VRIJE UNIVERSITEIT BRUSSEL

PATHFINDING ALGORITHMS: A PRACTICAL IMPLEMENTATION IN PYTHON

A Report Written for the Course: Techniques of Artificial Intelligence

SEPPE LAMPE
2019-2020

1. INTRODUCTION

This report accompanies python code which has been written for a project of the course 'Techniques of Artificial Intelligence' of Prof. Geraint Wiggins at the Vrije Universiteit Brussel. For this report option 1 'Implement yourself' was chosen, more specifically a practical implementation of the A* algorithm was constructed in python. Note that the description of the code in this report is general and not detailed, since the detailed explanation is commented in the code itself. However, the general strategy of the program, a detailed explanation on how to operate the GUI and a comparison with two other A* pathfinding implementations will be discussed here. In this personal adaptation four pathfinding algorithms i.e., A*, Uniform Cost Search (UCS), Breadth First Search (BFS) and Depth First Search (DFS) can be visualized on an interactive grid.

2. MATERIALS & METHODS

The code is split up into three files i.e., Board, Main and Comparison. The Board file contains a self-made class 'Board', which will be used to represent the field/world on which a path will need to be found. An instance of the Board class contains several attributes i.e., board, start, destination, size and visited. When creating a Board instance one parameter has to given i.e., size. Size is an integer which determines how large the (n x n)-matrix of the field will be. The board attribute contains an automatically, semi-randomly generated (size x size)-matrix with values between 1 and 5. Furthermore has one fourth of the board been transformed to obstacles, which have a value of 'X' instead. By default 'start' and 'destination' are tuples representing the coordinates of the top left (0, 0) and lower right (size – 1, size – 1) corner respectively. Lastly, the attribute visited is also an (size x size)-matrix but is completely initialised with False values.

The start location of the path can be changed manually via the 'setStart' method, which takes two parameters i.e., row and column. This selects the board[row][column] tile as the start. The 'setDestination' method operates similarly but for the destination. Furthermore has the instance three more methods which can be called. Firstly, setObstacle which takes two parameters, row and column, this method changes the value of board[row][column] to 'X'. Secondly, setValue, which takes three parameters, row, column and value. This method changes the value of board[row][column] to the value defined by third parameter 'value'. Lastly, resetVisitorFlags will turn all elements of the visited attribute back to False.

The Main file contains functions which define various pathfinding algorithms as well as the GUI. This is not ideal and would ideally be split into two different files. This was originally the case, however, both the algorithms and the GUI call on each other and require synchronisation. E.g., From the GUI a pathfinding algorithm can be called which then at each step needs to update the GUI. The only way this issue can be resolved is if the GUI environment is passed as argument to the algorithm. However, this makes the algorithms bloated and as they are so interweaved it was decided to merge them together into one file i.e., Main.

The Main file contains i.a., two functions which regulate the different pathfinding algorithms and two helper functions. The first function 'uniformCostOrAStar' takes three parameters i.e., sleeptime, tracepath and astar. This function is used for the calculations involving algorithm A* and UCS. As both algorithms can be implemented in a very similar manner, they were combined into one function. The parameter 'astar', when set to True (default)

will ensure that algorithm A* is executed, while UCS will be performed when the value is set to False. The sleeptime parameter indicates for how much seconds the system should sleep between each consecutive step of the algorithm, this value is by default 0 and can be used in order to slow down the calculations and inspect the logic of the algorithm from the GUI. Lastly, the tracepath parameter indicates whether between consecutive steps the GUI should show which path the algorithm is considering.

The function starts by saving the current timestamp in a variable, then it resets the colors on the board, this is done to erase any previously calculated path which is still shown on the GUI. This is followed by a reset of the visited flags (resetVisitorFlags) of the global variable board which is an instance of the Board class. Then it creates the agenda, a priority queue. For the UCS an element in the agenda will have the following structure: (travelled_distance, pathway), where travelled_distance is the distance travelled so far and pathway is a list of tuples which have been traversed so far. For the A* algorithm an agenda item will be: (f, pathway), where f is the sum of the heuristic and the travelled distance so far and pathway is identical as in UCS. The heuristic is the Manhattan distance (no diagonal movement is allowed in this implementation) and is calculated by the first helper function 'calculateHeuristic' which takes two points (tuples) as argument and returns their Manhattan distance. Continuing with the 'uniformCostOrAStar' function, the following step is to add the start point to the agenda and start the path finding (while loop).

In this loop the first item of the agenda, the item with the lowest travelled_distance or f, will be retrieved and stored in a variable, the last item in the pathway will be considered as the current position. If the current position equals the destination then the destination has been reached and the loop will stop, otherwise the visitor flag of the current position will be checked, if this position has already been visited before by travelling less then it will be disregarded and we will move on to the next item in the agenda. Note that this is not strictly necessary for the algorithm to operate properly, since its neighbours will all not be added further on, but doing this can shorten the whole execution time, especially if the field is large. If the current position is not the destination and it is not disregarded then the GUI will be updated to show the current pathway if tracepath is True and perform the sleeptime. After this, the eligible neighbours of the current position will be added to the agenda. The eligible neighbours are neighbours which are on the board, the 'findNeighbours' helper function is called to find these, which are not obstacles, not in the current pathway (this prevents cycles) and are not yet visited or only visited with a longer path. After this a new cycle of the loop will be executed until the destination is reached or until the agenda is empty, if the agenda is empty then the destination cannot be reached.

The 'depthOrBreadthFirstSearch' function operates in a similar manner as the 'uniformCostOrAStar' function. Similar to the 'astar' parameter it takes a 'breadth' parameter which by default is True and indicates that the algorithm should be BFS, otherwise DFS is executed. For BFS, the agenda is a FIFO queue, while DFS uses a LIFO stack. The eligibility of the neighbours is determined by checking whether they are on the board, they are not obstacles and not visited yet. An element of the agenda contains solely the pathway for both algorithms and the travelled distance is only calculated when a valid path is found. In this implementation DFS will favour going to the right first, when this is not possible it will try to go down then left and ultimately up. This means that this DFS implementation will be better in finding paths which have to go from the left upper corner to the right bottom corner than pathways which go in other directions, especially if the path is to the left upper corner.

For managing the GUI, the tKinter library was employed. Note that this was personally the first GUI I have ever created, therefore I do not claim that the GUI was implemented optimally, the focus was as well on functionality and not aesthetics.

The comparison file contains two other implementations of pathfinding through the A* algorithm. It was challenging to find other implementations which were comparable with the version written for this course. However, after some changes it was possible to construct a speed and reliability test for these implementations.

The first algorithm was retrieved from https://www.redblobgames.com/pathfinding/a-star/implementation.html, henceforth referred to as 'redblob implementation'. This code does not automatically generate different values for non-obstacle cells. The only change made to this implementation was the lay-out of the items in the priority queue (agenda). Instead of the form: 'next, priority' it was altered to '(priority, next)', this change was made because in some cases the original scenario would not return the shortest path. The official documentation of the queue.priorityqueue class states that input is preferred in the form of '(priority, data)' and by doing this the implementation seems to run better. Besides this small change a small function was written which makes it possible to reconstruct the same 'field' as in the self-written Board class. This function will be used to evaluate whether all the algorithms produce the same result.

The second retrieved implementation was retrieved from the following source: https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2,
henceforth referred to as 'swift implementation'. This implementation uses a class 'Node' which helps finding the path. Before this implementation could be compared however, a few modifications were necessary. In the code they have been marked as 'Modification 1-7'.

Modification 1 comprises of three lines which calculate the total travelled distance and returns it together with the result of the function (the pathway). This does not affect the speed of the algorithm by much since the original implementation already loops over each element in the pathway. The second modification changes the 'neighbours' of a cell, in the original implementation diagonal movement was allowed, while this is not allowed in the code written for this report. Note that enabling this for my code would not be complex but it was intentionally chosen to not include these steps since obstacles become easy to avoid when diagonal movement is allowed. The third modification is the smallest, the original code skipped neighbours which did not have a value of 0 (originally 0 was a crossable cell and 1 was an obstacle), the '!= 0' was changed into '== "X"' in order to match the implementation of the Board class.

Modification four and seven are closely related, both are caused by the same minor error in the source code (Fig. 1). The continue in both loops is supposed to jump back to the next child in children of the loop at the top of Figure 1. However, as it is implemented like in Figure 1 it jumps to the next element of the sub-for loops 'for closed_child ..' and 'for open_node …' respectively for modification 4 and 7. This was solved easily by adding two variables which indicate whether a continue should be performed or not which is checked after both sub-loops.

The fifth modification is small and adds the value of the current tile to the travelled distance instead of the original '1'. Lastly, the heuristic was originally the shortest distance squared (Fig. 2.), this was altered to be the Manhattan distance since diagonal movement was disabled. Note that the author calculates the shortest distance via Pythagoras theorem, yet he purposely does not take the root of the resultant (c^2). This actually means that the

author created an A algorithm implementation and not an A* implementation since the used heuristic is not admissible if c^2 is used instead of c even though the author states this in his text (Fig. 2).

```
78          # Loop through children
79          for child in children:
80
81              # Child is on the closed list
82              for closed_child in closed_list:
83                  if child == closed_child:
84                      continue
85
86              # Create the f, g, and h values
87              child.g = current_node.g + 1
88              child.h = ((child.position[0] - end_node.position[0]) ** 2) + ((child.position[1] -
89              child.f = child.g + child.h
90
91              # Child is already in the open list
92              for open_node in open_list:
93                  if child == open_node and child.g > open_node.g:
94                      continue
95
96              # Add the child to the open list
97              open_list.append(child)
```

4

7

Figure 1: The original code where modification 4 and 7 were applied. Note that modification 5 and 6 were performed on lines 87 and 88 respectively.

> *H is the heuristic — estimated distance from the current node to the end node.*

So let's calculate the distance. If we take a look we'll see that if we go over 7 spaces and up 3 spaces, we've reached our end node (`node(19)`).

Let's apply the Pythagorean Theorem! `a² + b² = c²`. After we've applied this, we'll see that `currentNode.h = 7² + 3²`. Or `currentNode.h = 58`.

But don't we have to apply the square root to 58? Nope! We can skip that calculation on every node and still get the same output. Clever!

With a heuristic, we need to make sure that we can actually calculate it. It's also very important that the heuristic is always an underestimation of the total path, as an overestimation will lead to A* searching for through nodes that may not be the *'best'* in terms of `f` value.

Figure 2: The heuristic as defined in the original implementation of the A(*) algorithm. By not taking the square root the heuristic loses its admissibility, turning it effectively into the A algorithm instead of A.*

3. RESULTS

Figure 3 depicts the GUI which the user gets to see. The slider 'Matrix size' denoted by A lets a user select the size of the (size x size)-field. The 'Generate Matrix' button marked by B automatically generates and shows a matrix (C) to the user. This field can be adjusted by the settings at D, which lets the user choose different start and destination points, create and delete obstacles and modify the values of single cells. The slider on the right (E) lets users slow down the calculation. This is only useful when the 'Trace calculations' option in F is toggled on. This lets the user see which possible pathway an algorithm is considering at any time. The selection box of G lets the user decide on which algorithm should be run and finally, the 'Calculate Pathway' button of H initiates the selected pathfinding algorithm. When an algorithm finds a pathway or decides that there is no possible pathway then it will give the user a message in the white box above I. When a user wants to clear the messages in this box he/she can press the button 'Clear Messages' denoted by I.

A comparison was made between the three A* implementations in terms of speed and correctness. Six tests were analysed, the first four tests were performed on all three algorithms, while the swift algorithm was excluded from the fifth test and the sixth test only included the personal implementation. The performed tests are still included in the comparison file, which can immediately be executed in order to redo these tests and confirm the results. In order to account for variability, each of these test has been run ten times. For each of these individual test runs a loop is performed a number of times where a board of a certain size is created and the start and destination randomly selected. The algorithms are each required to find the path according to their implementation, which is timed.

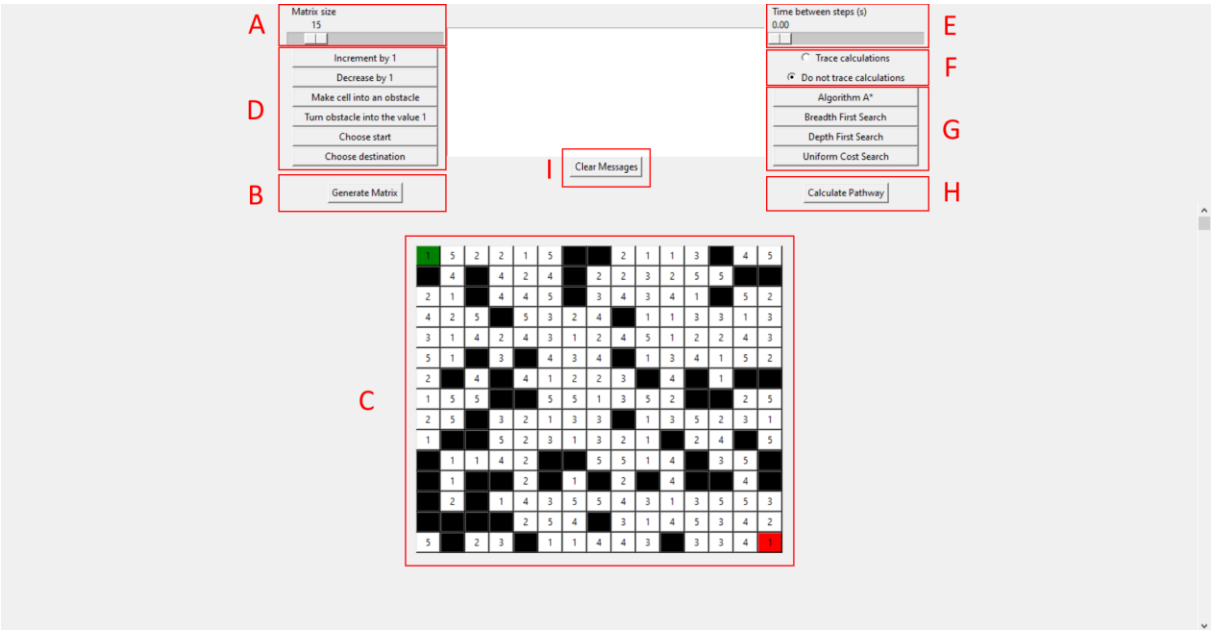

Figure 3: The GUI of the written pathfinding implementation in python. The slider 'A' adjusts the size of the matrix. The button 'B' generates and shows a matrix which is shown at 'C'. The options at 'D' let a user change this field. The slider 'E' lets the user adjust the speed of the algorithm if the first option has been enabled at 'F'. In 'G' the desired algorithm can be selected and 'H' initiates the calculation of a path.

The results of the individual tests can be found in Appendix A, the first values always represent the total elapsed time of the personal implementation, the second value is of the redblob implementation and the third value is the total processing time of the swift implementation.

Test 1 comprised of 2000 iterations on matrices of 5 x 5, in the second test 1000 iterations were performed on fields of 10 by 10. In the third test 500 iterations were completed on a matrix size of 20 x 20, while test 4 contained 100 iterations on 50 by 50 fields. Test 5, the first test without the swift implementation ran for 50 iterations on matrices of 100 x 100 and the last test was only performed on the personal algorithm and was comprised of 20 iterations on 200 by 200 fields.

The average of the ten runs of each of test is given in Table 1 as well as the standard deviation (S.D.) of test runs on the average. Note that the average values shown in this table do not indicate the average time it takes to solve one (n x n)-matrix but rather the average time it takes to solve m (n x n)-matrices e.g., 0.434 s is the average time it takes the personal adaptation to solve 2000 (5 x 5)-matrices and 11.227 s is the average time it takes the personal implementation to solve 20 (200 x 200)-matrices. The S.D. indicates the spread of these summed values for the individual runs.

Table 1: The results of the six tests on the personal, redblob and swift implementation of the A pathfinding algorithm. Note that the standard deviation does not indicate how much each separate calculation of a pathway varies from the average, it indicates how much the total sum of each one of the ten test runs differs from each other.*

	Average personal (in s)	*Average redblob (in s)*	*Average swift (in s)*	*S.D. personal*	*S.D. redblob*	*S.D. swift*
Test 1	*0.434*	*0.373*	*0.357*	*0.096*	*0.077*	*0.159*
Test 2	*0.809*	*0.709*	*1.548*	*0.124*	*0.100*	*0.236*
Test 3	*1.571*	*1.897*	*21.607*	*0.079*	*0.079*	*3.308*
Test 4	*2.095*	*7.835*	*1559*	*0.189*	*0.737*	*788*
Test 5	*5.124*	*56.428*	*n.a.*	*0.534*	*5.368*	*n.a.*
Test 6	*11.227*	*n.a.*	*n.a.*	*1.892*	*n.a.*	*n.a.*

4. DISCUSSION

Table 1 shows that for very small matrices the swift implementation is slightly faster than the personal and redblob applications and that the redblob is faster than the personal implementation. However, all three algorithms operate at more or less the same speed, if one takes into account the S.D. then their speeds cannot be distinguished confidently. Once the matrix size increases from 5 x 5 to 10 x 10, it can be noted that that the swift application gets drastically slower, it requires ~double the time which the personal and redblob implementations need, the S.D. shows that these observations are quite consistent. Both the redblob and personal implementation operate at the more or less the same speed, with redblob slightly outperforming the personal implementation. Though, if the size of matrix is increased again to 20 by 20 then the swift implementation starts operating very slowly. The redblob and personal versions run ~10 times faster with the

personal adaptation outperforming the redblob at this scale. At a scale of 50 x 50 the swift implementation has become ~750 times slower than the personal adaptation, even taking into account the large S.D. it is unlikely that these results are inconsistent. The personal version has also become ~4 times faster than the redblob implementation. Once the scale is increased even further to 100 by 100 fields then the redblob implementation begins to slow down as well, needing ~10 times the time the personal implementation requires. Furthermore does the sixth test (200 x 200 matrices) indicate that the personal algorithm scales very well and keeps operating at reasonable speeds consistently.

In order to demonstrate that the personal implementation is working correctly, its output can be compared to that of the redblob and swift versions. The personal and redblob adaptations both return the same result 100% of the time. Yet, once in a while the personal and swift versions return different pathways. This only happens in scenarios where there are two equal outcomes i.e., two different pathways with the same cost. So although the pathway which both versions return is not always the same, the total cost is. Which indicates that the three algorithms all operate equally good in terms of correctness. The reason why the redblob and the personal implementation always return the same result can be found when investigating the order by which the neighbours are added to the agenda. For the personal adaptation the order is determined by the order of the 'if clauses' in the 'findNeighbors' function, similarly this is done in the squareGrid.neighbors function for redblob. In the swift implementation, the order is defined by the list of tuples on the line where modification 2 happened 'for new_position in [(0, -1), (0, 1), (-1, 0), (1, 0)]:'.

When comparing the source code of the personal and redblob implementations, one can conclude that the redblob version of the A* algorithm will require less memory than the personal adaptation. This is mainly due to the use of dictionaries in the redblob implementation, there the agenda cannot contain the same cell twice with varying estimated cost. In the personal case this can happen i.e., when a path is found to a certain cell it is put in the agenda along with its priority but when later a shorter path is found before it is popped out of the agenda then there will be two paths to the same cell stored in the agenda. Each of these paths can be a list of considerable length depending on the board size.

Lastly, while operating the GUI the user might encounter some waiting time especially when creating large matrices, this can be up to ~10 s. This delay is not caused by the backend of the program but rather by the implementation of the GUI. As noted earlier, this was my first GUI, so I assume that there are still some better and faster ways to implement a few of the operations.

5. CONCLUSIONS

A practical implementation was made of the A* algorithm as well as UCS, DFS and BFS. The created A* version was compared to other, online-available pathfinding implementations of the algorithm. The self-written code underperformed slightly at very small matrix size, yet was not statistically relevant (within 1 S.D.). However, for normal to very big matrices it shows its strengths, it is often orders of magnitude faster than the compared implementations. It also yields the same results and thus is correct as well, when compared to the redblob adaptation it can be concluded that it requires more memory. Furthermore, was a GUI created which functions well and can give any user a better look at the logic of the A*, UFS, DFS and BFS algorithms without having to take a look at the code.

6. BIBLIOGRAPHY

Not applicable.

7. APPENDIX

APPENDIX A.

Test 1:

[0.718, 0.593, 0.831]

[0.41, 0.37, 0.322]

[0.414, 0.385, 0.332]

[0.371, 0.323, 0.335]

[0.415, 0.316, 0.287]

[0.384, 0.341, 0.31]

[0.391, 0.325, 0.286]

[0.39, 0.337, 0.285]

[0.417, 0.362, 0.303]

[0.429, 0.375, 0.282]

Test 2:

[0.77, 0.654, 1.373]

[0.753, 0.662, 1.356]

[0.719, 0.654, 1.471]

[0.714, 0.614, 1.391]

[1.152, 0.985, 2.183]

[0.902, 0.777, 1.734]

[0.789, 0.693, 1.576]

[0.768, 0.684, 1.444]

[0.766, 0.684, 1.454]

[0.76, 0.678, 1.5]

Test 3:

[1.474, 1.83, 18.653]

[1.575, 1.905, 19.538]

[1.59, 1.887, 22.371]

[1.521, 1.827, 19.785]

[1.54, 1.851, 19.387]

[1.768, 2.055, 22.611]

[1.641, 2.027, 23.569]

[1.509, 1.829, 19.167]

[1.525, 1.836, 21.9]

[1.563, 1.919, 22.088]

Test 4:

[2.289, 8.486, 3467.906]

[2.121, 8.218, 1731.865]

[2.058, 7.776, 715.706]

[2.151, 7.918, 1401.548]

[2.031, 7.406, 996.166]

[1.627, 5.987, 1753.886]

[2.081, 7.738, 947.615]

[2.065, 7.686, 1557.346]

[2.131, 8.294, 779.57]

[2.391, 8.842, 2240.468]

Test 5:

[5.091, 57.886]

[4.914, 53.213]

[4.483, 49.458]

[5.128, 57.25]

[5.597, 59.962]

[4.707, 51.792]

[6.203, 67.285]

[5.574, 59.397]

[5.206, 59.225]

[4.336, 48.814]

Test 6:

[10.526]

[7.126]

[14.624]

[10.109]

[11.38]

[10.634]

[11.741]

[13.0]

[10.643]

[12.487]