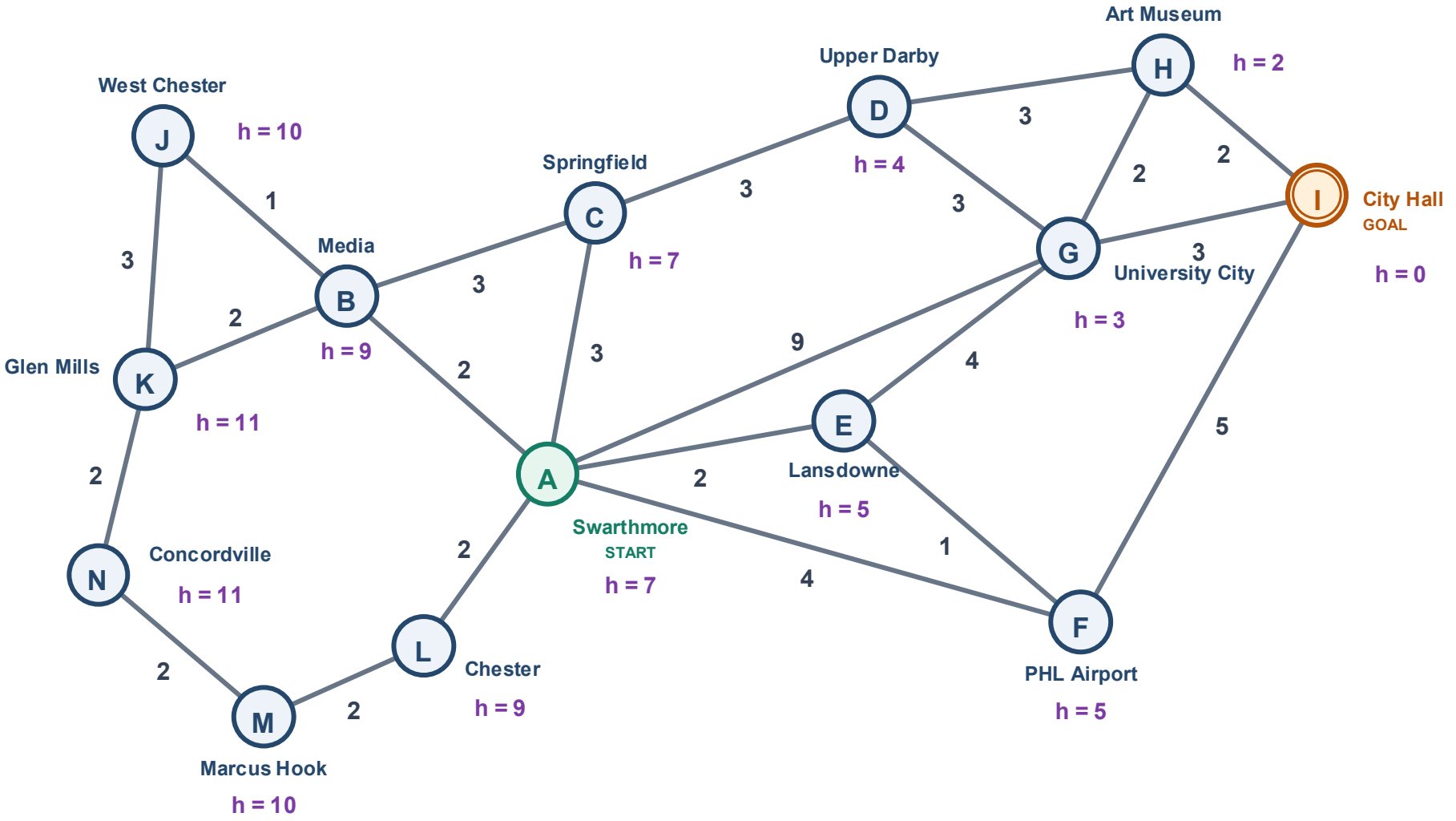
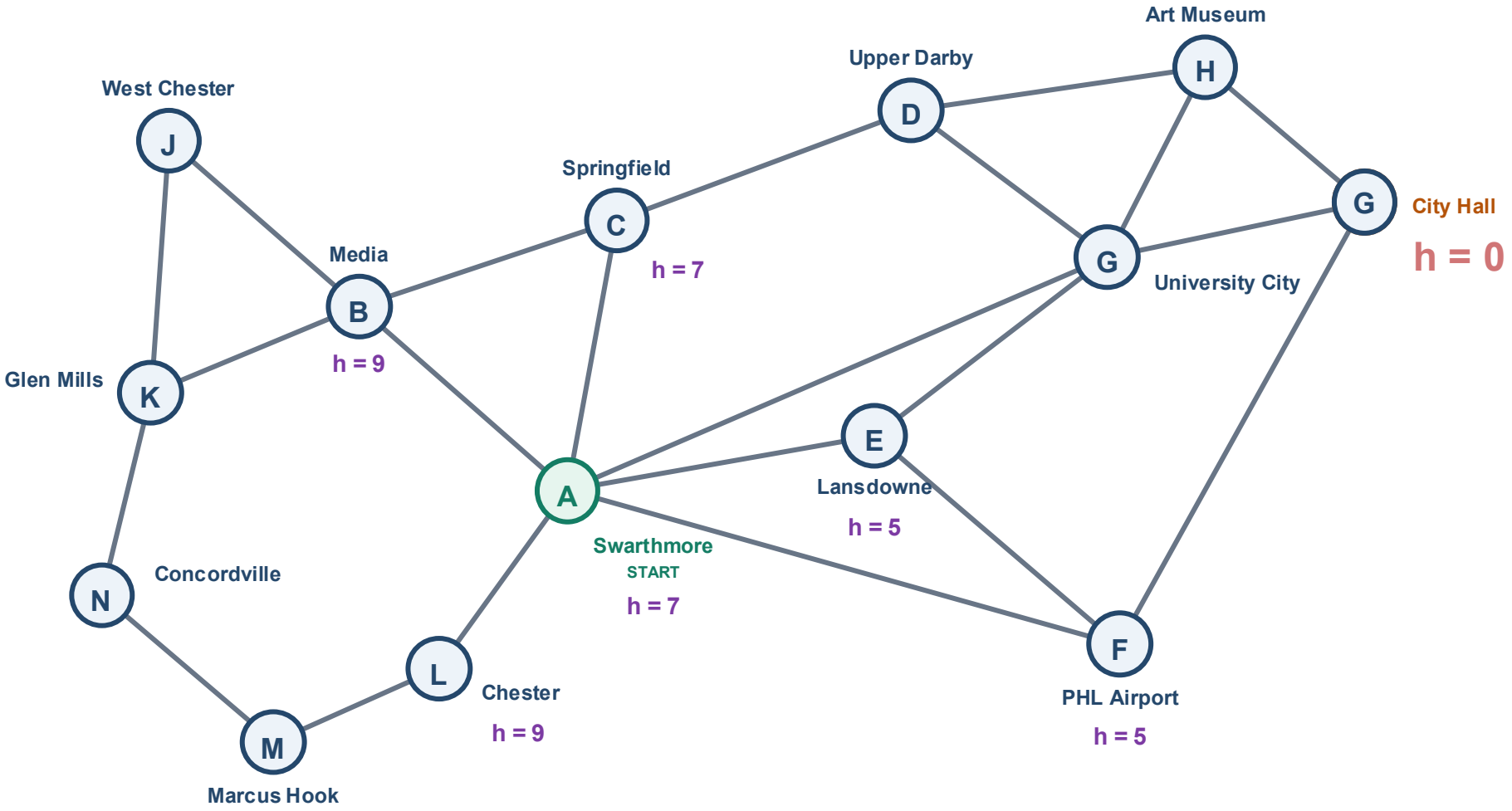
Outline

- Introduce local search
- Three local search algorithms
  1. Hill climbing
  2. Simulated annealing
  3. Beam search


Art Museum
h = 2
Upper Darby
H
D
h = 4
3
2
2
West Chester
J
h = 10
1
Springfield
C
3
3
I
City Hall
GOAL
h = 0
h = 7
3
Media
G
3
University City
h = 3
3
B
2
h = 9
9
4
Glen Mills
K
2
3
h = 11
E
5
Lansdowne
h = 5
2
A
Swarthmore
START
h = 7
1
4
2
2
N
Concordville
h = 11
F
PHL Airport
h = 5
2
L
Chester
h = 9
2
M
Marcus Hook
h = 10

Local search

- So far we have been focusing on searching for a path from a start state to a goal state
- Now we will turn to methods that search directly for a solution rather than a path


Art Museum
H
Upper Darby
D
West Chester
J
Springfield
C
h = 7
G
City Hall
h = 0
Media
B
h = 9
G
University City
Glen Mills
K
E
Lansdowne
h = 5
A
Swarthmore
START
h = 7
Concordville
N
F
L
Chester
h = 9
PHL Airport
h = 5
M
Marcus Hook

Introducing hill-climbing

With a literal hill

Characteristics of local search

- State description itself contains all of the information for a solution
- Operate using a single current node
- Move only to neighbors of that node
- Choose neighbor that looks “best” based on an objective function
- Do not maintain a path

Requirements of local search

1. State representation
2. An objective function that we are trying to maximize (or minimize)
3. A successor function

Advantages of local search

- Uses very little memory
- Often finds reasonably good solutions in large or infinite state space

Disadvantages of local search

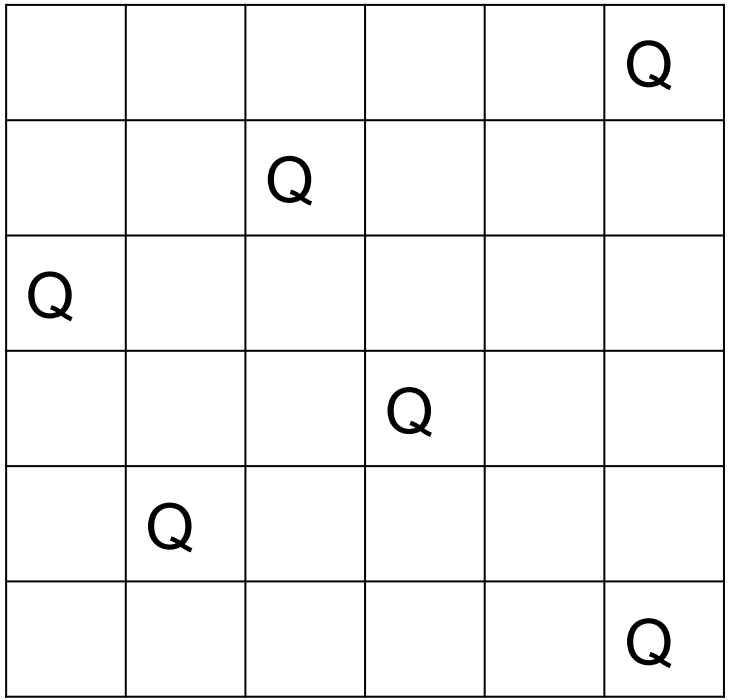
- Does not explore a state space systematically
- Is neither complete nor optimal

However, we typically use local search when a systematic approach will not work

Example problem: N-Queens

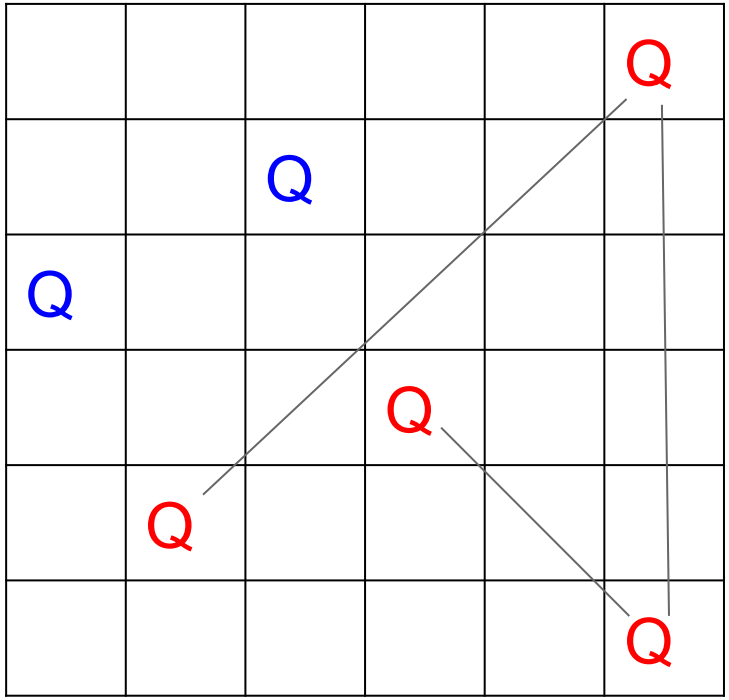
- Goal: place N Queens on an N by N board such that no Queen can attack another one
- Each Queen can move in a line horizontally, vertically, or diagonally
- Size of the search space is N^N
- For the 8-Queens problem this is approximately 17 million states

Example: A 6-Queens state



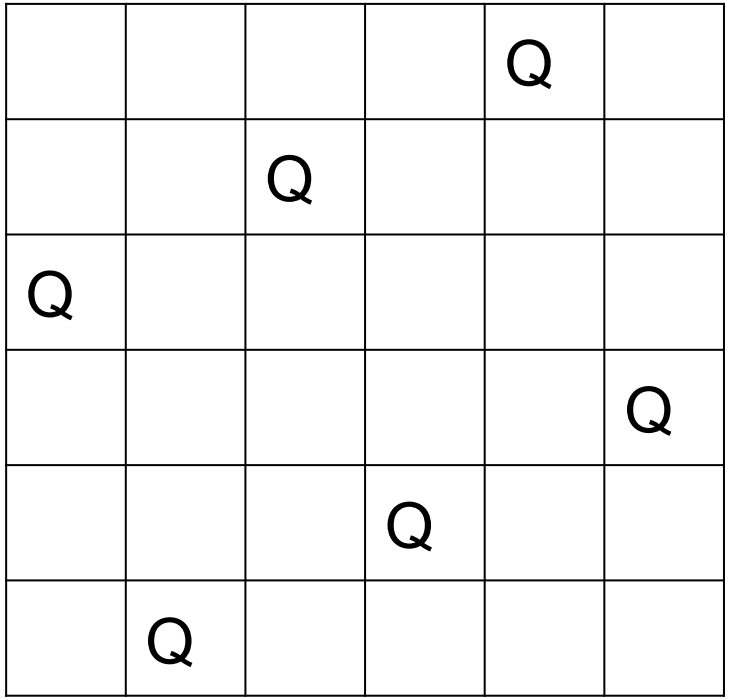
- How many Queens are safe in this state?
- Goal is to find a state where all 6 Queens are safe

Example: A 6-Queens state



- Only the two blue queens are safe on this board

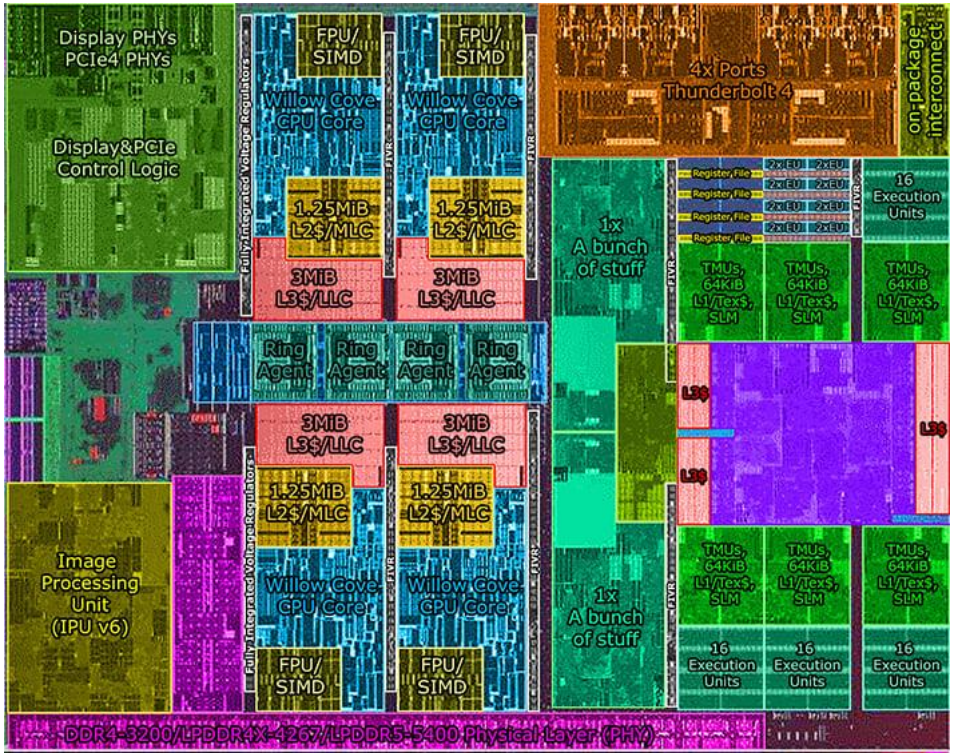
One solution for 6-Queens problem



Applying local search to N-Queens

1. State representation: A list of the (x, y) locations of each Queen on the board
2. An objective function: The number of safe Queens
3. A successor function: Start with a complete configuration of N Queens that may violate some of the constraints, make adjustments

VLSI


Display PHYs
PCIe4 PHYs
Display&PCIe
Control Logic
FPU/
SIMD
Willow Cove
CPU Core
1.25MiB
L2$/MLC
3MiB
L3$/LLC
Ring
Agent
Fully Integrated Voltage Regulators
FIVR
4x Ports
Thunderbolt 4
on-package
interconnect
1x
A bunch
of stuff
Register File
2x EU
16
Execution
Units
TMUs,
64KiB
L1/Tex$,
SLM
L2$
L3$
Image
Processing
Unit
(IPU v6)
DDR4-3200/LPDDR4X-4267/LPDDR5-5400 Physical Layer (PHY)

Successor function for N-Queens

- Start with each Queen in a unique row to avoid horizontal attacks
- Still potential problems with vertical and diagonal attacks
- To generate successors, allow each Queen to move across its row to any column

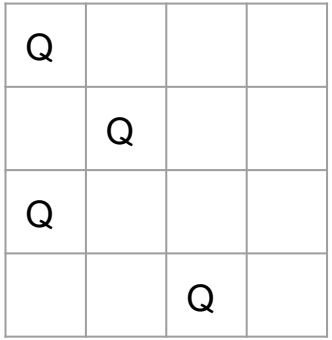
Successor function for N-Queens

For a 4-Queens problem

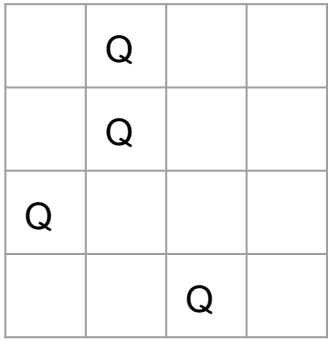
- Each Queen can move to 3 other columns
- There are 4 Queens to move
- For a total of 3*4 = 12 successors

Example successors for 4-Queens

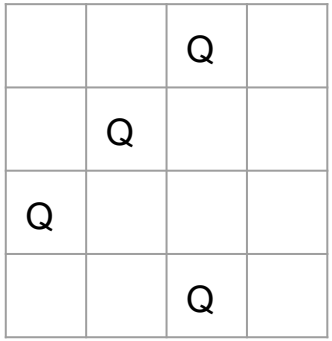
Start: 1 safe



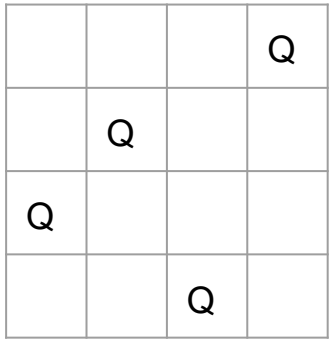
Next1: 1 safe



Next2: 0 safe



Next3:2 safe

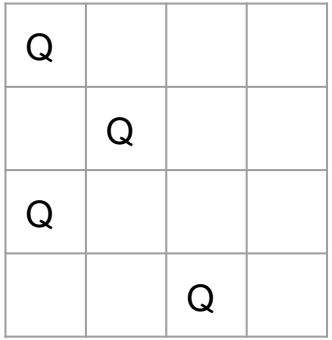


Moving first row Queen

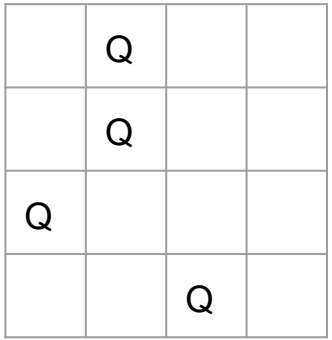
Moving second row Queen

Example successors for 4-Queens

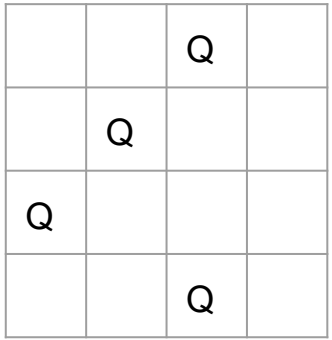
Start: 1 safe



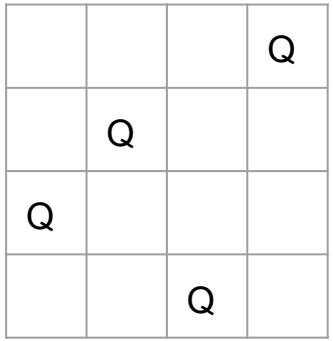
Next1: 1 safe



Next2: 0 safe

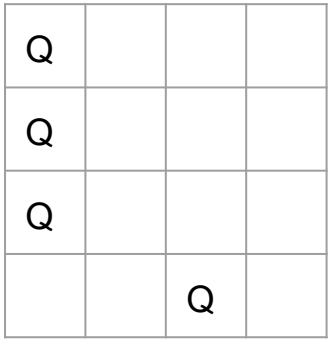


Next3:2 safe

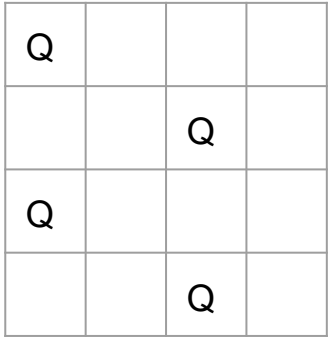


Moving first row Queen

Next4: 0 safe



Next5: 0 safe



...

Moving second row Queen

Successor function for N-Queens

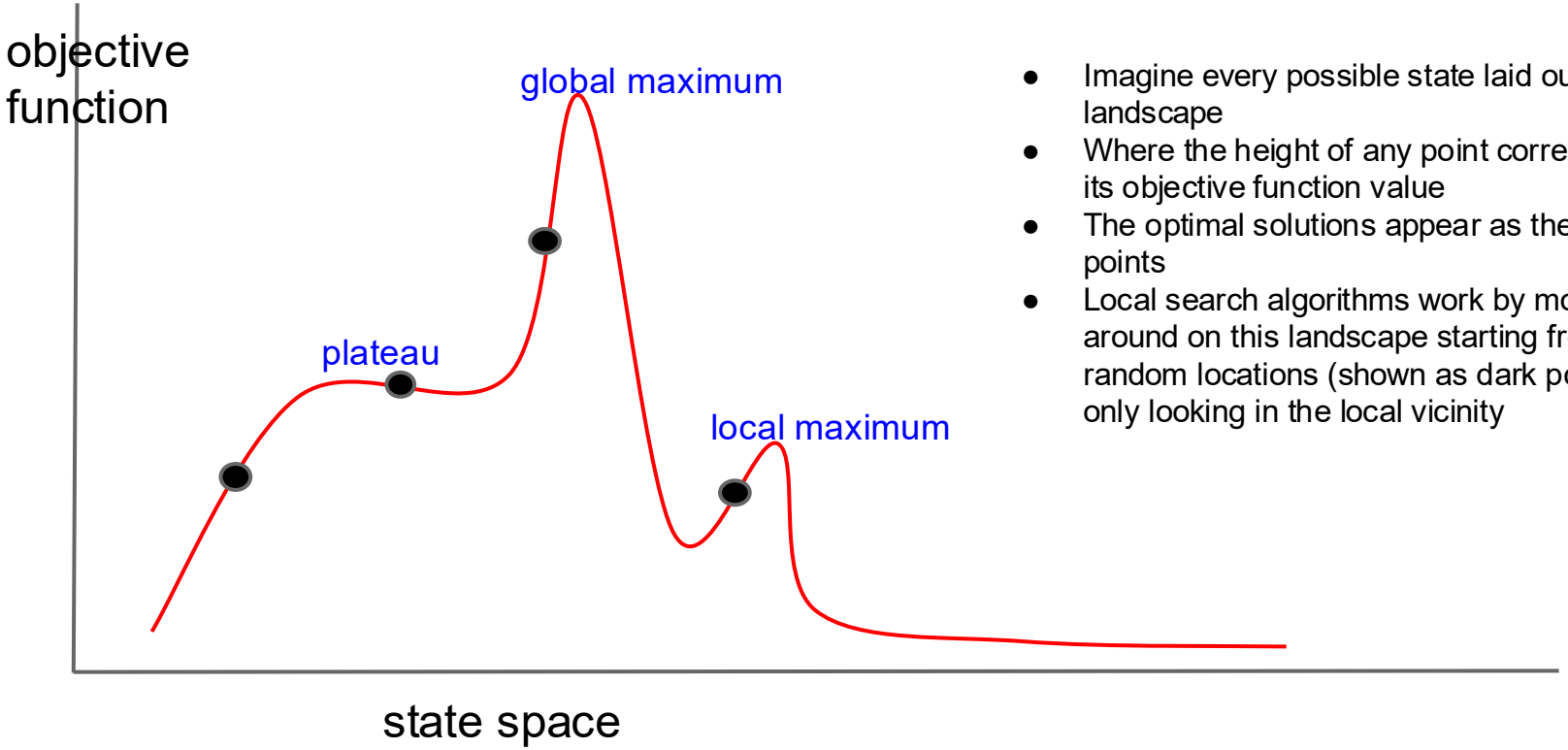
Recall that for the 4-Queens problem we determined there would be 4*3 successors because we have 4 queens each in their own row and each can move to 3 other columns

- How many successors would there be for an 8-Queens problem?
- Or a 25-Queens problem?

Successor function for N-Queens

- How many successors would there be for an 8-Queens problem? 7*8 = 56
- Or a 25-Queens problem? 24*25 = 600

Visualizing local search algorithms



- Imagine every possible state laid out on a landscape
- Where the height of any point corresponds to its objective function value
- The optimal solutions appear as the highest points
- Local search algorithms work by moving around on this landscape starting from random locations (shown as dark points) and only looking in the local vicinity

1. Hill Climbing

- A very straight-forward idea
- Start at a random spot in search landscape
  - Generate your immediate successors
  - If you find one that has a higher score, move to it
  - Repeat until you don’t find any higher successors
- Return the highest valued state found

Reading question

- The reading mentioned several issues that can cause hill climbing to get “stuck”
- Select the issue that seems most problematic to you and explain why

Hill Climbing Pseudocode v.0

```
function hillClimb(problem) returns best state found
  current = makeNode(problem.randomInitialState())
  loop
     neighbor = highest valued successor of current
     if goal(neighbor.state) return neighbor.state
     if neighbor.value <= current.value
        return current.state
     current = makeNode(neighbor)
```

Trace of successful run of v.0

```
Current score: 0 Next score: 2
Current score: 2 Next score: 3
Current score: 3 Next score: 4
Current score: 4 Next score: 5
Current score: 5 Next score: 6
Current score: 6 Next score: 8
-----------------
| |Q| | | | | | |
| | | | |Q| | | |
| | | | | | |Q| |
| | | |Q| | | | |
|Q| | | | | | | |
| | | | | | | |Q|
| | | | | |Q| | |
| | |Q| | | | | |
-----------------
Goal found in 6 steps
```

Trace of unsuccessful run of v.0

```
Current score: 0 Next score: 2
Current score: 2 Next score: 4
Current score: 4 Next score: 5
Current score: 5 Next score: 6
Current score: 6 Next score: 6
-----------------
| |Q| | | | | | |
| | | | |Q| | | |
| | | | | | | |Q|
| | | |Q| | | | |
| | | | | | |Q| |
| | | |Q| | | | |
| | | | | |Q| | |
|Q| | | | | | | |
-----------------
Goal NOT found in 4 steps
```

Average success rate of v.0

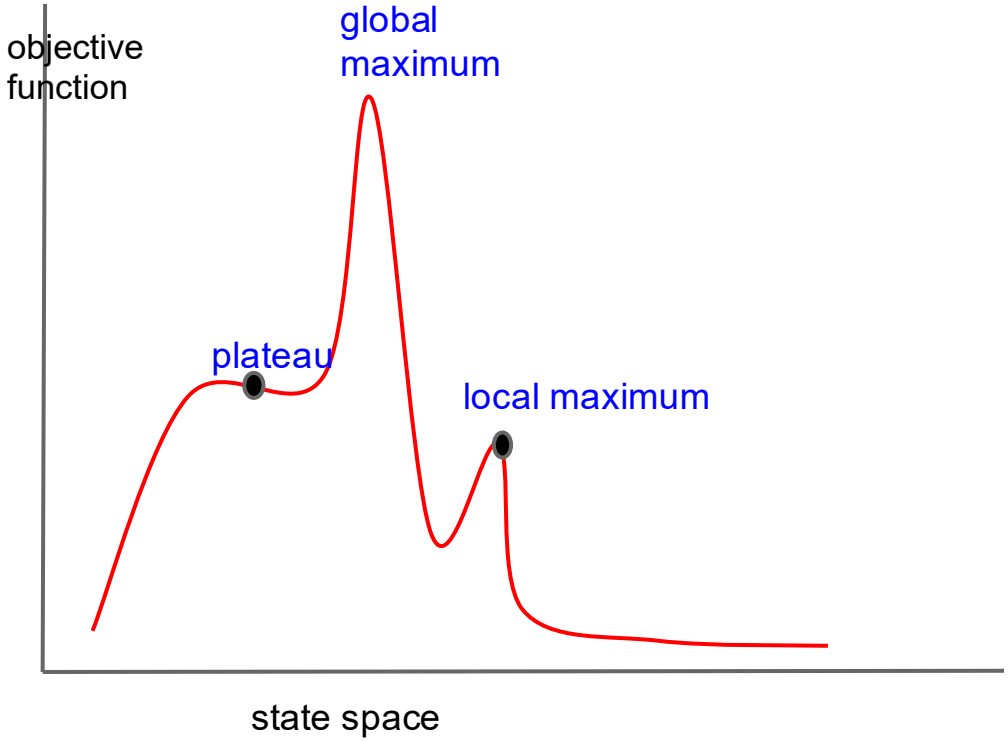
```
Test: Basic hill climbing

Results of 200 trials
Success rate 15%
Average steps when successful 4.2
Average steps when failure    3.0
```

- Not too bad for a search space with 17 million states
- But we can do better

Problems with v.0

- If a local maxima is reached, algorithm halts because every possible step leads down
- If a plateau is reached, the algorithm halts because every possible step is no better than the current location



Handling plateaus

- Allow the algorithm to take sideways steps
- This should improve the success rate, but at the cost of taking more steps

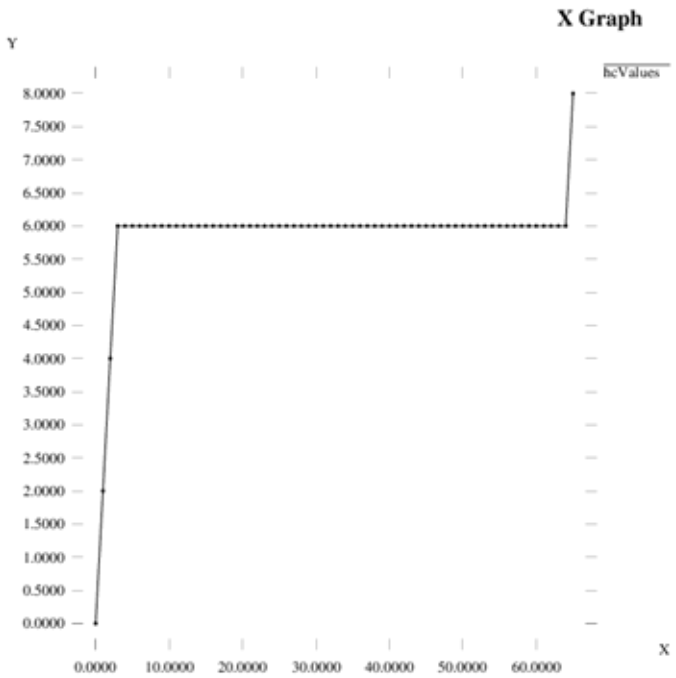
Hill Climbing Pseudocode v.1

```
function hillClimb(problem) returns best state found
  current = makeNode(problem.randomInitialState())
  stepsOnPlateau = 0; plateauLimit = 100
  loop
     neighbor = highest valued successor of current
          if goal(neighbor.state) return neighbor.state
     if neighbor.value == current.value
        stepsOnPlateau += 1
     else
        stepsOnPlateau = 0
     if neighbor.value < current.value or stepsOnPlateau >= plateauLimit
        return current.state
     current = makeNode(neighbor)
```

Trace of v.1

```
Current score: 0 Next score: 2
Current score: 2 Next score: 4
Current score: 4 Next score: 6
...
Current score: 6 Next score: 6
Current score: 6 Next score: 8
-----------------
| | | | |Q| | | |
| | | | | | | |Q|
| | | |Q| | | | |
|Q| | | | | | | |
| | | | | | |Q| |
| |Q| | | | | | |
| | | | | |Q| | |
| | |Q| | | | | |
-----------------
Goal found in 65 steps
```

Stagnation can now occur



- x-axis is steps
- y-axis is objective function values
- Initially the objective values increased
- Then stagnated for 50 steps
- Finally found a goal state

Average success rate of v.1

```
Test: Allow up to 100 moves with no progress

Results of 100 trials
Success rate 92%
Average steps when successful 17.7
Average steps when failure    54.6
```

- Dramatic improvement
- But we can still do better

Problem with v.1

- We still have the issue of local maxima
- Algorithm ends search whenever we have to take a step down
- What could we do to address this issue?

Problem with v.1

- We still have the issue of local maxima
- Algorithm ends search whenever we have to take a step down
- What could we do to address this issue?

Restart the search from scratch at a new randomly selected location

How many restarts will be needed?

- Each hill climbing search has a probability of success, which we saw was about 15% for the N-Queens problem
- So if we do about 7 random restarts we should be successful almost every time

Hill Climbing Pseudocode v.2

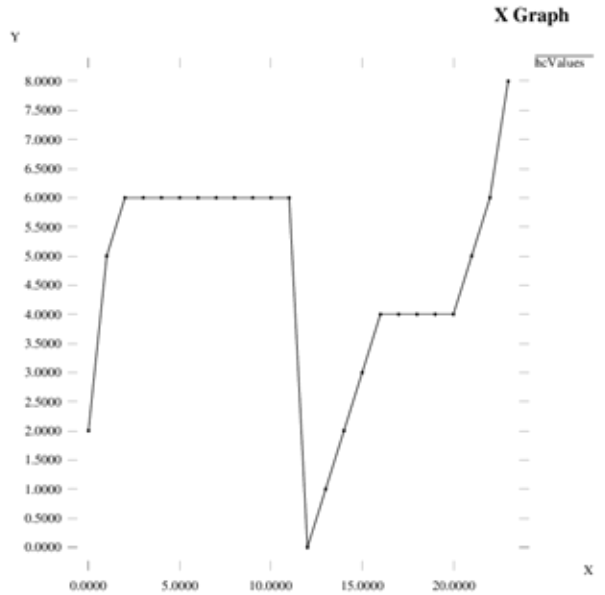
```
function hillClimb(problem) returns best state found
  current = makeNode(problem.randomInitialState())
  stepsOnPlateau = 0; plateauLimit = 10 # Reduce plateau limit
  restarts = 0; restartLimit = 7
  loop
     neighbor = highest valued successor of current
     if goal(neighbor.state) return neighbor.state
     if neighbor.value == current.value
        stepsOnPlateau += 1
     else
        stepsOnPlateau = 0
     if neighbor.value < current.value or stepsOnPlateau >= plateauLimit
        restarts += 1
        if restarts > restartLimit
           return current.state
        else
           neighbor = problem.randomInitalState()
     current = makeNode(neighbor)
```

Trace of v.2

```
Current score: 1 Next score: 2
Current score: 2 Next score: 3
Current score: 3 Next score: 4
Current score: 4 Next score: 6
Current score: 6 Next score: 6
Current score: 6 Next score: 6
Current score: 6 Next score: 6
Current score: 6 Next score: 6
Current score: 6 Next score: 6
Current score: 6 Next score: 6
Current score: 6 Next score: 6
Current score: 6 Next score: 6
Current score: 6 Next score: 6
Current score: 6 Next score: 6
```

```
Hill climbing restarted (1)...
Current score: 1 Next score: 3
Current score: 3 Next score: 4
Current score: 4 Next score: 5
Current score: 5 Next score: 6
Current score: 6 Next score: 6
Current score: 6 Next score: 6
Current score: 6 Next score: 6
Current score: 6 Next score: 6
Current score: 6 Next score: 8
-----------------
| | | | | |Q| | |
| | |Q| | | | | |
| | | | | | |Q| |
| | | |Q| | | | |
|Q| | | | | | | |
| | | | | | | |Q|
| |Q| | | | | | |
| | | | |Q| | | |
-----------------
Goal found in 23 steps
```

Profile of restart objective values



- Once stagnation occurs we quickly restart from a new random location
- Multiple tries at shorter searches proves to be more effective

Test of v.2

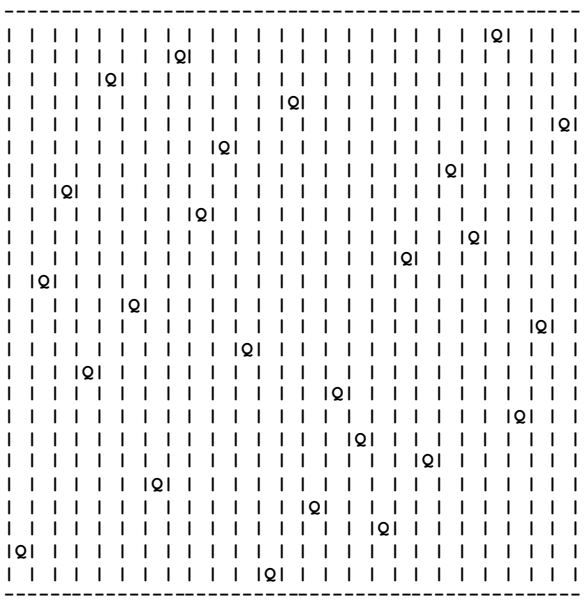
```
Test: Allow up to 7 random restarts
and 10 moves with no progress

Results of 100 trials
Success rate 100%
Average steps when successful 22.3
```

- Using a very simple strategy we are effectively searching 17 million states and finding a solution every time!

v.2 can solve the 25-Queens problem

```
Current score: 3 Next score: 5
Current score: 5 Next score: 7
Current score: 7 Next score: 10
Current score: 10 Next score: 12
Current score: 12 Next score: 13
Current score: 13 Next score: 15
Current score: 15 Next score: 16
Current score: 16 Next score: 17
Current score: 17 Next score: 18
Current score: 18 Next score: 20
Current score: 20 Next score: 21
Current score: 21 Next score: 21
Current score: 21 Next score: 21
Current score: 21 Next score: 22
Current score: 22 Next score: 22
Current score: 22 Next score: 22
Current score: 22 Next score: 22
Current score: 22 Next score: 22
Current score: 22 Next score: 22
Current score: 22 Next score: 22
Current score: 22 Next score: 23
Current score: 23 Next score: 23
Current score: 23 Next score: 23
Current score: 23 Next score: 23
Current score: 23 Next score: 23
Current score: 23 Next score: 23
Current score: 23 Next score: 23
Current score: 23 Next score: 23
Current score: 23 Next score: 25
```



```
Goal found in 30 steps
```

Summary of hill climbing results on N-Queens problem

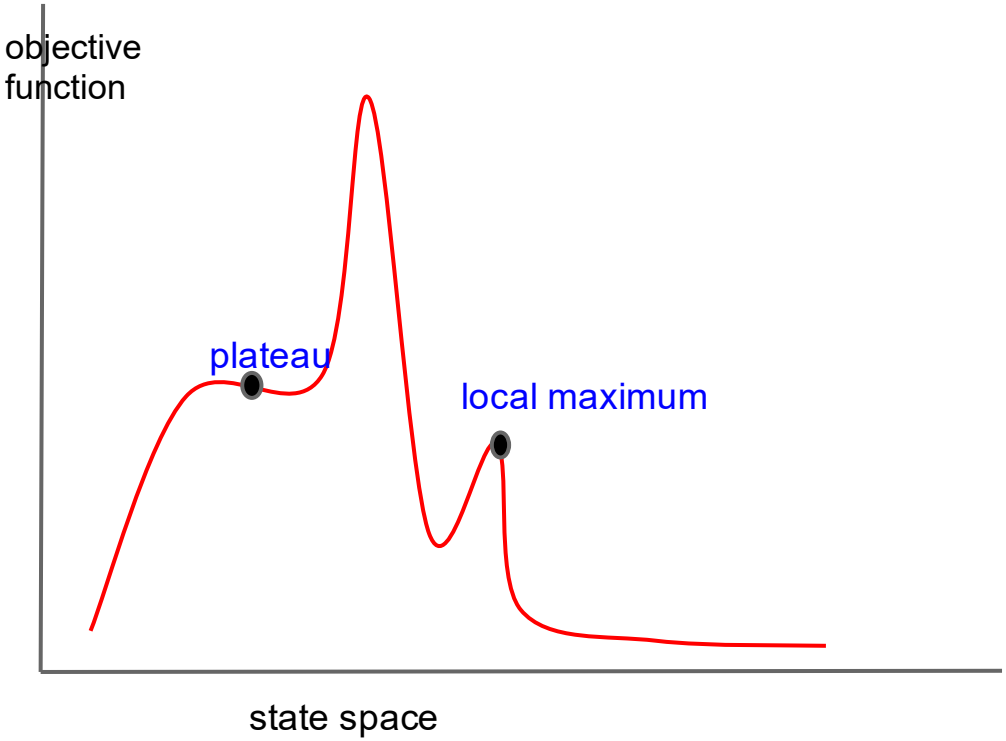
Version	Search	Success rate	Avg Steps Success	Avg Steps Failure
V.0	Hill Climbing	15%	4.2	3.0
V.1 100 side steps	HC with side steps	92%	17.7	54.6
V.2 10 side steps 7 restarts	HC with side steps & random restarts	100%	22.3	0

When hill climbing is appropriate

- The shape of the search space strongly affects the potential success of this method
- A very spiky search space which is flat between the spikes will be quite difficult to solve
- The final version that limits side steps and does a repeated random restarts is incomplete, but is likely to succeed on relatively smooth search spaces

Why might downward steps be helpful?

- Standard hill climbing only moves upward
- Improved hill climbing is willing to take limited sideways steps, but never downward steps (though it did restart at random states)



Employing randomness

Simulated Annealing is a method that is designed to take steps in the wrong direction with some probability that decreases over time

Implementing random options

```
# random function returns a uniform random number [0..1]
from random import random
# randomly do action with probability p
p = 0.33
if random() < p:
    do_action()
```

2. Simulated annealing

- Uses a temperature parameter to control the probability of taking steps in wrong direction
- High temperature equates with a high chance of trying locally bad moves
- Temperature begins high and gradually decreases over time based on a schedule

Simulated Annealing (Minimizing)

```
function simulatedAnnealing(problem, runs, steps, init_temp, temp_decay)
  bestState = None; bestCost = float("inf") # init to high value
  loop over runs
    temp = init_temp
    currState=problem.random_candidate(); currCost=problem.cost(currState)
    loop over steps # each step considers one random successor
       neighbor, neighborCost = problem.random_successor(currState)
       ...

      temp *= temp_decay
  return bestState, bestCost
```

Simulated annealing (Minimizing)

```
function simulatedAnnealing(problem, runs, steps, init_temp, temp_decay)
  bestState = None; bestCost = float("inf") # init to high value
  loop over runs
    temp = init_temp
    currState= problem.random_candidate();currCost= problem.cost(currState)
    loop over steps # each step considers one random successor
      neighbor, neighborCost = problem.random_successor(currState)
      delta = currCost - neighborCost
      # always accept better neighbors, accept worse neighbors with some prob.
      if delta > 0 or with probability e^(delta/temp)
         currState = neighbor; currCost = neighborCost
      if currCost < bestCost # minimizing
         bestState = currState; bestCost = neighborCost
      temp *= temp_decay
  return bestState, bestCost
```

Prob. of taking locally bad step: $e^{(delta/temp)}$

- delta represents the difference in value between the current state and the random successor
- negative deltas indicate bad moves when minimizing
- temp represents the current temperature
- higher temps lead to more randomness

Prob. of taking locally bad step: $e^{(delta/temp)}$

- Assume delta = -1
- Vary temp
- Higher temps lead to higher probabilities of accepting the downward step

delta	temp	$e^{(delta/temp)}$
-1	10.0	0.90
-1	5.0	0.82
-1	2.0	0.61
-1	1.0	0.37
-1	0.5	0.13

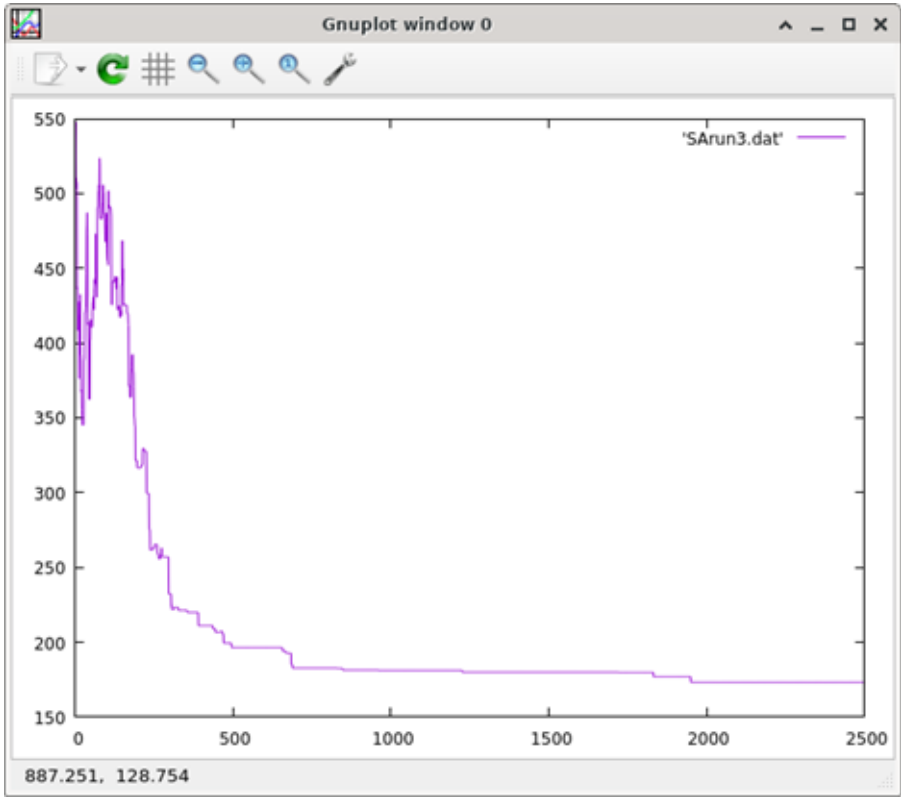
Prob. of taking locally bad step: $e^{(delta/temp)}$

- Assume temp = 3.0
- Vary delta
- More negative deltas lead to lower probabilities of taking those downward steps
- In other words, there is much less of a chance of moving to a very bad neighbor

delta	temp	$e^{(delta/temp)}$
-1	3.0	0.72
-2	3.0	0.51
-3	3.0	0.37
-4	3.0	0.26
-5	3.0	0.19

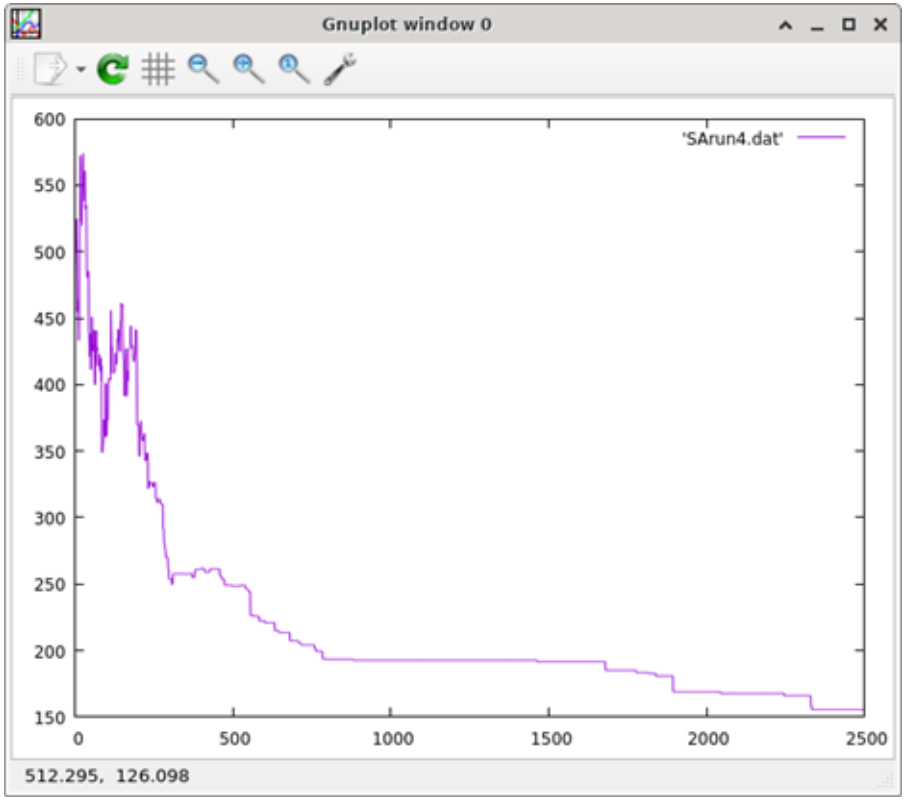
Example run of Simulated Annealing

- X axis is steps
- Y axis is the objective score that we want to minimize
- Notice that early on the objective score is very volatile, going up and down
- But as temperature decreases, the volatility decreases



Another run of Simulated Annealing

- X axis is steps
- Y axis is the objective score that we want to minimize
- Again we see a similar trend: high variation early on that diminishes over time



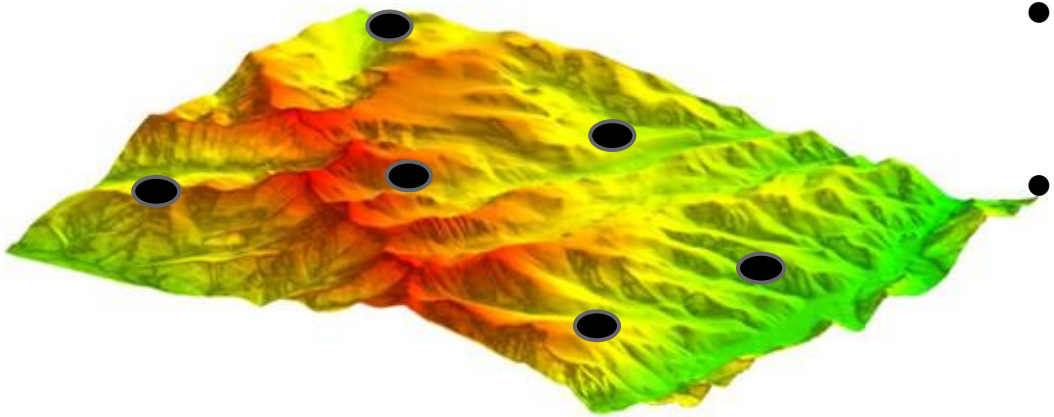
Simulated Annealing performance

- If the temperature is lowered slowly enough, optimal solutions can be found
- In practice such a schedule is too slow and we have to accept suboptimal solutions

3. Beam search

- Population-based search which operates on multiple individuals at once
- Essentially doing many local searches in parallel

Population-based local search



Green represents lower values, red higher

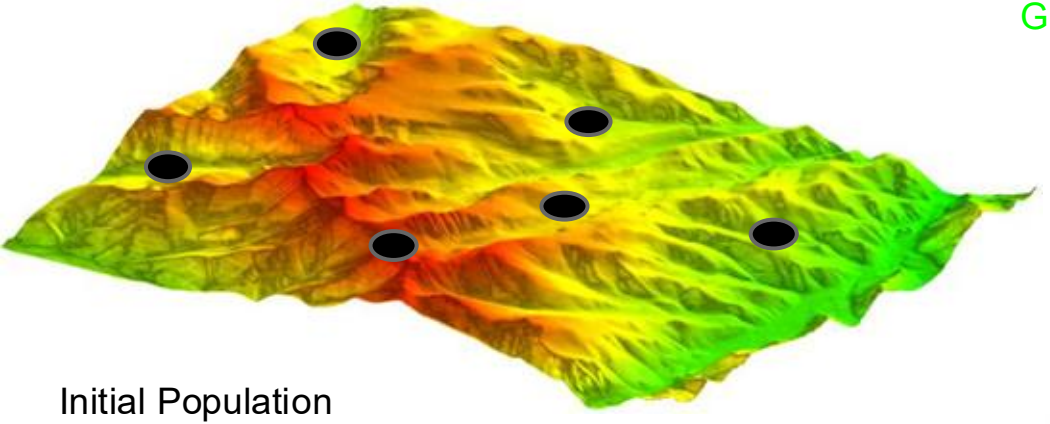
- Initially individuals are scattered randomly across the fitness landscape
- In Beam Search, a local search is conducted near each individual

Beam Search

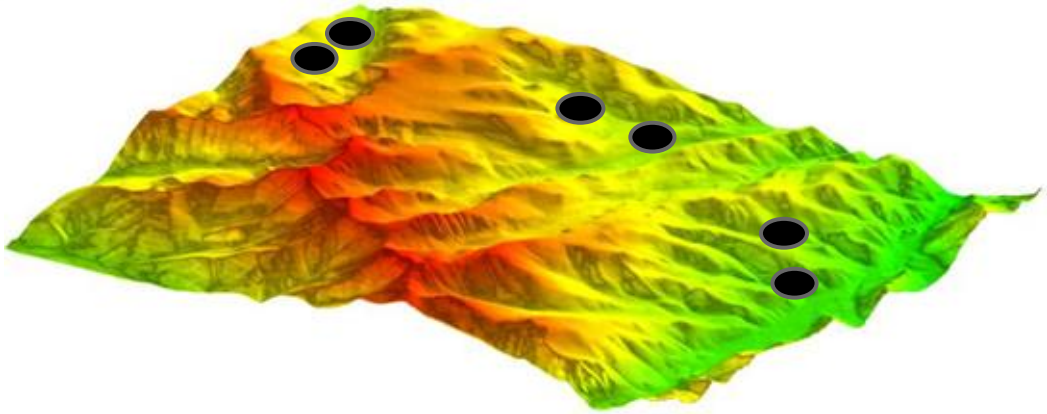
- Population of `pop_size` individuals
- Each iteration, generate neighbors of the current population
  - If generating all neighbors is too time consuming, instead generate a random subset of them
- Select the best `pop_size` neighbors for the next population
- Continue until an acceptable answer is found

Visualizing Minimizing Beam Search

Green represents lower values, red higher



Initial Population



Next Population

Beam Search performance

+ Quickly focuses on promising areas of the search space
+ Memory efficient
- Population may converge too quickly on suboptimal areas of the search space
- Neither complete nor optimal

Stochastic Beam Search

- Instead of selecting the best `pop_size` individuals, chooses randomly
- Probability of an individual being selected is proportional to its score
- Inspired by biological evolution and survival of the fittest

Stochastic Beam Search pseudocode

```
function StochasticBeamSearch(problem, pop_size, steps, init_temp,
                              decay_temp, max_neighbors)
  bestState = None; bestCost = float("inf") # init to high value
  pop = initial population of pop_size problem.random_candidate()
  temp = init_temp
  loop over steps
     ...

     temp *= decay_temp
  return bestState, bestCost
```

Stochastic Beam Search pseudocode

```
function StochasticBeamSearch(problem, pop_size, steps, init_temp,
                              decay_temp, max_neighbors)
  bestState = None; bestCost = float("inf") # init to high value
  pop = initial population of pop_size problem.random_candidate()
  temp = init_temp
  loop over steps
     generate max_neighbors neighbors of each individual in pop
     determine overall bestNeighborState and bestNeighborCost
     if bestNeighborCost < bestCost update bestState and bestCost
     generate probabilities for selecting neighbors based on costs
          pop = pop_size neighbors selected by random sampling based on
           probabilities
     temp *= decay_temp
  return bestState, bestCost
```

Determining Stochastic Beam Search probabilities

Suppose that pop_size=5 with individuals labeled A-E

Assume we are minimizing cost

Let's consider 2 neighbors per individual (where A's neighbors are A1 and A2)

Neighbor	A1	A2	B1	B2	C1	C2	D1	D2	E1	E2
Cost	7	9	1	2	10	7	3	4	8	9

Sorting these by lowest cost: B1, B2, D1, D2, A1, E1, A2, E2

Goal is to preferentially select the lower cost neighbors, but still have some small likelihood of selecting any neighbor

Implementing selection in stochastic beam search

- Compute costs of neighbors
- Compute $e^{(-cost/temp)}$ of neighbors
- Compute sum of these and normalize them
- Use these normalized values as the probability distribution
- NOTE: Probabilities can become so small as to cause `ZeroDivisionError`
  - Can catch this error and simply return the best state found so far
  - May indicate that the parameters need tweaking

Stochastic Beam Search probabilities (temp=10.0)

Neighbor	A1	A2	B1	B2	C1	C2	D1	D2	E1	E2
Cost	7	9	1	2	10	7	3	4	8	9
$e^{(-cost/temp)}$	0.50	0.41	0.90	0.82	0.37	0.50	0.74	0.67	0.45	0.41
Normalized	0.09	0.07	0.16	0.14	0.06	0.09	0.13	0.12	0.08	0.07

sum(Normalized) must be 1 to treat these as probabilities

B1 (the best neighbor) has the highest probability of selection at 16%

C1 (the worst neighbor) has the lowest probability of selection at 6%

For lower temps the range of probabilities would be wider

Stochastic Beam Search probabilities (temp=5.0)

Neighbor	A1	A2	B1	B2	C1	C2	D1	D2	E1	E2
Cost	7	9	1	2	10	7	3	4	8	9
$e^{(-cost/temp)}$	0.25	0.17	0.82	0.67	0.14	0.25	0.55	0.45	0.20	0.17
Normalized	0.07	0.05	0.22	0.18	0.04	0.07	0.15	0.12	0.06	0.04

With a lower temperature, the probabilities favor good neighbors even more:

B1 (the best neighbor) went from 16% to 22% chance of selection

C1 (the worst neighbor) went from 6% to 4% chance of selection

Implementing selection

Selection code:

```
from numpy.random import choice
items = ['a', 'b', 'c', 'd', 'e']
probs = [0.5, 0.25, 0.2, 0.05, 0]
n = 4
# select n times from the items
# based on the probabilities
s = list(choice(items,n,p=probs))
print("selection:", selection)
```

Output might be:

```
selection: ['a', 'b', 'a', 'c']
selection: ['a', 'a', 'b', 'a']
selection: ['c', 'a', 'a', 'a']
```