Dynamic Pathfinding Algorithm for Bike Routes in ASU Campus

Ajinkya Patil
ajinkya.patil@asu.edu

Saurabh Kardile
skardile@asu.edu

Ajay Kulkarni
aakulka9@asu.edu

School of Computing, Informatics, and Decision Systems Engineering
Arizona State University
Tempe, AZ 85281

ABSTRACT

Various search algorithms such as Dijkstra's algorithm and elementary A* search algorithm were created to solve the shortest path problem. But all these variants provide solutions for static graphs. A graph can be dynamic when some of its entities like edges, weights or vertices change over time. Finding the optimal path algorithm for this scenario is not only interesting but also challenging. In this paper, we present a modification of A* search algorithm whose objective is to provide pathfinding for bike routes in Arizona State University campus. This adaptive optimal path algorithm works under the constraint of variable travel time based on pedestrian traffic data that will depend on the classes scheduled in that particular area of campus. The key technical contributions of the proposed algorithm is the computation of crowd density during pathfinding, heuristic guidance and the temporal aspect of path costs.

1. INTRODUCTION

The pathfinding problem involves finding the optimum path between source and destination with minimum distance or minimum travel time. With tools like MapQuest and Google Maps becoming an essential part of everyday life, shortest path problems are certainly gaining back popularity in the field of navigation systems.

There are traditional pathfinding algorithms like Dijkstra's and Best First Search. However, they are more effective when the graph is static. The reason being those algorithms are distance-based route planning algorithms which do not consider time dependent traffic data. The change in edge weights or removal of certain nodes gives a dynamic characteristic to a graph. Besides this, A* search algorithm is more efficient as it drastically reduces the search space. The objective of the proposed algorithm is to take into account crowd density at different points in the map along with the estimated time it takes to reach the destination in order to find the quickest path.

The proposed algorithm modifies the A* search algorithm by adding dynamic weights to nodes in the map. The A* search algorithm uses a heuristic that aids in finding best path by estimating the total travel time. Our algorithm refers Arizona State University (ASU) semester class schedule to determine crowd density across different regions in the campus map.

2. PROBLEM DEFINITION

In this section we first define a set of key concepts: *pathfinding grid* and *crowd density factor*.

Definition 1. A pathfinding grid consists of nodes representing points through which routes are possible. These points are called nodes, a group of which represent roads, walkways and open areas suitable for biking. The grid also comprises of regions that lack nodes. Such regions are called obstacles and represent structures, buildings etc. through which a route is not possible.

The pathfinding grid is basically an undirected weighted graph $G(V, E)$ where V is the set of nodes and E is the set of edges. For our scenario of university campus, the graph is considered undirected to allow path in any direction between any two nodes in the graph. Although paths are best thought of as a sequence of edges, it 's convenient to store them as a sequence of nodes[12]. In context, the edges have equal weight in the pathfinding grid and this weight is attributed to the nodes each edge connects. The pathfinding grid is obtained when the graph G is superimposed on the campus map where the nodes align to roads and walkways and the buildings/structures do not have any nodes. Each node is like a tile in the grid. The representation of nodes and obstacles depends on the system used to represent geometric points on the map; for clarity we have represented it with cartesian (x, y) coordinates.

Definition 2. The crowd density factor cdf, is equivalent to the number of people estimated at a location at particular time with respect to the school schedule data.

The school schedule data is simply the semester class schedule of all programs in campus. The factor determines whether a particular area (node) near a class (obstacle) will be crowded with students at a given time.

With above definitions available, we are ready to state our problem:

Problem Statement 1. *For ASU campus, given a pathfinding grid, school schedule data and a query $q \rightarrow (s, e, T)$, compute a fast route q_r between nodes s and e starting from s at time T, such that q_r takes minimum time with respect to crowd density and distance between s and e.*

We can see from the problem definition that we are interested in finding fast paths, that are optimal by way of crowd density along the path and the distance taken by the path. These two factors decide the time it takes to reach the goal.

The problem definition encompasses factors beyond traditional pathfinding and are optimized for bike routes in ASU campus.

3. PATHFINDING

In pathfinding, algorithms like Dijkstra's wastes time exploring in directions which are not promising. On the contrary, Greedy Best First Search explores only nodes which are closer to the destination. But it may not find the optimum path. A* search algorithm combines the best of both worlds by considering minimum cost path in the direction of the destination. The cost function in basic A* search algorithm is given by:

$$f(n) = g(n) + h(n)$$

$g(n) \implies$ exact cost of path from start node to a node n
$h(n) \implies$ heuristic estimated cost of path from node n to destination node The modified A* search algorithm in this paper considers crowd density along the path as a variable of time. The cost function in our algorithm is given by:

$$f(n,t) = g(n,t) \times cdf_{n,t} + h(n,t)$$

$cdf_{n,t} \implies$ crowd density factor of node n at time t

The crowd density factor *cdf* is equivalent to the crowd density i.e. number of pedestrians at the node n at time t. Section 4 explains crowd density factor in detail.

3.1 Heuristic Function

The heuristic function estimates the cost for path from a node under consideration to the destination node. Depending on the value of the heuristic function, the algorithm prefers the nodes which are closer to the destination node thereby reducing the search space.

For our algorithm, the heuristic function considers the Manhattan distance between the current node and the destination node. Manhattan distance assumes 4 directions of movement on the grid. We have considered Manhattan distance over other distance measurements like Euclidean distance because most of the paths in the campus map coincide with a square grid and very few are diagonal.

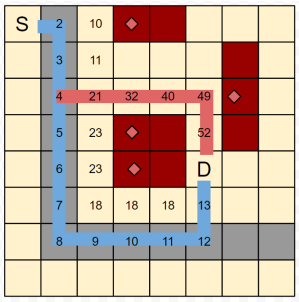

Figure 1: Example of pathfinding. S is start and D is goal. Diamonds represent buildings with scheduled classes. Numbers indicate path costs from S.

Figure 1 is an example of pathfinding by our algorithm. Red route is costlier because of the high crowd density along its path whereas blue route is cheaper (although longer in distance) to reach destination D. The heuristic function significantly improves the efficiency of A* search algorithm by reducing the search space. The heuristic costs of nodes that are farther from destination are more and the algorithm tends to avoid those nodes.

4. CROWD DENSITY FACTOR

The crowd density factor is derived from the crowd density data from the school schedule dataset. The school schedule dataset contains number of students per class in a class schedule. This gives an estimation of the quantity of students rushing in or out of the class before the class starts or after the class ends respectively. This class strength grouped together with strengths of classes in nearby buildings gives an estimate about the crowd density in the given neighborhood. The school schedule dataset is presumed to be in following format for ease of understanding:

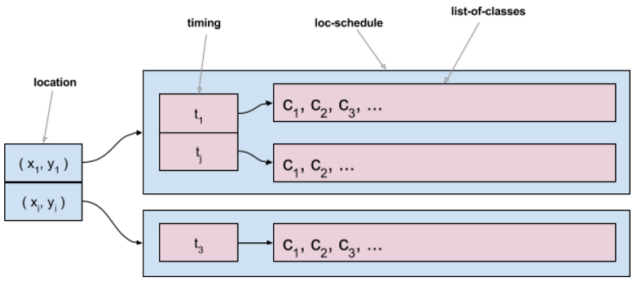

Figure 2: School schedule data structure

Figure 2 shows the structure of the schedule. It is a hash table with `<location, loc-schedule>` as key-value pair. `loc-schedule` represents schedule of a particular location and is also a hash table with `<timing, list-of-classes>` as key-value pair. The `list-of-classes` specify the class size (number of students) for each class. This class size number c_i is equivalent to crowd density and is fetched given the location and time; crowd density factor *cdf* is calculated using this number.

The points in the school schedule dataset are locations of buildings in campus that have the scheduled classes or events. These points are obstacles on the grid and never coincide with a node. The crowd density (class size) at a node is summation of crowd densities at obstacles adjacent to that node.

45	45	76	10	31
45	45	76	21	31
50	50	66	21	21
5	92	87	98	11
5	92	87	0	11

Figure 3: Example of net crowd densities $\mathcal{C}$ (blue) of nodes and crowd densities c (black) of obstacles in grid. Crowd density of a node is 0 but it may have a non-zero net crowd density as shown.

$$cdf = \begin{cases} 0.500 & \text{if } \mathcal{C} \geq 100 \\ 0.333 & \text{if } 40 \leq \mathcal{C} < 100 \\ 0.167 & \text{if } 0 \leq \mathcal{C} < 40 \end{cases}$$

The crowd density factor *cdf* of a node depends on the net crowd density $\mathcal{C}$ at that node . It is formulated as follows:

$$\mathcal{C} = \sum_{neighbors} c_i$$

5. THE ALGORITHM

At this point we are ready to state the fastest path algorithm, it dynamically computes path costs, takes into consideration the crowd density factor and heuristic estimation to find the quickest path. The crowd density factor and terrain cost are computed by the `cost`() function. The concepts of the algorithm are summarized below:

- The input to the algorithm is the map structure (grid) containing hash sets of obstacles and walk-only zones, school schedule data hashtable, start node, destination node and start time.
- The algorithm maintains a priority queue reach of expanded paths (represented by the last node of the path); new nodes are enqueued in the priority queue with the priority values as the sum of their travel cost and heuristic cost. At each step of the search process, node with lowest cost is picked. The algorithm prefers to travel through roads or walkways with less crowd density.

Algorithm 1: A* Search

Input : *map*, *start*, *goal*, *start_time*
Output: List of nodes that are antecedent to nodes represented by corresponding list indexes

```
1  reach ← PriorityQueue();
2  reach.put(start, 0);
3  antecedents ← {};
4  path_costs ← {};
5  antecedents[start] ← None;
6  path_costs[start] ← start_time;
7  while not reach.empty() do
8      current ← reach.get();
9      if current == goal then break;
10     for next in map.neighbors(current,
         path_costs[current]) do
11         new_cost ← path_costs[current] + map.cost(next,
             path_costs[current]);
12         if next not in path_costs or new_cost <
             path_costs[next] then
13             path_costs[next] ← new_cost;
14             priority ← new_cost+heuristic(goal, next);
15             reach.put(next, priority)
                 antecedents[next] ← current;
16 return antecedents;
```

- The next node to consider during pathfinding is one of the neighbors of the current node. The neighbors of the current node are nodes spatially separated along the horizontal and vertical axes of the pathfinding grid. Note that in pathfinding grid, neighbors of a node are given by the points to the left, right, top and bottom. If a point is an obstacle, it is not considered for pathfinding. The neighbor node is also checked against time as walk-only zones are treated as obstacles for bikes in particular time periods.

Algorithm 2: Cost function `cost`

Input : *node*, *time_at_current*
Output: Path cost for the *node*

```
1  time_from_current ← 0.125 if node in map.roads else
     0.133;
2  path_time ← time_from_current + time_at_current;
3  people ← 0;
4  cdf ← 1;
5  for neighbor in map.obstacles_nearby(node, path_time) do
6      if neighbor in map.schedule then
7          for event_strength in
             map.schedule[neighbor][path_time] do
8              people ← people + event_strength;
9          if people ≥ HIGH then cdf ← 1.5;
10         else if people > MEDIUM then cdf ← 1.333;
11         else if people > LOW then cdf ← 1.167;
12 return path_time × cdf;
```

- The `path_costs` hash table is maintained to store the total cost for each node considered by the algorithm. This cost is the exact cost (terrain and crowd density) to travel from start to the node. The `antecedents` hash table is maintained to store the antecedent or the previous node to the current node in the shortest path.
- The solution path can be recreated from the `antecedents` hash table.

Algorithm 3: Heuristic function `heuristic`

Input : *node*, *goal*
Output: Estimated time to travel from current *node* to destination *goal*

```
1  (x1, y1) ← node;
2  (x2, y2) ← goal;
3  return (| (x1 − x2) | + | (y1, −y2) |)/800;
```

- The time calculation for elapsed time to reach a node are conservative approximations by considering average value of bike speed through roadways, walkways and various crowd densities. The crowd density factor is multiplied to the terrain cost depending on the crowd density computed at that node. Following presumptions are made for pathfinding in campus:
  - Distance between adjacent nodes in the grid is 100 feet.
  - Average speed of bike is 800 feet per minute on road and 750 feet per minute on walkway.
  - With above figures, the time taken to travel from one node to its adjacent node is 0.125 minute and 0.133 minute on road and walkway respectively. Its takes slightly more time on walkway because of pedestrian traffic.

6. ANALYSIS

The proposed algorithm uses a priority queue. In the worst case, the algorithm may explore all the nodes making the length of the priority queue to n which equals the total

number of nodes in the grid. For each node in the queue, a list of neighbors excluding obstacles and walk only zones is retrieved. Two hash sets: one for obstacles and other for walk-only zones are used. Hence, checking whether the 8 neighbors of a given node are obstacles or not is a constant operation.

For every neighbor found, the `cost()` function is called which first finds the nearby obstacles and returns the crowd density factor after calculating the number of people in the neighborhood. Computation of finding nearby obstacles again takes constant time due to use of hash sets. Now for each obstacle found, schedule hash table returns a list of number of students per class at that particular time. If the number of classes in the list is m, summation of class sizes will asymptotically take $O(m)$ time. Therefore the time complexity of `cost()` function is $O(m)$.

Once the actual travel cost is calculated, heuristic cost can be found in constant time as per the Manhattan distance formula. Thus, for each node in n nodes, the algorithm calculates path costs in linear time making the overall complexity to $O(n \times m)$.

Now, we will see how our algorithm selects the best node.

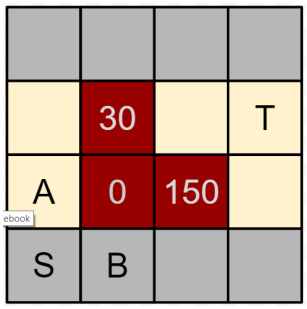

Figure 4: Sample grid. S is start, T is goal. Red cells represent obstacles and the numbers represent crowd densities. Yellow cells are walkways and grey cells represent plain roads. The distance between each cell is 100 feet.

In order decide which node should be visited first, $f(n)$ is calculated for both nodes A and B.

	Node A	**Node** B
$g(n)$	0.133	0.125
cdf_n	$0.133 \times 1.167 = 0.15$	$0.125 \times 1.5 = 0.19$
$h(n)$	4	4
$f(n)$	4.15	4.19

The algorithm gives different weights to nodes depending on its type. Our assumption is that if a node is walkway within a campus then it would take more time than plain roads. As per our observation, time required to travel from a current node to walkway is 0.133 minutes and if it's a plain road it would take 0.125 minutes (assuming speed at walkway node is 750 feet per min and for plain roads 800 feet per min). To calculate *cdf* at node A, algorithm scans all the neighboring nodes of A and finds the total crowd density which is equal to 30 and hence, can be classified as LOW . Thus, $g_A(n)$ will be multiplied by 1.167. Similarly, crowd density at B is 150 which is $HIGH$ and $g_B(n)$ will be multiplied by 1.5. Heuristic cost for A i.e. $h_A(n)$ is 4 which equals the Manhattan distance from A to T. $h_B(n)$ is also 4. So, the total cost $f_A(n)$ is 4.15 and $f_B(n)$ is 4.19. Thus, from *Table 1* it is clearly evident that node A will be preferred over B as $f_A(n) < f_B(n)$ best node if it hits a dead end.

7. IMPLEMENTATION

We have implemented our customized A* search algorithm in Python. As per the above algorithm, it returns a list of coordinates of nodes that form the optimal path. It is available under MIT License at the following online repository:

`https://github.com/AjinxPatil/asu-bikerouting-algorithm`

8. RELATED WORK

Shortest Path algorithms has been a hot research topic since a few decades now. There are many research papers published on this topic and we have highlighted few of them.

A* search algorithm is the most popular algorithm in pathfinding. [12] answers various basic questions related to A* in great detail like how to represent maps in A*, calculation of movement costs, selecting admissible heuristics to get optimal results from A*. It also has a number of interactive visualizations that aids in deep understanding of these algorithms and hence, it proved to be a great start for our research on A* search algorithm.

Heuristic function is one of the most important factor that impacts the A* performance. [11] provides a nice description about the use of heuristic, tradeoff between speed and accuracy, special heuristics for grid maps and their selection criteria. Effects of inconsistent heuristics on the performance of A* has been explained in [14] which guided us to use consistent admissible heuristics function for our grid map.

A number of optimization techniques for A* and some real examples of how A* is used for pathfinding in real games along with the comparison between different A*-based algorithms have been discussed in [2]. To reduce the time complexity of computation, the concept of edge hierarchies have been used by some algorithms[10][7]. Very recently, algorithms have used a combination of edge hierarchies[13][3] and pre-calculated chosen paths. Some approximation algorithms have also suggested the idea of splitting the graph using large roads[1]. Recent algorithms[4] have also concentrated on improvising on the heuristic function used by A* search algorithm. However, all of these papers do not consider the dynamic characteristics of a graph which is required by our problem definition. The first phase of our research work focused on understanding how A* works on static graphs and different factors that impacts the performance of A*. In the second phase, we studied how we can adapt A* search algorithm in order to account for the changes in the weights of graph edges due to crowd density factor.

One can interpret shortest path calculation for dynamic graphs in a couple of ways. The first type is the one in which edges are added, removed and changed. The method used to deal with this dynamic behaviour is to use Dijkstra 's[5] on the part of the graph where the quickest path changes after update. Another paper [6] compared some of the most frequently used shortest path algorithms for dynamic graphs. The second type is one in which speed on an edge is based on time. One paper [9] uses an adaption of A* to match the speed on an edge with the time of the day. In this paper a function of cost was used based on the start time. This paper was focused on giving the fastest path at a given time in

a day. [8] presents an adaptive fastest path algorithm that takes into account the important driving and speed patterns mined from a large set of traffic data. The algorithm presented in [8] gave us the idea about the assumptions and different factors we can consider while deciding the total cost of a node, strictly in terms of time factor. The contents of [8] were thoroughly studied in order to efficiently incorporate the class schedule database in calculating the crowd density factor which is one of the key elements of our algorithm.

9. CONCLUSION

We developed an algorithm that optimizes biking routes in the university campus. We have modified the existing A* search algorithm to also take into account the density of students in one particular area of the campus, when suggesting the shortest path to the destination. The number of students in a particular area depends on the finish times of the classes in that area and the number of students in each class. A person looking to travel using a bike will get the best possible route at that time. The algorithm also considers the amount of time a bike rider will require to reach that area and the crowd density in that area at that time. A phone application build on this algorithm can help a lot of students to reach their next class faster.

10. REFERENCES

[1] Y.-L. Chou, H. E. Romeijn, and R. L. Smith. Approximating shortest paths in large-scale networks with an application to intelligent transportation systems. *INFORMS Journal on Computing*, 10(2):163–179, February 1998.

[2] X. Cui and H. Shi. A*-based pathfinding in modern computer games. *International Journal of Computer Science and Network Security*, 11(1):125–130, 2011.

[3] D. Delling, P. Sanders, D. Schultes, and D. Wagner. Highway hierarchies star. In *9th DIMACS Implementation Challenge*, 2006.

[4] C. Demetrescu, S. Emiliozzi, and G. F. Italiano. Experimental analysis of dynamic all pairs shortest path algorithms. In *SODA '04 Proceedings*, pages 369–378, New Orleans, United States, January 2004. 15th Annual ACM-SIAM Symposium on Discrete Algorithms.

[5] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, December 1959.

[6] D. Frigioni, M. Ioffreda, U. Nanni, and G. Pasqualone. Experimental analysis of dynamic algorithms for the single source shortest paths problem. *Journal of Experimental Algorithmics*, 3(5), 1998.

[7] L. Fu, D. Sun, and L. R. Rilett. Heuristic shortest path algorithms for transportation applications: State of the art. *Computers and Operations Research*, 33(11):3324–3343, November 2006.

[8] H. Gonzalez, J. Han, X. Li, M. Myslinska, and J. P. Sondag. Adaptive fastest path computation on a road network: a traffic mining approach. In *VLDB '07 Proceedings*, pages 794–805, Vienna, Austria, September 2007. 33rd International Conference on Very Large Databases.

[9] E. Kanoulas, Y. Du, T. Xia, and D. Zhang. Finding fastest paths on a road network with speed patterns. In *ICDE '06 Proceedings*, page 10. 22nd International Conference on Data Engineering, April 2006.

[10] S. Pallottino and M. G. Scutella. Shortest path algorithms in transportation models: Classical and innovative aspects. Technical Report TR-97-06, Department of Computer Science, University of Pisa, April 1997.

[11] A. Patel. Heuristics — red blob games. http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html, 2014.

[12] A. Patel. Introduction to a* — red blob games. http://www.redblobgames.com/pathfinding/a-star/introduction.html, 2014.

[13] P. Sanders and D. Schultes. Highway hierarchies hasten exact shortest path queries. In *ESA'05 Proceedings*, pages 568–579, Palma de Mallorca, Spain, October 2005. 13th Annual European Conference on Algorithms.

[14] Z. Zhang, N. R. Sturtevant, R. Holte, J. Schaeffer, and A. Felner. A* search with inconsistent heuristics. In *IJCAI '09 Proceedings*, 2009.