COMP 521: Modern Computer Games
Project Report

The Use of Visibility Coverage for Making Smarter Decisions in A* Pathfinding on a Hexagonal Grid

ADIBPOUR Nima, VALET Matthieu

April 9, 2017

McGill
School of Computer Science

1 Introduction

1.1 Main objectives and overview of methodology and results

The main target of this project was to be able to experiment with ways that we could make A* pathfinding in a smart fashion, and experiment with the impacts of several optimizations that can be applied to the algorithm on hexagonal grids.

Our initial attempt started with finding a way to use visibility in the form of cameras placed throughout our static map.
This was to be used as a tool to detect a moving agent, detect the path that it has taken to its target, and use a smart approach into finding the best and quickest interception point for another agent to path-find to, which would simulate a "cop-robber" chase sort of situation, so that the cop does not have to attempt to chase the robber from across city, but to rather find a smart interception point along its path.
We achieved our goal at being able to simulate this but the issue was that it was just too slow. Pathfinding in a tightly packed map like the one we chose, with many closely located obstacles, proved to be not as fast as we needed it to be to be able to quickly find the smartest point along the robber's path to mark for interception.
As a result, we have attempted several explorations and optimizations, including the use of different data structures for our pathfinding, multi-threaded processing, and an exploration of the Jump Point Search extension onto A*, applied on hex grids.

The results we achieved with our research was fascinating as it involved several optimizations (JPS with many obstacles) not proving as efficient on hex grids, as well as multi-threading issues that forced us to finding a better alternative for reducing the number of pathfinding requests at runtime to be able to speed things up.

1.2 Work distribution

Nima:

- Implementation of hex grid
- Implementation of A* and adaptations needed for hex grid
- Production of heat maps
- Binary Heap optimization
- Multi-Threading
- JPS Extension onto A* and integration with hex grids
- Implementation of regions across the map using node weights
- *Attempts at making the pathfinding smarter*

Matthieu:

- City map and unit spawning
- Visibility algorithms and detection of robber through cameras

- Collision handling between robber and cop
- *Attempts at making the pathfinding smarter*

2 Game Context

In our static city map, a cop must protect the bank. Unknown to him is a robber walking in the shadows to try to rob the bank.
Cameras are scattered throughout the main city roads and will report back to the cop if any robber is seen. The influence of visibility is as follows: it will affect the cop's pathfinding to intercept the robber in time before he reaches the bank.
The objective of this project is to implement these ideas so as to make pathfinding more intelligent and realistic given the visibility of city cameras. Below is an image of our static map (*Fig.1*).

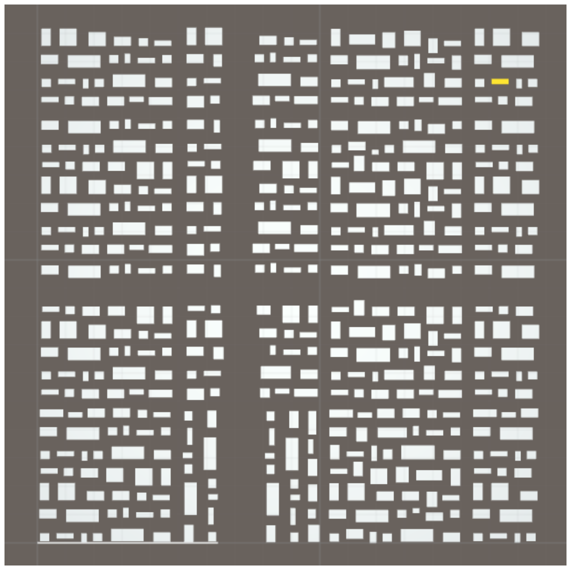

Figure 1: Static map

3 Background

3.1 Pathfinding (A*)

A* pathfinding is known to be more efficient than other classic pathfinding algorithms such as DFS or Dijkstra's (with modifications) algorithm.
However, for real time simulation, it is proven that A* pathfinding is known to still be considered slow as there are many nodes that are expanded prove to not be necessary to find the optimal path to the target as a lot of them represent symmetrical movements from an intermediate point A to another intermediate point B.
This means that given the limitations of this algorithm in terms of speed and performance, we wanted to start exploring the different ways with which we can analyze a path and make a decision in a smarter fashion regarding where along the path is the best location to intercept another agent such that the second agent attempting to catch the first is not chasing from behind, but manages to (efficiently) find a cut-off point to be able to intercept the path.

We address efficiency issues through minor attempts at changes in the algorithm with improvements in the data structures used, as well as exploration of a possible integration of the JPS algorithm onto hex grids to reduce the number expanded nodes[1][7] throughout the pathfinding process.

3.2 Visibility

Visibility is one of the fundamental aspects implemented into a game when making realistic, intelligent NPCs.
Indeed, it is used to define what an agent can or cannot see. In multiplayer games, for instance, situations will arise where a target will be *out of sight* and the NPC or player will be unable to interact with it.

This crucial game aspect will be implemented in our project in two ways: first, for the police officer who will arrest any robber it sees ; second, for city cameras to serve as extended eyes of the police and monitor any robber it sees, communicating that information to the police.

4 Methodology

The implementation of this project as well as the method that we approached it started simple to be able to be handled and we expanded from there, exploring the improvements in efficiencies that could be made to make our work fast.
We have also used GitHub and related software to be able to collaborate as a team on this so that our branches of exploration would not interfere with one another, yet be able to merge the work into one final product.
The implementation starts off as a pretty simple extension of A* onto the concepts of hexes as well as the implementation of cameras that act as radar units throughout the city to be able to detect the entry of a unit into a certain region and react accordingly. The details of each part of our implementation follows.

4.1 Hexagonal Grid

Hex grid are traditionally known to be an interesting alternative to square grids for representation of nodes

that represent a terrain or simply ground of an area in games.
What makes them interesting is the underlying geometry that simplifies neighboring structures and the heuristics such that the distance from a given hex to any of its neighbors is the same, regardless of direction, which is not the case with square grids as a diagonal movement on a square grid results in a larger distance.
After reading through Amit's[4] detailed guide on how hex grids form, we attempted at implementing our version of this in Unity.
The major difference between our implementation and the pseudo design represented in Amit's guide is that we had to find a way to visually construct a hex tile in unity, as it was not considered efficient to use 3rd party software made models for our representation given the large number of nodes that we wanted to test on; there would simply be way too many meshes to be rendered and that could affect performance while not being the primary area of exploration of this project.

So with that in mind, we attempted at creating hex meshes in Unity using vertices and constructing 6 triangles together to form the basic face of a hex tile.
We then expanded on that to be able to convert the 2D grid, offset representation of a hex within a grid, into the cube coordinates we have used to be able to later use cube coordinates for our distance heuristics of the pathfinding. We could ultimately ignore cube coordinates within a cell, but then it would be required to convert the offset coordinates of a node within the grid to cube or axial coordinates for each node explored in our pathfinding and that would be less efficient than calculating the values needed beforehand.

4.2 A* pathfinding without any optimizations

The first goal set was to be able to integrate the classic A* that we knew from square grids into hex grids. This goal was easily achievable through modifications on the get-neighbors function for each node, as well as changing our distance function that determines the distance between two hex tiles. For that, we have used the cube coordinate distance function in Amit's guide on Hexagonal Grids:

```
function cubeDistance(a, b):
  return max(abs(a.x - b.x), abs(a.y - b.y),
     abs(a.z - b.z));
```

4.3 A* With Heap Optimization

It seemed like every implementation of the A* algorithm on square grids that we observed had used a binary heap (sometimes referred to as priority queue) for the main data structure used to store the open list of the algorithm.
This makes sense as iterating through the whole list of nodes (in the worst case) to find a node with a lower F Cost compared to the current node is very redundant and it could be done using binary heaps to always store the node with the lowest F Cost on top of the binary heap tree, such that finding that element is done in constant time and inserting a node at its correct location within the tree can be done at log(n) time.
With that in mind, we attempted at an implementation of the binary heap data structure that would suite for our application and with that in place we achieved a reduction of 3-4 times in runtime for each pathfinding request between a start and end node (see Results section).
However, a very interesting (yet accidental) observation that we made throughout this implementation was that due to a bug in our program, we somehow managed to visit much less nodes using a list than using a binary heap. A brief summary of this is explained in the Results section.

4.4 Smart decisions and Multi-Threading

For the cop to find the best interception point along the path of the robber, it was necessary for the path to be analyzed first.
Our first, and most inefficient, approach was to take all the remaining nodes on the path of the robber to its target, and attempt to find a path to each of them, such that only a path that had a shorter length from the cop to the interception node than the robber to the same node was accepted, and then the rest would be discarded.
However once we decided to put our pathfinding calculations on separate threads to be able to take more advantage of multi-threaded CPU calculations, we realized the real problems with this.
For the pathfinding algorithm to not show irregular behavior, the path request manager locks the results queue (where the resulting paths of a pathfinding requests are stored) until the result has been processed and this means that other pathfinding requests are not able to put their results for evaluation onto the queue upon completion until the given result is processed.

4.5 JPS Extension

The attempt at implementing an algorithm meant for square grids onto our hexagonal grid has not failed to prove itself interesting.
On square grids, with an 8-way traversal fashion, there are four diagonal and four cardinal directions that a node can reach into one of its neighbors.
Comparing that to a hex grid, given the 6 neighbor structure that the grid forms, we observed that we could have up to 3 diagonal, and 3 cardinal directions. The

choice of which directions to take as diagonal and cardinal seems arbitrary given the following strict condition that no two cardinal directions can be consecutive, with the same applying to diagonal directions.

We represent directions on a hex grid as NE (north-east), E (east), SE (southeast), SW (southwest), W (west), and NW (northwest). With that, we have chosen NE, SE, and W as the cardinal directions and E, SW, and NW as the diagonal directions.
Since diagonal movements are always preferred in the JPS extension, first we check for those. Diagonal movements, however, are composed of checks for possible cardinal movements. For each diagonal movement, we check if we are opposed with forced neighbors through blocked cardinal movements.
So for a given diagonal direction (assuming indices of 0 to 5 for directions), we check if a move with a direction index less than and a direction index greater than the current direction index was blocked to detect corners. If a corner was detected, we add the current hex along the diagonal line to the open list. We then restart the expansion by moving the current hex from the open list to the closed list, and expand from there by checking the two possible cardinals.

This results in moving diagonally until you reach another corner. For the cardinal checks, we also check for forced neighbors and corners. The idea is to move in the same cardinal direction until an obstacle is reached, and that means we have to expand on the next hex in the open list. If we reach a forced neighbor, it means we are at a corner, and we need to add the current hex, and expand from there. We end the expansion and start from the next hex in the open list.

4.6 Visibility design

The purpose of visibility in our project is for city cameras and cop to detect a robber. This simple description requires methodology to achieve the results we want.
We first started by listing the necessary working features of our camera: it must detect a robber up to a certain range, with a specific angle and have its field of view (FOV) be blocked by obstacles.

4.7 Visibility Mesh

The first thought is for our visibility tool to be a Mesh. This is done by having a camera have a 'Visualization Mesh' as child. The shape and color of this Mesh will be determined by a radius, an angle and a color. This first step can be seen in the image below (*Fig.2*).

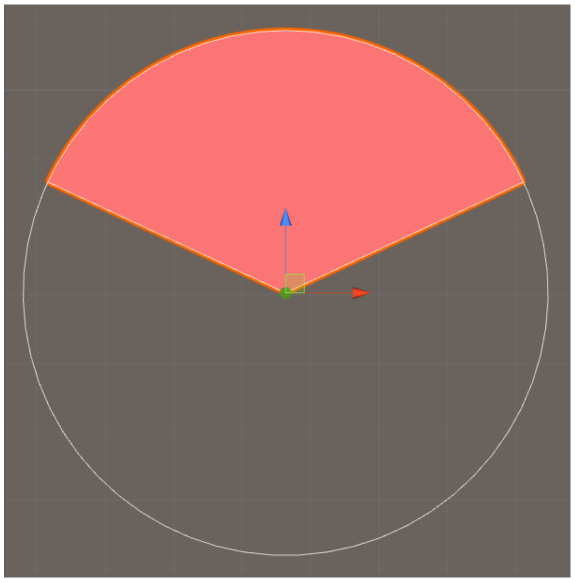

Figure 2: Basic prototype for visibility

This design makes it simple and fast to change its radius and angle to work well with any features of our environment.

4.8 Target detection

Now that the Mesh is established, we must record any collision between itself and a robber. This means that we must add a 'Robber Mask' as well as a collider to the robber so that the collision can be handled.
Below is an image of a FOV with multiple robbers (*Fig.3*).

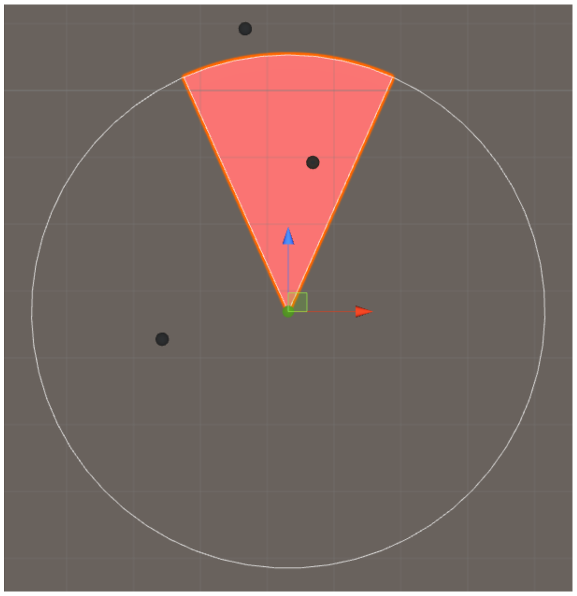

Figure 3: Target detection

Notice how only one robber is in the shown FOV, meaning that, in theory, we would have no idea of the other

two's existence. Should we want to keep track of multiple robbers in the scene, we made it so that the robbers are detected as a list of 'visibleTargets' instead of only one 'GameObject'.
Although this is a good start, the bulk of our work visibility is yet to be done: polygons interfering with the view, which will be discussed in the following section.

4.9 Polygon FOV interference

To quote our main paper on visibility, "The boundary of a simple polygon P consists of a sequence of straight-line edges such that they form a cycle and no two nonconsecutive edges intersect"[9]. From this, we can deduce that continually raycasting (e.g., every 5° of our FOV) is a sound and viable option. These raycasts will be in either of two states: stopped by a polygon P, or unstopped and going the full length of the FOV's radius.
Below is a snippet of code showing exactly that concept:

```
if (Physics.Raycast (transform.position, dir, out
    hit, viewRadius, obstacleMask)) {
   return new ViewCastInfo (true, hit.point,
       hit.distance, globalAngle);
} else {
return new ViewCastInfo (false, transform.position +
    dir * viewRadius, viewRadius, globalAngle);
}
```

Comparable to our second reference for visibility[10], we get the following image (*Fig.4*).

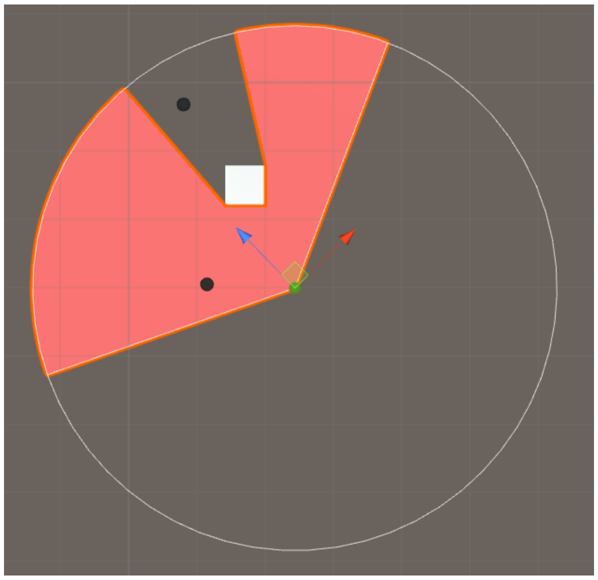

Figure 4: Polygon interfering with FOV

We can observe that, in the above image, only one of the two robbers is seen by our camera as it is in the FOV and the other is hidden *behind* an obstacle. Although this could be thought of as a good prototype, there is an optimization left to perform: making detection of the end of en edge sharper.

4.10 Optimizing end vertices detection

Below is an image of end vertices detection without optimizing the Mesh Resolution (*Fig.5*).

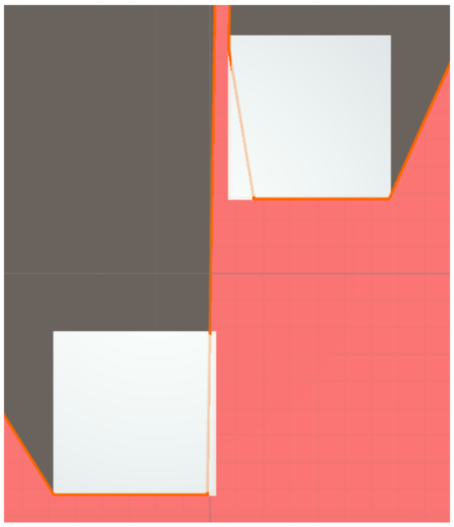

Figure 5: End vertices before optimization

We can clearly see how the general shapes of polygons are respected. However, the end vertices are not well detected.
This can be fixed by creating a variable called 'Mesh Resolution' that stores the definition with which to detect those end vertices. The following snippet of code is that very optimization, with an int variable which stores the angular step at which we are raycasting and a boolean variable which returns true if we have raycasted beyond an end vertex.

```
int stepCount = Mathf.RoundToInt(viewAngle *
    meshResolution);
[...]
bool edgeDstThresholdExceeded = Mathf.Abs
    (oldViewCast.dst - newViewCast.dst) >
    edgeDstThreshold;
```

After some trial and error, the first values to give accurate and sharp results are a meshResolution = 10 and an edgeDstThreshold = 4. With this optimization, the resulting detection is much sharper and accurate, as can be seen in the image below (*Fig.6*).

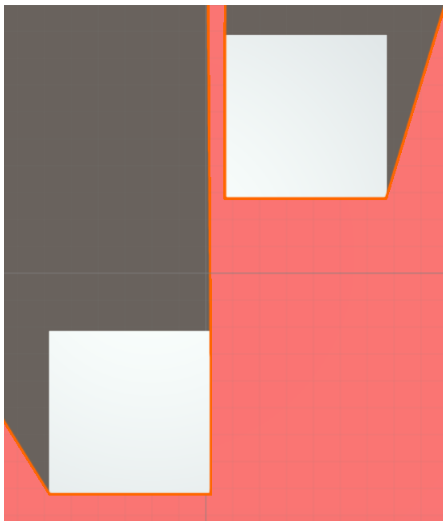

Figure 6: End vertices after optimization

We can see that the end vertices are now being correctly detected, and that the 'Visualization Mesh' is much closer to the actual scene given the polygons it hits.
A good way to think about this optimization is to draw the added raycast from the center of FOV to one of the end vertices that is detected. This new raycast is a good representation of this optimization.

4.11 Implementing FOV components into the city

Now that our model corresponds to our expectations of how FOV should work, we will implement it into our map. In addition to placing the camera, this process requires the addition of the 'Robber' Mask to our Robber prefab so that it will be detected as a visible target upon entering any FOV Mesh. Similarly, all obstacles must receive the 'Unwalkable' Mask so that they serve properly as obstacles in our FOV. Below is an image showing how FOV looks in our final static map (*Fig.7*).

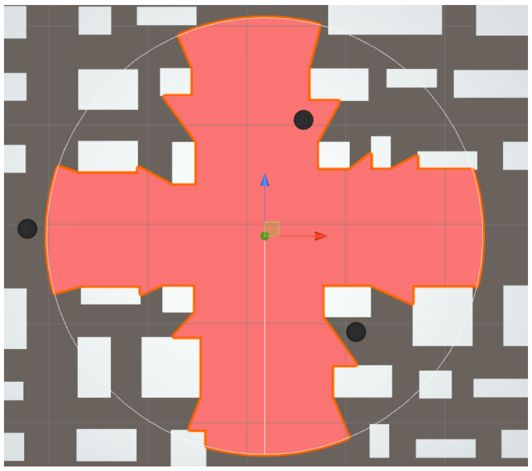

Figure 7: Optimized FOV integration

We can see, in this integrated example, all the components of our project's visualization aspect. Out of three robbers, one is outside the range of FOV, one is hidden by an obstacle, and the last one is the only one to be detected.

5 Results

5.1 A* with Heap Optimization

Implementing a binary heap data structure has indeed proven to make the pathfinding much quicker, such that our project's goal could be achieved with a more reasonable calculation runtime.

The A* pseudo code indicates a check for a node with lowest F Cost[5] throughout the open list as:

```
current := the node in openSet having the lowest fScore[] value
```

And we implemented this with a skip on nodes with higher F Cost, and used the H Cost as a tiebreaker on nodes that had equivalent F Cost.To skip the nodes with the given above conditions, in C we had to implement:

```
var currentHex = openSet[0];
for (int i = 1; i < openSet.Count; i++)
{
    if (openSet[i].FCost >= currentHex.FCost &&
        openSet[i].FCost != currentHex.FCost)
        continue;
    if (openSet[i].HCost < currentHex.HCost)
        currentHex = openSet[i];
}
```

However, due to a slight bug in our implementation, at the first attempt we wrote "... && openSet[i].FCost == currentHex.FCost) continue;". This decreased the number of nodes visited to find a path from start to target, drastically, as shown below:

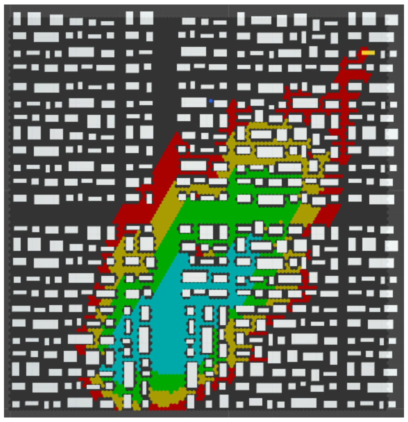

Figure 8: Correct A* implementation

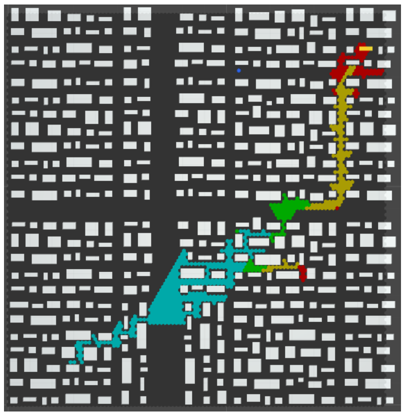

Figure 9: Wrong A* implementation

Given this, we observed for an unknown reason that we have not been able to figure out why, if we consider nodes with both less than and greater than F Cost than the current node, it appears much less nodes are considered in overall.

	time (in ms)
A* Runtime with list	129
A* Runtime with heaps	31
A* Runtime with list (with our bug)	33

The runtime for this dropped from ∼140 ms to find a path from the robber to the bank, to about ∼ 30 ms, which in fact competes with the heap implementation in terms of runtime and it ends up visiting much less nodes in the grid.

5.2 Smart decisions and Multi-Threading

While we attempted to use threads to spread out several requests on several threads to take advantage of processing power, the final outcome was (due to the lock), turned out such that our program froze until all those requests were handled and resolved, and then the best one would proceed.
This obviously denoted a huge drop in performance, and optimizations had to be made such that smarter and better paths would be requested first. So with that in mind, we decided that instead of searching through the remaining nodes of the path of the robber to its target location from the start or the end of the path, we would search from the node closest to the available cop.
We would analyze that path once found, and only if the path proved to not be an optimal one in terms of a possible interception, we request another path to be found, descending through the list of locations to path-find to with respect to their true distance from the cop. Indeed, this showed us that we could find a path to the best interception node much faster with much less extra paths analyzed.

5.3 JPS Extension

The results of this integration with hex tiles has proved to us that indeed JPS seems like a much better alternative than the base A* although it does so only in open spaces. Given our 100×120 grid size, JPS has only proven to be more efficient in cases with few obstacles between the start and end tiles as shown below:

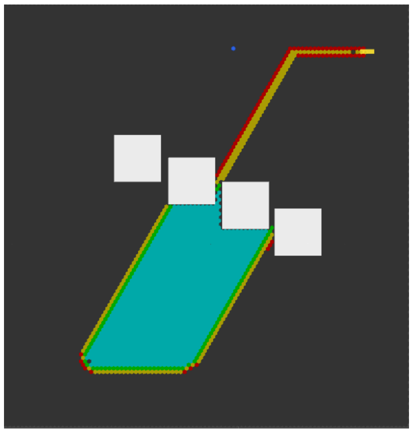

Figure 10: Heat map on nodes explored by A*

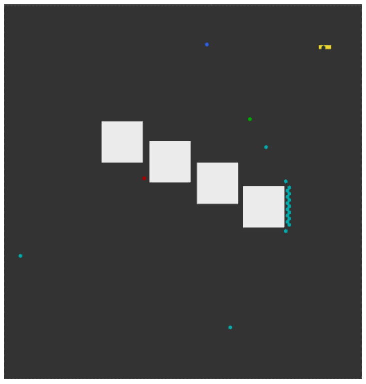

Figure 11: Heat map on nodes explored by JPS

Here is the runtime comparison with few obstacles (DT stands for data structure in the following table):

	time (in ms)
A* with List DT for OpenSet	113
A* with Heap DT for OpenSet	47
JPS with List DT for OpenSet	21
JPS with Heap DT for OpenSet	27

As visible in the images, far fewer nodes are visited by JPS in comparison to A*, when there is a lot of open space for exploration, and the runtime is dropped to half of what the heap optimized version of A* is able to reach.

However, it can be seen that due to the fact that there are very few nodes in the open set of JPS, there is not much optimization made with the use of heaps for the data structure behind the open set and perhaps searching through all (few) nodes of the open list by JPS is actually faster than sorting up/down the contents of the binary heap when there are such few nodes to consider.
However, when the map becomes populated with more tightly packed obstacles, JPS proves itself not much more efficient than the basic A*.

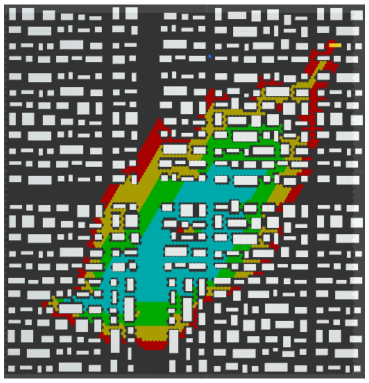

Figure 12: Heat map on nodes explored by A*

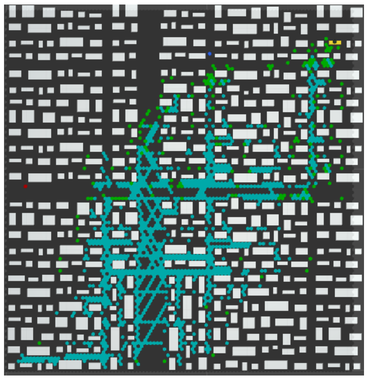

Figure 13: Heat map on nodes explored by JPS

It shows that due to the scattered nature of the nodes with a lot of unexplored nodes throughout its journey, JPS is exploring less nodes even with tightly packed obstacles. However, the runtime comparisons are where its performance truly drops with many obstacles (DT stands for data structure in the following table):

	time (in ms)
A* with List DT for OpenSet	145
A* with Heap DT for OpenSet	28
JPS with List DT for OpenSet	148
JPS with Heap DT for OpenSet	159

This was quite expected as the algorithm is has been known to be more efficient in open spaces and certainly its implementation on hexagonal grids is no exception. In both tests (few and many obstacles), heaps have proven to not optimize JPS at all, and indeed the list structure seems to be the better way of implementation when it gets to JPS.
We have also decided to not pursue the use of JPS at the state that we have implemented it, for the primary goal of our project, due to the fact that a path recovery is needed once JPS is calculated. The hexes that are considered as the "path" in our implementation are very scattered and this results in our agent jumping between points due to our discrete movement.
It does not provide a smooth path, and indeed to fix this we would need to further explore on JPS by recovering the nodes that have been skipped by the pathfinding algorithm between any two consecutive hexes of the path, such that all hexes in between are considered for the agent's traversal as well.

6 Conclusions and Future Work

We have managed to achieve the primary goal of this project, being able to make a smart decision on which interception point along a path to take, such that it can be done at a reasonable real-time simulation environment.

The optimizations that we have made throughout the project have proven that they are necessary given the limitations of A* and the high number of nodes that the algorithm visits until it is able to come up with a decision regarding the path it chooses as valid. However, we believe that the JPS extension on hex grids definitely deserves extra work to be able to figure out how to optimize it further, perhaps with a data structure that can make it compete better with A* in the case of tightly packed obstacles.
It might even be better to consider preprocessing a lot of the calculations that, as explained in Steve Rabin's goal bounding extension onto the JPS+ algorithm [8].
Another area in which we could extend this exploration is experimenting with continuous movement of agents, as such would require more advanced calculations regarding the time that it takes for an agent to travel between two points, as now it was assumed that since on every tick each agent moves 1 unit forward, the travel time is of

an integer value that is easy to compare against another agent.
Another interesting and more real-life application of this would be such that the robber does not have one fixed goal every time, and that the cameras are not aware of that goal; instead, as multiple cameras throughout the city detect the robber, they would constructively form a prediction of its final destination by discarding the possible target locations that are not valid given the robber's velocity and direction of movement, as well as the region in the city that it was detected in.
With that, a much smarter decision making of where to navigate the cop to would be made, and it would also allow for experimentation with multiple cop and robber units throughout the city, with the closest available cop for each pursuit chosen for interception.

7 References

[1] Harabor, D., & Grastien, A. (2011). Online Graph Pruning for Pathfinding on Grid Maps. NICTA and The Australian National University. Association for the Advancement of Artificial Intelligence

[2] Witmer, N. (2013, May 5). Jump Point Search Explained. Retrieved April 6, 2017, from http://zerowidth.com/2013/05/05/jump-point-search-explained.html

[3] Patel, A. (1997). Amit's A* Pages From Red Blob Games. Retrieved March 30, 2017, from http://theory.stanford.edu/ amitp/GameProgramming/

[4] Patel, A., Fu, C., & Verbrugge, C. (2013, March 13). Hexagonal Grids. Retrieved April 01, 2017, from http://www.redblobgames.com/grids/hexagons/

[5] A* search algorithm. (n.d.). Retrieved March 30, 2017, from https://en.wikipedia.org/wiki/A*_search_algorithm

[6] Adamchik, V. S. (2009). Binary Heaps. Retrieved March 30, 2017, from https://www.cs.cmu.edu/ adamchik/15-121/lectures/Binary%20Heaps/heaps.html

[7] Harabor, D., & Grastien, A. (2012). The JPS Pathfinding System. NICTA and The Australian National University. Association for the Advancement of Artificial Intelligence

[8] Rabin, S. (2015). JPS : Over 100x Faster than A*. Retrieved April 6, 2017, from http://www.gdcvault.com/play/1022094/JPS-Over-100x-Faster-than

[9] B., Joe, and Simpson R. B. 1985. *Visibility of a Simple Polygon from a Point.* Waterloo: University of Waterloo

[10] C. Nicky, 2015, "Sight & Light, How to create 2D visibility/shadow effects for your game" http://ncase.me/sight-and-light/