CS143: Artificial Intelligence

Project Assignment 3
Local search variants (Random Restart, Local Beam Search, Simulated Annealing, Genetic Algorithm)

Assignment 3

Assignment 3: A* Search

Use priority queue for FRINGE

1. if GOAL == (initial-state) then return initial-state
2. INSERT(initial-node, FRINGE)
3. repeat:
4. if empty(FRINGE) then return failure
5. s ← REMOVE(FRINGE)
6. if GOAL ==s then return s and/or path
7. for every state s' in Successor(s):
8. INSERT(s', FRINGE)

REMOVE node s with minimum value of f()

Compute f(s') then INSERT s' into FRINGE
f(s') = g(s')+ h(s')

Assignment 3

- You need modify the followings:
  - Finish the methods `set_g_value()`, `set_h_value()`, `set_f_value()` inside `SSWSearch` class
  - Finish the method `a_star_search()`
    - Most of the content are very similar to the `ucs()`
  - Finish the method `greedy_best_first_search()`

Computing h() for 8-puzzle

- For A* search, we need to compute h(s) for a state s before inserting it into the FRINGE
- Let's consider: h(s) = number of misplaced tiles (not including empty tile)

state **s**

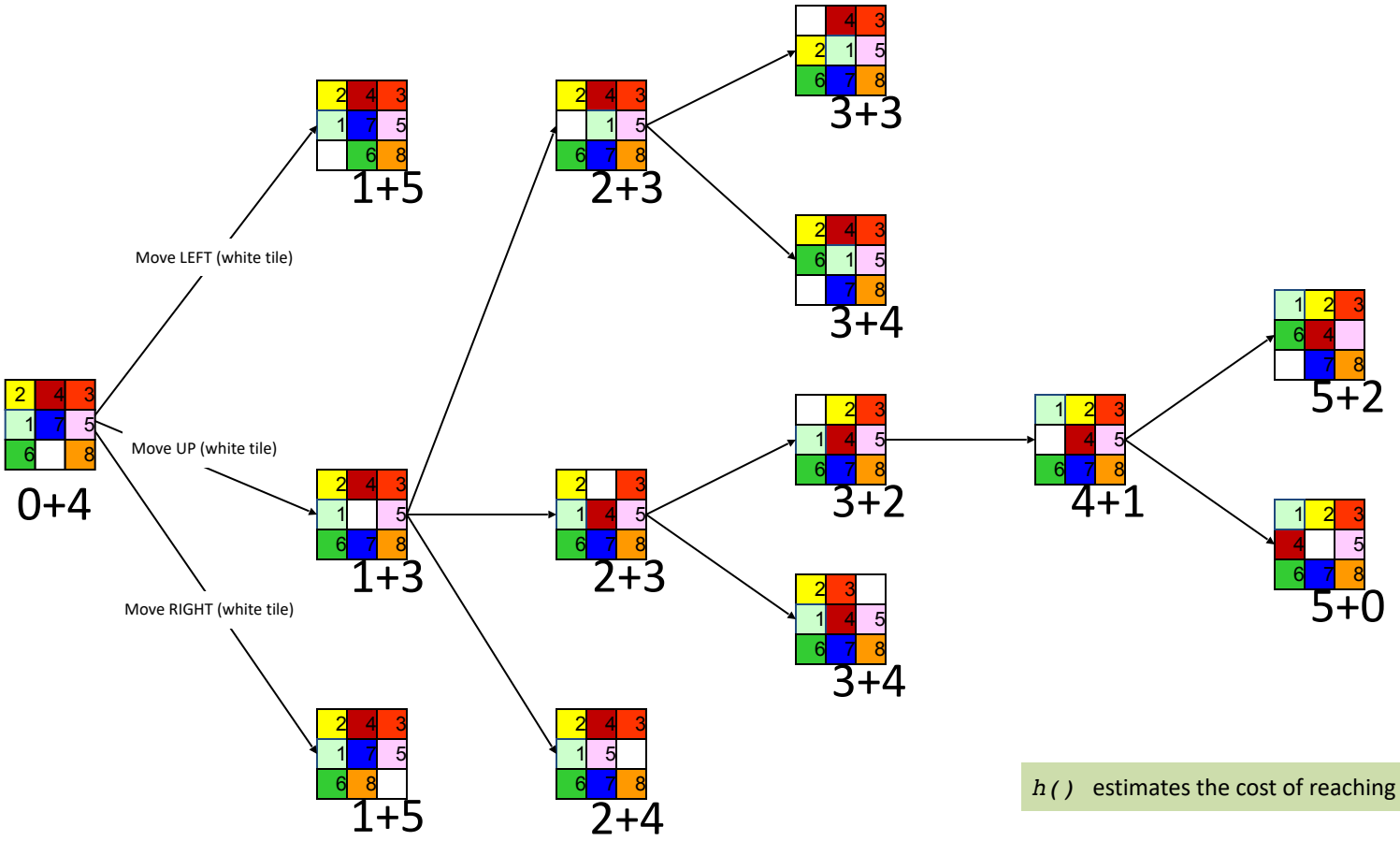
Goal state

Is tile#1 misplaced? =	1
Is tile#2 misplaced? =	1
Is tile#3 misplaced? =	0
Is tile#4 misplaced? =	1
Is tile#5 misplaced? =	0
Is tile#6 misplaced? =	0
Is tile#7 misplaced? =	1
Is tile#8 misplaced? =	0
h(s) = total # of misplaced tiles =	4

h( ) estimates the cost of reaching a goal from s

A* Search with misplaced tiles heuristic function

f(s) = g(s) + h(s)
with h(s) = number of misplaced numbered tiles

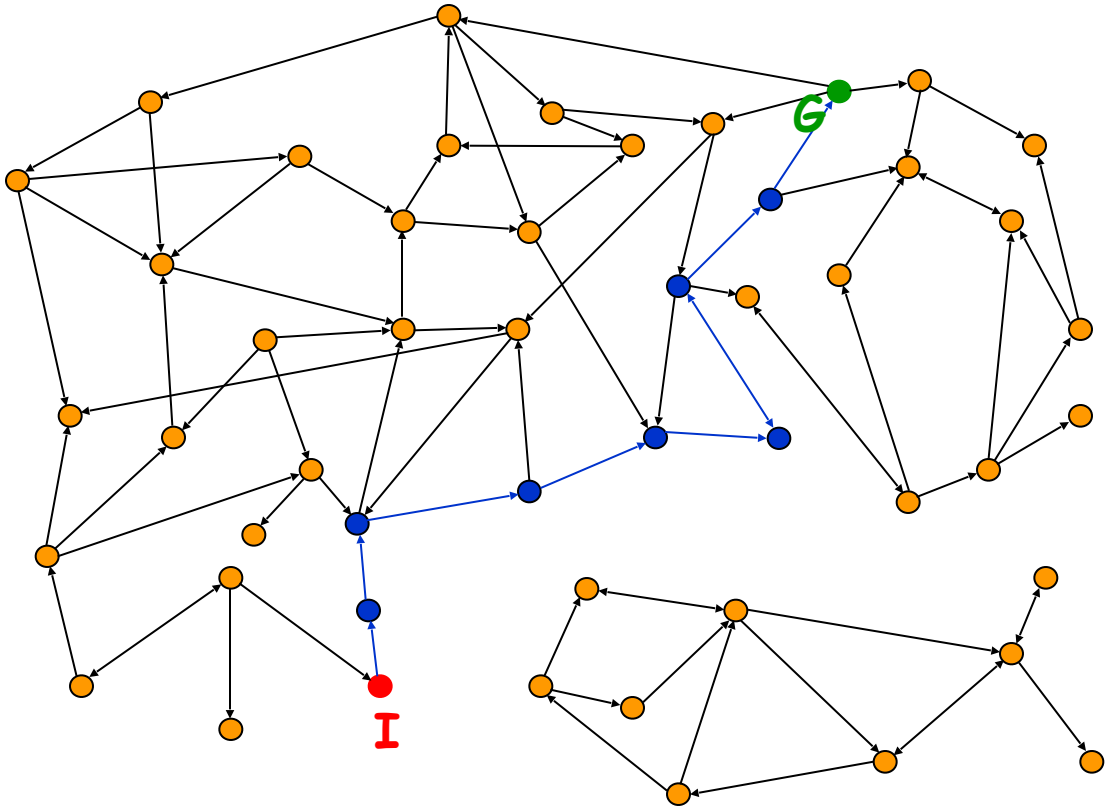


h( ) estimates the cost of reaching a goal from s

Local Search

Local Search for Pathless Problems

- Sometimes the path doesn't matter
- A solution is any goal node
- Edges represent potential state transformations
  - 8-queens, Map coloring



Review: Local Search

- Local search algorithms operate using a single, current node (rather than multiple paths).

- Local search is not systematic (i.e., it does not search all possibilities).
  - It may not identify the "best" or "optimal" solution.

- Advantages of local search
  - Use little memory
  - Can often find reasonable solutions in large spaces (where classical search fails)

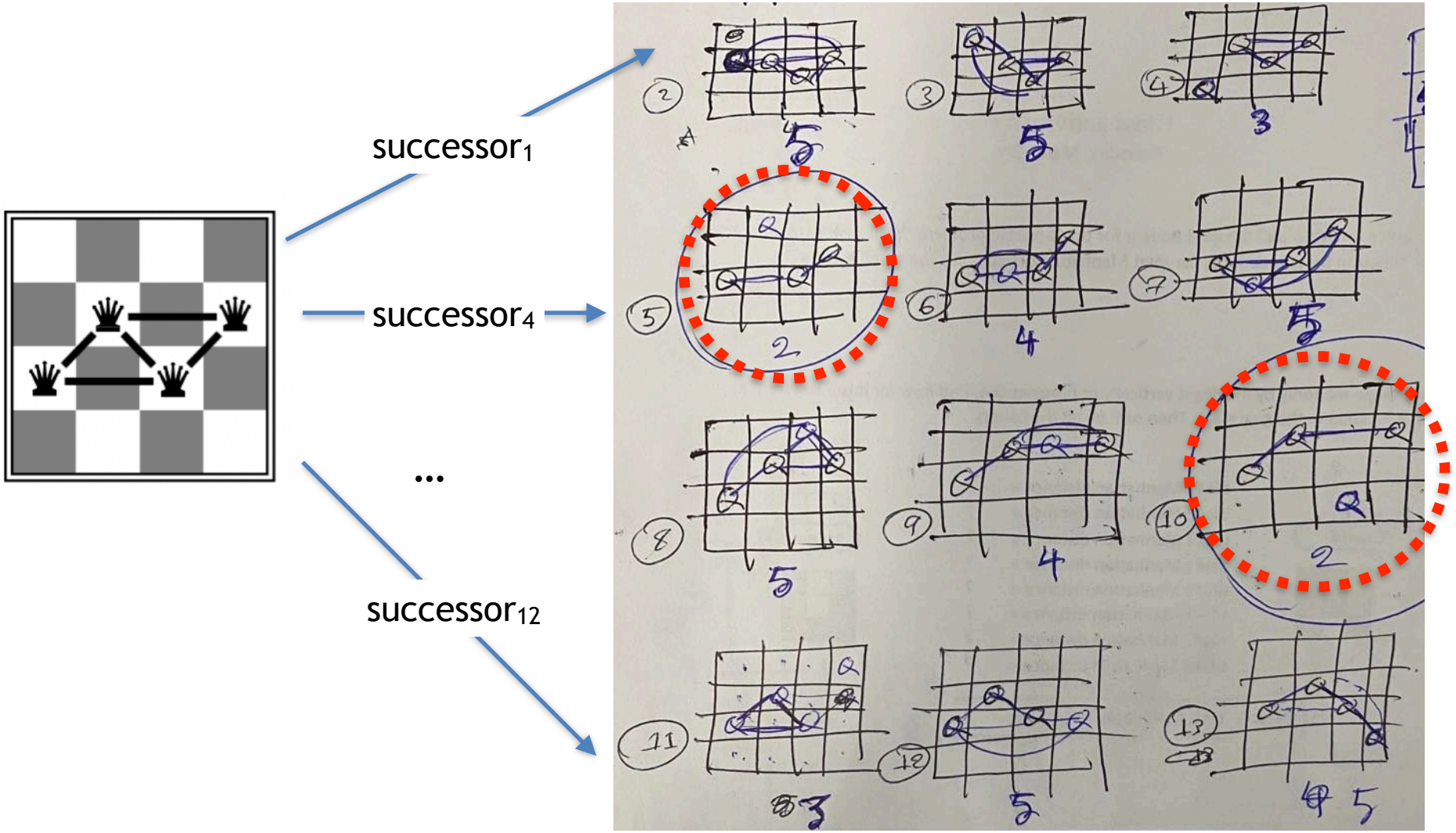
Review: Hill-Climbing Search

- Let's start with the first local search ie, hill-climbing search

- Depending on whether we are trying to find a global max or a global min, there are two versions of hill-climbing search
  - **Steepest-ascent:** Continually moves in the direction of increasing value
  - **Steepest-descent:** Continually moves in the direction of decreasing value

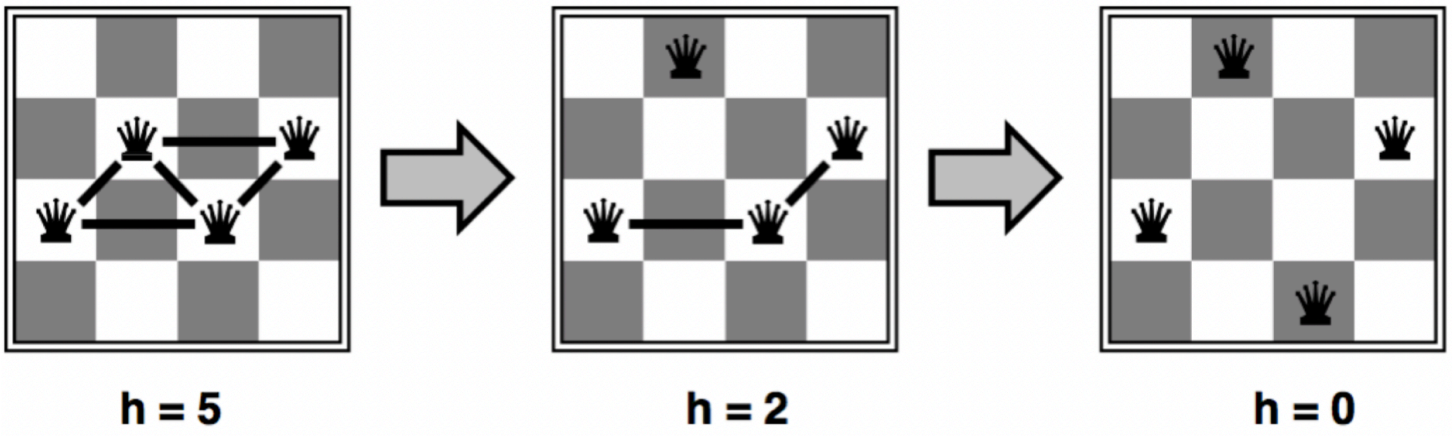
Review: Hill-Climbing to Solve 4-Queens

- What are the costs associated with each successor state?



Review: Hill-Climbing to Solve 4-Queens

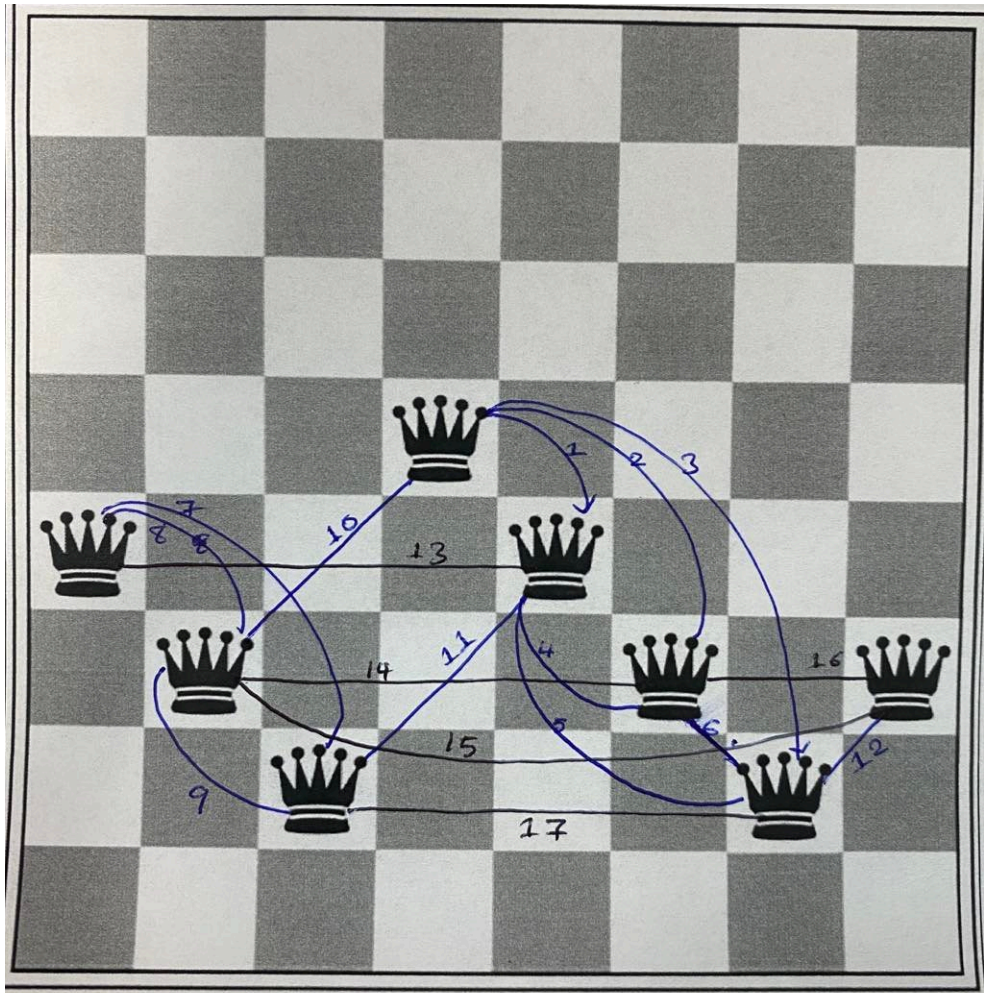
- With Hill-Climbing search, always choose the move that minimizes the *cost function*



- In other words, always choose the move that best reduces the *number of conflicting queens*

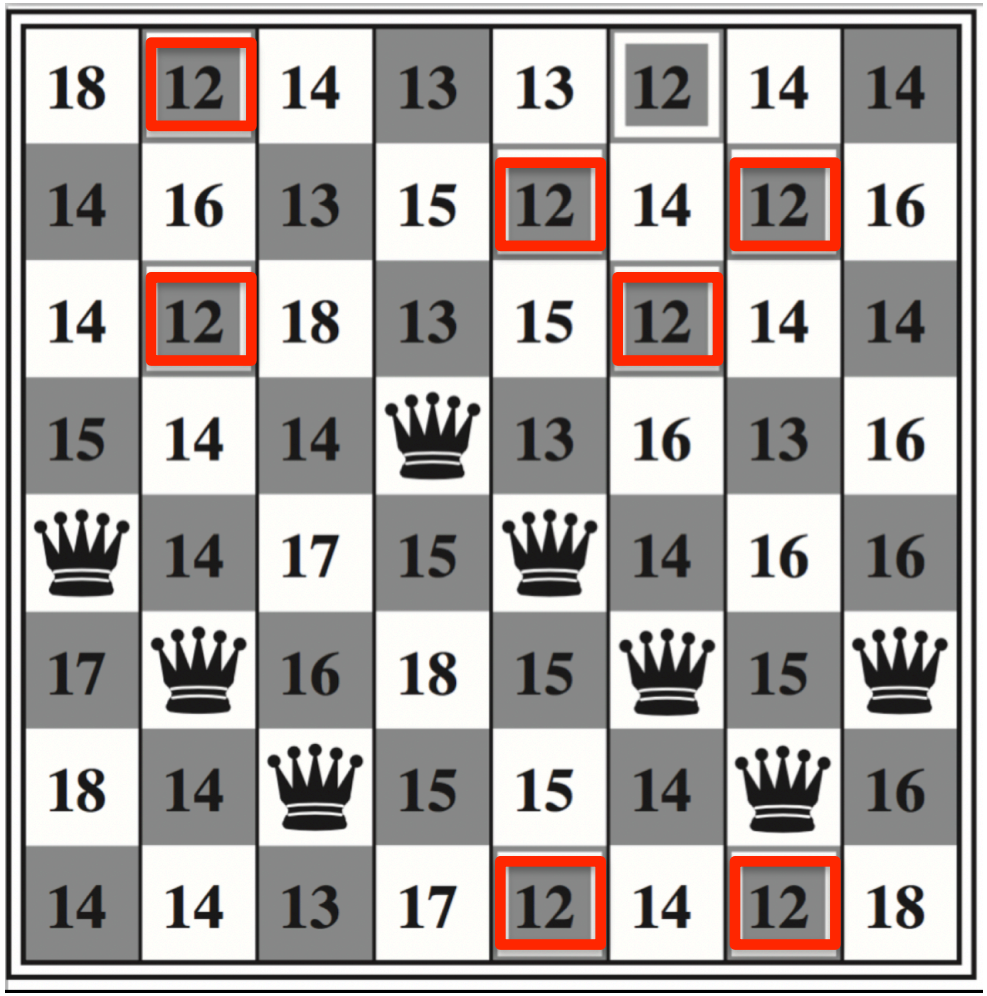
Review: Class activity#1: Solution

- How many pairs of queens are attacking each other?
- Answer: 17 pairs are attaching each other



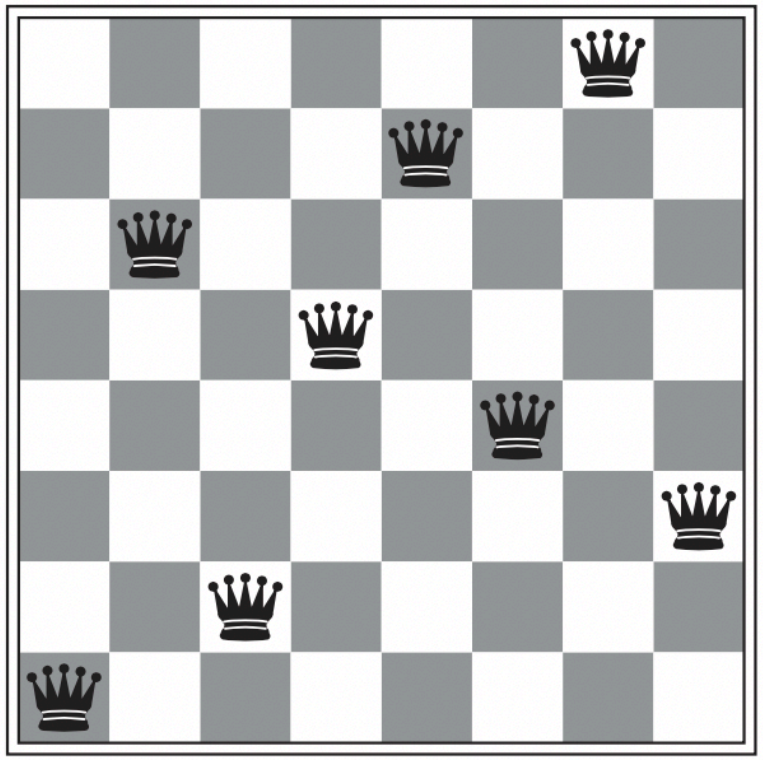
Review: Successor States for 8-Queens

- The cost of each move
- Hill-climbing tells us to make the lowest cost move (since we are trying to minimize the cost function)
- If there is a tie (multiple lowest cost moves), we break the tie randomly



Review: Stalling Out

- What is the cost of this configuration?
- Problem: Every successor has a higher cost
- So we get stuck with a near solution



New Material: Local Search

Issues with Hill-Climbing

- It's possible that hill-climbing will yield only a local minimum (or maximum)
  - not the optimal solution

- How to increase chances of getting the optimal solution?
  - Many variants have been invented

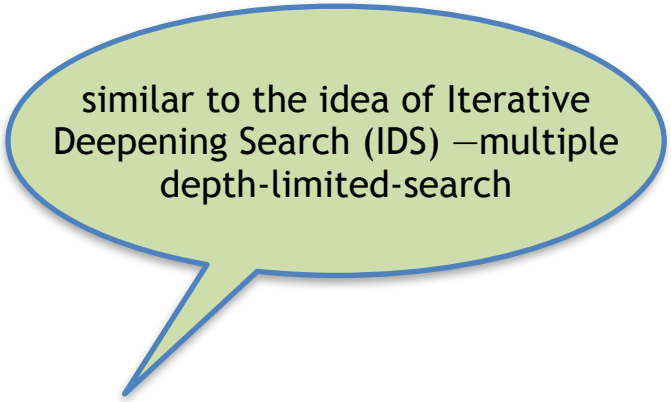
Idea#1: Random-Restart Hill Climbing

- Try Hill-climbing algorithm with different random initial state many times until you get the solution!

- That is, we start with some preset number of restarts, with a random initial state.

- After running through all of these restarts, we return the best solution found.

Idea#1: Random-Restart Hill Climbing




```
for i in range(number of restarts):
        shuffle start state
        apply hill-climbing algorithm
        determine if attempt gives best
        solution
return best solution
```

Why/when does it work well?

- There are **many goal states** that are well-distributed over the state space
- Building a search tree would be much less efficient because of the **high branching factor**
- If no solution has been found after a few steps, it's **better to start it all over again**

Idea #2: Local Beam Search

- Idea: Instead of keeping one state in memory (as in Hill Climbing or Random Restart Search), keep k states in memory

- Repeat
  - Begin with k randomly generated states —each one is a beam
  - At each step, create all successors of each of the k states
  - If any one of them is those successors is a goal state then the algorithm halts.
  - Otherwise, it selects the k best successors from the complete list.

Idea #2: Local Beam Search

- This is not the same as Random Restart search as k searches run in parallel!

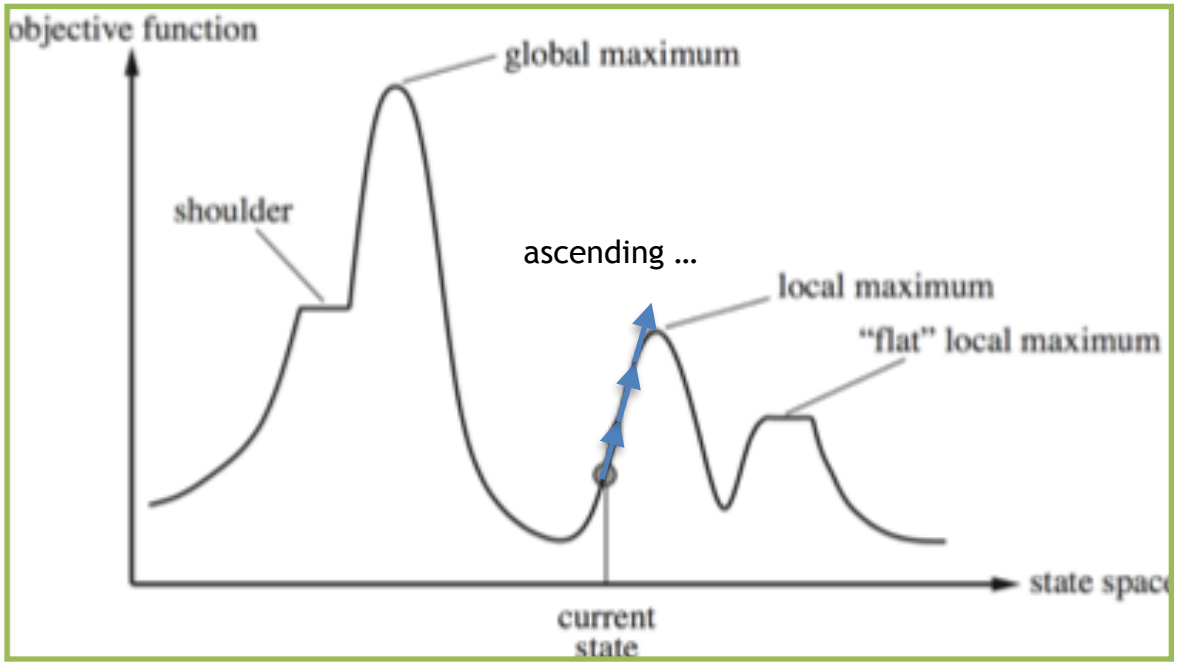
- In a local beam search, useful information is passed among the parallel search threads.

  - In effect, the states that generate the best successors say to the others, "Come over here, the grass is greener!"

  - The algorithm quickly abandons unfruitful searches and moves its resources to where the most progress is being made.

Idea #3: Simulated Annealing Search

- Hill-climbing algorithm that never makes a "downhill" moves towards states with lower value is guaranteed to be incomplete
  - It can get stuck at a local maximum



Idea #3: Simulated Annealing Search

- We make use of an idea that comes from metallurgy: annealing.
- Annealing is the process used to harden metals and glass by heating them then gradually cooling them
- Idea: escape local maxima by allowing some "bad" moves, i.e., some moves in the wrong direction, but we gradually decrease the size and frequency of these moves
- If the "temperature" is decreased slowly enough, the probability of finding an optimal solution approaches 1.

Idea #3: Simulated Annealing Search

a.k.a Stochastic Hill Climbing Search

```
function SIMULATED-ANNEALING( problem, schedule) returns a solution state
    inputs: problem, a problem
            schedule, a mapping from time to "temperature"
    local variables: current, a node
                     next, a node
                     T, a "temperature" controlling prob. of downward steps

    current ← MAKE-NODE(INITIAL-STATE[problem])
    for t ← 1 to ∞ do
        T ← schedule[t]
        if T = 0 then return current
        next ← a randomly selected successor of current
        ΔE ← VALUE[next] – VALUE[current]
        if ΔE > 0 then current ← next                    } Good move
        else current ← next only with probability e^(ΔE/T)  } Bad move
```

5000	4999	4998	4997	4996	...	2	1	0

Idea #4: Genetic Algorithm

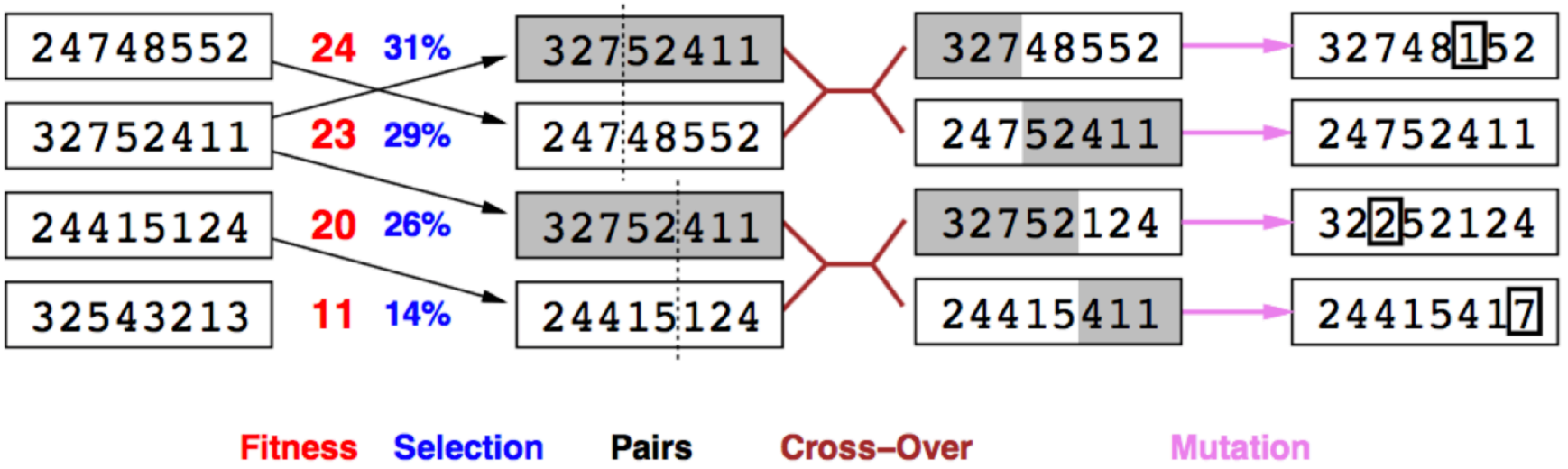
- Genetic algorithm is a variant of a stochastic beam search
- Idea: Combine *two* parent states rather than modifying a single state.
- We begin by selecting a "generation" of k randomly generated states.
  - We call this the "population"
- Produce the next generation by an evolutionary process.

Idea #4: Genetic Algorithm

- Each member is rated on a "fitness function"
  - The fitness function is a heuristic that measures how close it is to a solution.
- Pairs are selected in accordance with the probabilities associated with the fitness function
  - Note the analogy to natural selection
- New "successors" (offspring) are created by crossing the parent strings at a randomly-selected crossover point
- Each location is subject to a random mutation with a small probability

Idea #4: Genetic Algorithm for 8-Queens

= stochastic local beam search + generate successors from **pairs** of states



In-class Coding Activity

- Please, find link to the local-search-activity from blackboard

Coding Activity

- Inside the class HillClimbingSearch: implement the descent version of hill climbing
- Inside the class NQueensProblem: define the cost given by number of conflicting pairs of queens
- Once you've done these two things, solve the examples in main function at the bottom

How do we represent N-Queens?

- How would you represent a state of the n-queens problem?
- How would you represent the possible successors of a given state of the n-queens problem?
- How would you calculate the cost of a given state of the n-queens problem?

Steepest Descent Hill-Climbing

- In HillClimbingSearch.py: Implement the descent version of hill climbing

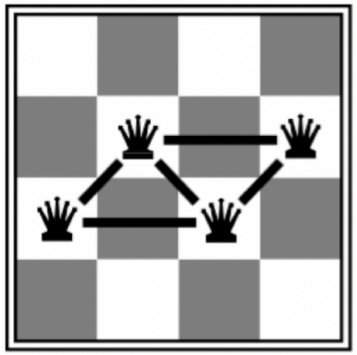
```
function HILL-CLIMBING-DESCENT(problem) returns a state that is a local minimum

  current_node ← MAKE-NODE(problem.INITIAL-STATE)
  loop do:
    neighbor_node ← a lowest-valued successor of current_node
    if neighbor_node.VALUE >= current_node.VALUE
        return current_node.STATE
    current_node ← neighbor_node
```

Coding Activity

- NQueensProblem Class: define the successor function:
  - number of conflicting pairs of queens in the same row
  - number of conflicting pairs of queens in the diagonal (upward direction)
  - number of conflicting pairs of queens in the diagonal (downward direction)

Successor states from the current state

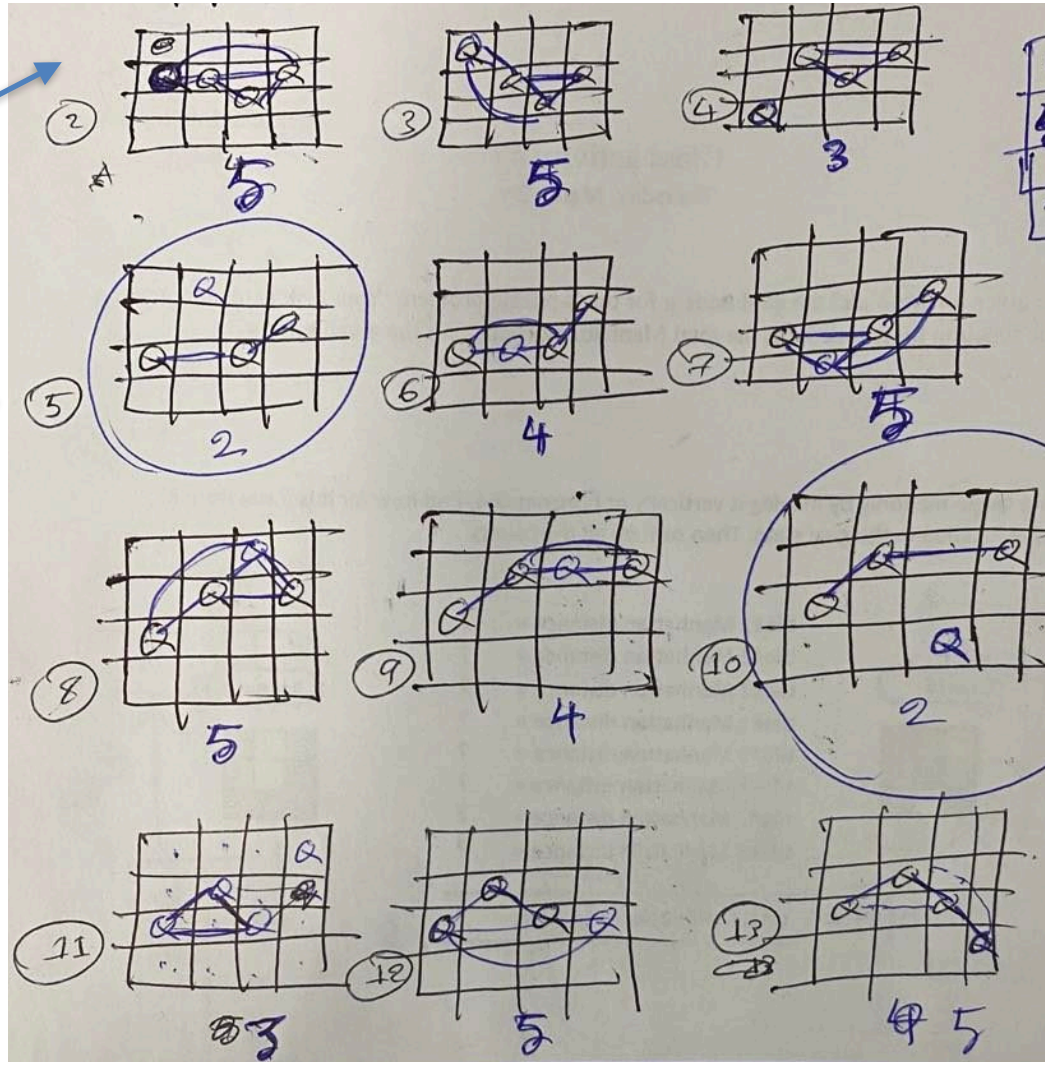
- What are the successor states from the one below?



successor$_1$

successor$_4$

successor$_{11}$

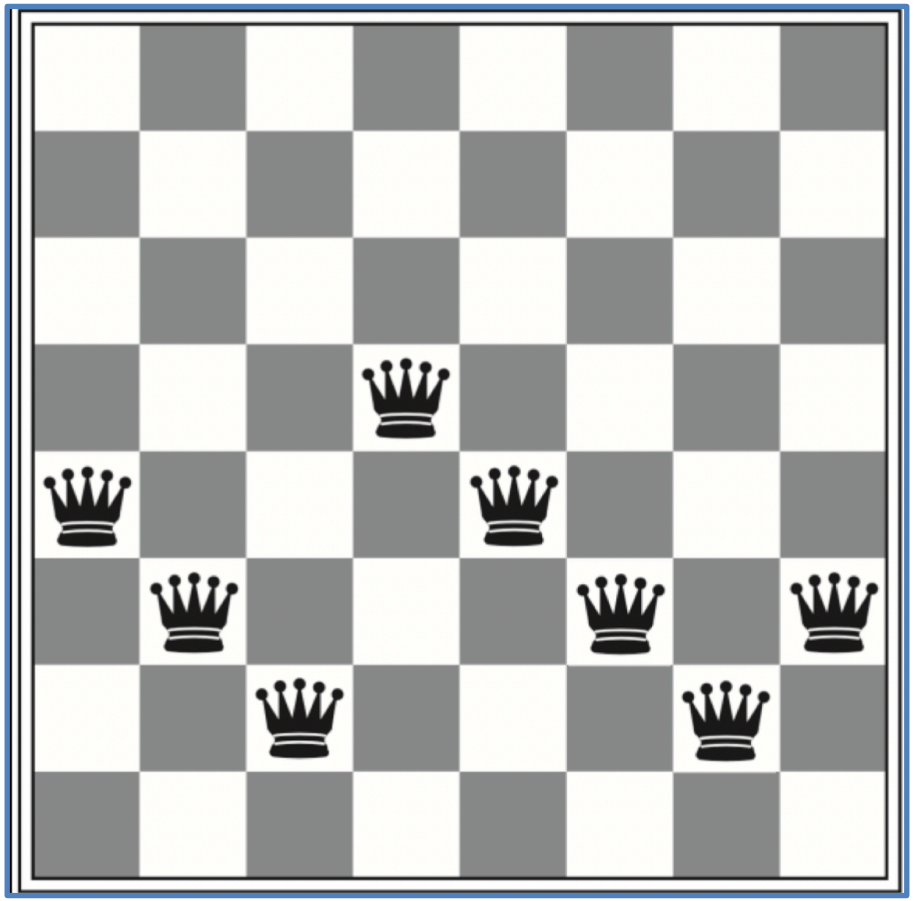


Successor states from the current state

- Following code will generate the successors

```
def actions(self, state):
    """
        state: a tuple that is the state of a NQueens configuration
        returns a list of successors or neighbors of possible
        configurations from the input state
     """
    possible_actions = []
    for col in range(self.N):
        for val in range(self.N):
            if state[col] != val:
                new_state = list(state)
                new_state[col] = val
                possible_actions.append(tuple(new_state))
    return possible_actions
```

Successor states from the current state



Coding Activity

- NQueensProblem Class: define the cost given by number of conflicting pairs of queens:
  - number of conflicting pairs of queens in the same row
  - number of conflicting pairs of queens in the diagonal (upward direction)
  - number of conflicting pairs of queens in the diagonal (downward direction)

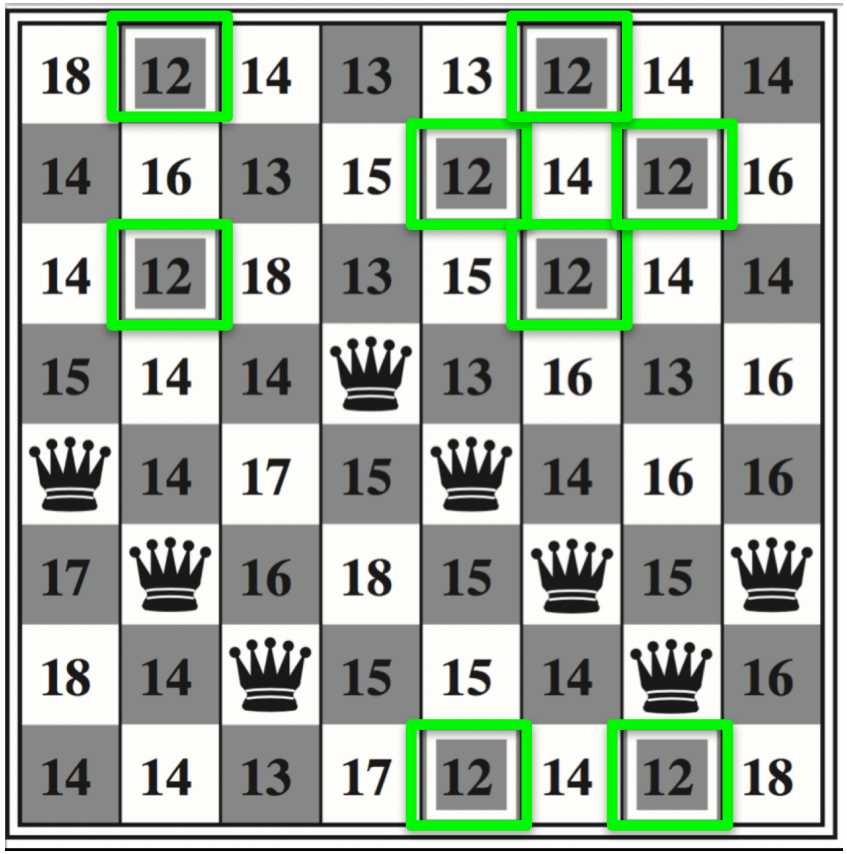
Coding Activity: Cost Function

- Following code where you will generate the cost of state by finding the number of conflicting pairs of queens:
  - number of conflicting pairs of queens in the same row
  - number of conflicting pairs of queens in the diagonal (upward direction)
  - number of conflicting pairs of queens in the diagonal (downward direction)

```
def cost(self, state):
    # TODO
    """"""# Return number of conflicting queens for a given node"""""
    num_conflicts = 0

    '''

    # hints: find the types of conflict
    #    i)   two queens in the same row
    #    ii)  two queens in the same diagonal (upward diretion)
    #    iii) two queens in the same diagonal (downward diretion)

    for col in range(self.N-1):
        for nstep in range(col+1,self.N):
            print("")
            # TODO
    '''
```

Successor states from the current state

- Generate 56 states from the current state
- The cost of each move is shown below
- Go through each successor state and keep track of the best one



Coding Activity

- HillClimbingSearch Class: finish the search function:

```
function HILL-CLIMBING-DESCENT(problem)
returns a state that is a local minimum

  current_node ← MAKE-NODE(problem.INITIAL-STATE)
  loop do:
    neighbor_node ← a lowest-valued successor of current_node
    if neighbor_node.VALUE >= current_node.VALUE
        return current_node.STATE
    current_node ← neighbor_node
```

Pseudocode

```
def hillclimb(self, startState):
    startTime = time.time()

    # create a node for the state;
    state     = startState
    best_cost = self.problem.cost(state)
    best_move = state

    #TODO:
    # implement Hill Climbing Algorithm here
    # return state with a local minimum
    found_new_state = True

    #-----------------------------------------
    # Your solution

    # ...
    # ...
    # ...


    #-----------------------------------------
    self.timeSpent = time.time() - startTime

    return state
```

Python Implementation