Dynamic Pathfinding Algorithms and Techniques

Tyler Loewen
University of Manitoba
Manitoba
Canada
loewent4@myumanitoba.ca

Abstract

Pathfinding is used in a large number of domains ranging from games, autonomous vehicles, drones, and road networks to name a few. Pathfinding is an important component in many applications and has been a focus of research for decades. As applications of pathfinding are becoming increasingly demanding with complex and dynamic environments the performance and efficiency of pathfinding is as crucial as ever. This paper explores some of the different algorithms and techniques used to create dynamic pathfinding solutions. It describes Brute-Force Replanning (BFR), Basic D* (BD*), and Focussed D* (FD*) algorithms and how they can be used to find paths from an agent to a goal in a dynamic environment. The performance of BFR, BD*, and FD* algorithms are compared, and conclusions are drawn about optimal use cases for the algorithms. An alternative method of implementing a dynamic pathfinding algorithm using neural networks is also explored. Finally, the benefits and drawbacks of these algorithms are compared and contrasted.

1. Introduction

Dynamic pathfinding is used across many different domains such as video games, robot navigation, and various networks such as road, water, electricity, and computer networks [Kumari and Geethanjali, 2010]. Dynamic pathfinding can be implemented using algorithms such as Brute-Force Replanner (BFR), Basic D* (BD*), and Focussed D* (FD*) [Stentz, 1995b, 1995a; Graham et al., 2005]. Each algorithm can be implemented using different techniques such as neural networks or waypoints, among others [Graham et al., 2005; Wang and Lin, 2012].

Many domains require agents to navigate from an initial position in an environment to a goal position. If the environment is static and the locations of all obstacles are previously known to the agent, traditional A* pathfinding can be used. In this case, the algorithm computes the optimal path off-line (i.e. while the agent is not navigating) before the agent begins navigating the path towards the goal [Stentz, 1995b]. Many domains do not have static environments such as game worlds with moving enemies or a vehicle's autopilot system with other moving vehicles. Besides completely static, the environment may contain a mixture of static and dynamic objects, making calculating the optimal path off-line using A* not feasible due to the constantly moving objects [Stentz, 1995b]. Also, the agent may only have either partial or no information about the environment which requires the agent to have some method of sensing its surroundings [Stentz, 1995b]. Many different algorithms have been created to solve this problem of pathfinding dynamic environments in real-time. The next three sections describe and compare three algorithms for achieving pathfinding in a dynamic environment.

2. Brute-Force Replanner

The Brute-Force Replanner algorithm (BFR) is an A*-based algorithm that can be used in a way that allows for dynamic pathfinding. First, the agent must have a form of sensors that can

be used to detect its nearby surroundings [Graham et al., 2005]. These may be physical sensors on a robot or rays cast from a game character [Stentz, 1995b, 1995a; Graham et al., 2005]. Next, the initial state of the environment is recorded which includes the current position of both static and dynamic objects. The algorithm then uses this static state of the environment to plan a path towards the goal. The agent does not begin moving towards the goal until the path calculation is complete [Stentz, 1995b]. If an obstacle is reached (i.e. detected by the agents sensing methods) at any point during the agent's traversal towards the goal, the agent stops, and the algorithm begins calculating a new path to the goal using the agent's new position [Stentz, 1995b]. That is, an updated snapshot of the current state of the environment is taken, and a new path is planned. Only once the computation of the new path is complete can the agent begin moving towards the goal again.

Multiple factors greatly decrease the computational efficiency of this technique of pathfinding. First, because BFR always produces an optimal path to the goal, if one exists, more computing power may be used to calculate an optimal path than what is necessary. For example, a car in a racing video game may not have to take the fastest path around a racetrack - it may be adequate to take a path that is slightly longer and slower [Wang and Lin, 2012]. Second, when the algorithm begins calculating a new path it will search its problem space in every direction around the agent even though it is unlikely the agent will perform a complete backtrack rather than slightly deflecting its path around the obstacle [Stentz, 1995b]. Third, if the environment is large or if the goal is far away, searching for an optimal path will take more time [Stentz, 1995b]. Due to these factors, among others, BFR is computationally heavy and therefore often too slow for the agent to move and respond quickly in a dynamic environment in real-time due to the constant need to stop at an obstacle and wait for a new path to be calculated before moving again.

3. Basic D* and Focussed D*

As described in [Stentz, 1995b, 1995a], the algorithm Basic D* (BD*) closely resembles Brute-Force Replanner (BFR) except BD* is dynamic in the way that it can modify cost heuristics dynamically while the algorithm is executing. Like BFR, BD* also produces an optimal solution. In BD*, the problem space (i.e. the environment) can be broken up into a set of states, each of which represent a location in the environment. Each location is connected by directional arcs that each have a cost associated with them. Arcs between states may be either directional or bi-directional. A directional state only allows the agent to travel in one direction across the two states connected by the arc. There may also not be an arc between two states meaning the agent cannot travel between the two states. Each state keeps an estimate of the cost to traverse to the goal state relative to its position. This cost is determined by summing the cost of all arcs between the current state and the goal state, where the cost associated with each arc is determined by some cost function. As the agent moves across arcs from one state to another, the cost of traversing between the two states is added to the total travel cost of reaching the goal state.

The improved efficiency of BD* compared to BFR is due to multiple factors. The first factor is the agents ability to detect an obstacle with its sensors during execution of the algorithm, allowing the algorithm to only update the cost associated with states in the local vicinity of the agent [Stentz, 1995a]. The Focussed D* algorithm (FD*) described in [Stentz, 1995b] improves on this by introducing a heuristic that only updates costs associated with states located in the general direction the agent is traveling. This reduces the number of states that need to have their cost updated and thus increases the efficiency by only computing the

optimal path for a limited number of states compared to the whole problem space [Stentz, 1995b, 1995a].

This can be seen from Figure 1 and Figure 2 from [Stentz, 1995b]. The figures contain arrows which represents the arcs between two different states, black shapes that represent obstacles unknown to the agent at the start of navigation, and a start and goal state S and G respectively. Figure 1 represents the BD* algorithm presented in [Stentz, 1995a] and Figure 2 represents the FD* algorithm presented in [Stentz, 1995b]. As can be seen from comparing the number of visited states with a calculated cost (arrows) between Figure 2 and Figure 1, it is clear that the FD* algorithm has a smaller search space compared to the BD* algorithm. This reduced number of visited states is the effect of focusing the cost propagation to states that are in the general direction the agent is traveling.

Figure 1: Basic D* Algorithm [Stentz, 1995b]

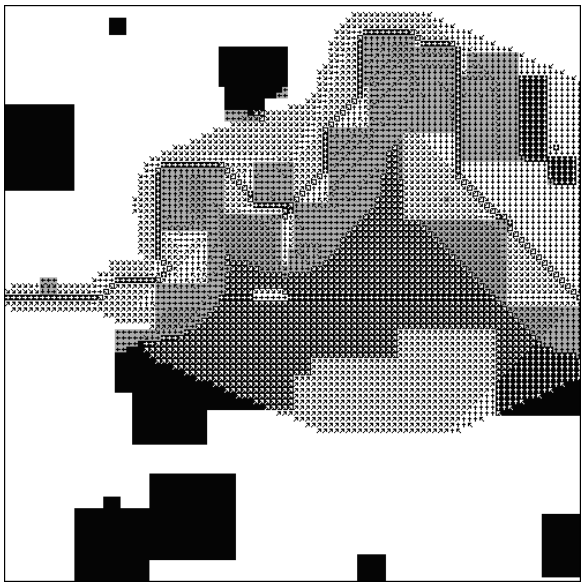

Figure 2: Focussed D* Algorithm [Stentz, 1995b]

The second factor making BD* more efficient than BFR is that the agent usually makes incremental progress towards its goal due to the higher cost imposed on the states previously visited by the agent [Stentz, 1995b]. This discourages backtracking and means as the agent gets closer to its goal the length of the path between the agent and its goal continues to get shorter

[Stentz, 1995b]. Because backtracking is discouraged through high-cost states and the cost associated with states is only updated for states local to the direction the agent is traveling, the performance of the algorithms is greatly improved [Stentz, 1995b]. These differences between the three algorithms allow BD* and FD* to be used for real-time agent navigation in an environment with static and dynamic obstacles and where the agent only has partial or no known knowledge of the obstacles in the environment.

4. Brute-Force Replanner vs Basic D* and Focussed D*

Four different algorithms were tested in [Stentz, 1995b] to verify the optimality of the algorithms and to compare their run-times. First, the Brute-Force Replanner (BFR) which is described in section 2. Second is Basic D* (BD*) as described in section 3. Third is Focussed D* with Minimal Initialization (FD*M) as described in section 3, which does not propagate path costs to every state. Fourth is Focussed D* with Full Initialization, which will not be included in this comparison. Each environment used for the test was square in dimensions with a start and goal state in the same position for each environment. Each environment had a mixture of static objects known to the agent and dynamic objects which were not known to the agent. The agent was equipped with a sensor to detect obstacles. Each algorithm was run on five randomly generated environments. The off-line time is the time it took to compute the initial path from the agent to the goal state using the known static environment. This is computed before the agent begins traversing the environment. The on-line time is the time needed to compute all replanning operations needed for the agent to reach the goal.

	Brute-Force Replanner **(Speedup)**	Basic D* **(Speedup)**	Focussed D* with Minimal Initialization **(Speedup)**
Off-line: 10^4	0.09 sec	1.02 sec	0.16 sec

On-line: 10^4	13.07 sec	1.31 sec	1.70 sec
Total Time: 10^4	**13.16 sec (1x)**	**2.33 sec (5.64x)**	**1.86 sec (7.08x)**
Off-line: 10^5	0.41 sec	12.55 sec	0.68 sec
On-line: 10^5	11.86 min	16.94 sec	18.20 sec
Total Time: 10^5	**11.87 min (1x)**	**29.49 sec (24.15x)**	**18.88 sec (37.72x)**
Off-line: 10^6	4.82 sec	129.08 sec	9.53 sec
On-line: 10^6	50.63 min	21.47 sec	41.72 sec
Total Time: 10^6	**55.71 min (1x)**	**150.55 sec (22.20x)**	**51.25 sec (65.22x)**

Table 1: Results from tests [Stentz, 1995b] (modified)

Table 1 describes the test results found in [Stentz, 1995b] with an additional time metric representing the sum of the off-line and on-line portion of an algorithms computation time as well as a speedup factor that represents how many times faster the algorithm is compared to BFR. As seen in Table 1, the BFR algorithm has the longest total compute time for every map size. On the other hand, FD*M is the fastest in total time for all map sizes making it good if off-line and on-line times are considered to be the same [Stentz, 1995b]. Looking at the off-line and on-line compute times, each algorithm can be the correct choice depending on the specific requirements of the problem. If reducing off-line computation time is the goal, then BFR is a good choice as the off-line computation time remains very short compared to the other algorithms, even on the largest 10^6 map. If real-time dynamic pathfinding is a requirement of the domain, then FD*M is a good choice. This algorithm has a short on-line computation time, almost as short as the on-line times for BD* with the added benefit of much shorter off-line times. This would be beneficial in situations where there is limited time available for off-line computation before the pathfinding begins [Stentz, 1995b]. As seen from the speedup factor for

each algorithm, FD*M has the largest speedup factor for every map size when compared to the other algorithms.

5. Neural Networks

Research has tackled the challenges of real-time pathfinding in games through the use of neural networks (NN) [Graham et al., 2005]. More games are becoming dynamic in nature due to an increase in the popularity of middleware physics engines that allow developers to easily add complex physics to their games [Graham et al., 2005]. Therefore, game worlds are often changing dynamically as the agent is actively navigating the world, leading to new obstacles the agent has to handle. Because of the larger number of games with dynamic environments, there is a higher demand than ever for pathfinding algorithms that can plan realistic and real-time paths for entities in a dynamic game world. There are often problems with real-time pathfinding approaches such as Basic D* (BD*) and Focussed D* (FD*) mentioned in [Stentz, 1995b, 1995a; Graham et al., 2005]. One of these problems not addressed in any previous section of this paper is the problem of agents only being able to react to obstacles once the agent is within a short range of the obstacle. This problem is inherent in the short range of the agent's sensors, whether the sensors or physical infrared sensors or digital ray casts, which can lead to agents only detecting obstacles once they have nearly collided already [Graham et al., 2005]. The second problem is agent's movements appearing unrealistic in nature. An agent may be expected to move in a natural curving or winding path but may appear to move in a straight line from one point to another [Graham et al., 2005]. This linear movement is often due to the decrease in the number of nodes in the search space, and therefore, granularity of the agent's movements [Graham et al., 2005].

As described in section 3 and [Stentz, 1995b, 1995a], the D* algorithms compute optimal paths in a real-time environment, but these algorithms work best for an agent that moves slowly and is okay with pausing for short periods of time while a path is being replanned. There are many different situations where the demands for real-time pathfinding are much higher, such as games. Many game worlds are constantly changing and require almost instantaneous responses from agents to make the world seem realistic and immersive to players. This is difficult to achieve with BFR, BD*, and FD* due to generally high replanning costs, especially when dozens or even hundreds of pathfinding agents are present [Graham et al., 2005].

The idea of using a NN for pathfinding presented in [Graham et al., 2005] allows for the agent to navigate a world by learning how to reach its goal on its own instead of simply following traditional paths created by A* and D* algorithms. Two requirements are defined in [Graham et al., 2005] for this type of learning to be possible. First, the agent needs a method of detecting its surroundings in the world so it can navigate around them successfully. To address this requirement, the agent can be configured to cast multiple rays into the game world in a forward direction which will detect any intersection with obstacles. Second, the agent needs a method of processing the information it receives from its sensors to determine the correct movement to make next. To address this requirement, [Graham et al., 2005] uses an Artificial Neural Network (ANN) with the data from the sensor as input to the NN which processes the data and produces output data that the agent uses to navigate.

For a NN to be able to solve a specific problem it needs to be trained through reinforcement learning [Graham et al., 2005]. Reinforcement learning works by executing the algorithm multiple times with agents, and through some rewarding technique, ranks the agent's performance by this score. The higher an agent's score, the better its performance and the

more desirable its traits are. By keeping the highest-scoring agent's traits, the next execution of the algorithm will hopefully produce even higher scoring agents that are closer to the optimal agent [Graham et al., 2005]. To help train the agents, [Graham et al., 2005] created a set of maps varying from simple to complex for the agents to navigate. To start, each agent was tasked with navigating towards a moving goal in a map with no obstacles or sensors on the agents. Next, they added obstacles to the map and sensors to the agents to receive sensor input and allowed the agents to roam the map learning how to navigate around obstacles. Next, they added a goal for the agents to reach while navigating the obstacle-filled map where input to the NN was the agent's sensor data and its relative position to the goal. During this last stage of learning, the agents had a difficult time learning how to successfully navigate the obstacles. This required creating a specific set of maps with carefully laid out obstacles.

There are some major benefits to using NNs. One benefit being the ability of the agent to successfully handle pathfinding situations it has never encountered before through training [Graham et al., 2005]. Another benefit is the low computation costs while the agent is pathfinding due to not needing to recalculate traditional node-based paths using slow algorithms. A drawback of pathfinding using NN is the difficulty of training agents [Graham et al., 2005]. The method of training can greatly affect the success the agent has in finding the goal, as seen by the difficulties found in [Graham et al., 2005] to train agents using a general configuration of obstacles in a map. Another downside to the NN pathfinding implementation in [Graham et al., 2005] is the jittery path of the agent as seen in Figure 3. Because NN pathfinding determines a path through reinforced learning and not pre-determined waypoints or nodes, applying a method of path smoothing to the agent's path may be difficult because the next target location of the agent is not known.

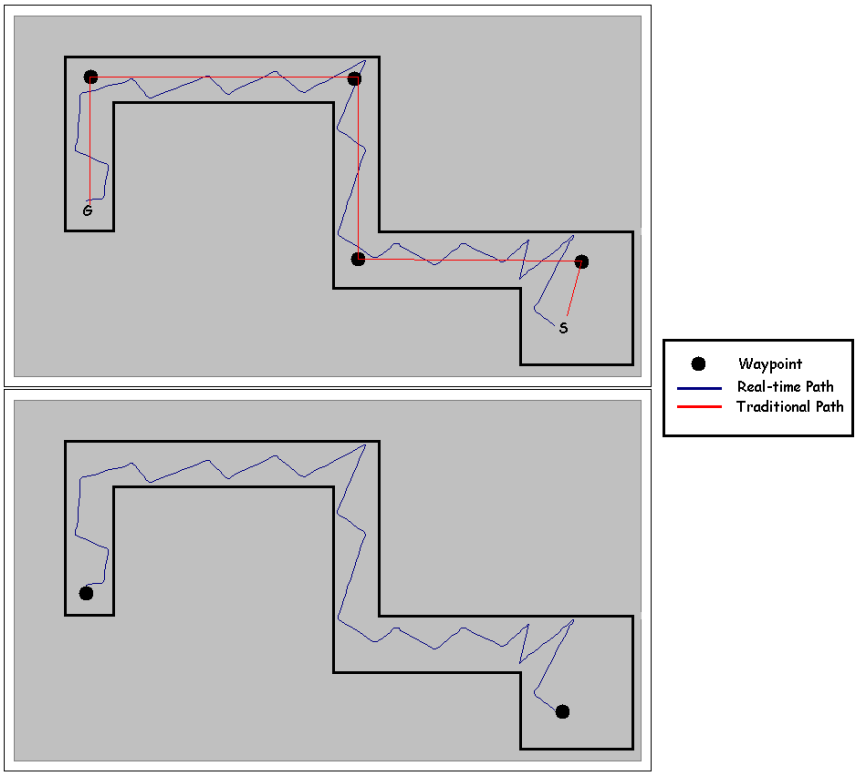

Figure 3: Real-time vs traditional pathfinding [Graham et al., 2005]

6. Conclusions

This paper introduces the concepts of dynamic pathfinding and the algorithms and variations that can be used to navigate a dynamic environment effectively. It describes Brute-Force Replanner, a variation of A* which computes a complete path before navigation begins and then as the agent reaches an obstacle, recomputes a new path given the agents updated position and the new state of the environment. This method was found to be much slower overall than Basic D* and Focussed D* which allow for the cost associated with states to be modified dynamically during execution of the algorithm. Even though the D* algorithms were a large improvement over the Brute-Force Replanner, they still lacked the speed required to be effective in domains that require real-time dynamic planning with fast response times such as games. To address the problem in the domain of games the paper described a pathfinding

algorithm implemented using neural networks instead of more traditional node-based traversals. The neural network allowed for true real-time pathfinding with large amounts of simultaneous agents through the use of a training method called reinforcement learning.

References

[Stentz, 1995b] Stentz, A., "The focussed D* algorithm for real-time replanning". *14th International Joint Conference on Artificial Intelligence (IJCAI)*, *95*(August), 1652–1659.

[Van Den Berg et al., 2006] Van Den Berg, J., Ferguson, D., & Kuffner, J., "Anytime path planning and replanning in dynamic environments". *Proceedings - IEEE International Conference on Robotics and Automation*, *2006*(i), 2366–2371.

[Graham et al., 2005] Graham, R., McCabe, H., & Sheridan, S., "Realistic agent movement in dynamic game environments". *Proceedings of DiGRA 2005 Conference: Changing Views - Worlds in Play*.

[Stentz, 1995a] Stentz, A., "Optimal and efficient path planning for unknown and dynamic environments". *International Journal of Robotics and Automation*, *10*(3), 89–100.

[Cui and Hao Shi, 2011] Cui, X., & Hao Shi., "Direction Oriented Pathfinding In Video Games". *International Journal of Artificial Intelligence & Applications*, *2*(4), 1–11.

[Elshamli et al., 2004] Elshamli, A., Abdullah, H. A., & Areibi, S., "Genetic algorithm for dynamic path planning". *Canadian Conference on Electrical and Computer Engineering*, *2*(June).

[Kumari and Geethanjali, 2010] Kumari, S., & Geethanjali, N., "A survey on shortest path routing algorithms for public transport travel". *Global Journal of Computer Science and Technology*, *9*(5), 73–76.

[Wang and Lin, 2012] Wang, J.-Y., & Lin, Y.-B., "Game AI: Simulating Car Racing Game by Applying Pathfinding Algorithms". *International Journal of Machine Learning and Computing*, *2*(1), 13–18.