**Chapter 4: Findings**

CHRISTEN FORD, Baldwin Wallace University

1. **Testing Environment**

The purpose of this section is to reintroduce the testing environment utilized in the thesis as well as the final list of software packages. Data presented here was collected on an MSI GT62VR 7RE laptop with the following specifications:

* Windows 10 Professional Version 1709
* Intel Core i7 7700HQ @ 2.80 GHz
* 16GB DDR4 2400 MHz Ram
* Samsung MZNLN256HMHQ 256 GB SSD Primary Drive
* Intel 535 480 GB SSD Data Drive
* Nvidia GeForce GTX 1070 8GB @ 1650 MHz

In addition, this thesis utilizes the following software packages:

* Project Malmo 0.31.0
* Python 3.6.4
* PyCuda 2017.1.1
* NumPy 1.14.0
* SciPy 1.0.0
* Pandas 0.22.0
* Seaborn 0.8.1

Finally, we have incorporated a hand built three-dimensional world and collision detection system, meant to simulate the Minecraft environment in a meaningful way. This was done as Project Malmo has no way to extract complete world states from the agent. It was also done to allow us to plan offline, as opposed to having the agent plan in real-time as the Minecraft simulation was running. We previously attempted this before building the world system and it caused minute but highly impactive timing errors to occur between the Minecraft server and client as well as the Project Malmo agent.

1. **The A\* Search Algorithm**

To gauge the performance of RRT we required a comparable algorithm to compare against. It was the decision of the team to utilize the A\* path planning algorithm for this purpose. A\* is a vastly popular path planning algorithm. It finds use in video game environments, movies, robotics, and many other forms of entertainment and electronics. A\* is an offline path planning algorithm. In this regard, A\* is run prior to the agent making any decisions. Traditionally, A\* would find a complete path from the agents starting state to the goal state. In real-time environments, A\* is modified to plan for a limited amount of time to find a state that looks the best to reaching the goal state. For this discussion we will discuss the offline, non-real-time variant of A\*.

The A\* search algorithm is a modified version of Dijkstra’s search algorithm, known in the literature as Dijkstra’s algorithm. Both algorithms work on the premise of creating a path that adheres to some constraint. The most common use case for these algorithms involves finding shortest-paths. This technique is employed in this thesis to find paths using A\* search. Where A\* and Dijkstra’s algorithm differ is in the way they evaluate neighbor nodes. A\* and Dijkstra’s algorithm both operate on the premise of a heuristic or best guess metric. Dijkstra’s algorithm utilizes a null heuristic, meaning the best guess distance to the goal node is always zero. The performance of A\* is highly dependent on the heuristic that is used. Finally, much of what has been done with path planning involves two-dimensional path planning. Many path planning algorithms can be generalized to n-dimensional space.

The classical implementation of A\* utilizes a priority queue (referred to in the literature as the open list or frontier) data structure to choose successor nodes. Upon initialization of the algorithm, the starting node is placed on the queue with priority zero. The algorithm then repeats the following procedure until either all nodes have been evaluated, or the goal is found. A\* utilizes three primary functions to generate paths. The first *f(n)* is determines the overall value of the node currently being evaluated.

For A\* to produce correct paths the heuristic used must be consistent and admissible. A heuristic in A\* search is said to be consistent if for every neighbor node of each vertex in the path the estimated distance remaining to the goal plus the cost of reaching the neighbor is less than or equal to the actual distance remaining to the goal. With the additional constraint that the goal node should always have a heuristic of zero. Mathematically, **consistency** can be represented as:

Where…

* *h* is the heuristic distance function
* *c* is the cost function that determines the cost of moving from one node to the next
* *n* is the node currently being evaluated
* *a* is the action taken at node *n*
* *n’* is the successor node generated from having taken action *a* at node *n*

For the condition of admissibility to hold true, the heuristic value must never overestimate the cost to reach the goal node from any node in the graph. The result of these two constraints will cause A\* to never evaluate unnecessary nodes, thereby producing the optimal path. Mathematically, **admissibility** can be represented as:

Where…

* h is the heuristic distance function
* c is the cost function
* n is any node in the search space
* g is the goal node

The heuristic and cost functions vary per use case. To successfully find paths, a reasonably appropriate heuristic function must be employed. The cost function can be implemented in such a way to encourage certain paths, while discouraging others, as was done in this thesis.

The heuristic and costs functions employed in this thesis appear as follows:

Where…

* n is a node in the search space
* n’ is any descendant of n
* g is the goal node

There are several distinctions in with these functions that were chosen specifically for this thesis. First, we weight the y-dimension in the heuristic function. Minecraft utilizes a three-dimensional coordinate system for placement of objects and entities. The x and z axes determine horizontal position, while the y axis determines vertical position. This led to a natural adaptation of the heuristic function where we bias the y-axis to ensure the agent properly traverses between floors. However, there are some issues with this approach.

Our 3D A\* implementation suffers from the locality problem. Wherein, the agent will become locally trapped on floors due to the way neighbor evaluation is handled. We have corrected for this situation with careful design of the maps utilized in the thesis, but because of this problem, the implementation is not fully portable to other maps. Another potential solution for this problem would involve increasing A\*s’ scanning range (the area around the agent where neighbors are evaluated). In the current implementation, the agents’ scanning range is 3x3x3 wherein the agent evaluates only adjacent cells on the floor below, the current floor, and the floor above it for a total of twenty-six cells (the current cell is never put up for evaluation). Increasing the scanning range would dramatically increase the number of evaluated cells thereby increasing evaluation time.

The other problem that our A\* implementation suffers from is tie-breaking. We do not break ties in this implementation. In cases where there are multiple shortest paths (as is the case in the last map), the agent will consistently choose the first shortest path it comes across and will never utilize any other. A possible solution to this problem would involve a multi-step process.

First, we evaluate all neighbor nodes for their f-score. After this we are presented with a few options. We could just randomly choose from the best nodes. However, this would require us to iterate through all the successors twice. On the first iteration we calculate all f-scores while simultaneously finding the max f-score. On the second iteration, we mark all successor nodes that match the maximum f-score we found during the previous iteration. We would then randomly pick a successor from this sample.

The second solution involves modifying the h-score to act as the tie-breaker. This is as the h-score times the sum of one plus a weighting factor *p*. We then proceed to calculate the f-scores. Across the literature, this is the more common way of breaking ties with A\*. The issue with this approach is that it risks creating an inadmissible. To reduce this risk, the literature recommends:

1. **The Rapidly Exploring Random Tree Algorithm**

A rapidly exploring random tree (RRT) is a randomized path planning algorithm. The algorithm is initialized with a starting point, a maximum distance between nodes, *epsilon*, and a maximum number of iterations. At each iteration, a point, *rand*, is randomly sampled from the search space. Then for each node, *p*, already in the tree, a distance function is used to find the nearest node, *nn*, in the tree to the randomly sampled point. A line-to function is used to determine whether to attach *nn* to *rand* or to generate a new point along the line created from *rand* to *nn*. This functionality is based on whether the distance between *rand* and *nn* is less than *epsilon*. If so, *rand* is used, otherwise the newly the algorithm generates a new point from *nn* to *rand*.

Our implementation differs from the standard implementation of RRT by utilizing NVidias’ CUDA library to accelerate the distance function using a discrete graphics card (GPU). In our implementation, the distance between *rand* and all neighbors contained in the tree is calculated simultaneously on the GPU. We utilize the Numpy and PyCuda libraries to make working with CUDA accessible within Python. We utilize the PyCuda ElementWise Kernel which allows for the simultaneous application of some predefined function to a given data set. The result of this kernel is a two-dimensional array of distances from *rand* to all the points in the tree. We then flatten the array to a single dimension and treat it as a parallel array of the tree. We then find the min index in the array and utilize the corresponding node in the array containing the tree as *nn*. We call this implementation RRT-GPU.

The Python implementation of this agent has some flaws that require discussion. First, the overhead incurred by utilizing the PyCuda library and Numpy dominate any potential gains from accelerating RRT over smaller maps. Second, the CUDA requires precisely aligned data sets to work with that adhere to strict standards based on the users GPU specifications. This affects the thesis in two ways: First, as it stands, the direct implementation of this thesis is not 100% portable to other hardware setups without some additional configuration; Second, due to the requirements of CUDA, GPU acceleration is only utilized every eight iterations of the algorithm. If this was not followed, the GPU would have to deal with jagged arrays that would require a complex striding algorithm to properly apply the distance function. This translates to increased run times for RRT-GPU.

This also impedes the algorithms ability to maintain and reconstruct paths. RRT-GPU keeps track of two arrays, an array containing the tree, and an array containing the distances as calculated by the ElementWise Kernel. PyCuda allows us to send and receive C structs to and from the GPU. However, this requires a custom Python class that contains a specialized wrapper. We did not opt for this approach and instead implemented a function in our Node class that converts the position contained within the node to a float3 vector object PyCuda can work with.

1. **A\* Results**

As mentioned prior, A\* was to serve as a benchmark of comparison that would allow us to draw conclusions from RRT-GPU. We expected A\* to produce similar results across all four maps. Aside from the runtime, this is what we found. What will follow is a presentation of the raw data of each map as well as a statistical summary of each with a discussion that follows. In each of the following discussions we will present a small

**Map One: Flat Land Pathing Agent**

In this map, the agent is placed on the same level as the goal with no variations in elevation. This map was to serve as a comparison for all the other maps, as it is the easiest to path since it is just a two-dimensional planning problem. Following is a sample of the performance of the A\* agent on this map.

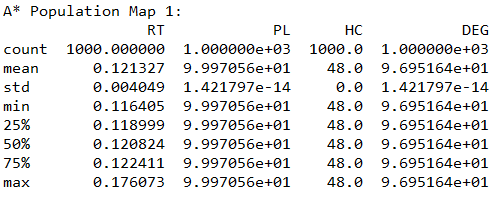


Figure 1: Map One A\* Population Data

**Map Two: Climbing Agent**

In this map, the agent is placed two levels below the goal. The goal sits on an elevated platform surrounded by climbing blocks. Other than that, the goal is placed in the same relative position as the first map. Presented is a sample of the results of this map categorized by run time, path length, heading changes, and total degrees in the path.

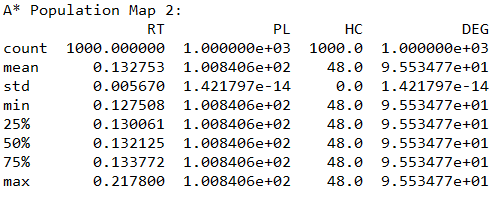


Figure 2: Map Two A\* Population Data

**Map Three: Descent Agent**

This map is an inverse of the second map and follows a similar layout. Rather than being elevated two levels above the agents starting position, the goal is placed two levels below the agent. Following is a sample of the performance of the A\* agent on this map:

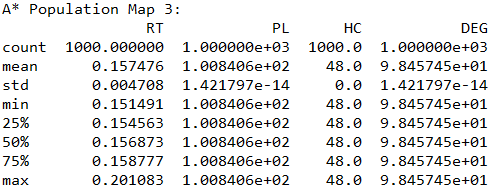


Figure 3: Map Three A\* Population Data

**Map Four: Mixed Agent**

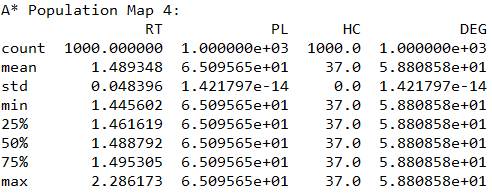


Figure 4: Map Four A\* Population Data

1. **RRT Results**

We provide data on the CPU bound version of real-time rapidly exploring random trees so that we may draw statistical conclusions between the CPU bound and GPU bound versions.

**Map One: Flat Land Agent**

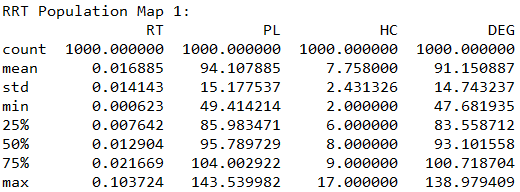


Figure 5: Map One RRT Population Data

**Map Two: Climbing Agent**

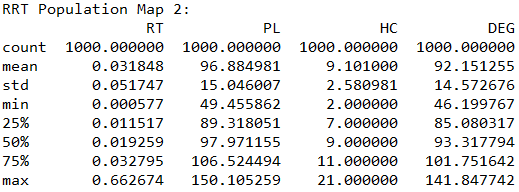


Figure 6: Map Two RRT Population Data

**Map Three: Descent Agent**

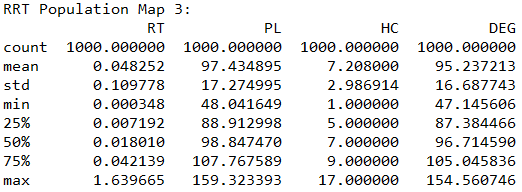


Figure 7: Map Three RRT Population Data

**Map Four: Mixed Agent**

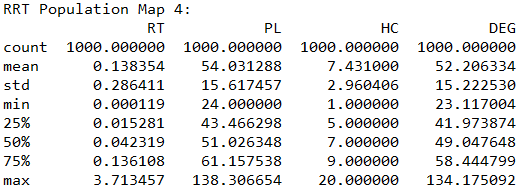


Figure 8: Map Four RRT Population Data

1. **RRT-GPU Results**

**Map One: Flat Land Agent**

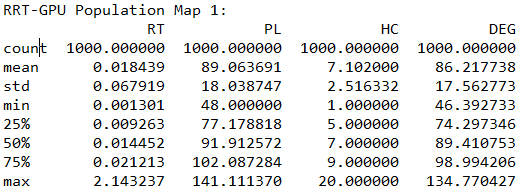


Figure 9: Map One RRT-GPU Population Data

**Map Two: Climbing Agent**

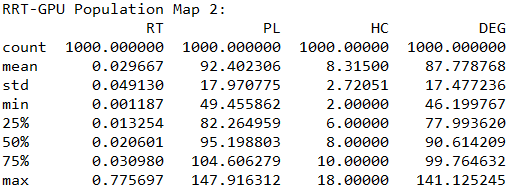


Figure 10: Map Two RRT-GPU Population Data

**Map Three: Descent Agent**

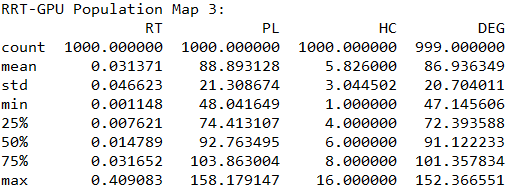


Figure 11: Map Three RRT-GPU Population Data

**Map Four: Mixed Agent**

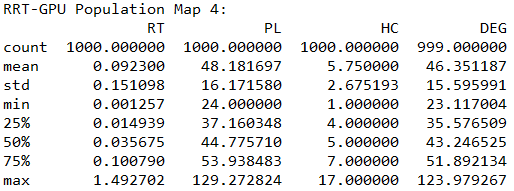


Figure 12: Map Four RRT-GPU Population Data

1. **Statistical Analysis**

All analysis conducted in this section will be performed utilizing the data from the RRT and RRT-GPU algorithms. A\* is excluded from this process and serves only as a benchmark to gauge the performance of the two RRT algorithms. Note that all tests were conducted utilizing populations of with n=1,000 across all four maps. In addition, we let RRT represent population one and RRT-GPU represent population two.

**F-Test for Equality of Variance: Two Populations**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Map One: Flat Land Agent (α=.05)** | | | | |
|  | **Hypothesis** | **Rejection Region** | **F-Statistic** | **Conclusion** |
| **Run Time** |  | F < 0.901 | 0.0043 | Very Strong Evidence to Reject the Null |
| **Path Length** |  |  | 0.708 | Weak Evidence to Reject the Null |
| **Heading Changes** |  |  | 0.934 | No Evidence to Reject the Null |
| **Total Degrees** |  |  | 0.705 | Moderate Evidence to Reject the Null |

As can be observed here, there is a very strong indication we should reject the null for the F-Test for run time. This suggests that the variance of RRT for map one is significantly different than the variance for RRT-GPU. Likewise, there is weak evidence to suggest a difference in variance for path length and moderate evidence to suggest a difference in variance amongst total degrees. Surprisingly, there is no evidence to suggest a difference in variance with the total number of heading changes.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Map Two Descent Agent (α=.05)** | | | | |
|  | **Hypothesis** | **Rejection Region** | **F-Statistic** | **Conclusion** |
| **Run Time** |  | F < 0.901 | 1.109 | No Evidence to Reject the Null |
| **Path Length** |  |  | 0.701 | Moderate Evidence to Reject the Null |
| **Heading Changes** |  |  | 0.9 | No Evidence to Reject the Null |
| **Total Degrees** |  |  | 0.695 | Moderate Evidence to Reject the Null |

Unlike map one, the F-Test for run time shows there is no evidence to suggest a difference in variance for map two. Note that this is what we expected to happen. The other three metrics generally follow along with the results from map one, although there is slight stronger evidence to suggest a difference in variance with the path lengths on map two as opposed to map one.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Map Three Climbing Agent (α=.05)** | | | | |
|  | **Hypothesis** | **Rejection Region** | **F-Statistic** | **Conclusion** |
| **Run Time** |  | F < 0.901 | 5.544 | No Evidence to Reject the Null |
| **Path Length** |  |  | 0.657 | Moderate Evidence to Reject the Null |
| **Heading Changes** |  |  | 0.963 | No Evidence to Reject the Null |
| **Total Degrees** |  |  | 0.65 | Moderate Evidence to Reject the Null |

We received the same results with map three as we did with map two.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Map Four Mixed Agent (α=.05)** | | | | |
|  | **Hypothesis** | **Rejection Region** | **F-Statistic** | **Conclusion** |
| **Run Time** |  | F < 0.901 | 3.593 | No Evidence to Reject the Null |
| **Path Length** |  |  | 0.933 | No Evidence to Reject the Null |
| **Heading Changes** |  |  | 1.225 | Moderate Evidence to Reject the Null |
| **Total Degrees** |  |  | 0.953 | No Evidence to Reject the Null |

Map four is the most different geographically from the other maps. We expected different results from the F-Tests conducted on this map then we did the other maps. Just like maps two and three we found no evidence to suggest a difference in variation for run time. However, both path length and total degrees are opposite from the results obtained in maps two and three. We specifically designed this map to force the agent to at least follow a specific path (the implementations of RRT and RRT-GPU do not explicitly force the agent to take a specific path).