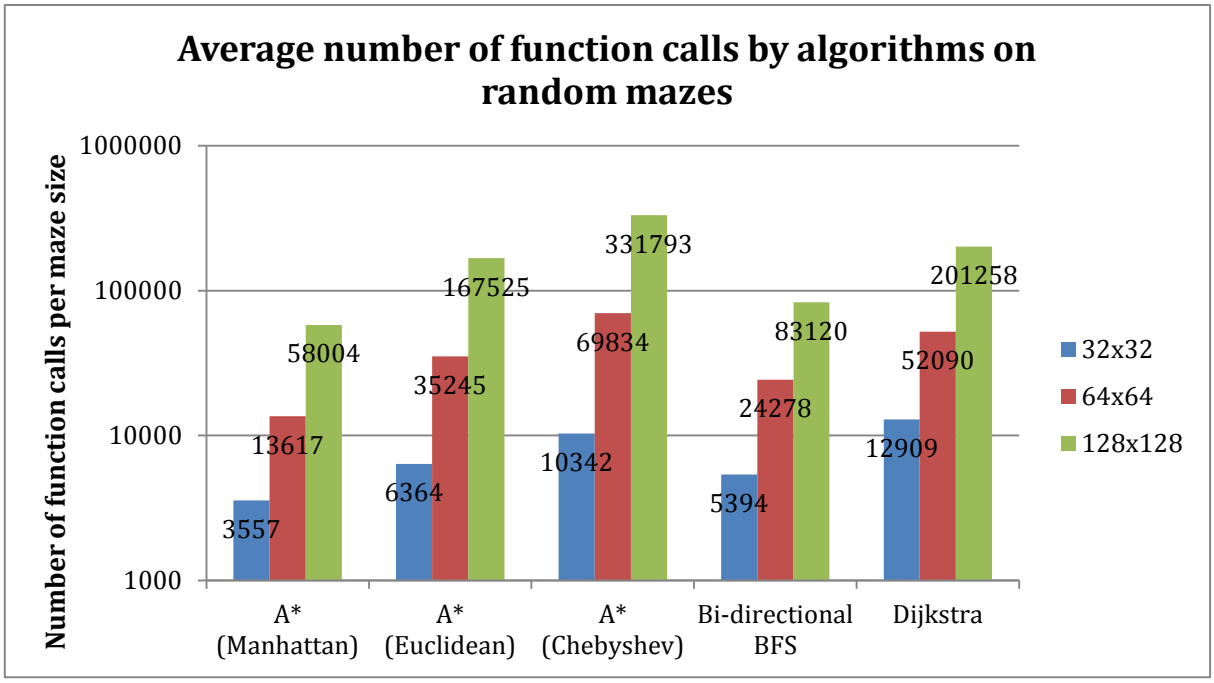
Table 2. Average number of function calls of algorithms on random mazes



Graph 1. Average computation time of algorithms on random mazes

A* (Manhattan) was the fastest algorithm, followed by Bi-directional BFS, A* (Euclidean), A* (Chebyshev) and Dijkstra's algorithm. This was expected as the usage of heuristics in A* allows the algorithm to move in the direction of the destination node, whereas Bi-directional BFS and Dijkstra's algorithm have to expand thousands of nodes in order to find the destination node which is not efficient. However, when the size of the maze was 128x128, Dijkstra's algorithm made fewer function calls than A* (Chebyshev), yet was slower. It can be attributed to the fact that Chebyshev heuristic has not admissible behaviour anymore, however usage of heuristic guarantees that the search progresses towards the destination node, therefore the better computation time.

As can be observed from the graph, usage of different heuristic functions resulted in difference in computation times, which means that the usage of the other heuristic functions is not admissible anymore and usage of A* with Manhattan heuristic is preferred on random mazes. Bi-directional BFS was faster than Dijkstra due to searching for the shortest-path both from start and destination nodes, unlike Dijkstra's expand-until-destination-not-found approach.



Graph 2. Average number of function calls by algorithms on random mazes

A* (Manhattan) was the algorithm which expanded the least amount of nodes, followed by Bi-directional BFS, Dijkstra's algorithm, A* (Euclidean) and A* (Chebyshev). The efficiency of A* (Manhattan) can be attributed to optimal behaviour of the heuristic function on this dataset, where as other heuristic function clearly over-estimated the distance to the destination node and as a result were slower than less-sophisticated algorithms such as Bi-directional BFS or Dijkstra's algorithm.

4.1.2 Dataset of game maps

18 maps of size smaller than 512 x 512 were chosen from dataset and first 20 scenarios (pre-defined sets of start and goal node coordinates) were used, together for 360 times as on dataset of random mazes. Priority of the application was set to real-time again.

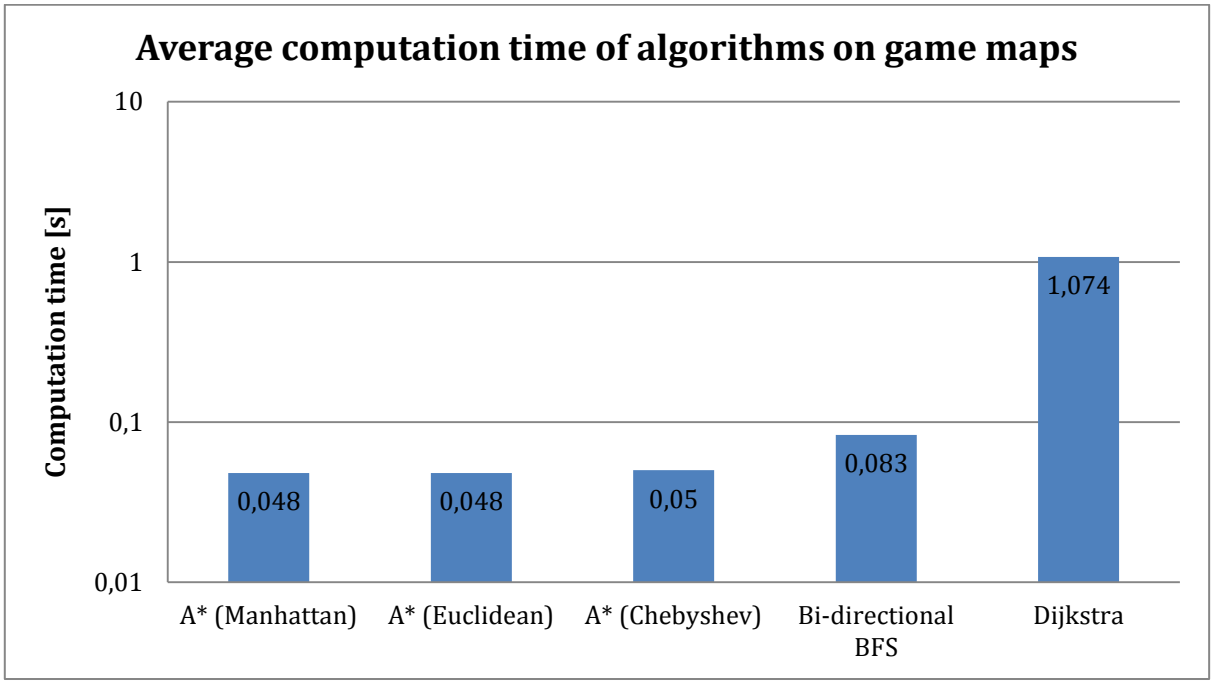
Average computation times of algorithms on game maps	
Algorithm	**Time [s]**
A* (Manhattan)	0.048
A* (Euclidean)	0.048
A* (Chebyshev)	0.050
Bi-directional BFS	0.083

Dijkstra	1.074

Table 3. Average computation times on dataset of game maps

Average number of function calls by algorithms on game maps	
Algorithm	**Number of function calls**
A* (Manhattan)	38867
A* (Euclidean)	38963
A* (Chebyshev)	39217
Bi-directional BFS	38933
Dijkstra	213772

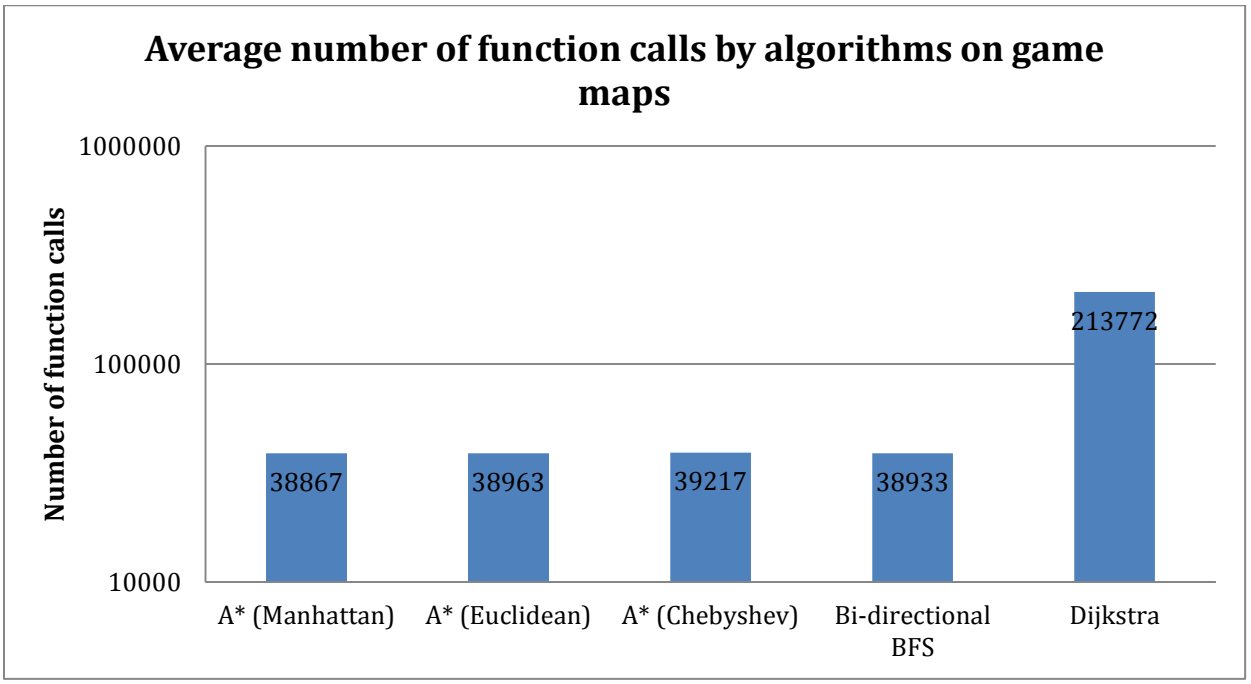
Table 4. Average number of function calls by algorithms on dataset on game maps



Graph 3. Average computation time of algorithms on game maps

As shown in Graph 3, A* (Manhattan) was the fastest path-finding algorithm followed by A* (Euclidean), A* (Chebyshev), Bi-directional BFS and Dijkstra's algorithm. Contrary to the results in the previous section (4.1.1 Dataset of random mazes), A* algorithm was the fastest one regardless of the heuristic used. It can be attributed to the different

properties of the maps in comparison to the artificially generated ones. (Sturtevant, 2012)



Graph 4. Average number of function calls by algorithms on game maps

Similarly as in Graph 3, A* (Manhattan) was the algorithm with the least number of function calls, followed by Bi-directional BFS, A* (Euclidean), A* (Chebyshev) and Dijkstra's algorithm. In conjunction with the computation time of the algorithms on this dataset, A* could be chosen with any of those heuristic functions, however if the path-finding process would be critical and real-time, A* (Manhattan) would be the best suited algorithm, as it made less function calls than A* (Euclidean), despite the computation taking the same time on this dataset.

5. Evaluation and conclusion

The aim of this essay was to compare the efficiency of three different path-finding algorithms to find the shortest path in square-sized maze. The efficiency was tested on two datasets:

1. Square-sized mazes of size $2^n x 2^n$, ranging from (32x32 to 128x128), as in (Silver, 2005; Standley, 2010; Standley & Korf, 2011)
2. Square-sized game maps from (Sturtevant, 2012) dataset

Designing the algorithms took a significant amount of time, as knowledge of the graph theory was required and algorithms were tested multiple times throughout the development to ensure their proper behaviour.

Comparing the efficiency of three different path-finding algorithms to find the shortest path in a square-shaped maze.

The three algorithms were:

- A*, path-finding algorithm using heuristic (Manhattan, Euclidean and Chebyshev)
- Bi-directional BFS, path-finding algorithm using expansion from both start and destination node
- Dijkstra's algorithm, path-finding algorithm expanding into every possible direction

Regarding random mazes, A* (Manhattan) was the fastest algorithm with a time of **0.006, 0.012 and 0.019 seconds per map** of respective maze size. (32x32, 64x64, 128x128) A* (Manhattan) was the algorithm which made the least number of function calls as well, **3557, 13617 and 58004 function calls per map** of respective maze size.

Regarding game maps, A* (Manhattan) was the fastest algorithm with a time **of 0.048 seconds per map**, requiring **38867 function calls per map** to find the shortest path.

From the results it can be concluded that A* algorithm with Manhattan heuristic is the most-efficient algorithm regarding both random mazes and game maps – therefore its usage is highly preferred to the usage of other algorithms.

Path-finding is an important problem in many various application areas – not only computer games, but also for navigation between cities and navigation of blind people – which has real-life significance. It is worth noting that the results of this Extended Essay apply to any type of path-finding on a grid – either square-shaped or rectangular-shaped, as the problem can be abstracted furthermore.

5.1 Future study

As one of the possibilities for the future study, multiple-destination path-finding could be implemented – calculating the shortest route among the given destinations, for example between cities on a route or shops in a shopping centre.

There is a possibility of examining the efficiency of other path-finding algorithms as well, such as D*, Fringe Search or IDA* (Iterative Deepening A*).

Lastly, to gain a wider spectrum of data, maps of larger sizes could be used.

Bibliography

Anderson, C. (2014, September 14). *CS440: Introduction to Artificial Intelligence*. Retrieved November 11, 2014, from CS440: Introduction to Artificial Intelligence: http://www.cs.colostate.edu/~anderson/cs440/index.html/doku.php?id=notes:week4c

Björnsson, Y., Enzenberger, M., Holte, R., Schaejfer, J., & Yap, P. (2003). Comparison of different grid abstractions for pathfinding on maps. *Proceedings of the 18th international joint conference on Artificial intelligence* (pp. 1511-1512). San Francisco: Morgan Kaufmann Publishers Inc.

Carsten, J., Ferguson, D., Rankin, A., & Stentz, A. (2007, March 3-10). Global Path Planning on Board the Mars Exploration Rovers. *2007 IEEE Aerospace Conference*, pp. 1-11.

Davis, I. (2000). Warp Speed: Path Planning for Star Trek: Armada. In *Artificial Intelligence and Interactive Entertainment* (pp. 18-21). Menlo Park, CA, USA: AAAI Press.

Dijkstra, E. (1959, December 1). A note on two problems in connexion with graphs. *Numerische Mathematik, 1*(1), pp. 269-271.

Hart, P., Nilsson, N., & Raphael, B. (1968, July). A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics, 4*(2), pp. 100-107.

Jacob, R., Marathe, M. V., & Nagel, K. (1999, December 31). A Computational Study of Routing Algorithms for Realistic Transportation Networks. *ACM Journal of Experimental Algoriths, 4*(6).

Lee, J.-Y., & Yu, W. (2009, October 10-15). A coarse-to-fine approach for fast path finding for mobile robots. *IROS*, pp. 5414-5419.

Millington, I., & Funge, J. (2009). Artificial Intelligence for Games. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Moore, E. (1959). The shortest path through a maze. In H. Aiken (Ed.), *Proceedings of an International Symposium on the Theory of Switching* (pp. 285-292). Cambridge: Harvard University Press.

Osteman, L. (2009, November 3). *What is the 'realtime' setting for, for process priority?* Retrieved November 29, 2014, from StackOverflow: http://stackoverflow.com/a/1665003/825916

Patel, A. (2014a). *Heuristics*. Retrieved October 16, 2014, from Amit's Thoughts on Pathfinding: http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html

Patel, A. (2014b). *Map representations*. Retrieved November 17, 2014, from Amit's Thoughts on Pathfinding: http://theory.stanford.edu/~amitp/GameProgramming/MapRepresentations.html

Pohl, I. (1969). *Bi-directional and heuristic search in path problems.* Stanford, CA, USA: Stanford University.

Python Speed - Python Wiki. (2012, January 31). Retrieved October 25, 2014, from Python Wiki: https://wiki.python.org/moin/PythonSpeed

Silver, D. (2005). Cooperative Pathfinding. *Proceedings of the first artificial intelligence and interactive digital entertainment conference* (pp. 117-122). Marina Del Ray, California, USA: The AAAI Press.

Standley, T. (2010). Finding Optimal Solutions to Cooperative Pathfinding Problems. *AAAI Conference on Artificial Intelligence*, (pp. 173-178).

Standley, T., & Korf, R. (2011). Complete Algorithms for Cooperative Pathfinding Problems. *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence* (pp. 668-673). Barcelona, Catalania, Spain: AAAI Press.

Stout, B. (1999, February 12). *Gamasutra - Smart Move: Intelligent Path-Finding*. Retrieved October 18, 2014, from http://www.gamasutra.com/view/feature/131724/smart_move_intelligent_.php

Sturtevant, N. (2007). Memory-Efficient Abstractions for Pathfinding. In *Proceedings of the Third Artificial Intelligence and Interactive Digital Entertainment Conference* (pp. 31-36). Stanford, CA, USA: The AAAI Press.

Sturtevant, N. (2012, May 3). Benchmarks for Grid-Based Pathfinding. *Transactions on Computational Intelligence and AI in Games, 4*(2), pp. 144-148.

van Rossum, G. (2008, December 3). *27.4. The Python Profilers — Python 3.4.2 documentation*. Retrieved October 8, 2014, from Python 3.4.2 documentation: https://docs.python.org/3.4/library/profile.html

Appendices

Appendix A – E-mail communication with David Silver

On Thu, Oct 2, 2014 at 7:48 AM, Jaromir Latal <latal.jaromir@gmail.com> wrote:
Dear Mr. Silver,

My name is Jaromír Látal and I am a student of the International Baccalaureate Diploma in Bratislava, Slovakia.
I am in the last year before going to university and I am writing a big assignment called Extended Essay on Pathfinding for my Computer Science class.

I stumbled upon your paper "Cooperative Pathfinding Problems" and I would like to ask you following questions:

1. Why were only the grids of size 32x32 used? Would not it be better to use larger sized grids in order to check the algorithms efficiency on them as well?

2. What is the reason for generating the grid with 20% probability for cell being an obstacle?

Thanks in advance for your answers.

Best regards,
Látal Jaromír

On Thu, Oct 2, 2014 at 10:50 AM, David Silver <davidsilver@google.com> wrote:

Hi Jaromir

1. 32x32 produced interesting mazes but could still run at real-time (many years ago when I ran these experiments).
2. 20% gives interesting mazes with a high probability of destination being reachable, higher probabilities tend to have many separate sections that are not always reachable.

Dave

Appendix B – E-mail communication with Richard Korf

On Thu, Oct 2, 2014 at 7:48 AM, Jaromir Latal <latal.jaromir@gmail.com> wrote:
Dear Dr. Korf,

My name is Jaromír Látal and I am a student of the International Baccalaureate Diploma in Bratislava, Slovakia.
I am in the last year before going to university and I am writing a big assignment called Extended Essay on Pathfinding for my Computer Science class.

I stumbled upon your paper "Complete Algorithms for Cooperative Pathfinding Problems" and I would like to ask you following questions:

1. Why were only the grids of size 32x32 used? Would not it be better to use larger sized grids in order to check the algorithms efficiency on them as well?

2. What is the reason for generating the grid with 20% probability for cell being an obstacle?

Thanks in advance for your answers.

Best regards,
Látal Jaromír

On Wed, Oct 22, 2014 at 3:18 AM, Richard E. Korf <korf@cs.ucla.edu> wrote:
Jaromir,
Sorry for the delay in my response, but I've been swamped lately. I didn't do most of the research on that paper, but I'll try to answer your questions.

1) Regarding the 32x32 grids, that was probably the largest size where we could get a reasonable amount of data in a reasonable time.

2) There is no particular reason for 20% obstacles. You want sufficient obstacles so that the pathfinding is not trivial, but not so many that the space becomes disconnected.
-rich

Appendix C – Source code

C.1 – astar.py (original author: Xueqiao Xu)

```
from constants import *
from graph import is_walkable, InvalidMap
from heapq import *
from math import hypot


class _Node(object):

    """This class works as the container of the nodes' info.
    Refer to the class AStar's documents for the meanings
    of g, h and f.
    """
    __slots__ = ['parent', 'status', 'f', 'g', 'h']

    def __init__(self):
        self.status = None
        self.parent = None
        self.g = None
        self.h = None
        self.f = None


class _QNode(object):

    """This class is used for storing the coordinates of the
    opened nodes in the open list. It provides a comparison
    method used by the priority queue.
    """
    __slots__ = ['pos', '_nodes']

    def __init__(self, pos, nodes):
        """Creates a new instance of QNode.
        :Parameters:
            pos : (x, y)
                the coordinate of this node.
            nodes: [str]
                a reference to AStar.nodes.
        """
        self.pos = pos
        self._nodes = nodes

    def __lt__(self, other):
        """Used by heapq for maintaining the priority queue.
        """
        x, y = self.pos
        ox, oy = other.pos
        if self._nodes[y][x].f == self._nodes[oy][ox].f:
            return self._nodes[y][x].h < self._nodes[oy][ox].h
        else:
            return self._nodes[y][x].f < self._nodes[oy][ox].f


class AStar(object):

    """Each node has three main properties:
        F, G and H
```

```
      where
        F = G + H
        G = the cost from source to this node
        H = estimated cost from this node to the destination.
When we explore a node, we put all its adjacent nodes into a open
list. At each loop, we take one node from the open list for next
inspection. The node with smaller F will have higher priority.
There are serveral ways to calculate the value of H:
    let x = horizontal distance between this node and destination
        y = vertical distance between this node and destination
    1. Manhattan: h = x + y
    2. Euclidean: h = hypot(x, y)
    3. Chebyshev: h = max(x, y)
    and so on.
    (Of course you can define your own heuristic methods.)
also, you can give H a weight, i.e.
    F = G + W * H
The higher the W value is, The more important the heuristic will
be in this algorithm.
"""

def __init__(self, raw_graph, heuristic=MANHATTAN):
    """Create a new instance of A* path finder.

    :Parameters:
        raw_graph : str
            A multi-line string representing the graph.
            example:
            s = '''
                S000
                1110
                T000
                '''
        heuristic :
            Currently three types of heuristic are supported,
            namely MANHATTAN, EUCLIDEAN and CHEBYSHEV
    """
    self.graph = raw_graph.split()
    self.size = len(self.graph)

    # determine heuristic function
    self.h_list = {MANHATTAN: self._manhattan,
                   EUCLIDEAN: self._euclidean,
                   CHEBYSHEV: self._chebyshev}
    if heuristic not in self.h_list:
        self.heuristic = MANHATTAN
    else:
        self.heuristic = heuristic
    self.h_func = self.h_list[self.heuristic]

    self.source = None
    self.target = None
    self.path = []

    # 2D array of nodes
    self.nodes = [[_Node()
                   for x in range(self.size)]
                  for y in range(self.size)]

    # get source and target coordinates
    for y in range(self.size):
```

```
            for x in range(self.size):
                if self.graph[y][x] == SOURCE:
                    self.source = (x, y)
                elif self.graph[y][x] == TARGET:
                    self.target = (x, y)

        # guarantee that both source and target is present on the graph
        if not all((self.source, self.target)):
            raise InvalidMap("No source or target given")

        # a priority queue holding the coordinates waiting
        # for inspection
        self.open_list = []

    def step(self, record=None):
        """Starts the computation of the shortest path.
        *Note* :
            This function works as a generator, a yield statement
            is appended at the bottom of each loop to make this
            function capable of being executed step by step.

            If you does not want the step feature, simply delete
            the yield statements.

        :Parameters:
            record : deque or list
                if a queue is specified, a record of each operation
                (OPEN, CLOSE, etc) will be pushed into the queue.
        """
        # add the source node into the open list
        sx, sy = self.source
        self.nodes[sy][sx].g = 0
        self.nodes[sy][sx].f = 0
        self.open_list.append(_QNode(self.source, self.nodes))
        self.nodes[sy][sx].status = OPENED

        # while the open list is not empty
        while self.open_list:

            # get the node with lowest F from the heap
            x, y = heappop(self.open_list).pos
            self.nodes[y][x].status = CLOSED
            if record is not None:
                record.append(('CLOSE', (x, y)))

            # if the node is the target, reconstruct the path
            # and break the loop
            if (x, y) == self.target:
                self._retrace()
                break

            # inspect the horizontal and vertical adjacent nodes
            for i in range(len(XOFFSET)):

                # next x and y
                nx = x + XOFFSET[i]
                ny = y + YOFFSET[i]

                # if the next coordinate is walkable, inspect it.
                if is_walkable(nx, ny, self.size, self.graph):
                    self._inspect_node((nx, ny), (x, y), False, record)
```

```
                # further investigate the diagonal nodes
                nx1 = x + DAXOFFSET[i]
                ny1 = y + DAYOFFSET[i]
                nx2 = x + DBXOFFSET[i]
                ny2 = y + DBYOFFSET[i]
                npos = ((nx1, ny1), (nx2, ny2))
                for nx, ny in npos:
                    if is_walkable(nx, ny, self.size, self.graph):
                        self._inspect_node((nx, ny), (x, y),
                                           True, record)
        yield

    def _retrace(self):
        """After the search completes, this method will be called to
        reconstruct the path according to the nodes' parents.
        """
        self.path = [self.target]
        while self.path[-1] != self.source:
            x, y = self.path[-1]
            self.path.append(self.nodes[y][x].parent)
        self.path.reverse()

    def _inspect_node(self, node_pos, parent_pos, diagonal, record):
        """Push the node into the open list if this node is not
        in the open list. Otherwise, if the node can be accessed
        with a lower cost from the given parent position, update
        its parent and cost, then heapify the open list.
        """
        x, y = node_pos
        px, py = parent_pos

        if self.nodes[y][x].status != CLOSED:
            if self.nodes[y][x].status != OPENED:
                self.nodes[y][x].status = OPENED
                if record is not None:
                    record.append(('OPEN', (x, y)))
                self._try_update((x, y), (px, py), diagonal, record)
                heappush(self.open_list,
                         _QNode((x, y), self.nodes))
            else:
                if self._try_update((x, y), (px, py), diagonal,
                                    record):
                    heapify(self.open_list)

    def _try_update(self, node_pos, parent_pos, diagonal, record):
        """Try to update the node's info with the given parent.
        If this node can be accessed with the given parent with lower
        G cost, this node's parent, G and F values will be updated.
        :Return:
            updated : bool
                whether this node's info has been updated.
        """
        x, y = node_pos
        px, py = parent_pos
        dd = DDIST if diagonal else DIST  # whether is diagonal
        ng = self.nodes[py][px].g + dd  # next G value
        node = self.nodes[y][x]

        if node.g is None or ng < node.g:
            # if this node has not been opened or
```

```
            # it can be accessed with lower cost
            node.parent = (px, py)
            node.g = ng
            node.h = self._calc_h((x, y))
            node.f = node.g + node.h

            if record is not None:
                record.append(('VALUE', ('g', (x, y), node.g)))
                record.append(('VALUE', ('h', (x, y), node.h)))
                record.append(('VALUE', ('f', (x, y), node.f)))
                record.append(('PARENT', ((x, y), (px, py))))
            return True
        return False

    def _calc_h(self, pos):
        """Caculate the H value of the node.
        """
        dx = abs(pos[0] - self.target[0])
        dy = abs(pos[1] - self.target[1])
        return self.h_func(dx, dy)

    def _manhattan(self, dx, dy):
        return (dx + dy) * SCALE

    def _euclidean(self, dx, dy):
        return int(hypot(dx, dy) * SCALE)

    def _chebyshev(self, dx, dy):
        return max(dx, dy) * SCALE
```

C.2 – bidirbfs.py (original author: Xueqiao Xu)

```
from graph import is_walkable, InvalidMap
from heapq import *
from constants import *


class _Node(object):

    """This class works as the container of the nodes' info.
    """

    def __init__(self, status):
        self.g = None  # cost from source
        self.h = None  # cost from target
        self.parent = None
        self.visited_by = None
        self.status = status


class BiDirBFS(object):

    """ Bi-Directional Breadth-First-Search.
    Explores the map simultaneously from source and target. When the two
    trees meet, a shortest path is found.

    *NOTE* This class is designed for solving graphs with equal
    weighted edges. That is to say, on graphs with various weights,
    this algorithm doesn't gurantee to find the shortest path.
    """

    def __init__(self, raw_graph):
        """Create a new instance of Bi-Directional Breadth-First-Search
        path finder.

        :Parameters:
            raw_graph : str
                A multi-line string representing the graph.
                example:
                s = '''
                    S000
                    1110
                    T000
                    '''
        """
        self.graph = raw_graph.split()
        self.size = len(self.graph)
        self.source = None
        self.target = None
        self.path = []
        self.success = False

        # nodes grid
        self.nodes = [[_Node(self.graph[y][x])
                       for x in range(self.size)]
                      for y in range(self.size)]

        # get source and target coordinates
        for y in range(self.size):
            for x in range(self.size):
                if self.graph[y][x] == SOURCE:
```

```
                    self.source = (x, y)
                elif self.graph[y][x] == TARGET:
                    self.target = (x, y)
        if not all((self.source, self.target)):
            raise InvalidMap('No source or target given')

        self.queue_source = []
        self.queue_target = []

    def step(self, record=None):
        """Starts the computation of shortest path.
        :Parametes:
            record : deque
                if a queue is specified, a record of each operation
                (OPEN, CLOSE, etc) will be pushed into the queue.
        """
        # push the source node into the source queue and the
        # target node into the target queue
        sx, sy = self.source
        tx, ty = self.target
        self.queue_source.append((0, self.source))
        self.queue_target.append((0, self.target))
        self.nodes[sy][sx].g = 0
        self.nodes[ty][tx].h = 0
        self.nodes[sy][sx].visited_by = CSOURCE
        self.nodes[ty][tx].visited_by = CTARGET

        # while both source queue and target queue is not empty
        # expand them.
        while self.queue_source and self.queue_target and \
                not self.success:
            self._expand_source(record)
            if self.success:
                break
            yield
            self._expand_target(record)
            yield
        yield

    def _expand_source(self, rec):
        """Searches from the source. Until it meets a node which
        has been visited by the other tree.
        """
        # take the first node from the source queue
        v, (x, y) = heappop(self.queue_source)
        node = self.nodes[y][x]

        diagonal_can = []  # stores the diagonal positions that can be
accessed

        if rec is not None:
            rec.append(('CLOSE', (x, y)))

        # inspect horizontally and vertically adjacent nodes
        for i in range(len(XOFFSET)):
            nx = x + XOFFSET[i]
            ny = y + YOFFSET[i]
            if is_walkable(nx, ny, self.size,
                           self.graph):
                # if this node can be accessed, then then correponding
                # diagonal node can be accessed.
```

```
                diagonal_can.append(i)

                nxt_node = self.nodes[ny][nx]
                # if this node has been visited by source queue before,
                # then there's no need to inspect it again.
                if nxt_node.visited_by == CSOURCE:
                    continue

                # if this node has been visited by *target* queue.
                # Then a path from source to target exists.
                # Reconstructs the path and return.
                if nxt_node.visited_by == CTARGET:
                    if rec:
                        rec.append(('CLOSE', (nx, ny)))
                    self._retrace((x, y), (nx, ny))
                    self.success = True
                    return

                # mark this node and update its info, then push the node
                # into the source queue
                nxt_node.visited_by = CSOURCE
                nxt_node.g = node.g + DIST
                nxt_node.parent = (x, y)
                heappush(self.queue_source, (nxt_node.g, (nx, ny)))

                if rec is not None:
                    rec.append(('OPEN', (nx, ny)))
                    rec.append(('VALUE', ('g', (nx, ny), nxt_node.g)))
                    rec.append(('PARENT', ((nx, ny), (x, y))))

        # further investigate the diagonal nodes, the procedure is
identical
        # with above
        for i in diagonal_can:
            nx1 = x + DAXOFFSET[i]
            ny1 = y + DAYOFFSET[i]
            nx2 = x + DBXOFFSET[i]
            ny2 = y + DBYOFFSET[i]
            npos = ((nx1, ny1), (nx2, ny2))
            for nx, ny in npos:
                if is_walkable(nx, ny, self.size, self.graph) and \
                        self.nodes[ny][nx].visited_by != CSOURCE:
                    nxt_node = self.nodes[ny][nx]
                    if nxt_node.visited_by == CTARGET:
                        if rec:
                            rec.append(('CLOSE', (nx, ny)))
                        self._retrace((x, y), (nx, ny))
                        self.success = True
                        return
                    nxt_node.visited_by = CSOURCE
                    nxt_node.g = node.g + DDIST
                    nxt_node.parent = (x, y)
                    heappush(self.queue_source, (nxt_node.g, (nx, ny)))
                    if rec is not None:
                        rec.append(('OPEN', (nx, ny)))
                        rec.append(('VALUE', ('g', (nx, ny),
                                              nxt_node.g)))
                        rec.append(('PARENT', ((nx, ny), (x, y))))

    def _expand_target(self, rec):
        """Searches from the target. Until it meets a node which
```

```
        has been visited by the other tree.
        """
        # the procedure is identical with _expand_source.
        v, (x, y) = heappop(self.queue_target)
        node = self.nodes[y][x]
        diagonal_can = []
        if rec is not None:
            rec.append(('CLOSE', (x, y)))
        for i in range(len(XOFFSET)):
            nx = x + XOFFSET[i]
            ny = y + YOFFSET[i]
            if is_walkable(nx, ny, self.size, self.graph):
                diagonal_can.append(i)
                if self.nodes[ny][nx].visited_by == CTARGET:
                    continue
                nxt_node = self.nodes[ny][nx]
                if nxt_node.visited_by == CSOURCE:
                    if rec:
                        rec.append(('CLOSE', (nx, ny)))
                    self._retrace((nx, ny), (x, y))
                    self.success = True
                    return
                nxt_node.visited_by = CTARGET
                nxt_node.h = node.h + DIST
                nxt_node.parent = (x, y)
                heappush(self.queue_target, (nxt_node.h, (nx, ny)))
                if rec is not None:
                    rec.append(('OPEN', (nx, ny)))
                    rec.append(('VALUE', ('h', (nx, ny),
                                          nxt_node.h)))
                    rec.append(('PARENT', ((nx, ny), (x, y))))

        # further investigate the diagonal nodes
        for i in diagonal_can:
            nx1 = x + DAXOFFSET[i]
            ny1 = y + DAYOFFSET[i]
            nx2 = x + DBXOFFSET[i]
            ny2 = y + DBYOFFSET[i]
            npos = ((nx1, ny1), (nx2, ny2))
            for nx, ny in npos:
                if is_walkable(nx, ny, self.size, self.graph) and \
                        self.nodes[ny][nx].visited_by != CTARGET:
                    nxt_node = self.nodes[ny][nx]
                    if nxt_node.visited_by == CSOURCE:
                        if rec:
                            rec.append(('CLOSE', (nx, ny)))
                        self._retrace((nx, ny), (x, y))
                        self.success = True
                        return
                    nxt_node.visited_by = CTARGET
                    nxt_node.h = node.h + DDIST
                    nxt_node.parent = (x, y)
                    heappush(self.queue_target, (nxt_node.h, (nx, ny)))
                    if rec is not None:
                        rec.append(('OPEN', (nx, ny)))
                        rec.append(('VALUE', ('h', (nx, ny),
                                              nxt_node.h)))
                        rec.append(('PARENT', ((nx, ny), (x, y))))

    def _retrace(self, s_pos, t_pos):
        """This method will be called when the two search trees meet.
```

```
    Since the two trees have different directions, the path must
    be reconstructed seperatedly and then combined.
    """
    s_path = [s_pos]
    t_path = [t_pos]

    while s_path[-1] != self.source:
        x, y = s_path[-1]
        s_path.append(self.nodes[y][x].parent)

    while t_path[-1] != self.target:
        x, y = t_path[-1]
        t_path.append(self.nodes[y][x].parent)

    s_path.reverse()
    self.path = s_path + t_path
```

C.3 – constants.py (original author: Xueqiao Xu)

```
XOFFSET = (0, 1, 0, -1)
YOFFSET = (-1, 0, 1, 0)
DAXOFFSET = (1, 1, -1, -1)
DAYOFFSET = (-1, 1, 1, -1)
DBXOFFSET = (-1, 1, 1, -1)
DBYOFFSET = (-1, -1, 1, 1)

NORMAL = '0'
BLOCKED = '1'
SOURCE = 'S'
TARGET = 'T'

OPENED = 'P'
CLOSED = 'C'

SCALE = 10
DIST = 10
DDIST = 14  # diagonal distance
INF = int(1e9)

MANHATTAN = 0
EUCLIDEAN = 1
CHEBYSHEV = 2

OSOURCE = 1
OTARGET = 2
CSOURCE = 1
CTARGET = 2
```

C.4 – dijkstra.py (original author: Xueqiao Xu)

```
from constants import *
from graph import make_graph
from heapq import *


class Dijkstra(object):

    """This class is designed for solving general graphs without
    negative weighted edges, not limited to grid maps.
    """

    def __init__(self, graph, source, target):
        """Create a new instance of Dijkstra path finder.

        :Parameters:
            graph : {nodeid1: {nodeid2: dist, ... }, ... }
                The graph is in adjacency list representation.
                The nodeid can be any hashable object.
                Sample graphs are as follows:
                    graph = {(1, 2): {(2, 2): 1, (1, 3): 1},
                             (2, 2): {(1, 2): 1},
                             (1, 3): {(1, 2): 1}}
                or
                    graph = {'A': {'B': 1, 'C': 1},
                             'B': {'A': 1},
                             'C': {'A': 1}}

            source : nodeid
                Source coordinate.

            target : nodeid
                Destination coordinate.
        """
        self.graph = graph
        self.source = source
        self.target = target
        self.path = []

        # record of each node's parent
        self.parent = {}

        # record of each node's estimate distance
        self.dist = dict([(pos, INF) for pos in graph])

        # set of open nodes
        self.nodes = set([pos for pos in graph])

    def step(self, record=None):
        """Starts the computation of shortest path.
        :Parameters:
            record : deque
                if a queue is specified, a record of each operation
                (OPEN, CLOSE, etc) will be pushed into the queue.
        """
        self.dist[self.source] = 0
        while self.dist:
            # get the node with minimum estimated distance
            node = min(self.nodes, key=self.dist.__getitem__)
            self.nodes.remove(node)
```

```
            if record is not None:
                record.append(('CLOSE', node))

            # if the node with minimum estimated distance has the
            # distance of infinity, then there is no such path from
            # source to distance.
            if self.dist[node] == INF:
                break

            # if the node is the target, then the path exists.
            if node == self.target:
                self._retrace()
                break

            # inspect the adjacent nodes.
            for adj in self.graph[node]:
                if adj in self.nodes:
                    self._relax(node, adj, record)
                    if record is not None:
                        record.append(('OPEN', adj))
            yield
        yield

    def _relax(self, u, v, record_):
        """Relax an edge.
        :Parameters:
            u : nodeid
                Node u
            v : nodeid
                Node v
        :Return:
            suc : bool
                whether the node v can be accessed with a lower
                cost from u.
        """
        d = self.dist[u] + self.graph[u][v]
        if d < self.dist[v]:
            self.dist[v] = d
            self.parent[v] = u
            if record_ is not None:
                record_.append(('VALUE', ('f', v, d)))
                record_.append(('PARENT', (v, u)))
            return True
        return False

    def _retrace(self):
        """This method will reconstruct the path according to the
        nodes' parents.
        """
        self.path = [self.target]
        while self.path[-1] != self.source:
            self.path.append(self.parent[self.path[-1]])
        self.path.reverse()


class GridDijkstra(Dijkstra):

    """This class is specified to grid maps.

    *Note*: On grid maps with all horizontal and vertical weights
```

```
    set to be 10 and all diagonal weights set to be 14, like
    we presumed in this scenario, Dijkstra's algorithm explores
    nodes in exactly the same way as a generic Breadth-First-Search
    algorithm.
    """

    def __init__(self, raw_graph):
        g, s, t = make_graph(raw_graph)
        Dijkstra.__init__(self, g, s, t)
```

C.5 – ee.py

```
import cProfile
import pstats
import psutil
import os
import sys

from astar import AStar
from bidirbfs import BiDirBFS
from constants import *
from dijkstra import GridDijkstra
from io import StringIO
from mazegen import mgen
from mapparser import parsemap_WHOLE


# based on the answer of Triptych on StackOverflow:
# http://stackoverflow.com/a/616672/825916
class Logger(object):

    """Class for duplication of stdout both to terminal and file."""

    def __init__(self):
        self.terminal = sys.stdout
        self.log = open("results_h", "w")

    def write(self, message):
        self.terminal.write(message)
        self.log.write(message)

    def flush(self):
        self.terminal.flush()
        self.log.flush()

# set stdout to Logger() class
sys.stdout = Logger()

At = []
Af = []

Aet = []
Aef = []

Act = []
Acf = []

Bt = []
Bf = []

Dt = []
Df = []


# A*
def A(m, h=0):
    a = AStar(m, h)
    for i in a.step():
        pass
    if not a.path:
        raise Exception("\a[!] A* PATH NOT FOUND!")
```

```
# Bi-directional BFS
def B(m):
    bdbfs = BiDirBFS(m)
    for i in bdbfs.step():
        pass
    if not bdbfs.path:
        raise Exception("\a[!] Bi-dirBFS PATH NOT FOUND!")


# Dijkstra
def D(m):
    d = GridDijkstra(m)
    for i in d.step():
        pass
    if not d.path:
        raise Exception("\a[!] Dijkstra PATH NOT FOUND!")

# get the ID assigned to the testing script
p = psutil.Process(os.getpid())
# set the priority to realtime
p.set_nice(psutil.REALTIME_PRIORITY_CLASS)

# walk to current working folder to find all the maps which are there
for root, dirs, files in os.walk(os.getcwd()):
    for file in files:
        if file.endswith(".map") and "map" in root:
            r, n = parsemap_WHOLE(os.path.join(root, file))
            if r is not None:
                print(n)
                for i, j in enumerate(r):
                    case = j
                    p = cProfile
                    #A* - manhattan
                    p.run("A(case)", filename="statsfile")
                    stream = StringIO()
                    stats = pstats.Stats('statsfile', stream=stream)
                    stats.print_stats()
                    s = str(stream.getvalue()).split()

                    Af.append(s[6])
                    At.append(s[10])
                    print("[*]A*M - {}/20 done. F: {} T: {}".format(
                        i + 1, Af[-1], At[-1]))

                    #A* - euclidean
                    p.run("A(case, 1)", filename="statsfile")
                    stream = StringIO()
                    stats = pstats.Stats('statsfile', stream=stream)
                    stats.print_stats()
                    s = str(stream.getvalue()).split()

                    Aef.append(s[6])
                    Aet.append(s[10])
                    print("[*]A*E - {}/20 done. F: {} T: {}".format(
                        i + 1, Aef[-1], Aet[-1]))

                    #A* - chebyshev
                    p.run("A(case, 2)", filename="statsfile")
                    stream = StringIO()
```

```
                    stats = pstats.Stats('statsfile', stream=stream)
                    stats.print_stats()
                    s = str(stream.getvalue()).split()

                    Acf.append(s[6])
                    Act.append(s[10])
                    print("[*]A*C - {}/20 done. F: {} T: {}".format(
                        i + 1, Acf[-1], Act[-1]))

                    # Bi-directional BFS
                    p.run("B(case)", filename="statsfile")
                    stream = StringIO()
                    stats = pstats.Stats('statsfile', stream=stream)
                    stats.print_stats()
                    s = str(stream.getvalue()).split()

                    Bf.append(s[6])
                    Bt.append(s[10])
                    print("[*]B - {}/20 done. F: {} T: {}".format(
                        i + 1, Bf[-1], Bt[-1]))

                    # Dijkstra
                    p.run("D(case)", filename="statsfile")
                    stream = StringIO()
                    stats = pstats.Stats('statsfile', stream=stream)
                    stats.print_stats()
                    s = str(stream.getvalue()).split()

                    Df.append(s[6])
                    Dt.append(s[10])
                    print("[*]D - {}/20 done. F: {} T: {}".format(
                        i + 1, Df[-1], Dt[-1]))

print("MAPS")
#A* - manhattan
print("A*")
print("AVG CALLS: {}".format(sum(map(int, Af)) // len(Af)))
print("AVG TIMES: {}".format(sum(map(float, At)) / len(At)))
#A* - euclidean
print("A*E")
print("AVG CALLS: {}".format(sum(map(int, Aef)) // len(Aef)))
print("AVG TIMES: {}".format(sum(map(float, Aet)) / len(Aet)))
#A* - chebyshev
print("A*C")
print("AVG CALLS: {}".format(sum(map(int, Acf)) // len(Acf)))
print("AVG TIMES: {}".format(sum(map(float, Act)) / len(Act)))
# Bi-dirBFS
print("B")
print("AVG CALLS: {}".format(sum(map(int, Bf)) // len(Bf)))
print("AVG TIMES: {}".format(sum(map(float, Bt)) / len(Bt)))
# Dijkstra
print("D")
print("AVG CALLS: {}".format(sum(map(int, Df)) // len(Df)))
print("AVG TIMES: {}".format(sum(map(float, Dt)) / len(Dt)))

# RANDOM MAZES
for size in (32, 64, 128):
    At = []
    Af = []

    Aet = []
```

```
    Aef = []

    Act = []
    Acf = []

    Bt = []
    Bf = []

    Dt = []
    Df = []
    for i in range(120):
        # generate a random maze of size *size*
        case = mgen(size)
        p = cProfile
        #A* - manhattan
        p.run("A(case)", filename="statsfile")
        stream = StringIO()
        stats = pstats.Stats('statsfile', stream=stream)
        stats.print_stats()
        s = str(stream.getvalue()).split()

        Af.append(s[6])
        At.append(s[10])
        print("[*]A*M - {}/20 done. F: {} T: {}".format(i + 1, Af[-1], At[-
1]))

        #A* - euclidean
        p.run("A(case, 1)", filename="statsfile")
        stream = StringIO()
        stats = pstats.Stats('statsfile', stream=stream)
        stats.print_stats()
        s = str(stream.getvalue()).split()

        Aef.append(s[6])
        Aet.append(s[10])
        print("[*]A*E - {}/20 done. F: {} T: {}".format(
            i + 1, Aef[-1], Aet[-1]))

        #A* - chebyshev
        p.run("A(case, 2)", filename="statsfile")
        stream = StringIO()
        stats = pstats.Stats('statsfile', stream=stream)
        stats.print_stats()
        s = str(stream.getvalue()).split()

        Acf.append(s[6])
        Act.append(s[10])
        print("[*]A*C - {}/20 done. F: {} T: {}".format(
            i + 1, Acf[-1], Act[-1]))

        # Bi-directional BFS
        p.run("B(case)", filename="statsfile")
        stream = StringIO()
        stats = pstats.Stats('statsfile', stream=stream)
        stats.print_stats()
        s = str(stream.getvalue()).split()

        Bf.append(s[6])
        Bt.append(s[10])
        print("[*]B - {}/20 done. F: {} T: {}".format(i + 1, Bf[-1], Bt[-
1]))
```

```
        # Dijkstra
        p.run("D(case)", filename="statsfile")
        stream = StringIO()
        stats = pstats.Stats('statsfile', stream=stream)
        stats.print_stats()
        s = str(stream.getvalue()).split()

        Df.append(s[6])
        Dt.append(s[10])
        print("[*]D - {}/20 done. F: {} T: {}".format(i + 1, Df[-1], Dt[-
1]))

    print("{}x{} MAZE".format(size, size))
    #A* - manhattan
    print("A*")
    print("AVG CALLS: {}".format(sum(map(int, Af)) // len(Af)))
    print("AVG TIMES: {}".format(sum(map(float, At)) / len(At)))
    #A* - euclidean
    print("A*E")
    print("AVG CALLS: {}".format(sum(map(int, Aef)) // len(Aef)))
    print("AVG TIMES: {}".format(sum(map(float, Aet)) / len(Aet)))
    #A* - chebyshev
    print("A*C")
    print("AVG CALLS: {}".format(sum(map(int, Acf)) // len(Acf)))
    print("AVG TIMES: {}".format(sum(map(float, Act)) / len(Act)))
    # Bi-dirBFS
    print("B")
    print("AVG CALLS: {}".format(sum(map(int, Bf)) // len(Bf)))
    print("AVG TIMES: {}".format(sum(map(float, Bt)) / len(Bt)))
    # Dijkstra
    print("D")
    print("AVG CALLS: {}".format(sum(map(int, Df)) // len(Df)))
    print("AVG TIMES: {}".format(sum(map(float, Dt)) / len(Dt)))
```

C.6 – graph.py (original author: Xueqiao Xu)

```
from constants import *


class InvalidMap(Exception):
    pass


def is_walkable(x, y, s, m):
    """Return whether the given coordinate resides inside the map
    and its status is NORMAL.
    """
    return x >= 0 and x < s and \
        y >= 0 and y < s and \
        m[y][x] != BLOCKED


def make_graph(s):
    """
    Generate an adjacency-list-represented graph from a multi-line string.

    :Parameters:
        s : str
            A multi-line string representing the maze.
            A sample string is as follows:
            s = '''
                1001
                0100
                1001
                '''

    :Return:
        graph : {(x1, y1): {(x2, y2): dist, ... }, ... }
            The graph is in ajacency list representation.
            The graph generated using the sample input above is as follows:
            graph = {(0, 1): {},
                     (1, 2): {(2, 1): 14, (2, 2): 10},
                     (3, 1): {(2, 0): 14, (2, 1): 10, (2, 2): 14},
                     (2, 1): {(1, 2): 14, (2, 0): 10, (1, 0): 14, (3, 1):
10, (2, 2): 10},
                     (2, 0): {(1, 0): 10, (3, 1): 14, (2, 1): 10},
                     (2, 2): {(1, 2): 10, (3, 1): 14, (2, 1): 10},
                     (1, 0): {(2, 0): 10, (2, 1): 14}}

        source : (x, y)
            source coordinate

        target : (x, y)
            target coordinate

    """
    try:
        nodes_map = [list(row) for row in s.split()]
        size = len(nodes_map)
    except:
        raise InvalidMap("The given raw map may be invalid")

    # put all available nodes into the graph
    g = dict([((x, y), {})
              for x in range(size)
```

```
            for y in range(size)
            if nodes_map[y][x] != BLOCKED])
    source = None
    target = None

    for x in range(size):
        for y in range(size):
            if nodes_map[y][x] == SOURCE:
                source = (x, y)
            elif nodes_map[y][x] == TARGET:
                target = (x, y)
            if is_walkable(x, y, size, nodes_map):
                for i in range(len(XOFFSET)):
                    # inspect horizontal and vertical adjacent nodes
                    nx = x + XOFFSET[i]
                    ny = y + YOFFSET[i]
                    if is_walkable(nx, ny, size, nodes_map):
                        g[(x, y)][(nx, ny)] = DIST
                        # further inspect diagonal nodes
                        nx = x + DAXOFFSET[i]
                        ny = y + DAYOFFSET[i]
                        if is_walkable(nx, ny, size, nodes_map):
                            g[(x, y)][(nx, ny)] = DDIST
                        nx = x + DBXOFFSET[i]
                        ny = y + DBYOFFSET[i]
                        if is_walkable(nx, ny, size, nodes_map):
                            g[(x, y)][(nx, ny)] = DDIST
    return g, source, target
```

C.7 – mapparser.py

```
import os

from constants import *

D = {".": "0", "G": "0", "@": "1", "O": "1", "T": "1", "S": "1", "W": "1"}


def replace_all(t, d=D):
    """Replaces the map to be compatible with the testing script."""
    for i, j in d.items():
        t = t.replace(i, j)
    return t


def parsescen(scen):
    r = []
    i = 0
    with open(scen) as f:
        f.readline()
        for line in f:
            # split the scenario line either by space or tab
            # depends on the version of the scenario file
            l = line.split(" ") if not "\t" in line else line.split("\t")
            # parse the coordinates of start, goal and distance between
them
            s = (int(l[4]), int(l[5]))
            g = (int(l[6]), int(l[7]))
            d = float(l[8])
            # if the start is different from goal and the shortest-path
exists
            if s != g and d > 0.0:
                r.append([s, g])
                i += 1
            # return the 20 scenarios
            if i == 20:
                return r


def parsemap_WHOLE(map):
    """Given a path to the map, returns array of maps according to the
scenario"""
    r = []
    with open(map) as f:
        d = map.split("\\")[-2][3:]
        n = map.split("\\")[-1]

        # read type, height, width, map (notes begin of the map)
        f.readline()
        h = int(f.readline().split()[-1])
        w = int(f.readline().split()[-1])
        f.readline()

        # if the map is not square-shaped or it is too big
        if (h != w) or (h * w >= 512 * 512):
            return None, None

        # replaces map to be compatible with the testing script
        v = [replace_all(line) for line in f]
        # path to scenario
```

```
        s = os.getcwd().replace("\\", "\\\\") + \
            "\\\\scen" + d + "\\\\" + n + ".scen"
        # parsescenarios
        t = parsescen(s)
        # iterate through each case in testcases
        for c in t:
            m = v
            #start, goal = testcase
            s, g = c
            # split the string, assign start and target and indices
            m = [list(x) for x in v]
            m[s[1]][s[0]] = 'S'
            m[g[1]][g[0]] = 'T'
            # re-create the map back
            m = ''.join(['',join(l) for l in m])
            r.append(m)
    # return maps and name of the map
    return r, n
```

C.8 – mazegen.py

```
from astar import AStar
from random import random, randint, seed

seed()


def solveable(m):
    def A(m, h=0):
        a = AStar(m, h)
        for i in a.step():
            pass
        return None if not a.path else True
    return A(m)


def mgen(size):
    """Generates a random square-sized maze of size *size*."""
    maze = [["0" for i in range(size)] for i in range(size)]

    for i in range(size):
        for j in range(size):
            # each field has a 20% change of being an obstacle
            if random() <= 0.2:
                maze[i][j] = "1"

    while 1:
        # generate source
        a, b = randint(0, size - 1), randint(0, size - 1)
        # if the current tile is blocked
        while(maze[a][b] == '1'):
            a, b = randint(0, size - 1), randint(0, size - 1)
        maze[a][b] = 'S'

        # generate target
        a, b = randint(0, size - 1), randint(0, size - 1)
        # if the current tile is either blocked or start node
        while(maze[a][b] in ("1", "S")):
            a, b = randint(0, size - 1), randint(0, size - 1)
        maze[a][b] = 'T'

        break

    # put the maze back together
    maze = "".join("".join(maze[i]) + "\n" for i in range(size))

    return maze if solveable(maze) else mgen(size)
```