Re-implementation of Dynamic Terrain Abstraction

Megan Sumner

Cmput 658 - Single Agent Search

Introduction

Pathfinding in games is continually growing in requirements as games continue to improve in fidelity. The game Nightingale by Inflexion Games has created scenarios through their procedural generated content that has expanded the needs of navigation systems. (Sturtevant et al. 2019) lay out these pathfinding problems and develop a new abstraction of navigation on a grid to handle these new challenges. This paper displays the result of the re-implementation of the (Sturtevant et al. 2019) paper.

Problem

Pathfinding in games is a well known task and many engines, including Unreal Game Engine, have built in navigation systems that can handle most use cases. However, often times companies have to extend this functionality to fit the needs of their particular game. For Nightingale, this means developing new abstraction techniques to create more realistic worlds.

There are a few problems that Nightingale focuses on solving for their game. The first being that there are multiple creature types in the game. Different creatures will have different movement strategies. For example, a human faction may try to path towards roads, whereas deer may avoid roads and prefer forests. If all creatures used generic A*, they would behave similarly and also look less realistic. A human walking through a forest instead of taking a road can break players immersion. They may get to the goal in the shortest path, but within the realm of games, realism is more important than optimal path. Figure 1 shows a potential example of these different pathing goals

To solve this problem, there are a few options. Weighted A* can be used to weight different terrain types for each creature. Doing so will find good paths, but as (Sturtevant et al. 2019) discovered, the weights needed to make creatures not just try to go shortest path from start to goal were quite high. Shown in Figure 1, even with weighted terrain, weighted A* determined the best path was through undesirable terrain. This generally means that weighted A* ignores the terrain types and returns sub-optimal costs. Since terrain paths are important to the game direction of Nightingale, weighted A* doesn't provide a path that respects terrain paths well enough. To solve this Dynamic Terrain Abstraction (DTA) is used.

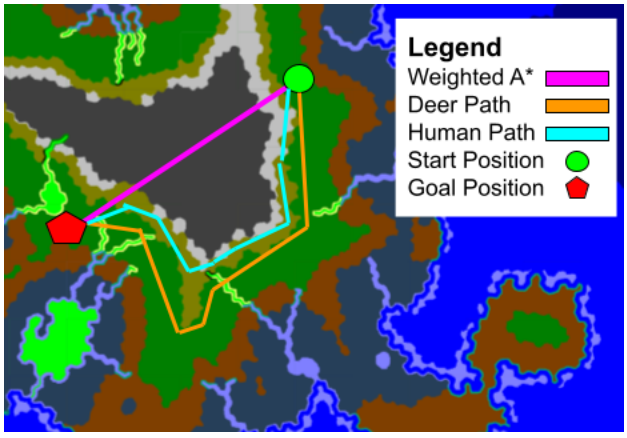

Figure 1: The path using Dynamic Terrain Abstraction for two different pathfinding types. The human uses the roads, while the deer uses the forest. Weighted A* ignores terrain type all together.

.

Dynamic Terrain Abstraction

DTA does a better job of respecting terrain pathing, though it does do a bit more work than weighted A* (Sturtevant et al. 2019). Since the focus is on having unique pathing styles for different creature types this becomes a useful strategy. DTA has a big upfront cost, but can be pre-computed similar to how navigation mesh is done in Unreal Game Engine. This makes the higher cost of DTA over weighted A* worth the extra amount of work.

DTA works by abstracting the level by terrain types onto a grid system. The world is split into a specific grid size and assigned a terrain type. This terrain type is then used to create regions for the abstraction. Regions being used as abstraction has been studied in (Sturtevant 2007) and DTA is a continuation of this work. The regions that are abstracted are done so in a way that uses the terrain ID as the region created. Breadth-First Search (BFS) is done to connect a region of terrain which is then stored in the abstract graph. The abstract graph can then be traversed and since we are prioritizing the region navigation in the lowest weighted terrain type, navigating in that terrain will happen first.

Dynamic Terrain

Terrain abstraction is not the only focus of Nightingale. The game also involves building structures/modifying the terrain which means that the abstract graph needs to handle dynamic terrain.

The pathfinding data changes because building in the game is allowed. This creates a challenge because users can create bridges, or remove areas that used to be navigable. This causes the terrain abstraction to extend its functionality and allow for dynamic updates to terrain types when changes occur. The cost of this can be quite expensive, but with DTA, whenever an update to the level is done another pass over the abstract graph can determine if the change causes the graph to be updated. The whole abstract graph does not need to be regenerated, which saves significantly on cost.

Performance

Overall games have to stay relatively quick on updates to keep up with the frame speed of the running game. Not only this, but the game also has to stay low on memory usage to keep up with the machine the code is running on. DTA is able to do this well due to the framework of the Minimum Memory (MM) abstraction proposed in (Sturtevant 2007).

Implementation

Assumptions

A few assumptions were made in this re-implementation. The random weights used for the terrain costs were kept as integers. The movement was kept to being 4 directions (up, down, left and right) and diagonal movement was ignored. Lastly, instead of regenerating the regions dynamically, re-running the code was done. As future work, using the previously generated abstraction and only modifying the changed code would help performance significantly.

Breadth-First Search

The first step for DTA is to create the regions based on terrain type. To do this, Breadth-First Search (BFS) was used. BFS allows us to get all connected regions of a graph. This splits the maps into segmented regions, each region consisting of the same terrain types. This was done by ensuring that the only children generated for a specific state had the same terrain ID. Once BFS was complete, a vector of states was created that contained information about which states had edges between each other.

Terrain Edge Pass

Once the first pass of BFS was done to form edges between all states in the same, connected regions, another pass was needed to link different terrain types together. This allowed the A* implementation to jump from one region to the other without getting trapped indefinitely in one region.

The re-expansions were generated by looking at all the states in the BFS state vector and checking to see if any of those states bordered a different terrain type. If this was the case, a new child was added to that state linking it to the state of the different terrain.

Once the terrain edges have been added to the BFS state vector, this vector can be saved to a file. Pathfinding in games is often pre-computed due to the high cost of generating navigation mesh for a whole level, this same tactic can be done for the abstract graph. Only when there are updates between two different terrain types, does another pass on the abstract graph need to be done. For this reason, a file of states is stored. The children were not stored as this would increase the file size significantly and the children can be regenerated during the A* pass.

Now that the state vector of connected terrains has been created, A* can be done to determine the optimal path from one location on the abstract graph to a goal location.

A*

For A*, a priority queue was implemented to keep track of the open list of states and an unordered set was used to keep track of the visited list. This was done because the priority queue is quick at sorting states by f cost and getting the next lowest f cost state quickly. The unordered set was used for the visited list because it allows us to override the functionality for finding states and therefore can find states by their current location. The current location was used to determine if a state had been visited before.

The heuristic that was used was the Manhattan distance as it is good for grid levels and allows us to have an admissible heuristic.

A* was run on specific terrain types, starting from the terrain type of the start location. This ensured that it exhausted its options on the specific terrain type before moving onto the next. We kept running A* on the abstract regions until the goal location was reached. For example if our abstract regions (r) were ordered from the start location to the goal as follows $start- > r0- > r1- > r2- > end$ then the first A* iteration would attempt to get from the start to r0, the next iteration would try to get from r0 to r1 and so on until the goal is reached.

To make the terrain abstraction more useful, the terrains are weighted. In this implementation they were weighted randomly between 1-5, but this could be changed depending on the creature that is pathfinding and the type of terrain they navigate best in. When A* is run, the heuristic is weighted depending on the weight of that specific terrain. Weighting it this way means that A* will navigate through the current terrain type until the next lowest f cost is a new terrain type. If the current terrain type has a lower weight, it will try to stay in that specific terrain type as long as possible. This behaviour is what provides the desired realism ie) for characters walking on roads instead of through a forest.

Results

The hardware used for these runs was a Microsoft Windows machine. It has 32 GB of RAM and a Intel(R) Core(TM) i7-8086K CPU @ 4.00GHz.

Abstraction

This implementation did not work well for long areas of connected terrain. Memory became a big issue with the bigger

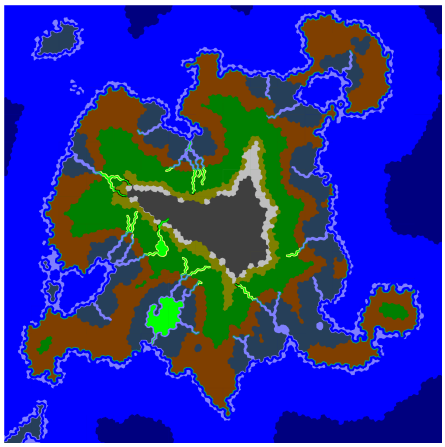

Figure 2: One of the maps that Dynamic Terrain Abstraction was generated on.

.

maps, possibly due to how the children states were stored. Using figure 2 as an example, it shows that there is a lot of light blue in the map. This light blue area is connected throughout the whole map and therefore, in this implementation, becomes a huge split of the region. BFS has to work for quite a while to generate this whole blue area and runs out of memory before it can generate. An extension to my implementation that might help to generate the regions better would be to split the map up into sectors as is done in (Sturtevant 2007). This way, if a region is too big, BFS will split the job up into multiple steps.

The next big issue that stopped the abstraction from being efficient was connecting the edges of two terrains. My implementation stores the children into the BFS states created and then does an iteration through the states created, checks to see if there is another state that has a different terrain type that borders it, and adds it to the children of the current state. Doing this, involves 2 for loops and therefore made my implementation $O(n^2)$. However, as shown in figure 1 it appears that my edge time taken is much more than exponential. Some refactoring of the code needs to be done in order to make this code anywhere near efficient enough to generate abstraction data for the whole map.

Map Size	BFS Time Taken (s)	Edges Time Taken (s)
2048 x 5	2	18
2048 x 10	5	50
2048 x 20	45	300
2048 x 50	600	DNF

Table 1: Time taken for different sizes of maps for the initial region generation (BFS Time Taken) and for the terrain edge generation.

A*

The A* implementation also ran into a few pitfalls. Firstly, I never resolved the issue I was having in assignment 2 around getting efficient times with A*. I think this is due to the closed list and open list not properly detecting if a state has been seen before. I ran into some issues with unordered sets that could not be resolved in time.

The A* implementation does include the code to walk through the abstract regions through weighted terrain, but when run in practice, an infinite loop of jumping between 2 region types was hit. More debugging is necessary to determine how to resolve this infinite loop. A guess to what is happening is that the previous location is not being handled at all and so it is jumping back to the previous region type as soon as it hits it.

Conclusion

Many different genres of games rely on the pathfinding of their games to have good performance, but also appear realistic to the player's eyes. Weighted terrain with different creatures having different behaviours is a good way to help the pathfinding feel realistic. Using dynamic terrain abstraction allows for this realism to also have good performance and update whenever terrain changes. It is definitely more difficult to implement than weighted A*, but when done correctly, as with (Sturtevant et al. 2019) it can have significant advantages.

Additional Information

Online Resources

A zip with the code, report and a few log files of the run have been provided. A read-me also exists in the zip with more details on how to run the make-file for the program.

- Logs of the runs are located: **AbstractionWithWeightedCosts/logs/**
- Executable is located at **AbstractionWithWeightedCosts/out/build/x64Release/AbstractionWithWeightedCosts.exe**.
- Read-me for how to run is located at **AbstractionWithWeightedCosts/readme.txt**.
- The GitHub repository containing the code and a runable executable can also be found at https://github.com/megsum/DynamicTerrainAbstraction.

References

Sturtevant, N. 2007. Memory-Efficient Abstractions for Pathfinding. 31–36.

Sturtevant, N. R.; Sigurdson, D.; Taylor, B.; and Gibson, T. 2019. Pathfinding and Abstraction with Dynamic Terrain Costs. In *Artificial Intelligence and Interactive Digital Entertainment Conference*.