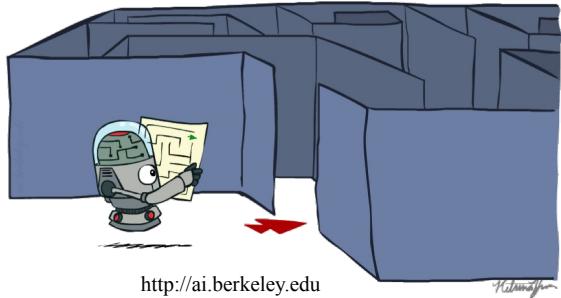




# Chapter 1

## Search



<http://ai.berkeley.edu>

COMP 3270  
Artificial Intelligence

Dirk Schnieders

# Outline

- Abstraction
- Types of Search
- Search Problem Definition
- State Space Graph vs. Search Tree
- Search Algorithm (TSA, GSA)
- Uninformed Search
  - BFS, DFS, UCS
- Informed Search
  - Greedy, A\*
- Local Search
- Constraint Satisfaction Problems
- Adversarial Search
  - Minimax
  - DLS
  - Horizon Effect
  - $\alpha$ - $\beta$  Pruning
  - Expectimax
  - Multi-Agent Utilities

# Search



<http://ai.berkeley.edu>

# Abstraction

- Search problems are models
  - Simplifications of the real world



- Don't deal with the unnecessary complexities of the real world
  - World states vs. search states

# Types of Search

- Uninformed search
  - No information about the problem other than its definition is given
- Informed search
  - A heuristic is used that leads to better overall performance in getting to the goal state
- Local Search
  - Evaluate and modify a current state to move closer to a goal state
- Constraint Satisfaction Problems
  - For certain types of problems we can search for solution faster by understanding states better
- Adversarial Search
  - Search in the presence of an adversary

# Search Problem Definition

- States: Details of what constitutes a state
- Initial state: The state the agent starts in
- Actions and transition model
  - Description of possible actions available
  - Description of what each action does
- Goal test: Is a given state a goal state?
- Path cost: A function that assigns a zero or positive numeric cost to each path

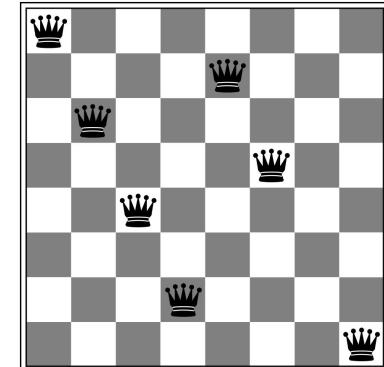
A solution is a sequence of actions (a plan) which transforms the start state to a goal state

# N Queens Puzzle - An Example Problem

- Problem Definition A

Eight Queens Puzzle on [Wikipedia](#)

- States
  - Any arrangement of 0 to n queens on the board
- Initial state
  - No queens on the board
- Actions and Transition model
  - Add a queen to an empty square
- Goal test
  - n queens are on the board, none attacked



<http://aima.cs.berkeley.edu/>

# N Queens Puzzle - An Example Problem

- Problem Definition B
  - States
    - One queen per column, none attacking another
  - Initial state
    - No queens on the board
  - Actions and Transition model
    - Add a queen to an empty column such that no other queen is under attack
  - Goal test
    - n queens are on the board

# State Space

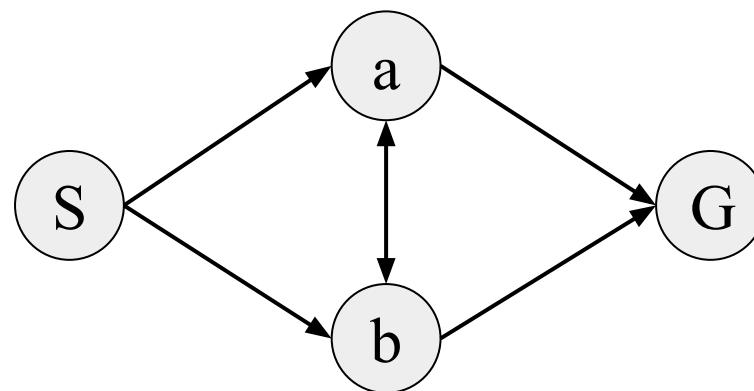
- The set of all states reachable from the initial state by any sequence of actions is the state space
  - Usually a graph
  - The possible action sequences form a search tree
- The nodes are states, and the links between nodes are actions
- A solution is a sequence of actions (i.e., a path) leading from the initial state to a goal state
- Task: Which state space has more states?
  - Problem Definition A
  - Problem Definition B

# State Space Graph vs. Search Tree

- State space graph
  - A mathematical representation of a search problem
    - Nodes are (abstracted) world configurations
    - Arcs represent successors (action results)
    - The goal test is a set of goal nodes
  - Each state occurs only once
- Search tree
  - Root of the tree is the start state
  - Nodes represent the possible action sequences
  - States may occur more than once

# State Space Graph vs. Search Tree

- Consider the following state space graph
  - Let S be the start state and G be the goal state



- Task: Draw the complete search tree

# States vs. State Sequences

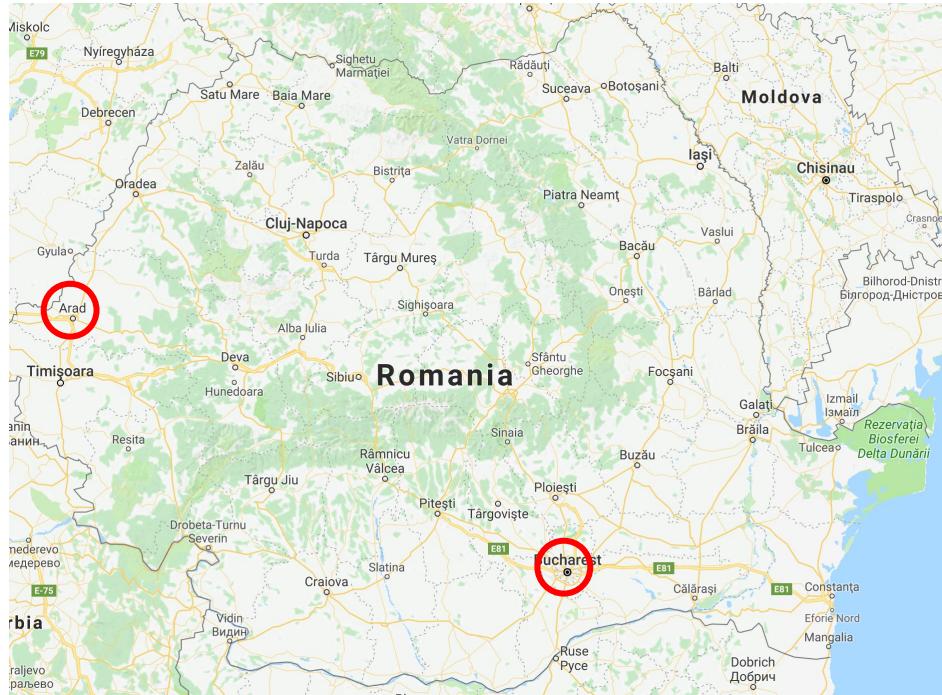
- A large state space results in a huge number of state sequences
  - I.e., a large number of nodes in the search tree
- Example: Chess
  - Claude Shannon estimated  $10^{43}$  possible states and  $10^{120}$  possible state sequences in his 1950 paper “Programming a Computer for Playing Chess”

“There are only  $10^{15}$  total hairs on all the human heads in the world,  $10^{23}$  grains of sand on Earth, and about  $10^{81}$  atoms in the universe. The number of typical chess games is many times as great as all those numbers multiplied together - an impressive feat for 32 wooden pieces lined up on a board.”

Extracted from: Answers to the World's Greatest Questions, By Bjorn Carey

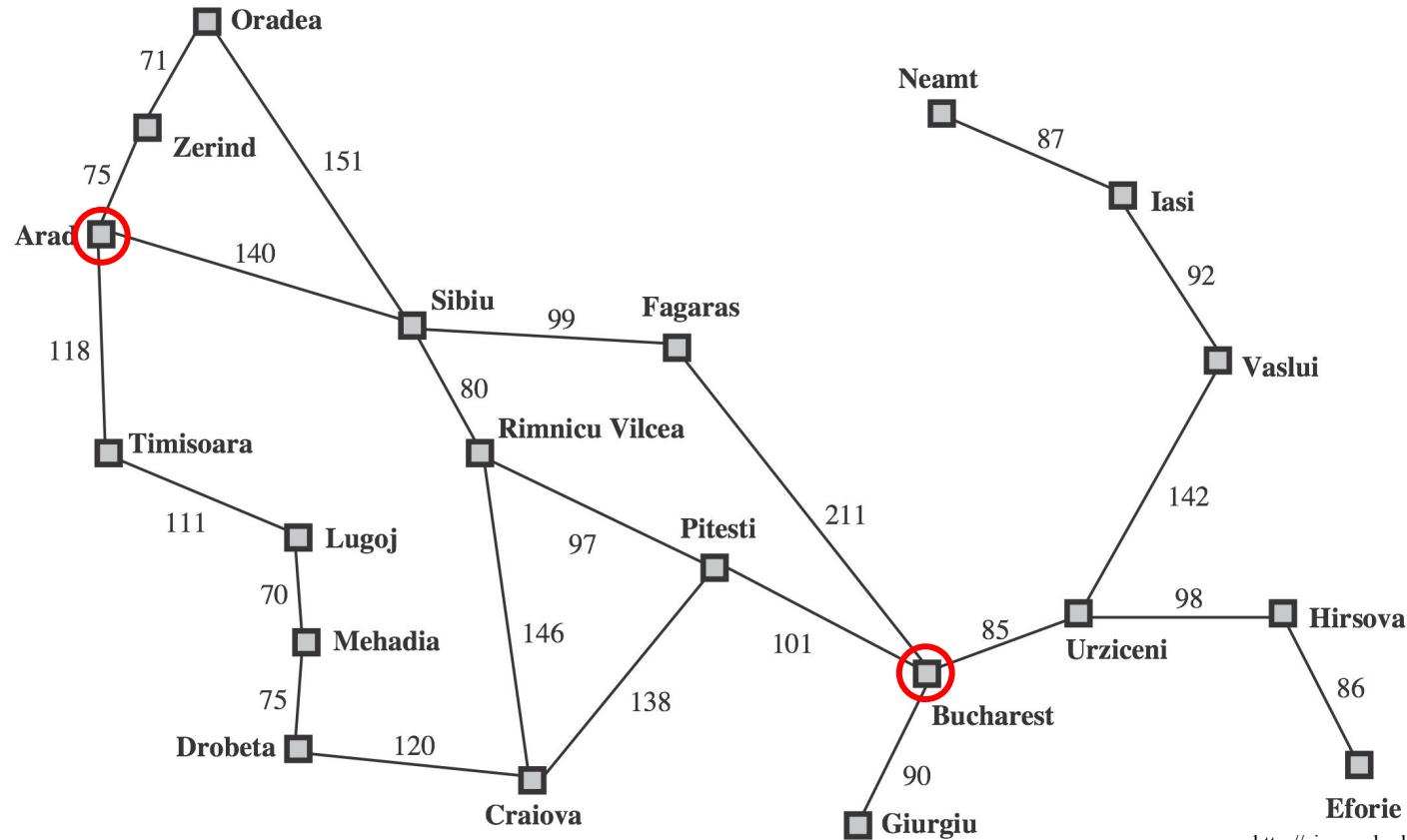
# Romania Problem - A Toy Problem

- States
  - The cities
- Initial state
  - Arad
- Actions and Transition model
  - Go to neighboring city
- Goal test
  - In Bucharest?
- Path cost
  - Distance between the cities



[www.google.com/maps/place/Romania/](http://www.google.com/maps/place/Romania/)

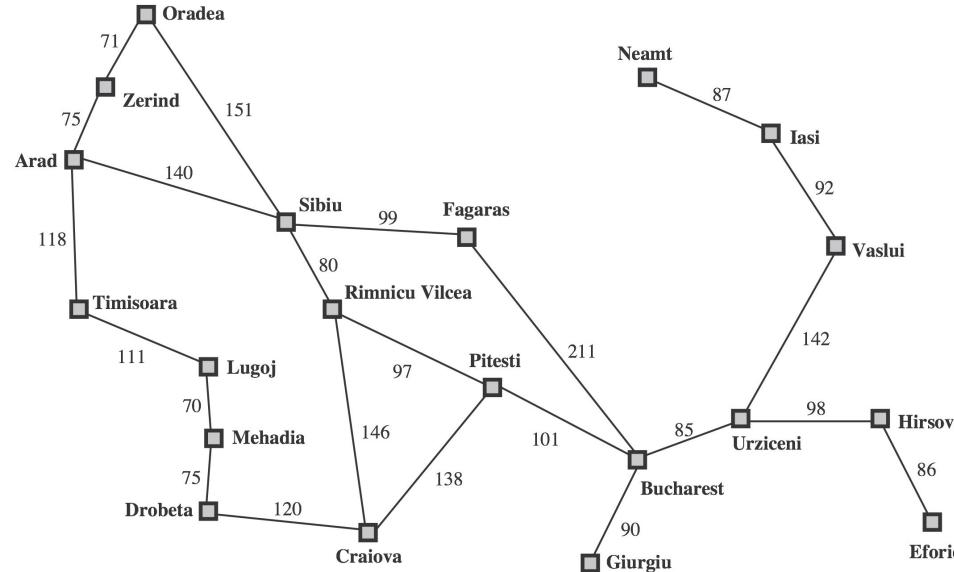
# Romania Problem - State Space



# Romania Problem Definition

Use dictionary to store neighbors for each cities

```
>>> romania['A']
['S', 'T', 'Z']
```



```
romania = {
    'A':[['S', 'T', 'Z'], 'Z':[['A', 'O'], 'O':[['S', 'Z'], 'T':[['A', 'L'], 'L':[['M', 'T'], 'M':[['D', 'L'], 'D':[['C', 'M'], 'S':[['A', 'F', 'O', 'R'], 'R':[['C', 'P', 'S'], 'C':[['D', 'P', 'R'], 'F':[['B', 'S'], 'P':[['B', 'C', 'R'], 'B':[]}}]]]]]]]]]
```

# Search Strategy

- The search strategy defines the order of node expansion
  - I.e., the order in which nodes in the search tree are discovered
- Search strategies are evaluated along the following dimensions
  - Completeness: Always find a solution if one exists?
  - Optimality: Always find a least-cost solution?
  - Time complexity: Number of nodes generated (= edges traversed)
  - Space complexity: Max number of nodes in memory
- Time / space complexity are measured in terms of
  - $b$ : maximum branching factor of the search tree
  - $d$ : distance to root of the shallowest solution
  - $m$ : maximum length of any path in the state space

# Uninformed (Blind) Search Strategies

- Uninformed search strategies (also called blind search) use only the information available in the problem definition
- Examples
  - Breadth-first search (BFS)
  - Depth-first search (DFS)
  - Uniform-cost search (UCS)

# Search Algorithms

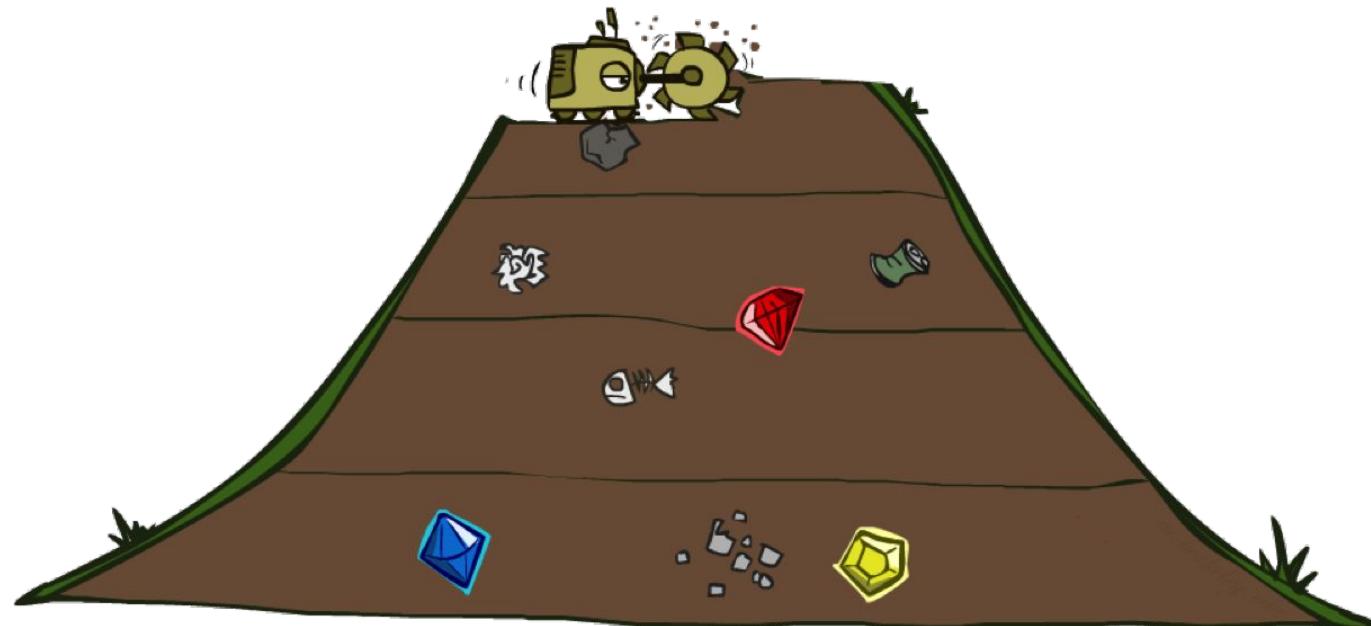
- Search algorithms come in two different flavors
  - Tree search algorithm (TSA)
  - Graph search algorithm (GSA)

# Tree Search Algorithm (TSA)

```
function TSA(problem) returns solution
    initialize frontier using initial state of problem
    while frontier is not empty
        choose a node and remove it from frontier
        if node contains a goal state then return corresponding solution
        explore the node, adding the resulting nodes to the frontier
```

# BFS

- The BFS search strategy explores the shallowest node in the search tree



<http://ai.berkeley.edu>

frontier is a queue (first in first out data structure)

# Queue in Python

- We are going to use the deque in Python
  - Example usage:

```
>>> import collections
>>> queue = collections.deque(['A', 'B', 'C', 'D'])
>>> queue.popleft()
'A'
>>> queue
deque(['B', 'C', 'D'])
>>> queue.popleft()
'B'
>>> queue
deque(['C', 'D'])
```

# Examples

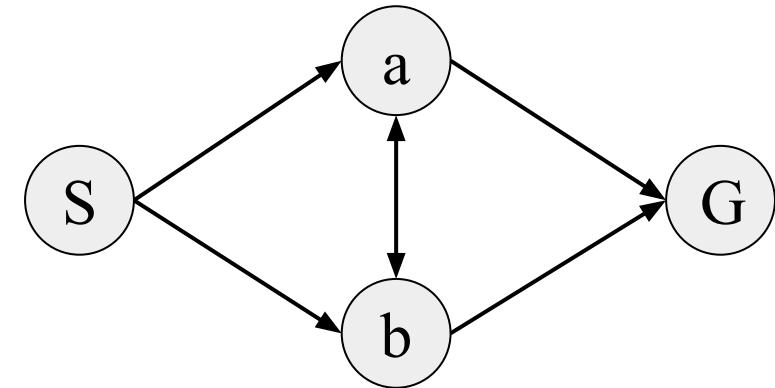
- In the following examples we will use

- characters for states, e.g,

- 'S'
    - 'a'

- strings for state sequences

- 'Sa'
    - 'SabG'



# Tree Search Algorithm (TSA) - BFS Version

```
function TSA(problem) returns solution
    initialize frontier using initial state of problem
    while frontier is not empty
        choose a node and remove it from frontier
        if node contains a goal state then return corresponding solution
        explore the node, adding the resulting nodes to the frontier
```

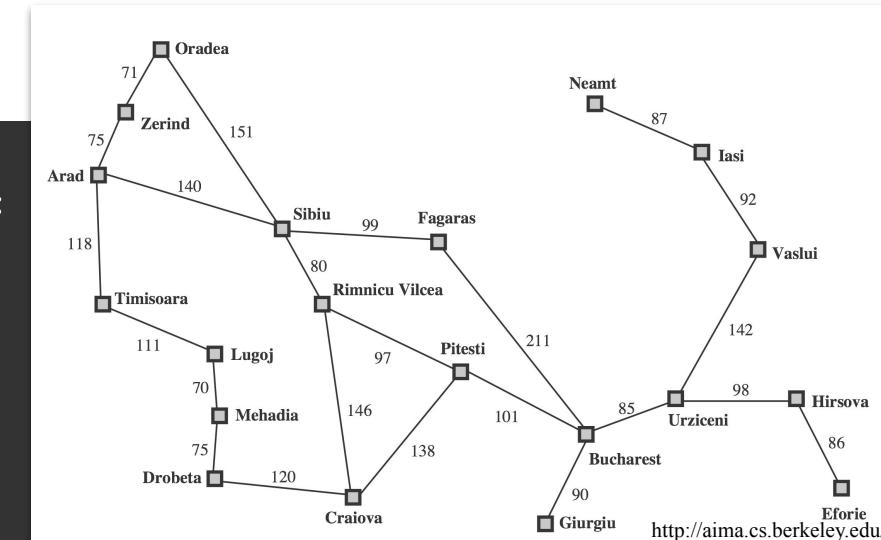
```
import collections
def bfsTsa(stateSpaceGraph, startState, goalState):
    frontier = collections.deque([startState])
    while frontier:
        node = frontier.popleft()
        if (node.endswith(goalState)): return node
        for child in stateSpaceGraph[node[-1]]: frontier.append(node+child)
```

# BFS-TSA Romania Code

```

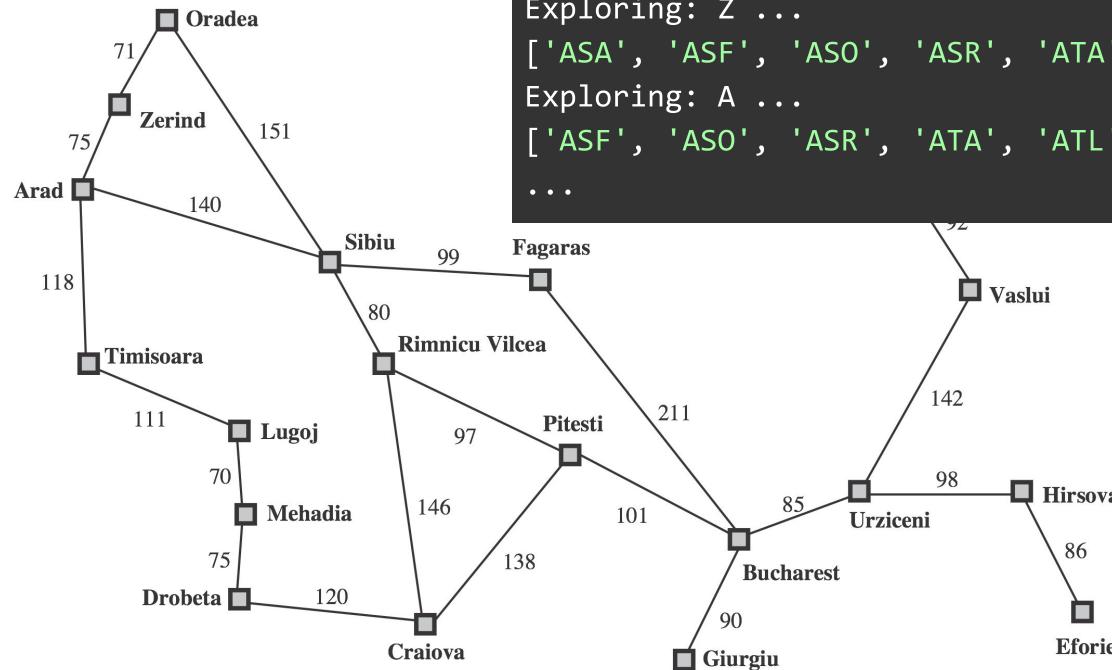
import collections
def bfsTsa(stateSpaceGraph, startState, goalState):
    frontier = collections.deque([startState])
    print('Initial frontier:',list(frontier))
    input()
    while frontier:
        node = frontier.popleft()
        if (node.endswith(goalState)): return node
        print('Exploring:',node[-1],...)
        for child in stateSpaceGraph[node[-1]]: frontier.append(node+child)
        print(list(frontier))
        input()
    romania = {
        'A':[['S','T','Z'],['A','O'],['S','Z'],['A','L'],['M','T'],['D','L']],
        'D':[['C','M'],['A','F','O','R'],['C','P','S'],['D','P','R']],
        'F':[['B','S'],['B','C','R']],
        'P':[],
        'R':[]
    }
    print('Solution path:',bfsTsa(romania, 'A', 'B'))

```



<http://aima.cs.berkeley.edu/>

# BFS-TSA Romania



```
Initial frontier: ['A']
Exploring: A ...
['AS', 'AT', 'AZ']
Exploring: S ...
['AT', 'AZ', 'ASA', 'ASF', 'ASO', 'ASR']
Exploring: T ...
['AZ', 'ASA', 'ASF', 'ASO', 'ASR', 'ATA', 'ATL']
Exploring: Z ...
['ASA', 'ASF', 'ASO', 'ASR', 'ATA', 'ATL', 'AZA', 'AZO']
Exploring: A ...
['ASF', 'ASO', 'ASR', 'ATA', 'ATL', 'AZA', 'AZO', 'ASAS', 'ASAT', 'ASAZ']
...
...
```

Will `bfsTsa(romania, 'A', 'B')` terminate ?

# Graph Search Algorithm (GSA) - BFS Version

```
function GSA (problem) returns solution
    initialize frontier using initial state of problem
    initialize explored set to be empty
    while frontier is not empty
        choose a node and remove it from frontier
        if node contains a goal state then return corresponding solution
        If node is not in explored set
            add node to explored set
            explore the node, adding the resulting nodes to the frontier
```

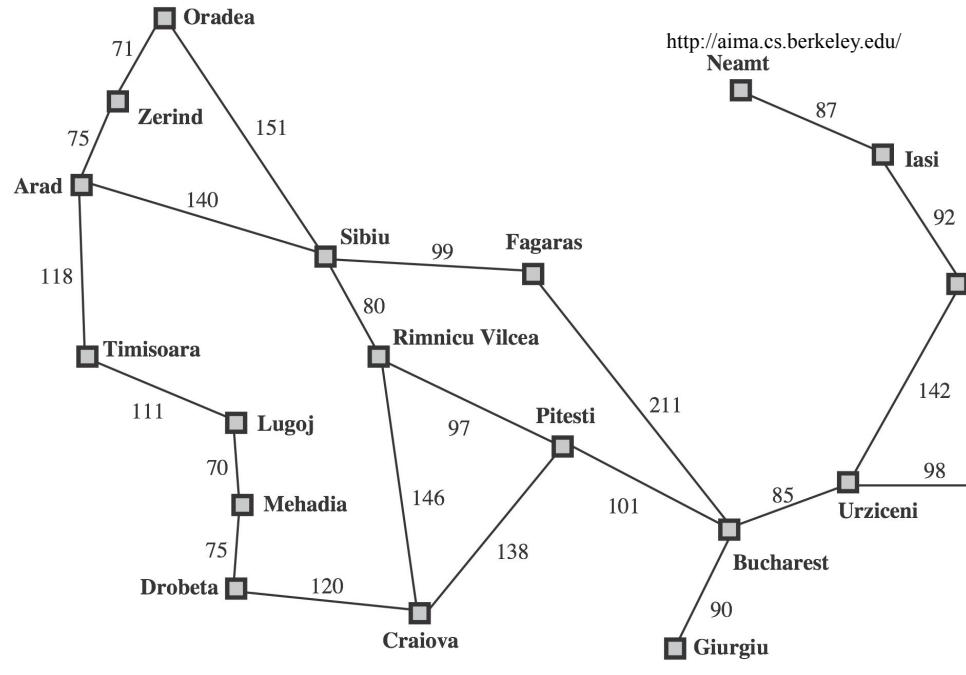
```
import collections
def bfsGsa(stateSpaceGraph, startState, goalState):
    frontier = collections.deque([startState])
    exploredSet = set()
    while frontier:
        node = frontier.popleft()
        if (node.endswith(goalState)): return node
        if node[-1] not in exploredSet:
            exploredSet.add(node[-1])
            for child in stateSpaceGraph[node[-1]]: frontier.append(node+child)
```

```
import collections
def bfsGsa(stateSpaceGraph, startState, goalState):
    frontier = collections.deque([startState])
    exploredSet = set()
    print('Initial frontier:',list(frontier))
    input()
    while frontier:
        node = frontier.popleft()
        if (node.endswith(goalState)): return node
        if node[-1] not in exploredSet:
            print('Exploring:',node[-1],...)
            exploredSet.add(node[-1])
            for child in stateSpaceGraph[node[-1]]: frontier.append(node+child)
            print(list(frontier))
            print(exploredSet)
            input()
romania = {
    'A': ['S', 'T', 'Z'], 'Z': ['A', 'O'], 'O': ['S', 'Z'],
    'T': ['A', 'L'], 'L': ['M', 'T'], 'M': ['D', 'L'],
    'D': ['C', 'M'], 'S': ['A', 'F', 'O', 'R'],
    'R': ['C', 'P', 'S'], 'C': ['D', 'P', 'R'],
    'F': ['B', 'S'], 'P': ['B', 'C', 'R'], 'B': []
}
print('Solution path:',bfsGsa(romania, 'A', 'B'))
```

```

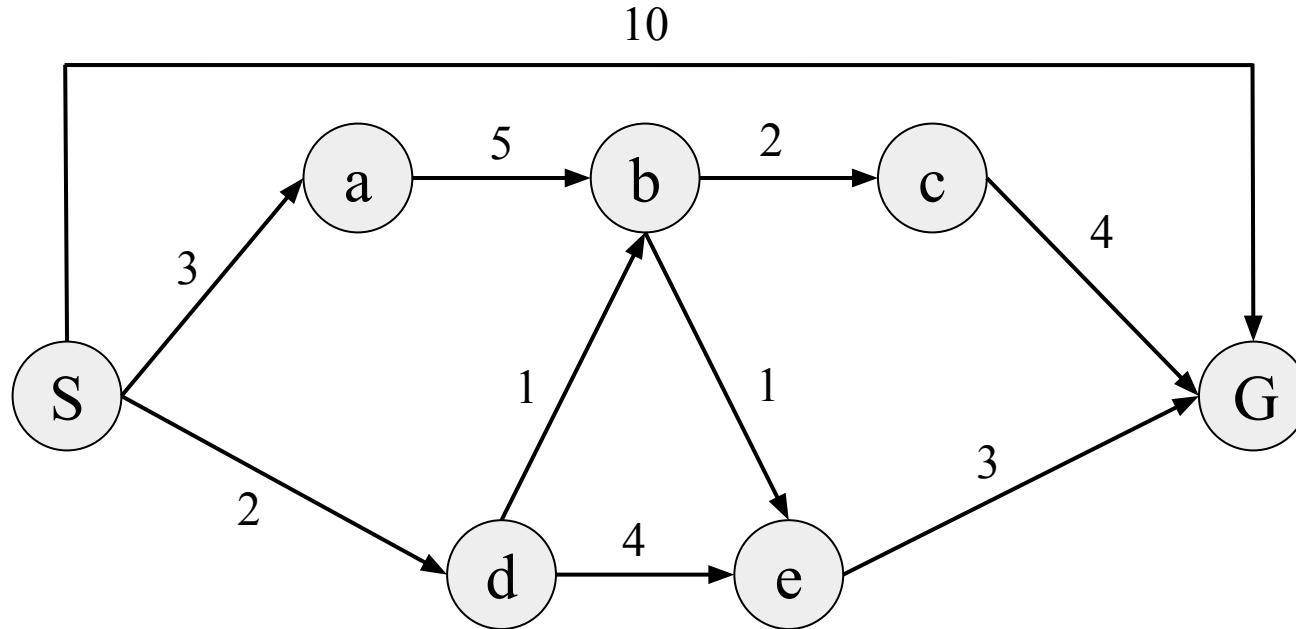
Initial frontier: ['A']
Exploring: A ...
['AS', 'AT', 'AZ']
{'A'}
Exploring: S ...
['AT', 'AZ', 'ASA', 'ASF', 'ASO', 'ASR']
{'S', 'A'}
Exploring: T ...
['AZ', 'ASA', 'ASF', 'ASO', 'ASR', 'ATA', 'ATL']
{'S', 'A', 'T'}
Exploring: Z ...
['ASA', 'ASF', 'ASO', 'ASR', 'ATA', 'ATL', 'AZA', 'AZO']
{'S', 'A', 'Z', 'T'}
Exploring: F ...
['ASO', 'ASR', 'ATA', 'ATL', 'AZA', 'AZO', 'ASFB', 'ASFS']
{'A', 'Z', 'T', 'S', 'F'}
Exploring: O ...
['ASR', 'ATA', 'ATL', 'AZA', 'AZO', 'ASFB', 'ASFS', 'ASOS', 'ASOZ']
{'A', 'Z', 'T', 'S', 'O', 'F'}
Exploring: R ...
['ATA', 'ATL', 'AZA', 'AZO', 'ASFB', 'ASFS', 'ASOS', 'ASOZ', 'ASRC', 'ASRP', 'ASRS']
{'A', 'R', 'Z', 'T', 'S', 'O', 'F'}
Exploring: L ...
['AZA', 'AZO', 'ASFB', 'ASFS', 'ASOS', 'ASOZ', 'ASRC', 'ASRP', 'ASRS', 'ATLM', 'ATLT']
{'A', 'R', 'L', 'Z', 'T', 'S', 'O', 'F'}
Solution path: ASFB

```

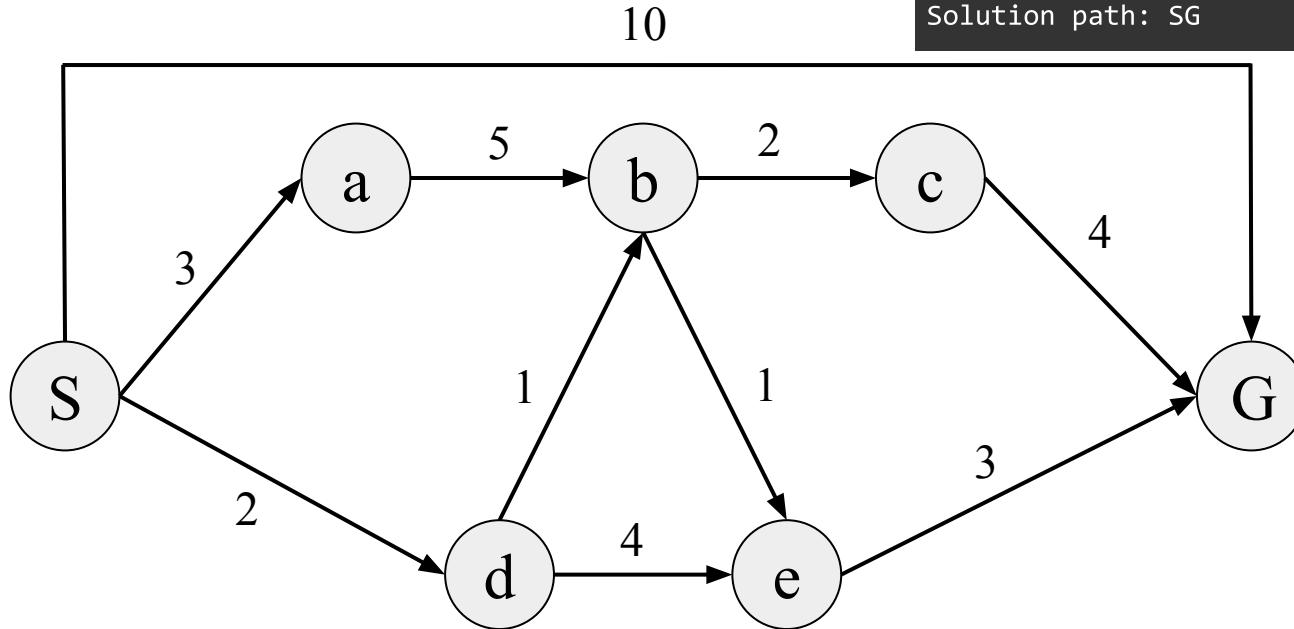


# BFS-GSA Practice

- Consider the following state space graph
  - Let S be the start state and G be the goal state
- Perform **BFS-GSA** and write down the order of explored states
  - Assume children are added into the frontier in alphabetical order



# BFS-GSA Practice Solution



```
Initial frontier: ['S']
Exploring: S ...
['Sa', 'Sd', 'SG']
{'S'}
Exploring: a ...
['Sd', 'SG', 'Sab']
{'S', 'a'}
Exploring: d ...
['SG', 'Sab', 'Sdb', 'Sde']
{'S', 'a', 'd'}
Solution path: SG
```

Try it [here](#)  
[bfsTsaPractice.py](#)  
[bfsGsaPractice.py](#)

# BFS Properties

- Complete
- Optimal
- Complexity
  - Time
  - Space

Assume:  $b = 10$ , 1 million nodes/second, 1000 bytes/node

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	$10^6$	1.1 seconds	1 gigabyte
8	$10^8$	2 minutes	103 gigabytes
10	$10^{10}$	3 hours	10 terabytes
12	$10^{12}$	13 days	1 petabyte
14	$10^{14}$	3.5 years	99 petabytes
16	$10^{16}$	350 years	10 exabytes

<http://aima.cs.berkeley.edu/>

b: maximum branching factor of the search tree

d: depth of the shallowest solution

# DFS

- DFS explores the deepest node in the search tree



<http://ai.berkeley.edu>

frontier is a stack (last in first out data structure)

# Stack in Python

- We are going to use the deque in Python
  - Example usage:

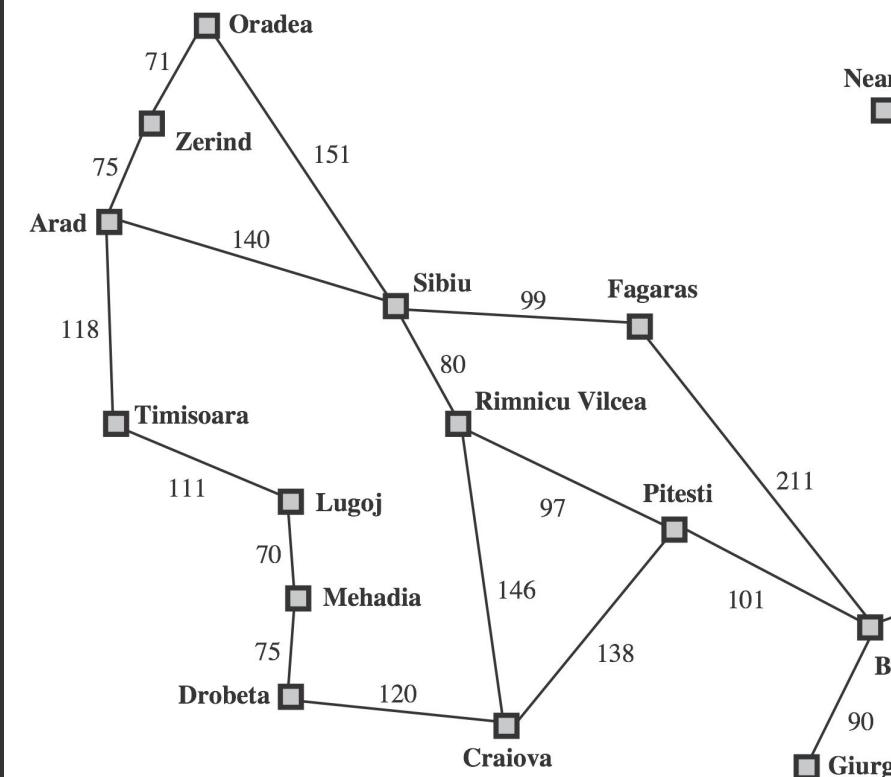
```
>>> import collections
>>> queue = collections.deque(['A', 'B', 'C', 'D'])
>>> queue.pop()
'D'
>>> queue
deque(['A', 'B', 'C'])
>>> queue.pop()
'C'
>>> queue
deque(['A', 'B'])
```

```
import collections
def dfsGsa(stateSpaceGraph, startState, goalState):
    frontier = collections.deque([startState])
    exploredSet = set()
    print('Initial frontier:',list(frontier))
    input()
    while frontier:
        node = frontier.pop()
        if (node.endswith(goalState)): return node
        if node[-1] not in exploredSet:
            print('Exploring:',node[-1],...)
            exploredSet.add(node[-1])
            for child in stateSpaceGraph[node[-1]]: frontier.append(node+child)
            print(list(frontier))
            print(exploredSet)
            input()
romania = {
    'A':['S','T','Z'],'Z':['A','O'],'O':['S','Z'],'T':['A','L'],'L':['M','T'],'M':['D','L'],
    'D':['C','M'],'S':['A','F','O','R'],'R':['C','P','S'],'C':['D','P','R'],
    'F':['B','S'],'P':['B','C','R'],'B':[] }
print('Solution path:',dfsGsa(romania, 'A', 'B'))
```

```

Initial frontier: ['A']
Exploring: A ...
['AS', 'AT', 'AZ']
{'A'}
Exploring: Z ...
['AS', 'AT', 'AZA', 'AZO']
{'Z', 'A'}
Exploring: O ...
['AS', 'AT', 'AZA', 'AZOS', 'AZOZ']
{'Z', 'O', 'A'}
Exploring: S ...
['AS', 'AT', 'AZA', 'AZOSA', 'AZOSF', 'AZOSO', 'AZOSR']
{'Z', 'O', 'S', 'A'}
Exploring: R ...
['AS', 'AT', 'AZA', 'AZOSRC', 'AZOSRP', 'AZOSRS']
{'A', 'Z', 'R', 'O', 'S'}
Exploring: P ...
['AS', 'AT', 'AZA', --- 'AZOSRPB', 'AZOSRPC', 'AZOSRPR']
{'A', 'P', 'Z', 'R', 'O', 'S'}
Exploring: C ...
['AS', 'AT', 'AZA', --- 'AZOSRPCD', 'AZOSRPCP', 'AZOSRPCR']
{'C', 'A', 'P', 'Z', 'R', 'O', 'S'}
Exploring: D ...
['AS', 'AT', 'AZA', --- 'AZOSRPB', 'AZOSRPCDC', 'AZOSRPCDM']
{'D', 'C', 'A', 'P', 'Z', 'R', 'O', 'S'}

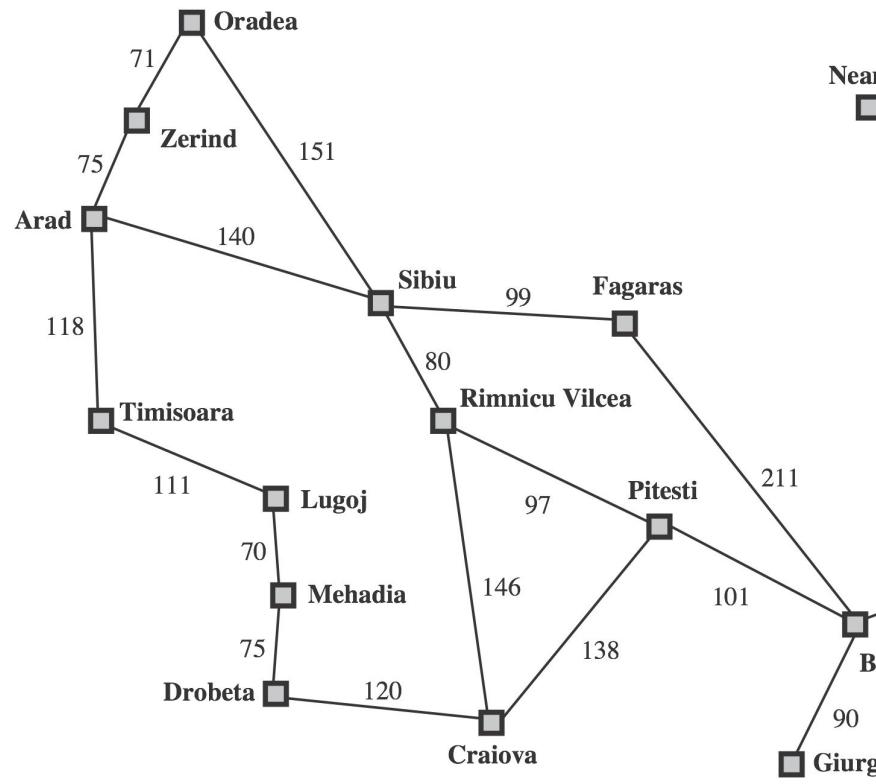
```



```

...
Exploring: M ...
['AS', 'AT', 'AZA', --- 'AZOSRPCDC', 'AZOSRPCDMD', 'AZOSRPCDML'
{'D', 'C', 'A', 'P', 'Z', 'R', 'O', 'S', 'M'}
Exploring: L ...
['AS', 'AT', 'AZA', --- 'AZOSRPCDMD', 'AZOSRPCDMLM', 'AZOSRPCDMLT']
{'D', 'C', 'A', 'P', 'Z', 'R', 'L', 'O', 'S', 'M'}
Exploring: T ...
['AS', 'AT', 'AZA', --- 'AZOSRPCDMLM', 'AZOSRPCDMLTA', 'AZOSRPCDMLTL']
{'D', 'C', 'A', 'P', 'Z', 'T', 'R', 'L', 'O', 'S', 'M'}
Solution path: AZOSRPB

```

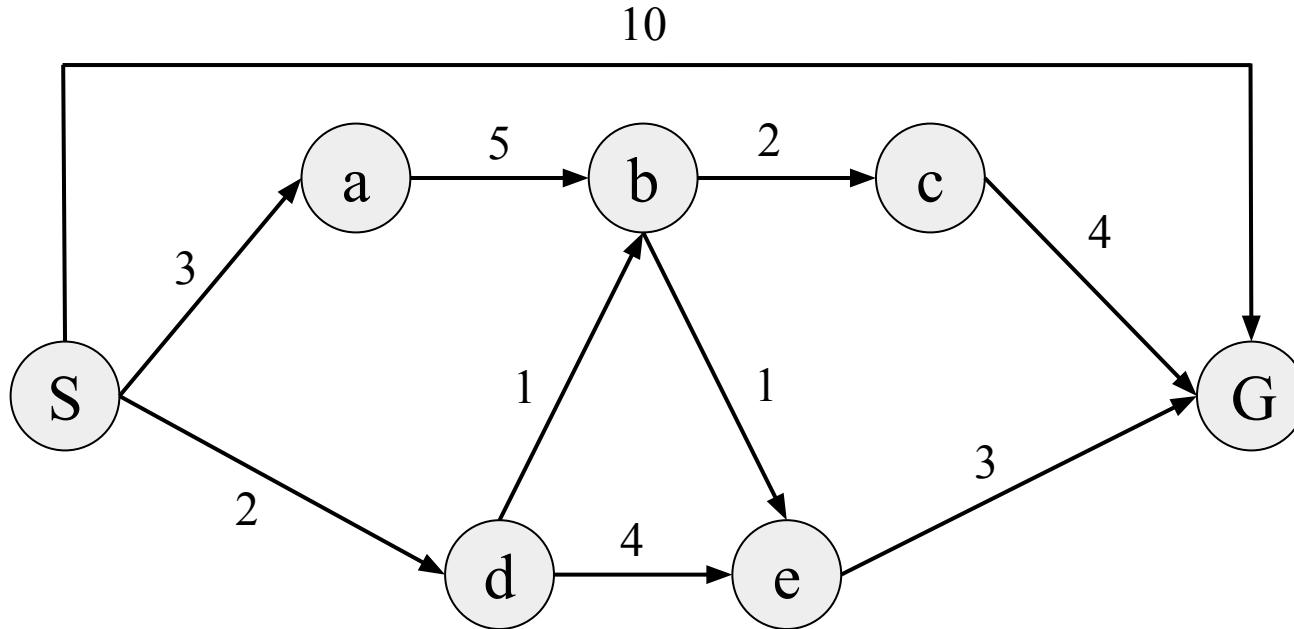


<http://aima.cs.berkeley.edu/>

Try it [here](#)  
[dfsTsaRomania.py](#)  
[dfsGsaRomania.py](#)

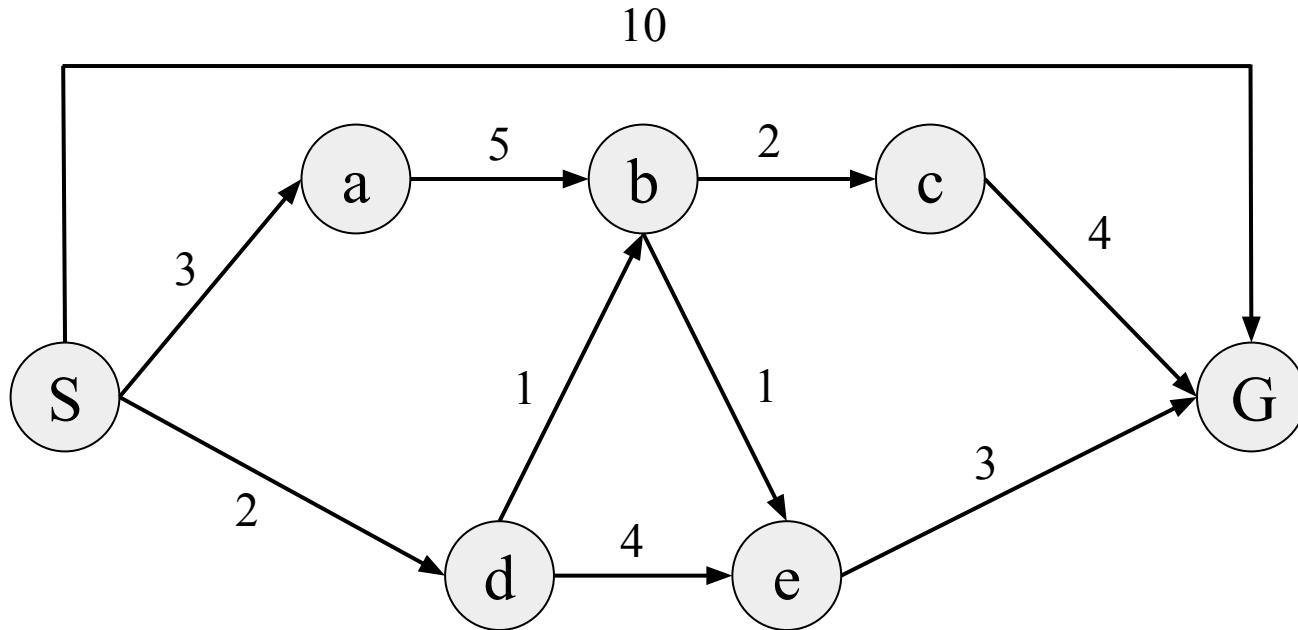
# DFS-TSA Practice

- Consider the following state space graph
  - Let S be the start state and G be the goal state
- Perform **DFS-TSA** and write down the order of explored states
  - Assume children are added into the frontier in alphabetical order



# DFS-TSA Practice Solution

```
Initial frontier: ['S']
Exploring: S ...
['Sa', 'Sd', 'SG']
Solution path: SG
```



# DFS Properties

- Complete
- Optimal
- Complexity
  - Time
  - Space

b: maximum branching factor of the search tree

m: maximum length of any path in the state space

# BFS vs. DFS

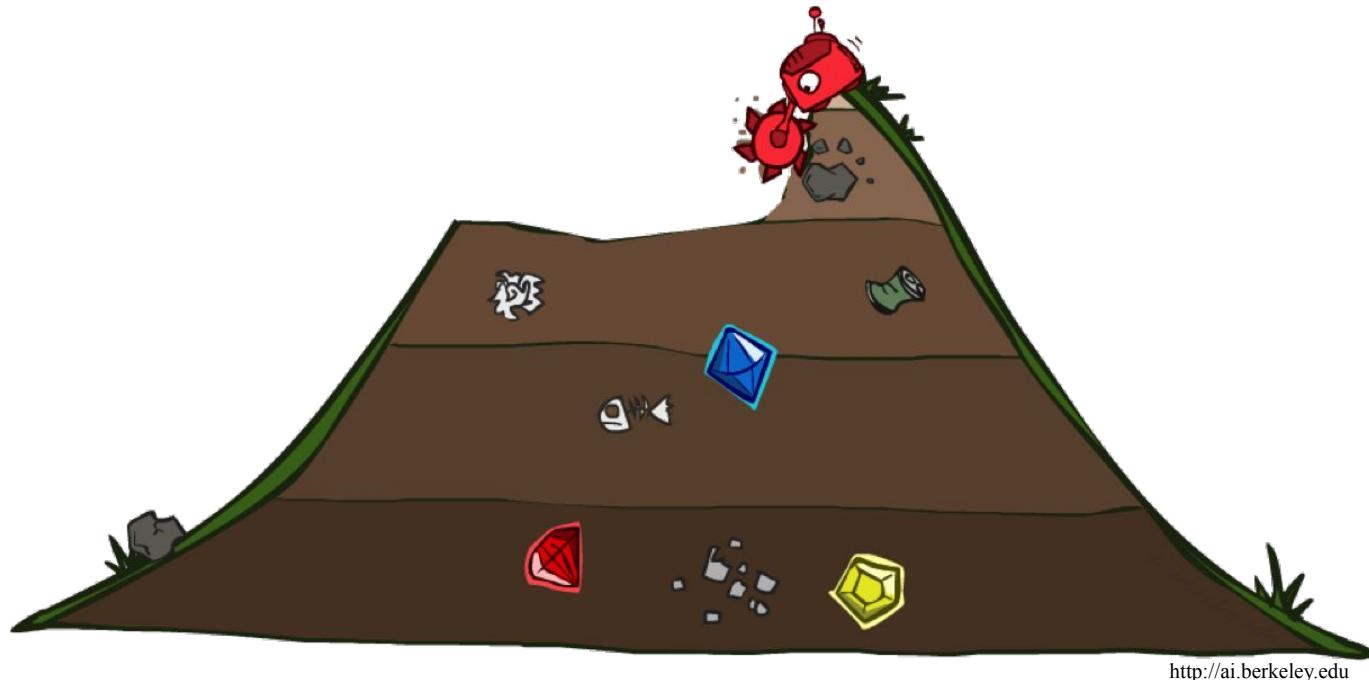
- When will BFS outperform DFS?
  - If solutions are close to the root of the search tree
- When will DFS outperform BFS?
  - If all solutions are deep inside the search tree

# GSA vs. TSA

- GSA
  - Avoids infinite loops
  - Eliminates exponentially many redundant paths
  - Requires memory proportional to its runtime
- TSA
  - Could be stuck in infinite loops
  - Explores redundant paths
  - Requires less memory
  - Easier to implement

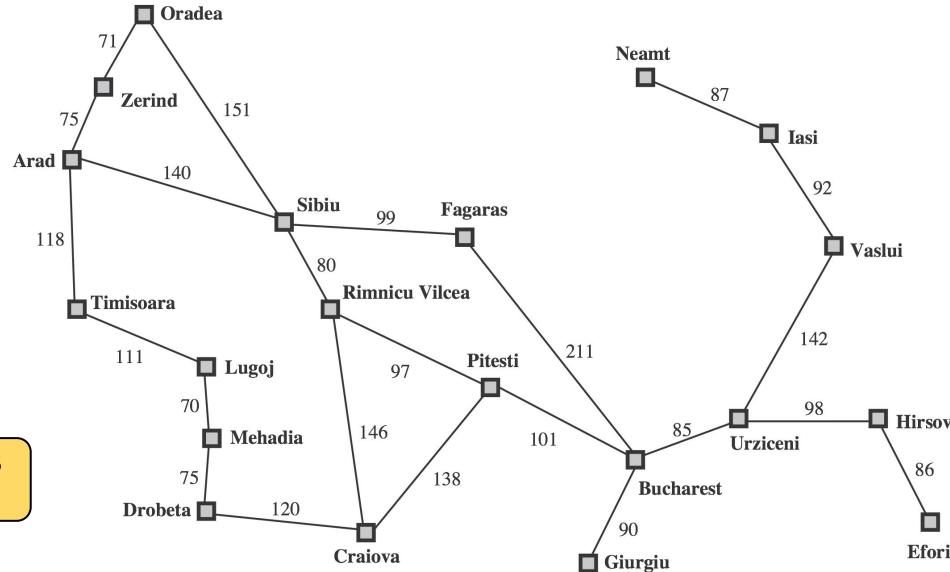
# UCS

- UCS explores the cheapest node first



frontier is a priority queue (queue data structure where each element has a priority)

# Updated Romania Problem Definition



Updated version includes  
the distances

This is the cost  
from O to Z

```
romania = {
    'A':[(140, 'S'), (118, 'T'), (75, 'Z')], 'Z':[(75, 'A'), (71, 'O')], 'O':[(151, 'S'), (71, 'Z')],
    'T':[(118, 'A'), (111, 'L')], 'L':[(70, 'M'), (111, 'T')], 'M':[(75, 'D'), (70, 'L')],
    'D':[(120, 'C'), (75, 'M')], 'S':[(140, 'A'), (99, 'F'), (151, 'O'), (80, 'R')], 'F':[(211, 'B'), (99, 'S')],
    'R':[(146, 'C'), (97, 'P'), (80, 'S')], 'C':[(120, 'D'), (138, 'P'), (146, 'R')], 'P':[(101, 'B'), (138, 'C'), (97, 'R')], 'B':[]}
```

# Priority Queue in Python

We will use this library  
for our Priority Queue

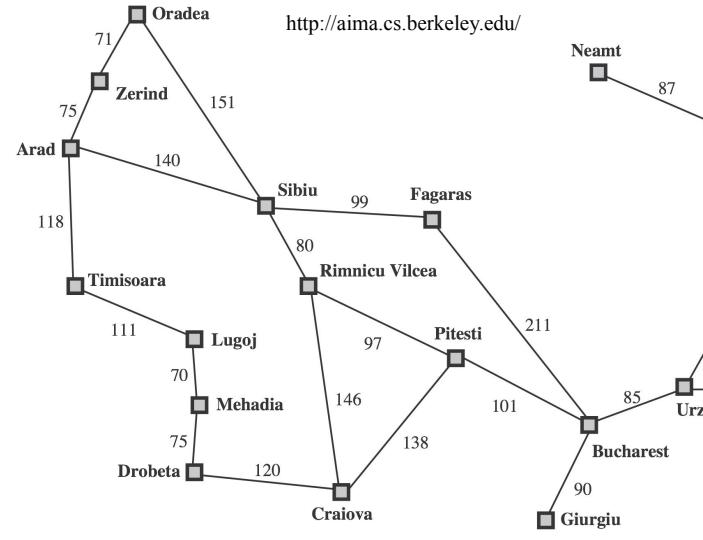
```
frontier = []
>>> from heapq import heappush, heappop
import random
>>> heappush(frontier, (random.randint(1, 10), 'A'))
>>> heappush(frontier, (random.randint(1, 10), 'B'))
>>> heappush(frontier, (random.randint(1, 10), 'C'))
>>> frontier
[(2, 'C'), (7, 'B'), (6, 'A')]
>>> heappop(frontier)
(2, 'C')
>>> heappop(frontier)
(6, 'A')
>>> heappop(frontier)
(7, 'B')
>>> (1, 'B') < (2, 'A')
True
>>> (1, 'B') < (1, 'A')
False
```

This is how tuples are  
compared in Python

```

from heapq import heappush, heappop
def ucsGsa(stateSpaceGraph, startState, goalState):
    frontier = []
    heappush(frontier, (0, startState))
    exploredSet = set()
    print('Initial frontier:', list(frontier)); input()
    while frontier:
        node = heappop(frontier)
        if (node[1].endswith(goalState)): return node
        if node[1][-1] not in exploredSet:
            print('Exploring:', node[1][-1], 'at cost', node[0])
            exploredSet.add(node[1][-1])
            for child in stateSpaceGraph[node[1][-1]]:
                heappush(frontier, (node[0]+child[0], node[1]+child[1]))
            print(list(frontier)); print(exploredSet); input()
romania = {
'A':[(140, 'T'),(118, 'T'),(75, 'Z')], 'Z':[(75, 'A'),(71, 'O')], 'O':[(151, 'S'),(71, 'Z')], 
'T':[(118, 'A'),(111, 'L')], 'L':[(70, 'M'),(111, 'T')], 'M':[(75, 'D'),(70, 'L')], 
'D':[(120, 'C'),(75, 'M')], 'S':[(140, 'A'),(99, 'F'),(151, 'O'),(80, 'R')], 
'R':[(146, 'C'),(97, 'P'),(80, 'S')], 'C':[(120, 'D'),(138, 'P'),(146, 'R')], 
'F':[(211, 'B'),(99, 'S')], 'P':[(101, 'B'),(138, 'C'),(97, 'R')], 'B':[]}
print('Solution path:', ucsGsa(romania, 'A', 'B'))

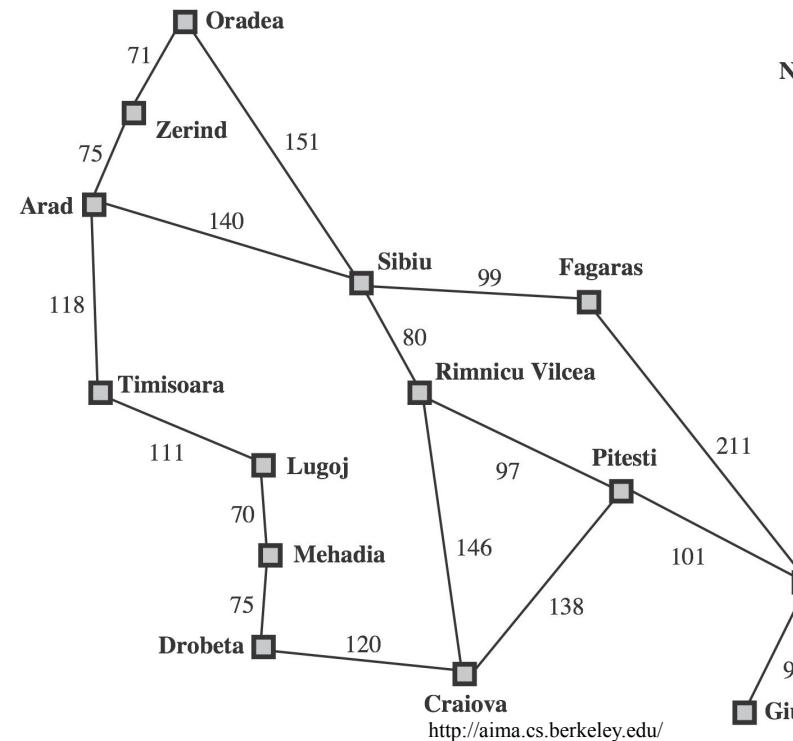
```



```

Initial frontier: [(0, 'A')]
Exploring: A at cost 0
[(75, 'AZ'), (140, 'AS'), (118, 'AT')]
{'A'}
Exploring: Z at cost 75
[(118, 'AT'), (140, 'AS'), (150, 'AZA'), (146, 'AZO')]
{'Z', 'A'}
Exploring: T at cost 118
[(140, 'AS'), (146, 'AZO'), (150, 'AZA'), (236, 'ATA'), (229, 'ATL')]
{'Z', 'T', 'A'}
Exploring: S at cost 140
[(146, 'AZO'), (220, 'ASR'), --- (291, 'ASO'), (236, 'ATA')]
{'Z', 'T', 'A', 'S'}
Exploring: O at cost 146
[(150, 'AZA'), (217, 'AZOZ'), --- (297, 'AZOS'), (229, 'ATL')]
{'A', 'O', 'S', 'Z', 'T'}
Exploring: R at cost 220
[(229, 'ATL'), (280, 'ASA'), --- (317, 'ASRP'), (300, 'ASRS')]
{'R', 'A', 'O', 'S', 'Z', 'T'}
Exploring: L at cost 229
[(236, 'ATA'), (280, 'ASA'), --- (299, 'ATLM'), (340, 'ATLT')]
{'R', 'A', 'O', 'S', 'L', 'Z', 'T'}
Exploring: F at cost 239
[(280, 'ASA'), (291, 'ASO'), --- (450, 'ASFB'), (338, 'ASFS')]
{'R', 'A', 'O', 'S', 'L', 'F', 'Z', 'T'}

```

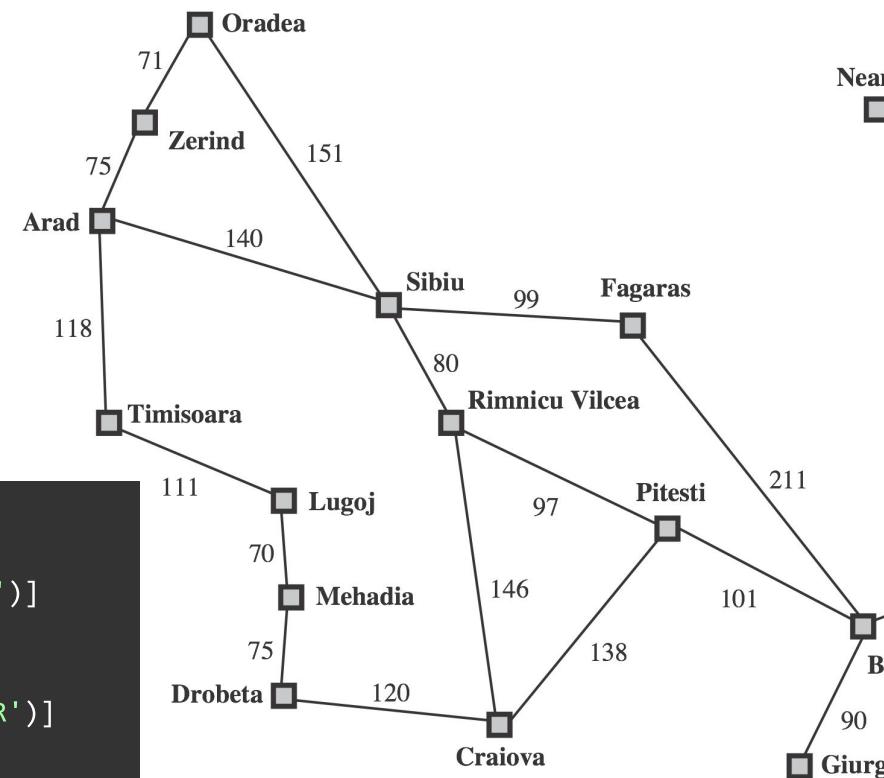


<http://aima.cs.berkeley.edu/>

```

...
Exploring: M at cost 299
[(300, 'ASRS'), (317, 'ASRP'), --- (374, 'ATLMD'), (369, 'ATMLM')]
{'M', 'R', 'A', 'O', 'S', 'L', 'F', 'Z', 'T'}
Exploring: P at cost 317
[(338, 'ASFS'), (369, 'ATMLM'), --- (455, 'ASRPC'), (414, 'ASPRP')]
{'M', 'P', 'R', 'A', 'O', 'S', 'L', 'F', 'Z', 'T'}
Exploring: C at cost 366
[(369, 'ATMLM'), (374, 'ATLMD'), --- (504, 'ASRCP'), (512, 'ASRCR')]
{'M', 'P', 'R', 'A', 'O', 'S', 'L', 'F', 'Z', 'T', 'C'}
Exploring: D at cost 374
[(414, 'ASPRP'), (418, 'ASRPB'), --- (504, 'ASRCP'), (494, 'ATLMDC')]
{'M', 'P', 'R', 'A', 'D', 'O', 'S', 'L', 'F', 'Z', 'T', 'C'}
Solution path: (418, 'ASRPB')

```

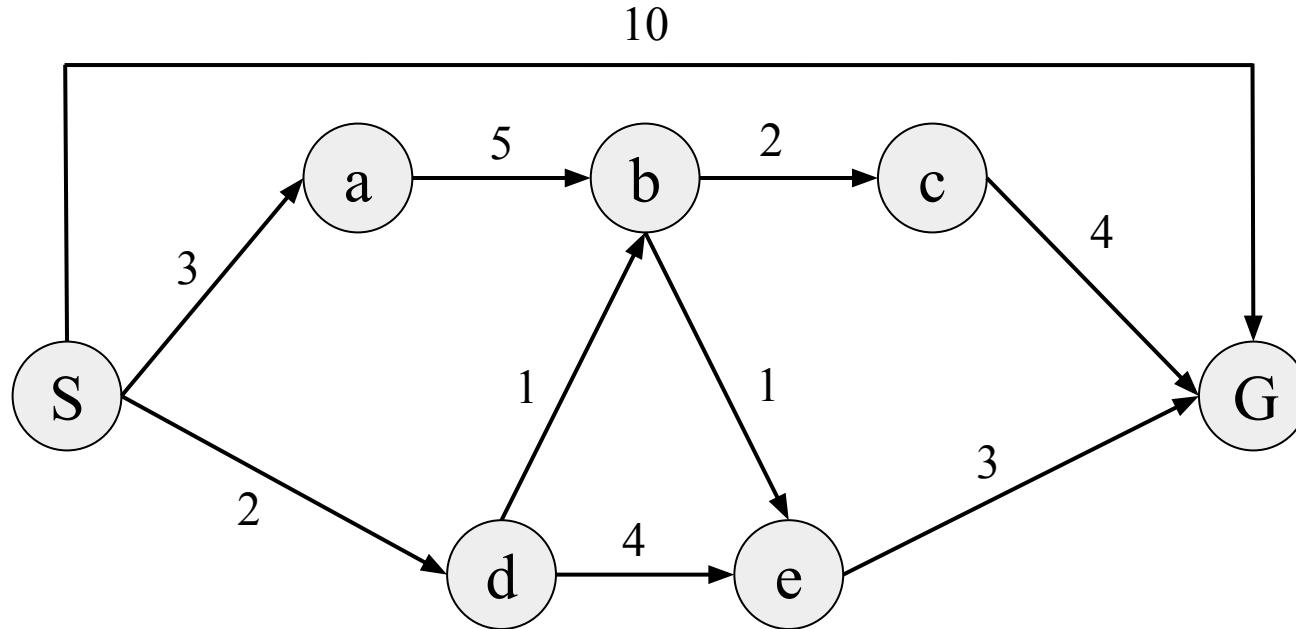


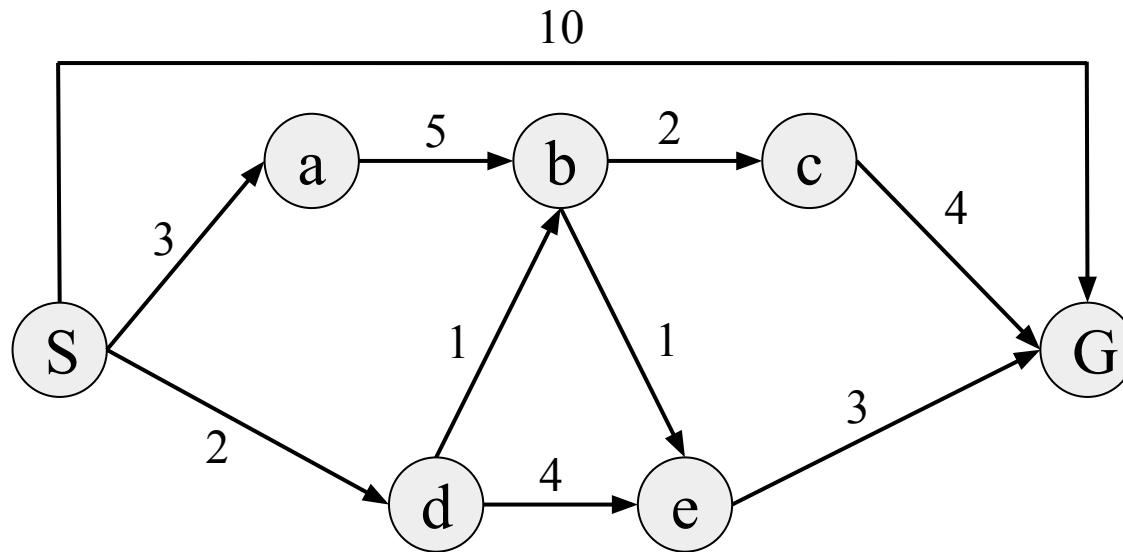
<http://aima.cs.berkeley.edu/>

Try it [here](#)  
[ucsTsaRomania.py](#)  
[ucsGsaRomania.py](#)

# UCS-GSA Practice

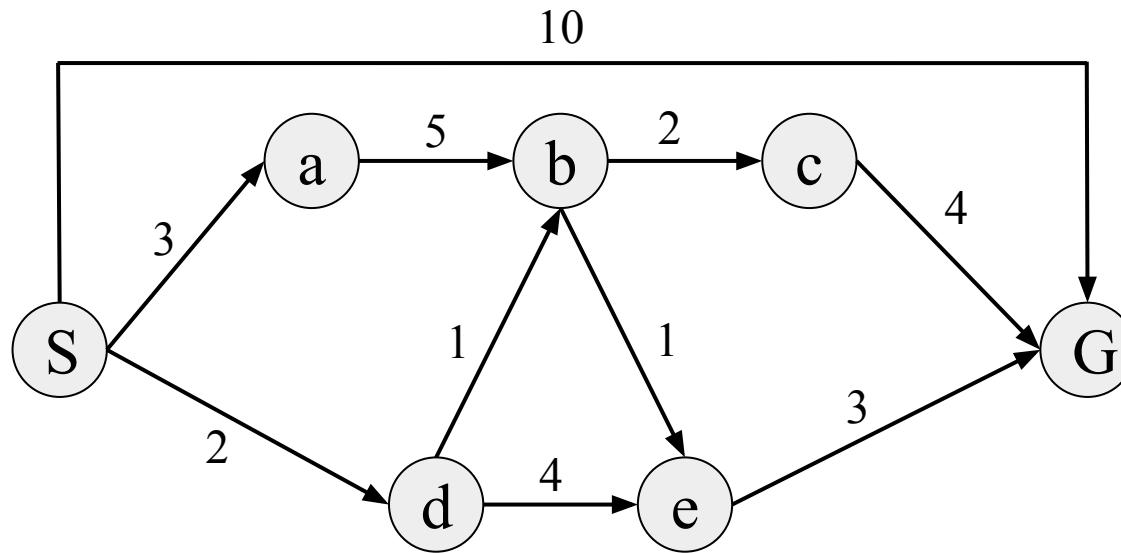
- Consider the following state space graph
  - Let S be the start state and G be the goal state
- Perform **UCS-GSA** and write down the order of explored states
  - Assume children are added into the frontier in alphabetical order





```

Initial frontier: [(0, 'S')]
Exploring: S at cost 0
[(2, 'Sd'), (3, 'Sa'), (10, 'SG')]
{'S'}
Exploring: d at cost 2
[(3, 'Sa'), (6, 'Sde'), (3, 'Sdb'), (10, 'SG')]
{'S', 'd'}
Exploring: a at cost 3
[(3, 'Sdb'), (6, 'Sde'), (10, 'SG'), (8, 'Sab')]
{'S', 'a', 'd'}
  
```



```

...
Exploring: b at cost 3
[(4, 'Sdbe'), (5, 'Sdbc'), (10, 'SG'), (8, 'Sab'), (6, 'Sde')]
{'b', 'S', 'a', 'd'}
Exploring: e at cost 4
[(5, 'Sdbc'), (6, 'Sde'), (10, 'SG'), (8, 'Sab'), (7, 'SdbeG')]
{'S', 'e', 'd', 'a', 'b'}
Exploring: c at cost 5
[(6, 'Sde'), (7, 'SdbeG'), (10, 'SG'), (8, 'Sab'), (9, 'SdbcG')]
{'S', 'e', 'd', 'c', 'a', 'b'}
Solution path: (7, 'SdbeG')

```

Try it [here](#)  
[ucsTsaPractice.py](#)  
[ucsGsaPractice.py](#)

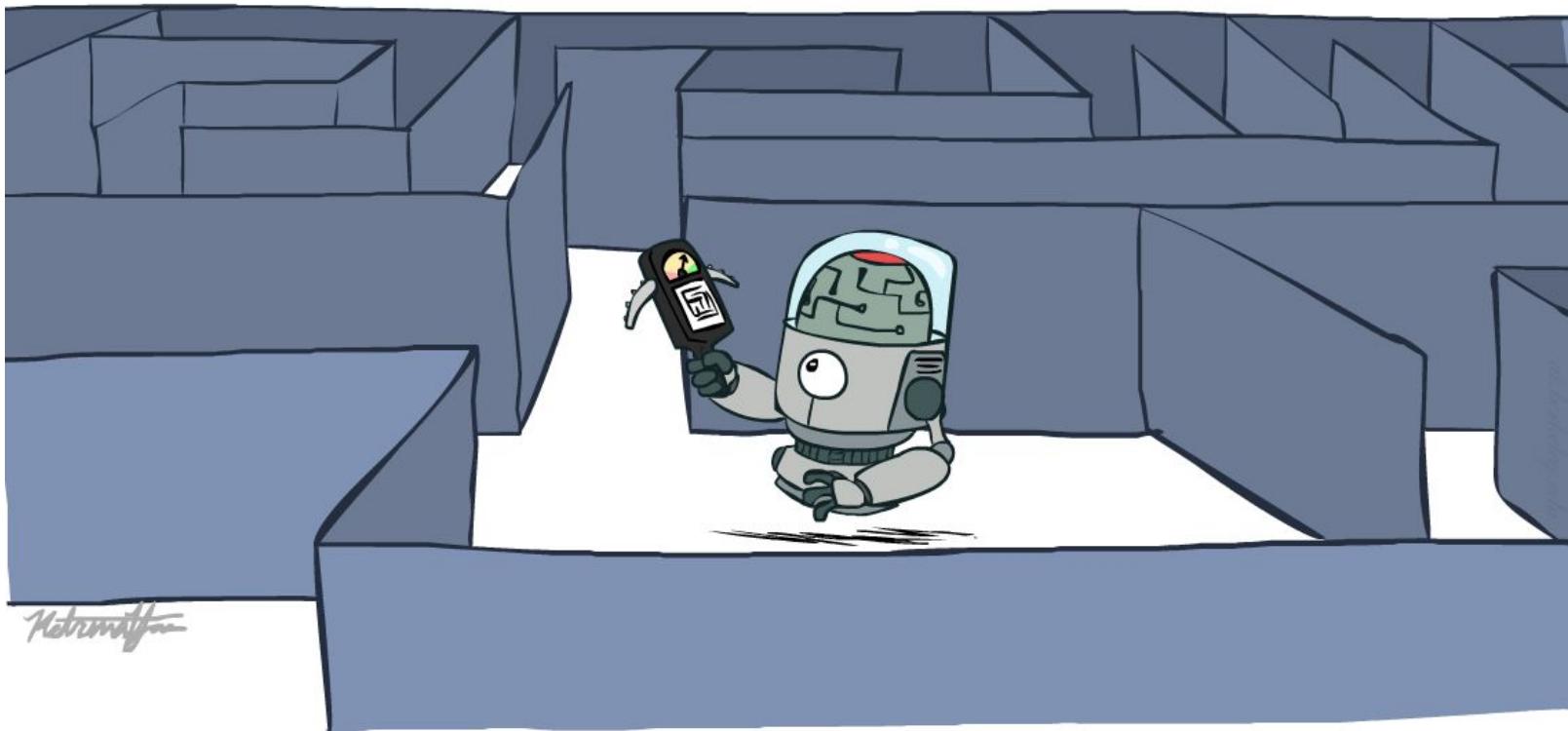
# UCS Properties

- Complete
- Optimal
- Complexity
  - Time
  - Space

$C^*$ : cost of optimal solution

epsilon: smallest path cost in search graph

# Informed Search

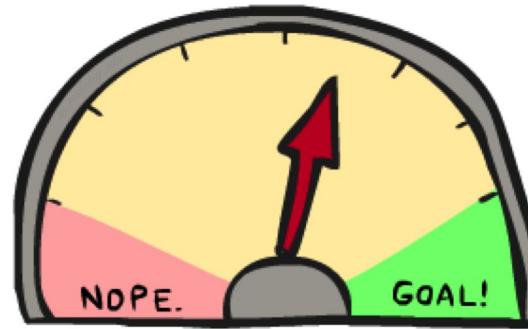


# Informed Search

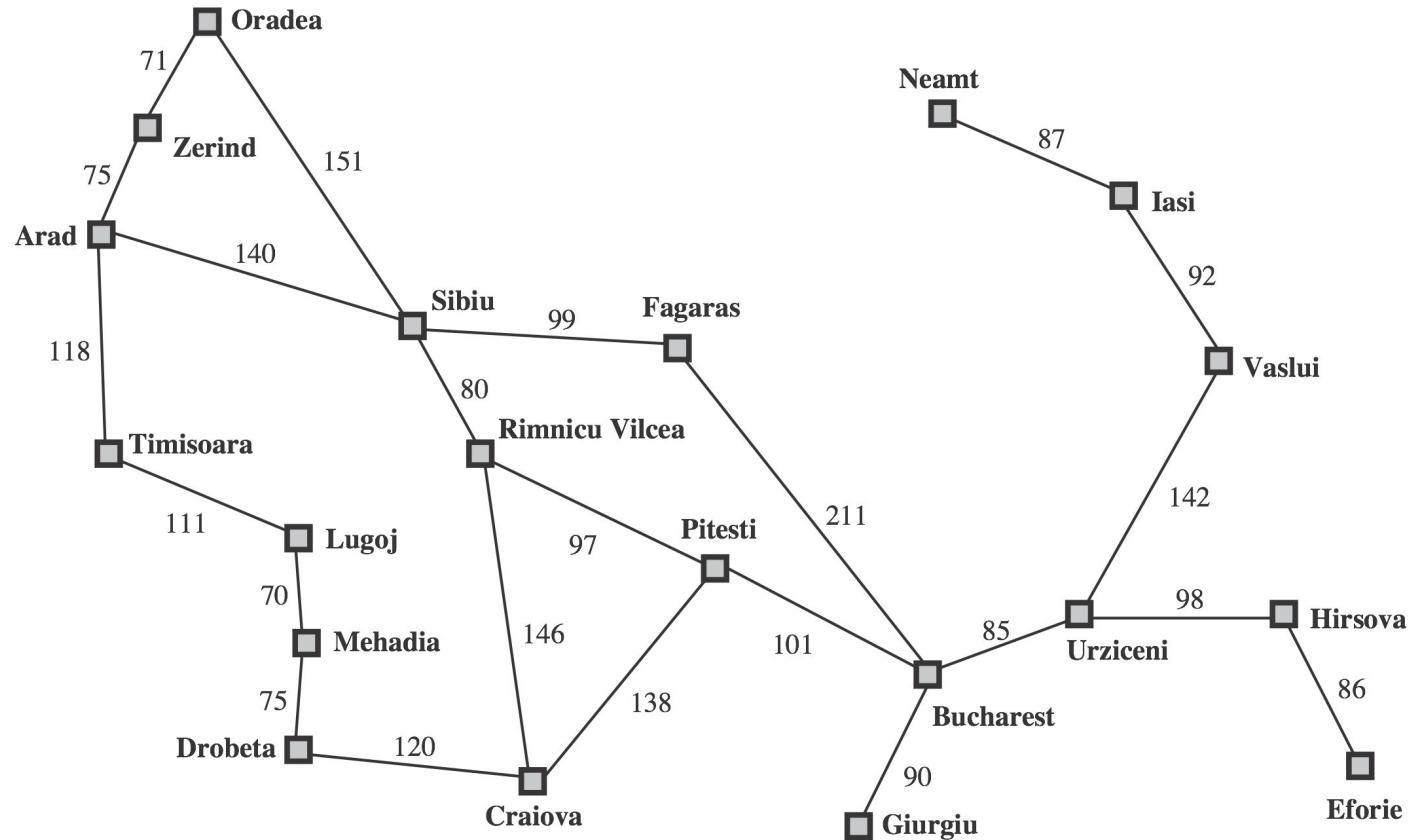
- Informed search strategies find solutions more efficiently than uninformed search strategies
- They employ problem specific knowledge beyond the definition of the problem itself
  - Heuristic function
- Examples
  - Greedy best-first search
  - A\* search

# Heuristic Function

- A function that estimates how close you are to the goal
- Designed for a particular search problem
- $h(n)$ 
  - Cost (estimate) of the cheapest path from the state at node  $n$  to a goal state
  - If  $n$  is a goal node  $h(n) = 0$



# Romania Problem



Straight-line distance to B	
n	h(n)
A	366
B	0
C	160
D	242
E	161
F	176
G	77
H	151
I	226
L	244
M	241
N	234
O	380
P	100
R	193
S	253
T	329
U	80
V	199
Z	374

# Greedy

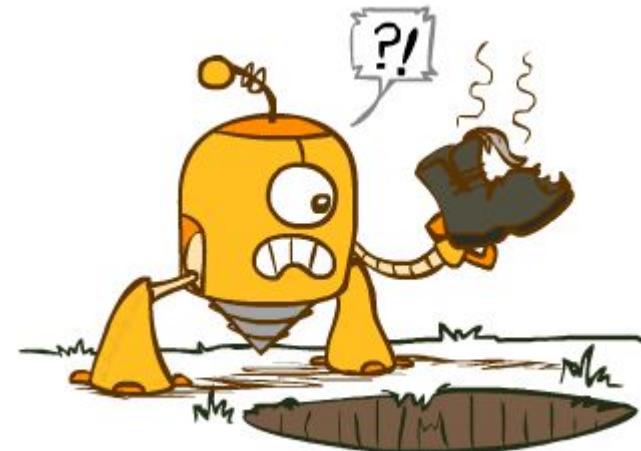


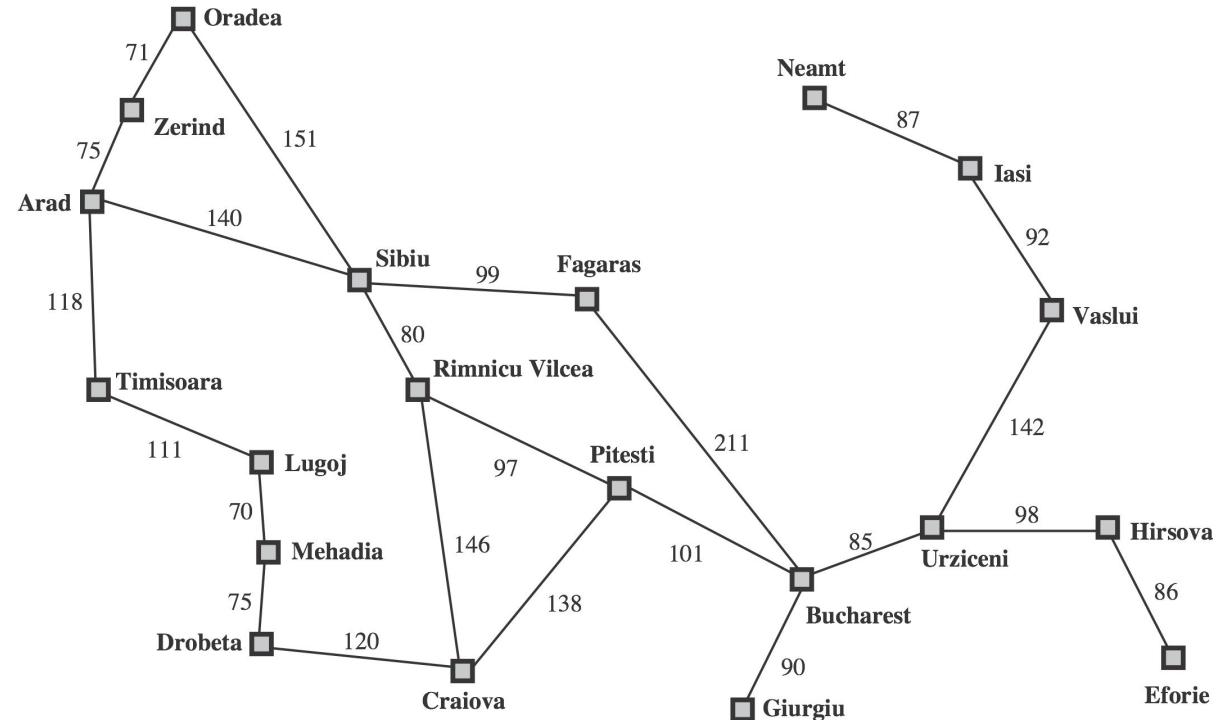
<http://ai.berkeley.edu>

frontier is a priority queue (queue data structure where each element has a priority)

# Greedy Search

- Also known as Best-first Search
- Expand the node that has the lowest  $h(n)$ 
  - What can possibly go wrong with this approach?





```

romaniaH = {
    'A':366, 'B':0, 'C':160, 'D':242, 'E':161, 'F':176, 'G':77, 'H':151, 'I':226,
    'L':244, 'M':241, 'N':234, 'O':380, 'P':100, 'R':193, 'S':253, 'T':329, 'U':80,
    'V':199, 'Z':374}
    
```

```
Initial frontier: [(366, 'A')]
```

```
Exploring: A at cost 366
```

```
[(253, 'AS'), (329, 'AT'), (374, 'AZ')]
{'A'}
```

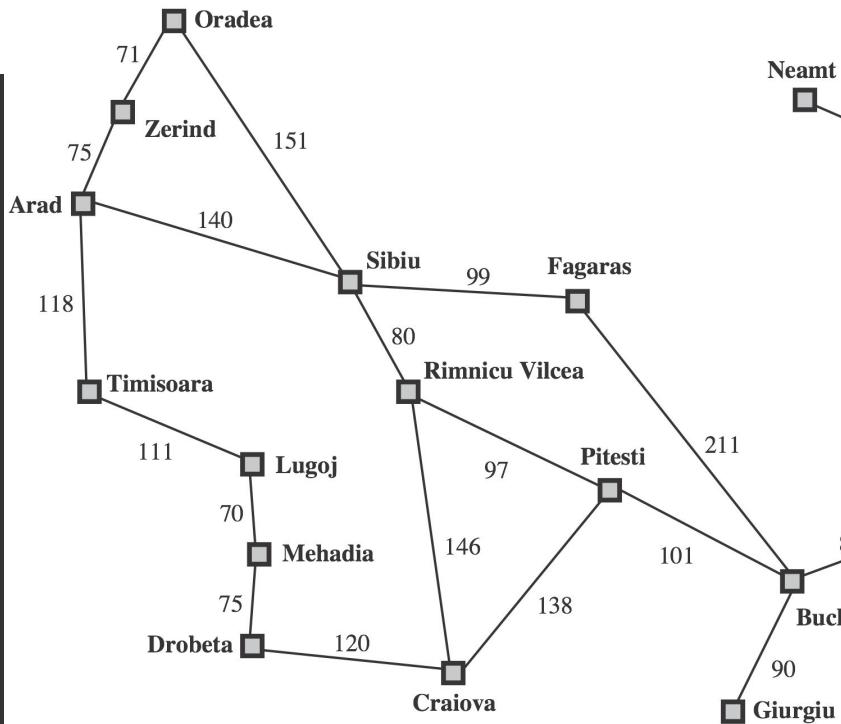
```
Exploring: S at cost 253
```

```
[(176, 'ASF'), (329, 'AT'), (193, 'ASR'),
(374, 'AZ'), (380, 'ASO'), (366, 'ASA')]
{'A', 'S'}
```

```
Exploring: F at cost 176
```

```
[(0, 'ASFB'), (329, 'AT'), (193, 'ASR'),
(374, 'AZ'), (380, 'ASO'), (366, 'ASA'),
(253, 'ASFS')]
{'F', 'A', 'S'}
```

```
Solution path: (0, 'ASFB')
```



Straight-line  
distance to B

n	h(n)
---	------

A	366
B	0
C	160
D	242
E	161
F	176
G	77
H	151
I	226
L	244
M	241
N	234
O	380
P	100
R	193
S	253
T	329
U	80
V	199
Z	374

Try it [here](#)

[greedyTsaRomania.py](#)

[greedyGsaRomania.py](#)

```

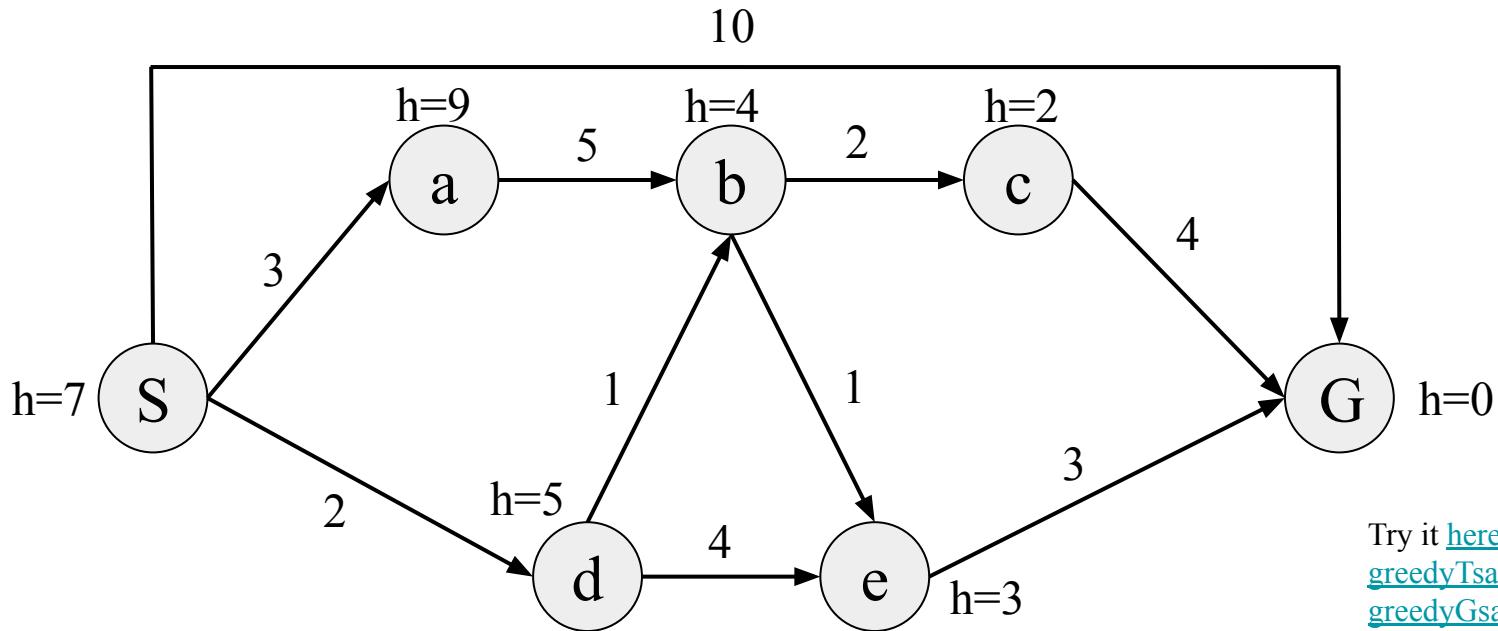
from heapq import heappush, heappop
def greedyTsa(stateSpaceGraph, h, startState, goalState):
    frontier = []
    heappush(frontier, (h[startState], startState))
    print('Initial frontier:',list(frontier)); input()
    while frontier:
        node = heappop(frontier)
        if (node[1].endswith(goalState)): return node
        print('Exploring:',node[1][-1],'at cost',node[0])
        for child in stateSpaceGraph[node[1][-1]]:
            heappush(frontier, (h[child[1]], node[1]+child[1]))
        print(list(frontier)); input()
romania = {
    'A':[(140,'S'),(118,'T'),(75,'Z')], 'Z':[(75,'A'),(71,'O')], 'O':[(151,'S'),(71,'Z')],
    'T':[(118,'A'),(111,'L')], 'L':[(70,'M'),(111,'T')], 'M':[(75,'D'),(70,'L')],
    'D':[(120,'C'),(75,'M')], 'S':[(140,'A'),(99,'F'),(151,'O'),(80,'R')], 
    'R':[(146,'C'),(97,'P'),(80,'S')], 'C':[(120,'D'),(138,'P'),(146,'R')], 
    'F':[(211,'B'),(99,'S')], 'P':[(101,'B'),(138,'C'),(97,'R')], 'B':[]
}
romaniaH = {
    'A':366, 'B':0, 'C':160, 'D':242, 'E':161, 'F':176, 'G':77, 'H':151, 'I':226,
    'L':244, 'M':241, 'N':234, 'O':380, 'P':100, 'R':193, 'S':253, 'T':329, 'U':80,
    'V':199, 'Z':374}
print('Solution path:',greedyTsa(romania, romaniaH, 'A', 'B'))

```

# Greedy-TSA Practice

- Consider the following state space graph
  - Let S be the start state and G be the goal state
- Perform **Greedy-TSA** and write down the order of explored states

```
Initial frontier: [(7, 'S')]
Exploring: S at cost 7
[(0, 'SG'), (9, 'Sa'), (5, 'Sd')]
Solution path: (0, 'SG')
```



Try it [here](#)  
[greedyTsaPractice.py](#)  
[greedyGsaPractice.py](#)

```
from heapq import heappush, heappop
def greedyTsa(stateSpaceGraph, h, startState, goalState):
    frontier = []
    heappush(frontier, (h[startState], startState))
    print('Initial frontier:',list(frontier)); input()
    while frontier:
        node = heappop(frontier)
        if (node[1].endswith(goalState)): return node
        print('Exploring:',node[1][-1],'at cost',node[0])
        for child in stateSpaceGraph[node[1][-1]]:
            heappush(frontier, (h[child], node[1]+child))
        print(list(frontier)); input()
practice = {
    'S':[(3,'a'),(2,'d'),(10,'G')], 'a':[(5,'b')],
    'd':[(1,'b'),(4,'e')], 'G':[], 'b':[(1,'e'),(2,'c')],
    'e':[(3,'G')], 'c':[(4,'G')]}

practiceH = {'S':7, 'a':9, 'b':4, 'c':2, 'd':5, 'e':3, 'G':0}
print('Solution path:',greedyTsa(practice, practiceH, 'S', 'G'))
```

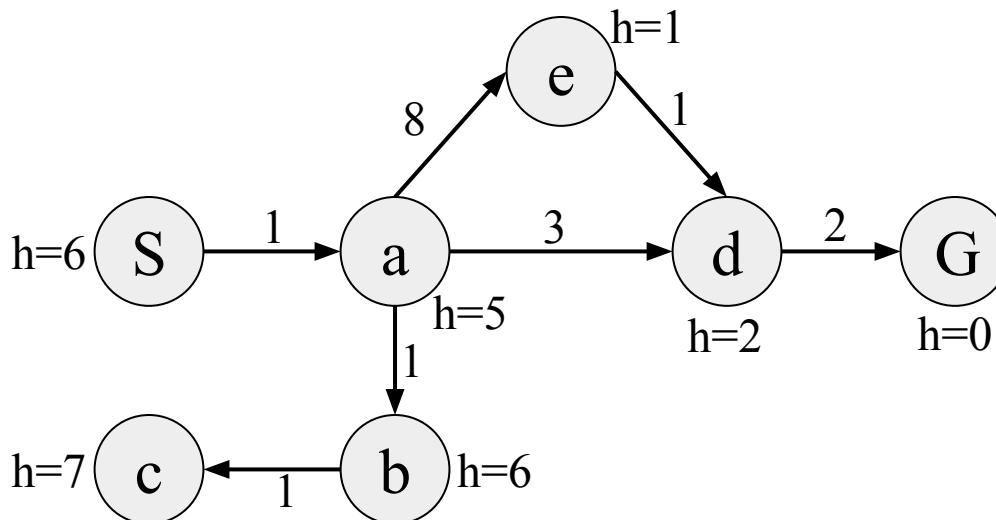
## A\*



frontier is a priority queue (queue data structure where each element has a priority)

# A\* Motivation UCS-TSA

- UCS orders by **backward cost**
  - $g(n)$



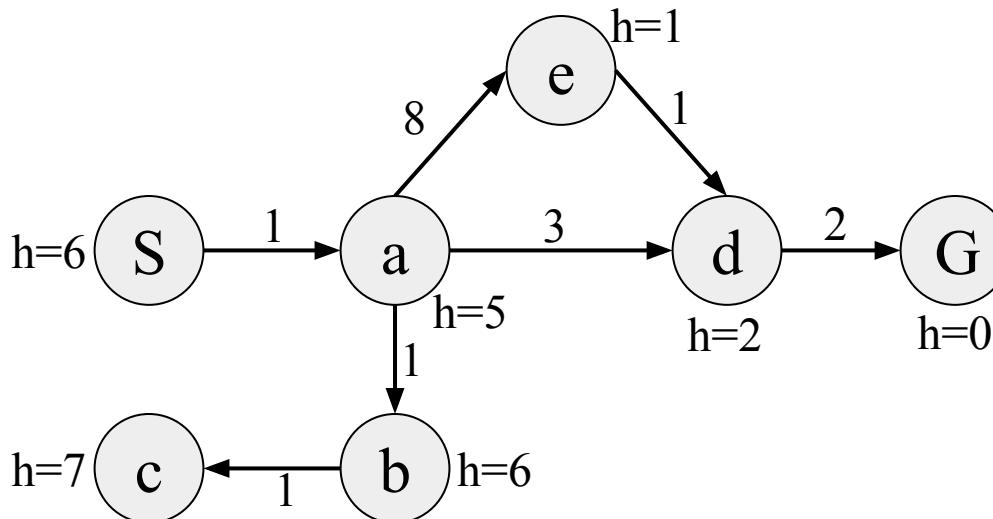
```

Initial frontier: [(0, 'S')]
Exploring: S at cost 0
[(1, 'Sa')]
Exploring: a at cost 1
[(2, 'Sab'), (4, 'Sad'), (9, 'Sae')]
Exploring: b at cost 2
[(3, 'Sabc'), (9, 'Sae'), (4, 'Sad')]
Exploring: c at cost 3
[(4, 'Sad'), (9, 'Sae')]
Exploring: d at cost 4
[(6, 'SadG'), (9, 'Sae')]
Solution path: (6, 'SadG')

```

# A\* Motivation Greedy-TSA

- Greedy orders by **forward cost**
  - $h(n)$



```

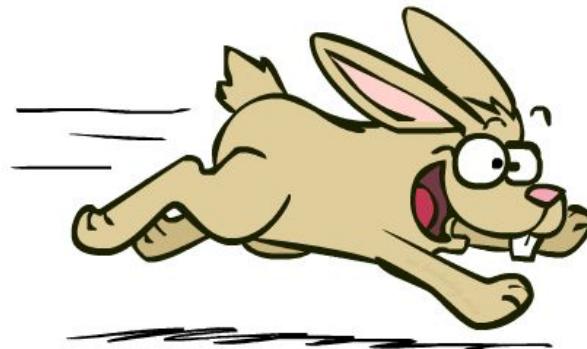
Initial frontier: [(6, 'S')]
Exploring: S at cost 6
[(5, 'Sa')]
Exploring: a at cost 5
[(1, 'Sae'), (6, 'Sab'), (2, 'Sad')]
Exploring: e at cost 1
[(2, 'Sad'), (6, 'Sab'), (2, 'Saed')]
Exploring: d at cost 2
[(0, 'SadG'), (6, 'Sab'), (2, 'Saed')]
Solution path: (0, 'SadG')
  
```

# The Tortoise and the Hare



<http://ai.berkeley.edu>

UCS



<http://ai.berkeley.edu>

Greedy

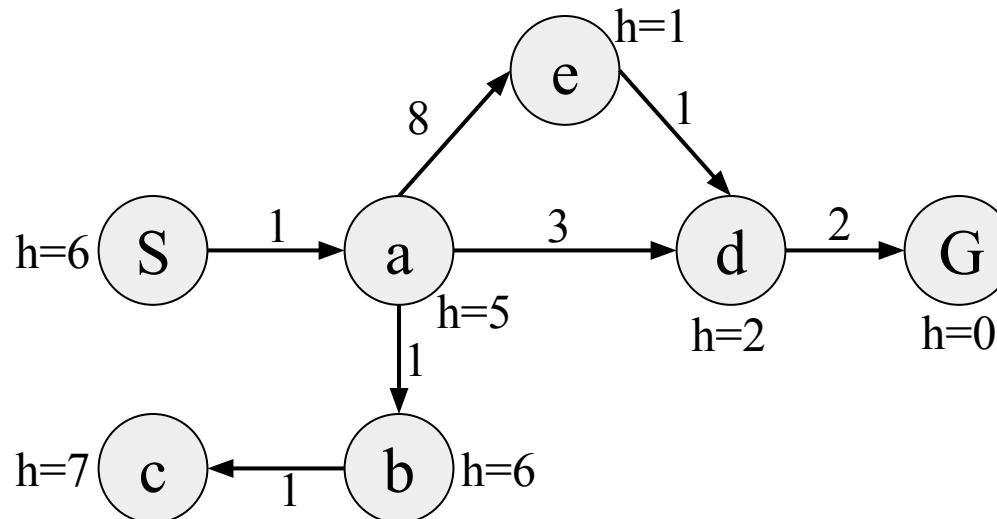


<http://ai.berkeley.edu>

A\*

# A\* Motivation A\*-TSA

- A\* orders by **backward cost + forward cost**
  - $f(n) = g(n) + h(n)$



```

Exploring: S at cost 6
[(6, 'Sa')]
Exploring: a at cost 6
[(6, 'Sad'), (8, 'Sab'), (10, 'Sae')]
Exploring: d at cost 6
[(6, 'SadG'), (10, 'Sae'), (8, 'Sab')]
Solution path: (6, 'SadG')
  
```

Try it [here](#)  
[aStarTsaAStarMotivation.py](#)

```

from heapq import heappush, heappop
def aStarTsa(stateSpaceGraph, h, startState, goalState):
    frontier = []
    heappush(frontier, (h[startState], startState))
    print('Initial frontier:',list(frontier)); input()
    while frontier:
        node = heappop(frontier)
        if (node[1].endswith(goalState)): return node
        print('Exploring:',node[1][-1],'at cost',node[0])
        for child in stateSpaceGraph[node[1][-1]]:
            heappush(frontier, (node[0]+child[0]-h[node[1][-1]]+h[child[1]], node[1]+child[1]))
        print(list(frontier)); input()
aStarMotivation = {
'S':[(1,'a')], 'a':[(1,'b'),(3,'d'),(8,'e')], 'b':[(1,'c')], 'c':[], 'd':[(2,'G')], 'e':[(1,'d')]}
aStarMotivationH = { 'S':6, 'a':5, 'b':6, 'c':7, 'd':2, 'e':1, 'G':0}
print('Solution path:',aStarTsa(aStarMotivation, aStarMotivationH, 'S', 'G'))

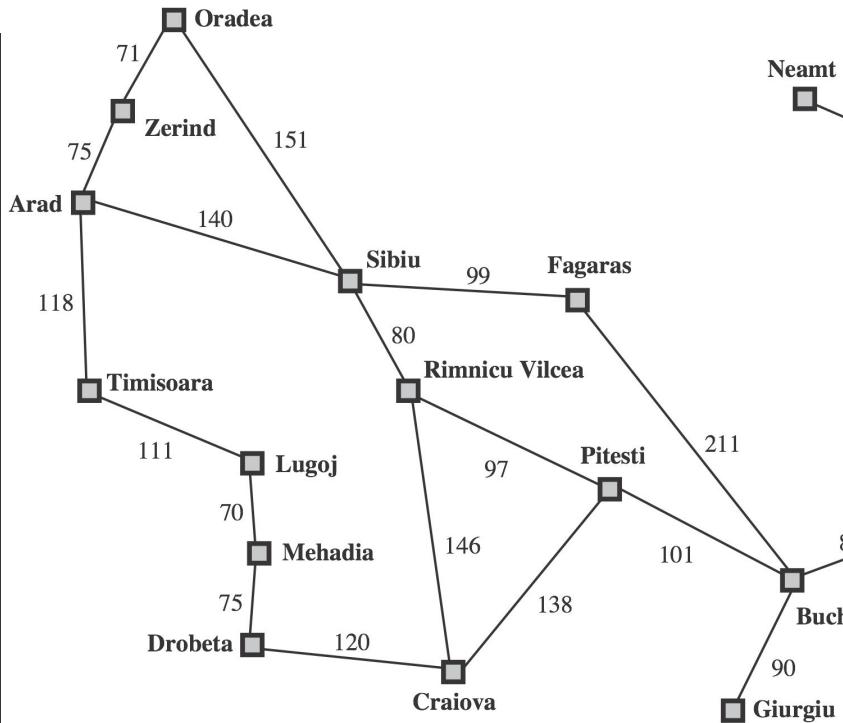
```

```

Initial frontier: [(366, 'A')]
Exploring: A at cost 366
[(393, 'AS'), (447, 'AT'), (449, 'AZ')]
Exploring: S at cost 393
[(413, 'ASR'), (447, 'AT'), (415, 'ASF'),
(449, 'AZ'), (671, 'ASO'), (646, 'ASA')]
Exploring: R at cost 413
[(415, 'ASF'), (447, 'AT'), (417, 'ASRP'),
(449, 'AZ'), (671, 'ASO'), (646, 'ASA'),
(526, 'ASRC'), (553, 'ASRS')]
Exploring: F at cost 415
[(417, 'ASRP'), (447, 'AT'), (526, 'ASRC'),
(449, 'AZ'), (671, 'ASO'), (646, 'ASA'),
(553, 'ASRS'), (450, 'ASFB'), (591, 'ASFS')]
Exploring: P at cost 417
[(418, 'ASRPB'), (447, 'AT'), (526, 'ASRC'),
(449, 'AZ'), (607, 'ASRPR'), (646, 'ASA'),
(553, 'ASRS'), (591, 'ASFS'), (450, 'ASFB'),
(671, 'ASO'), (615, 'ASRPC')]

Solution path: (418, 'ASRPB')

```



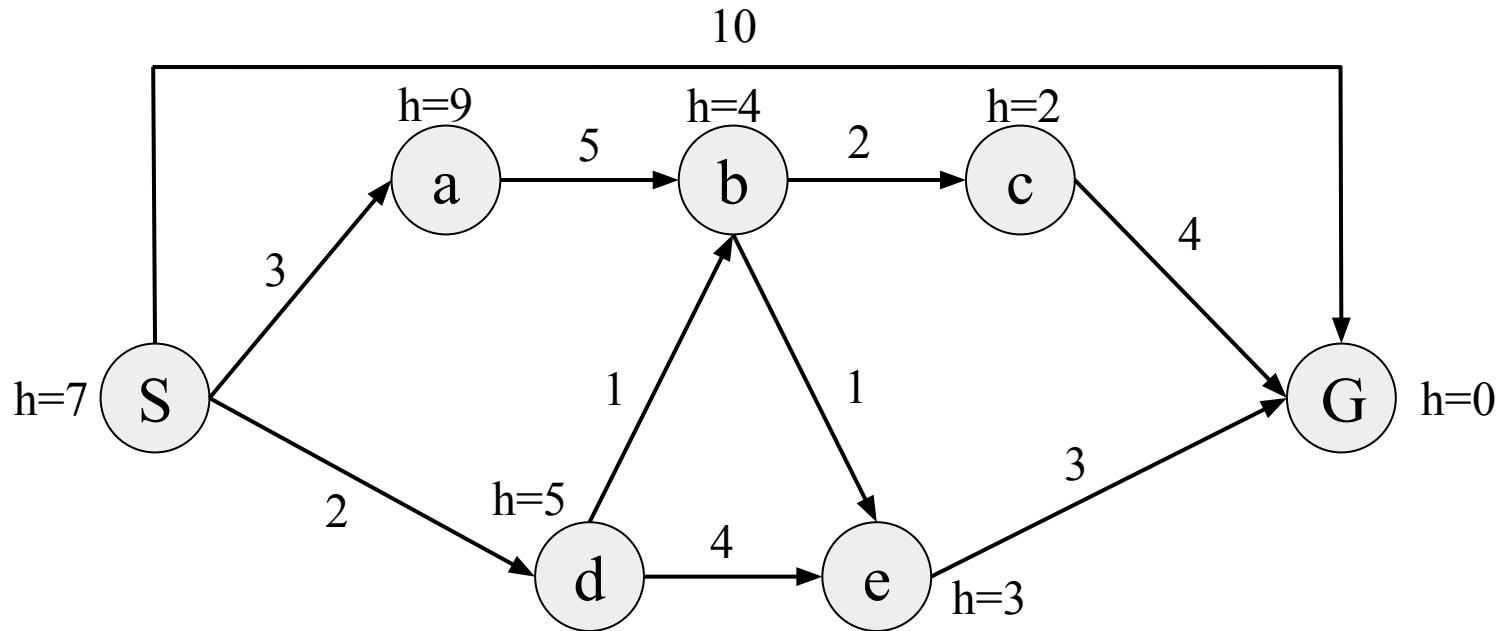
Straight-line  
distance to B

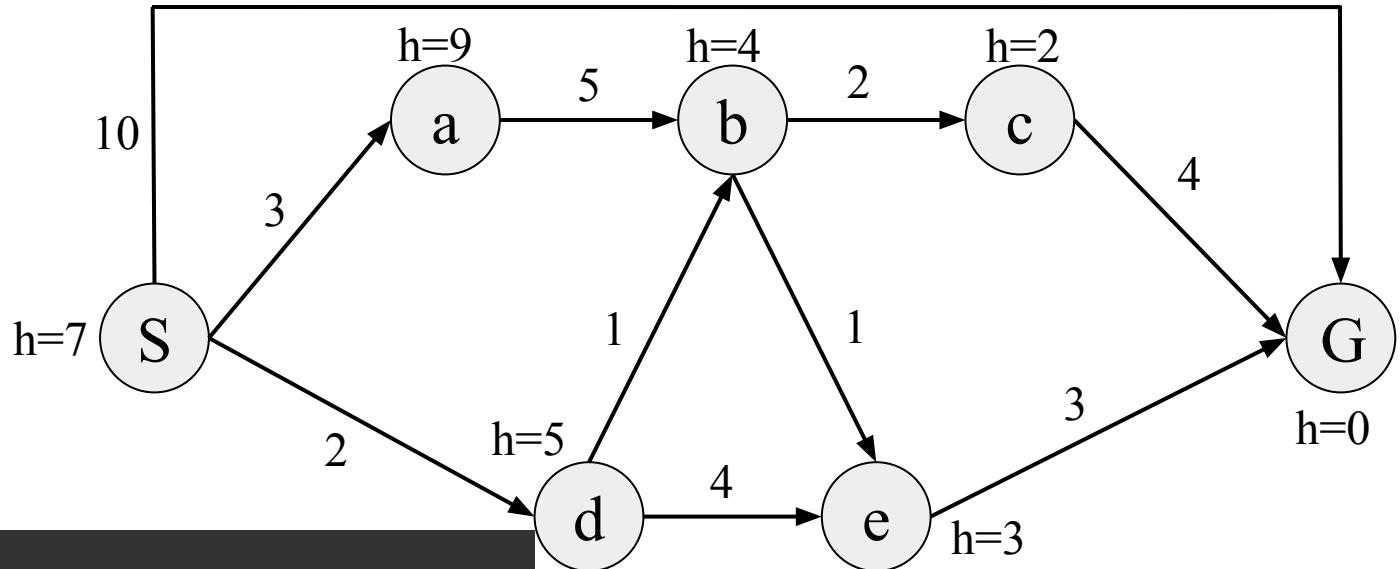
n	h(n)
A	366
B	0
C	160
D	242
E	161
F	176
G	77
H	151
I	226
L	244
M	241
N	234
O	380
P	100
R	193
S	253
T	329
U	80
V	199
Z	374

Try it [here](#)  
[aStarTsaRomania.py](#)  
[aStarGsaRomania.py](#)

# A\*-TSA Practice

- Consider the following state space graph
  - Let S be the start state and G be the goal state
- Perform **A\*-TSA** and write down the order of explored states





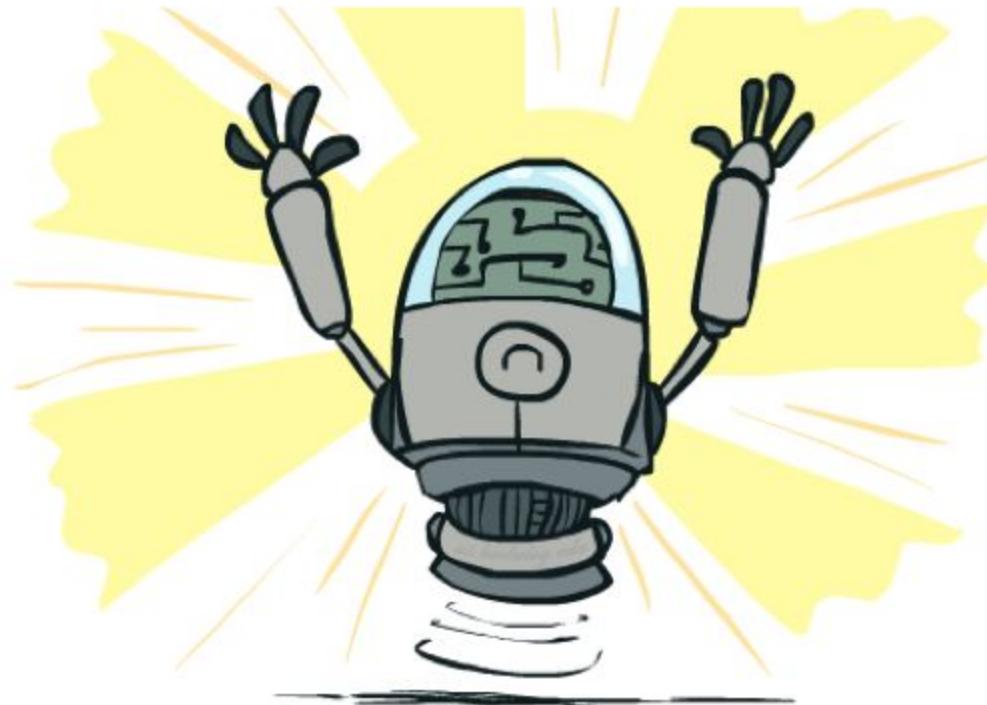
```

Initial frontier: [(7, 'S')]
Exploring: S at cost 7
[(7, 'Sd'), (12, 'Sa'), (10, 'SG')]
Exploring: d at cost 7
[(7, 'Sdb'), (9, 'Sde'), (10, 'SG'), (12, 'Sa')]
Exploring: b at cost 7
[(7, 'Sdbc'), (7, 'Sdbe'), (10, 'SG'), (12, 'Sa'), (9, 'Sde')]
Exploring: c at cost 7
[(7, 'Sdbe'), (9, 'SdbcG'), (10, 'SG'), (12, 'Sa'), (9, 'Sde')]
Exploring: e at cost 7
[(7, 'SdbeG'), (9, 'SdbcG'), (10, 'SG'), (12, 'Sa'),
(9, 'Sde')]
Solution path: (7, 'SdbeG')

```

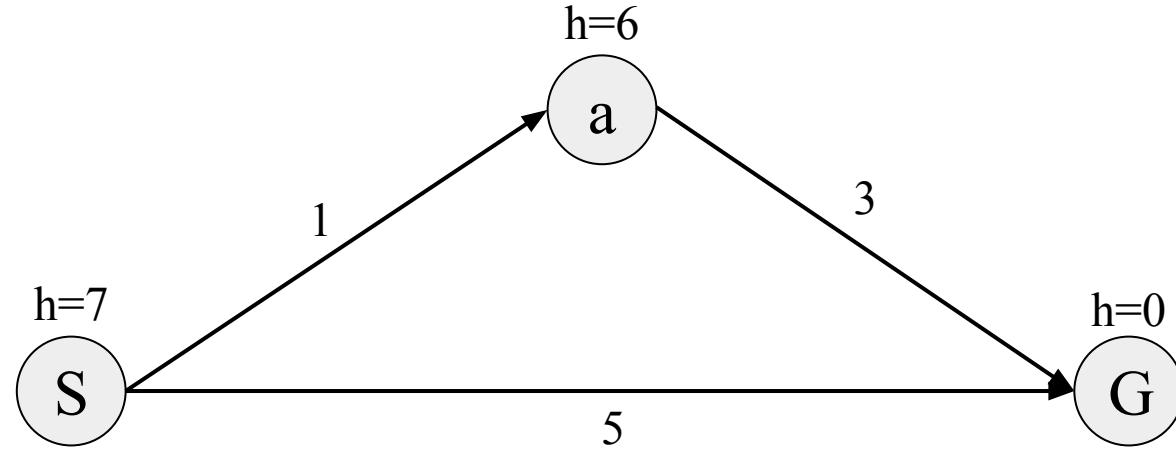
Try it [here](#)  
[aStarTsaPractice.py](#)  
[aStarGsaPractice.py](#)

# Optimality of A\*



<http://ai.berkeley.edu>

# Is A\* Optimal ?



```
Initial frontier: [(7, 'S')]  
Exploring: S at cost 7  
[(5, 'SG'), (7, 'Sa')]  
Solution path: (5, 'SG')
```

What went wrong?

Try it [here](#)  
[aStarTsaInadmissible.py](#)  
[aStarGsaInadmissible.py](#)

# Admissibility of Heuristic

A heuristic  $h(n)$  is admissible (optimistic)

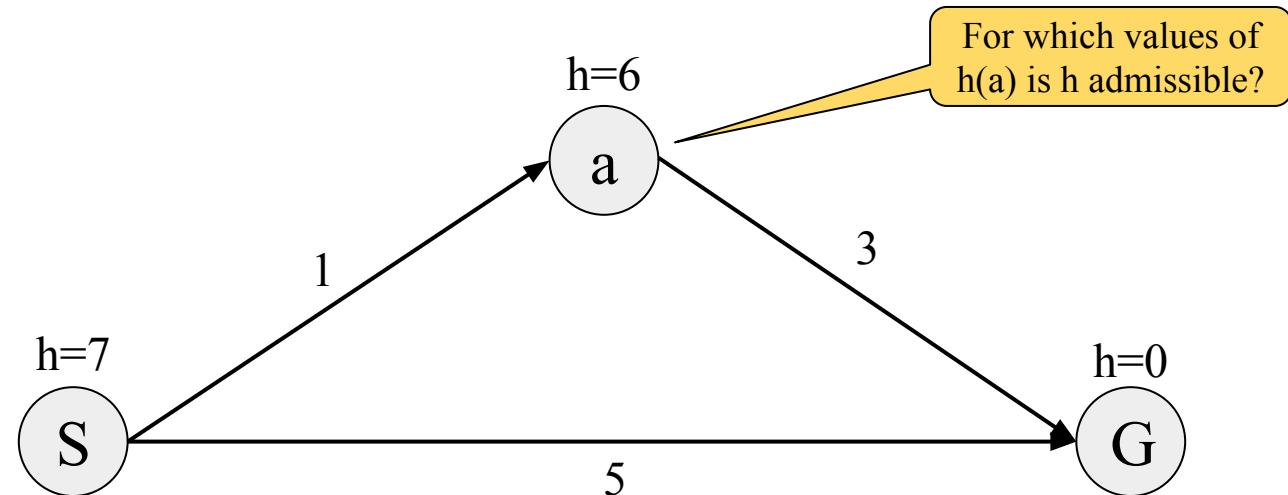
$$0 \leq h(n) \leq h^*(n)$$

where  $h^*(n)$  is the true cost to the nearest goal

# Admissibility of Heuristic

A heuristic  $h(n)$  is admissible (optimistic) if  
 $0 \leq h(n) \leq h^*(n)$ ,

where  $h^*(n)$  is the true cost to the nearest goal from  $n$

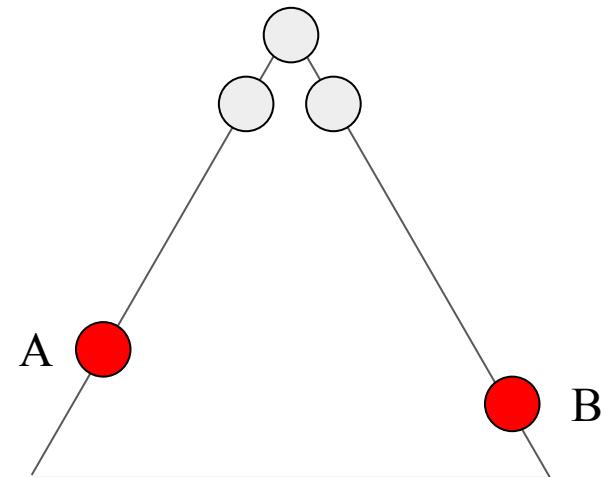


# Optimality of A\*

- Claim
  - A\* is optimal if an admissible heuristic is used

# Optimality of A\*

- Let
  - A be an optimal goal node
  - B be a suboptimal goal node
  - $h$  be admissible
- Claim
  - A will exit the frontier before B

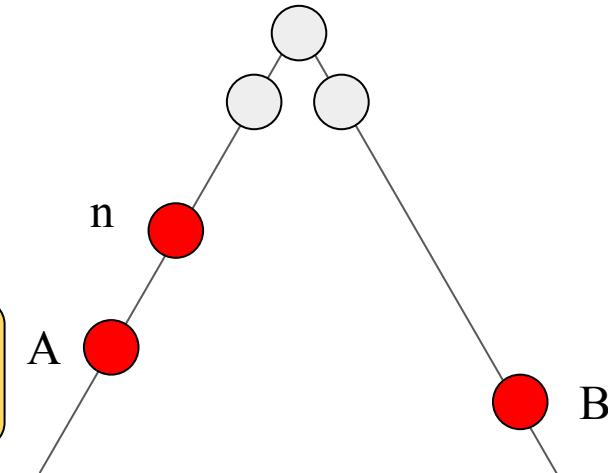


# Optimality of A\*

- Let  $n$  be an ancestor of  $A$
- Let both  $B$  and  $n$  be on the frontier
- Claim
  - $n$  will exit frontier before  $B$
- Proof
  - $f(n) \leq f(A)$
  - $f(A) < f(B)$
  - $n$  exits frontier before  $B$

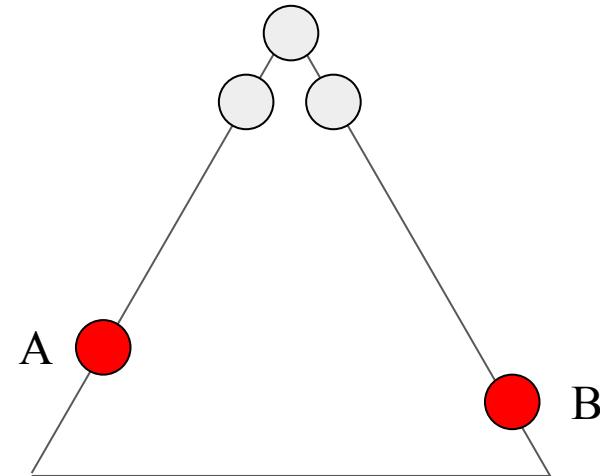
Follows from definition of  
admissibility and the fact that  $n$  is  
an ancestor of  $A$

$B$  is suboptimal

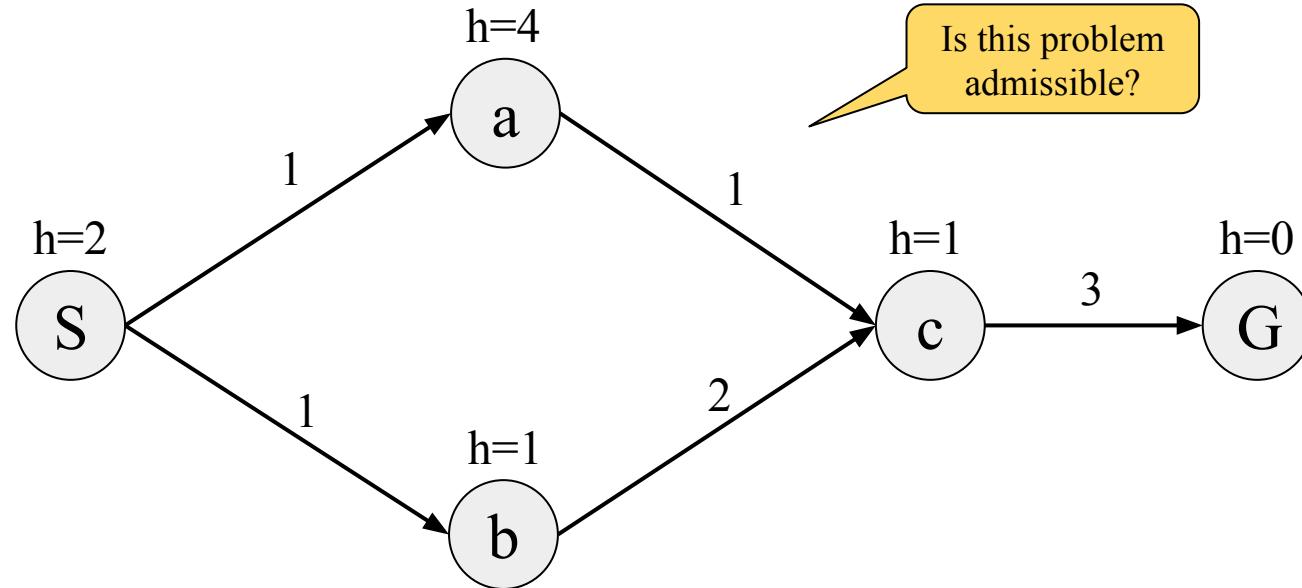


# Optimality of A\*

- All ancestors of A will exit frontier before B
- A will exit frontier before B
- A\* TSA is optimal



# Optimality of A\*



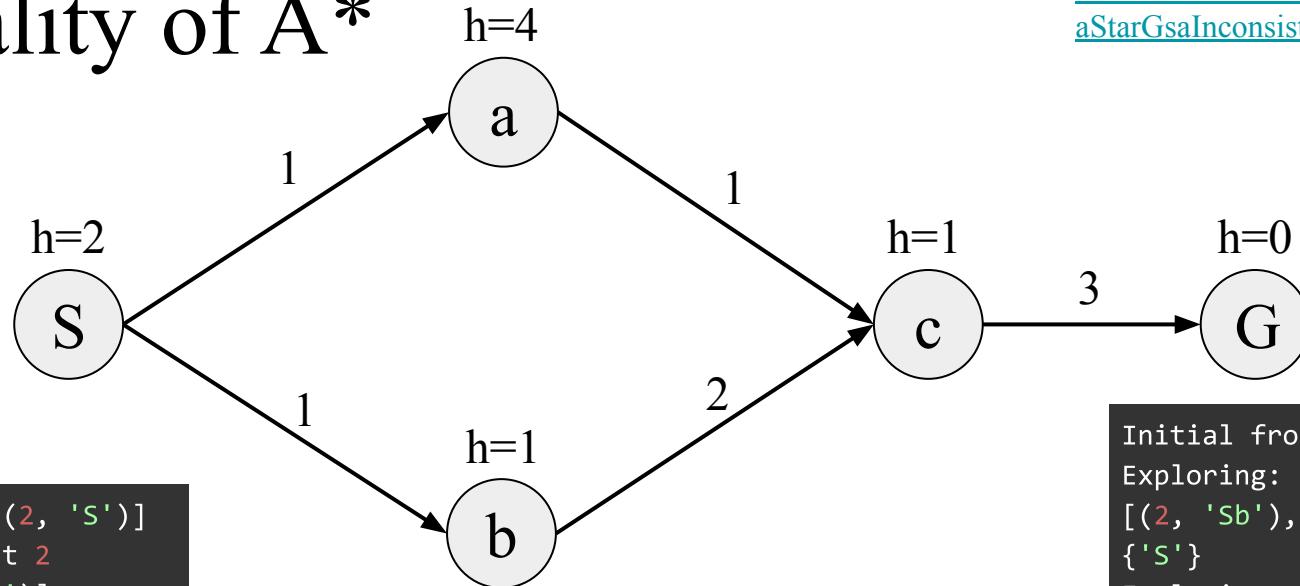
A heuristic  $h(n)$  is admissible (optimistic) if

$$0 \leq h(n) \leq h^*(n),$$

where  $h^*(n)$  is the true cost to the nearest goal from  $n$

Try it [here](#)  
[aStarTsaInconsistent.py](#)  
[aStarGsaInconsistent.py](#)

# Optimality of A\*



```

Initial frontier: [(2, 'S')]
Exploring: S at cost 2
[(2, 'Sb'), (5, 'Sa')]
Exploring: b at cost 2
[(4, 'Sbc'), (5, 'Sa')]
Exploring: c at cost 4
[(5, 'Sa'), (6, 'SbcG')]
Exploring: a at cost 5
[(3, 'Sac'), (6, 'SbcG')]
Exploring: c at cost 3
[(5, 'SacG'), (6, 'SbcG')]
Solution path: (5, 'SacG')
  
```

Looks good, this is  
the optimal  
solution

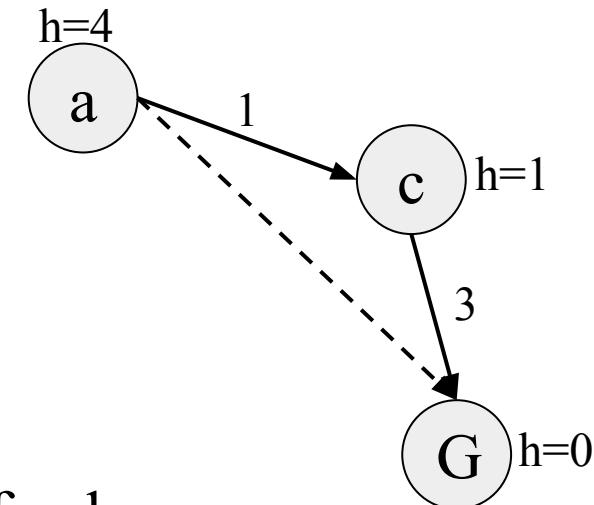
What went  
wrong ?

```

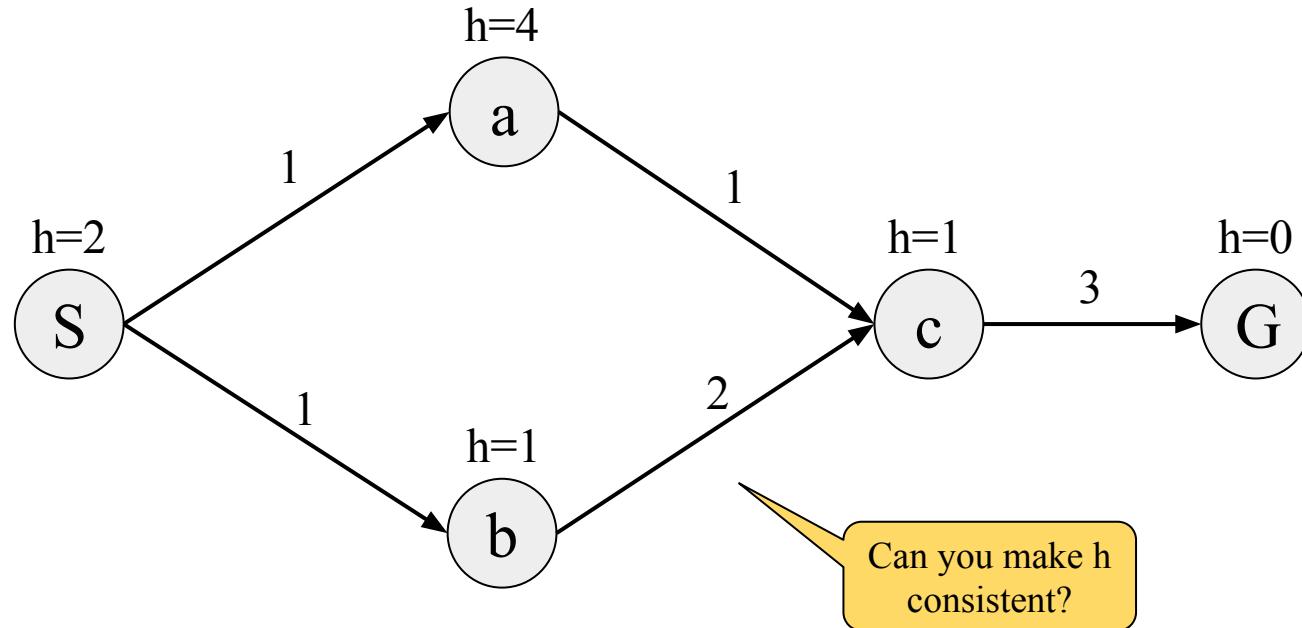
Initial frontier: [(2, 'S')]
Exploring: S at cost 2
[(2, 'Sb'), (5, 'Sa')]
{'S'}
Exploring: b at cost 2
[(4, 'Sbc'), (5, 'Sa')]
{'S', 'b'}
Exploring: c at cost 4
[(5, 'Sa'), (6, 'SbcG')]
{'S', 'c', 'b'}
Exploring: a at cost 5
[(3, 'Sac'), (6, 'SbcG')]
{'S', 'c', 'a', 'b'}
Solution path: (6, 'SbcG')
  
```

# Consistency of Heuristic

- Definition
  - Heuristic cost  $\leq$  actual cost for each arc
    - $h(a) - h(c) \leq \text{cost}(a \text{ to } c)$
- Consequence of consistency
  - The f value along a path never decreases
    - $h(a) \leq \text{cost}(a \text{ to } c) + h(c)$
- Can you prove that A\* GSA is optimal if the f value never decreases?



# Consistency of Heuristic



# Local Search

# Planning vs. Identification

- Planning: sequences of actions
  - The path to the goal is the important thing
  - Paths have various costs, depths
  - Heuristics to guide, frontier to keep backups
- Identification: assignments to variables
  - The goal itself is important, not the path
- Local Search can find solutions faster for specific types of identification problems

# Local Search

- Evaluate and modify one current state rather than systematically explore paths from an initial state
- Suitable for problems where all that matters is the solution state and not the path cost to reach it
- Although local search algorithms are not systematic they have two advantages
  - Require very little memory
  - Often find reasonable solutions in large spaces

# Example: 8-Queens

- States
  - Each state has 8 queens on the board, one per column
- Successors
  - All possible states generated by moving a single queen to another square in the same column
- Cost function
  - Number of pairs of queens that are attacking each other, either directly or indirectly

# Example: 8-Queens

- What is the cost of the following state?

Q <sub>1</sub>			Q <sub>4</sub>				
				Q <sub>5</sub>			
Q <sub>2</sub>					Q <sub>6</sub>		Q <sub>8</sub>
		Q <sub>3</sub>				Q <sub>7</sub>	

- Answer:
- Consider unique attacks
  - Q<sub>1</sub> is attacking Q<sub>2</sub>, Q<sub>3</sub>, Q<sub>5</sub>
  - Q<sub>2</sub> is attacking Q<sub>3</sub>, Q<sub>4</sub>, Q<sub>6</sub>, Q<sub>8</sub>
  - Q<sub>3</sub> is attacking Q<sub>5</sub>, Q<sub>7</sub>
  - Q<sub>4</sub> is attacking Q<sub>5</sub>, Q<sub>6</sub>, Q<sub>7</sub>
  - Q<sub>5</sub> is attacking Q<sub>6</sub>, Q<sub>7</sub>
  - Q<sub>6</sub> is attacking Q<sub>7</sub>, Q<sub>8</sub>
  - Q<sub>7</sub> is attacking Q<sub>8</sub>
- The cost of the state is 17

# Example: 8-Queens

- How many neighbors does the following state have?

$Q_1$				$Q_5$			
	$Q_2$				$Q_6$		$Q_8$
		$Q_3$				$Q_7$	

- Answer:
- Every queen can move to 7 locations
  - There are  $7*8 = 56$  neighbors

# Example: 8-Queens

- Costs of all neighbors with minimas shown in red

18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14	$Q_4$	13	16	13	16
$Q_1$	14	17	15	$Q_5$	14	16	16
17	$Q_2$	16	18	15	$Q_6$	15	$Q_8$
18	14	$Q_3$	15	15	14	$Q_7$	16
14	14	13	17	12	14	12	18

# Local Search - Algorithm

1. Randomly initialize currentState
2. If cost of currentState == 0 return currentState
3. If  $\min(\text{cost}(\text{getNeighbors}(\text{currentState}))) > \text{cost}(\text{currentState})$   
    goto step 1 (we have reached a local minimum)
4. Select cheapest neighbor as currentState and goto step 2

# Example: 8-Queens

- Starting from a randomly generated 8-queens state, local search gets stuck 86% of the time, solving only 14% of problem instances
- It just takes 4 steps on average when it succeeds and 3 when it gets stuck
  - This is remarkable considering the huge state space

# Constraint Satisfaction Problems

# Constraint Satisfaction Problems

- CSPs use a factored representation for states
  - A set of variables, each of which has a value
- A problem is solved when each variable has a value that satisfies certain constraints
- CSPs can often be solved more efficiently
  - They eliminate large portions of the search space by identifying variable/value combinations that violate constraints

# Defining CSPs

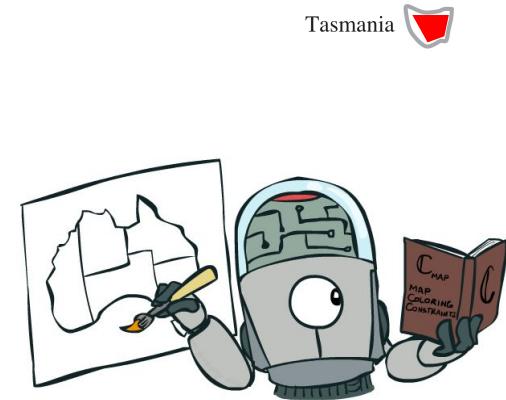
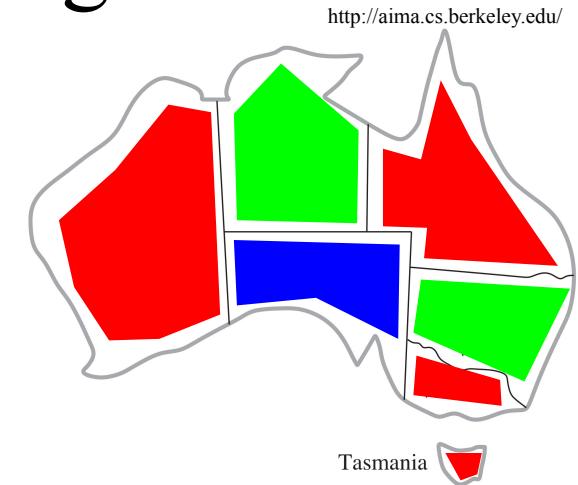
- A CSP consists of three components
  - A set of variables,  $X = \{X_1, \dots, X_n\}$
  - A set of domains,  $D = \{D_1, \dots, D_n\}$ ,  
where  $D_i = \{v_1, \dots, v_k\}$  for each variable  $X_i$
  - A set of constraints  $C$  which specify allowable combinations of values
- To solve a CSP we need to define a state space
  - Each state is defined by an assignment of values to some or all variables  $\{X_i = v_i, X_j = v_j, \dots\}$

# Solutions to CSPs

- If no constraints are violated we call the assignment consistent
- If every variable is assigned we call the assignment complete
- A solution is an assignment that is both consistent and complete

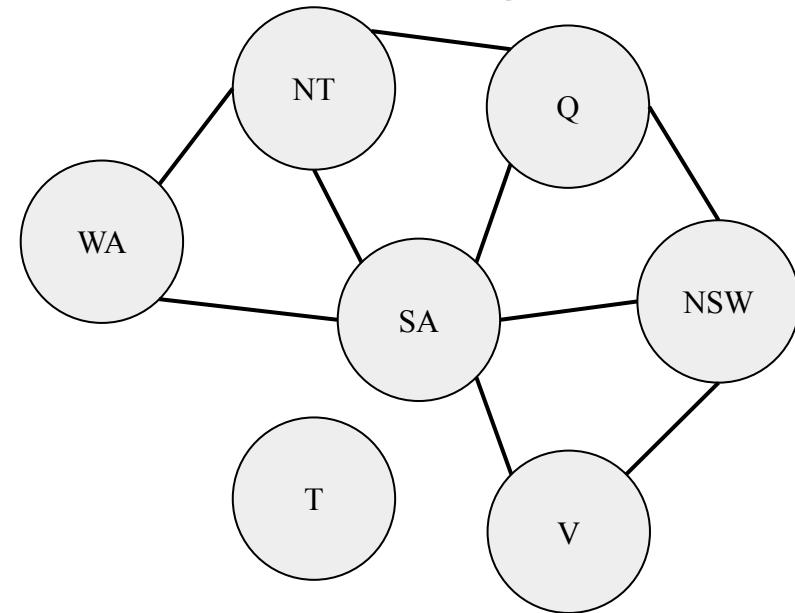
# Example: Australia - Map Coloring

- Variables
  - $X = \{\text{WA, NT, Q, NSW, V, SA, T}\}$
- Domains
  - $D = \{\text{red, green, blue}\}$
- Constraints (different colors for neighbors)
  - Implicit
    - $C = \{\text{SA} \neq \text{WA}, \dots\}$
  - Explicit
    - $C = \{(\text{SA, WA}) \in \{(\text{red, green}), \dots\}\}$



# Constraint Graph

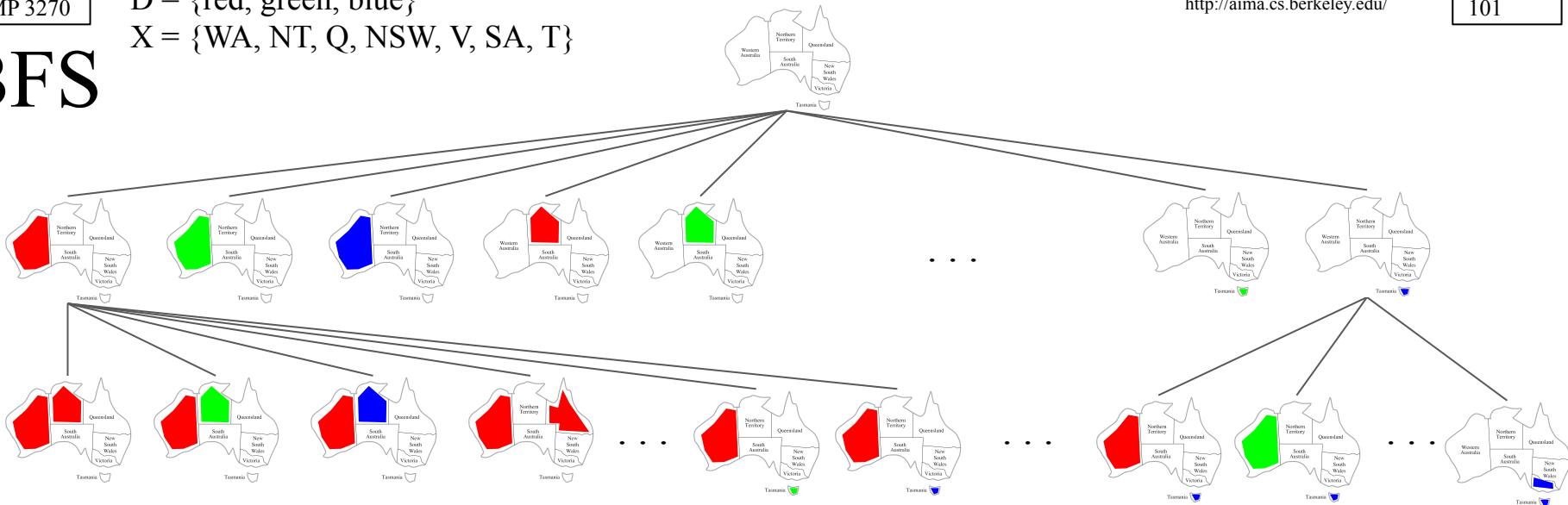
- It can be helpful to visualize a CSP as a constraint graph
  - Nodes correspond to variables
  - Arcs show existence of constraints



# Solving CSPs

- Let's start with a naive approach
  - States are defined by the values assigned so far
  - Initial state
    - Empty assignment { }
  - Successor function
    - Assign a value to an unassigned variable
  - Goal test
    - Current assignment is complete and consistent

## BFS

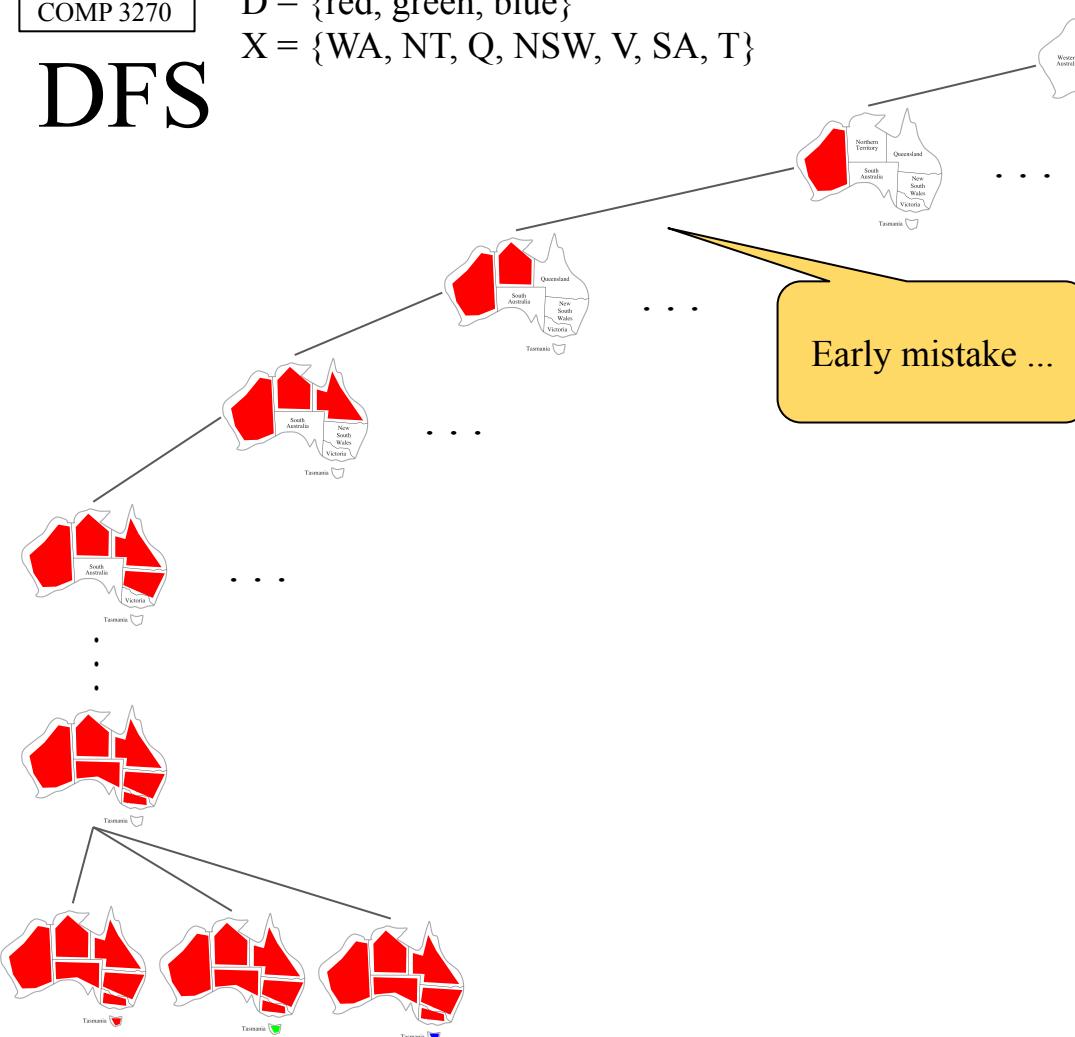


Note that all solutions will be at the bottom of this search tree :-(

This calls  
for DFS  
:-)

D = {red, green, blue}  
X = {WA, NT, Q, NSW, V, SA, T}

## DFS

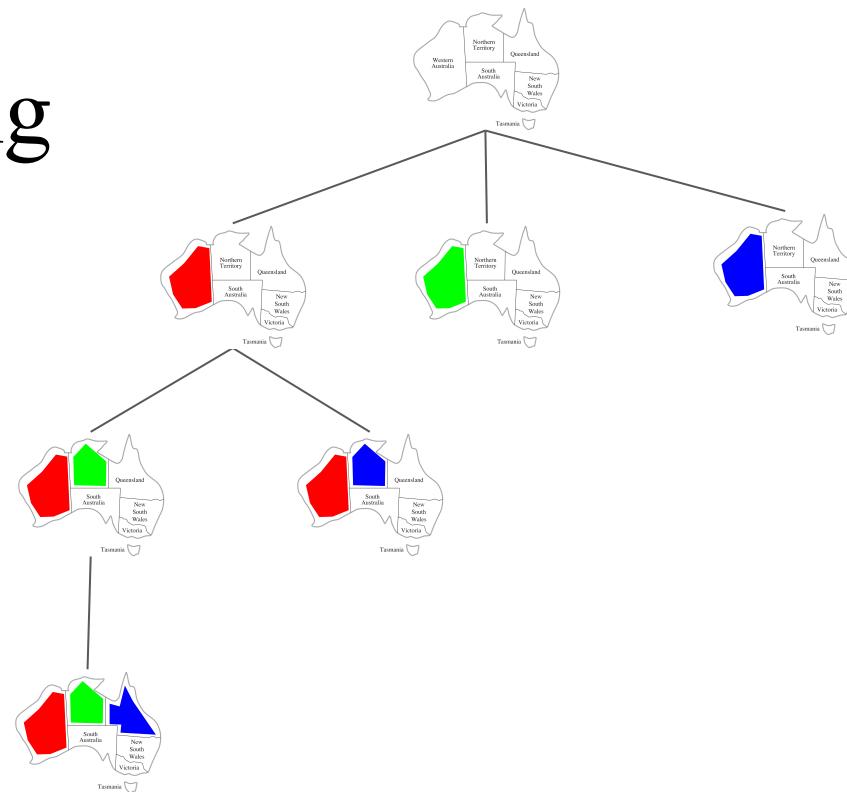


# Backtracking Search

- The basic algorithm for solving CSPs
- Idea
  - a. Only consider assignments to a single variable at each point
    - Note that variable assignments are commutative
      - Therefore, we need only consider a single variable at each node
  - b. Only allow legal assignments at each point
    - Consider only values which do not conflict previous assignments
      - Incremental goal test
- DFS with these two ideas is called backtracking search

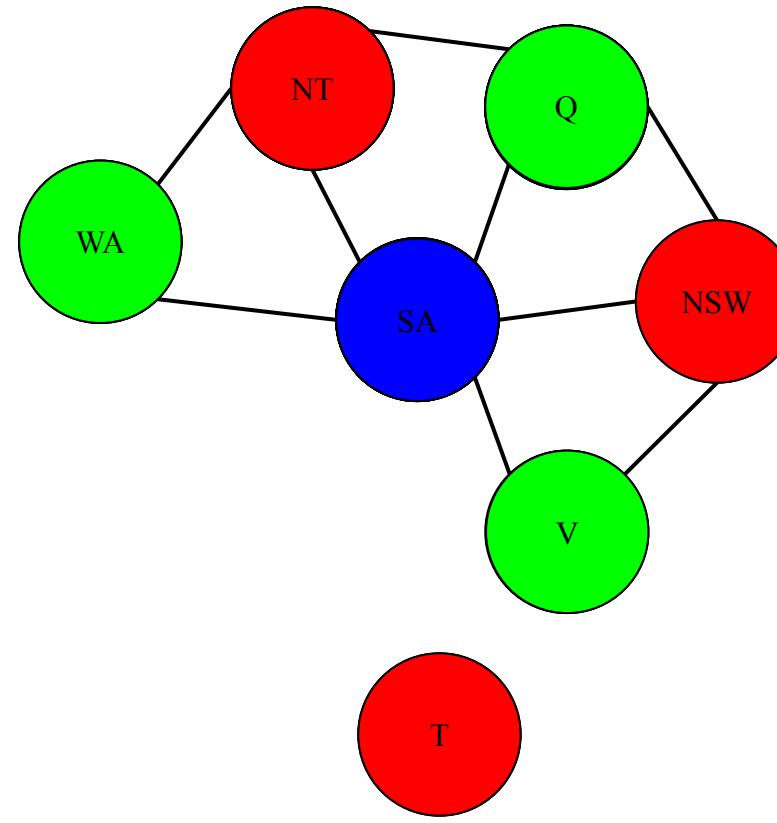


# Backtracking



$D = \{\text{red, green, blue}\}$   
 $X = \{\text{WA, NT, Q, NSW, V, SA, T}\}$

# Example: Backtracking



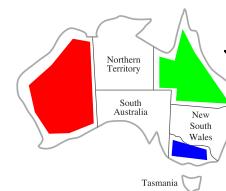
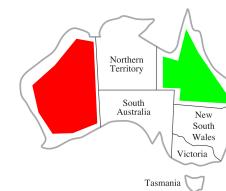
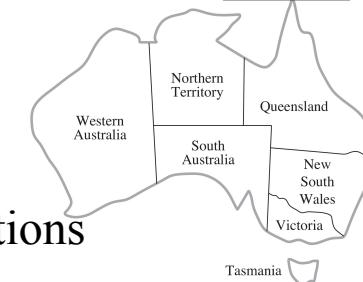
$D = \{\text{red, green, blue}\}$   
 $X = \{\text{NSW, WA, NT, Q, SA, V, T}\}$

# Improving Backtracking

- Idea
  - Can we detect inevitable failure early?
    - Forward checking (FC)
    - Constraint propagation (AC-3)

# Filtering: Forward Checking

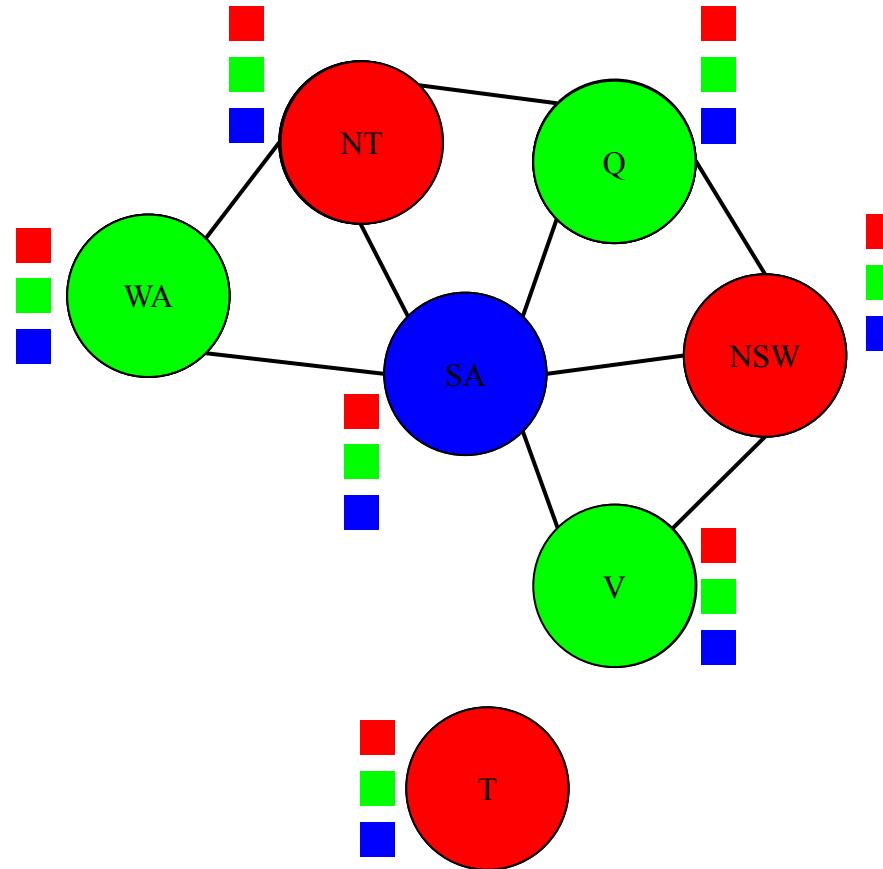
- Filtering
  - Keep track of domains for unassigned variables and cross off bad options
- Forward checking
  - Cross off values that violate a constraint when added to the existing assignment



Detect failure earlier

WA	NT	Q	NSW	V	SA	T
Red Green Blue						
Red	Green Blue	Red Green Blue	Red Green Blue	Red Green Blue	Green Blue	Red Green Blue
Red	Blue	Green	Red Blue	Red Green Blue	Blue	Red Green Blue
Red	Blue	Green	Red	Blue		Red Green Blue

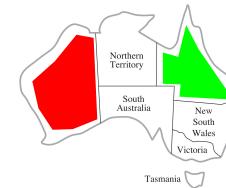
# Example: Backtracking + Forward Checking



$D = \{\text{red, green, blue}\}$   
 $X = \{\text{NSW, WA, NT, Q, SA, V, T}\}$

# Filtering: Forward Checking

- Filtering
  - Keep track of domains for unassigned variables and cross off bad options
- Forward checking
  - Cross off values that violate a constraint when added to the existing assignment



**Doomed to fail**

Can we detect this earlier?

WA	NT	Q	NSW	V	SA	T



# Consistency of an Arc

- An arc  $X \rightarrow Y$  is consistent iff for every  $x$  in the tail there is some  $y$  in the head which could be assigned without violating a constraint

WA	NT	Q	NSW	V	SA	T
Red	Red Green Blue					

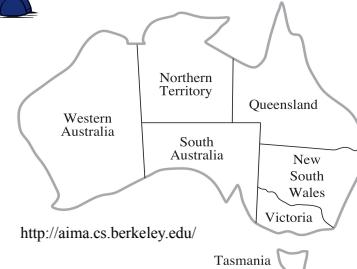
A curved arrow points from the 'WA' column to the 'NT' column.

- Arc consistency explained by Prof. Alan Mackworth

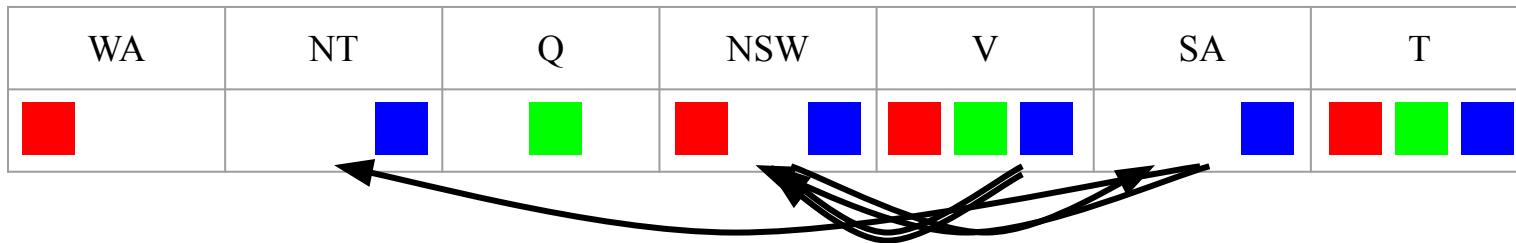
# Constraint propagation



Delete from tail!



- Constraint propagation repeatedly enforces constraints
- Note
  - Arcs can become inconsistent
    - If X loses a value, neighbors of X need to be rechecked
  - Arc consistency detects failure earlier than forward checking
  - Can be run as a preprocessor or after each assignment



An arc  $X \rightarrow Y$  is consistent iff for every  $x$  in the tail there is some  $y$  in the head which could be assigned without violating a constraint

# AC-3 to Enforce Arc Consistency

**function** AC-3(*csp*) **returns** the CSP, possibly with reduced domains

**inputs:** *csp*, a binary CSP with components  $\{X, D, C\}$

**local variables:** *queue*, a queue of arcs, initially all the arcs in *csp*

**while** *queue* is not empty **do**

$(X_i, X_j) \leftarrow \text{REMOVE-FIRST}(\text{queue})$

**if** REVISE(*csp*,  $X_i$ ,  $X_j$ ) **then**

**if** size of  $D_i = 0$  **then return** *false*

**for each**  $X_k$  **in**  $X_i.\text{NEIGHBORS} - \{X_j\}$  **do**

            add  $(X_k, X_i)$  to *queue*

**return** *true*

Why - {  $X_j$  } ?

**function** REVISE(*csp*,  $X_i$ ,  $X_j$ ) **returns** true iff revise domain of  $X_i$

*revised*  $\leftarrow$  *false*

**for each**  $x$  **in**  $D_i$  **do**

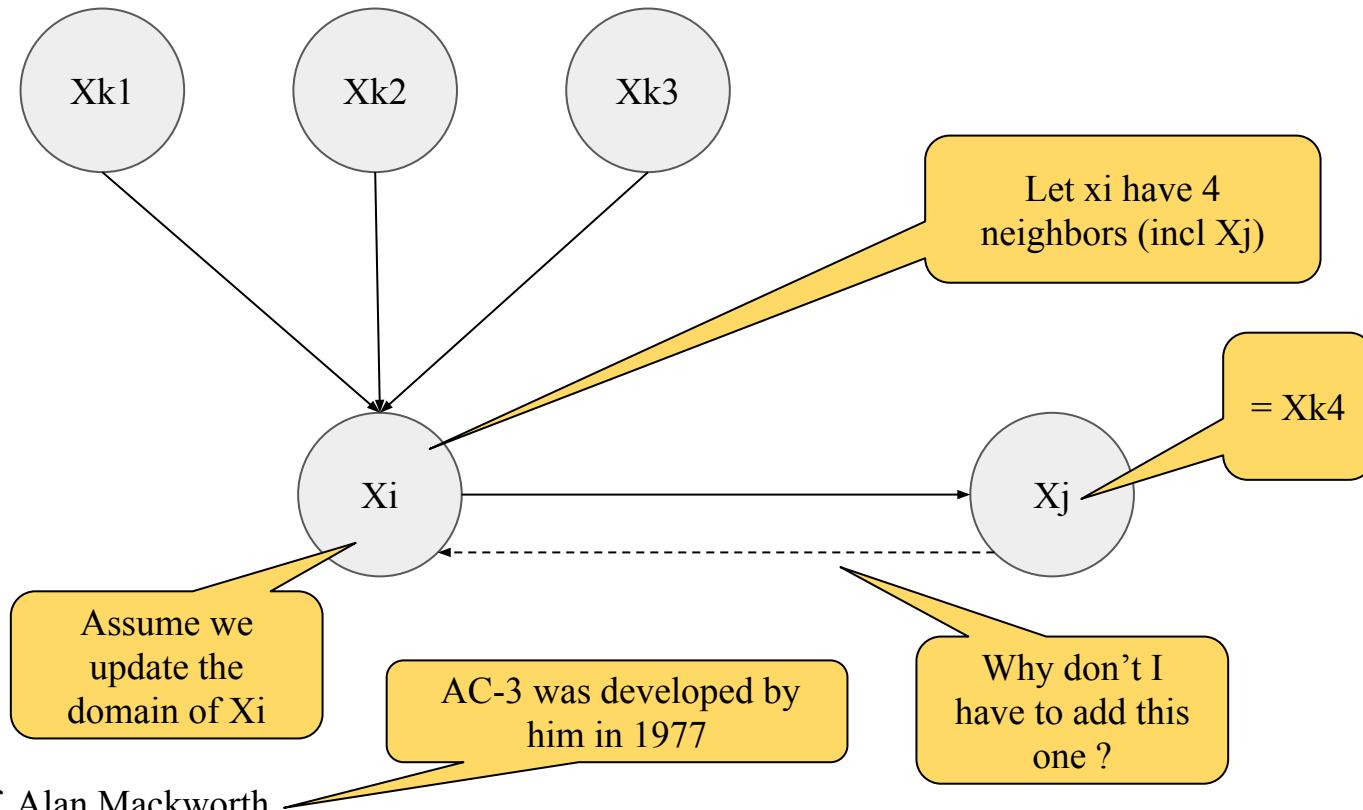
**if** no value  $y$  in  $D_j$  allows  $(x, y)$  to satisfy constraint between  $X_i$  and  $X_j$

            delete  $x$  from  $D_i$

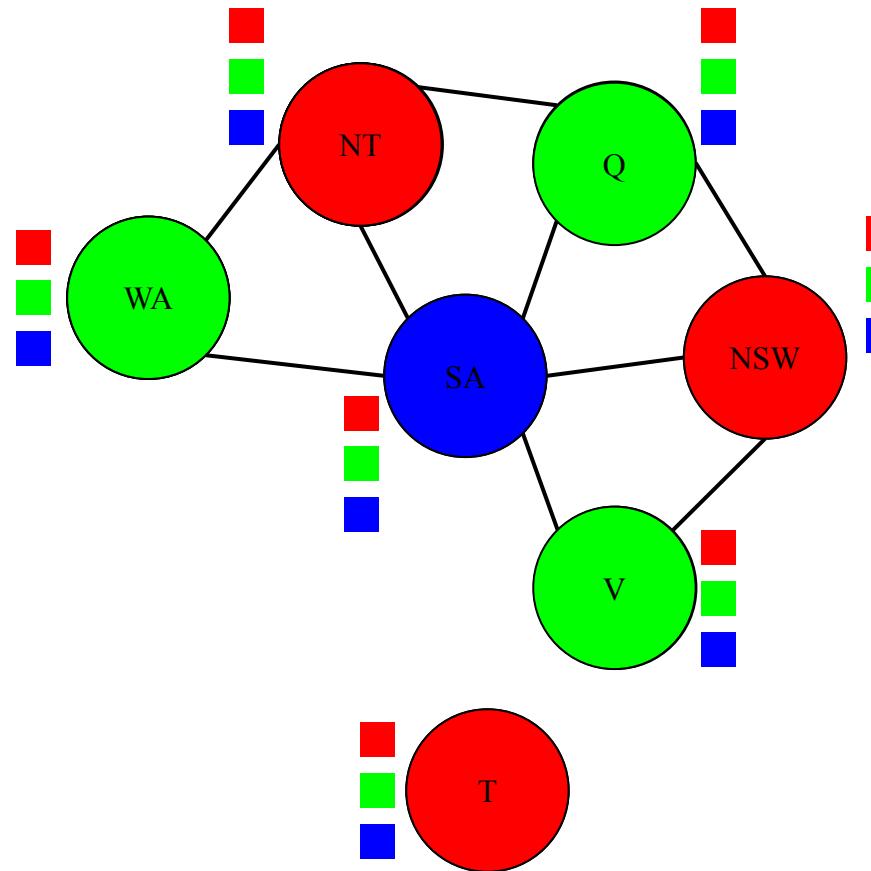
*revised*  $\leftarrow$  *true*

**return** *revised*

# Why - { $X_j$ } ?



# Example: Backtracking + AC-3



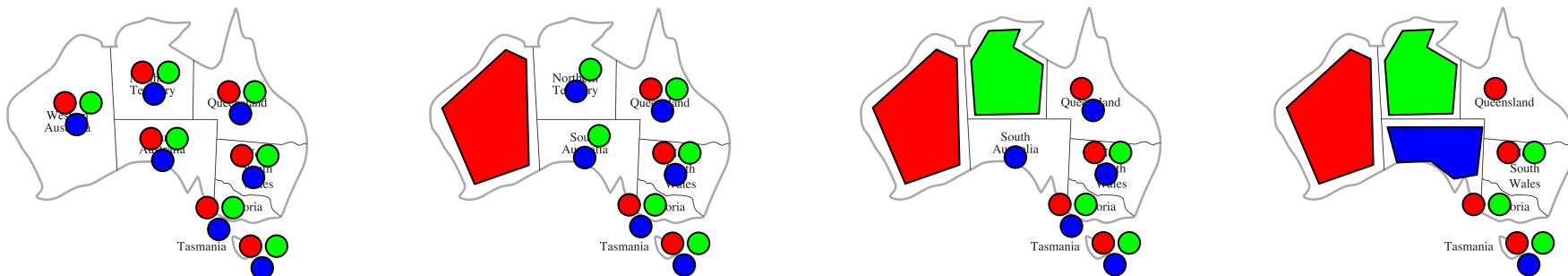
$$\begin{aligned} D &= \{\text{red, green, blue}\} \\ X &= \{\text{NSW, WA, NT, Q, SA, V, T}\} \end{aligned}$$

# Improving Backtracking Further

- Variable Ordering
  - Minimum remaining values (MRV)
    - Choose the variable with the fewest legal left values in its domain
      - Most constrained variable
      - Fail-first heuristic
    - Tie-breaker among MRV variables
      - Degree Heuristic (Deg)
        - Choose the variable with the most constraints on remaining variables
  - Value Ordering
    - Least constraining value (LCV)
      - Choose the value that rules out the fewest values in the remaining variables

# Variable Ordering (MRV)

- Minimum remaining values
  - Choose the variable with the fewest legal left values in its domain
    - Most constrained variable
    - Fail-first heuristic

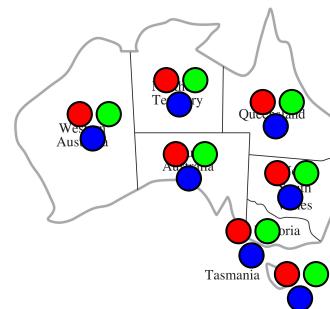


using forward checking

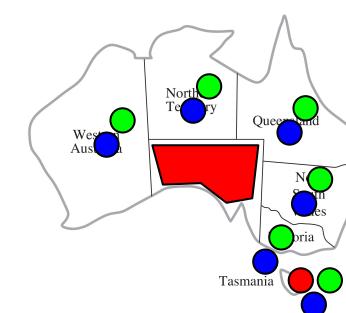
$$\begin{aligned} D &= \{\text{red, green, blue}\} \\ X &= \{\text{WA, NSW, NT, Q, SA, V, T}\} \end{aligned}$$

# Variable Ordering (MRV + Deg)

- Degree Heuristic (Deg)
  - Choose the variable with the most constraints on remaining variables



using forward checking

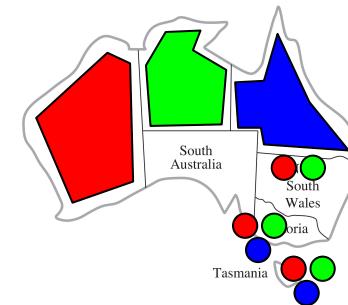
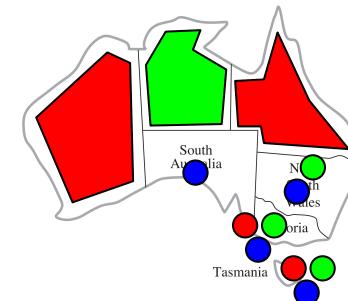
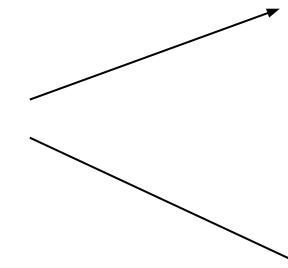
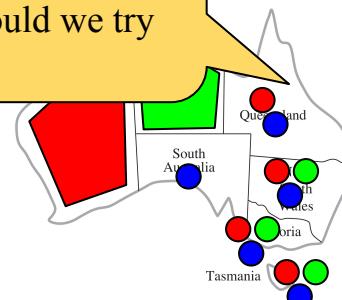


$$\begin{aligned} D &= \{\text{red, green, blue}\} \\ X &= \{\text{NSW, WA, NT, Q, SA, V, T}\} \end{aligned}$$

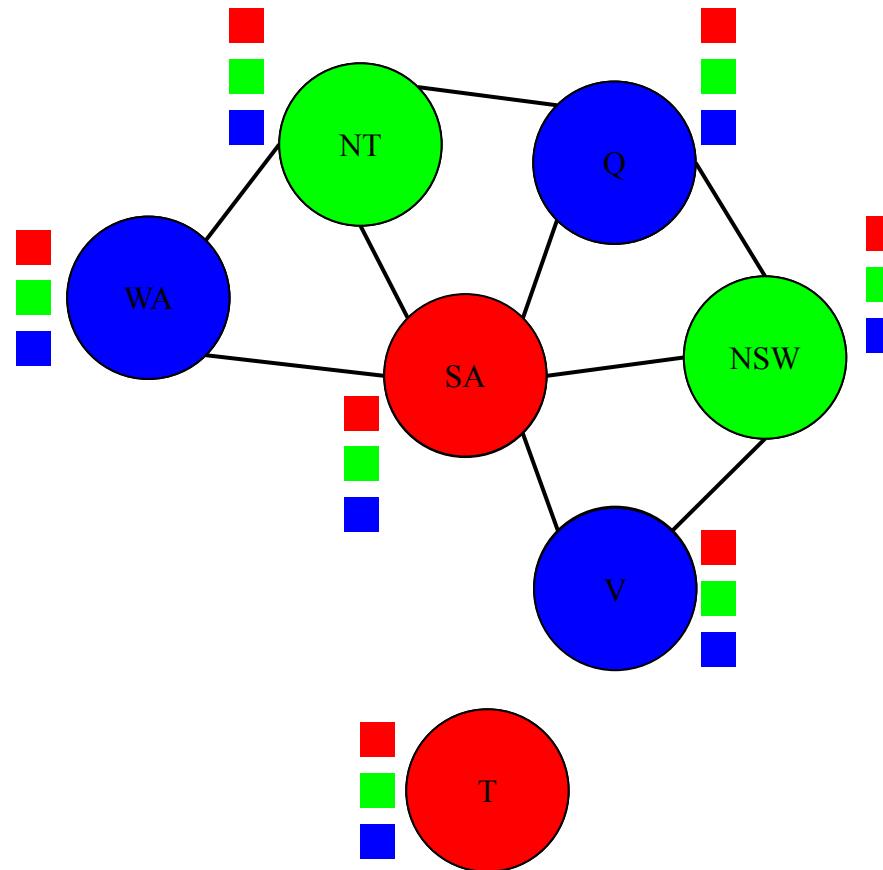
# Value Ordering (LCV)

- Choose the value that rules out the fewest values in the remaining variables
  - Least constraining value (LCV)

Suppose we have decided to assign this variable next.  
What value should we try first?

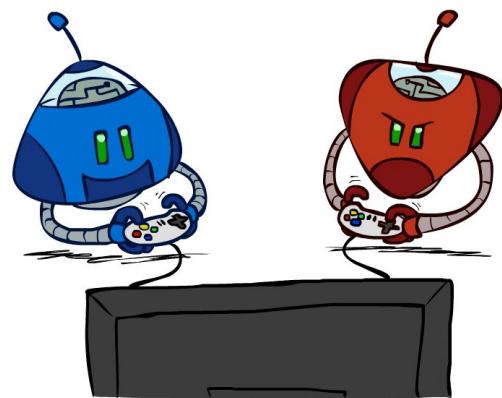


# Example: Backtracking +AC-3 + MRV & Deg + LCV



$D = \{red, green, blue\}$   
 $X = \{NSW, WA, NT, Q, SA, V, T\}$

# Adversarial Search

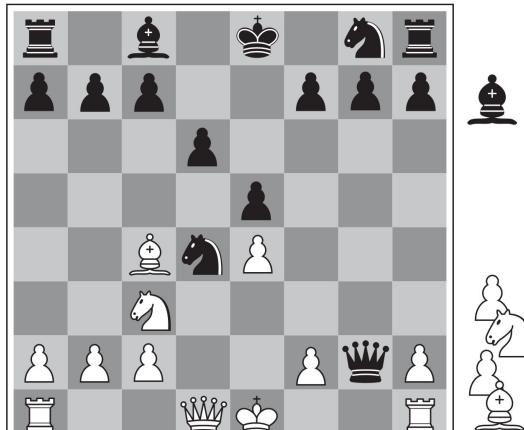


# Adversarial Search

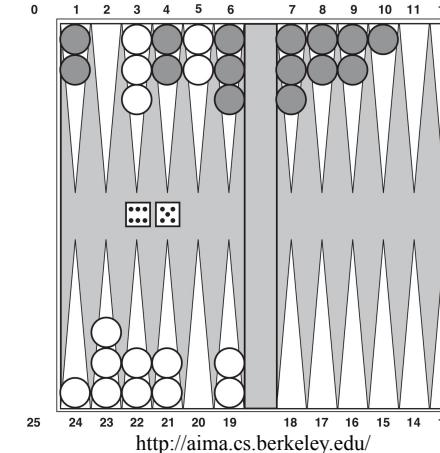
- A multi-agent competitive environment
  - We try to plan ahead in a world where other agents are planning against us
  - Goals are in conflict (not necessarily)



<http://ai.berkeley.edu>



<http://aima.cs.berkeley.edu/>

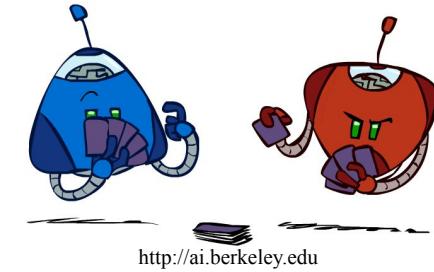


<http://aima.cs.berkeley.edu/>

X	O
X	
O	

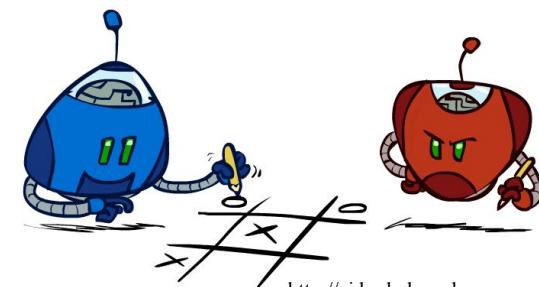
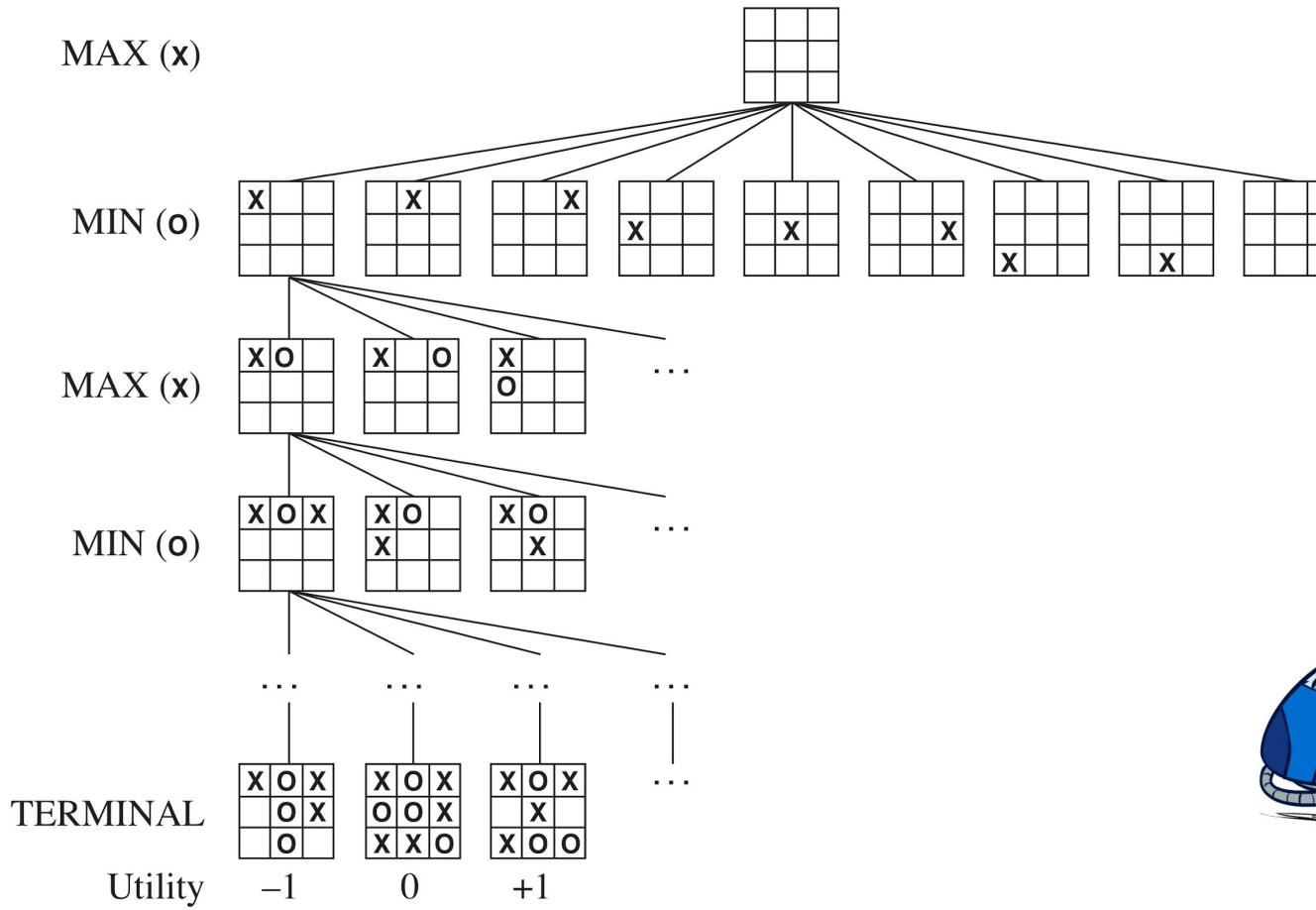
# Game Definition

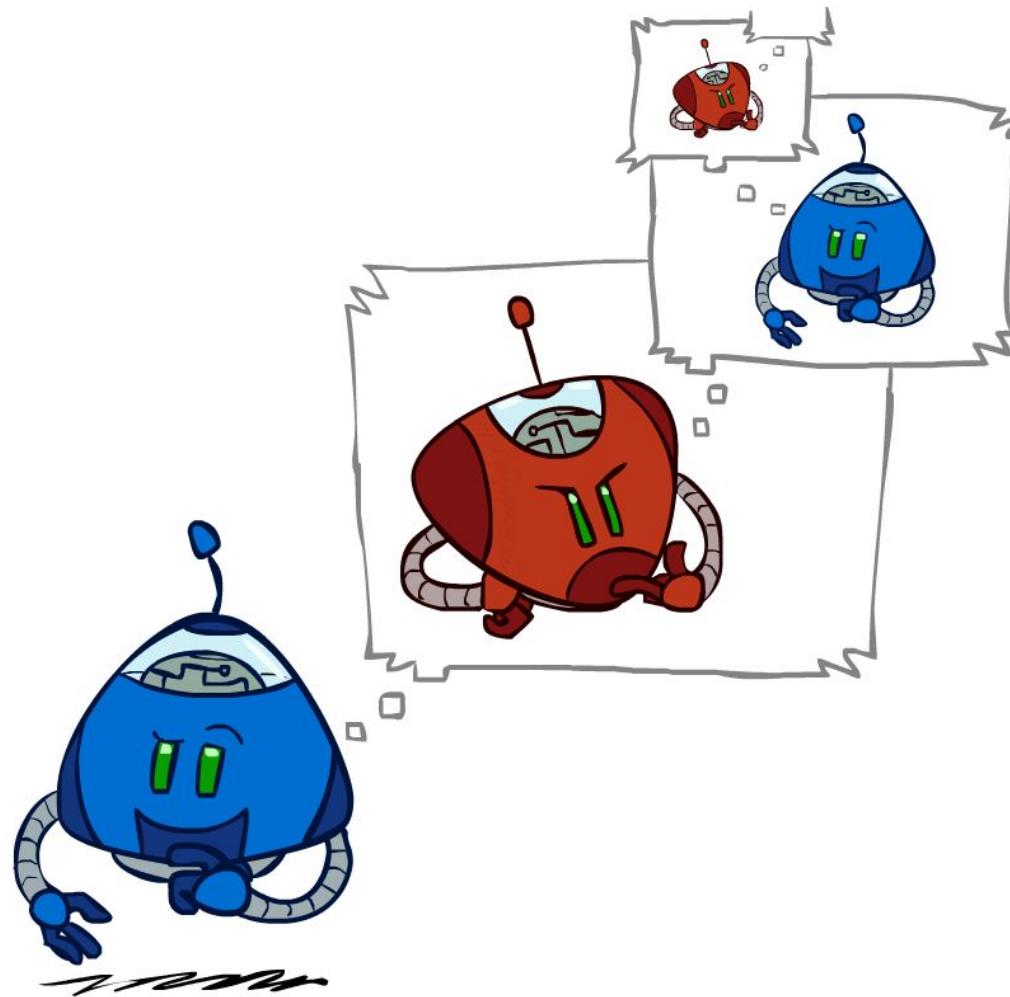
- A game can be defined as
  - $s$  : States
  - $s_0$ : Initial state
  - Player(s) : Defines which player has the move
  - Actions(s) : Returns a set of legal moves
  - Result( $s,a$ ) : Defines the result of a move
  - TerminalTest( $s$ ) : True when game is over, false otherwise
  - Utility( $s,p$ ) : Defines the final numeric value for a game that ends in terminal state  $s$  for player  $p$
- A game tree can be constructed
  - Nodes are game states and edges are moves



<http://ai.berkeley.edu>

# Tic-Tac-Toe Game Tree





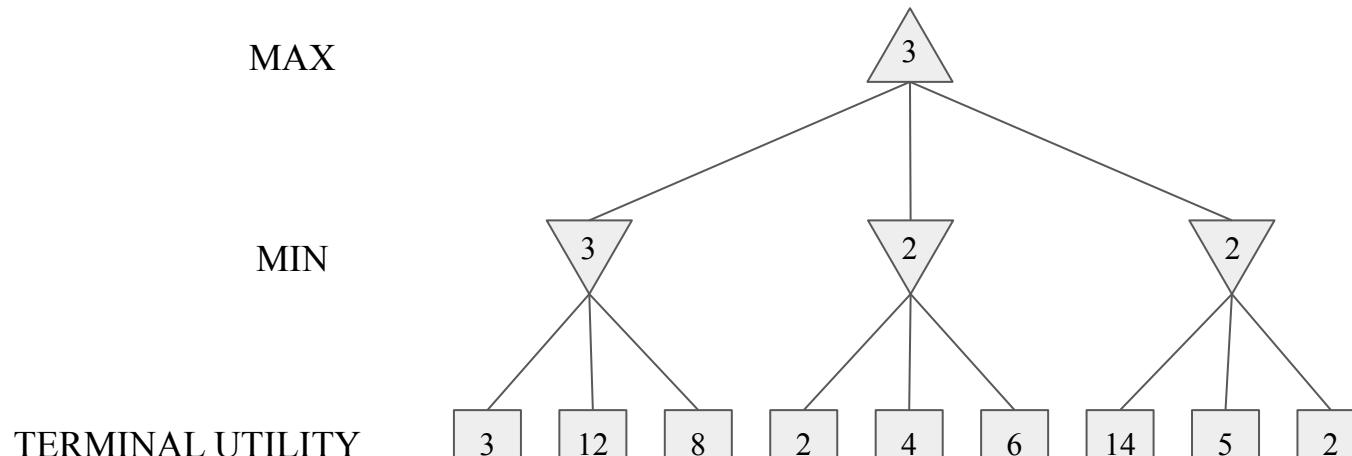
# Minimax Search

- A state-space search tree
- Players alternate turns
- Compute each node's minimax value
  - the best achievable utility against a rational (optimal) adversary

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if } \text{TERMINAL-TEST}(s) \\ \max_{a \in \text{Actions}(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if } \text{PLAYER}(s) = \text{MAX} \\ \min_{a \in \text{Actions}(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if } \text{PLAYER}(s) = \text{MIN} \end{cases}$$

- Will lead to optimal strategy
  - Best achievable payoff against best play

# Minimax Search - Example



$\text{MINIMAX}(s) =$

$$\begin{cases} \text{UTILITY}(s) & \text{if } \text{TERMINAL-TEST}(s) \\ \max_{a \in \text{Actions}(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if } \text{PLAYER}(s) = \text{MAX} \\ \min_{a \in \text{Actions}(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if } \text{PLAYER}(s) = \text{MIN} \end{cases}$$

# Minimax Implementation

```
function MINIMAX-DECISION(state) returns an action
    return  $\arg \max_{a \in \text{ACTIONS}(s)} \text{MIN-VALUE}(\text{RESULT}(s, a))$ 
```

---

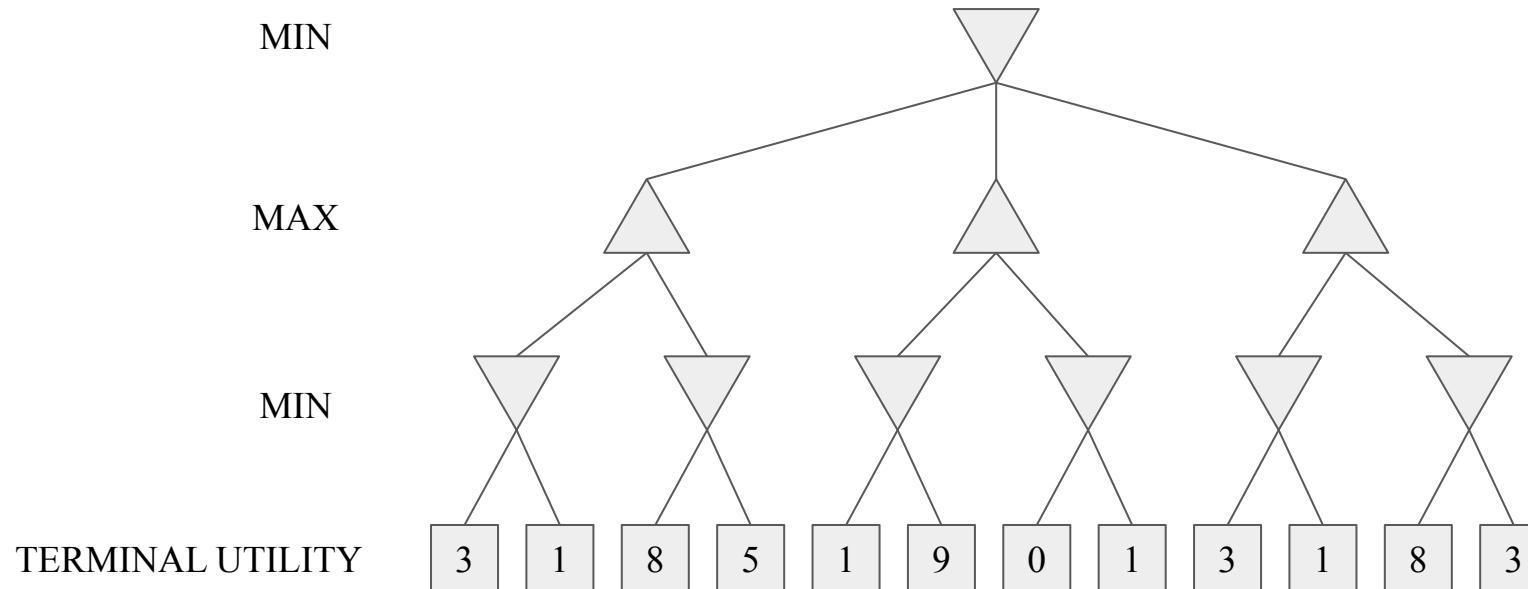
```
function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
     $v \leftarrow -\infty$ 
    for each a in ACTIONS(state) do
         $v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(\text{RESULT}(s, a)))$ 
    return v
```

---

```
function MIN-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
     $v \leftarrow \infty$ 
    for each a in ACTIONS(state) do
         $v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(\text{RESULT}(s, a)))$ 
    return v
```

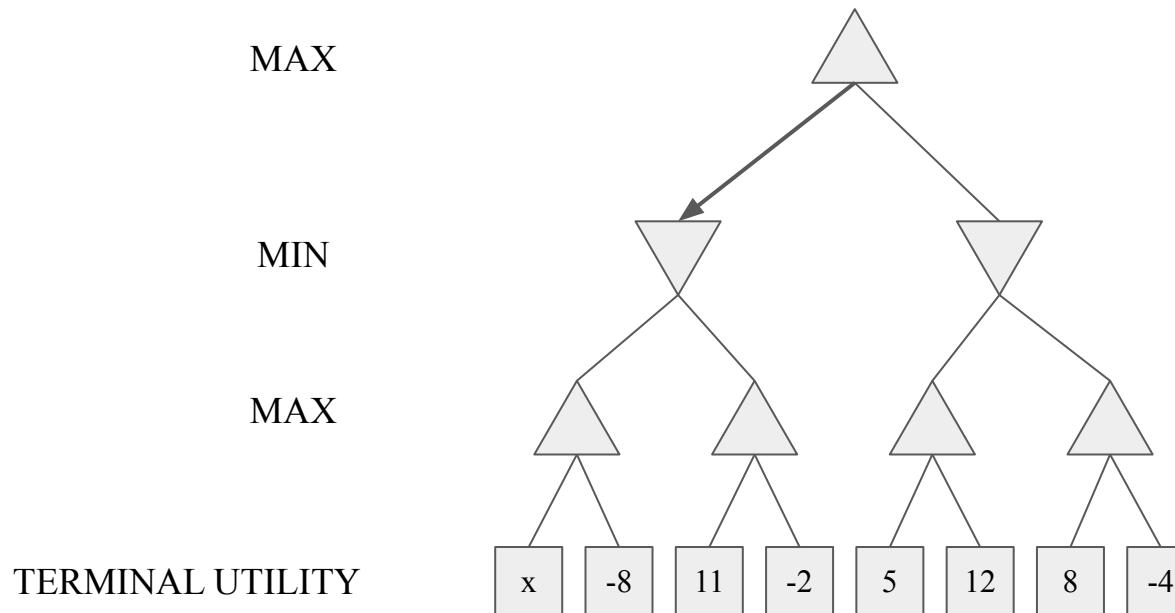
# Quiz

- Consider the simple game tree shown below
  - What is the minimax value of the game tree?
  - Which action will the minimizer take when playing according to the minimax strategy?



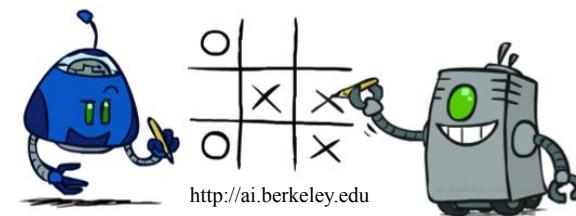
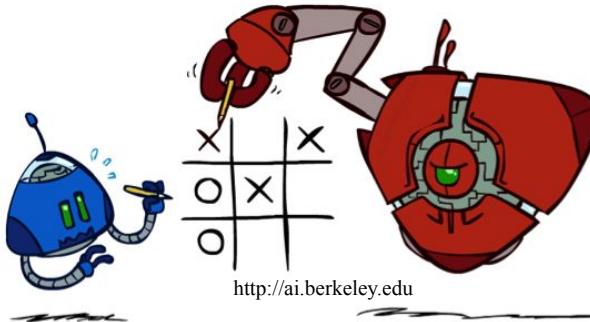
# Quiz

- Consider the following game tree, where one of the leaves has an unknown payoff  $x$
- For what values of  $x$  is MAX guaranteed to choose the initial left action?



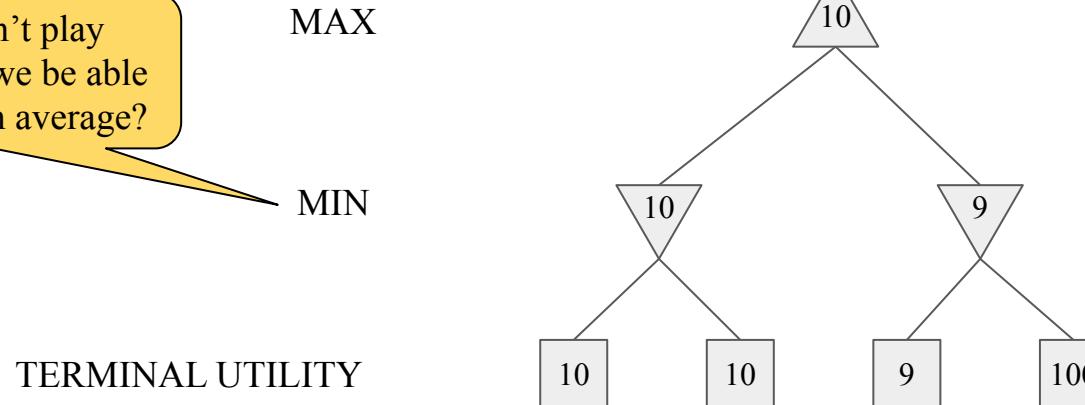
# Minimax Properties

- Will minimax lead to optimal play?
  - No
  - It will lead to an optimal strategy
    - I.e., it will be optimal against perfect play



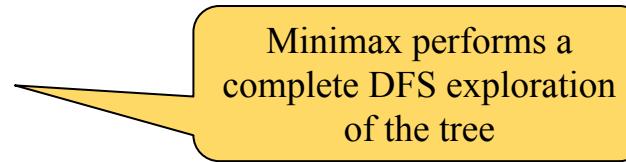
# What if MIN does not play optimally?

If MIN doesn't play  
optimally, will we be able  
to play better on average?



# Minimax Properties

- Complete?
  - Yes, if tree is finite
- Optimal?
  - In general no, yes against an optimal opponent
- Time complexity?
  - $O(b^m)$
- Space complexity
  - $O(bm)$



Minimax performs a complete DFS exploration of the tree

# Example: Chess

- $b \approx 35, m \approx 100$ 
  - Cannot search to leaves !



# Resource Limits

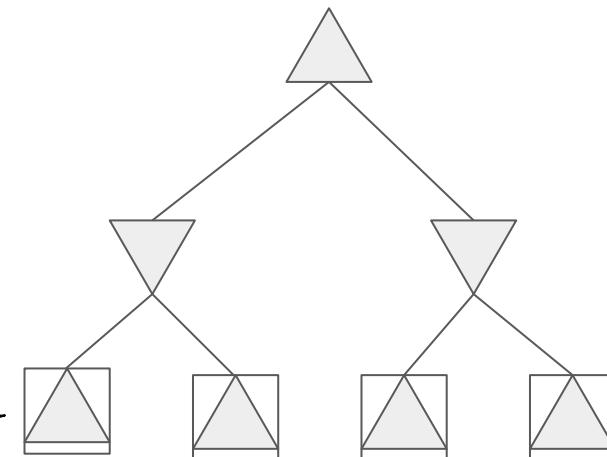


<http://ai.berkeley.edu>

# DLS

- Depth-Limit Search
  - DFS-TSA with a depth limit and an evaluation function
    - i.e., nodes at depth  $l$  have no successors and use value of evaluation function
  - Properties
    - Complete ?
      - No, if  $l < d$ ; Yes, if  $l \geq d$
    - Optimal ?
      - No
    - Time ?
      - $O(b^l)$
    - Space ?
      - $O(bl)$

Cut the tree here

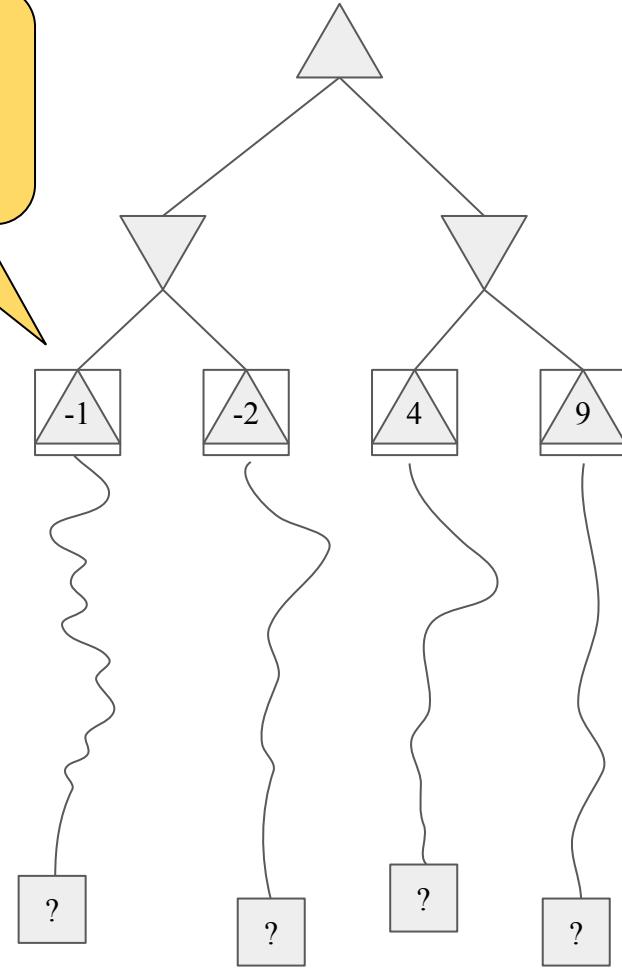


$d$ : depth of the shallowest solution

# DLS

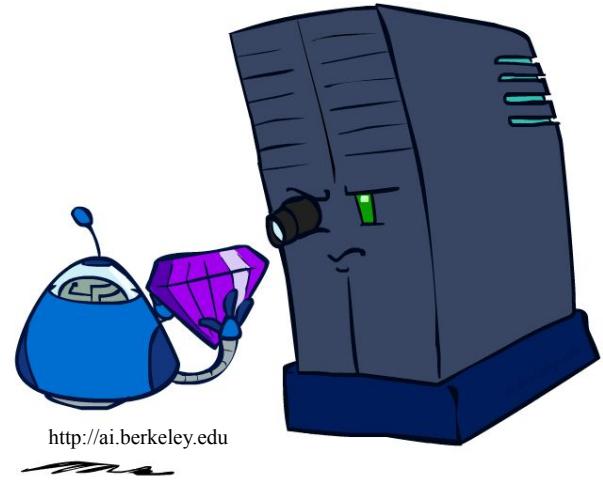
- Depth limited search
  - Search only to a limited depth in the tree
  - Replace terminal utilities with an evaluation function for non-terminal positions
- Problems:
  - Guarantee of optimal play is gone
  - Need to design evaluation function

Cut the tree here  
and plug in these  
numbers from the  
evaluation  
function



# Evaluation Function

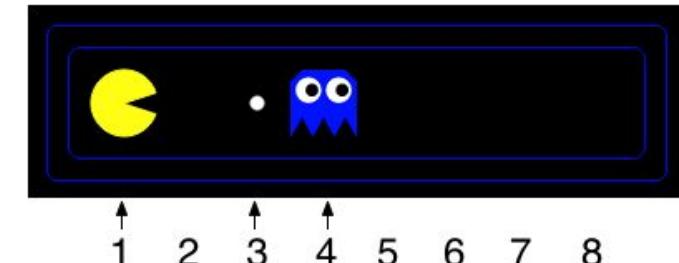
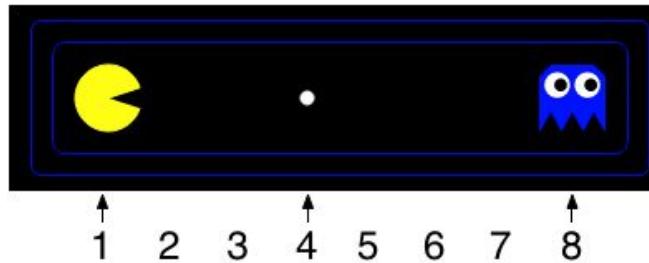
- An evaluation function  $\text{Eval}(s)$  scores non-terminals in depth-limited search
  - An estimate of the expected utility of the game from a given position
- Ideal function
  - The actual minimax value of the position
- The performance of a game-playing program depends strongly on the quality of its evaluation function



<http://ai.berkeley.edu>

# Quiz

- For the two situations shown below, which evaluation functions will give the situation on the left a higher score than the situation on the right?



- $1 / (\text{Pacman's distance to the nearest food pellet})$
- Pacman's distance to the nearest ghost
- Pacman's distance to the nearest ghost +  $1 / (\text{Pacman's distance to the nearest food pellet})$
- Pacman's distance to the nearest ghost +  $1000 / (\text{Pacman's distance to the nearest food pellet})$

# Evaluation Function

$$\text{Eval}(s) = w_1 f_1(s) + w_2 f_2(s) + \dots + w_n f_n(s)$$

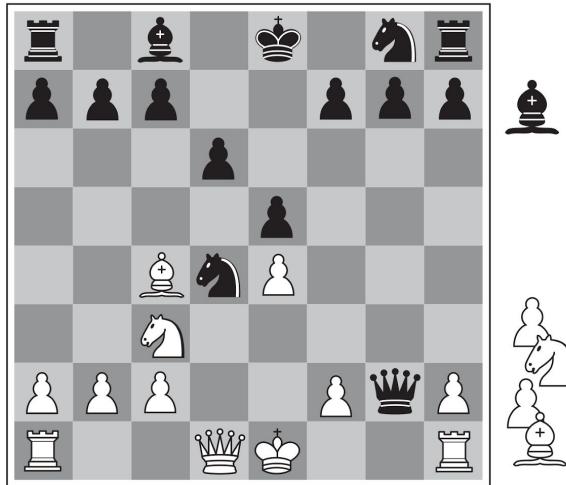
- Example: Chess
  - Successful evaluation functions for chess compute numerical contributions from each feature
    - E.g., each pawn is worth 1, a knight or bishop is worth 3, a rook 5 and a queen 9
    - Other features such as “king safety” or “good pawn structure” might be worth half a pawn
    - The features and their weights are then combined



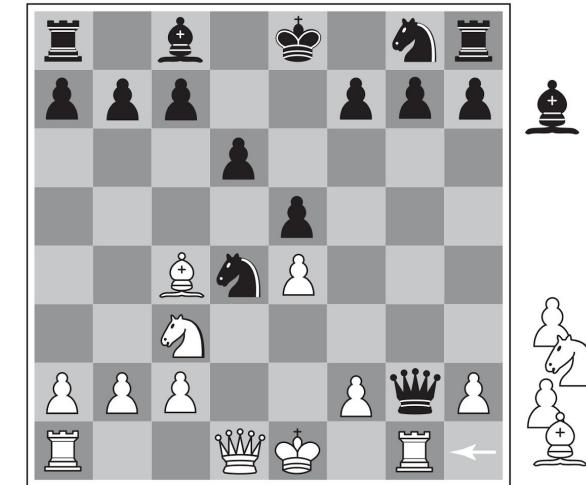
<http://aima.cs.berkeley.edu/>

# Quiz

- Which player has an advantage for (a) and for (b) ?



(a) White to move



(b) White to move

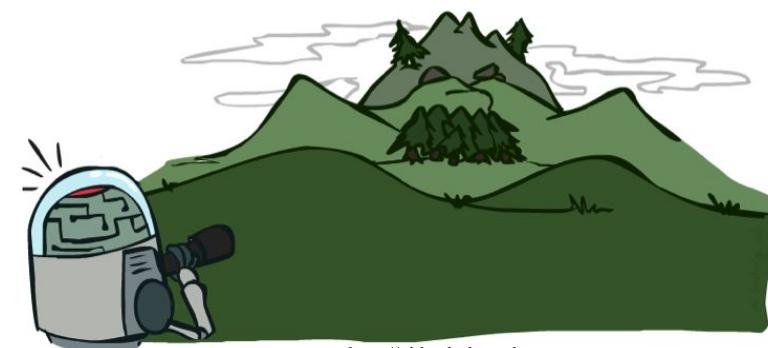
**Figure 5.8** Two chess positions that differ only in the position of the rook at lower right. In (a), Black has an advantage of a knight and two pawns, which should be enough to win the game. In (b), White will capture the queen, giving it an advantage that should be strong enough to win.

# Depth Matters

- Evaluation functions are always imperfect
- The deeper in the tree the evaluation function is buried, the less the quality of the evaluation function matters
- An important example of the trade off between complexity of features and complexity of computation



<http://ai.berkeley.edu>



<http://ai.berkeley.edu>

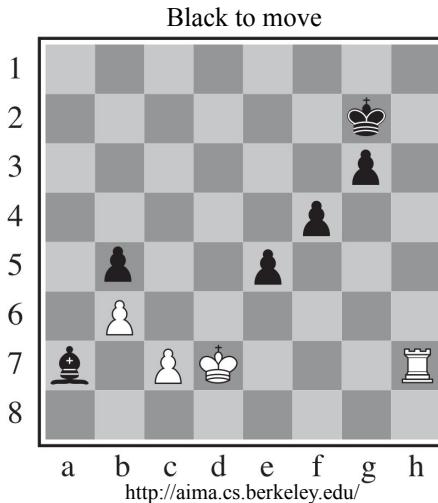
# Horizon Effect

- Consider an opponent's move that causes serious damage and is ultimately unavoidable
- With a low depth limit we don't know that the damage is ultimately unavoidable
- Employing delaying tactics to temporality avoided the damage may cause even more damage



# Horizon Effect - Example

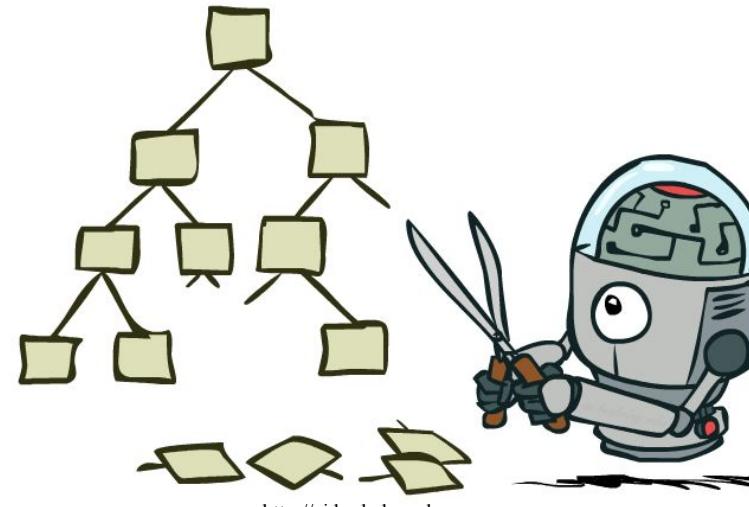
- Consider the following example
  - Can black save its bishop?



**Figure 5.9** The horizon effect. With Black to move, the black bishop is surely doomed. But Black can forestall that event by checking the white king with its pawns, forcing the king to capture the pawns. This pushes the inevitable loss of the bishop over the horizon, and thus the pawn sacrifices are seen by the search algorithm as good moves rather than bad ones.

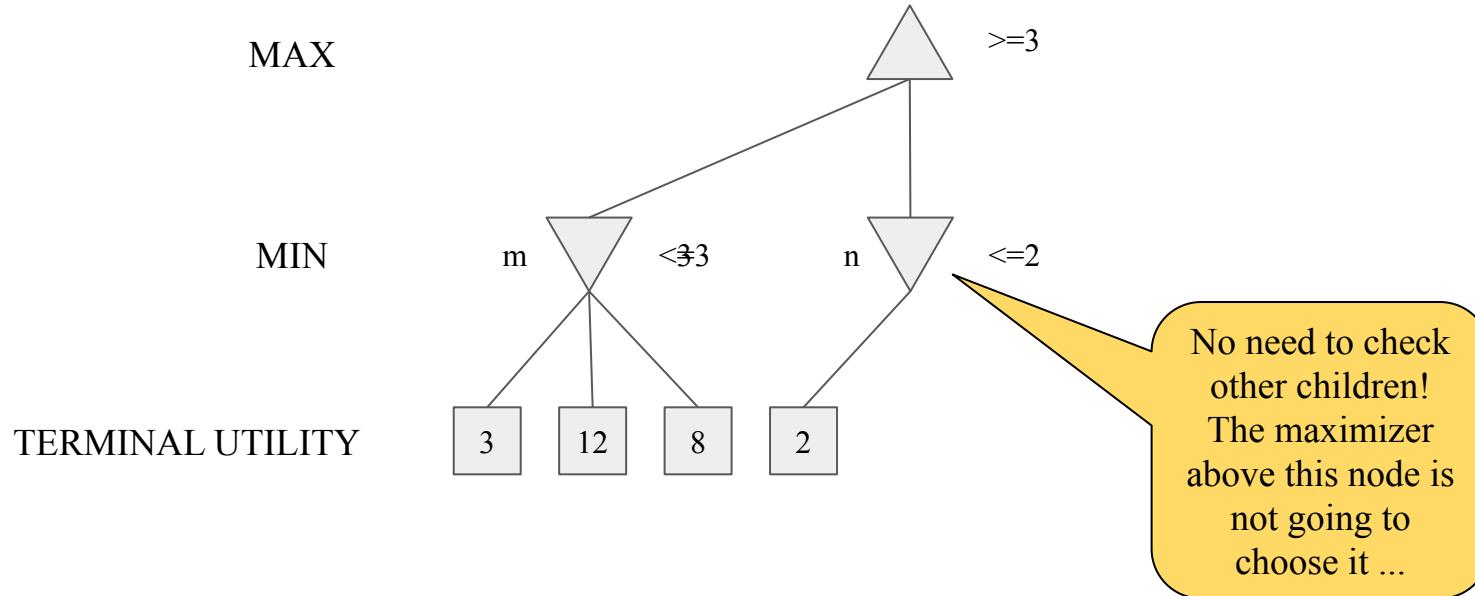
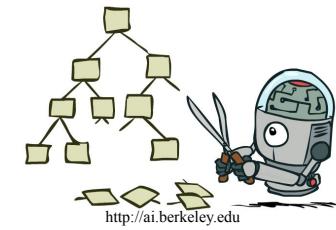
# Game Tree Pruning

- Is it possible to compute the correct minimax value without looking at every node?



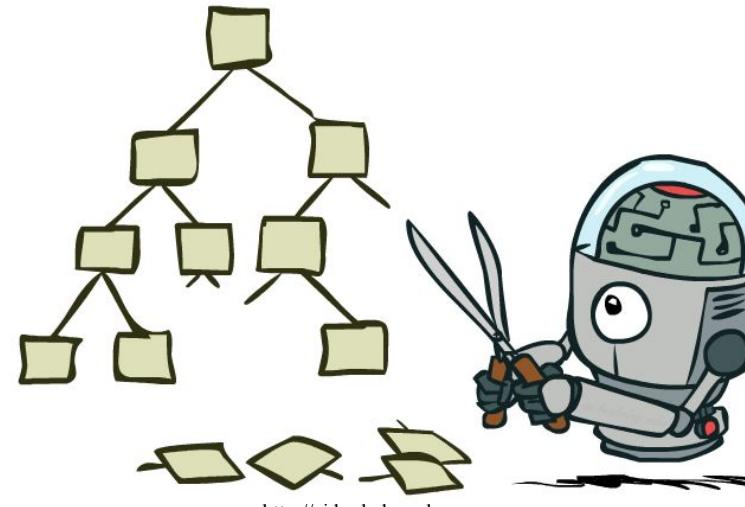
<http://ai.berkeley.edu>

# Pruning - Motivation



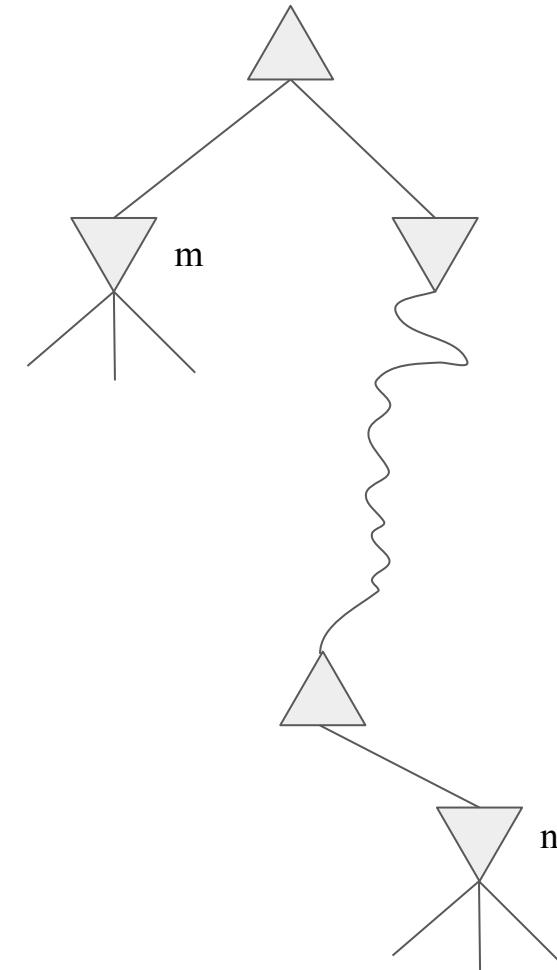
# Pruning

- The  $\alpha$ - $\beta$  pruning algorithm can determine the minimax value without looking at all the nodes



# $\alpha$ - $\beta$ Pruning Algorithm

- Min version
  - Consider Min's value at some node n
  - n will decrease (or stay constant) while the descendants of n are examined
  - Let m be the best value that Max can get at any choice point along the current path from the root
  - If n becomes worse (<) than m
    - Max will avoid it
    - Stop considering n's other children
- Max version is symmetric



# $\alpha$ - $\beta$ Pruning Algorithm

---

**function** MINIMAX-DECISION(*state*) **returns** an action  
**return**  $\arg \max_{a \in \text{ACTIONS}(s)} \text{MIN-VALUE}(\text{RESULT}(s, a))$

---

**function** MAX-VALUE(*state*) **returns** a utility value  
**if** TERMINAL-TEST(*state*) **then return** UTILITY(*state*)  
 $v \leftarrow -\infty$   
**for each** *a* **in** ACTIONS(*state*) **do**  
 $v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(\text{RESULT}(s, a)))$   
**return** *v*

---

**function** MIN-VALUE(*state*) **returns** a utility value  
**if** TERMINAL-TEST(*state*) **then return** UTILITY(*state*)  
 $v \leftarrow \infty$   
**for each** *a* **in** ACTIONS(*state*) **do**  
 $v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(\text{RESULT}(s, a)))$   
**return** *v*

---

$\alpha$  = best explored option along path to root for max  
 $\beta$  = best explored option along path to root for min

**function** ALPHA-BETA-SEARCH(*state*) **returns** an action  
 $v \leftarrow \text{MAX-VALUE}(s, -\infty, +\infty)$   
**return** the action in ACTIONS(*state*) with value *v*

---

**function** MAX-VALUE(*state*,  $\alpha$ ,  $\beta$ ) **returns** a utility value  
**if** TERMINAL-TEST(*state*) **then return** UTILITY(*state*)  
 $v \leftarrow -\infty$   
**for each** *a* **in** ACTIONS(*state*) **do**  
 $v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(\text{RESULT}(s, a), \alpha, \beta))$   
**if** *v*  $\geq \beta$  **then return** *v*  
 $\alpha \leftarrow \text{MAX}(\alpha, v)$   
**return** *v*

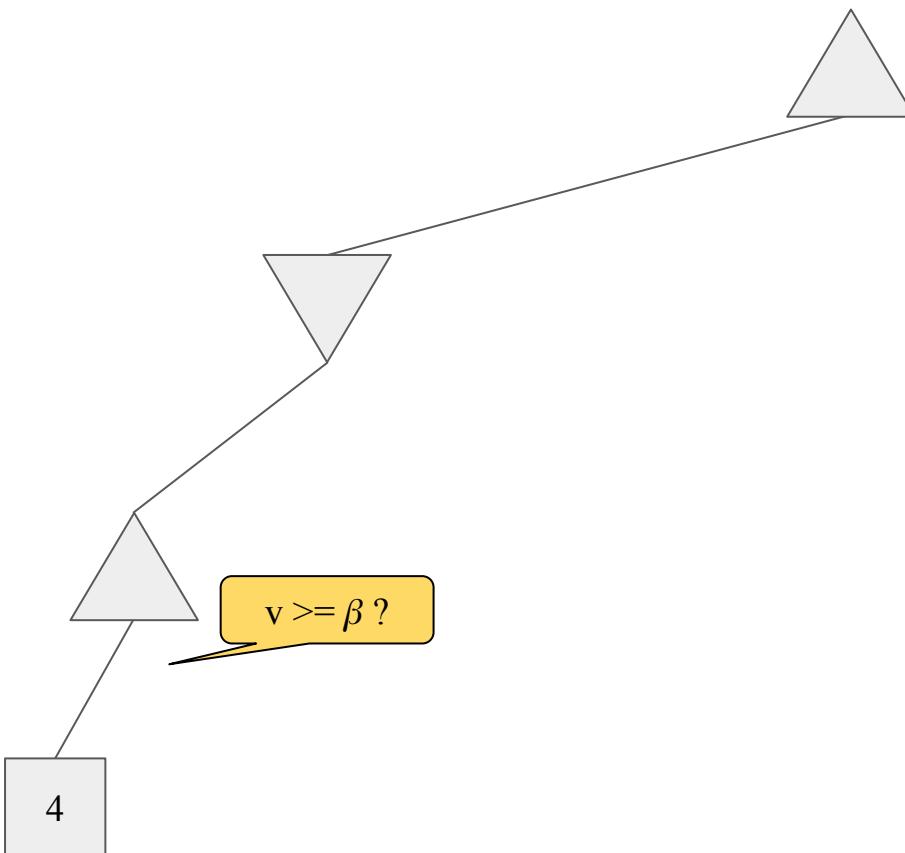
---

**function** MIN-VALUE(*state*,  $\alpha$ ,  $\beta$ ) **returns** a utility value  
**if** TERMINAL-TEST(*state*) **then return** UTILITY(*state*)  
 $v \leftarrow +\infty$   
**for each** *a* **in** ACTIONS(*state*) **do**  
 $v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(\text{RESULT}(s, a), \alpha, \beta))$   
**if** *v*  $\leq \alpha$  **then return** *v*  
 $\beta \leftarrow \text{MIN}(\beta, v)$   
**return** *v*

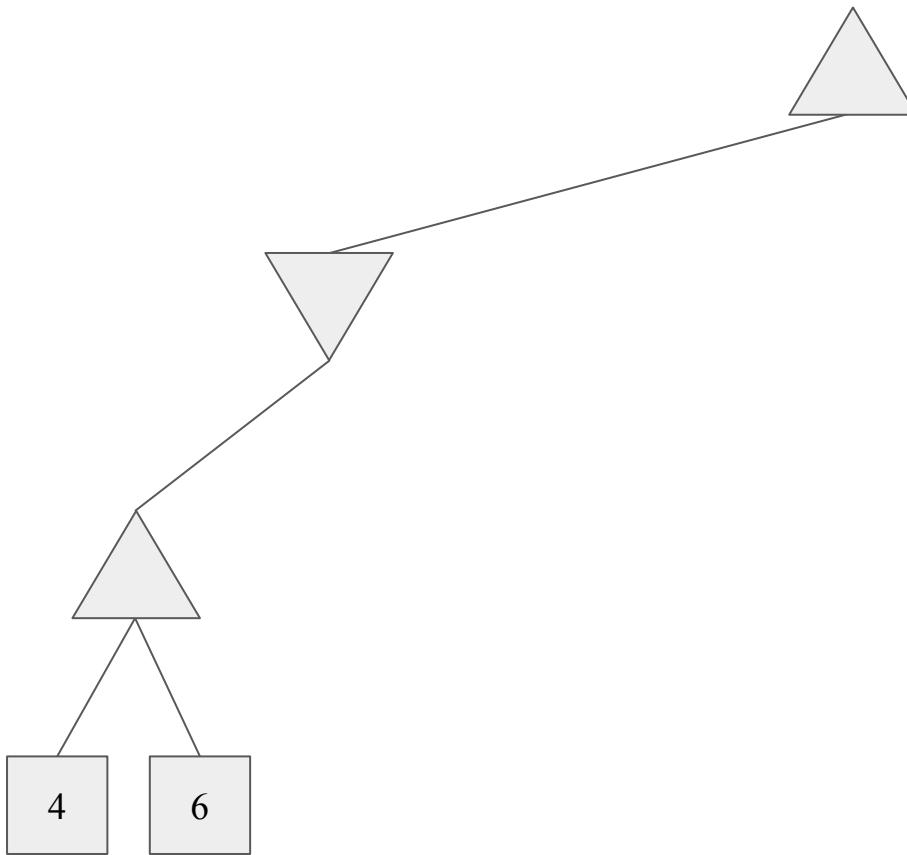
# $\alpha$ - $\beta$ Pruning Example

$\alpha$  = best explored option along path to root for max  
 $\beta$  = best explored option along path to root for min

Pruning Conditions:  $v \geq \beta$  (max),  $v \leq \alpha$  (min)  
No option:  $-\infty$  (max),  $\infty$  (min)



# $\alpha$ - $\beta$ Pruning Example



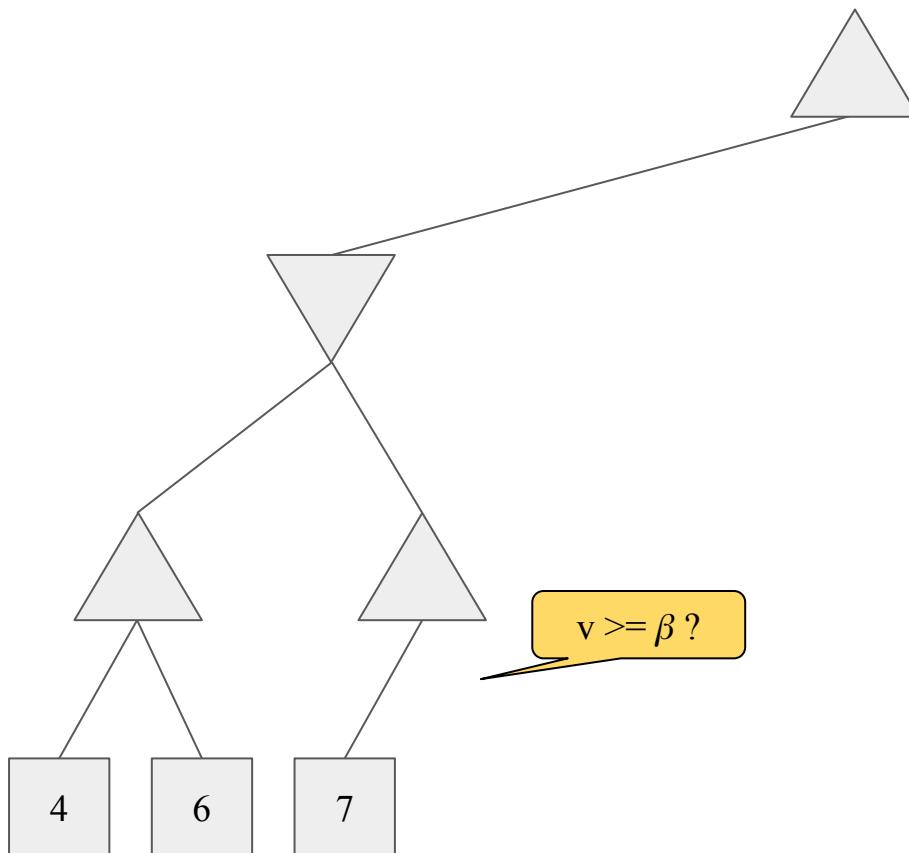
$\alpha$  = best explored option along path to root for max  
 $\beta$  = best explored option along path to root for min

Pruning Conditions:  $v \geq \beta$  (max),  $v \leq \alpha$  (min)  
No option:  $-\infty$  (max),  $\infty$  (min)

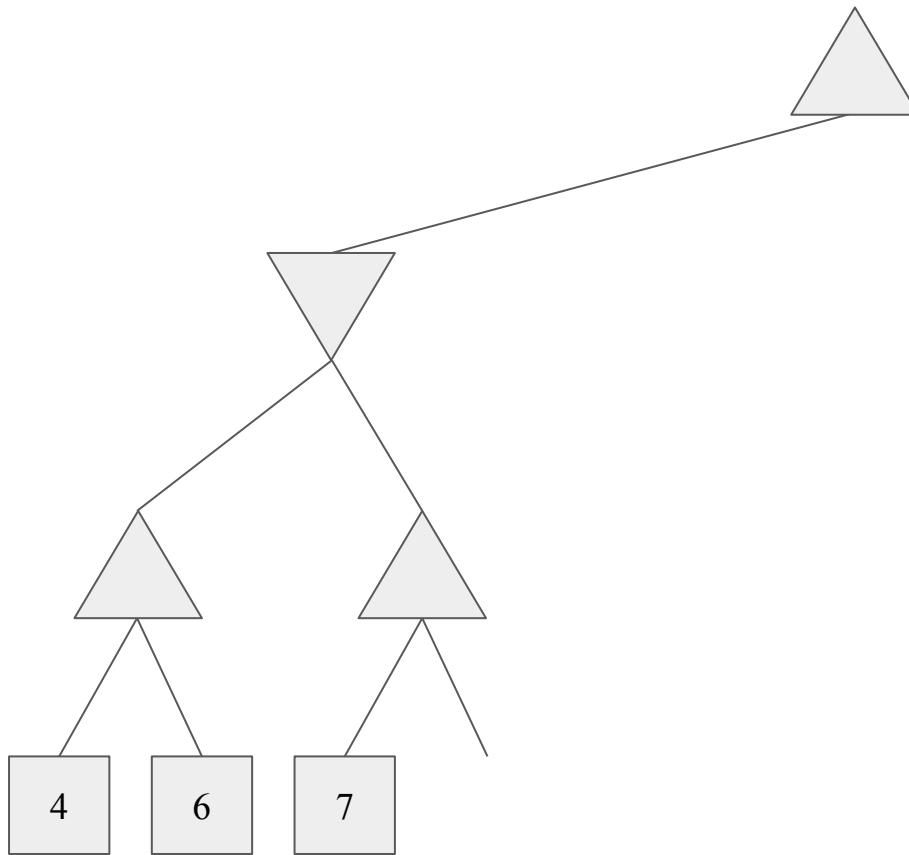
# $\alpha$ - $\beta$ Pruning Example

$\alpha$  = best explored option along path to root for max  
 $\beta$  = best explored option along path to root for min

Pruning Conditions:  $v \geq \beta$  (max),  $v \leq \alpha$  (min)  
No option:  $-\infty$  (max),  $\infty$  (min)



# $\alpha$ - $\beta$ Pruning Example



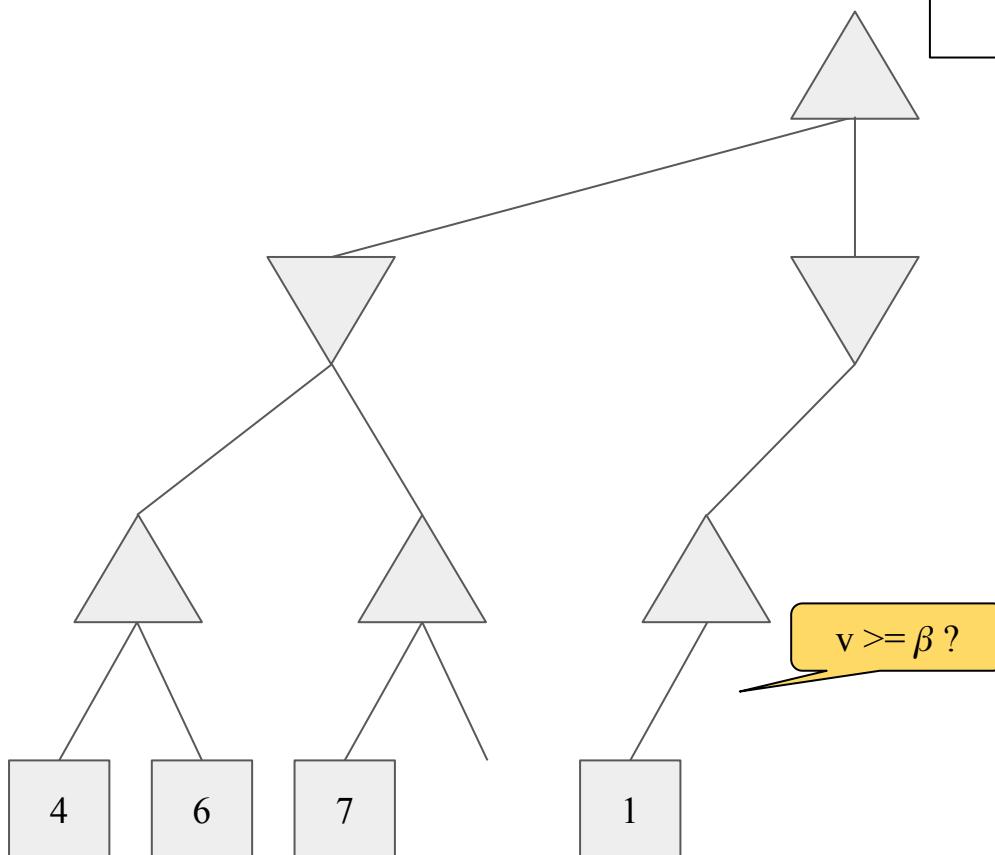
$\alpha$  = best explored option along path to root for max  
 $\beta$  = best explored option along path to root for min

Pruning Conditions:  $v \geq \beta$  (max),  $v \leq \alpha$  (min)  
No option:  $-\infty$  (max),  $\infty$  (min)

# $\alpha$ - $\beta$ Pruning Example

$\alpha$  = best explored option along path to root for max  
 $\beta$  = best explored option along path to root for min

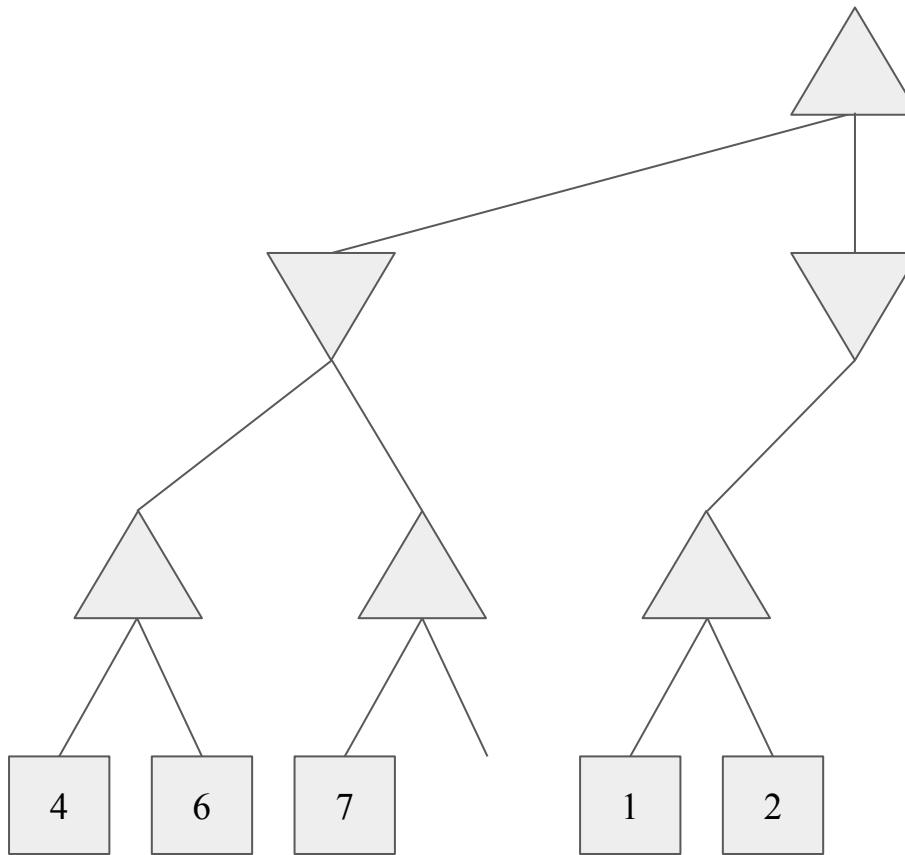
Pruning Conditions:  $v \geq \beta$  (max),  $v \leq \alpha$  (min)  
No option:  $-\infty$  (max),  $\infty$  (min)



# $\alpha$ - $\beta$ Pruning Example

$\alpha$  = best explored option along path to root for max  
 $\beta$  = best explored option along path to root for min

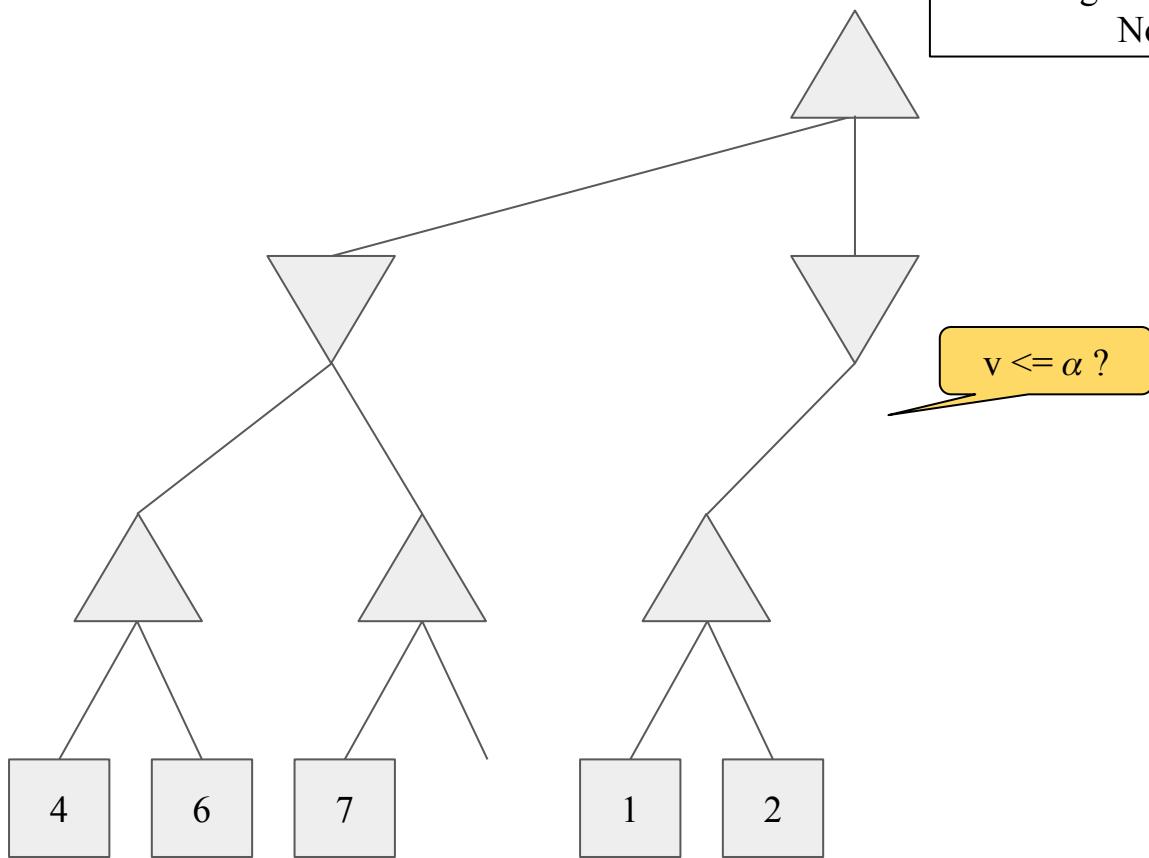
Pruning Conditions:  $v \geq \beta$  (max),  $v \leq \alpha$  (min)  
No option:  $-\infty$  (max),  $\infty$  (min)



# $\alpha$ - $\beta$ Pruning Example

$\alpha$  = best explored option along path to root for max  
 $\beta$  = best explored option along path to root for min

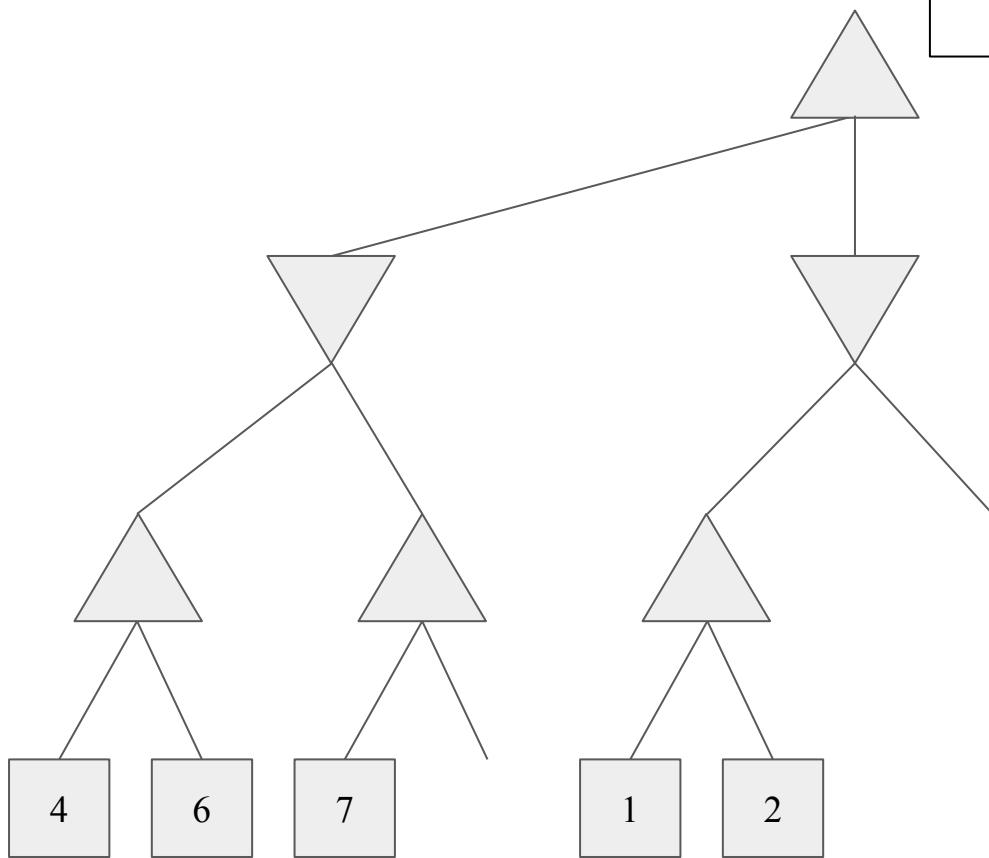
Pruning Conditions:  $v \geq \beta$  (max),  $v \leq \alpha$  (min)  
No option:  $-\infty$  (max),  $\infty$  (min)



# $\alpha$ - $\beta$ Pruning Example

$\alpha$  = best explored option along path to root for max  
 $\beta$  = best explored option along path to root for min

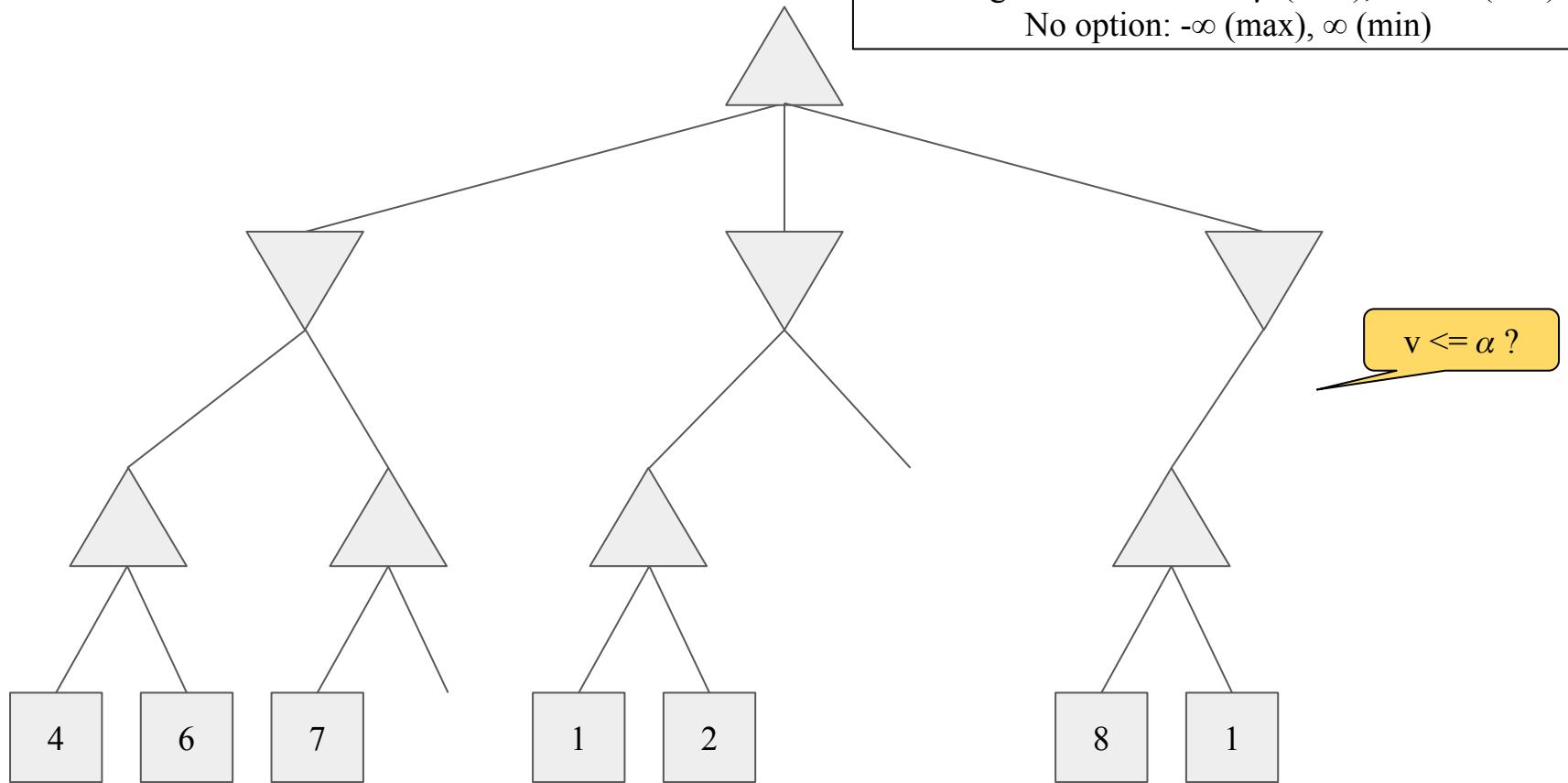
Pruning Conditions:  $v \geq \beta$  (max),  $v \leq \alpha$  (min)  
No option:  $-\infty$  (max),  $\infty$  (min)



# $\alpha$ - $\beta$ Pruning Example

$\alpha$  = best explored option along path to root for max  
 $\beta$  = best explored option along path to root for min

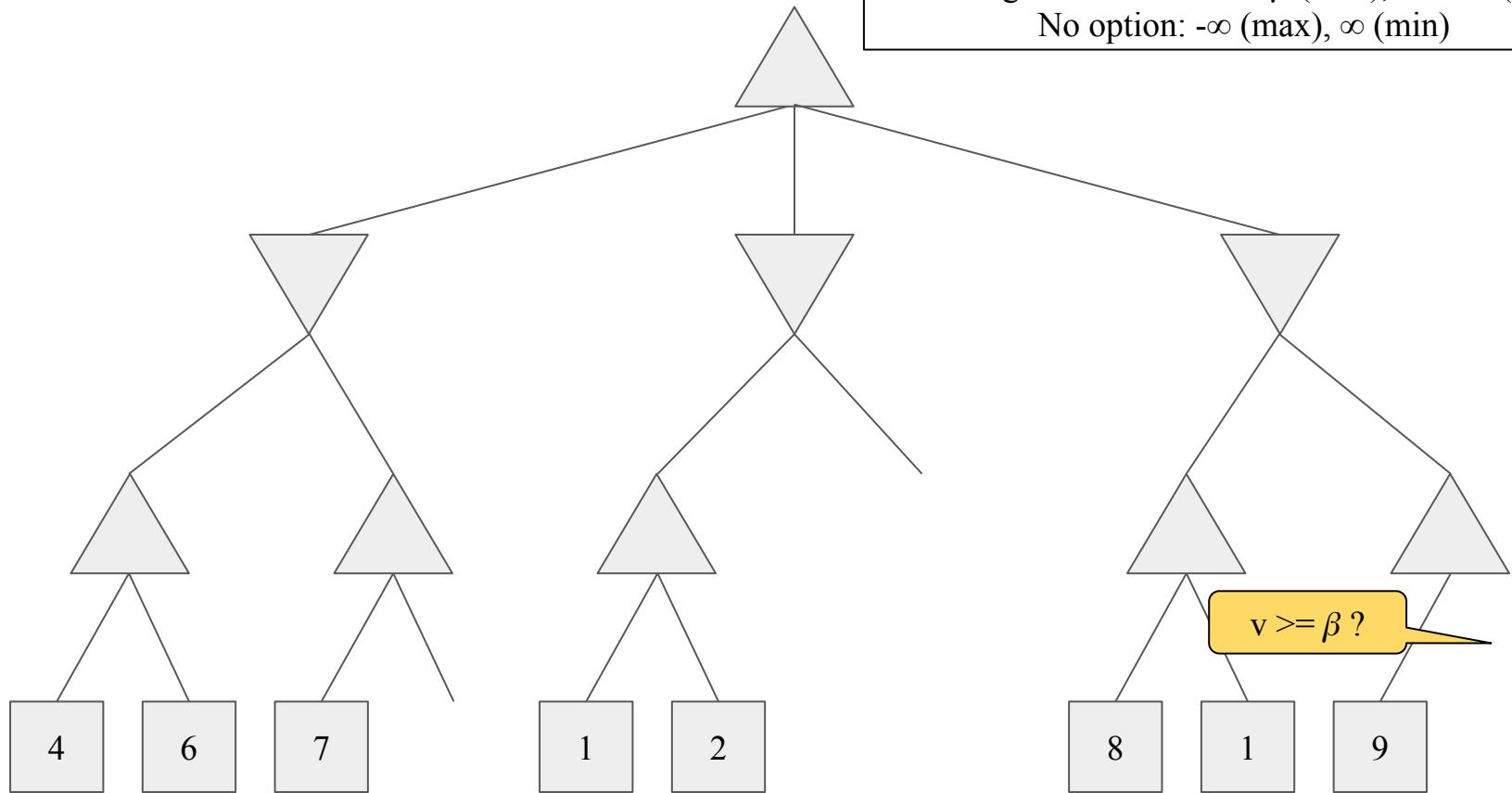
Pruning Conditions:  $v \geq \beta$  (max),  $v \leq \alpha$  (min)  
No option:  $-\infty$  (max),  $\infty$  (min)



# $\alpha$ - $\beta$ Pruning Example

$\alpha$  = best explored option along path to root for max  
 $\beta$  = best explored option along path to root for min

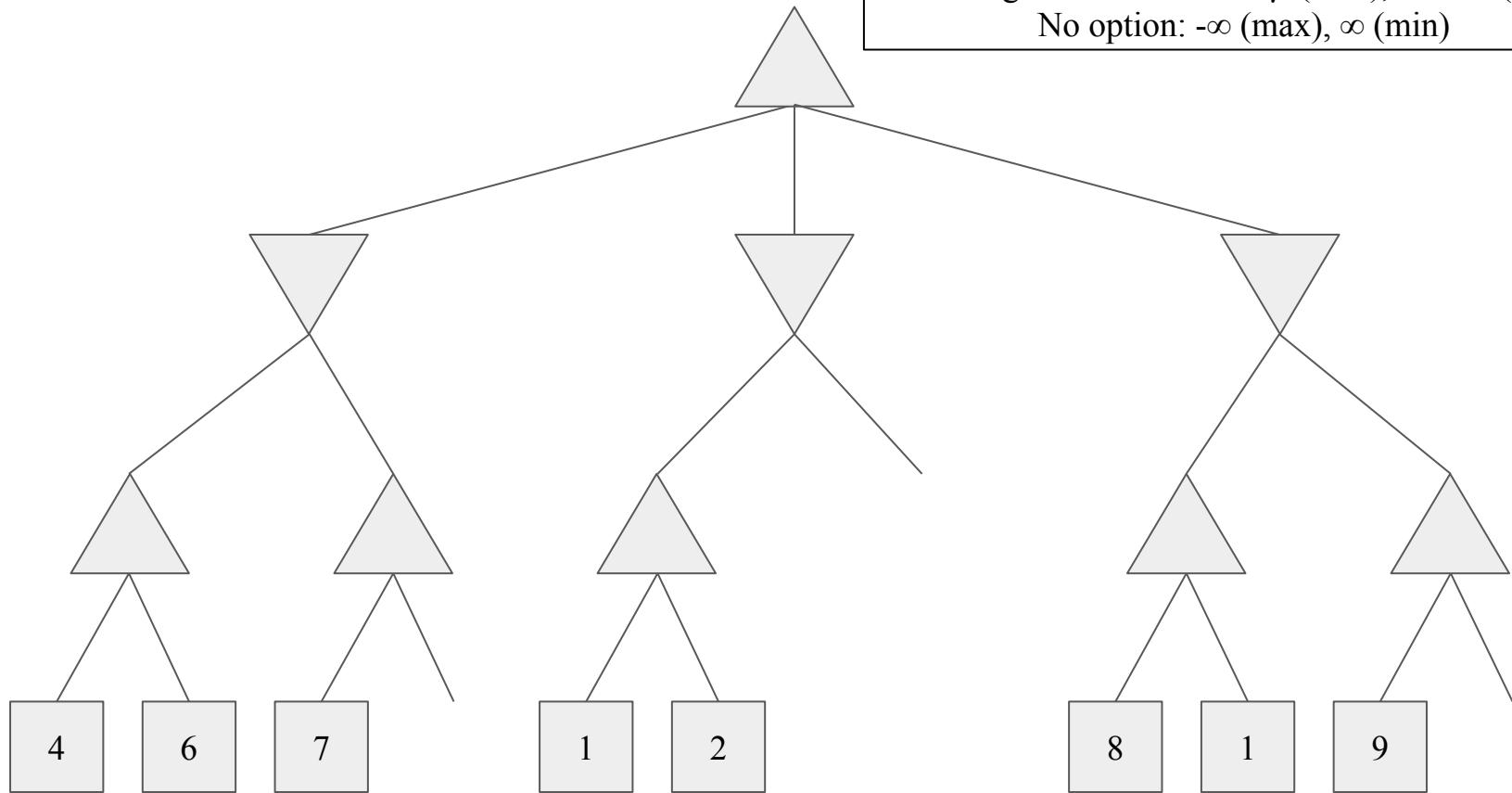
Pruning Conditions:  $v \geq \beta$  (max),  $v \leq \alpha$  (min)  
No option:  $-\infty$  (max),  $\infty$  (min)



# $\alpha$ - $\beta$ Pruning Example

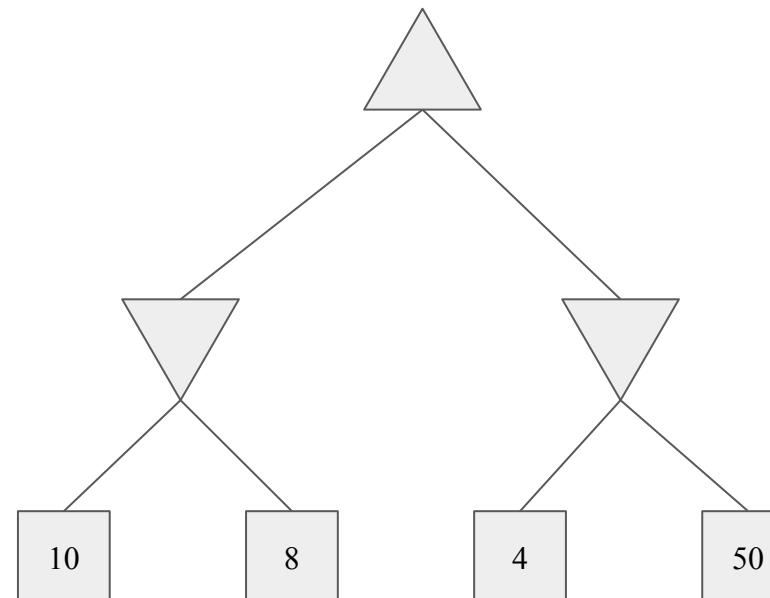
$\alpha$  = best explored option along path to root for max  
 $\beta$  = best explored option along path to root for min

Pruning Conditions:  $v \geq \beta$  (max),  $v \leq \alpha$  (min)  
No option:  $-\infty$  (max),  $\infty$  (min)



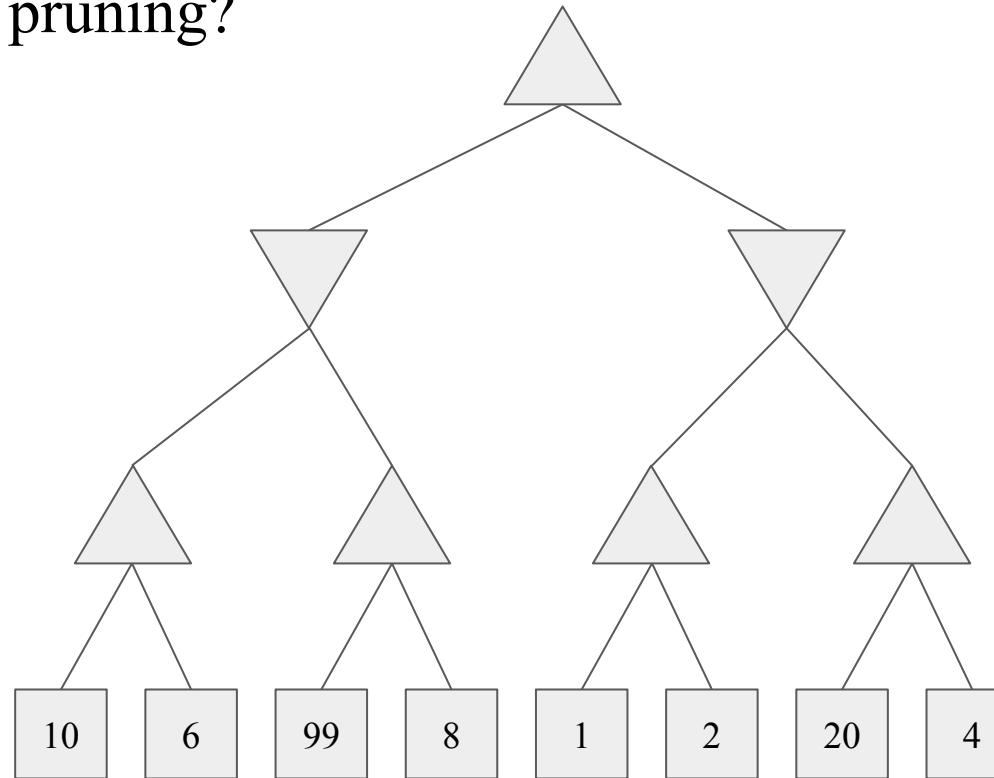
# Quiz

- For the game tree shown below, which branches will be pruned by alpha-beta pruning?



# Quiz

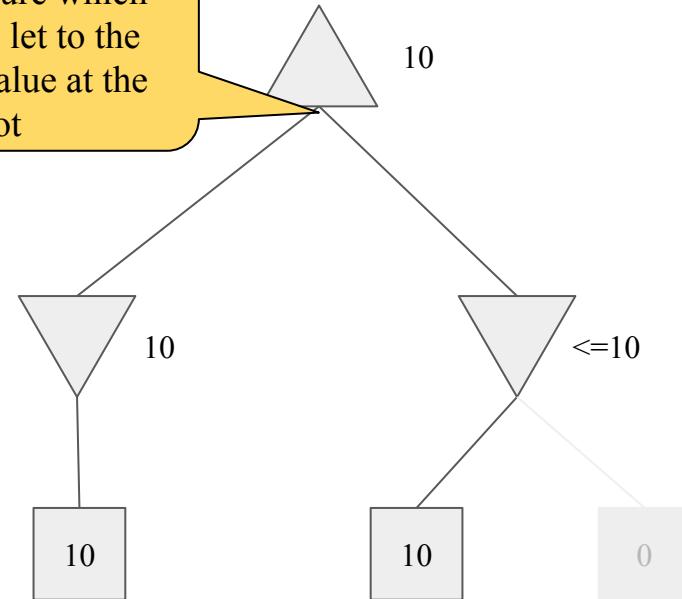
- For the game tree shown below, which branches will be pruned by alpha-beta pruning?



# $\alpha$ - $\beta$ Pruning Properties

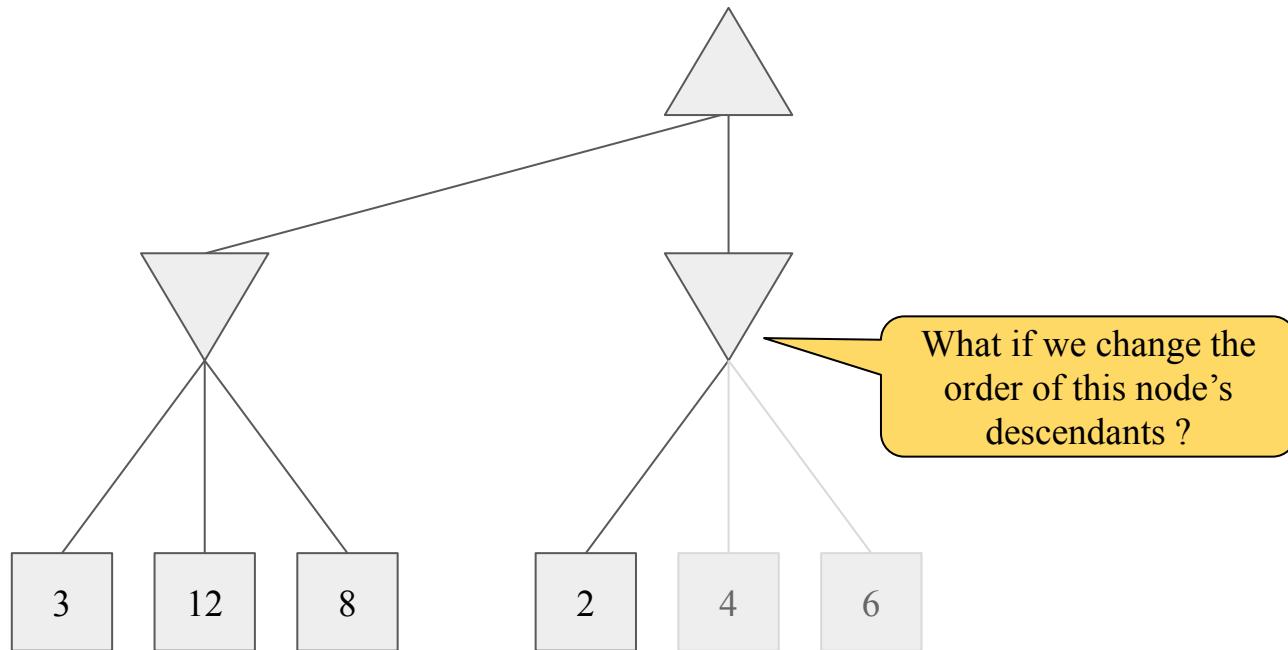
- Pruning has no effect on the minimax value at the root
- However, values of intermediate nodes might be wrong
  - Action selection not appropriate for this simple version of alpha-beta pruning

We are unsure which action will lead to the minimax value at the root



# Move Ordering

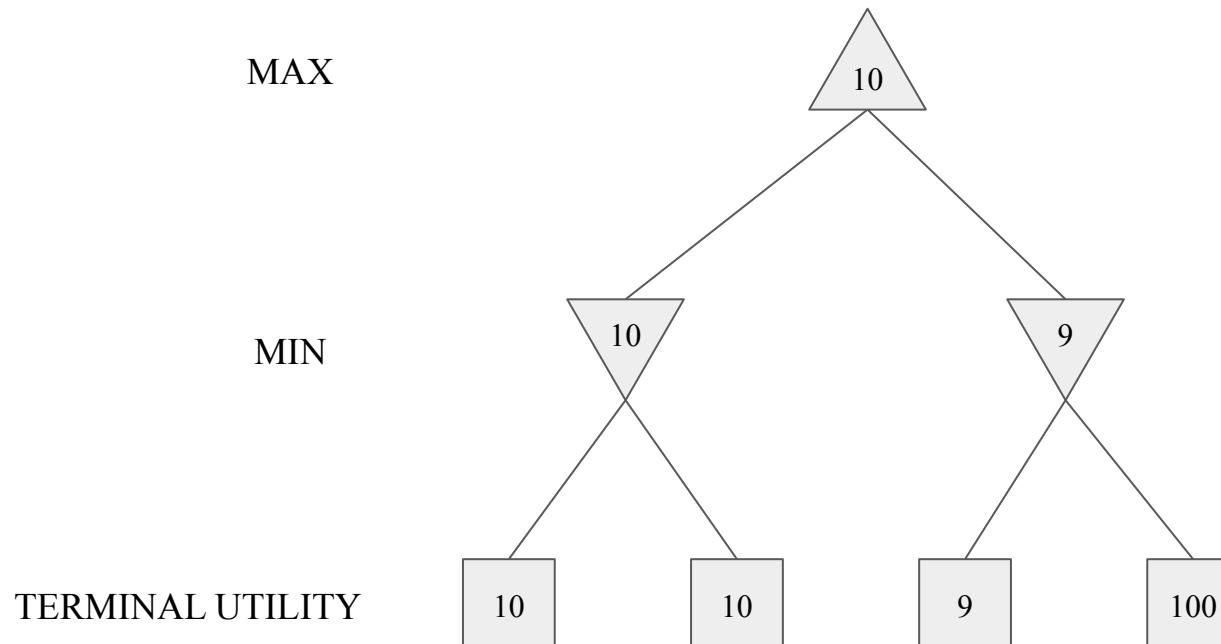
- The effectiveness of alpha-beta pruning is highly dependent on the order in which states are examined



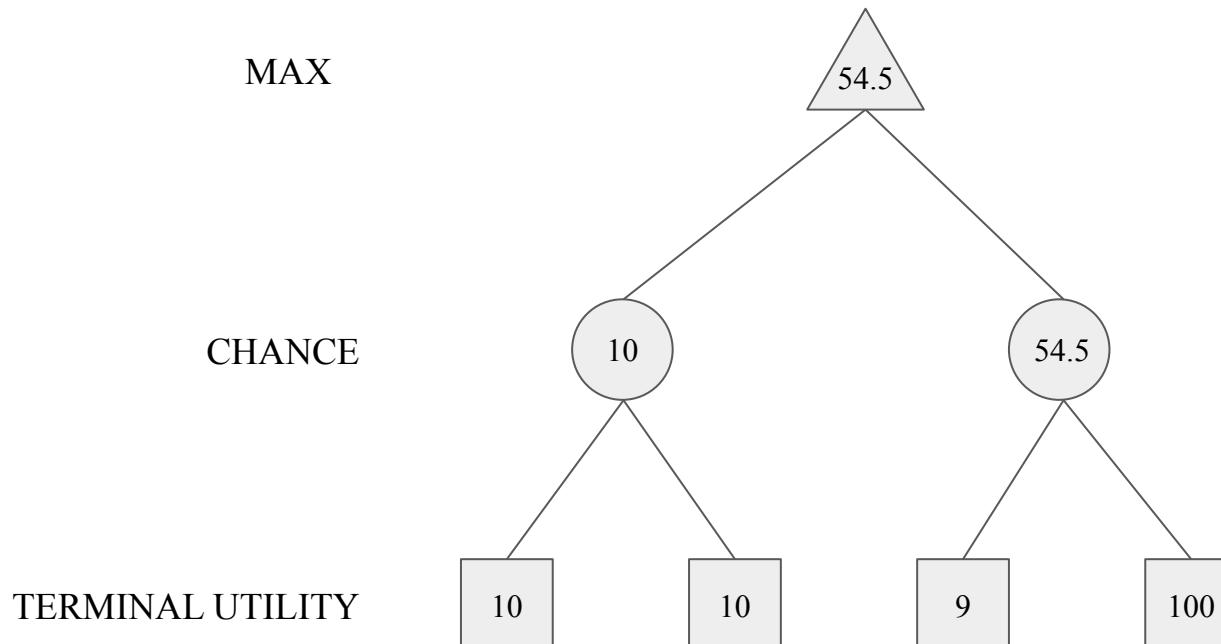
# Alpha-Beta Pruning Properties

- It is worthwhile to try to examine first the successors that are likely best
- If this can be done, then it turns out that alpha-beta needs to examine only  $O(b^{m/2})$  nodes to pick the best move, instead of  $O(b^m)$  for minimax
  - Knuth, D. E., and Moore, R. W. [An analysis of Alpha-Beta pruning](#), 1975

# Worst Case vs. Average Case



# Worst Case vs. Average Case

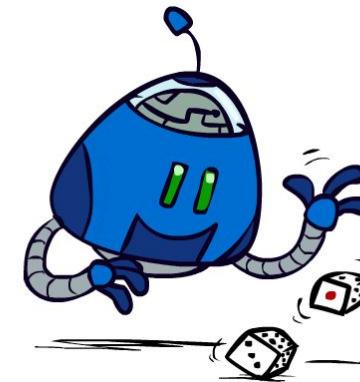


# Chance Note

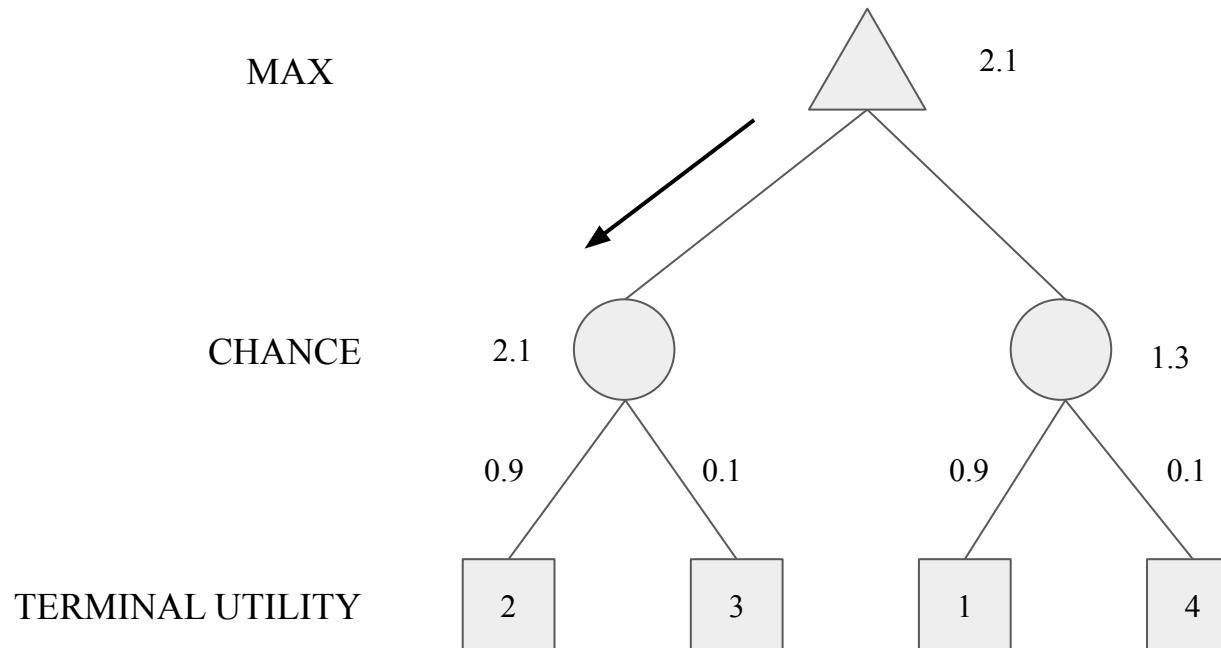
- Why wouldn't we know what the result of an action will be?
  - Explicit randomness: rolling dice
  - Unpredictable opponents: the ghosts respond randomly
  - Actions can fail: when moving a robot, wheels might slip

# Expectimax Search

- Values reflect average case outcomes, not worst case outcomes
- Expectimax search computes the expected score under optimal play
  - Max nodes as in minimax search
  - Chance nodes are like min nodes but the outcome is uncertain
  - Calculate their expected utilities
    - I.e., take weighted average of children

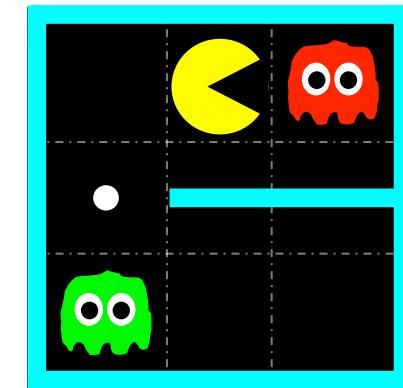


# Expectimax Search Example



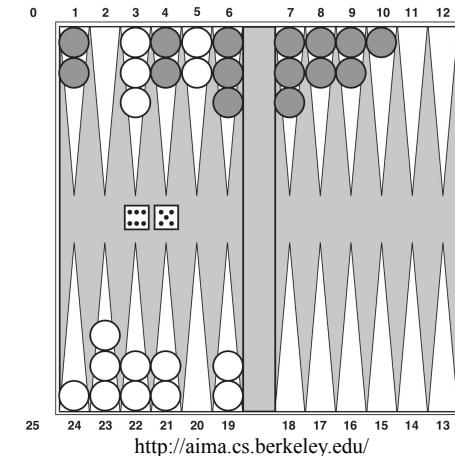
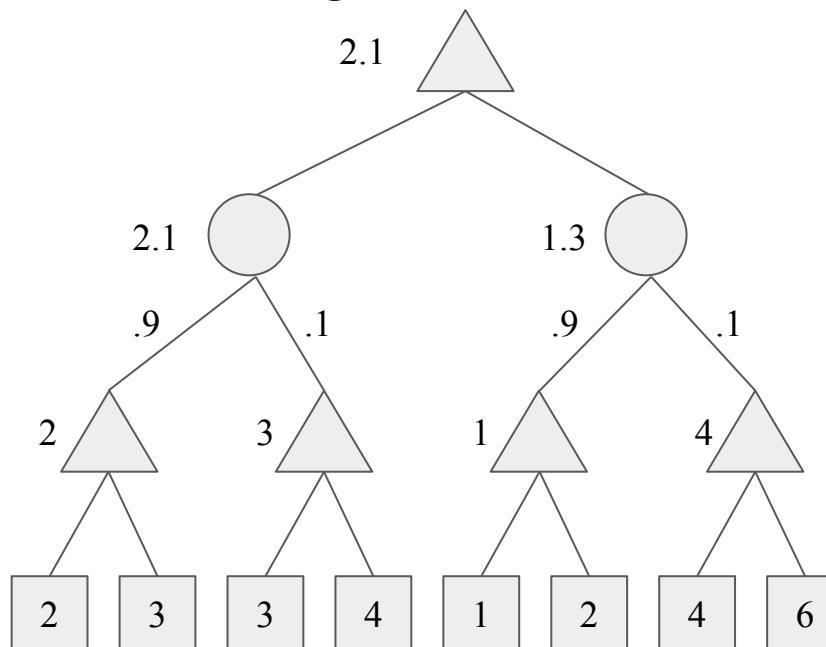
# Quiz

- Given the maze below, build the expectimax tree. Assume that Pacman moves first, followed by the lower left ghost, then the top right ghost
- Rules:
  - Ghosts cannot change direction unless they are facing a wall. The possible actions are east, west, south, and north (not stop). Initially, they have no direction and can move to any adjacent square
  - We use random ghosts which choose uniformly between legal moves
  - Assume that Pacman cannot stop
  - The game is scored as follows
    - -1 for each action Pacman takes
    - 10 for each food dot eaten
    - -500 for losing (if Pacman is eaten)
    - 500 for winning (all food dots eaten)

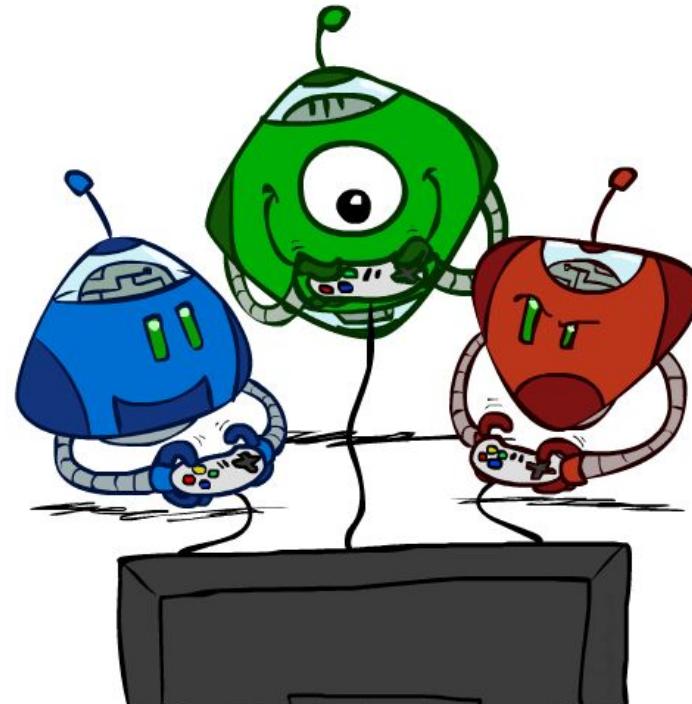


# Expectiminimax Search

- Expectiminimax
  - Environment is an extra “random agent” player that moves after each min/max agent

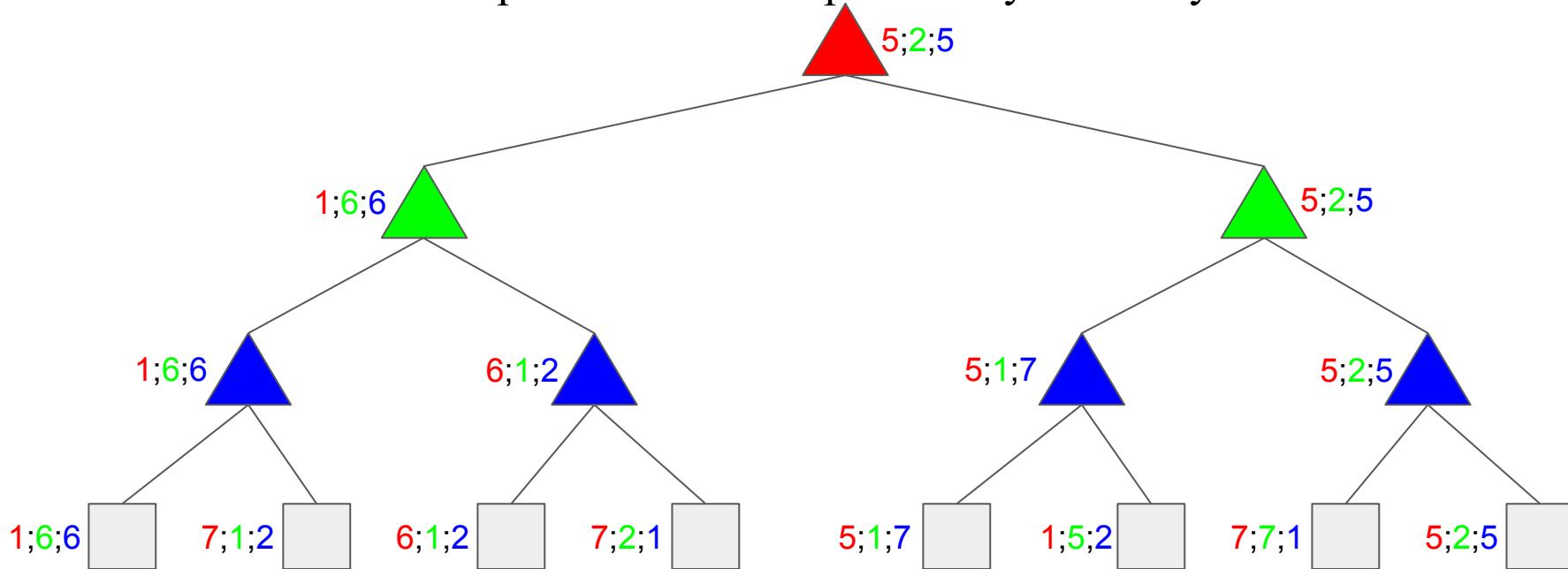


# Multi-Agent Utilities

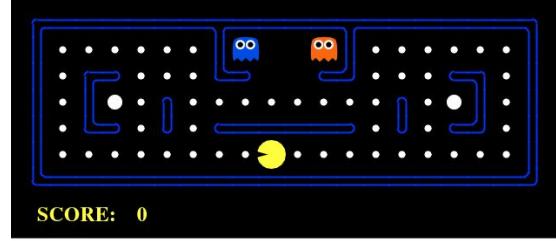


# Multi-Agent Utilities

- Generalisation of minimax
  - Terminals and nodes have utility vectors
  - Each player maximizes its own component
  - Gives rise to cooperation and competition dynamically



<http://ai.berkeley.edu>



# Assignment 1