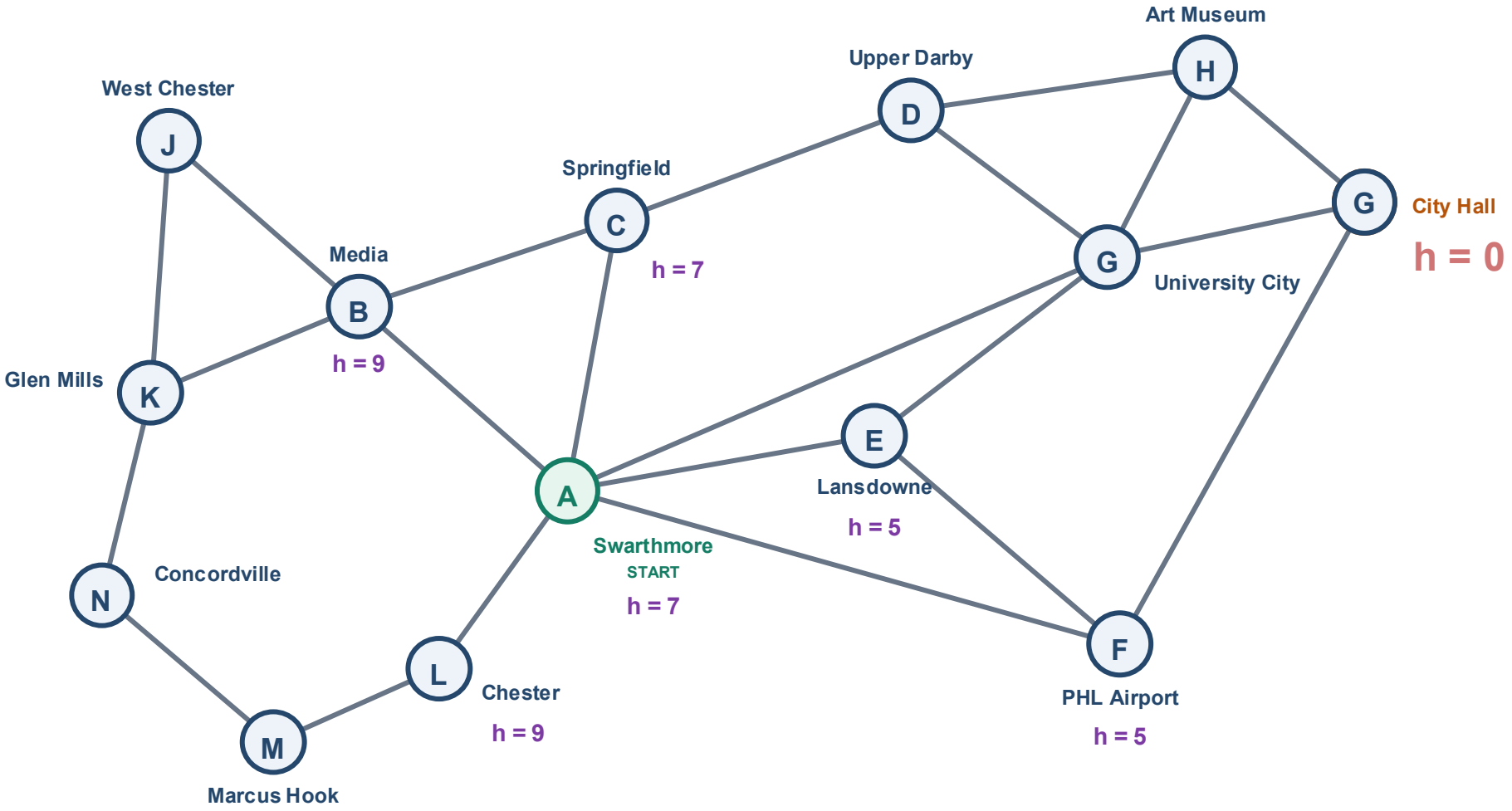
Outline

- Review
  - Simulated annealing
- Another local search method
  - Beam Search

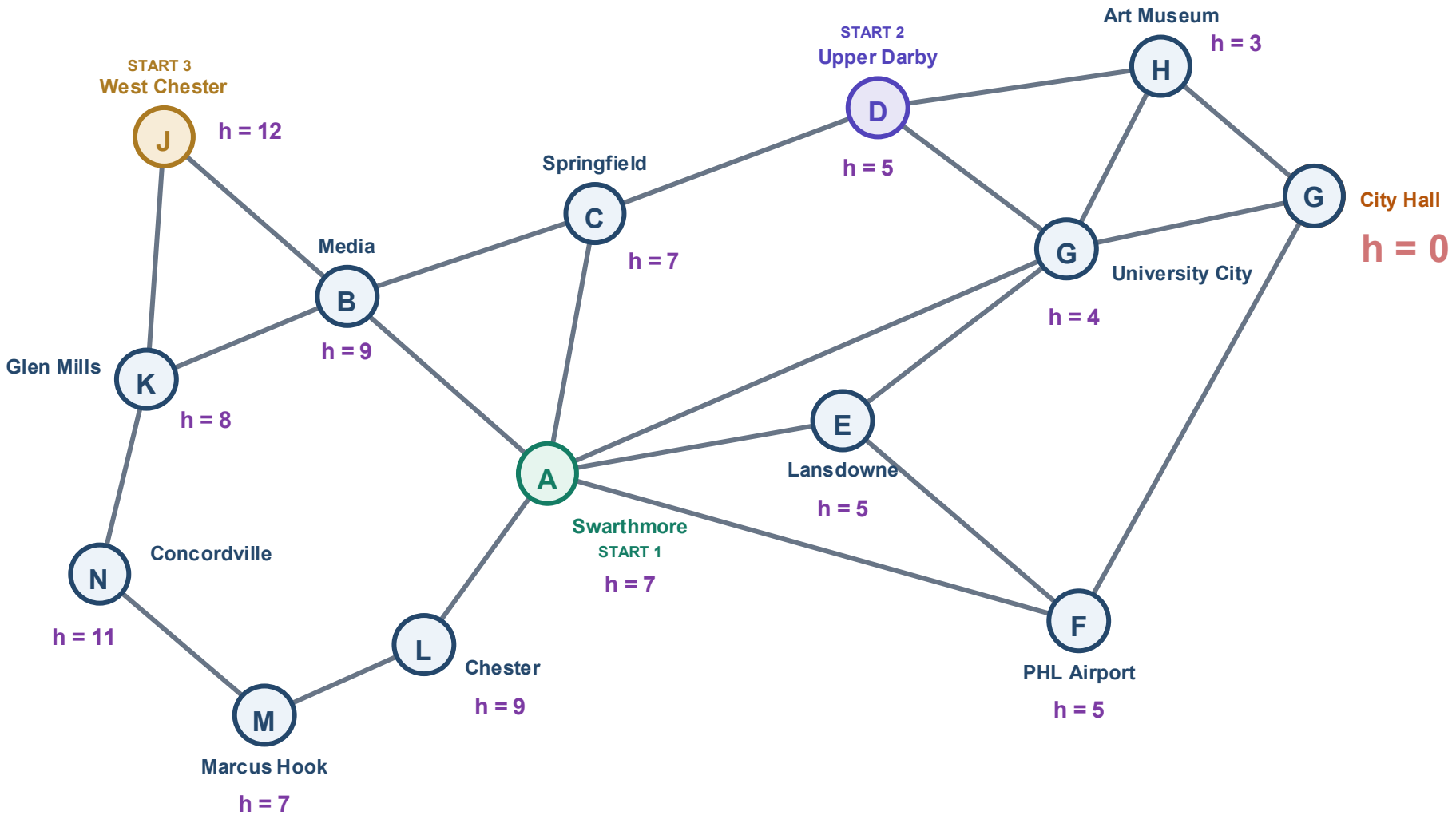
Review

- 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)
- Simulated annealing is willing to take downward steps, especially early on in the search

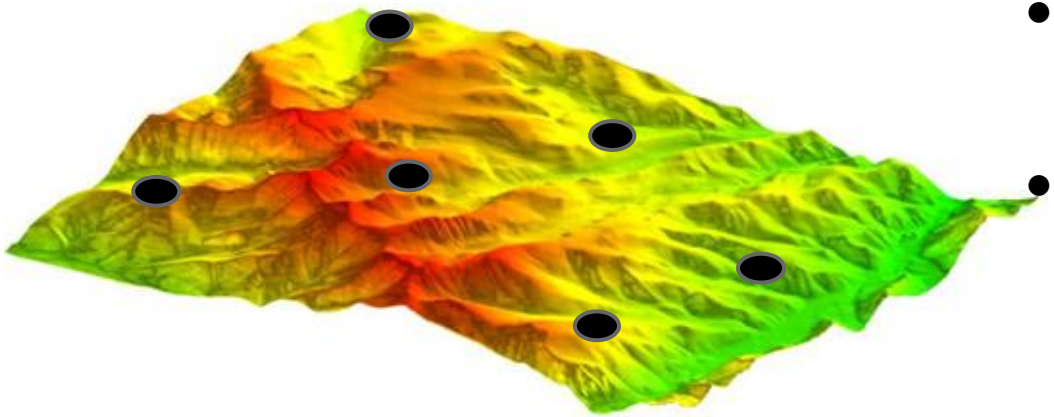

Art Museum
H
Upper Darby
D
West Chester
J
Springfield
C
h = 7
G
City Hall
h = 0
Media
B
h = 9
G
University City
Glen Mills
K
E
Lansdowne
h = 5
A
Swarthmore
START
h = 7
Concordville
N
F
L
Chester
h = 9
PHL Airport
h = 5
M
Marcus Hook

Population-based local search

- Operates on multiple individuals at once
- Essentially doing many local searches in parallel
- Examples include:
  - Beam Search
  - Stochastic Beam Search


START 3
West Chester
J
h = 12
START 2
Upper Darby
D
h = 5
Art Museum
h = 3
H
Springfield
C
h = 7
G
City Hall
h = 0
Media
B
h = 9
G
University City
h = 4
Glen Mills
K
h = 8
E
Lansdowne
h = 5
A
Swarthmore
START 1
h = 7
Concordville
N
h = 11
L
Chester
h = 9
F
PHL Airport
h = 5
M
Marcus Hook
h = 7

Population-based local search

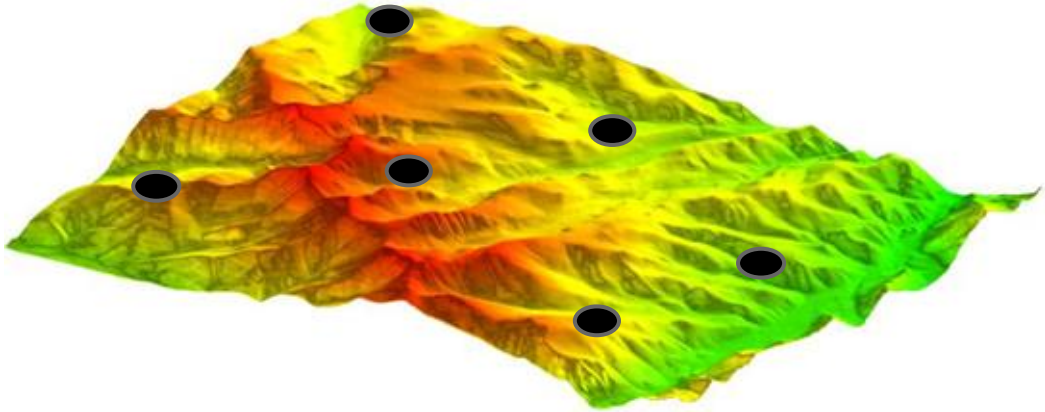


Green represents lower values, red higher

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

What if we *share* information?

Green represents lower values, red higher



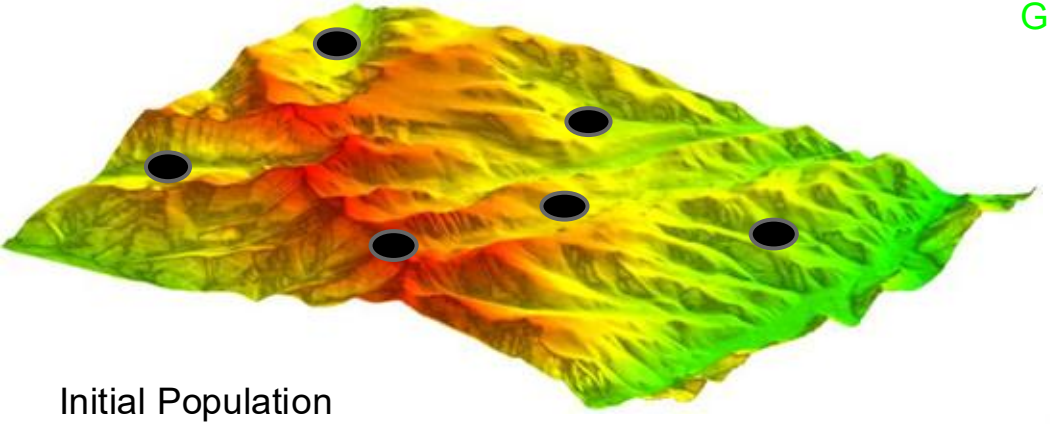
- Initially individuals are scattered randomly across the fitness landscape
- In Beam Search, a local search is conducted near each individual
  - Select the best neighbors from what everyone can see!

Beam Search

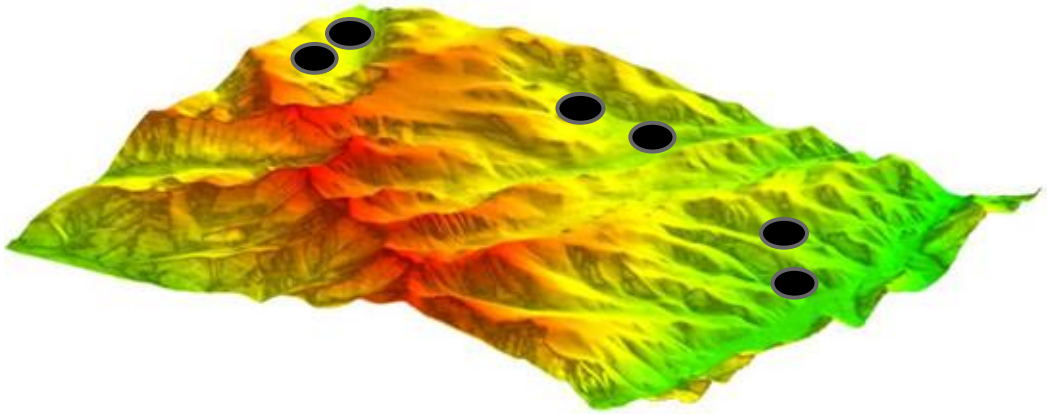
- Population of `pop_size` individuals
- Each iteration, generate neighbors of the current population
  - If generating all neighbors is too time consuming, instead generate a random subset of them
- Select the best `pop_size` neighbors for the next population
- Continue until an acceptable answer is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

Visualizing Minimizing Beam Search

Green represents lower values, red higher



Initial Population



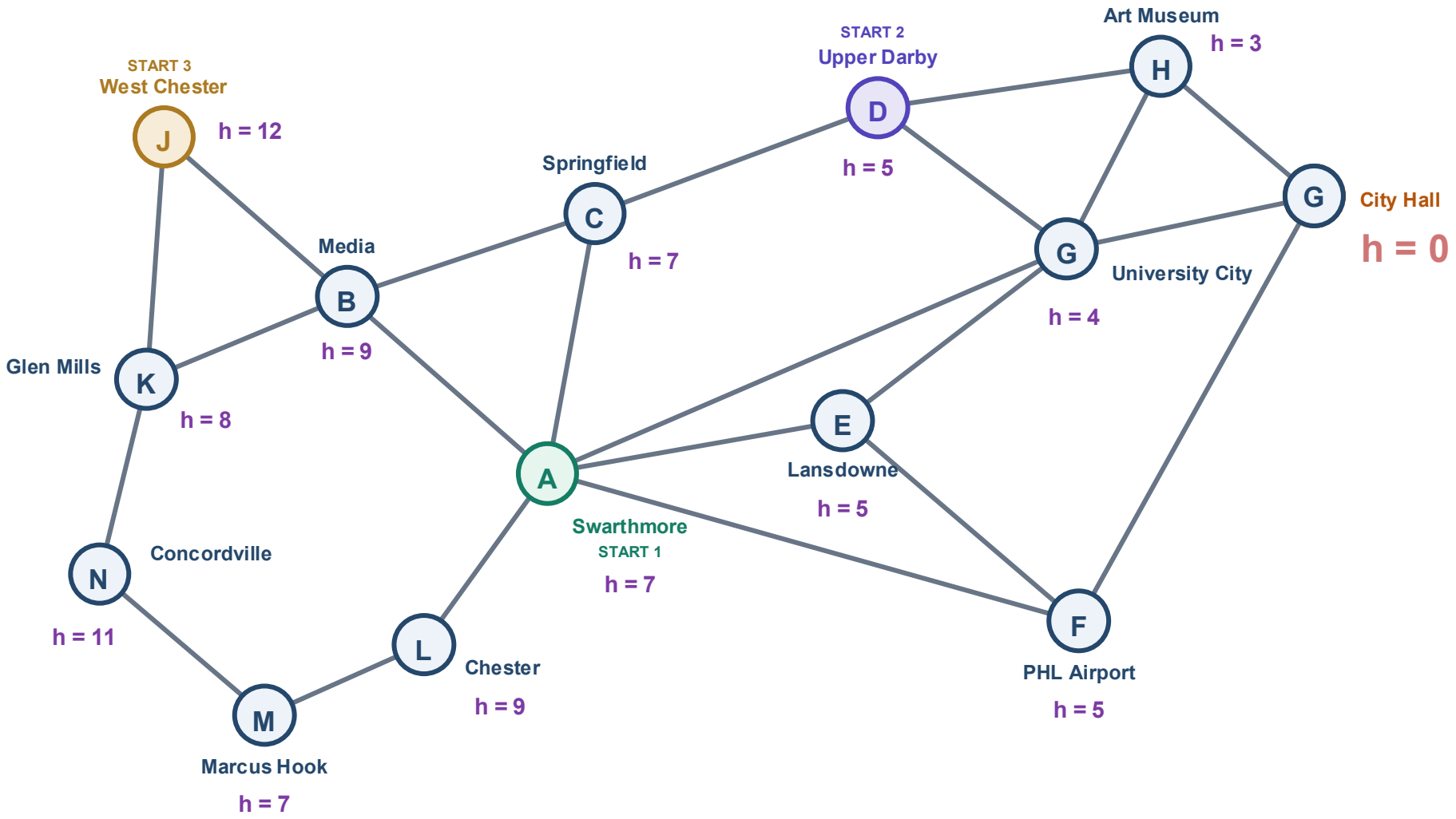
Next Population

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

Same problems as hill-climbing

- Easily gets *stuck* at local maxima/minima
- *Can we allow for downward steps like simulated annealing?*


START 3
West Chester
J
h = 12
START 2
Upper Darby
D
h = 5
Art Museum
h = 3
H
Springfield
C
h = 7
G
City Hall
h = 0
Media
B
h = 9
G
University City
h = 4
Glen Mills
K
h = 8
E
Lansdowne
h = 5
A
Swarthmore
START 1
h = 7
Concordville
N
h = 11
L
Chester
h = 9
F
PHL Airport
h = 5
M
Marcus Hook
h = 7

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 = 3
# 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']
selection: ['a', 'a', 'b']
selection: ['c', 'a', 'a']
```

Without replacement

Selection code:

```
from numpy.random import choice
items = ['a', 'b', 'c', 'd', 'e']
probs = [0.5, 0.25, 0.2, 0.05, 0]
n = 3
# select n times from the items
# based on the probabilities
s = list(choice(items,n,p=probs, replace=False))
print("selection:", selection)
```

Output might be:

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

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

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

Sampling without replacement prevents us from sampling the same choice twice

- After first sample, remove it from the choices and *renormalize again*
- Numpy handles this automatically
- Either is acceptable for the lab assignment!

Outline

- Adversarial games and AI
- Game trees
  - Similar to state space trees, but swap perspectives on each level
- How the Minimax algorithm operates
- Evaluation functions

Game Playing

- Board games provide an ideal environment for AI
  - Examples: Chess, Checkers, Othello, Backgammon
  - Fixed set of rules
  - Well-defined interactions between players
- Physical games are much more challenging
  - Examples: softball, basketball, tennis, soccer

Desirable properties of board games

- Discrete
- Static
- Deterministic

The main difficulty is dealing with the uncertainty created by the opponent

Consider the game Nim

- There are initially N pieces
- Each turn a player must remove 1, 2, or 3 pieces
- The player who removes the last piece loses
- Let’s play a game where N=9, you go first

Explore the game further

- Try playing a few games with each other
- Try different values for N=5, 4, 3, and 2
- Are certain states of the game especially good or bad to be in?

Outcomes for player who goes first

N	Outcome	Move
1	lose	take1
2	win	take1
3	win	take2
4	win	take3
5	lose	any
6	win	take1
7	win	take2
8	win	take3
9	lose	any
10	win	take1
11	win	take2
12	win	take3
13	lose	any

What strategy did you use?

- People tend to think, “If I do this, then my opponent can do that…”
- This type of thinking is essentially search
- You are looking ahead to see what the possible outcomes of your actions will be

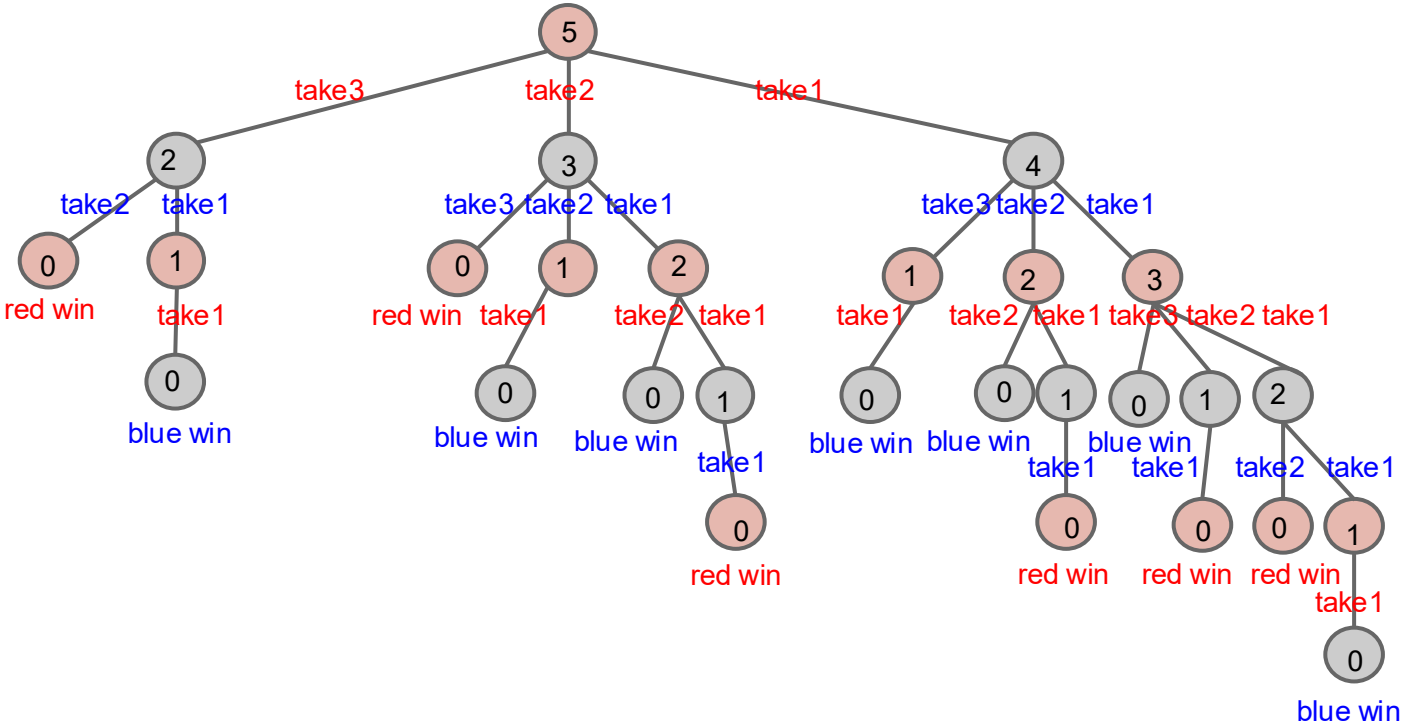
Adversarial search

- Searching in the presence of an adversary adds uncertainty, because you don’t know what the opponent will do
- Let’s assume that the opponent is behaving optimally, trying to make the best moves possible
- Then we can use state space search to create a game tree

Game tree

- Each node of the tree represents a state of the game
  - For Nim, the state of the game is how many pieces are left
- Branches from a node represent a player's possible moves from that state
  - For Nim, there are at most 3 moves (take3, take2, or take1)
- Levels of the game tree switch perspective each time (player1 then player2, and so on)
- Leaves of the tree represent terminal states where the game is over
  - For Nim, this is when 0 pieces are left

Nim game tree for N=5



Zero-Sum Games

- Games where a positive reward for one player is a negative reward for the other
- Nim is an example of such a game
- The outcome of the game can be represented as a single number
- One player is trying to maximize this number
- The other is trying to minimize this number

Setting up a game tree for search

- Players
  - MAX: the player who goes first
  - MIN: the player who goes second
- Values of terminal states
  - +1 = win for MAX
  - -1 = win for MIN
  - In some games it is possible to tie, which would be represented as 0

Figuring out the best move

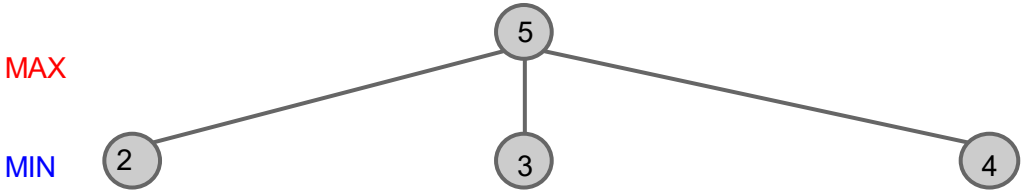
- Use a depth-first search approach
- Backup the terminal values through the parent nodes to the root
- If it is MAX's turn, backup the maximum value from the successors
- If it is MIN's turn, backup the minimum value from the successors
- Choose the move at the root with the maximum value
- This method is called MINIMAX

Tracing through Minimax, for Nim N=5

Player1 is the maximizer

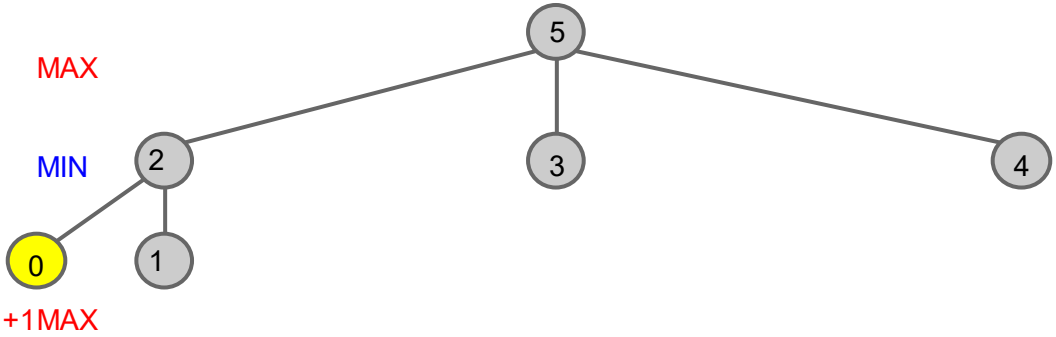
Tracing through Minimax, for Nim N=5

Initially, look at all possible moves from current game state



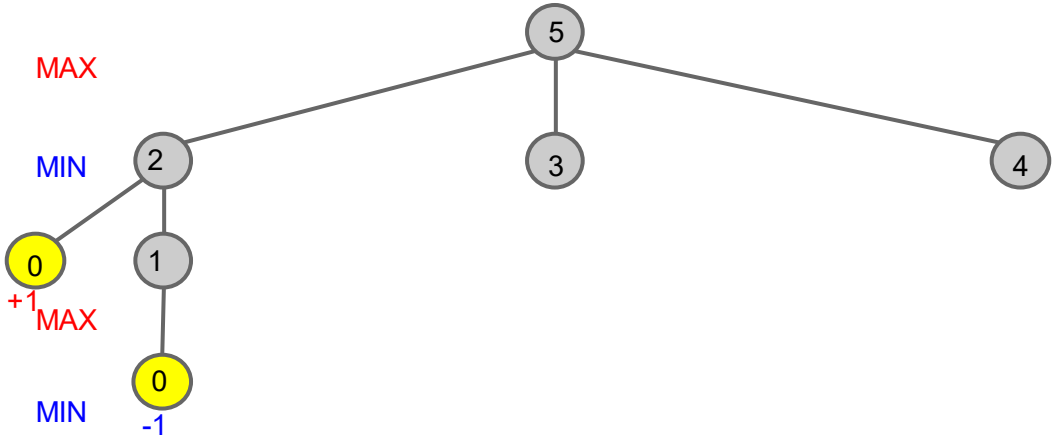
Tracing through Minimax, for Nim N=5

In a DFS way, explore deeper into the game tree



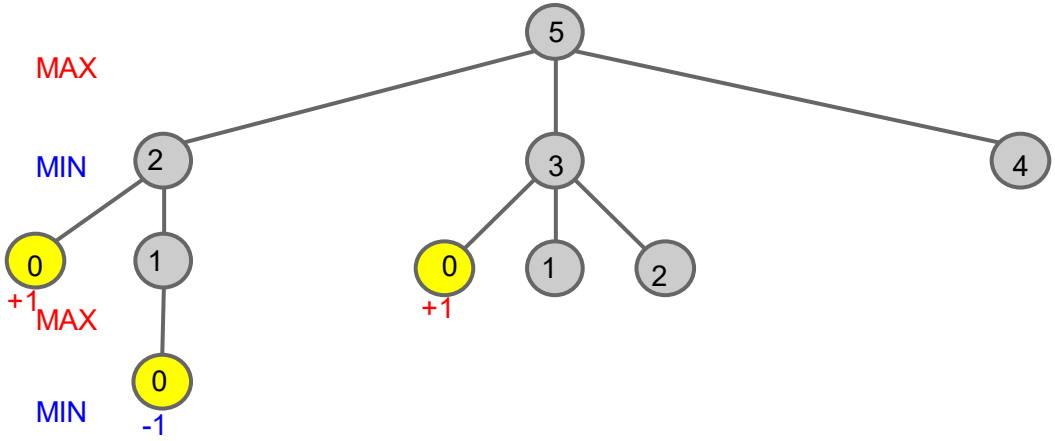
Tracing through Minimax, for Nim N=5

When terminal states are reached, backup the values from each player's perspective



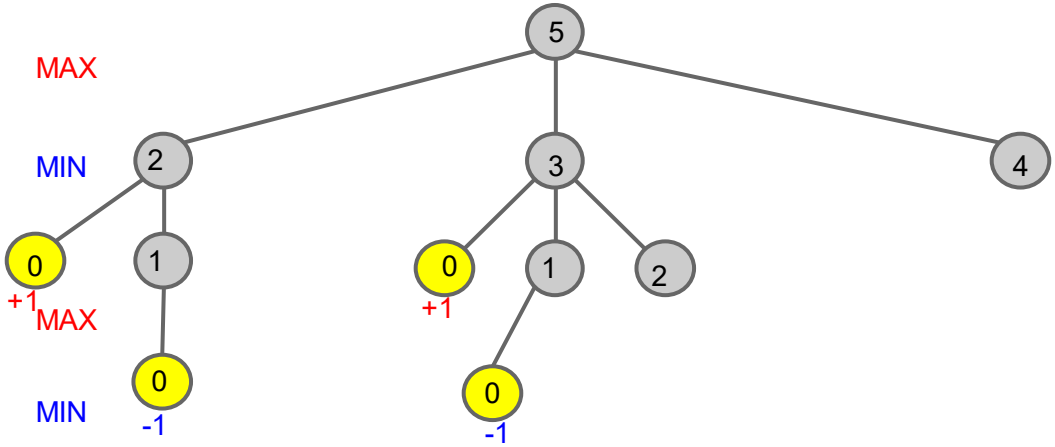
Tracing through Minimax, for Nim N=5

Continue exploring other moves in a DFS way



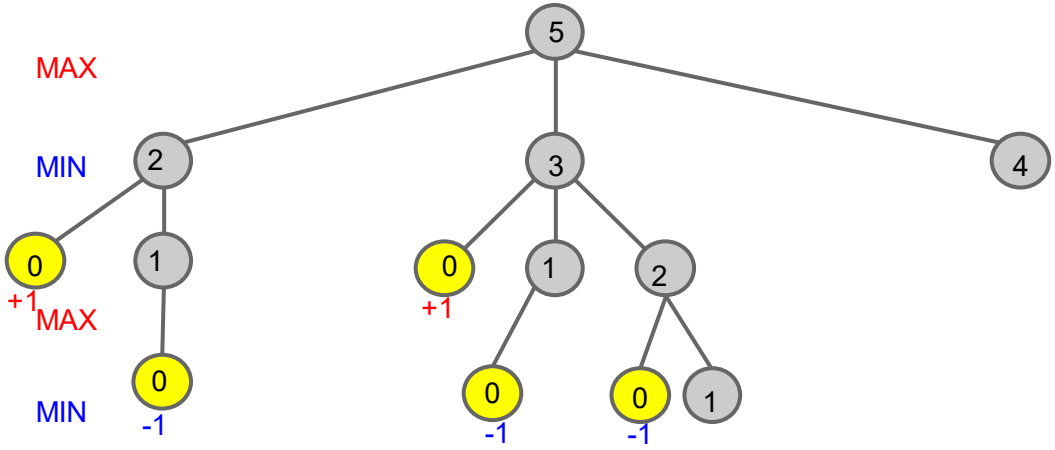
Tracing through Minimax, for Nim N=5

Continue exploring other moves in a DFS way



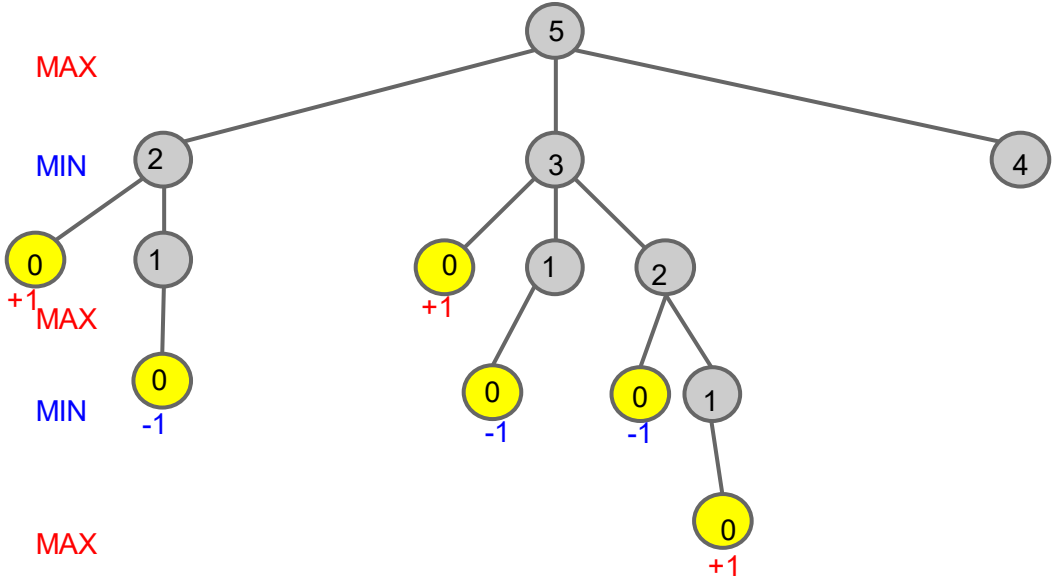
Tracing through Minimax, for Nim N=5

Continue exploring other moves in a DFS way



Tracing through Minimax, for Nim N=5

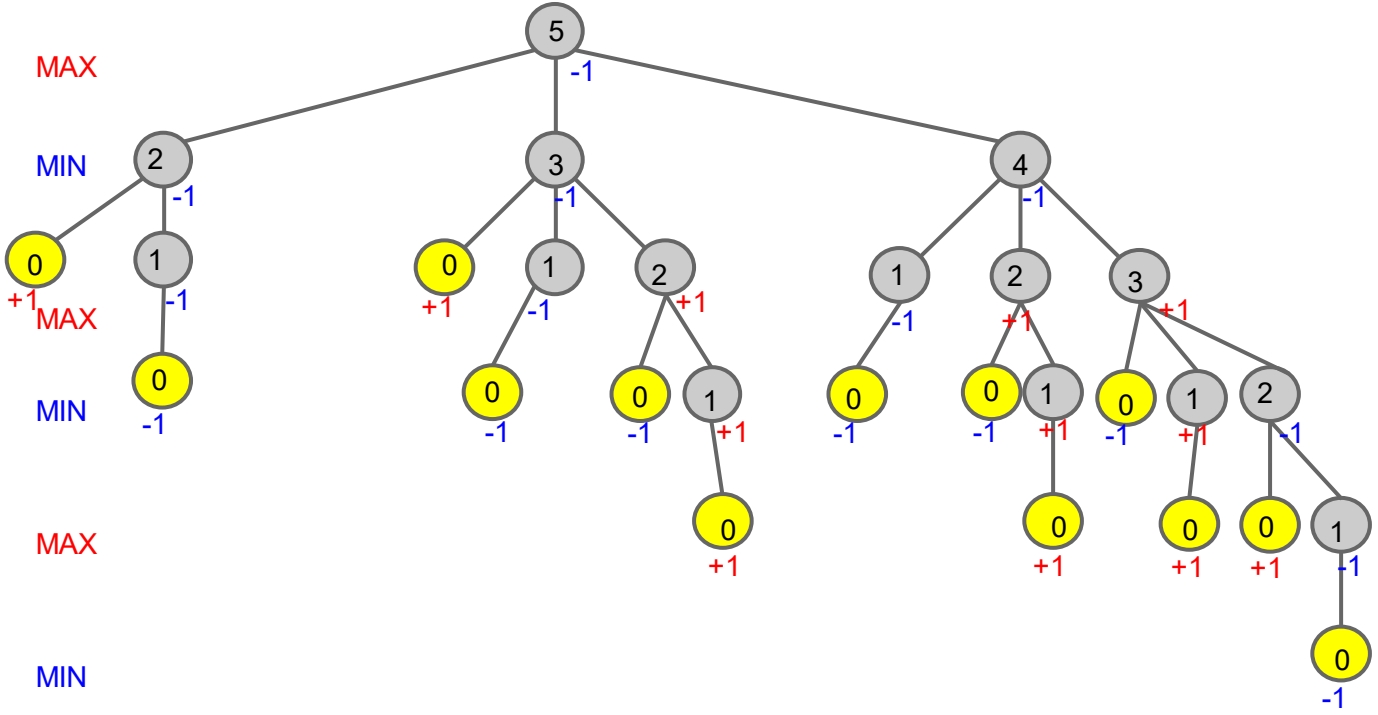
When terminal states are reached, backup the values from each player's perspective



Tracing through Minimax, for Nim N=5

Outcome of the entire search

Notice that with optimal play, starting at N=5 is always a losing state



Minimax algorithm

- Performs a depth-first traversal of the game tree while also determining the best outcome from each player’s perspective
- Minimax is optimal assuming that the leaf node evaluations are correct

Practicality of Minimax

- For most interesting games the game tree is too large to search to the end and to find optimal moves
- In chess, the branching factor is approximately 35 and games can last for 100 moves
- This creates a game tree of 35^{100}