Monterrey Institute of Technology
and Higher Education

Intelligent Agents Final Project

Smart Parking using A* Algorithm

Authors:
Jose Ramon Obeso
Obed N Muñoz

Professor:
Dr. Gildardo Sanchez

May 13, 2014

Contents

Chapter 1

Background

Testing stuff

Chapter 2

Problem

Chapter 3

Solution

3.1 A* Algorithm

A* uses a best-first search and finds a least-cost path from a given initial node to one goal node (out of one or more possible goals). As A* traverses the graph, it follows a path of the lowest expected total cost or distance, keeping a sorted priority queue of alternate path segments along the way.

It uses a knowledge-plus-heuristic cost function of node x (usually denoted f(x)) to determine the order in which the search visits nodes in the tree. The cost function is a sum of two functions:

- The past path-cost function, which is the known distance from the starting node to the current node x (usually denoted g(x))
- A future path-cost function, which is an admissible "heuristic estimate" of the distance from x to the goal (usually denoted h(x)).

3.1.1 A* Algorithm Process

The h(x) part of the f(x) function must be an admissible heuristic; that is, it must not overestimate the distance to the goal. Thus, for an application like routing, h(x) might represent the straight-line distance to the goal, since that is physically the smallest possible distance between any two points or nodes.

Like all informed search algorithms, it first searches the routes that appear to be most likely to lead towards the goal. What sets A* apart from a

greedy best-first search is that it also takes the distance already traveled into account; the g(x) part of the heuristic is the cost from the starting point, not simply the local cost from the previously expanded node.

Starting with the initial node, it maintains a priority queue of nodes to be traversed, known as the open set or fringe. The lower f(x) for a given node x, the higher its priority. At each step of the algorithm, the node with the lowest f(x) value is removed from the queue, the f and g values of its neighbors are updated accordingly, and these neighbors are added to the queue. The algorithm continues until a goal node has a lower f value than any node in the queue (or until the queue is empty). (Goal nodes may be passed over multiple times if there remain other nodes with lower f values, as they may lead to a shorter path to a goal.) The f value of the goal is then the length of the shortest path, since h at the goal is zero in an admissible heuristic.

The algorithm described so far gives us only the length of the shortest path. To find the actual sequence of steps, the algorithm can be easily revised so that each node on the path keeps track of its predecessor. After this algorithm is run, the ending node will point to its predecessor, and so on, until some node's predecessor is the start node.

```
function A*(start,goal)
    closedset := the empty set    // The set of nodes already evaluated.
    openset := {start}    // The set of tentative nodes to be evaluated, initially containing the start node
    came_from := the empty map    // The map of navigated nodes.

    g_score[start] := 0    // Cost from start along best known path.
    // Estimated total cost from start to goal through y.
    f_score[start] := g_score[start] + heuristic_cost_estimate(start, goal)

    while openset is not empty
        current := the node in openset having the lowest f_score[] value
        if current = goal
            return reconstruct_path(came_from, goal)

        remove current from openset
        add current to closedset
        for each neighbor in neighbor_nodes(current)
            if neighbor in closedset
                continue
            tentative_g_score := g_score[current] + dist_between(current,neighbor)

            if neighbor not in openset or tentative_g_score < g_score[neighbor]
                came_from[neighbor] := current
                g_score[neighbor] := tentative_g_score
                f_score[neighbor] := g_score[neighbor] + heuristic_cost_estimate(neighbor, goal)
                if neighbor not in openset
                    add neighbor to openset

    return failure

function reconstruct_path(came_from, current_node)
    if current_node in came_from
        p := reconstruct_path(came_from, came_from[current_node])
        return (p + current_node)
    else
        return current_node
```

Figure 3.1: A* Algorithm Pseudocode

3.2 A* for path finding in ITESM Parking Lot

The ITESM parking lot has more than 1500 parking lots that are distributed near to common areas like Library, High School, Administrative offices, Conferences Center, etc. In order to represent the parking lot distribution as a conected graph, it's required to separate parking sections based on the location and the nearest buildings areas it's set. As we show in Figure 3.1, here's how each node type is being configured:

- Parking of Lot Start

  List of Destinations with (Name and Estimated Distance h(x))

  Node Type

- Section

  List of Destinations with (Name and Estimated Distance h(x))

  Calculated path cost from start to section g(x)

  Neighbors (Section or Slots)

  Node Type

- Parking Slot

  List of Destinations with (Name and Estimated Distance h(x))

  Calculated path cost from start to section g(x)

  Neighbors (Slots)

  Node Type

- Destinations

  The final visited node(stot) should be conected to the final destination (Example: Conferences Center)

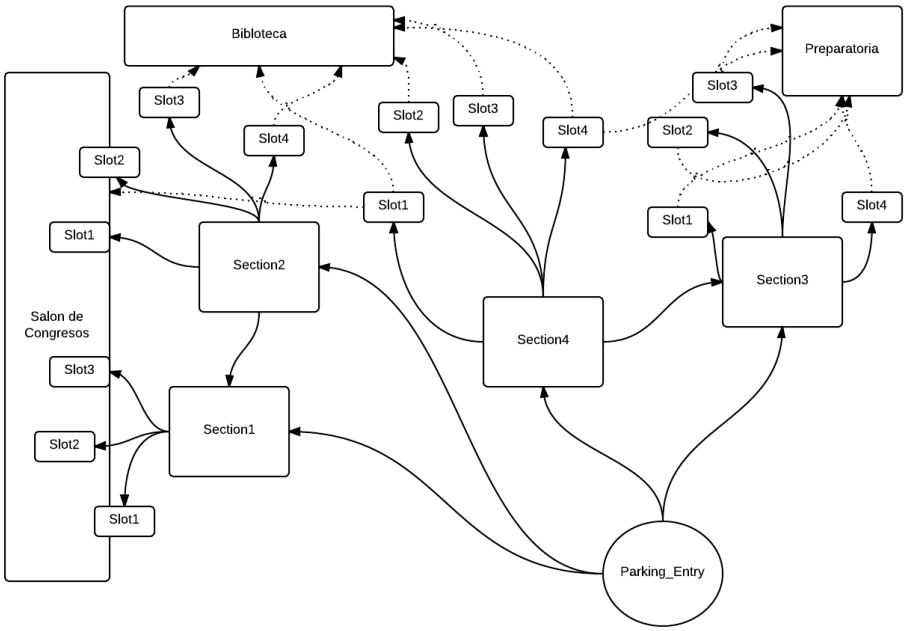

Figure 3.2: Example of Graph used for representing Parking Lot sections/slots conections

Once we have our graph representation we are able to perform slot searches. It's really keep the type of Node when visiting it in the A* Algorithm. The type of Node will be saying if it's an end node or if it cannot be a final node. For example, a Section Node, cannot be the end because it's a forbidden area to park, you need to park in the slot-type nodes.

3.3 Source Code Implementation

3.3.1 Node class

```
class Node():

    def __init__(self, type, h):
        self.type = type
        self.h = h
        self.neighbors = []

    def add_neighbor(self,neighbor):
        neighbor.parent = self
        self.neighbors.append(neighbor)

    def add_section_neighbor(self, neighbor):
        self.neighbors.append(neighbor)
```

Figure 3.3: Node Class Definition

3.3.2 Graph Builder

```
def build_graph(self, filename):
    paths = loader.json_to_dict(filename)

    self.graph = Node("start", paths["destinations"])

    for section in paths["sections"]:
        section_neighbor = Node("section", section["destinations"])
        section_neighbor.name = section["name"]
        section_neighbor.path_from_start = section["path_from_start"]
        self.graph.add_neighbor(section_neighbor)
        for slot in section["slots"]:
            slot_neighbor = Node("slot", slot["destinations"])
            slot_neighbor.id = slot["id"]
            slot_neighbor.path_from_section_start = slot["path_from_section_start"]
            slot_neighbor.status = slot["status"]
            slot_neighbor.type = slot["type"]
            section_neighbor.add_neighbor(slot_neighbor)

        for section in paths["sections"]:
            if section["near_sections"]:
                for current_section in self.graph.neighbors:
                    if current_section.name == section["name"]:
                        current_section.near_sections = section["near_sections"]
                        for near_section in current_section.near_sections:
                            for dest_section in self.graph.neighbors:
                                for key in near_section:
                                    if dest_section.name == key:
                                        current_section.add_section_neighbor(dest_section)
```

Figure 3.4: Graph Builder Function

3.3.3 f(x) = g(x) + h(x) Calculation

```
def get_g(self,node):
    try:
        return node.g
    except AttributeError:
        pass
    if node.type == "start":
        g = 0
    elif node.type == "section":
        g = len(node.path_from_start) * 5
    elif node.type == "slot":
        g = len(node.parent.path_from_start) * 5 + (node.path_from_section_start - 1) * 5
    return g

def get_h(self,node, goal):

    for destination in node.h:
        if destination["name"] == goal:
            return destination["distance"]

def get_f(self,node,goal):
    try:
        return node.f
    except AttributeError:
        pass
    return get_g(node) + get_h(node,goal)
```

Figure 3.5: Functions to calculate f(x) = g(x) + h(h)

3.3.4 A Star Algorithm

```
def a_star(self,start,goal):

    closed_set = []
    open_set = [start]
    came_from = []

    g_score = 0
    f_score = g_score + self.get_h(start,goal)

    while open_set:
          current = self.get_lowest_f(open_set, goal)
          print current.type
          if current.type == "slot":
             return [self.reconstruct_path(current, goal), current]

          open_set.remove(current)
          closed_set.append(current)

          for neighbor in current.neighbors:
              if neighbor in closed_set:
                  continue
              tentative_g_score = self.get_g(current) + self.distance_between(current, neighbor)
              if neighbor not in open_set or tentative_g_score < self.get_g(neighbor):

                  neighbor.came_from = current
                  neighbor.g = tentative_g_score
                  neighbor.f = neighbor.g + self.get_h(neighbor,goal)
                  if neighbor not in open_set:
                      open_set.append(neighbor)
    return None

def get_lowest_f(self, open_set, goal):
```

Figure 3.6: A* Algorithm

3.4 Tech Behind

Chapter 4

Results

Chapter 5

Optimizations and future work

Chapter 6

Conclusion

Chapter 7

Refereces