COMP 4752
Computational Intelligence

Lecture 6

Assignment Tips and

A* Search Improvements

Assignment 1 Marking Scheme

- code style / modularity / readability 10%
- use of decent admissible heuristic 05%
- is_connected all correct for size = 1 15%
- is_connected all correct for size = 2, 3 10%
- get_path costs correct for size = 1 40%
- get_path costs correct for size = 2,3 20%
  - You MUST use the A* algorithm for get_path

A* Graph-Search (GENERAL)

```
1.  Function AStar(problem, h(n))
2.      closed = empty set                  # stores states (tile locations)
3.      open = [(Node(problem.initial_state)]   # stores Nodes
4.      while (true)
5.          if (open.empty) return fail
6.          node = remove_min_f_node(open)
7.          if (node.state is goal) return solution
8.          closed.add(node.state)
9.          for child in Expand(node, problem)
10.             if (child.state in closed) continue
11.             child.f = child.g + h(child)
12.             open.add(child)
```

A* Graph-Search (Improvement)

```
1.  Function AStar(problem, h(n))
2.      closed = empty set                          # stores states (tile locations)
3.      open = [(Node(problem.initial_state)]       # stores Nodes
4.      while (true)
5.          if (open.empty) return fail
6.          node = remove_min_f_node(open)
7.          if (node.state is goal) return solution
8.          closed.add(node.state)
9.          for child in Expand(node, problem)
10.             if (child.state in closed) continue
11.             child.f = child.g + h(child)
12.             if (any node in open with state = child.state and g < child.g) continue
13.             open.add(child)
```

Node Implementation

```
class Node:
    def __init__(self, tile):
        self.state = tile
        self.action = (0, 0)
        self.g, self.f = 0
        self.parent = None
```

Node Implementation

```
class Node:
    def __init__(self, tile):
        self.state = tile
        self.action = (0, 0)
        self.g, self.f = 0
        self.parent = None

    def __lt__(self, other):
        return self.f < other.f
```

General AI Programming Tips

- Algorithm should read like pseudocode wherever possible
- All get / set data functionality should be handled by functions
- Don’t dirty up algorithm with indices or details about data storage / locations
- Be modular wherever possible

A* Search (Python)

```
1.  def a_star_search(self, start, goal, size):
2.          closed = []
3.          open = [Node(start)]
4.          while (len(open) > 0):
5.              node = remove_min_from(open)
6.              if (node.state == goal) return self.reconstruct_path(node)
7.              closed.append(node.state)
8.              for child in self.expand(node):
9.                  if (child.state in closed) continue
10.                 child.f = child.g + self.grid.estimate_cost(start, goal)
11.                 open.append(child)
12.         return []
```

Open List Implementation

- Implement functions in A* class
  - add_to_open(node)
  - remove_min_from_open()
  - is_in_open(node)
  - is_in_closed(state)
- Start out with open / closed as Python list
- Algorithm calls above functions, shouldn’t care about how the data is stored
- Gradually move to more clever Node storage

Open / Closed List Query Time

- “Is a given state on the open / closed list”
- Scanning list = O(n)
- Closed list as set = O(logn) query
- Open list
  - If we sort it, will sort on f values
  - Can’t query a state in logn time
  - How to speed up this query?

Open List Query Time

- If problem is small, use a lookup table
- 2D array [map_width][map_height]
- Array[x][y] = # of (x,y) on the open list
- Constant time queries! O(1)
- Updating the table
  - add_to_open(node) a[x][y] += 1
  - remove_min_from_open() a[x][y] -= 1

A* Graph-Search (Improvement)

```
1.  Function AStar(problem, h(n))
2.      closed = empty set                          # stores states (tile locations)
3.      open = [(Node(problem.initial_state)]       # stores Nodes
4.      while (true)
5.          if (open.empty) return fail
6.          node = remove_min_f_node(open)
7.          if (node.state is goal) return solution
8.          closed.add(node.state)
9.          for child in Expand(node, problem)
10.             if (child.state in closed) continue
11.             child.f = child.g + h(child)
12.             if (any node in open with state = child.state and g < child.g) continue
13.             open.add(child)
```

Informing the Algorithm

- **NOTE: FOR OUR ASSIGNMENT PROBLEM ONLY**
- Think about the case in red

```
if (any node in open with state = child.state
      and g < child.g) continue
open.add(child)
```

- “If there is a node on the search tree that’s on a shorter path through this state, then don’t bother searching this child node’s path”
- Our problem can’t generate a shorter path through a node with a higher g cost

Informing the Algorithm

- What about the opposite case?
- “If the new path through this node is shorter than any other node on the open list, don’t bother searching **those** paths!”
- Instead of putting a new node onto the open list, we can instead **replace** the existing node on the open list with the newly generated one

A* Graph-Search (2D Grid Improve)

```
1.  Function AStar(problem, h(n))
2.      closed = empty set                          # stores states (tile locations)
3.      open = [(Node(problem.initial_state)]       # stores Nodes
4.      while (true)
5.          if (open.empty) return fail
6.          node = remove_min_f_node(open)
7.          if (node.state is goal) return solution
8.          closed.add(node.state)
9.          for child in Expand(node, problem)
10.             if (child.state in closed) continue
11.             child.f = child.g + h(child)
12.             if (any node in open with state = child.state and g <= child.g) continue
13.             if (any node in open with state = child.state and g > child.g)
14.                 update_in_open_list(node_in_open, child)
15.             else open.add(child)
```

A* Example

- Example Removed, see improved version in Lecture 8 slides

Update Open = Huge Savings!

- By updating the node in the open list instead of adding a new one, we now only have max of 1 state on the open list
- We can now pre-allocate all nodes in a 2D grid of size of the map
- all_nodes[x][y] = Node((x,y))
- Max nodes generated = x*y
- Open list now stores pointers to nodes

Constant Time Node Lookup

- Want to know if a state is in a node in the open list to look up its g cost
- Previously had to scan the list for node
- We can now simply look up in array
  - all_nodes[x][y]
- IMPORTANT:
  - Must re-sort open list upon updating a node