COMP3702 Tutorial 3

Matt Choy | matthew.choy@uq.edu.au

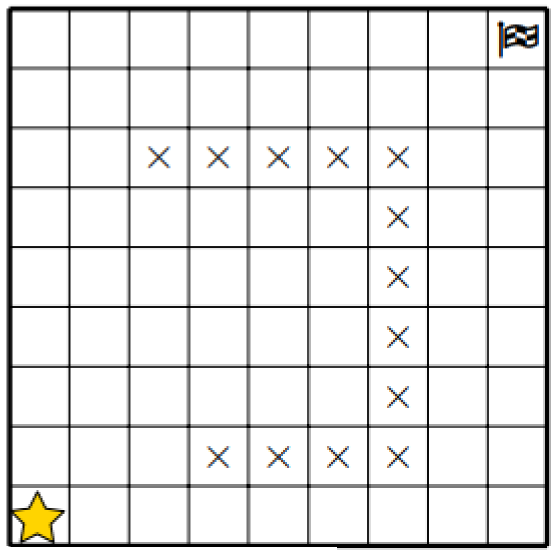
Breadth First Search

- In BFS, the successors of a node get added to a First-In, First Out (FIFO) queue
- This can be implemented in Python using container.append(next_node) and container.pop()

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(frontier) then return failure
     node ← POP(frontier)  /* chooses the shallowest node in frontier */
     add node.STATE to explored
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
           if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
           frontier ← INSERT(child, frontier)
```

Figure 3.11 Breadth-first search on a graph.

Exercise 3.1



Exercise 3.1. Consider the path planning problem on a 9×9 grid world in the figure below. The goal is to move from the star to the flag in as few steps as possible. Crosses indicate obstacles, ~~and attempting to traverse either an obstacle or a boundary wall is an invalid move, and does not move your agent (it should have no cost or effect).~~

a) Develop a state graph representation for this search problem, and develop a `step()` method for finding the next legal steps this problem, i.e. for generating successor nodes (vertices).

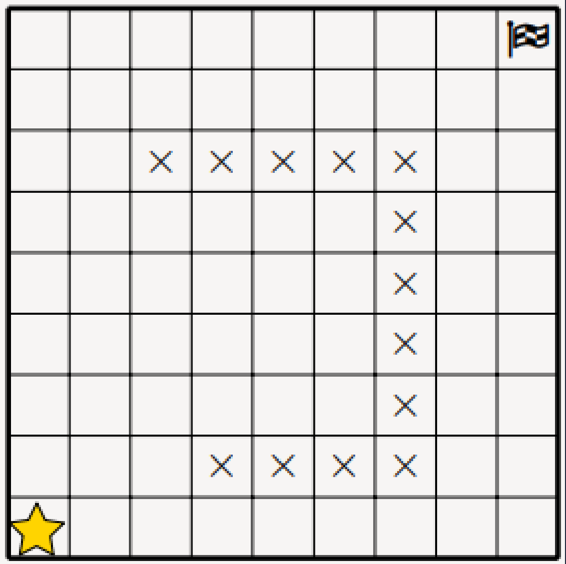
b) Implement BFS for this problem (using a *FIFO queue*) using your `step()` function.

c) Implement iterative-deepening DFS (IDDFS) for this problem using a length-limited *LIFO queue*, and reusing `step()`.

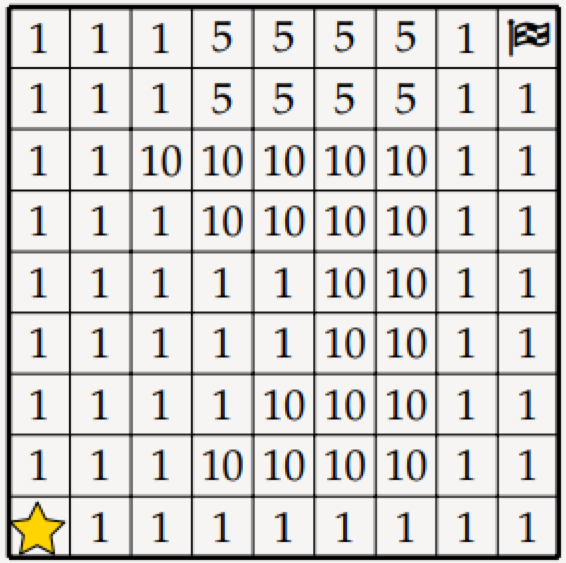
d) Compare the performance of BFS and IDDFS in terms of (i) the number of nodes generated, (ii) the number of nodes on the fringe when the search terminates (iii) the number of nodes on the explored set (if there is one) when the search terminates, and (iv) the run time of the algorithm (e.g. in units such as mins:secs). Discuss your findings.

Gridworld Environment

- Consider the path planning problem on a 9 x 9 grid world in the figure below. The goal is to move from the start to the flag in as few steps as possible.
- In the uniform cost problem, crosses indicate obstacles.



Uniform cost



Costly moves

Tactics for Exercise 3.1 and 3.2

- Skim through the problem, and try to map what is required for designing (modelling) an agent, and the environment interaction:
  - What is the agent?
  - What is the environment?
  - What is the agent’s goal?
  - What are its costs?
- Recall by *model*, we mean the state and action spaces as well as the transition function).
  - Try to design the environment module in such a way that you can re-use it – this will be helpful in later problems.
  - How do you want to structure the interaction with the agent, and use this to define different classes and their methods.

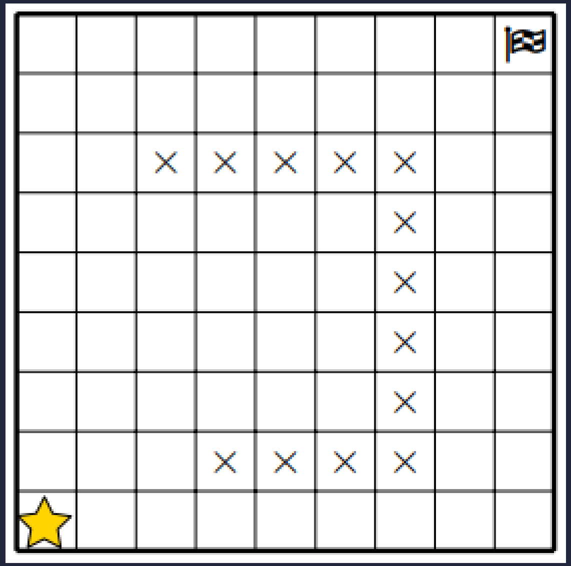
Environment Module

- Two reasonable classes that we could implement might be a GridWorld class, and a GridWorldState class.
- GridWorld should provide:
  - A parser for the concrete problem you are trying to solve
  - A step function that encodes the dynamics of the environment and
  - A function that returns a list of neighbouring states and the transition costs to move between them.
  - You may also want to have a method to heuristics here, although that could also be placed inside your search agent module.
- The GridWorldState class can provide a place to put information and methods relevant to each specific state, e.g. its parents in the search tree.

Search Agent Module

- Recall the three important parts of the search agent:
  - Data structure (container, e.g. a queue or stack)
  - Goal state checker
  - A way to obtain the neighbours (successors) of the currently explored (expanded) state
- You can build separate code blocks for each section, or use inheritance to develop one module with an abstract search agent class (template) that is re-used for each concrete algorithm (implementation).

Exercise 3.1



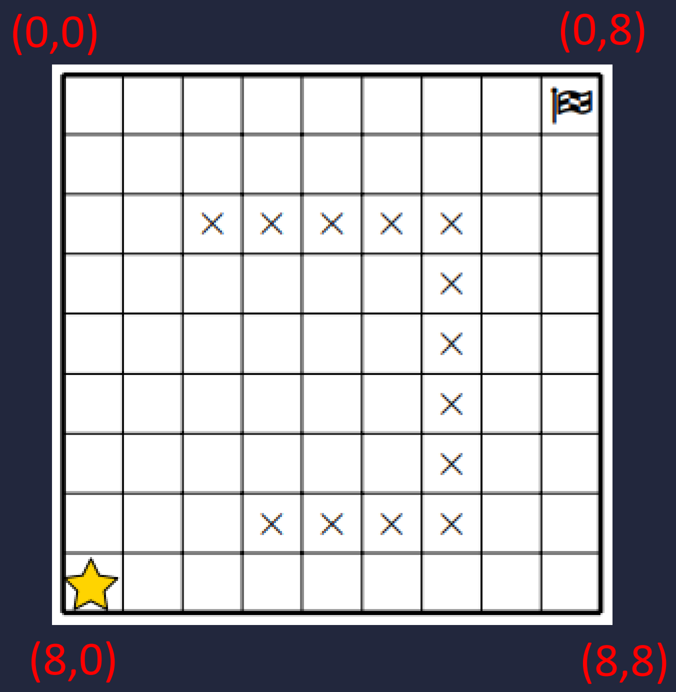
Template code
https://gist.github.com/MattPChoy/763da403f3dcf31c9c2d8f18a19d8a83

1. Create a class that represents the current state of the GridWorld
2. Implement Breadth-First Search (you may take inspiration from last week's tutorial solutions)
3. Make modifications to a Depth-First Search implementation to implement Iteration-Deepening Depth First Search (ID DFS)
4. Compare the performance difference of BFS and ID DFS

Exercise 3.1 - Solution

State Representation – Agent Design

- Agent Design Components:
  - State Representation: S = {(row, col) | 0 <= row, col <= 8}
  - Actions = {U, D, L, R}
  - World Dynamics: Move one square in the selected direction (unless the next square is a boundary or an obstacle)
  - Utility:
    - 1 if (row, col) == goal state → (row, col) == (0, 8)
    - 0 otherwise



State Graph Representation

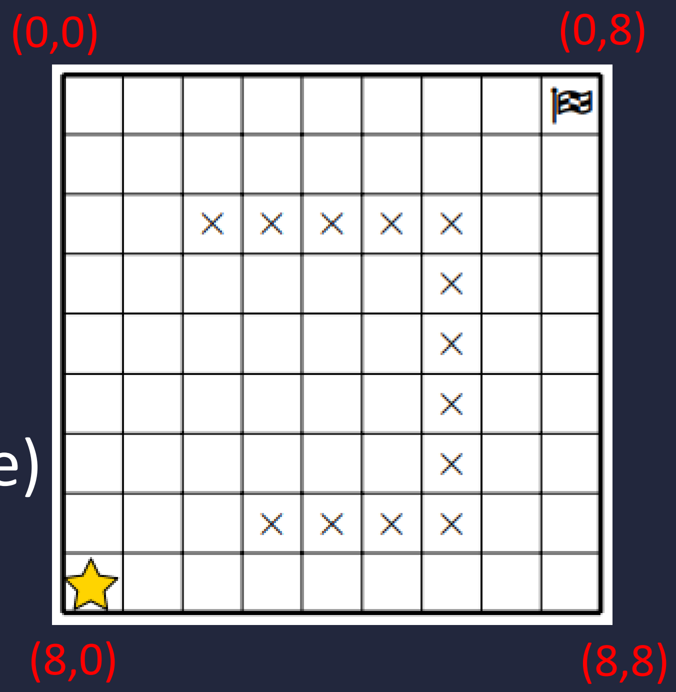
```python
class GridWorld():
    ACTIONS = ['U', 'D', 'L', 'R']

    def __init__(self):
        self.n_rows = 9
        self.n_cols = 9

        # (row, column) where indexing is top to bottom,
        # left to right (like a matrix)
        start = (8, 0) #bottom, left
        goal  = (0, 8)

        self.obstacles = [[0, 0, 0, 0, 0, 0, 0, 0, 0],
                          [0, 0, 0, 0, 0, 0, 0, 0, 0],
                          [0, 0, 1, 1, 1, 1, 1, 0, 0],
                          [0, 0, 0, 0, 0, 0, 1, 0, 0],
                          [0, 0, 0, 0, 0, 0, 1, 0, 0],
                          [0, 0, 0, 0, 0, 0, 1, 0, 0],
                          [0, 0, 0, 0, 0, 0, 1, 0, 0],
                          [0, 0, 0, 1, 1, 1, 1, 0, 0],
                          [0, 0, 0, 0, 0, 0, 0, 0, 0]]
```

```python
self.costs = [[1, 1,  1,  5,  5,  5,  5, 1, 1],
              [1, 1,  1,  5,  5,  5,  5, 1, 1],
              [1, 1, 10, 10, 10, 10, 10, 1, 1],
              [1, 1,  1, 10, 10, 10, 10, 1, 1],
              [1, 1,  1,  1,  1, 10, 10, 1, 1],
              [1, 1,  1,  1,  1, 10, 10, 1, 1],
              [1, 1,  1,  1, 10, 10, 10, 1, 1],
              [1, 1,  1, 10, 10, 10, 10, 1, 1],
              [1, 1,  1,  1,  1,  1,  1, 1, 1]]
```

State Graph Representation – Step Method

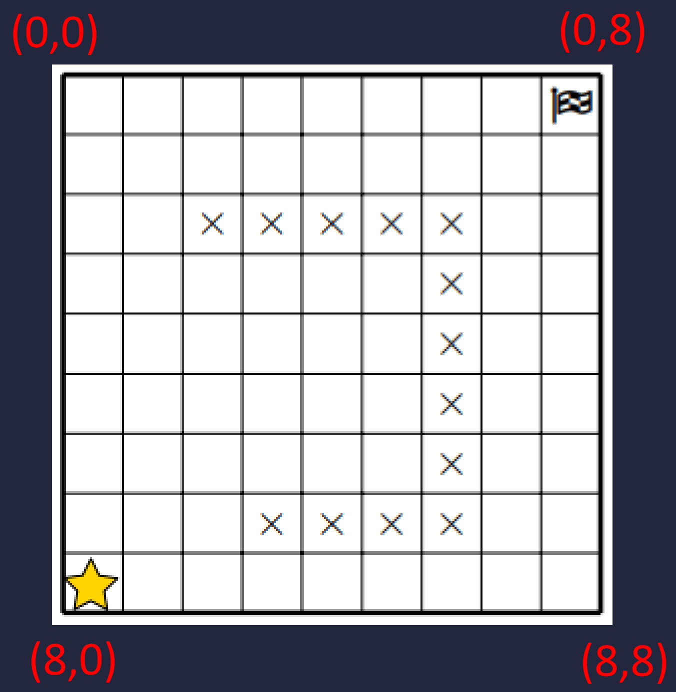
- Given a current state (here, represented by a 2-tuple – (row, col) and an action (L, R, U, D) how do we determine whether an action is valid or not?
- We need to come up with a function that tells us whether this is valid or not – this is the purpose of the step() function.
- The step method has:
  - Inputs: (state, action)
  - Outputs: [success / collision, next state, action cost]

1. Compute the new position given current state & action
2. Test for collision (with boundary or obstacles)
3. Compute cost of performing action (in costly moves case)



State Graph Representation – Step Method

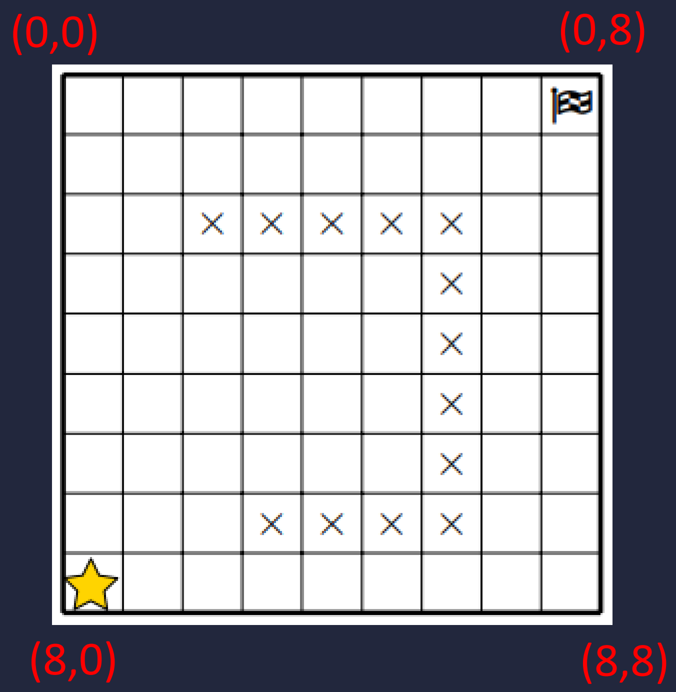
```python
def step(self, state, action):
    """
    :param state: (row, col) tuple
    :param action: 'U', 'D', 'L' or 'R'
    :return: (success [True/False], new state, action cost)
    """
    r, c = state

    if action == 'U':
        new_r = r - 1
        new_c = c
    elif action == 'D':
        new_r = r + 1
        new_c = c
    elif action == 'L':
        new_r = r
        new_c = c - 1
    elif action == 'R':
        new_r = r
        new_c = c + 1
    else:
        assert False, '!!! invalid action !!!'

    if (not (0 <= new_r < 9)) or (not (0 <= new_c < 9)) or self.obstacles[new_r][new_c] == 1:
        # collision occurs
        return False, (r, c), self.costs[r][c]
    else:
        return True, (new_r, new_c), self.costs[new_r][new_c]
```



State Graph Representation – Helper Functions

- is_goal(state) -> Boolean
  - Here, we use tuple equality to check if (row, col) == (goal_row, goal_col)
  - Allows a modularity for problems with more complex goal conditions.

```
def is_goal(self, state):
    """
    :param state: (row, col) tuple
    :return: True/False
    """
    return state == self.goal_state

def get_state_cost(self, state):
    r, c = state
    return self.costs[r][c]
```



Exercise 3.1b - BFS

BFS - Search Node

```python
class StateNode:
    def __init__(self, env, state, actions, path_cost):
        self.env = env
        self.state = state
        self.actions = actions
        self.path_cost = path_cost

    def get_successors(self):
        successors = []
        for a in GridWorldEnv.ACTIONS:
            success, new_state, a_cost = self.env.step(self.state, a)
            if success:
                successors.append(StateNode(self.env,
                                            new_state,
                                            self.actions + [a],
                                            self.path_cost + self.env.get_state_cost(new_state)))
        return successors
```

BFS - Algorithm

```
def bfs(env, verbose=True):
    container = [StateNode(env, env.init_state, [], 0)]
    visited = set()

    n_expanded = 0
    while len(container) > 0:
        # expand node
        node = container.pop(0)

        # test for goal
        if env.is_goal(node.state):
            if verbose:
                print(f'Visited Nodes: {len(visited)},\t\tExpanded Nodes: {n_expanded},\t\t'
                      f'Nodes in Container: {len(container)}')
                print(f'Cost of Path (with Costly Moves): {node.path_cost}')
            return node.actions

        # add successors
        successors = node.get_successors()
        for s in successors:
            if s.state not in visited:
                container.append(s)
                visited.add(s.state)
        n_expanded += 1

    return None
```

Exercise 3.1c - IDDFS

IDDFS Concept

- DFS traverses nodes going through successors of roots
  - O(d) space (where d is the depth of the tree)
- BFS traverses level by level
  - O(n) space (where n is the number of nodes in the tree)
- IDDFS combines DFS's space efficiency with BFS's time efficiency.
  - Calls DFS for increasing maximum depths
  - Basically call a **depth-limited DFS search** is a breadth-first search style fashion

IDDFS

- An implementation of depth-limited DFS is required as a component of IDDFS
  - Has an additional input, max_depth
  - If successor has len(actions) > max_depth, do not add it to the container
  - Successors should be added to the container if they are not visited, or if visited at a cost higher than that of thec urrent path
  - This is needed for the optimality condition (same applies for UCS and A*)

DL DFS - Algorithm

```
def depth_limited_dfs(env, max_depth, verbose=True):
    container = [StateNode(env, env.init_state, [], 0)]
    # revisiting should be allowed if cost (depth) is lower than previous visit (needed for optimality)
    visited = {}    # dict mapping states to path cost (here equal to depth)
    n_expanded = 0
    while len(container) > 0:
        # expand node
        node = container.pop(-1)

        if env.is_goal(node.state):        # test for goal
            if verbose:
                print(f'Visited Nodes: {len(visited.keys())},\t\tExpanded Nodes: {n_expanded},\t\t'
                      f'Nodes in Container: {len(container)}')
                print(f'Cost of Path (with Costly Moves): {node.path_cost}')
            return node.actions

        successors = node.get_successors()         # add successors
        for s in successors:
            if (s.state not in visited or len(s.actions) < visited[s.state]) and len(s.actions) < max_depth:
                container.append(s)
                visited[s.state] = len(s.actions)
        n_expanded += 1

    return None
```

ID DFS - Algorithm

- Call Depth-Limited DFS with incrementing depth until a solution is found
- To allow termination when no solution exists, set an arbitrary maximum depth (e.g. n = 1000)
- Alternatively, DL DFS return status to indicate whether unvisited successors exist at greater depths.

```
def iddfs(env, verbose=True):
    depth_limit = 1
    while depth_limit < 1000:
        actions = depth_limited_dfs(env, depth_limit, verbose)
        if actions is not None:
            return actions
        depth_limit += 1
    return None
```

Exercise 3.1d - Performance

Comparison - Code

```python
n_trials = 100
print('== Exercise 3.1 ========================================================================')
gridworld = GridWorldEnv()

print('BFS:')
t0 = time.time()
for i in range(n_trials):
    actions_bfs = bfs(gridworld, verbose=(i == 0))
t_bfs = (time.time() - t0) / n_trials
print(f'Num Actions: {len(actions_bfs)},\t\tActions: {actions_bfs}')
print(f'Time: {t_bfs}')
print('\n')

print('IDDFS:')
t0 = time.time()
for i in range(n_trials):
    actions_iddfs = iddfs(gridworld, verbose=(i == 0))
t_iddfs = (time.time() - t0) / n_trials
print(f'Num Actions: {len(actions_iddfs)},\t\tActions: {actions_iddfs}')
print(f'Time: {t_iddfs}')
print('\n')
```

Performance

BFS:

Visited Nodes: 68, Expanded Nodes: 68, Nodes in Container: 0

Cost of Path (with Costly Moves): 32

Num Actions: 16, Actions: ['U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'R', 'R', 'R', 'R', 'R', 'R', 'R', 'R']

Time: 0.00031942367553710937

IDDFS:

Visited Nodes: 27, Expanded Nodes: 25, Nodes in Container: 5

Cost of Path (with Costly Moves): 16

Num Actions: 16, Actions: ['R', 'R', 'R', 'R', 'R', 'R', 'R', 'R', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U']

Time: 0.005034787654876709

Exercise 3.2

1	1	1	5	5	5	5	1	
1	1	1	5	5	5	5	1	1
1	1	10	10	10	10	10	1	1
1	1	1	10	10	10	10	1	1
1	1	1	1	1	10	10	1	1
1	1	1	1	1	10	10	1	1
1	1	1	1	10	10	10	1	1
1	1	1	10	10	10	10	1	1
	1	1	1	1	1	1	1	1

Exercise 3.2. Now consider the path planning problem **with costly moves** on a 9×9 grid world in the figure below, where costs of arriving at a state are indicated on the board and the goal is to move from the star to the flag:

a) Run BFS for this problem, reusing your answer from Exercise 3.1 (nb. it should not use the costs on the grid).

b) Implement UCS for this problem using a *priority queue*.

c) Compare the performance of BFS and UCS in terms of (i) the number of nodes generated, (ii) the number of nodes on the fringe when the search terminates (iii) the cost of the solution path. Discuss how and why these results differ from those for Exercise 3.1.

d) Now derive an *admissible* heuristic for this path planning problem.

e) Using your heuristic, implement A* search for solving this path planning problem.

f) Compare the performance of UCS and A* search in terms of (i) the number of nodes generated, (ii) the number of nodes on the fringe when the search terminates (iii) the cost of the solution path. Explain these results.

Costly Paths

- We are now considering paths with a cost – the “shortest” path now may not be a straight line from A to B (avoiding obstacles)
- We need to use search algorithms that take this into account such as UCS and A*.
- It is interesting to note that when the costs are all the same, UCS returns the same result as BFS.

Exercise 3.2 - Solution

1	1	1	5	5	5	5	1	
1	1	1	5	5	5	5	1	1
1	1	10	10	10	10	10	1	1
1	1	1	10	10	10	10	1	1
1	1	1	1	1	10	10	1	1
1	1	1	1	1	10	10	1	1
1	1	1	1	10	10	10	1	1
1	1	1	10	10	10	10	1	1
	1	1	1	1	1	1	1	1

Exercise 3.2. Now consider the path planning problem **with costly moves** on a 9×9 grid world in the figure below, where costs of arriving at a state are indicated on the board and the goal is to move from the star to the flag:

a) Run BFS for this problem, reusing your answer from Exercise 3.1 (nb. it should not use the costs on the grid).

b) Implement UCS for this problem using a *priority queue*.

c) Compare the performance of BFS and UCS in terms of (i) the number of nodes generated, (ii) the number of nodes on the fringe when the search terminates (iii) the cost of the solution path. Discuss how and why these results differ from those for Exercise 3.1.

d) Now derive an *admissible* heuristic for this path planning problem.

e) Using your heuristic, implement A* search for solving this path planning problem.

f) Compare the performance of UCS and A* search in terms of (i) the number of nodes generated, (ii) the number of nodes on the fringe when the search terminates (iii) the cost of the solution path. Explain these results.

Exercise 3.2a – BFS

BFS:
Visited Nodes: 68, Expanded Nodes: 68, Nodes in Container: 0
Cost of Path (with Costly Moves): 32
Num Actions: 16, Actions: ['U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'R', 'R', 'R', 'R', 'R', 'R', 'R', 'R']
Time: 0.00031942367553710937

Exercise 3.2b – UCS

UCS

- We are now considering paths with a cost – the "shortest" path now may not be a straight line from A to B (avoiding obstacles)
- We need to use search algorithms that take this into account such as UCS and A*.
- It is interesting to note that when the costs are all the same, UCS returns the same result as BFS.

UCS

- Sometimes there are costs associated with edges
- The cost of a path is the sum of the costs of its edges

$$cost(n_0, \cdots, n_k) = \sum_{i=1}^{k} cost(n_{i-1}, n_i)$$

- At each stage, uniform-cost search selects a path on the frontier with the lowest cost.
  - The first path to a goal is a least-cost path to a goal node
- UCS treats the frontier as a priority queue, ordered by path cost.
  - It always selects the node of the highest priority added to the frontier.
- If the list of paths on the frontier (PQ) is $[p_1, p_2, \cdots]$ then:
  - p_1 is selected to be expanded
  - Its successors are added into the PQ
  - The highest-priority vertex is selected (and it might be a newly-expanded vertex).

UCS

```python
import queue
q = queue.PriorityQueue()
```

- We can use the inbuilt priority queue so we don't have to implement one ourselves!
- UCS generates the optimal solution (guarantees that it can find a path of lowest cost) if **all of the edges have positive cost**
- UCS has both time and space complexity of $O\left(b^{1+\left\lfloor\frac{C^*}{c}\right\rfloor}\right)$
  - b: Branching Factor
  - C* Cost of optimal solution
  - c: cost
  - ϵ: Minimum cost of step (smallest cost for a single step)

Exercise 3.2c – BFS vs UCS

Exercise 3.2c – BFS vs UCS

BFS:
Visited Nodes: 68, Expanded Nodes: 68, Nodes in Container: 0
Cost of Path (with Costly Moves): 32
Num Actions: 16, Actions: ['U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'R', 'R', 'R', 'R', 'R', 'R', 'R', 'R']
Time: 0.00031915187835693357

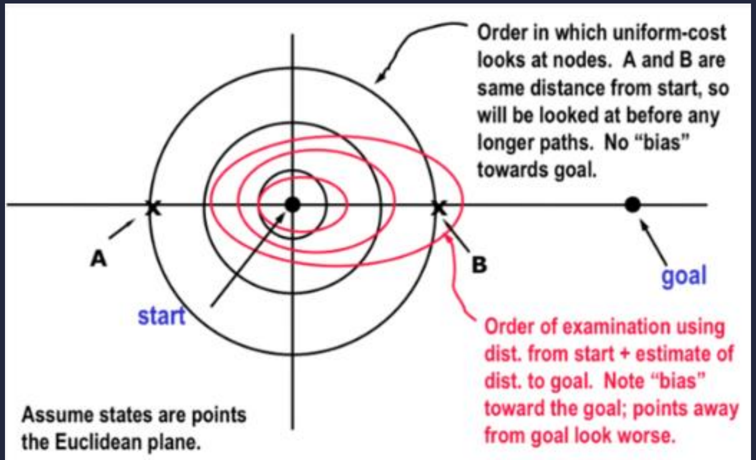
UCS:
Visited Nodes: 65, Expanded Nodes: 56, Nodes in Container: 9
Cost of Path (with Costly Moves): 16
Num Actions: 16, Actions: ['R', 'R', 'R', 'R', 'R', 'R', 'R', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'R']

Comparison - Code

```python
print('== Exercise 3.2 ===========================================================================')
print('UCS:')
t0 = time.time()
for i in range(n_trials):
    actions_ucs = ucs(gridworld, verbose=(i == 0))
t_ucs = (time.time() - t0) / n_trials
print(f'Num Actions: {len(actions_ucs)},\t\tActions: {actions_ucs}')
print(f'Time: {t_ucs}')
print('\n')

print('A*:')
t0 = time.time()
for i in range(n_trials):
    actions_a_star = a_star(gridworld, manhattan_dist_heuristic, verbose=(i == 0))
t_a_star = (time.time() - t0) / n_trials
print(f'Num Actions: {len(actions_a_star)},\t\tActions: {actions_a_star}')
print(f'Time: {t_a_star}')
print('\n')
```

Exercise 3.2d – Admissible Heuristic

Heuristics



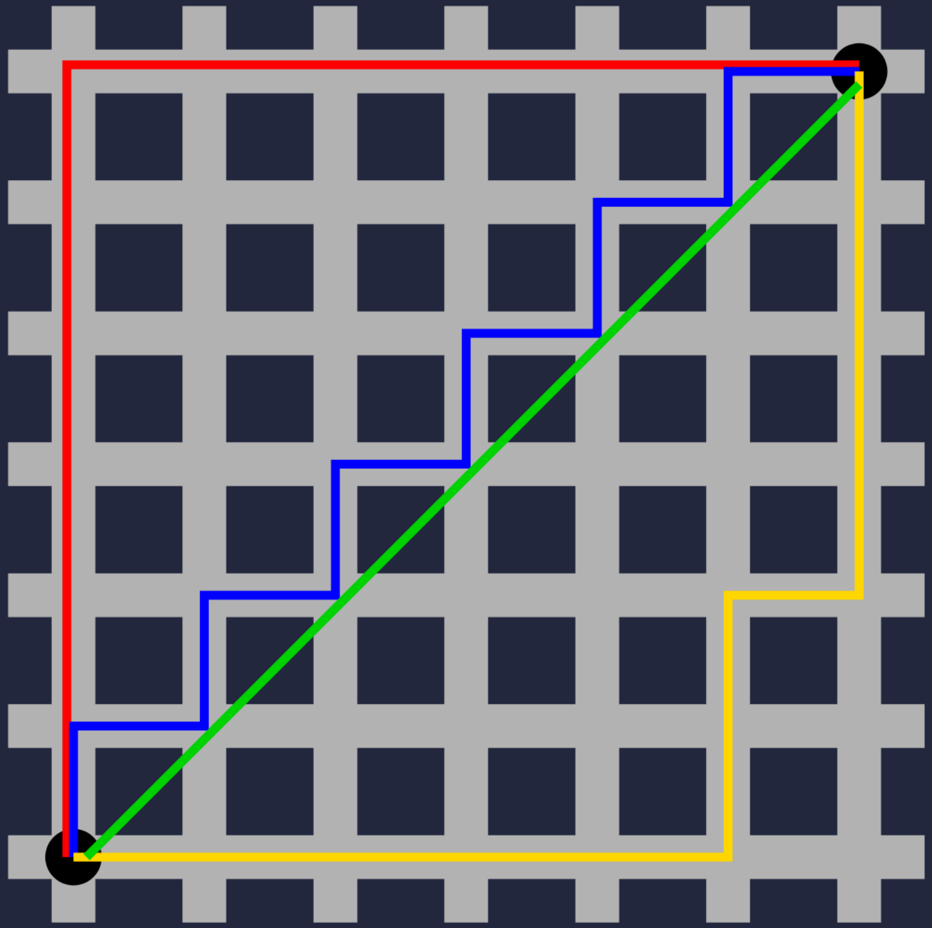
- Heuristics are a way for us to guide our search algorithms so that they can work more efficiently
  - A way of providing the search algorithm more information to work with.
- h(n) is an estimate of the cost of the shortest path from some node \`n\` to the goal node.
- Whilst h(n) doesn't need to be a perfect heuristic, it must be quick to compute!
  - Otherwise, we might not save any time with respect to uninformed search algorithms, if the heuristic is inefficient to compute
- h(n) is an underestimate if there is no path from n to the goal node with a cost less than h(n) (i.e. it underestimates the cost of the path to the goal).
- An admissible heuristic is non-negative (>=0) heuristic function that does not overestimate the actual cost of a path to the goal.

Exercise 3.2d – Solution

3.2d – Admissible Heuristics

- Simplest Grid-World Heuristic
- Given points (x1, y1) and (x2, y2) the distance is computed using the following formula:

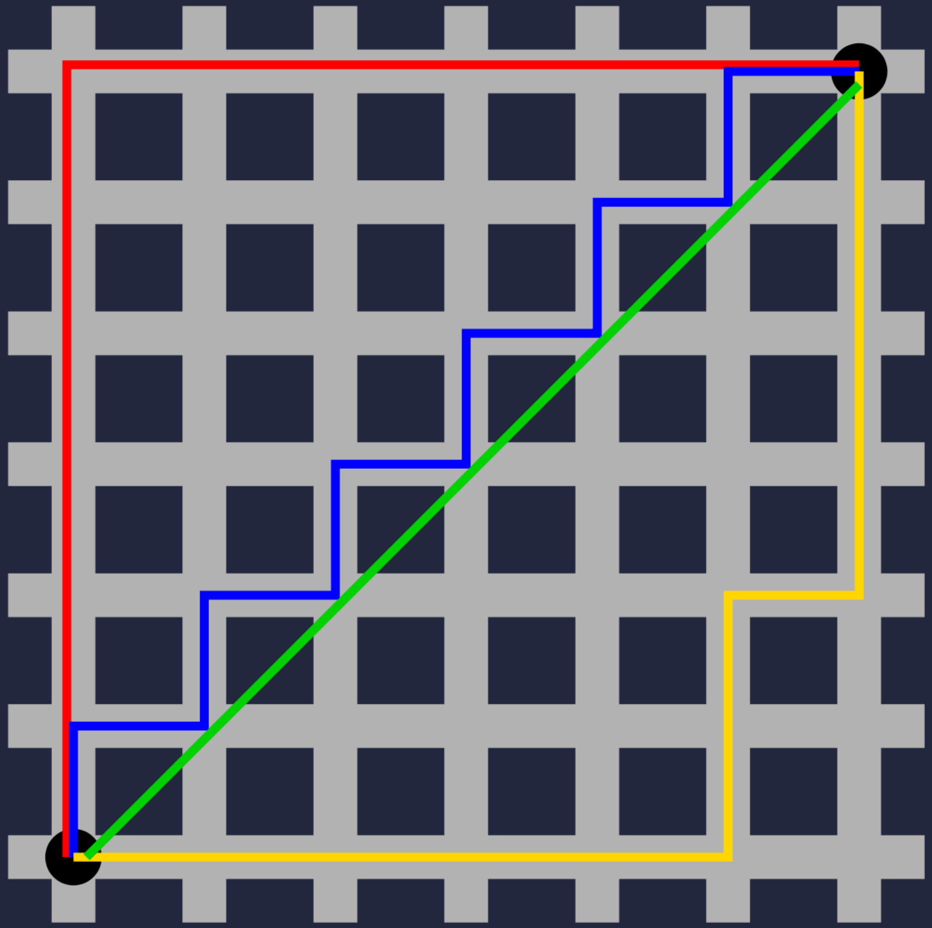
$$dist\big((x_1, y_1), (x_2, y_2)\big) = |x_2 - x_1| + |y_2 - y_1|$$



3.2d – Admissible Heuristics

- Simplest Grid-World Heuristic
- Given points (x1, y1) and (x2, y2) the distance is computed using the following formula:

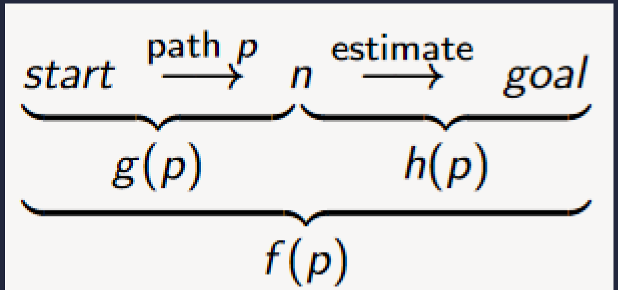
$$dist\big((x_1, y_1), (x_2, y_2)\big) = |x_2 - x_1| + |y_2 - y_1|$$



```
def manhattan_dist_heuristic(env, state):
    return abs(env.goal_state[0] - state[0]) + abs(env.goal_state[1] - state[1])
```

Exercise 3.2e – A* Algorithm

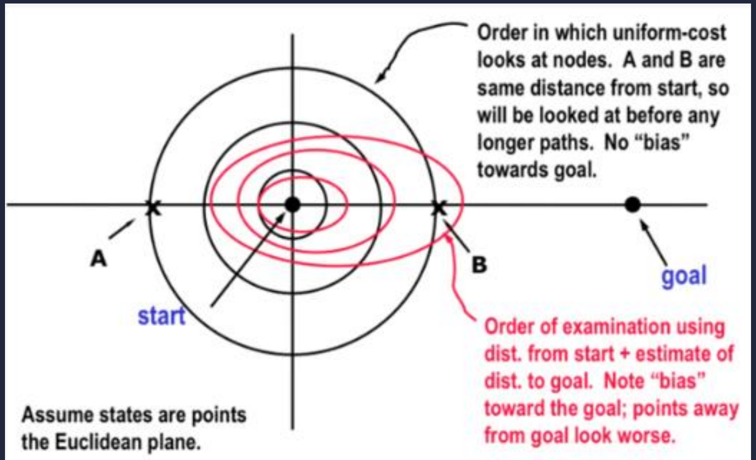
A* Search Algorithm



- We now want to use 'smarter' search algorithms.
- Often, there is extra knowledge that can be used to guide our search algorithm – these are called 'heuristics', denoted h(n)

- A* search uses both path cost and heuristic values
- It is as mix of uniform-cost and best-first search
- Given that g(p) is the cost of path p from initial state to a node, and h(p) estimates the cost from the end of the path p to a goal node
- A* uses the formula f(p) = g(p) + h(p) – note that this is still an estimate.
- The search algorithm treats the frontier as a priority queue, ordered by f(p), with the highest priority node with the lowest f(p) value (lowest estimated distance)

Heuristics



- Heuristics are a way for us to guide our search algorithms so that they can work more efficiently
  - A way of providing the search algorithm more information to work with.
- h(n) is an estimate of the cost of the shortest path from some node \`n\` to the goal node.
- Whilst h(n) doesn't need to be a perfect heuristic, it must be quick to compute!
  - Otherwise, we might not save any time with respect to uninformed search algorithms, if the heuristic is inefficient to compute
- h(n) is an underestimate if there is no path from n to the goal node with a cost less than h(n) (i.e. it underestimates the cost of the path to the goal).
- An admissible heuristic is non-negative (>=0) heuristic function that does not overestimate the actual cost of a path to the goal.

Exercise 3.2e – Solution

3.2e – A* Solution

```
def a_star(env, heuristic, verbose=True):
    container = [(0 + heuristic(env, env.init_state), StateNode(env, env.init_state, [], 0))]
    heapq.heapify(container)     # dict: state --> path_cost
    visited = {env.init_state: 0}
    n_expanded = 0
    while len(container) > 0:
        n_expanded += 1
        _, node = heapq.heappop(container)
        if env.is_goal(node.state):          # check if this state is the goal
            if verbose:
                print(f'Visited Nodes: {len(visited.keys())},\t\tExpanded Nodes: {n_expanded},\t\t'
                      f'Nodes in Container: {len(container)}')
                print(f'Cost of Path (with Costly Moves): {node.path_cost}')
            return node.actions

        # add unvisited (or visited at higher path cost) successors to container
        successors = node.get_successors()
        for s in successors:
            if s.state not in visited.keys() or s.path_cost < visited[s.state]:
                visited[s.state] = s.path_cost
                heapq.heappush(container, (s.path_cost + heuristic(env, s.state), s))
    return None
```

Exercise 3.2f – UCS vs A*

Exercise 3.2f – UCS vs A*

BFS:
Visited Nodes: 68, Expanded Nodes: 68, Nodes in Container: 0
Cost of Path (with Costly Moves): 32
Num Actions: 16, Actions: ['U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'R', 'R', 'R', 'R', 'R', 'R', 'R', 'R']
Time: 0.00031915187835693357

UCS:
Visited Nodes: 65, Expanded Nodes: 56, Nodes in Container: 9
Cost of Path (with Costly Moves): 16
Num Actions: 16, Actions: ['R', 'R', 'R', 'R', 'R', 'R', 'R', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'R']
Time: 0.0003191423416137695

A*:
Visited Nodes: 27, Expanded Nodes: 17, Nodes in Container: 10
Cost of Path (with Costly Moves): 16
Num Actions: 16, Actions: ['R', 'R', 'R', 'R', 'R', 'R', 'R', 'R', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U']

Exercise 3.3 – A* Search in 8-Puzzle

Exercise 3.3. Implement A* for solving the 8-puzzle problem, assuming the goal of the 8-puzzle agent is to find the solution in as few steps as possible.

a) Discuss the heuristics you could use with your classmates.

b) Reuse the container in the program you created in the last tutorial, by modifying it to a priority queue. Along with this, revise the program, so as to use the cost from initial to the current node and the heuristics to identify which node to expand next.

c) Implement the heuristic as part of your A* search algorithm.

3.3 - A* Search in 8-Puzzle

- Re-use the 8-Puzzle code that you wrote last week, and using the same tactics above implement A*
- To make the 8-puzzle environment from the previous tutorial compatible with the env.step(…) paradigm, we need to have:
- Env class:
  - Stores initial state and goal state
  - Implements step(…), is_goal(…), get_state_cost(…)
- State class
  - Stores current position of each tile
  - Implements get_successors()
  - Supports less than operator for priority queue use (implement __lt__(…) method)

3.3 - A* Search in 8-Puzzle

```python
class EightPuzzleEnv:
    ACTIONS = ['U', 'D', 'L', 'R']

    def __init__(self, init, goal):
        self.init_state = EightPuzzleState(init)
        self.goal_state = EightPuzzleState(goal)

    @staticmethod
    def move_left(state):
        new_squares = state.squares[:]
        new_squares[state.idx] = state.squares[state.idx - 1]
        new_squares[state.idx - 1] = state.squares[state.idx]
        return EightPuzzleState(new_squares)

    @staticmethod
    def move_right(state):
        new_squares = state.squares[:]
        new_squares[state.idx] = state.squares[state.idx + 1]
        new_squares[state.idx + 1] = state.squares[state.idx]
        return EightPuzzleState(new_squares)

    @staticmethod
    def move_up(state):
        new_squares = state.squares[:]
        new_squares[state.idx] = state.squares[state.idx - 3]
        new_squares[state.idx - 3] = state.squares[state.idx]
        return EightPuzzleState(new_squares)

    @staticmethod
    def move_down(state):
        new_squares = state.squares[:]
        new_squares[state.idx] = state.squares[state.idx + 3]
        new_squares[state.idx + 3] = state.squares[state.idx]
        return EightPuzzleState(new_squares)
```

3.3 - A* Search in 8-Puzzle

```
def step(self, state, action):
    """
    :param state: EightPuzzle state
    :param action: 'U', 'D', 'L' or 'R'
    :return: (success [True/False], new state, action cost)
    """
    if action == 'U' and (state.idx // 3) > 0:
        return True, self.move_up(state), 1
    elif action == 'D' and (state.idx // 3) < 2:
        return True, self.move_down(state), 1
    elif action == 'L' and (state.idx % 3) > 0:
        return True, self.move_left(state), 1
    elif action == 'R' and (state.idx % 3) < 2:
        return True, self.move_right(state), 1
    else:
        return False, EightPuzzleState(state.squares), 1
```

```
def is_goal(self, state):
    """
    :param state: EightPuzzleState
    :return: True/False
    """
    return state == self.goal_state

def get_state_cost(self, state):
    # same cost for all states in
EightPuzzle
    return 1
```

3.3 - A* Search in 8-Puzzle

```
class StateNode:
    def __init__(self, env, state, actions, path_cost):
        self.env = env
        self.state = state
        self.actions = actions
        self.path_cost = path_cost

    def get_successors(self):
        successors = []
        for a in GridWorldEnv.ACTIONS:
            success, new_state, a_cost = self.env.step(self.state, a)
            if success:
                successors.append(StateNode(self.env,
                                            new_state,
                                            self.actions + [a],
                                            self.path_cost + self.env.get_state_cost(new_state)))
        return successors

    def __lt__(self, other):
        # we won't use this as a priority directly, so result doesn't matter
        return True
```

3.3 - A* Search in 8-Puzzle Solutions

- Re-use the 8-Puzzle code that you wrote last week, and using the same tactics above implement A*
- Possible heuristics include:
  - Hamming distance (i.e. the number of tiles that are different, between the current state and the goal state)
  - Manhattan distance of each tile to its goal position
  - The number of inversions (look at last week's tutorial sheet for explanation on inversions)
- All of these heuristics are admissible, but vary in how efficiently they can find the solution.

3.3 - A* Search in 8-Puzzle Solutions

```
print('== Exercise 3.3 ==============================================================================')
puzzle = EightPuzzleEnv('281463_75', '1238_4765')

print('BFS:')
t0 = time.time()
for i in range(n_trials):
    actions_bfs = bfs(puzzle, verbose=(i == 0))
t_bfs = (time.time() - t0) / n_trials
print(f'Num Actions: {len(actions_bfs)},\t\tActions: {actions_bfs}')
print(f'Time: {t_bfs}')
print('\n')

print('A* (num mismatches):')
t0 = time.time()
for i in range(n_trials):
    actions_a_star = a_star(puzzle, num_mismatches_heuristic, verbose=(i == 0))
t_a_star = (time.time() - t0) / n_trials
print(f'Num Actions: {len(actions_a_star)},\t\tActions: {actions_a_star}')
print(f'Time: {t_a_star}')
print('\n')

print('A* (summed manhattan):')
t0 = time.time()
for i in range(n_trials):
    actions_a_star = a_star(puzzle, summed_manhattan_heuristic, verbose=(i == 0))
t_a_star = (time.time() - t0) / n_trials
print(f'Num Actions: {len(actions_a_star)},\t\tActions: {actions_a_star}')
print(f'Time: {t_a_star}')
print('\n')
```

Exercise 3.4 – Admissible Heuristics

Exercise 3.4. Let h_1 be an *admissible* heuristic, where the lowest value is 0.1 and the highest value is 2.0. Suppose that for any state s:

- $h_2(s) = h_1(s) + 5$,
- $h_3(s) = 2h_1(s)$,
- $h_4(s) = cos(h_1(s) * \pi)$, and
- $h_5(s) = h_1(s) * |cos(h_1(s) * \pi)|$.

Answer the following questions, and provide convincing arguments to support your answers:

a) Can you guarantee that h_2 be admissible? How about h_3, h_4 and h_5?

b) Can we guarantee that A* with heuristic h_2 will generate an optimal path? How about A* with heuristic h_3, h_4 or h_5?

Heuristics

- h(n) is an underestimate if there is no path from n to the goal node with a cost less than h(n) (i.e. it underestimates the cost of the path to the goal).
- An admissible heuristic is non-negative (>=0) heuristic function that does not overestimate the actual cost of a path to the goal.

3.4a – Admissible Heuristics

- We have that a heuristic h_1 is admissible for the range $0.1 \leq h_1 \leq 2.0$
    a) $h_2(s) = h_1(s) + 5$

Admissible: Heuristic is non-negative, and doesn't overestimate the cost to the goal

3.4a – Admissible Heuristics

- We have that a heuristic h_1 is admissible for the range $0.1 \leq h_1 \leq 2.0$
  - *a)* $h_2(s) = h_1(s) + 5$ - We cannot guarantee that this heuristic is admissible, as $h_2 > h_1$. Because of this, we cannot say for sure whether or not $h_2(s)$ overestimates the true cost to the goal, and therefore it is not admissible.

Admissible: Heuristic is non-negative, and doesn't overestimate the cost to the goal

3.4a – Admissible Heuristics

- We have that a heuristic h_1 is admissible for the range $0.1 \leq h_1 \leq 2.0$
  a) $h_2(s) = h_1(s) + 5$ - We cannot guarantee that this heuristic is admissible, as $h_2 > h_1$. Because of this, we cannot say for sure whether or not $h_2(s)$ overestimates the true cost to the goal, and therefore it is not admissible.
  b) $h_3(s) = 2\ h_1(s)$

Admissible: Heuristic is non-negative, and doesn't overestimate the cost to the goal

3.4a – Admissible Heuristics

- We have that a heuristic h_1 is admissible for the range $0.1 \leq h_1 \leq 2.0$
  a) $h_2(s) = h_1(s) + 5$ - We cannot guarantee that this heuristic is admissible, as $h_2 > h_1$. Because of this, we cannot say for sure whether or not $h_2(s)$ overestimates the true cost to the goal, and therefore it is not admissible.
  b) $h_3(s) = 2\, h_1(s)$ This heuristic will always be larger than or equal to $h_1(s)$ and therefore, we cannot guarantee its admissible (as above)

Admissible: Heuristic is non-negative, and doesn’t overestimate the cost to the goal

3.4a – Admissible Heuristics

- We have that a heuristic h_1 is admissible for the range $0.1 \leq h_1 \leq 2.0$
  a) $h_2(s) = h_1(s) + 5$ - We cannot guarantee that this heuristic is admissible, as $h_2 > h_1$. Because of this, we cannot say for sure whether or not $h_2(s)$ overestimates the true cost to the goal, and therefore it is not admissible.
  b) $h_3(s) = 2\, h_1(s)$ This heuristic will always be larger than or equal to $h_1(s)$ and therefore, we cannot guarantee its admissible (as above)
  c) $h_4(s) = \cos(h_1(s) \times \pi)$

Admissible: Heuristic is non-negative, and doesn’t overestimate the cost to the goal

3.4a – Admissible Heuristics

- We have that a heuristic h_1 is admissible for the range $0.1 \leq h_1 \leq 2.0$
  a) $h_2(s) = h_1(s) + 5$ - We cannot guarantee that this heuristic is admissible, as $h_2 > h_1$. Because of this, we cannot say for sure whether or not $h_2(s)$ overestimates the true cost to the goal, and therefore it is not admissible.
  b) $h_3(s) = 2\ h_1(s)$ This heuristic will always be larger than or equal to $h_1(s)$ and therefore, we cannot guarantee its admissible (as above)
  c) $h_4(s) = \cos(h_1(s) \times \pi)$ We know that $-1 \leq \cos(x) \leq 1$ and $0.1 \leq h_1(s) \leq 2.0$. This means that there may be some state s such that $\cos(h_1(s) \times \pi) \geq h_1(s)$. For example, when $h_1(s) = 0.1$, $h_{4(s)} = \cos(0.1 \times \pi) = 0.99998 > 0.1$. Therefore for this reason we cannot guarantee the admissibility of $h_{4(s)}$

Admissible: Heuristic is non-negative, and doesn't overestimate the cost to the goal

3.4a – Admissible Heuristics

- We have that a heuristic h_1 is admissible for the range $0.1 \leq h_1 \leq 2.0$
  a) $h_2(s) = h_1(s) + 5$ - We cannot guarantee that this heuristic is admissible, as $h_2 > h_1$. Because of this, we cannot say for sure whether or not $h_2(s)$ overestimates the true cost to the goal, and therefore it is not admissible.
  b) $h_3(s) = 2\, h_1(s)$ This heuristic will always be larger than or equal to $h_1(s)$ and therefore, we cannot guarantee its admissible (as above)
  c) $h_4(s) = \cos(h_1(s) \times \pi)$ We know that $-1 \leq \cos(x) \leq 1$ and $0.1 \ \leq h_1(s) \leq 2.0$. This means that there may be some state s such that $\cos(h_1(s) \times \pi) \geq h_1(s)$. For example, when $h_1(s) = 0.1$, $h_{4(s)} = \cos(0.1 \times \pi) = 0.99998 > 0.1$. Therefore for this reason we cannot guarantee the admissibility of $h_{4(s)}$
  d) $h_5(s) = h_1(s)|\cos(h_1(s) \times \pi)|$

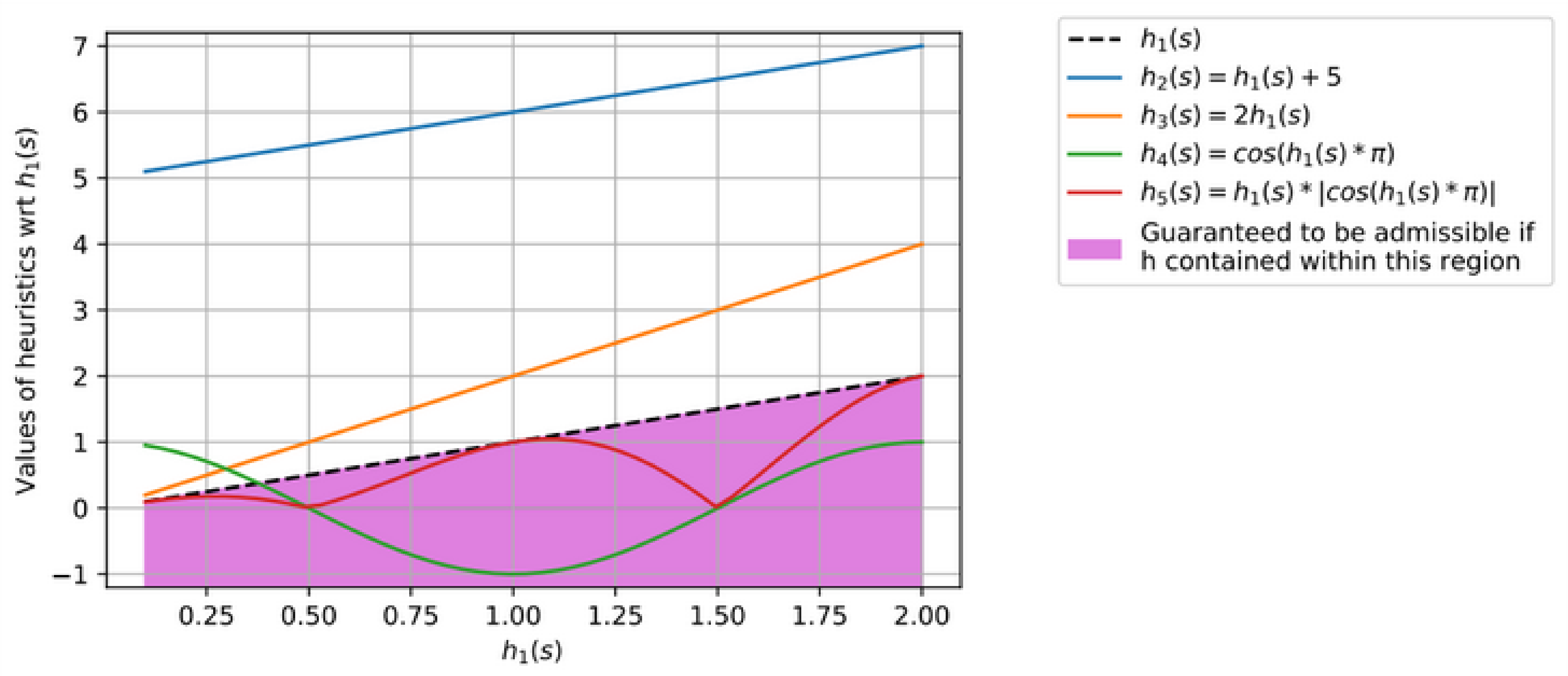
Admissible: Heuristic is non-negative, and doesn't overestimate the cost to the goal

3.4a – Admissible Heuristics

- We have that a heuristic h_1 is admissible for the range $0.1 \leq h_1 \leq 2.0$
  a) $h_2(s) = h_1(s) + 5$ - We cannot guarantee that this heuristic is admissible, as $h_2 > h_1$. Because of this, we cannot say for sure whether or not $h_2(s)$ overestimates the true cost to the goal, and therefore it is not admissible.
  b) $h_3(s) = 2\ h_1(s)$ This heuristic will always be larger than or equal to $h_1(s)$ and therefore, we cannot guarantee its admissible (as above)
  c) $h_4(s) = \cos(h_1(s) \times \pi)$ We know that $-1 \leq \cos(x) \leq 1$ and $0.1 \ \leq h_1(s) \leq 2.0$. This means that there may be some state s such that $\cos(h_1(s) \times \pi) \geq h_1(s)$. For example, when $h_1(s) = 0.1$, $h_{4(s)} = \cos(0.1 \times \pi) = 0.99998 > 0.1$. Therefore for this reason we cannot guarantee the admissibility of $h_{4(s)}$
  d) $h_5(s) = h_1(s)|\cos(h_1(s) \times \pi)|$ This heuristic is equivalent to $h_1(s)|h_4(s)|$. Since $h_1(s)$ is being scaled by the absolute value of the output of $h_4(s)$, in which $0 \leq h_4(s) \leq 1$ we can guarantee that $h_5(s) \leq h_1(s) \forall s$. Thus, since $h_1(s)$ is admissible, $h_5(s)$ is also admissible.

Admissible: Heuristic is non-negative, and doesn't overestimate the cost to the goal

3.4a – Admissible Heuristics



3.4b – Optimal Heuristics

- In the context of A* search, is a heuristic guaranteed to generate an optimal path?
  - When choosing between two possible nodes to expand in the A* search algorithm, the priority queue is ordered using the quantity $f(s) = g(s) + h(s)$ in which nodes with lower $f(s)$ values are chosen first.

Admissible: Heuristic is non-negative, and doesn't overestimate the cost to the goal

3.4b – Optimal Heuristics

$a)\ \mathrm{h}_2(s) = h_1(s) + 5$

Admissible: Heuristic is non-negative, and doesn't overestimate the cost to the goal

3.4b – Optimal Heuristics

a) $\mathrm{h}_2(s) = h_1(s) + 5$ If $h_1(s)$ is an admissible heuristic, replacing it with $\mathrm{h}_2(s) = h_1(s) + 5$ will not change the ordering the queued nodes – this just shifts the f(s) value of all of the nodes by a constant amount.

Admissible: Heuristic is non-negative, and doesn't overestimate the cost to the goal

3.4b – Optimal Heuristics

b. $h_3(s) = 2h_1(s)$

Admissible: Heuristic is non-negative, and doesn't overestimate the cost to the goal

3.4b – Optimal Heuristics

b. $h_3(s) = 2h_1(s)$ is not admissible and it is not guaranteed to generate an optimal path – the shift here is not constant.

- This can create cases where the h term in $f = g + h$ becomes too prominent and changes which node will be chosen

Admissible: Heuristic is non-negative, and doesn't overestimate the cost to the goal

3.4b – Optimal Heuristics

c. $h_4(s) = \cos(h_1(s) \times \pi)$

Admissible: Heuristic is non-negative, and doesn't overestimate the cost to the goal

3.4b – Optimal Heuristics

c. $h_4(s) = \cos(h_1(s) \times \pi)$ this can change the priority order of the queue, compared to the order determined by h_1

- Consider $h_1(s_1) = \frac{1}{4}$, $h_1(s_2) = \frac{1}{2}$ and $h_1(s_3) = \frac{3}{4}$
- In this case, $h_4(s_1) = \frac{\sqrt{2}}{2}$, $h_4(s_2) = 0$ and $h_4(s_3) = -\frac{\sqrt{2}}{2}$

- This means that the order in which the nodes are expanded from the frontier of the PQ can differ between A* search using heuristic h_1 or h_4
- Therefore, h_4 is not guaranteed to be an admissible heuristic, so we cannot guarantee that using $h_4(s)$ will generate an optimal path

Admissible: Heuristic is non-negative, and doesn't overestimate the cost to the goal

3.4b – Optimal Heuristics

d. $h_5(s) = h_1(s) \times |\cos(h_1(s) \times \pi)|$

Admissible: Heuristic is non-negative, and doesn't overestimate the cost to the goal

3.4b – Optimal Heuristics

d. $h_5(s) = h_1(s) \times |\cos(h_1(s) \times \pi)|$ is always equal to or less than the admissible heuristic h_1. This means it is also an admissible heuristic, and will generate an optimal solution

Admissible: Heuristic is non-negative, and doesn't overestimate the cost to the goal