Pathfinding for Agents in a Dynamic Environment

Abstract

Pathfinding is an increasingly important problem in Artificial Intelligence (AI). It involves finding the most efficient path for an agent to take in order reach a goal state. The simplest form of the this problem is found when a single agent is working in a static environment, in other words the environment doesn't change. However, this problem becomes increasingly more complex when there are multiple agents in a dynamic environment. Examples of this can be found in turn-by-turn directions (e.g. Google Maps), game development, and even robotics. In this report we will analyze this problem in present-day. Below, section 1 summarizes common approaches/applications to this problem today along with an analysis of methods for handling dynamic environments. Section 2 depicts the main focus: a simulation of Frogger using pathfiding in dynamic environments. Section 3 lays out the experiment that is ran on top of the Frogger in order to analyze the three algorithms in terms of efficiency in section 4.

Introduction

In everyday life us, as humans, take actions that appear to require little to no thought. Some simple examples are: driving, walking, facial-recognition, or even writing/speaking. These actions, as simple as they may seem, require extensive processing. For example: driving a car requires a human to steer, and press the gas/brake pedals at crucial points in time. It is up to the human to process the surrounding environment in order to make these decisions accurately. The majority of the work is done when processing the environment. For now lets assume there is a single-agent, the driver, in the environment in which they're operating which is static (i.e. not changing). The driver must actively be checking its field of vision, along with its blind spots for obstructions in the path being took. It can then correct its direction, when necessary, based on the data collected. If the driver fails to process the environment correctly/fully, an error can occur. In turn, these errors can lead to severe consequences.

In Artificial Intelligence (AI), this example can be simplified as a pathfinding problem. There are several algorithms to solve this: Dijkstra's, A* to name a few. Now lets expand on this problem in order to see the full extent of the processing necessary. Previously, the environment was defined as single-agent and static; let us properly redefine the environment to be multi-agent, and dynamic (i.e. changing). With this newly defined environment, the driver is accompanied on the road by other drivers. Not only are the other drivers brought into play, but pedestrians and other moving agents as well. The driver must process the environment that now includes monitoring other agents actions. This now leads to more complex errors that can occur. Lets assume the driver

has processed the environment correctly, this does not account for other agents miscalculations and/or errors. This simple problem of pathfinding has now become more complex and includes error correction for not only the agent in focus, but others as well.

This problem of pathfinding in a dynamic environment has been increasingly popular. Self-driving cars have appeared in blockbuster movies for years now (e.g. I, Robot). This futuristic concept has started to become a reality in recent years. Companies, such as Google, have worked on autonomous vehicles for years now. Even though they have developed working prototypes that have drove amongst civilians, the technology is still in the works.

Throughout this paper, we will be focusing on a much simpler problem than autonomous vehicles: walking. In specific, we will be examining an agent who operates in a dynamic, multi-agent environment. In everyday life humans walking are faced with several obstacles: crossing a road, etc. We will simulate this problem using a 1990's kids childhood favorite: frogger. This game involves a frog who has to cross a road, river, and even does mazes depending on the level. This game will allow us to examine the efficiency of pathfinding algorithms and how they can be used in a multi-agent, dynamic environment.

1 Applications

Pathfinding is a problem that has existed for years. In the past there have been several methods of solving it. In doing so, several algorithms have been introduced to allow us to find an efficient solution. Below is an analysis of common approaches in practice, along with the introduction to the problem in AI using dynamic environments.

1.1 Approach

Pathfinding involves finding the best path for an agent to reach a goal state within an environment. Before the problem can be attempted, the environment must be expressed in a form that can be easily manipulated. A graph is the simplest form of this. This graph (G) can be defined as $G = (V, E)$ where:

- **V = Vertices**: A set of points in the graph.
- **E = Edges**: A set of edges that connect the vertices. These edges can either be directed or non-directed.

How the graph is extracted from the environment is situational based on the type of problem one is working with. One example is in game development, where using navigation meshes and waypoints is a common approach [2]. A navigation mesh is simply a 3D surface in which the agent can travel on, similar to a floor plan of a building. Waypoints can be described as a set of points visible to the agent, in which it can travel between.

Once this graph is created, common pathfinding algorithms can be implemented solve the problem. These algorithms can be classified into two sub-groups: single-source shortest path problem (SSSP), and all pairs shortest path problem (APSP) [5]. A classic SSSP algorithm is Dijkstra's. Given a start vertex s and a goal vertex t, the algorithm will begin to relax (i.e. remove constraints) all edges between the start index and goal by exploring all the nodes in between. The time complexity in worst-case is $O(n^2)$ [3].

Dijkstra's algorithm is efficient, when working with a small data set, but the run time is exponentially greater as the data set increases. This leads us to A*, another SSSP algorithm. A* uses a cost function to examine the costs between nodes of the graph [4]. This cost function is defined by *f(n) = g(n) + h(n)*; where *f(n)* is the total cost, *g(n)* is the total distanced traveled thus far, and *h(n)* being the heuristic cost for node *n*. The heuristic is defined to be an underestimate of the actual cost from node n to the goal index. [3]. One of the most common APSP algorithms is the Floyd-Warshall algorithm. This algorithm approaches the problem in a different fashion. Using a two dimensional array D, which is used to store the distances between each vertex. This algorithm will iterate over every node finding the shortest path between each of them [1]. The Floyd-Warshall algorithm has a tight bound $\theta(\mathrm{n}^3)$. In practice, A* has been shown to be used in both robotics and game development [6][9]. The use of a specific search algorithm is unknown when it comes to turn-by-turn directions due to proprietary proposes.

1.2 Dynamic Environments

The pathfinding problem has been has been solved, but the issue with a dynamic environment still exists. In a dynamic environment an obstruction to the path, in which was returned by the pathfinding algorithm, that the agent is traveling on can occur. When this happens, the agent must be able reroute itself in order to handle this error.

In robotics, a robot can be equipped with motion planning to solve this problem [8]. Motion planning algorithms will relax the completeness of the the problem. Two methods of this are path modification and re-planning. Both of these methods allow the robot to handle dynamic environments, but are quite costly [8]. In game development, direct modifications to the search algorithm A* can be utilized to account for a dynamic environment [9]. Local Repair A* adds rerouting measures through brute-force. With this method an agent will ignore all other agents (if present). In the event of a possible collision at the next step taken in the agents path, A* will be ran again in order to reroute the agent. This method can lead to cycles in the path being taken by the agent. If an area among the agents path is heavily occupied, A* will be called several times. These recalculations are expensive and will increase the time for the agent to reach the goal immensely. In some cases the agent can even become deadlocked if a surrounding area on the path being taken becomes completely occupied.

2 Approach

As stated previously, Frogger (i.e. http://www.frogger.net/), a common game for kids, will be used to simulate an agent walking in a dynamic environment. This allow us to analyze different methods of solving this pathfinding problem. In order to this, we will need to define mechanisms that will allow us to: control the agent (frog), extract the environment in graph form, parse the graph using pathfinding algorithms, and implement motion planning/error handling. This section will expand on each of these mechanisms in order for us to understand the internals of the problem being defined.

2.1 Java Robot

By itself, the frog within the game is by no means intelligent. The master mind behind it is normally the games player. This player is capable of viewing the game, and moving the frog using simple key

presses on a standard computer keyboard. In programming, there are several methods of simulating this. Java's Robot Class is one of the most common. This class will allow one to define a robot that is capable of generating input events for the applications running on a computer. This class was originally designed for automated testing of Java applications, but in our case will allow us to simulate the game player.

First a *Robot* must be declared using the *Robot()* constructor. Let us refer to our *Robot* as froggie. Once instantiated, froggie will be able to be able to start utilizing the classes functions. A select few will be necessary in order to fully simulate an actaul player. These functions are as follows [7]:

1. *mouseMove(int x, int y)*: Moves mouse pointer to given screen coordinates.
2. *mousePress(int buttons)*: Presses one or more mouse buttons.
3. *mouseRelease(int buttons)*: Releases one or more mouse buttons.
4. *createScreenCapture(Rectangle screenRect)*: Creates an image containing pixels read from the screen.
5. *getPixelColor(int x, int y)*: Returns the color of a pixel at the given screen coordinates.
6. *keyPress(int keycode)*: Presses a given key.
7. *keyRelease(int keycode)*: Releases a given key.

Functions one, two, and three will allow froggie to move the mouse to coordinates on the screen (as shown in Figure 1) in order to start the game itself in the browser of choice.

Figure 1: Moving the mouse pointer to begin the game.

Function four will be used to allow froggie to save a *BufferedImage* of the screen in a temporary buffer. This will be used alongside function five to extract the environment froggie is currently in.

More detail on this process can be found in the next subsection: *Extracting the Environment*. Finally, functions six and seven will used to allow froggie to move about the game. The selection/order of in which the keys will be pressed is examined in further detail in *Pathfinding Algorithms* later in this section.

2.2 Extracting the Environment

In order for froggie to move about the game properly, it must be able to view the environment it is currently in. This environment will be referred to as the game area. Our goal is to extract the game area and represent in a form that can be easily manipulated. Utilizing graph theory is one of the most common practices.

First the function *createScreenCapture(Rectangle screenRect)* will be utilized to allow froggie to grab a *BufferedImage* of the screen. This procedure is expensive in terms memory and runtime. For this reason, the goal is to minimize the amount of calls to function as much as possible. Unfortunately when a conflict occurs, this function call will need to be made again. We will examine this situation further in subsection *Conflict and Resolution*.

Now that we have *BufferedImage* of our game are, let us define our graph as a double indice Boolean array: *G[][]*. Each indice of the array will signify a section of the game area. Figure 2 helps one visualize how the game area will be divided into sections in which can be represented as *G*.

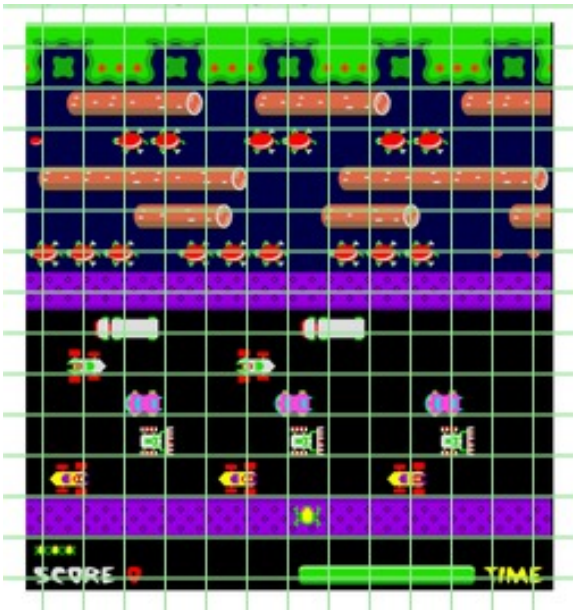

Figure 2: The game area divided into sections to represent our graph *G*.

Each section is a square that is sized in pixels. According to the constructs of the game itself, froggie can occupy a square section 25x25 pixels. This allows us to easily divide the entire game area in sections of this size. We can then traverse the game area and utilize the As stated previously, the *getPixelColor(int x, int y)* function to assign Boolean values to *G* based on the color returned. In our case *true* will signify a section that is safe for froggie to go, and *false* will signify a unsafe

zone. This simplified representation of the game area will make running pathfinding algorithms on efficient.

2.3 Pathfinding Algorithms

Shortest path problems are some of the most common problems in day-to-day life. A shortest path problem must first be conceptualized graphically. This graph can be represented by G which contains *(V, E)*, where V is a set of vertices with edges E [1]. With the graph defined, the shortest path problem comes to life. The only question is how do we find it? Several algorithms have been made to do so such as: Dijkstra's, Floyd-Warshall, and A*. Froggie will utilize all three of these algorithms within three separate implementations. An analysis/comparison of these three algorithms in use is described in further detail in section *Experiment*.

One of the first and most famous pathfinding algorithms is Dijkstra's algorithm [3]. Given a start vertex s and a goal vertex t, the algorithm will begin to relax all edges between the start index and goal by exploring all the nodes in between. The time complexity in worst-case is $O(n^2)$ [3]. Another famous shortest path algorithm is the Floyd-Warshall algorithm. This algorithm approaches the problem in a different fashion. Using a two dimensional array D, which is used to store the distances between each vertex. This algorithm will iterate over every node finding the shortest path between each of them [1]. The Floyd-Warshall algorithm has a tight bound $\theta(n^3)$. Lastly is the A* algorithm, which uses a cost function to examine the costs between nodes of the graph [4]. This cost function is defined by *f(n) = g(n) + h(n)*; where *f(n)* is the total cost, *g(n)* is the total distanced traveled thus far, and *h(n)* being the heuristic cost for node n. The heuristic is defined to be an underestimate of the actual cost from a node n to the goal index. [3].

Each of these algorithms discussed thus far are used to solve the shortest path problem, but are applied in different ways. Both Dijkstra and A* are used for single sourced shortest path (SSSP) problems [1] [3]. This problem is used to analyze the shortest path from a start vertex and a goal vertex. The Floyd-Warshall algorithm is used for the all pairs shortest path (APSP) problem. This algorithm will find the shortest path between all vertices in the graph G, but comes at cost of a higher run time [1]. When comparing the the two SSSP algorithms, there are only minor differences. The A* algorithm is informed due to the fact that it uses a heuristic function to help narrow down the nodes explored, instead of exploring all nodes possible in the uninformed Dijkstra's algorithm [4]. When looking at the big picture, all three algorithms appear to run on directed acyclic graphs. Dijkstra's and A* don't accept negative weight edges, whereas Floyd-Warshall's can [1] [4].

2.4 Conflict and Resolution

As discussed previously, when working in a multi-agent dynamic environment, the agent in focus must be able to process the environment, take actions when necessary and error handle. Lets refer back to our previous example of the agent as a driver. When the single-agent static environment was redefined to be mutli-agent and dynamic, special cases were introduced. For example: if at any point the drivers path is obstructed by another agent, a conflict has arisen. Unless properly handled, severe ramifications will occur.

In our simulation, if froggie's path is interrupted by another agent a conflict has occurred. In order for a resolution to be made, froggie must be able to identify the conflict before it occurs. Upon running one of the pathfinding algorithms on G, a path will be returned for froggie to take. Before taking each step in the path, froggie must verify that the next section in the graph has a Boolean

value of *true*. This preliminary checks are not expensive since we have simplified the representation of the game area. If the next value is *false*, then a conflict will occur if froggie is to take the next step in the path. In order to correct this, froggie will have to call *createScreenCapture(Rectangle screenRect)* again and iterate over the image using *getPixelColor(int x, int y)* to assign new Boolean values to G again. This will allow froggie to run the given pathfinding algorithm on G again to find a new path to the goal state. As discussed in *Applications* section, this method is expensive but is the best option given at this point in time.

3 Experiment

In practice, there are several pathfinding algorithms. If you recall Dijkstra's, Floyd-Warshall, and A* were selected to be used in our simulation. Which one in turn is the most efficient for pathfinding in a multi-agent, dynamic environment? This is the exact question we will try to answer by recording the run-time of each algorithm and keeping track of how many times a conflict occurs in which a resolution (i.e. running the algorithm again to find a new shortest path).

3.1 Implementation

In order to measure the run-time of the pathfinding algorithms being utilized, timing must be implemented. To do this, system calls can be made. The following is a simple example of how we can do this:

1. *long startTime = System.currentTimeMillis() ;*
2. *// Run algorithm of choice.*
3. *long endTime = System.currentTimeMillis() ;*
4. *long totalTime = (endTime - startTime) ;*

This method will allow for easy tracking of the run-time of each algorithm in milliseconds. These values will be recorded after each time the program is ran for future analysis (see section: *Analysis of Results*). In order for accuracy, the run-times will be recorded averaged among several executions.

Now that we are able to record the run-times of each algorithm, let us formulate a method of counting how many conflicts arise in the path returned. To implement this, a simple counter can be used. The key thing is placement for the counter itself. Our counter won't be incremented the first time each algorithm is ran, but rather only if a conflict occurs. Notice that we are also not timing the consecutive calls to each algorithm. When a conflict occurs, the start vertex of froggie will be different. This means that only a subsection of G will be parsed compared to the first iteration. Since it is unknown where froggie will be each time a conflict arises, it won't make sense to compare these run-times between each algorithm.

3.2 Results

After understanding the implementation of the experiment itself, we are able to extract our results by simply running our program and recording the data after each iteration. As stated previously,

we will execute our program multiple times to ensure accuracy. In our case, the program was executed twenty-five times for each the three implementations of the three pathfinding algorithms. Upon computation, the results can be found in the following figure:

Algorithm	Average Run-Time (ms)	Number of Conflicts
Dijkstra's	6 ms	2
Floyd-Warshall	13 ms	4
A*	4 ms	1

Table 1: Results given by running each algorithm twenty-five times and averaging the results.

4 Analysis of Results

Given the results from the experiment conducted in the previous section, lets us expand on them and try to reason with them. In order to do so we will utilized our knowledge of how each algorithm works internally. Without further ado, the average run-times were recorded in milliseconds along with the number of conflicts (each algorithm was ran twenty-five times). Based on the data, we can now sort each algorithm from "best" to "worst" based on our implementation. Upon doing so the three algorithms can be listed as follows: A* (4 ms), Dijkstra's (6 ms), and Floyd-Warshall (13 ms); respectively 1 conflicts, 2 conflicts, and 4 conflicts.

Lets first examine the Floyd-Warshall algorithm. This algorithm came in last place in terms of run-time. As previously stated, Floyd-Warshall is an algorithm created for APSP problems. In turn this means that extensive calculations need to made to find the shortest path from an given node in G to another. This in itself would explain the slower run-time when compared to Dijkstra's and A*. The run-time being slower can result in an increase in conflicts as well. Since froggie is in a dynamic environment, after a new *createScreenCapture(Rectangle screenRect)* call has been made, G needs to be setup again before the algorithm can even commence. As a result, G may contain significantly outdated information in terms of milliseconds.

Excluding the Floyd-Warshall algorithm for now, lets analyze our two SSSP algorithms: Dijkstra's and A*. At the end of the day, A* came on top, let us discover why this is the case. As stated previously, Dijkstra's algorithm is an uninformed search. That is to say that Dijkstra's has no knowledge of the distance from its current vertex to the goal vertex. A*, on the other hand, is an informed search. As we saw earlier, this comes with a cost function $f(n) = g(n) + h(n)$; where $f(n)$ is the total cost, $g(n)$ is the total distanced traveled thus far, and $h(n)$ being the heuristic cost for node n. The heuristic is defined to be an underestimate of the actual cost from a node n to the goal index. This allows the algorithm to direct its search towards the goal instead of "blindly" searching. I turn, the lower run-time will amount for less conflicts to occur.

Conclusion

Pathfinding is a common problem in AI and is increasingly important. There are several efficient methods at solving this, when in a static environment. As discussed, utilizing graph theory is key in order to run a pathfinding algorithm on it, whether this algorithm belongs to the SSSP or the APSP family. Dijkstra's has ruled for years, but A* has become its superior predecessor in recent years. When working with a dynamic environment extra measures must be taken in order to avoid

collisions. Methods such as motion planning, and Local Repair A* exist, but have their pitfalls. As more research progresses, a more efficient method of handling this is in eyes sight. For the time being, precautions and error handling must be dealt with on a per-situation basis.

In our simulation of the game Frogger, we implemented a *Robot* (froggie) that was able to utilize the three pathfinding algorithms chosen. Furthermore, and experiment was conducted in order to analyze the efficiency of each algorithm when placed in a mulit-agent, dynamic environment. As shown in practice, A* was shown to be superior in both run-time and the number of conflicts. This was due to its important heuristic and/or cost function that allows the algorithm reduce the amount of nodes visited. Dijkstra's came in second, showing quick times and a small number of conflicts. The reason it lagged behind was due to it being an uninformed pathfinding algorithm. The Floyd-Warshall came in dead last in both run-time and number of conflicts. Upon analysis, this was found to be a result due to the type of algorithm it is. Both A* and Dijkstra's are SSSP algorithms while Floyd-Warshall is an APSP algorithm. In turn this significantly increases the number of calculations necessary.

References

[1] J. S. Baras and G. Theodorakopoulos. Path problems in networks: Synthesis lectures on communication networks, 2010.

[2] R. Graham, H. McCabe, and S. Sheridan. Pathfinding in computer games. *The ITB Journal*, 4(2):6, 2015.

[3] A. G.-E. Hector Ortega-Arranz, Diego R. Llanos. The shortest-path problem analysis and comparison of methods, 2014.

[4] W. P. Imad S. AlShawi, Lianshan Yan and B. Luo. Lifetime enhancement in wireless sensor networks using fuzzy approach and a-star algorithm. pages 3013–3014, 2012.

[5] G. R. B. Karishma Talan. Shortest path finding using a star algorithm and minimum weight node first principle. *International Journal of Innovative Research in Computer and Communication Engineering*, 3, 2015.

[6] J. D. R. Millán and C. Torras. A reinforcement connectionist approach to robot path finding in non-maze-like environments. *Machine Learning*, 8(3-4):363–395, 1992.

[7] Oracle. Class robot.

[8] M. C. L. D. M. Russell Gayle, Avnees Sud. Reactive deformation roadmaps: Motion planning of multiple robots in dynamic environments.

[9] D. Silver. Cooperative pathfinding. In *AIIDE*, pages 117–122, 2005.