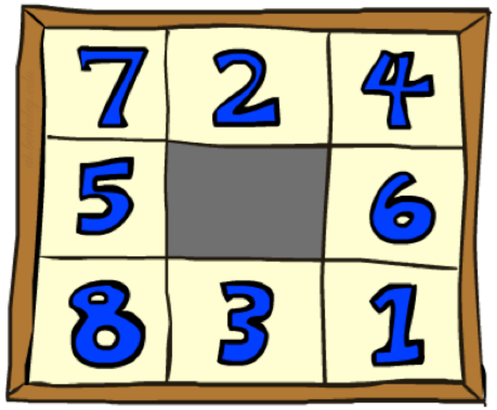
PURDUE CS47100

INTRODUCTION TO AI

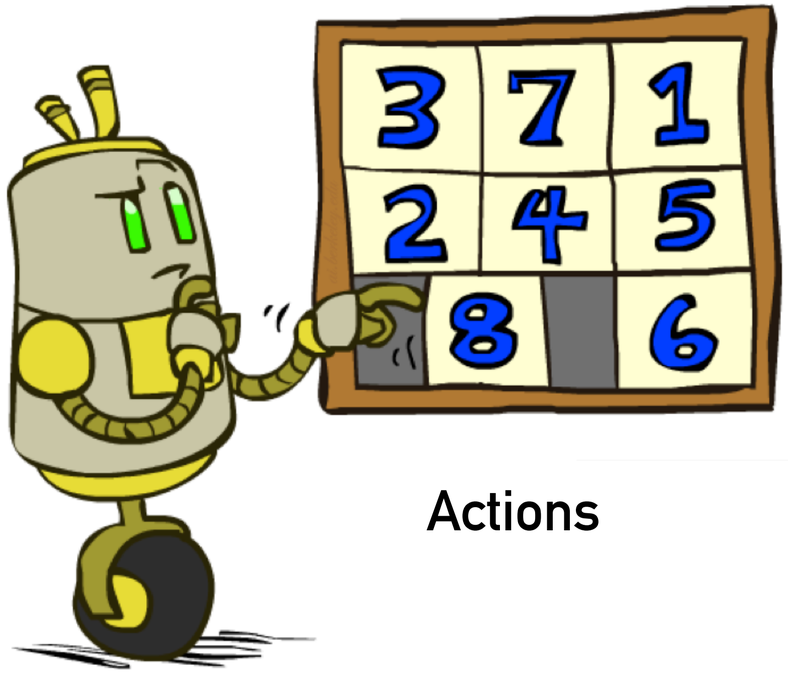
RECAP: COST-SENSITIVE SEARCH

- Uninformed search: Uniform cost search
  - Prioritize nodes with smaller cumulative cost g(n)
- Informed search
  - Greedy search: Prioritize nodes with smaller heuristic function value h(n)
  - A* search: Prioritize nodes with smaller evaluation function value f(n)=g(n)+h(n)

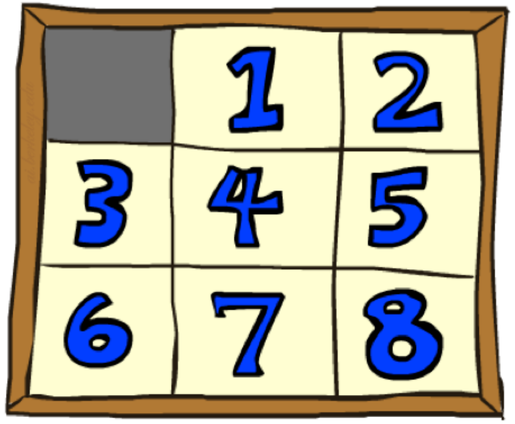
ADMISSIBLE HEURISTICS EXAMPLE: 8 PUZZLE



Start State



Actions



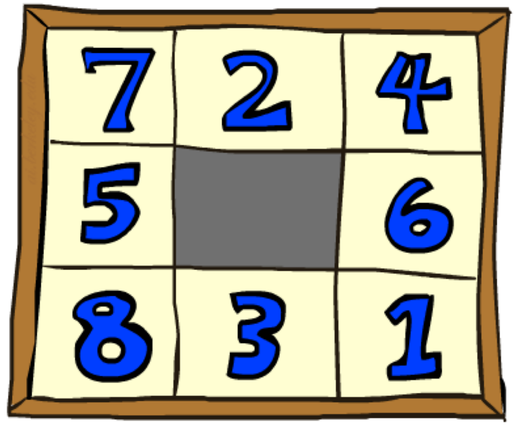
Goal State

What could be an admissible heuristic function for 8-puzzle problems?

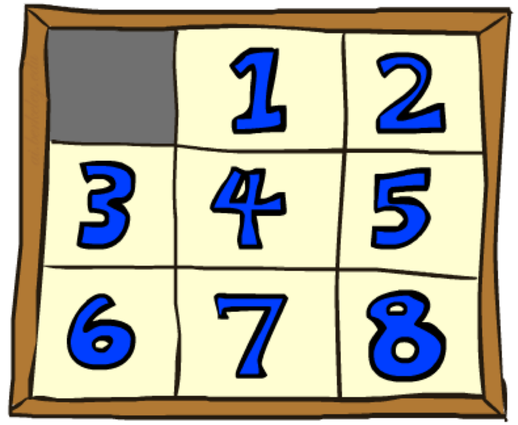
Hint: A tile can move from Location A to B if A and B is adjacent and B is blank…Relax it?

8 PUZZLE HEURISTIC I

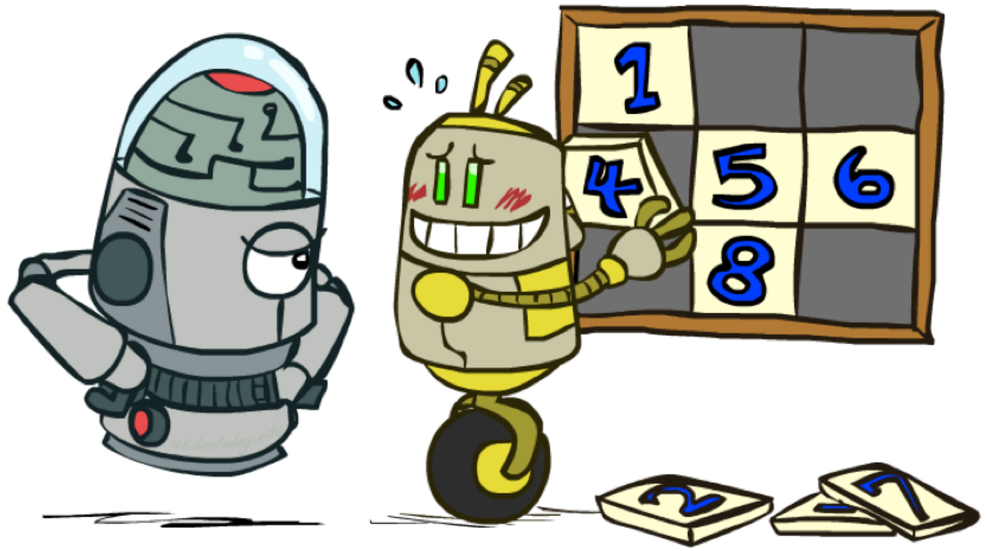
- Heuristic: Number of misplaced tiles
- Why is it admissible?
- h(start) = 8
- This is a *relaxed-problem* heuristic



Start State



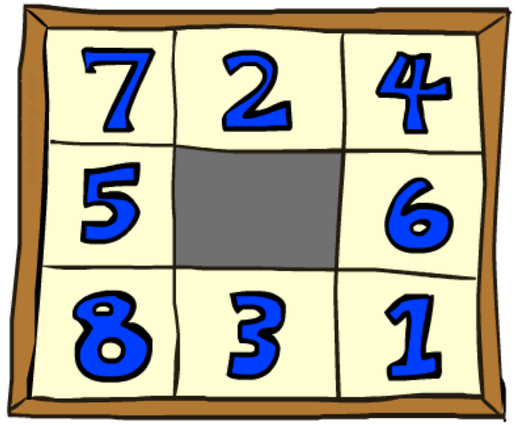
Goal State



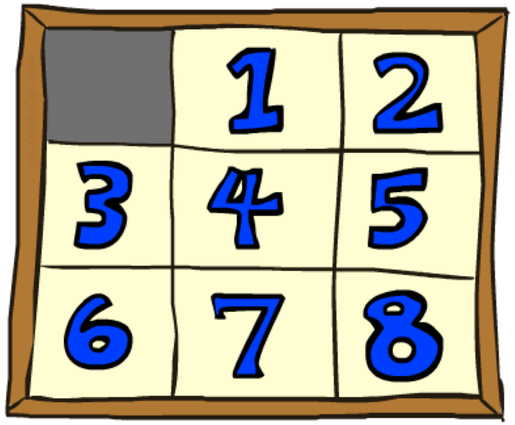
	Average nodes expanded when the optimal path has…		
	…4 steps	…8 steps	…12 steps
UCS	112	6,300	3.6×10^6
TILES	13	39	227

8 PUZZLE HEURISTIC II

- Heuristic: Sum of Manhattan distance for each tile
- Why is it admissible?
- h(start) = **3 + 1 + 2 + … = 18**
- This is still a *relaxed-problem* heuristic



Start State



Goal State

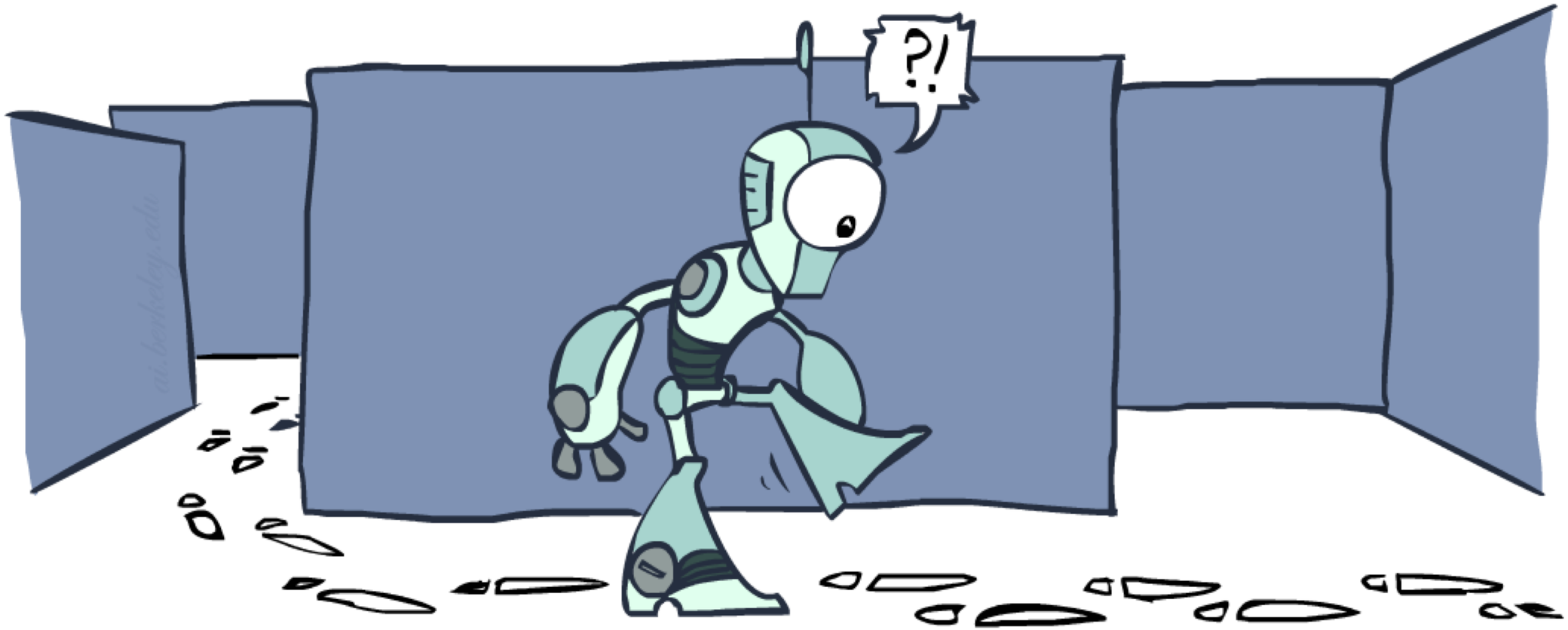
	Average nodes expanded when the optimal path has…		
	…4 steps	…8 steps	…12 steps
TILES	13	39	227
MANHATTAN	12	25	73

8 PUZZLE III

- How about using the *actual cost* as a heuristic?
  - Would it be admissible?
  - Would we save on nodes expanded?
  - What's wrong with it?

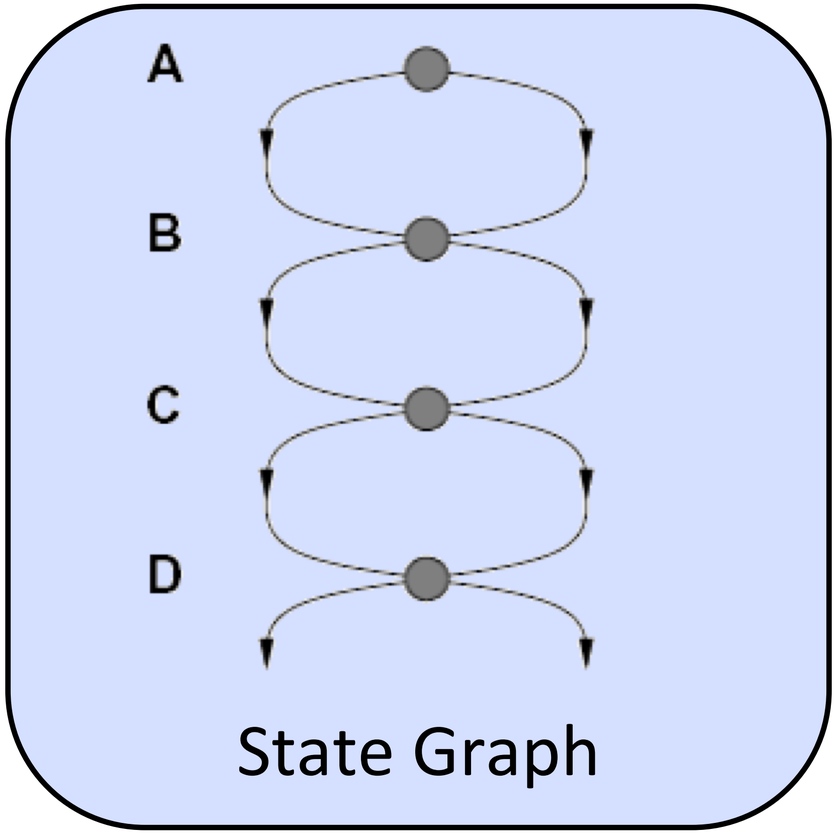
- With A*: a trade-off between quality of estimate and work per node
  - As heuristics get closer to the true cost, you will expand fewer nodes but usually do more work per node to compute the heuristic itself
  - Max of admissible heuristics is admissible, and it is a "better" heuristic

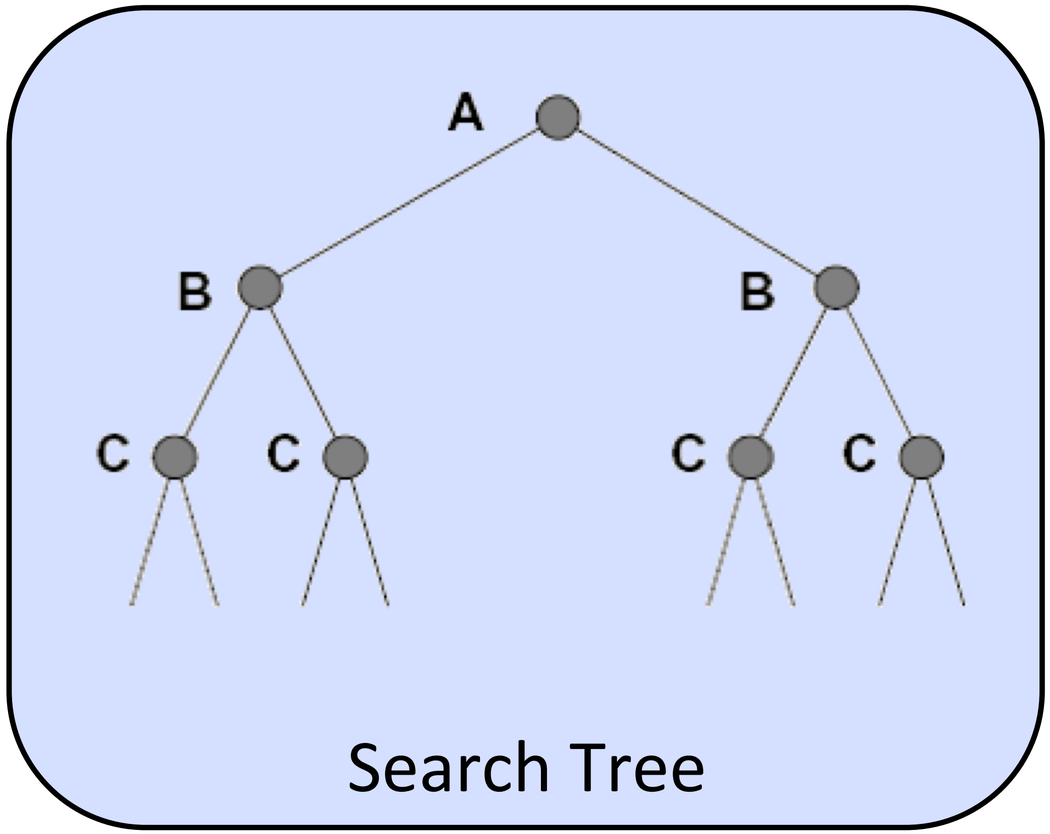
GRAPH SEARCH



TREE SEARCH: EXTRA WORK

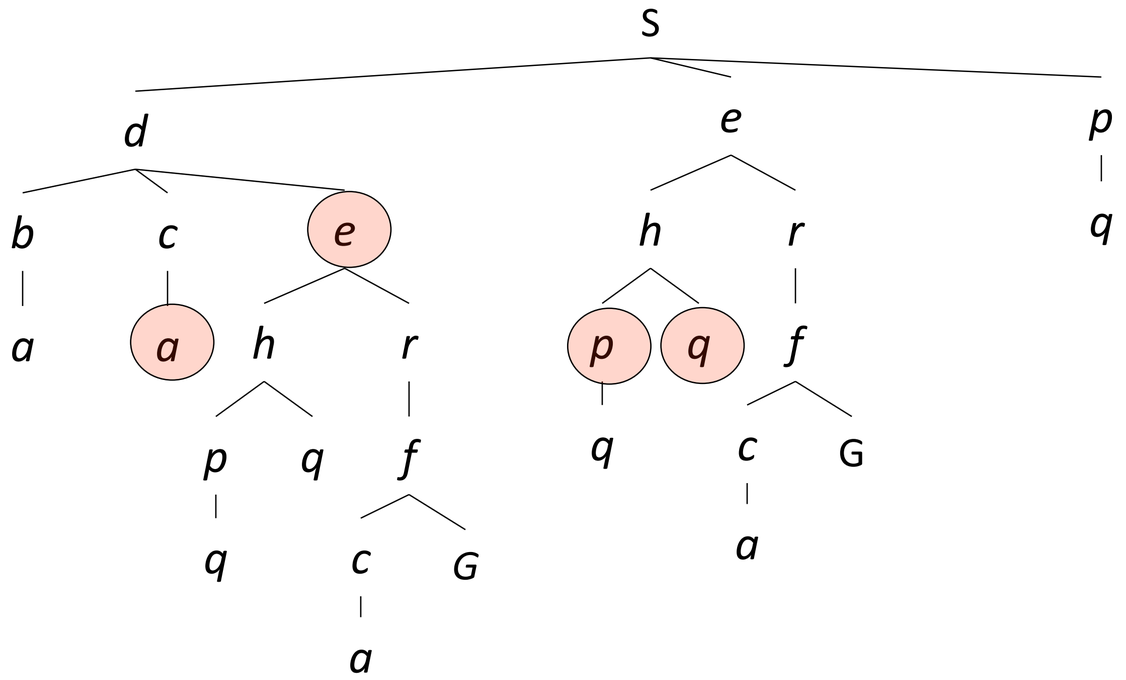
- Failure to detect repeated states can cause exponentially more work





GRAPH SEARCH

- In BFS, for example, we shouldn't bother expanding the circled nodes (why?)



GRAPH SEARCH

- Idea: never **expand** a state twice
- How to implement:
  - Tree search + set of expanded states (“**closed set**”)
  - Expand the search tree node-by-node, but…
  - Before expanding a node, check to make sure its state has never been expanded before (i.e., not in “closed set”)
  - If not new, skip it, if new add to closed set
- Important: **store the closed set as a set**, not a list

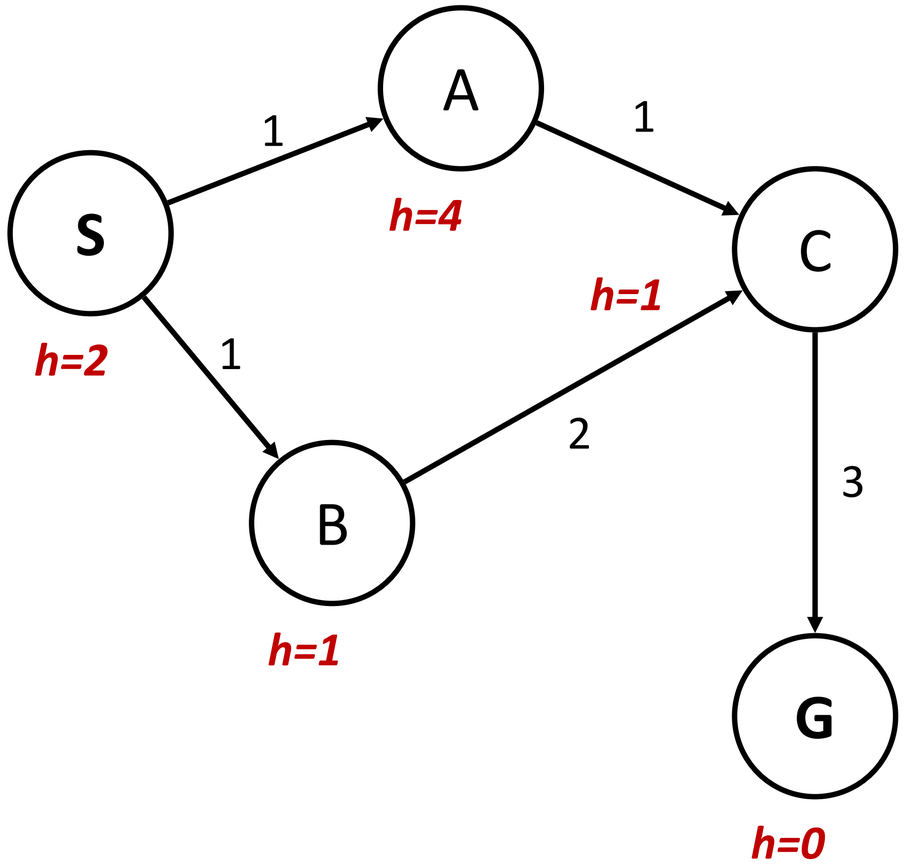
GRAPH SEARCH ALGORITHMS

```
function GRAPH_SEARCH (problem) return a solution, or failure
    fringe ← {initial_state}; explored ← ∅
    repeat
        if fringe = ∅ then return failure
        node ← POP(fringe)
        if GOAL_TEST(node)  then return the corresponding solution
        if node not in explored:
            explored ←  explored ∪ {node}
            for a in Action(node):
                a' = Result(node, a)
                INSERT(fringe, a')
```

Can graph search wreck completeness?

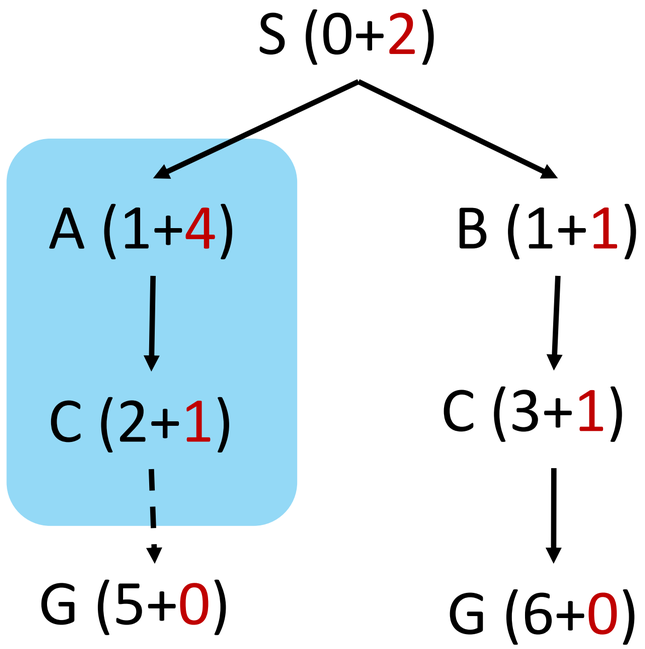
How about optimality?

A* GRAPH SEARCH GONE WRONG?



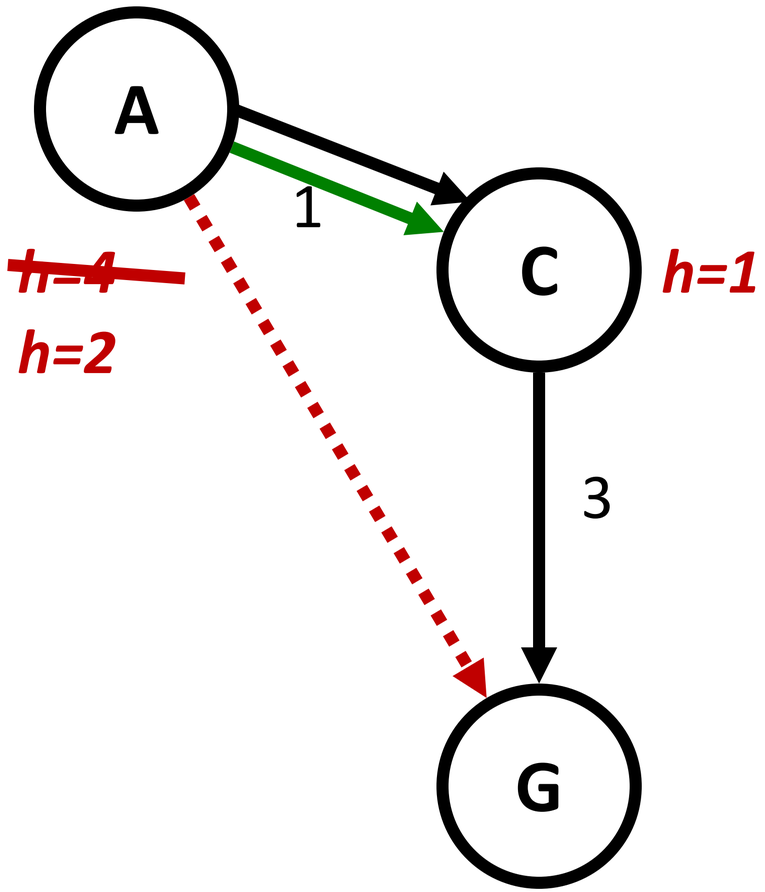
State space graph

f(n) is decreasing



Search tree

CONSISTENCY OF HEURISTICS



- Main idea: estimated heuristic costs ≤ actual costs
  - Admissibility: heuristic cost ≤ actual cost to goal

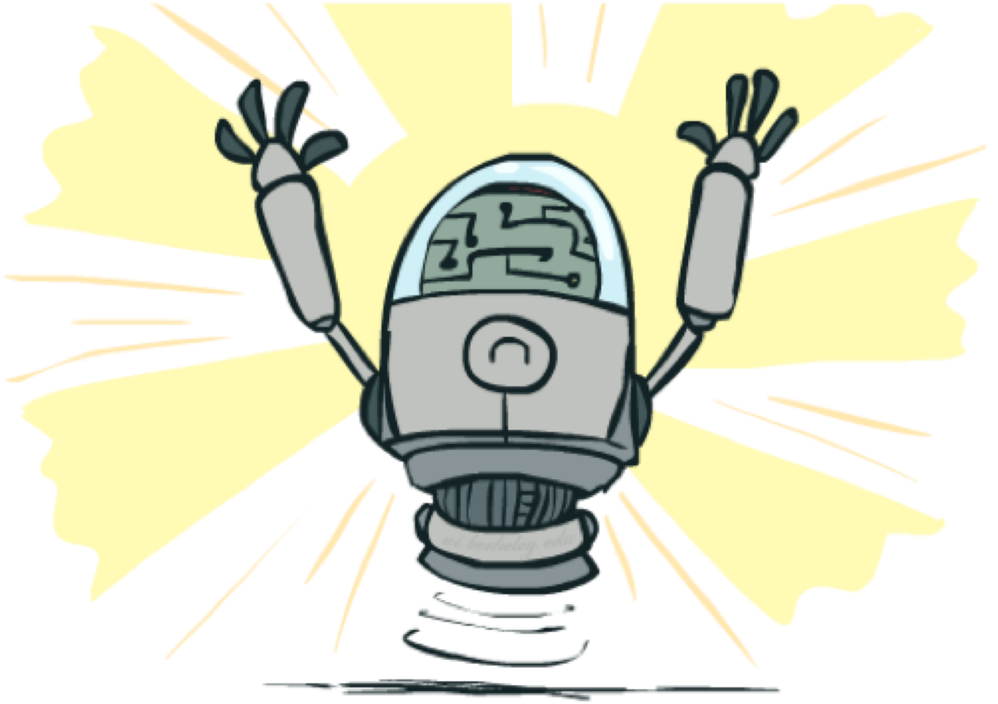
    h(A) ≤ actual cost from A to G
  - Consistency: heuristic "arc" cost ≤ actual cost for each arc

    h(A) – h(C) ≤ cost(A to C)

    h(A) ≤ cost(A to C) + h(C)
- Consequences of consistency:
  - The f value along a path never decreases
  - A* graph search is optimal

OPTIMALITY

- Tree search:
  - A* is optimal if heuristic is admissible
  - UCS is a special case (h = 0)

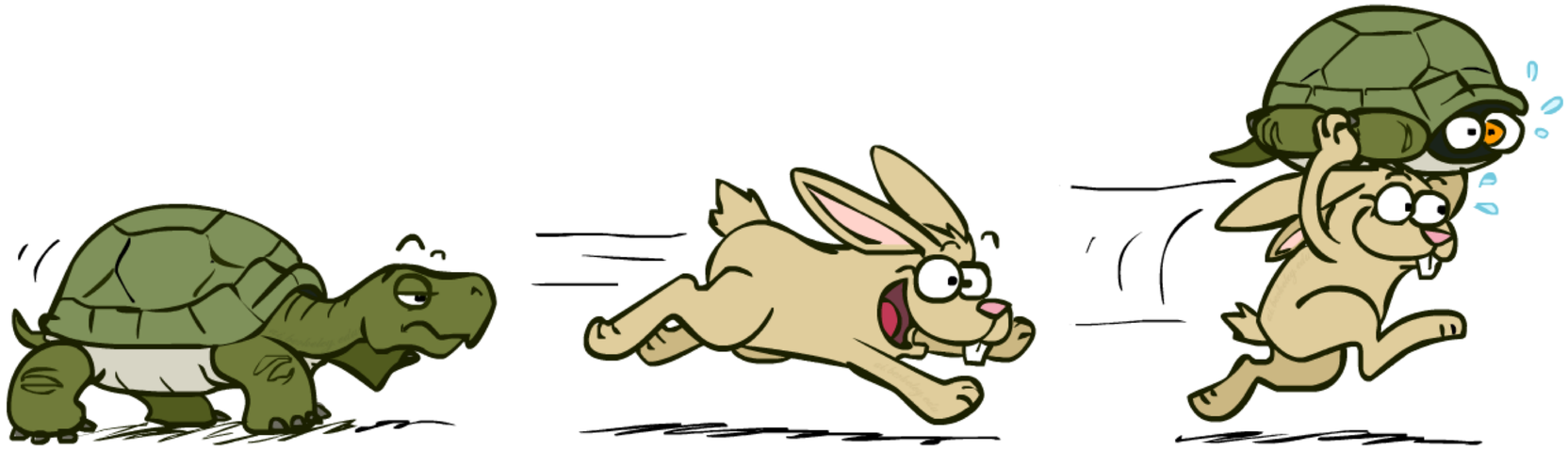


- Graph search:
  - A* optimal if heuristic is consistent
  - UCS optimal (h = 0 is consistent)
- Consistency implies admissibility
- In general, most natural admissible heuristics tend to be consistent, especially if from relaxed problems

A*: SUMMARY

A*: SUMMARY

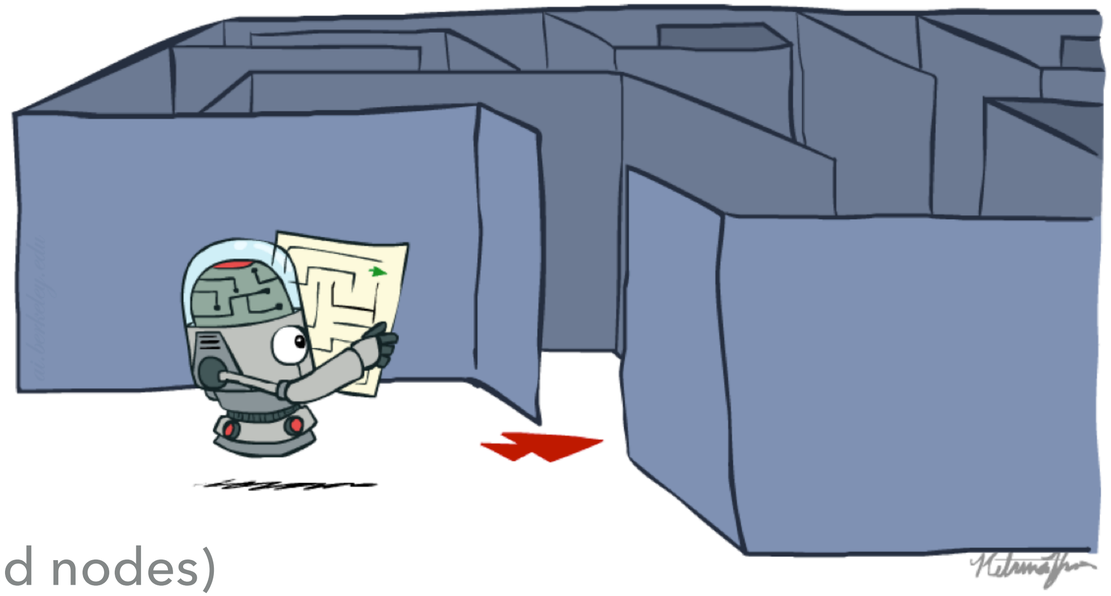
- A* uses both backward costs and (estimates of) forward costs
- A* is optimal with admissible / consistent heuristics
- Heuristic design is key: often use relaxed problems



RECAP: SYSTEMATIC SEARCH

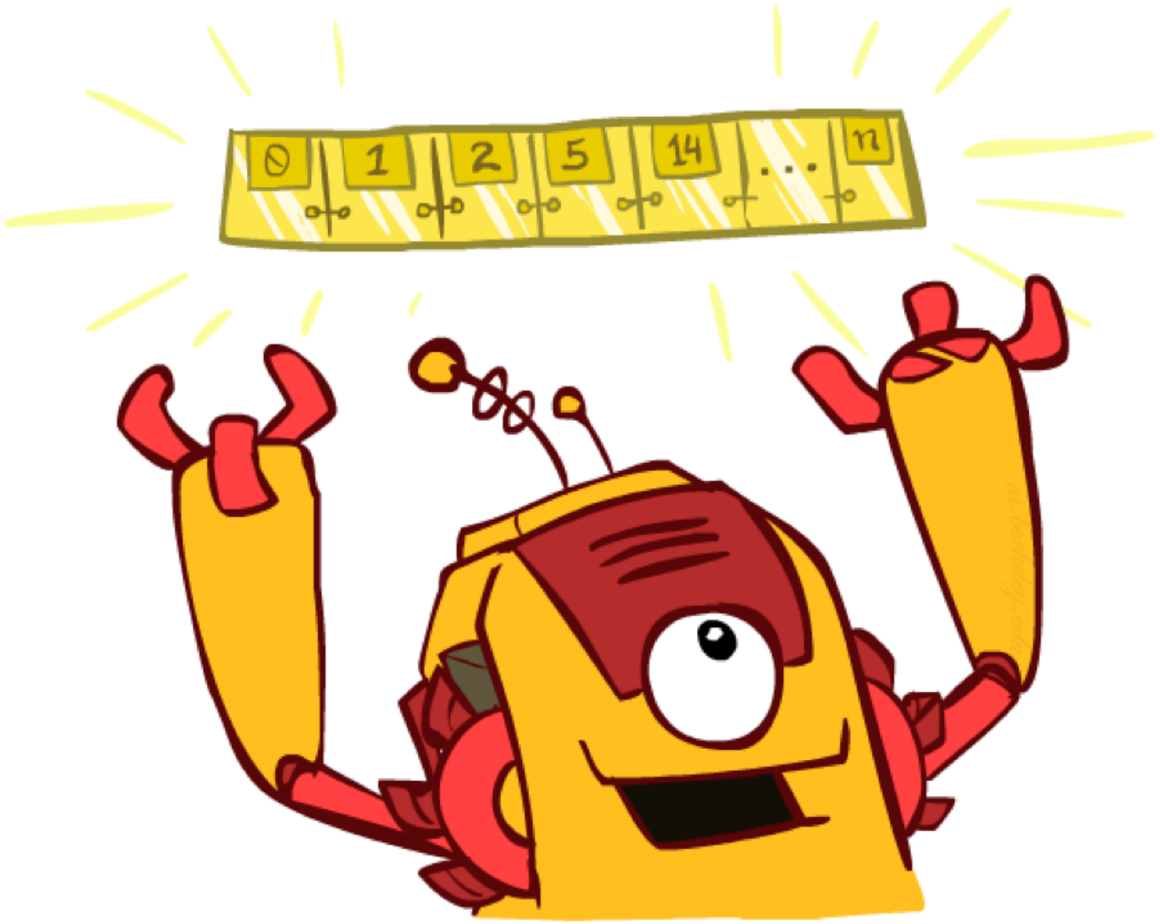
- Search tree:
  - Nodes: represent partial plans
  - Plans have costs (sum of action costs)

- Search algorithm:
  - Systematically builds a search tree
  - Chooses an ordering of the fringe (unexplored nodes)
  - Optimal: finds least-cost plans



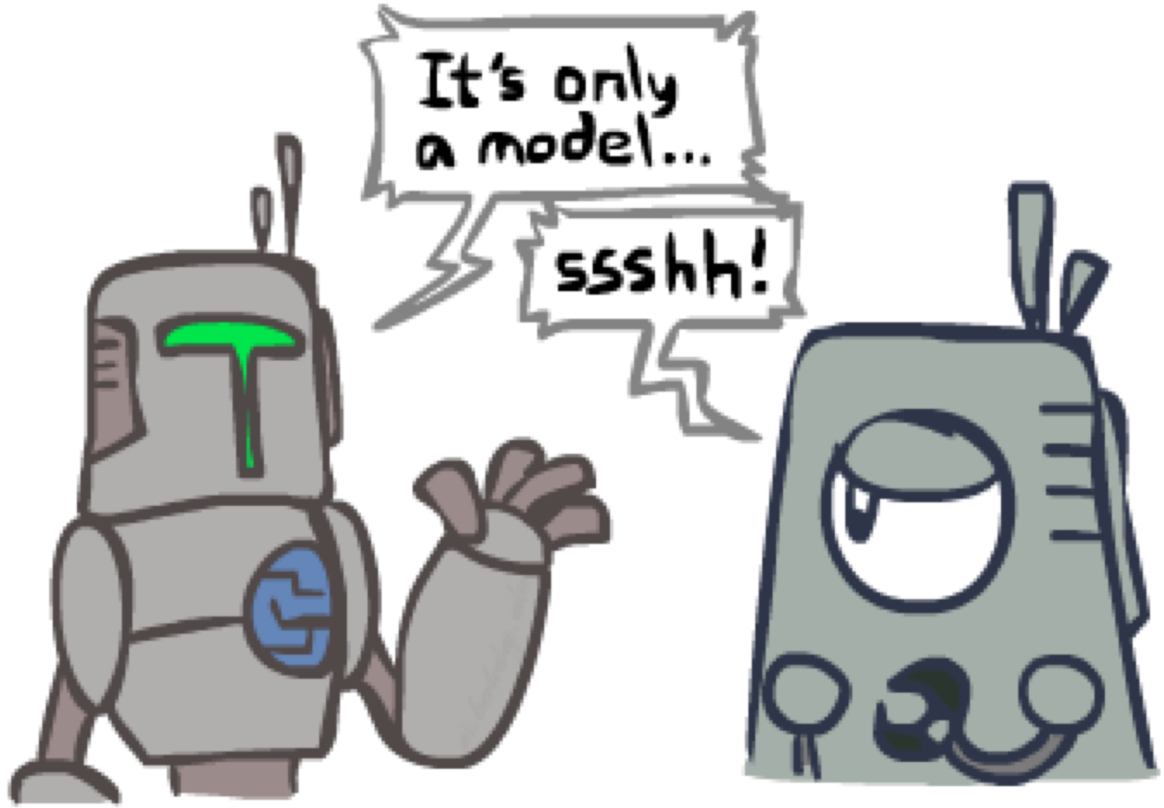
ONE SEARCH QUEUE

- All these search algorithms are the same except for fringe strategies
  - Conceptually, all fringes are priority queues (i.e. collections of nodes with attached priorities)
  - Practically, for DFS and BFS, you can avoid the log(n) overhead from an actual priority queue, by using stacks and queues
  - Can even code one implementation that takes a variable queuing object



SEARCH AND MODELS

- Search operates over models of the world
  - The agent doesn't actually try all the plans out in the real world
  - Planning is all "in simulation"
  - Your search is only as good as your models…



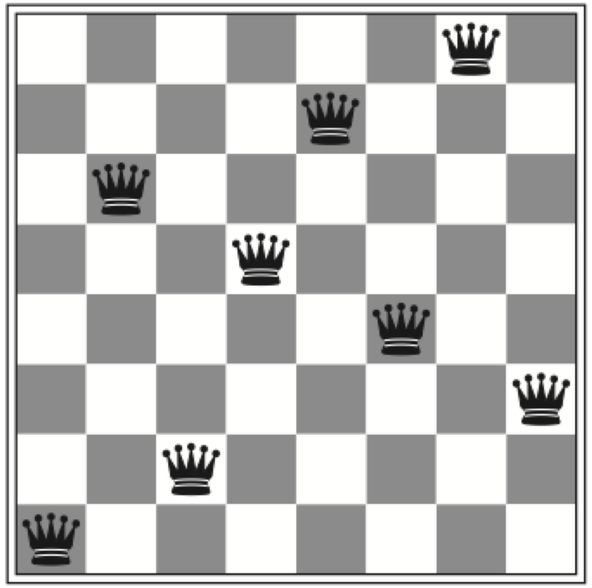
FROM PATH-BASED TO LOCAL SEARCH

- So far we have considered search methods that:
  - Systematically explore the full search space
  - Return the path from start node to the goal
- But in many problems, path is irrelevant; the goal state itself is the only thing you care about. Example applications:
  - 8-queens problems
- Sometimes the goal test itself is unclear, e.g., you are really solving an optimization problem. Example applications:
  - Traveling salesperson

LOCAL SEARCH

- Useful when path to goal state does not matter / solving pure optimization problem
- Basic idea:
  - Only keep a single “current” state
  - Try to improve iteratively
  - Don’t save paths followed
- Characteristics
  - Low memory requirements—usually constant
  - Effective—can often find good solutions in extremely large state spaces

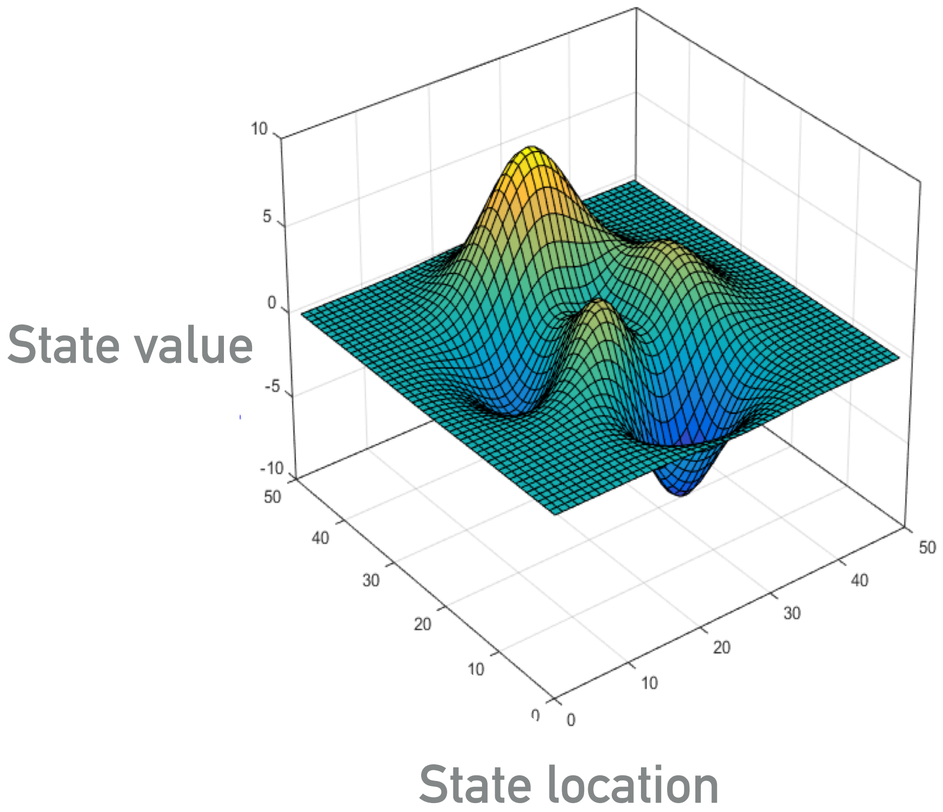
EXAMPLE: EIGHT-QUEENS PROBLEM



- Place eight queens on a chessboard so that no queen can attack another
  - Queen can move in the same row, column, or diagonally
- How to formulate search?
  - Incremental formulation
  - Complete-state formulation
- Heuristic/Objective: number of pairs of attacking queens

HILL-CLIMBING SEARCH

- Consider state-space landscape where:
  - Location=state
  - Elevation=evaluation of state (objective function value)
- Continually move in direction of increasing value (if trying to maximize objective function)



HILL-CLIMBING SEARCH

```
function HILL_CLIMBING (problem) return a state that is a local maximum
    current ← initial_state;
    repeat:
        best_neighbor ← current
        for state in Neighbors(current):
            if state.value > best_neighbor.value:
                best_neighbor ← state
        if best_neighbor.value > current.value:
            current ← best_neighbor
        else:
            return current
```

HILL-CLIMBING EXAMPLE: EIGHT-QUEENS PROBLEM

Heuristic/Objective:
number of pairs of attacking queens

CHALLENGES FOR HILL-CLIMBING

- Local maxima
  - Once a local maximum is reached, there is no way to backtrack or move out of that maximum

Image: http://www.ndsu.nodak.edu/instruct/juell/vp/cs724s00/hill_climbing/

CHALLENGES FOR HILL-CLIMBING

- Plateaux
  - Hillclimbing can have difficult time finding its way off of a flat portion of the state space landscape

Image: http://www.ndsu.nodak.edu/instruct/juell/vp/cs724s00/hill_climbing/

CHALLENGES FOR HILL-CLIMBING

- Ridges
  - Ridges can produce a series of local maxima that are difficult to navigate out of

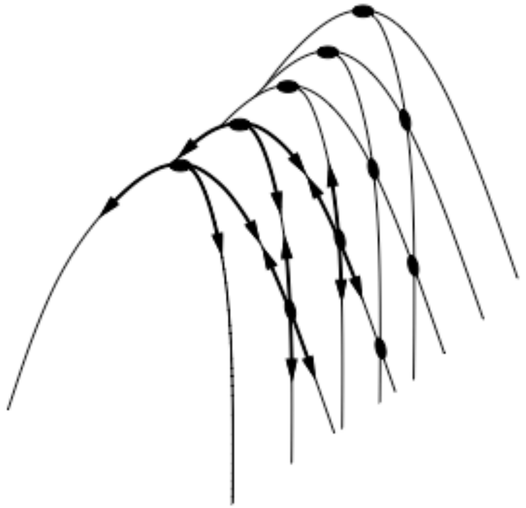
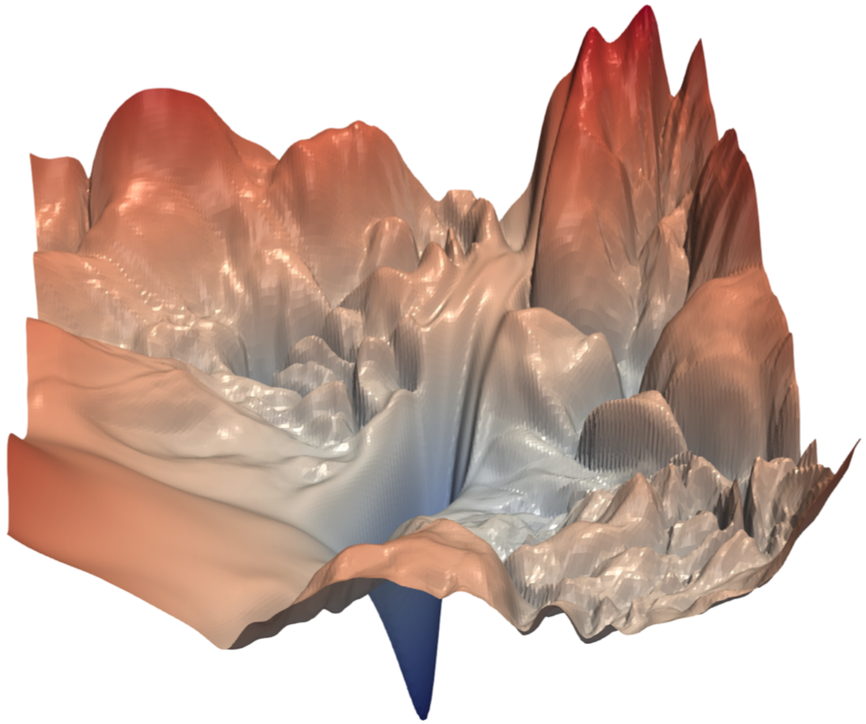


Image: http://www.ndsu.nodak.edu/instruct/juell/vp/cs724s00/hill_climbing/

STATE SPACE LANDSCAPES IN PRACTICE (FOR NEURAL NETWORKS)

State value



State location