King Fahd University of Petroleum & Minerals
College of Computing and Mathematics
Information and Computer Science Department
ICS 381: Introduction to AI – First Semester 2022-2023 (222)
PA 3 – Programming Instructions

General Helpful Tips:

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

local_search.py: Implement functions for the TSP problem

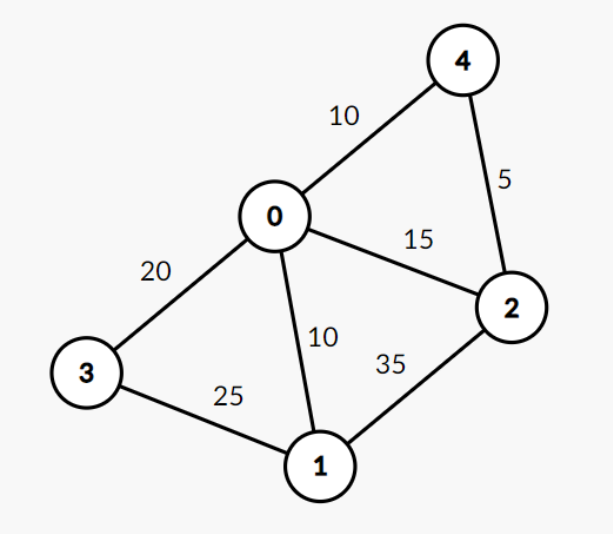
Consider the famous Travelling-Salesman-Problem (TSP). You are given a graph $G = (V, E)$ where V are vertices representing locations and E are undirected edges with costs. An agent wants to find the least-cost tour visiting every location in the graph exactly once. We can model this problem as a local search problem and use a local search algorithm to solve it.

When using a local search algorithm, the state should represent a complete solution to the given problem. When thinking of how transitions/neighbors are generated for a state, we need to limit how many neighbors are generated (otherwise we would generate the entire state-space as 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.

For this given problem, one possible representation for a state is to use a list of locations whose ordering indicates the path the agent should take in visiting the locations. Note that in this list all locations must be present and each location must occur exactly once. For the graph $G = (V, E)$, we will represent it as an adjacency matrix. For the objective function, we will use the tour cost (this is the state cost).

For example, consider the graph to the right. We represent with a **numpy** 2D array:

	0	1	2	3	4
0	np.nan	10	15	20	10
1	10	np.nan	35	25	np.nan
2	15	35	np.nan	np.nan	5
3	20	25	np.nan	np.nan	np.nan
4	10	np.nan	5	np.nan	np.nan



Notice that we use np.nan to indicate no edges. Our state is going to be a valid tour, so here are some possible states for this example:

- $state = [0, 4, 2, 1, 3]$ this tour has cost of $10 + 5 + 35 + 25 = 75$
- $state = [3, 1, 0, 2, 4]$ this tour has cost of $25 + 10 + 15 + 5 = 55$
- $state = [4, 3, 1, 2, 0]$ this non-valid tour has cost of $np.nan + 25 + 35 + 15 = np.nan$

Notice that cost of non-valid tours are $np.nan$, so we should block such states from being neighbors.

Note that for a graph with N locations, there are $N!$ possible tours. From the current state (a tour), how do we generate neighbor states? Since our state is a tour list, we will consider swapping two locations in the list; there are $N(N-1)$ possible swaps which is much less than the state-space $N!$. So, this neighbor generation method is $O(N^2)$ which is not bad. We can also limit this further by considering only $O(N)$ randomized swaps. We are going to use default simulated annealing whereby we select a random neighbor, so we can just sample a single random swap.

Before implementing simulated annealing to solve this problem, we should implement some helper functions. Assume that the N vertex locations are labelled as integers $0, 1, \ldots, N-1$. Assume that the adjacency matrix given is $N \times N$ indexed by the integer label of the locations.

Name	Arguments	Returns	Implementation hints/clarifications
`tour_cost`	`state, adj_matrix`	Returns the tour cost of the tour given in state.	This is a simple $O(N)$ loop that sums the tour cost using edge costs in adjacency matrix.
`random_swap`	`state`	Selects two random indices in state and swaps them.	`idx1, idx2 = np.random.choice(N, size=2, replace=False)` Samples two random index locations from 0 to N. You can then swap the locations. **NOTE1**: be sure to deepcopy state and then modify it. Return the modified deepcopy. Do not modify the state variable directly. **NOTE2:** Ignore checking for non-valid tours here. We will handle it in simulated annealing function.

local_search.py: Implement simulated annealing

Your task is to implement a simulated annealing algorithm similar to what was described in **slide 20 of Local Search module**. Your implementation will be used to solve the N-queens problem.

Function Name: `simulated_annealing`
Arguments: `initial_state, adj_matrix, initial_T = 1000`
Returns: `final_state, iters`

Implementation: Do the following steps:

1. The temperature T is set to the given argument initial_T.
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:
    a. The scheduler is $T = T * 0.99$
    b. Stop looping when $T < 10^{-14}$ and return the current state and number of iterations.
    c. Generate a random successor of current using your random_swap implementation.
    d. Compute deltaE = tour_cost(current) – tour_cost(next).
    e. **IF deltaE > 0** set current to the successor.
    f. **ELIF deltaE <= 0**, sample a number $u = np.random.uniform()$ and if $u \leq e^{detlaE/T}$, then set current to the successor (i.e. accept the bad move).
    g. At end of loop be sure to increment iters += 1

NOTE: our use of deltaE > 0 and deltaE <= 0 means that if deltaE = np.nan (which happens when our next is a non-valid tour) will not result in a transition to another state. In other words, if deltaE = np.nan, current_state will stay the same and in the next iteration a new swap will be sampled.

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:

```
TSP state: [4, 6, 16, 0, 19, 12, 7, 17, 10, 3, 18, 5, 9, 11, 13, 14, 2, 1, 15, 8]
f(s) = 940.0
_____________________________________________________________________________
TSP state: [3, 10, 6, 12, 2, 5, 8, 14, 18, 13, 19, 9, 17, 11, 16, 7, 4, 0, 1, 15]
f(s) = 920.0
_____________________________________________________________________________
Initial state objective value: 1043.0
Final state objective value: 403.0
# iterations: 3894
_____________________________________________________________________________
Initial state objective value: 5064.0
Final state objective value: 2099.0
# iterations: 4284
```

Note: for this local search implementation, when using random sampling functions like choice, be sure to call them exactly as specified; i.e. do not make two calls when the directions is to use one. The reason is because the autograder will control the random sampling using a random seed, this way you should get the same results.

csp.py: Implement AC3 algorithm

You are given an implementation of graph-coloring CSP problem in csp_graphcolor.csp. Check out the code to see its implementation. We will be using this graph-color implementation to test your AC3 and backtracking implementation. Also, note that we will implement other CSP problems and solve it using your AC3 and backtracking implementation.

Your first task is to implement the AC3 algorithm similar to Figure 5.3. Be sure to create a queue of arcs in both directions at the start of the algorithm.

Function Name: `ac3`
Arguments: `csp, arcs_queue=None, current_domains=None, assignment=None`
Returns: `is_consistent, updated_domains`; is_consistent is true if arc-consistency was enforced, false if it was not possible. updated_domains is the updated dictionary of variable domains after enforcing arc-consistency (the caller may want to use this).

Implementation hints:

- You can implement the revise function as a helper. Check for constraint violations using `csp.constraint_consistent`
- The arcs_queue is a queue of arcs which may be passed by the caller (when used with backtracking), or if set as None then you should create an arcs_queue of all the arcs in the csp **in both directions**. You can use a python **set** as the queue with pop, add, and remove operations. The items in the queue are just tuples of the form (var1, var2) indicating the arc var1→var2. You can assume that csp.adjacency is a dictionary of key, val where key is a variable and value is list of its neighbor variables (so you can use this to construct the queue of arcs).
- If the caller passes their own arcs_qeueu, then to be safe in case a list is passed wrap it in a set. That is, arcs_queue = set(arcs_queue)
- The current_domains argument is a dictionary where the key is a variable and the value is the domain list. If passed as None, then set current_domains of all variables to be the same csp.domains. **NOTE** you should make a deepcopy of csp.domains. For this, you can import copy and use deepcopy() function (see link).
- The assignment argument is the current partial assignment dictionary (variable: value). You will need this to make sure that when you add more arcs to the queue, you do not add variables that were already assigned (see CSP slides). With this, we can use AC3 in backtracking search that you will implement next.

- Be careful when dealing with the `csp` object, it is passed as reference and so you do not want to modify its instance variables; treat it strictly as read-only.

csp.py: Implement backtracking algorithm

Your task is to implement the backtracking algorithm similar to Figure 5.5; use it as a guide but your implementation will most likely deviate from it.

Function Name: `backtracking`
Arguments: `csp`
Returns: If succeeds, then returns a complete-consistent `assignment` (a dictionary where the key is a variable and the value is a color). Otherwise, if fails then returns None.

Implementation hints/clarifications:

- You can implement whatever helper functions you want. It may be helpful to implement `backtracking_helper(csp, assignment ={}, current_domains=None)`
- If you do a recursive implementation as in Figure 5.5, then you may want to keep track of the current_domains in each recursive call. You should also pass deep copies so that you do not modify earlier calls' domains. For this, you can import copy and use deepcopy() function (see link). There are more optimized ways of doing this, but no need to optimize in this assignment.
- After a variable is assigned a value, be sure to modify assignment dictionary, and check if the assignment is valid by calling csp.check_partial_assignment. Then you can do AC3 checking afterwards.
- For the variable and value ordering, you may use whatever implementation you want. For **most** problems, inference checking (AC3) helps much more than variables/value ordering. For problems we are implementing, you may want to use MRV + AC3.
- Since you implemented AC3, you may also want to use it for inference. This will speed up the search significantly. Be careful of what you pass to AC3 as you may make modifications in AC3 that will then modify them in the caller. This may be or may not be what you want. When in doubt, make deep copies.
- **If you do use AC3**, then make sure to do the following: restrict the domain of the variable you just assigned a value current_domains[var] = [val], get the unassigned neighbors of var, create an arcs_queue of just the unassigned neighbors of var. Pass this current_domains and arcs_queue to the ac3 call along with other necessary arguments.
- The autograder is setup to check the output of your implementation. It is expecting a dictionary (variable: value) in case there is a complete consistent solution, or None in case there is no possible solution. It is not looking for a specific

complete and consistent assignment; as long as your implementation returns a complete consistent assignment you'll get the grade.

- Some test cases will have a large number of variables and will have a time-limit to incentivize you to use AC3+variable-and-value-ordering in your implementation. For a graph-color problem with 36 variables, it can take more than 20 minutes if you do not use AC3, but less than 4 seconds if you do use AC3. You can pass the large test cases by using AC3+MRV.

Test your code on test_graphcolor.py

You can test your csp code by running test_graphcolor.py. You can inspect the code and add your own problem cases. The following are my implementation results:

```
False
True
True
_____________________________________________________________________
False
True
_____________________________________________________________________
False
True
_____________________________________________________________________
True {'SA': ['red', 'blue', 'green'], 'NT': ['red', 'blue', 'green'], 'V': ['red', 'blue', 'green'], 'Q': ['red', 'blue', 'green'], 'WA': ['red',
'blue', 'green'], 'NSW': ['red', 'blue', 'green'], 'T': ['red', 'blue', 'green']}
Testing AC3 with inconsistenct example:
False {'WA': ['red'], 'NT': ['blue'], 'Q': ['green'], 'NSW': ['red', 'blue'], 'V': ['red', 'green', 'blue'], 'SA': [], 'T': ['red', 'green', 'blue
']}
_____________________________________________________________________
Sol: {'SA': 'green', 'NT': 'blue', 'NSW': 'blue', 'V': 'red', 'Q': 'red', 'WA': 'red', 'T': 'green'}
Is sol complete and consistent: True
Time taken: 0.015624523162841797 sec
_____________________________________________________________________
```

Note: in my implementation of AC3, the queue was implemented as a python set. Because python sets are unordered, this can result in different ordering of pop, add, remove operations. Furthermore, I used variable and value ordering with tie-breaking. So, you can get different complete and consistent assignments which is okay; e.g. var1 is assigned green in one correct assignment and in the other var1 is assigned red. The autograder will only check if your assignment is complete and consistent.

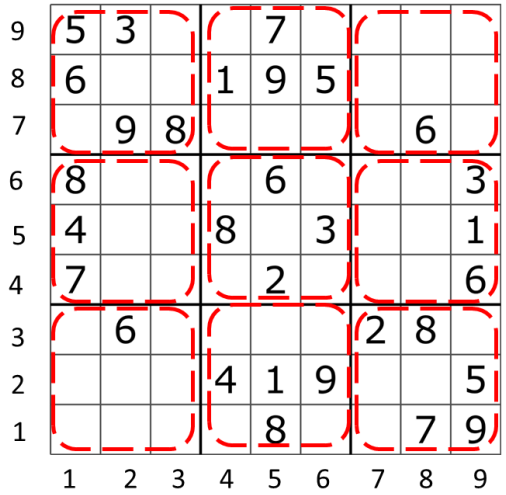
To test your code on large graphs, I've added test_large_graphcolor.py. This code will load in large graph-color objects using pickle. It will then run backtracking. Below is the output of my implementations (uses AC3+MRV+LCV). You may get slightly different runtimes due to machine-spec differences:

```
Is sol complete and consistent: True
Time taken: 0.788285493850708 sec
_____________________________________________________________________
Is sol complete and consistent: True
Time taken: 0.7933402061462402 sec
_____________________________________________________________________
Is sol complete and consistent: True
Time taken: 0.7446434497833252 sec
_____________________________________________________________________
Is sol complete and consistent: True
Time taken: 0.7859344482421875 sec
_____________________________________________________________________
Is sol complete and consistent: True
Time taken: 0.784935474395752 sec
_____________________________________________________________________
```

csp.py: Implement class for SudokuCSP

Recall the Sudoku CSP problem from class. You are given a 9×9 grid where each grid can be assigned values from $\{1, 2, \ldots, 9\}$. So, in total we have 81 variables. For some variables, they may already be assigned. The goal is complete the grid assignments to satisfy the following constraints:

- Row constraints: for all rows from 1 to 9, all variables on a row must be different values.
- Column constraints: for all columns from 1 to 9, all variables on a column must be different values.
- 3×3 square constraint: for all 3×3 squares, all variables in the square must be different values. Note in total there are 9 squares Sudoku (you can see them highlighted in red below)



Most of your implementation logic will be in the constructor to setup self.variables, self.domains, and self.adjacency. To denote a variable, we will use a tuple (row, col) where row and col range from 1 to 9. self.adjacency is a dictionary where the key is a variable and the value is a list of neighbor variables indicating binary constraints. For example, for variable (1,1) we have:

```
self.adjacency[(1,1)] = [(2,1), (3,1), (4,1), (5,1), (6,1), (7,1), (8,1), (9,1),
                         (1,2), (1,3), (1,4), (1,5), (1,6), (1,7), (1,8), (1,9),
                         (2,2), (3,2), (2,3), (3,3)]
```

Class name	
`SudokuCSP`	
SudokuCSP Constructor	
Constructor arguments	Constructor body
`partial_assignment={}`	`partial_assignment` is a dictionary of variable: value assignments where some of the grids are assigned values. Note that this can be empty dictionary; meaning an empty Sudoku puzzle. Setup the following: `self.variables` is a **list of tuples** (row, col) for the 81 variables. This can be done using a double loop on range(1, 10). `self.domains` is a dictionary of variable: list of domain values for the 81 variables. The key should be the variable tuple (row, col) and value is domain [1, 2, …, 9]. For the variables in partial_assignment, you should restrict their domains to what is assigned: `self.domains[var] =[partial_assignment[var]]`. `self.adjacency` is a dictionary where the key is a variable and the value is a list of neighbor variable tuples indicating binary constraints. You can do this by looping on each variable (row, col) and doing the following: • Add row constraints by enumerating (row, j) for $j \neq col$. • Add column constraints by enumerating (i, col) for $i \neq row$. • Add square constraints based (row, col) position. Be careful not to duplicate neighbors in self.adjcacency[var] when adding square constraints.

SudokuCSP Functions			
Name	Arguments	Returns	Implementation hints/clarifications
`constraint_consistent`	`var1, val1, var2, val2`	Returns true if var1 and var2 are neighbors and val1 and val2 are the **NOT** the same.	We want to implement the difference constraint for neighbors. Use self.adjacency to determine neighboring. If var1 and var2 are **not** neighbors, just return True. If var1 and var2 are neighbors, and val1 and val2 are not the same return True (constraint satisfied), otherwise false.
`check_partial_assignment`	`assignment`	Returns true if the partial assignment is consistent.	`assignment` is a dictionary where the key is a variable and the value is a value. If assignment is None, then return False. Just check if the assignment does not violate a constraint. Treat a complete assignment as a partial assignment.
`is_goal`	`assignment`	Returns true if assignment is **complete** and **consistent**. Otherwise, false.	`assignment` is a dictionary where the key is a variable and the value is a value. If assignment is None, then return False. You need to check if assignment is complete. If it is **complete**, check that it is **consistent**.

Test your code on test_sudoku.py

You can test your Sudoku code by running test_sudoku.py. You can inspect the code and add your own problem cases. The following are my implementation results:

```
81
_____________________________________________________________________________
[(1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (1, 6), (1, 7), (1, 8), (1, 9), (2, 1), (2, 2), (2, 3), (2, 4), (2, 5), (2, 6), (2, 7), (2, 8), (2, 9), (
3, 1), (3, 2), (3, 3), (3, 4), (3, 5), (3, 6), (3, 7), (3, 8), (3, 9), (4, 1), (4, 2), (4, 3), (4, 4), (4, 5), (4, 6), (4, 7), (4, 8), (4, 9), (5,
 1), (5, 2), (5, 3), (5, 4), (5, 5), (5, 6), (5, 7), (5, 8), (5, 9), (6, 1), (6, 2), (6, 3), (6, 4), (6, 5), (6, 6), (6, 7), (6, 8), (6, 9), (7, 1
), (7, 2), (7, 3), (7, 4), (7, 5), (7, 6), (7, 7), (7, 8), (7, 9), (8, 1), (8, 2), (8, 3), (8, 4), (8, 5), (8, 6), (8, 7), (8, 8), (8, 9), (9, 1),
 (9, 2), (9, 3), (9, 4), (9, 5), (9, 6), (9, 7), (9, 8), (9, 9)]
_____________________________________________________________________________
[4]
_____________________________________________________________________________
[1, 2, 3, 4, 5, 6, 7, 8, 9]
_____________________________________________________________________________
[8]
_____________________________________________________________________________
[(1, 2), (1, 3), (1, 4), (1, 5), (1, 6), (1, 7), (1, 8), (1, 9), (2, 1), (3, 1), (4, 1), (5, 1), (6, 1), (7, 1), (8, 1), (9, 1), (2, 2), (2, 3), (
3, 2), (3, 3)]
_____________________________________________________________________________
[(6, 1), (6, 3), (6, 4), (6, 5), (6, 6), (6, 7), (6, 8), (6, 9), (1, 2), (2, 2), (3, 2), (4, 2), (5, 2), (7, 2), (8, 2), (9, 2), (4, 1), (4, 3), (
5, 1), (5, 3)]
_____________________________________________________________________________
[(3, 1), (3, 2), (3, 3), (3, 4), (3, 5), (3, 6), (3, 7), (3, 8), (1, 9), (2, 9), (4, 9), (5, 9), (6, 9), (7, 9), (8, 9), (9, 9), (1, 7), (1, 8), (
2, 7), (2, 8)]
_____________________________________________________________________________
Sol: {(1, 1): 4, (3, 6): 5, (4, 8): 5, (5, 2): 7, (6, 4): 4, (7, 3): 8, (8, 7): 6, (9, 5): 8, (1, 9): 6, (2, 1): 7, (5, 4): 9, (6, 2): 1, (7, 7):
7, (9, 3): 5, (1, 8): 3, (8, 9): 3, (2, 3): 9, (4, 6): 2, (5, 8): 6, (1, 5): 7, (3, 7): 2, (7, 4): 3, (9, 2): 3, (4, 1): 3, (5, 6): 3, (6, 9): 2,
(8, 3): 7, (1, 7): 8, (1, 6): 9, (3, 3): 3, (2, 5): 3, (3, 5): 4, (5, 1): 5, (9, 9): 9, (6, 8): 7, (4, 2): 8, (7, 5): 9, (4, 4): 7, (2, 2): 2, (7,
 6): 1, (8, 8): 8, (2, 7): 5, (3, 1): 8, (9, 4): 6, (4, 9): 1, (6, 1): 9, (8, 2): 9, (2, 6): 6, (5, 9): 8, (9, 7): 1, (3, 8): 9, (6, 6): 8, (7, 2)
: 4, (8, 4): 5, (1, 3): 1, (8, 5): 2, (2, 8): 1, (7, 9): 5, (3, 4): 1, (4, 3): 4, (5, 5): 1, (6, 7): 3, (9, 1): 2, (1, 2): 5, (2, 4): 8, (3, 9): 7
, (4, 5): 6, (5, 3): 2, (7, 1): 6, (8, 6): 4, (9, 8): 4, (4, 7): 9, (6, 3): 6, (1, 4): 2, (2, 9): 4, (3, 2): 6, (5, 7): 4, (6, 5): 5, (7, 8): 2, (
8, 1): 1, (9, 6): 7}
Is sol complete and consistent: True
Time taken: 0.20023179054260254 sec
_____________________________________________________________________________
```

[Optional Bonus] csp.py: Visualizing Sudoku solution

Your task is to implement a helper function in csp.py to help us visualize the SudokuCSP solution. We will use the **matplotlib** and **seaborn** packages to ease this implementation, so add these imports to local_search.py as follows:

```
import matplotlib.pyplot as plt
import seaborn as sns
```

First let us define a function that will visualize the solution of a RouteProblem instance.

Name: `visualize_sudoku_solution`
Arguments: `assignment_solution, file_name`
Returns: does not return anything.

Implementation: assignment_solution argument is the dictionary (row, col): value solution for the 81 Sudoku variables. First, you want to construct a 2D integer array of size 9×9 representing the Sudoku puzzle with the value assignments. Implement this as an 9×9 list of lists. For example, here is the array for solution to a Sudoku puzzle:

sudoku_array = [[5, 3, 4, 6, 7, 8, 9, 1, 2],
[6, 7, 2, 1, 9, 5, 3, 4, 8],
[1, 9, 8, 3, 4, 2, 5, 6, 7],
[8, 5, 9, 7, 6, 1, 4, 2, 3],
[4, 2, 6, 8, 5, 3, 7, 9, 1],
[7, 1, 3, 9, 2, 4, 8, 5, 6],
[9, 6, 1, 5, 3, 7, 2, 8, 4],
[2, 8, 7, 4, 1, 9, 6, 3, 5],
[3, 4, 5, 2, 8, 6, 1, 7, 9]]

Now to plot this solution, do the following.

- Define the figure size by calling `plt.figure(figsize=(9, 9))`.
- Draw the solution as a seaborn heatmap: set the data argument to your 2D array, set annot to True, set linewidths to 1.5 and linecolor to ‘k’, and set cbar to False.

- Invert the y-axis (see link).
- Finally, save the resulting figure as a png using savefig. Also, make sure at the end to call plt.close().

Hint: setting the figure size, creating seaborn heatmap, inverting the y-axis, saving it to file, and closing it should take just five lines of code. If you run visualize_sudoku.py on your code, you can check your resulting images against the reference images: [ref_sudoku1.png, ref_sudoku2.png]. You should reproduce the same image.