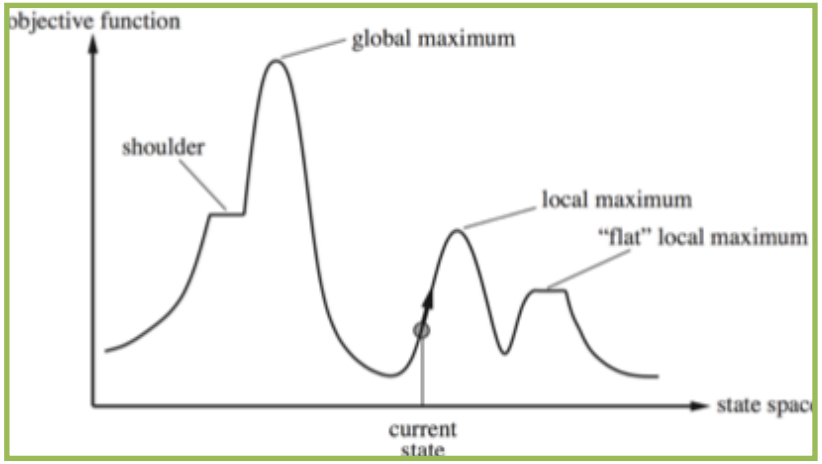
CS143: Artificial Intelligence



Introduction to Local Search
Hill-Climbing

Drake UNIVERSITY

Review: Search Strategies

- Uninformed ("blind") search:
  - algorithms have no additional information about states.
    - Breadth-First Search (BFS)
    - Depth-First Search (DFS)
    - Iterative Deepening Search (IDS)

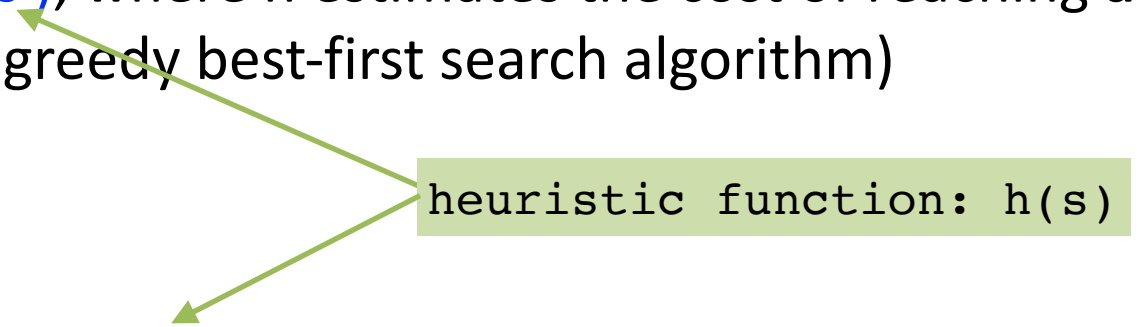
- Informed ("heuristic") search:
  - more information given about whether a non-goal state is "more promising" with respect to a goal state
    - Greedy Best-First Search
    - A* Search

Review: Informed (Heuristic) Search

- Explores most 'promising' state from FRONTIER first
  - typically implemented with a priority queue
  - requires *evaluation function* $f(s') \geq 0$ that estimates the **"cost" from initial state, through s', to a goal state**

Review: Informed (Heuristic) Search

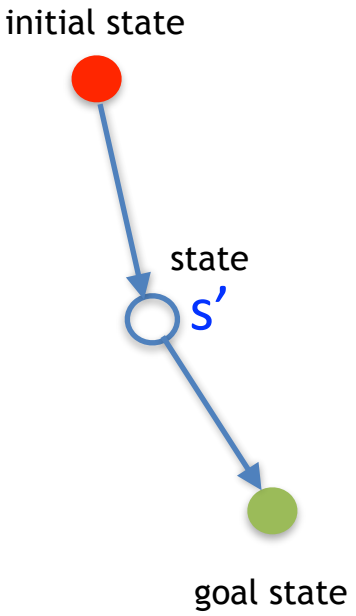
- Typical choices for the evaluation function f(s):
  - f(s') = h(s'), where h estimates the cost of reaching **a goal from s'** (greedy best-first search algorithm)



  - f(s') = g(s') + h(s'), where g() is **cost of path from start state to s'** and heuristic function h() estimates the **cost of reaching a goal from s** (A* search algorithm)

Review: A* Search

- Main idea: avoid expanding paths that are already expensive



- Use an evaluation function f(s'):
  - f(s') = g(s') + h(s')
  - g(s') = cost of path from initial state to s'
  - h(s') = cost of reaching a goal state from s'

`heuristic function: h(s)`

- f(s') measures an estimated total cost of path through s' to goal

Review: A* Search

- Use Graph Search algorithm by making three changes:
  - use priority queue as data structure instead of stack/queue
  - Insert a node s' into the FRONTIER after computing its evaluation function f(s') = g(s') + h(s')
  - remove a node s' from the FRONTIER based on the lowest estimate of f(s') among all the nodes inside the FRONTIER

Review: A* Search

Use priority queue for FRONTIER

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

REMOVE node s with minimum value of f()

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

Review: A* Search 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**

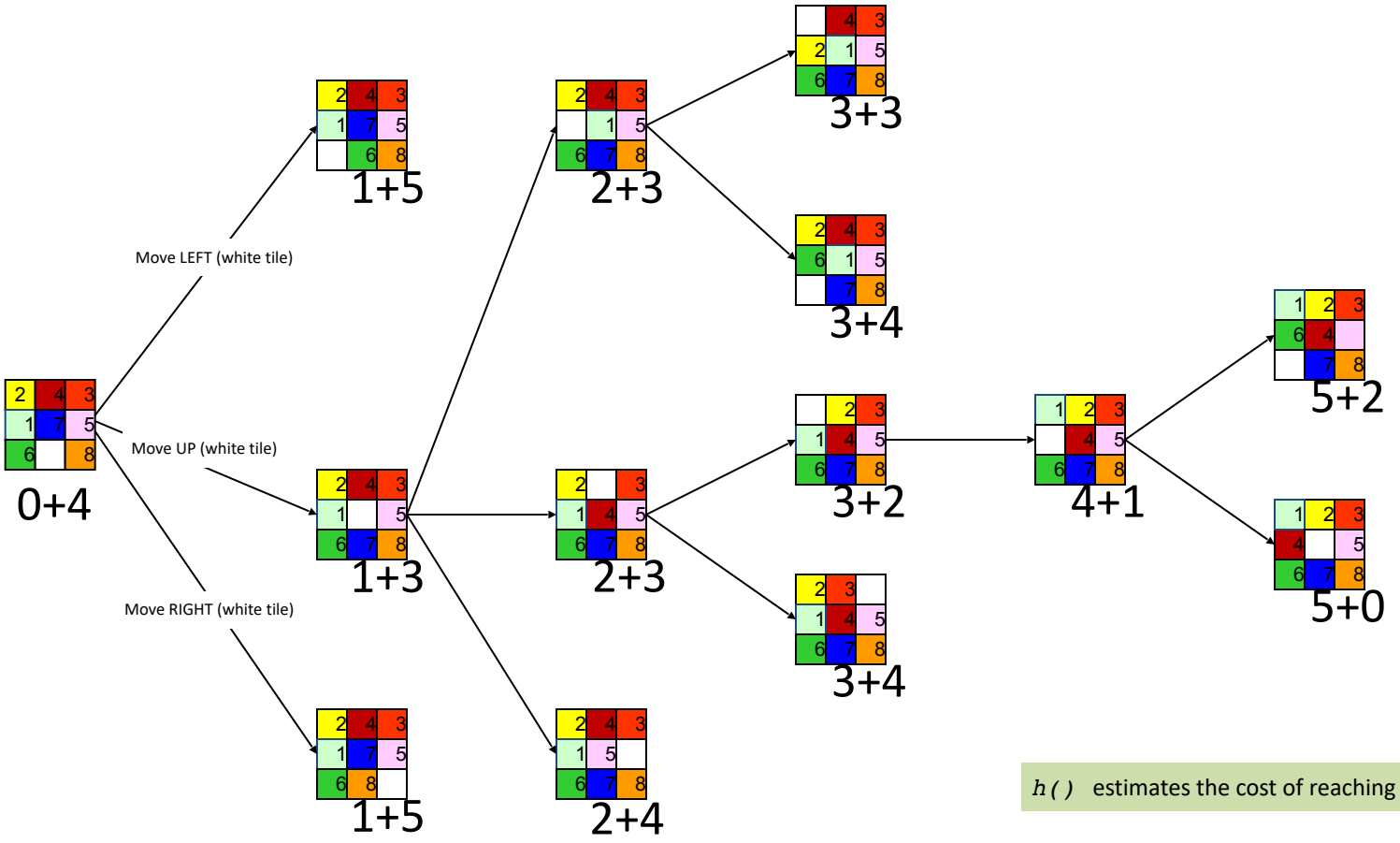
Is tile#1 misplaced? =	1
Is tile#2 misplaced? =	1
Is tile#3 misplaced? =	0
Is tile#4 misplaced? =	1
Is tile#5 misplaced? =	0
Is tile#6 misplaced? =	0
Is tile#7 misplaced? =	1
Is tile#8 misplaced? =	0
h(s) = total # of misplaced tiles =	4

Goal state

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

Review: A* Search

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

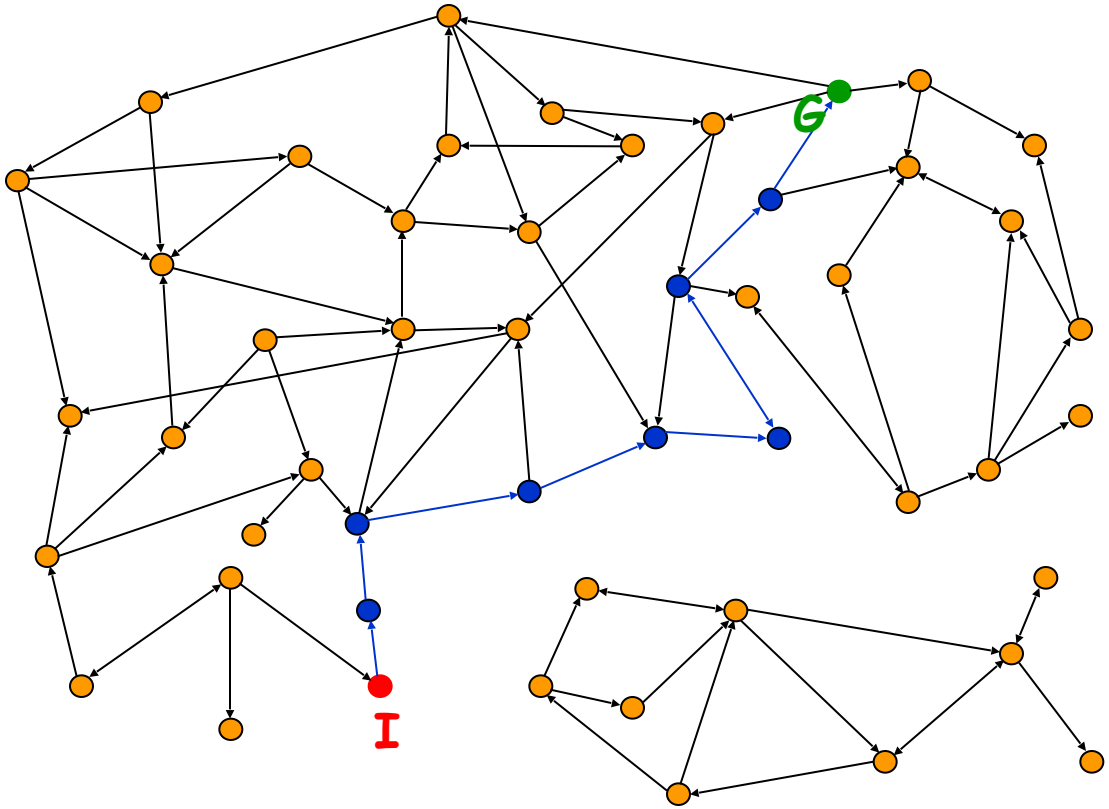


h( ) estimates the cost of reaching a goal from s

New Material: Local Search

Pathless Problems

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



Beyond Classical Search

- In many optimization problems, path is irrelevant; the goal state itself is the solution
  - "I don't care about the path the led me to the optimal configuration of 8-Queens, I just want an answer"
  - "I want my auto-driving car to park within the lines (now!); I don't care if the algorithm is optimal"

- In such cases, can use iterative improvement algorithms
  - keep a single "current" state, try to improve it

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)

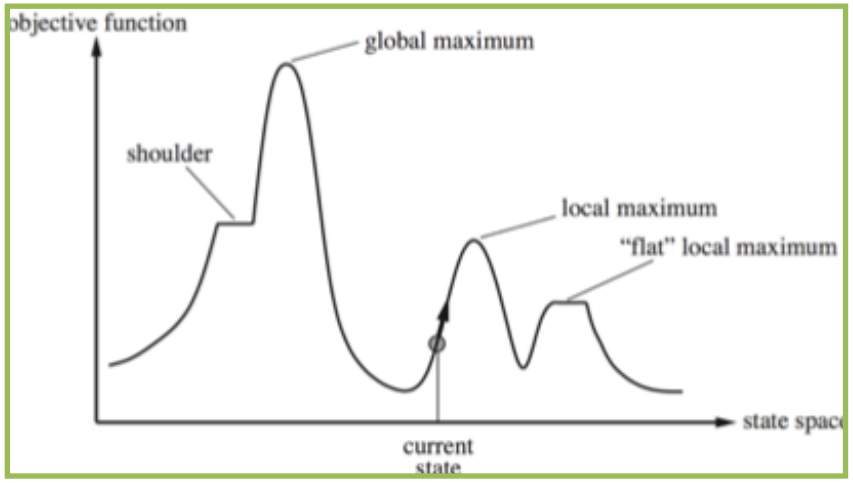
Hill-Climbing Search

- 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

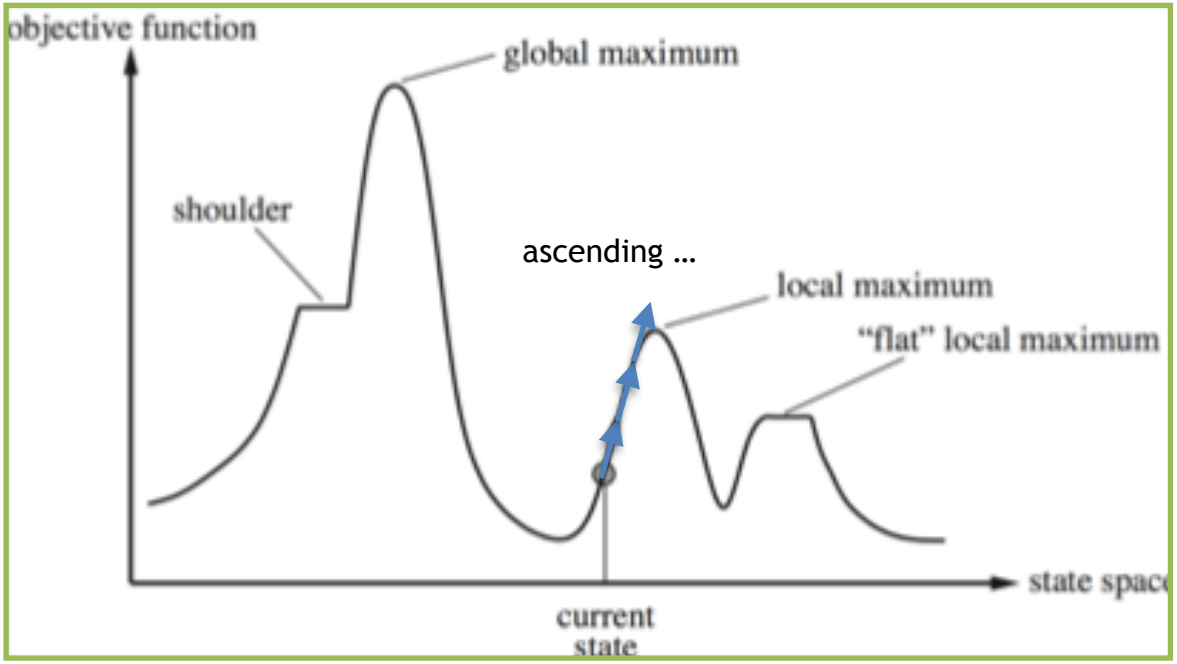
State-Space Landscape

- If elevation corresponds to the value of an objective function, we want to find the highest peak (global max)



- If, however, elevation corresponds to cost, we want to find the lowest valley (global min)

Steepest Ascent Hill-Climbing



Steepest Ascent Hill-Climbing

`function` HILL-CLIMBING-ASCENT(*problem*) `returns` a state that is a local maximum

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

If all my neighbors are lower than me, then I am at the highest location

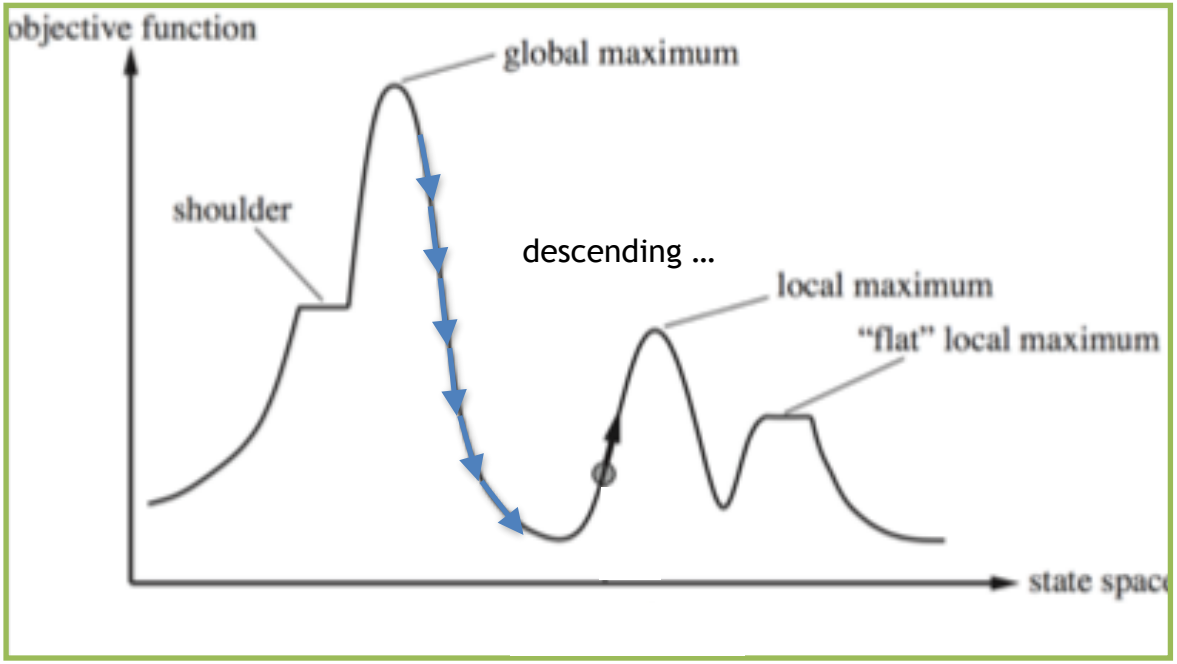
Steepest Descent Hill-Climbing

`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
```

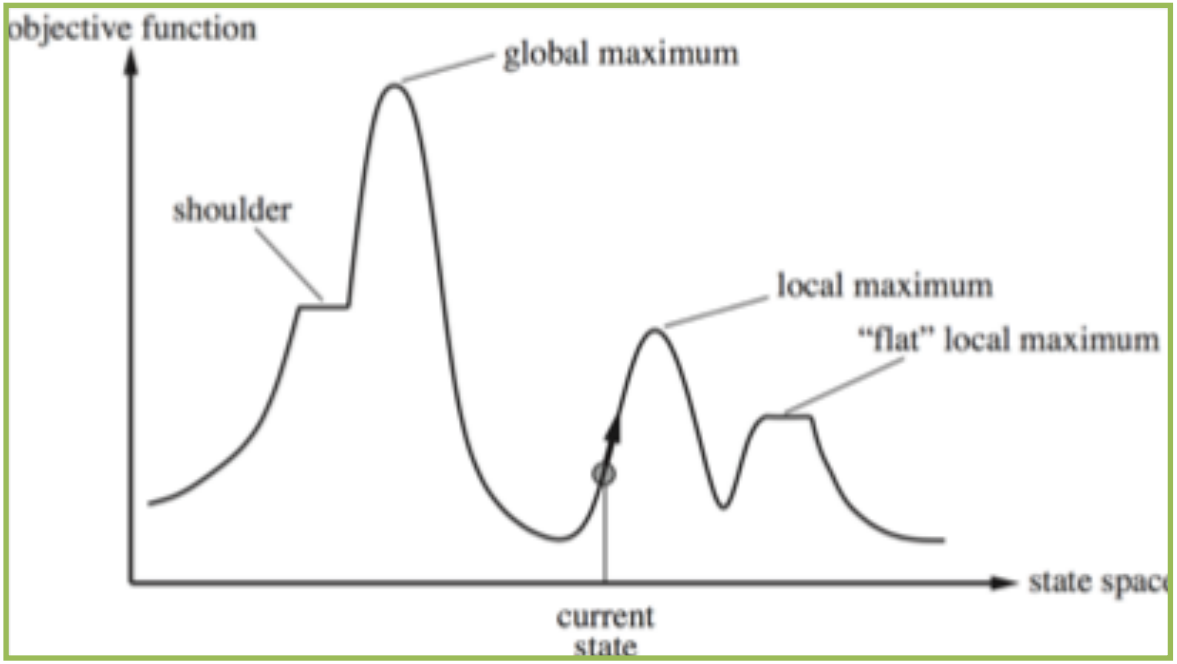
If all my neighbors are higher than me, then I am at the lowest location.

Steepest Descent Hill-Climbing



Problems with Hill-Climbing

- Gets stuck at local extrema (local max or local min)

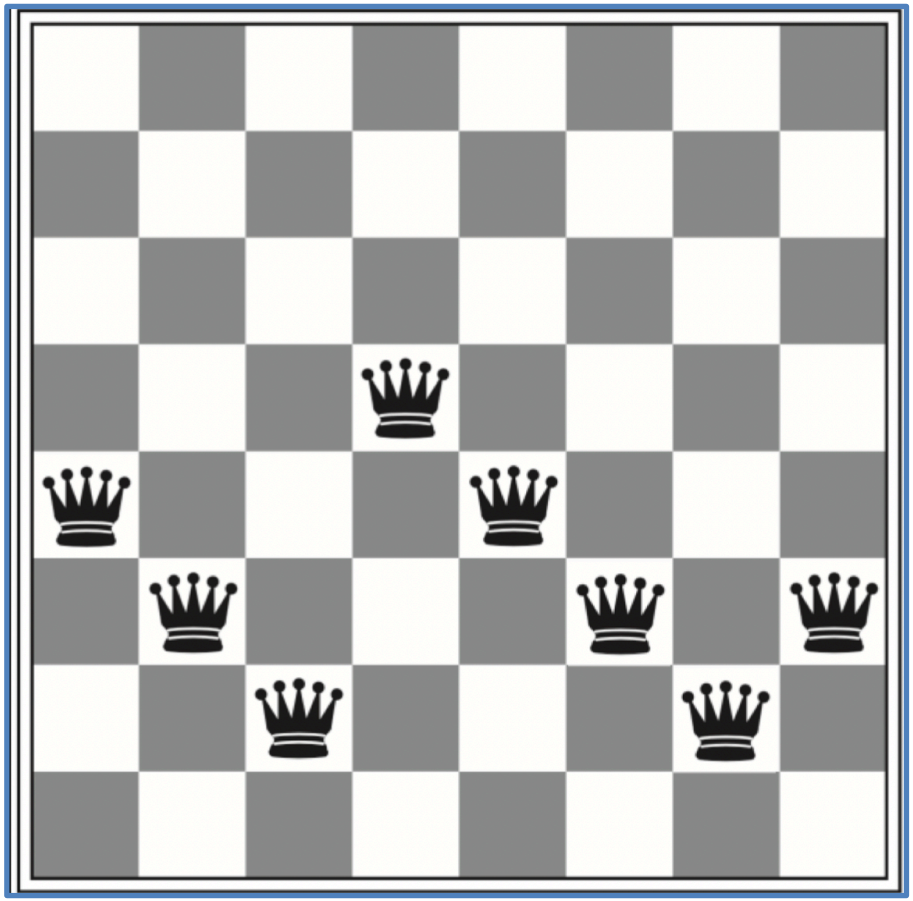


- search can get lost on shoulders or stuck on plateaus

Problem Solving with Hill-Climbing: N-Queens Problem

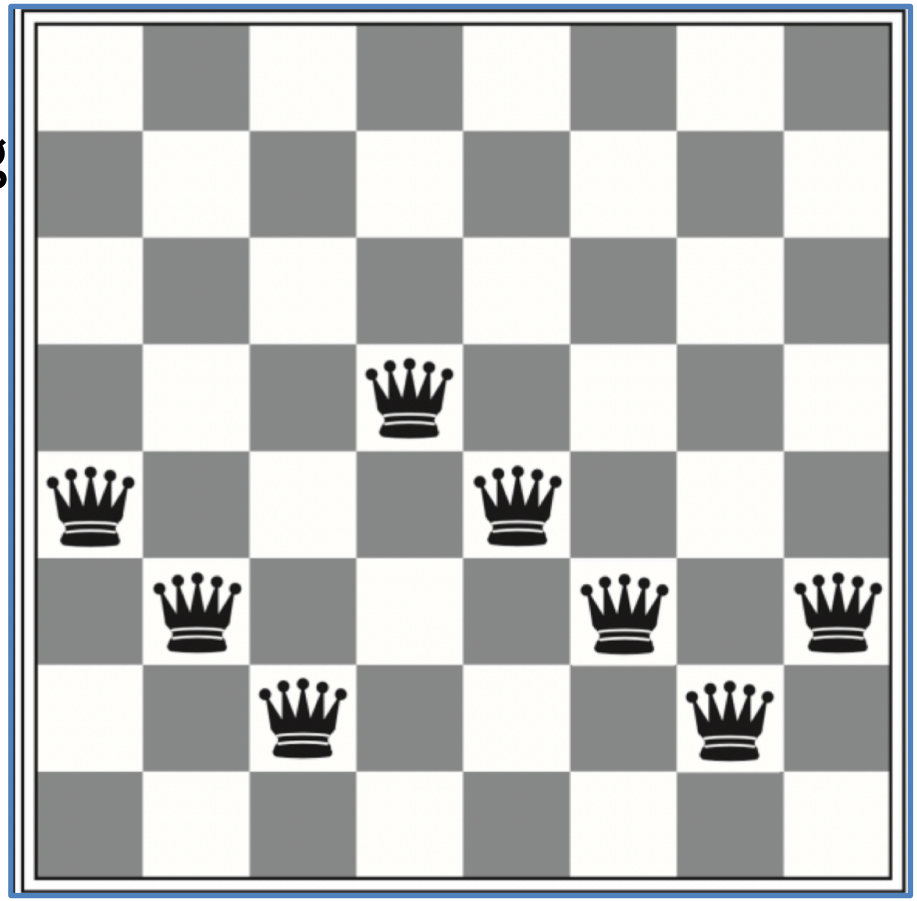
Place n queens in a $n \times n$ chessboard so that no two queens are in the:

- same row or
- same column or
- same diagonal



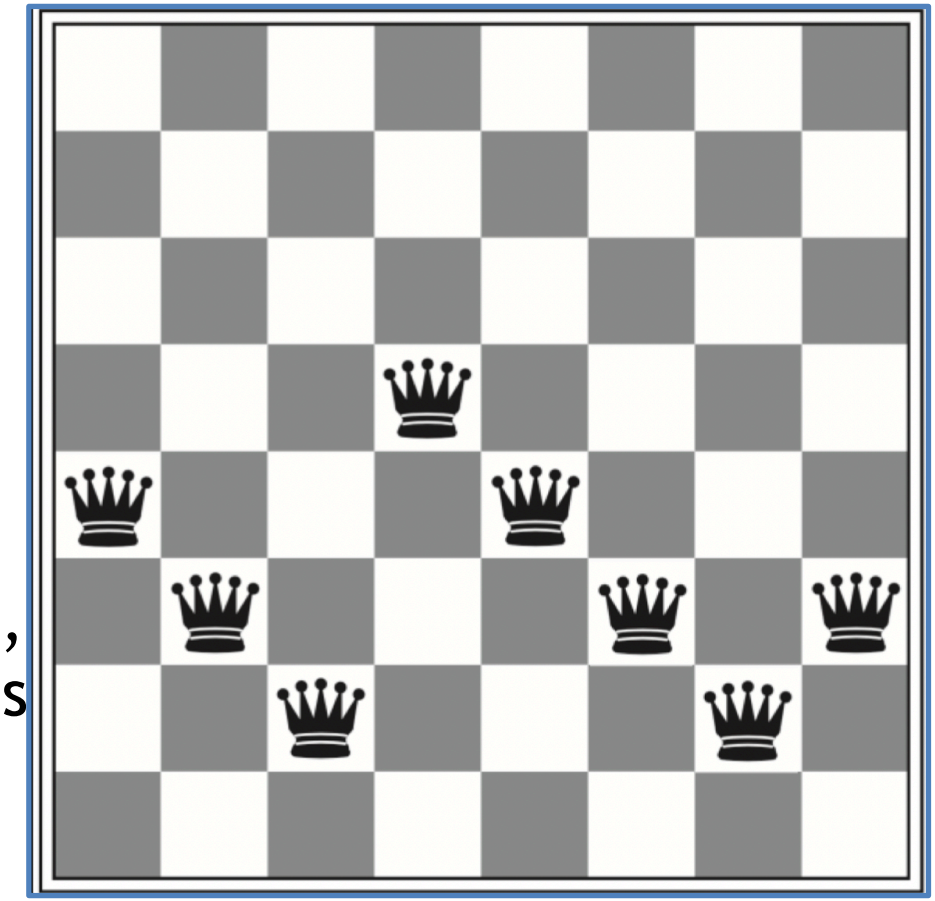
Problem Solving with Hill-Climbing: N-Queens Problem

- The number of pairs of queens that are attacking one another defines a *cost function* which we are seeking to minimize
- How many pairs of queens are attacking each other?



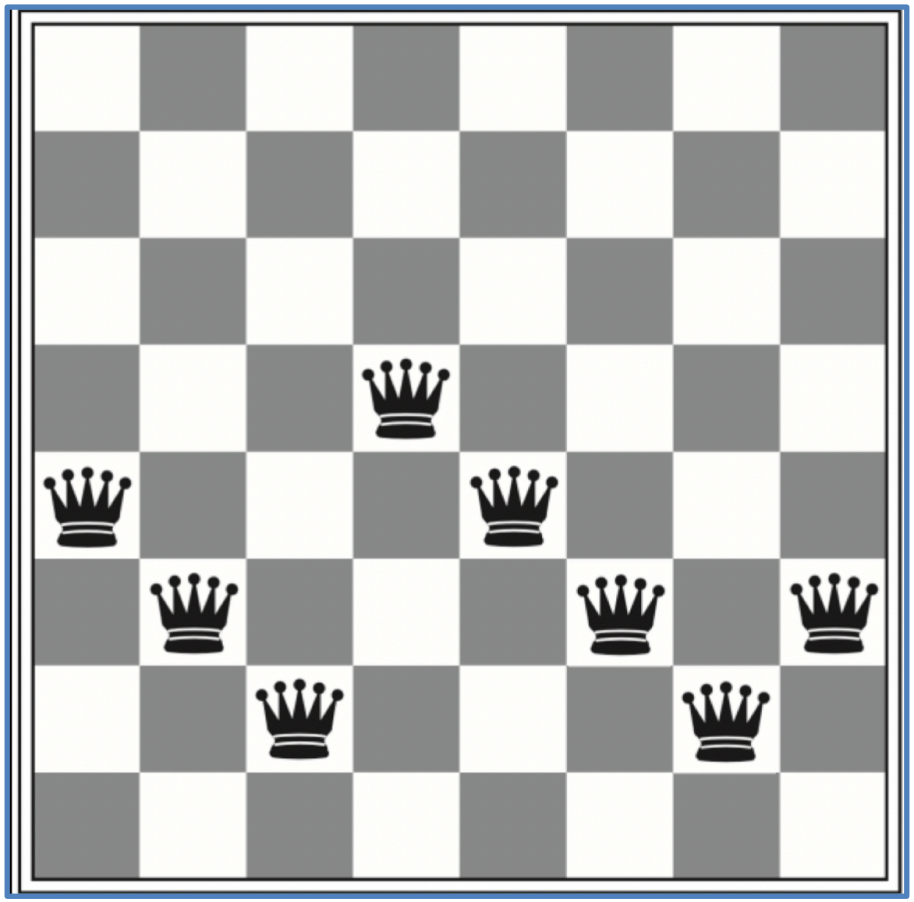
Successor States in the N-Queens Problem

- Given a configuration of the board, a successor state is obtained by moving a single queen to any other square in its column.
- For the 8-queens problem, how many successor states are there?



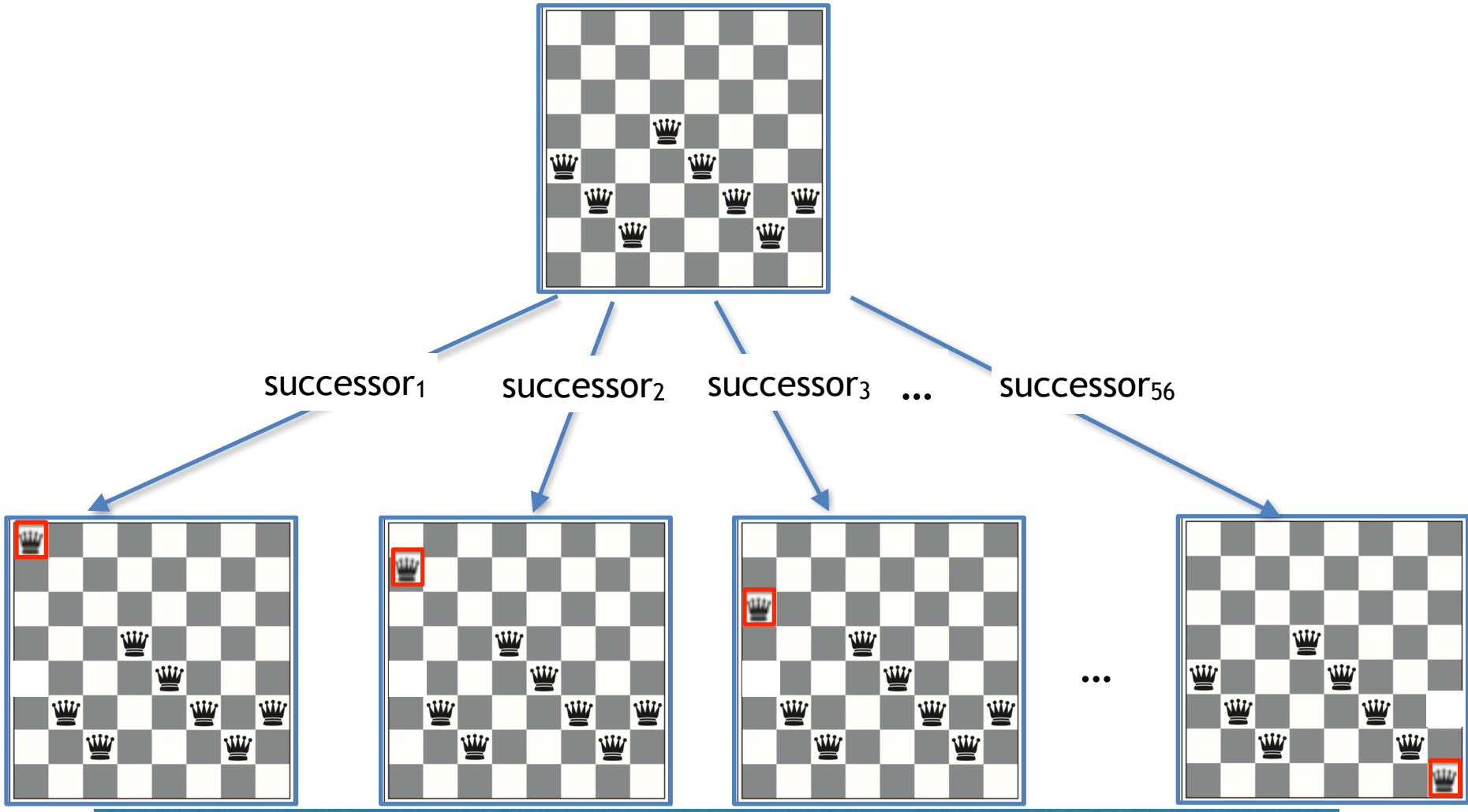
Successor states from the current state

- What are the successor states from the one below?



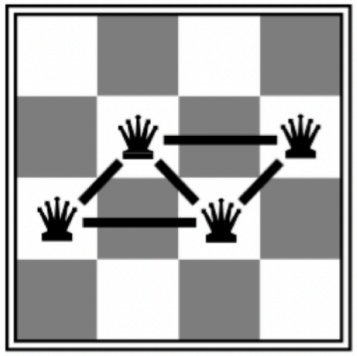
Successor states from the current state

- What are the successor states from the one below?



Cost associated with each successor states

- What are the costs associated with each successor state?

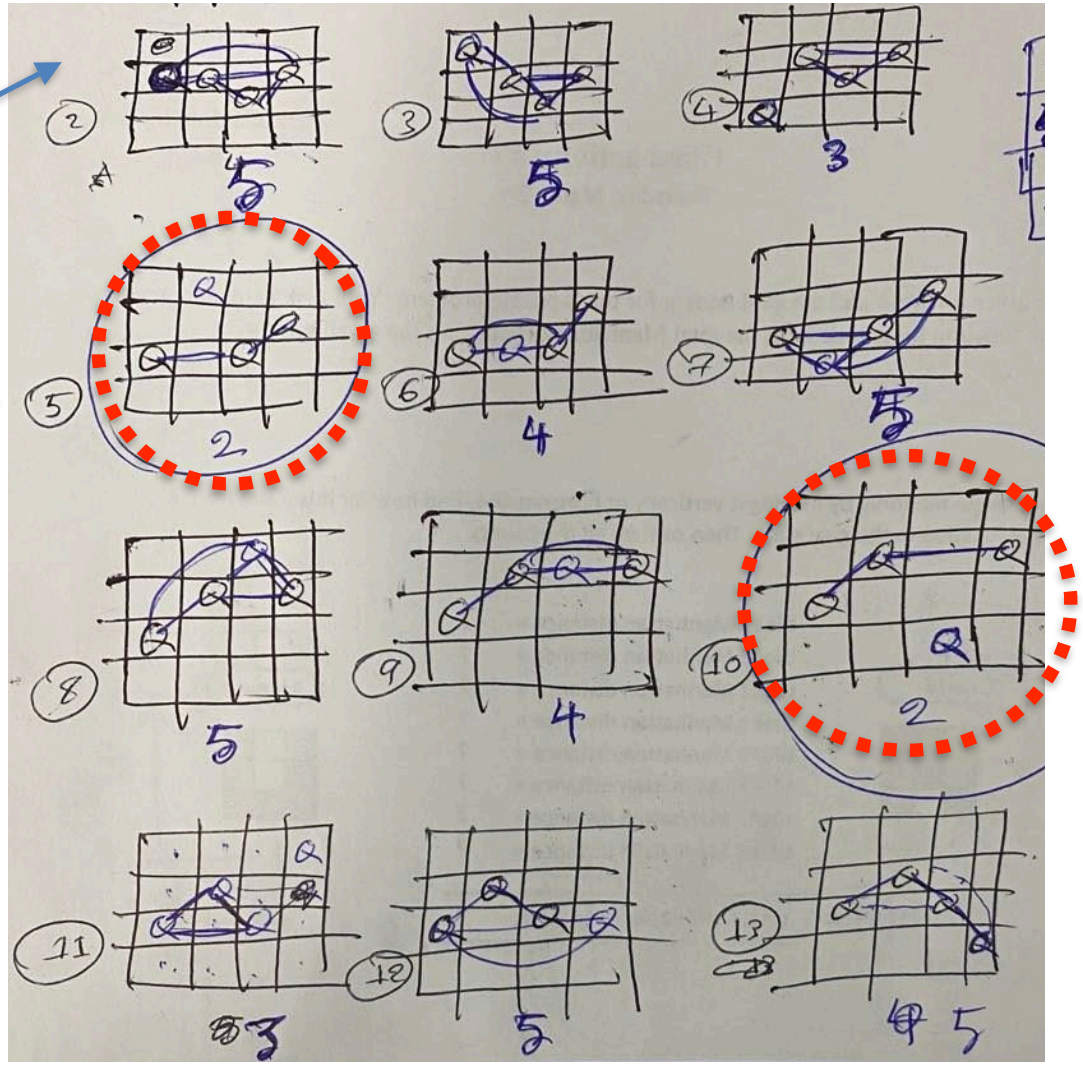


successor$_1$

successor$_4$

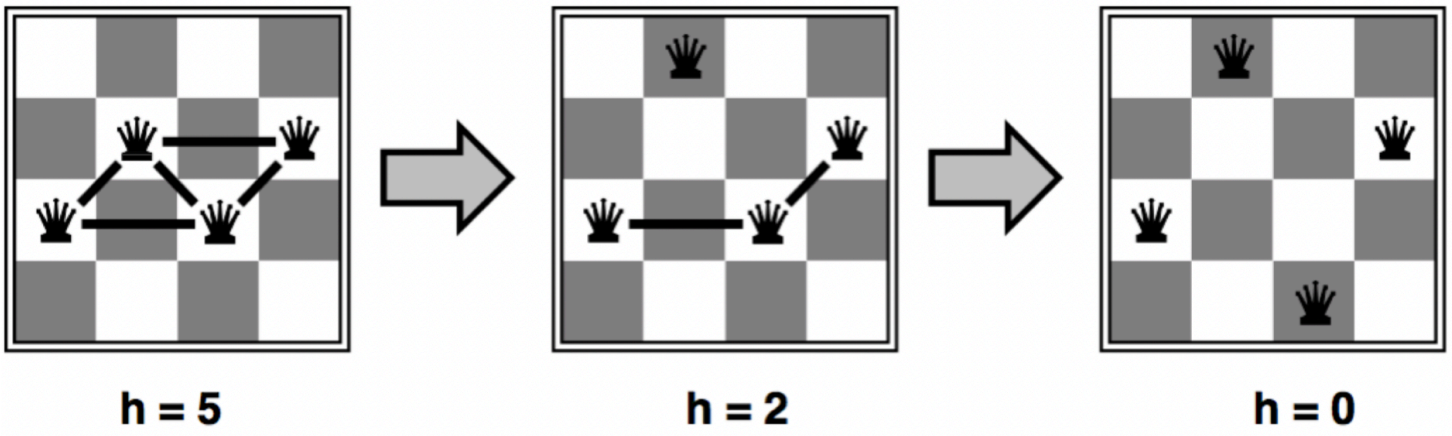
...

successor$_{12}$



Hill-Climbing to Solve N-Queens Problem

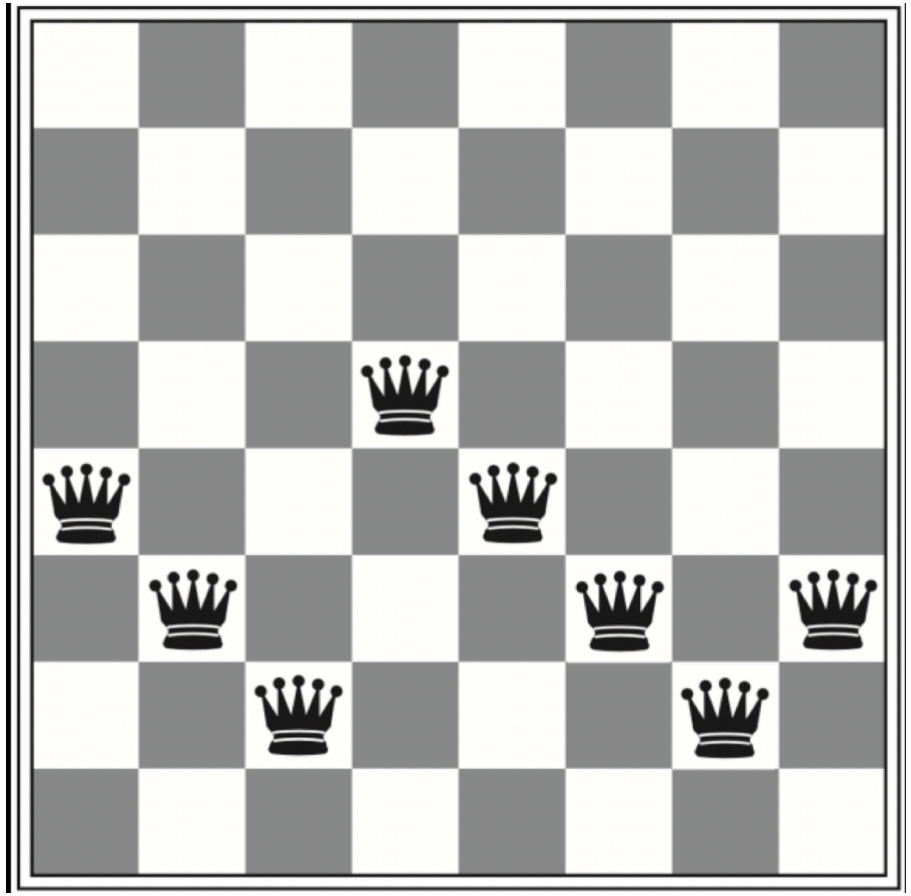
- With Hill-Climbing search, always choose the move that minimizes the *cost function*



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

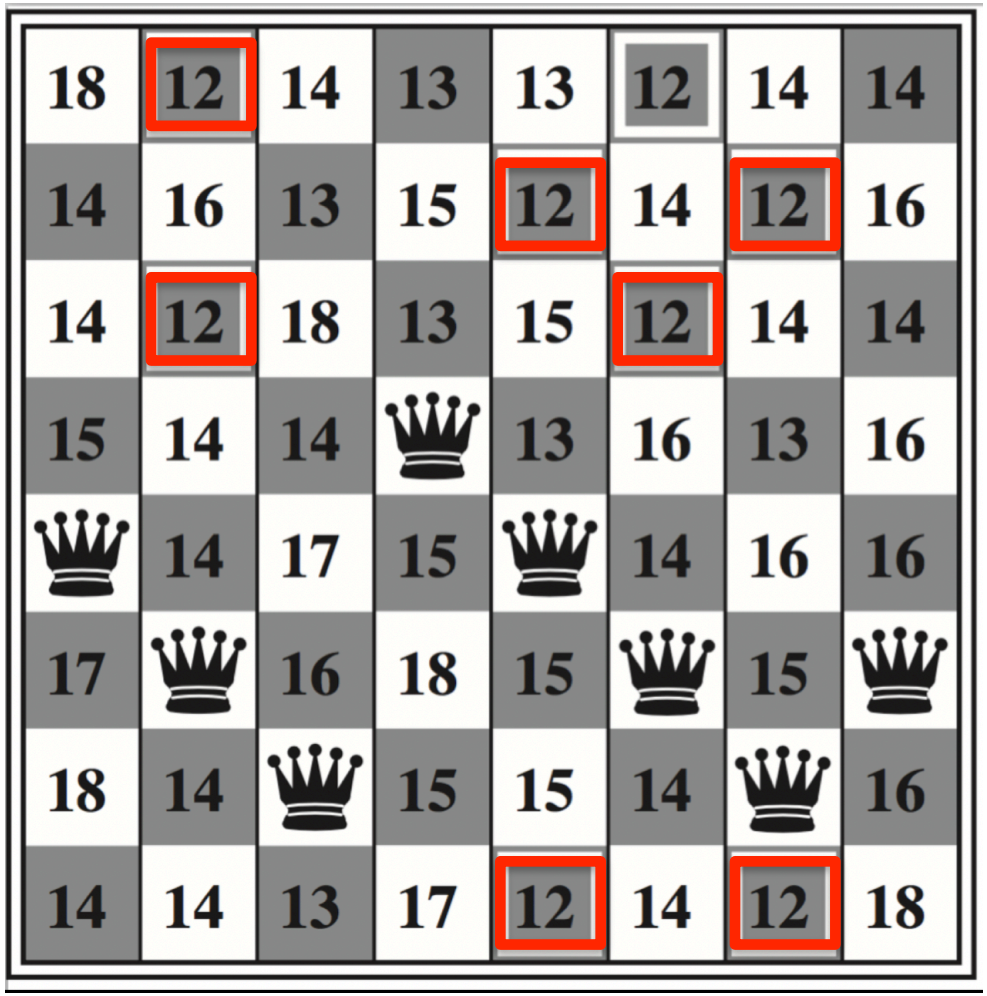
Class activity#1: Cost Function for 8-Queens Problem

- How many pairs of queens are attacking each other?



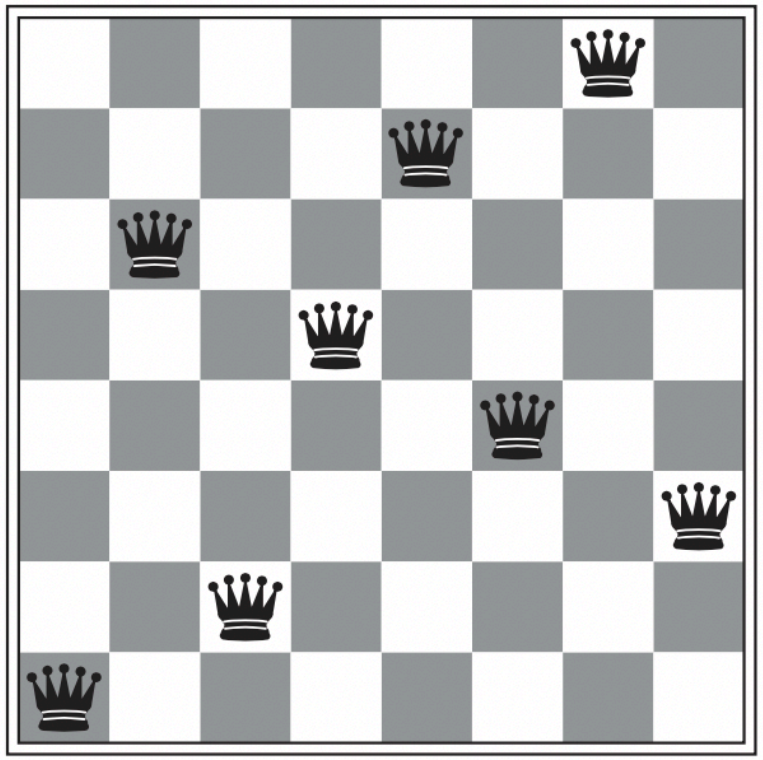
Successor States for 8-Queens Problem

- 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



Stalling Out

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



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?

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

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 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)

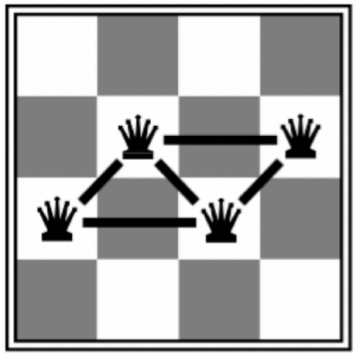
Coding Activity: Cost Function

- 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

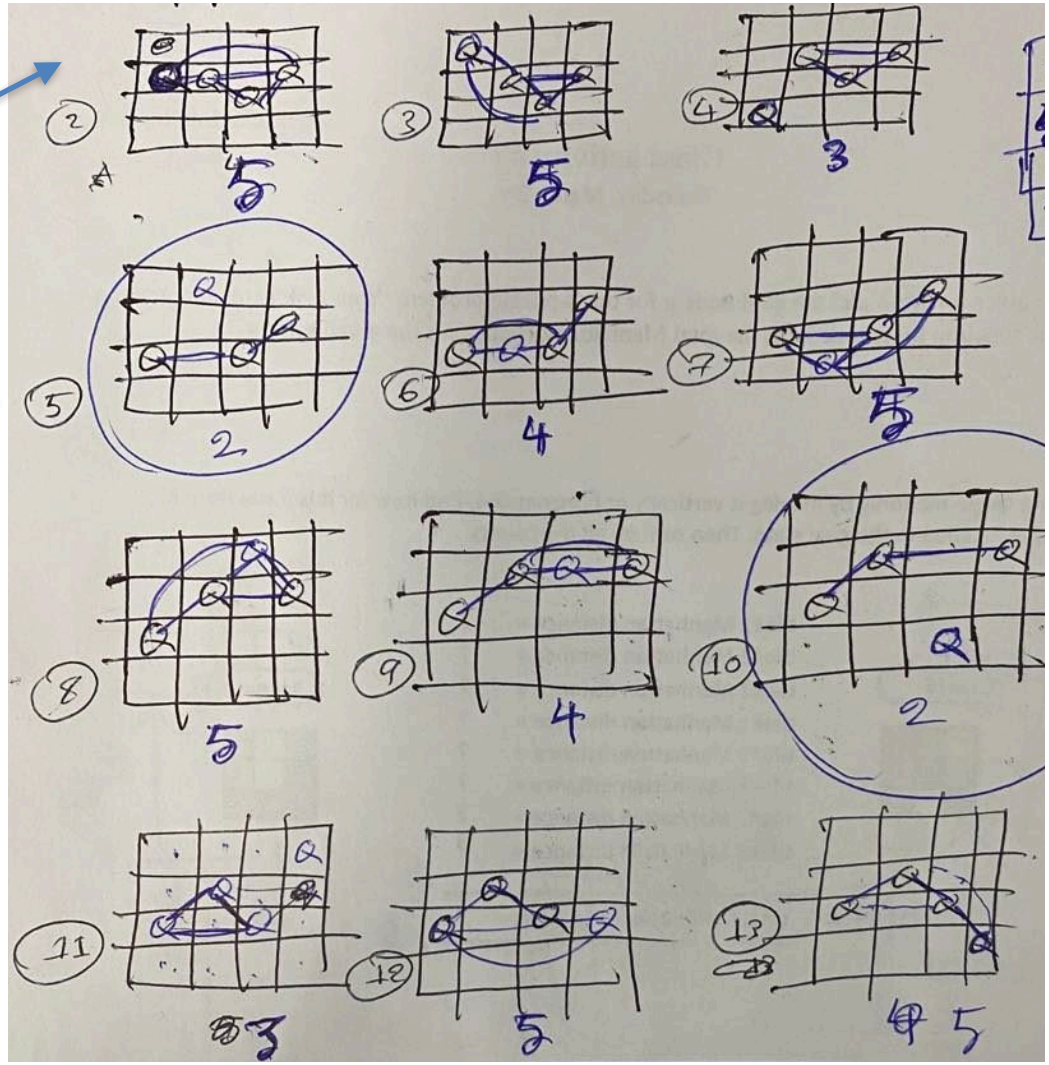
- What are the successor states from the one below?



successor$_1$

successor$_4$

successor$_{11}$

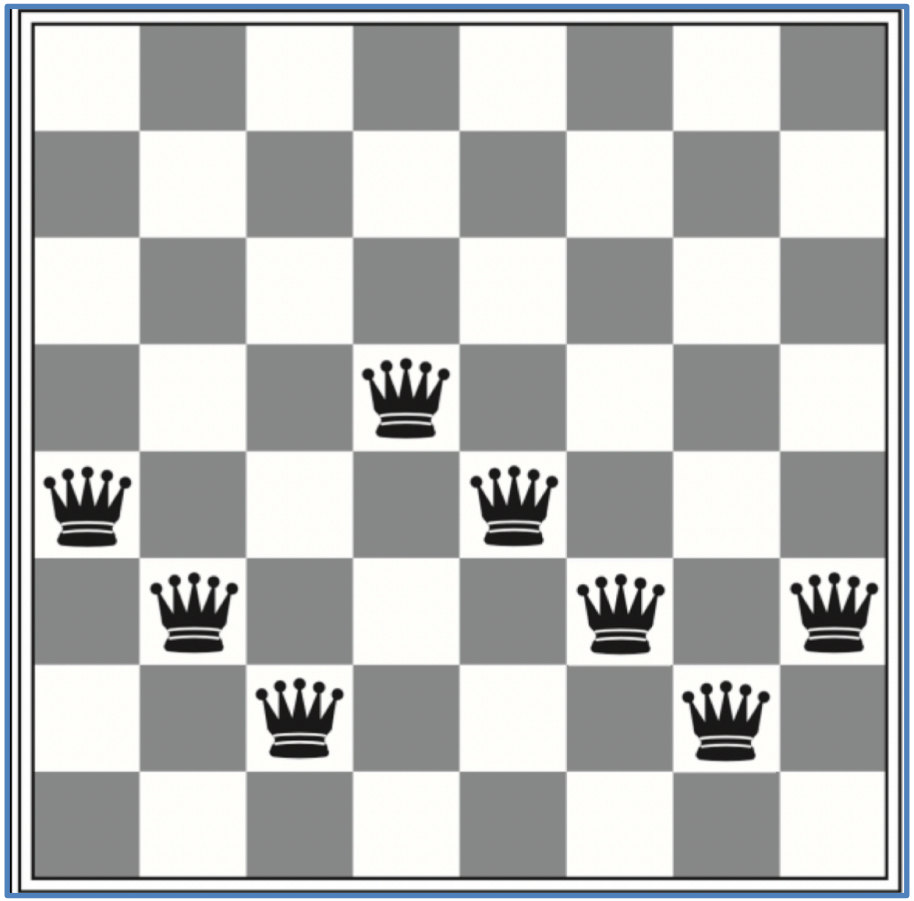


Successor states from the current state

- 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



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