A comparison of Breadth First Search(BFS) and Depth-First-Search(DFS) using a pursuit-evasion game

Yixue Zhang
Boston University
jedzhang@bu.edu

Abstract

Breadth-First Search(BFS) and Depth-First Search (DFS) are two mainstream algorithms for traversing or searching tree or graph structures. Their searching complexity are the same, while their approach is not. In order to compare these two algorithms, we developed a pursuit-evasion game to test these methods.

The game is basically a combination of Pac-Man and Snake. The snake is going to eat a pea while avoiding the pursuers in a 2D maze.

By using BFS or DFS we can get different results and find out the advantage of each one. The performance of these algorithms will vary from the structure of map.

1. Introduction

Reaching a designated area for a robot without any collision on its way, is the goal of robot motion planning.The discipline of robot motion planning is evolving rapidly in recent years. Further study shows that motion planning has not only robotics applications, such as autonomy, automation, and robot design in CAD software, but also applications in other fields, such as animating digital characters, video games artificial intelligence, architectural design, robotic surgery, and the study of biological molecules. With the advance of modern control theory and computer science, new algorithm for robot motion planning (like A*, wavefront and D*) has been replacing outdated ones (like the bug algorithm) in the recent decades.

Graph traversal searching techniques are wildly used for measuring large unknown maps [1]. In this paper, we are comparing two mainstream graphic searching method, which is Breadth First Search(BFS) and Depth First Search(DFS).

Breadth First Search(BFS) is a blind search method, which will check every node in the entire system. In other word, BFS won't consider the structure of the entire map so that the algorithm will keep searching until it finds the goal. The algorithm follows the following steps. The search first starts at the root and then visits all of the children. Next, the search then visits all of the grandchildren, and so forth until it finds the goal [2]. The method contributed to many modern searching algorithms, in this paper, we use wave front planner algorithm as an implementation of Breadth First Search(BFS) for the further testing.

Depth First Search(DFS), like the BFS, is also a blind search method. Starting at a root, chooses a child, then that node's child, and so on until finding either the goal of a dead end. If a dead end is met, the search then backstage up a level and then searches through an unvisited child until finding the desired node or a dead end, repeating this process until the algorithm finds the goal [2]. Based on the method described above, we developed a simple algorithm for future testing.

Theoretically, these two algorithms have their own shortage and advantage. In general cases, though it would take more time for Breadth First Search(BFS) algorithm to compute, the result would be an optimal path. Computation time for Depth First Search(DFS) is shorter then Breadth First Search(BFS). However, this approach would stop once it finds a feasible path, the quality of the path, therefore, cannot be guaranteed.

The paper developed a pursuit-evasion game as a testing environment for these two algorithms. By comparing computation time and score, we can evaluate the performance of these two algorithms in a controllable environment. The rule of the game is taken from Pac-Man and Snake. By recording the score and computing time on the same testing environment, the difference of these two algorithms can be found.

2. Related Work

As two mainstream variation of graph traversal searching techniques, comparison among Breadth First Search(BFS) and Depth First Search(DFS) has long been a topic in the field of robot autonomous control. Works comparing these two concept including Tom Everitt and Marcus Hutter's [3],

which have carried out different analytical results with both methods in tree search and graph search [3][4], also, Bin Jiang's in paging environment [5]. However, comparing these two algorithms directory in practice, such as in the form of pursuit-evasion game has rarely been done before.

The concept of the differential game was first originated by von Neumann and Morgenstern [6], designed for using dynamic systems to produce results of computational interest [7]. The theory of the game is then be developed formally in 1954 by R, Isaacs [8]. As a variation of this type of game, pursuit-evasion game is one of the most common, challenging, and important adaptive problems that been studied in recent years and have been used to test the performance of Artificial Intelligence (AI) and Machine Learning (ML) [9].

Ms.Pac-Man computation [10] is held for groups around the world to compete with their algorithms, serves as a open arena for path planning methods. Common algorithms like Cell Decomposition Approach [11] and Sample Tree Search [9] are been used in the contest and proved that the Pac-Man platform is a effective and equal evolution environment for different algorithms.

Therefore, this paper adapted a similar environment as Pac-Man. With element from the game Snake added into the testing environment, we hope the new combined testing platform would prove itself a solid and fair testing environment for the two algorithms we are intended to compare.

3. Game Design

The pursuit-evasion game is achieved with MATLAB and is designed to be a 2-D maze, although taking the graphic interface from the game Pac-Man, the rule of the game is not the same. Instead of eating every pea in the map to get the next level, this game adopted element from the game Snake, in another word, the goal of the evader is not only to avoid the searcher but also to get the pea.

There are four key elements in the game, which are the maze, the snake, the pursuer, and pea.

3.1. The Maze

The map of the game is a 2-D maze and is formed with basic element called cell element. The map is consisted of 1296 (36X36) cell elements. Every cell element represent one step and have the same dimension and property.

Let U be the collection of all the possible moving direction of a single cell element, 1 represent accessible(feasible), 0 represent inaccessible(wall), each row represent a possible direction (up, right, down, left). There are 15 reasonable combinations,

$$U = \left\{ \begin{bmatrix}1\\0\\0\\0\end{bmatrix} \begin{bmatrix}0\\1\\0\\0\end{bmatrix} \begin{bmatrix}0\\0\\1\\0\end{bmatrix} \begin{bmatrix}0\\0\\0\\1\end{bmatrix} \begin{bmatrix}1\\1\\0\\0\end{bmatrix} \begin{bmatrix}1\\0\\1\\0\end{bmatrix} \begin{bmatrix}1\\0\\0\\1\end{bmatrix} \begin{bmatrix}0\\1\\1\\0\end{bmatrix} \begin{bmatrix}0\\1\\0\\1\end{bmatrix} \begin{bmatrix}0\\0\\1\\1\end{bmatrix} \right\}$$

$$\left\{ \begin{bmatrix}1\\1\\1\\0\end{bmatrix} \begin{bmatrix}1\\1\\0\\1\end{bmatrix} \begin{bmatrix}1\\0\\1\\1\end{bmatrix} \begin{bmatrix}0\\1\\1\\1\end{bmatrix} \begin{bmatrix}1\\1\\1\\1\end{bmatrix} \right\}$$

By assign corresponding possible moving direction to each cell element, the map can be drew as follow,

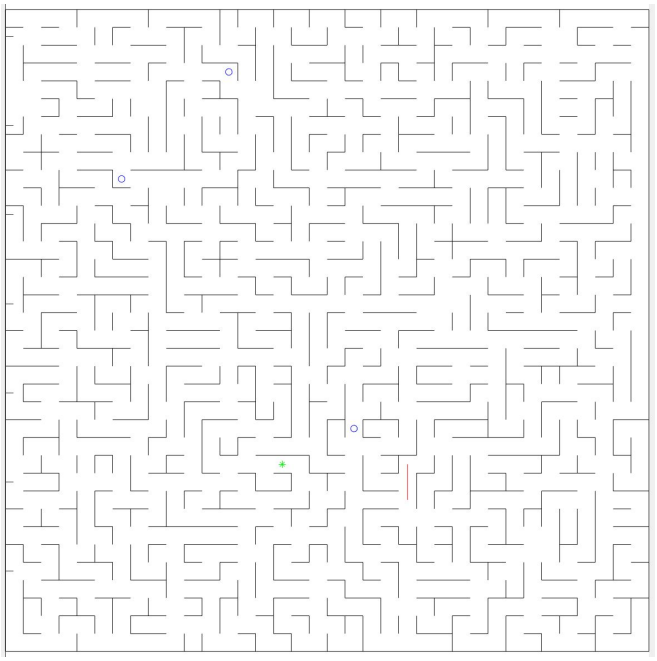

Figure 1. Example of the map. Red line represents the snake, blue dots represent the ghosts and the green dot represents the pea.

3.2. The Snake

The snake in the game is first represented as a back dot since it has not get any pea yet. With the proceed of the game, the snake would get longer every time it gets a pea. Meanwhile, capture avoidance is another vital factor for the snake. The game would be over once the ghost touched any part of the snake body.

We let the snake know the location information of the ghost at all times, by applying different algorithms, the snake would try find a feasible path to the pea without running into a ghost.

3.3. The Ghost

The only goal of the pursuer, also known as the ghost, is to capture the snake. Represented as a red dot, the ghost

would first go casually by following a path generate by A* pointing at randomly selected location(patrol mode). The ghost doesn't know the location of the snake until it was spotted. We define the ghost would find the snake when there is no obstacle in the line of sight between ghost and any part of the snake. When the ghost spotted the snake, it would change it's course immediately and follow the snake to the next turning point. At that intersection, if the snake is still in sight of the ghost, the ghost would continue follow it, if not, it would randomly select a direction and go to the other end of his sight, if the snake is still not in sight, the ghost would go back to patrol mode.

To reduce the difficulty for the ghost to catch the snake, a new ghost would appear randomly every three peas was eaten by the snake.

3.4. Pea

Pea is represented in the map as a stationary green dot. It would regenerate randomly on the map once it has been eaten.

3.5. Random Feature

Random features can be added to the game to make the game more interesting. Notice, that these random features are only effective to ghosts.

3.5.1 Frozen Trap

When a cell element is experiencing a Frozen Trap(FT), the background color of the cell would be blue and any unit goes over it would be trapped for two rounds.

3.5.2 Wind Trap

When a cell element is experiencing a Wind Trap(WT), the background color of the cell would be yellow and any unit goes over it would be blown to another random location.

4. Game Algorithm

The vital part of the game is its algorithm setting and variables. This game is consisted with four main functions and several variables. Following sections are detailed introduction.

4.1. Key Variables

Mapinfo.idx: Store the index of the node.
Mapinfo.neighbors: Stores its neighbor information.
Mapinfo.neighborsbackup: Stores its neighbor information for easy recovery.
Mapinfo.x: Stores its coordinate information
Mapinfo.(up/down/right/left): Stores its accessibility to each direction.
Tinfo.currentT: Stores the current location index of MR.Pac Man.
Tinfo.xPath: Stores the following steps for MR.Pac Man.
Tinfo.length: Store the information of snake length.
CTinfo.idx: Stores the current location index of this ghost.
CTinfo.xPath: Stores the following steps for this ghost.
CTinfo.vision: Stores the nodes that is visible to this ghost.
CTinfo.mode: Stores the current mode of the ghost.(1 for petrol mode, 2 for chasing mode).
pea: Stores the location index of the current pea.
gameoverindicator: The game would be over if the gameoverindicator is 1.
plannercount: Stores the number of times that a planner been called in the game.
Totaltime: Stores the total time consumed by planner in the game.

4.2. danger area

The function of danger area is to predict the next move of the ghost. This function mark all possible node that is accessible to each ghost in N steps. N is usually the length of the snake.

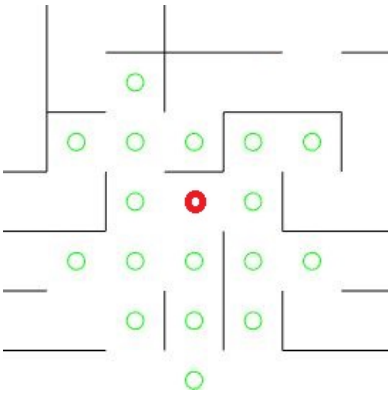

Figure 2. Example of danger area. Danger area are marked as green dot, the ghost is marked as red dot, all these green dot would be accessible for the ghost within N(N=3) steps.

4.3. Tpathplanner

The function of Tpathplanner is to provide MR.Pac Man with relatively safe path, detailed algorithm is as follows:

First, the function would check the remaining length of T.xPath, if it is less then 2 or or the following N step is in danger area then do the followings:1.Modify the value of every node marked as 'danger area' so that the planner would treat then as obstacle and the planner won't generate a path over it.2.Call planner to generate a feasible path.3.If there is no feasible path then repeat the followings: a.Recompute danger area with N=N-counter. b.Set counter

= counter + 1. c. Modify the map again with new danger area. d.Call planner Until a path is found or the counter reaches 5. (Due to the limited size of the map, the length of the snake and the number of the ghosts, this algorithm cannot provide a path that guarantee it's safe)

Algorithm 1 Tpathplanner

```
if The length of T.xPath is less then 2 or the following N step is in danger area then
    Modify the value of every node marked as 'danger area'
    Call planner
    if No feasible path can be found then
        repeat
            Recompute danger area with N - counter
            counter = counter+1
            Modify the map
            Call planner to generate a feasible path
        until a path is found or counter reaches 5
        if counter is 5 then
            Call planner to generate a path directly from current T location to pea.
        end if
    end if
    Reset Mapinfo to default settings.
end if
```

4.4. CTvisibilitycheck

This function is to return the visible node for each ghost. Visible, in this special circumstance include every node between the center node and the nearest wall in a giving direction.

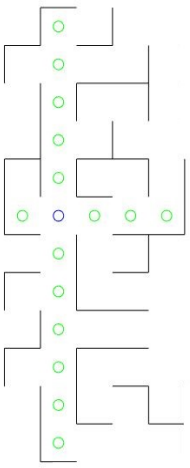

Figure 3. Example of visible node. The blue dot represent the current location of a certain ghost, green dots shows it's visible nodes

4.5. CTpathplanner

The function of CTpathplanner is to provide each ghost with their path. This planner does the followings':
First, it would check if the ghost can see the snake, that is, whether any part of the snake can be found in visible nodes. If so, change it's current mode to mode 2 (chasing mode), if not, stay in mode 1 (patrol mode). Second, if this ghost is in mode 2 (chasing mode), call planner to generate a path directly from it's current location to the closest node where the snake is found and replace CTinfo.xPath with new path. If the ghost is still in mode 1 (patrol mode), check it's remaining path. If the length of CTinfo.xPath is less then 2, call planner to generate a path from current location to a randomly selected node.

Algorithm 2 CTpathplanner

```
if The ghost found the snake in it's sight then
    Switch it's present mode to mode 2 (chasing mode)
else
    Switch it's present mode to mode 1 (patrol mode)
end if
if The ghost is in mode 2(chasing mode) then
    Call planner to generate a path from it's current location to the snake.
    Replace CTinfo.xPath with new generated path
else
    The ghost is in mode 1 (patrol mode)
    if The length of CTinfo.xPath is less then 2 then
        Call planner to generate a path from its current location to a random selected node.
    end if
end if
```

4.6. CTmove

The function of CTmove is to move the position of each ghost.

First, the algorithm would scatter a white dot on it's previous location to erase it's last step. Next, it would extract next step from CTinfo.xPath and scatter a blue dot, representing it's new location.Finally, it would check if the ghost has been caught. If the new position is on top of any part of the snake, then the game is over.

4.7. Tmove

The function of Tmove is to move the position of MR.Pac Man. This algorithm does the followings:
First, it would check the value of Peacount.If Peacount is 0, the snake has not eaten any pea yet. In this case, the snake is represented as a red dot. Then the algorithm would erase it's previous step with white dot on it's old location. Extracting from Tinfo.xPath, the algorithm would get it's next step

and scatter it with red dot, representing the new position of MR.Pac Man.After taking it's next move, the algorithm would check if the new position is on top if a pea. If so, the pea would be eaten, in another word, the algorithm would add 1 to peacount,add a new ghost into the map, erase the pea with white dot, generate a new pea randomly and plot it with green dot representing the new position of the new pea.

If the peacount is not 0 (the snake has eaten a pea before). In this circumstance, the snake is represented as a red line. The algorithm would first read from Tinfo.snakebody and get it's previous snake body information and erase it's old body with white line. Extraction from peacount, the algorithm would know the length of the snake and combining it's path information from Tinfo.xPath, a new position of the entire snake can be found. Plot the new found snake body with red line and store in into Tinfo.snakebody. Finally, the algorithm would check if the snake's new body is on top of the pea. If so, 1 will be added to peacount, a new ghost would be added to the map, old pea would be erased with white dot and new pea would be generated randomly and plot as a green dot.

Algorithm 3 Tmove

```
if Peacount is 0 then
    Erase it's previous step with white dot
    Scatter it's next step with red dot
    if The snake is on top of a pea then
        Peacount = Peacount + 1
        Erase the pea
        Add a new ghost into the map
        Generate and plot a new pea randomly
    end if
else
    Erase its previous body with white line
    Plot it's new body with red line
    Refresh snakebody
    if The snake if on top of a pea then
        Peacount = Peacount + 1
        Add a new ghost into the map
        Erase the pea
        Generate and plot a new pea randomly
    end if
end if
```

4.8. Testing related variables

Several variables are been created for the purpose of algorithm comparing. Every time a planner is called by Tpathplanner or CTpathplanner, 1 would be added to plannercount and the time spent on computing path would be recorded in Totaltime.

When the game is over, the plannercount and Totaltime would be store into a local file along with peacount for futher analysis.

4.9. The main algorithm

The main algorithm of the game is a summary of previous sub-functions. The main function does the followings:

First, the algorithm would call map-draw to plot the maze. Second, it would initialize the ghost, pea and MR.Pac Man by creating their variable and plot their first location. After initialization, the main function would call these sub-functions repeatedly until the game is over. Finally, store plannercount, peacount and Totaltime to local file.

Algorithm 4 The main function

```
Initialize the map
Initialize the first position of MR.Pac Man, ghost and pea.
repeat
    Call Tpathplanner
    Call CTVisibilitycheck
    Call CTpathplanner
    Call CTmove
    Call Tmove
    Check whether the game is over
until Game is over
Store plannercount, peacount and Totaltime to local file
```

5. Experiment and Result

The aim of the experiment is to compare the difference of BFS and DFS under different circumstances. The game provided a good platform for the test because the two planner would be called multiple times and their path-planning environment is vary much alike. Also, randomly appeared goals provided the algorithm with tasks of difference difficulties, which would test every possible aspect of a path-planning algorithm.Random features are not enable in the test.

The experiment would stop after collecting 100 effective data set of each path-planning algorithm.

5.1. Testing Environment

System:Microsoft Windows 10 Professional x64
CPU: Intel Core i5-7600K 3.8Ghz
Matlab: Matlab 2017b

5.2. Score Comparison

According to the analysis, the two algorithms provided different results. The average score for BFS is 5.71, while the average of DFS is 3.19.

The reason for that difference may because the two algorithms have different path quality. While BFS would expand the whole map to find the optimal path to the goal, DFS would stop when it found the first feasible path. The longer it takes to get to the goal, the bigger the chance is to be caught by the ghosts, that is the reason the average score for DFS is lower then BFS.

Figure 4. Score comparison

5.3. Time Comparison

After analyzing the result, we get the average CPU time for BFS is 0.06s and average for DFS is 0.05s. Though the difference on average CPU time between these two algorithms is not much, the pattern is different. CPU time for BFS lies mostly in the section between 0.04 to 0.1 while the majority time for DFS is distributed between 0.02 to 0.08. On the other hand, the maximum CPU time for BFS is 0.18 while the maximum time for DFS is 0.22. We can conclude from the above pattern that in most cases, the DFS algorithm would provide us with a path in a very short time. In extreme circumstances, however, it would take a little bit longer for it to get a path. The BFS algorithm, on the other hand, would provide us a path within a relatively consist time period, in another word, more stable.

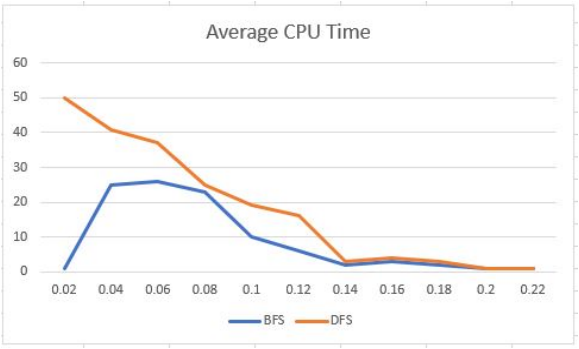

Figure 5. Time comparison

6. Conclusion

This two algorithm have their own advantage in different circumstances.

In most cases, the DFS would provide us with a valid path faster then BFS. In extreme cases, however, it would take more time for DFS to come up with a path then BFS, which indicated that time consumption for BFS is more consistent and stable.

The performance of the two algorithms on the platform shows the importance of the quality of a path. Regardless the time consumption and using the same evasive strategy, MR.Pac Man would have a higer score if the planner always provide it with an optimal path. Because BFS provides MR.Pac Man with a optimal path and DFS does not, the average score by using BFS is higher the DFS by 179%.

7. Future Works

Thouth the platform uses a evasive strategy, the average score for either algorithm is still not very high. The rule that a new ghost would be add to the map after every pea is been eaten seems not fair to MR.Pac Man, a upper limit for the number of the ghost should be set to provide more opportunity for MR.Pac Man. With a higher score, the number of times that the testing algorithm is been called would increase dramatically, which would give us more testing samples and insures the accuracy of the reslut.

Also, apart from CPU time and score, more variable should be collected to evaluate the performance of the testing algorithm. By collect more data from the platform, we can understand the algorithm much better.

References

[1] M. Kurant, A. Markopoulou, and P. Thiran. Towards unbiased bfs sampling. *IEEE Journal on Selected Areas in Communications*, 29(9):1799–1809, October 2011.

[2] Howie Choset / Kevin M. Lynch / Seth Hutchinson / George A. Kantor / Wolfram Burgard / Lydia E. Kavraki / Sebastian Thrun. Principles of robot motion. pages 580–588, May 2005.

[3] Tom Everitt and Marcus Hutter. Analytical results on the bfs vs. dfs algorithm selection problem. part i: tree search. In *Australasian Joint Conference on Artificial Intelligence*, pages 157–165. Springer, 2015.

[4] Tom Everitt and Marcus Hutter. Analytical results on the bfs vs. dfs algorithm selection problem: Part ii: Graph search. In *Australasian Joint Conference on Artificial Intelligence*, pages 166–178. Springer, 2015.

[5] Bin Jiang. Traversing graphs in a paging environment, bfs or dfs? *Information Processing Letters*, 37(3):143–147, 1991.

[6] J. von Neumann and O. Morgenstern. Theory of games and economic behavior. 1944.

[7] W. Willman. Formal solutions for a class of stochastic pursuit-evasion games. *IEEE Transactions on Automatic Control*, 14(5):504–509, October 1969.

[8] Isaacs R. Differential games i, ii, iii, iv. *RAND Corp. Res. Memo. RM-1391, 1399, 1411, 1468*, 1954.

[9] D. Robles and S. M. Lucas. A simple tree search method for playing ms. pac-man. In *Proc. IEEE Symp. Computational Intelligence and Games*, pages 249–255, September 2009.

[10] S. M. Lucas. Ms pac-man versus ghost-team competition. In *Proc. IEEE Symp. Computational Intelligence and Games*, page 1, September 2009.

[11] G. Foderaro, A. Swingler, and S. Ferrari. A model-based cell decomposition approach to on-line pursuit-evasion path planning and the video game ms. pac-man. In *Proc. IEEE Conf. Computational Intelligence and Games (CIG)*, pages 281–287, September 2012.