An Application of Collaborative Diffusion to Smart Parking Systems

Nicholas Bedford
School of Computer Science
Newcastle University
England

ABSTRACT

In this paper, collaborative pathfinding is explored as a solution to traffic congestion in metropolitan areas by looking into its possible applications to smart parking. The smart parking systems that have been utilised in the past typically rely on some derivative of A*. By expanding on the concepts introduced in other research, this project seeks to show the validity of using collaborative diffusion as a pathfinding solution by creating a simulation of the metropolitan area of Newcastle where AI agents avoid each other to reduce congestion.

ADDITIONAL KEYWORDS AND PHRASES

Smart Parking, Collaborative Diffusion, Collaborative Agents, Game AI, Multi-agent Architecture, Collaborative Pathfinding

1 INTRODUCTION

1.1 Smart Parking

Since other solutions to heavy traffic in metropolitan areas, such as park and ride systems, have been proven to be limited in their success to alleviate the congestion problems of city centres [1] other solutions are required to prevent excessive traffic build up.

Drivers searching for an available parking space cause more than 30% of the traffic congestion generated in most modern cities [2]. France for example has reportedly had many Parisien and Lyonnais drivers abandon their trips due to the excessive time involved in searching for available parking, which causes a negative impact to the economy by reducing the cities' potential revenue stream[3]. With parking generating a positive cash flow for most cities, many parking technologies have provided a promising solution to both their parking and potential economic problems.

Many cities try to alleviate their parking issues by utilizing Parking Guidance and Information (PGI) systems to assist drivers in finding available parking spaces within the city[4, 5].

PGI systems are important to cities that employ them, as they attempt to avoid traffic issues found in other large metropolitan areas, by reducing the number of drivers manually searching for available parking spaces. Most PGI systems monitor parking spaces by using devices to automatically detect when spaces become available, assign a driver to a specific available parking space[6, 7, 8] and then direct the driver to the location of the designated space.

1.2 Collaborative Pathfinding

Pathfinding is a way for AI agents to navigate their environment. There are multiple ways of creating this behaviour and extensive research has been carried out to optimize the procedure as much as possible. Most forms of pathfinding are built on the premise of nodes that are points in space that the AI agent can travel to in order to traverse the environment.

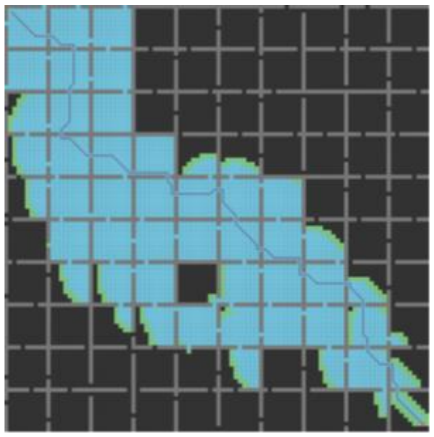


Fig. 1. An example of an A* pathfinding algorithm in action. The dark blue line representing the path generated by the algorithm, the light blue squares representing the nodes which have been searched and the light green squares representing the nodes which are next to be searched.[9].

Nodes in the environment normally represent areas where the agent can make a choice to change direction, for example a T-junction in a road. These nodes store all of the paths to neighbouring nodes that are used for pathing calculations. These routes are commonly referred to as edges.

There are many different approaches to agent pathfinding such as Dijkstra’s algorithm, A*, JPS+ as well as other well-developed algorithms [10, 11, 12]. An example of A* can be seen in Figure 1. These approaches, while working well for solo agents travelling through an environment, can cause problems when multiple agents are involved.

The issues arising from multi-agent pathfinding are not due to the number of agents, as path planning is known to be compatible with GPU computation as the calculations are easily parallelisable [13]. The issue is in fact with the way agents communicate their paths with each other.

Where A* is the most common form of pathfinding found in AI development, due to its optimal path generation and it’s relative ease to setup, the fact that it does not factor in other agents in any way is an issue that needs to be addressed. Other work has proposed variations to A* to factor in other agents such as splitting up the area to path find through [14, 15] or even using machine learning to arrive at a viable solution [16].

1.3 Agent Based Diffusion

Agent based diffusion has been the basis of multiple avenues of research, such as image feature extraction [17] and distributed optimization [18]. The use of it as a pathfinding mechanism is a relatively recent development [19].

Agent based diffusion is based upon the behaviour of particle diffusion, where atoms move from an area of high concentration to an area of lower concentration. Diffusion of the goal value is simulated using a simplified form of the diffusion equation:

$$u_{0,t+1} = u_{0,t} + D\sum_{i=1}^{n}\left(u_{i,t} - u_{0,t}\right) \tag{1}$$

where:

n = number of neighbouring agents used as input for the diffusion equation

$u_{0,t}$ = diffusion value of centre agent

$u_{i,t}$ = diffusion value of neighbour agent (i > 0)

D = diffusion coefficient [0..0.5]

The diffusion coefficient controls the speed of the value propagation. The larger the value the faster the diffusion. Increasing the number of neighbours increases the time needed for a nodes calculation by a factor of O(n).

This diffusion pattern is the main component of agent pathfinding in agent based diffusion. The diffusion value that permeates from the goal node, in this case Pacman, spreads to other nodes, as seen in Figure 2, distributing the diffusion value throughout the entire grid.

Creating a simple hill-climbing [20] based behaviour, where ghost agents constantly move to the next highest diffusion value, an emergent collaborative behaviour occurs as seen in Figure 3. By making the ghost agents block the diffusion value propagating from the goal node, the agents will change course to avoid one another’s path due to the diffusion value being higher in the other nodes of the grid.

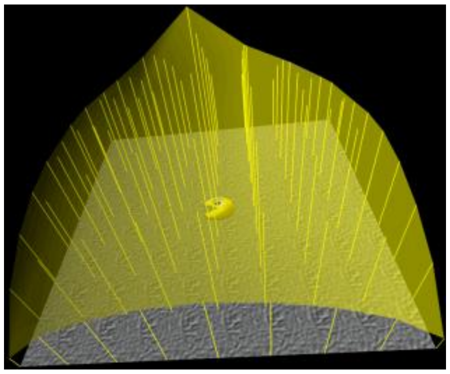


Fig 2. A 9 x 9 floor tile grid with a Pacman agent in the middle. The diffusion value is diffused from pacman via the tiles in the grid with the diffusion values shown as a plot above the grid [19].

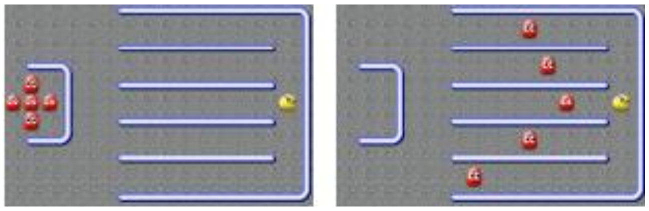


Fig 3. Five ghosts move from their start point to the end goal (Pacman) without colliding into each other by taking their own unobstructed paths [19].

The major advantage of collaborative diffusion over A* is that if anything should cause a path to change with A* the whole path must be recalculated whereas with collaborative diffusion no such recalculation needs to take place as the diffusion values change accordingly [21].

2 RELATED WORK

2.1 Smart Parking

Once a driver is allocated a parking space by a PGI system, a Global Positioning System (GPS) is commonly employed to direct the driver to their destination. In early iterations of these PGI systems, they caused undue congestion due to the GPS pathfinding used not taking into account other cars en-route to the same location.

Without the data on other driver's paths, many drivers were given overlapping routes through the city to the same location thus causing a traffic buildup. With traffic volumes having a clear pattern of growth year on year as seen in Figure 4, the need for smart parking systems to reduce road traffic is only going to increase with time.

Only limited research has been performed to explore how drivers are given a path to their assigned parking space within these PGI technologies. As most of these systems do not take into account other drivers paths and their positions, a lot of unnecessary traffic congestion can be produced from following similar paths to the designated locations.

Many of the issues arising from the PGI systems version of pathfinding can be mitigated. This is done by having a system in place where each driver's pathfinding solution collaborates to avert each other's paths.

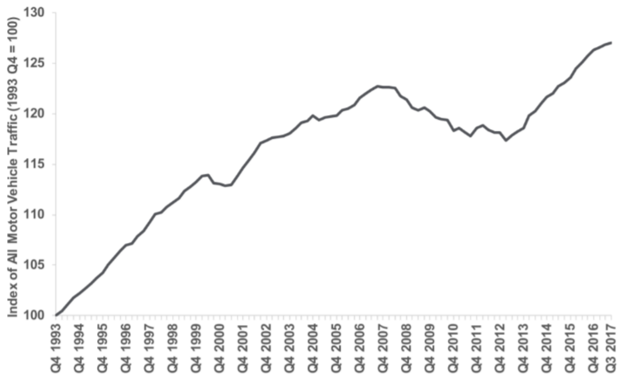


Fig. 4. Rolling Annual Index of Road Traffic in Great Britain, from 1993 [22].

One of the few research teams to look into a solution to the non-collaborative pathing problem was Rhodes et al [23]. Rhodes et al used a modified form of the A* pathfinding algorithm to allow concurrent pathing of multiple agents while also taking into consideration other agents current paths and positions by expanding upon the work of David Silver [24].

The purpose of creating suboptimal paths in this modified version of A* is to create paths that avoid other drivers. These less optimal paths prevent traffic build up caused by agents all seeking the same goal on the overlapping paths.

By creating a pathfinding algorithm that takes other agents positions and paths into account, a viable solution could be created for the issues highlighted in the section 1.1. If a GPS utilised a variation of a collaborative pathfinding algorithm, much of the traffic generated by drivers following similar paths would in theory be reduced, if not eliminated, with little impact on journey time.

2.2 Collaborative Diffusion

The approach this project will take is utilizing an algorithm known as collaborative diffusion [19]. Collaborative diffusion works by having goal nodes possessing a static diffusion value, which propagates through neighbouring nodes via its edges. Agents follow the increasing diffusion value to each node until they reach their goal.

Other agents reduce the diffusion value of the node they are currently on and their next destination node. This produces an emergent behaviour where each agent avoids each other while traveling to their destination, thus mitigating traffic issues.

$$u_{0,t+1} = \lambda \left[u_{0,t} + D \sum_{i=1}^{n} \left(u_{i,t} - u_{0,t} \right) \right] \tag{2}$$

where:

n = number of neighbouring agents used as input for the diffusion equation

$u_{0,t}$ = diffusion value of centre agent

$u_{i,t}$ = diffusion value of neighbour agent (i > 0)

D = diffusion coefficient [0..0.5]

λ = agent interaction variable

As seen above in equation 2, which is the collaborative diffusion formula, it is similar to the diffusion equation in section 1.3. The collaborative diffusion equation differs as it has an extra λ variable. This extra λ value is used to change agents' behaviour towards the diffusion value calculated.

When the formula uses λ <1 this displays competition between agents and conversely when λ>1 is used this shows collaboration. The full list of λ value behavioural effects can be found in Table 1. In terms of the Pacman examples used earlier in section 1.3, if a ghost agent had the λ>1 trait, ghosts would move towards that friendly ghost provided it does not block the diffusion values like in Figure 3.

Table 1. **Agent Interaction**

λ	Agent Interaction
>>1	Extreme Collaboration
>1	Collaboration
=1	Autonomy
<1	Competition
<<1	Extreme Competition

The simple nature of the formula is one of the algorithm's many strengths, making it exceptionally fast to compute. This is one of the reasons this algorithm was considered to be a promising candidate due to its exceptional calculation speed, compatibility with GPU calculations [25] and ease to implement in an already existing code base with a A* node grid in place.

3 DESIGN AND IMPLEMENTATION

3.1 Proof of Concept

In the early stages of development, the primary goal was to provide a visual representation of the data for each pathfinding method tested. A number of different features were decided as essential for the application whilst comparative testing was being performed:

- A flexible and easily extensible node system with the plan of further development later in the project.
- An easy to follow visualisation of the path generated.
- Show diffusion based pathfinding avatars avoiding each other.
- Clearly displayed statistics generated during the pathfinding which had to include the path length and time taken to create the path.
- Easily modifiable variables.
- Easy to use controls displayed on screen.

In order to produce a testing environment conducive to running pathfinding, a solid and flexible node based system was developed. There are three classes that make the node system function as it does: Node, Edge and GridManager.

The Node class at this stage was simple to make and the class was easy to return to and extend as needed. The Node class spreads the diffusion value (which is required for collaborative diffusion) by using its neighbours' diffusion with a function named CalculateDiffusion() which is called every update on each Node object, as seen in Table 2.

Lines 7-20 are the key part of calculating the diffusion value of the node. The foreach loop cycles through each of the node's neighbours and incorporates their diffusion value into their own. Essentially this is the result of converting equation 2 into code. Lines 18 and 19 apply a decay value to reduce the diffusion value slightly. This is a required feature, as without this the entire grid will slowly become the diffusion goal value, thus rendering the pathfinding based on diffusion inoperable.

Table 2. **CalculateDiffusion**

```
1   private void CalculateDiffusion()
2   {
3       if(myManager.GetEndAllocated() && NodeStatus !=
4          NODE_STATUS.END && NodeStatus !=
5          NODE_STATUS.BLOCKED                        )
6       {
7           float originalDiff = diffusion;
8           foreach (EdgeScript edge in edges)
9           {
10              if (edge.GetTo().NodeStatus !=
11                  NODE_STATUS.BLOCKED       )
12              {
13                  diffusion += (myManager.diffusionCoeff *
14                               (edge.GetTo().diffusion -
15                               originalDiff));
16              }
17          }
18          diffusion = (diffusion - (diffusion *
19                      myManager.decayRate));
20      }
21      else if (!myManager.GetEndAllocated())
22      {
23          diffusion = (diffusion - (diffusion *
24                      myManager.decayRate));
25      }
26      if (avatarsOverlaping >= 1)
27      {
28          if (NodeStatus != NODE_STATUS.END)
29          {
30              diffusion /= myManager.evasionStrength +
31                     (1 / (float)myManager.avatars.Count);
32          }
33      }
34      avatarsOverlaping = 0;
32  }
```

Lines 26-34 reduce the diffusion value further based whether an avatar is present on the node. This is a departure from the method used to navigate ghosts discussed in section 1.3 as the value is not completely blocked by the presence of an AI agent. This choice was made as when high numbers of AI agents were placed in a similar environment to Figure 3 agents behind those blocking the value of the goal node would wander aimlessly which is not the desired behaviour. The difference this change made can be seen in Figure 5.

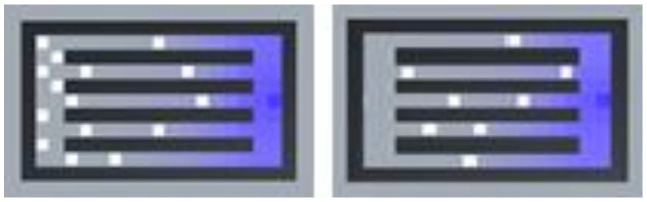


Fig. 5. The left figure shows AI agents blocking the value leading to wandering near the start point by those in the area of no diffusion. The right shows the AI not completely blocking the value and all heading towards the goal node. The darker blue the node is, the greater the diffusion value.

The Node class also possessed a NodeStatus enum which was used to store what state the node was in. The six states are: unsearched, start, end, searched, path and blocked. While the application is running, the node changes its appearance in accordance to which state the node is in. This makes a clear way of visualising the data the path generation creates. The appearance of the node colours can be seen in Figure 6.

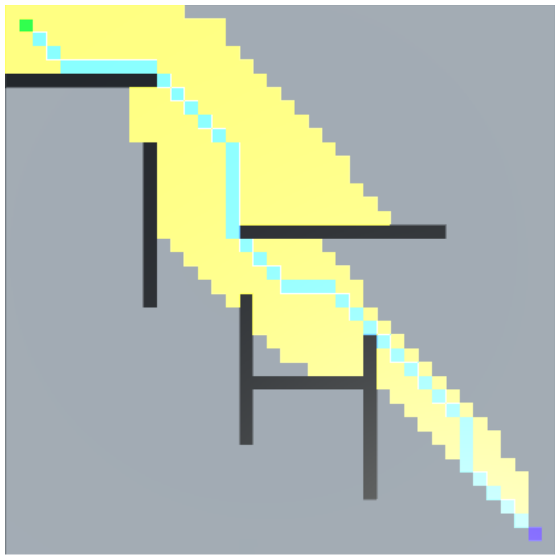


Fig. 6. An A* path from the green start node to the dark blue goal node. Light blue nodes are the path; yellow signifies nodes that have been searched as part of pathfinding, black nodes, which are impassable and grey nodes that have not taken place as part of the search.

The Edge class is a simple class as seen in Table 3. The Edge class is important to the way each node is linked to each other. This class features heavily throughout the pathfinding code as it is essential to know which node is neighbouring which. The decision was made to make this a class rather than a struct as structs are passed by value while classes are passed by reference in C# and moving it through functions by value was deemed unnecessary.

Table 3. **Edge class**

```
public class EdgeScript
{
    private NodeScript from;
    private NodeScript to;
    private float cost;

    public NodeScript GetFrom()
    {
        return from;
    }

    public void SetFrom(NodeScript newFrom)
    {
        if (newFrom != null)
        {
            from = newFrom;
        }
    }

    public NodeScript GetTo()
    {
        return to;
    }

    public void SetTo(NodeScript newTo)
    {
        if (newTo != null)
        {
            to = newTo
        }
    }

    public float GetCost()
    {
        return cost;
    }

    public void SetCost(float newCost)
    {
        cost = newCost;
    }
}
```

The GridManager class creates and maintains the grid in the proof of concept scene. The grid can be created to be any size or dimensions for the sake of testing. The GridManager also runs the different types of pathfinding in the project: Pythagoras based A*, Manhattan based A*, Dijkstra's algorithm and collaborative diffusion.

A departure from typical A* pathing was the use of lambda functions to calculate the h cost of the A* algorithm. CalculateHCost, as seen in Table 4, has a function delegate as a parameter making the function run with any new heuristics. This was in keeping with the plan to make the node system easily extensible by allowing any lambda heuristic to operate within the function prevented the need for an individual CalculateHCost function for each heuristic created.

Table 4. **CalculateHCost**

```
float CalculateHCost(NodeScript node,
          Func<NodeScript, NodeScript, float> heuristic,
          float weight)
{
    if (node && heuristics.Count > 0
        && weights.Length >= heuristics.Count)
    {
        float HCost = 0;
        for (short i = 0; i < heuristics.Count; ++i)
        {
            HCost += heuristics[i](node,
                                  paths[pathNumber].end)
                                  * weights[i];

            return HCost;
        }
    }
    return -1;
}
```

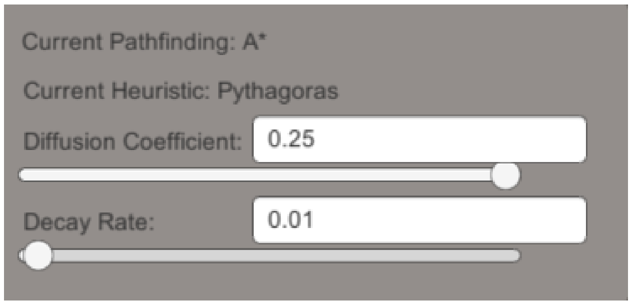
Each path calculated is stored in one of nine path structs in GridManager, this struct can be seen in Table 5. The reason for the encapsulation of all of the path data into a struct was to make comparisons of different paths clean and easy to follow within the code and so all of the data was within a single array. By mapping button presses to indexes of the array, rapid access of previous test data was possible making the whole comparative process much simpler.

Table 5. **Path struct**

```
public struct Path
{
public List<NodeScript> pathNodes;
public List<NodeScript> searchedNodes;
public NodeScript start;
public NodeScript end;
public bool startAllocated;
public bool endAllocated;
public bool startEndReallocated;
public float pathTime;
public float pathDist;
public AvatarScript avatar;
};
```

During the pathfinding process, the GridManager changes the NodeStatus of each node as required while pathfinding occurs. This creates the pattern seen in Figure 6.

To create easily modifiable variables, Unity's UI objects were utilised. By using Unity's Slider and Input Field UI objects, this creates a simple and easy to use system for changing the collaborative diffusion. Both the diffusion coefficient and the decay rate can be changed with the sliders for quick changes, or the input fields for more precise values. Figure 7 shows these features.



Current Pathfinding: A*

Current Heuristic: Pythagoras

Diffusion Coefficient: 0.25

Decay Rate: 0.01

Fig. 7. The GUI present in the proof of concept for changing the parameters at runtime.

To display the input onscreen yet again Unity's UI was used. A simple UI Text box was utilised and placed in a clear position on the screen so any user can use the application without difficulty. When the mode of the application changes, the text changes as required to display the changed controls.

3.2 Newcastle Simulation

After the proof of concept showed promise using collaborative diffusion for agent pathfinding, the next step was to create a simulation based on the city centre of Newcastle. The plan was to modify the node based system created in the proof of concept to overlay nodes on the roads of Newcastle. At this stage, a number of different goals were identified as essential features of the application:

- A map of Newcastle with accurate road representation with frame rate above 30fps.
- Avatars both entering and leaving the city to more realistically represent traffic
- Creation of a selection of real parking lots with the actual pricing for the simulation to use.
- Clearly displayed statistics generated during the simulation, which had to include the number of cars driving in the simulation and the total revenue generated by the parking lots.
- Easily modifiable variables.
- A way to restart the simulation at any time.

The map of Newcastle used for the simulation was taken from Google maps [26] to provide an accurate and clear texture to be used in the Newcastle simulation as seen in Figure 8. Google maps due to its ease of use and clarity of the images produced, was chosen to get the required images of Newcastle. Several images were taken and merged together to maintain the detailed street names throughout the map. Satellite imagery of the same map is also present in the project, seen in Figures 9 and 10; however, the more complex imagery of the satellite view made identifying which roads the avatars were on, difficult. This led to the satellite images being cut from the final application.

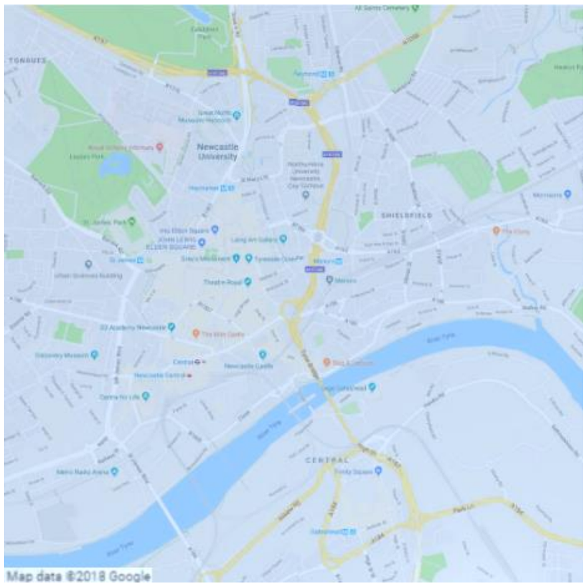


Fig. 8. The view of Newcastle used in the completed simulation.

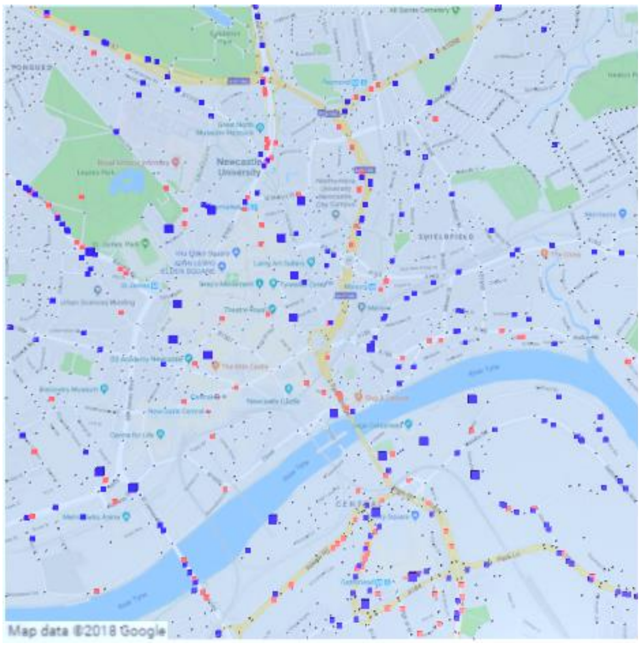


Fig. 9. The view of Newcastle used in the completed simulation.

Fig. 10. The cut satellite view of Newcastle as seen in the earlier versions of the simulation.

In order to have the avatars move into and out of the city the node system was re-used. However unlike in the proof of concept the node system was not a perfect grid and because of this some changes had to be made.

Due to the complexity of the city of Newcastle, no automatic generation of the node placement could be developed in a reasonable amount of time, so all 1280 nodes were placed by hand into the simulation on each junction and in turn found on the map. These nodes can be seen as the small black dots in Figure 9.

Once the nodes were placed on the map, each node was assigned their neighbours by hand. This was once again due to the lack of an easy way of generating the edges. In the proof of concept, as all of a node's neighbours were immediately next to each other, edge generation was simple and a node's neighbours would also generate an edge to the original node making it possible to move from one node to its neighbour or vice versa. In the Newcastle simulation, the two way edge would be a poor assumption to make due to the many one way roads present in Newcastle. To make this work, the node class was changed slightly. Now instead of relying only on the auto generation of edges, each node can accept other nodes as a public variable so that manual neighbour creation was possible.

Once all of the nodes had been placed with each of their neighbours declared, the next step was to come up with a way of making two types of goals. Due to the proof of concept only having one goal value, the agents could not enter and leave the city, only one or the other. To circumvent this issue, a new value was introduced to the nodes. This new value called exit diffusion was used for agents leaving the city. The calculation for exit diffusion and the way it propagated through the node system was identical to the previous diffusion value used. The issue that arose from this was that the increase in computations per update by the nodes drove the simulation into an unusable frame rate of >10 frames per second. To solve this issue, a toggle was created so that on one frame the standard diffusion is calculated and in the other the exit diffusion is calculated, this can be seen in Table 6.

In order to make the cars entering and exiting the city visually distinct, cars entering the city were made blue and those exiting made red as seen in Figure 9.

Table 6. **NodeUpdate**

```
public void NodeUpdate()
{
    if (myManager.exitAvailable)
    {
        if (calculatedDiffOnLastFrame)
        {
            CalculateExitDiffusion();
        }
        else
        {
            CalculateDiffusion();
        }
        calculatedDiffOnLastFrame =
                              !calculatedDiffOnLastFrame;
    }
    ... //Other code
}
```

Once the entering and exiting of the avatars had been accomplished, the next step was to create a way of monitoring the revenue generated by the parking lots. With the brief stating that real world parking lots had to be used along with their parking prices, research into parking lots around Newcastle was conducted. The Parkopedia website, which showed all parking lots and their pricing, was used to gather this data quickly [27].

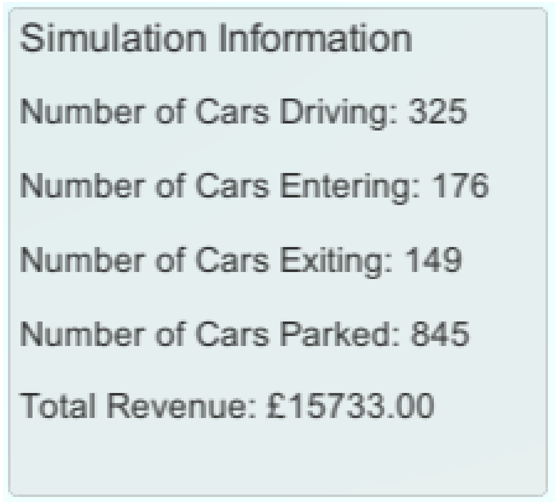
Once all of the relevant data had been acquired, the implementation of the parking lots was a relatively simple one. A new class called ParkingLot was created to store and manage all data relevant to the parking lot chosen for representation in the simulation. The most important part of the class was the AddCar and CalculateRevenue functions found in Table 7.

As seen in lines 8-11, the stay time is randomly generated between the maximum and minimum stay times, multiplied by the time step. The time step is the time in seconds that an hour is represented by in the simulation. The stayTime value is then sent over to the calculate revenue function and then used as a timer to release the car at the correct time. The way CalculateRevenue works is it cycles through an array of floats called stayBandDuration which is used to store the different times in hours a car may stay. Once the stayTime is less than the stayBandDuration the index is used to get the corresponding price stored in the pricingBand array and added to the totalRevenue.

Table 7. **AddCar and CalculateRevenue**

```
1   public void AddCar()
2   {
3       if (currentOccupancy < capacity)
4       {
5           ++currentOccupancy;
6           percentageFilled = currentOccupancy /
7                              capacity;
8           float stayTime = Random.Range(stayMin *
9                           timeStep, stayMax * timeStep);
10          CalculateRevenue(stayTime);
11          Invoke("ReleaseCar", stayTime);
12      }
13  }
14
15  void CalculateRevenue(float stayTime)
16  {
17      int index = 0;
18      for(int i = 0; i < stayBandDuration.Length; ++i)
19      {
20          index = i;
21          if(stayTime < stayBandDuration[i])
22          {
23              break;
24          }
25      }
26      totalRevenue += pricingBand[index];
27  }
```

To create the clearly displayed statistics on the screen, Unity's UI system was utilised by displaying the values stored inside GridManager and ParkingLot manager. These were clearly displayed in the top right of the screen as seen in Figure 11. Specific parking lot information can be seen by hovering over a parking lot with the mouse. The values shown can be seen in Figure 12.



Simulation Information

Number of Cars Driving: 325

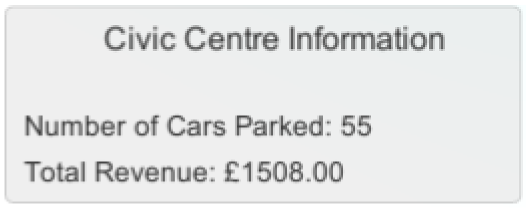
Number of Cars Entering: 176

Number of Cars Exiting: 149

Number of Cars Parked: 845

Total Revenue: £15733.00

Fig. 11. The information displayed at the top right of the simulation.



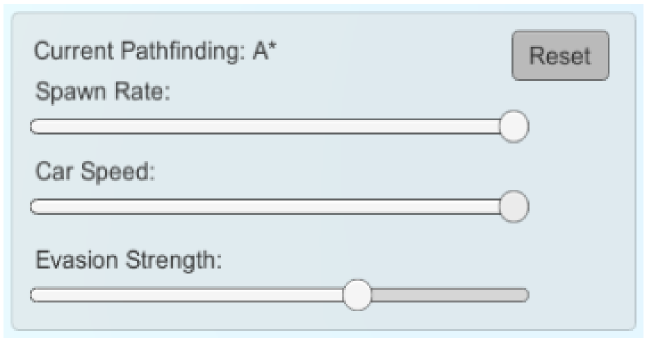
Civic Centre Information

Number of Cars Parked: 55

Total Revenue: £1508.00

Fig. 12. The information displayed at the bottom right of the simulation when a parking lot is highlighted with the mouse.

To create easily modifiable variables Unity's UI objects were utilised again. By using the Unity's Slider UI object to create a simple and easy to use system for changing the speed of the cars, spawn rate of cars and the avoidance factor of the cars. Figure 13 shows these features.



Current Pathfinding: A*

Spawn Rate:

Car Speed:

Evasion Strength:

Fig. 13. The GUI present in the Newcastle simulation for changing the parameters at runtime.

4 RESULTS AND EVALUATION

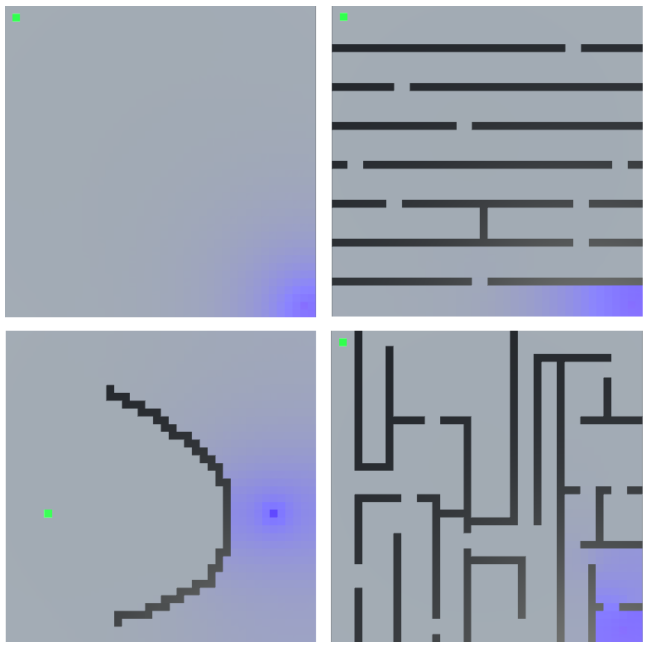
4.1 Project Progress

The project as a whole has been a success, as the objective of creating a collaborative diffusion based simulation over a map of Newcastle was achieved. The agents used in the simulation can be seen taking different routes around the city to reduce congestion, as originally proposed and as seen in Figure 9.

4.2 Pathfinding Evaluation

The tests performed during the proof of concept were based on four different maps, with four different types of pathfinding: Pythagoras A*, Greedy Best First Search A*, Dijkstra and Diffusion. The four maps that were used for the test scenarios along with the start and end points of each test can be found in Figure 14.

Fig. 14. The four different maps used for testing.



As seen in Appendix A.1, the results clearly show that diffusion pathfinding provided the fastest path every time. However Pythagoras A* is consistently the best for finding the optimal route as seen by its route and node length statistics. This result clearly shows the huge speed increase diffusion has over all other forms of pathfinding and despite not having the most optimal path this is inconsequential, as the collaborative pathfinding problem demands a suboptimal resultant route due to the nature of the problem [23].

4.3 FPS Evaluation

In the stress tests of the system whilst running the Newcastle simulation, two cases were tested on two different machines to ensure the goal frame rate of 30fps was always met. The test consisted of sampling the frame rate at 10 different time steps one minute apart after the program had ran the simulation with the selected parameters for 10 minutes.

The two machines consisted of PC one with: Intel Core i7-4790S 8 core CPU running at 3.20GHz, 8.00GB RAM and a NVIDIA GeForce GTX 970 graphics card and PC two with: Intel i7-4790 8 core CPU running at 3.60 GHz, 32.00GB RAM and a AMD Radeon R9 270.

The first stress test was running the program with the maximum spawn rate with the lowest possible speed, to maximise the number of cars in the simulation. The second test was running at the maximum spawn rate and maximum car speed to see if increasing the car movement rate affected the results in any way.

As seen in Appendix A.2 the program successfully runs above the 30fps goal set during the planning of this part of the application. With the worst framerate recorded of 36.3, in Test 1 Time Step 6, this clearly shows the application runs within the desired parameters of performance.

5 CONCLUSIONS AND FUTURE WORK

Overall as seen by the previous section, the application produced has been a success. However when comparing this application to many other state of the art technologies in the same field, many potential improvements to the application can be seen.

One improvement would be to improve the entrance of cars into the city by basing the volume of traffic on past data as opposed to a random number slider. This would improve the validity of some of the data created by the application

Another improvement would be to visualise the data created in the simulation on a variable time step. At the moment, only the hard numbers are displayed on screen to show the revenue generated, number of cars parked and driving etc. It would be a great improvement if this data could be in an easily digestible format, such as a graph displaying the revenue generated over time. This would have applications for businesses using the software to see how changing pricing at a certain times of the day could affect their revenue. This in combination with the previously mentioned improvement would be a major selling point of this technology.

Another feature, which could be of use, would be to generate a heat map of the traffic over time. This has the potential to see where further traffic mitigation strategies or improved road works could be employed within a city. In addition, this feature would be relatively simple to implement, as the nodes monitor how many avatars overlap them on a frame by frame basis, so it would simply be a case of storing this data and applying a visual overlay to the map to see problem areas.

An efficiency-based improvement of the project would be to move many of the processes currently residing on the CPU onto the GPU by applying GPGPU principals. As highlighted in earlier sections many of the processes in collaborative diffusion are highly parallelisable such as the node diffusion calculations and the avatar pathing [25]. Due to the primary bottleneck of this application being the high volume of parallelisable functions, the performance benefit has the potential to be huge.

6 REFERENCES

[1] Parkhurst, G., 1995. Park and Ride: could it lead to an increase in car traffic? Transport Policy, 2(1), 15-23.

[2] R. Arnott, T. Rave, and R. Schob, “Alleviating urban traffic congestion,” ¨ MIT Press Books, vol. 1, 2005.

[3] E. Gantelet, A. Lefauconnier, and SARECO. The time looking for a parking space: strategies, associated nuisances and stakes of parking management in France. Association for European Transport and Contribution, 2006.

[4] D. Teodorović and P. Lučić, “Intelligent parking systems,” European Journal of Operational Research, vol. 175, no. 3, pp. 1666–1681, 2006.

[5] T. Lin, “Smart Parking: Network, Infrastructure and Urban Service”.
Diss. INSA Lyon, 2015.

[6] G. Yan, W. Yang, D. B. Rawat, and S. Olariu, “Smartparking: a secure and intelligent parking system,” Intelligent Transportation Systems Magazine, IEEE, vol. 3, no. 1, pp. 18–30, 2011.

[7] H. Wang and W. He, “A reservation-based smart parking system,” in Computer Communications Workshops (INFOCOM WKSHPS), 2011 IEEE Conference on. IEEE, 2011, pp. 690–695.

[8] Y. Geng and C. G. Cassandras, “A new smart parking? System based on optimal resource allocation and reservations,” in Intelligent Transportation Systems (ITSC), 2011 14th International IEEE Conference on. IEEE, 2011, pp. 979–984.

[9] Buckland, M. (2005). *Programming Game AI by Example* (1st ed.). (MatBuckland, Ed.) Sudbury: Wordware Publishing, Inc.

[10] E.W. Dijkstra. A note on two problems in connection with graphs. Numerische Mathematik, 1:269–271, 1959.

[11] Hart, P. E.; Nilsson, N. J.; Raphael, B. (1968). “A Formal Basis for the Heuristic Determination of Minimum Cost Paths”. IEEE Transactions on Systems Science and

Cybernetics SSC4. 4 (2): 100–107.
[12] Rabin, S., & Silva, F. (2015). JPS+: An Extreme A* Speed Optimization for Static Uniform Cost Grids. In S. Rabin, & S. Rabin (Ed.), Game AI Pro 2: Collected Wisdom of Game AI Professionals (pp. 131-143). Boca Raton: CRC Press.
[13] W. Blewitt, G. Ushaw, and G. Morgan, "Applicability of gpgpu computing to real-time ai solutions in games," 2013.
[14] M. E. Falou, M. Bouzid, and A.-I. Mouaddib, "Dec-a*: A decentralized multiagent pathfinding algorithm," in IEEE 24th International Conference on Tools with Artificial Intelligence, 2012.
[15] K.-H. C. Wang, "Tractable massively multi-agent pathfinding with solutions quality and completeness guarantees," in Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence, 2011.
[16] Stone, P., and Veloso, M. 1997. Multiagent Systems: A Survey from the MachineLearning Perspective, Technical report, CMU-CS-97-193, School of Computer Science, Carnegie Mellon University
[17] Liu, J., Tang, Y.Y. and Cao, Y.C. An Evolutionary Autonomous Agents Approach to Image Feature Extraction IEEE Transactions on Evolutionary Computation, 1 (1997). 141-158
[18] Tsui, K.C. and Liu, J., Multiagent Diffusion and Distributed Optimization. in Proceedings of the second international joint conference on Autonomous agents and multiagent systems, (Melbourne, Australia, 2003), ACM Press, New York, NY, USA, 169 - 176.
[19] Repenning A. Collaborative diffusion: Programming antiobjects. Technical Report, University of Colorado 2006.
[20] Genesereth, M.R. and Nilson, N.J. Logical Foundations of Artificial Intelligence. Morgan Kaufman Publishers, Inc., Los Altos, 1987.
[21] Gelperin, D. On the Optimality of A*. AI, 8 (1977). 69- 76.
[22] Department for Transport. Provisional road traffic estimates, Great Britain: October 2016 to September 2017 report. Retrieved from https://www.gov.uk/government/statistics/provisional-road-traffic-estimates-great-britain-october-2016-to-september-2017
[23] Rhodes, C., Blewitt, W., Sharp, C., Ushaw, G., and G. Morgan, 2014. Smart routing: A novel application of collaborative pathfinding to smart parking systems. In: Proc. of "2014 IEEE 16th Conf. on Business Informatics (CBI)," Geneva, Switzerland, 14- 17 July 2014, pp. 119-126, DOI:10.1109/CBI.2014.22.
[24] D. Silver, "Cooperative pathfinding," in AI Game Programming Wisdom 3, S. Rabin, Ed. Charles River Media, Inc., Massachusetts, 2006, ch. 2.1, pp. 99–112.
[25] C. McMillan, E. Hart, K. Chalmers, "Collaborative Diffusion on the GPU for Path-Finding in Games" in European Conf on the Applications of Evolutionary Computation, Springer, pp. 418-429, 2015.

[26] Google, 'Google Maps', 2018. Retrieved from https://www.google.co.uk/maps
[27] Parkopedia, 'Parkopedia', 2018. Retrieved from https://en.parkopedia.co.uk/parking/carpark/gateshead_college_mscp/ne8/gateshead

APPENDIX

A.1 PATHFINDING TEST DATA

A.1.1 Map 1 Tests

Test 1	Pythagoras A*	Greedy Best First Search A*	Dijkstra	Collaborative Diffusion
Nodes in Path	38	38	38	38
Path Distance (m)	523.259	523.259	523.259	523.259
Nodes Searched	37	37	1580	37
Time Taken (ms)	24.271	5.947	2969.490	5.737

Test 2	Pythagoras A*	Greedy Best First Search A*	Dijkstra	Collaborative Diffusion
Nodes in Path	38	38	38	38
Path Distance (m)	523.259	523.259	523.259	523.259
Nodes Searched	37	37	1580	37
Time Taken (ms)	25.529	6.326	2968.912	5.633

Test 3	Pythagoras A*	Greedy Best First Search A*	Dijkstra	Collaborative Diffusion
Nodes in Path	38	38	38	38
Path Distance (m)	523.259	523.259	523.259	523.259
Nodes Searched	37	37	1580	37
Time Taken (ms)	25.072	6.174	2969.412	5.692

Test 4	Pythagoras A*	Greedy Best First Search A*	Dijkstra	Collaborative Diffusion
Nodes in Path	38	38	38	38
Path Distance (m)	523.259	523.259	523.259	523.259
Nodes Searched	37	37	1580	37
Time Taken (ms)	24.831	5.983	2970.472	5.744

A.1.2 Map 2 Tests

Test 1	Pythagoras A*	Greedy Best First Search A*	Dijkstra	Collaborative Diffusion
Nodes in Path	119	131	119	119
Path Distance (m)	1312.549	1449.117	1312.549	1329.117
Nodes Searched	1130	485	1381	118
Time Taken (ms)	2288.041	158.203	2300.047	36.254

Test 2	Pythagoras A*	Greedy Best First Search A*	Dijkstra	Collaborative Diffusion
Nodes in Path	119	131	119	119
Path Distance (m)	1312.549	1449.117	1312.549	1329.117
Nodes Searched	1130	485	1381	118
Time Taken (ms)	2289.078	160.650	2304.883	37.525

Test 3	Pythagoras A*	Greedy Best First Search A*	Dijkstra	Collaborative Diffusion
Nodes in Path	119	131	119	119
Path Distance (m)	1312.549	1449.117	1312.549	1312.549
Nodes Searched	1130	485	1381	118
Time Taken (ms)	2289.658	158.613	2997.207	35.994

Test 4	Pythagoras A*	Greedy Best First Search A*	Dijkstra	Collaborative Diffusion
Nodes in Path	119	131	119	119
Path Distance (m)	1312.549	1449.117	1312.549	1329.117
Nodes Searched	1130	485	1381	118
Time Taken (ms)	2288.041	158.203	2300.047	37.729

A.1.3 Map 3 Tests

Test 1	Pythagoras A*	Greedy Best First Search A*	Dijkstra	Collaborative Diffusion
Nodes in Path	38	45	38	38
Path Distance (m)	461.127	502.132	461.127	461.127
Nodes Searched	728	403	1377	37
Time Taken (ms)	543.121	82.519	2463.538	23.681

Test 2	Pythagoras A*	Greedy Best First Search A*	Dijkstra	Collaborative Diffusion
Nodes in Path	38	45	38	38
Path Distance (m)	461.127	502.132	461.127	461.127
Nodes Searched	728	403	1377	37
Time Taken (ms)	541.473	80.058	2465.320	23.724

Test 3	Pythagoras A*	Greedy Best First Search A*	Dijkstra	Collaborative Diffusion
Nodes in Path	38	45	38	38
Path Distance (m)	461.127	502.132	461.127	461.127
Nodes Searched	728	403	1377	37
Time Taken (ms)	542.296	83.144	2465.908	23.662

Test 4	Pythagoras A*	Greedy Best First Search A*	Dijkstra	Collaborative Diffusion
Nodes in Path	38	45	38	38
Path Distance (m)	461.127	502.132	461.127	461.127
Nodes Searched	728	403	1377	37
Time Taken (ms)	543.479	82.092	2462.740	24.773

A.1.4 Map 4 Tests

Test 1	Pythagoras A*	Greedy Best First Search A*	Dijkstra	Collaborative Diffusion
Nodes in Path	124	133	124	124
Path Distance (m)	1362.548	1489.827	1362.548	1362.548
Nodes Searched	1206	813	1294	123
Time Taken (ms)	2278.002	364.216	2287.626	32.449

Test 2	Pythagoras A*	Greedy Best First Search A*	Dijkstra	Collaborative Diffusion
Nodes in Path	124	133	124	124
Path Distance (m)	1362.548	1489.827	1362.548	1461.959
Nodes Searched	1206	813	1294	123
Time Taken (ms)	2274.139	369.812	2291.626	32.109

Test 3	Pythagoras A*	Greedy Best First Search A*	Dijkstra	Collaborative Diffusion
Nodes in Path	124	133	124	124
Path Distance (m)	1362.548	1489.827	1362.548	1461.959
Nodes Searched	1206	813	1294	123
Time Taken (ms)	2274.839	370.812	2293.281	31.358

Test 4	Pythagoras A*	Greedy Best First Search A*	Dijkstra	Collaborative Diffusion
Nodes in Path	124	133	124	124
Path Distance (m)	1362.548	1489.827	1362.548	1461.959
Nodes Searched	1206	813	1294	123
Time Taken (ms)	2274.139	369.812	2291.626	34.094

A.2 FRAMES PER SECOND TEST DATA

PC 1: Intel Core i7-4790S 8 core CPU running at 3.20GHz, 8.00GB RAM and a NVIDIA GeForce GTX 970 graphics card

PC 2: Intel i7-4790 8 core CPU running at 3.60 GHz, 32.00GB RAM and a AMD Radeon R9 270.

All data is measured in frames per second

Test 1: High Spawn Rate & Low Car Speed	PC 1	PC2
Time Step 1	37.6	43.3
Time Step 2	39.5	39.6
Time Step 3	40.2	44.9
Time Step 4	38.9	45.3
Time Step 5	39.8	43.2
Time Step 6	36.3	42.6
Time Step 7	41.6	44.0
Time Step 8	38.4	39.1
Time Step 9	37.2	46.3
Time Step 10	40.2	42.5
Average	38.97	43.08

Test 2: High Spawn Rate & High Car Speed	PC 1	PC2
Time Step 1	49.2	55.4
Time Step 2	50.1	53.1
Time Step 3	47.6	54.3
Time Step 4	50.2	56.2
Time Step 5	45.3	49.8
Time Step 6	47.3	60.2
Time Step 7	49.8	53.4
Time Step 8	47.3	58.9
Time Step 9	51.4	54.0
Time Step 10	46.8	53.6
Average	48.50	54.89