

CSC 665: Artificial Intelligence

Local Search

Pooyan Fazli

San Francisco State University

Assignment 1: Feedback

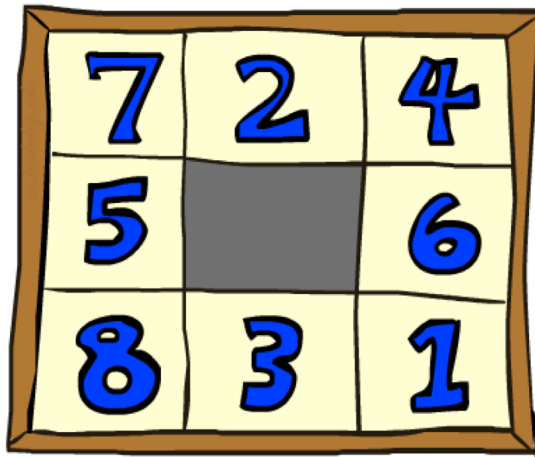
- You will fill in portions of `search.py` and `searchAgents.py` during the assignment.
- Your code will be autograded for technical correctness. Please do not change the names of any provided functions or classes within the code, or you will wreak havoc on the autograder.
- Make sure to use the `Stack`, `Queue` and `PriorityQueue` data structures provided to you in `util.py`! These data structure implementations have particular properties which are required for compatibility with the autograder.

Assignment 1: Feedback

- **Hint:** Each algorithm is very similar. Algorithms for DFS, BFS, UCS, and A* differ only in the details of how the frontier is managed. So, concentrate on getting DFS right and the rest should be relatively straightforward. Indeed, one possible implementation requires only a single generic search method which is configured with an algorithm-specific queuing strategy.

8-Puzzle

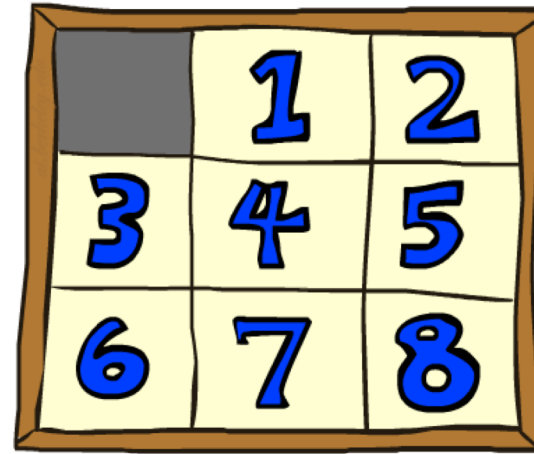
Path is Important!



A 3x3 grid representing the start state of an 8-puzzle. The tiles are numbered 1 through 8, with the middle cell (row 2, column 2) being empty and shaded gray. The numbers are in blue on a yellow background.

7	2	4
5		6
8	3	1

Start State



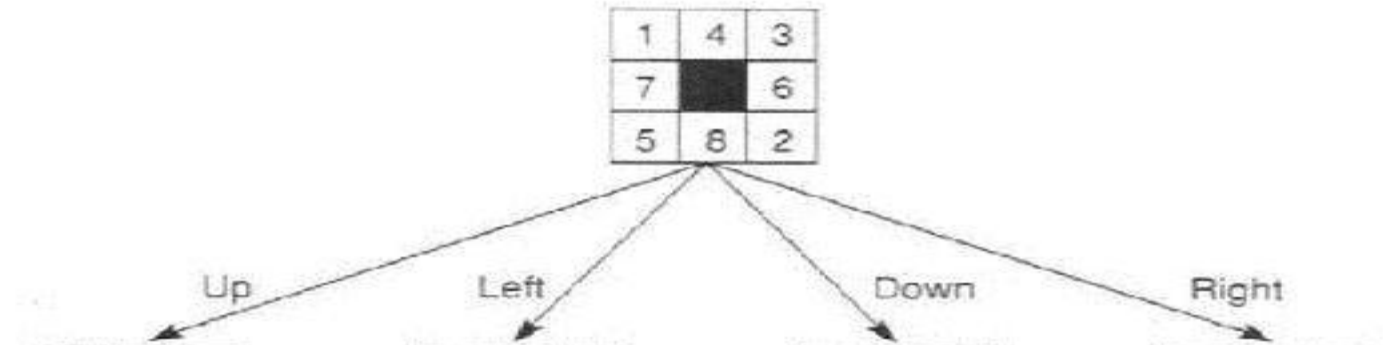
A 3x3 grid representing the goal state of an 8-puzzle. The tiles are numbered 1 through 8, with the top-left cell (row 1, column 1) being empty and shaded gray. The numbers are in blue on a yellow background.

	1	2
3	4	5
6	7	8

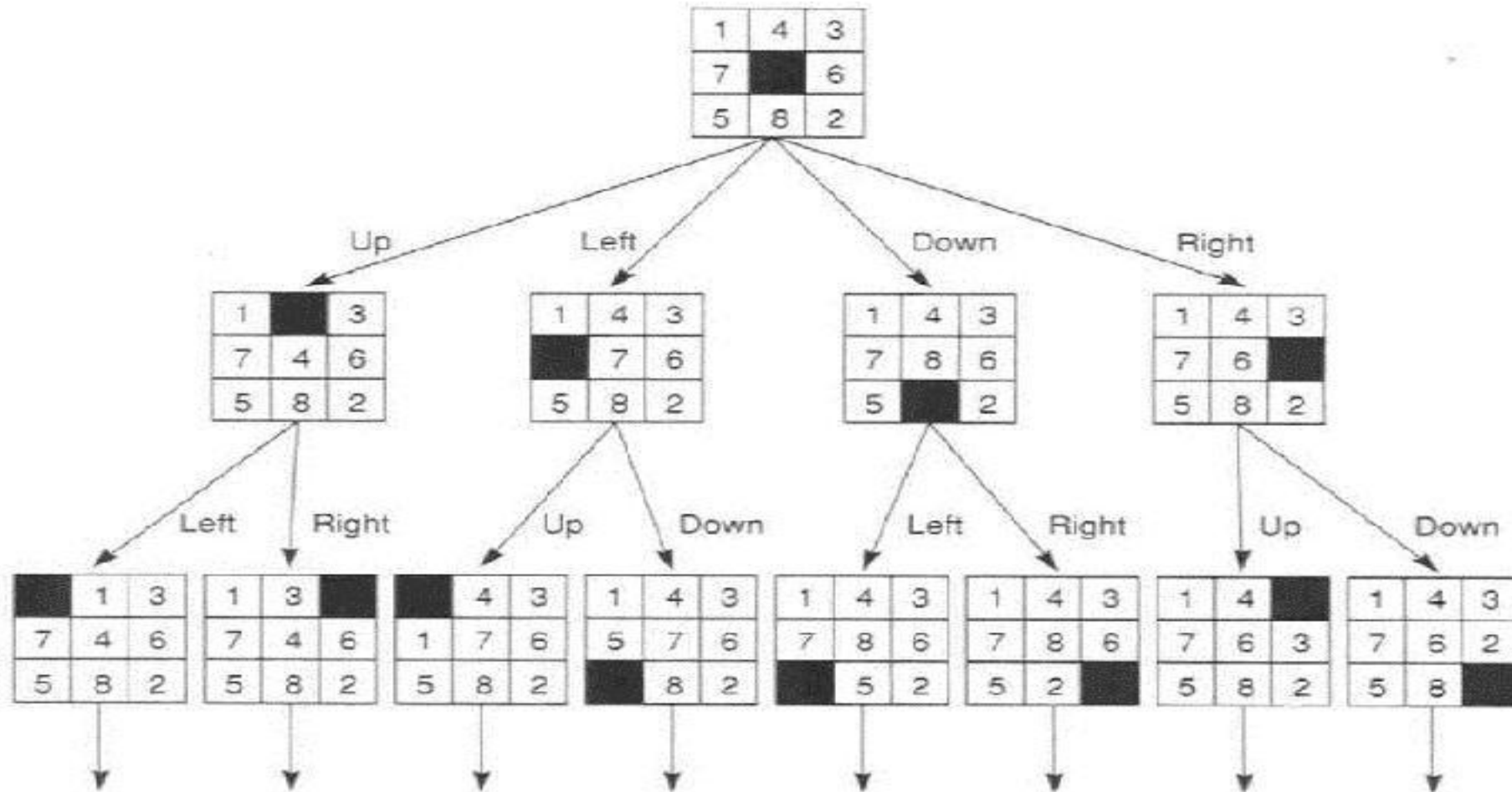
Goal State

When a goal is found, the *path* to that goal also constitutes a *solution* to the problem.

Search Space for 8-Puzzle

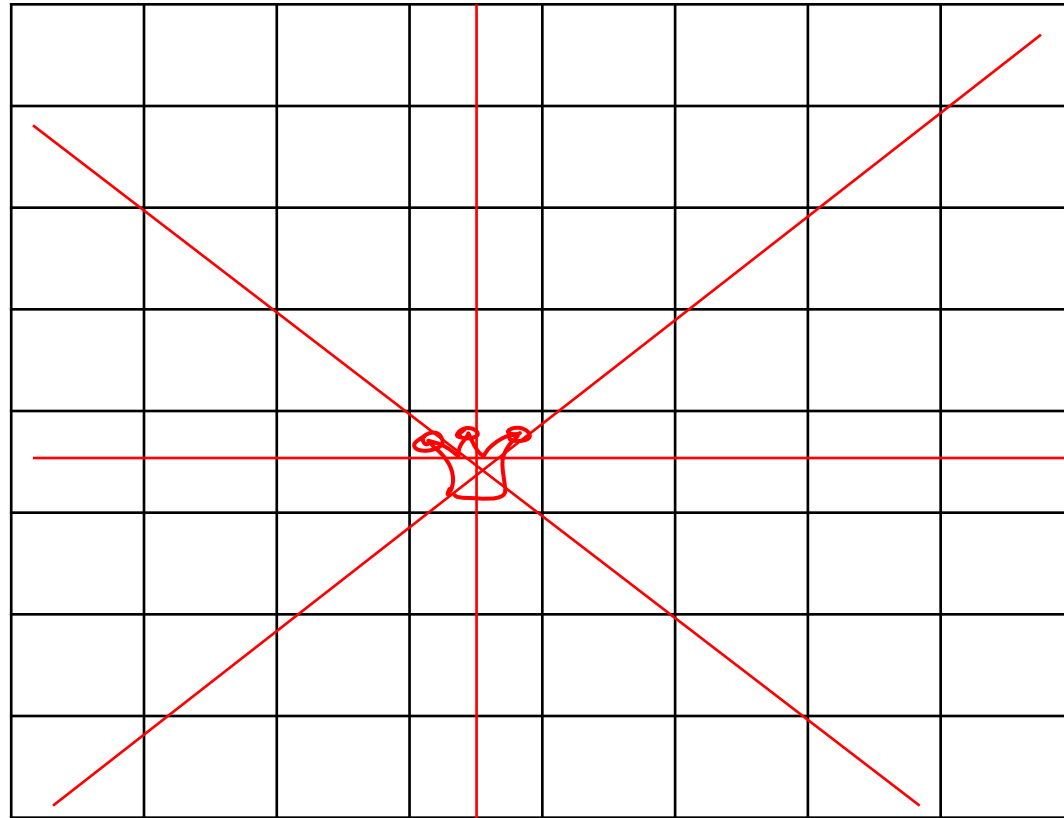


Search Space for 8-Puzzle



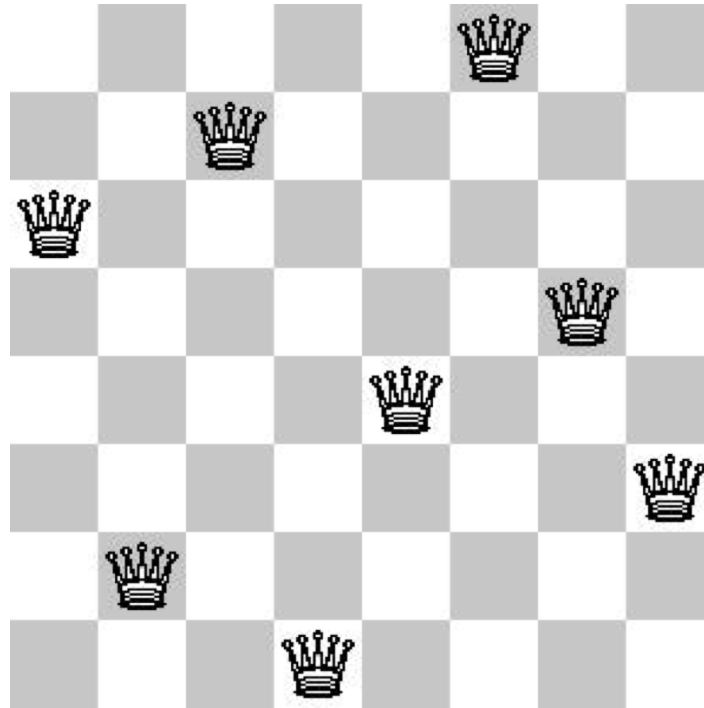
N-Queens

- Put n queens on an $n \times n$ board with **no two queens** on the same row, column, or diagonal (i.e, attacking each other)



N-Queens

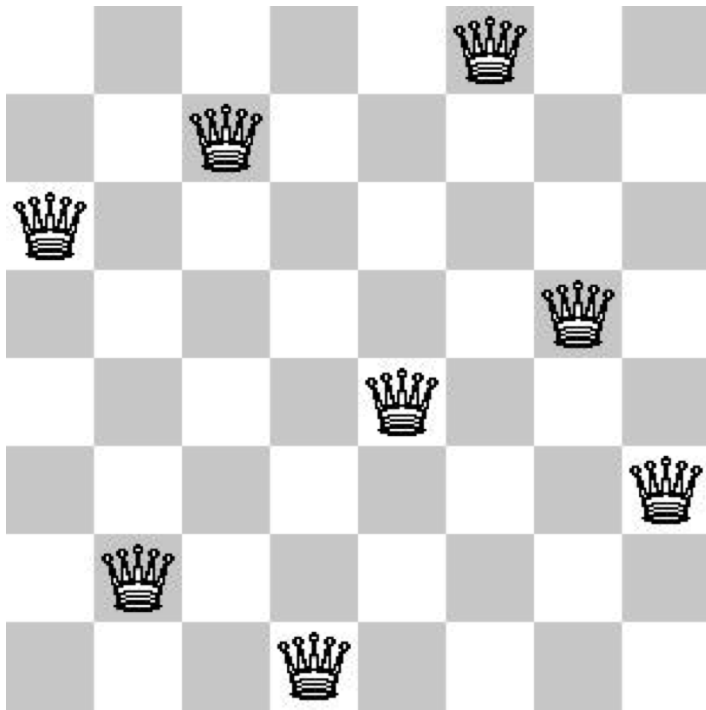
The path to the goal does not matter!



What matters is the final configuration of queens, not the order in which they are added.

N-Queens

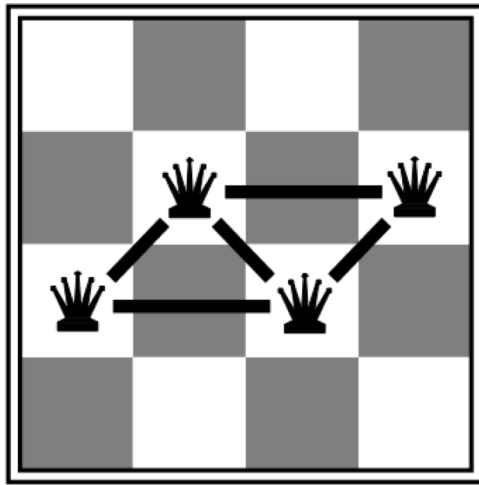
The path to the goal does not matter!



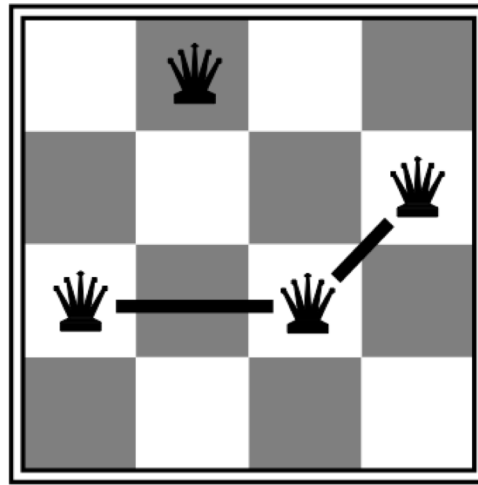
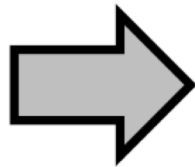
- Initial state = a “complete” configuration (e.g., 8 queens on the board, one per column).
- Then, we can use iterative improvement algorithms; keep a single “current” state, try to improve it

Objective Function for N-Queens

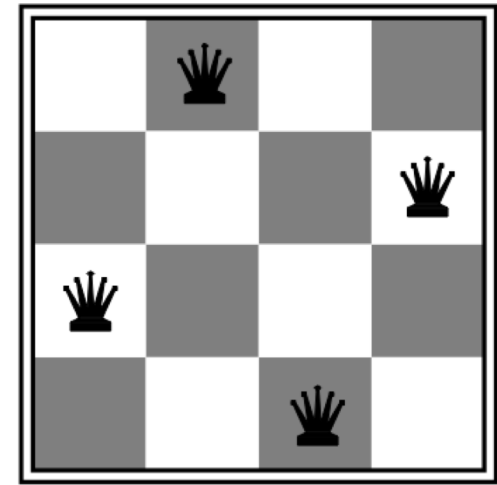
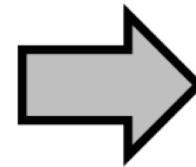
- Objective Function $h(n)$: number of conflicts



$h = 5$



$h = 2$



$h = 0$

Local Search

- Simple, general idea:
 - Start wherever
 - Repeat: move to the best neighboring state
 - If no neighbors better than current, quit
- What's bad about this approach?
 - Complete?
 - Optimal?
- What's good about it?
 - Fast and memory efficient!



Local Search

- **Local search:** improve a single option until you can't make it better.

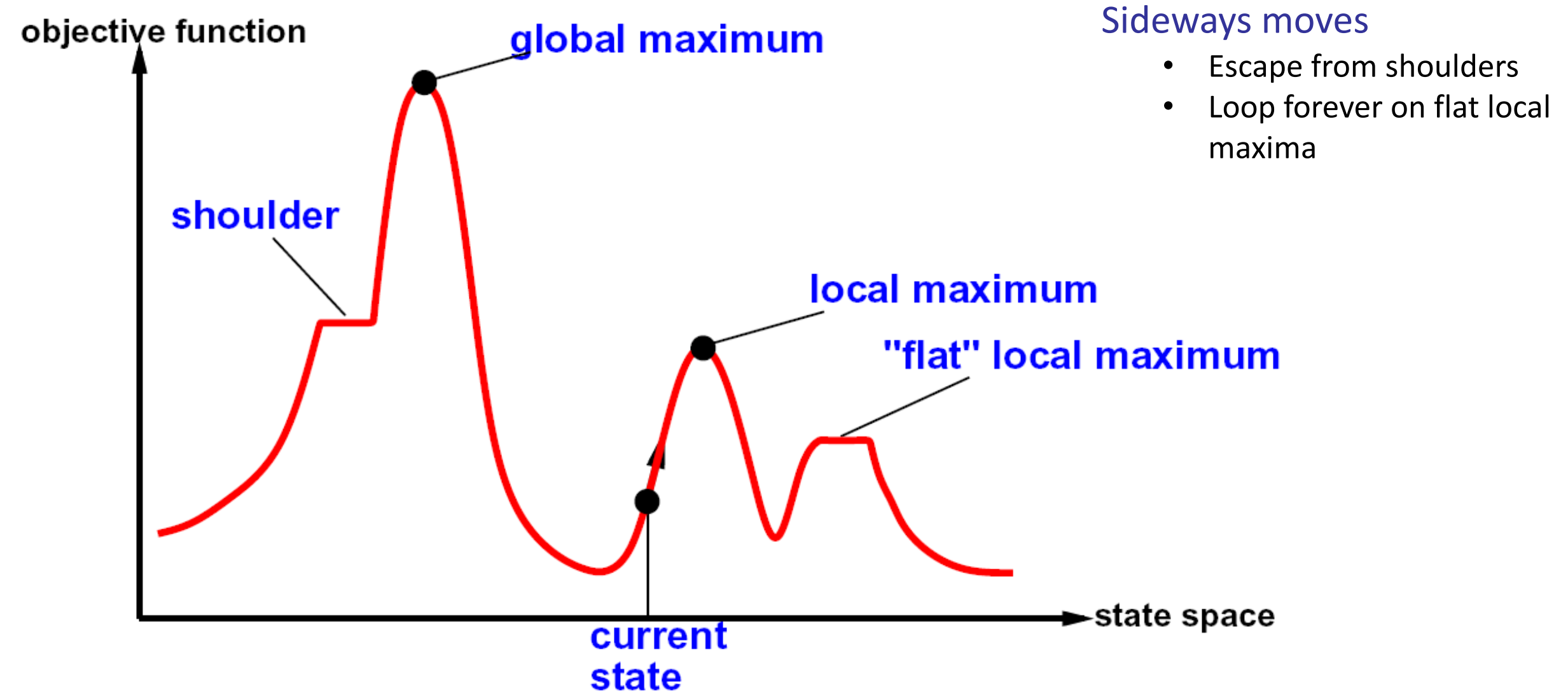
Local Search

- How to determine the neighbor node to be selected?
 - Select the neighbor that optimizes some objective function $h(n)$
 - e.g. Minimize the number of conflicts in N-Queens
- **Greedy Descent**: evaluate $h(n)$ for each neighbor, pick the neighbor n with minimal $h(n)$
- **Hill Climbing**: equivalent algorithm for maximization problems

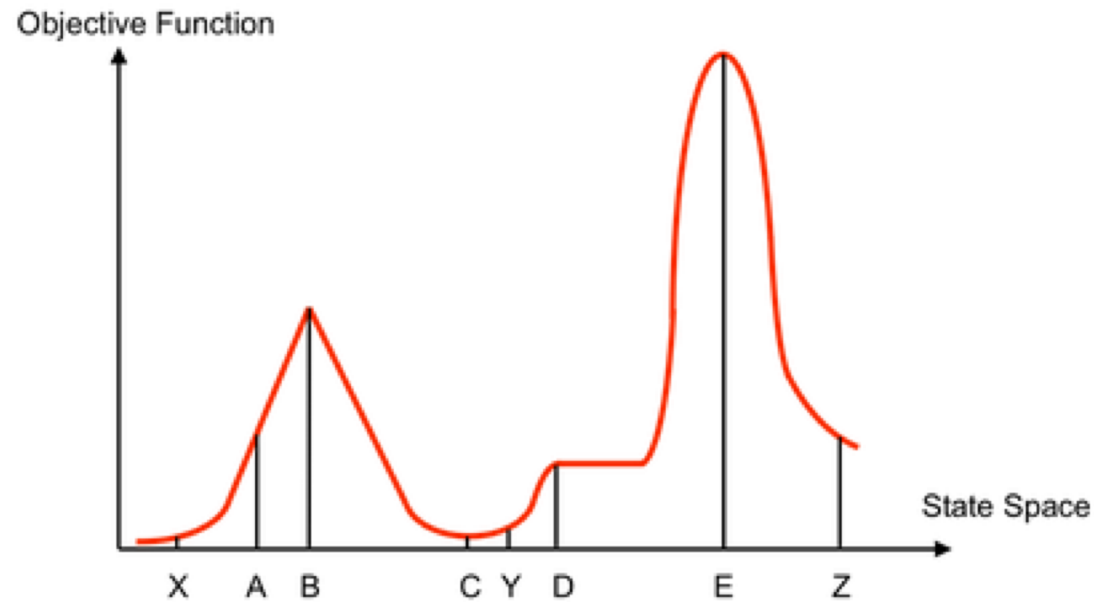
Hill-Climbing Algorithm

```
function HILL-CLIMBING(problem) returns a state
  current ← getStartState(problem)
  loop do
    neighbor ← a highest-valued successor of current
    if neighbor.value ≤ current.value then
      return current.state
    current ← neighbor
```

Hill Climbing Diagram



Hill Climbing Quiz



Starting from X, where do you end up ?

Starting from Y, where do you end up ?

Starting from Z, where do you end up ?

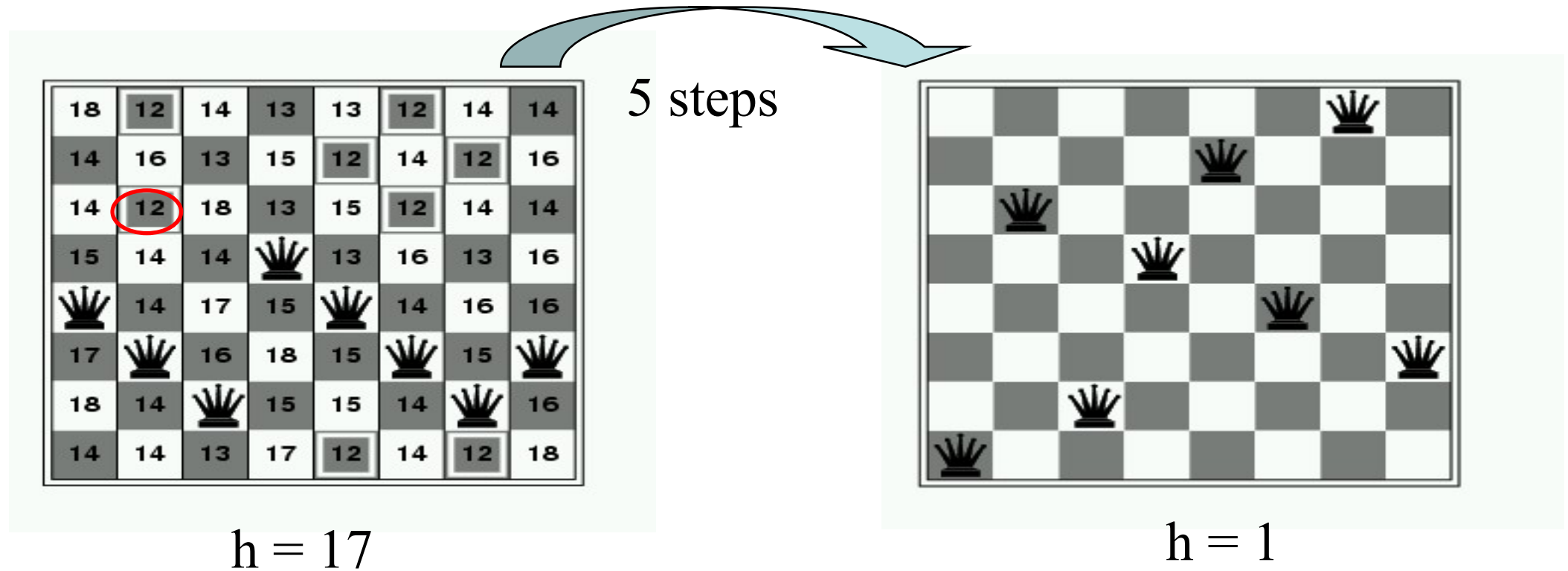
Greedy Descent on 8-Queens

- Randomly generated 8-queens starting states...
- 14% of the time it solves the problem
- 86% of the time it get stuck at a local minimum or a shoulder
- However...
 - Takes only 4 steps on average when it succeeds
 - And 3 on average when it gets stuck
 - (for a state space with $8^8 \approx 17$ million states)

Escaping Shoulders: Sideways Moves

- Move to a neighbor even if it is not strictly better (just equal). Allow sideways moves in the hope that algorithm can escape
 - Need to place a limit on the possible number of sideways moves to avoid infinite loops
- For 8-queens
 - Now allow sideways moves with a limit of 100
 - Raises percentage of problem instances solved from 14% to 94%
- However....
 - 21 steps for every successful solution
 - 64 for each failure

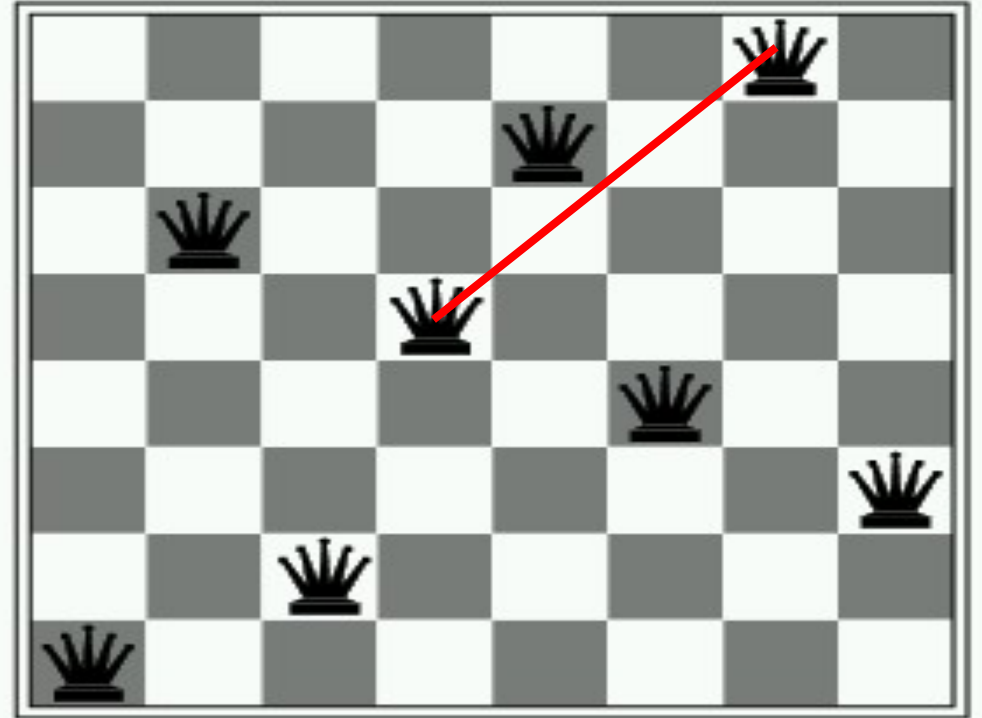
Example: 8-Queens



Each cell lists h (i.e. #conflicts) if you move the queen from that column into the cell

The Problem of Local Minima

- Which move should we pick in this situation?
 - Current cost: $h=1$
 - No single move can improve on this
 - In fact, every single move only makes things worse ($h \geq 2$)
- Since we are minimizing: **local minimum**



Stochastic Local Search

- We will use randomness to avoid getting trapped in local minima

Stochastic Local Search

- **Random Restart**

- Generate a state randomly. This lets the search start from a completely different part of the search space.

- **Random Step**

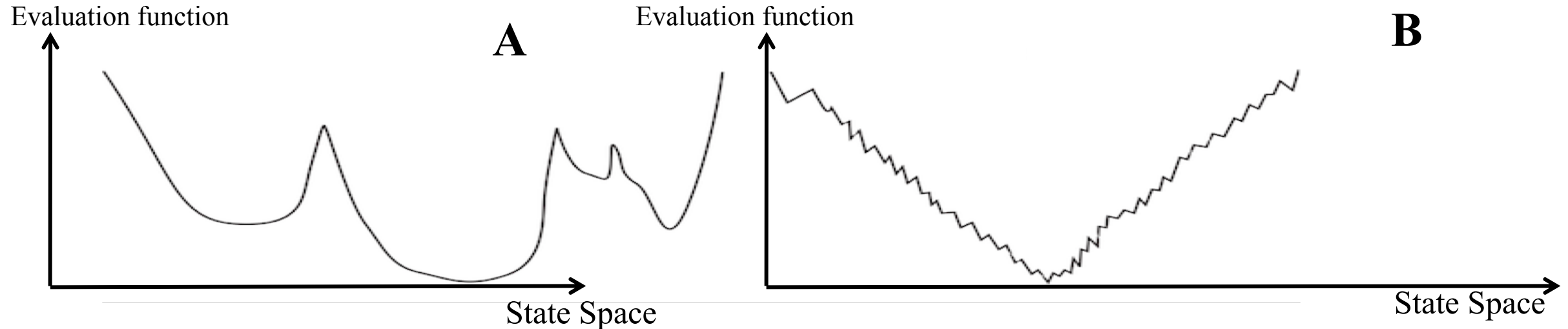
- Randomly pick a neighbor of the current state

Stochastic Local Search

- **Stochastic local search** is a mix of:
 - Greedy Descent: move to neighbor with lowest h
 - Random Restart: Generate a state randomly
 - Random Step: take some random steps

SLS Quiz

Which randomized method would work best in each of these two search spaces?



Greedy descent with random steps best on A

Greedy descent with random restart best on B

✓ Greedy descent with random steps best on B
Greedy descent with random restart best on A

Equivalent

Reading

- Chapter 4.1 in the ALMA textbook