**Artificial Intelligence**
**Assignment 1**

**Semester 1 2022**
**Due 11:59pm Wednesday 23 March 2022**

# 1   Pathfinding

Pathfinding is the problem of finding a path between two points on a plane. It is a fundamental task in robotics and AI. Perhaps the most obvious usage of pathfinding is in computer games, when an object is instructed to move from its current position to a goal position, while avoiding obstacles (e.g., walls, enemy fire) along the way.

   Pathfinding in commercial games is frequently accomplished using search algorithms[1]. We consider a simplified version in this assignment. The following shows a 2D map drawn using ASCII characters:

```
1 1 1 1 1 1 4 7 8 X
1 1 1 1 1 1 1 5 8 8
1 1 1 1 1 1 1 4 6 7
1 1 1 1 1 X 1 1 3 6
1 1 1 1 1 X 1 1 1 1
1 1 1 1 1 1 1 1 1 1
6 1 1 1 1 X 1 1 1 1
7 7 1 X X X 1 1 1 1
8 8 1 1 1 1 1 1 1 1
X 8 7 1 1 1 1 1 1 1
```

Given a start position and an end position on the map, our aim is to find a path from the start position to the end position. The character 'X' denotes an obstacle that cannot be traversed by a path, while the digits represent the elevation at the respective positions.

   Any position is indicated by the coordinates $(i, j)$, where $i$ is the row number (ordered top to bottom) and $j$ is the column number (ordered left to right). For example, the

---

[1]http://theory.stanford.edu/~amitp/GameProgramming/

top left position is $(1, 1)$, the bottom right is $(10, 10)$, while the position with elevation
'3' is $(4, 9)$. Given start position $(1, 1)$ and end position $(10, 10)$, a possible path is

```
* * * 1 1 1 4 7 8 X
1 1 * 1 1 1 1 5 8 8
1 1 * * * * * * * 7
1 1 1 1 1 X 1 1 * 6
1 1 1 1 1 X 1 * * 1
1 1 1 1 1 1 1 * 1 1
6 1 1 1 1 X 1 * * *
7 7 1 X X X 1 1 1 *
8 8 1 1 1 1 1 1 1 *
X 8 7 1 1 1 1 1 1 *
```

Note that we use **4-connectedness** for paths, which means any two points on the path
are only connected if they are vertically or horizontally (but not diagonally!) adjacent.

## 1.1 Problem formulation

Following the lecture notes, we formulate the problem as follows:

- **States:** Any obstacle-free position $(i, j)$ on the map.

- **Initial state:** A position $(i_0, j_0)$ given by the user.

- **Actions:** Since we consider 4-connectedness, only four actions are available: Up,
  down, left and right (**your program must expand each node in this order**).
  Available actions for positions adjacent to the map boundary or obstacles are
  reduced accordingly.

- **Transition model:** Moving left moves the current position to the left, etc.

- **Goal test:** Check if the current state is the end position $(i_*, j_*)$ given by the user.

- **Path cost:** Given a map $M$ and a path $P = \{(i_0, j_0), (i_1, j_1), \ldots, (i_N, j_N)\}$, the
  cost of the path is calculated as

$$g(P) = \sum_{k=1}^{N} c(i_{k-1}, j_{k-1}, i_k, j_k, M),$$

where

$$c(a, b, c, d, M) = \begin{cases} 1 + M(c, d) - M(a, b) & \text{if } M(c, d) - M(a, b) > 0 \\ 1 & \text{otherwise} \end{cases}$$

and $M(a, b)$ is the elevation at position $(a, b)$. In words, the cost of a path is the sum of the costs between two adjacent points of the path, and the cost between adjacent points is 1 plus the difference between the elevation of the two points if we climb "uphill", or simply 1 if we stay "level" or slide "downhill".

This means shorter paths which avoid climbing cost less. As an example, the cost in the path in the previous page is 25. What is the optimal (cheapest) path?

## 1.2    Your tasks

Solve pathfinding using Breadth-First Search (BFS), Uniform-Cost Search (UCS) and A* Search. You should base your program on the pseudocode GRAPH-SEARCH in the lecture slides, and carefully think about the appropriate data structures to use. For A* Search, you must implement two heuristics:

- Euclidean distance between current position and end position.

- Manhattan distance between current position and end position.

For the map in Page 1 with start position $(1, 1)$ and end position $(10, 10)$, your program should help you answer these questions:

1. Are the paths returned by the three methods different?

2. What about the optimality of the returned paths?

3. Which method is the most computationally and memory efficient?

4. Do the two heuristics for A* Search provide different solutions?

5. Does checking for repeated states matter in this problem?

## 1.3 Deliverables

Write your pathfinding program in Python 3 in a file called `pathfinder.py`. Your program must be able to be run as follows:

```
$ python pathfinder.py [map] [algorithm] [heuristic]
```

The inputs/options to the program are as follows.

- [map] specifies the path to map, which is a text file formatted according to this example (see next page):

  ```
  10 10
  1 1
  10 10
  1 1 1 1 1 1 4 7 8 X
  1 1 1 1 1 1 1 5 8 8
  1 1 1 1 1 1 1 4 6 7
  1 1 1 1 1 X 1 1 3 6
  1 1 1 1 1 X 1 1 1 1
  1 1 1 1 1 1 1 1 1 1
  6 1 1 1 1 X 1 1 1 1
  7 7 1 X X X 1 1 1 1
  8 8 1 1 1 1 1 1 1 1
  X 8 7 1 1 1 1 1 1 1
  ```

  The first line indicates the size of the map (rows by columns), while the second and third line represent the start and end positions respectively. The map data then follows, where all elevation values are integers from 0 to 9 inclusive.

- [algorithm] specifies the search algorithm to use, with the possible values of `bfs`, `ucs`, and `astar`.

- [heuristic] specifies the heuristic to use for A* search, with the possible values of `euclidean` and `manhattan`. This input is ignored for BFS and UCS.

Your program must then **print to standard output** the path returned by the search algorithm, in the following format:

```
* * * 1 1 1 4 7 8 X
1 1 * 1 1 1 1 5 8 8
1 1 * * * * * * 7
1 1 1 1 1 X 1 1 * 6
1 1 1 1 1 X 1 * * 1
1 1 1 1 1 1 1 * 1 1
6 1 1 1 1 X 1 * * *
7 7 1 X X X 1 1 1 *
8 8 1 1 1 1 1 1 1 *
X 8 7 1 1 1 1 1 1 *
```

where the path is indicated by asterisks '*' superimposed on the original map beginning from the start position and leading to the end position. **Do not include extraneous spaces or other characters in the output.**

If the given map or problem does not have a feasible path, your program must print

`null`

**Again, do not include extraneous spaces or other characters in the output.**

### 1.3.1 Python libraries

You are allowed to use NumPy to write your pathfinding program. The marking program will not be able to run your program to completion if other Python libraries are used.

## 1.4 Submission

You must submit your program files on Gradescope. Instructions on accessing Gradescope and submitting assignments are provided at `https://help.gradescope.com/article/5d3ifaeqi4-student-canvas`. Please use the course code **X3ZJZE** to enrol into the course. **For undergraduates**, please submit your pathfinding program (`pathfinder.py`) to **Assignment 1 - Undergraduates**. If there are any questions or issues with Gradescope, please contact Andrew via email at andrew.du@adelaide.edu.au.

## 1.5 Assessment

I will compile and run your code on several test problems. If it passes all tests you will get 15% (undergrads) or 12% (postgrads) of the overall course mark.

There will be no further manual inspection/grading of your program to award marks on the basis of coding style, commenting or "amount of code written.

## 1.6   Using other source code

You may not use other source code for this assignment. You should personally and carefully implement the search algorithms to fully understand the concept.

## 1.7   Due date and late submission policy

This assignment is due by 11:59pm Wednesday 23 March 2022. If your submission is late, the maximum mark you can obtain will be reduced by 25% per day (or part thereof) past the due date or any extension you are granted.

<div align="center">Continues next page for postgraduate section.</div>

# 2  Pathfinding by direct optimisation

For postgraduate students, completing this section successfully will give you the remaining 3% of the marks.

Here we shall attempt to directly optimise the path instead of step-by-step searching. We consider the simulated annealing algorithm shown in Algorithm 1. For more background on simulated annealing, see Section 4.1 of Russell and Norvig (3rd ed.).

---

**Algorithm 1** Simulated annealing for path optimisation

---

1:  **input** Initial path $P^0$, initial temperature $T_{ini}$, final temperature $T_{fin}$, cooling rate $\alpha$, segment length $d$.
2:  **output** Optimised path $P$.
3:  Initialise $T \leftarrow T_{ini}$, $P \leftarrow P^0$.
4:  **while** $T > T_{fin}$ **do**
5:      $P^h \leftarrow$ rand-local-adjust$(P, d)$.
6:      $\Delta g \leftarrow g(P) - g(P^h)$
7:      **if** $\Delta g > 0$ **then**
8:          $P \leftarrow P^h$
9:      **else**
10:         With probability $e^{\Delta g / T}$, $P \leftarrow P^h$.
11:     **end if**              /* Record $T$ and $g(P)$ here for bookkeeping.*/
12:     $T \leftarrow \alpha T$
13: **end while**
14: **return**  $P$

---

The algorithm receives as input a feasible (but non-optimal) path joining a start position and an end position on a map. The core idea is to iteratively perform random local adjustments to the path, and accept the new path if the adjustments improve the path cost (defined in Sec. 1.1), or accept it probabilistically if the cost is not improved. The process is repeated until the annealing temperature $T$ falls below a small value $T_{fin}$ given by the user. The temperature reduction is conducted as $T = \alpha T$ , where $0 < \alpha < 1$ is the cooling rate (also supplied by the user). See Section 4.1 of Russell and Norvig (3rd ed.) for more details.

The main body of the algorithm is conceptually simple — the hardest part is the routine to perform the random adjustments. Fortunately we can rely on the BFS program written in the previous section. The method is shown in Algorithm 2.

Note that the adjustments cannot make the path infeasible, i.e., any resulting path still joins the original start position and end position required by the user.

**Algorithm 2** Make random local adjustment on path

1: **input** Path $P$, segment length $d$.
2: **output** Adjusted path $P^h$.
3: Random pick a point $(u, v)$ on $P$.
4: Pick as $(x, y)$ the point of $d$ positions away from $(u, v)$ along $P$ towards the end position. If such a point does not exist, use the end position for $(x, y)$.
5: Find a random path $S$ joining $(u, v)$ and $(x, y)$ using **randomised BFS** (see text below).
6: Replace path segment in $P$ between $(u, v)$ and $(x, y)$ with $S$. Store new path as $P^h$.
7: **return** $P^h$.

To perform randomised BFS, only a minor tweak to the original BFS algorithm is required — the order of actions for expanding each node in the search tree is randomised every time. For example, while in Sec. 1.1 the order is fixed as UDLR (up, down, left, right), we randomise this at every instance to be LURD, DLUR, etc. The following shows randomised adjustments with $d = 5$, and $(u, v) = (8, 1)$ and $(x, y) = (10, 4)$.

```
* 1 8 1 1 2 4 7 8 X     * 1 8 1 1 2 4 7 8 X     * 1 8 1 1 2 4 7 8 X
* 1 1 5 1 5 1 5 8 8     * 1 1 5 1 5 1 5 8 8     * 1 1 5 1 5 1 5 8 8
* 4 2 2 1 6 1 4 6 7     * 4 2 2 1 6 1 4 6 7     * 4 2 2 1 6 1 4 6 7
* 5 1 7 0 3 5 1 1 6     * 5 1 7 0 3 5 1 1 6     * 5 1 7 0 3 5 1 1 6
* 7 8 1 2 6 8 1 5 1     * 7 8 1 2 6 8 1 5 1     * 7 8 1 2 6 8 1 5 1
* 7 4 1 1 4 2 2 4 2     * 7 4 1 1 4 2 2 4 2     * 7 4 1 1 4 2 2 4 2
* 5 1 2 1 2 7 5 1 6     * 5 1 2 1 2 7 5 1 6     * 5 1 2 1 2 7 5 1 6
* 7 1 3 4 2 0 4 2 1     * * 1 3 4 2 0 4 2 1     * 7 1 3 4 2 0 4 2 1
* * 1 1 1 5 1 1 9 1     8 * 1 1 1 5 1 1 9 1     * * * 1 5 1 1 9 1
X * * * * * * * * *     X * * * * * * * * *     X 8 7 * * * * * * *
```

## 2.1 Your tasks

Implement simulated annealing for path optimisation. As a sanity check, test your program on the following map with start position $(1, 1)$ and end position $(10, 10)$, with the initial path given by your (deterministic) BFS method from the previous section.

```
2 1 8 1 1 2 4 7 8 X
3 1 1 5 1 5 1 5 8 8
4 4 2 2 1 6 1 4 6 7
8 5 1 7 0 3 5 1 1 6
```

```
3 7 8 1 2 6 8 1 5 1
2 7 4 1 1 4 2 2 4 2
6 5 1 2 1 2 7 5 1 6
7 7 1 3 4 2 0 4 2 1
8 8 1 1 1 5 1 1 9 1
X 8 7 1 3 1 7 1 0 0
```

Use parameter values $T_{ini} = 10$, $T_{fin} = 0.001$, $\alpha = 0.99$, and $d = 5$. Your program should help you answer the following questions:

1. Does simulated annealing find the optimal path every time?

2. How important is it to be able to accept an inferior path? Investigate by disabling Step 10 in Algorithm 1.

3. How sensitive is the performance to the parameter settings? Investigate by changing the values of $T_{ini}$, $T_{fin}$, $\alpha$ and $d$.

## 2.2 Deliverables

Write your simulated annealing pathfinder program in Python 3 in a file called `sapathfinder.py`. Your program must be able to be run as follows:

```
$ python sapathfinder.py [map] [init] [tini] [tfin] [alpha] [d]
```

The test program will assume that you would use the same programming language as in Sec. 1, and that you have a working program (`pathfinder.py`) for the tasks in Sec. 1.

The inputs/options to the program are as follows.

- `[map]` specifies the path to a map, formatted according to Sec. 1.3.

- `[init]` specifies the path to an initial path, encoded according to the output of the program in Sec. 1.3.

- `[tini]` and `[tfin]` specifies the initial and final temperature respectively.

- `[alpha]` specifies the cooling rate.

- `[d]` specifies the segment length for random local path adjustments.

Your program must then **print to standard output** the optimised path, as well as the evolution of the temperature and path cost, in the manner of this example:

```
* 1 8 1 1 2 4 7 8 X
* 1 1 5 1 5 1 5 8 8
* 4 2 2 1 6 1 4 6 7
* 5 1 7 0 3 5 1 1 6
* 7 8 1 2 6 8 1 5 1
* 7 4 1 1 4 2 2 4 2
* * * * 1 2 7 5 1 6
7 7 1 * 4 2 0 4 2 1
8 8 1 * 1 5 1 1 9 1
X 8 7 * * * * * * *
T = 10.000000, cost = 38
T = 9.900000, cost = 44
T = 9.801000, cost = 42
.
.
.
T = 5.151371, cost = 40
T = 5.099857, cost = 40
T = 5.048859, cost = 41
.
.
.
T = 0.001014, cost = 23
T = 0.001004, cost = 23
```

**Do not include extraneous spaces or other characters in the output.**

Submit your program in the same way as the submission for Sec. 1. **For postgraduates**, please submit your pathfinding programs (`pathfinder.py` and `sapathfinder.py`) to **Assignment 1 - Postgraduates**. The due date, late submission policy and code reuse policy are also the same as in Sec. 1.

∼∼∼ The End ∼∼∼