PATHFINDING THROUGH TIME: ARE WE THERE YET?

Luke Jones
McGill University, Montreal, Quebec
260620255
luke.jones@mail.mcgill.ca

Zain Virani
McGill University, Montreal, Quebec
260567997
zainilabideen.virani@mail.mcgill.ca

INTRODUCTION

In this paper we analyze the behaviour of an agent given access to the A* pathfinding algorithm and the ability to travel backwards in time upon reaching a difficult pathfinding environment. We believe that this topic of study has practical application in the video game industry, as well as machine learning and artificial intelligence. We are attempting to test if, given the ability to time travel, having more opportunities to rewind, and more states to rewind to, allows the agent to find a more efficient path than through normal A* pathfinding. Luke Jones worked primarily on the game system itself, creating methods for saving and loading game states, as well as map generation, map behavior, and time flow. Zain Virani worked on implementing A* pathfinding behavior and agent rewind management, as well as final cost management and automated testing. Both script specification and report sections are outlined below for each contributor.

Luke Jones
- GameSystem.cs
- GameTile.cs
- GameMapBehavior.cs
- TimeFlowUIController.cs
- FiniteList.cs
- Background
- Results

Zain Virani
- PlayerController.cs
- AStarBehaviour.cs
- Introduction
- Methodology
- Results
- Conclusion
- Future Works

Topics

Pathfinding, Alternative Physics

1. BACKGROUND

In the field of video game AI and pathfinding there is plenty of resources regarding 2D and 3D pathfinding. However, when it comes to adding a time dimensions there is very little work that has been done. The idea of pathfinding forward in time isn't new, and has been presented as a means to resolve group cooperative pathfinding. Some of these solutions are mentioned in David Silver's paper titled *Cooperative Pathfinding* [3]. If you look at the single agent case for Silver's solution, you'll notice that there is little to no merit for pathing forward in time. The result will end up the same as with a standard 2D A*. The algorithm only becomes interesting when there are dynamic obstacles in the scene where it would be beneficial to look into the future to optimally avoid such obstacles. This brings us back to our project. Consider now instead of looking forward in time, we look back in time. The semantics shift from avoiding any future problems, to getting a do-over when things get tough. Similarly to Silver's algorithm,

this is only interesting when there is a dynamically changing environment. The way these states are represented also differ. In Silver's you must predict the changes in the environment. In a deterministic setting, this is easy enough to do. However many games make heavy use of randomness. Enemies may spawn randomly, move randomly, terrain may interact randomly. This presents a fundamental problem to Silver's approach of pathfinding forward through time. You must now fix this random behavior to make it pseudo-random and use its sub-textual determinism to make predictions. For our algorithm which considers going back in time this is a non-issue. We have no need to predict states of the game. Instead we only need a means to store previous states. Just like Silver's algorithm we have a window into the past that we will look at. We also have the luxury that these states don't have to be recorded every frame since we are not trying to achieve optimality, but simple trying to converge on it. Instead of avoiding problems, we ask our algorithm "Did things turn out how you planned? Were they better? Were they worse? Given the chance should you go back and re roll the dice again?" Here lies the fundamental problem of our solution. Given the opportunity to go back in time, should you? We answer this question with a simple comparison of the path then and now, as well as a scaling cost to rewinding.

Now in order to support our work we need a physics system capable of rewinding through time. Though not many games support this, there are a few out there than have implemented very efficient solutions. One of the most well-known and ground breaking solutions was made by Jonathan Blow, the creator of *Braid* [1]. His implementation is quite complex and multi-tiered, however we took inspiration from how his solution only includes records parts of the state that update regularly. Another example of this system was used to integrate time into pathfinding through visualization[2], where timer based visualization accuracy can be increased by considering speed and discrete time steps into account, wherein game state data structures are sent to a simulation manager, giving agents more autonomy and collision avoidance.

Rewinding through time presents its own set of problems. How do we keep physics and movement consistent? To which our solution was we don't. We simply store positions. How do we keep other programmers from affecting the scene during a rewinding period? Again, we don't enforce that. We make the state of our engine publicly accessible and it is the programmer's responsibility to check for that state when operating on the environment. How do we visually indicate when we are rewinding, and when we are simply moving backwards? *Braid* helped us answer this problem. *Braid* provides a visual indication via screen tinting, and also rewinds at a different pace from real-time. We found this was an effective way of indicating to a user that the game is in a rewind state.

2. METHODOLOGY

Our experiment consists of a very basic map, made as generic as possible in order to generalize results. The agent uses A* to find the most efficient path to its goal, and on every time step has the ability to compare its path efficiency from previous game states. From here the agent can decide to revert back to a previous game state, if it decides it can fare better from that position. Each rewind has an associated cost, depending on how many rewinds have occurred so far. In order to test our hypothesis we will run a total of 6 tests, each with 100 runs (where one run consists of the agent starting on (0,0) and ending on (10,10)), with a variety of tweaked parameters in each test. The program was built in *Unity3D*, with scripts written in C#.

2.1 Map Conditions

The experiment will take place on a small 10x10 unit discrete grid. An agent, with access to the A* pathfinding algorithm, is placed on the green grid tile (0, 0). The agent's underlying goal is to reach the gold grid tile (10, 10). Walkable grid tiles are green, gold, or light grey. Obstructed grid tiles are dark grey. The starting state of the map is shown below.

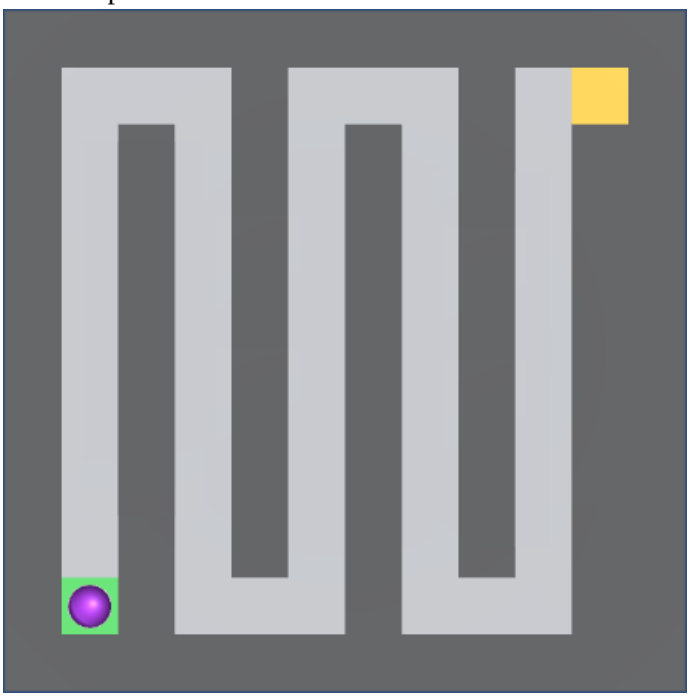


Figure 1. Game map.

The four walls in the center of the map are made up of 9 individual grid units. On each time step, each of these units is randomly chosen to be either walkable or obstructed (unless the agent is standing on the tile, in which case it must stay walkable). The state of the game is saved on every time step, however only a certain number of game states are saved.

2.2 Pathfinding

The agent makes use of the A* pathfinding algorithm to find its way to the goal, and moves one unit either vertically, horizontally, or diagonally on each time step. On each step, however, the map is re-randomized, and it must deploy A* again to find a potentially new path. Each path found has an attributed cost the agent will use to compare to other potential paths when it considers time travel. This cost is simply the total distance it will travel along that path, provided no more map randomization occurs. The distance is calculated from each tile along the path, thus including paths around obstructions. A vertical or horizontal move has an associated cost of 1. A diagonal move has an associated cost of 1.4.

2.3 Time-travel

An integral part of our project is the incorporation of time travel into the physics engine. The game system keep snapshots of the game in a bounded array whose size is determined by the Max Snapshots parameter. Theses snapshots are not 1:1 representations of the world in order to conserve space. Only the details that change are tracked. At every snapshot period we save and store a snapshot of the game and at any point in time you can rewind to a previous state up to Max Snapshots in the past. While rewinding,

simulation is halted and the game enters “rewind” mode which is separate from the standard game loop. When the rewinding is done, the game returns to a playable state and the agent is free to move. Upon reaching a new tile, the agent compares the cost of the current path to previous paths at previous game states. If it finds it can fare better from that game state, even with the added rewind cost, the game will revert back to the state it was at when the better path was found. Rewind costs increase each time a rewind is used, preventing the agent from rewinding forever.

2.4 Testing Conditions

The parameters to be adjusted are outlined in the table below.

Parameter	Type	Description	Script
Max Snapshots	Int	The maximum number of game states saved at any moment.	GameSystem.cs
Rewinds Allowed	Bool	If true, the agent is allowed to rewind.	PlayerController.cs
Comparison Window	Int	How often the agent can consider a rewind.	PlayerController.cs

Table 1. Adjustable parameters.

The parameters to be tested are outlined in the table below.

Parameter	Type	Description	Script
Sum of Total Costs / Total Runs	Float	The average path cost per run.	PlayerController.cs
Sum of Rewinds Used / Total Runs	Float	The average number of rewinds used per run.	PlayerController.cs

Table 2. Tested parameters.

We will run one as control, consisting of 100 runs. In this test, the agent cannot rewind at all. This represents normal A* pathfinding. We will run five tests as our main experiment, consisting of 100 runs each. In these five tests, the agent is given the ability to rewind, with a rewind cost of 1 (equivalent to moving one horizontal or vertical units), and a cost equation of *rewind_cost * (rewinds_used + 1)*. This rewind cost was chosen because it implies if even a single movement can be saved by rewinding, the agent will take advantage of it. The cost function increases to prevent overuse of the rewind function, simulating a game environment in which the user or AI cannot abuse a feature. The final path cost of a run is the actual distance travelled, where a horizontal or vertical movement has a cost of 1, and a diagonal movement has a cost of 1.4. The differences between the tests are outlined in the table below.

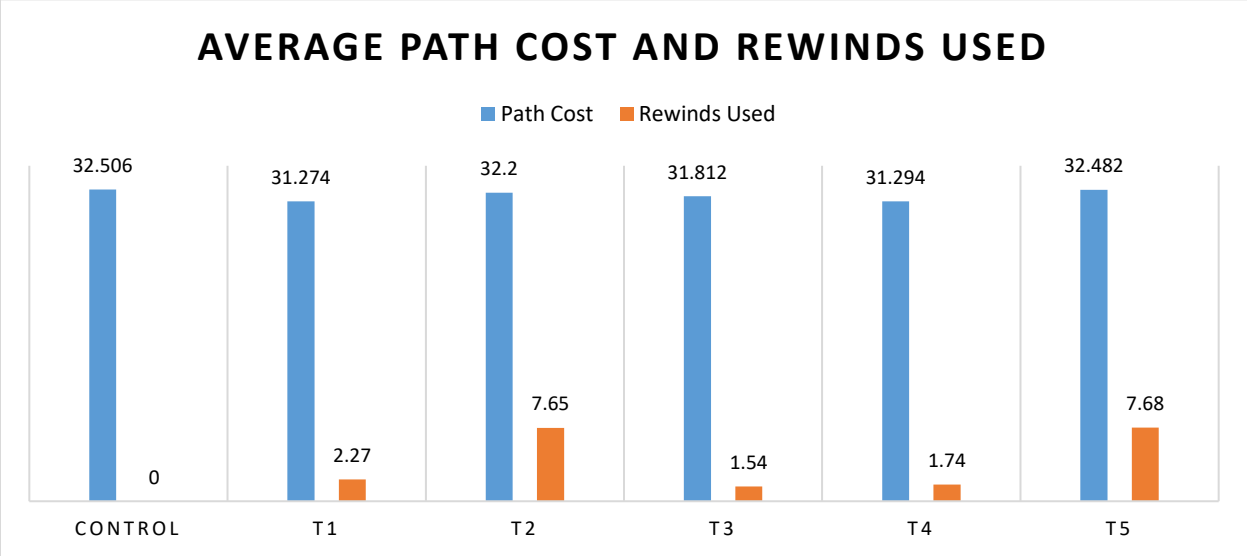
Test	Max Snapshots	Comparison Window
1	10	2
2	15	0
3	5	4
4	15	4
5	5	0

Table 3. Tests.

2.5 Hypothesis

Hypothesis: Given the ability to time travel will always provide an equal or better path to normal A*. Additionally, having more opportunities to rewind (a lower comparison window), and more states to rewind to (a higher max snapshot value), will allow the agent to find a more efficient path than normal A* pathfinding. Specifically, we expect test 2 to have the lowest average path cost among the 5 main tests and the control test. We expect test 3 to have the highest average path cost among the 5 main tests and the control test.

3. RESULTS



Graph 1. Average results over 100 runs.

To test if there is a difference between the control and our tests, we will first find if there is a significant difference between our 5 main tests. If there is, we will then look for a significant difference between tests

1, 3, and 4, as well as between tests 2, and 5, using ANOVA tests. These tests have been chosen due to similar results between their mean path cost and mean rewinds used. From there we will search for significant differences between the control and tests 1, 3, and 4, and between the control and tests 2, and 5. A brief summary of raw data can be found below.

TEST	Control	T1	T2	T3	T4	T5
1-20	33.76	30.17	32.14	31.52	31.6	31.58
21-40	32.15	31.15	32.09	33.74	31.1	34.13
41-60	31.5	31.89	32.42	30.84	30.89	32.24
61-80	32.46	30.81	31.8	31.42	30.6	33.61
81-100	32.66	32.35	32.55	31.54	32.28	30.85
Average Path Cost	32.506	31.274	32.2	31.812	31.294	32.482
1-20	0	2.05	6.75	1.35	1.8	7.9
21-40	0	2.4	6.35	1.85	1.5	8.75
41-60	0	2.4	7.85	1.5	1.6	6.95
61-80	0	2.15	7.9	1.6	1.75	8.2
81-100	0	2.35	9.4	1.4	2.05	6.6
Average Rewinds Used	0	2.27	7.65	1.54	1.74	7.68

Table 4. Summary of raw data.

3.1 ANOVA Tests

	Treatments					
	T1	T2	T3	T4	T5	Total
N	25	25	25	25	25	125
$\sum X$	781.85	805	795.3	782.35	812.05	3976.55
Mean	31.274	32.2	31.812	31.294	32.482	31.8124
$\sum X^2$	24466.5105	25922.733	25325	24491.6025	26414.6275	126620.4315

Std.Dev.	0.7888	0.2687	1.0181	0.6035	1.252	0.9707

Table 5. Summary of data.

Result Details				
Source	SS	df	MS	
Between-treatments	28.9303	4	7.2326	F = 9.87359
Within-treatments	87.902	120	0.7325	
Total	116.8323	124		

Table 6 ANOVA tests between T1-T5.

With an f-ratio value of 9.87 and a p-value of < 0.00001, the result is significant at $p < 0.01$. There is a significant difference between the means of all 5 tests.

Result Details				
Source	SS	df	MS	
Between-treatments	4.6514	2	2.3257	F = 3.44906
Within-treatments	48.5496	72	0.6743	
Total	53.201	74		

Table 7. ANOVA tests between T1, T3, T4.

With an f-ratio value of 3.44906 and a p-value of 0.037117, the result is not significant at $p < 0.01$. There is no significant difference between the means of tests 1, 3, and 4.

Result Details				
Source	SS	df	MS	
Between-treatments	0.9941	1	0.9941	F = 1.21249
Within-treatments	39.3524	48	0.8198	
Total	40.3465	49		

Table 8. ANOVA tests between T2, T5.

With an f-ratio value of 1.21249 and a p-value of 0.276331, the result is not significant at $p < 0.01$. There is no significant difference between the means of tests 2, and 5.

3.2 Control Comparison

With 95% confidence (assuming normal distribution), we can say that there is a difference of between 0.8562 and 1.2358 of the means of the control and (T1, T3, T4). We can conclude that tests 1, 3, and 4 provide better results than the control with 95% confidence.
With 95% confidence (assuming normal distribution), we can say that there is a difference of between -0.0436 and 0.3736 of the means of the control and (T2, T5). We can conclude that tests 2, and 5 provide the same results as the control with 95% confidence.

3.3 Anomalies, Trends, and Observations

Some of the highest recorded path costs were in the mid 40's, and the lowest path costs were in the high 10's (with a path cost of 14 being the absolute lowest bound possible).

Generally, we found that the number of steps the agent would rewind ranged from 1-3 for each rewind. When attempting to discover why this was, we played with parameters like max snapshots, comparison window, and rewind cost, but found that this range did not change. We reason that this is because the randomization is less of a factor the closer to the goal the agent gets. In other words, a large rewind would subject the agent to more obstructions, which is reflected in larger path costs, leading to the agent's decision to only rewind by short chunks of 1 to 3 steps.

We also attempted to find any correlation between number of rewinds used and the path cost. Using Pearson's Correlation Coefficient formula, we found $R = 0.5357$. This is a moderate positive correlation, meaning there is a moderate tendency for high path costs to pair with high rewind usage, and low path costs to pair with low rewind usage.

4. CONCLUSIONS and FUTURE WORK

4.1 Conclusions

Our hypothesis has been proven false. We expected all 5 tests to prove better or equal than normal A* pathfinding, which they were, however, we also expected test 2 to provide the best results, and test 3 to provide the worst results. The opposite was true. We concluded there was a significant decrease in average path costs from normal A* and tests 1, 3, and 4. Tests 2, and 5 provided equal average path costs. Upon analysis of test parameters, it seems more than likely that the number of game states available for rewind has no effect on the efficiency of the found path as the range of values was uniformly split among T1, 3, 4 and T2, 5. This claim is in agreement with our observation that, generally, the number of steps rewound was between 1 and 3, so having a large number of states available to rewind to would not make a substantial difference. When we look at the comparison window, however, in both test 2 and test 5, the value was 0, meaning the agent considered rewinding at every step towards the goal. Considering the fact that we found a moderate positive correlation between high path costs and high rewind usage, it is possible that considering rewinds more often (and consequently rewinding more often), results in a higher path cost. Indeed, we can back up these claims with further statistical analysis. Comparing the average rewinds used between tests 1, 3, and 4 and tests 2, and 5, we find (assuming normal distribution and 95% confidence) that there is a significant difference lying between 5.6717 and 5.9583. This could mean that capping rewinds, or changing the decision method for when a rewind is a good choice is the next step in improving pathfinding through time. Either way, it implies there certainly is a sweet spot of rewinding that can give a more efficient path. One could argue that this relies heavily on the rewind cost function.

Our cost function was linear. A constant cost would result in more rewinds, whereas an exponential cost function would result in less rewinds. So perhaps an exponential cost function could result in fewer rewinds, but one must also consider that when fewer rewinds are used, the agent is closer to following a regular A* pathfinding algorithm. To summarize, an agent given the ability to time travel will always provide an equal or better path to normal A*, however, having more opportunities to rewind (a lower comparison window), and more states to rewind to (a higher max snapshot value), will not allow the agent to find a more efficient path than normal A* pathfinding. Instead, using a reasonable number of rewinds provides the most efficient path.

4.2 Future Work

In order for an agent to use a reasonable number of rewinds, it must be able to decide, given that rewinding in this moment would save steps later, if rewinding now or saving a rewind for later is the better choice. The obvious solution to this problem is machine learning. Machine learning would be able to give an agent the distinct ability to make this decision, emulating a human decision, and avoiding the need for a rewind cap while improving the decision method.

Obviously, time travel is a tricky subject. Whenever time travel is implemented in a video game, tough choices have to be made to preserve linearity and the appearance of stable physics. What events are immune to rewinding? How do we ensure that motion is preserved when rewinding? Even in our system, one could make the case that time is not linear: in a truly linear timeline, map randomization would not vary after a rewind. Time has many more applications besides games, despite that being the focus of this paper. Even so, the subject has barely been explored, and even in this paper we only scratch the surface. The future of pathfinding through time, especially in a machine learning context, holds many opportunities. So, to conclude, we will respond to the titular question: are we there yet? Not even close.

5. REFERENCES

[1] Blow, J. (2010). *The Implementation of Rewind in Braid*.

[2] Lewen, T. and Derksen, C. (2011). *Integration of the time aspect into the visualization*. [Online] SIGCHI Conference. Available at: http://s3.amazonaws.com/academia.edu.documents/30934611/ISPG-Day-2011-proceedings.pdf?AWSAccessKeyId=AKIAIWOWYYGZ2Y53UL3A&Expires=1489701084&Signature=W1oaawBb84NQPoEtwSiEHtTYGFI%3D&response-content-disposition=inline%3B%20filename%3DABC_and_Perceived_Value.pdf#page=28

[3] Silver, D. (n.d.). *Cooperative Pathfinding*. [Online] University College London. Available at: http://www0.cs.ucl.ac.uk/staff/D.Silver/web/Applications_files/coop-path-AIWisdom.pdf