COMPX216 Assignment 1
Uninformed and Heuristic Search in a Zen Puzzle Garden

Abstract

In this assignment, you will implement a Zen puzzle garden and apply various uninformed and heuristic search algorithms to find solutions. In the last step, you will examine the source code of A* search and modify it to implement a beam-search version of A*. But first, you will need to set up the Python environment used in this course.

1 Setup

First, we need to create a so-called "virtual environment" in Python and install dependencies (i.e., packages/libraries needed for our program) in this environment. Some installation steps may take a while, so it is recommended that you plan ahead and have things to work on in the meantime. We assume a Linux system in what follows.

Open a command-line terminal, and the default working directory should be your home directory. Although the virtual environment can be installed here, it is recommended that you install it in a dedicated COMPX216 folder. Let us create and navigate to it by inputting the following lines in the terminal and hitting "Enter" after each line. In a Linux terminal, use `Ctrl-Shift-C` to copy and `Ctrl-Shift-V` to paste.

```
mkdir Documents/COMPX216
cd Documents/COMPX216
```

The prompt should now show that we are in `~/Documents/COMPX216`. Let us create our virtual environment here.

```
python3 -m venv venv
```

To actually work in this environment, we need to activate it:

```
source venv/bin/activate
```

Note that we need to activate the virtual environment every time we open a new terminal. When installing packages, check the left of the command-line prompt to ensure you have the desired virtual environment activated. After the previous step, the prompt should show `(venv)` on the left.

Download the `requirements.txt` file from Moodle and move it to your working directory. Install the requirements using `pip`.

```
pip install -U pip
pip install wheel
pip install -r requirements.txt
```

We will be using the `aima-python` GitHub repository. Access it using a browser: `https://github.com/aimacode/aima-python/tree/master`. We will walk through its installation guide. Note that we will not be using the `requirements.txt` file provided in the repository because it does not specify the versions of packages, which can lead to installation problems.

Click the green “Code” button near the top of the page on GitHub. The option to clone the repository locally with https should be shown as default. Copy the given URL and clone it using the `git clone` command. The command in the terminal should look like this:

```
git clone https://github.com/aimacode/aima-python.git
```

Check that the repository has been cloned:

```
ls
```

This is a very useful Linux command that lists the content of a directory. You can look up commands like `ls` and `cd` using a search engine. A command’s usage can also be looked up in a terminal using the `man` command.

```
man ls
```

Now, navigate to the local cloned code repository.

```
cd aima-python
```

When typing a directory or file name in a Linux terminal, press “Tab” for auto-completion and suggestions.

Fetch the datasets required for the AIMA software:

```
git submodule init
git submodule update
```

Install `pytest` and run tests to verify your setup.

```
py.test
```

If the tests return no error, you have successfully set up the environment.

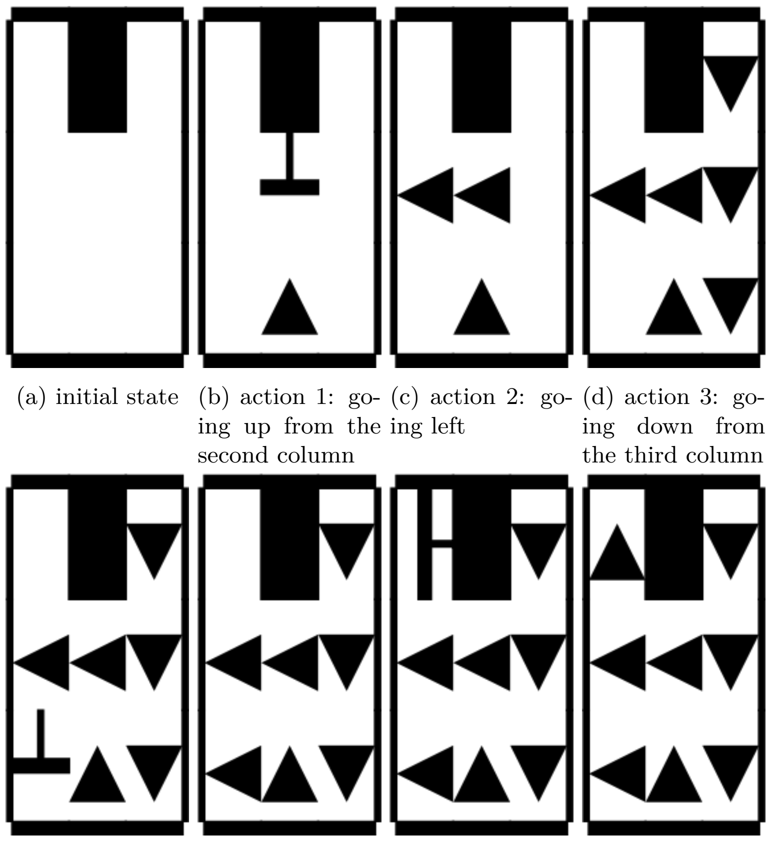
2 Zen Puzzle Garden

We will implement a simplified version of the Zen puzzle garden game in Amos, M., & Coldridge, J. (2012). A genetic algorithm for the Zen Puzzle Garden game. Natural Computing, 11, 353-359. It is a commercially available game, and example videos can be found online.

Consider a Zen garden to be raked by a monk. The rules are as follows:

1. The garden is surrounded by a perimeter.
2. The garden may contain immovable rocks.
3. Each tile of the garden is in one of three states: unraked, raked, or rock.
4. The garden starts unraked, i.e., the initial state may contain only unraked and rock tiles.
5. The monk's objective is to rake every unraked tile, i.e., a goal state may contain only raked and rock tiles.
6. The monk starts on the perimeter, i.e., outside the garden, and must finish on the perimeter when the last tile is raked.
7. When the monk is on the perimeter, he can choose to enter the garden via any unraked tile.
8. When in the garden, the monk can only move on unraked tiles.
9. Once moving in the garden, including when entering, the monk has to keep moving in the same direction until a) he is stopped by a rock or a raked tile, or b) he exits the garden by moving onto the perimeter.
10. When stopped in the garden, the monk chooses to go in a different direction, assuming that direction starts with an unraked tile or the perimeter. One exception is that the monk is not allowed to step into the garden and immediately "backstep" out where he entered. In other words, the monk is allowed to turn 90° either way but not 180° when stopped.
11. The monk can only move horizontally or vertically, but not diagonally.
12. When the monk exits a tile, it becomes raked. Specifically, the monk leaves behind a trail of raked tiles as he moves, but the tile he is currently standing on is considered unraked.

Figure 1 shows an example solution to a 3x3 garden. The border lines represent the perimeter. A rock sits at the middle of the top row, shown as a solid block. Raked tiles and the direction of raking are represented by triangles. When the monk is in the garden, he and the direction he faces are represented by a "T" symbol, resembling the rake he uses. The actions of entering the garden and turning each have unit cost. There is no cost associated with moving on



(a) initial state (b) action 1: going up from the second column (c) action 2: going left (d) action 3: going down from the third column

(e) action 4: going up from the first column (f) action 5: going left (g) action 6: going right from the first row (h) action 7: going up

Figure 1: A zen garden from its initial state to a finished state action by action

the perimeter. The solution presented here involves 7 actions and has a cost of 7. Note that due to the "no backstep" rule, when the monk enters from the bottom in action 4, he can only exit on the left in action 5, and likewise when he enters from the left in action 6, he can only exit on the top in action 7.

3 Tasks

There are four tasks you need to perform for this assignment. Download the provided `assignment1.py` file from Moodle and move it into the `aima-python` directory. This file contains the skeleton code for this assignment. You can follow the comments in this code to complete the assignment. Test code for each task is provided in the lower half of the file. After completing a task, uncomment the corresponding block of test code by removing the `'''` marks around it to test it in your run. Do not modify the names of classes, functions, or variables provided in the file. This is the file you will submit for marking. It is permitted to import additional modules required by your implementation. Download the provided `assignment1aux.py` and `assignment1config.txt` files to the same location.

Hints are provided at the end of this document. You are encouraged to read through the hints before attempting each task.

Write code in `assignment1.py` using a text editor or IDE of your choosing (e.g., IDLE).

Once you have written the code for a task, uncomment its test code and run the assignment code in a command-line terminal as shown below. Remember to ensure that your working directory is `aima-python`, and your virtual environment is activated.

```
python assignment1.py
```

3.1 Task 1

Your first task is to complete the `read_initial_state_from_file()` function. This function takes a filename as a string, e.g., the `assignment1config.txt` file, and returns a state built from the configuration read from the file.

A configuration file's first line is a single number indicating the height (number of rows) of the garden. The second line is a single number indicating the width (number of columns) of the garden. Each line that may follow contains a pair of numbers separated by a comma indicating the position of a rock tile. A position of `i,j` indicates a rock tile at the *i*-th row and *j*-th column (with indexing starting from 0). For example, the garden in Figure 1 can be described using the configuration below. The name of the file containing this configuration is used as *input* to the function.

```
3
3
0,1
```

A state is a tuple of three values. The first value is the map (garden), the second value is the position of the agent (monk), and the third value is the direction the agent faces. A map is a tuple of rows, and each row is a tuple of strings each representing a tile: `'rock'` indicates a rock tile, `'left'`/`'up'`/`'right'`/`'down'` indicate a raked tile and the direction in which it has been raked, and `''` indicates an unraked tile. When the agent is not on the map, i.e., the monk is on the perimeter, its position and direction are represented as `None`. When the agent is on the map, i.e., the monk is in the garden, its position at the *i*-th row and *j*-th column is encoded as a tuple of integers `(i, j)`, and its direction is encoded as a string `'left'`/`'up'`/`'right'`/`'down'`. For example, the configuration above should return the following *output* state via the `read_initial_state_from_file()` function. It is a tuple and only formatted for easy viewing.

```
(
 (
  ('', 'rock', ''),
  ('', '',     ''),
  ('', '',     '')
 ),
 None,
 None
)
```

3.2 Task 2

Your second task is to complete implementation of the Zen puzzle garden game described in Section 2. The `ZenPuzzleGarden` class' `__init__()` function has been written, allowing it to be initialised from a configuration file or a state. You need to implement three functions in the class: `actions()`, `result()`, and `goal_test()`.

Refer to Figure 1 to help with your implementation. Refer to the `search.py` file in the `aima-python` directory (or on GitHub) for source code of functions used.

Below is an example state, the state corresponding to the outcome of action 4 in Figure 1. Consider what actions are available in this state, and what state each action leads to.

```
(
 (
  ('',     'rock', 'down'),
  ('left', 'left', 'down'),
  ('',     'up',   'down')
 ),
 (2, 0),
 'up'
)
```

After implementing all three functions of `ZenPuzzleGarden` in Task 2, uncomment its corresponding test code block to solve it using breadth-first graph search. As each action in this problem has unit cost, breadth-first graph search is equivalent to uniform cost search.

3.2.1 Task 2.1

You need to implement the `actions()` function. It takes a state as input and returns a list of all allowed actions in that state. You may use any data structure you deem suitable for encoding an action, but make sure to be consistent with how you will apply an action in the `result()` function.

3.2.2 Task 2.2

You need to implement the `result()` function. It takes a state and an action as input and returns a state resulting from the action applied to the state.

3.2.3 Task 2.3

You need to implement the `goal_test()` function. It takes a state as input and returns a Boolean value indicating whether the state is a goal state. A goal state is one where the objective of the game is accomplished.

3.3 Task 3

Your third task is to implement a heuristic cost function for A* search. Assign it to the `astar_heuristic_cost` variable for marking. Note that for A* search to return an optimal solution, the heuristic cost function needs to be *admissible*, i.e., it should never overestimate the cost of the path from a state to a goal state. You can define this function with `def`, but you are encouraged to write it as a lambda expression.

This heuristic function will be used in the `astar_search()` function provided by AIMA's `search.py` code as its `h` argument.[1] Refer to the source code of `astar_search()` in `search.py`.

3.4 Task 4

Your fourth task is to complete the `beam_search()` function which is a beam-width version of A* search. Vanilla A* search is prone to frontiers that increase in size rapidly in large search spaces. Beam-width search restricts the frontier to never exceed a certain size, i.e., the width of the search "beam", by discarding nodes in the frontier that have comparatively high estimated cost.

The `beam_search()` function takes three arguments: `problem` is an instance of a (Zen puzzle garden) problem, `f` is a cost function that takes a search node as input and returns an estimated cost for a solution path through this node, and `beam_width` is the maximum number of nodes allowed in a frontier. Note that `f` in `beam_search()` is the sum of `g` and `h`, where `g` is the path cost to the node under consideration. (In contrast, the A* implementation in `search.py` takes `h` as an argument and adds `g` later.)

4 Running Your Own Tests

You can run your own code and tests using the Python interpreter in a terminal.

```
    python
```

[1]Remember that functions are first-class citizens in Python and can be used like other objects when passing arguments to functions.

IDLE's shell functions as an interpreter as well. Note that due to the shell's limitations, the `animate()` function in `assignment1aux.py` cannot draw over existing visualisation as intended.

In an interpreter, import modules you would like to use and test.

```
from search import *
from assignment1 import *
from assignment1aux import *
```

You can then write code as you would in a Python file. Every time you hit `Enter`, the interpreter runs the code you have inputted. When the interpreter detects that code you have entered is incomplete, e.g., the start of a loop, an if statement, or missing closing brackets, it will display `...` instead of `>>>` on the left, and you will be allowed to enter multiple lines of code before the interpreter runs them all together. Signal for the interpreter to run by inputting an empty line.

You are encouraged to make your own configuration files for different garden layouts to test your implementation. Some example interpreter code is given below.

```
>>> garden = ZenPuzzleGarden('my_config.txt')
>>> print('Running beam search.')
>>> before_time = time()
>>> h = astar_heuristic_cost
>>> node = beam_search(garden, lambda n: n.path_cost + h(n), 50)
>>> after_time = time()
>>> print(f'Beam search took {after_time-before_time} seconds.')
>>> if node:
...     print(f'Solution with a cost of {node.path_cost} found.')
...     animate(node)
... else:
...     print('No solution was found.')
...
```

A Hints

A nice way to interact with Python code is through the use of so-called "notebooks". The `search.ipynb` notebook file in `aima-python` contains good examples of `Problem` subclasses and search algorithms. Open Jupyter Notebook in `aima-python` with your virtual environment activated.

```
jupyter notebook
```

In the notebook window, find and open `search.ipynb` in the listed files and view its code examples.

A.1 Hints for Task 1

The `split()` function can split a string into a list of strings based on a given separator string.

The state data structure employs tuples, which are immutable, because each state needs to be hashable by the search algorithm to check for states for which nodes have been generated previously. However, when building a map from a configuration, it may be easier to use lists instead of tuples, because lists are mutable, so that their values can be modified. You can convert lists into tuples using the `tuple()` function after the modifications have been performed.

You can use list comprehension to build lists in a convenient manner. For example, the code below builds a list of ten values of zero.

```
my_list = [0 for _ in range(10)]
```

List comprehension can be nested to build a nested list of lists. For example, the code below builds a nested list of ten by five values of zero.

```
my_nested_list = [[0 for _ in range(5)] for _ in range(10)]
```

Comprehension can also be used to convert a nested list into a nested tuple.

```
my_nested_tuple = tuple(tuple(i) for i in my_nested_list)
```

A.2 Hints for Task 2.1

An action needs to contain all information required to encode a move unambiguously. Note that the direction of the move alone is insufficient when the agent needs to enter the map. The location of entry is also needed.

When the agent needs to make a turn on the maps, it can only turn 90°, so there are only two potential directions to consider. Also note that each direction is only a valid choice if not immediately blocked by a rock or raked tile.

An action should be encoded as a position-direction pair/tuple. For example, `((0, 2), 'down')` means starting from the zeroth row and second column and moving down.

You can initialise the list of available actions for a state as an empty list and use the `append()` function to add actions to the list depending on the state.

Below are a few examples of input and output of the `actions()` function using states in Figure 1.

A.2.1 Example 1

The input is the initial state, where the agent is outside the map and needs to enter via an unraked tile.

```
(
 (
  ('', 'rock', ''),
  ('', '',     ''),
  ('', '',     '')
 ),
 None,
 None
)
```

The output is a list of actions entering the map, whose order does not matter. Note that when the agent enters via a corner tile, it has two potential directions, whereas it has only one direction to go if entering via a non-corner side tile.

```
[
 ((0, 0), 'down'),
 ((0, 2), 'down'),
 ((0, 2), 'left'),
 ((1, 2), 'left'),
 ((2, 2), 'left'),
 ((2, 2), 'up'),
 ((2, 1), 'up'),
 ((2, 0), 'up'),
 ((2, 0), 'right'),
 ((1, 0), 'right'),
 ((0, 0), 'right')
]
```

A.2.2 Example 2

The input is the state reached by action 1, where the agent is on the map.

```
(
 (
  ('', 'rock', ''),
  ('', '',     ''),
  ('', 'up',   '')
 ),
 (1, 1),
 'up'
)
```

The output is a list of actions turning from the current location on the map.

```
[
 ((1, 1), 'left'),
 ((1, 1), 'right')
]
```

A.2.3 Example 3

The input is the state reached by action 4, where the agent just entered the map from the bottom.

```
(
 (
  ('',     'rock', 'down'),
  ('left', 'left', 'down'),
  ('',     'up',   'down')
 ),
 (2, 0),
 'up'
)
```

The output is a list of actions turning from the current location on the map.

```
[
 ((2, 0), 'left')
]
```

Note that the agent cannot step back in the direction going down, and it cannot turn right as the tile on the right is already raked.

A.3 Hints for Task 2.2

Use a `for` loop when you want to iterate for a number of iterations. Use a `while` loop when you want to iterate until a condition is (no longer) met.

Use `break` to jump out of the loop. Use `continue` to jump to the next iteration in the loop. Naturally, `return` breaks out of the loop.

Remember that carrying out an action has an exact cost of 1, unless the `path_cost()` function is overridden. It is not recommended to override this function in this assignment. An action should keep the agent going in the same direction until it either a) is stopped by a rock or raked tile, or b) moves off the map.

Use the `list()` function or list comprehension to create lists from tuples to enable changing of their values as the agent moves on the map. Remember to convert lists back to tuples when returning them in a state.

Below are a few examples of input and output of the `result()` function using states in Figure 1.

A.3.1 Example 1

The input is the initial state and an action to move up from the middle column.

```
(
 (
  ('', 'rock', ''),
  ('', '',     ''),
  ('', '',     '')
 ),
 None,
 None
),
((2, 1), 'up')
```

The output is the state after action 1.

```
(
 (
  ('', 'rock', ''),
  ('', '',     ''),
  ('', 'up',   '')
 ),
 (1, 1),
 'up'
)
```

A.3.2 Example 2

The input is the state after action 2 and an action to move down from the right column.

```
(
 (
  ('',     'rock', ''),
  ('left', 'left', ''),
  ('',     'up',   '')
 ),
 None,
 None
),
((0, 2), 'down')
```

The output is the state after action 3.

```
(
 (
  ('',     'rock', 'down'),
  ('left', 'left', 'down'),
  ('',     'up',   'down')
 ),
 None,
 None
)
```

A.3.3 Example 3

The input is the state after action 4 and an action to move left.

```
(
 (
  ('',     'rock', 'down'),
  ('left', 'left', 'down'),
  ('',     'up',   'down')
 ),
 (2, 0),
 'up'
),
((2, 0), 'left')
```

The output is the state after action 5.

```
(
 (
  ('',     'rock', 'down'),
  ('left', 'left', 'down'),
  ('left', 'up',   'down')
 ),
 None,
 None
)
```

A.4 Hints for Task 2.3

The `all()` function built into Python returns `True` if and only if an iterable contains only `True` values. The `any()` function returns `True` if and only if an iterable contains at least one `True` value. You can use comprehension to handle nested iterables.

A.4.1 Exampled 1

The following input state should return `False` when passed to the `goal_test()` function, because all tiles need to be raked or occupied by a rock, and the agent needs to finish outside the map.

```
(
 (
  ('',     'rock', 'down'),
  ('left', 'left', 'down'),
  ('left', 'up',   'down')
 ),
 None,
 None
)
```

A.4.2 Exampled 2

The following input state should return `True` when passed to the `goal_test()` function, because all tiles are raked or occupied by a rock, and the agent is outside the map.

```
(
 (
  ('up',   'rock', 'down'),
  ('left', 'left', 'down'),
  ('left', 'up',   'down')
 ),
 None,
 None
)
```

A.5 Hints for Task 3

The A* heuristic cost function takes a `Node` object as input. View the source code of the `Node` class in `search.py` to determine what it is used for and how to access its values.

Think of this cost as an optimistic estimate of the number of actions to take from the current state to a goal state. What are some simple ways of estimating how many actions it will take? Such estimates do not necessarily have to be accurate, as the point of a heuristic is to provide computationally efficient guidance.

A.6 Task 4

Since `beam_search()` is a variant of `astar_search()`, you may want to view its source code and the functions it calls. The only differences between these two functions are that a) `beam_search()` restricts the maximum size of its frontier, and b) `beam_search()` takes a cost function `f` which is the sum of `g` and `h`, while the A* implementation in `search.py` takes `h` as an argument and adds `g` later, as mentioned in Section 3.4.

You can copy code from `search.py` and modify it in order to implement `beam_search()`. Should you use the source code of `astar_search()` or the function it calls? Also remember you are allowed to import additional modules.

B Further Discussion

This section discusses the assignment further for students who are interested. It is completely voluntary reading, and you will not be evaluated on its content.

You may have realised that a raked tile is functionally the same as a rock, i.e., it stops a moving agent. In this assignment, each tile is encoded as a string to help with visualisation and analysis, but a more efficient way would be to use Boolean values: `False` represents an unraked tile, and `True` represents a raked tile or a rock. Another advantage to using Boolean values here is that it reduces the number of duplicate copies saved in the dictionary holding generated states because two maps are functionally equivalent if they have the same arrangement of unraked tiles: it does not matter whether an inaccessible tile is a raked tile or a rock, nor does the direction in which each tile is raked matter.

Zen puzzle garden has been confirmed as NP-complete. As of now there is no algorithm that is able to solve it in polynomial time. Heuristics can be employed to attempt to find solutions more quickly than uninformed search. However, certain heuristics may end up exploring more nodes than uninformed search before finding a solution, because they are prone to extensively exploring map layouts that are actually unsolvable. Due to the puzzle's NP-completeness, there is unlikely to be an algorithm that can determine the solvability of every layout in polynomial time. However, there are some general rules of thumb. Consider how these rules can be incorporated into a heuristic and whether their incorporation could potentially nullify a search algorithm's optimal solution guarantee.