King Fahd University of Petroleum & Minerals
College of Computing and Mathematics
Information and Computer Science Department

ICS 381: Principles of AI – First Semester 2022-2023 (221)

PA 3 – Programming Instructions

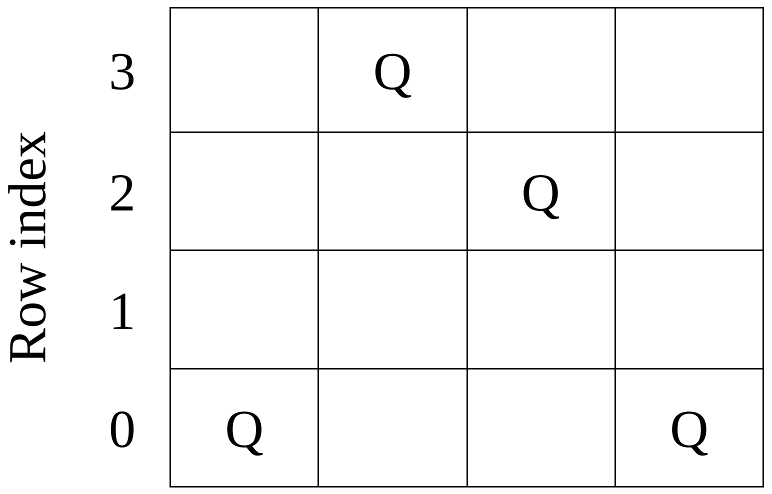
General Helpful Tips:

- 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 N-queen problem

Recall that the N-queen problem is the task of placing N queens on an NxN grid such that no two queens attack each other. A queen can attack other pieces on the same row, same column, left-to-right diagonal, and right-to-left diagonal.

Your task is to implement functions for the N-Queen problem that uses a **complete formulation**. We will use the concise formulation of forcing the N-queens on the N columns of the grid. Thus, a state can be represented by a **tuple**. For example for 4-queens, s = (0,3,2,0) represents the state (note the usage of 0-indexing) of positioning the first queen in the first column at row 0, the second queen in the second column at row 3, the third queen in the third column at row 2, and the fourth queen in the fourth column at row 0. Note that this state is not a goal state since there are still queens that are attacking each other.



We will solve the N-queens problem using simulated annealing algorithm. But first, implement the following helper functions.

Name	Arguments	Returns	Implementation hints/clarifications
`objective_function`	`state`	Returns the objective function value of a state. In N-queen, count the number of **attacking** queens for a state. A low objective value value indicates a state that is closer to the goal.	Be sure not to overcount; i.e. the same two queens attacking each other should be counted as 1 attack.
`is_goal`	`state`	Returns true if the given state is the goal solution; i.e. no two queens attack each other. Otherwise, returns false.	Hint: check if objective_function(state) is 0 Return true or false.

Hint1: because of our formulation, no need to check for column attacks. So just need to check row and diagonal attacks

Hint2: To check diagonal attacks for two queens A and B with positions (row_A, col_A) and (row_B, col_B), respectively, you can simply check if the absolute value difference of their rows is equal to the absolute value difference of their columns.

$$abs(row_A - row_B) == abs(col_A - col_B)$$

The nice thing about this check is that it accounts for both diagonal directions.

local_search.py: Implement simulated annealing to solve N-queens

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, initial_T = 1000`
Returns: `current_stat, iters`

Implementation: We will follow the pseudocode from slide 20 of local search module. 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 $T = T * 0.95$
   b. Stop looping when $T < 1e - 14$ and return the current state and number of iterations.
   c. Generate a random successor of current. (see details below)
   d. Compute deltaE = objective_function(current) - objective_function(next).
   e. If deltaE > 0 set current to the successor.
   f. Else, 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

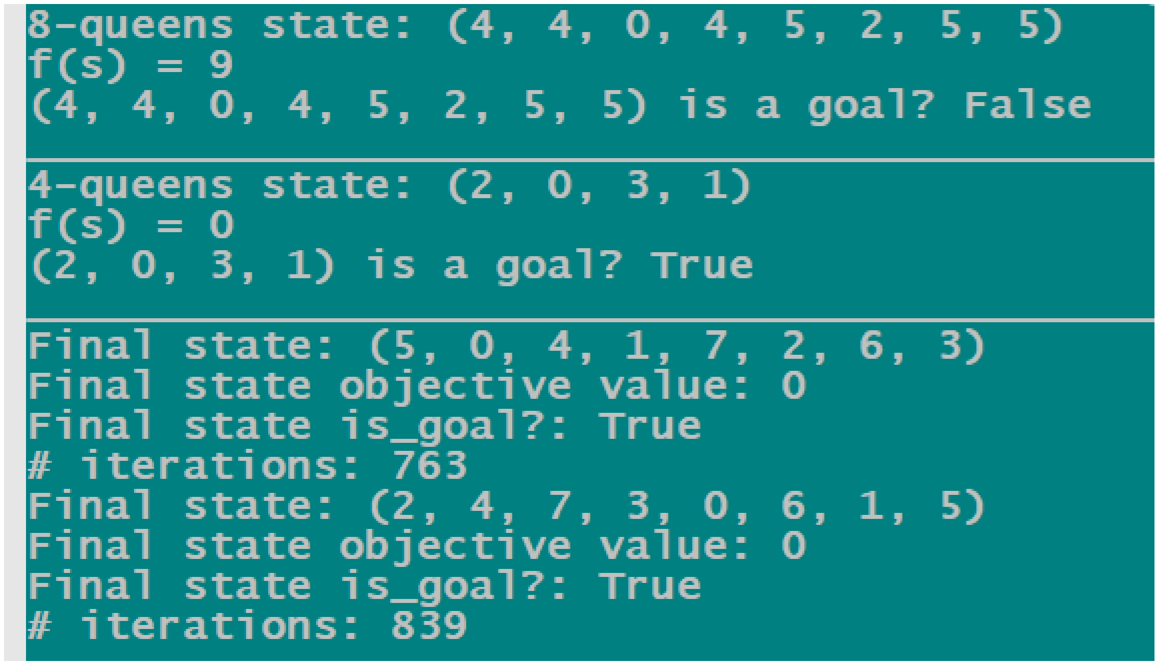
To generate a random successor of current, do the following:

- Uniformly random select a queen to modify. You can do this using `rnd_queen = np.random.choice(range(N))` which returns a random number between 0 and N-1.
- Now that you have identified the queen to modify, you want to randomly select a value for that queen. You can do this using `rnd_val = np.random.choice([i for i in range(N) if i != current[rnd_queen]])`
- The new successor will be the same as current but with the queen at column `rnd_queen` changed to `rnd_val`.

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:



```
8-queens state: (4, 4, 0, 4, 5, 2, 5, 5)
f(s) = 9
(4, 4, 0, 4, 5, 2, 5, 5) is a goal? False
```

```
4-queens state: (2, 0, 3, 1)
f(s) = 0
(2, 0, 3, 1) is a goal? True
```

```
Final state: (5, 0, 4, 1, 7, 2, 6, 3)
Final state objective value: 0
Final state is_goal?: True
# iterations: 763
Final state: (2, 4, 7, 3, 0, 6, 1, 5)
Final state objective value: 0
Final state is_goal?: True
# iterations: 839
```

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 class for GraphColorCSP

Recall the graph coloring CSP problem from class. We want to assign colors to vertices with colors from a domain such that no two connected vertices have the same color. First, let's implement a class for this graph coloring problem.

Class name	Inherits from
`GraphColorCSP`	`object`

GraphColorCSP Constructor	
Constructor arguments	Constructor body
`variables`, `colors`, `adjacency`	Add arguments to self. `variables` is a **set** of strings denoting the name of variables (vertices). `colors` is a **set** of strings denoting the name of the colors. Using colors, add the dictionary self.domains. For each variable `var`, set `self.domains[var] = [c for c in colors]` `adjacency` is a dictionary where the key is a vertex and the value is a list of neighbors; example {var1: [var2,var3], var2: [var1], var3: [var1]} indicating the constraints between variables.

GraphColorCSP Functions			
Name	Arguments	Returns	Implementation hints/clarifications
`constraint_consistent`	`var1, color1, var2, color2`	Returns true if var1 and var2 are neighbors and color1 and color2 are the **NOT** the same.	We want to implement the difference constraint in graph coloring. If var1 and var2 are **not** neighbors, just return True. If var1 and var2 are neighbors, and color1 and color2 are not the same return True (constraint satisfied), otherwise false.
`is_goal`	`assignment`	Returns true if assignment is **complete** and **consistent**. Otherwise, false.	`assignment` is a dictionary where the key is a vertex and the value is a color. You need to check if assignment is complete. If it is **complete**, check that it is **consistent**.
`check_partial_assignment`	`assignment`	Returns true if the partial assignment is consistent.	Just check if the assignment does not violate a constraint. Treat a complete assignment as a partial assignment.

csp.py: Implement AC3 algorithm

Your 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.
- 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.

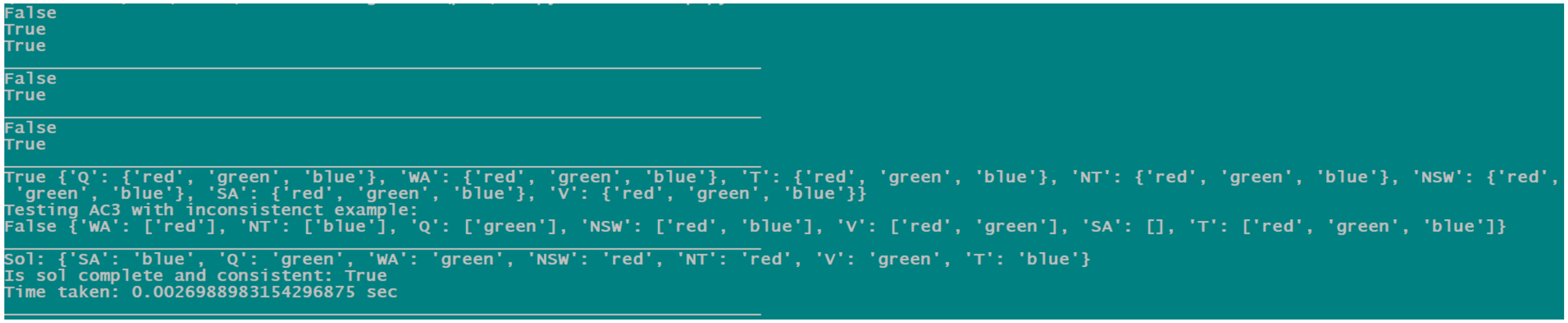
Function Name: `backtracking`
Arguments: `csp`
Returns: If succeeds, then returns a complete-consistent `assignment` (a dictionary where the key is a variable and the value is a color). Otherwise, if fails then returns None.

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.
- For the variable and value ordering, you may use whatever implementation you want. For **most** problems, inference checking (forward checking or AC3) helps much more than variables/value ordering. However, for a **general** graph-coloring problem you also want to use MRV + LCV heuristics to *guarantee* speedup.
- 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.
- 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.

Test your code on test_csp.py

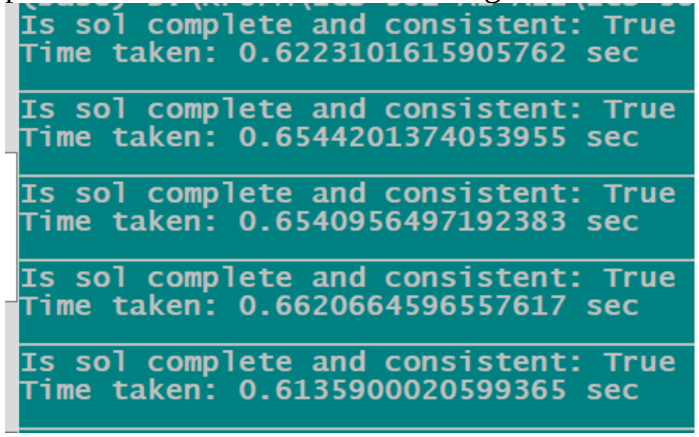
You can test your csp code by running test_csp.py. You can inspect the code and add your own problem cases. The following are my implementation results:



```
False
True
True
______________________________________________________________________
False
True
______________________________________________________________________
False
True
______________________________________________________________________
True {'Q': {'red', 'green', 'blue'}, 'WA': {'red', 'green', 'blue'}, 'T': {'red', 'green', 'blue'}, 'NT': {'red', 'green', 'blue'}, 'NSW': {'red',
 'green', 'blue'}, 'SA': {'red', 'green', 'blue'}, 'V': {'red', 'green', 'blue'}}
Testing AC3 with inconsistenct example:
False {'WA': ['red'], 'NT': ['blue'], 'Q': ['green'], 'NSW': ['red', 'blue'], 'V': ['red', 'green'], 'SA': [], 'T': ['red', 'green', 'blue']}
______________________________________________________________________
Sol: {'SA': 'blue', 'Q': 'green', 'WA': 'green', 'NSW': 'red', 'NT': 'red', 'V': 'green', 'T': 'blue'}
Is sol complete and consistent: True
Time taken: 0.0026988983154296875 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_csp_large_graph.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):



```
Is sol complete and consistent: True
Time taken: 0.6223101615905762 sec
__________________________________
Is sol complete and consistent: True
Time taken: 0.6544201374053955 sec
__________________________________
Is sol complete and consistent: True
Time taken: 0.6540956497192383 sec
__________________________________
Is sol complete and consistent: True
Time taken: 0.6620664596557617 sec
__________________________________
Is sol complete and consistent: True
Time taken: 0.6135900020599365 sec
__________________________________
```

[Optional Bonus] local_search.py: Visualizing N-queens solution

Your task is to implement a helper function in local_search.py to help us visualize the N-queens 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_nqueens_solution`
Arguments: `n_queens, file_name`
Returns: does not return anything.

Implementation: n_queens argument is just an n-queens tuple indicating the row position of the queens. First, you want to construct a 2D integer array of size NxN with 1 where the queens are and 0 otherwise. Implement this as an NxN list of lists. For example, if n_queens = (0,1,1,3), then this NxN array will be:

```
nqueens_array = [[1, 0, 0, 0],
                 [0, 1, 1, 0],
                 [0, 0, 0, 0],
                 [0, 0, 0, 1]]
```

Notice that the board is viewed from top-to-bottom; this makes implementation easier.

- Define the figure size by calling `plt.figure(figsize=(N, N)).`
- Draw the n-queens as a seaborn heatmap: set the data argument to your 2D array, set cmap argument to 'Purples', 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().

Your code should work for any N. 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_runner.py on your code, you can check your resulting images against the reference images: [nqueens1.png, nqueens2.png]. You should reproduce the same image.