ICS 381: Principles of AI – 242
PA 1 – Programming Instructions

General Helpful Tips:

- You can submit your code on Gradescope any time before the submission deadline. The autograder will test your code and give you scores feedback on various test cases.
- **Do not copy others' work.** You can discuss general approaches with students, but do not share specific coding solutions.

- **NOTE1:** For this homework, we will be using numpy for random sampling. So, you can import numpy.
- **NOTE2:** You can import copy for the deepcopy function.
- **NOTE3**: You are not allowed to use any other external python packages; i.e. pandas, etc.
- **NOTE4:** Be sure to implement the functions as specified in this document. In addition, you can implement any extra helper functions as you see fit with whatever names you like.
- **NOTE5:** You do not need to adhere to good coding standards and style (although this is recommended for your own sake). The autograder just checks that your code gives the correct output (the logic is correct).
- **NOTE6:** Your code must work on the Gradescope autograder to achieve a score. Request for manual grading will not be entertained.
- **NOTE7: All** tests on Gradescope's autograder are expected to take around **140 seconds** to complete. If it takes longer than 4 minutes, then most likely there is an infinite loop bug in your code.

- **NOTE8:** Submit the required files only: [**search_problem.py, local_search.py**]

Searching

search_problem.py: Implement GridHunterProblem Class

Your task is to implement a variant of the grid problem. This grid problem consists of an N-by-N grid (N rows and N columns). A grid location is denoted by a tuple (row, col) with domains $\{0, 1, \ldots, N-1\}$. The agent starts at some grid location (row_A, col_A). The agent has **five actions available: turn-right, turn-left, move-forward, stay, and shoot-arrow**. Note that the agent cannot move outside of the grid limits defined by N.

In this grid problem, we have monsters that move in a pattern and destroy the agent if they come into contact (occupy the same grid). The goal is for the agent to shoot all the monsters.

Here is more info on the monsters:

- There are M monsters in total. The **initial** (row, col) starting location of these monsters is a list `monster_coords`.
- The monsters move in a fixed pattern. A monster moves left of its initial location, then back to its initial location, then right of its initial location, then back to its initial location, and so on. It keeps oscillating: initial, left, initial, right, initial, left, etc.
- In actions() implementation, we only remove the move-forward action in the case that it leads to outside of grid bounds.
- The agent has ***health*** represented as an integer. If the agent comes into contact with a monster (they occupy the same grid), then the agent takes 1 point of damage. If agent health <= 0, they cannot do any more actions.
- In the result() function, implement monster movement, agent actions, and agent damage logic.
- To further make the implementation memory efficient, we will use a simple trick to keep track of monster locations during the search. Since the monsters move with a fixed pattern, and we have stored their initial starting locations in `monster_coords`, then we can use an integer $mstep$ to keep track of which grid location the monsters should be in. When $mstep = 0$, the monsters are at their initial locations. When $mstep = 1$, then monsters are at the left of their initial locations. When $mstep = 2$, then monsters move back to initial locations. When $mstep = 3$, then monsters are the right of their initial locations. That is all we need to represent the fixed pattern. We store this timestep as part of the state, and each time `result` function is called, them $mstep = (mstep + 1)\%4$ that is modulo 4.
- When the agent shoots, the arrow moves forward from the agent's forward-direction. The arrow continues travelling until it hits a wall. Any monster in the arrow's path dies. Dead monsters can be ignored by the agent when moving (no damage).
- **The goal is for the agent to shoot all *M* monsters and survive**. We will maintain the status of the M monsters as Booleans in the state. The status of agent being alive is also a Boolean.

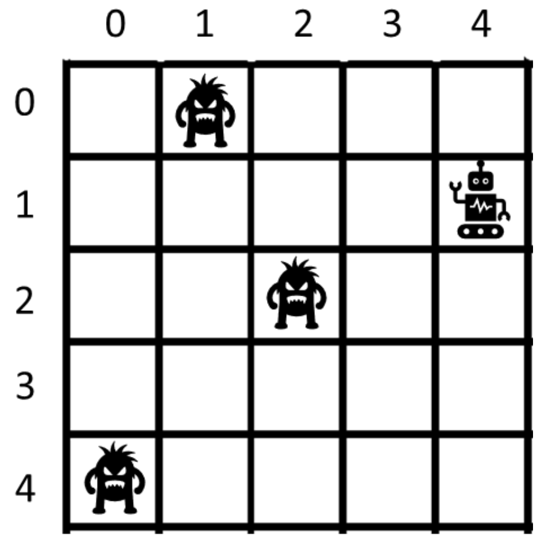
We will represent a state as a tuple $state = (row, col, forw, health, mstep, d_1, d_2, \dots, d_M)$, where:

- row, col is the current agent location. Both are in the domain {0, 1, …, N-1}.
- $forw$ ∈ {‘north, ‘south’, ‘east’, ‘west’} represents agent’s forward orientation. When the agent moves forward, it moves in the direction of its orientation. When the agent shoots an arrow, the arrow moves in direction of agent orientation.
- $health \in \{0, 1, 2, 3, \dots\}$ non-negative integer representing agent health.
- $mstep \in \{0, 1, 2, 3\}$ represents the monster timestep to keep track of their locations.
- $d_1, d_2, \dots, d_M$ are Booleans indicating if the corresponding monster in `monster_coords` is dead. Initially all false.

The goal condition is reached when all M monsters are dead and agent survives:

$$all(d_1, d_2, \dots, d_M) \wedge (health > 0)$$

To clarify the usage of GridHunterProblem, consider the following 5-by-5 grid instance, $M = 3$:

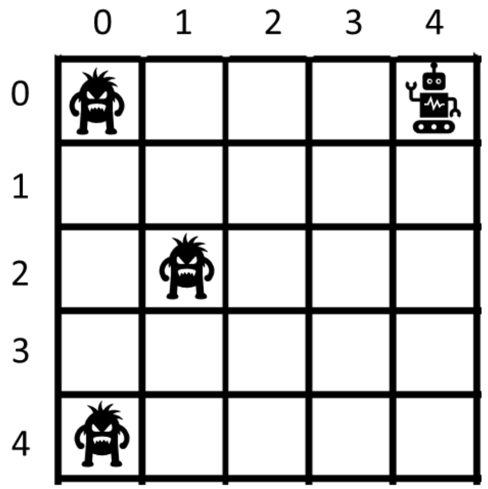


The agent is $(row, col, forw) = (1, 4, north, 10)$ where 10 is staring health. The monsters start at $[(0, 1), (2, 2), (4, 0)]$.
We can create this GridHunterProblem as follows:

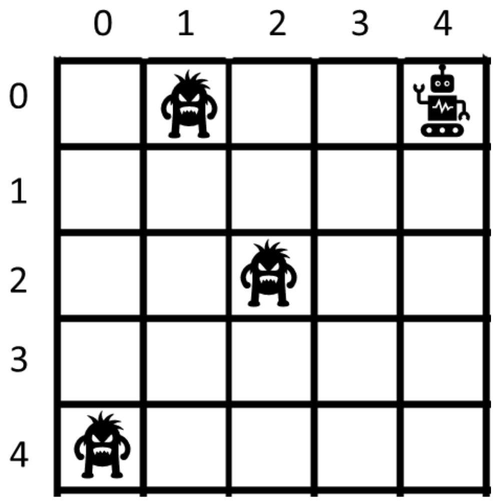
```
example_monster_coords = [(0,1), (2,2), (4,0)]

example_grid_problem = GridHunterProblems(initial_agent =(1,4, 'north', 10)
                                          N=5,
                                          monster_coords=example_monster_coords)
```

From the initial state above $state = (1, 4, north, 10, 0, False, False, False)$, let's say the agent moves-forward, then we will have the following where $mstep = 1$, so $state = (0, 4, north, 10, 1, False, False, False)$. This corresponds to the grid visualization:



Let's say the agent then does turn-left, then we get $mstep = 2$, so $state = (0,4, west, 10, 2, False, False, False)$



If the agent now shoots an arrow, then next state becomes $state = (0,4, west, 10, 3, True, False, False)$. **Note** that state Booleans and coordinate tuples in `monster_coords` are in same order (use the same index).

Using this information, now implement the GridHunterProblem class in search_problem.py

Class name	
`GridHunterProblem`	
GridHunterProblem Constructor	
Constructor arguments	Constructor body

`initial_agent_info,`
`N,`
`monster_coords`
	Add arguments to self.
	`initial_agent_info` is a tuple $(row, col, forw, health)$ indicating the agent's starting grid location, forward orientation, and starting health.
	Assume `N` is a positive integer for row and column-size of the grid problem.
	Assume `monster_coords` is a list of tuples (row, col) grid locations indicating the starting positions for the monsters.
	Appropriately set the initial state as a **tuple** described above
	`self.initial_state = initial_agent_info + (0, False, False, …, False)`

GridHunterProblem Functions			
Name	Arguments	Returns	implementation
`move_monsters`	`timestep`	Returns a new list of tuple (row, col) for the new monster locations based on timestep argument.	This is a helper function used inside actions() and result() functions.
`actions`	`state`	Returns a list of actions that can be reached by `state.` If no action is legal, then return an empty list [].	The agent should return all five actions except in two cases: - if move-forward action will result in out-of-grid bounds, then remove the move-forward action. - if health <= 0, then return empty list. **Note**: Format of the list of actions I used are strings 'move-forward, 'turn-left, 'turn-right, 'shoot-arrow', 'stay'. Stick to this format. Also, make sure to return the list of legal actions in the **order** move-forward, turn-left, turn-right, shoot-arrow, and stay.
`result`	`state,` `action`	Returns the transition-state that results from executing action.	Do the following when implementing: • Update $mstep$. Compute next monster movements. Be sure to block monsters from moving **outside of grid**. Use this updated monster coordinates when implementing the remaining below details. • If action is shoot-arrow, then update monster Booleans accordingly. The arrow starts in grid in front of the agent and keeps moving until it hits a wall, killing any monster in its path. • If action move-forward, update location of agent. • If action turn-left or turn-right, update agent orientation. • After taking care of the above, now process the **alive** monsters, and check if they contact the agent (occupy the same grid). If yes, then apply damage to agent health.

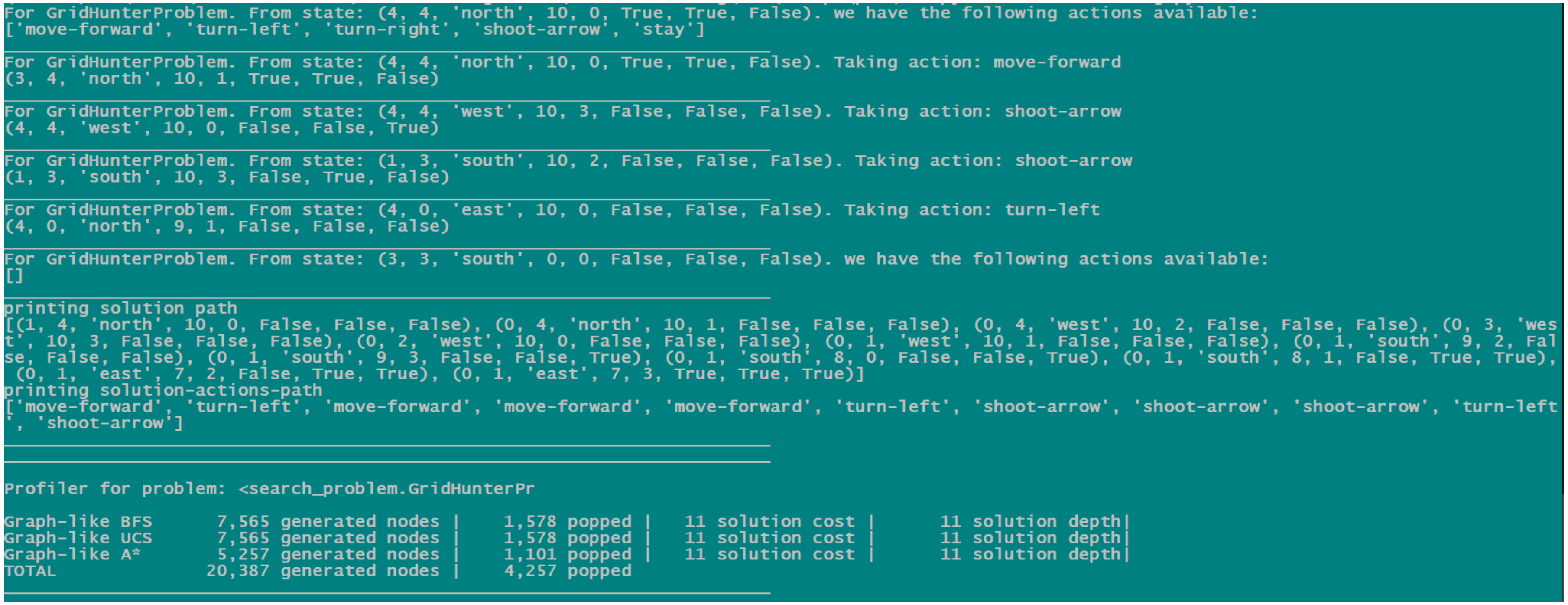
`action_cost`	`state1, action, state2`	Returns the cost of transition from state1 to state2 using action.	Any action costs 1.
`is_goal`	`state`	Returns true all monsters are dead **and** agent is alive.	Hint: recall that a state contains the monster dead statuses and agent health.
`h`	`node`	Returns the heuristic distance to the goal.	Implement a heuristic that computes the absolute row difference between the agent and **each alive monster,** and returns the minimum of these distances. **Note1**: for monster coordinates, look at current node.state's mstep coordinates. Do not use (mstep+1). **Note2**: make sure to return 0 if node is a goal-node. Note3: this given heuristic can be modified further to get a better one. Think on this more. However, do not submit a modified heuristic for the autograder.

Test your code on test_searching.py

You can test your code by running test_searching.py. You can inspect the code and add your own problem cases.

Note that you will need search_algorithms.py which we worked on in class. You can find a copy of it on blackboard, but you need to complete the implementation of the wrapper searches (dfs, bfs, etc.)

The following are my implementation results:



```
For GridHunterProblem. From state: (4, 4, 'north', 10, 0, True, True, False). we have the following actions available:
['move-forward', 'turn-left', 'turn-right', 'shoot-arrow', 'stay']
______________________________________________________________________
For GridHunterProblem. From state: (4, 4, 'north', 10, 0, True, True, False). Taking action: move-forward
(3, 4, 'north', 10, 1, True, True, False)
______________________________________________________________________
For GridHunterProblem. From state: (4, 4, 'west', 10, 3, False, False, False). Taking action: shoot-arrow
(4, 4, 'west', 10, 0, False, False, True)
______________________________________________________________________
For GridHunterProblem. From state: (1, 3, 'south', 10, 2, False, False, False). Taking action: shoot-arrow
(1, 3, 'south', 10, 3, False, True, False)
______________________________________________________________________
For GridHunterProblem. From state: (4, 0, 'east', 10, 0, False, False, False). Taking action: turn-left
(4, 0, 'north', 9, 1, False, False, False)
______________________________________________________________________
For GridHunterProblem. From state: (3, 3, 'south', 0, 0, False, False, False). we have the following actions available:
[]
______________________________________________________________________
printing solution path
[(1, 4, 'north', 10, 0, False, False, False), (0, 4, 'north', 10, 1, False, False, False), (0, 4, 'west', 10, 2, False, False, False), (0, 3, 'wes
t', 10, 3, False, False, False), (0, 2, 'west', 10, 0, False, False, False), (0, 1, 'west', 10, 1, False, False, False), (0, 1, 'south', 9, 2, Fal
se, False, False), (0, 1, 'south', 9, 3, False, False, True), (0, 1, 'south', 8, 0, False, False, True), (0, 1, 'south', 8, 1, False, True, True),
 (0, 1, 'east', 7, 2, False, True, True), (0, 1, 'east', 7, 3, True, True, True)]
printing solution-actions-path
['move-forward', 'turn-left', 'move-forward', 'move-forward', 'move-forward', 'turn-left', 'shoot-arrow', 'shoot-arrow', 'shoot-arrow', 'turn-left
', 'shoot-arrow']
______________________________________________________________________
______________________________________________________________________

Profiler for problem: <search_problem.GridHunterPr

Graph-like BFS      7,565 generated nodes |    1,578 popped |   11 solution cost |      11 solution depth|
Graph-like UCS      7,565 generated nodes |    1,578 popped |   11 solution cost |      11 solution depth|
Graph-like A*       5,257 generated nodes |    1,101 popped |   11 solution cost |      11 solution depth|
TOTAL              20,387 generated nodes |    4,257 popped
______________________________________________________________________
```

Note: if your code hangs/freezes at A*, you may want to check is_goal and h function implementations.

Implementing Local-Search

local_search.py: Implement functions for the Subset-Sum problem

Consider the subset-sum problem (SSP). We will work on a variant of it. You are given a set of non-negative integers S, and a sum integer T. The goal is to find a subset $B \subseteq S$ such that $sum(B)$ equals to T. This problem is in NP, so no known efficient algorithm. However, we can model this problem as a local search problem and use a local search algorithm to *try* to find a solution (or approximately close solution). We'll use simulated annealing.

When using a local search algorithm for this problem, the state should represent a subset that we can then sum over to check how good it is. One easy to do this is to model our state as a tuple of on/off bits (0 or 1) to indicate which elements of the set S is in the subset.

$$state = (e_1, e_2, \dots, e_{|S|})$$

Where $e_i \in \{0,1\}$ indicates if ith element of S is in the subset state.

As an example, consider $S = \{1, 54, 22, 43, 100\}$ and $T = 122$, then one possible state is $state = (0, 1, 1, 0, 1)$, this represents the subset $B = \{54, 22, 100\}$ with $sum(B) = 176$ (side note: solution is $state = (0, 0, 1, 0, 1)$).

How should we formulate our objective function? Well, we have T to act as a target. We want to favor subset states that are close to T as possible. So, we can use the following function for a subset B:

$$f(B) = abs(T - sum(B))$$

The goal subset B achieves $f(B) = 0$. We favor subsets that make $f(B)$ get closer and closer to 0. So, we are doing minimization.

From a current state, we need to consider how to generate neighbors. One simple heuristic is to make few modifications to the current state. The idea is to move a 'little' in the state-space to better states. How to design these few modifications is a design decision and largely depends on the problem itself. Here are some options:

- Consider generating all possible subsets of S, this is $O(2^{|S|})$ which is not feasible.
- From current subset state, consider generating a random neighbour via small modifications. For example, we can consider removing one item from the current subset state, or adding one item from S into the current subset state. This is what we'll do. Side-note: we can also consider generating K such random neighbours from a current subset state.

Before implementing simulated annealing to solve this problem, we should implement some helper functions. **Note: *S* is a numpy 1D array of integers (numpy makes indexing easier), *T* is an integer, and state is a python list with 0/1 values.**

Name	Arguments	Returns	Implementation hints/clarifications
`objective_f`	`state`, `S`, `T`	Returns the objective value $f(B)$ of `state`. S is the numpy 1D array of integers. T is the target sum.	Construct the subset B represented by `state`. You can do this easily as follows: `B = S[[idx for idx in range(len(S)) if state[idx] == 1]]` Then compute $f(B) = abs\big(T - sum(B)\big)$
`get_neighbor`	`state`	Generates one new random neighbor for state.	Implement this in the following **order**: • Make a deep copy of state called `n_state`. Do not modify `state` variable directly. Can be done as follows `n_state = copy.deepcopy(state)` • **Three Cases:** if no on-bits exist in state, then always add an item. If no off-bit exists in state, then always remove an item. It is easier to code this by having three if-elif-else statements. First case is when there are both on and off bits, this case you would do the u sampling described below. Second case is when there are on-bits but no off-bits, this is if sum(state) == len(state), this case you would remove an item. Last case is when there are off-bits, but no on-bits, then is if sum(state) == 0, this case you would add an item. **First case:** Sample `u = np.random.uniform()` If `u < 0.5` then uniformly at random remove an element from state. This can be done as follows: o Get the `indices` in state that have value 1.

			o `idx = np.random.choice(indices)` o `n_state[idx] = 0` Else uniformly at random add an element from state. This can be done as follows: o Get the `indices` in state that have value 0. o `idx = np.random.choice(indices)` o `n_state[idx] = 1` return `n_state` **Second case:** there are on-bits but no off-bits remove an item as described above. **Third case:** there are off-bits, but no on-bits add an item as described above.

Important Note: it is important that you random sample as described exactly. For example, only sample u if you are in the first case.

local_search.py: Implement simulated annealing

Your task is to implement a simulated annealing algorithm similar to what was described in slides of Local Search module.

Function Name: `simulated_annealing`
Arguments: `initial_state, S, T, initial_temp = 1000`
Returns: `final_state, iters`

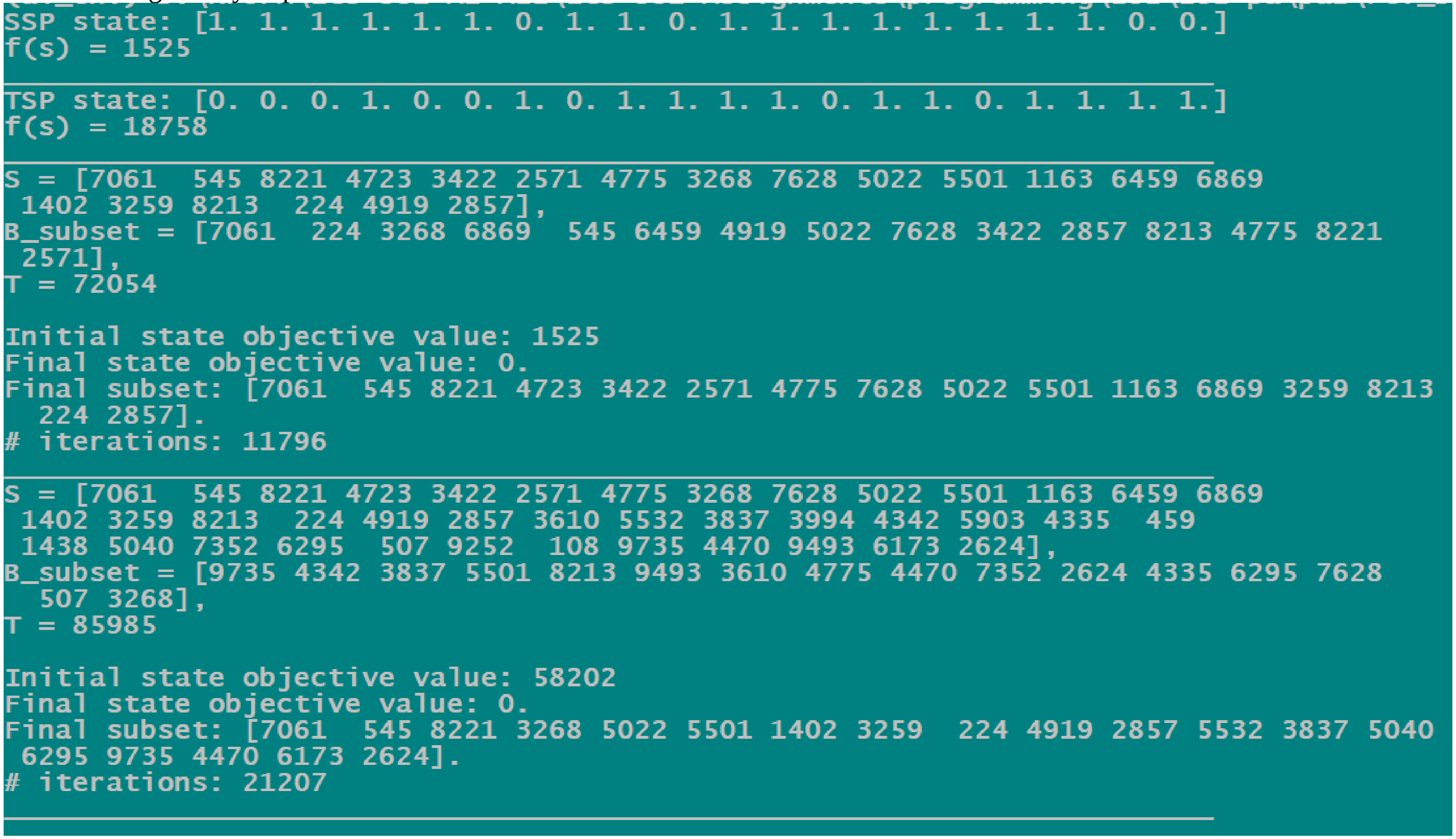
Implementation **in the following order (implement exactly in this order):**

1. The temperature variable is set to the given argument initial_temp.
2. The current state is set to the given argument initial_state.
3. Keep track of the number of iterations of the while loop, so set iters = 0
4. In the main while loop (code while header as `while temp >= 0:`):
   a. The scheduler update is $Temp = Temp * 0.9999$
   b. Add an if-check to stop looping when $Temp < 10^{-14}$ and return the current state and number of iterations. Note you can write 10^{-14} in python as `1e-14`
   c. Add an if-check to stop looping when `objective_f(current) == 0`, and return the current state and number of iterations.
   d. Generate a random successor of current using your `get_neighbor` implementation.
   e. Compute deltaE = `objective_f(current) - objective_f(next).`
   f. **IF $\boldsymbol{deltaE > 0}$** set current to the successor.
   g. **ELIF $\boldsymbol{deltaE \leq 0}$**, sample a number $u = np.random.uniform()$ and if $u \leq e^{detlaE/Temp}$, then set current to the successor (i.e. accept the bad move).
   h. **At end of loop be sure to increment iters += 1**
5. Outside of while loop, return current state and iteration count.

Test your code on test_local_search.py

You can test your local search code by running test_local_search.py. You can inspect the code and add your own problem cases.

The following are my implementation results



```
SSP state: [1. 1. 1. 1. 1. 1. 0. 1. 1. 0. 1. 1. 1. 1. 1. 1. 1. 1. 0. 0.]
f(s) = 1525
________________________________________________________________________
TSP state: [0. 0. 0. 1. 0. 0. 1. 0. 1. 1. 1. 1. 0. 1. 1. 0. 1. 1. 1. 1.]
f(s) = 18758
________________________________________________________________________
S = [7061  545 8221 4723 3422 2571 4775 3268 7628 5022 5501 1163 6459 6869
 1402 3259 8213  224 4919 2857],
B_subset = [7061  224 3268 6869  545 6459 4919 5022 7628 3422 2857 8213 4775 8221
 2571],
T = 72054

Initial state objective value: 1525
Final state objective value: 0.
Final subset: [7061  545 8221 4723 3422 2571 4775 7628 5022 5501 1163 6869 3259 8213
  224 2857].
# iterations: 11796
________________________________________________________________________
S = [7061  545 8221 4723 3422 2571 4775 3268 7628 5022 5501 1163 6459 6869
 1402 3259 8213  224 4919 2857 3610 5532 3837 3994 4342 5903 4335  459
 1438 5040 7352 6295  507 9252  108 9735 4470 9493 6173 2624],
B_subset = [9735 4342 3837 5501 8213 9493 3610 4775 4470 7352 2624 4335 6295 7628
  507 3268],
T = 85985

Initial state objective value: 58202
Final state objective value: 0.
Final subset: [7061  545 8221 3268 5022 5501 1402 3259  224 4919 2857 5532 3837 5040
 6295 9735 4470 6173 2624].
# iterations: 21207
________________________________________________________________________
```

I had to play around with the scheduler value till I found one that works well for these two instances (i.e. 0.9999).