Improved Runtime Heuristic Functions on General Maps

Abstract

For A* search to work optimally, an effective heuristic function is necessary. Manhattan distance and Euclidean distance are widely used for solving path-finding problems in maps, but they fail to deliver optimal solutions within a short runtime and do not scale well on large and complex maps. This paper introduces and examines two advanced heuristics for path-finding problems in maps: waypoint heuristic and gateway heuristic, and the technique of map abstraction, which is significant for scalability of these heuristics. We conclude that waypoint heuristic and gateway heuristic deliver optimal paths while preserving acceptable runtime complexity.

Tianye Ding
Northeastern University
Boston, MA
ding.tian@northeastern.edu

Jiacheng Pang
Northeastern University
Boston, MA
pang.ji@northeastern.edu

1 INTRODUCTION

With the modern progression of computer science and its expanding applications, path-finding algorithms are undoubtedly going to be applied to more complicated maps and are still required to find the optimal solution within limited running time and resources for the best performance.

1.1 Motivation

A* search, as one of the most stable and widely used modern path-finding algorithms, takes the cost of the path into consideration combined with a heuristic to guide its exploring direction towards the final goal we desire. For A* search to produce a reliable result, the heuristic functions are given two standards: admissibility - never overestimating the actual cost from the current vertex to the goal, and consistency - never overestimating the cost of an edge connecting two neighboring vertices.

Apart from the abstract graphs which we gain no insight into how to get the heuristic by solely looking at the graph itself, for graphical representation of any game map or real-world map by converting them to grid maps, the most primitive and widely used heuristic functions under these scenarios are Manhattan distance and Euclidean distance. Since they compute the cost to the goal under the best-case scenario - a plain field with no obstacles, it is guaranteed to be admissible and consistent. However, with the map becoming more complicated, the issue of computational performance raises, since the naive approaches can only show us the general direction but neglect the detailed setups within the map (Figure 1), which results in search agents exploring unnecessary paths that lead to a dead-end, or paths that lead to suboptimal solutions. The most ideal heuristic function should be capable of calculating solutions with cost as close to the minimum as possible from any arbitrary point on the map to the goal, this paper will discuss different approaches to develop ideal heuristic functions under any map setups with optimal running time and least nodes expanded.

1.2 Challenges

Recall that the decision made by A* search upon which node to explore next is reliant on the sum of *g* - cost from starting point to next state and *h* - heuristic which is the estimated cost from next state to the goal state. The major challenge of developing a general heuristic function is to find attributes that can be applied to any circumstance (generalization) and will not cause an exponential increase in its running time as the map size grows bigger or increase in complexity (scalability).

The way we deliver the final heuristic value to the A* search gives us the flexibility to control the behavior of our search agent. For instance, if we weigh down *h* and make *g* plays a bigger role in the path-finding process, the search agent will explore more vertices. Its running speed becomes slower, but it guarantees to find the optimal path. If we weigh up *h* instead, the algorithm will turn into a greedy Breadth-First Search following the decreasing flow of the heuristic values. This enables the agent to automatically break ties when deciding which vertex to explore next, resulting in exploring fewer nodes and consuming less time but, as a tradeoff, it does not guarantee to find the lowest cost path and may sacrifice admissibility and consistency. The desirable case is that the heuristic function can produce the exact cost from any point to the goal state, given the perfect information then we can make our A* search agent behave perfectly.

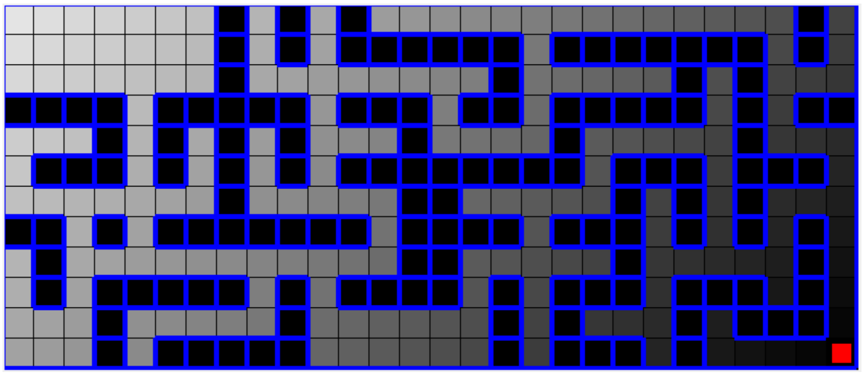
Figure 1: Example map 1: visualize naive heuristic function using gradient color change

1.3 Overview

Before running the A* search and heuristic function, one trick can be used to reduce the computation time during the search is by storing a precomputed list of important data which we can use afterward, but running time and space usage of the list need to be small and grow linearly in the worst case when the map increases in size. Our approach used to tackle the ideal heuristic function is inspired by the human perspective, namely how we humans tend to solve any given maze. We attempt to formulate the methodology into functions.

The first and most essential step is to compress the complexity of the map, when we examine a map, we do not pay close attention to every single grid but pay more attention to the bigger picture, automatically extract all the important nodes within the map, such as crossings, turning points, gateways, and the final goal, etc. Finding a reliable and efficient solution to abstract the entire map into generalized properties can greatly reduce the running time and resources occupied by the heuristic function.

The second step is back propagation, as a heuristic is an increasing count starting from the goal state, we can use a certain dynamic programming approach to propagate the heuristic values from the goal to all the other vertices within our abstracted graph of the map and store the values into the pre-computed list. This enables us to compress the complexity of mazes and generalize attributes of mazes.

2 RELATED WORKS

Decomposition Algorithm (Figure 2)

A decomposition algorithm from *Improved Heuristics for Optimal Pathfinding on Game Maps* (Yngvi B. & Kari H. 2006) uses a flooding-filling technique for all passable tiles within the map. Starting from the top left passable tile that has not been assigned to a zone, it fills to the right until it hits a non-free tile (a previously assigned tile or an impassable tile). It then proceeds to the next row by selecting another free tile from the top left.

The algorithm also detects if the left or right border of the current zone has grown or shrunk from the previous row. If either left or right border grows after shrinking, it possibly has gone into the next zone, which causes the filling process of the current zone to stop. The same detection rule also applies to top and bottom boundaries. When flooding into a narrow path, if either top or bottom of the next tile is opened up to more free tiles, the filling process stops as it goes into another neighboring zone. The process repeats until all tiles within the map are assigned to a zone.

This method dissects a complex maze into zones. Each zone can be represented as a node in a graph and the entrances of zones as the edges in a graph. This way, a complex maze can be turned into a multi-graph, which is much easier to analyze.

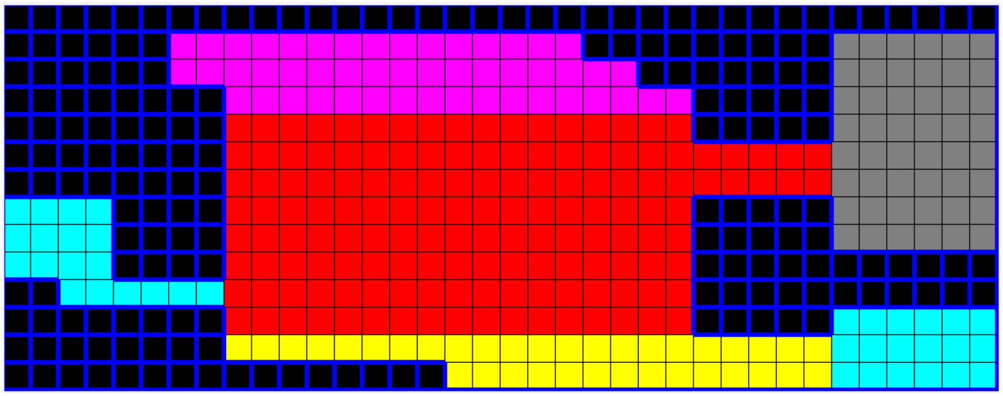
Figure 2: Example map 2: decomposition algorithm being applied to a room-like map

3 APPROACHES

3.1 The Testing Ground

To program, visualize, and evaluate various heuristics, we developed a maze simulation program similar to a grid world. The program supports customizable mazes, search algorithms, and heuristics. It also keeps track of the performance of heuristics in terms of time consumption and the number of nodes expanded. The program contains a GUI that displays the map, the position of the agent, nodes explored, and different attributes of the maze, which we will introduce in the following sections.

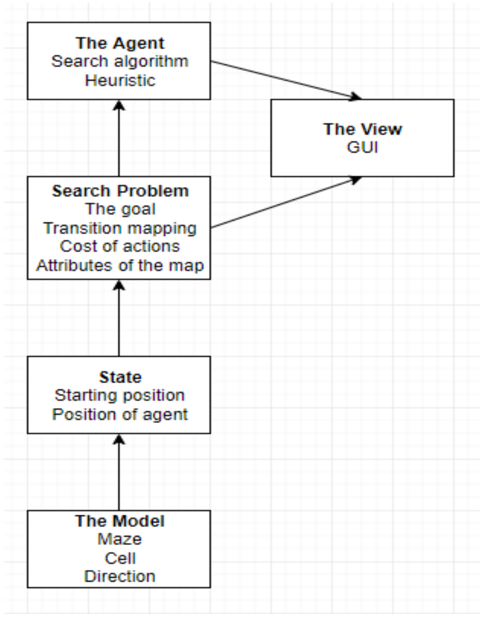

Figure 3: Flowchart of the testing ground program

3.2 Waypoint Heuristic

For a map where all paths are exactly one unit wide, the waypoint heuristic can immediately generate an abstract graph by marking out all turning points and crossings of paths within the map. This heuristic proceeds in three phases.

Extraction Phase The waypoint extraction process is done by traversing through all passable cells within the map and checking the opening directions of each cell.

Algorithm 1 Automatic Waypoint Extraction

```
Function generateSuccessors((x, y))
    if not passable((x, y)) then
        return None
    successors ← []
    for all available directions do
        target ← (x, y) + direction
        if passable(target) then
            successors ← target
    return successors

waypoints ← empty set
for all passable cells within the map do
    successors ← generateSuccessors(cell)
    { Node at crossing point }
    if len(successors) > 2 then
        waypoints ← cell
    else if len(successors) = 2 then
        s1 ← successors[0]
        s2 ← successors[1]
        { Check not on a straight line}
        if s1.x != s2.x ∧ s1.y != s2.y do
            waypoints ← cell
```

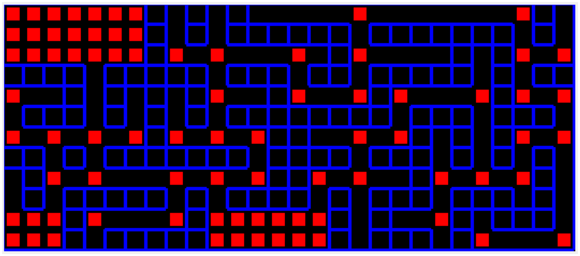
Figure 4: Example map 1: waypoint extraction

From this conditional extraction of waypoints, we can omit all the nodes along a straight line (with two successors) and the dead-end nodes (with only one successor). The property of such methodology is that every vertex in the abstracted graph can reach its nearby vertices through simply a straight-line translation.

Back Propagation Phase Given a certain goal state, we will use back propagation by utilizing the property described above to generate the pre-computed list of exact heuristics from goal to all waypoints, following an approach that resembles Dijkstra's algorithm:

$$h_v = min_{u \in U} \{g(u, v) + h_u\}$$

Starting from the goal, we denote it as a waypoint with a heuristic value of 0. Any unexplored vertices within the graph will be initialized with a heuristic value of $+\infty$. Ideally, from bottom-up dynamic programming, h_v stands for an unexplored vertex, U represents all nearby vertices within the graph connected with v, and $g(u, v)$ stands for the cost of the straight-line path connecting u and v, under the ideal scenario where passing every tile on a straight path within the map has approximately the same cost, $g(u, v)$ can be computed with only O(1) running time.

Runtime Phase During runtime, the heuristic function used by the A* star search is a simplified version of the algorithm used for back propagation. From the starting point, expand a straight line upon its all available directions and is guaranteed to hit a waypoint that is stored in the precomputed list, the resultant heuristic value will be the minimum sum of the path costs from starting to the waypoints and the stored heuristic values of the waypoints.

This heuristic function is guaranteed to produce values that are close estimates to the exact cost from any point on the map to the goal state that we are looking for. But the limitation of the Waypoint Heuristic is also obvious, it will only be efficient on certain types of preprocessed maps that match the criteria of all paths are exactly one unit wide. As shown in Figure 4, if it encounters even a slightly opened space, the waypoint extraction process will simply denote every single tile within the area which is not a desirable way of reducing overall complexity if the provided map has a room-like feature as shown in Example map 2.

3.3 Gateway Heuristic

Gateway heuristic combines the map decomposition function, as mentioned above, with the concept of extracting critical points within the map as waypoints and precomputing their heuristic values to speed up the calculation of the heuristic function during runtime. This heuristic works by extracting the nodes on both sides of the boundaries between different zones as waypoints. "Gateways" will be the more appropriate term under this circumstance, the heuristic has four phases.

Decomposition Phase The first phase will preprocess the map by applying the decomposition function, mentioned in the Related Work section, to dissect the entire map into zones. Based on the mechanism of how the zones are divided (a conditional flood filling algorithm) and the observation of the visualized result of its running result (Figure 2), we can make an assumption about a property of this decomposition algorithm: for any two arbitrary points within the same zone, we can calculate a theoretically exact distance between them by applying the Manhattan distance function. The main issue that causes the naive Manhattan distance function to be unable to handle maps that grow in complexity is the distance function cannot give an estimate of the cost of bypassing the obstacle when a single path involves more than one turning point. The decomposition algorithm will separate two sides of a turning into different zones, especially for vertical turnings, which makes the Manhattan distance function a reliable way to calculate the distance between two points within the same zone.

Gateway Extraction Phase After different zones are properly divided across the map, the process of extracting gateways from the map is fairly simple, by traversing through the entire map and denoting all tiles located along the boundaries of two different zones. But this time we also need to store the corresponding gateways on the other side of the boundary to be used for back propagation.

Algorithm 2 Automatic Gateway Extraction

```
gateways ← empty map
for all passable cells within the map do
    currentZone ← zone(cell)
    gatewaySwitch ← false
    targetGates ← []
    for all available directions do
        targetCell ← cell + direction
        if zone(targetCell) != currentZone do
            targetGates ← targetCell
            gatewaySwitch ← true
    if gatewaySwitch do
        gateways[currentZone]
        ← (cell, targetGates)
```

Within the gateways map, for every zone section, we store its gateways and the gateways' corresponding other gateways to nearby zones. One thing to notice is that the distance from each gateway to its corresponding gateways in the nearby zone is exactly one unit.

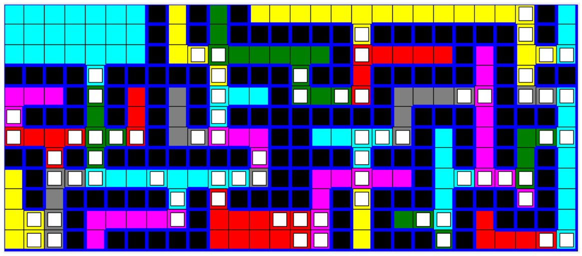
Figure 5: Example map 1: gateway extraction

Back Propagation Phase When given a certain goal state, the back propagation of heuristic value is done through a bottom-up dynamic programming approach. First, initialize the heuristic value of all gateways to be +∞, then confirm which zone the goal is located in from the existing zone division. Second, for all gateways of that zone, update their heuristic values to be the Manhattan distance to the goal, then propagate the heuristic values to the nearby zone by incrementing the values by 1 and assigning them to the target gateways stored in the map. The recursive step takes place when propagating the heuristic within the same zone - from one side of the gateways to another side - and passing down to the next nearby zone.

Algorithm 3 Heuristic Propagation

```
gateways ← generate gateway map through extraction
goal ← position of the goal state
Function heuristicPropagate(gate)
  currentZone ← zone(gate)
  waypoints ← gateways[currentZone]
  { Propagate heuristic within the zone }
  if len(waypoints) > 1 then
    for all starts in waypoints do
      for all ends in waypoints do
        startHeu ← heuristic(start[0])
        targetHeu ← startHeu
        + manhattanDistance(start[0], end[0])
        currentHeu ← heuristic(end[0])
        heuristic(end[0]) ← min(targetHeu, currentHeu)
  { Propagate to the next zone }
  for all gateways in waypoints do
    heu ← heuristic(gateway[0])
    targetHeu ← heu + 1
    for all nextGates in gateway[1] do
      if targetHeu < heuristic(nextGate) then
        heuristic(nextGate) ← targetHeu
        heuristicPropagate(nextGate)

goalZone ← zone(goal)
heuristic(goal) ← 0
gates ← gateways[goalZone]
for all gates in gates do
  pos ← gate[0]
  heu ← manhattanDistance(pos, goal)
  heuristic(pos) ← heu
  for all nextGates in gate[1] do
    heuristic(nextGate) ← heu + 1
    heuristicPropagate(nextGate)
```

The back propagation process terminates when reaching a gateway of the nearby zone having a stored heuristic value less than or equal to the heuristic value of current gateway, and the stored heuristic values of all gateways will eventually converge to the minimum estimate of the distance towards the goal following an optimal path.

Runtime Phase During runtime, the heuristic function used by the A* search follows a simple approach of finding the minimal sum of the Manhattan distance between the current point to all gateways of the current zone and their stored heuristic values:

$$h^Z_v = min_{g \in G^Z} \{manhattanDistance(g, v) + h_g\}$$

h^Z_v stands for the heuristic value of any point v located in zone Z, G^Z represents all gateways within zone Z and h_g is the stored heuristic of gateway g.

This heuristic function is theoretically guaranteed to produce an exact estimate of the distance between any point within the map and the goal and is generalized enough to be applied to any map setup.

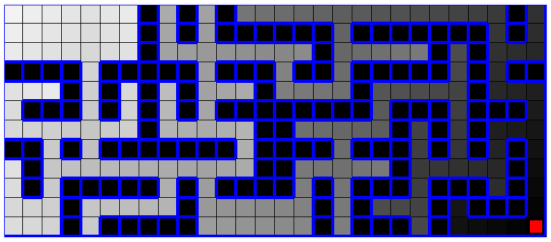
Figure 6: Example map 1: Gateway Heuristic

3.4 Approach Summaries

As shown in Figure 6, a visualization of the Gateway Heuristic in Example map 1 setup, brighter areas represent a higher heuristic value. Compared to the visualization of the Manhattan distance in Figure 1, the obvious brightness increase in certain parts of the map indicates the actual cost from that point to the goal is higher than the value produced by the naive approach. This mostly consists of dead-ends. The gradient color changes form some visible flow pattern from any point on the map towards the red dot at the bottom right corner of the maze - representing the search goal.

The Waypoints Heuristic does a better job when encountering a narrow path map with possible cost changes for passing different paths within the map, and can produce a close estimate of the minimal actual cost. The Gateway Heuristic is more generalized and significantly better on maps with more room setups and open areas by extracting the critical points for entering and exiting divided zones, whereas it cannot reflect the actual cost changes when there are subtle variations in the costs for passing different tiles.

Moreover, if we are under the most ideal map setups for each heuristic, with identical path costs across the entire map, we can utilize the flowing pattern within the maps generated by our improved heuristics. By weighing up the output from the heuristic function to suppress the effect of *g* when making exploring decisions and force our search agent to follow the gradient change of the heuristic value. Although this results in the heuristic values becoming inadmissible and inconsistent, our search agent now can explore significantly fewer nodes while still producing the optimal path.

4 EVALUATIONS

The evaluation of the heuristic will be focusing more on the running time for each stage of the heuristic functions and their space complexity using Big O notation, with the factors that cause them to increase. In the following evaluations, *m* and *n* represent the height and width of the map.

4.1 Waypoint Heuristic

The waypoint extraction phase basically traverses through every tile on the map so that its running time is always O(mn). Disregard maps with many open areas as they are not the subject of interest for applying the Waypoint Heuristic. The number of stored waypoints increases as the map has more crossing points and the limit for space usage of precomputed values is O(mn/4) which is also the running time for back propagating the heuristic values. During runtime when calculating the heuristic value for any point within the map, the limit of running time is O(m + n) as it only needs to find a waypoint along the straight line of available directions.

Therefore, the complexity of the waypoint extraction phase is O(mn) and grows linearly as the size of the maze gets bigger. It is only run once at the beginning of the program.

4.2 Gateway Heuristic

The decomposition phase needs to flood through the entire map which takes a running time of O(mn). Extracting the gateways from separated zones also takes a running time of O(mn), the space usage for storing the waypoints depends on the number of bottleneck-like obstacles that the map has. In the most extreme cases, the space complexity is O(mn). While in back propagation it needs to update the heuristic values of each extract gateway, which makes the running time also be O(*mn*) under the worst cases. During runtime, the time complexity for calculating the heuristic for any points on the map depends on the number of gateways that the current zone has, which is O(2*max*(*m*, *n*)) as the maximum number of gateways on one side of any zone cannot surpass either height or width of the map.

Therefore, the complexity within the preprocessing stages is O(*mn*) and grows linearly with the size of the maze. It is only run once at the beginning of the program.

5 CONCLUSIONS

The waypoint heuristic and gateway heuristic are both attempts that keep beyond the simplistic Manhattan and Euclidean distance, which fail to consider the obstacles or the complexity of maps. These two advanced heuristics achieve optimal solutions within an acceptable runtime, and scalability towards highly sizable and complex maps. They share a common technique: map abstraction, which proved to be an excellent way to convert a seemingly large and complex map into graphs for easier analysis.

6 FUTURE WORKS

In the scope of this paper, the heuristics we examined are designed to work on single-goal search problems in a maze. They can potentially be further designed to solve multi-goal scheduling problems. We recognize that multi-goal scheduling is an NP-hard problem in graphs and that our techniques convert maps into graphs. Therefore, these heuristics will also belong to NP-hard problems. Further attempts are required by seeking heuristics that offer to provide optimal solutions with scalability.

7 ACKNOWLEDGEMENTS

We would like to thank Professor Stacy Marsella for providing us with insights into our topic and resources for our research.

The prototype of the decomposition algorithm used in the research is provided by Yngvi Bjornsson's and Kari Halldorsson's papers in the References section.

8 REFERENCES

Andrew V. G. and Chris H., 2004, Microsoft Research: *Computing the Shortest Path: A* Search Meets Graph Theory*, pp. 1-24. http://research.microsoft.com/pubs/154937/soda05.pdf

Chris R. and Michael B., 2011, E*uclidean Heuristic Optimization*, pp. 1-6. http://webdocs.cs.ualberta.ca/~bowling/papers/11aaai-heuristicopt.pdf

Yngvi B. and Kari H., 2006, Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment: *Improved Heuristics for Optimal Pathfinding on Game Maps*, vol. 2, pp. 9-14