COMPX216 Assignment 2
Local Search and Optimisation in KNetWalk

Abstract

In this assignment, you will implement a fitness function for the KNetWalk game. You will also apply and implement various local search and optimisation algorithms to find solutions.

1 KNetWalk

The task in this assignment is to solve KNetWalk puzzles. KNetWalk is an open source game, and you can install and play it for free on a Linux system. Examples of the game can also be found online.

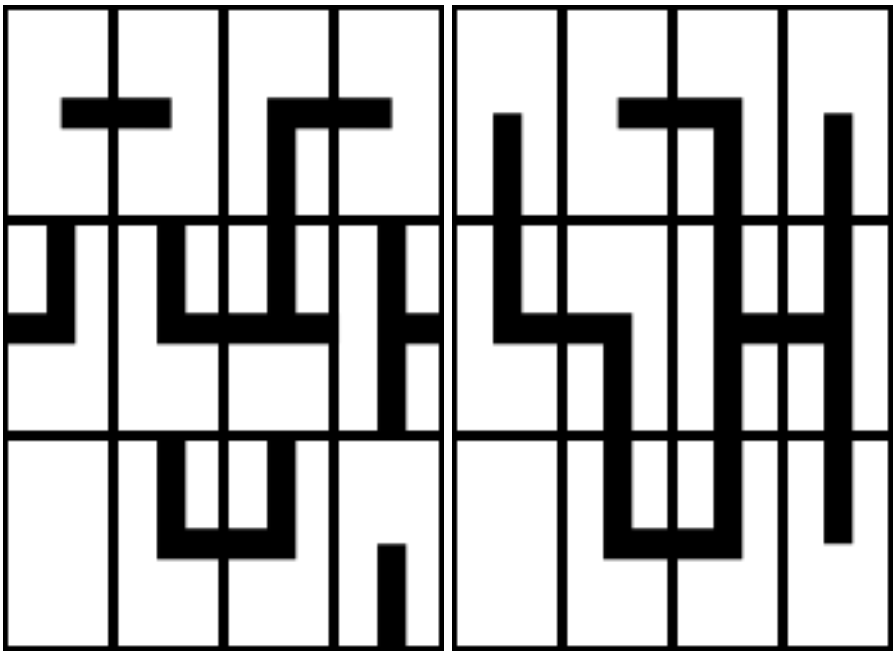
Consider a board of network tiles, and the objective is to rotate the tiles to connect them into one single network. Figure 1 shows an example of KNetWalk with three rows and four columns. All tiles start in random orientations, and they form a single connected network after appropriate rotations. A tile can be in one of four orientations:

- the default orientation, which we will encode as `0`,
- rotated 90° counterclockwise, which we will encode as `1`,
- rotated 180° counterclockwise, which we will encode as `2`, and
- rotated 270° counterclockwise, which we will encode as `3`.

Note that we will be tackling this puzzle as an optimisation problem and are not interested in the solution path showing how to get from the start state to the goal state.

2 Tasks

There are six tasks you need to perform for this assignment. Download the provided `assignment2.py` file from Moodle and move it into the `aima-python` directory. This file contains the skeleton code for this assignment. You can



(a) initial state (b) goal state

Figure 1: KNetWalk example

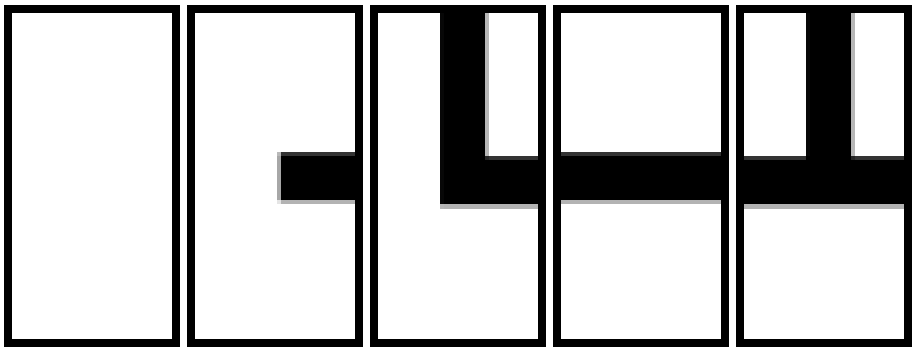
follow the comments in this code to complete the assignment. Test code for each task is provided in the lower half of the file. After completing a task, uncomment the corresponding block of test code by removing the `'''` marks around it to test it in your run. Do not modify the names of classes, functions, 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 `assignment2aux.py` and `assignment2config.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 `assignment2.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 assignment2.py
```



(a) empty (b) terminal (c) L-shaped (d) straight (e) T-shaped

Figure 2: KNetWalk tile types

2.1 Task 1

Your first task is to complete the `read_tiles_from_file()` function. This function takes a filename as a string, e.g., the `assignment2config.txt` file, and returns a state built from the configuration read from the file.

A configuration file contains a board of characters, with each character representing one of the five KNetWalk tile types shown in Figure 2. We will encode a connection to the right as `0`, one to the top as `1`, one to the left as `2`, and one to the bottom as `3`. The `read_tiles_from_file()` function needs to convert each tile from its character representation to what we call its "connection representation" as a tuple:

- empty tile: from `' '` to `()` with no connections,
- terminal tile: from `'i'` to `(0,)` with a connection to the right,
- L-shaped tile: from `'L'` to `(0, 1)` with connections to the right and top,
- straight tile: from `'I'` to `(0, 2)` with connections to the right and left, and
- T-shaped tile: from `'T'` to `(0, 1, 2)` with connections to the right, top, and left.

The connection representations facilitate easy transformation of a tile's connections using an orientation, as show in the Python code below.

```
my_tile = (0, 1, 2) # T-shaped tile
my_orientation = 2 # rotate 180 degrees
my_oriented_tile =
  tuple((con + my_orientation) % 4 for con in my_tile)
# (2, 3, 0) with connections to the left, bottom, and right
```

The converted tiles should be returned in a data structure representing the KNetWalk game board. You can use a nested tuple or a nested list for this. A nested tuple is recommended as this board does not need to be modified after being generated. Given the configuration in `assignment2config.txt`, the output as a nested tuple is as follows.

```
(
 ((0,),   (0,),   (0, 1),    (0,)),
 ((0, 1), (0, 1), (0, 1, 2), (0, 1, 2)),
 ((),     (0, 1), (0, 1),    (0,))
)
```

2.2 Task 2

Your second task is to complete implementation of KNetWalk's fitness/objective function. We will set the problem up so that the puzzle is solved when the fitness reaches its maximum value.

The `KNetWalk` class' `__init__()`, `actions()`, `result()`, and `goal_test()` functions have been written. You need to implement the `value()` function. It takes a state as input and returns an integer value representing its fitness. Fitness starts at 0, and increases by 1 for each connection from one tile to a neighbour. Note that for each connection from tile A to tile B, there is also a connection from tile B to tile A. Therefore, fitness is always an even number. Using Figure 1 as an example, the initial state has a fitness value of 10 due to 5 pairs of successful connections, while the goal state's fitness is 20.

After implementing `value()`, uncomment the task's corresponding test code block to solve the puzzle using hill-climbing with restart.

The `__init__()` function can initialise an object from a configuration file or a tile board. Inspect its code, which computes the board's maximum fitness and initialises a state corresponding to the given board. The maximum fitness value represents a solved board where all tiles are oriented correctly to form a single connected network. Note that `__init__()` initialises a random state of the board. Random initialisation facilitates restarts in search.

A state is a flat list of tile orientations.[1] Each orientation corresponds to a tile on the game board. Given a board with height `h` and width `w`, i.e., containing `h` rows each containing `w` tiles, a corresponding state should contain `(h * w)` orientations, and the `(i * w + j)`-th orientation in the state corresponds to the tile at the `i`-th row and `j`-th column on the game board. An orientation is a single integer number that is `0`, `1`, `2`, or `3`, as described in Section 1. The initial and goal states in Figure 1 are shown below.

```
initial_state = [0, 2, 3, 2, 1, 0, 0, 3, 0, 0, 1, 3]
goal_state = [3, 0, 2, 3, 0, 2, 3, 1, 0, 0, 1, 1]
```

[1] A state has to be a flat list to enable crossover in aima-python's genetic algorithm.

The `actions()` function takes a state as input and returns a list of all allowed actions in that state. Each action is a tuple containing three values: row index `i`, column index `j`, and new orientation `k`. Such an action would rotate the tile at the `i`-th row and `j`-th column to the new orientation `k`. The new orientation should be different from the tile's current orientation, which means there are three actions to rotate each tile, and the total number of actions returned by each call to `actions()` should always be `(3 * h * w)`.

The `result()` function takes a state and an action as input and returns a state resulting from the action applied to the state. To prevent errors, make sure you do not modify an existing state, e.g., do not write code like

```
state[i * w + j] = k
```

but instead produce a new state with the desired information like

```
new_state = state[:i*w+j] + [k] + state[i*w+j+1:]
return new_state
```

The `goal_test()` function takes a state as input and returns a Boolean value indicating whether the state is a goal state. This is a trivial function that simply uses `value()` to compute the state's fitness and checks whether it reaches the maximum fitness computed in `__init__()`.

2.3 Task 3

Your third task is to configure an exponential scheduling function for simulated annealing. The function has been provided to you. Run this task's test code to see that simulated annealing struggles to find a solution with its scheduling function in its default configuration. Examine the source code of `simulated_annealing()` and `exp_schedule()` in `search.py`. Configure the scheduling function's parameters, namely `k`, `lam`, and/or `limit`, so that simulated annealing can generally find a solution with fewer than ten restarts.

2.4 Task 4

Your fourth task is to configure a genetic algorithm. Examine the source code of `genetic_algorithm()` in `aima-python`'s `search.py`. Configure `pop_size`, `num_gen`, and `mutation_prob` so that a genetic algorithm can find solutions efficiently with these parameters.

2.5 Task 5

Your fifth task is to complete the `local_beam_search()` function, which can be viewed as "hill-climbing with a population of states". The population is kept at a certain size, i.e., the so-called "beam width". With a beam width of b, at each iteration, child states of all states in the population are computed, and the top-b fittest child states are kept as the new population for the next iteration. Like hill-climbing, local beam search returns a goal state if it is found in the

population. Without finding a goal state, it terminates if the fittest state in the next population is not fitter than the fittest state in the current population, and the fittest state in the current population is returned.

The `local_beam_search()` function takes two arguments: `problem` is an instance of a (KNetWalk) problem, and `population` is a list of states. The size of the initial population is used as the width of the beam.

2.6 Task 6

Your sixth task is to complete the `stochastic_beam_search()` function. It is similar to the genetic algorithm with only mutations and no crossover, but the two differ in that stochastic beam search uses search node expansion to obtain child states, while the genetic algorithm mutates states directly to obtain child states. Feel free to reuse your code from the previous task, as the two functions are largely the same. Compared to local beam search, its stochastic variant uses fitness-weighted random sampling to select the next population instead of selecting the top-b fittest states. Due to the stochastic nature of the algorithm, it does not terminate when the fittest state in the next population is not fitter than the fittest state in the current population, but it does return a goal state if one is found. It also terminates if a specified number of iterations have been performed, and the fittest state in the current population is returned in that case.

The `stochastic_beam_search()` function takes three arguments: `problem` is an instance of a (KNetWalk) problem, `population` is a list of states, and `limit` is the number of iterations allowed before termination.

3 Running Your Own Tests

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

```
python
```

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

```
from search import *
from assignment2 import *
from assignment2aux import *
```

You can then write code as you would in a Python file. Refer to the brief of the previous assignment for detailed instructions.

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

```
>>> network = KNetWalk('my_config.txt')
>>> network.actions(network.initial)
```

A Hints

A.1 Task 2

When testing whether a connection is successfully made, pay attention to tiles on the edges of the board. They do not have all four neighbours, i.e., above, below, left, and right.

Connections are made in pairs: for a connection from tile A to tile B to be successful, there has to be a connection from tile B to tile A as well.

A.2 Task 3 & 4

If a search algorithm does not find a solution efficiently, consider two common issues:

- Each search run terminates too early without sufficiently exploring the search space.
- Each search run takes too long, exploring parts of the search space unlikely to contain a solution.

Observe the search algorithms' behaviour and ask these questions: is it restarting too often with low fitness values, or is it not restarting enough? Configure the parameters to find a sweet spot where each search run explores the search space efficiently, yielding high fitness values without taking too long.

A.3 Task 5

Inspect the source code of `hill_climbing()` in `search.py`. You might find the `sorted()` function useful. In addition to the list to be sorted, it takes two additional optional arguments: a) `key`, which is a function that takes an item and assigns a value to it for sorting and b) `reverse`, which sorts from highest to lowest if set to true. The `key` is by default the identity function, and `reverse` is by default set to false.

A.4 Task 6

To perform weighted sampling without replacement, a good way is to use `numpy`'s `random.choice()` function.

```
import numpy as np
sample = np.random.choice(my_list, size, False, my_prob)
```

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.

It is clear that any state with at least one connection going to the edge of the board is not a valid solution. A smarter initialisation can avoid this, and all actions leading to this scenario can be discarded safely. These changes can reduce the search space considerably and lead to finding a solution faster.

The genetic algorithm by `aima-python` can only perform crossover between two lists. Consider the limitations of this approach in KNetWalk. Consider how crossover can be performed to take advantage of the structure of the board.

In KNetWalk, even a random state's fitness tends to be substantial compared to a goal state's. For example, in Figure 1, the randomly initialised state's fitness is 10, while the goal state's fitness is 20. This can lead to a substantial presence of poor-performing states in fitness-weighted samples, e.g., when performing the genetic algorithm. The fitness function can be modified to adjust the likelihood of a state being selection. For example, the `fitness_fn` parameter of `genetic_algorithm` can take a function that weights simple fitness quadratically or exponentially.