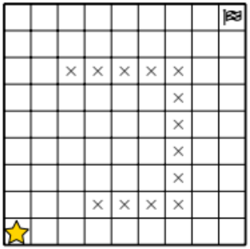
COMP3702 Tutorial 3

Aryaman Sharma

Aug 2022

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 costs 1 time step and does not move your agent.



Uniform cost

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).

Tactics for approaching Exercises 3.1 and 3.2

Try to map out what is required for designing (modelling) agent and environment interaction:

Tactics for approaching Exercises 3.1 and 3.2

Try to map out what is required for designing (modelling) agent and environment interaction:

- What is the agent?

Tactics for approaching Exercises 3.1 and 3.2

Try to map out what is required for designing (modelling) agent and environment interaction:

- What is the agent?
- What is the environment?

Tactics for approaching Exercises 3.1 and 3.2

Try to map out what is required for designing (modelling) agent and environment interaction:

- What is the agent?
- What is the environment?
- What is the agents goal?

Tactics for approaching Exercises 3.1 and 3.2

Try to map out what is required for designing (modelling) agent and environment interaction:

- What is the agent?
- What is the environment?
- What is the agents goal?
- What are its costs?

Modules

Modules

The **Environment Module** will model the world

- Within the module, think about how you want to structure the interaction with the agent, and use this to define different classes.

Modules

The **Environment Module** will model the world

- Within the module, think about how you want to structure the interaction with the agent, and use this to define different classes.

Search agent module

- Recall the common ingredients of search agents
  - Data structure (eg. queue, stack).
  - goal state checker.
  - A way to obtain neighbours of currently explored states.

Modules

The **Environment Module** will model the world

- Within the module, think about how you want to structure the interaction with the agent, and use this to define different classes.

Search agent module

- Recall the common ingredients of search agents
  - Data structure (eg. queue, stack).
  - goal state checker.
  - A way to obtain neighbours of currently explored states.
- One way to approach this is to develop one module with an abstract search agent class that is reused for each concrete algorithm, another way you could seperate code for each algorithm.

3.1 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).

3.1 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).

https://gist.github.com/aryaman-sh/e220d845ed0bb36f6335c49008dbcb1d

3.1

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

3.1

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().

3.1

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().

https://gist.github.com/aryaman-sh/62d8da41f0963264149034b24cd504bb

3.1 d

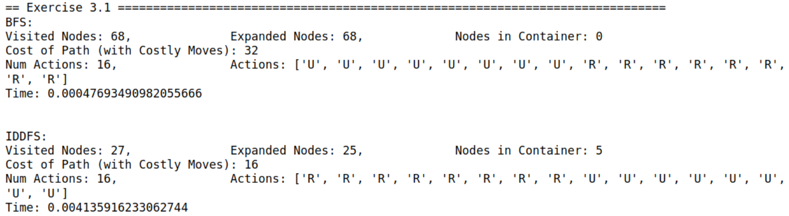
Compare the performance of BFS and IDDFS in terms of

- the number of nodes generated
- number of nodes on fringe when search terminates
- number of nodes on the expored set (if there is one) when the search terminates
- run time of the algorithm

3.1 d

Compare the performance of BFS and IDDFS in terms of

- the number of nodes generated
- number of nodes on fringe when search terminates
- number of nodes on the expored set (if there is one) when the search terminates
- run time of the algorithm



```
== Exercise 3.1 ===============================================================================
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: 0.00047693490982055666


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: 0.004135916233062744
```

3.2

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:

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

Costly moves

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

3.2 b

Implement UCS for this problem using a priority queue.

3.2 b

Implement UCS for this problem using a priority queue.

- consider heapq.heapify(container)

3.2 b

Implement UCS for this problem using a priority queue.

- consider heapq.heapify(container)

https://gist.github.com/aryaman-sh/2c75adc2f425c0bac5c83817eb9b173f

3.2 c

Compare the performances of BFS and UCS in terms of

- the number of nodes generated
- the number of nodes on the fringe when the search terminated
- the cost of the solution path

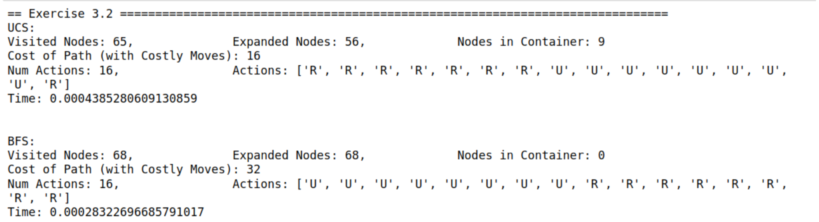
Discuss how and why these results differ from those for 3.1

3.2 c

Compare the performances of BFS and UCS in terms of

- the number of nodes generated
- the number of nodes on the fringe when the search terminated
- the cost of the solution path

Discuss how and why these results differ from those for 3.1



```
== Exercise 3.2 ==================================================================
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: 0.0004385280609130859


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: 0.00028322696685791017
```

3.2 d

Now derive and **admissible** heuristic for this path planning problem.

Informed search using heuristics

Idea: Don't ignore the goal when selecting paths.

Informed search using heuristics

Idea: Don't ignore the goal when selecting paths.

- Often there is extra knowledge that can be used to guide the search :**heuristics**.

Informed search using heuristics

Idea: Don't ignore the goal when selecting paths.

- Often there is extra knowledge that can be used to guide the search :**heuristics**.
- h(n) is an estimate of the cost of the shortest path from node n to goal node.

Informed search using heuristics

Idea: Don't ignore the goal when selecting paths.

- Often there is extra knowledge that can be used to guide the search :**heuristics**.
- h(n) is an estimate of the cost of the shortest path from node n to goal node.
- h(n) needs to be efficient to compute.

Informed search using heuristics

Idea: Don't ignore the goal when selecting paths.

- Often there is extra knowledge that can be used to guide the search :**heuristics**.
- h(n) is an estimate of the cost of the shortest path from node n to goal node.
- h(n) needs to be efficient to compute.
- h(n) does not overestimate, if there is no path from n to a goal with cost less than h(n).

Informed search using heuristics

Idea: Don't ignore the goal when selecting paths.

- Often there is extra knowledge that can be used to guide the search :**heuristics**.
- h(n) is an estimate of the cost of the shortest path from node n to goal node.
- h(n) needs to be efficient to compute.
- h(n) does not overestimate, if there is no path from n to a goal with cost less than h(n).
- An **admissible heuristic** is a nonnegative (≥ 0) heuristic function that never overestimates the actual cost of a path to a goal (it is optimistic)

3.2 d

Possible heuristic to use:

3.2 d

Possible heuristic to use:

- Manhattan distance of each tile to its goal position

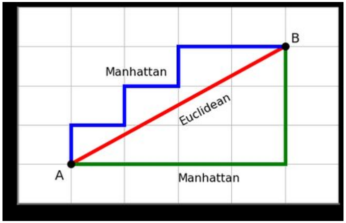
3.2 d

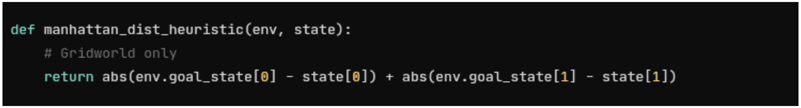
Possible heuristic to use:

- Manhattan distance of each tile to its goal position

Manhattan distance is admissible because every tile will have to be moved at least the number of spots in between itself and its correct position.

Manhattan distance





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

3.2 e

Using your heuristic, implement A^* search for solving this path problem.

3.2 e

Using your heuristic, implement A^* search for solving this path problem.

https://gist.github.com/aryaman-sh/c2ec9f4fde8e92b3f5a95f35a5298392

3.2 f

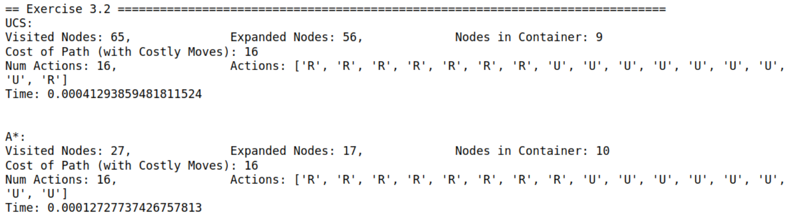
Compare the performances of UCS and A^* search in terms of

- the number of nodes generated
- the number of nodes on the fringe when the search terminates
- the cost of the solution path

3.2 f

Compare the performances of UCS and A^* search in terms of

- the number of nodes generated
- the number of nodes on the fringe when the search terminates
- the cost of the solution path



```
== Exercise 3.2 ==============================================================================
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: 0.00041293859481811524


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']
Time: 0.00012727737426757813
```

3.3

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

Possible heuristics to include:

3.3 a

Possible heuristics to include:

- Hamming distance (i.e. number of tiles that are different between the current state and the goal state)

3.3 a

Possible heuristics to include:

- Hamming distance (i.e. number of tiles that are different between the current state and the goal state)
- Manhattan distance of each tile to its goal position

3.3 a

Possible heuristics to include:

- Hamming distance (i.e. 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

3.3 a

Possible heuristics to include:

- Hamming distance (i.e. 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

All these heuristics would be admissible, but they may vary in how efficiently we can find the solution.

3.3 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.

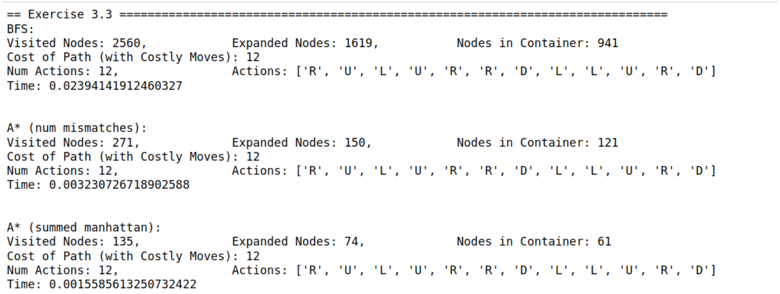
3.3 b

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.

https://gist.github.com/aryaman-sh/169224a843fdd4d2db58094439aaf293

3.3 c

Comparing performance



```
== Exercise 3.3 ===============================================================================
BFS:
Visited Nodes: 2560,          Expanded Nodes: 1619,          Nodes in Container: 941
Cost of Path (with Costly Moves): 12
Num Actions: 12,              Actions: ['R', 'U', 'L', 'U', 'R', 'R', 'D', 'L', 'L', 'U', 'R', 'D']
Time: 0.02394141912460327


A* (num mismatches):
Visited Nodes: 271,           Expanded Nodes: 150,           Nodes in Container: 121
Cost of Path (with Costly Moves): 12
Num Actions: 12,              Actions: ['R', 'U', 'L', 'U', 'R', 'R', 'D', 'L', 'L', 'U', 'R', 'D']
Time: 0.003230726718902588


A* (summed manhattan):
Visited Nodes: 135,           Expanded Nodes: 74,            Nodes in Container: 61
Cost of Path (with Costly Moves): 12
Num Actions: 12,              Actions: ['R', 'U', 'L', 'U', 'R', 'R', 'D', 'L', 'L', 'U', 'R', 'D']
Time: 0.0015585613250732422
```

3.4

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?

$$h_2(s) = h_1(s) + 5$$

$h_2(s) = h_1(s) + 5$

We know that h1 is an admissible heuristic. That is to say, it does not overestimate the true cost to the goal.
$h_2(s) = h_1(s) + 5$.

- We cant gurantee that $h_2(s)$ is asmissible because $h_2(s)$ is larger than $h_1(s)$.

$h_2(s) = h_1(s) + 5$

We know that h1 is an admissible heuristic. That is to say, it does not overestimate the true cost to the goal.
$h_2(s) = h_1(s) + 5$.

- We cant gurantee that $h_2(s)$ is asmissible because $h_2(s)$ is larger than $h_1(s)$.
- We cannot say for sure whether or not $h_2(s)$ overestimates the true cost to the goal, and it is not admissible.

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

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

- This heuristic will always be larger than $h_1(s)$. Thus for the same reason we cannot gurantee its admissibility.

$$h_4(s) = cos(h_1(s) * \pi)$$

$h_4(s) = cos(h_1(s) * \pi)$

- We know $-1 \leq cos(x) \leq 1 \implies$ there may be some state s such that $cos(h_1(s)\pi) > h_1(s)$.
- $h_1(s) = 0.1 \implies h_4(s) = 0.99998 > 0.1$.

$$h_5(s) = h_1(s) * |cos(h_1(s) * \pi)|$$

$h_5(s) = h_1(s) * |cos(h_1(s) * \pi)|$

- Since $h_1(s)$ is being scaled by a values between 0 and 1. We can gurantee $h_5(s) \leq h_1(s)$.
- Since we know that $h_1(s)$ is admissible and thus never overestimates the true cost to the goal, we can gurantee the same for $h_5(s)$.

3.4 b: Optimality

- We can gurantee h_2 generates optimal path. Why?

3.4 b: Optimality

- We can gurantee h_2 generates optimal path. Why?
- When choosing between two possible nodes to expand in A^*, the priority queue used $f(s) = g(s) + h(s)$.

3.4 b: Optimality

- We can gurantee h_2 generates optimal path. Why?
- When choosing between two possible nodes to expand in A^*, the priority queue used $f(s) = g(s) + h(s)$.
- Whichever node has lowest f value will be chosen next.

3.4 b: Optimality

- We can gurantee h_2 generates optimal path. Why?
- When choosing between two possible nodes to expand in A^*, the priority queue used $f(s) = g(s) + h(s)$.
- Whichever node has lowest f value will be chosen next.
- Thus if $h_1(s)$ is admissible heuristic (will lead to optimal solution), replacing with $h_1(s) + 5$ will never change the order of queued nodes.

Optimality $h_3(s)$

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

Optimality $h_3(s)$

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

- A similar argument cannot be used for h_3

Optimality $h_3(s)$

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

- A similar argument cannot be used for h_3
- The shift here is not constant but multiplicative.

Optimality $h_4(s)$, $h_5(s)$

Optimality $h_4(s)$, $h_5(s)$

$h_4(S) = cos(h_1(s)\pi)$

- The values can change fairly drastically, and this heuristic is no longer admissible.

Optimality $h_4(s)$, $h_5(s)$

$h_4(S) = cos(h_1(s)\pi)$

- The values can change fairly drastically, and this heuristic is no longer admissible.

$h_5(s) = h_1(s)(cos(h_1(s)\pi)$

Optimality $h_4(s)$, $h_5(s)$

$h_4(S) = cos(h_1(s)\pi)$

- The values can change fairly drastically, and this heuristic is no longer admissible.

$h_5(s) = h_1(s)(cos(h_1(s)\pi)$

- h_5 is always equal to or less than h_1. h_1 is being scaled down by a number that is always positive and ≤ 1.
- $\implies h_5$ is admissible and will generate an optimal solution.