

C1

Search

COMP3270
Introduction to Artificial Intelligence

Chapter 1: Search Roadmap

Chapter 1: Search Roadmap

Define the Problem

Abstraction, states, actions, goals, and costs

Chapter 1: Search Roadmap

Define the Problem

Abstraction, states, actions, goals, and costs

Find a Solution

Uninformed, informed, local, and constraint-based search

Chapter 1: Search Roadmap

Define the Problem

Abstraction, states, actions, goals, and costs

Find a Solution

Uninformed, informed, local, and constraint-based search

Compete Against an Opponent

Minimax, pruning, and uncertainty

Search Looks Ahead: Which Actions Reach the Goal?

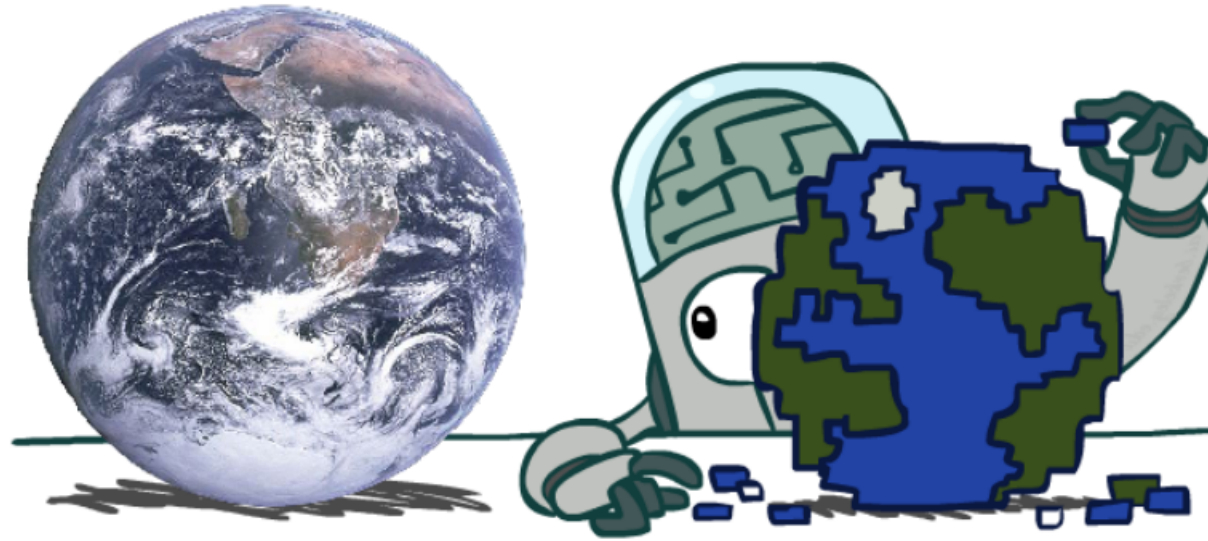


To Look Ahead, Build a Simpler Model

Real World



Search Model

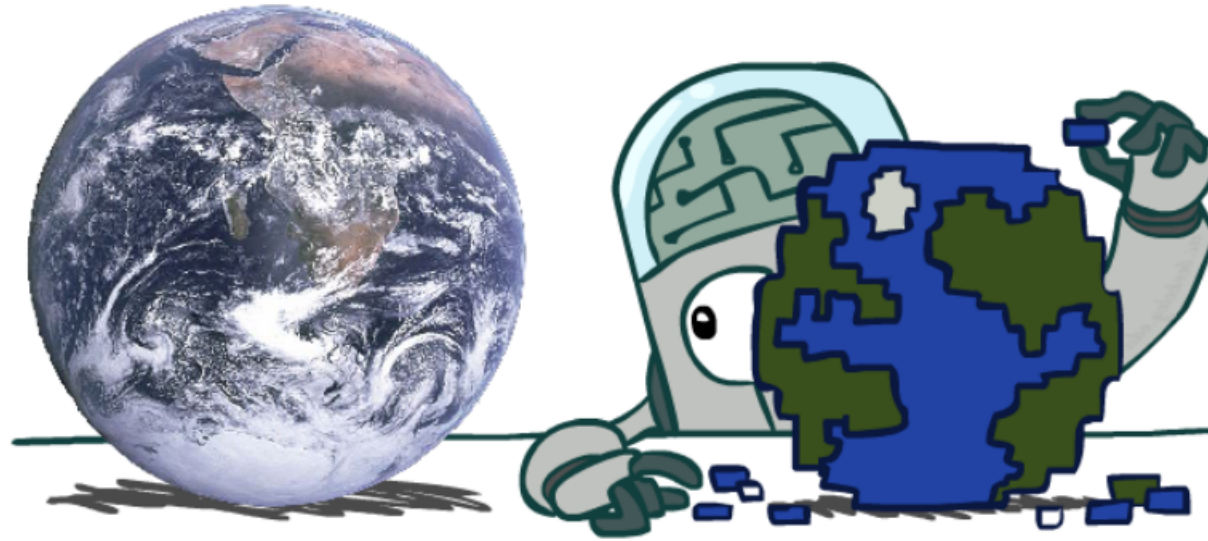


To Look Ahead, Build a Simpler Model

Real World



Search Model



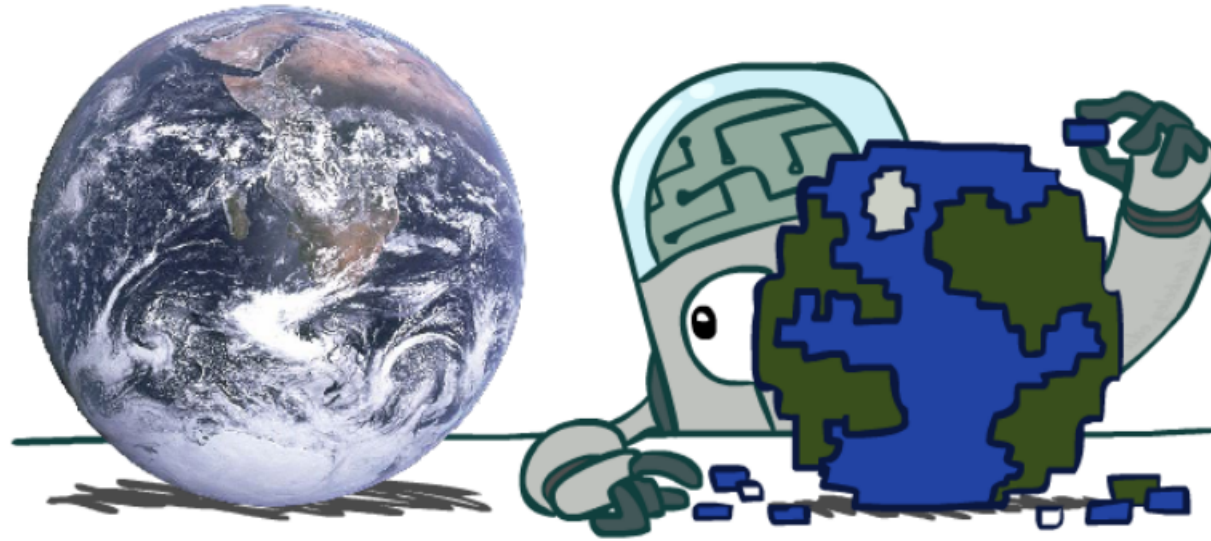
Abstraction keeps only what the decision needs

To Look Ahead, Build a Simpler Model

Real World



Search Model



Abstraction keeps only what the decision needs

States · Actions · Goal · Costs

Choose the Search Strategy to Match the Problem

Choose the Search Strategy to Match the Problem

No Guidance → Uninformed Search

Choose the Search Strategy to Match the Problem

No Guidance → Uninformed Search

A Helpful Estimate → Informed Search

Choose the Search Strategy to Match the Problem

No Guidance → Uninformed Search

A Helpful Estimate → Informed Search

Improve One Current State → Local Search

Choose the Search Strategy to Match the Problem

No Guidance → Uninformed Search

A Helpful Estimate → Informed Search

Improve One Current State → Local Search

Hard Constraints → Constraint Satisfaction

Choose the Search Strategy to Match the Problem

No Guidance → Uninformed Search

A Helpful Estimate → Informed Search

Improve One Current State → Local Search

Hard Constraints → Constraint Satisfaction

An Opponent → Adversarial Search

Defining a Search Problem

Defining a Search Problem

State Representation What information describes a situation?

Defining a Search Problem

State Representation What information describes a situation?

Initial State Where does the agent begin?

Defining a Search Problem

State Representation	What information describes a situation?
Initial State	Where does the agent begin?
Actions and Transition Model	What can the agent do, and what changes?

Defining a Search Problem

State Representation	What information describes a situation?
Initial State	Where does the agent begin?
Actions and Transition Model	What can the agent do, and what changes?
Goal Test	How do we recognise success?

Defining a Search Problem

State Representation	What information describes a situation?
Initial State	Where does the agent begin?
Actions and Transition Model	What can the agent do, and what changes?
Goal Test	How do we recognise success?
Path Cost	How do we compare solutions?

Defining a Search Problem

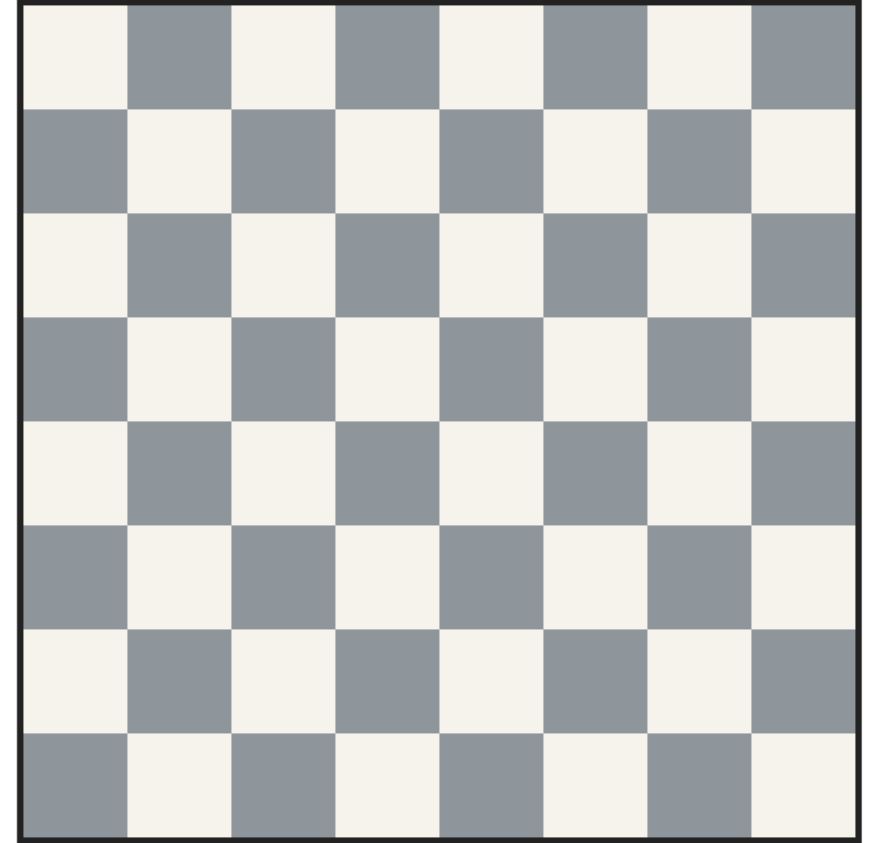
State Representation	What information describes a situation?
Initial State	Where does the agent begin?
Actions and Transition Model	What can the agent do, and what changes?
Goal Test	How do we recognise success?
Path Cost	How do we compare solutions?

A solution is a sequence of actions from the initial state to a goal state

N-Queens Definition A: Place Anywhere, Check at the End

Start

Empty board



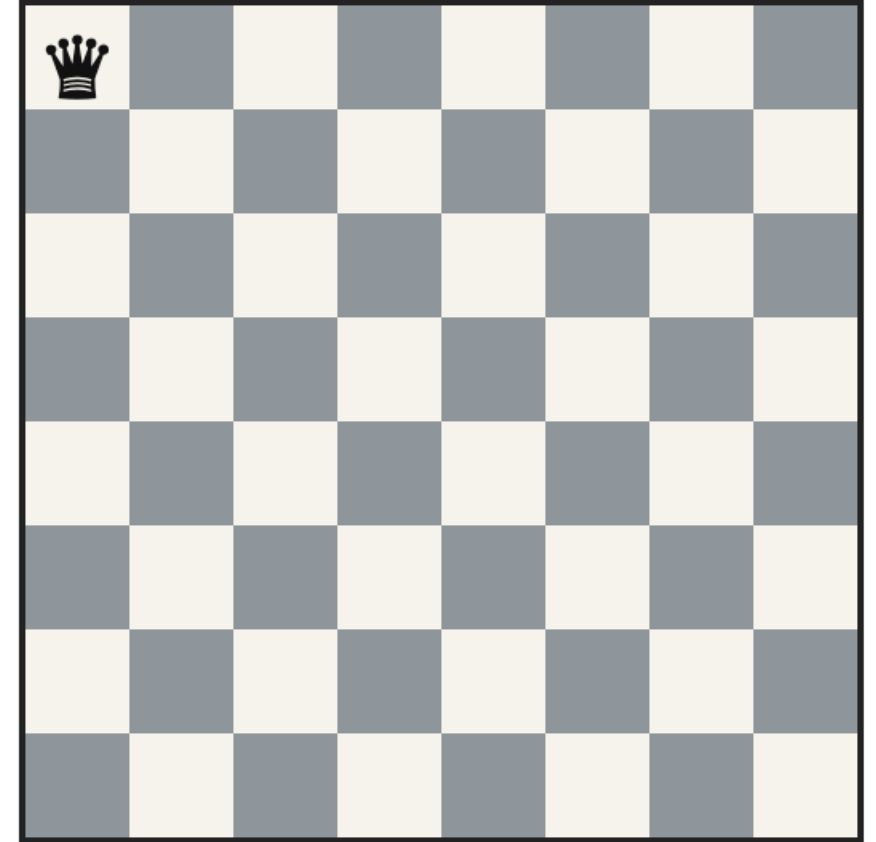
N-Queens Definition A: Place Anywhere, Check at the End

Start

Empty board

Action

Add a queen to any empty square



N-Queens Definition A: Place Anywhere, Check at the End

Start

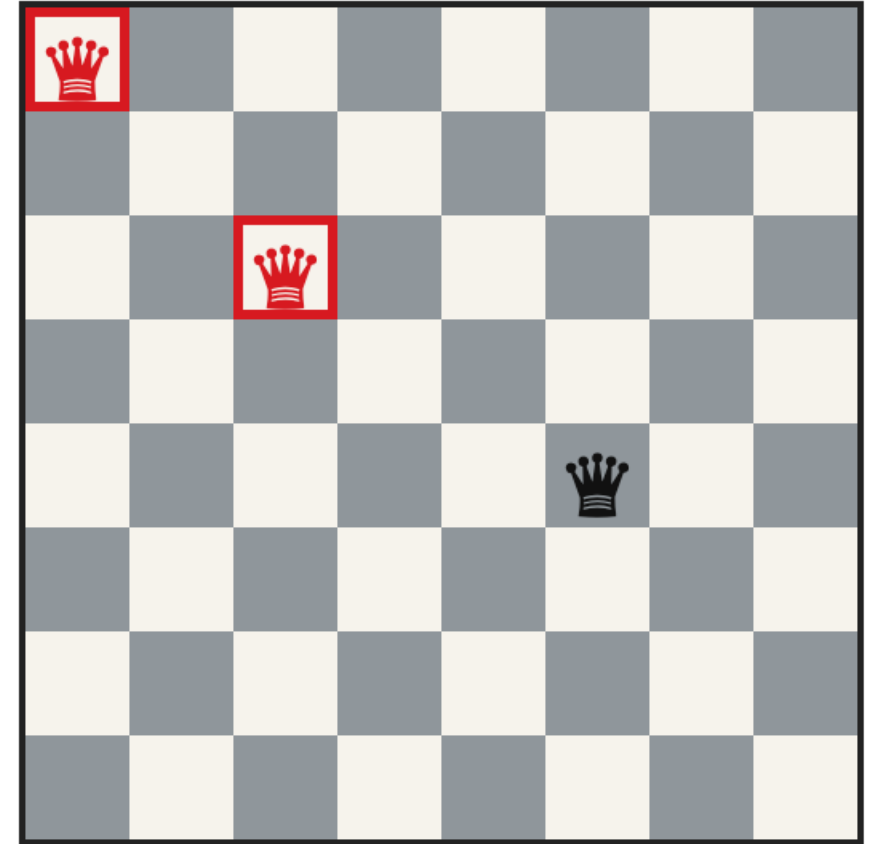
Empty board

Action

Add a queen to any empty square

State

Any arrangement of 0–n queens



N-Queens Definition A: Place Anywhere, Check at the End

Start

Empty board

Action

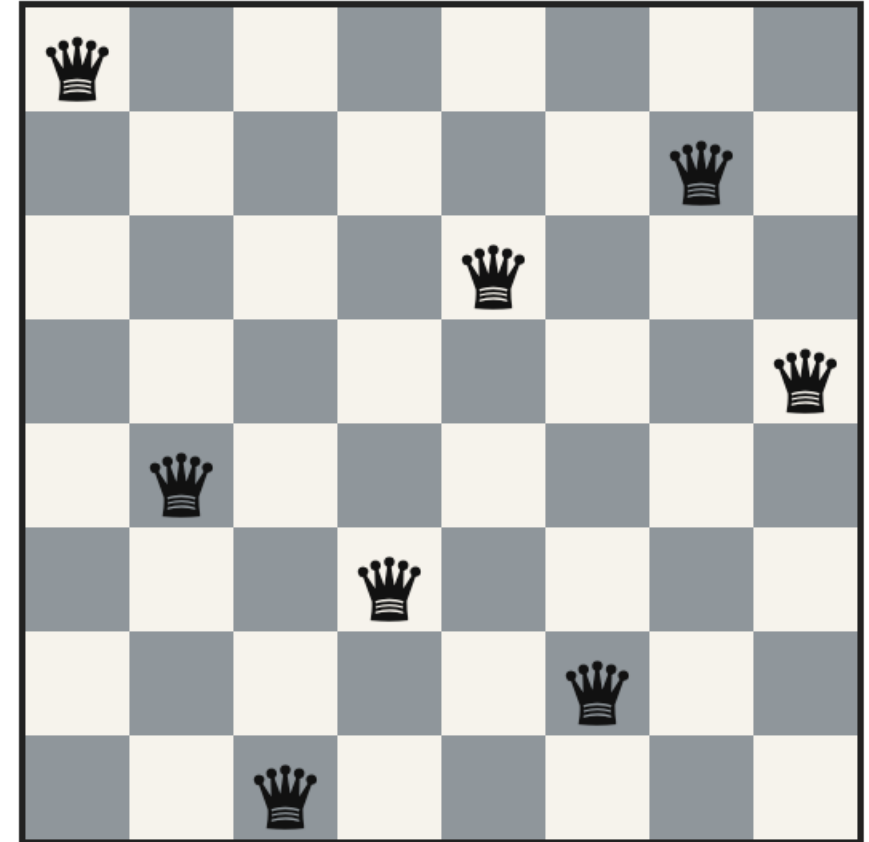
Add a queen to any empty square

State

Any arrangement of 0– n queens

Goal

n queens, none attacking



N-Queens Definition A: Place Anywhere, Check at the End

Start

Empty board

Action

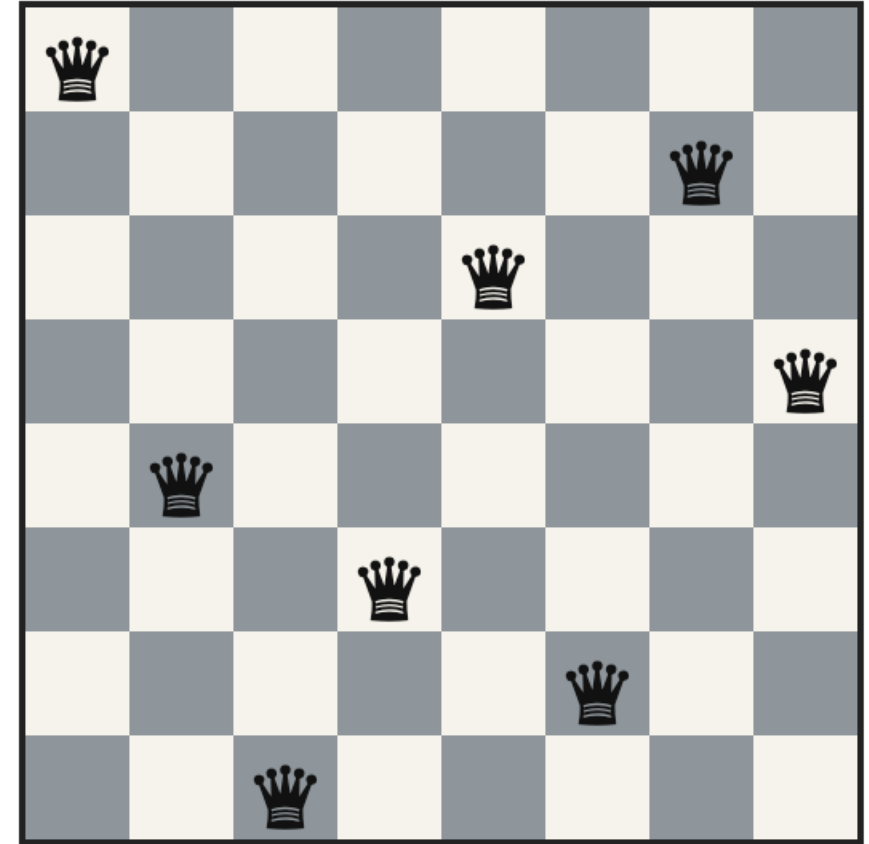
Add a queen to any empty square

State

Any arrangement of 0– n queens

Goal

n queens, none attacking

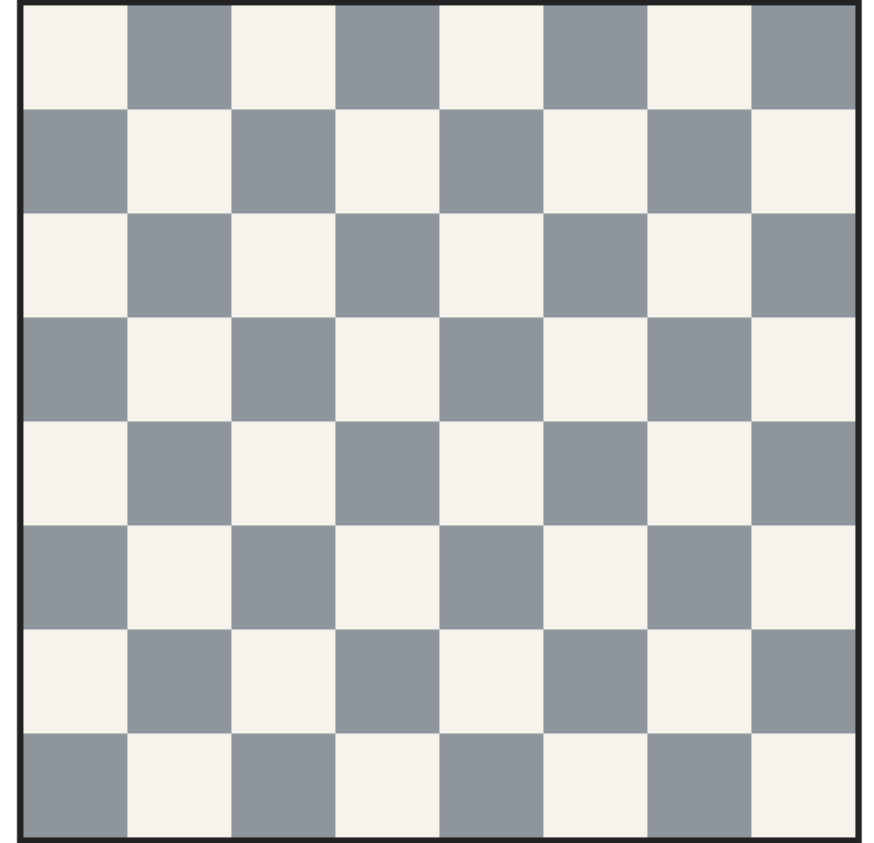


Attacking partial boards are still states

N-Queens Definition B: Rule Out Bad States Early

Start

Empty board



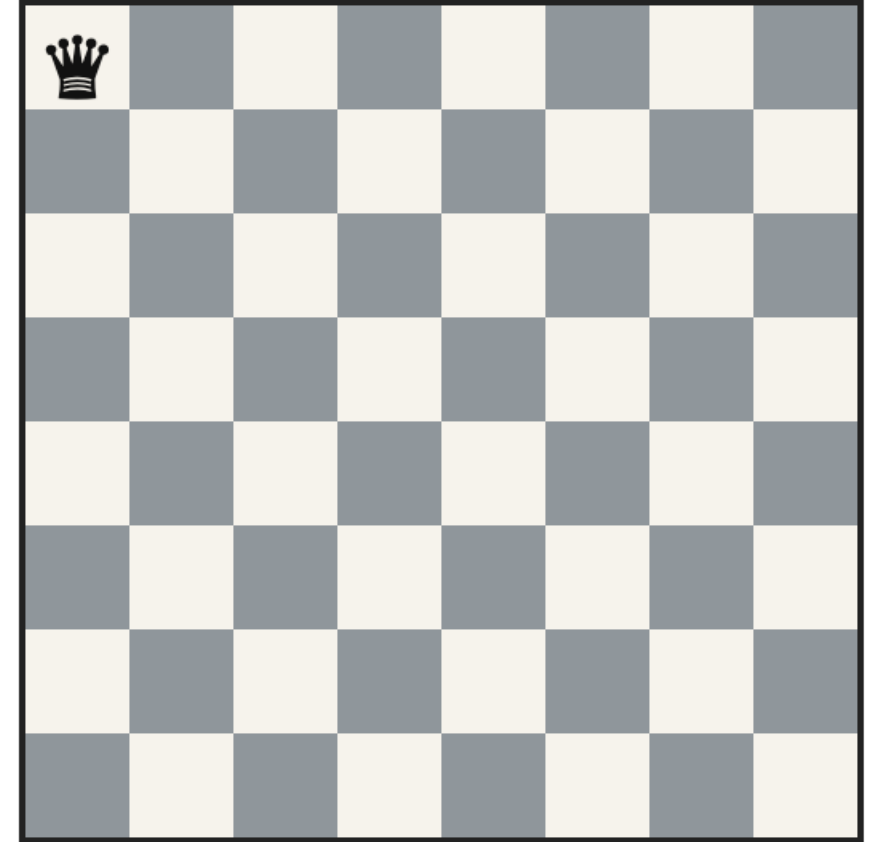
N-Queens Definition B: Rule Out Bad States Early

Start

Empty board

Action

Add one non-attacking queen to an empty column



N-Queens Definition B: Rule Out Bad States Early

Start

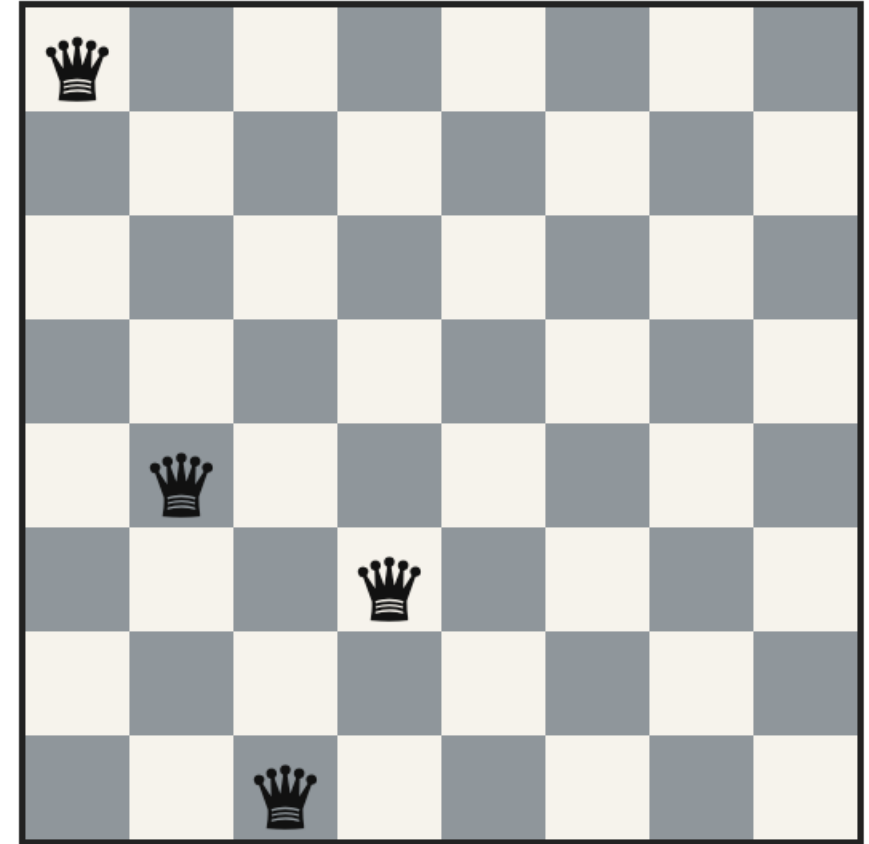
Empty board

Action

Add one non-attacking queen to an empty column

State

One queen per column, none attacking



N-Queens Definition B: Rule Out Bad States Early

Start

Empty board

Action

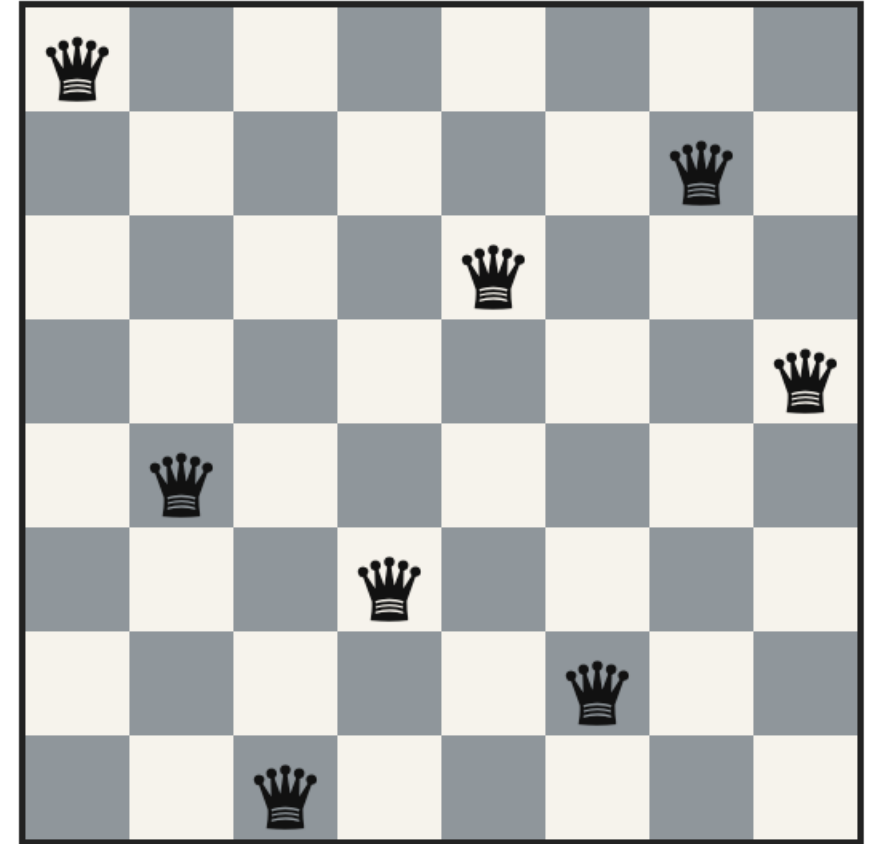
Add one non-attacking queen to an empty column

State

One queen per column, none attacking

Goal

n queens on the board



N-Queens Definition B: Rule Out Bad States Early

Start

Empty board

Action

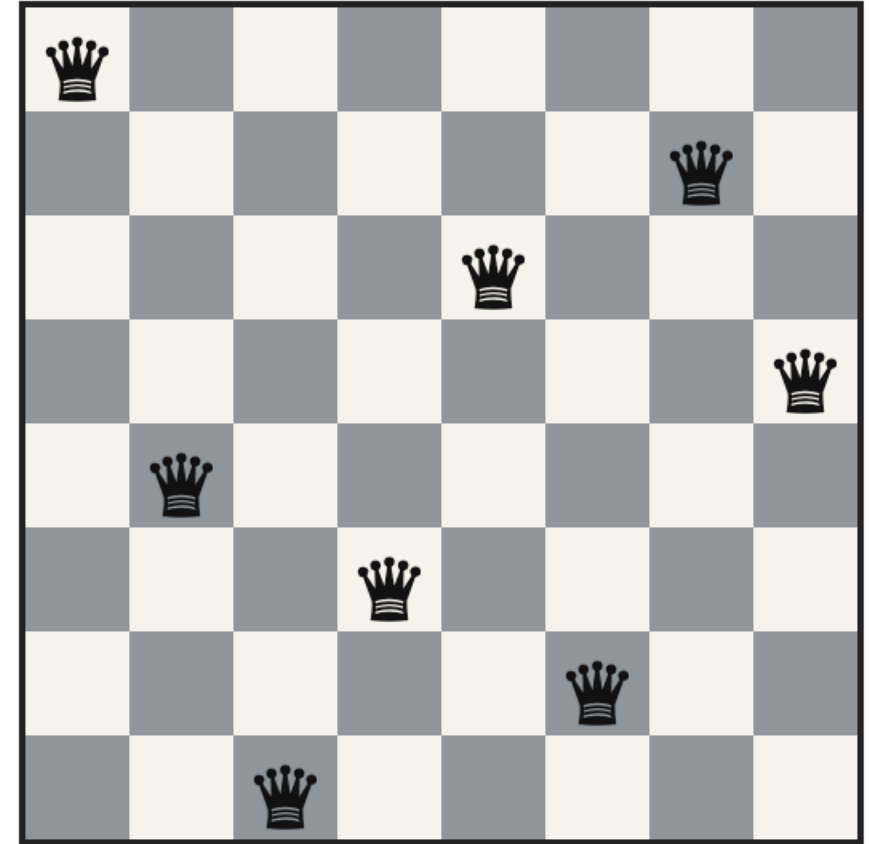
Add one non-attacking queen to an empty column

State

One queen per column, none attacking

Goal

n queens on the board



Invalid partial boards never enter the state space

A State Space Contains Every Reachable State

Different problem definitions produce different state spaces

A State Space Contains Every Reachable State

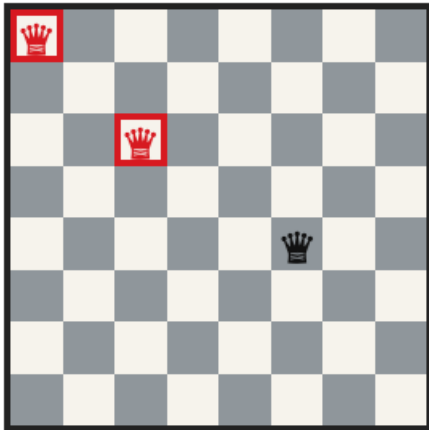
Different problem definitions produce different state spaces

TASK

Which definition has the larger state space?

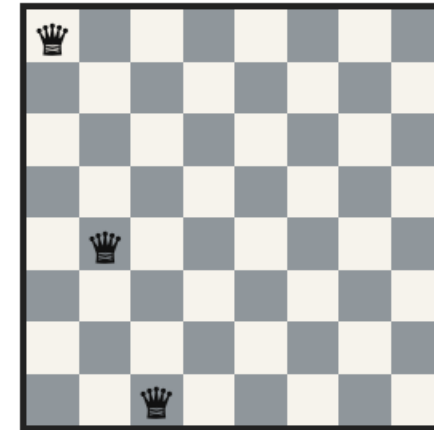
Definition A

Place anywhere. Check at the end.



Definition B

Rule out bad states early.



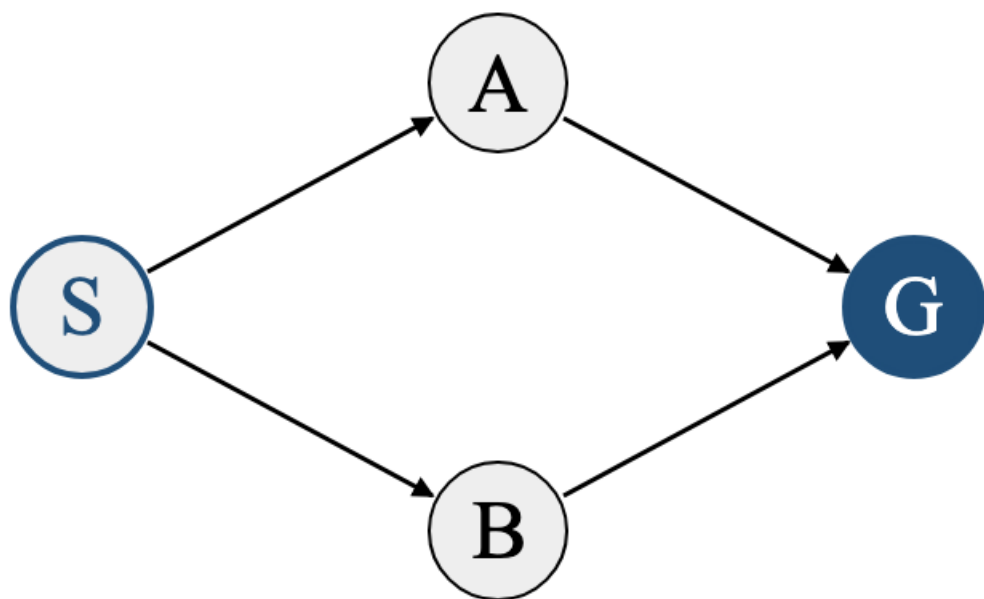
A Search Tree Records Action Sequences

State Space Graph

Each distinct state appears once

Search Tree

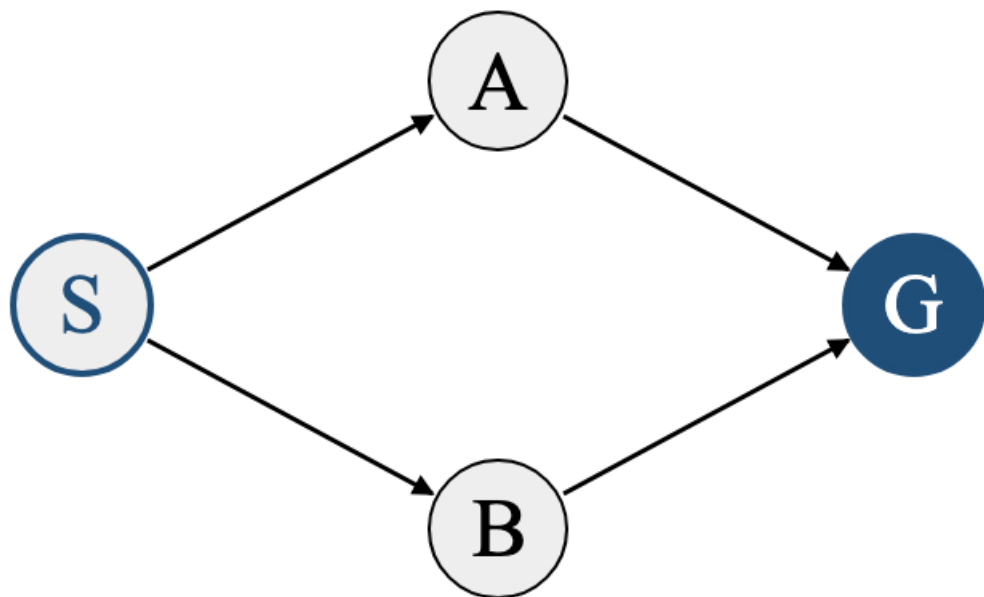
Each action sequence creates a tree node



A Search Tree Records Action Sequences

State Space Graph

Each distinct state appears once



Search Tree

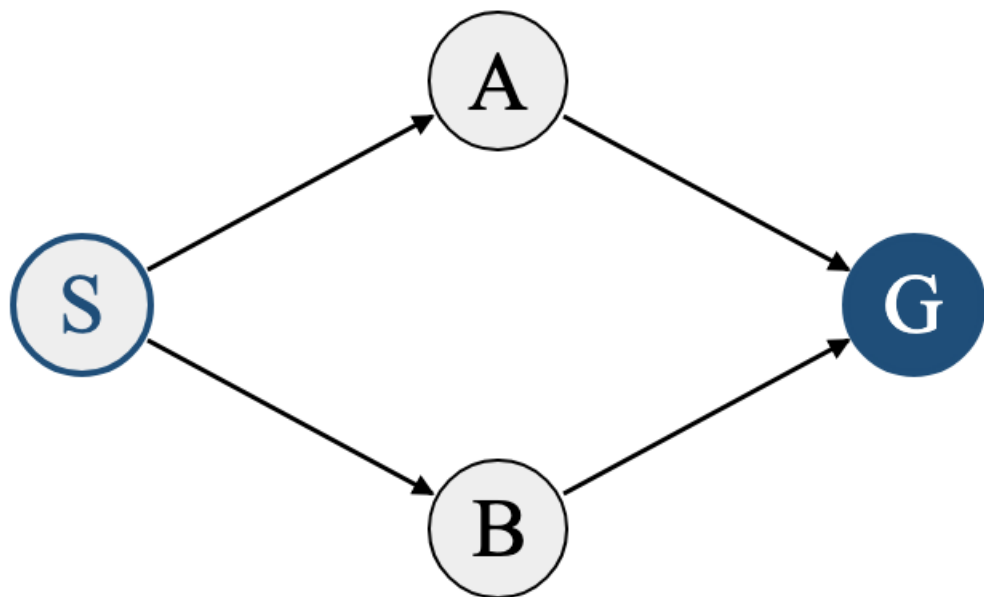
Each action sequence creates a tree node



A Search Tree Records Action Sequences

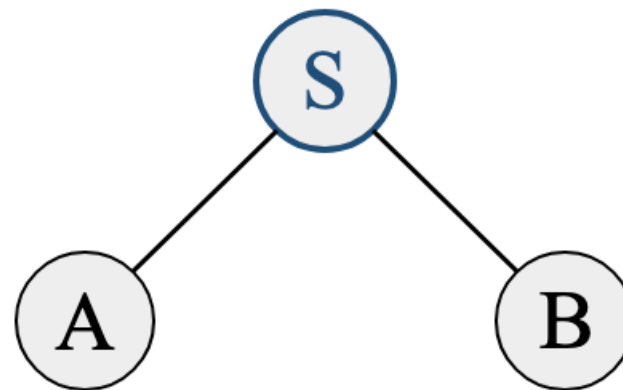
State Space Graph

Each distinct state appears once



Search Tree

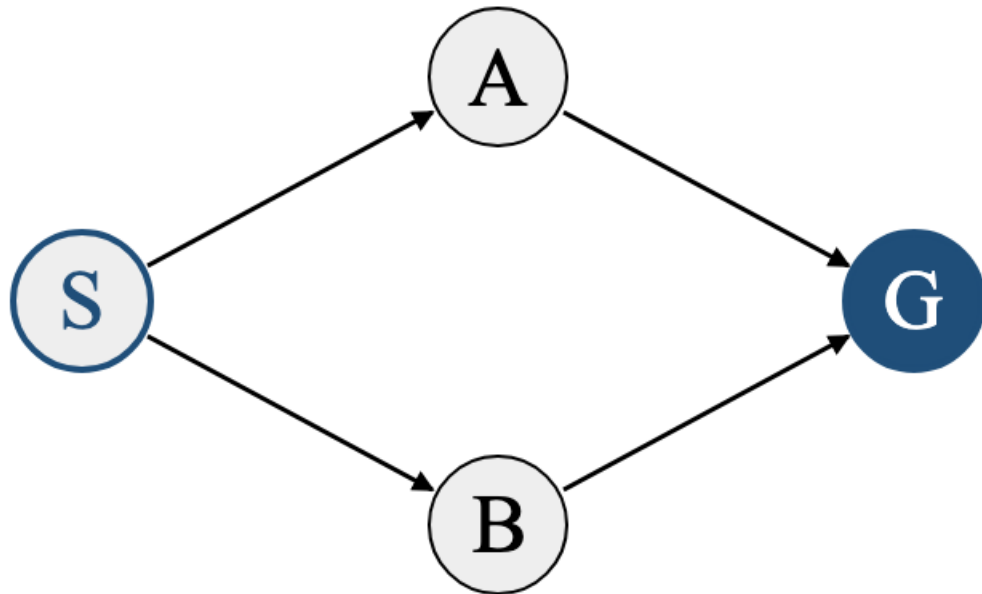
Each action sequence creates a tree node



A Search Tree Records Action Sequences

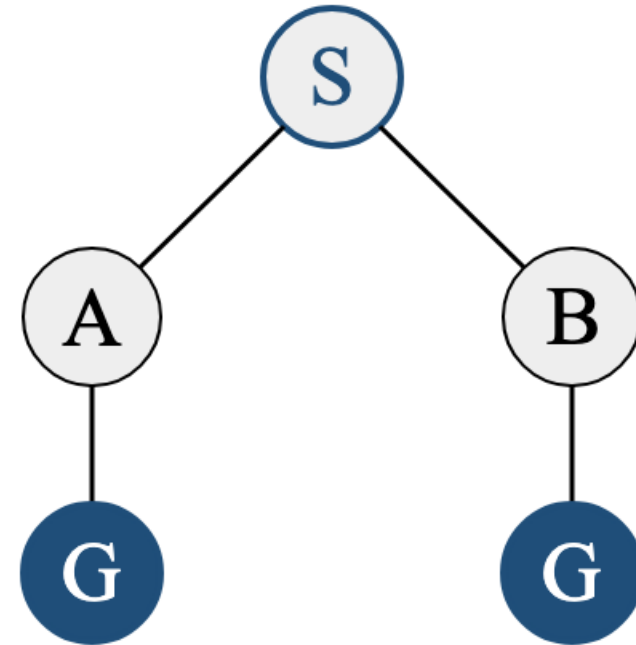
State Space Graph

Each distinct state appears once



Search Tree

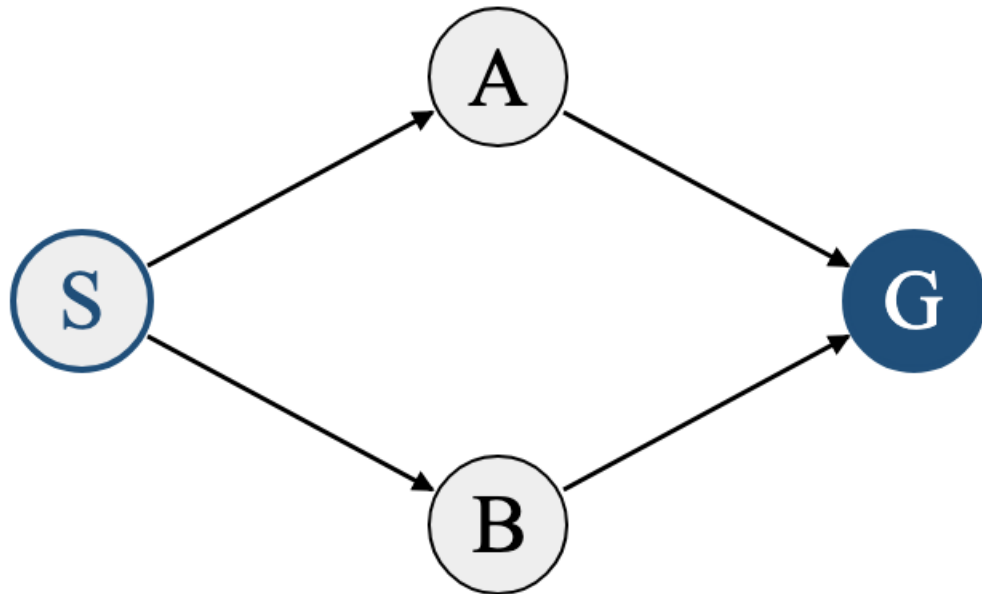
Each action sequence creates a tree node



A Search Tree Records Action Sequences

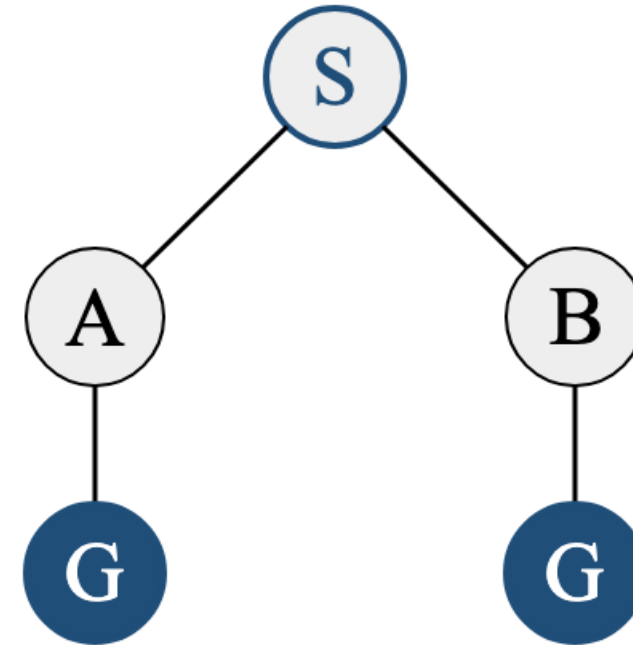
State Space Graph

Each distinct state appears once



Search Tree

Each action sequence creates a tree node



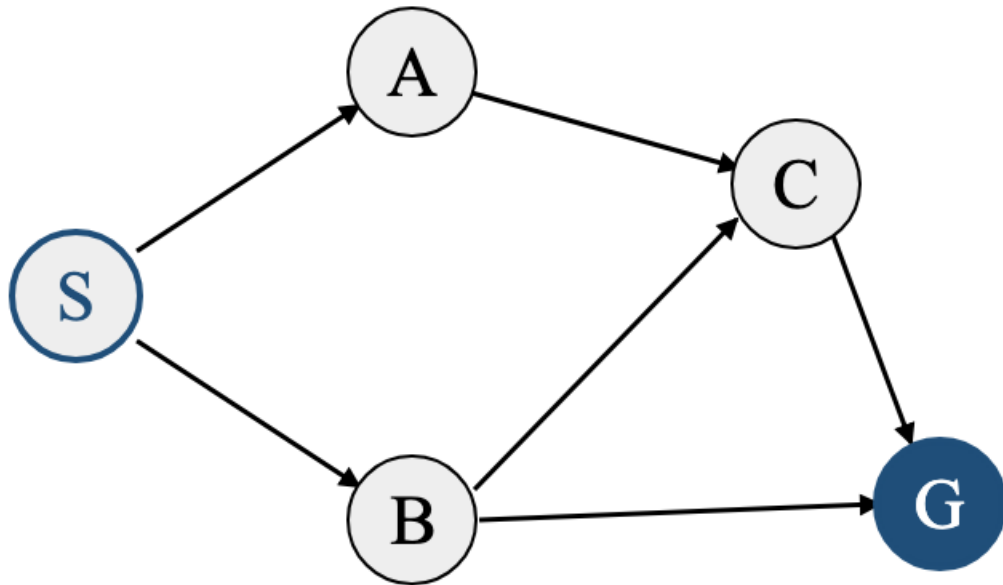
Different paths to G create different search-tree nodes

Draw the Search Tree

Starting at S, draw the complete search tree. How many times does G appear?

TASK

State Space Graph

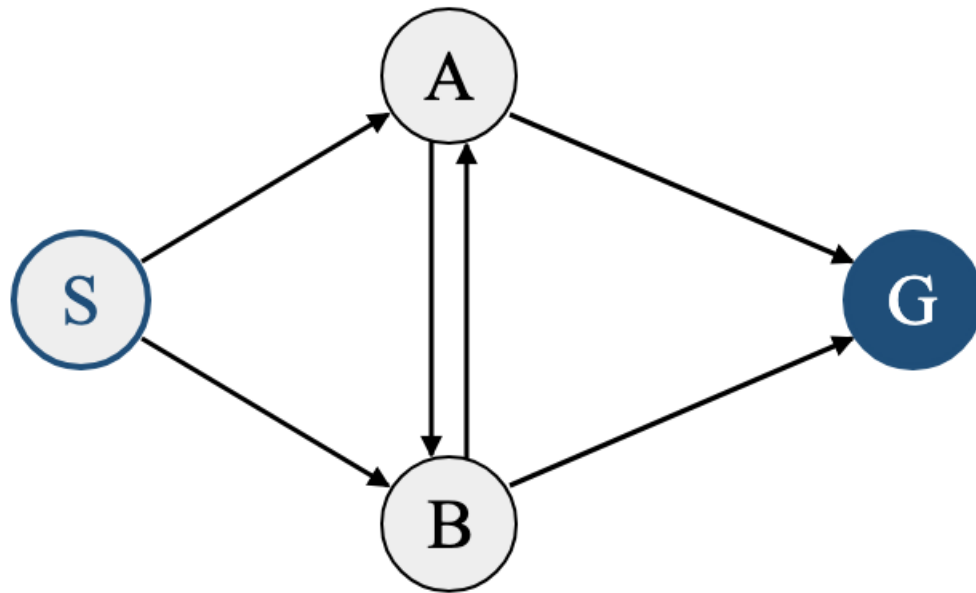


Trace the Search Tree

How many times does G appear in the complete search tree?

TASK

Cyclic State Space Graph



Define the Romania Route Problem

Find a route from Arad to Bucharest

TASK

State

Start

Actions and
Transition

Goal

Path Cost



Define the Romania Route Problem

Find a route from Arad to Bucharest

TASK

State

Start

Actions and
Transition

Goal

Path Cost



Define the Romania Route Problem

Find a route from Arad to Bucharest

TASK

State

Start

Actions and
Transition

Goal

Path Cost



Define the Romania Route Problem

Find a route from Arad to Bucharest

State

Start

Actions and
Transition

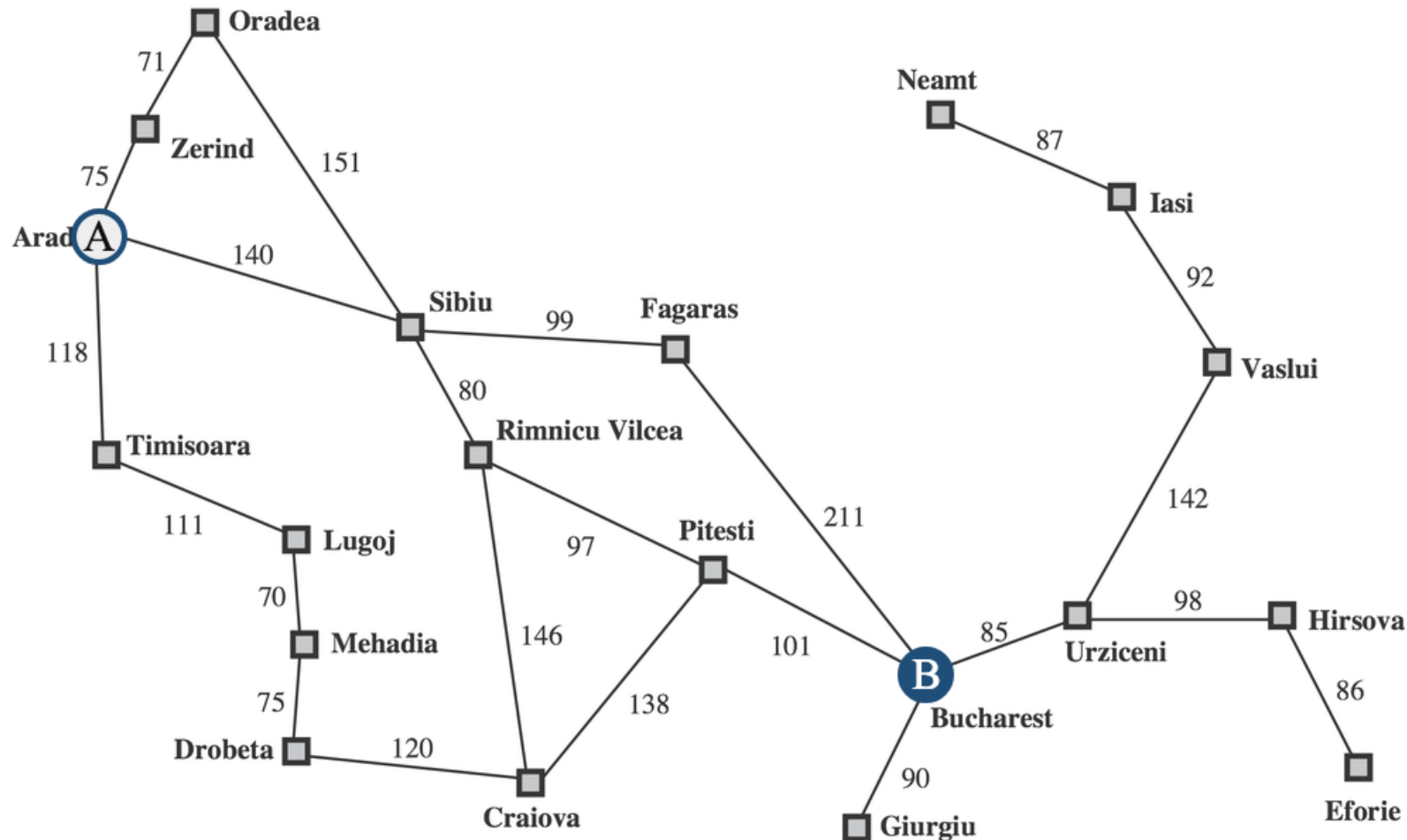
Goal

Path Cost



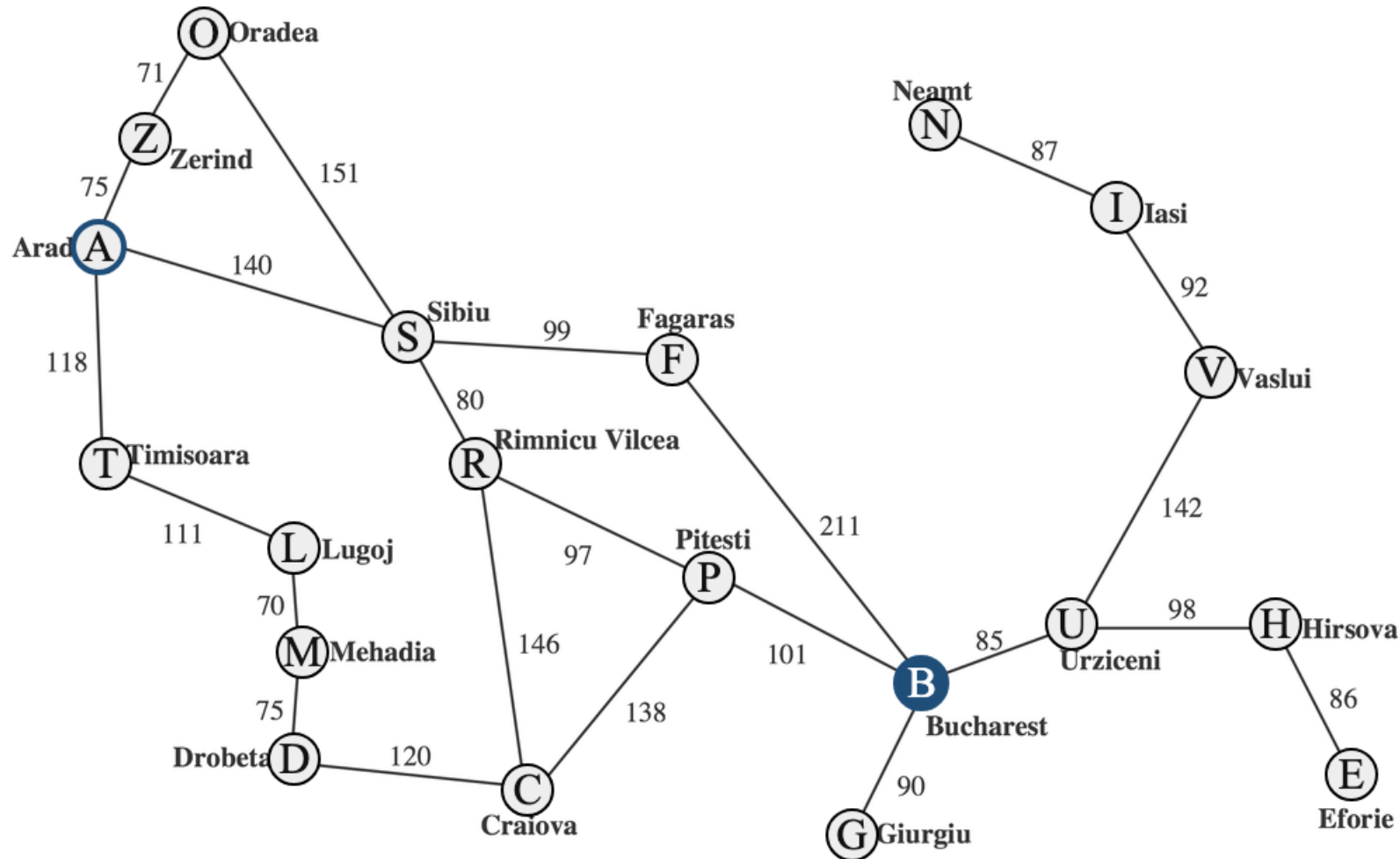
Romania as a State-Space Graph

Cities are states · roads are transitions · distances are step costs (ignored for now)

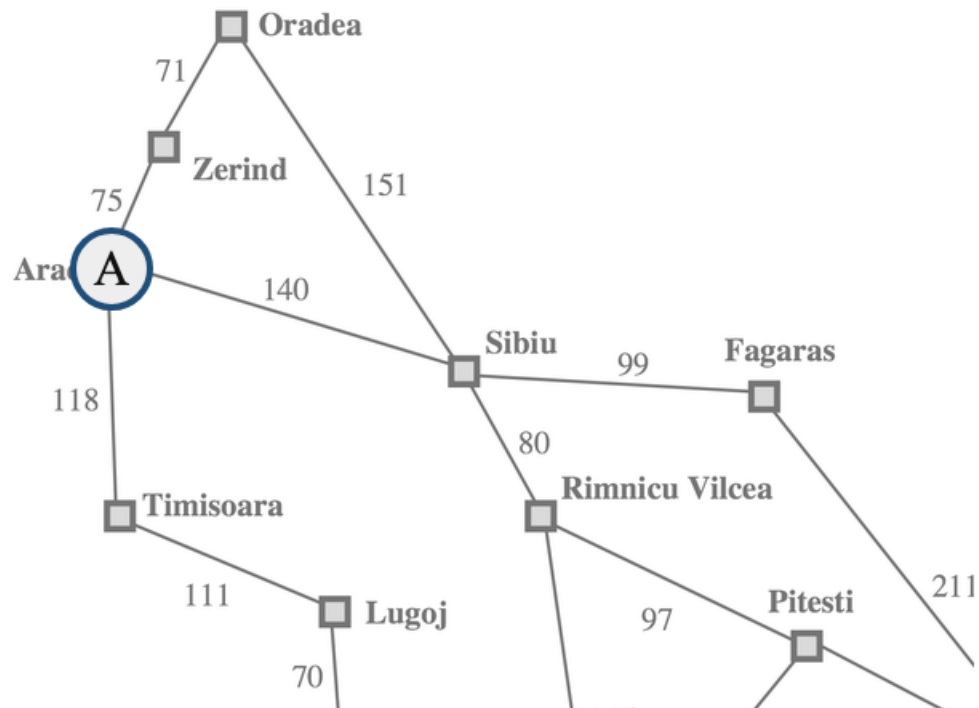


Romania as a State-Space Graph

Cities are states · roads are transitions · distances are step costs (ignored for now)



Expanding A Generates Three Successors

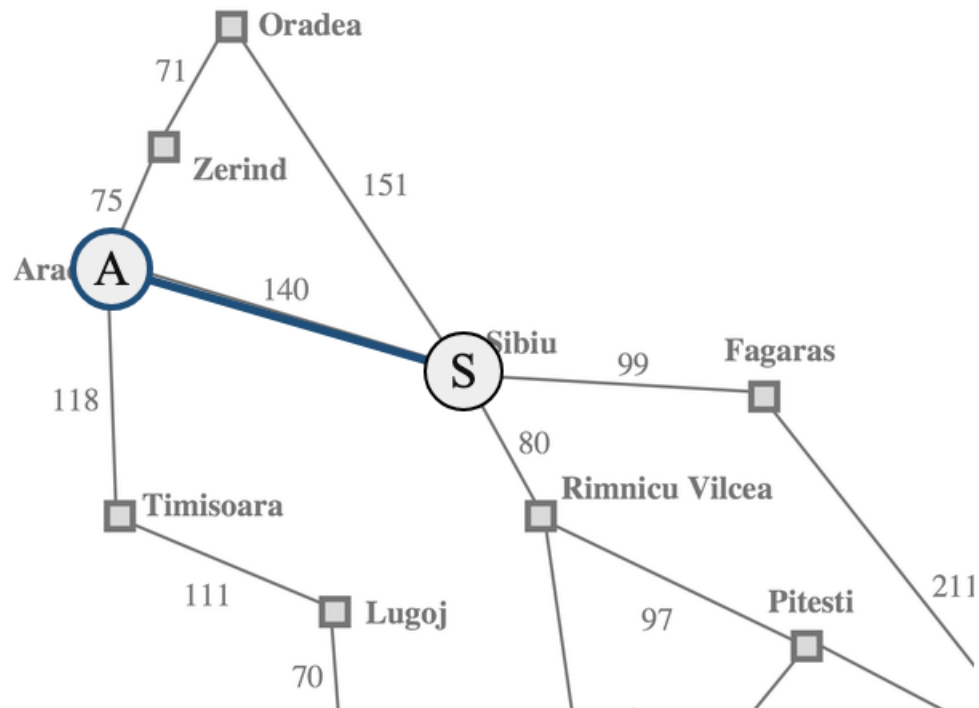


PY main.py

```
1- 1. romania = {
2     'A': ['S', 'T', 'Z'],
3     'Z': ['A', 'O'], 'O': ['S', 'Z'],
4     'T': ['A', 'L'], 'L': ['M', 'T'],
5     'M': ['D', 'L'], 'D': ['C', 'M'],
6     'S': ['A', 'F', 'O', 'R'],
7     'R': ['C', 'P', 'S'],
8     'C': ['D', 'P', 'R'],
9     'F': ['B', 'S'],
10    'P': ['B', 'C', 'R'], 'B': []
11 }
12 print(romania['A'])
13
```

Download: comp3270.dirk.hk/roads_from_arad.py

Expanding A Generates Three Successors

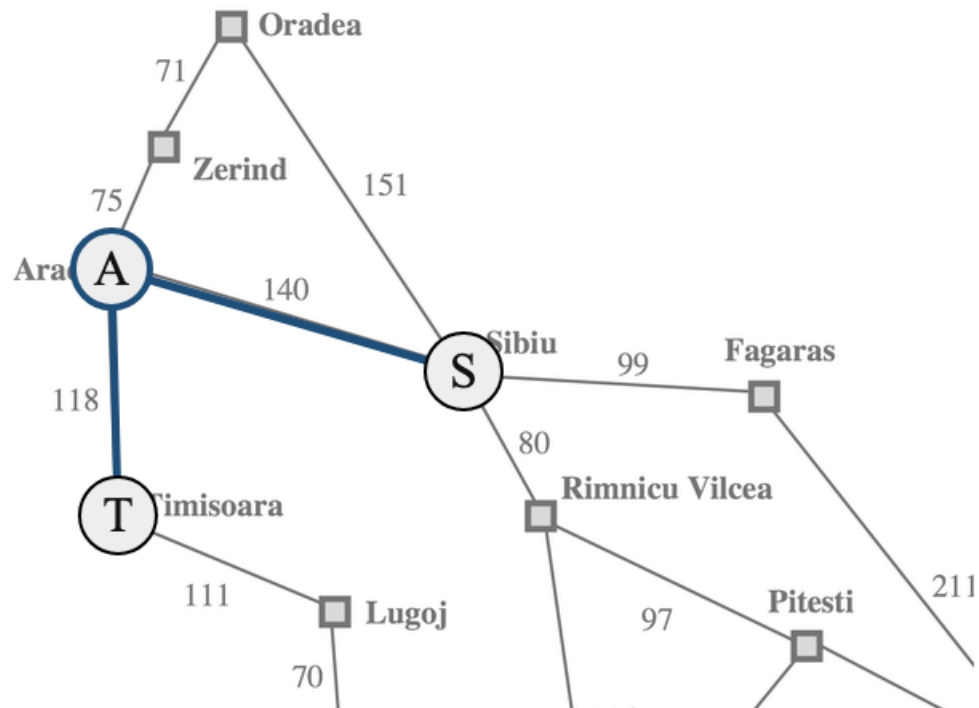


PY main.py

```
1- 1- romania = {
2     'A': ['S', 'T', 'Z'],
3     'Z': ['A', 'O'], 'O': ['S', 'Z'],
4     'T': ['A', 'L'], 'L': ['M', 'T'],
5     'M': ['D', 'L'], 'D': ['C', 'M'],
6     'S': ['A', 'F', 'O', 'R'],
7     'R': ['C', 'P', 'S'],
8     'C': ['D', 'P', 'R'],
9     'F': ['B', 'S'],
10    'P': ['B', 'C', 'R'], 'B': []
11 }
12 print(romania['A'])
13
```

Download: comp3270.dirk.hk/roads_from_arad.py

Expanding A Generates Three Successors

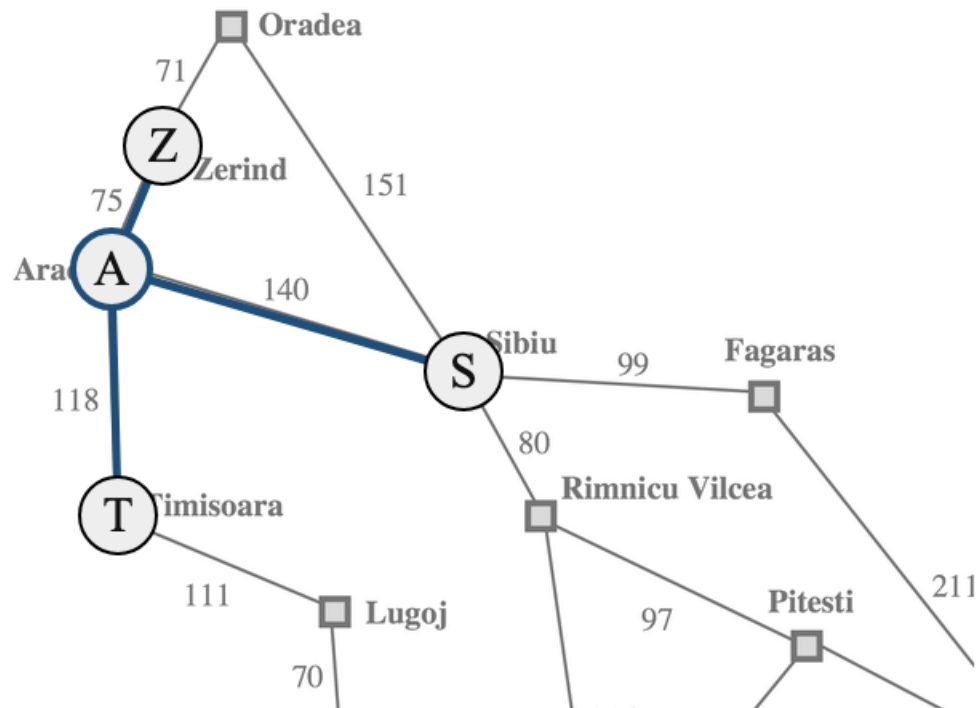


PY main.py

```
1- 1- romania = {
2     'A': ['S', 'T', 'Z'],
3     'Z': ['A', 'O'], 'O': ['S', 'Z]
4     'T': ['A', 'L'], 'L': ['M', 'T]
5     'M': ['D', 'L'], 'D': ['C', 'M]
6     'S': ['A', 'F', 'O', 'R'],
7     'R': ['C', 'P', 'S'],
8     'C': ['D', 'P', 'R'],
9     'F': ['B', 'S'],
10    'P': ['B', 'C', 'R'], 'B': []
11 }
12 print(romania['A'])
13
```

Download: comp3270.dirk.hk/roads_from_arad.py

Expanding A Generates Three Successors



PY main.py

```
1- romania = {
2     'A': ['S', 'T', 'Z'],
3     'Z': ['A', 'O'], 'O': ['S', 'Z]
4     'T': ['A', 'L'], 'L': ['M', 'T]
5     'M': ['D', 'L'], 'D': ['C', 'M]
6     'S': ['A', 'F', 'O', 'R'],
7     'R': ['C', 'P', 'S'],
8     'C': ['D', 'P', 'R'],
9     'F': ['B', 'S'],
10    'P': ['B', 'C', 'R'], 'B': []
11 }
12 print(romania['A'])
13
```

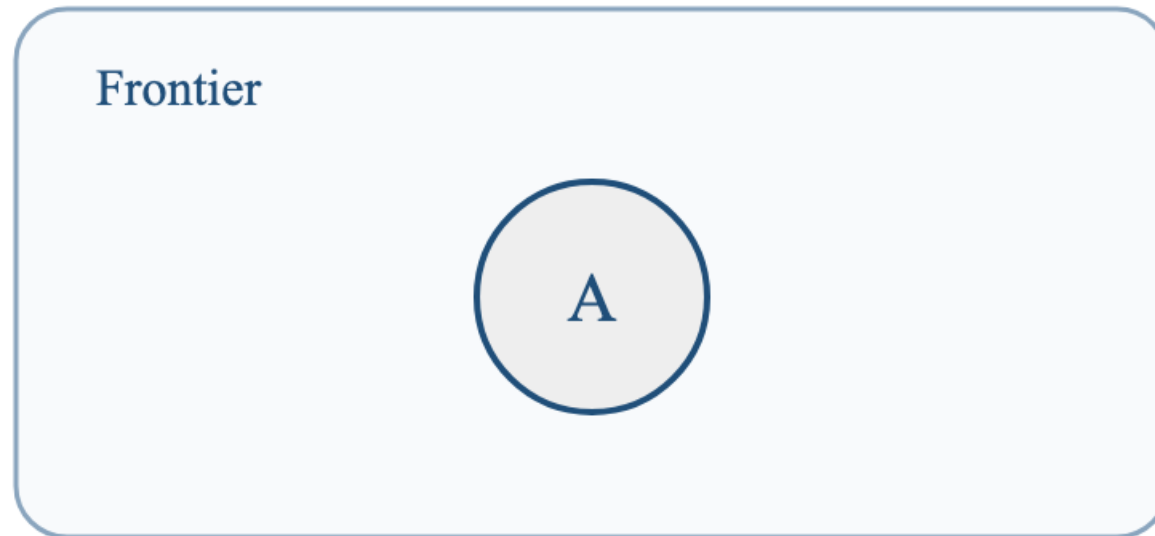
Download: comp3270.dirk.hk/roads_from_arad.py

The Frontier Stores Nodes Waiting to Be Expanded

Frontier = generated nodes waiting to be expanded.

It is the search algorithm's current set of choices.

Initially: one generated node



The Frontier Stores Nodes Waiting to Be Expanded

Frontier = generated nodes waiting to be expanded.

It is the search algorithm's current set of choices.

Frontier after selection

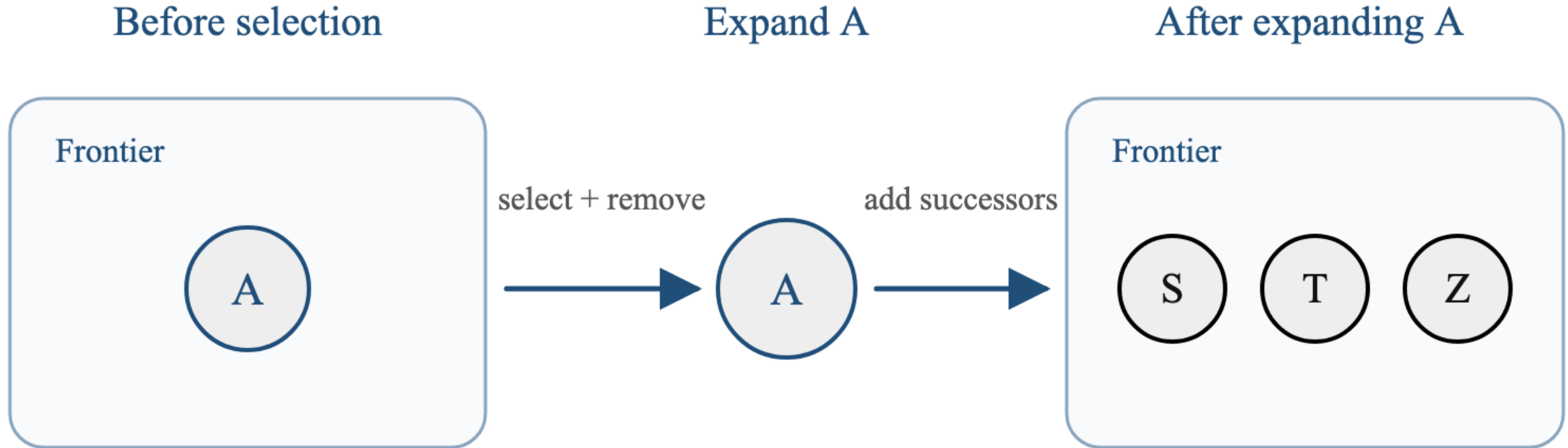
Expand A



The Frontier Stores Nodes Waiting to Be Expanded

Frontier = generated nodes waiting to be expanded.

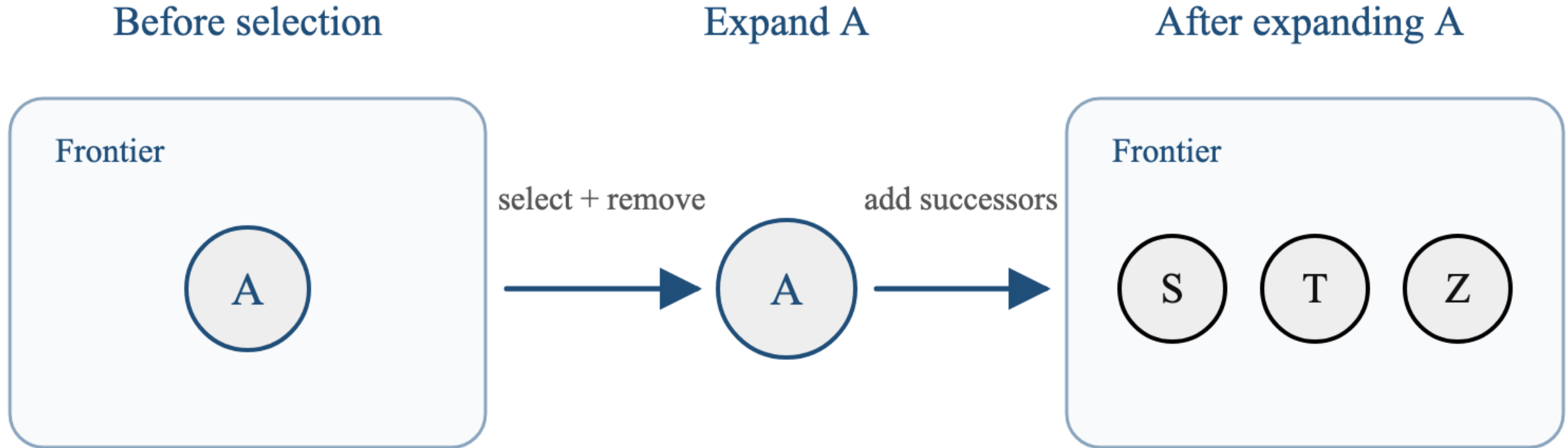
It is the search algorithm's current set of choices.



The Frontier Stores Nodes Waiting to Be Expanded

Frontier = generated nodes waiting to be expanded.

It is the search algorithm's current set of choices.



Which node should be expanded next: S, T, or Z?

A Search Strategy Chooses the Next Node to Expand

It is the rule that determines the order of node expansion.



A Search Strategy Chooses the Next Node to Expand

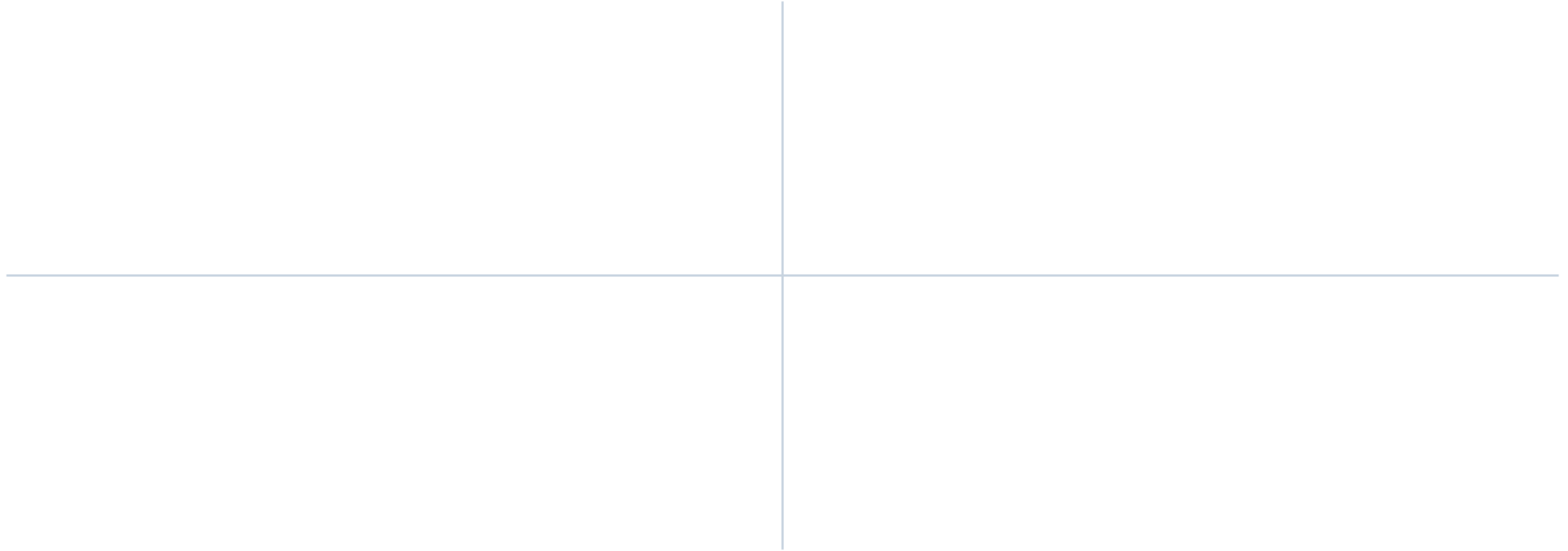
It is the rule that determines the order of node expansion.



How should we compare search strategies?

Four Questions Compare Search Strategies

We compare what a strategy guarantees and what resources it requires.



Four Questions Compare Search Strategies

We compare what a strategy guarantees and what resources it requires.

Completeness

Will it find a solution whenever one
exists?

Four Questions Compare Search Strategies

We compare what a strategy guarantees and what resources it requires.

Completeness

Will it find a solution whenever one exists?

Optimality

Will it find a least-cost solution?

Four Questions Compare Search Strategies

We compare what a strategy guarantees and what resources it requires.

Completeness

Will it find a solution whenever one exists?

Optimality

Will it find a least-cost solution?

Time

How many nodes are generated?

Four Questions Compare Search Strategies

We compare what a strategy guarantees and what resources it requires.

Completeness

Will it find a solution whenever one exists?

Optimality

Will it find a least-cost solution?

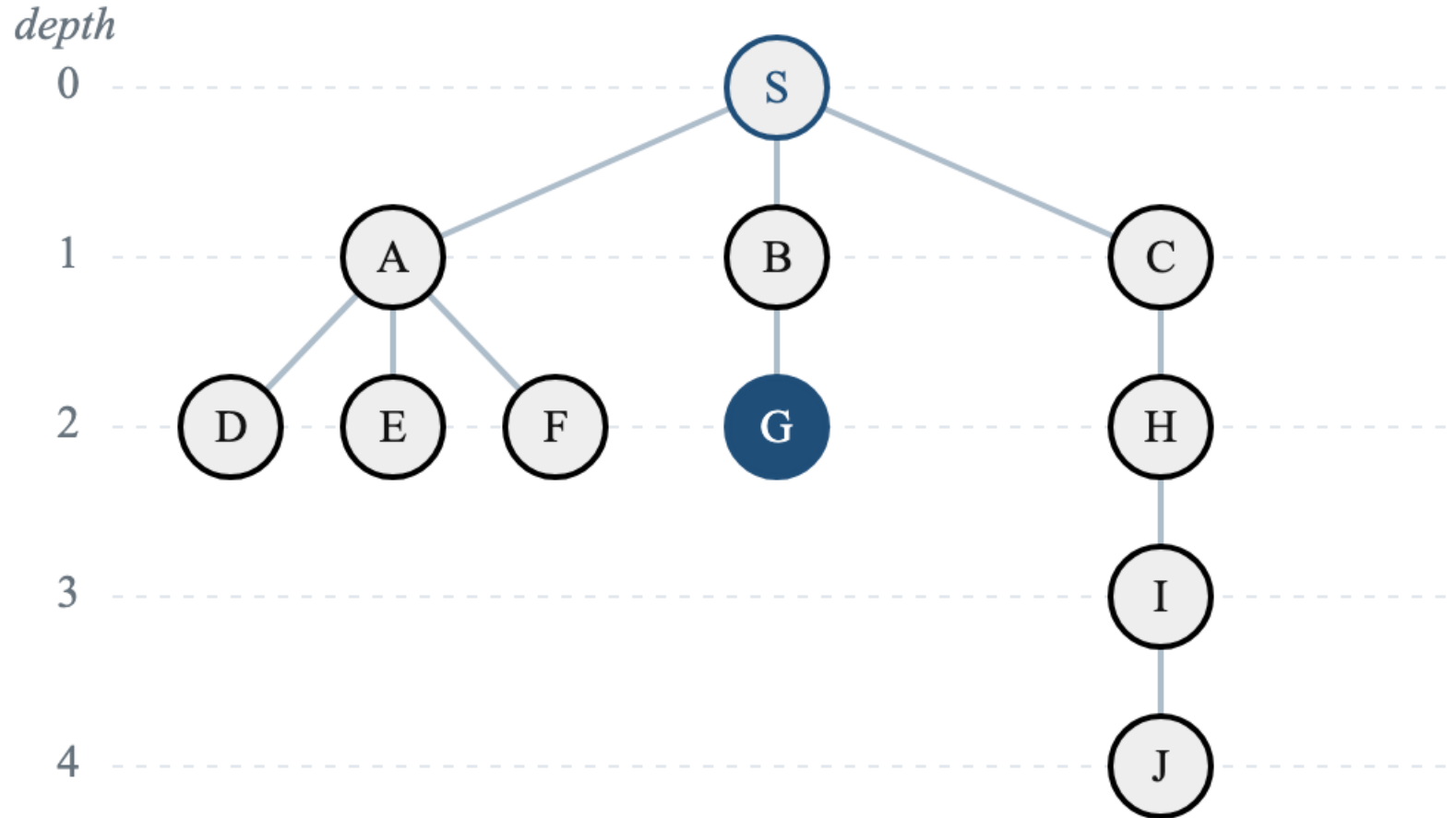
Time

How many nodes are generated?

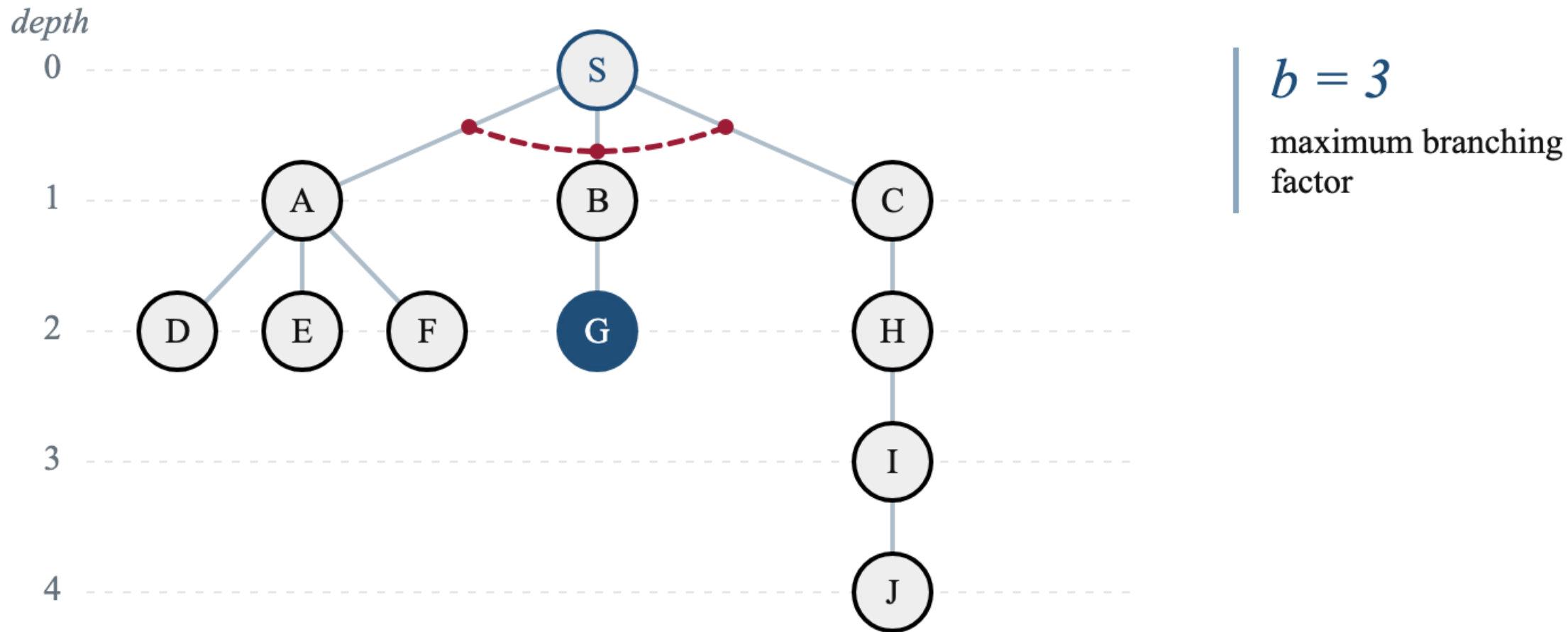
Space

What is the maximum number of nodes held in memory?

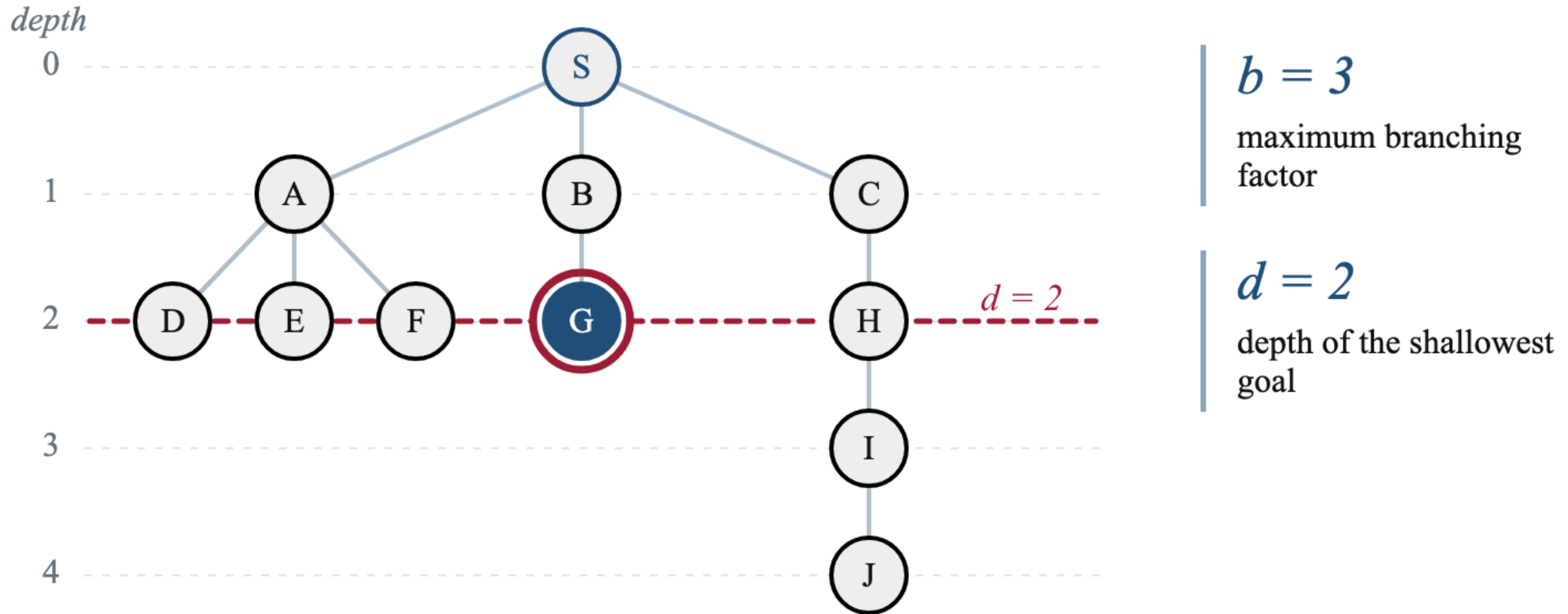
Search Complexity Depends on b , d , and m



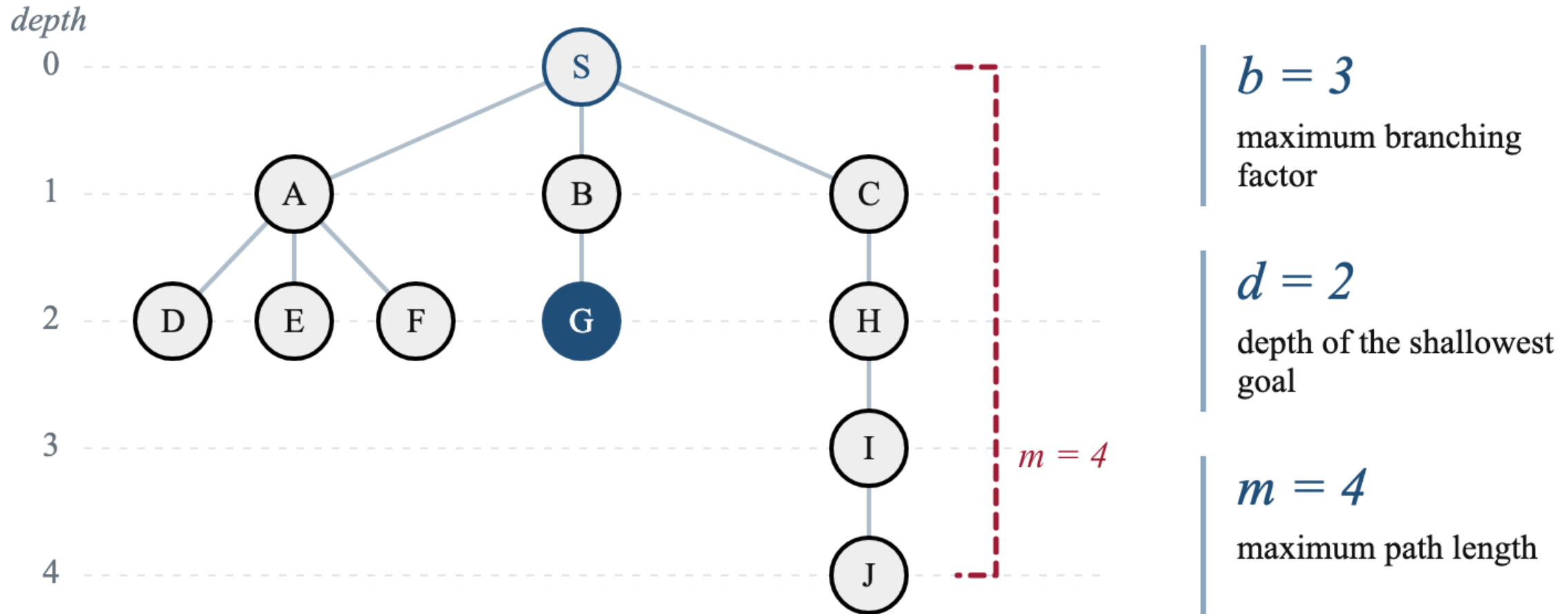
Search Complexity Depends on b , d , and m

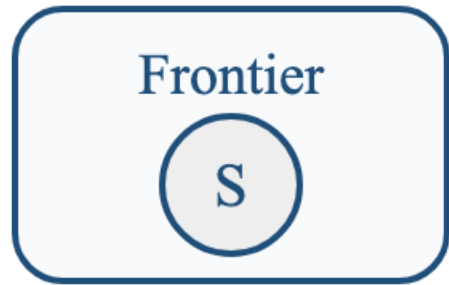


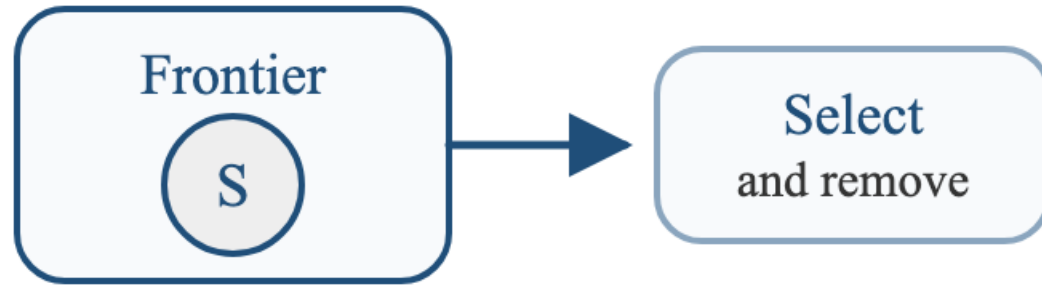
Search Complexity Depends on b , d , and m

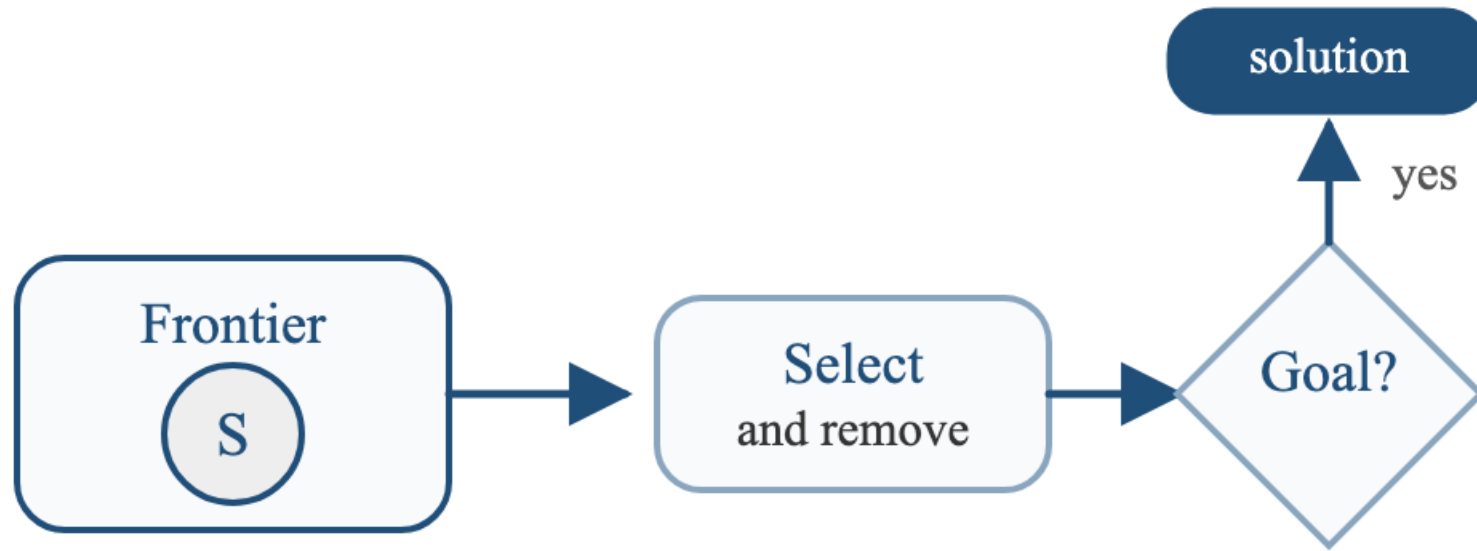


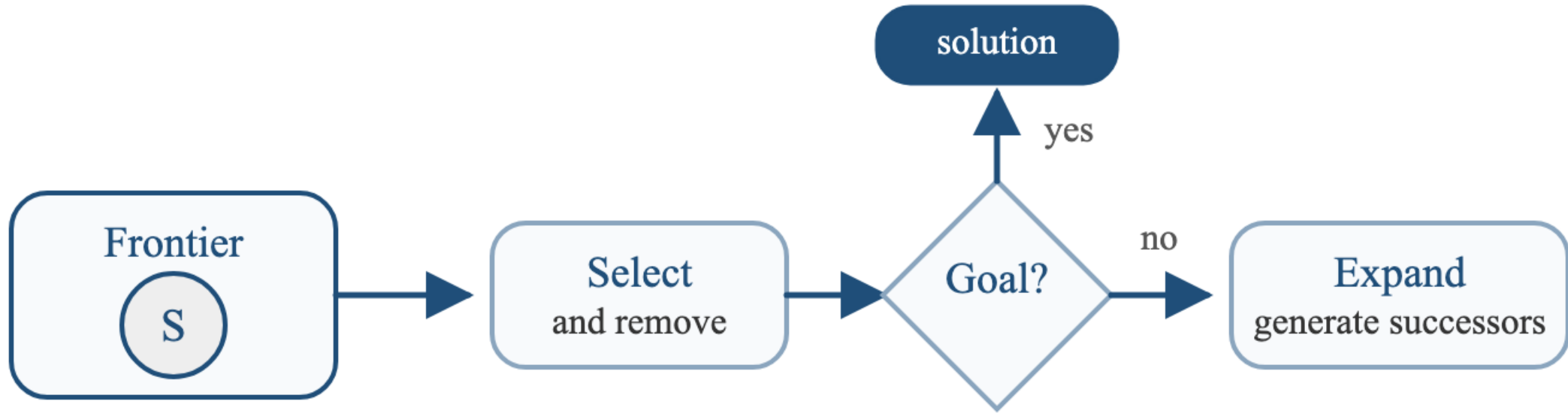
Search Complexity Depends on b , d , and m

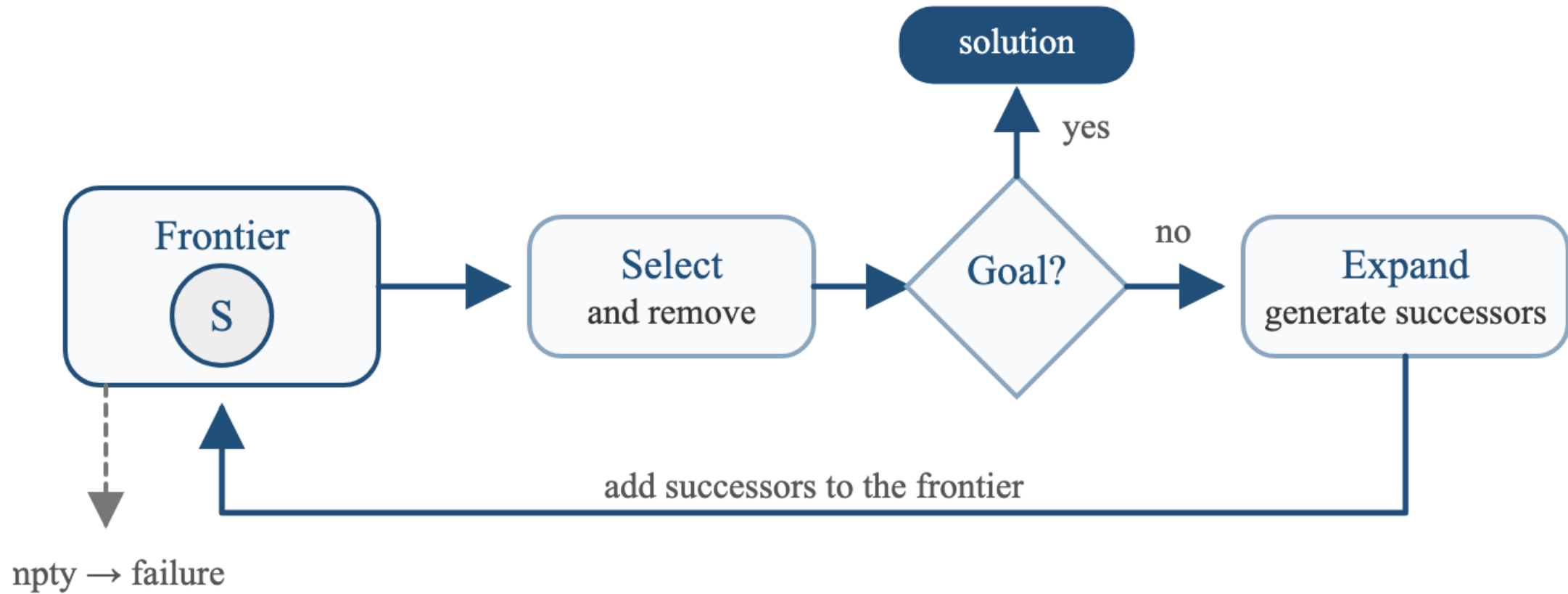












The search strategy supplies the rule used at **Select**.

Uninformed Search and Informed Search

Uninformed Search

Problem definition

No estimate to a goal

BFS · DFS · UCS

Informed Search

Problem definition

+ estimate to a goal

Greedy · A*

Uninformed Search and Informed Search

Uninformed Search

Problem definition

No estimate to a goal

BFS · DFS · UCS

Informed Search

Problem definition

+ estimate to a goal

Greedy · A*

We begin with BFS.

One Strategy, Two Search Algorithms

The strategy orders the frontier; TSA and GSA handle repeated states differently.

Tree Search Algorithm (TSA)

Keeps a new search node for every path

Graph Search Algorithm (GSA)

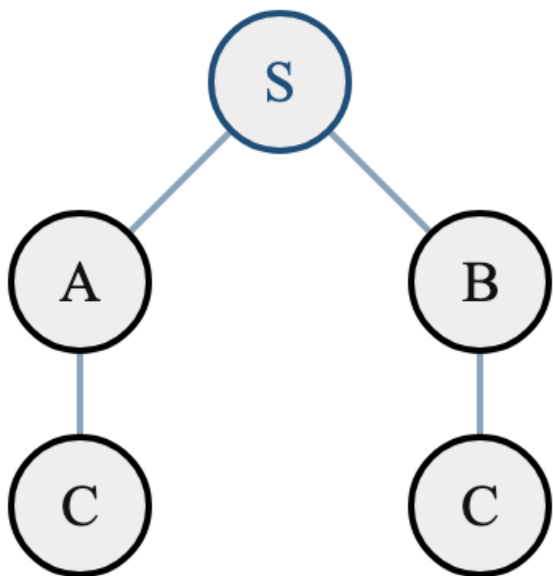
Remembers explored states and can skip repeats

One Strategy, Two Search Algorithms

The strategy orders the frontier; TSA and GSA handle repeated states differently.

Tree Search Algorithm (TSA)

Keeps a new search node for every path



State C appears in two search nodes

Graph Search Algorithm (GSA)

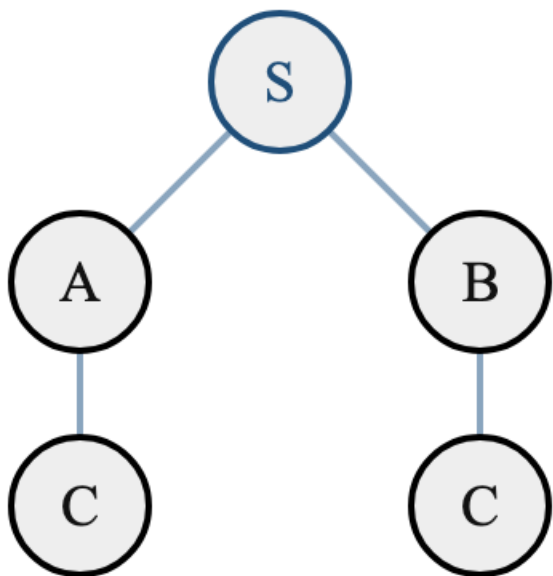
Remembers explored states and can skip repeats

One Strategy, Two Search Algorithms

The strategy orders the frontier; TSA and GSA handle repeated states differently.

Tree Search Algorithm (TSA)

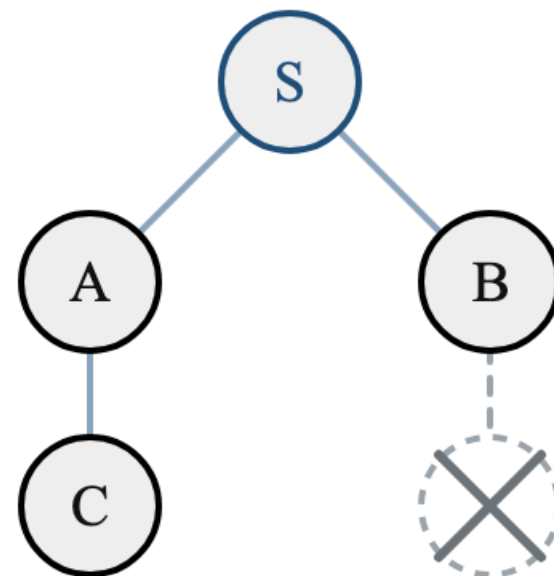
Keeps a new search node for every path



State C appears in two search nodes

Graph Search Algorithm (GSA)

Remembers explored states and can skip repeats



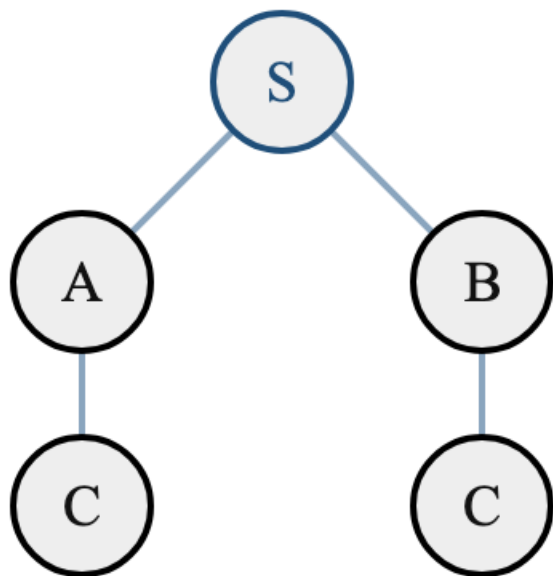
A node with explored state C can be skipped

One Strategy, Two Search Algorithms

The strategy orders the frontier; TSA and GSA handle repeated states differently.

Tree Search Algorithm (TSA)

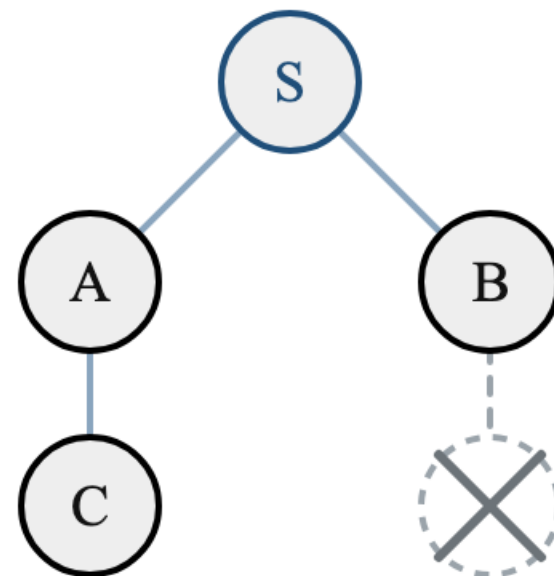
Keeps a new search node for every path



State C appears in two search nodes

Graph Search Algorithm (GSA)

Remembers explored states and can skip repeats

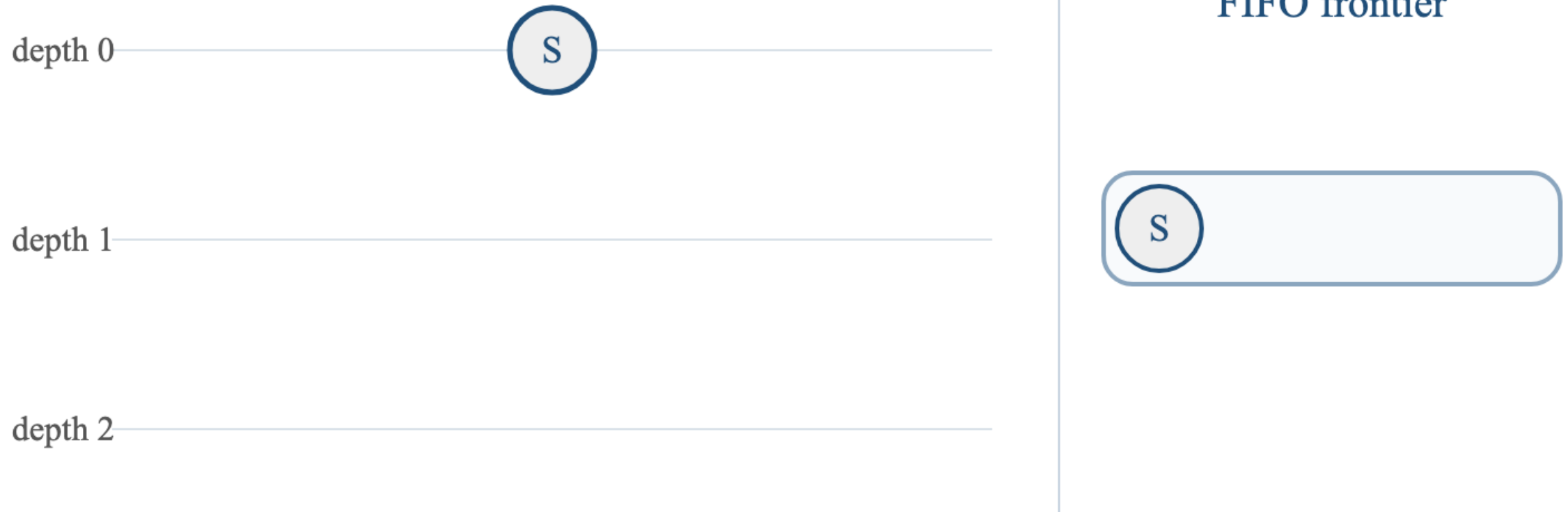


A node with explored state C can be skipped

We begin with BFS-TSA.

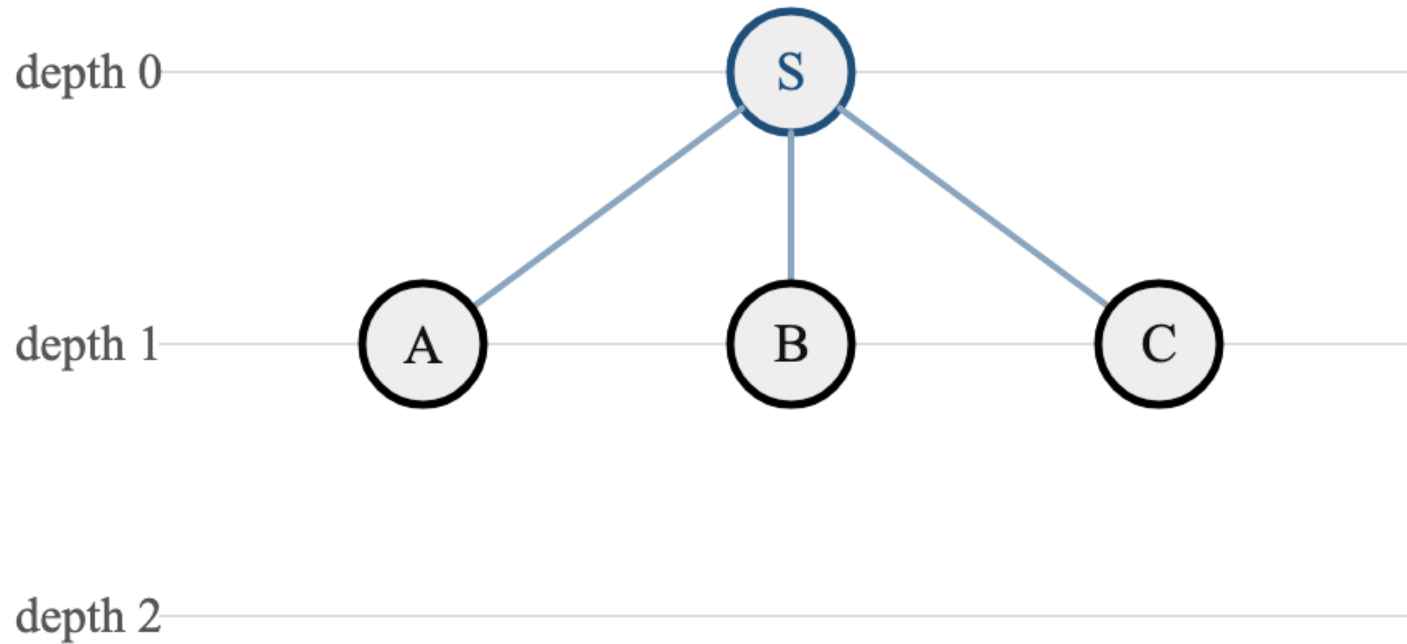
BFS Explores One Depth at a Time

BFS uses a FIFO frontier: earlier nodes stay ahead of newly generated nodes.

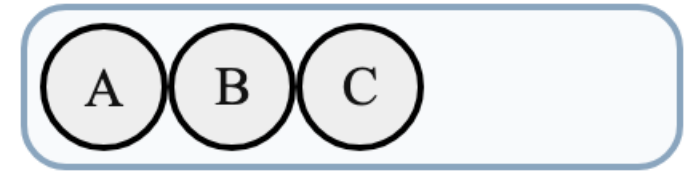


BFS Explores One Depth at a Time

BFS uses a FIFO frontier: earlier nodes stay ahead of newly generated nodes.

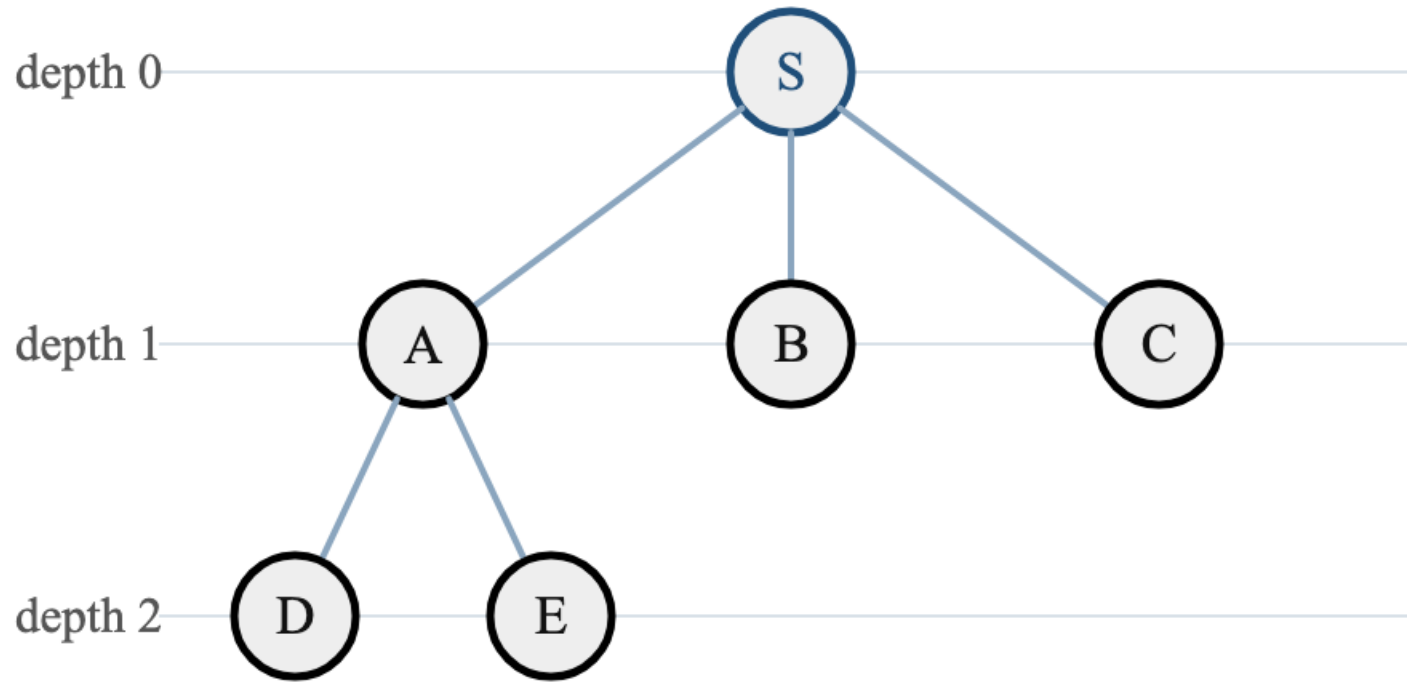


FIFO frontier

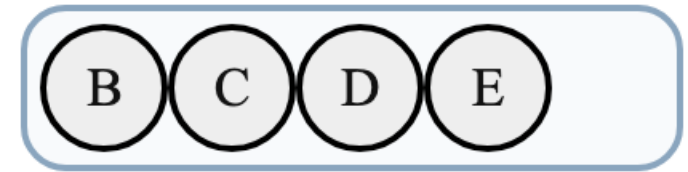


BFS Explores One Depth at a Time

BFS uses a FIFO frontier: earlier nodes stay ahead of newly generated nodes.

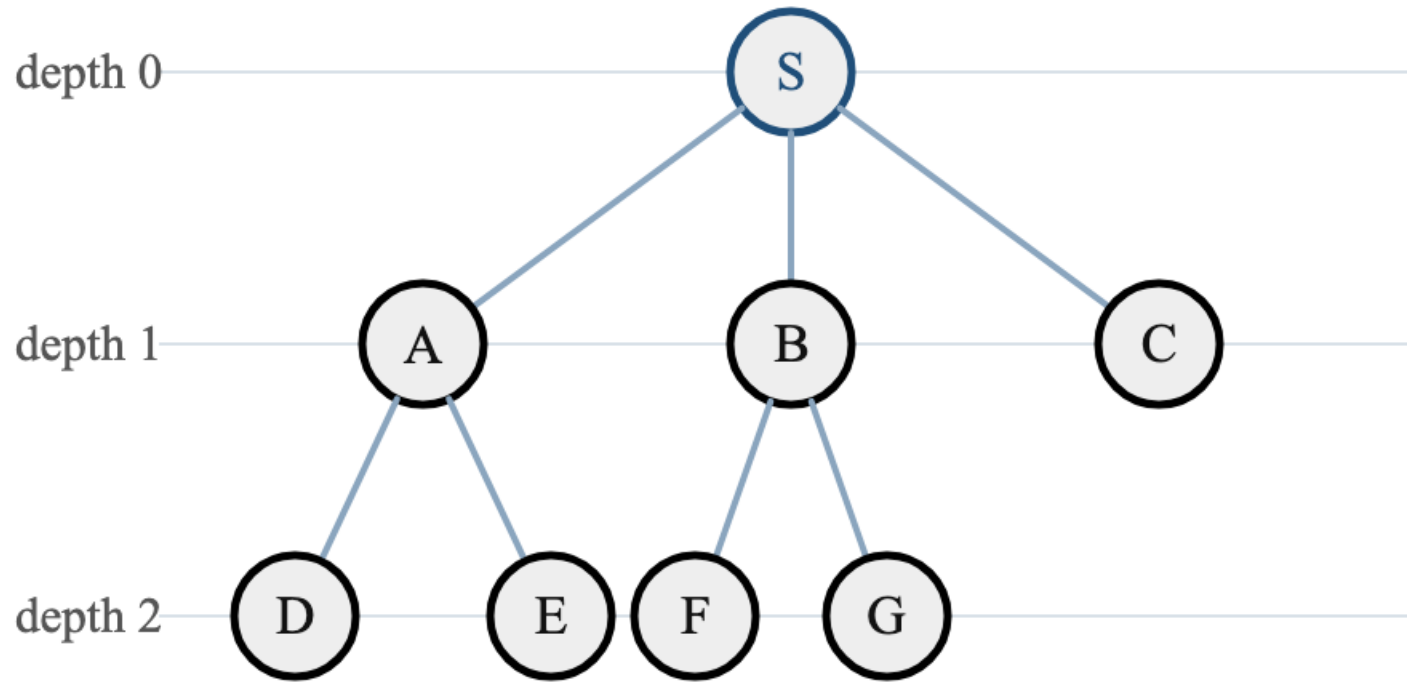


FIFO frontier

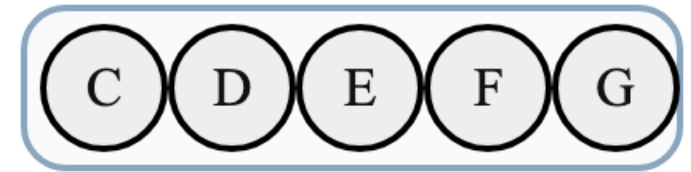


BFS Explores One Depth at a Time

BFS uses a FIFO frontier: earlier nodes stay ahead of newly generated nodes.



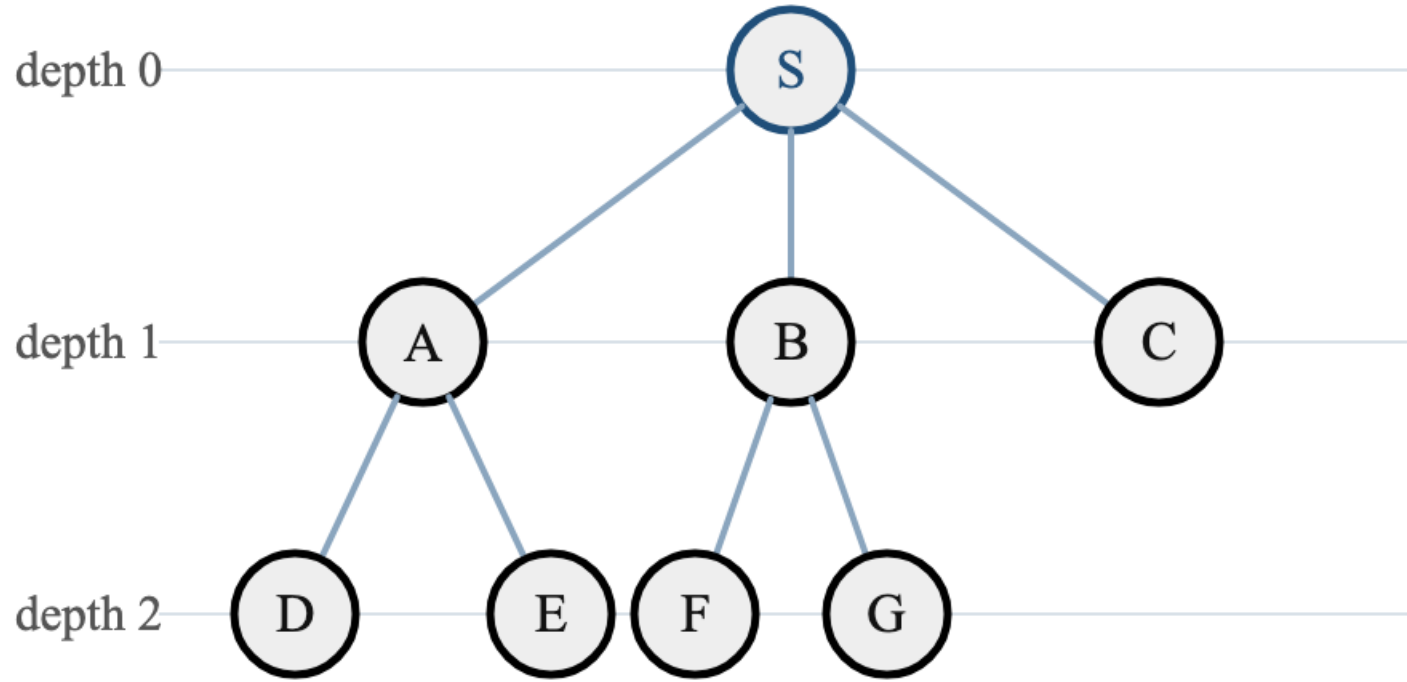
FIFO frontier



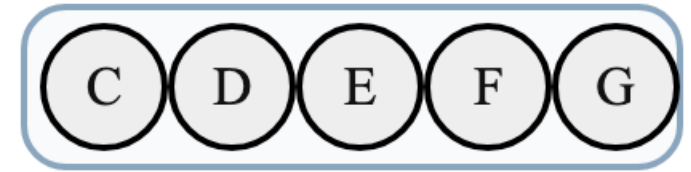
BFS Explores One Depth at a Time

TASK

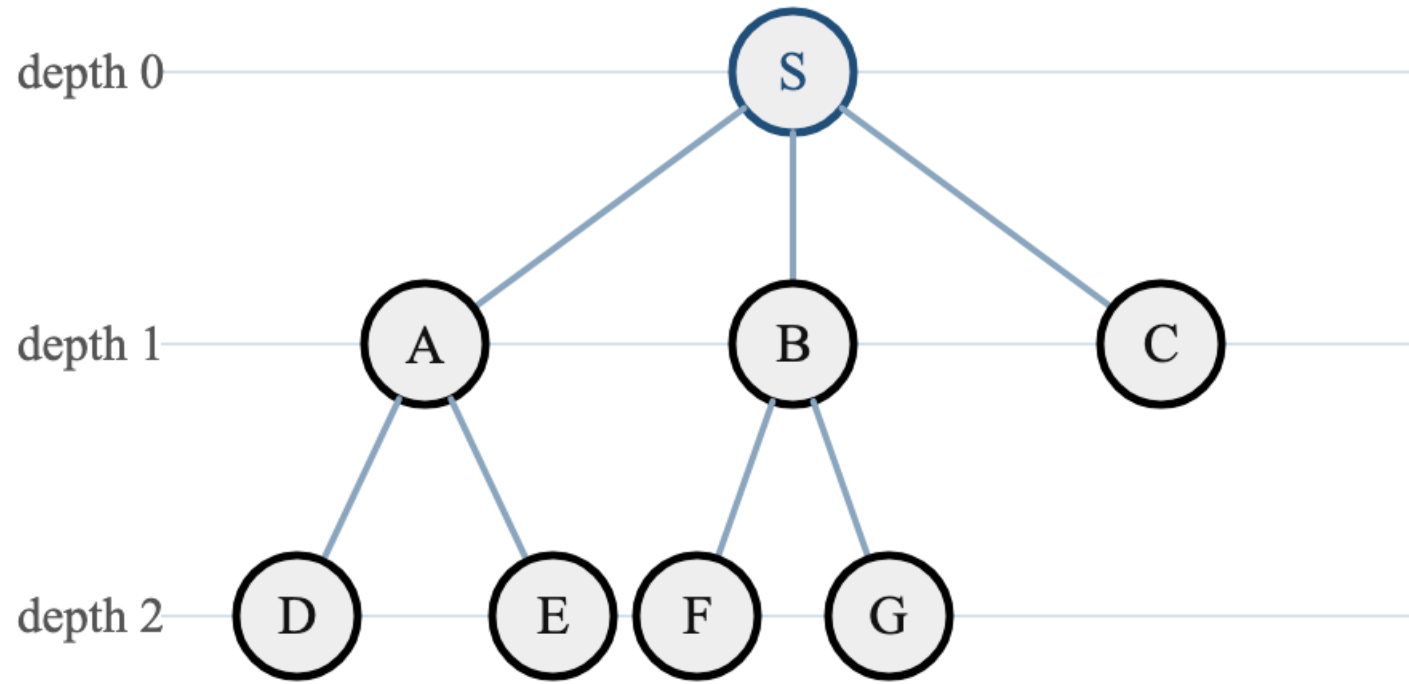
Which node should be expanded next?



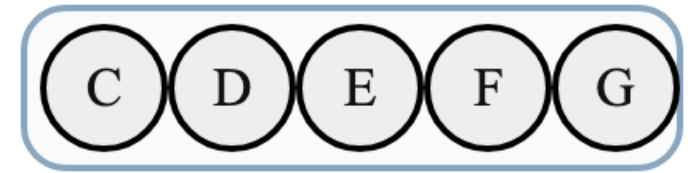
FIFO frontier



BFS Explores One Depth at a Time



FIFO frontier



Frontier

front

back



PY main.py

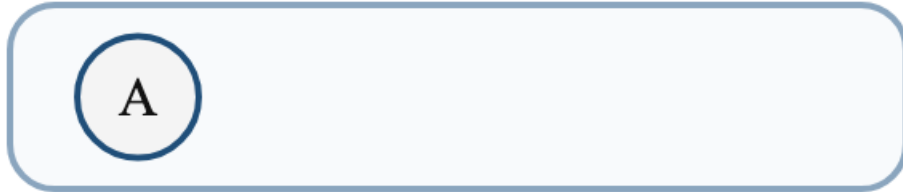
```
1 from collections import deque
2 frontier = deque()
3 frontier.append("A") # 1st
4 frontier.append("B") # 2nd
5 frontier.append("C") # 3rd
6 print("frontier:", list(frontier))
7 print("next:", frontier.popleft())
8 print("remaining:", list(frontier))
9
```

Download: comp3270.dirk.hk/queue_demo.py

Frontier

front

back



`append("A")`

PY main.py

```
1 from collections import deque
2 frontier = deque()
3 frontier.append("A")  # 1st
4 frontier.append("B")  # 2nd
5 frontier.append("C")  # 3rd
6 print("frontier:", list(frontier))
7 print("next:", frontier.popleft())
8 print("remaining:", list(frontier))
9
```

Download: comp3270.dirk.hk/queue_demo.py

Frontier

front

back



`append("A")`

`append("B")`

PY main.py

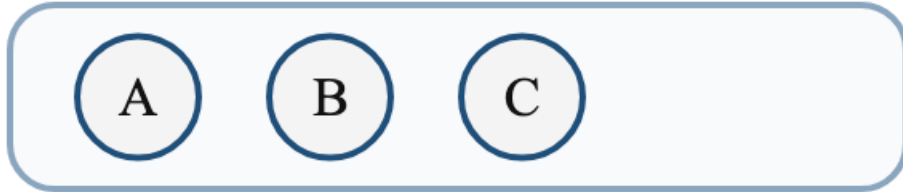
```
1 from collections import deque
2 frontier = deque()
3 frontier.append("A") # 1st
4 frontier.append("B") # 2nd
5 frontier.append("C") # 3rd
6 print("frontier:", list(frontier))
7 print("next:", frontier.popleft())
8 print("remaining:", list(frontier))
9
```

Download: comp3270.dirk.hk/queue_demo.py

Frontier

front

back



```
append("A")
```

```
append("B")
```

```
append("C")
```

PY main.py

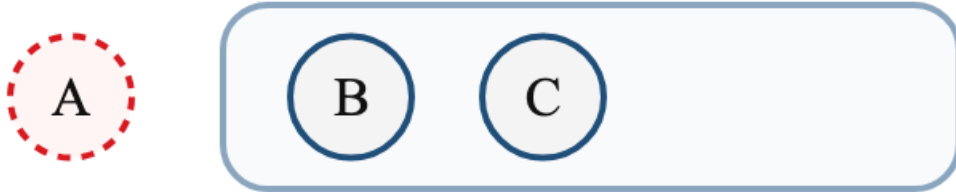
```
1 from collections import deque
2 frontier = deque()
3 frontier.append("A") # 1st
4 frontier.append("B") # 2nd
5 frontier.append("C") # 3rd
6 print("frontier:", list(frontier))
7 print("next:", frontier.popleft())
8 print("remaining:", list(frontier))
9
```

Download: comp3270.dirk.hk/queue_demo.py

Frontier

removed front

back



```
append("A")
```

```
append("B")
```

```
append("C")
```

```
popleft() → "A"; frontier = [B, C]
```

PY main.py

```
1 from collections import deque
2 frontier = deque()
3 frontier.append("A") # 1st
4 frontier.append("B") # 2nd
5 frontier.append("C") # 3rd
6 print("frontier:", list(frontier))
7 print("next:", frontier.popleft())
8 print("remaining:", list(frontier))
9
```

Download: comp3270.dirk.hk/queue_demo.py

A Search Node Stores the Path So Far

For our lecture code, a path string records one route through the search tree.

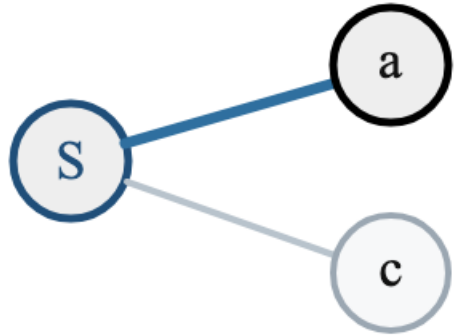
S

'S'

search tree	many possible routes
path	one route
current state	last character

A Search Node Stores the Path So Far

For our lecture code, a path string records one route through the search tree.

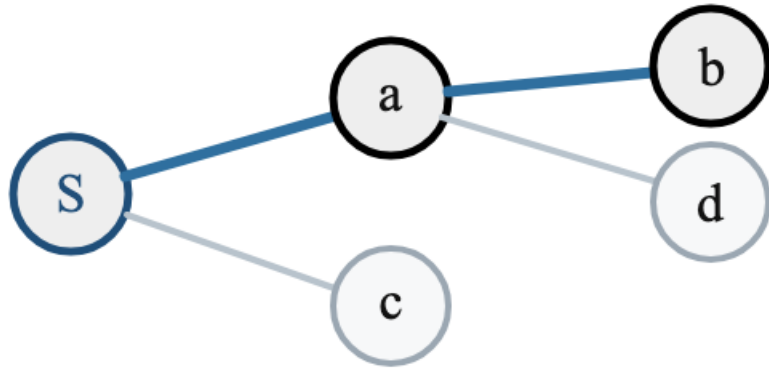


'S' → 'Sa'

search tree	many possible routes
path	one route
current state	last character

A Search Node Stores the Path So Far

For our lecture code, a path string records one route through the search tree.



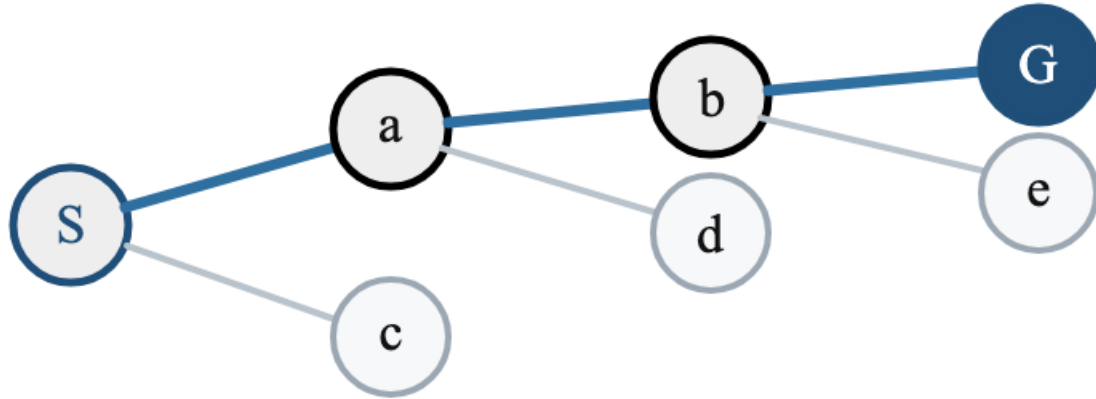
'S' → 'Sa' → 'Sab'

path[-1] gives 'b'

search tree	many possible routes
path	one route
current state	last character

A Search Node Stores the Path So Far

For our lecture code, a path string records one route through the search tree.



'S' → 'Sa' → 'Sab' → 'SabG'

path[-1] gives 'G'

PY main.py

```
1 path = "Sab"
2 print("current:", path[-1])
3 path += "G"
4 print("path:", path)
5 print("current:", path[-1])
6
```

Download: comp3270.dirk.hk/search_path_demo.py

BFS-TSA Repeats Three Operations

PY main.py

PY romania.py

```
1 from collections import deque
2 from romania import romania
3 def bfs_tsa(graph, start, goal, trace=False):
4     frontier = deque([start])
5     while frontier:
6         path = frontier.popleft()
7         state = path[-1]
8         if state == goal: return path
9         paths = [path + child for child in graph[state].neighbors]
10        frontier.extend(paths)
11        if trace: print(f"{path} -> {list(frontier)}")
12 print("solution:", bfs_tsa(romania, "A", "B", True))
13
```

Download: comp3270.dirk.hk/bfs_tsa_romania.py

BFS-TSA Repeats Three Operations

Remove the next path

Frontier ['A'] -> []

Path 'A'

PY main.py

PY romania.py

```
1 from collections import deque
2 from romania import romania
3 def bfs_tsa(graph, start, goal, trace=False):
4     frontier = deque([start])
5     while frontier:
6         path = frontier.popleft()
7         state = path[-1]
8         if state == goal: return path
9         paths = [path + child for child in graph[state]]
10        frontier.extend(paths)
11        if trace: print(f"{path} -> {list(frontier)}")
12    print("solution:", bfs_tsa(romania, "A", "B", trace=True))
13
```

Download: comp3270.dirk.hk/bfs_tsa_romania.py

BFS-TSA Repeats Three Operations

Remove the next path

Frontier ['A'] -> []

Path 'A'

Test its final state

Frontier []

Goal test
A ≠ B

PY main.py

PY romania.py

```
1 from collections import deque
2 from romania import romania
3 def bfs_tsa(graph, start, goal, trace=False):
4     frontier = deque([start])
5     while frontier:
6         path = frontier.popleft()
7         state = path[-1]
8         if state == goal: return path
9         paths = [path + child for child in graph[state]]
10        frontier.extend(paths)
11        if trace: print(f"{path} -> {list(frontier)}")
12    print("solution:", bfs_tsa(romania, "A", "B", trace=True))
13
```

Download: comp3270.dirk.hk/bfs_tsa_romania.py

BFS-TSA Repeats Three Operations

Remove the next path

Frontier ['A'] -> []

Path 'A'

Test its final state

Frontier []

Goal test
A ≠ B

Append every extension

Frontier [] -> ['AS', 'AT', 'AZ']

PY main.py

PY romania.py

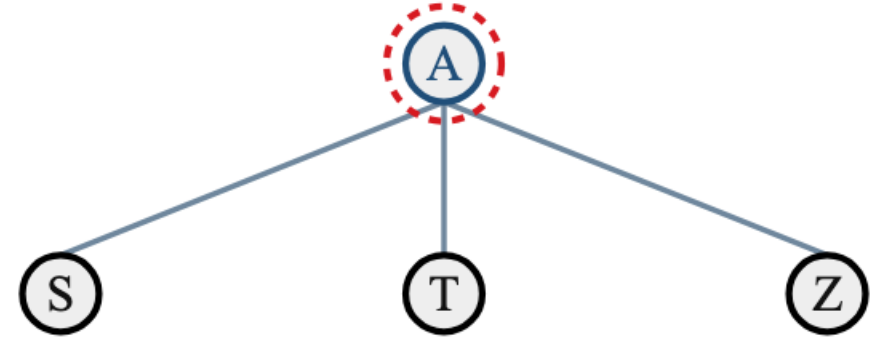
```
1 from collections import deque
2 from romania import romania
3 def bfs_tsa(graph, start, goal, trace=False):
4     frontier = deque([start])
5     while frontier:
6         path = frontier.popleft()
7         state = path[-1]
8         if state == goal: return path
9         paths = [path + child for child in graph[state]]
10        frontier.extend(paths)
11        if trace: print(f"{path} -> {list(frontier)}")
12    print("solution:", bfs_tsa(romania, "A", "B", trace=True))
13
```

Download: comp3270.dirk.hk/bfs_tsa_romania.py

Program Output

A -> ['AS', 'AT', 'AZ']

Search Tree

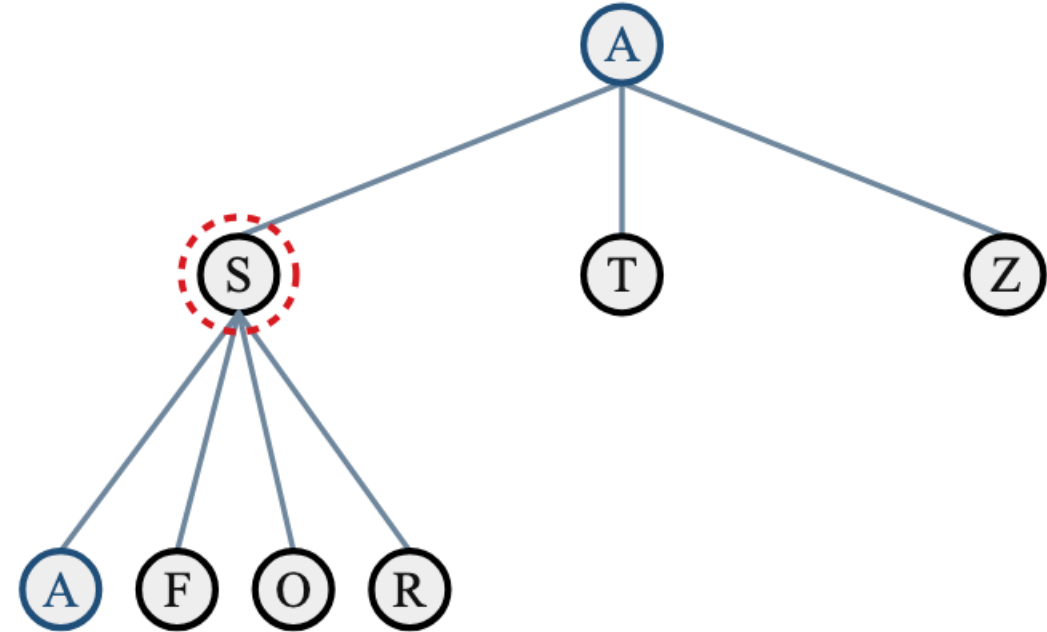


Program Output

A -> ['AS', 'AT', 'AZ']

AS -> ['AT', 'AZ', 'ASA', 'ASF',
'ASO', 'ASR']

Search Tree



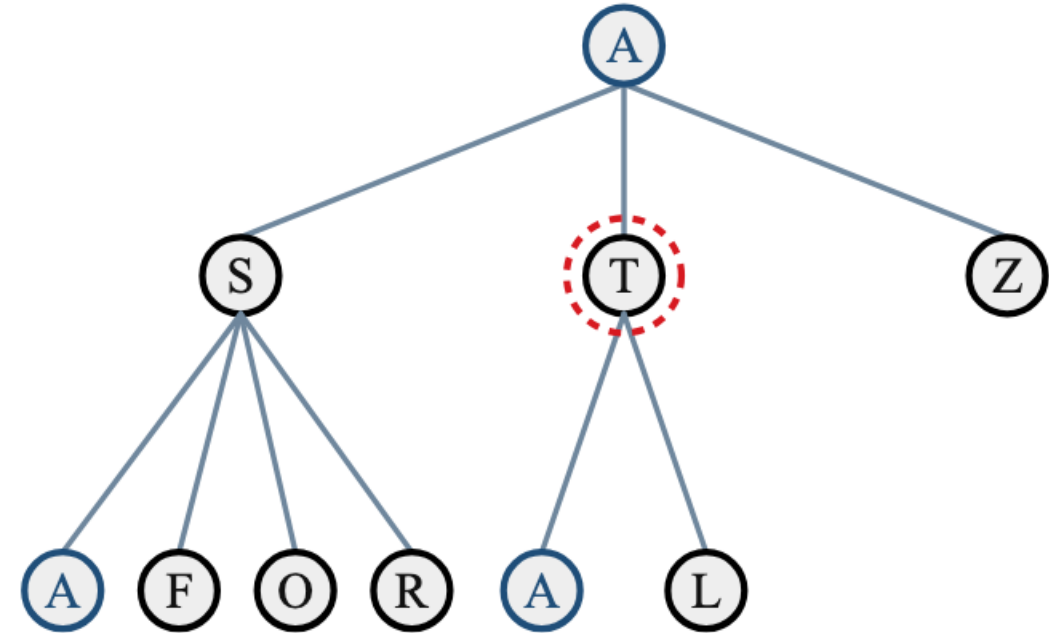
Program Output

A → ['AS', 'AT', 'AZ']

AS → ['AT', 'AZ', 'ASA', 'ASF',
'ASO', 'ASR']

AT → ['AZ', 'ASA', 'ASF', 'ASO',
'ASR', 'ATA', 'ATL']

Search Tree



Program Output

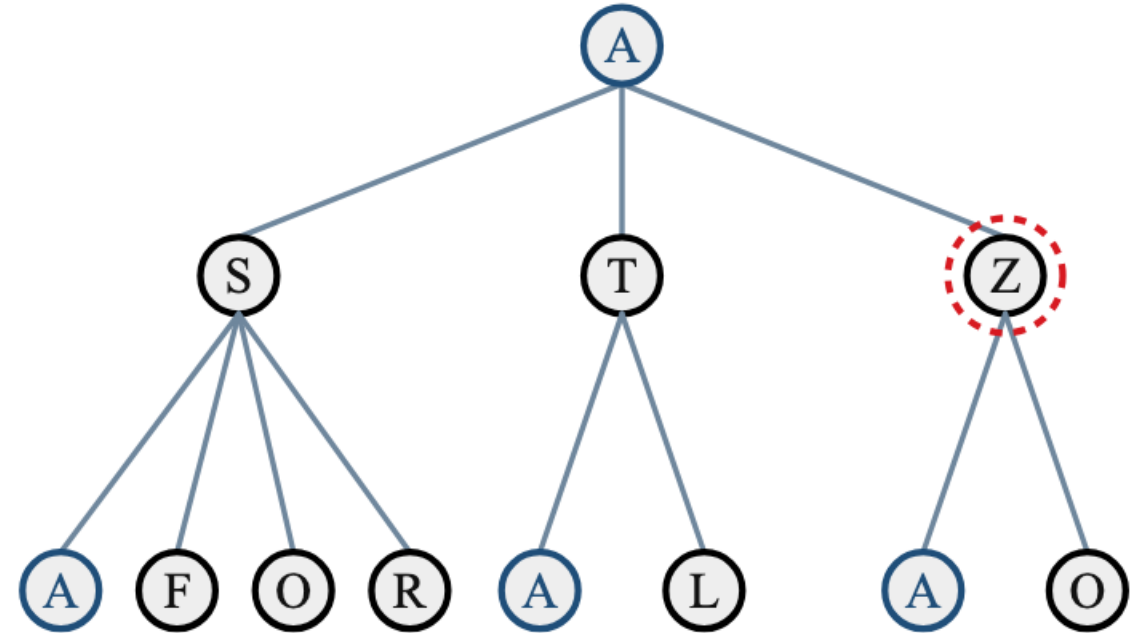
A → ['AS', 'AT', 'AZ']

AS → ['AT', 'AZ', 'ASA', 'ASF',
'ASO', 'ASR']

AT → ['AZ', 'ASA', 'ASF', 'ASO',
'ASR', 'ATA', 'ATL']

AZ → ['ASA', 'ASF', 'ASO', 'ASR',
'ATA', 'ATL', 'AZA', 'AZO']

Search Tree



Program Output

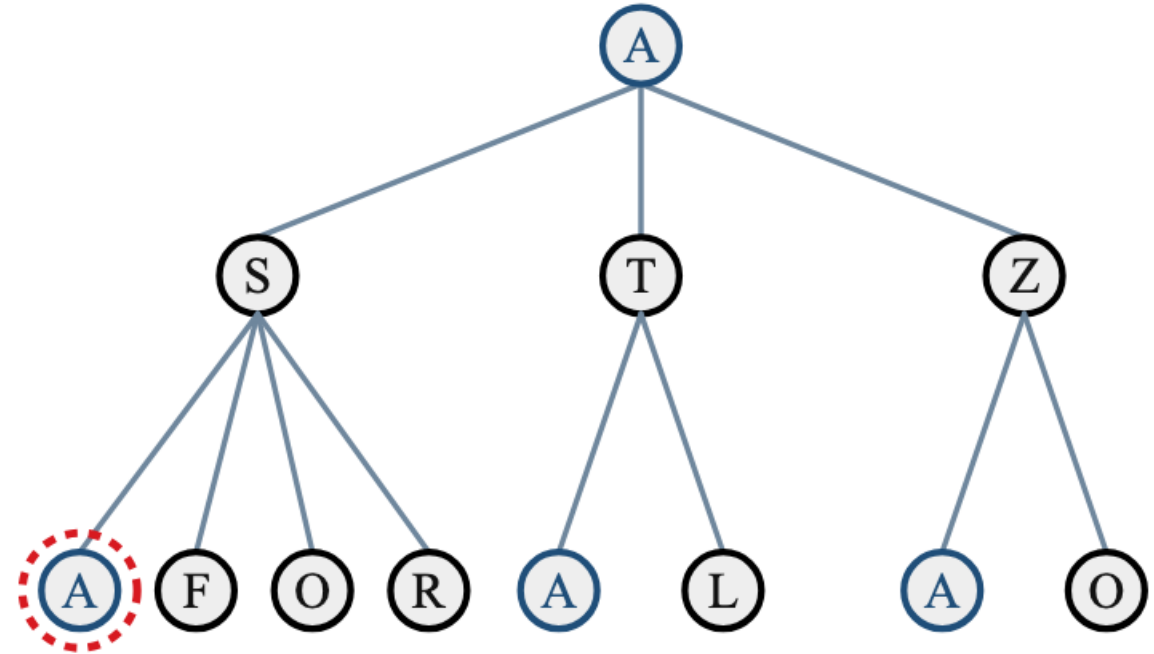
A → ['AS', 'AT', 'AZ']

AS → ['AT', 'AZ', 'ASA', 'ASF',
'ASO', 'ASR']

AT → ['AZ', 'ASA', 'ASF', 'ASO',
'ASR', 'ATA', 'ATL']

AZ → ['ASA', 'ASF', 'ASO', 'ASR',
'ATA', 'ATL', 'AZA', 'AZO']

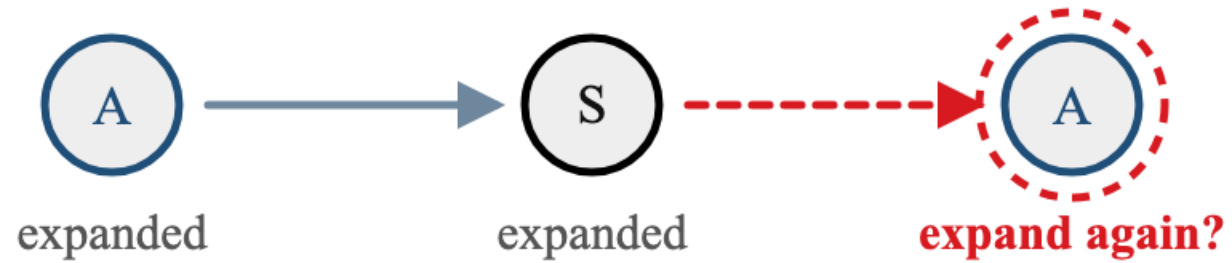
Search Tree



Explore A again?

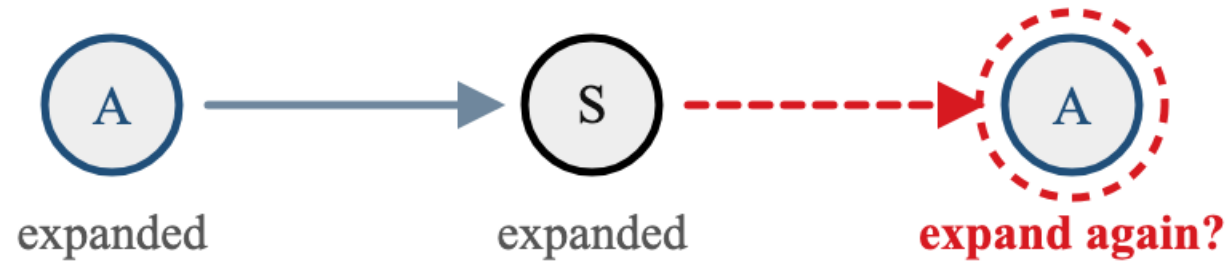
PROBLEM TSA revisits states

It does not remember where it has been



PROBLEM TSA revisits states

It does not remember where it has been



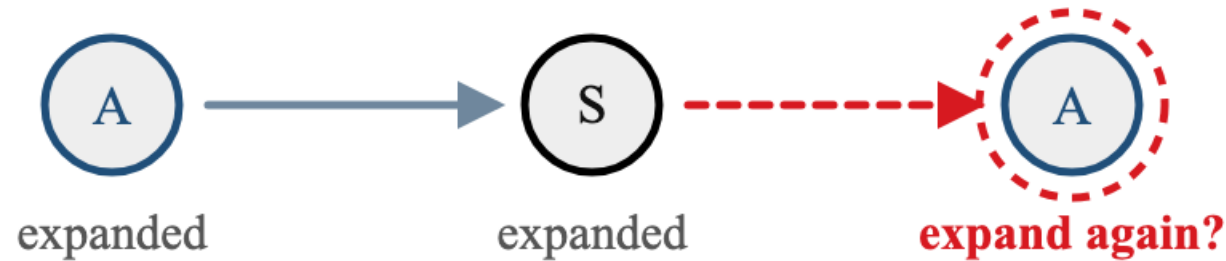
SOLUTION Remember explored states

`explored = {'A', 'S'}`

$A \in \text{explored} \rightarrow \text{skip}$

PROBLEM TSA revisits states

It does not remember where it has been



SOLUTION Remember explored states

`explored = {'A', 'S'}`

$A \in \text{explored} \rightarrow \text{skip}$

TSA + explored states = **GSA**

Repeated states are skipped instead of expanded again

Frontier ['AS', 'AT', 'AZ']
stores path strings

Explored {'A'}
stores states only

PY main.py

PY romania.py

```
1 from collections import deque
2 from romania import romania
3 def bfs_gsa(graph, start, goal, trace=False):
4     frontier = deque([start])
5     explored = set()
6     while frontier:
7         path = frontier.popleft()
8         state = path[-1]
9         if state == goal: return path
10        if state in explored: continue
11        explored.add(state)
12        paths = [path + child for child in graph[state]]
13        frontier.extend(paths)
14        if trace:
15            print(f"{path} -> {list(frontier)}")
16            print(f"explored: {sorted(explored)}\n")
17 print("solution:". bfs_gsa(romania, "A", "B", True))
```

Download: comp3270.dirk.hk/bfs_gsa_romania.py

Frontier ['AS', 'AT', 'AZ']
stores path strings

Explored {'A'}
stores states only

Create the memory

```
explored = set()
```

PY main.py

PY romania.py

```
1 from collections import deque
2 from romania import romania
3 def bfs_gsa(graph, start, goal, trace=False):
4     frontier = deque([start])
5     explored = set()
6     while frontier:
7         path = frontier.popleft()
8         state = path[-1]
9         if state == goal: return path
10        if state in explored: continue
11        explored.add(state)
12        paths = [path + child for child in graph[state]]
13        frontier.extend(paths)
14        if trace:
15            print(f"{path} -> {list(frontier)}")
16            print(f"explored: {sorted(explored)}\n")
17 print("solution:". bfs_gsa(romania, "A", "B", True))
```

Download: comp3270.dirk.hk/bfs_gsa_romania.py

Frontier ['AS', 'AT', 'AZ']
stores path strings

Explored {'A'}
stores states only

Create the memory

```
explored = set()
```

Skip a repeated state

```
state in explored
```

PY main.py

PY romania.py

```
1 from collections import deque
2 from romania import romania
3 def bfs_gsa(graph, start, goal, trace=False):
4     frontier = deque([start])
5     explored = set()
6     while frontier:
7         path = frontier.popleft()
8         state = path[-1]
9         if state == goal: return path
10        if state in explored: continue
11        explored.add(state)
12        paths = [path + child for child in graph[state]]
13        frontier.extend(paths)
14        if trace:
15            print(f"{path} -> {list(frontier)}")
16            print(f"explored: {sorted(explored)}\n")
17 print("solution:", bfs_gsa(romania, "A", "B", True))
```

Download: comp3270.dirk.hk/bfs_gsa_romania.py

Frontier ['AS', 'AT', 'AZ']
stores path strings

Explored {'A'}
stores states only

Create the memory

```
explored = set()
```

Skip a repeated state

```
state in explored
```

Remember before expanding

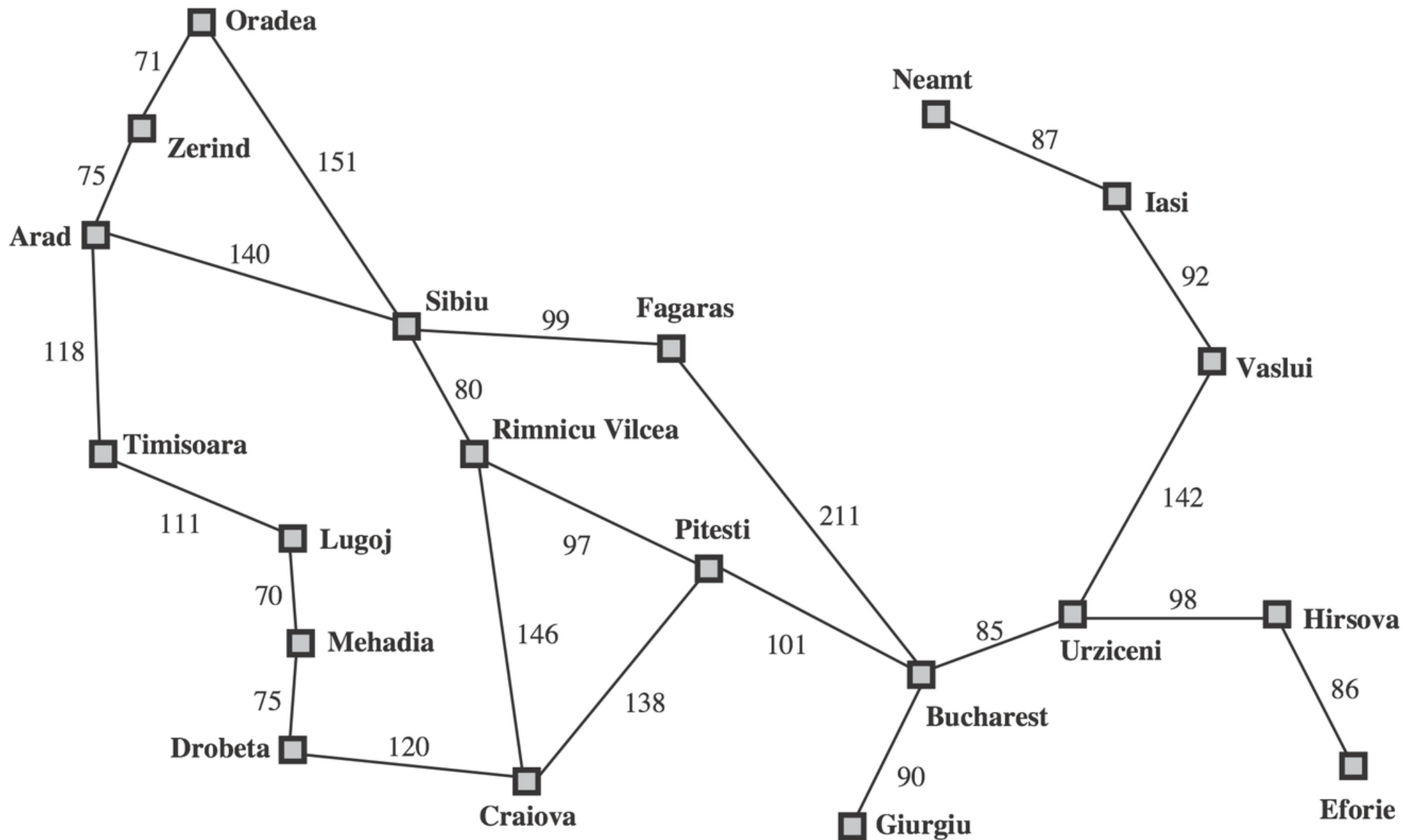
```
explored.add(state)
```

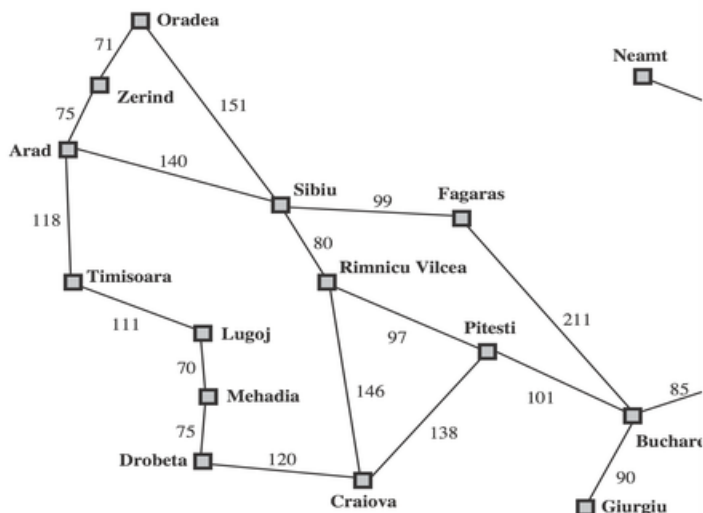
PY main.py

PY romania.py

```
1 from collections import deque
2 from romania import romania
3 def bfs_gsa(graph, start, goal, trace=False):
4     frontier = deque([start])
5     explored = set()
6     while frontier:
7         path = frontier.popleft()
8         state = path[-1]
9         if state == goal: return path
10        if state in explored: continue
11        explored.add(state)
12        paths = [path + child for child in graph[state]]
13        frontier.extend(paths)
14        if trace:
15            print(f"{path} -> {list(frontier)}")
16            print(f"explored: {sorted(explored)}\n")
17 print("solution:". bfs_gsa(romania, "A", "B", True))
```

Download: comp3270.dirk.hk/bfs_gsa_romania.py





frontier:
paths

explored:
states

PY main.py

PY romania.py

```

1 from collections import deque
2 from romania import romania
3 def bfs_gsa(graph, start, goal, trace=False):
4     frontier = deque([start])
5     explored = set()
6     while frontier:
7         path = frontier.popleft()
8         state = path[-1]
9         if state == goal: return path
10        if state in explored: continue
11        explored.add(state)
12        paths = [path + child for child in graph[state]]
13        frontier.extend(paths)
14        if trace:
15            print(f"{path} -> {list(frontier)}")
16            print(f"explored: {sorted(explored)}\n")
17 print("solution:", bfs_gsa(romania, "A", "B", True))

```

BFS-GSA Practice

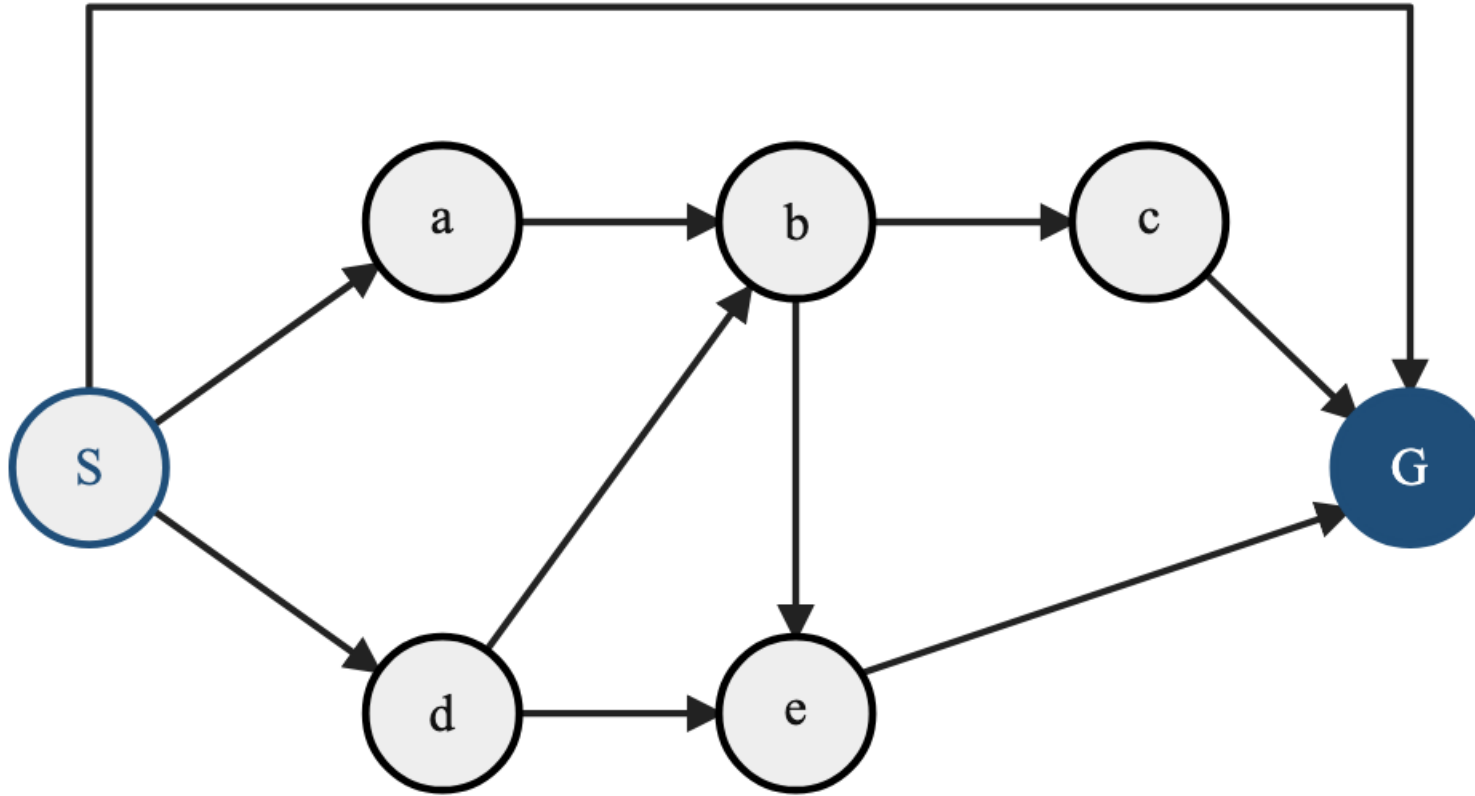
TASK

Run BFS-GSA from **S** to **G**

Record Explored order

Returned path

Add successors alphabetically



BFS-TSA Properties

TASK

Complete?

Optimal?

Time complexity?

Space complexity?

BFS-TSA Properties

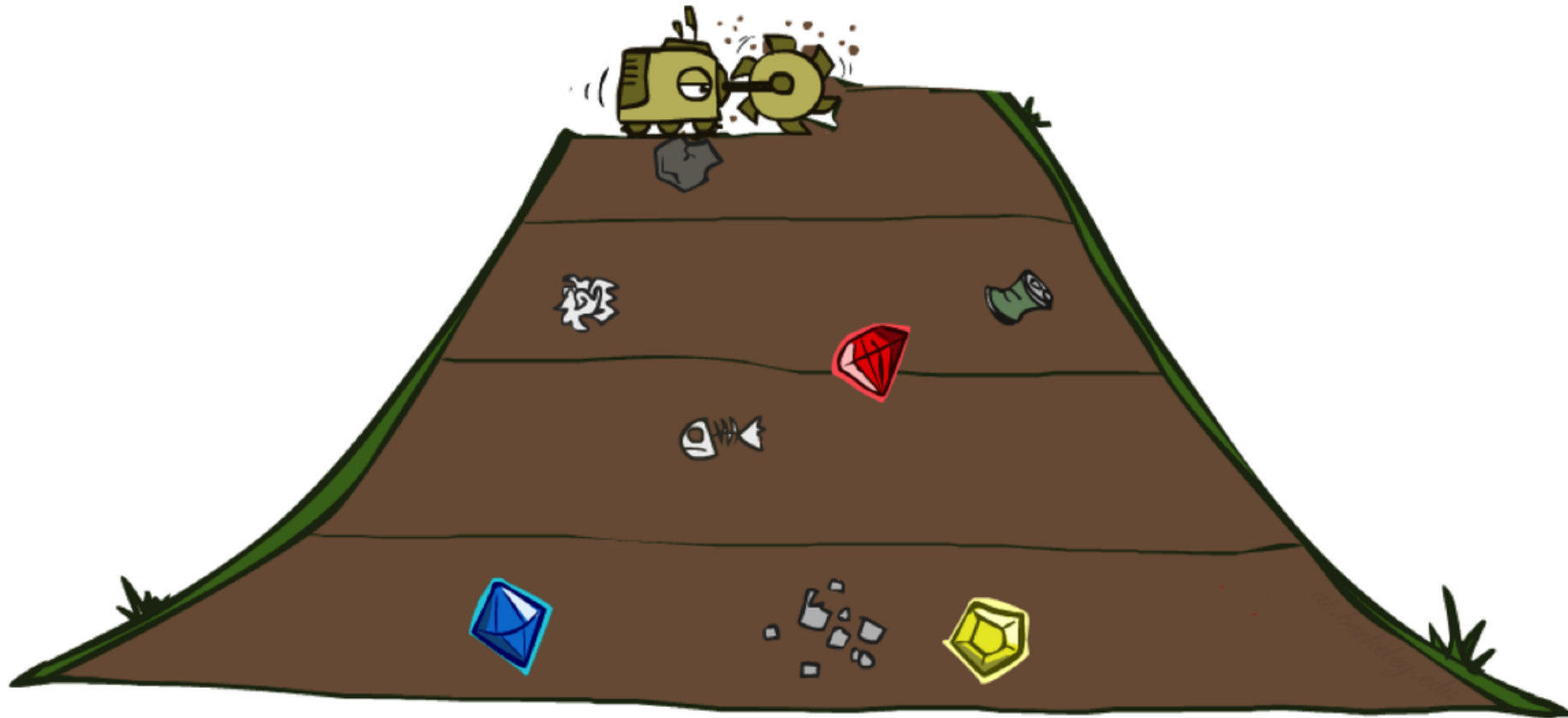
Complete?

Optimal?

Time complexity?

Space complexity?

BFS Explores the Shallowest Node First



DFS Explores the Deepest Node First



Frontier

bottom

top



PY main.py

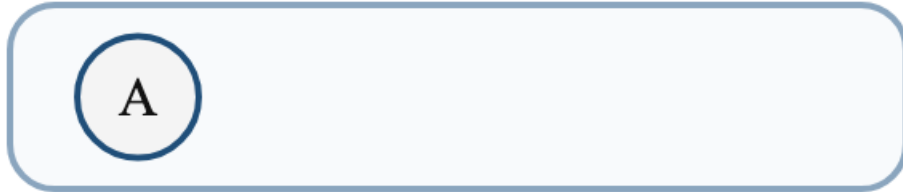
```
1 from collections import deque
2 frontier = deque()
3 frontier.append("A") # 1st
4 frontier.append("B") # 2nd
5 frontier.append("C") # 3rd
6 print("frontier:", list(frontier))
7 print("next:", frontier.pop())
8 print("remaining:", list(frontier))
9
```

Download: comp3270.dirk.hk/stack_demo.py

Frontier

bottom

top



append("A")

PY main.py

```
1 from collections import deque
2 frontier = deque()
3 frontier.append("A") # 1st
4 frontier.append("B") # 2nd
5 frontier.append("C") # 3rd
6 print("frontier:", list(frontier))
7 print("next:", frontier.pop())
8 print("remaining:", list(frontier))
9
```

Download: comp3270.dirk.hk/stack_demo.py

Frontier

bottom

top



`append("A")`

`append("B")`

PY main.py

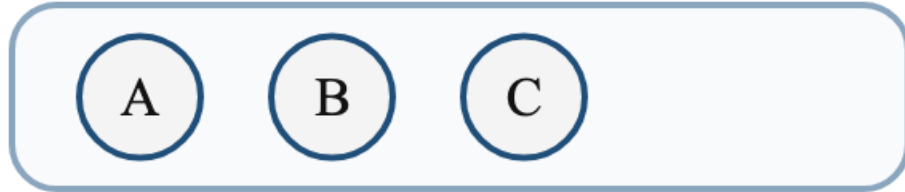
```
1 from collections import deque
2 frontier = deque()
3 frontier.append("A") # 1st
4 frontier.append("B") # 2nd
5 frontier.append("C") # 3rd
6 print("frontier:", list(frontier))
7 print("next:", frontier.pop())
8 print("remaining:", list(frontier))
9
```

Download: comp3270.dirk.hk/stack_demo.py

Frontier

bottom

top



`append("A")`

`append("B")`

`append("C")`

PY main.py

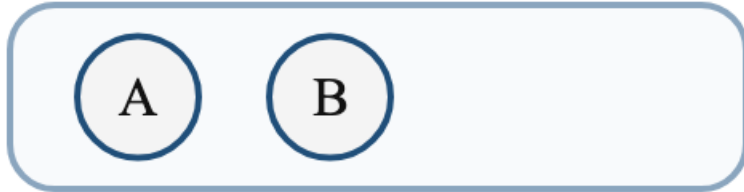
```
1 from collections import deque
2 frontier = deque()
3 frontier.append("A") # 1st
4 frontier.append("B") # 2nd
5 frontier.append("C") # 3rd
6 print("frontier:", list(frontier))
7 print("next:", frontier.pop())
8 print("remaining:", list(frontier))
9
```

Download: comp3270.dirk.hk/stack_demo.py

Frontier

bottom

top removed



append("A")

append("B")

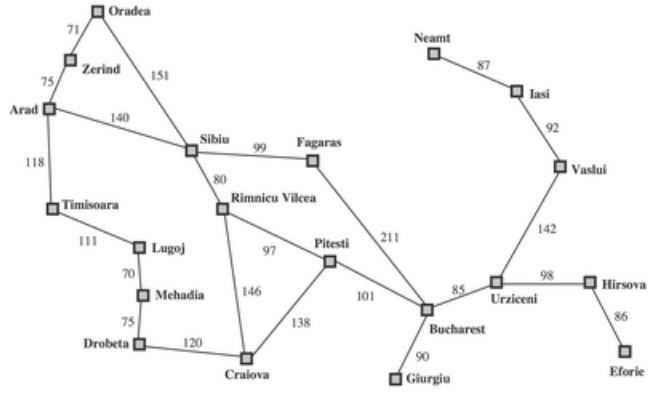
append("C")

pop() → "C"; frontier = [A, B]

PY main.py

```
1 from collections import deque
2 frontier = deque()
3 frontier.append("A") # 1st
4 frontier.append("B") # 2nd
5 frontier.append("C") # 3rd
6 print("frontier:", list(frontier))
7 print("next:", frontier.pop())
8 print("remaining:", list(frontier))
9
```

Download: comp3270.dirk.hk/stack_demo.py



frontier:
paths

explored:
states

PY main.py

PY romania.py

```

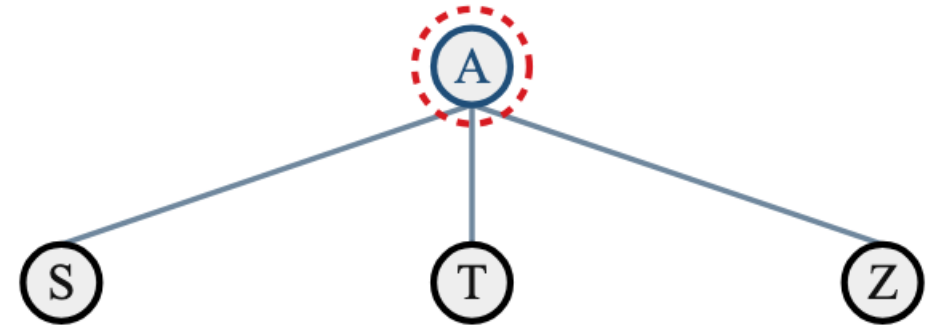
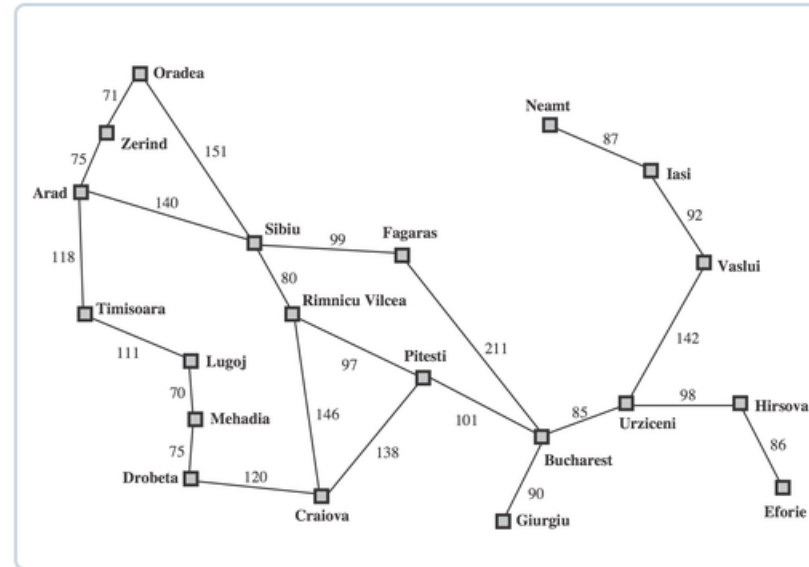
1  |from collections import deque
2  |from romania import romania
3  |def dfs_gsa(graph, start, goal, trace=False):
4  |    frontier = deque([start])
5  |    explored = set()
6  |    while frontier:
7  |        path = frontier.pop()
8  |        state = path[-1]
9  |        if state == goal: return path
10 |        if state in explored: continue
11 |        explored.add(state)
12 |        paths = [path + child for child in graph[state]]
13 |        frontier.extend(paths)
14 |        if trace:
15 |            print(f"{path} -> {list(frontier)}")
16 |            print(f"explored: {sorted(explored)}\n")
17 |    print("solution:", dfs_gsa(romania, "A", "B", True))

```

Program Output

```
A -> ['AS', 'AT', 'AZ']  
explored: ['A']
```

Search Tree

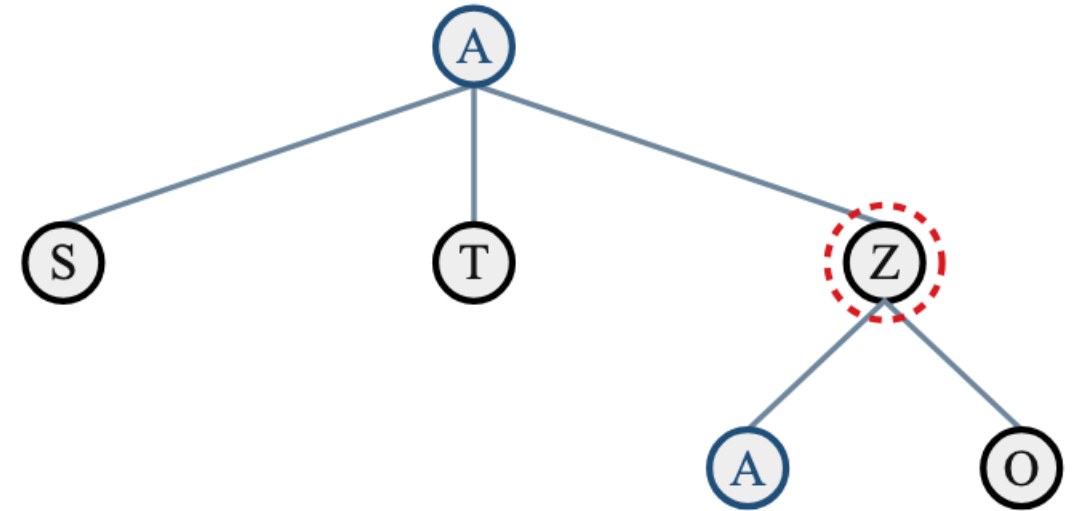
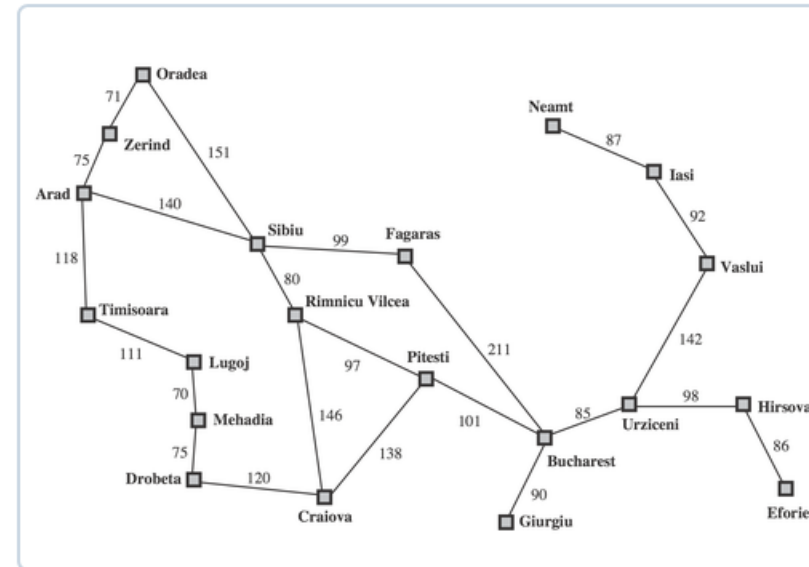


Program Output

```
A -> ['AS', 'AT', 'AZ']  
explored: ['A']
```

```
AZ -> [..., 'AZA', 'AZO']  
explored: ['A', 'Z']
```

Search Tree



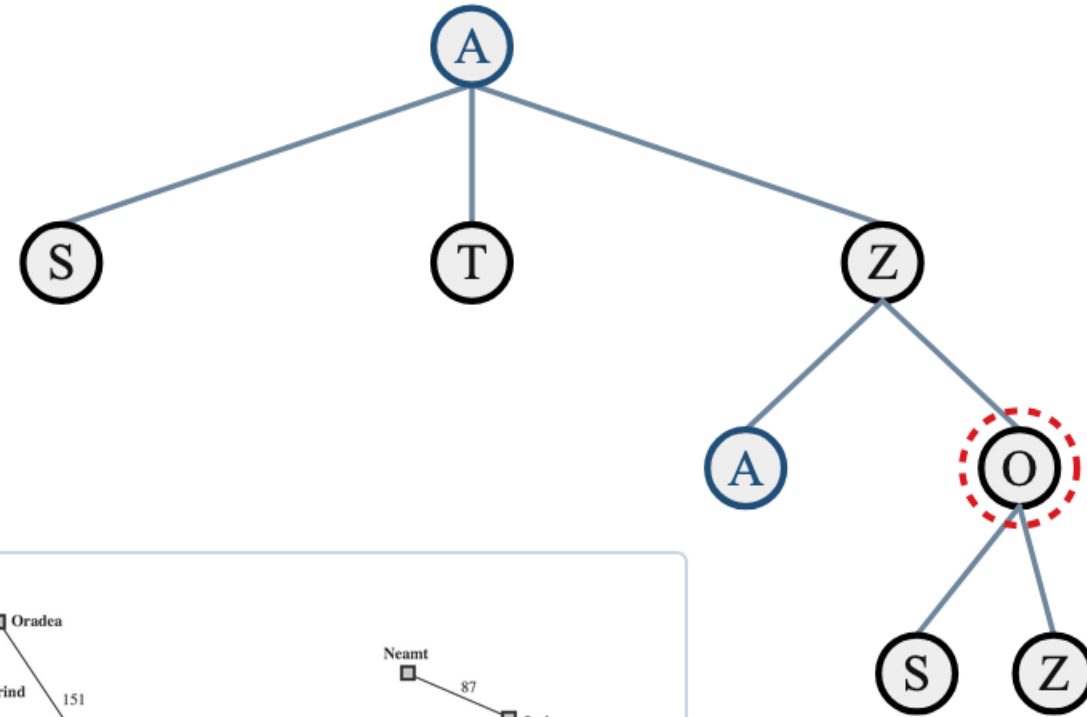
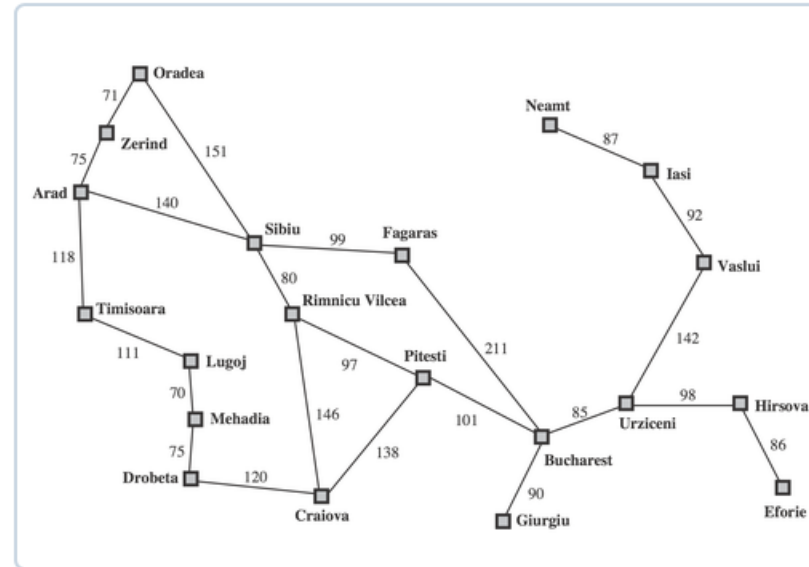
Program Output

A → ['AS', 'AT', 'AZ']
explored: ['A']

AZ → [..., 'AZA', 'AZO']
explored: ['A', 'Z']

AZO → [..., 'AZOS', 'AZOZ']
explored: ['A', 'O', 'Z']

Search Tree



Program Output

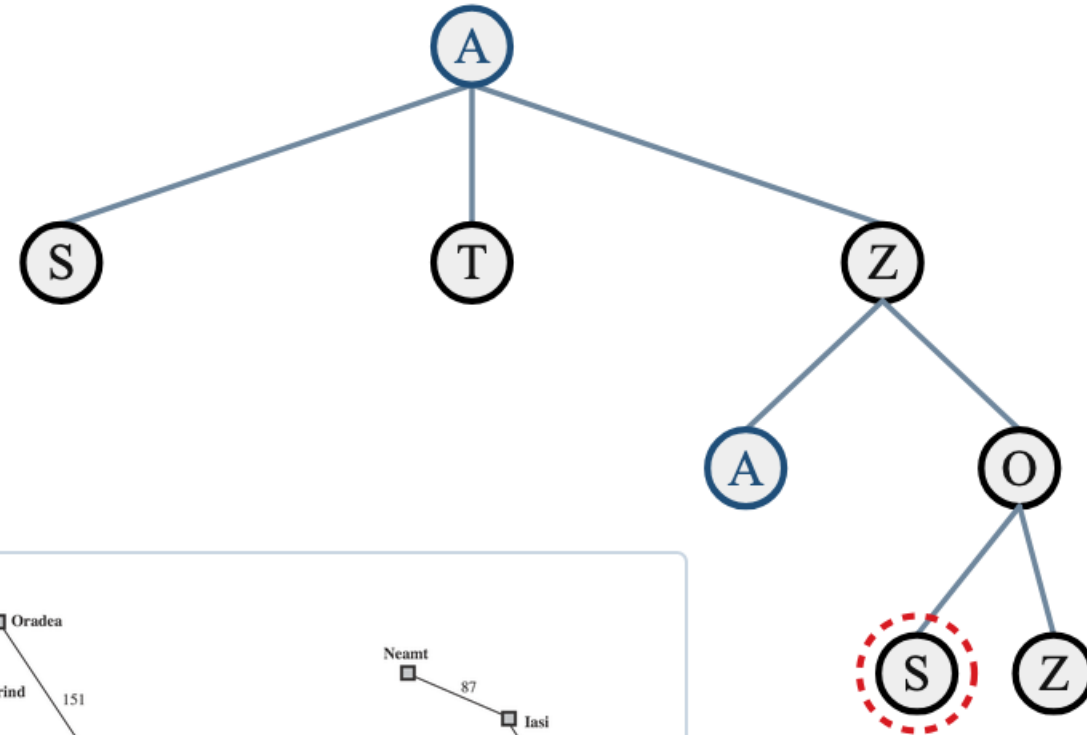
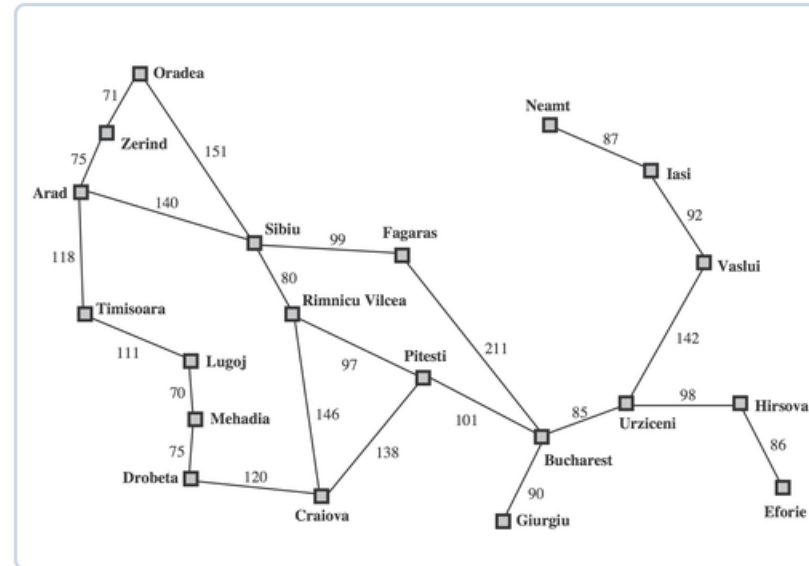
A → ['AS', 'AT', 'AZ']
explored: ['A']

AZ → [..., 'AZA', 'AZO']
explored: ['A', 'Z']

AZO → [..., 'AZOS', 'AZOZ']
explored: ['A', 'O', 'Z']

AZOS → [..., 'AZOSR']
explored: ['A', 'O', 'S', 'Z']

Search Tree



Program Output

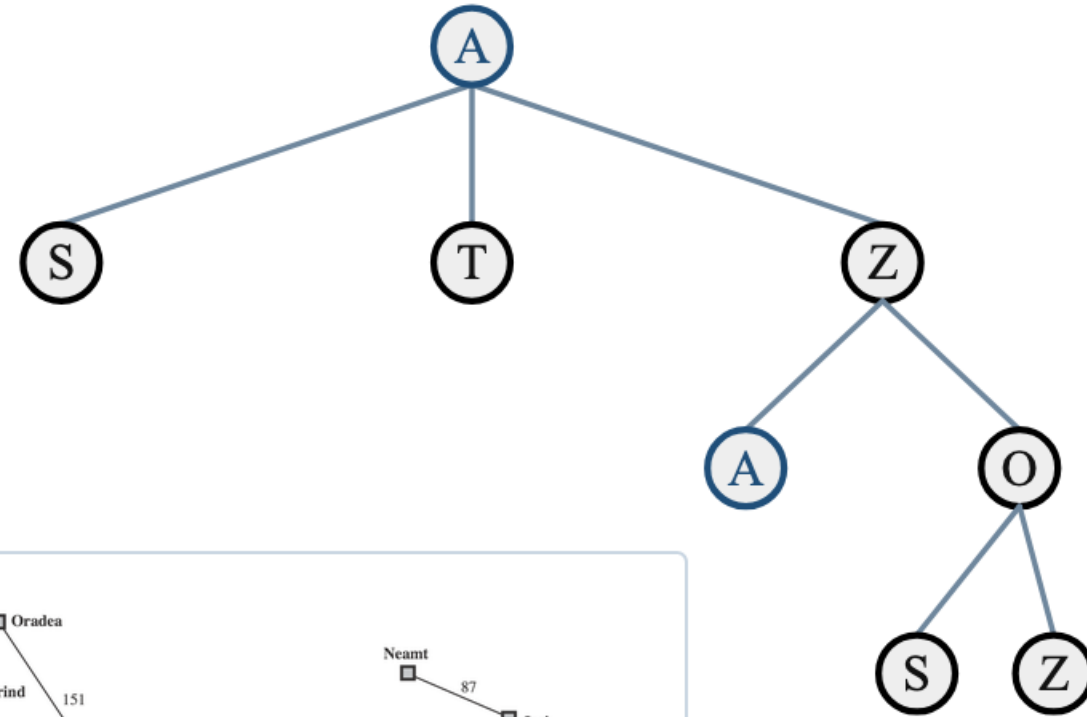
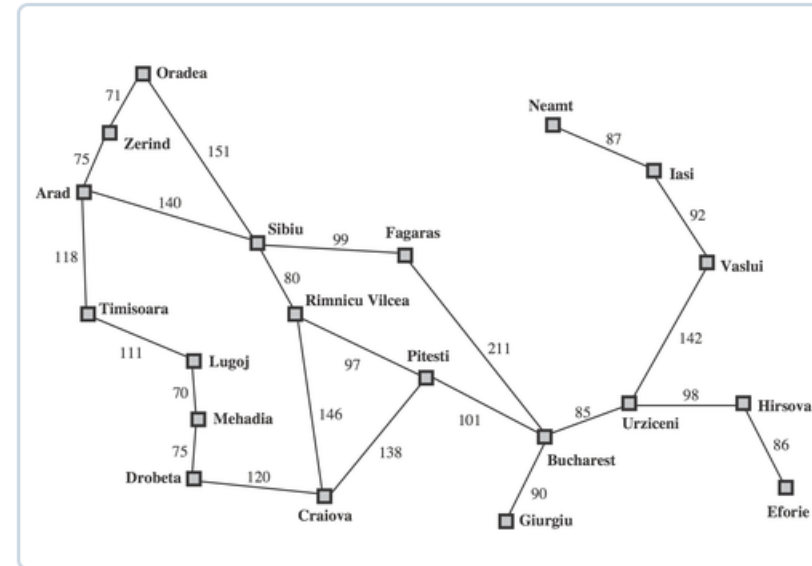
A → ['AS', 'AT', 'AZ']
explored: ['A']

AZ → [..., 'AZA', 'AZO']
explored: ['A', 'Z']

AZO → [..., 'AZOS', 'AZOZ']
explored: ['A', 'O', 'Z']

AZOS → [..., 'AZOSR']
explored: ['A', 'O', 'S', 'Z']

Search Tree



LIFO follows the newest path.

DFS-GSA Practice

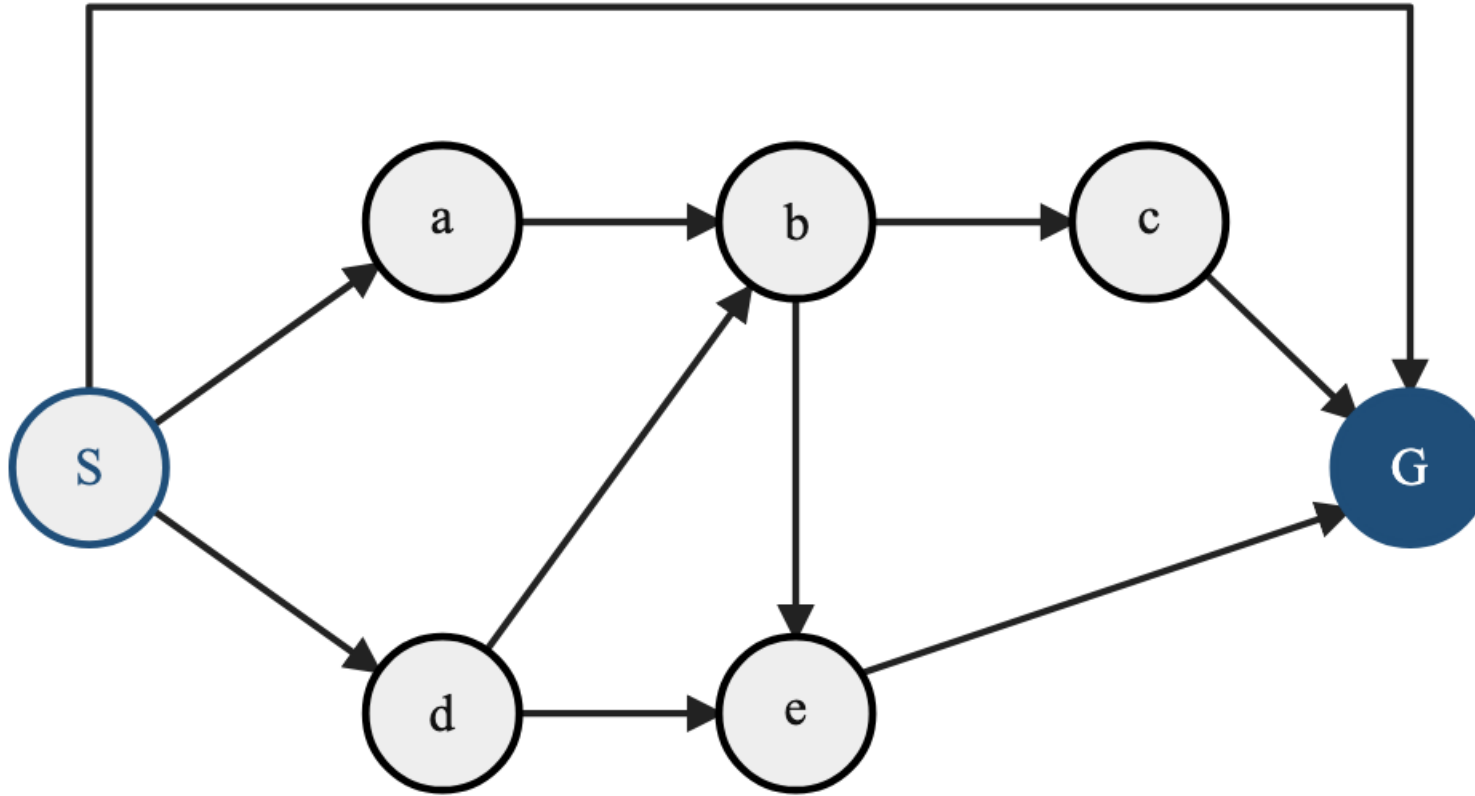
TASK

Run DFS-GSA from **S** to **G**

Record Explored order

Returned path

Add successors alphabetically



DFS-TSA Properties

TASK

Complete?

Optimal?

Time complexity?

Space complexity?

DFS-TSA Properties

Complete?

Optimal?

Time complexity?

Space complexity?

BFS vs. DFS

TASK

BFS

When can it outperform DFS?

DFS

When can it outperform BFS?

BFS vs. DFS

BFS

When can it outperform DFS?

DFS

When can it outperform BFS?

TSA vs. GSA

TSA

GSA

Memory

Repeated state

Trade-off

TSA vs. GSA

TSA

GSA

Memory

Frontier

Repeated state

Trade-off

TSA vs. GSA

TSA

GSA

Memory

Frontier

Frontier + explored set

Repeated state

Trade-off

TSA vs. GSA

TSA

GSA

Memory

Frontier

Frontier + explored set

Repeated state

Expand it again

Trade-off

TSA vs. GSA

TSA

GSA

Memory

Frontier

Frontier + **explored set**

Repeated state

Expand it again

Skip it

Trade-off

TSA vs. GSA

TSA

GSA

Memory

Frontier

Frontier + **explored set**

Repeated state

Expand it again

Skip it

Trade-off

Less memory

TSA vs. GSA

TSA

GSA

Memory

Frontier

Frontier + explored set

Repeated state

Expand it again

Skip it

Trade-off

Less memory

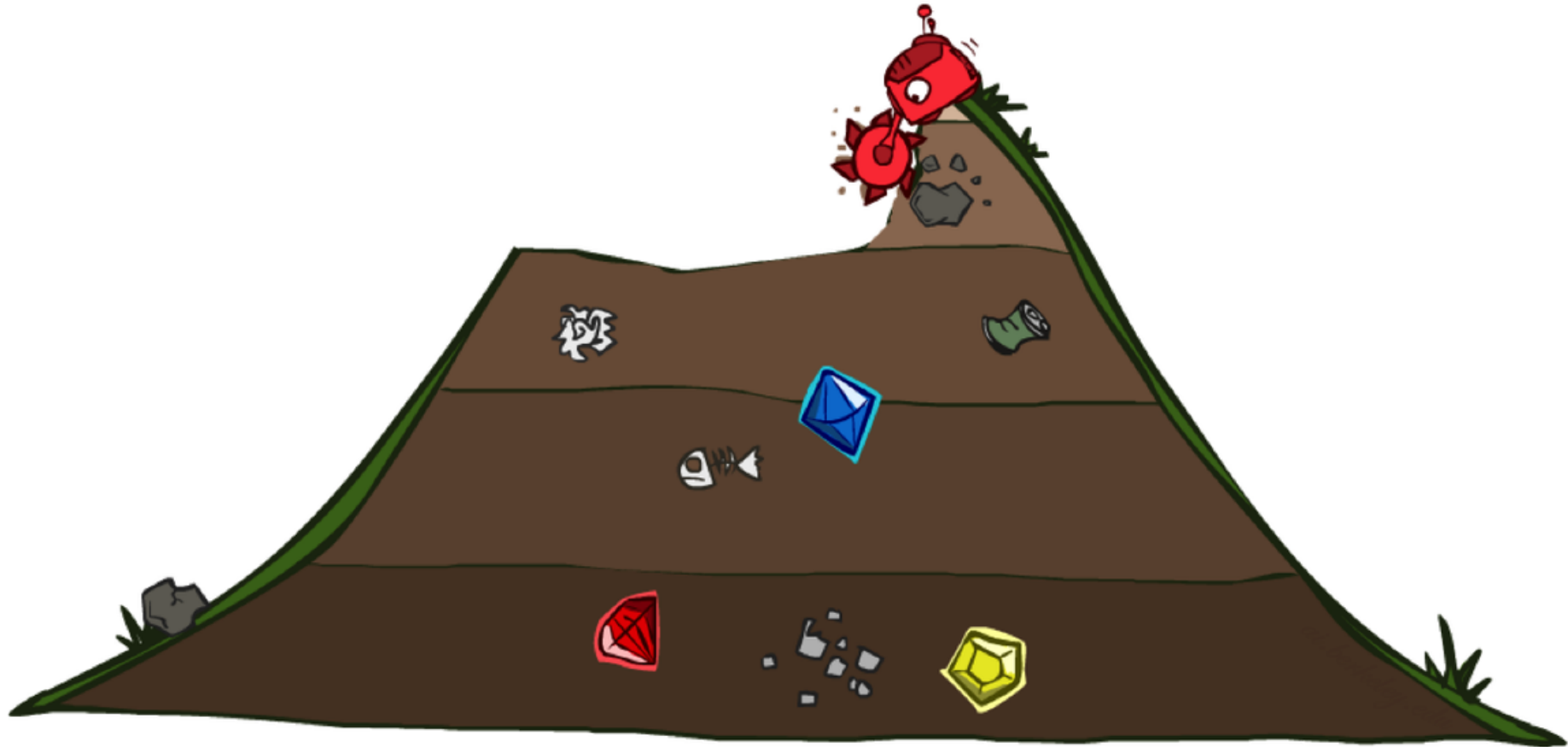
Fewer repeated expansions

TSA vs. GSA

	TSA	GSA
Memory	Frontier	Frontier + explored set
Repeated state	Expand it again	Skip it
Trade-off	Less memory	Fewer repeated expansions

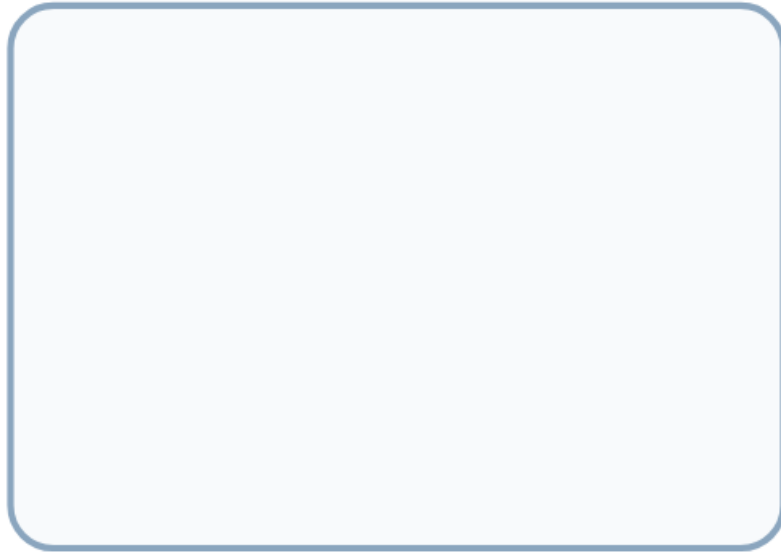
BFS, DFS, and UCS can each use either algorithm

UCS Explores the Cheapest Node First



Frontier

smallest total path cost first



PY main.py

```
1 from heapq import heappop, heappush
2 frontier = []
3 heappush(frontier, (140, "AS"))
4 heappush(frontier, (118, "AT"))
5 heappush(frontier, (75, "AZ"))
6 print("frontier:", frontier)
7 while frontier:
8     print("remove:", heappop(frontier))
```

Download: comp3270.dirk.hk/priority_queue_demo.py

Frontier

smallest total path cost first

AS 140

`heappush(frontier, (140, "AS"))`

PY main.py

```
1 from heapq import heappop, heappush
2 frontier = []
3 heappush(frontier, (140, "AS"))
4 heappush(frontier, (118, "AT"))
5 heappush(frontier, (75, "AZ"))
6 print("frontier:", frontier)
7 while frontier:
8     print("remove:", heappop(frontier))
```

Download: comp3270.dirk.hk/priority_queue_demo.py

Frontier

smallest total path cost first

AT	118
----	-----

AS	140
----	-----

```
heappush(frontier, (140, "AS"))
```

```
heappush(frontier, (118, "AT"))
```

PY main.py

```
1 from heapq import heappop, heappush
2 frontier = []
3 heappush(frontier, (140, "AS"))
4 heappush(frontier, (118, "AT"))
5 heappush(frontier, (75, "AZ"))
6 print("frontier:", frontier)
7 while frontier:
8     print("remove:", heappop(frontier))
```

Download: comp3270.dirk.hk/priority_queue_demo.py

Frontier

smallest total path cost first

AZ	75
AT	118
AS	140

```
heappush(frontier, (140, "AS"))  
heappush(frontier, (118, "AT"))  
heappush(frontier, (75, "AZ"))
```

PY main.py

```
1 from heapq import heappop, heappush  
2 frontier = []  
3 heappush(frontier, (140, "AS"))  
4 heappush(frontier, (118, "AT"))  
5 heappush(frontier, (75, "AZ"))  
6 print("frontier:", frontier)  
7 while frontier:  
8     print("remove:", heappop(frontier))
```

Download: comp3270.dirk.hk/priority_queue_demo.py

Frontier

smallest total path cost first

AT	118
AS	140
AZ	75

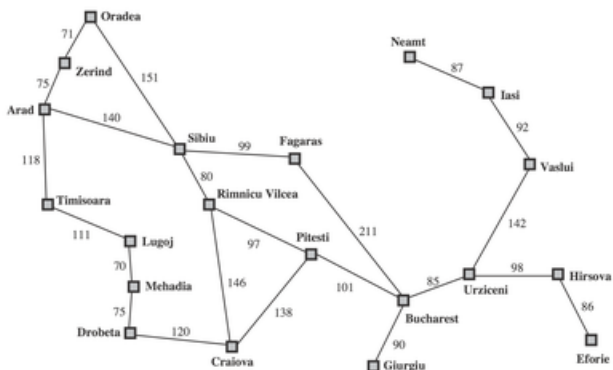
removed first

heappop(frontier)
returns (75, "AZ")

PY main.py

```
1 from heapq import heappop, heappush
2 frontier = []
3 heappush(frontier, (140, "AS"))
4 heappush(frontier, (118, "AT"))
5 heappush(frontier, (75, "AZ"))
6 print("frontier:", frontier)
7 while frontier:
8     print("remove:", heappop(frontier))
```

Download: comp3270.dirk.hk/priority_queue_demo.py



frontier: (cost, path)

explored: states

PY main.py

PY romania.py

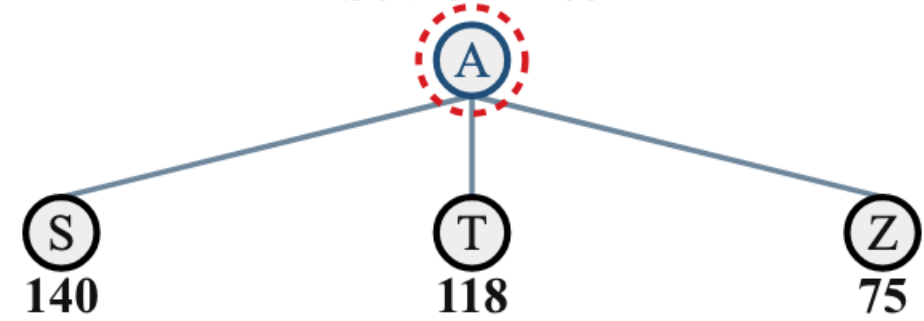
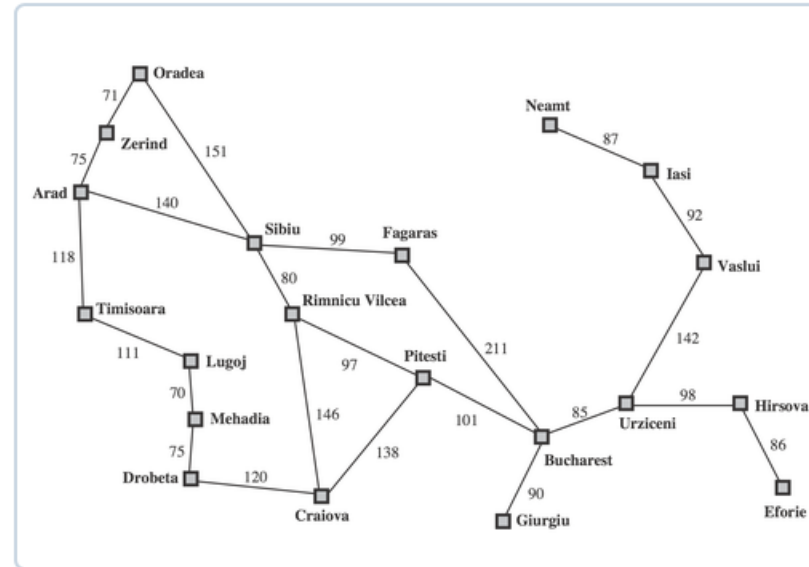
```

1 from heapq import heappop, heappush
2 from romania import romania
3 def ucs_gsa(graph, start, goal, trace=False):
4     frontier = [(0, start)]
5     explored = set()
6     explored_order = []
7     while frontier:
8         cost, path = heappop(frontier)
9         state = path[-1]
10        if state == goal: return cost, path, explored_order
11        if state in explored: continue
12        explored.add(state)
13        explored_order.append(state)
14        for step_cost, child in graph[state]:
15            heappush(frontier, (cost + step_cost, path + child))
16        if trace:
17            print(f"{path} @ {cost} -> {sorted(frontier)}")
18            print(f"explored: {sorted(explored)}\n")
19    cost, path, order = ucs_gsa(romania, "A", "B", True)
20    print("solution:", (cost, path))
21    print("explored order:", " -> ".join(order))
  
```


Program Output

```
A @ 0 -> [(75, 'AZ'), (118, 'AT'),  
(140, 'AS')]  
explored: ['A']
```

Search Tree

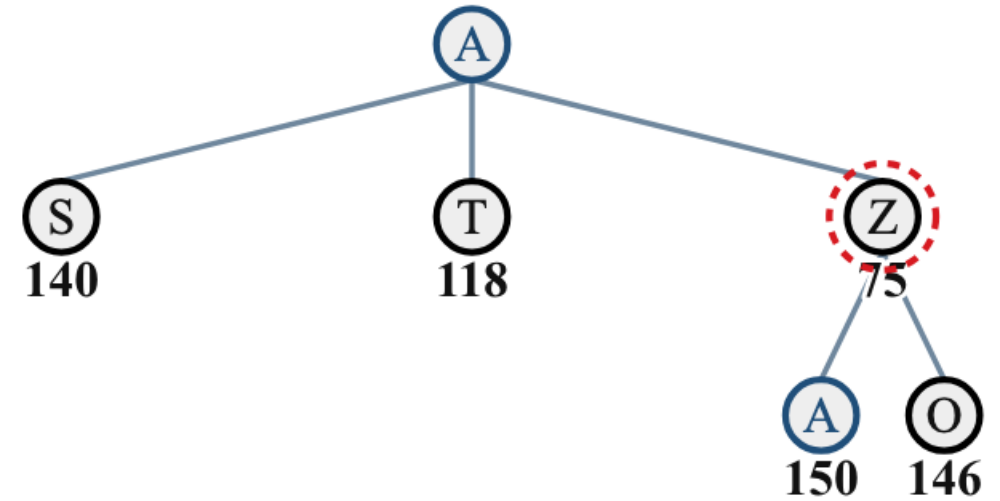
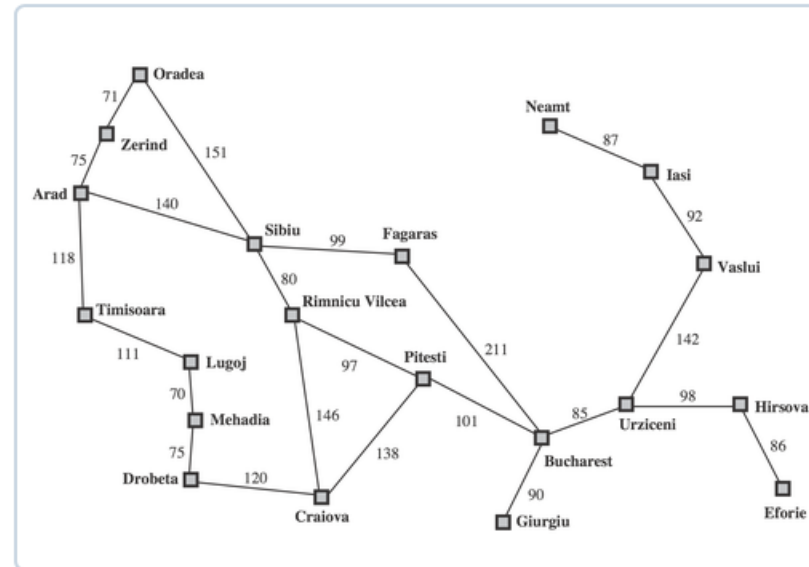


Program Output

```
A @ 0 -> [(75, 'AZ'), (118, 'AT'),  
(140, 'AS')]  
explored: ['A']
```

```
AZ @ 75 -> [(118, 'AT'), (140, 'AS'),  
...]  
explored: ['A', 'Z']
```

Search Tree



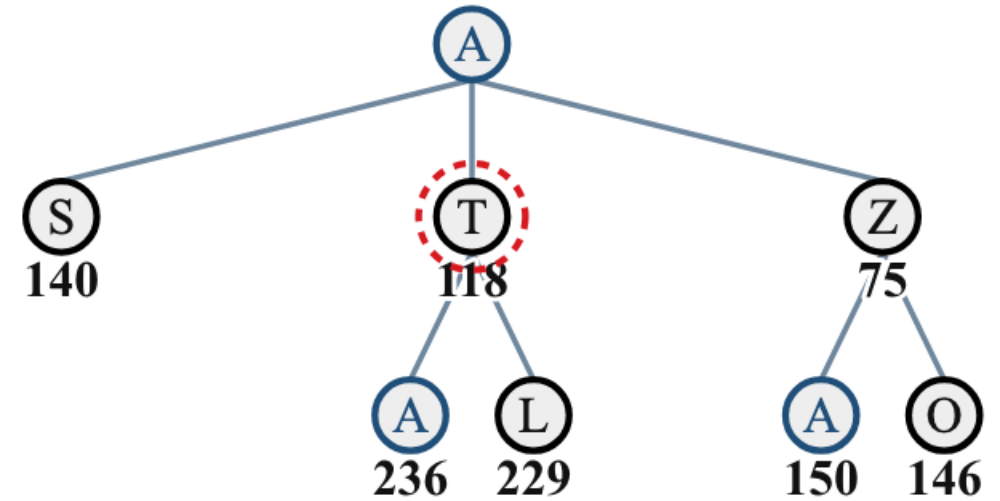
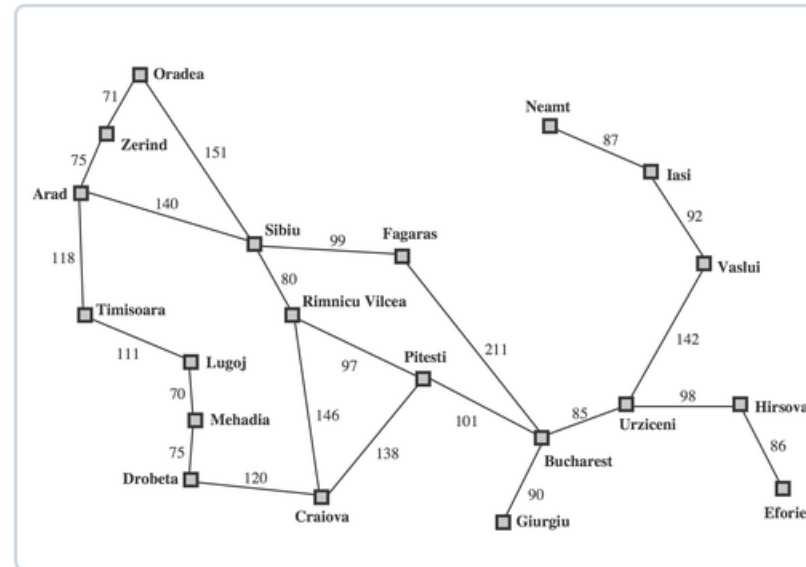
Program Output

A @ 0 → [(75, 'AZ'), (118, 'AT'),
(140, 'AS')]
explored: ['A']

AZ @ 75 → [(118, 'AT'), (140, 'AS'),
...]
explored: ['A', 'Z']

AT @ 118 → [(140, 'AS'), (146, 'AZO'),
...]
explored: ['A', 'T', 'Z']

Search Tree



Program Output

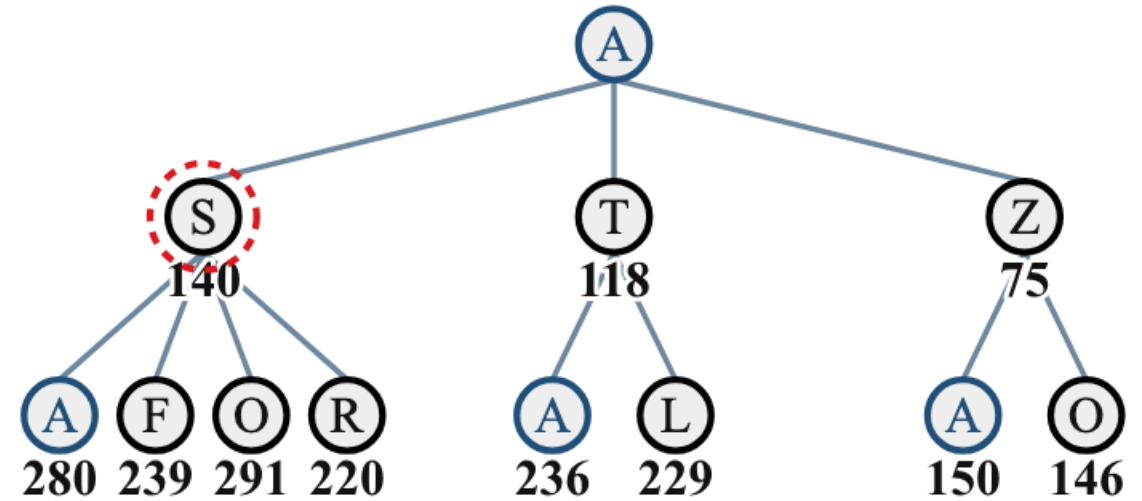
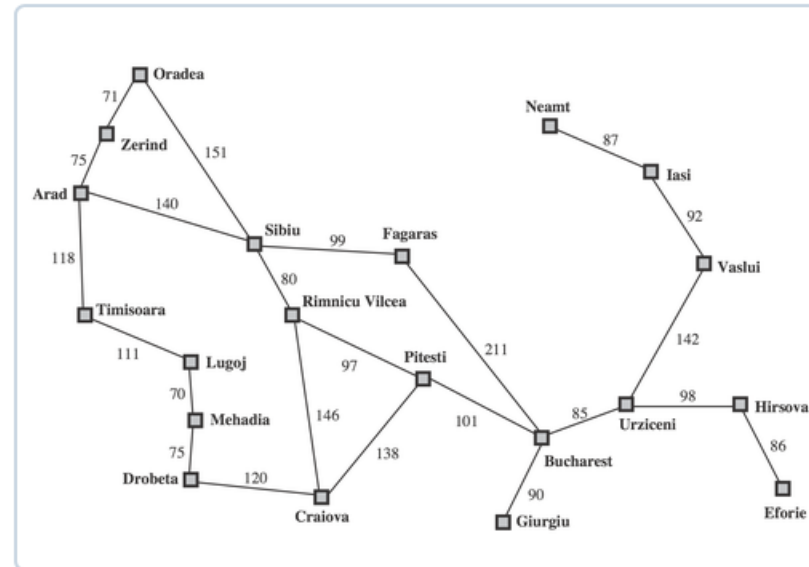
A @ 0 → [(75, 'AZ'), (118, 'AT'),
(140, 'AS')]
explored: ['A']

AZ @ 75 → [(118, 'AT'), (140, 'AS'),
...]
explored: ['A', 'Z']

AT @ 118 → [(140, 'AS'), (146, 'AZO'),
...]
explored: ['A', 'T', 'Z']

AS @ 140 → [(146, 'AZO'), (150,
'AZA'), ...]
explored: ['A', 'S', 'T', 'Z']

Search Tree



Program Output

A @ 0 → [(75, 'AZ'), (118, 'AT'), (140, 'AS')]
explored: ['A']

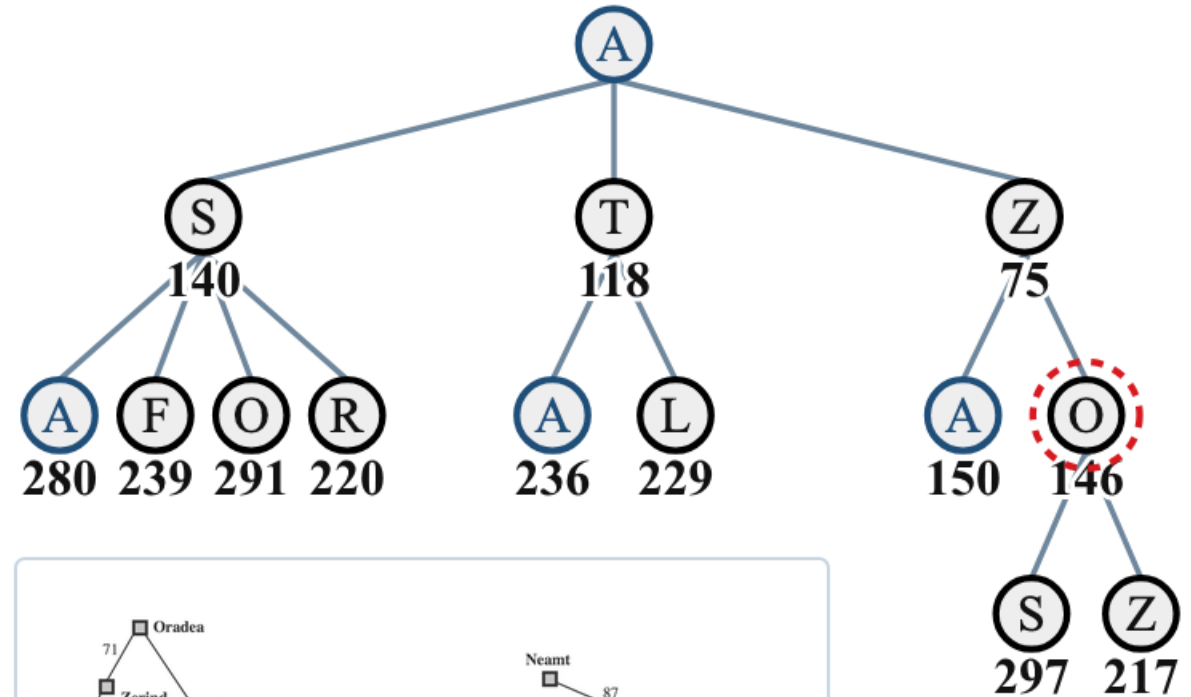
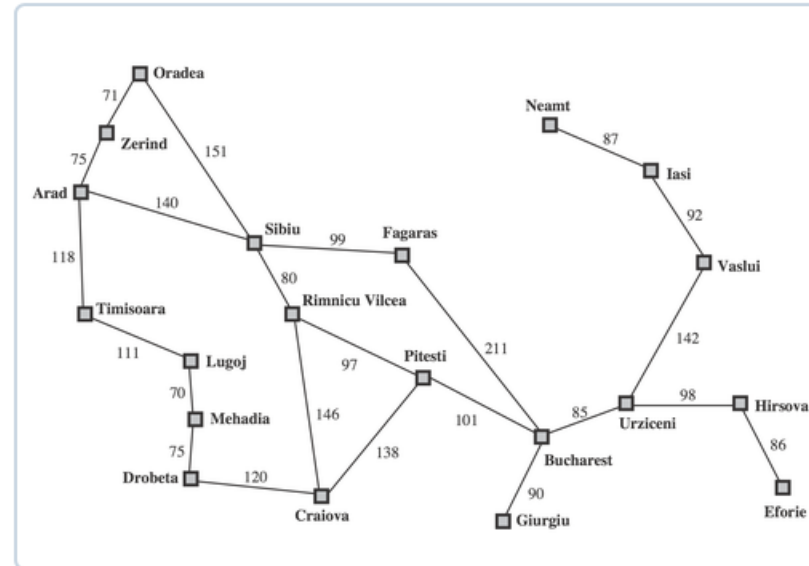
AZ @ 75 → [(118, 'AT'), (140, 'AS'), ...]
explored: ['A', 'Z']

AT @ 118 → [(140, 'AS'), (146, 'AZO'), ...]
explored: ['A', 'T', 'Z']

AS @ 140 → [(146, 'AZO'), (150, 'AZA'), ...]
explored: ['A', 'S', 'T', 'Z']

AZO @ 146 → [(150, 'AZA'), (217, 'AZOZ'), ...]
explored: ['A', 'O', 'S', 'T', 'Z']

Search Tree



Program Output

```
A @ 0 -> [(75, 'AZ'), (118, 'AT'),
(140, 'AS')]
explored: ['A']
```

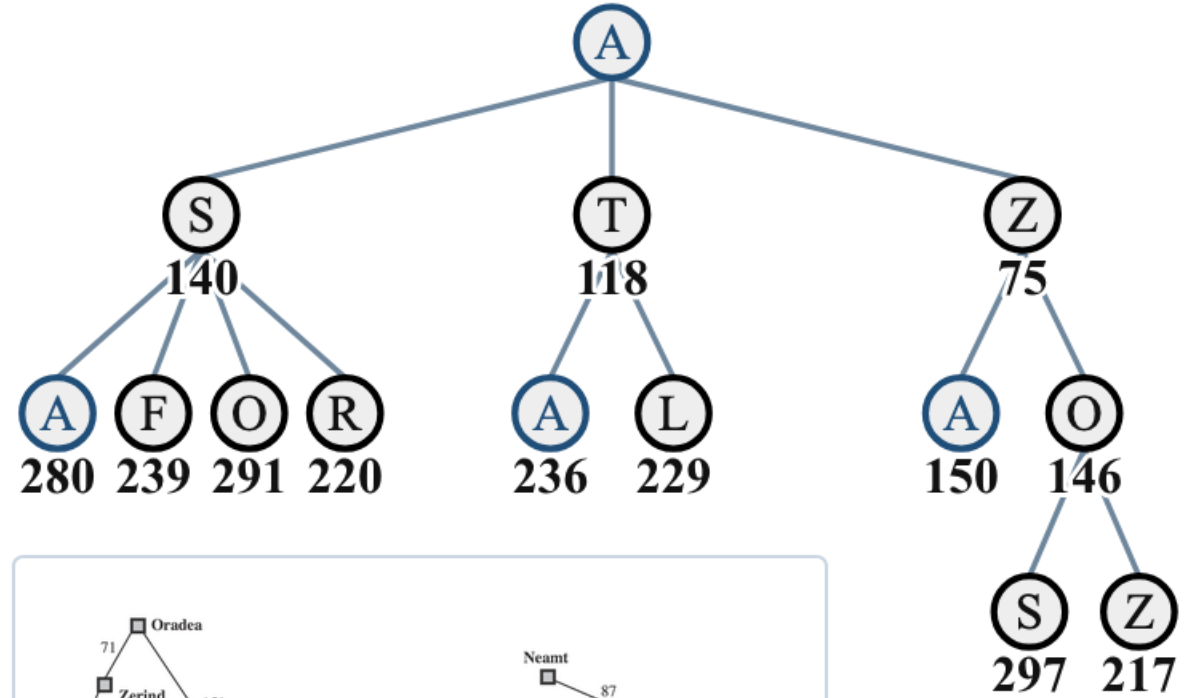
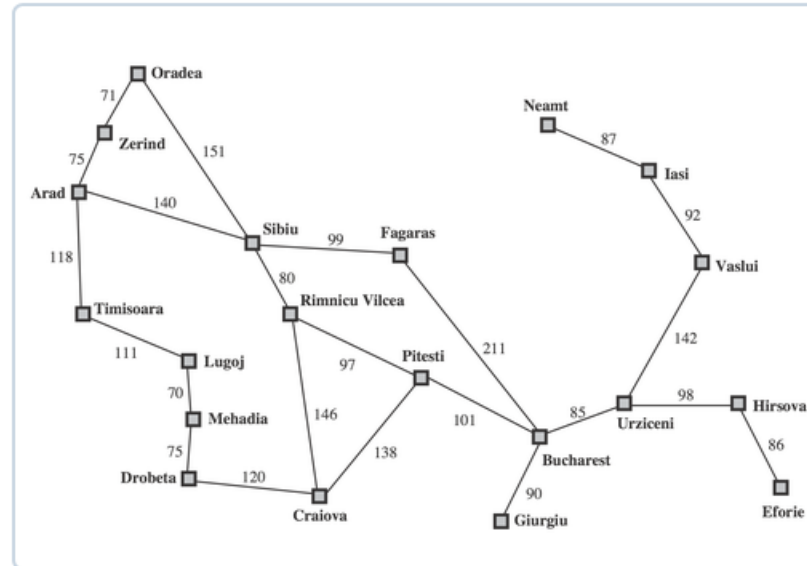
```
AZ @ 75 -> [(118, 'AT'), (140, 'AS'),
...]
explored: ['A', 'Z']
```

```
AT @ 118 -> [(140, 'AS'), (146, 'AZO'), ...]  
explored: ['A', 'T', 'Z']
```

```
AS @ 140 -> [(146, 'AZ0'), (150, 'AZA'), ...]  
explored: ['A', 'S', 'T', 'Z']
```

```
AZ0 @ 146 -> [(150, 'AZA'), (217, 'AZ0Z'), ...]  
explored: ['A', '0', 'S', 'T', 'Z']
```

Search Tree



Next path = smallest total path cost

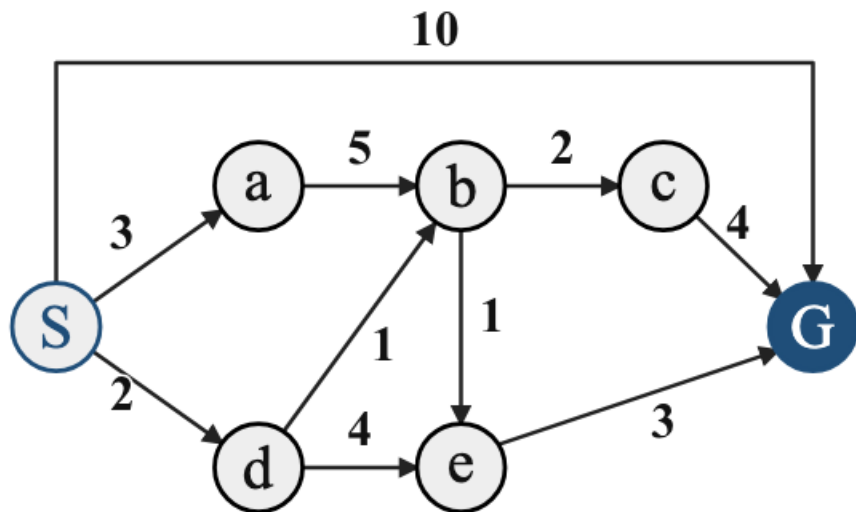
UCS-GSA Practice

TASK

Run UCS-GSA from **S** to **G**

Record Explored order
 Returned path and cost

Add successors alphabetically

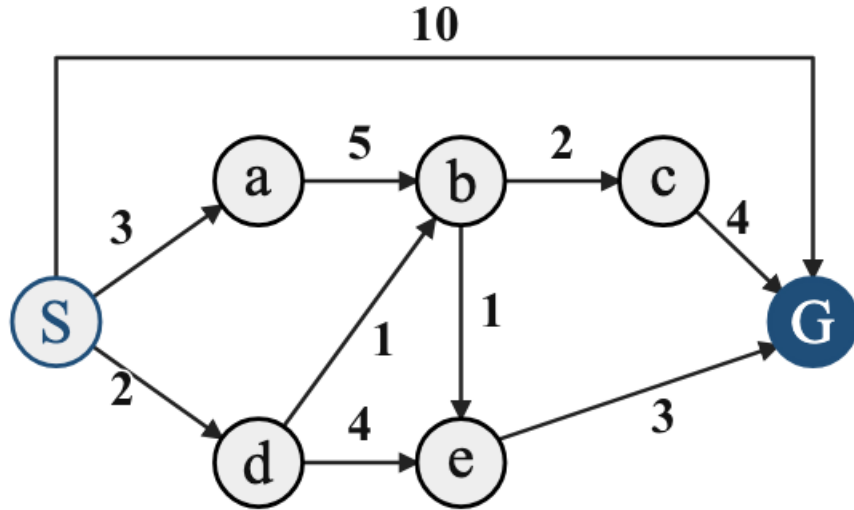


UCS-GSA Practice

Run UCS-GSA from **S** to **G**

Record Explored order
 Returned path and cost

Add successors alphabetically



UCS-TSA Properties

TASK

Complete?

Optimal?

Time complexity?

Space complexity?

UCS-TSA Properties

Complete?

Optimal?

Time complexity?

Space complexity?

Uninformed Search Strategies

Strategy	Frontier	Next path	Optimal?
BFS	FIFO queue	Shallowest	Only with equal step costs

Uninformed Search Strategies

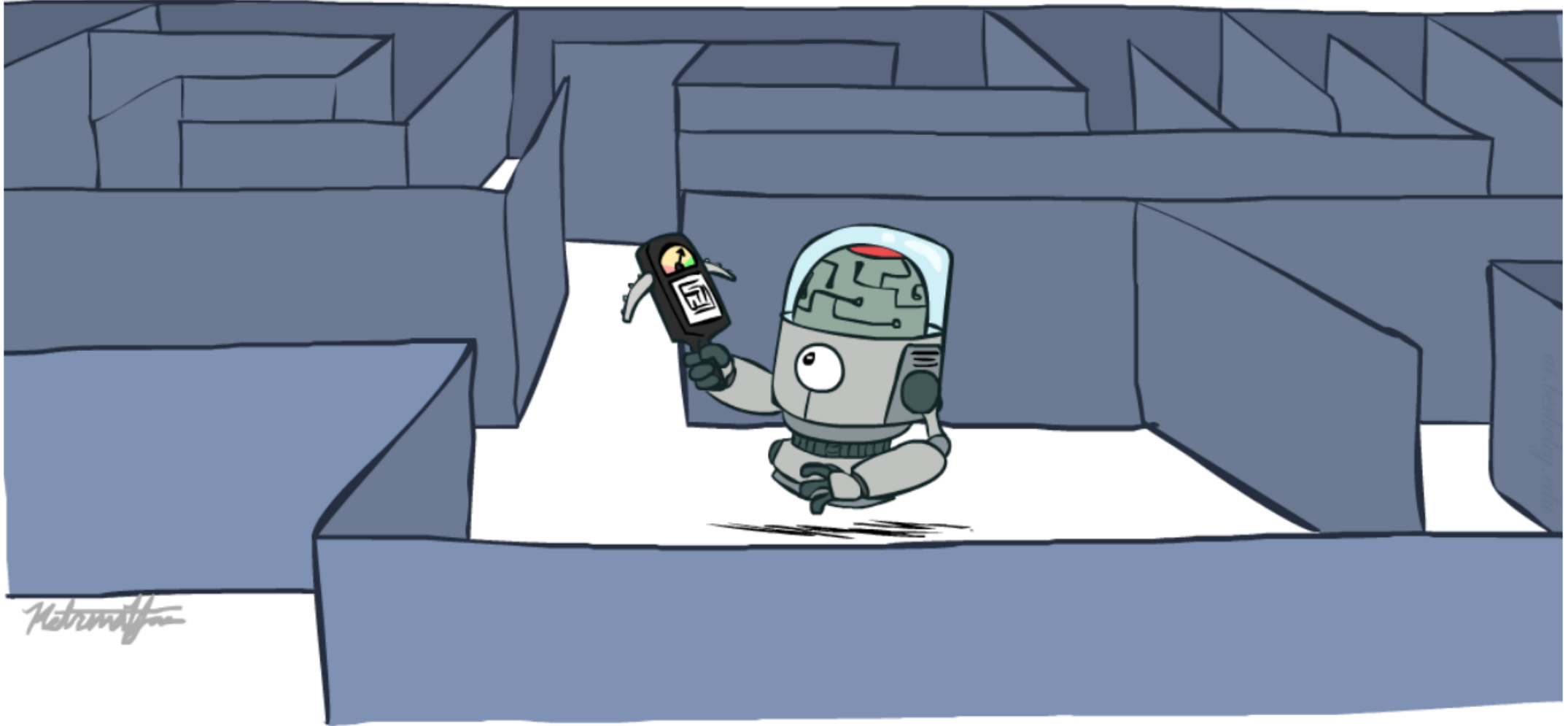
Strategy	Frontier	Next path	Optimal?
BFS	FIFO queue	Shallowest	Only with equal step costs
DFS	LIFO stack	Deepest	No

Uninformed Search Strategies

Strategy	Frontier	Next path	Optimal?
BFS	FIFO queue	Shallowest	Only with equal step costs
DFS	LIFO stack	Deepest	No
UCS	Priority queue	Cheapest	Yes

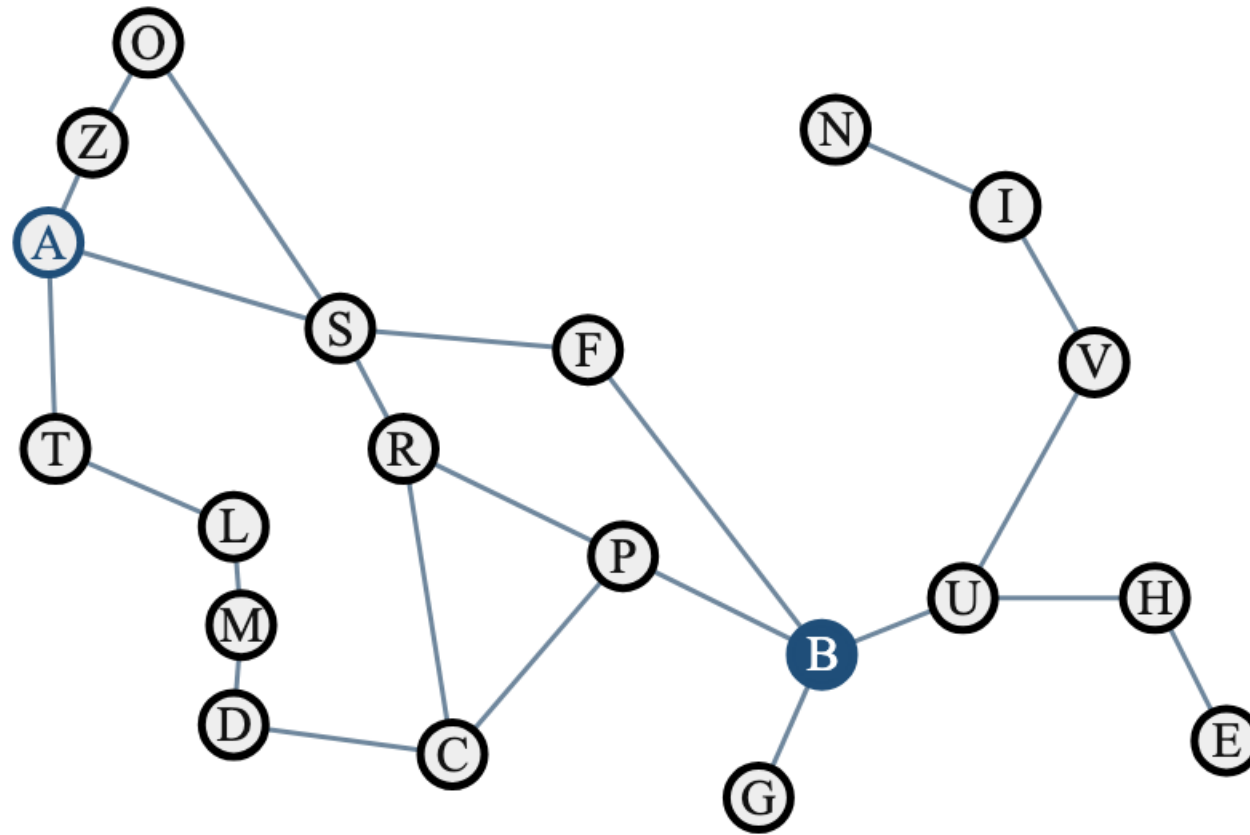
UCS uses only costs already incurred: $g(\text{path})$

Informed Search



A Heuristic Estimates the Remaining Cost

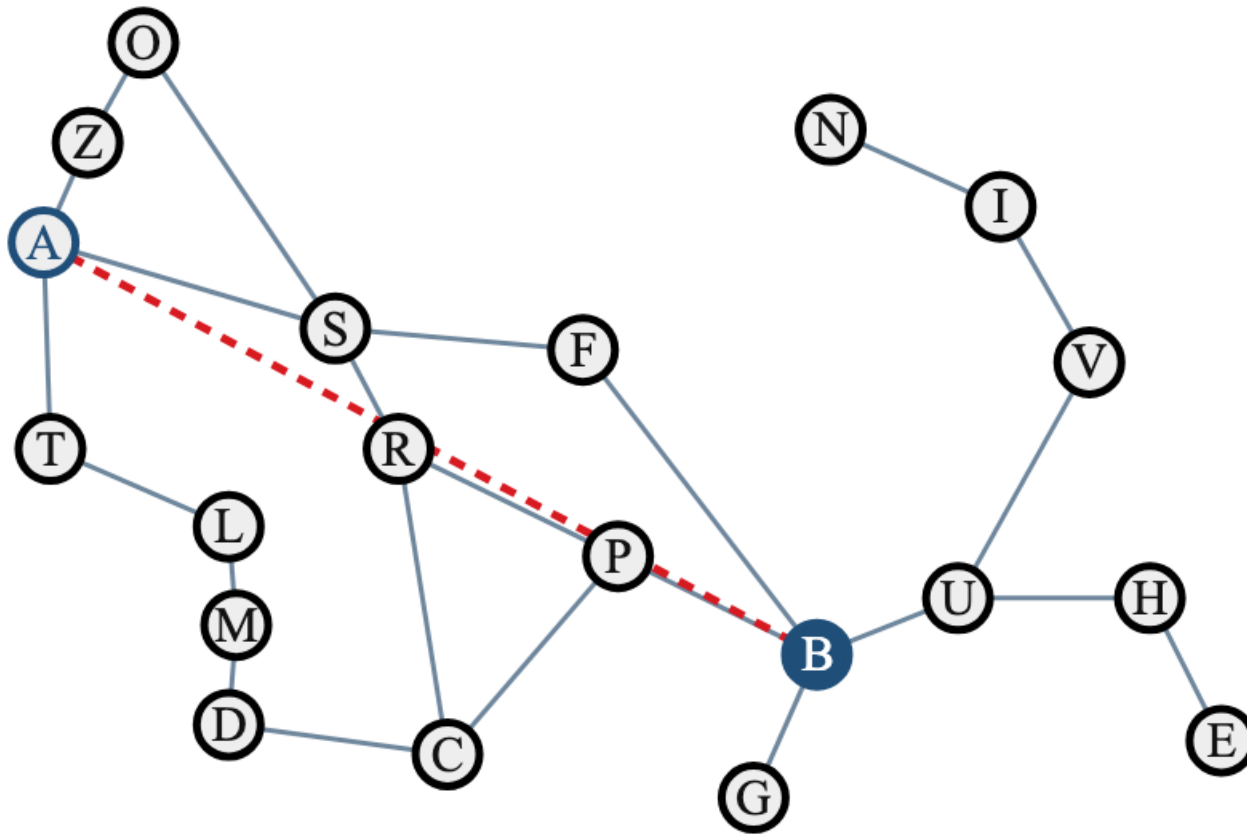
How far is a node from the goal?



$h(n)$: Estimated cost of the cheapest path from the state at node n to a goal

A Heuristic Estimates the Remaining Cost

How far is a node from the goal?



$h(n)$: Estimated cost of the cheapest path from the state at node n to a goal

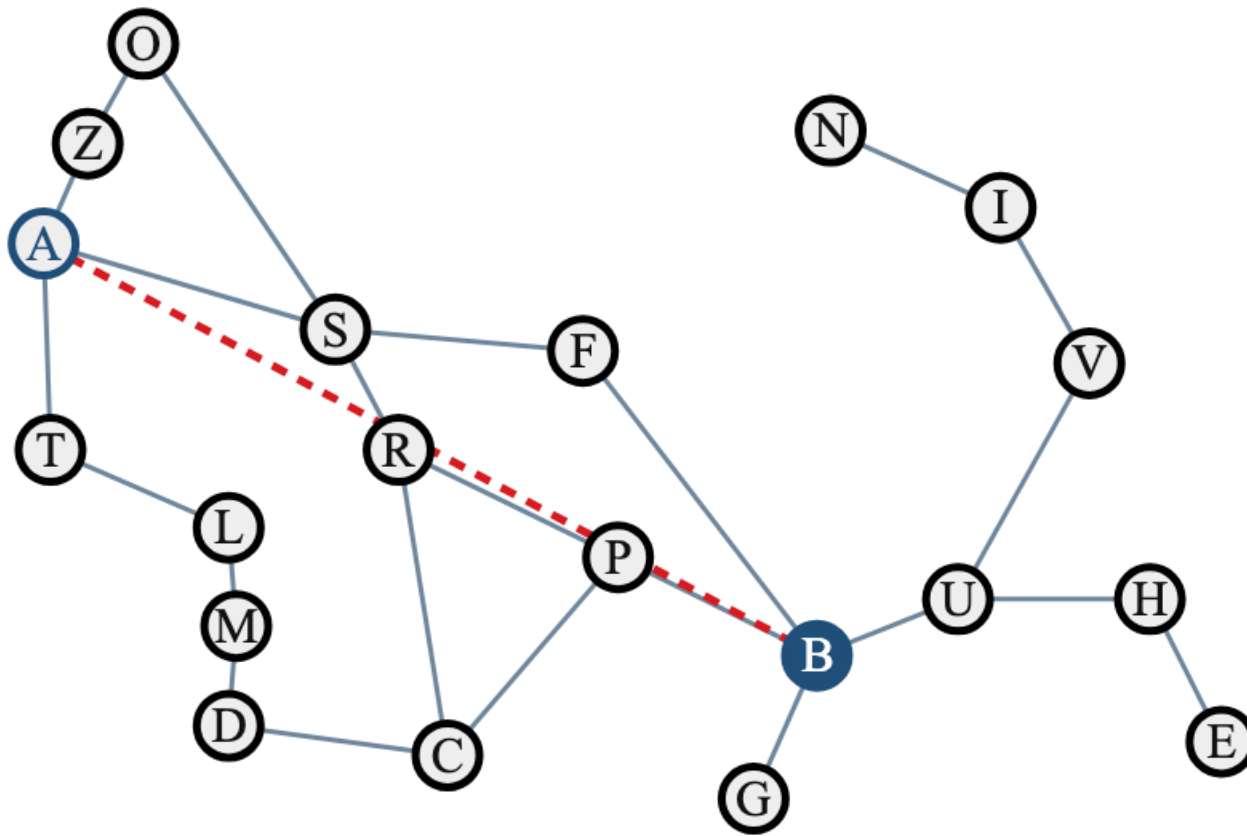
Straight-line distance to B

$h(A) = 366 \text{ km}$

An estimate, not a road route

A Heuristic Estimates the Remaining Cost

How far is a node from the goal?



$h(n)$: Estimated cost of the cheapest path from the state at node n to a goal

Straight-line distance to B

$h(A) = 366$ km

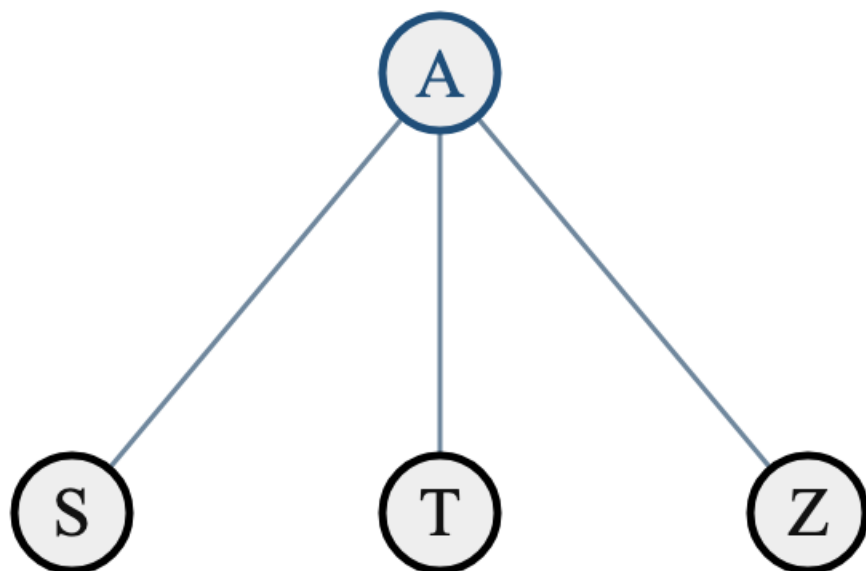
An estimate, not a road route

$h(B) = 0$

Use the estimate to guide which frontier node to explore next

Greedy Search Selects the Lowest $h(n)$

Expand the frontier node estimated closest to the goal



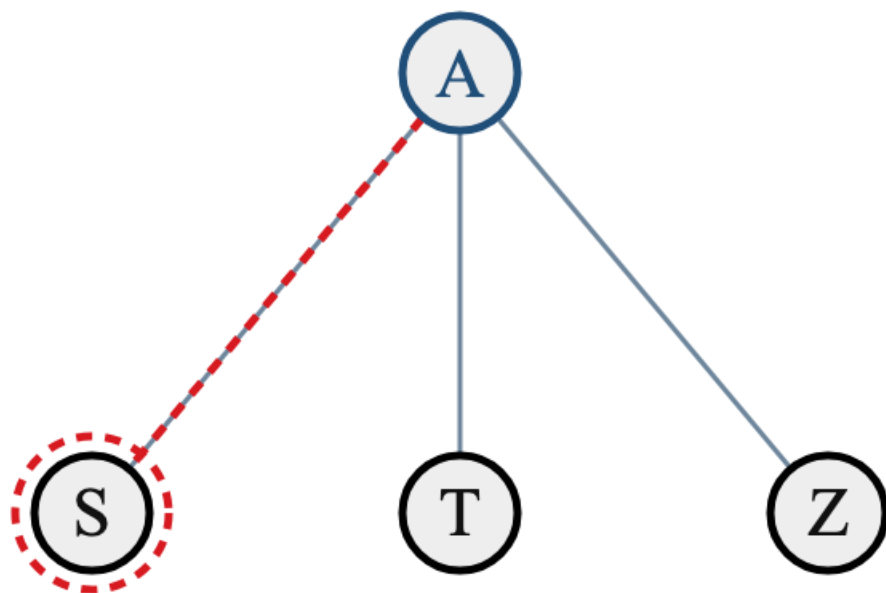
Frontier after expanding A

Priority queue · smallest h first

Path	Priority
AS	$h(S) = 253$
AT	$h(T) = 329$
AZ	$h(Z) = 374$

Greedy Search Selects the Lowest $h(n)$

Expand the frontier node estimated closest to the goal



Frontier after expanding A

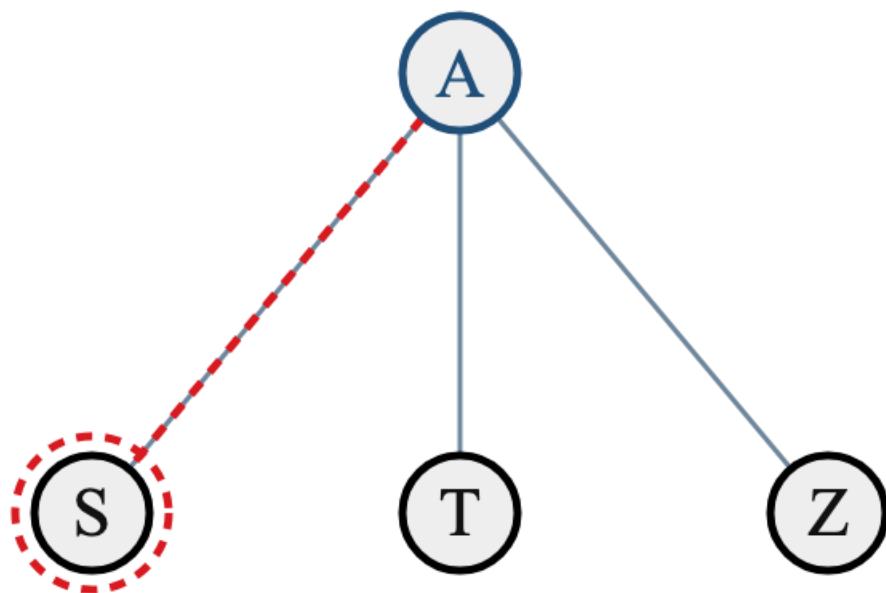
Priority queue · smallest h first

Path	Priority
AS	$h(S) = 253$
AT	$h(T) = 329$
AZ	$h(Z) = 374$

Remove AS; explore S next

Greedy Search Selects the Lowest $h(n)$

Expand the frontier node estimated closest to the goal



Frontier after expanding A

Priority queue · smallest h first

Path	Priority
AS	$h(S) = 253$
AT	$h(T) = 329$
AZ	$h(Z) = 374$

Remove AS; explore S next

Greedy uses the estimated cost remaining; UCS uses the cost so far

Greedy Follows the Heuristic

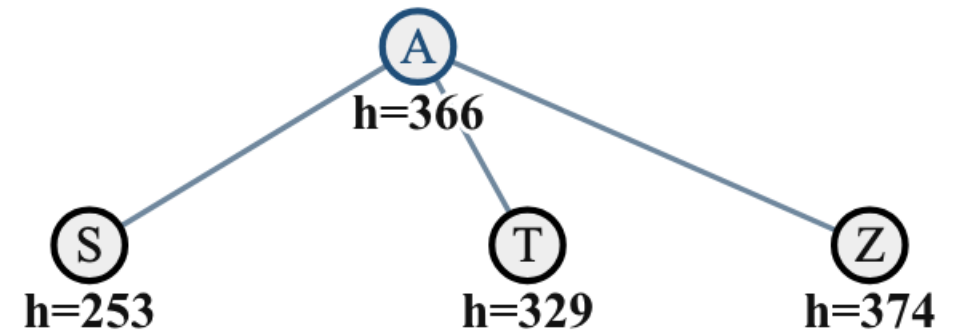
Select the frontier node with the smallest $h(n)$



Search Trace · @ = h(final state)

```
A @ 366 →  
[(253, 'AS'), (329, 'AT'), (374, 'AZ')]  
explored: ['A']
```

Search Tree

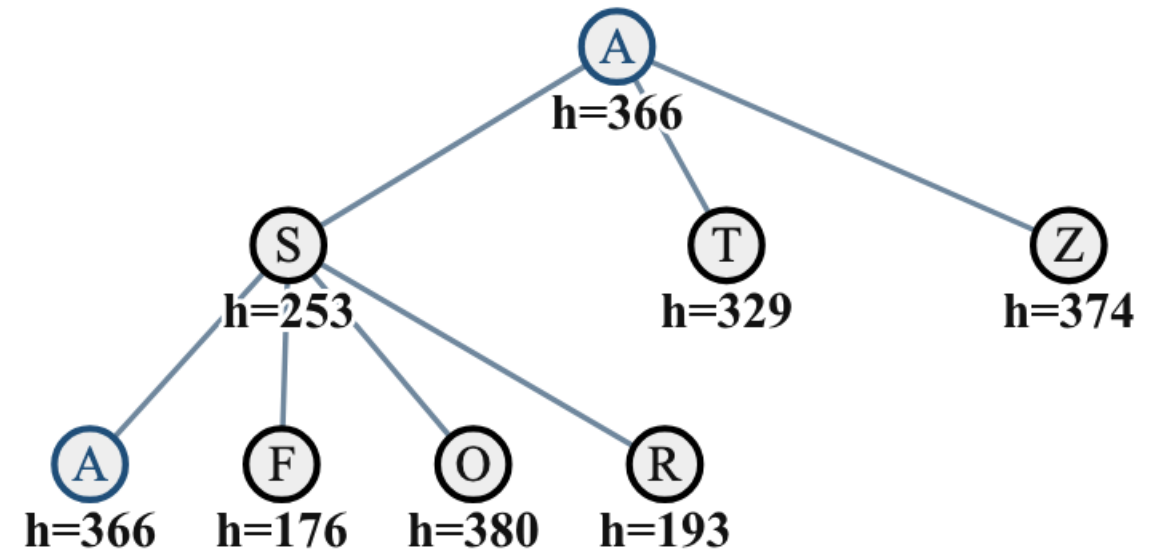


Search Trace · @ = h(final state)

A @ 366 →
[(253, 'AS'), (329, 'AT'), (374, 'AZ')]
explored: ['A']

AS @ 253 →
[(176, 'ASF'), (193, 'ASR'), ...]
explored: ['A', 'S']

Search Tree



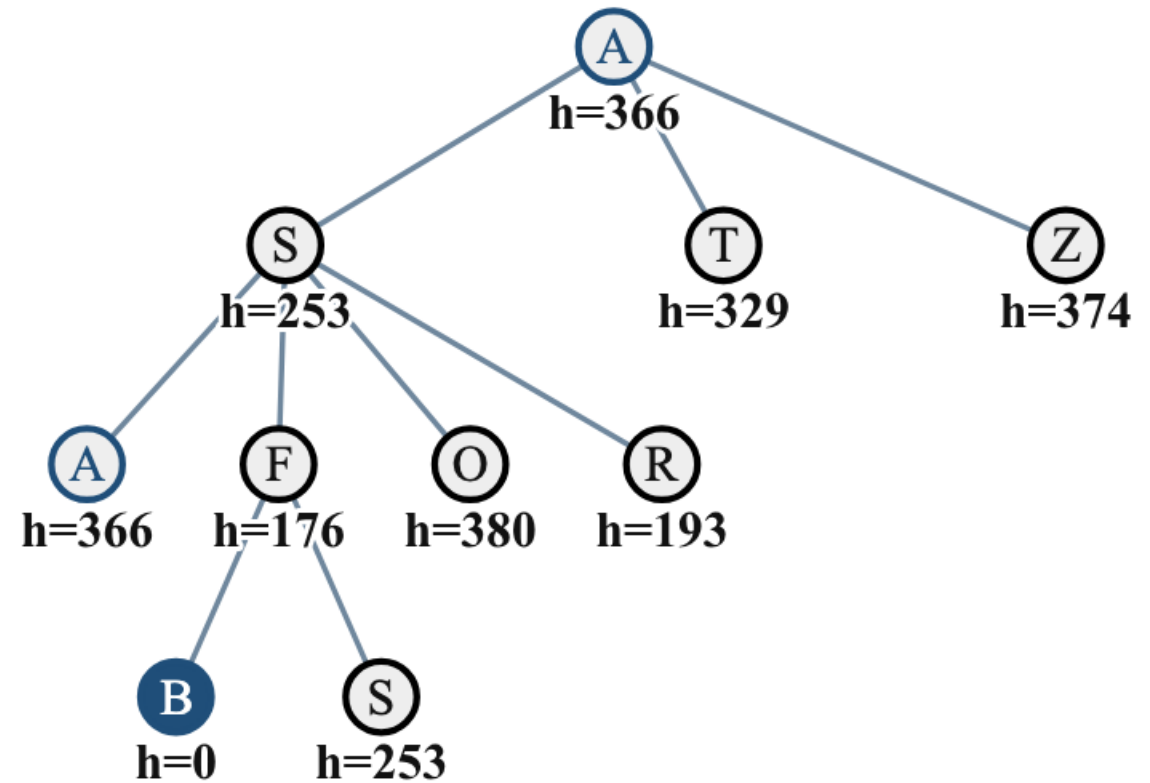
Search Trace · @ = h(final state)

A @ 366 →
[(253, 'AS'), (329, 'AT'), (374, 'AZ')]
explored: ['A']

AS @ 253 →
[(176, 'ASF'), (193, 'ASR'), ...]
explored: ['A', 'S']

ASF @ 176 →
[(0, 'ASFB'), (193, 'ASR'), ...]
explored: ['A', 'F', 'S']

Search Tree



Search Trace · @ = h(final state)

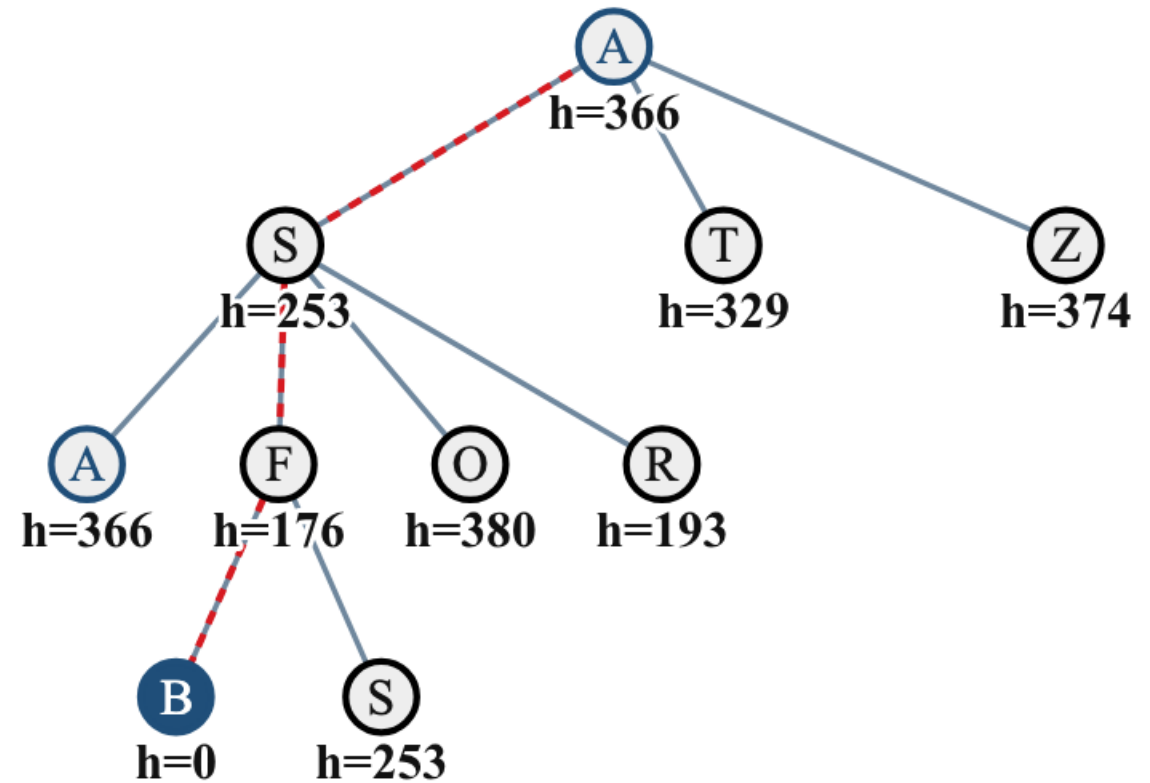
A @ 366 →
[(253, 'AS'), (329, 'AT'), (374, 'AZ')]
explored: ['A']

AS @ 253 →
[(176, 'ASF'), (193, 'ASR'), ...]
explored: ['A', 'S']

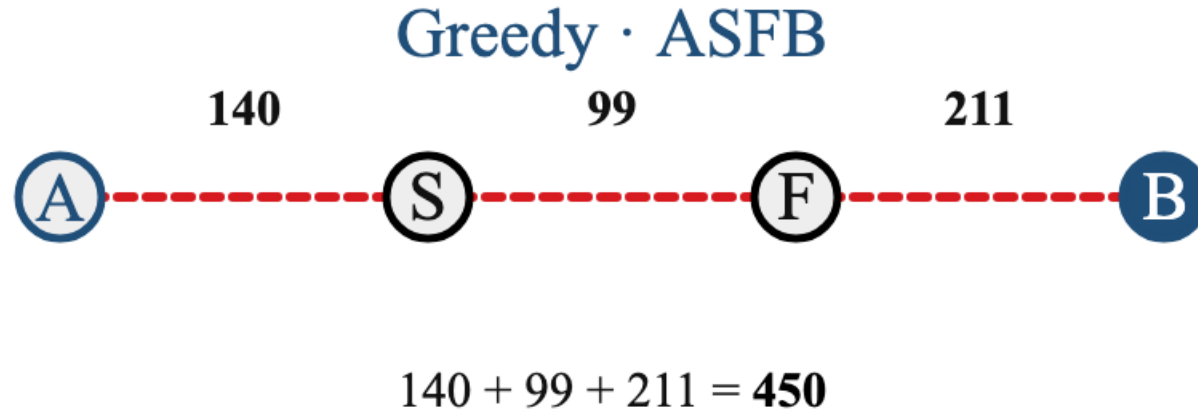
ASF @ 176 →
[(0, 'ASFB'), (193, 'ASR'), ...]
explored: ['A', 'F', 'S']

ASFB @ 0 → goal B
solution: ASFB

Search Tree



Greedy Search Is Not Always Optimal



Greedy Search Is Not Always Optimal

Greedy · ASFB



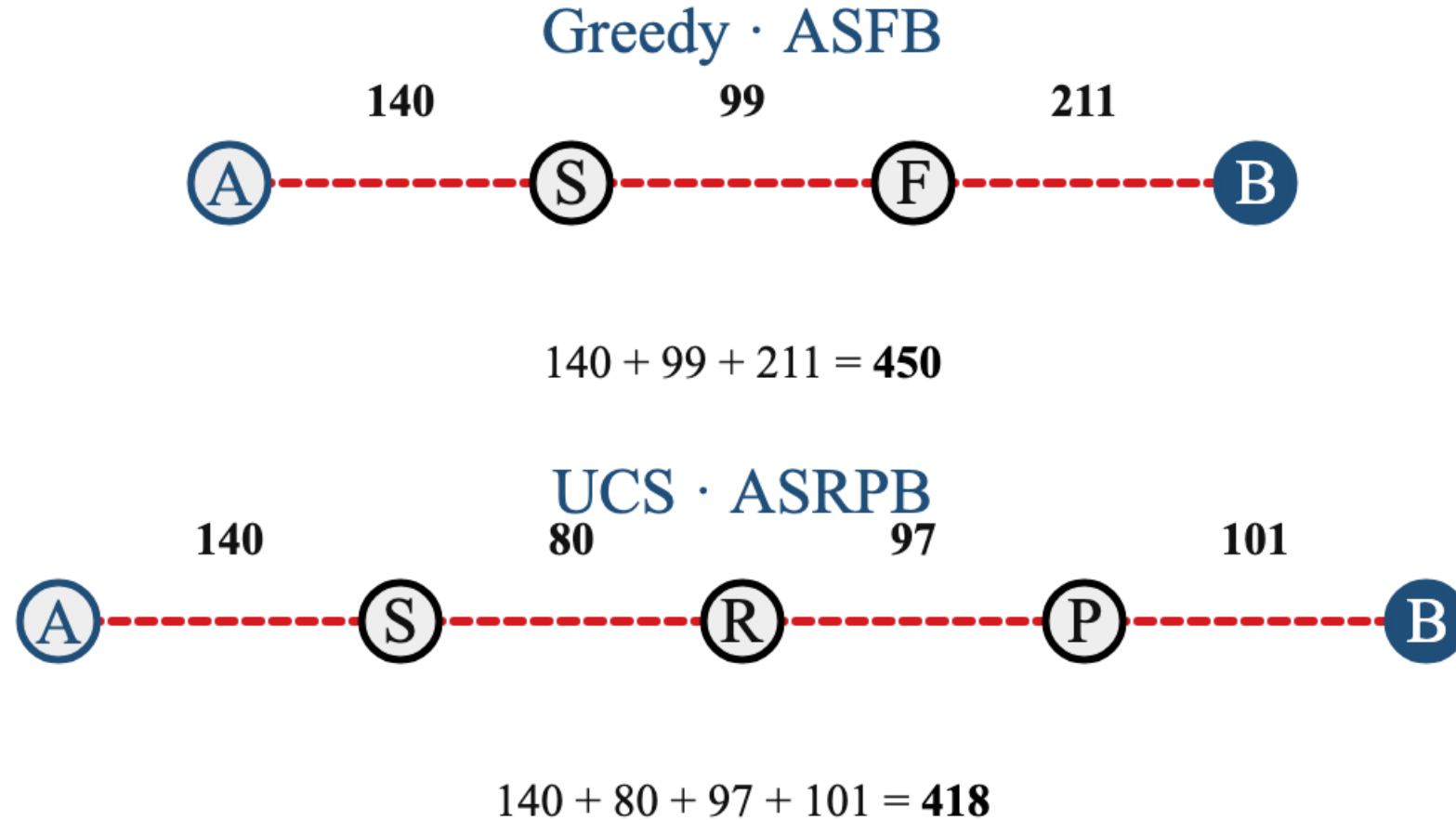
$$140 + 99 + 211 = 450$$

UCS · ASRPB



$$140 + 80 + 97 + 101 = 418$$

Greedy Search Is Not Always Optimal



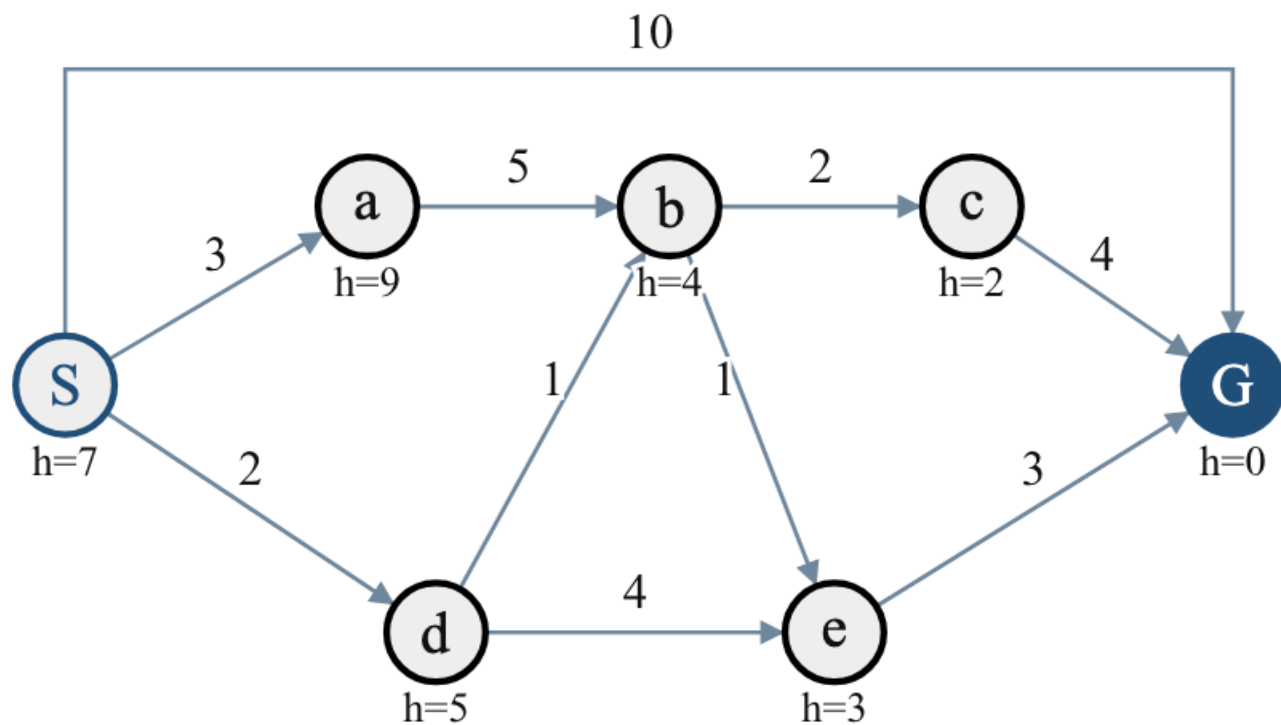
A smaller h-value does not guarantee a cheaper complete path

Greedy-TSA Practice

Start: S · Goal: G

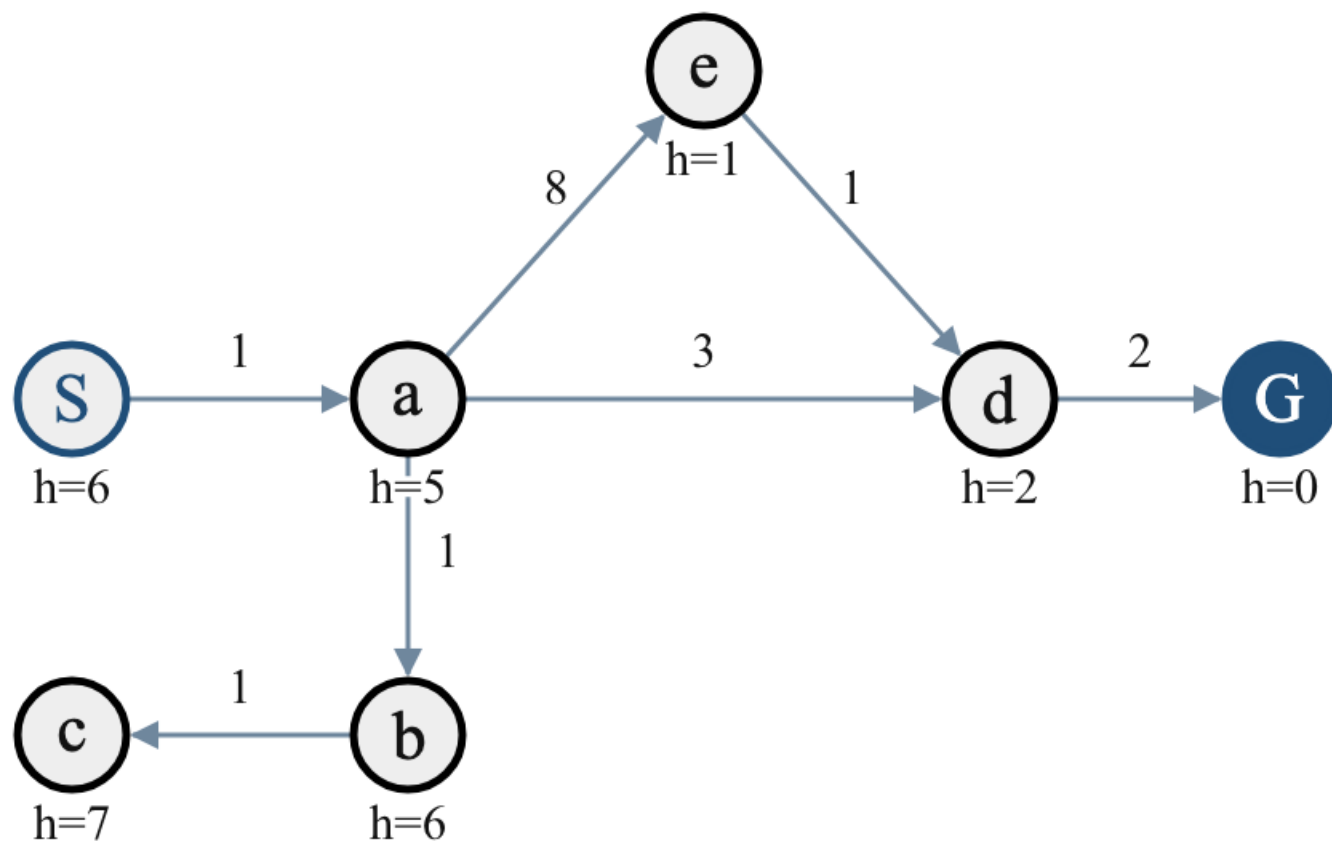
Which states are expanded?

Which path is returned? Is it the cheapest?



A* Motivation: UCS

Cheap so far can still lead away from the goal



After expanding a

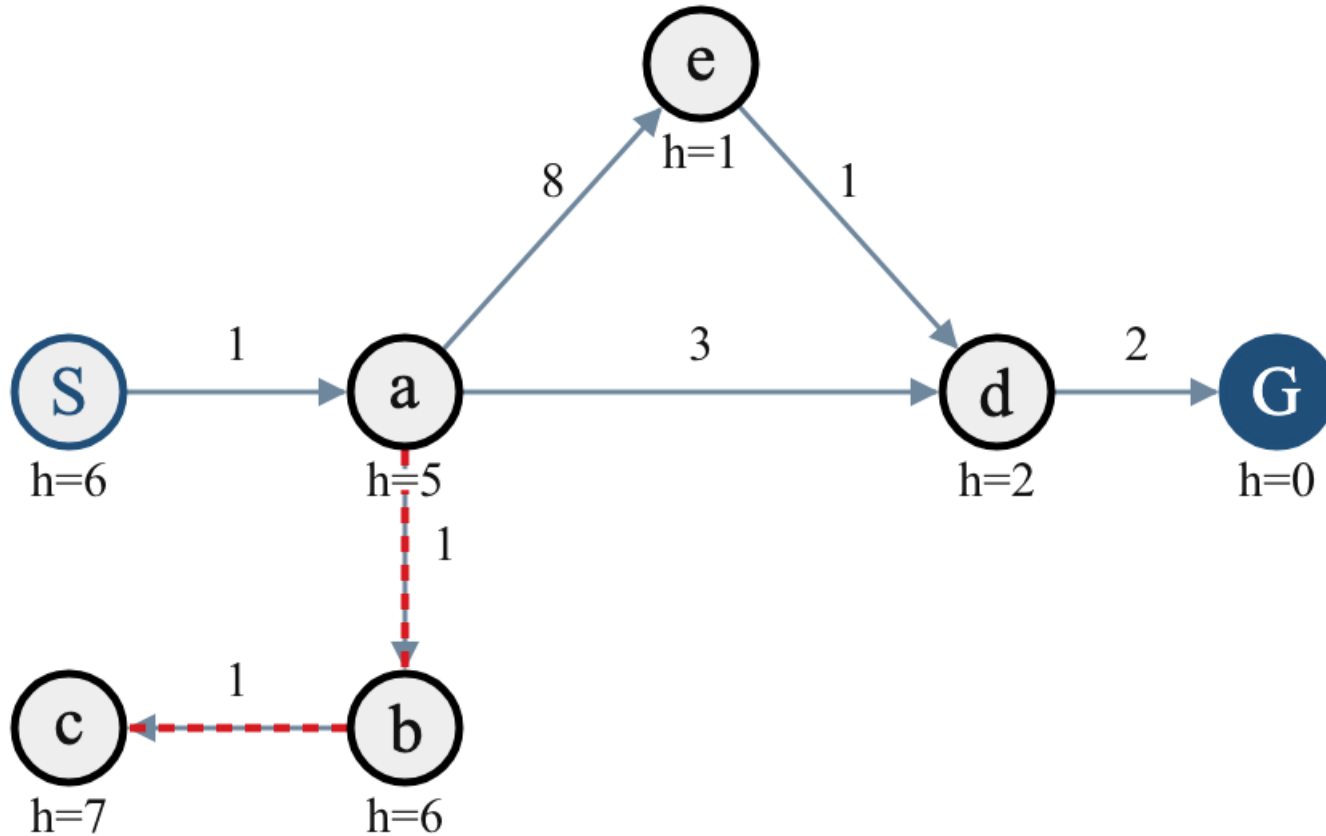
Sab: g=2

Sad: g=4

Sae: g=9

A* Motivation: UCS

Cheap so far can still lead away from the goal



After expanding a

Sab: $g=2$

Sad: $g=4$

Sae: $g=9$

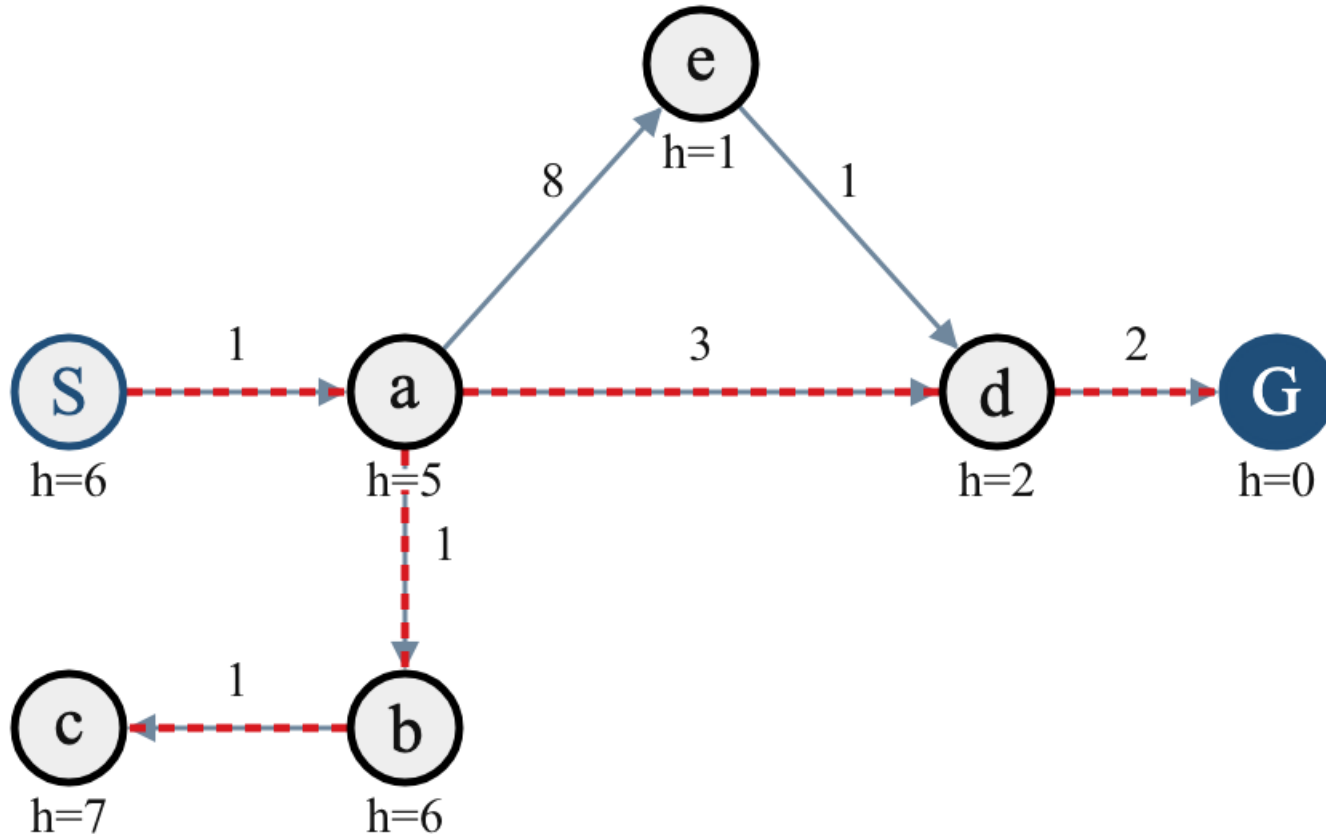
Follow the cheap branch

Sab $\rightarrow$ Sabc

c is a dead end

A* Motivation: UCS

Cheap so far can still lead away from the goal



After expanding a

Sab: $g=2$

Sad: $g=4$

Sae: $g=9$

Follow the cheap branch

Sab → Sabc

c is a dead end

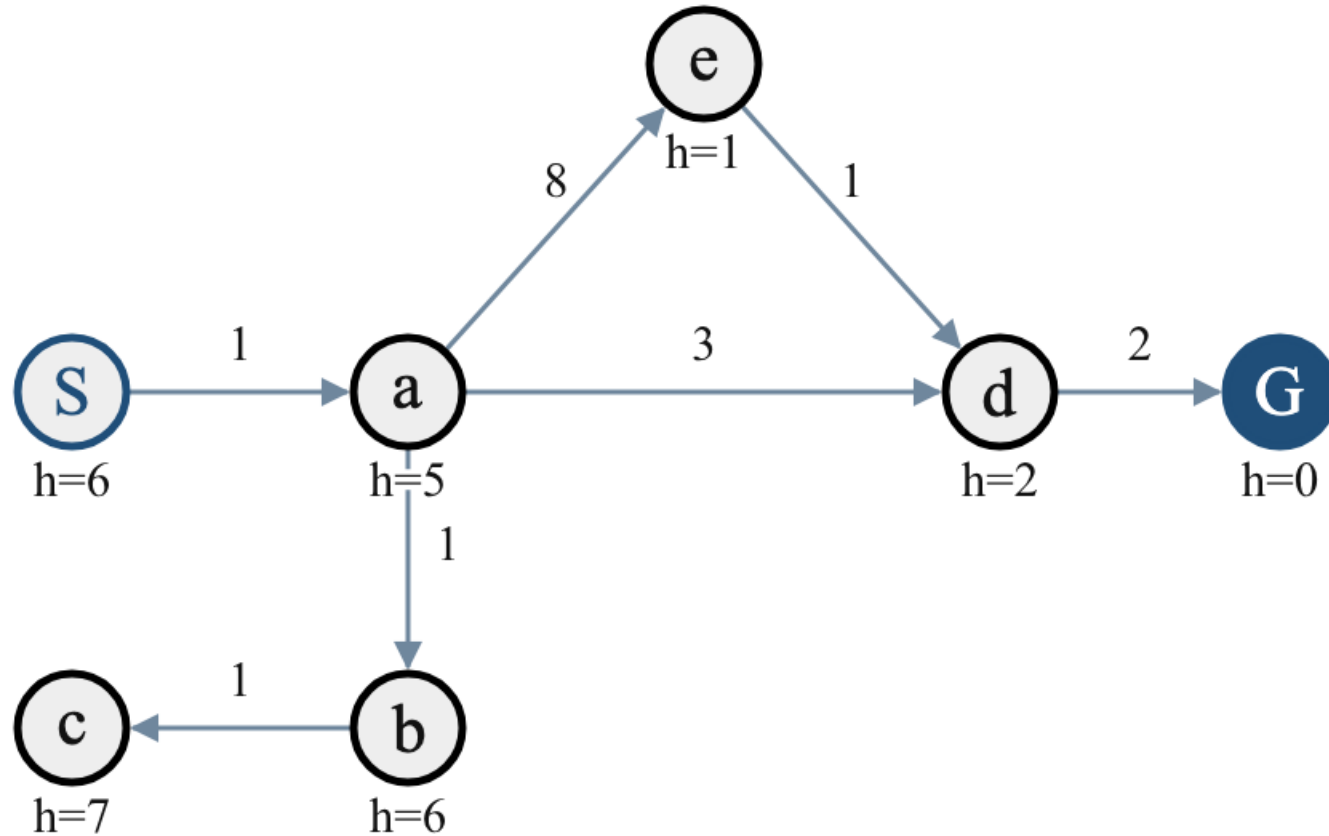
Return to the waiting path

Sad → SadG

Solution cost: 6

A* Motivation: Greedy Search

Greedy selects using $h(n)$ alone



Same frontier, new priority

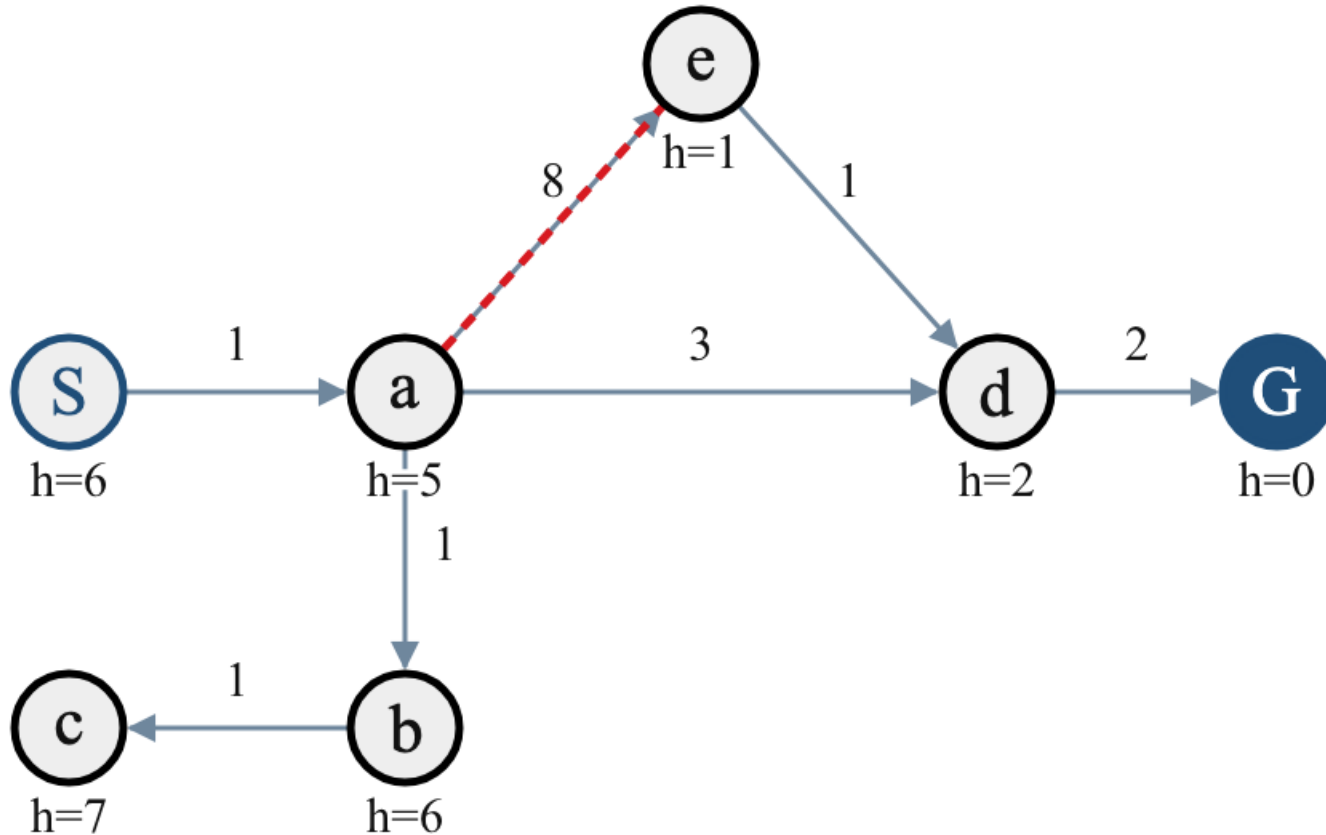
Sab: $h(b) = 6$

Sad: $h(d) = 2$

Sae: $h(e) = 1$

A* Motivation: Greedy Search

Greedy selects using $h(n)$ alone



Same frontier, new priority

Sab: $h(b)=6$

Sad: $h(d)=2$

Sae: $h(e)=1$

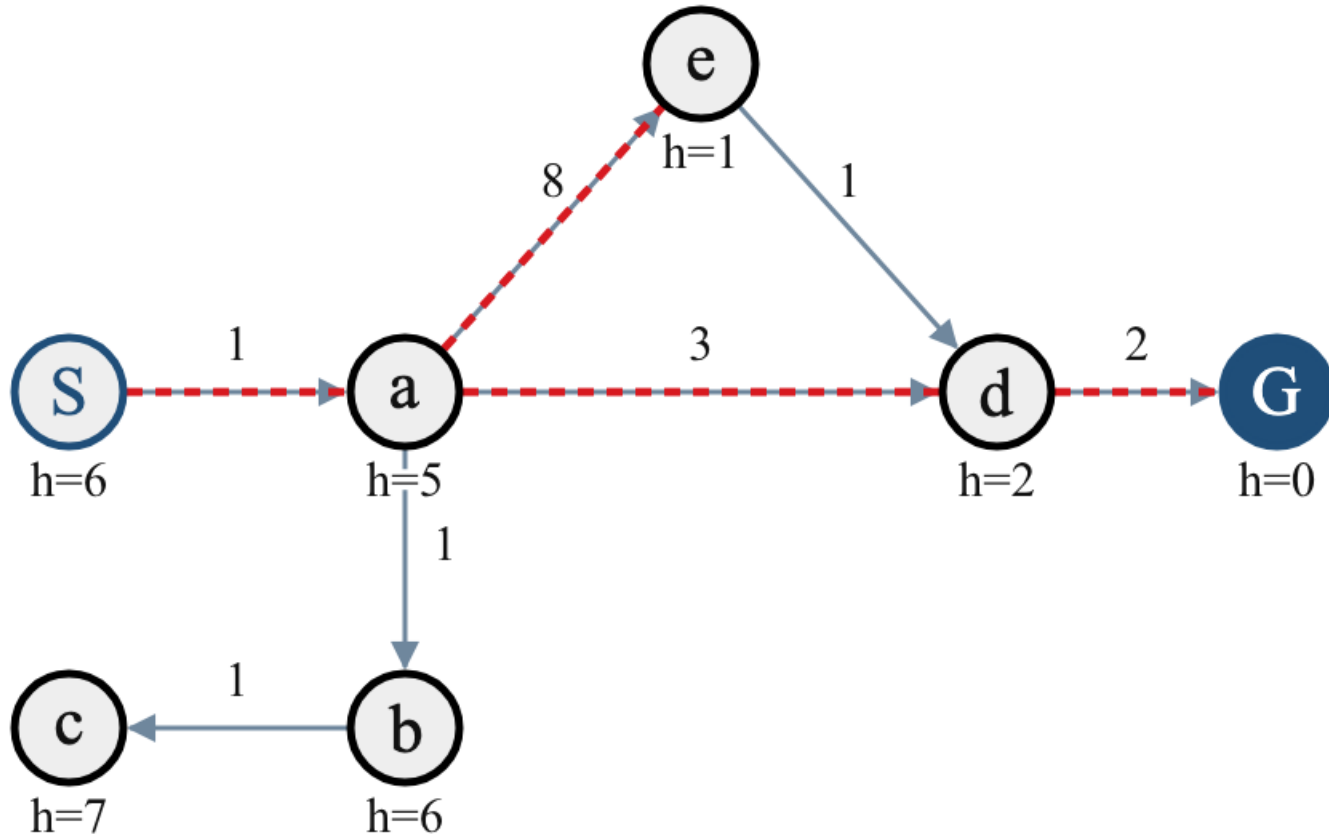
Chase the smallest estimate

Sae is selected first

$h(e)=1$ is the smallest estimate

A* Motivation: Greedy Search

Greedy selects using $h(n)$ alone



Same frontier, new priority

Sab: $h(b)=6$

Sad: $h(d)=2$

Sae: $h(e)=1$

Chase the smallest estimate

Sae is selected first

$h(e)=1$ is the smallest estimate

Then select **Sad**

Sad and **Saed** tie at $h=2$

Path-string order selects **Sad**

The Tortoise and the Hare



UCS

Cost so far

The Tortoise and the Hare



UCS

Cost so far



Greedy

Estimated cost remaining

The Tortoise and the Hare



UCS

Cost so far



Greedy

Estimated cost remaining

Can we combine their strengths?

The Tortoise and the Hare



UCS

Cost so far



Greedy

Estimated cost remaining



A*

Use both costs

A* Combines Cost So Far and Cost Remaining

$$g(n)$$

Cost so far

Actual path cost from A to S



Node n represents path AS

A* Combines Cost So Far and Cost Remaining

$$g(n) + h(n)$$

Cost so far

Estimated cost remaining

Actual path cost from A to S

Heuristic estimate from S to B



Node n represents path AS

A* Combines Cost So Far and Cost Remaining

$$g(n) + h(n) = f(n)$$

Cost so far

Estimated cost remaining

Actual path cost from A to S

Heuristic estimate from S to B

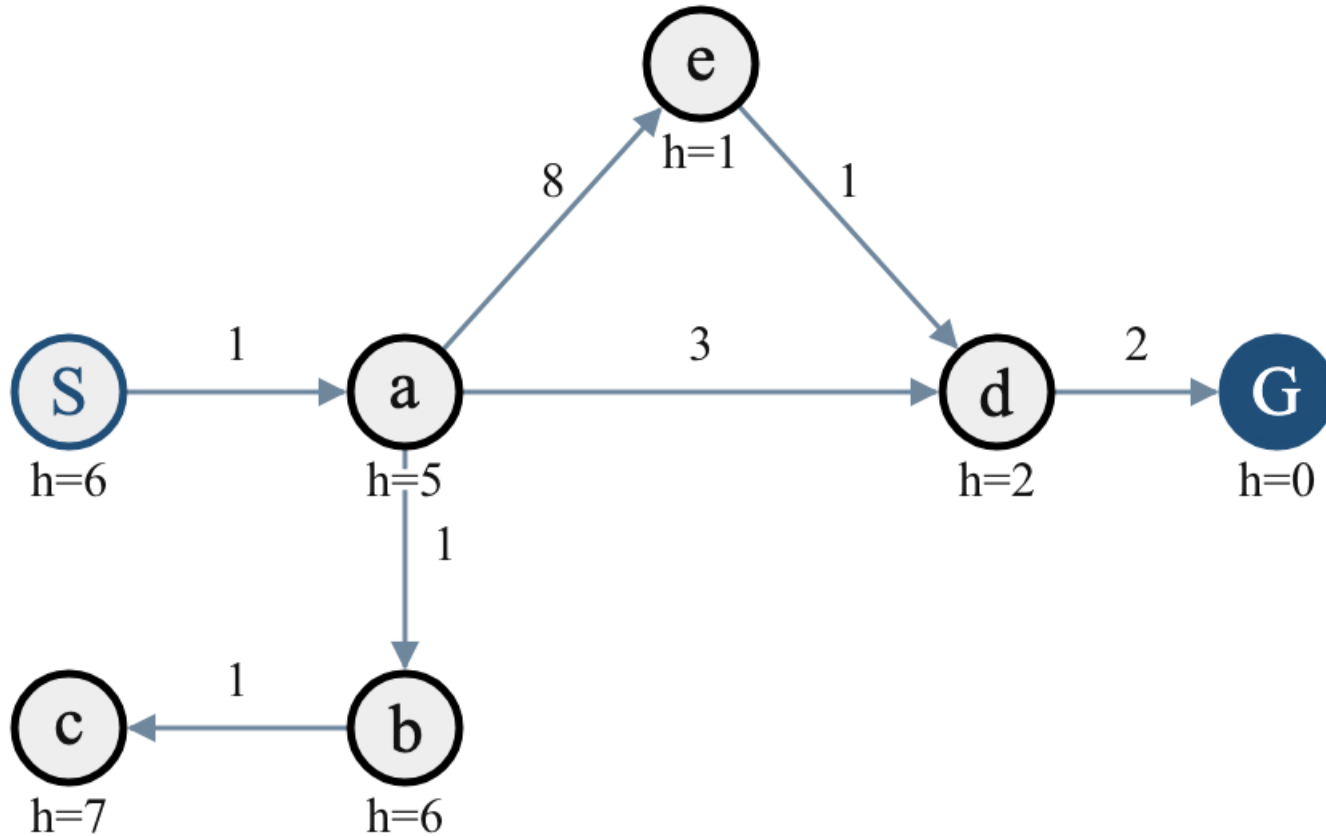


Node n represents path AS

Select the frontier node with the smallest $f(n)$

A* Motivation: Combine Both Costs

Now revisit the same choices using $g + h$



After expanding a

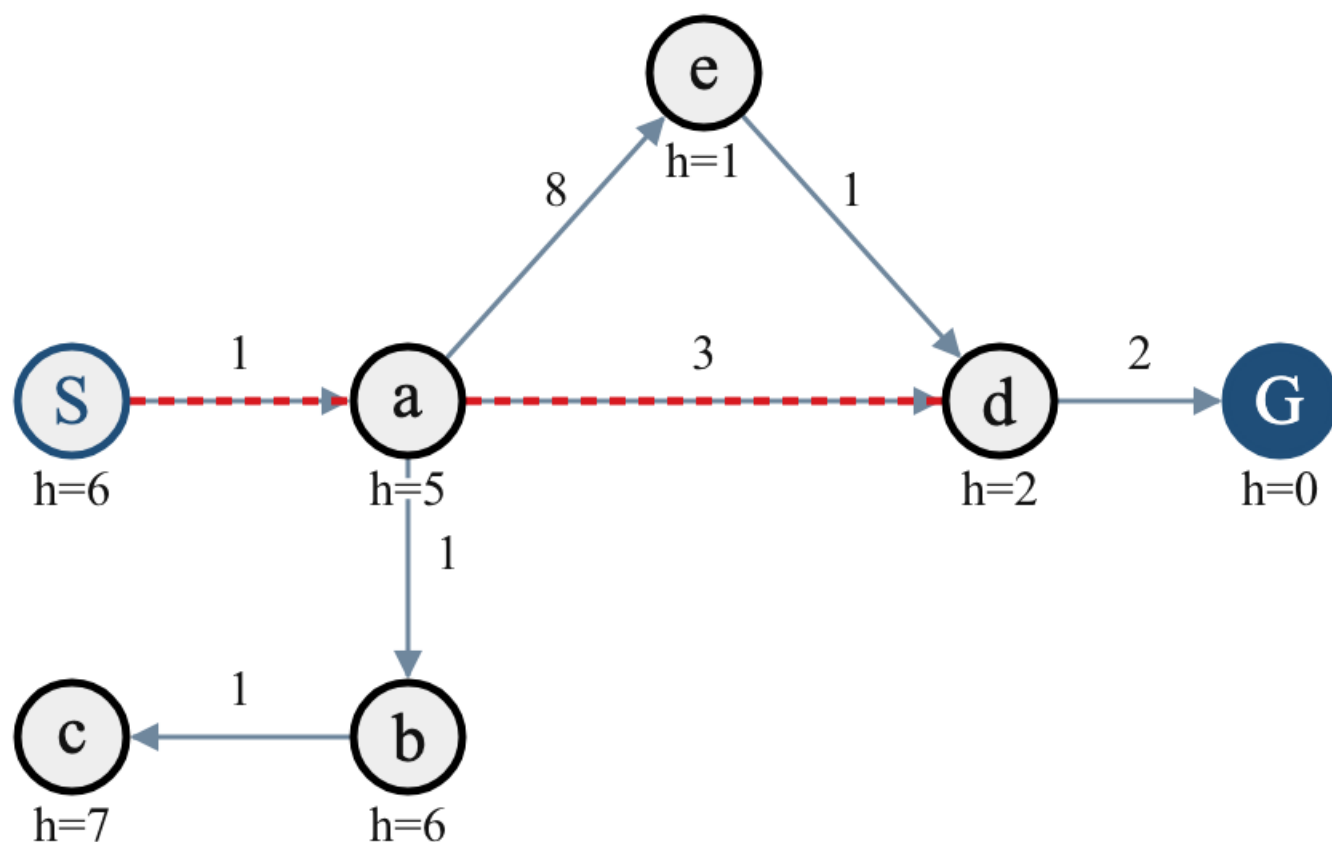
$$S_{ab}: 2 + 6 = 8$$

$$S_{ad}: 4 + 2 = 6$$

$$S_{ae}: 9 + 1 = 10$$

A* Motivation: Combine Both Costs

Now revisit the same choices using $g + h$



After expanding a

$$\text{Sab: } 2 + 6 = 8$$

$$\text{Sad: } 4 + 2 = 6$$

$$\text{Sae: } 9 + 1 = 10$$

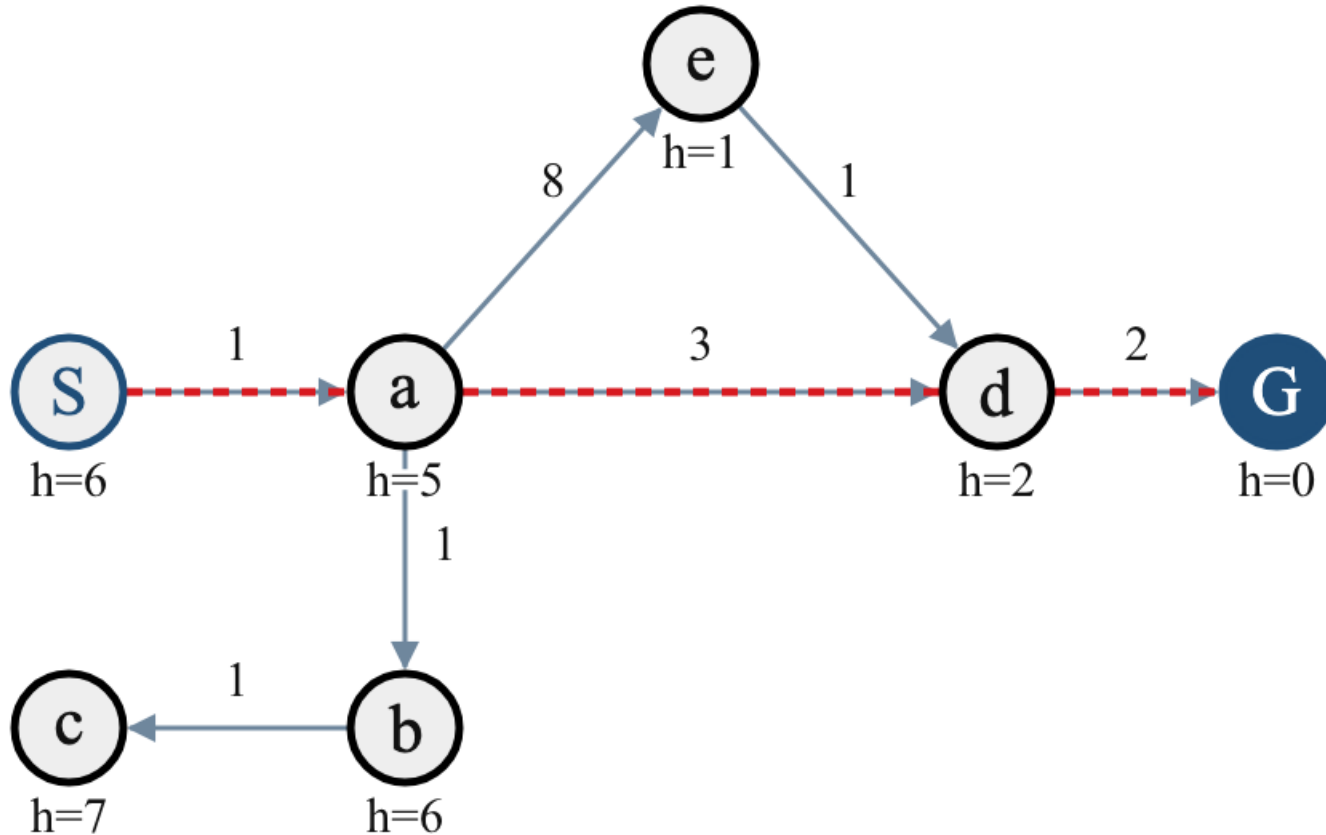
Select Sad

Its estimated total cost is smallest

$$\text{SadG: } 6 + 0 = 6$$

A* Motivation: Combine Both Costs

Now revisit the same choices using $g + h$



After expanding a

$$S_{ab}: 2 + 6 = 8$$

$$S_{ad}: 4 + 2 = 6$$

$$S_{ae}: 9 + 1 = 10$$

Select S_{ad}

Its estimated total cost is smallest

$$S_{adG}: 6 + 0 = 6$$

Reach G without either detour

Expanded: $S \rightarrow a \rightarrow d$

Returned path: S_{adG} , cost 6

A* Uses Both Costs

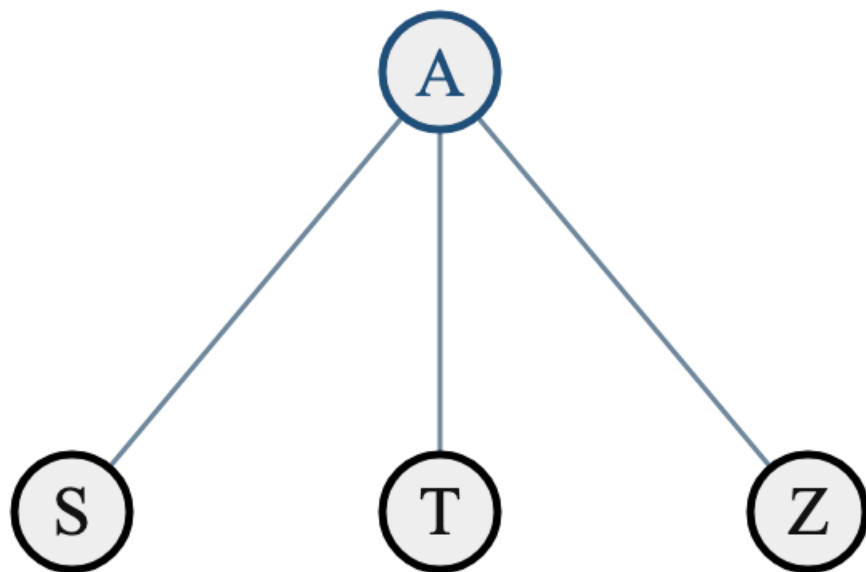
Select the frontier node with the smallest $g(n) + h(n)$



```
1 from heapq import heappush, heappop
2 def aStarTsa(graph, h, start, goal):
3     frontier = [(h[start], start)]
4     print("Initial frontier:", frontier)
5     while frontier:
6         f, path = heappop(frontier)
7         state = path[-1]
8         if state == goal:
9             return f, path
10        print("Exploring:", state, "at priority", f)
11        for cost, child in graph[state]:
12            new_f = f + cost - h[state] + h[child]
13            heappush(frontier, (new_f, path + child))
14        print(frontier)
15 graph = {'S': [(1, 'a')], 'a': [(1, 'b'), (3, 'd'), (8, 'e')],
16         'b': [(1, 'c')], 'c': [], 'd': [(2, 'G')],
17         'e': [(1, 'd')], 'G': []}
18 h = {'S': 6, 'a': 5, 'b': 6, 'c': 7, 'd': 2, 'e': 1, 'G': 0}
19 print("Solution path:", aStarTsa(graph, h, 'S', 'G'))
```

A* Selects the Smallest $g + h$

Same frontier paths · a new priority



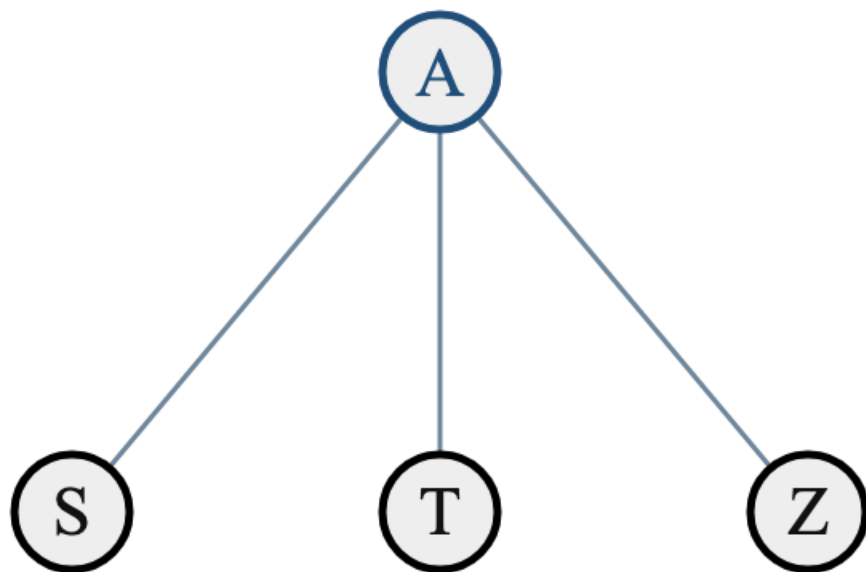
Frontier after expanding A

g : path cost · h : estimate at final state

Path	g	h	$f = g + h$
AS	140	253	
AT	118	329	
AZ	75	374	

A* Selects the Smallest $g + h$

Same frontier paths · a new priority



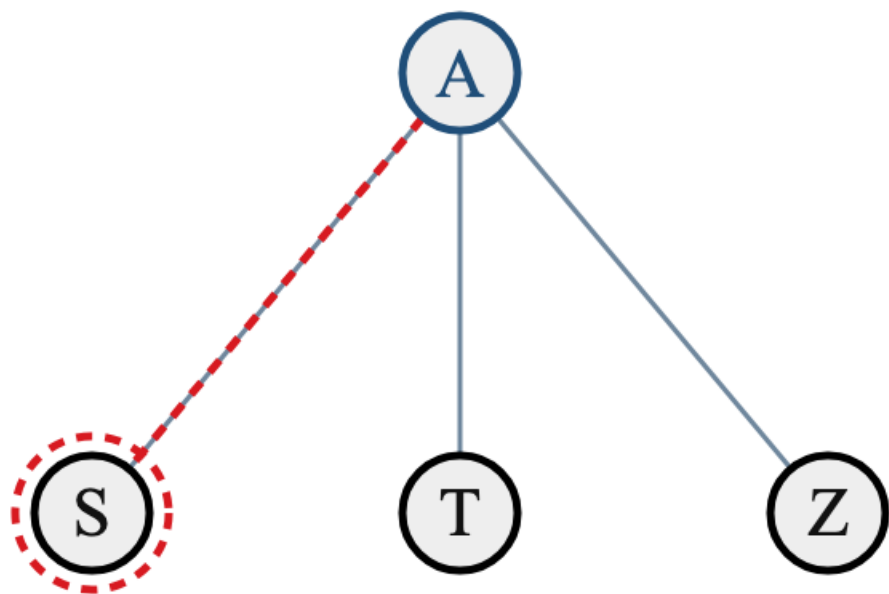
Frontier after expanding A

g : path cost · h : estimate at final state

Path	g	h	$f = g + h$
AS	140	253	393
AT	118	329	447
AZ	75	374	449

A* Selects the Smallest $g + h$

Same frontier paths · a new priority



Frontier after expanding A

g : path cost · h : estimate at final state

Path	g	h	$f = g + h$
AS	140	253	393
AT	118	329	447
AZ	75	374	449

Remove AS: 393 is smallest

A* Keeps Comparing Frontier Paths

Which path leaves the frontier next?

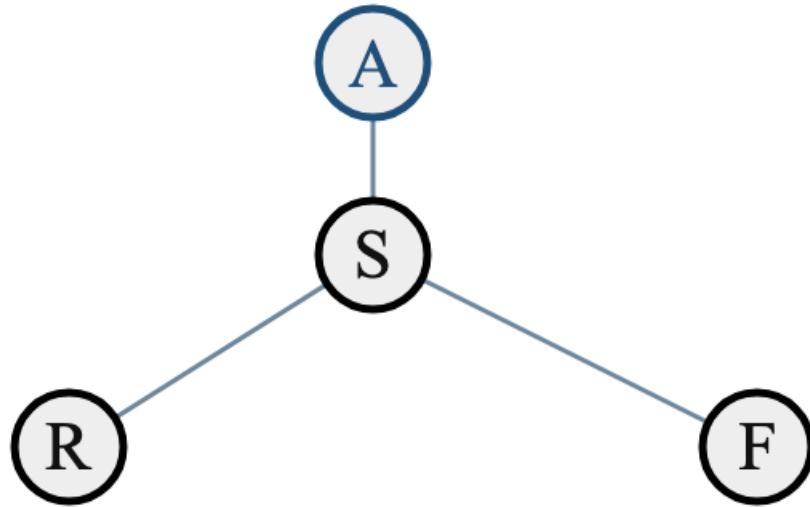


Expanded path	f	Two lowest frontier priorities
A	366	AS: 393 · AT: 447

Tree shows the competing routes

A* Keeps Comparing Frontier Paths

Which path leaves the frontier next?

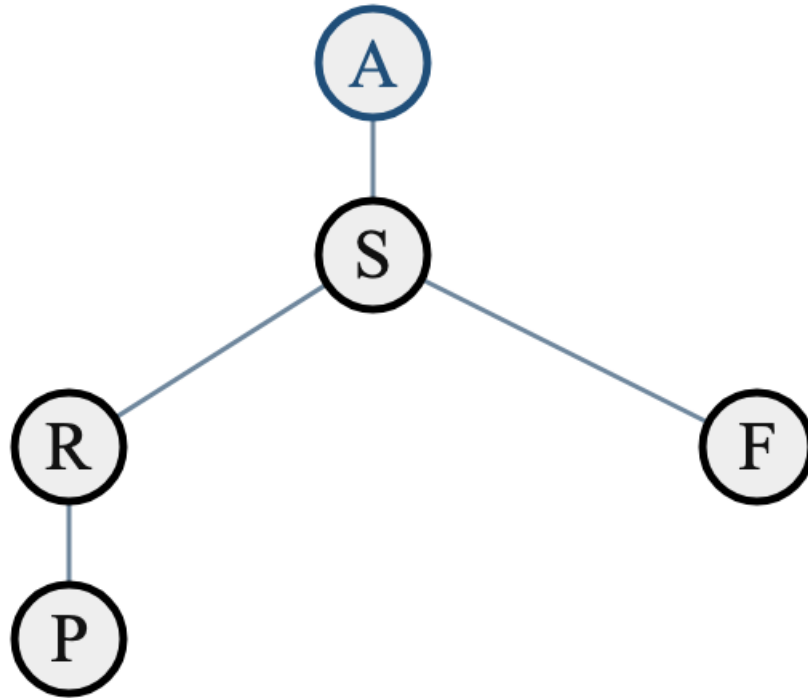


Expanded path	f	Two lowest frontier priorities
A	366	AS: 393 · AT: 447
AS	393	ASR: 413 · ASF: 415

Tree shows the competing routes

A* Keeps Comparing Frontier Paths

Which path leaves the frontier next?

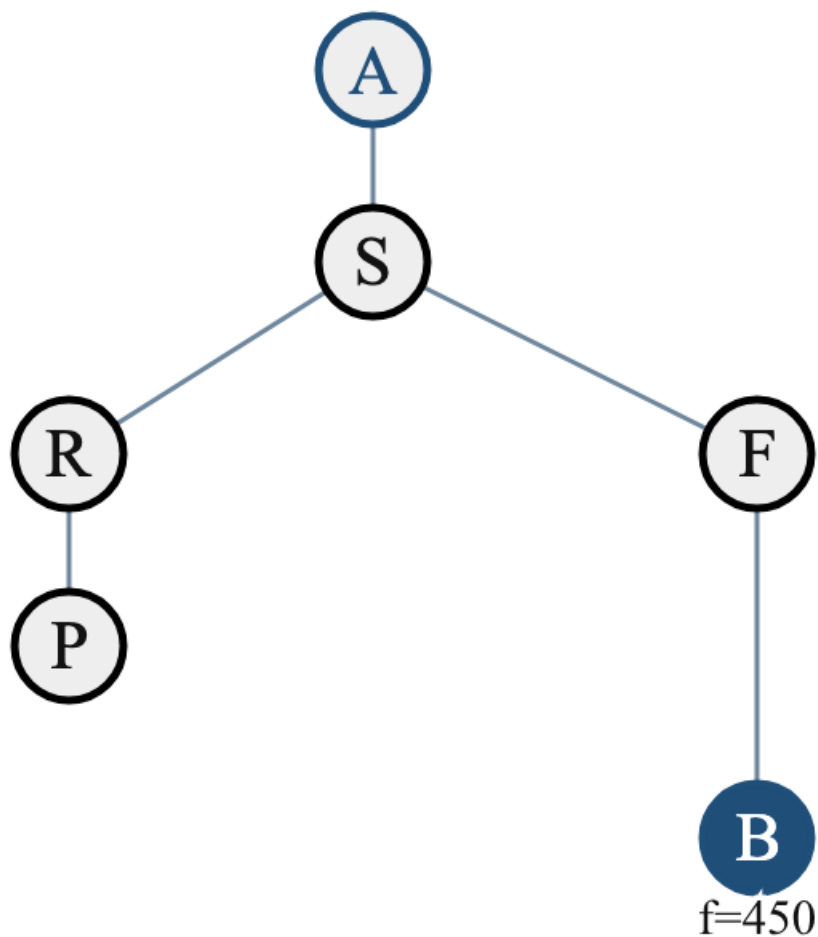


Expanded path	f	Two lowest frontier priorities
A	366	AS: 393 · AT: 447
AS	393	ASR: 413 · ASF: 415
ASR	413	ASF: 415 · ASRP: 417

Tree shows the competing routes

A* Keeps Comparing Frontier Paths

Which path leaves the frontier next?

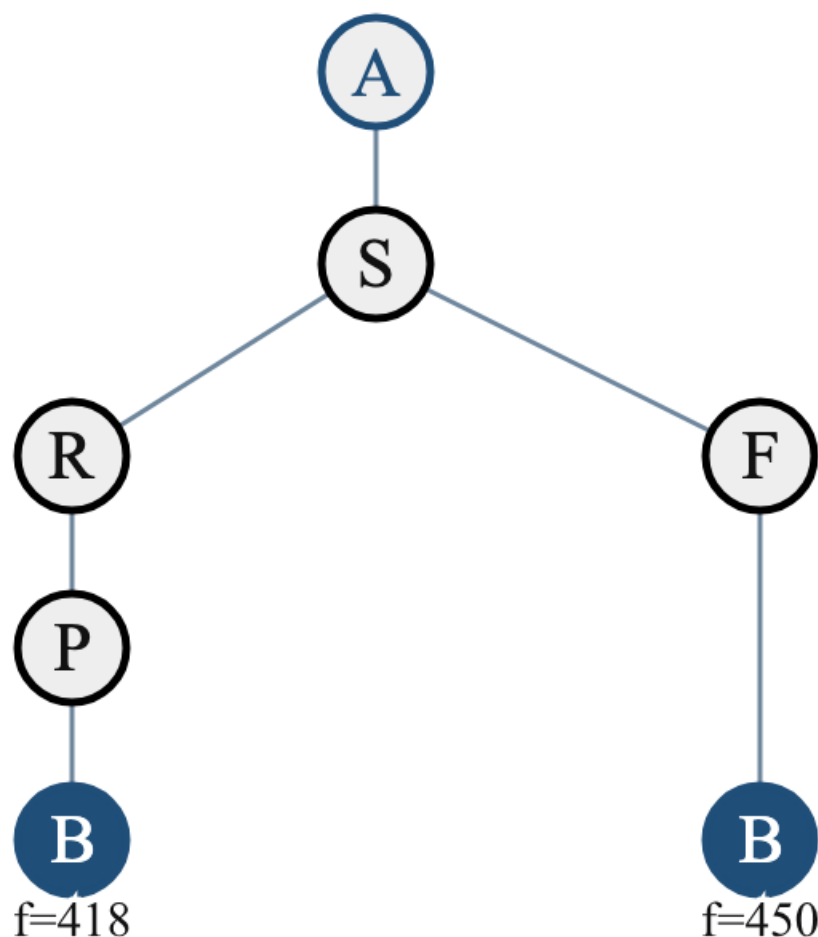


Expanded path	f	Two lowest frontier priorities
A	366	AS: 393 · AT: 447
AS	393	ASR: 413 · ASF: 415
ASR	413	ASF: 415 · ASRP: 417
ASF	415	ASRP: 417 · AT: 447

Tree shows the competing routes

A* Keeps Comparing Frontier Paths

Which path leaves the frontier next?



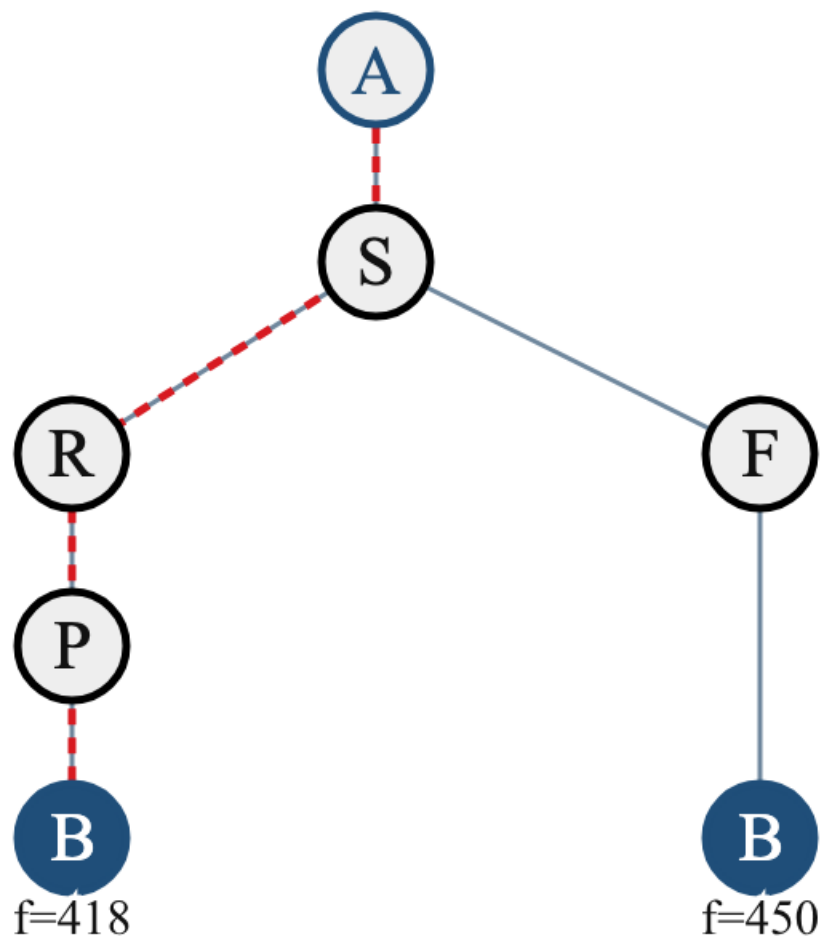
Tree shows the competing routes

other generated branches are omitted

Expanded path	f	Two lowest frontier priorities
A	366	AS: 393 · AT: 447
AS	393	ASR: 413 · ASF: 415
ASR	413	ASF: 415 · ASRP: 417
ASF	415	ASRP: 417 · AT: 447
ASRP	417	ASRPB: 418 · AT: 447
ASRPB: g=418, h(B)=0, f=418		

A* Keeps Comparing Frontier Paths

Which path leaves the frontier next?



Tree shows the competing routes

other generated branches are omitted

Expanded path	f	Two lowest frontier priorities
A	366	AS: 393 · AT: 447
AS	393	ASR: 413 · ASF: 415
ASR	413	ASF: 415 · ASRP: 417
ASF	415	ASRP: 417 · AT: 447
ASRP	417	ASRPB: 418 · AT: 447

ASRPB: $g=418$, $h(B)=0$, $f=418$

Remove ASRPB and stop

Solution cost: 418

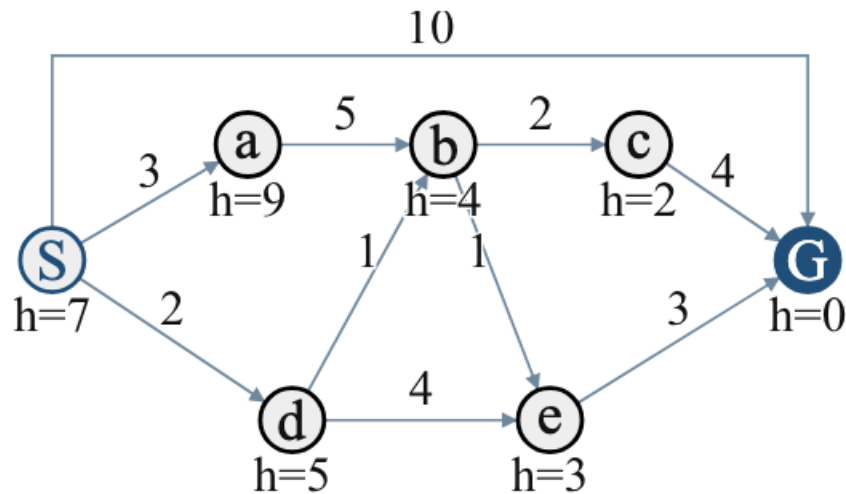
A*-TSA Practice

Start: S · Goal: G

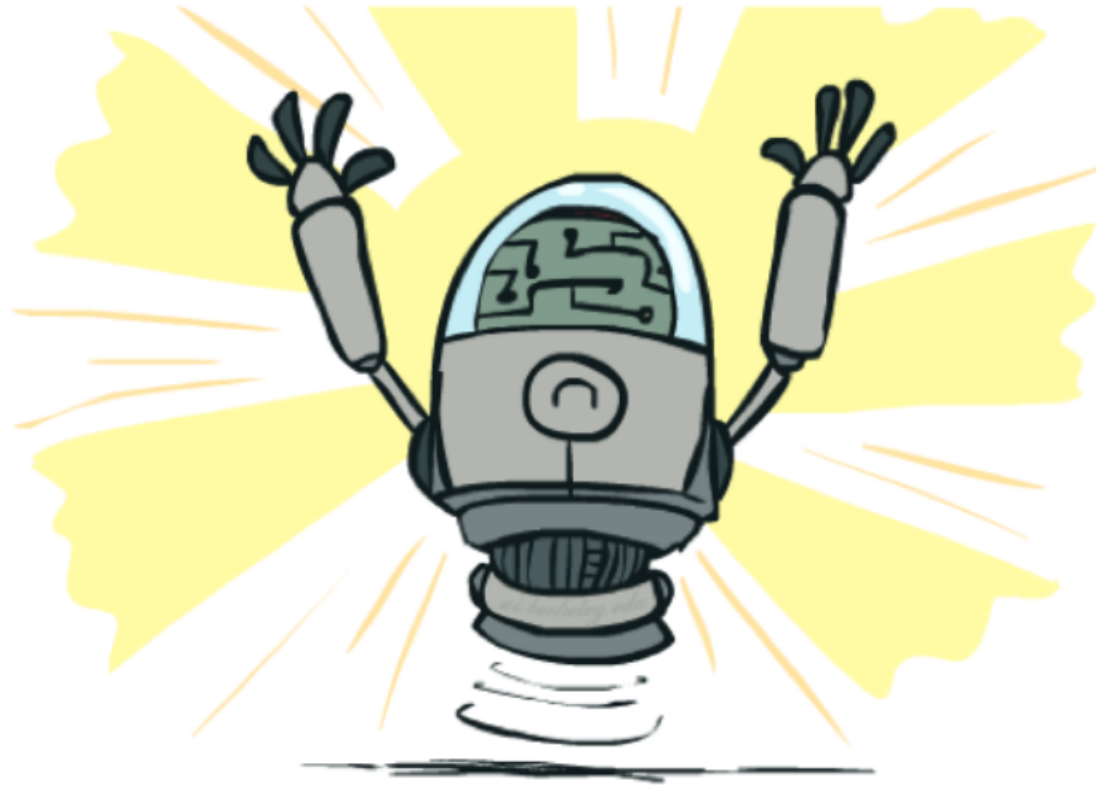
Use $f(n)=g(n)+h(n)$

Predict the expansion order
and returned path

Break ties by path-string order



Is A^* Always Optimal?

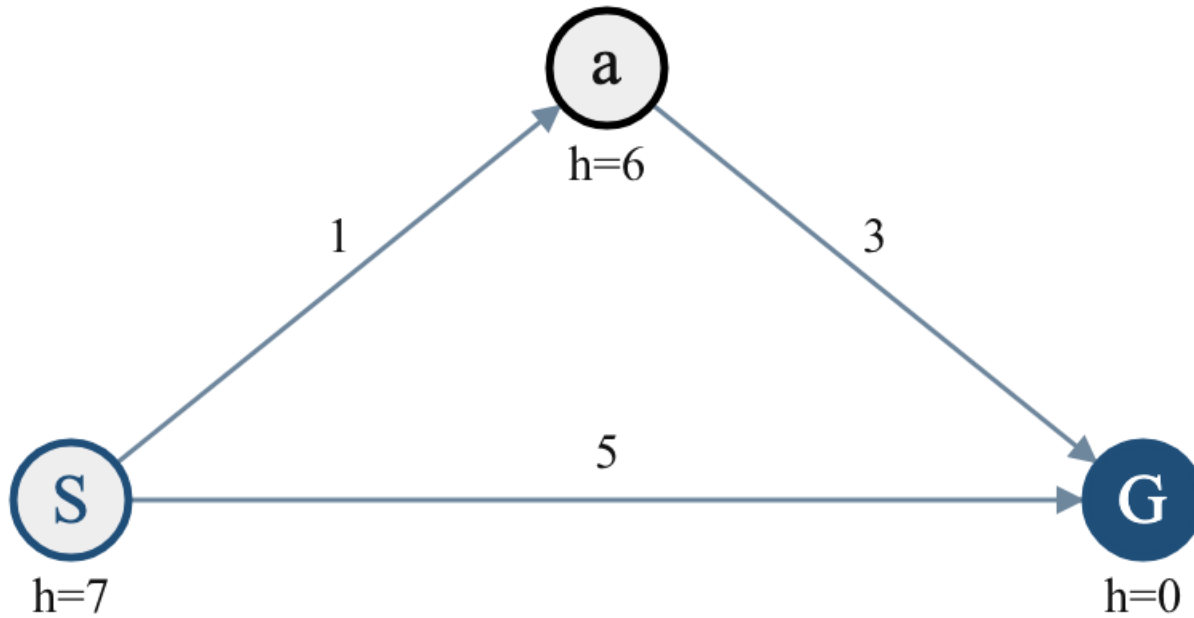


It worked on Romania. Is that a guarantee?

Is A*-TSA Optimal?

Start: S · Goal: G

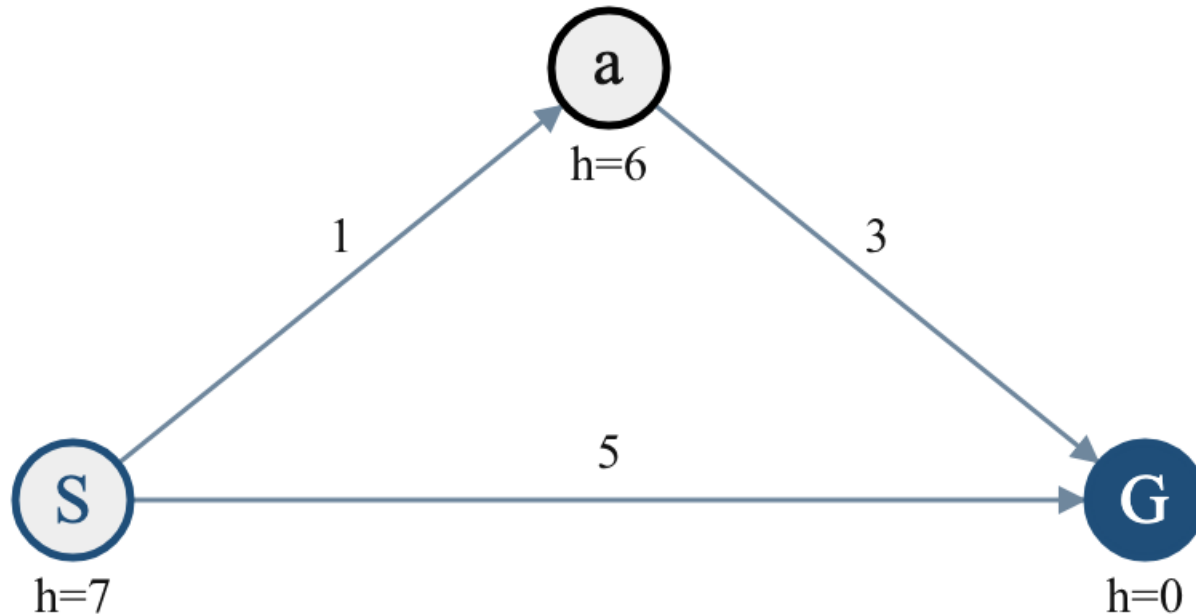
Which path will A*-TSA return?



Is A*-TSA Optimal?

Start: S · Goal: G

Which path will A*-TSA return?

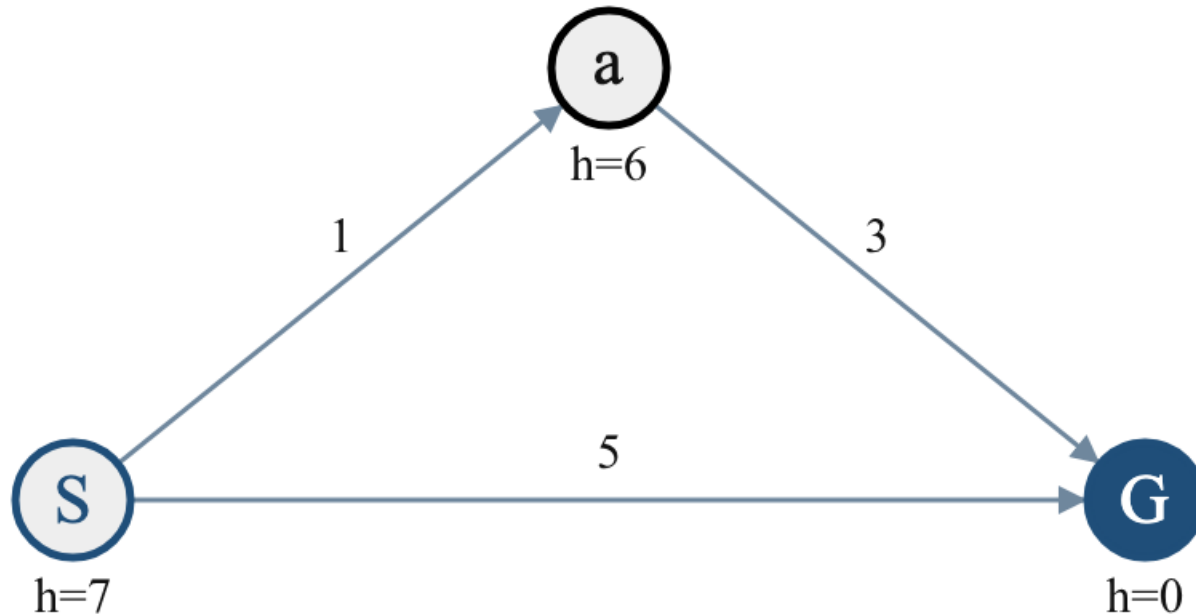


```
Initial frontier: [(7, 'S')]
S @ 7 -> [(5, 'SG'), (7, 'Sa')]
```

Is A*-TSA Optimal?

Start: S · Goal: G

Which path will A*-TSA return?



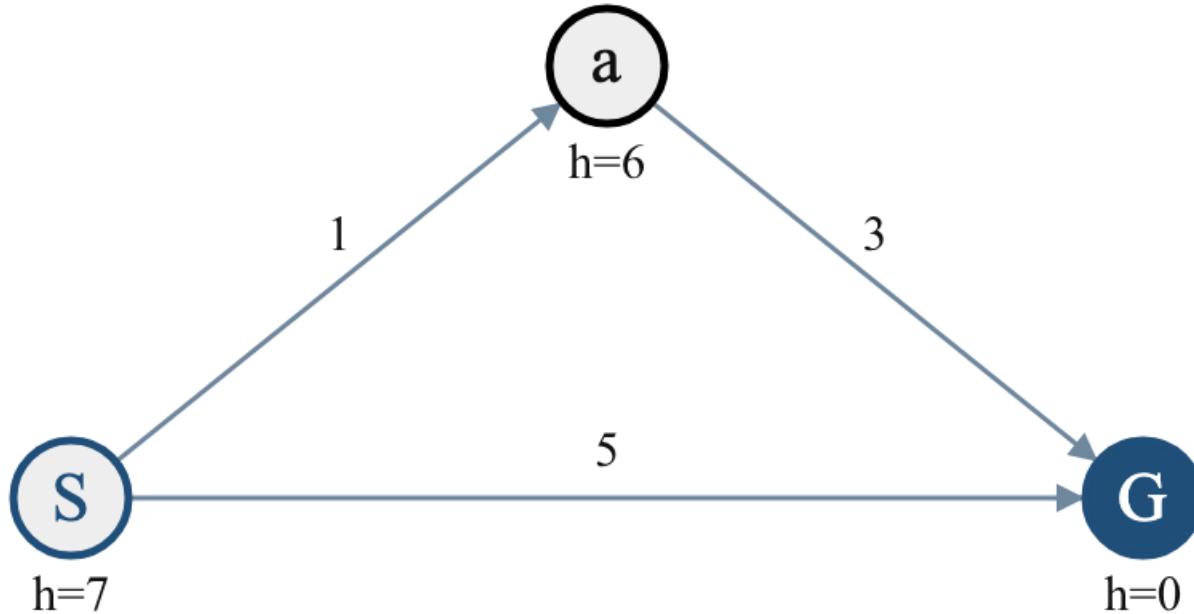
Initial frontier: [(7, 'S')]
S @ 7 → [(5, 'SG'), (7, 'Sa')]

Solution path: (5, 'SG')

Is A*-TSA Optimal?

Start: S · Goal: G

Which path will A*-TSA return?



```
Initial frontier: [(7, 'S')]
S @ 7 -> [(5, 'SG'), (7, 'Sa')]
```

```
Solution path: (5, 'SG')
```

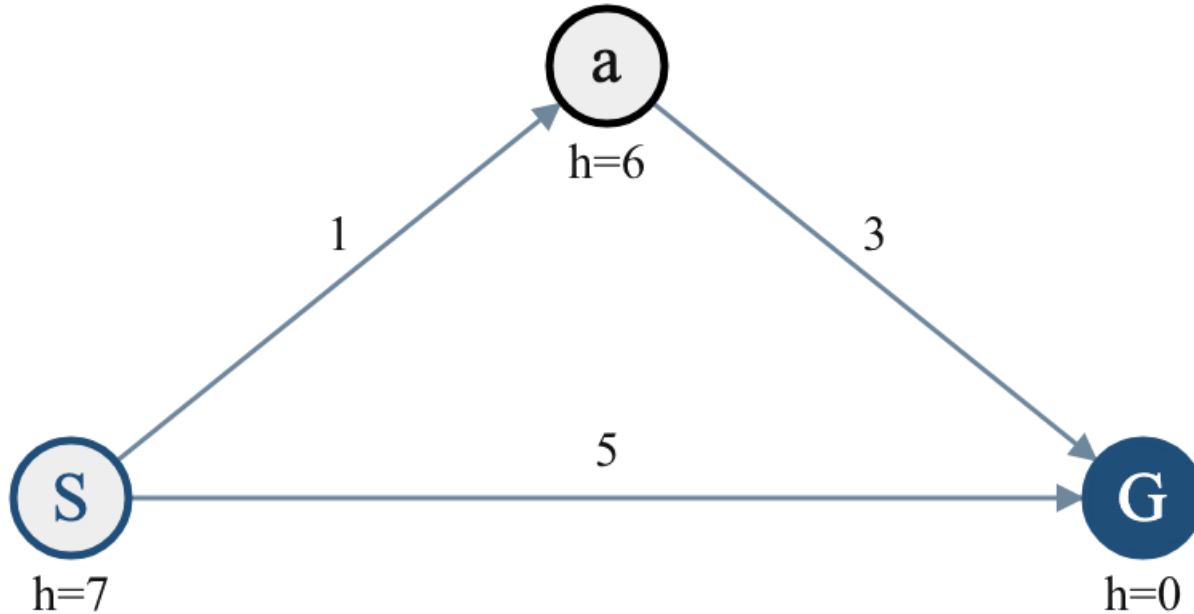
Returned: SG, cost 5

Cheaper path: SaG, cost $1 + 3 = 4$

Is A*-TSA Optimal?

Start: S · Goal: G

Which path will A*-TSA return?



```
Initial frontier: [(7, 'S')]
S @ 7 -> [(5, 'SG'), (7, 'Sa')]
```

```
Solution path: (5, 'SG')
```

Returned: SG, cost 5

Cheaper path: SaG, cost $1 + 3 = 4$

$h(a) = 6$ overestimates the
true remaining cost of 3

Admissibility of Heuristic

An optimistic estimate

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

Admissibility of Heuristic

An optimistic estimate

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

$h^*(n)$ is the true cheapest remaining cost

$h(n)$ may underestimate, but must not overestimate

Admissibility of Heuristic

An optimistic estimate

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

$h^*(n)$ is the true cheapest remaining cost

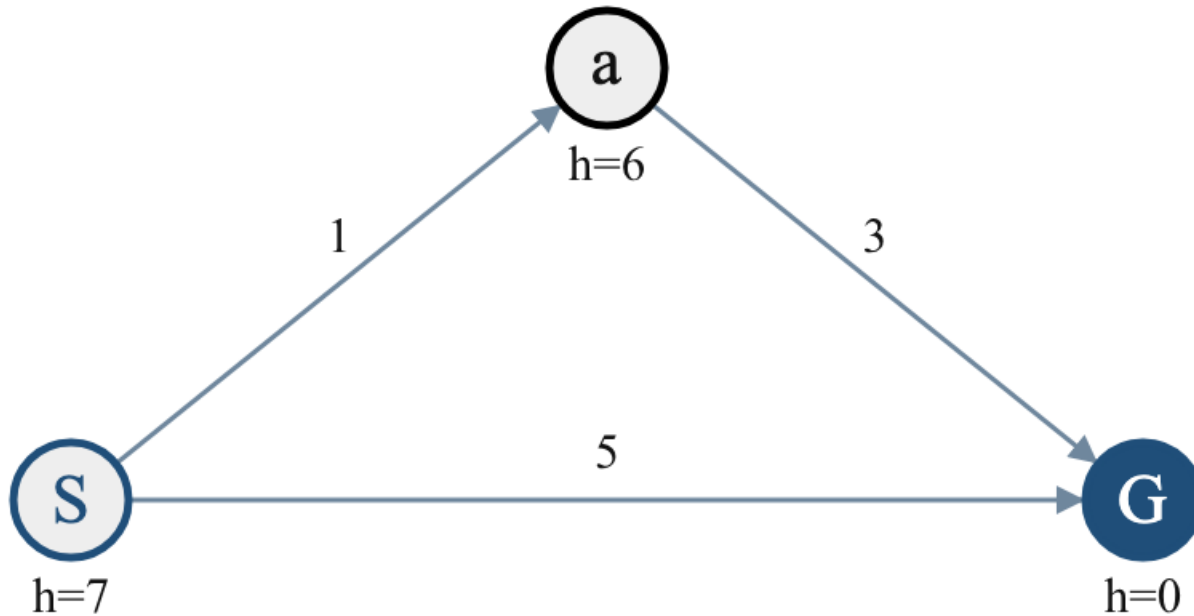
$h(n)$ may underestimate, but must not overestimate

At a goal

$$h(\text{goal})=0$$

Admissibility of Heuristic

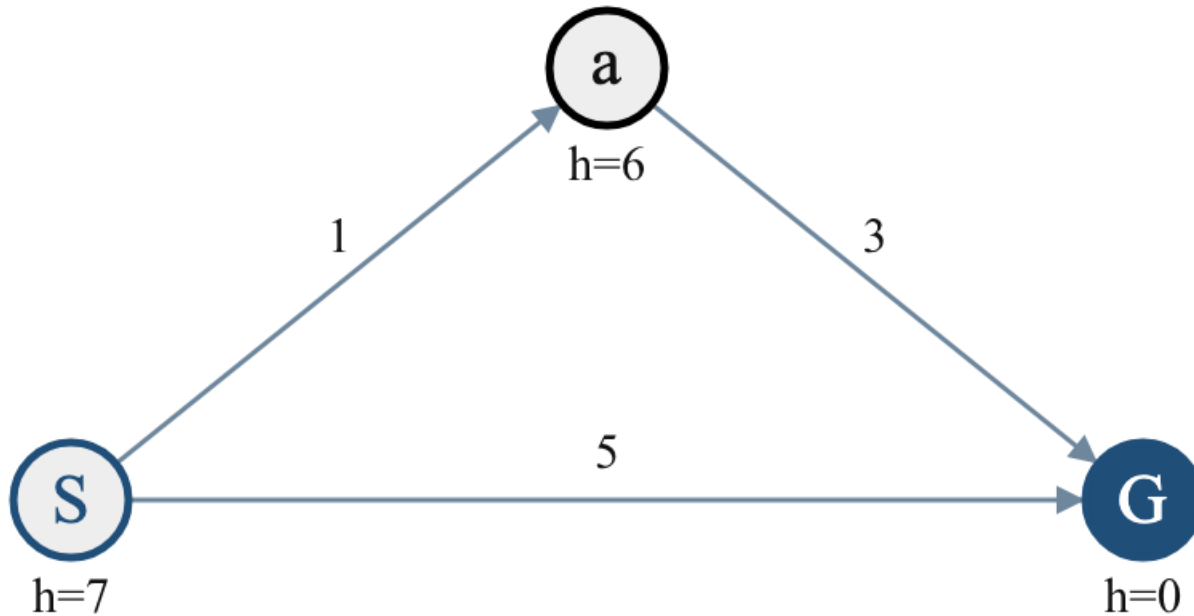
Is h admissible for this graph?



Admissibility of Heuristic

Is h admissible for this graph?

$$h(a) = 6 > h^*(a) = 3$$

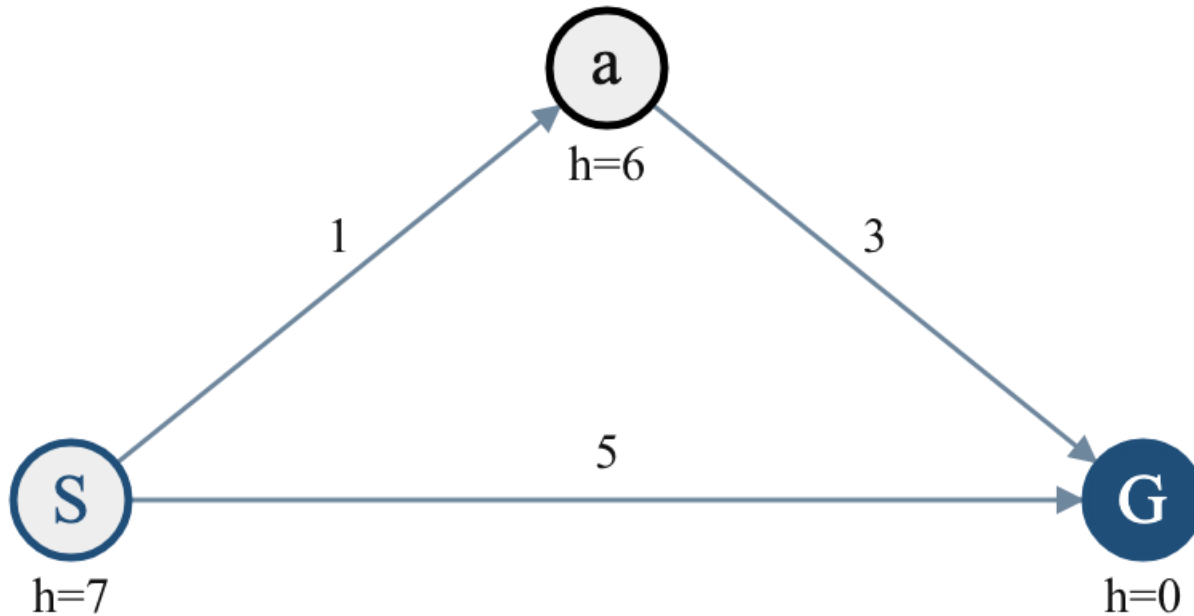


Admissibility of Heuristic

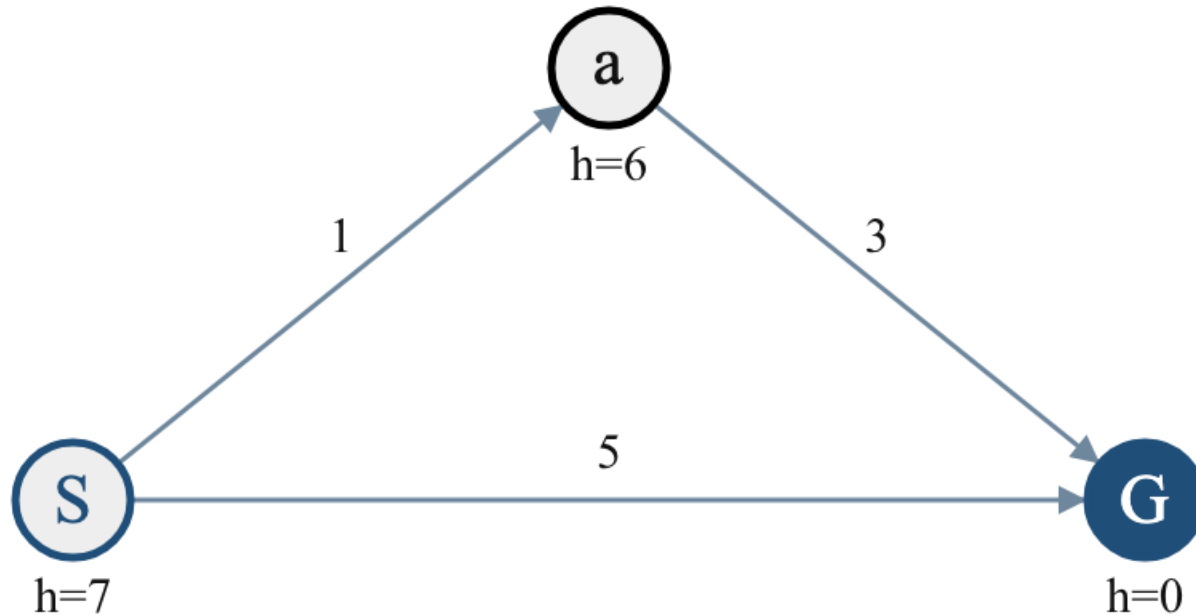
Is h admissible for this graph?

$$h(a) = 6 > h^*(a) = 3$$

$$h(S) = 7 > h^*(S) = 4$$



Admissibility of Heuristic



Is h admissible for this graph?

$$h(a) = 6 > h^*(a) = 3$$

$$h(S) = 7 > h^*(S) = 4$$

Keep the same edge costs

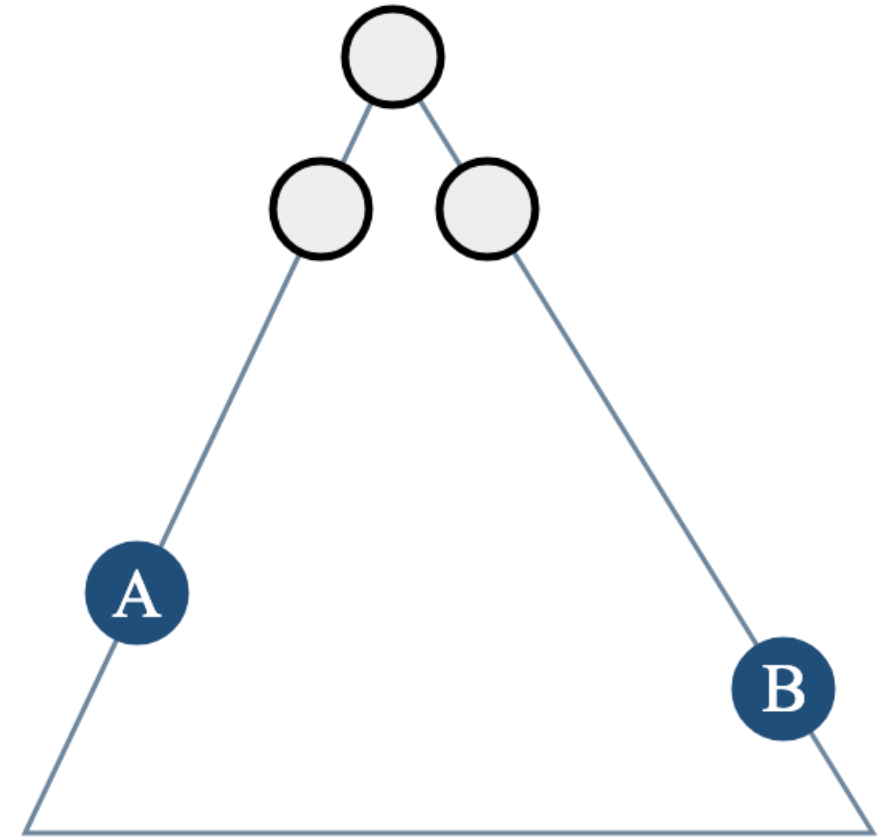
What values are allowed for $h(a)$?

What values are allowed for $h(S)$?

Optimality of A*

Let

- A be an optimal goal node
- B be a suboptimal goal node
- h be admissible



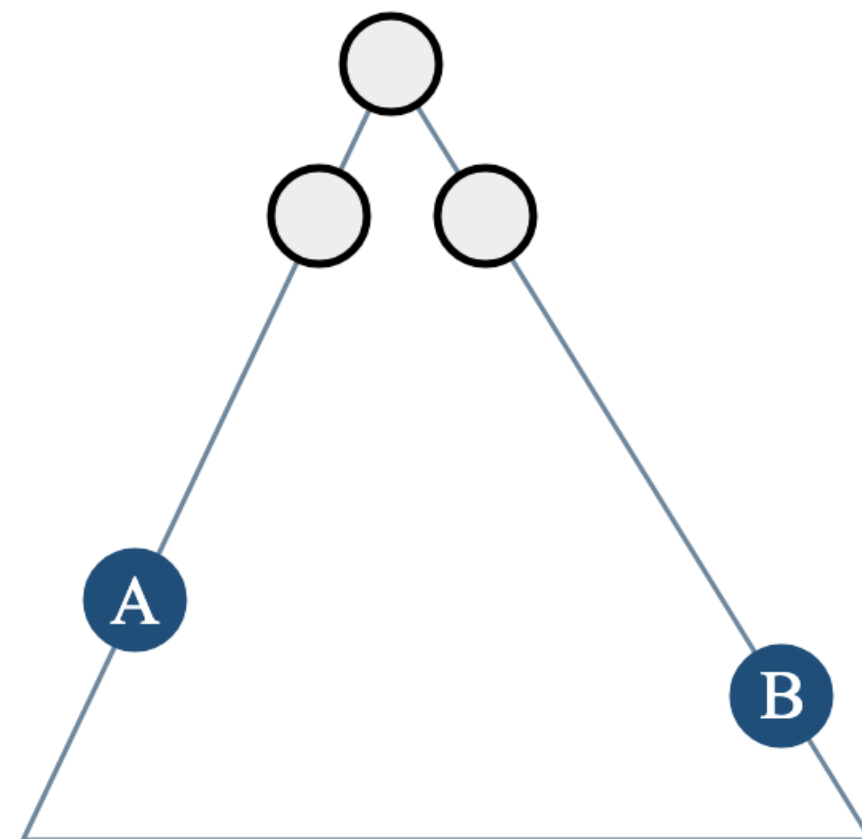
Optimality of A*

Let

- A be an optimal goal node
- B be a suboptimal goal node
- h be admissible

Claim

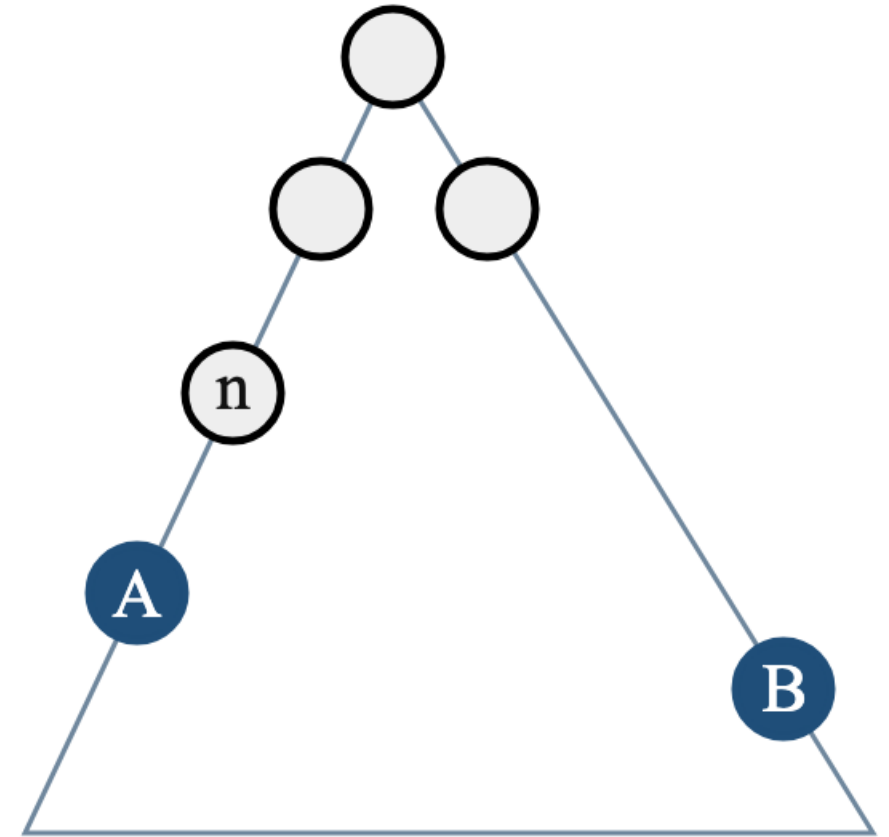
- A will exit the frontier before B



Optimality of A^*

Let n be an ancestor of A

Let B and n both be on the frontier

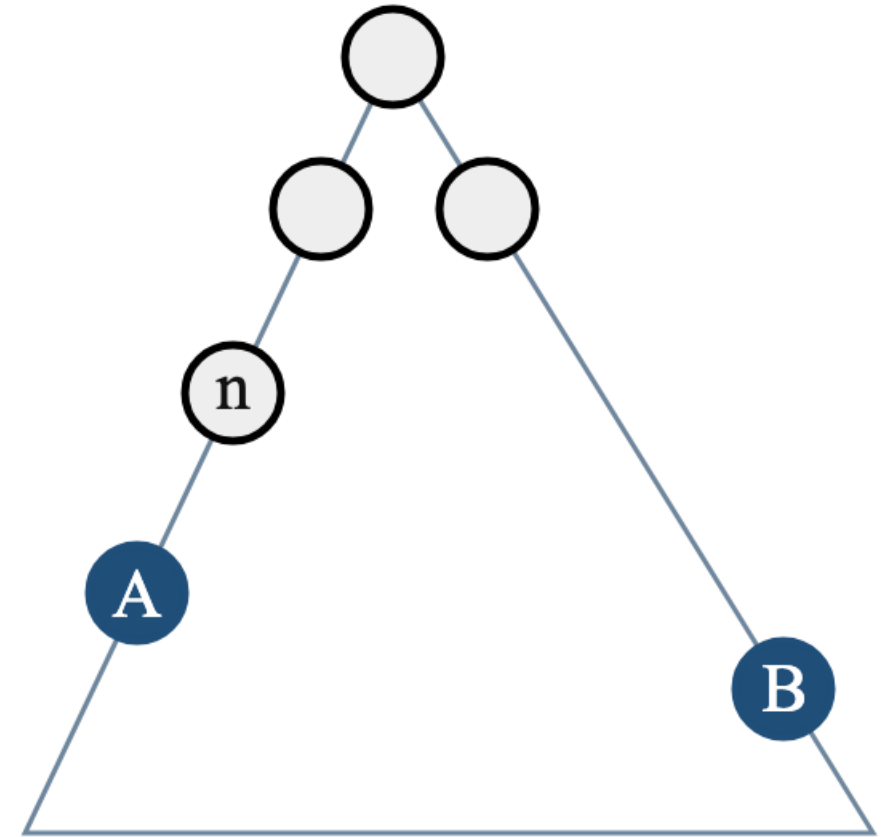


Optimality of A^*

Let n be an ancestor of A

Let B and n both be on the frontier

Claim: n exits the frontier before B



Optimality of A^*

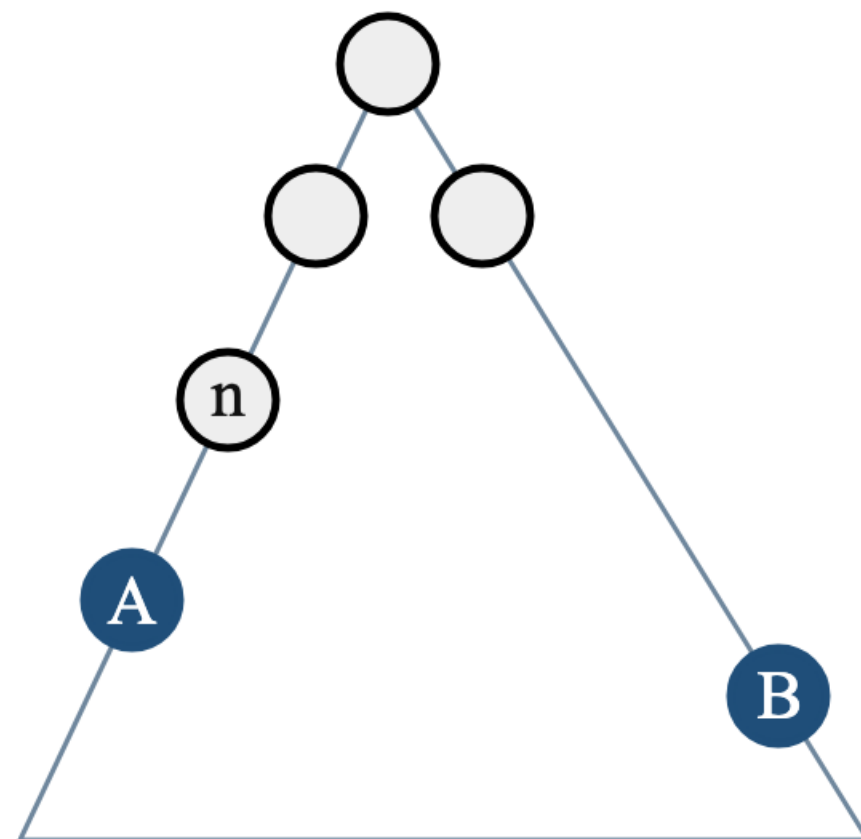
Let n be an ancestor of A

Let B and n both be on the frontier

Claim: n exits the frontier before B

$$f(n) \leq f(A)$$

Admissibility, with n on the optimal route to A



Optimality of A*

Let n be an ancestor of A

Let B and n both be on the frontier

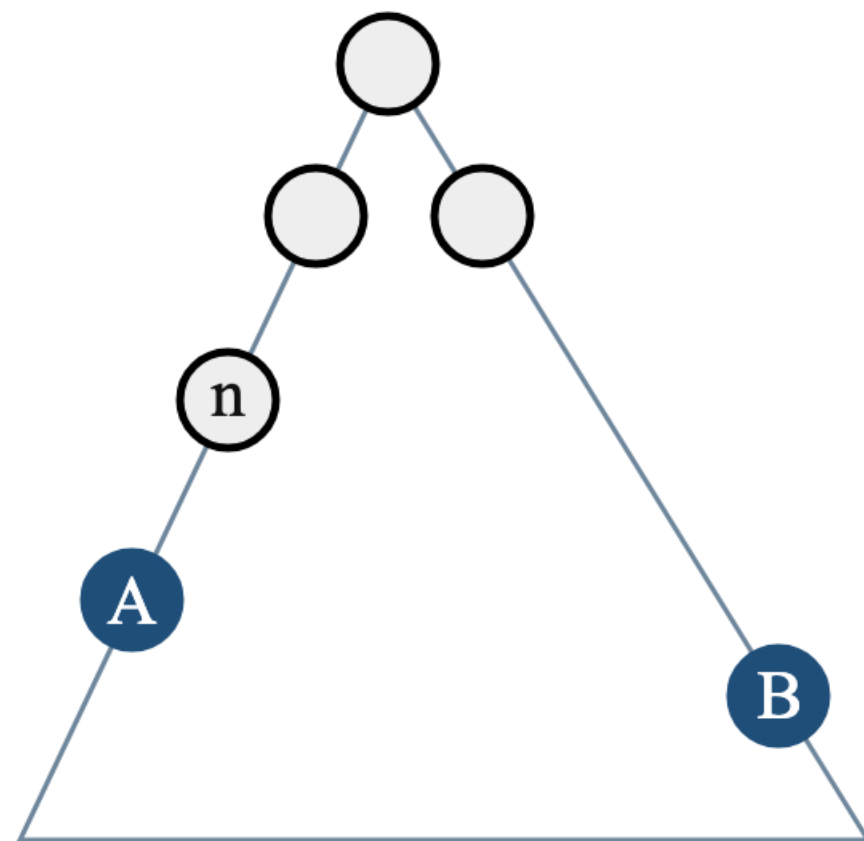
Claim: n exits the frontier before B

$$f(n) \leq f(A)$$

Admissibility, with n on the optimal route to A

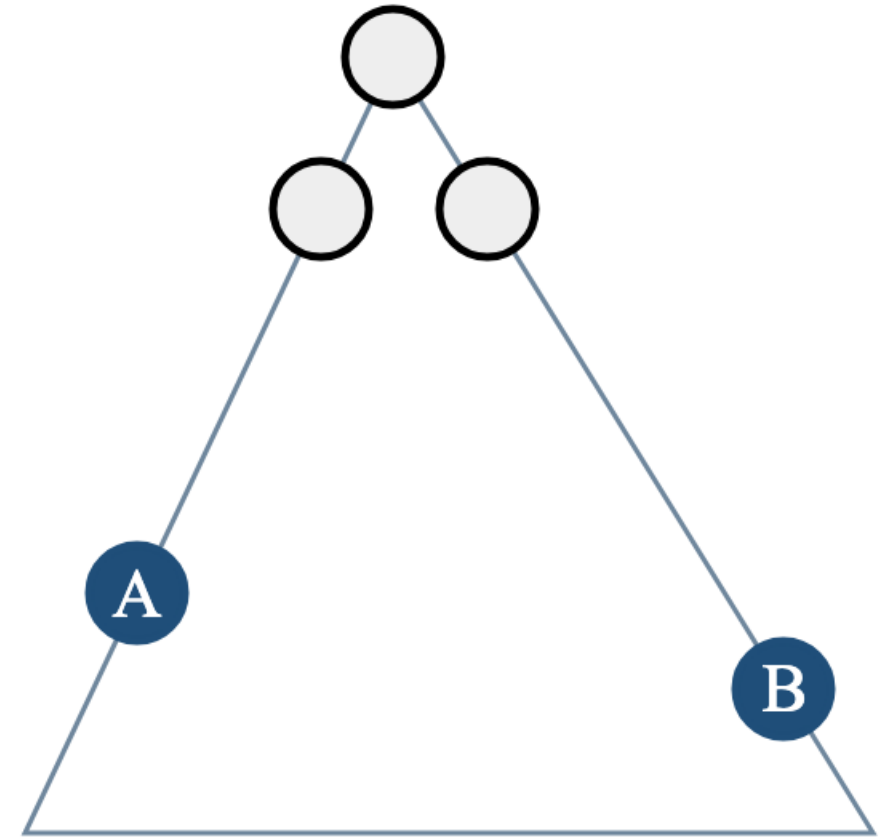
$$f(A) < f(B)$$

B is suboptimal and both goal heuristics are zero



Optimality of A^*

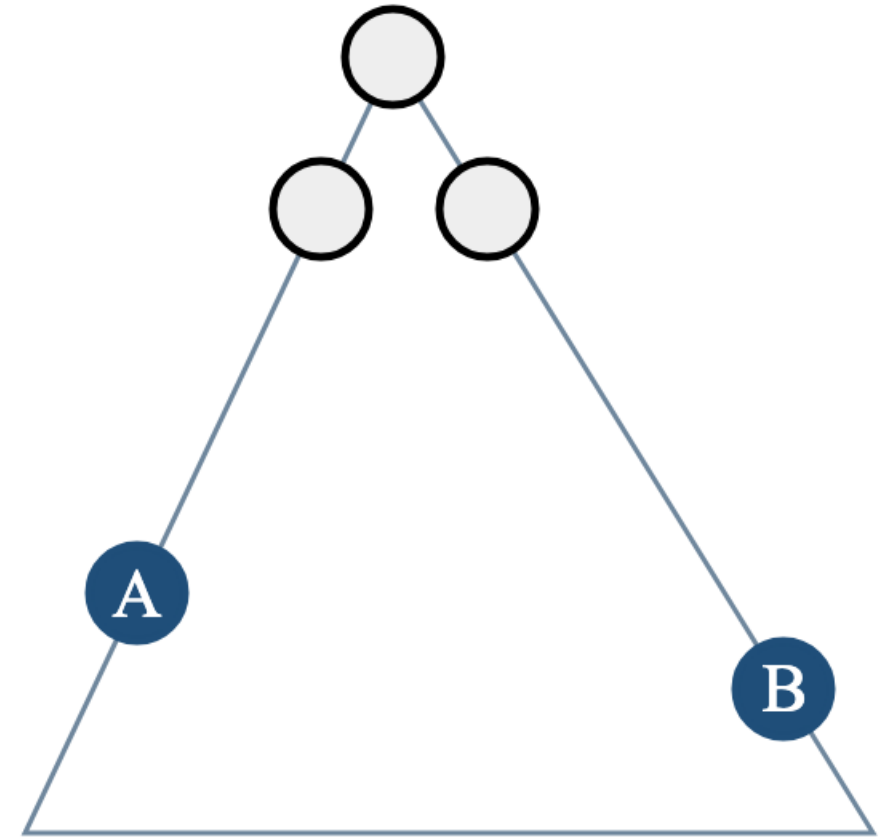
All ancestors of A exit the frontier before B



Optimality of A^*

All ancestors of A exit the frontier before B

A exits the frontier before B

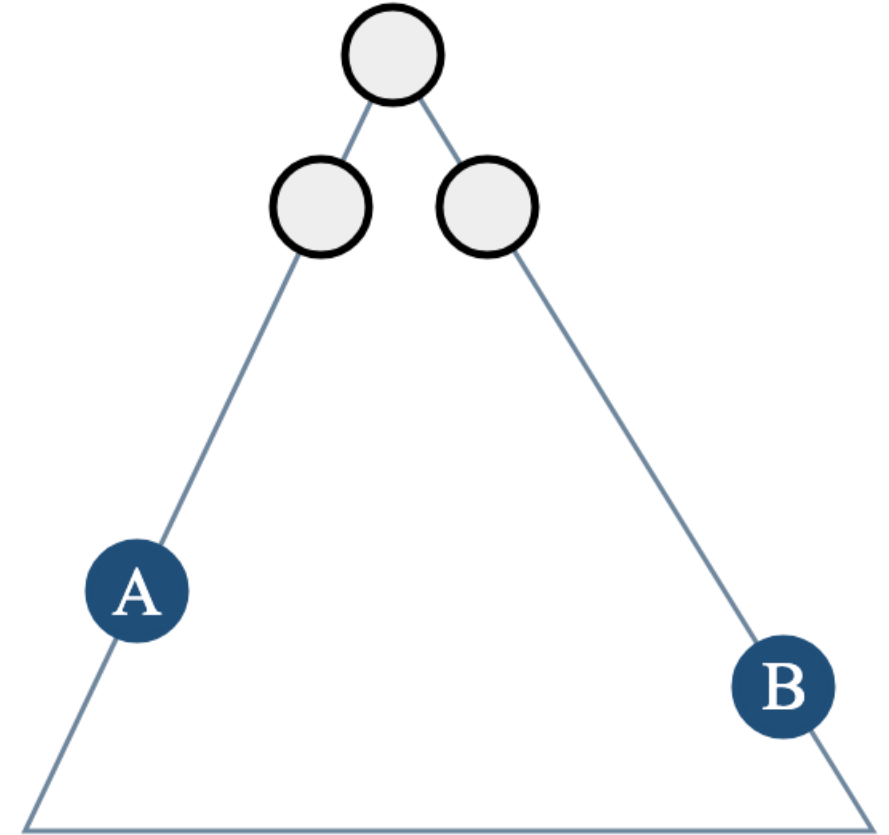


Optimality of A^*

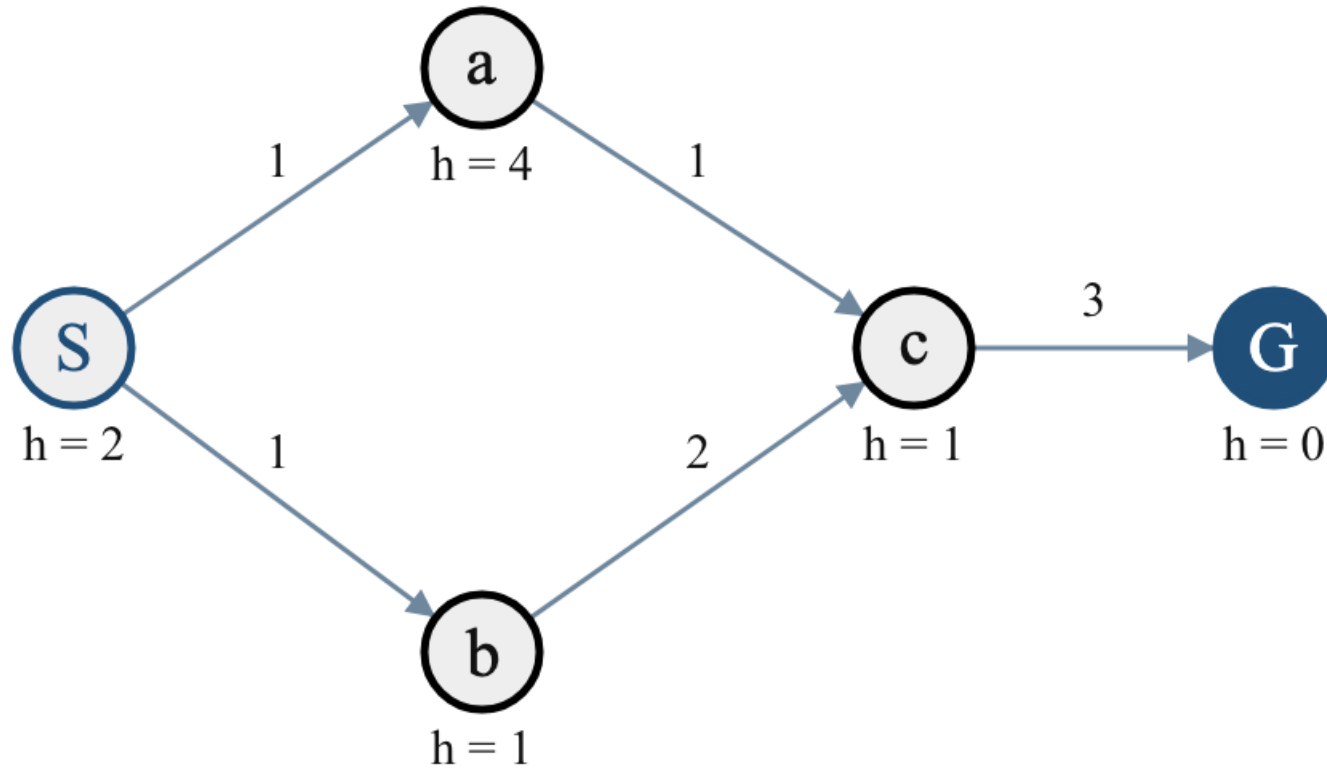
All ancestors of A exit the frontier before B

A exits the frontier before B

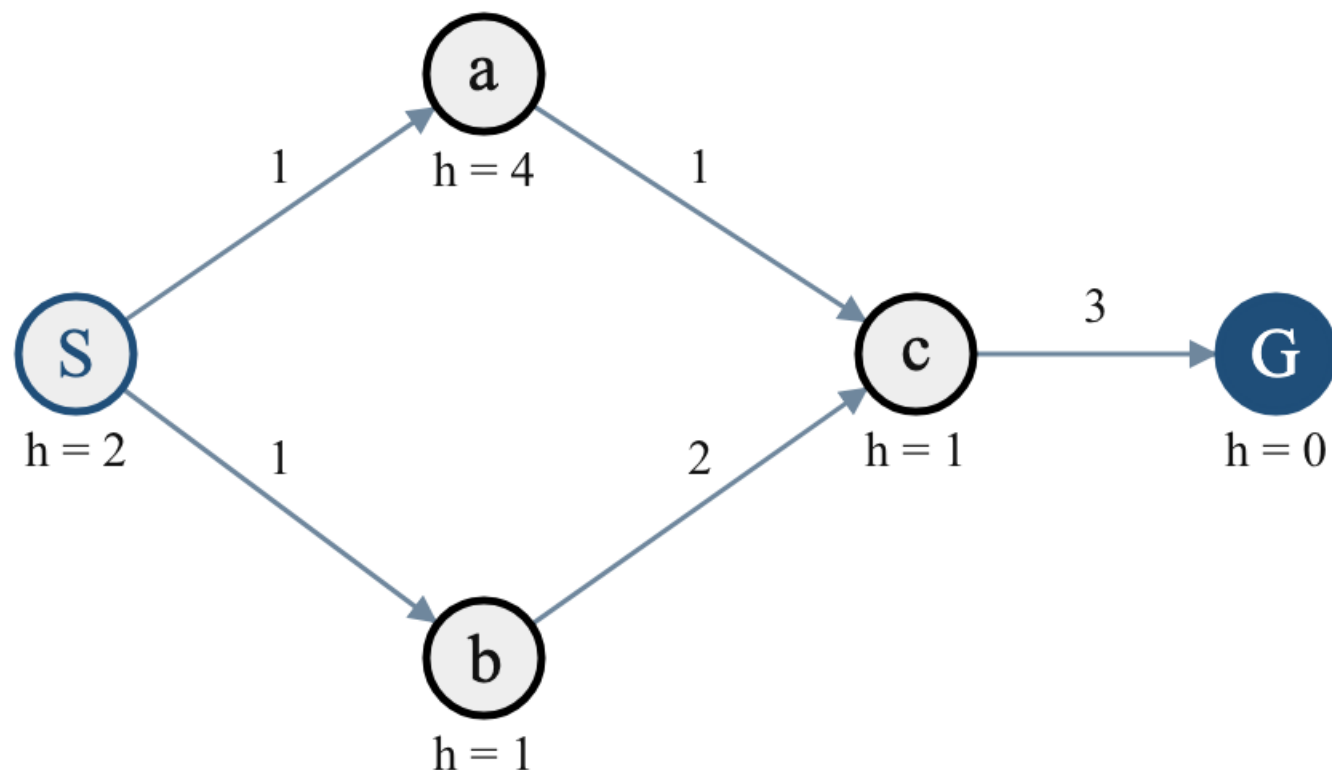
A^* -TSA is optimal with admissible h



Is This Heuristic Admissible?



Is This Heuristic Admissible?



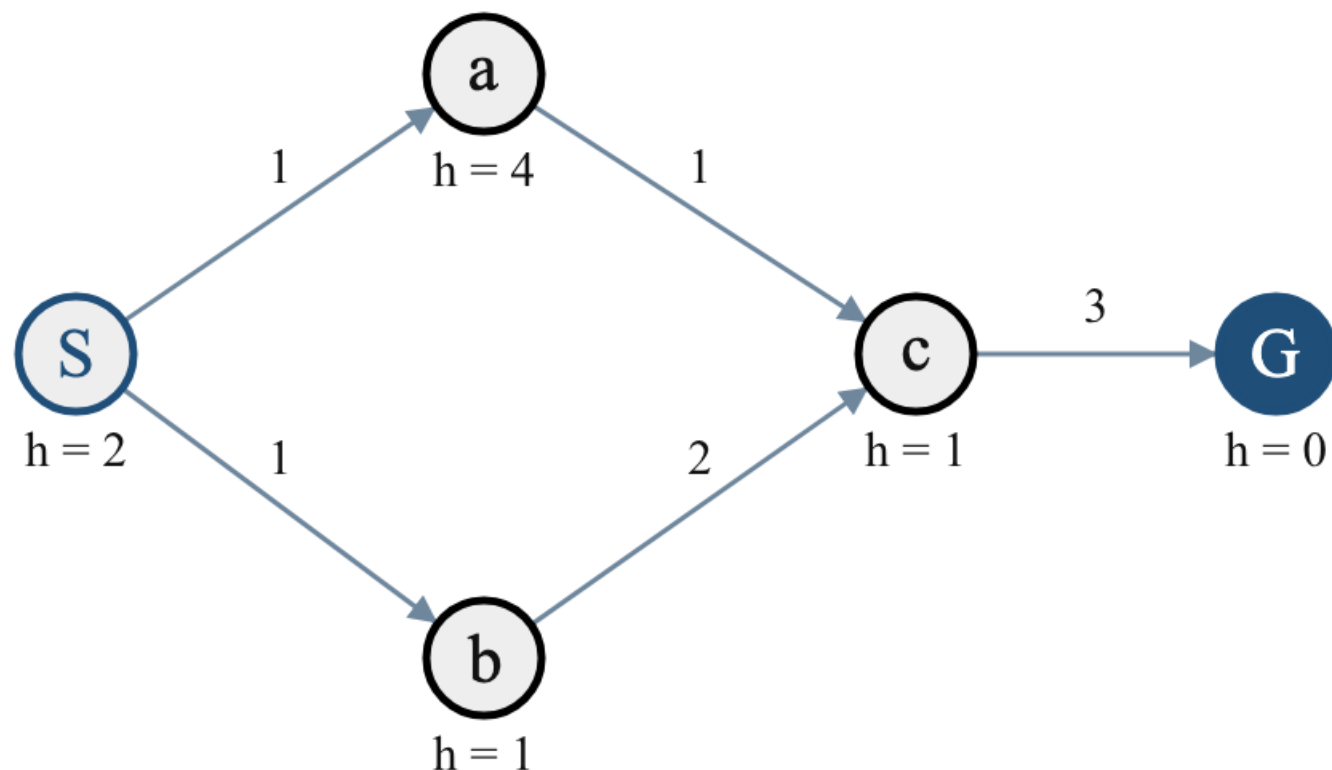
True remaining costs

$$h^*(G) = 0, h^*(c) = 3$$

$$h^*(a) = 4, h^*(b) = 5$$

$$h^*(S) = 5$$

Is This Heuristic Admissible?



True remaining costs

$$h^*(G) = 0, h^*(c) = 3$$

$$h^*(a) = 4, h^*(b) = 5$$

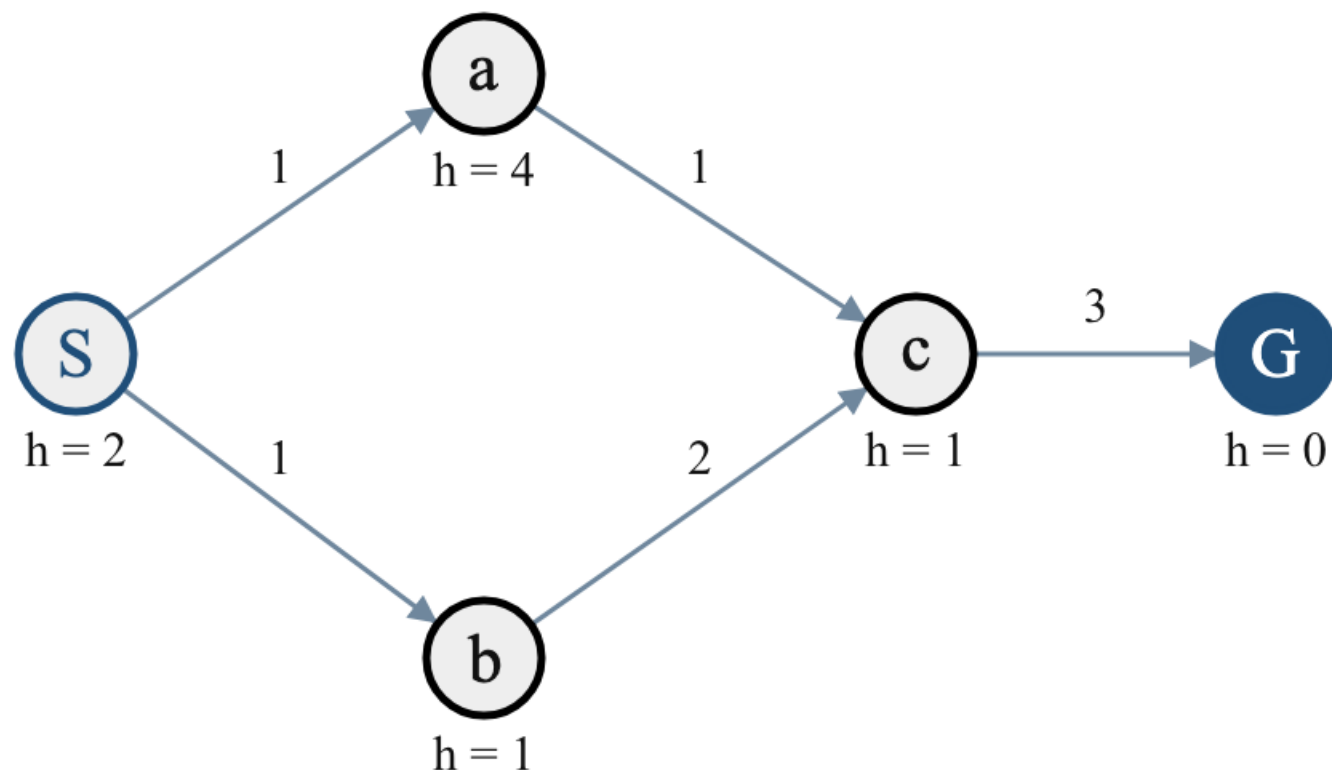
$$h^*(S) = 5$$

Compare with h

$$0 \leq 0 \quad \cdot \quad 1 \leq 3 \quad \cdot \quad 4 \leq 4$$

$$1 \leq 5 \quad \cdot \quad 2 \leq 5$$

Is This Heuristic Admissible?



True remaining costs

$$h^*(G) = 0, h^*(c) = 3$$

$$h^*(a) = 4, h^*(b) = 5$$

$$h^*(S) = 5$$

Compare with h

$$0 \leq 0 \quad \cdot \quad 1 \leq 3 \quad \cdot \quad 4 \leq 4$$

$$1 \leq 5 \quad \cdot \quad 2 \leq 5$$

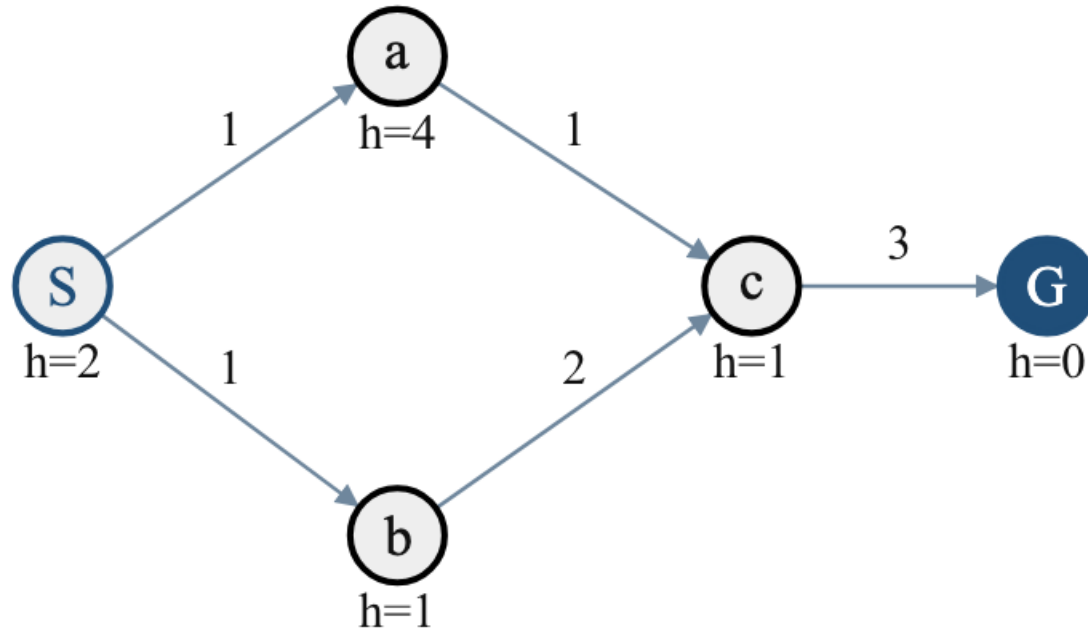
All estimates pass

Yes, this heuristic is admissible

Optimality of A*: TSA

Start: S · Goal: G

Initial frontier: [(2, 'S')]

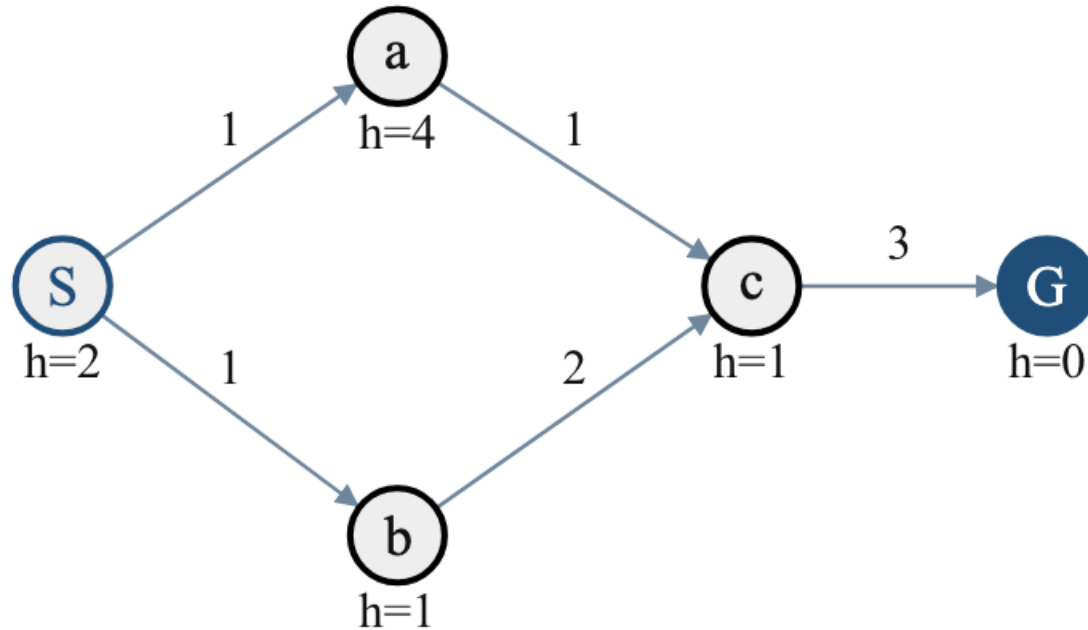


Optimality of A*: TSA

Start: S · Goal: G

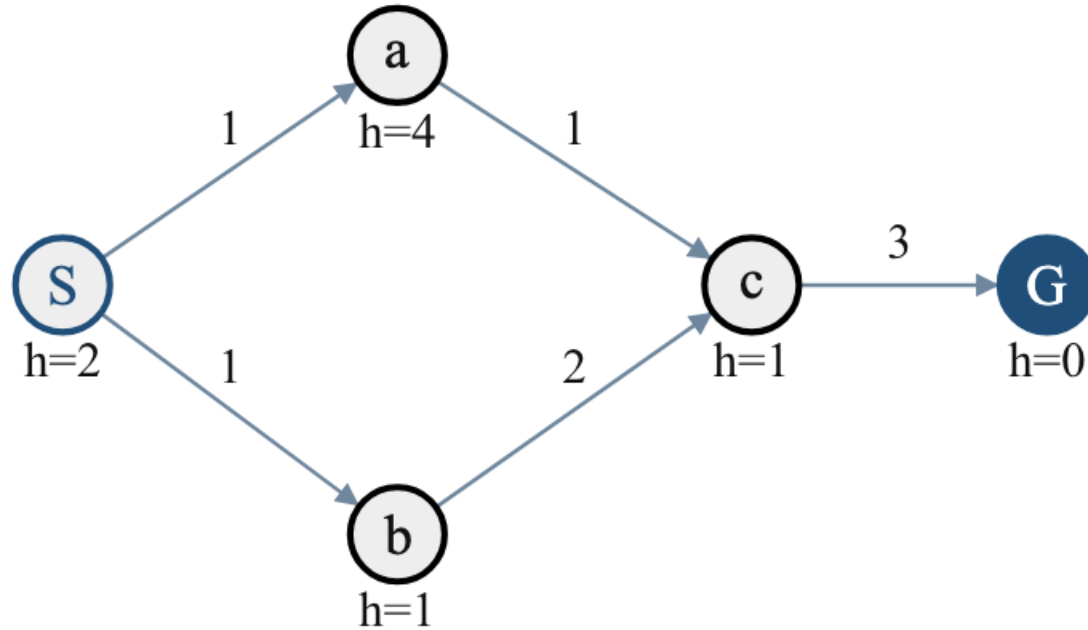
Initial frontier: [(2, 'S')]

S @ 2 $\rightarrow$ [(2, 'Sb'), (5, 'Sa')]



Optimality of A*: TSA

Start: S · Goal: G



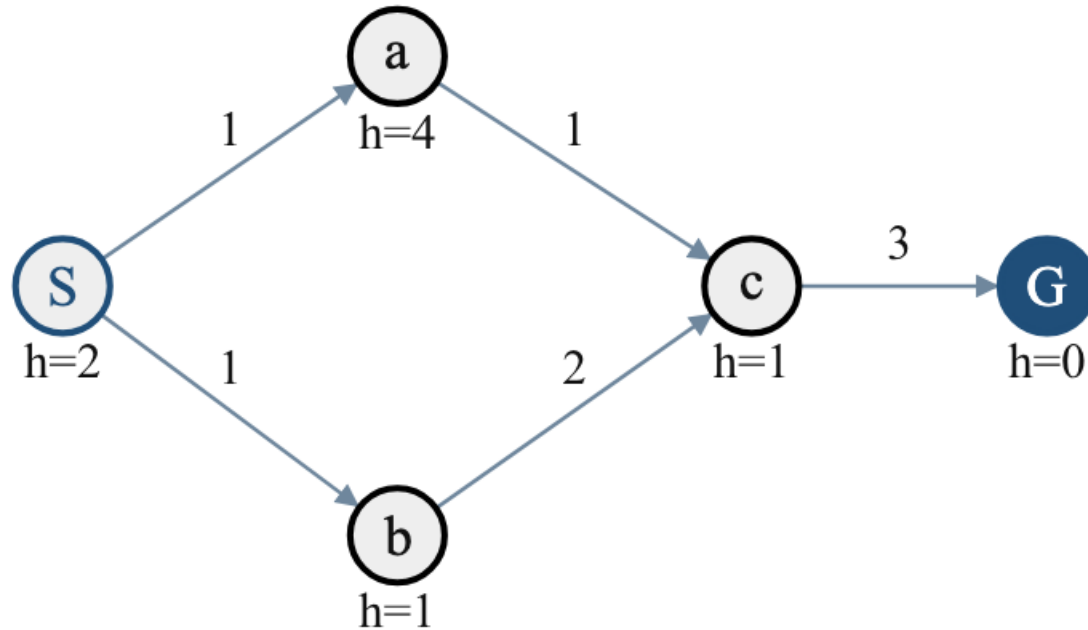
Initial frontier: [(2, 'S')]

S @ 2 $\rightarrow$ [(2, 'Sb'), (5, 'Sa')]

Sb @ 2 $\rightarrow$ [(4, 'Sbc'), (5, 'Sa')]

Optimality of A*: TSA

Start: S · Goal: G



Initial frontier: [(2, 'S')]

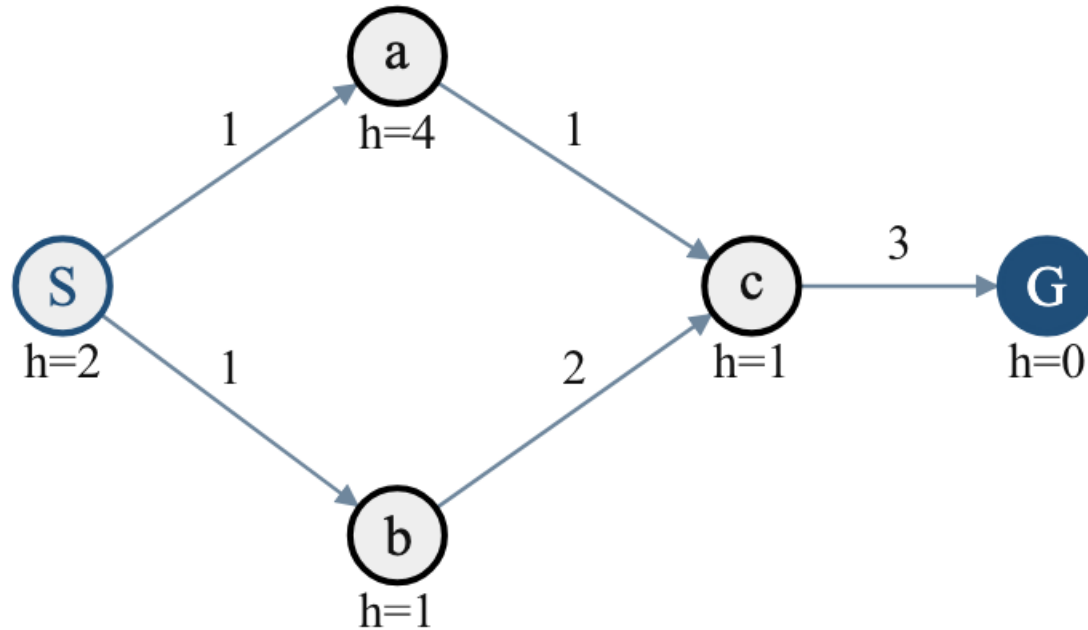
S @ 2 $\rightarrow$ [(2, 'Sb'), (5, 'Sa')]

Sb @ 2 $\rightarrow$ [(4, 'Sbc'), (5, 'Sa')]

Sbc @ 4 $\rightarrow$ [(5, 'Sa'), (6, 'SbcG')]

Optimality of A*: TSA

Start: S · Goal: G



Initial frontier: [(2, 'S')]

S @ 2 $\rightarrow$ [(2, 'Sb'), (5, 'Sa')]

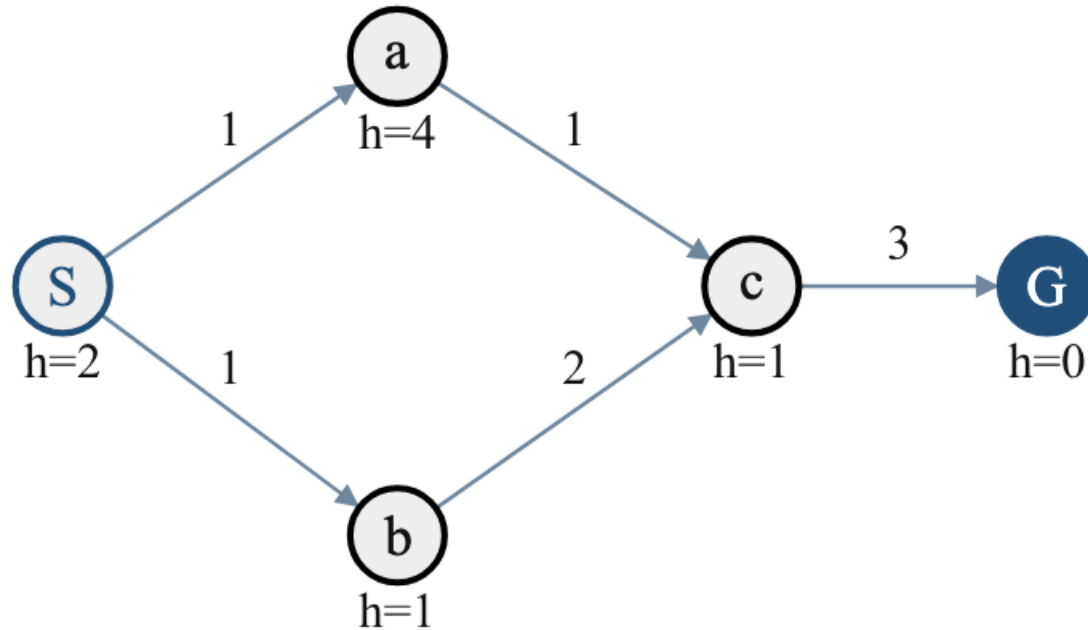
Sb @ 2 $\rightarrow$ [(4, 'Sbc'), (5, 'Sa')]

Sbc @ 4 $\rightarrow$ [(5, 'Sa'), (6, 'SbcG')]

Sa @ 5 $\rightarrow$ [(3, 'Sac'), (6, 'SbcG')]

Optimality of A*: TSA

Start: S · Goal: G



Initial frontier: [(2, 'S')]

S @ 2 $\rightarrow$ [(2, 'Sb'), (5, 'Sa')]

Sb @ 2 $\rightarrow$ [(4, 'Sbc'), (5, 'Sa')]

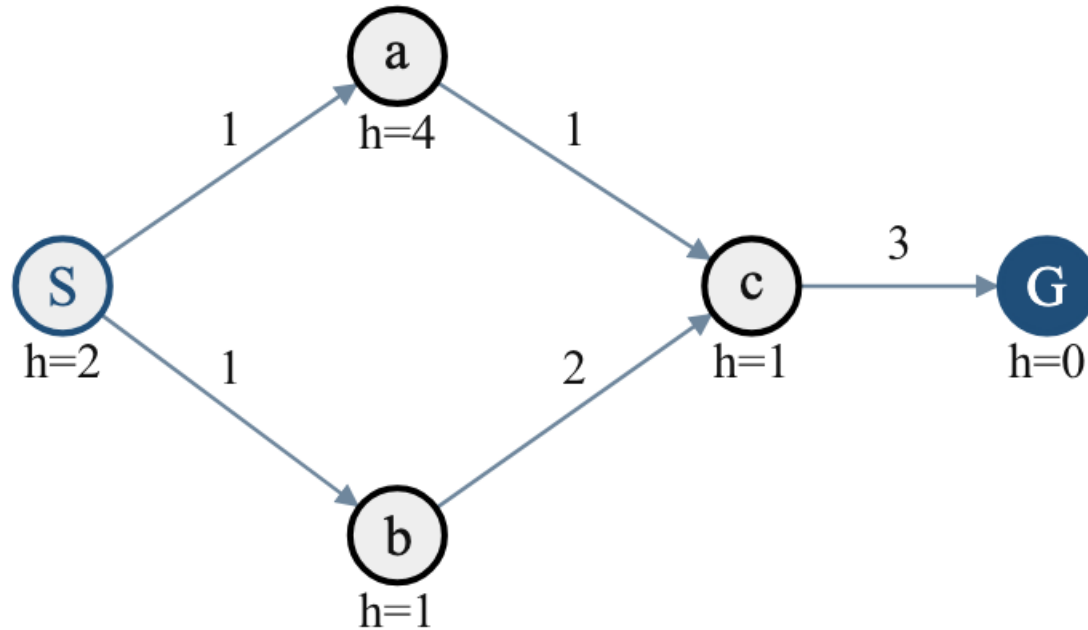
Sbc @ 4 $\rightarrow$ [(5, 'Sa'), (6, 'SbcG')]

Sa @ 5 $\rightarrow$ [(3, 'Sac'), (6, 'SbcG')]

Sac @ 3 $\rightarrow$ [(5, 'SacG'), (6, 'SbcG')]

Optimality of A*: TSA

Start: S · Goal: G



Initial frontier: [(2, 'S')]

S @ 2 $\rightarrow$ [(2, 'Sb'), (5, 'Sa')]

Sb @ 2 $\rightarrow$ [(4, 'Sbc'), (5, 'Sa')]

Sbc @ 4 $\rightarrow$ [(5, 'Sa'), (6, 'SbcG')]

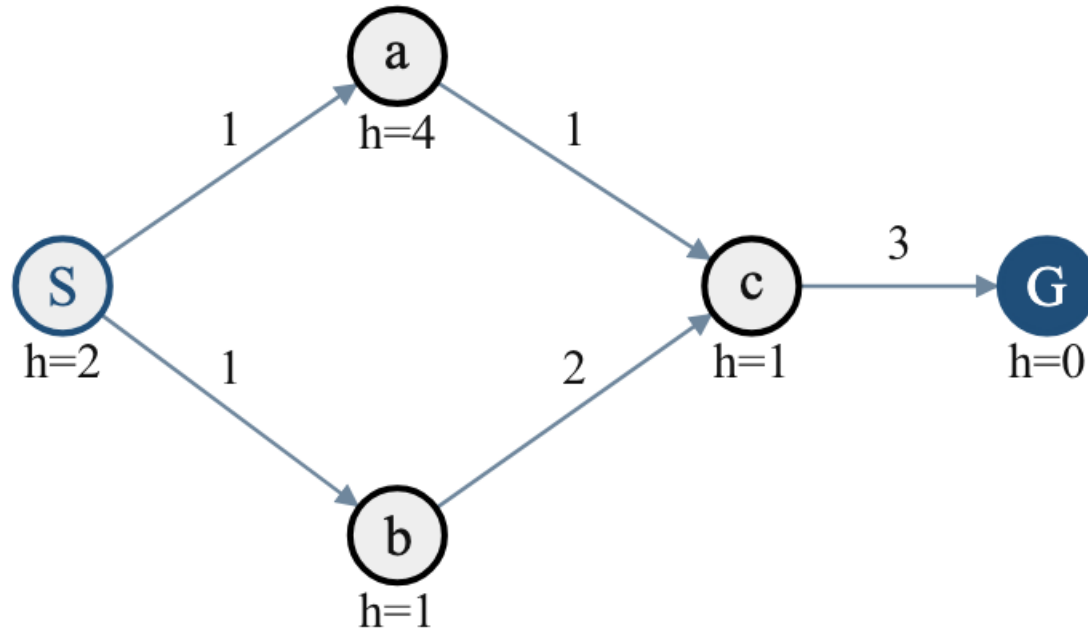
Sa @ 5 $\rightarrow$ [(3, 'Sac'), (6, 'SbcG')]

Sac @ 3 $\rightarrow$ [(5, 'SacG'), (6, 'SbcG')]

Solution path: (5, 'SacG')

Optimality of A*: TSA

Start: S · Goal: G



Initial frontier: [(2, 'S')]

S @ 2 $\rightarrow$ [(2, 'Sb'), (5, 'Sa')]

Sb @ 2 $\rightarrow$ [(4, 'Sbc'), (5, 'Sa')]

Sbc @ 4 $\rightarrow$ [(5, 'Sa'), (6, 'SbcG')]

Sa @ 5 $\rightarrow$ [(3, 'Sac'), (6, 'SbcG')]

Sac @ 3 $\rightarrow$ [(5, 'SacG'), (6, 'SbcG')]

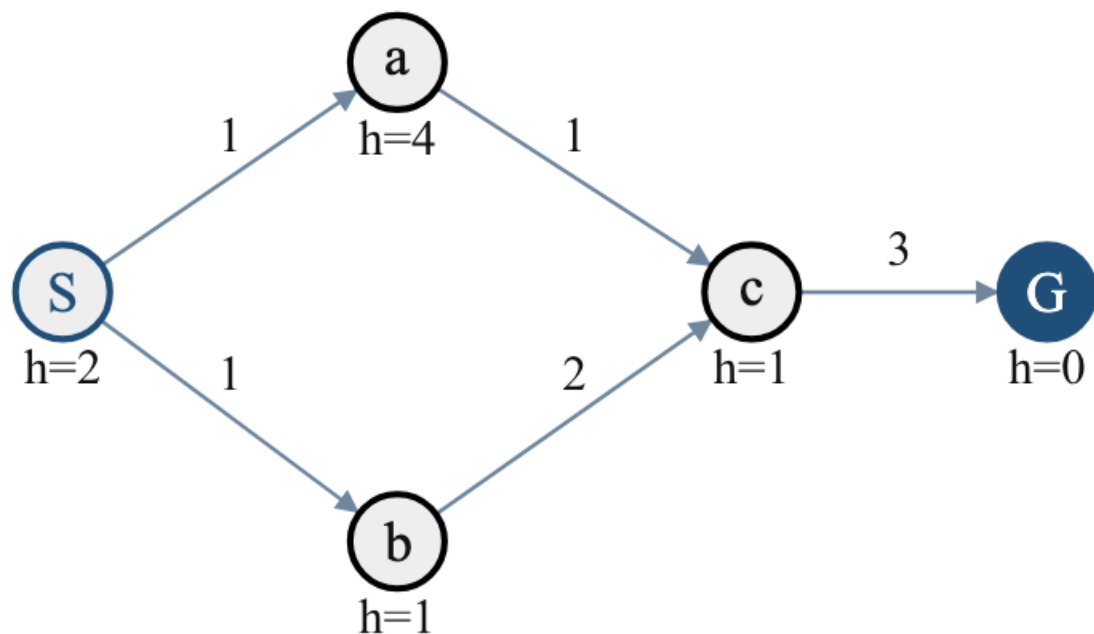
Solution path: (5, 'SacG')

Optimal path: SacG, cost 5

Optimality of A*: GSA

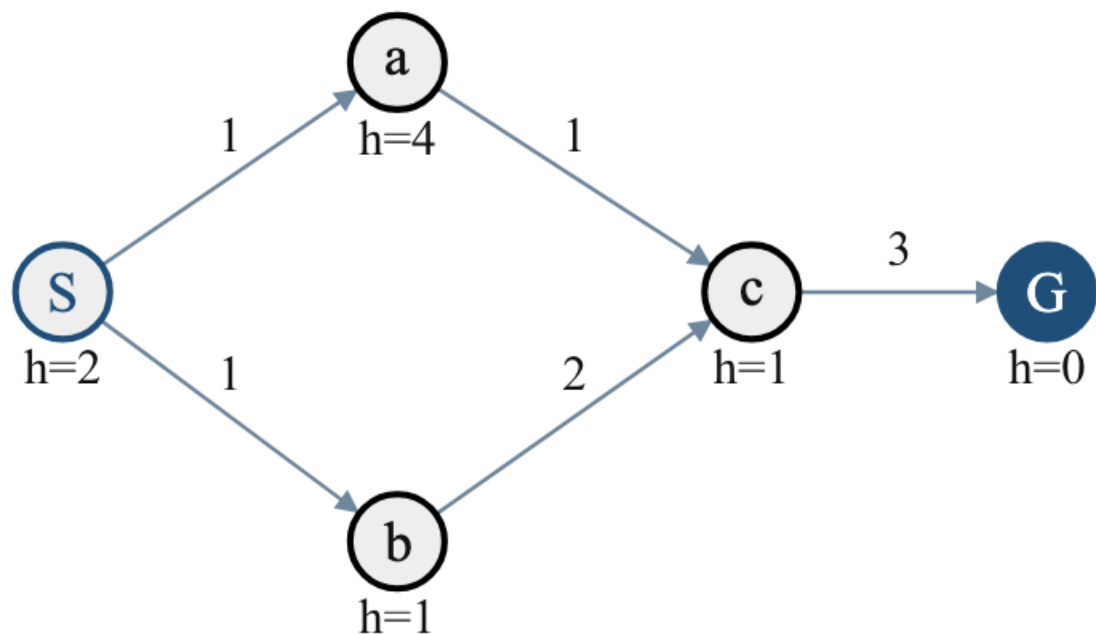
Our GSA never expands a state twice

Initial frontier: [(2, 'S')]
Explored: $\emptyset$



Optimality of A*: GSA

Our GSA never expands a state twice



Initial frontier: [(2, 'S')]

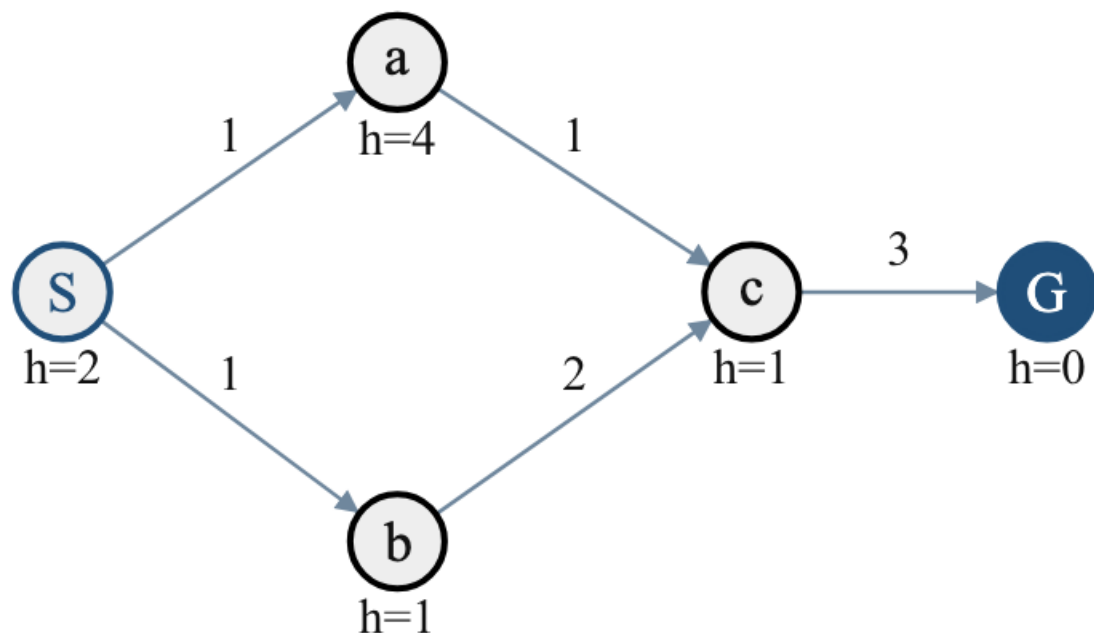
Explored: $\emptyset$

S @ 2 $\rightarrow$ [(2, 'Sb'), (5, 'Sa')]

Explored: {S}

Optimality of A*: GSA

Our GSA never expands a state twice



Initial frontier: [(2, 'S')]

Explored: $\emptyset$

S @ 2 $\rightarrow$ [(2, 'Sb'), (5, 'Sa')]

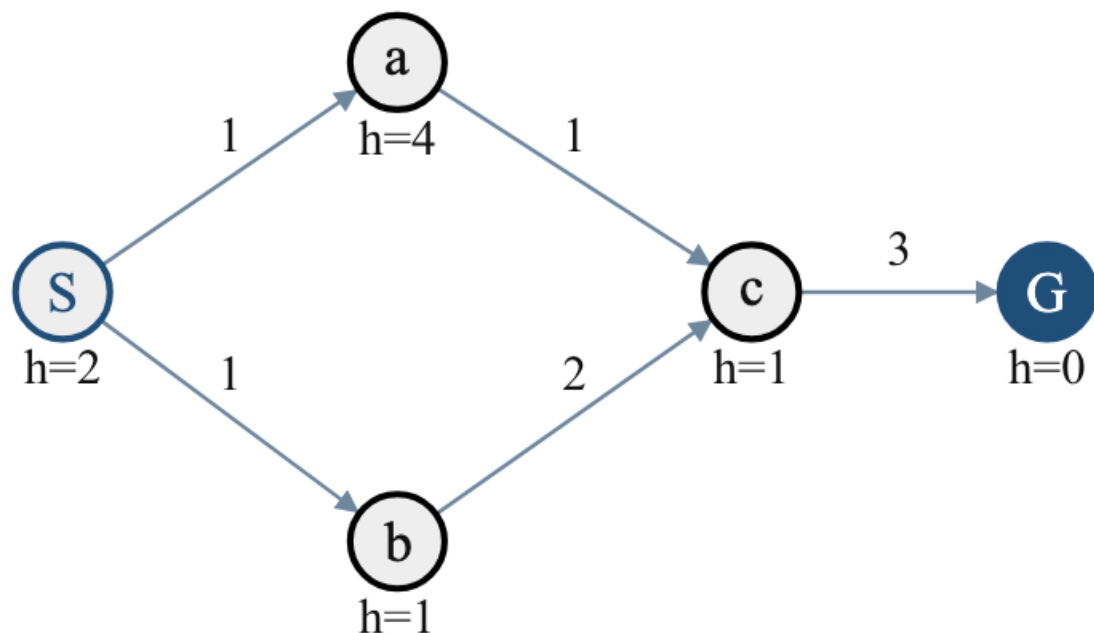
Explored: {S}

Sb @ 2 $\rightarrow$ [(4, 'Sbc'), (5, 'Sa')]

Explored: {S, b}

Optimality of A*: GSA

Our GSA never expands a state twice



Initial frontier: [(2, 'S')]

Explored: $\emptyset$

S @ 2 $\rightarrow$ [(2, 'Sb'), (5, 'Sa')]

Explored: {S}

Sb @ 2 $\rightarrow$ [(4, 'Sbc'), (5, 'Sa')]

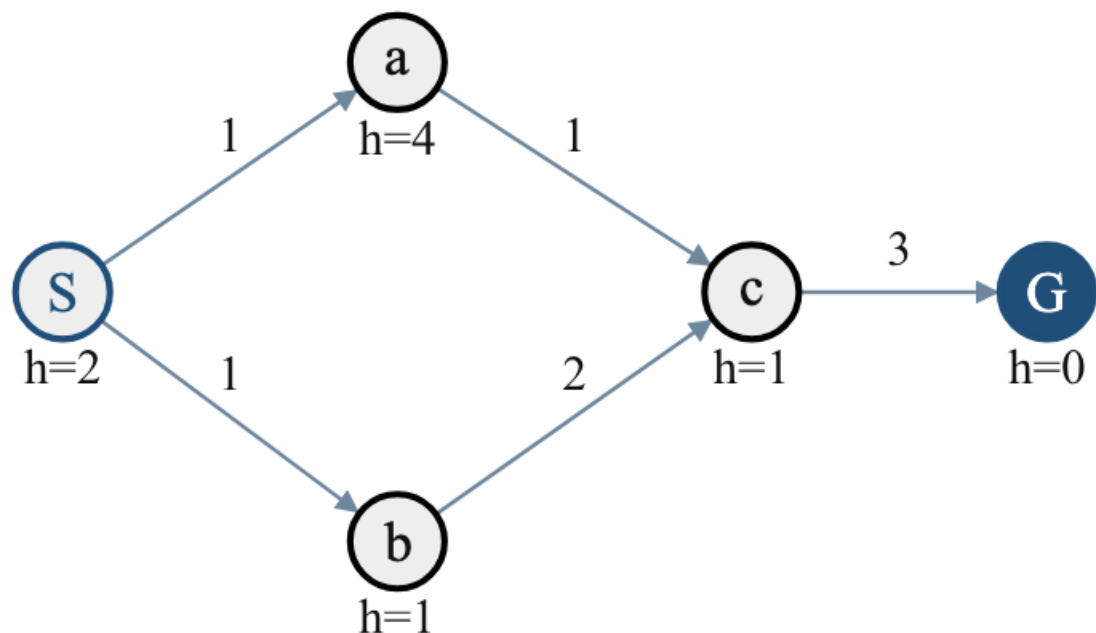
Explored: {S, b}

Sbc @ 4 $\rightarrow$ [(5, 'Sa'), (6, 'SbcG')]

Explored: {S, b, c}

Optimality of A*: GSA

Our GSA never expands a state twice



Initial frontier: [(2, 'S')]

Explored: $\emptyset$

S @ 2 $\rightarrow$ [(2, 'Sb'), (5, 'Sa')]

Explored: {S}

Sb @ 2 $\rightarrow$ [(4, 'Sbc'), (5, 'Sa')]

Explored: {S, b}

Sbc @ 4 $\rightarrow$ [(5, 'Sa'), (6, 'SbcG')]

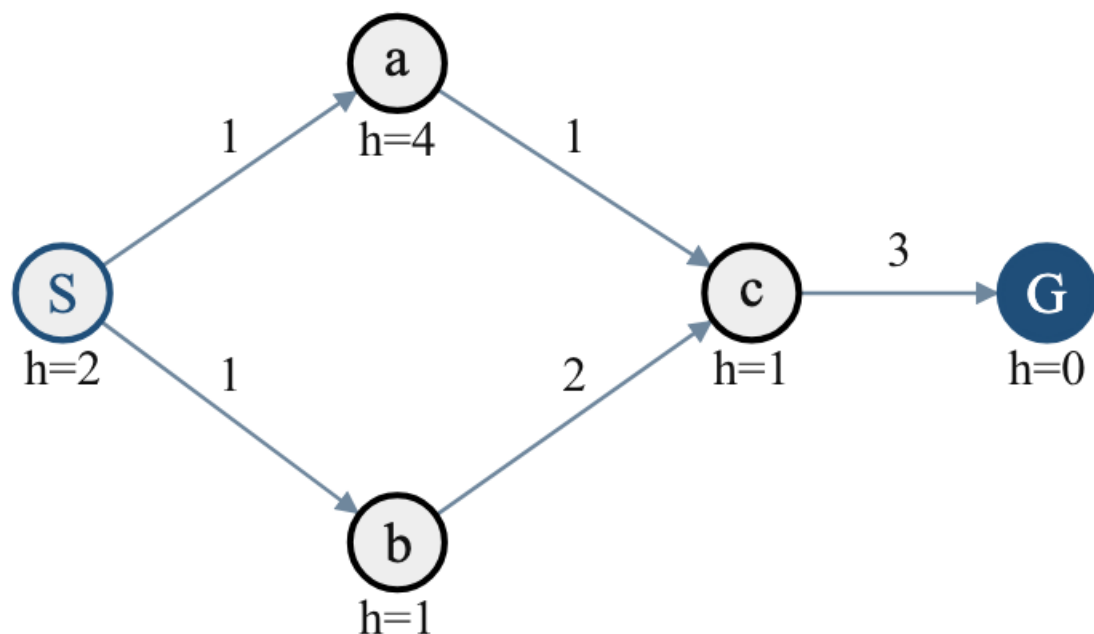
Explored: {S, b, c}

Sa @ 5 $\rightarrow$ [(3, 'Sac'), (6, 'SbcG')]

Explored: {S, b, c, a}

Optimality of A*: GSA

Our GSA never expands a state twice



Initial frontier: [(2, 'S')]

Explored: $\emptyset$

S @ 2 $\rightarrow$ [(2, 'Sb'), (5, 'Sa')]

Explored: {S}

Sb @ 2 $\rightarrow$ [(4, 'Sbc'), (5, 'Sa')]

Explored: {S, b}

Sbc @ 4 $\rightarrow$ [(5, 'Sa'), (6, 'SbcG')]

Explored: {S, b, c}

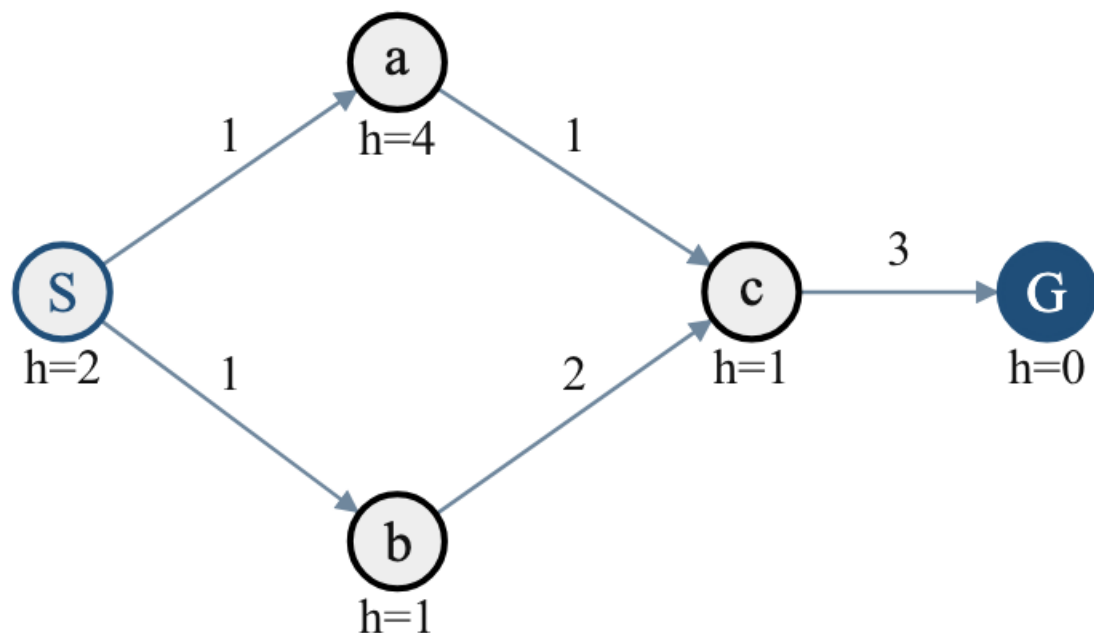
Sa @ 5 $\rightarrow$ [(3, 'Sac'), (6, 'SbcG')]

Explored: {S, b, c, a}

Skip Sac: c already explored

Optimality of A*: GSA

Our GSA never expands a state twice



Initial frontier: [(2, 'S')]

Explored: $\emptyset$

S @ 2 $\rightarrow$ [(2, 'Sb'), (5, 'Sa')]

Explored: {S}

Sb @ 2 $\rightarrow$ [(4, 'Sbc'), (5, 'Sa')]

Explored: {S, b}

Sbc @ 4 $\rightarrow$ [(5, 'Sa'), (6, 'SbcG')]

Explored: {S, b, c}

Sa @ 5 $\rightarrow$ [(3, 'Sac'), (6, 'SbcG')]

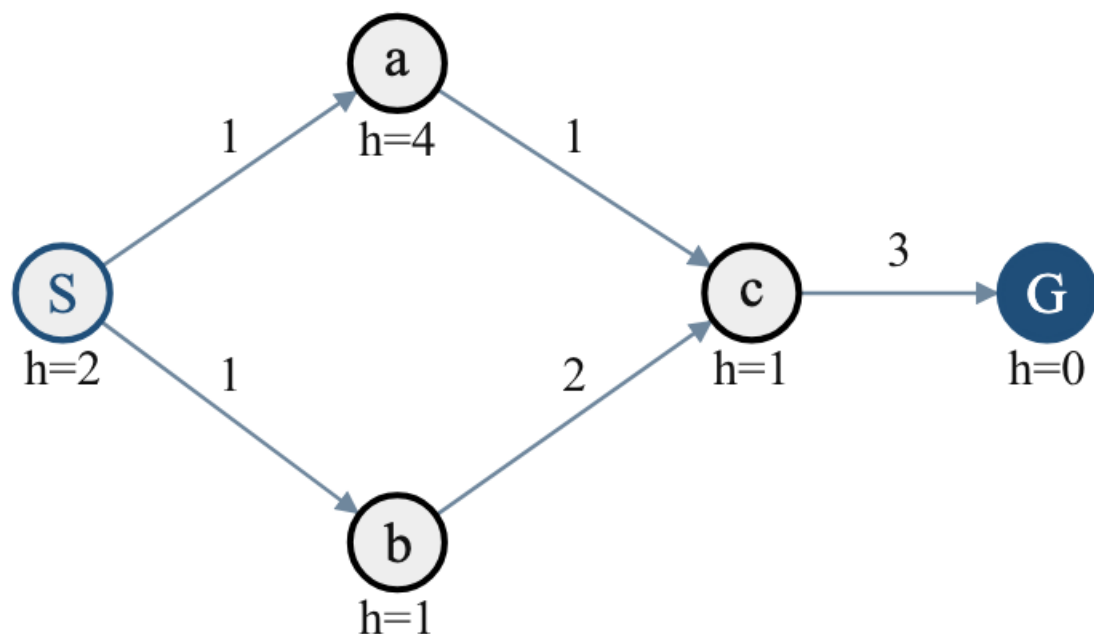
Explored: {S, b, c, a}

Skip Sac: c already explored

Solution path: (6, 'SbcG')

Optimality of A*: GSA

Our GSA never expands a state twice



Initial frontier: [(2, 'S')]

Explored: $\emptyset$

S @ 2 $\rightarrow$ [(2, 'Sb'), (5, 'Sa')]

Explored: {S}

Sb @ 2 $\rightarrow$ [(4, 'Sbc'), (5, 'Sa')]

Explored: {S, b}

Sbc @ 4 $\rightarrow$ [(5, 'Sa'), (6, 'SbcG')]

Explored: {S, b, c}

Sa @ 5 $\rightarrow$ [(3, 'Sac'), (6, 'SbcG')]

Explored: {S, b, c, a}

Skip Sac: c already explored

Solution path: (6, 'SbcG')

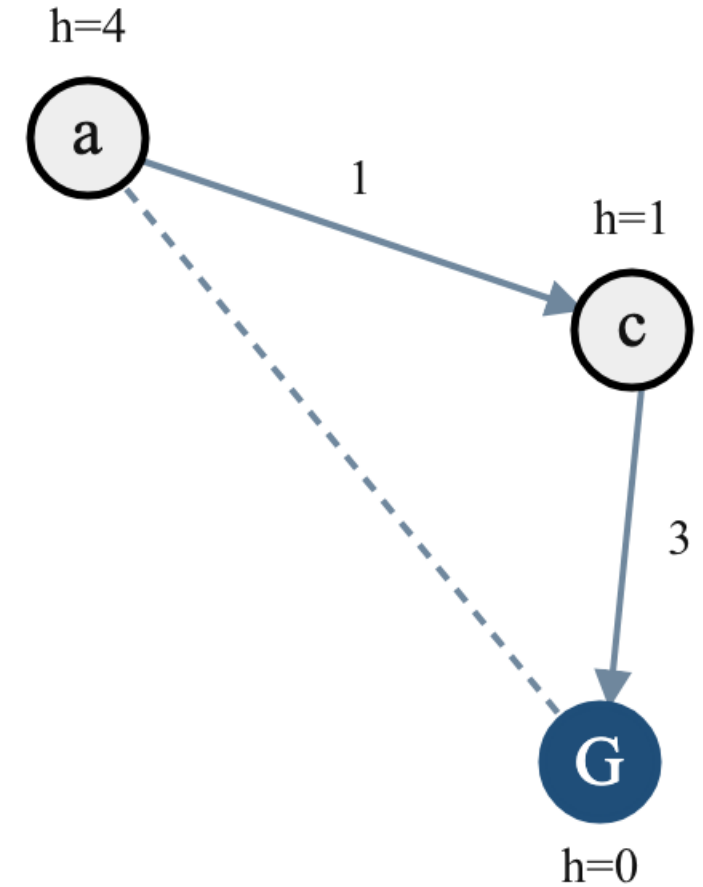
What went wrong?

Consistency of Heuristic

Definition

For every arc $a \rightarrow c$

$$h(a) - h(c) \leq \text{cost}(a,c)$$



Consistency of Heuristic

Definition

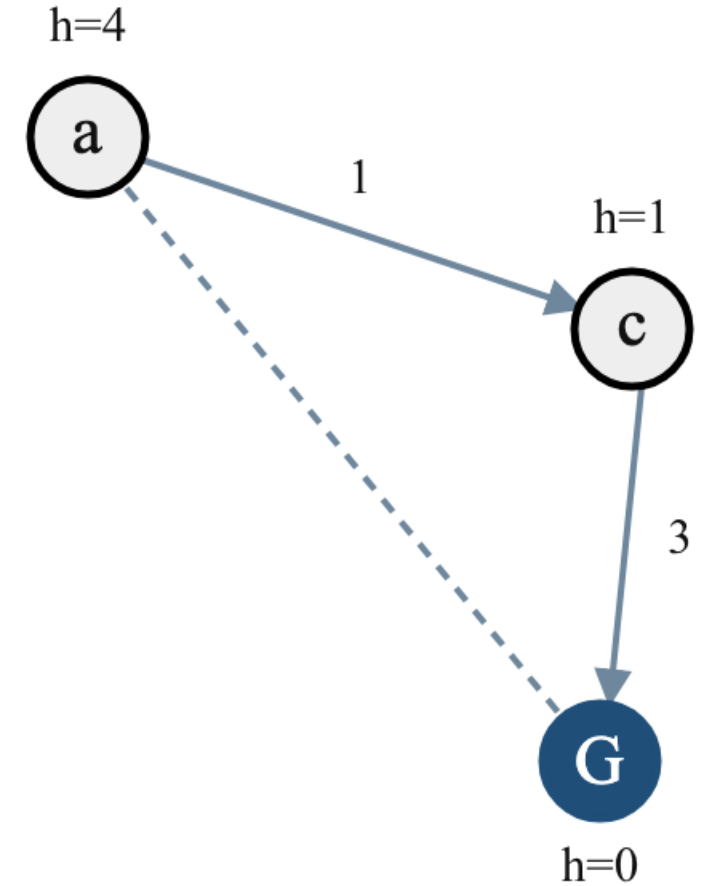
For every arc $a \rightarrow c$

$$h(a) - h(c) \leq \text{cost}(a,c)$$

Equivalently

$$h(a) \leq \text{cost}(a,c) + h(c)$$

Here: $4 \leq 1 + 1$ is false



Consistency of Heuristic

Definition

For every arc $a \rightarrow c$

$$h(a) - h(c) \leq \text{cost}(a,c)$$

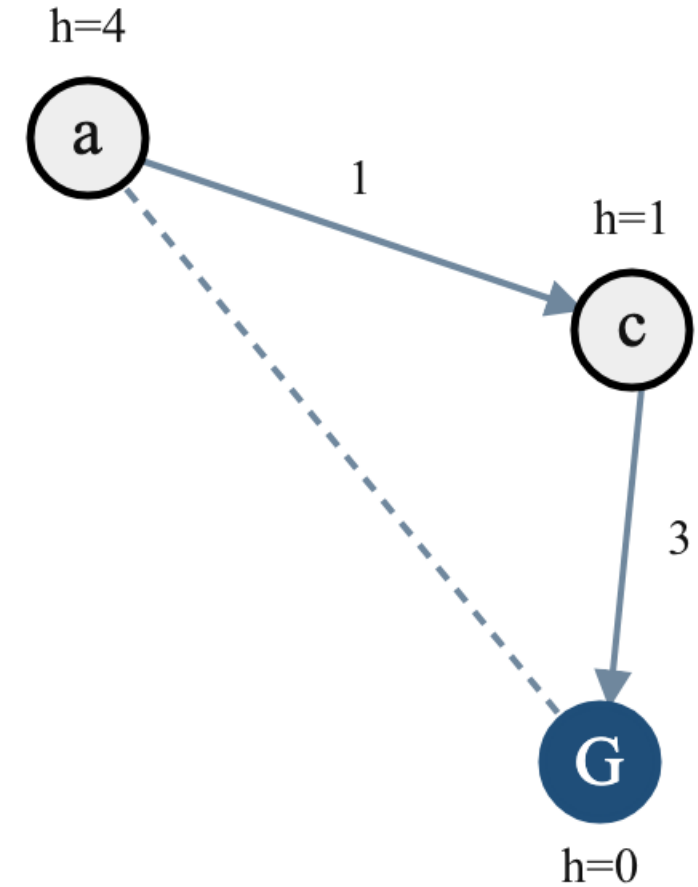
Equivalently

$$h(a) \leq \text{cost}(a,c) + h(c)$$

Here: $4 \leq 1 + 1$ is false

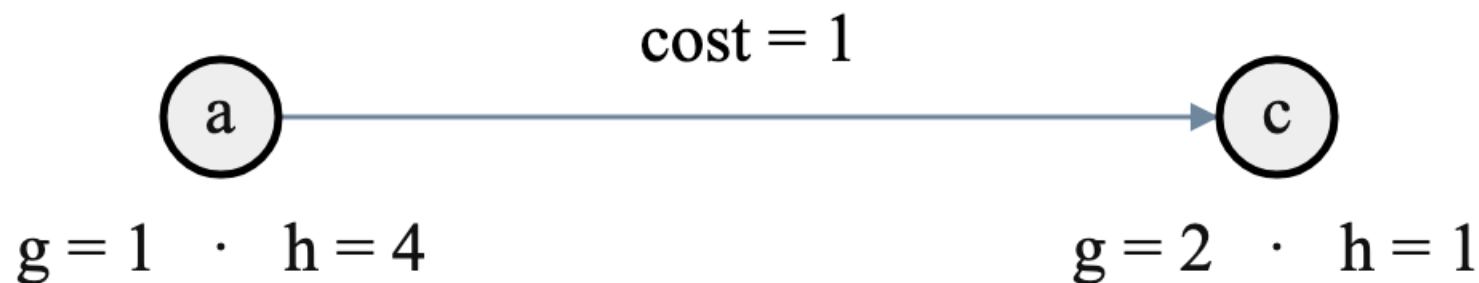
Consequence of consistency

The f value along a path never decreases



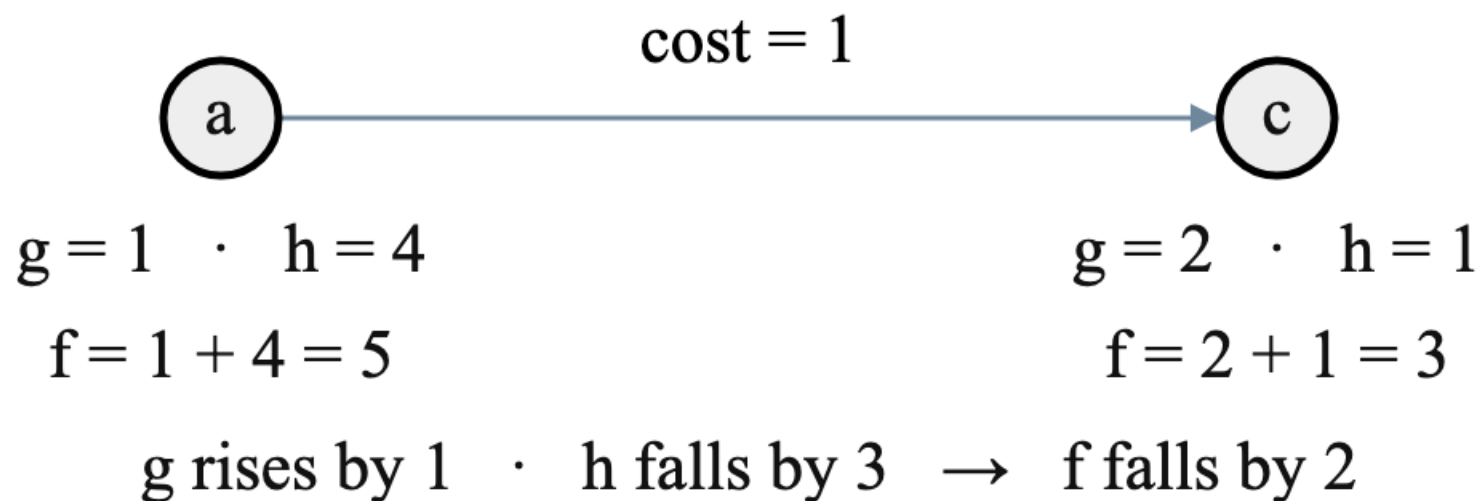
Consistency Keeps f from Decreasing

Recall the cheaper path $S \rightarrow a \rightarrow c$



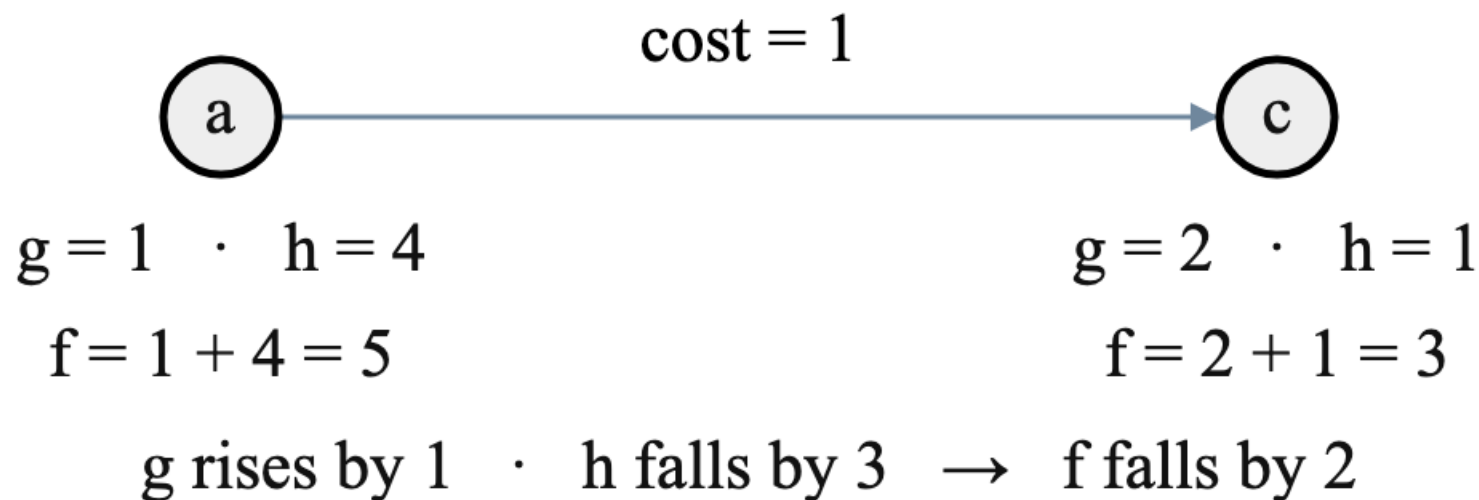
Consistency Keeps f from Decreasing

Recall the cheaper path $S \rightarrow a \rightarrow c$



Consistency Keeps f from Decreasing

Recall the cheaper path $S \rightarrow a \rightarrow c$

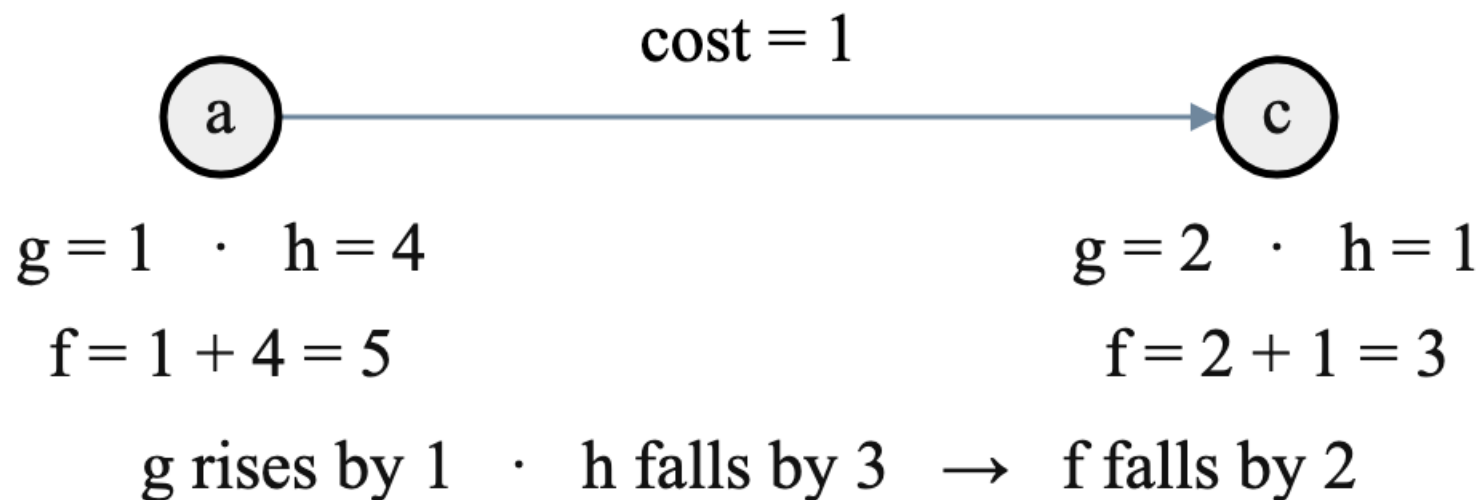


Consistency: h cannot fall by more than the step costs

$$f(v) - f(u) = \text{cost}(u, v) - (h(u) - h(v)) \geq 0$$

Consistency Keeps f from Decreasing

Recall the cheaper path $S \rightarrow a \rightarrow c$



Consistency: h cannot fall by more than the step costs

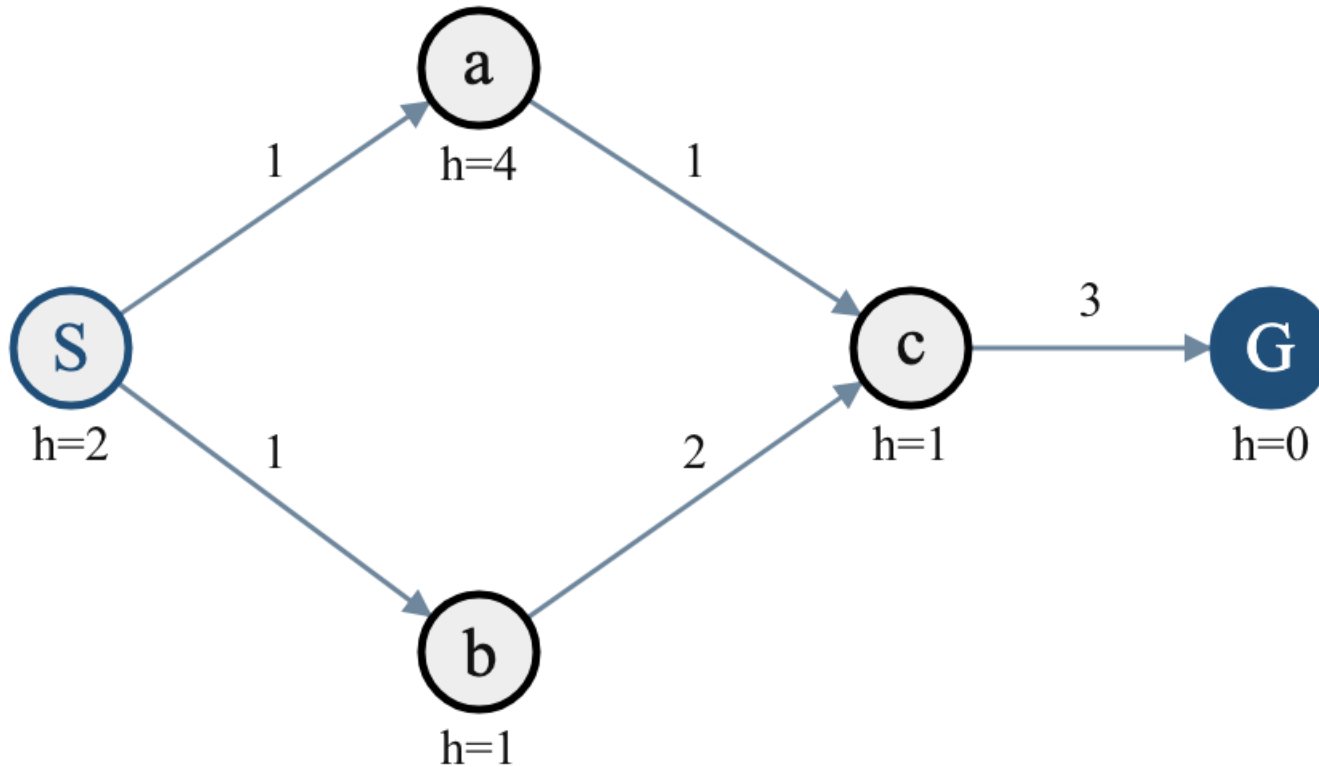
$$f(v) - f(u) = \text{cost}(u, v) - (h(u) - h(v)) \geq 0$$

With consistent h , GSA first expands each state via a cheapest path
Closing the state cannot block a cheaper route later

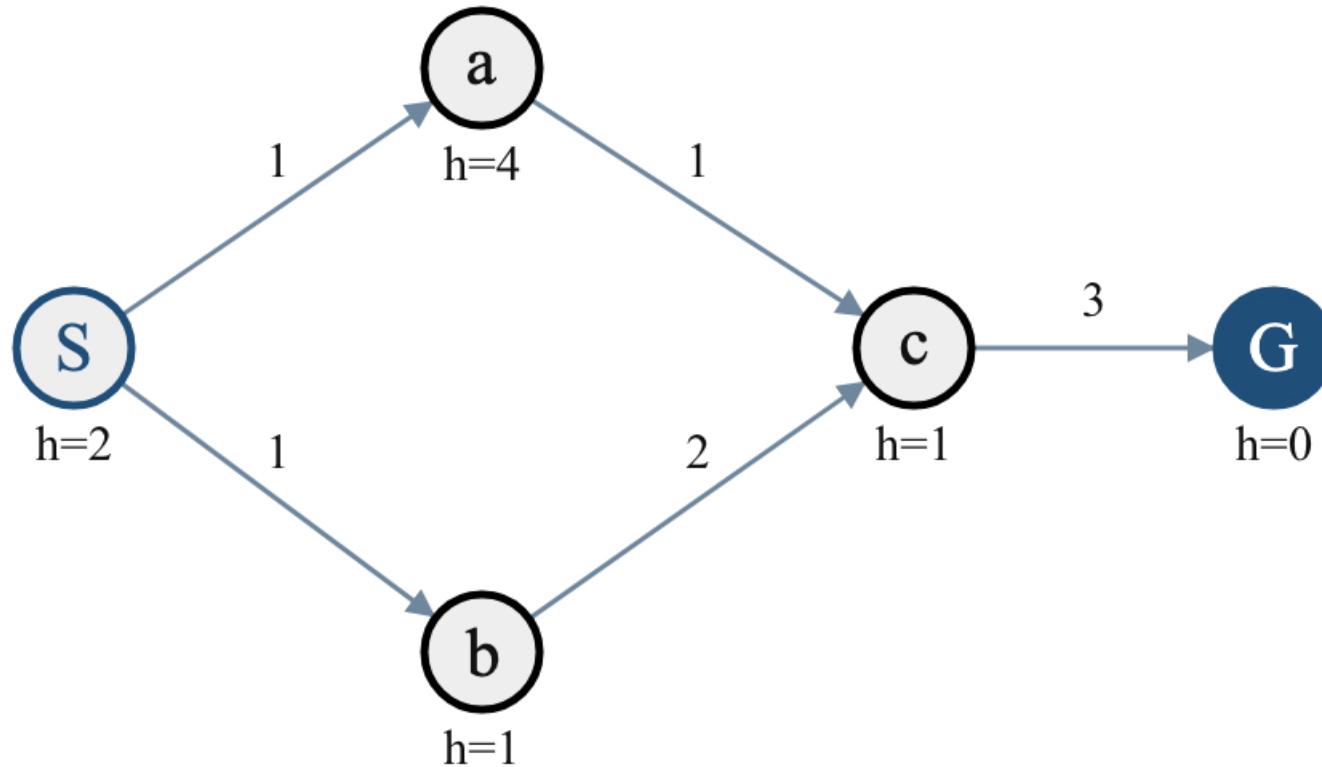
Consistency of Heuristic

Is h consistent for this graph?

$$h(u) \leq \text{cost}(u, v) + h(v)$$



Consistency of Heuristic

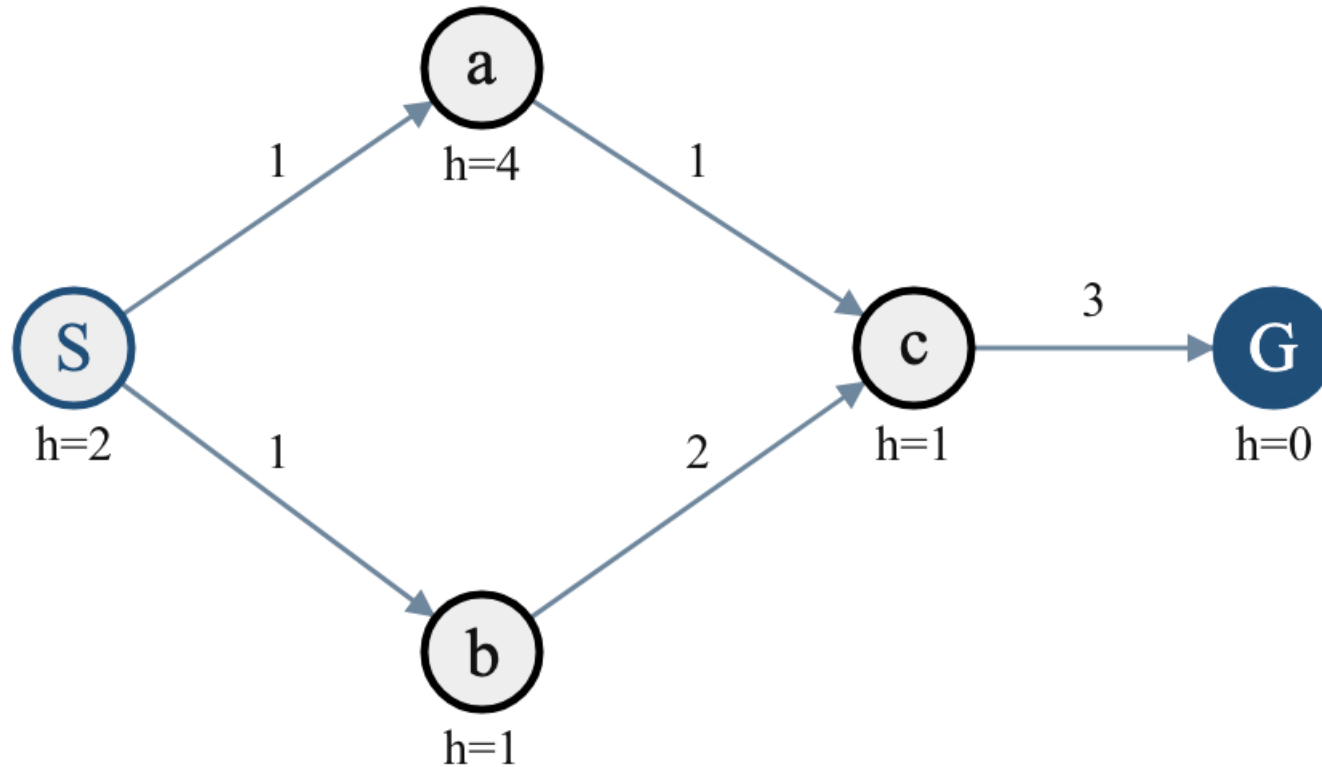


Is h consistent for this graph?

$$h(u) \leq \text{cost}(u, v) + h(v)$$

$$S \rightarrow a: 2 \leq 1 + 4$$

Consistency of Heuristic



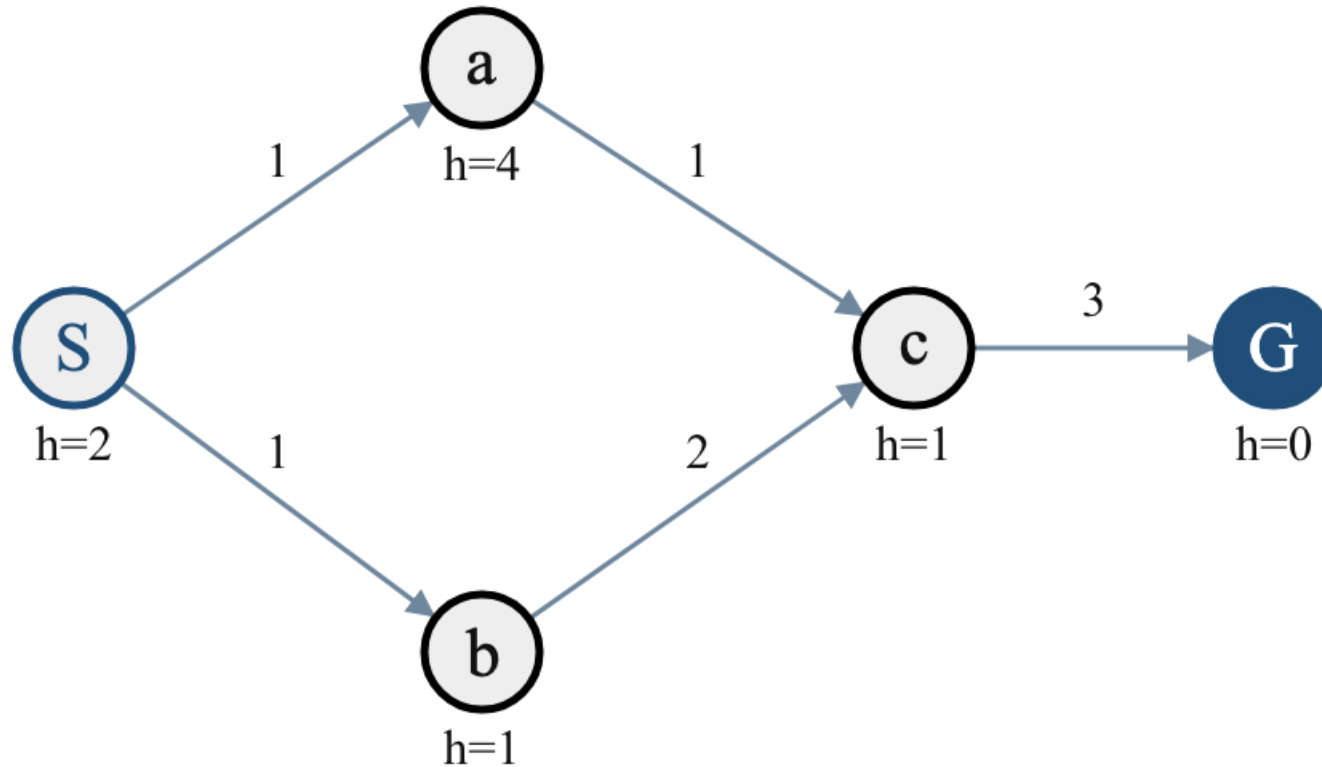
Is h consistent for this graph?

$$h(u) \leq \text{cost}(u, v) + h(v)$$

$$S \rightarrow a: 2 \leq 1 + 4$$

$$S \rightarrow b: 2 \leq 1 + 1$$

Consistency of Heuristic



Is h consistent for this graph?

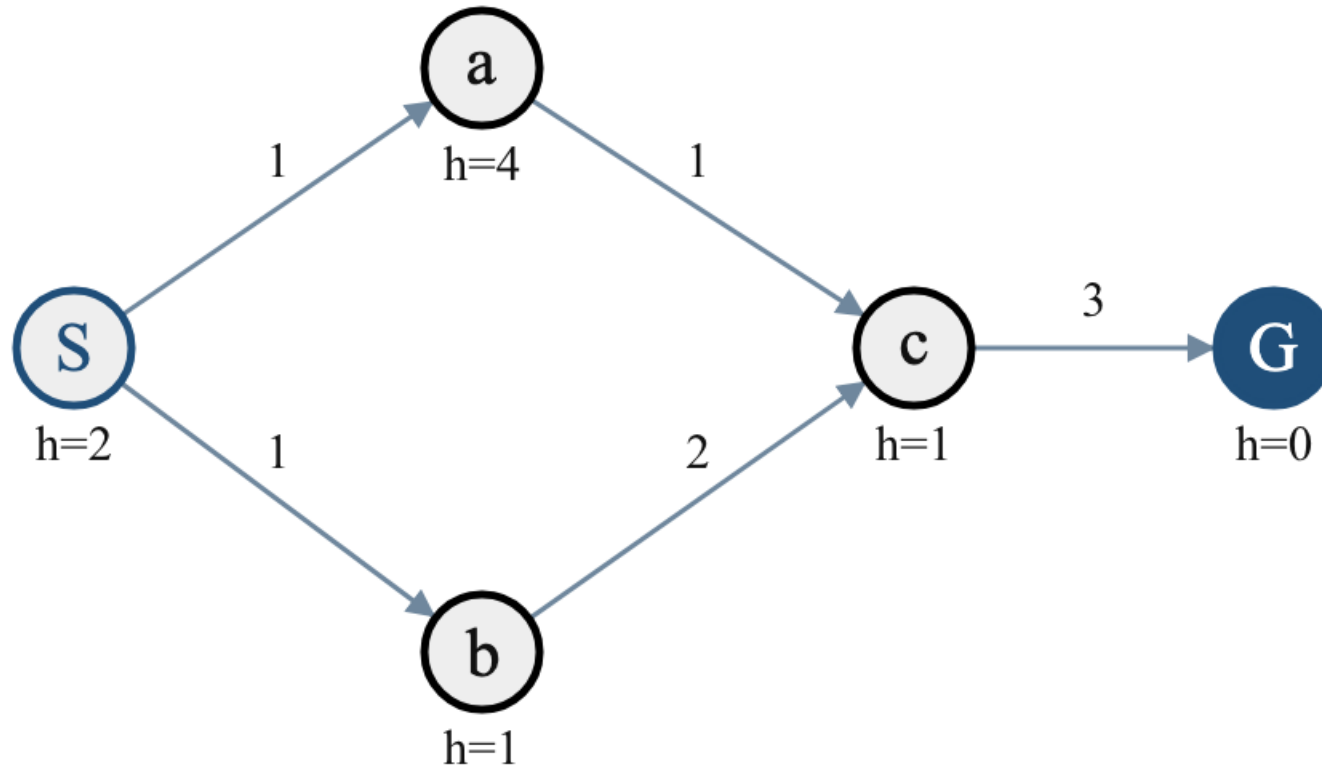
$$h(u) \leq \text{cost}(u, v) + h(v)$$

$$S \rightarrow a: 2 \leq 1 + 4$$

$$S \rightarrow b: 2 \leq 1 + 1$$

$$a \rightarrow c: 4 > 1 + 1$$

Consistency of Heuristic



Is h consistent for this graph?

$$h(u) \leq \text{cost}(u, v) + h(v)$$

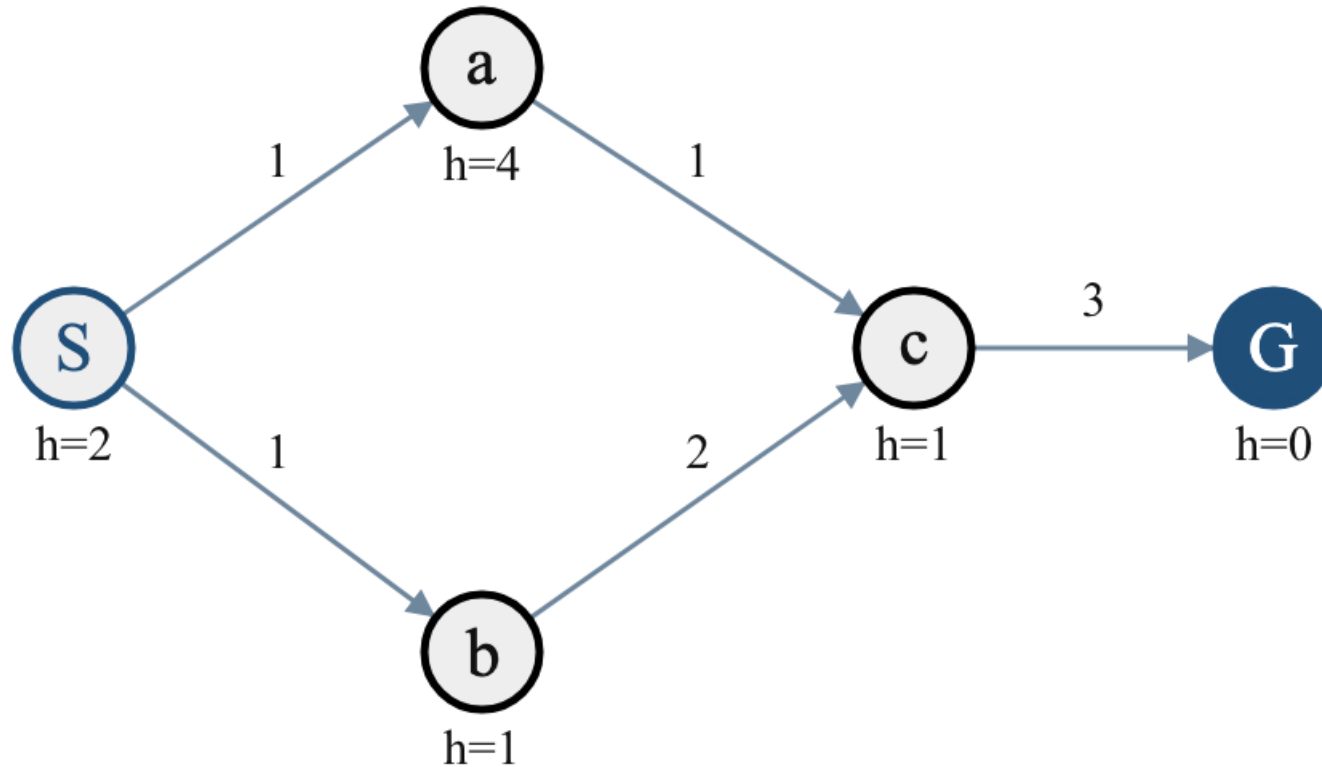
$$S \rightarrow a: 2 \leq 1 + 4$$

$$S \rightarrow b: 2 \leq 1 + 1$$

$$a \rightarrow c: 4 > 1 + 1$$

$$b \rightarrow c: 1 \leq 2 + 1$$

Consistency of Heuristic



Is h consistent for this graph?

$$h(u) \leq \text{cost}(u, v) + h(v)$$

$$S \rightarrow a: 2 \leq 1 + 4$$

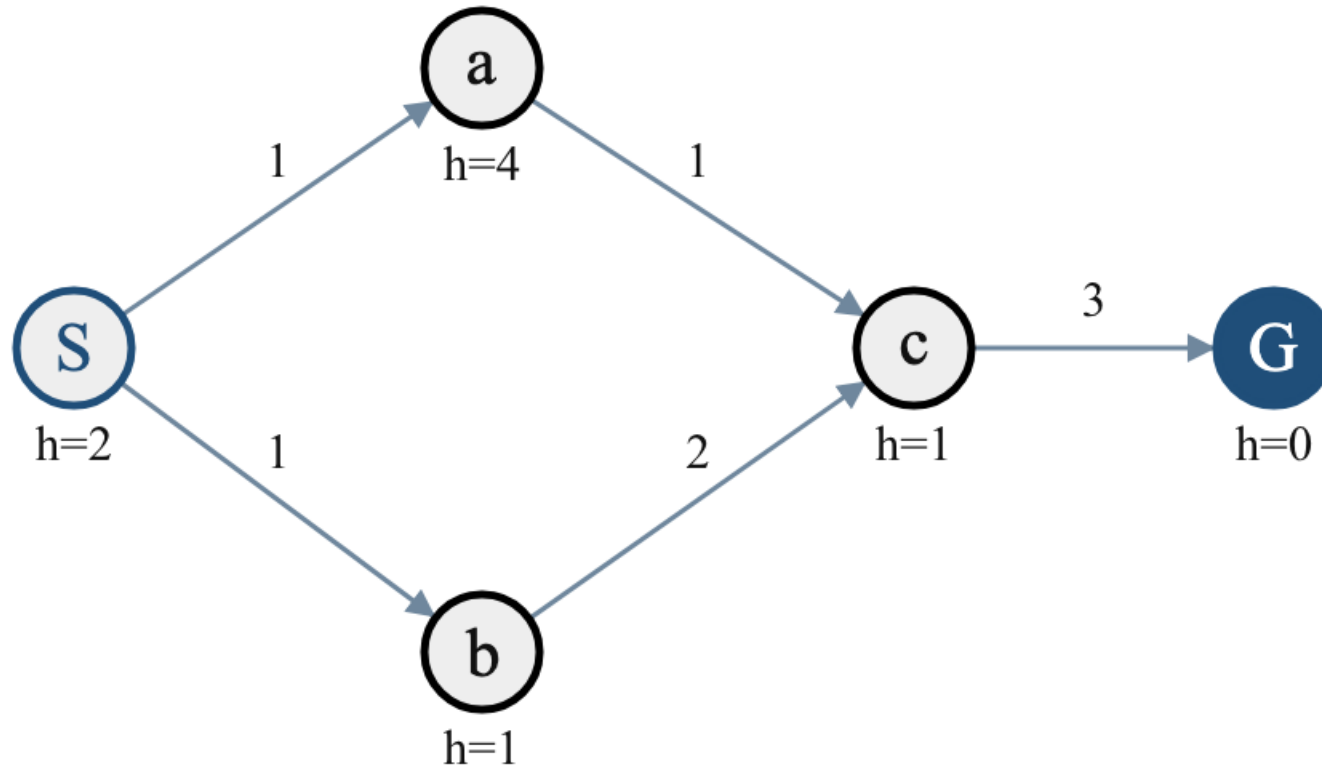
$$S \rightarrow b: 2 \leq 1 + 1$$

$$a \rightarrow c: 4 > 1 + 1$$

$$b \rightarrow c: 1 \leq 2 + 1$$

$$c \rightarrow G: 1 \leq 3 + 0$$

Consistency of Heuristic



Is h consistent for this graph?

$$h(u) \leq \text{cost}(u, v) + h(v)$$

$$S \rightarrow a: 2 \leq 1 + 4$$

$$S \rightarrow b: 2 \leq 1 + 1$$

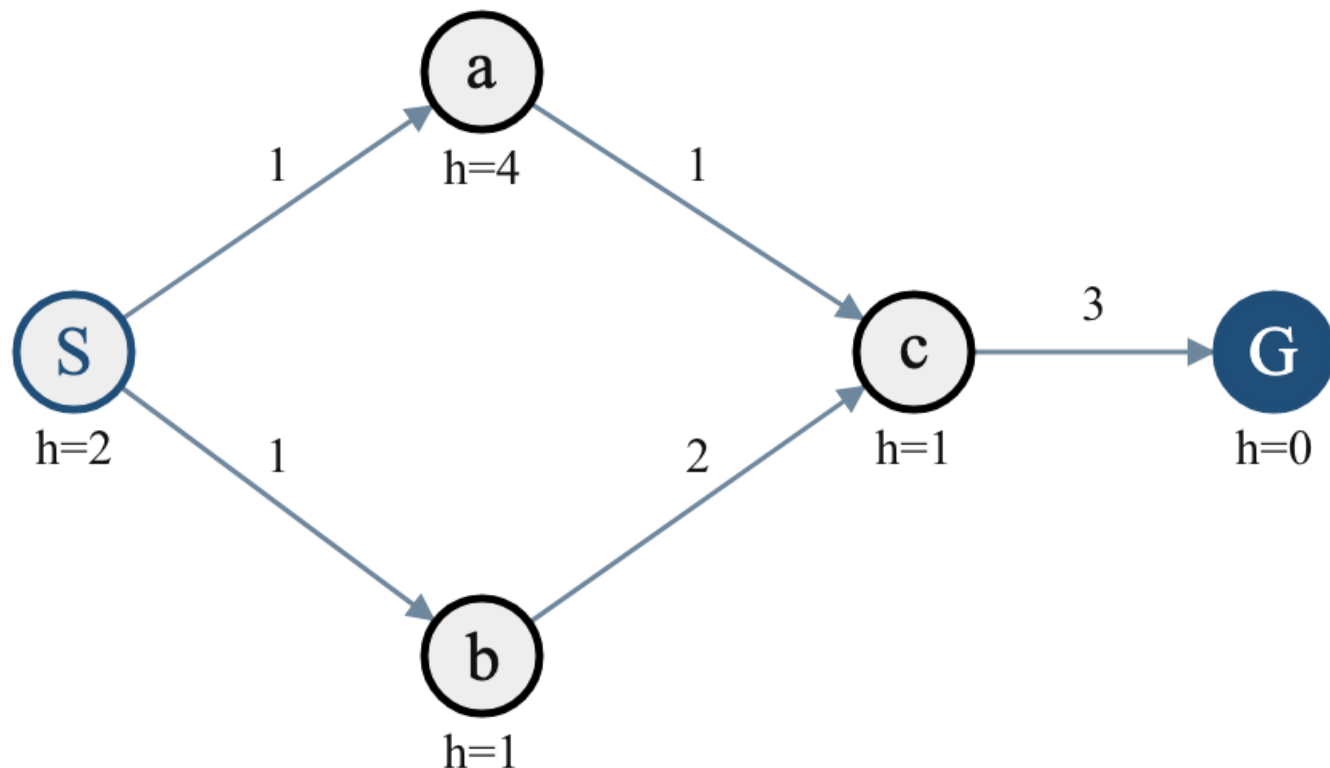
$$a \rightarrow c: 4 > 1 + 1$$

$$b \rightarrow c: 1 \leq 2 + 1$$

$$c \rightarrow G: 1 \leq 3 + 0$$

No: $a \rightarrow c$ violates consistency

Consistency of Heuristic



Is h consistent for this graph?

$$h(u) \leq \text{cost}(u, v) + h(v)$$

$$S \rightarrow a: 2 \leq 1 + 4$$

$$S \rightarrow b: 2 \leq 1 + 1$$

$$a \rightarrow c: 4 > 1 + 1$$

$$b \rightarrow c: 1 \leq 2 + 1$$

$$c \rightarrow G: 1 \leq 3 + 0$$

No: $a \rightarrow c$ violates consistency

Keep all edge costs fixed

How could we repair h ?

A*: Key Takeaways

A*-TSA

Returns a cheapest solution when h is admissible

A*: Key Takeaways

A*-TSA

Returns a cheapest solution when h is admissible

Our A*-GSA

Returns a cheapest solution when h is consistent

A*: Key Takeaways

A*-TSA

Returns a cheapest solution when h is admissible

Our A*-GSA

Returns a cheapest solution when h is consistent

Will it find a solution?

Yes, if one exists, with finite branching
and every step cost $\geq \epsilon > 0$

A*: Key Takeaways

A*-TSA

Returns a cheapest solution when h is admissible

Our A*-GSA

Returns a cheapest solution when h is consistent

Will it find a solution?

Yes, if one exists, with finite branching
and every step cost $\geq \epsilon > 0$

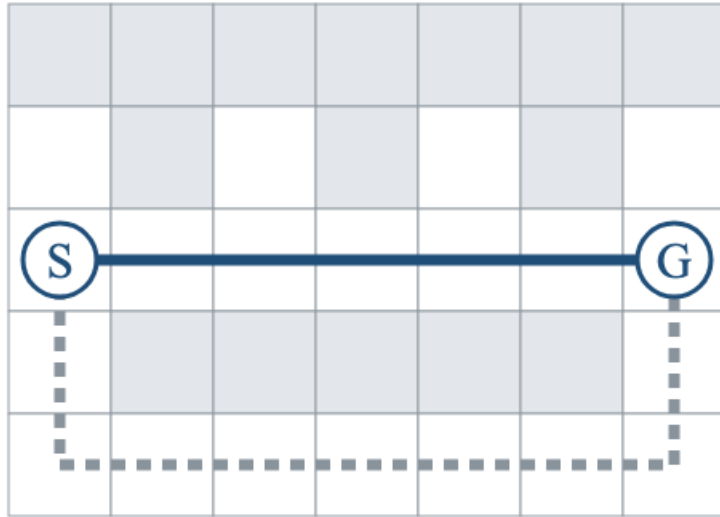
Time and memory

Can still grow exponentially

Local Search

Planning vs. Identification

Planning



The path matters

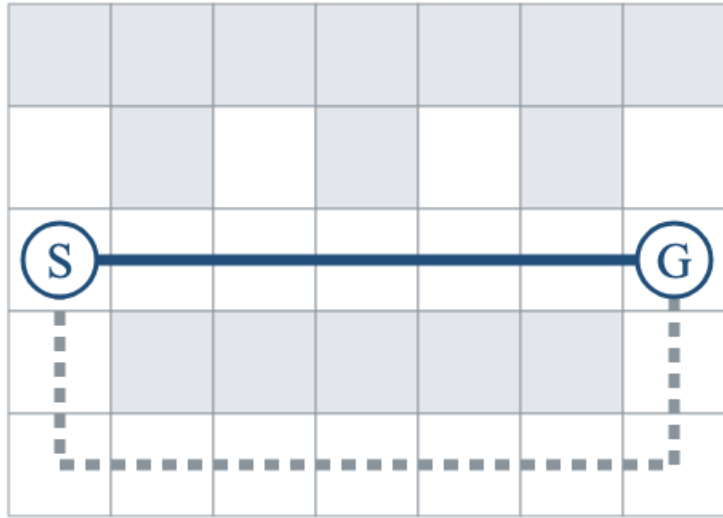
Find a sequence of actions

Compare path costs and depths

Use heuristics and a frontier

Planning vs. Identification

Planning



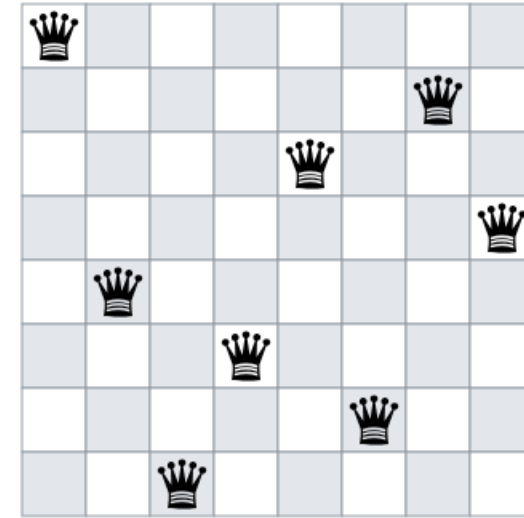
The path matters

Find a sequence of actions

Compare path costs and depths

Use heuristics and a frontier

Identification



The final arrangement matters

Find an assignment to variables

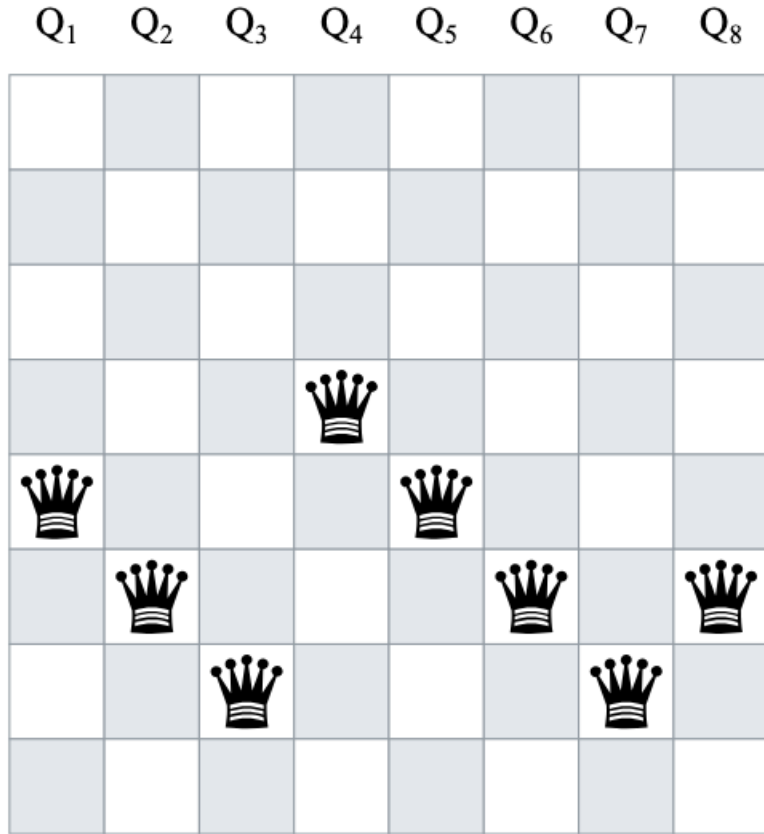
Example: assign a position to each queen

Local search can be faster for some problems

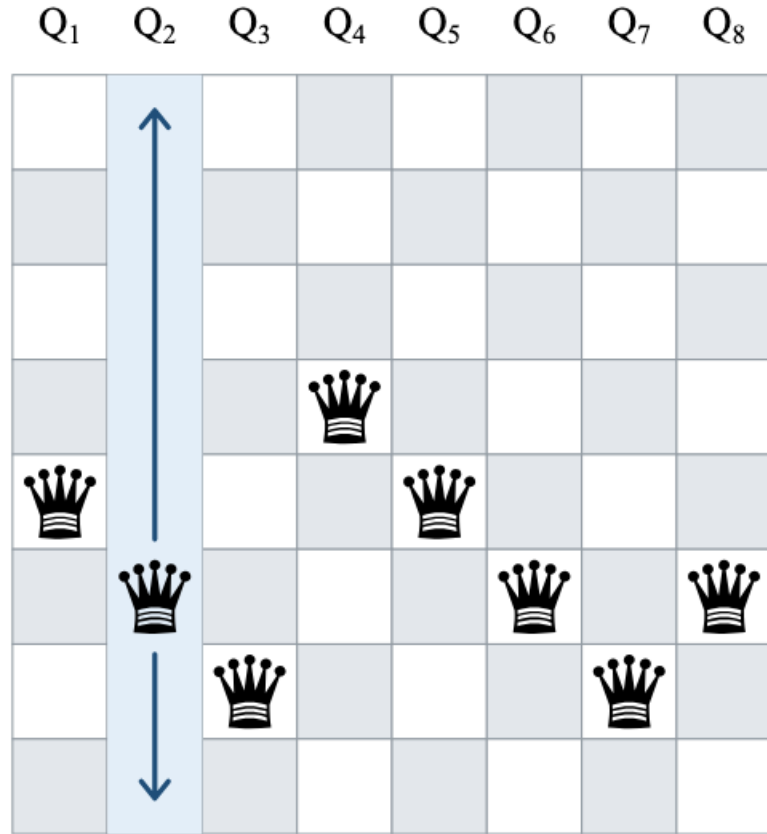
Example: 8-Queens

State

Eight queens on the board, one per column



Example: 8-Queens



State

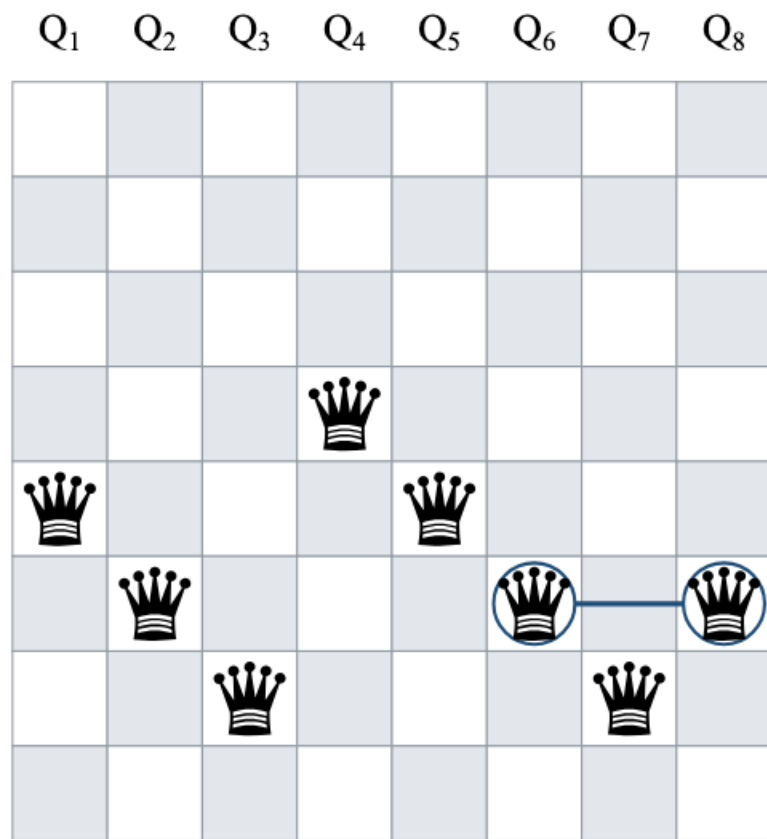
Eight queens on the board, one per column

Successors

Move one queen to another square
in the same column

Every such move gives a neighbour

Example: 8-Queens



Lower cost is better

Goal: zero attacking pairs

State

Eight queens on the board, one per column

Successors

Move one queen to another square
in the same column

Every such move gives a neighbour

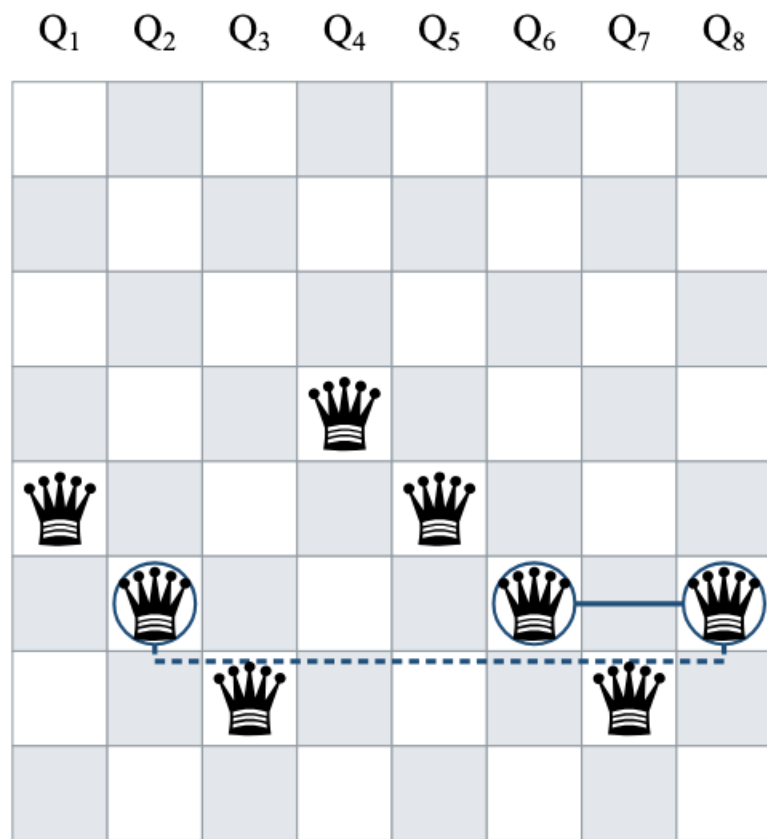
Cost function

Count pairs of queens that attack each other

Same row or diagonal

Direct: Q_6 and Q_8

Example: 8-Queens



Lower cost is better

Goal: zero attacking pairs

State

Eight queens on the board, one per column

Successors

Move one queen to another square in the same column

Every such move gives a neighbour

Cost function

Count pairs of queens that attack each other

Same row or diagonal

Direct: Q_6 and Q_8

Indirect: Q_2 and Q_8 , with Q_6 between

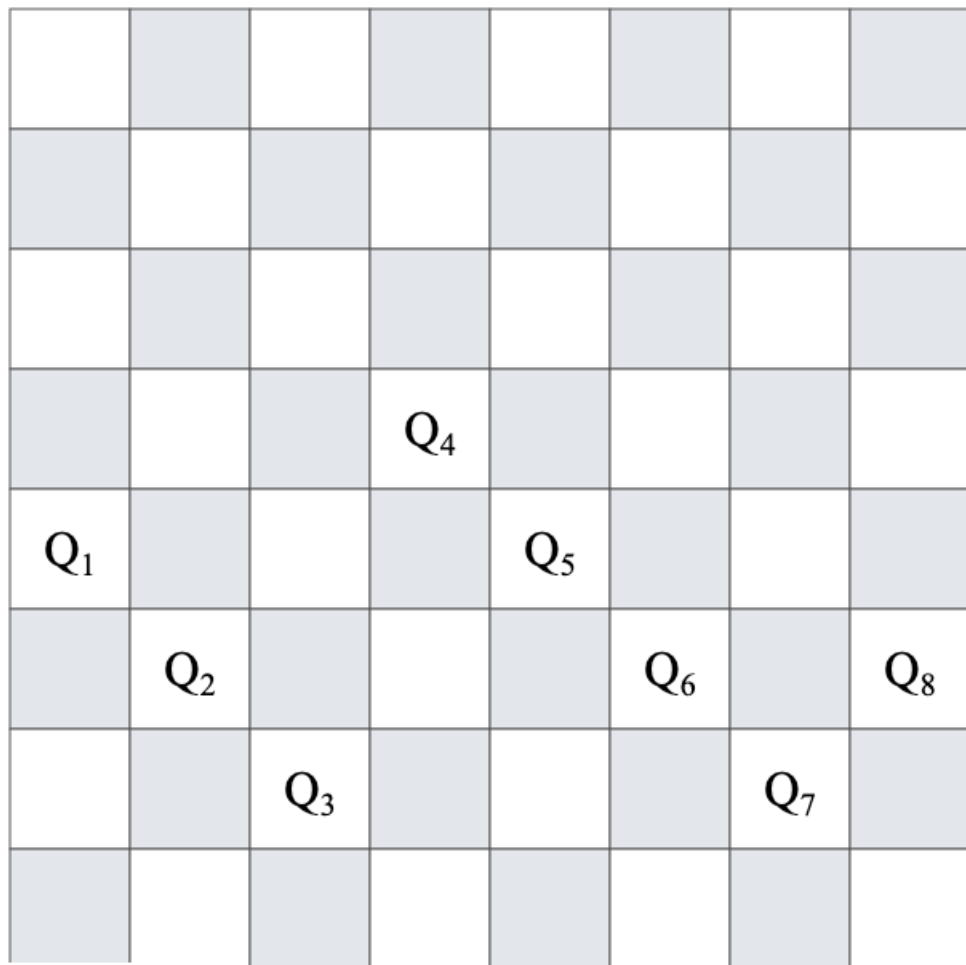
Count Attacking Pairs

What is the cost of this state?

			Q_4				
Q_1				Q_5			
	Q_2				Q_6		Q_8
		Q_3				Q_7	

Count the Neighbours

How many different one-queen moves are possible?



Evaluate the Neighbours

What would the cost be after each move?

			Q ₄				
Q ₁				Q ₅			
	Q ₂				Q ₆		Q ₈
		Q ₃				Q ₇	

Each number is the cost of the whole board
after moving that column's queen

Evaluate the Neighbours

What would the cost be after each move?

	12						
			Q ₄				
Q ₁				Q ₅			
	Q ₂				Q ₆		Q ₈
		Q ₃				Q ₇	

Each number is the cost of the whole board
after moving that column's queen

Example: move Q₂ to the top row

The resulting board costs 12

Evaluate the Neighbours

What would the cost be after each move?

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

Each number is the cost of the whole board after moving that column's queen

Example: move Q₂ to the top row
The resulting board costs 12

The current board costs 17

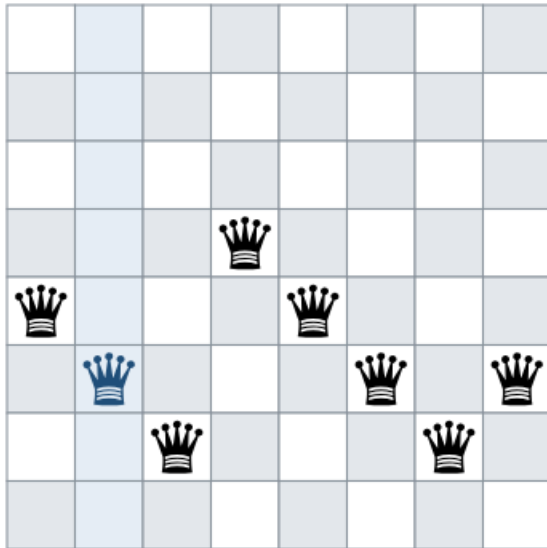
Lowest of all 56 costs: 12

Several moves tie; choose one at random

Improve the Current State

Cost = attacking pairs · Lower is better

Current state



Cost: 17

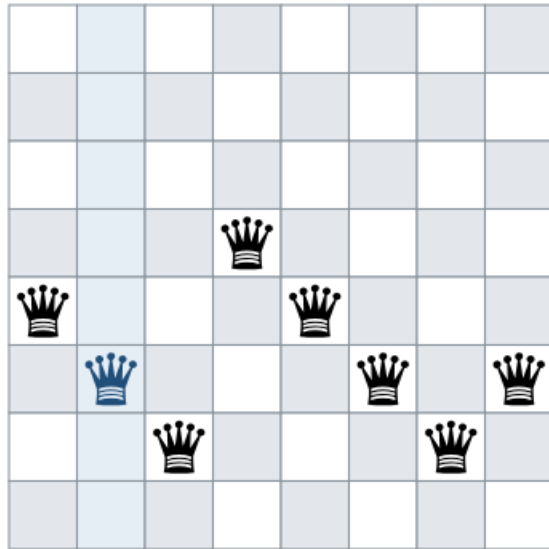
Improve the Current State

Cost = attacking pairs · Lower is better

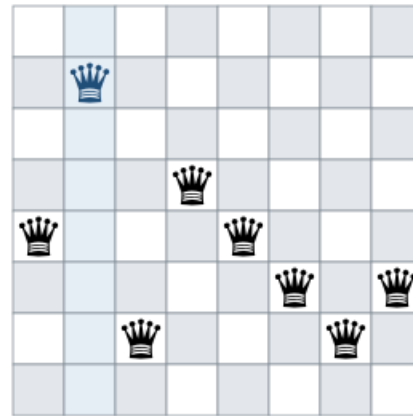
Current state

Compare neighbouring states

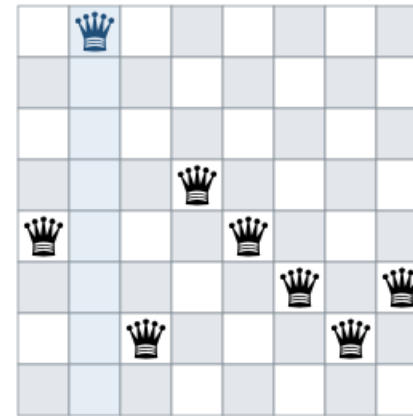
Three of the 56 neighbours; each moves the highlighted queen



Cost: 17



Cost: 16



Cost: 12



Cost: 14

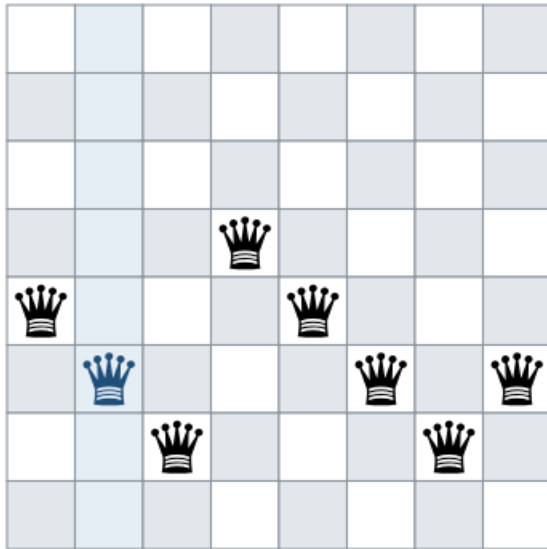
Improve the Current State

Cost = attacking pairs · Lower is better

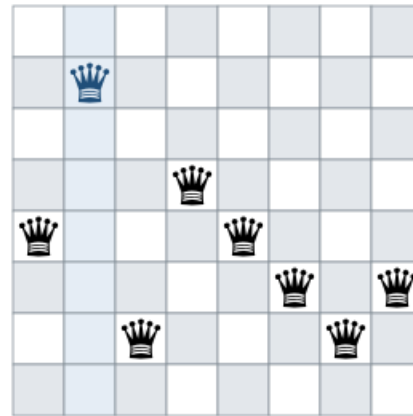
Current state

Compare neighbouring states

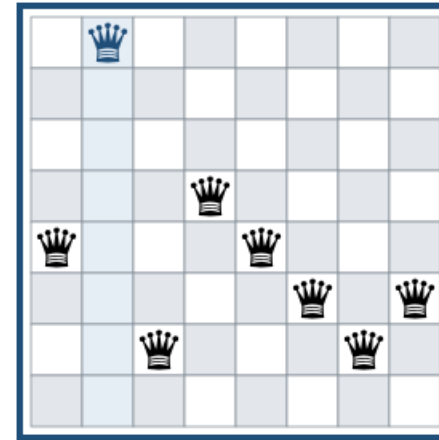
Three of the 56 neighbours; each moves the highlighted queen



Cost: 17



Cost: 16



Cost: 12

Choose this state



Cost: 14

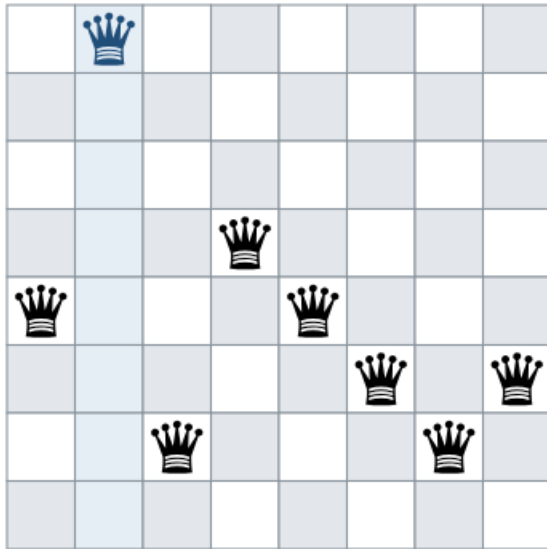
Improve the Current State

Cost = attacking pairs · Lower is better

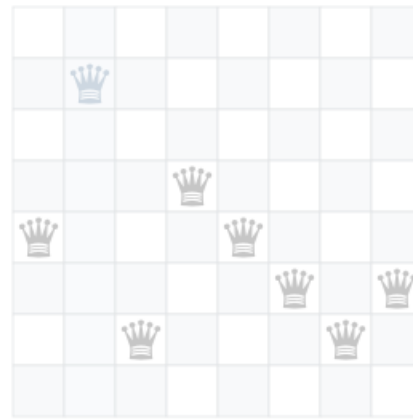
Current state

Compare neighbouring states

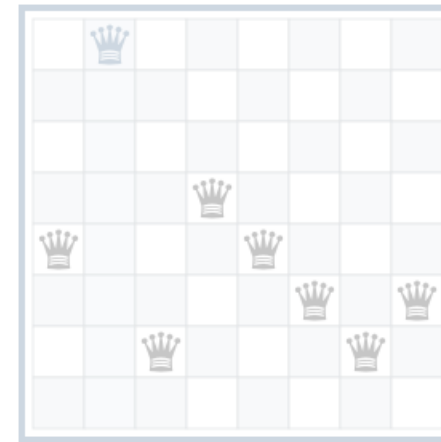
Three of the 56 neighbours; each moves the highlighted queen



Cost: 12

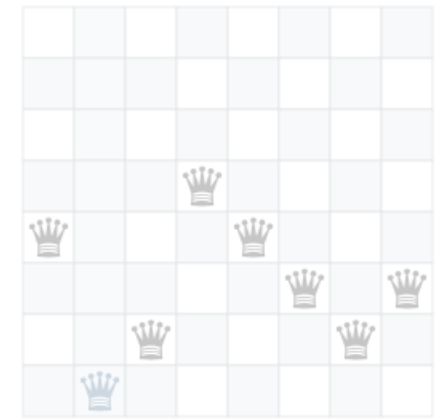


Cost: 16



Cost: 12

Choose this state



Cost: 14

Replace the current state, then repeat

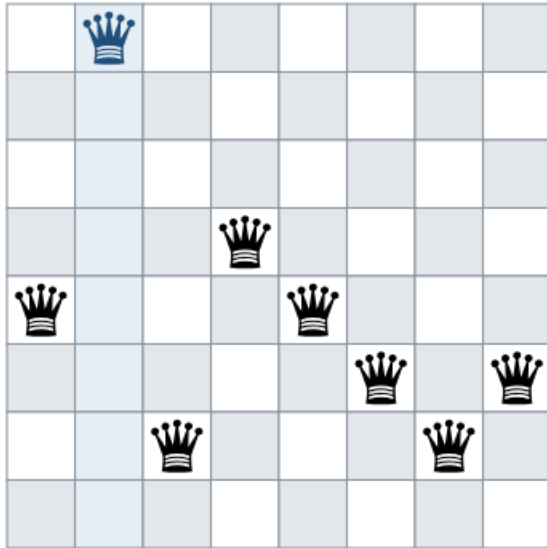
Improve the Current State

Cost = attacking pairs · Lower is better

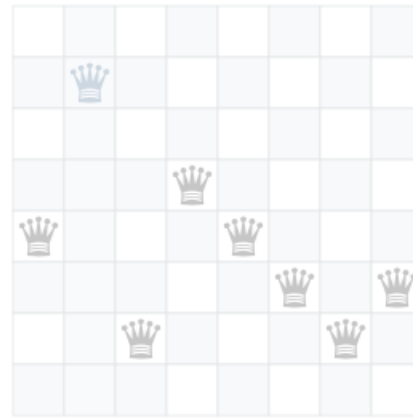
Current state

Compare neighbouring states

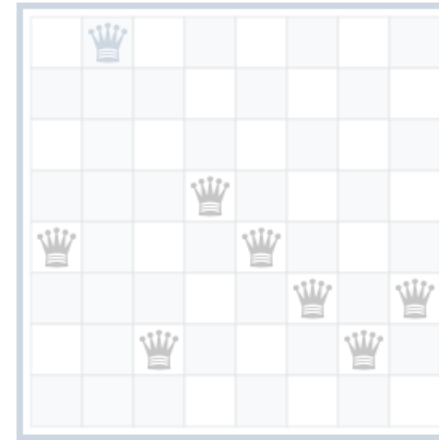
Three of the 56 neighbours; each moves the highlighted queen



Cost: 12

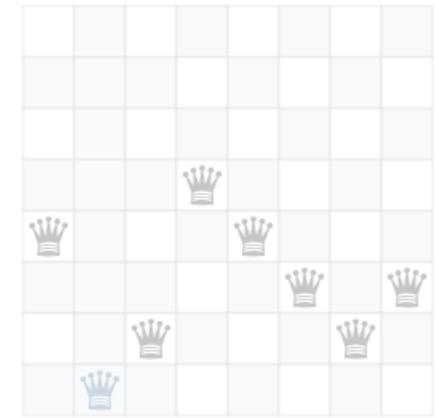


Cost: 16



Cost: 12

Choose this state



Cost: 14

Replace the current state, then repeat

Very little memory · Often finds good solutions in large spaces

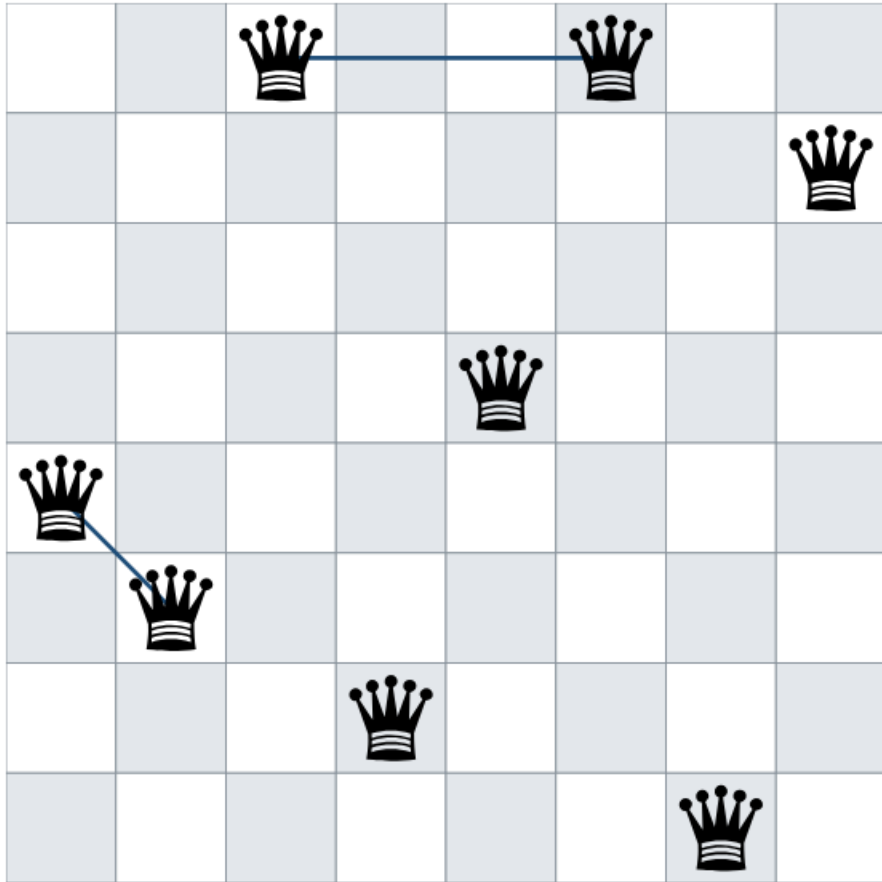
Hill Climbing: One Attempt

Steepest descent for a cost we want to minimise

1. Start with a random state
2. If cost = 0, return the solution
3. Find the lowest-cost neighbour
Break ties at random
4. If its cost $\geq$ current cost, stop this attempt
5. Otherwise replace the current state; return to step 2

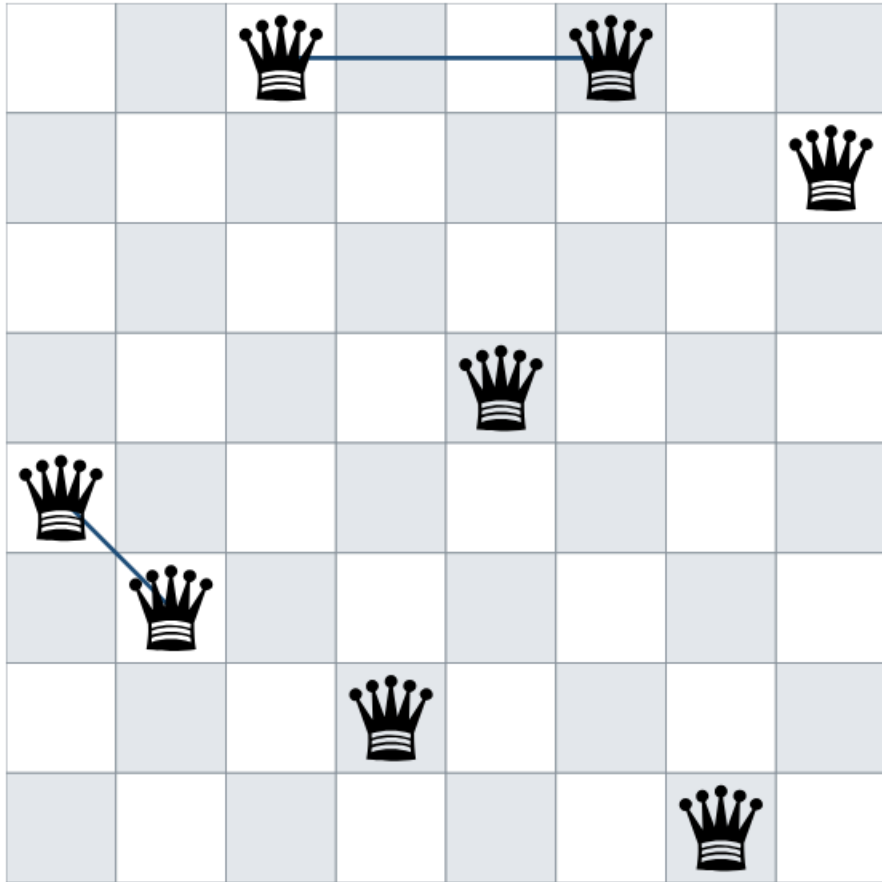
Accept only strictly lower costs

When Improvement Stops



Current cost: 2

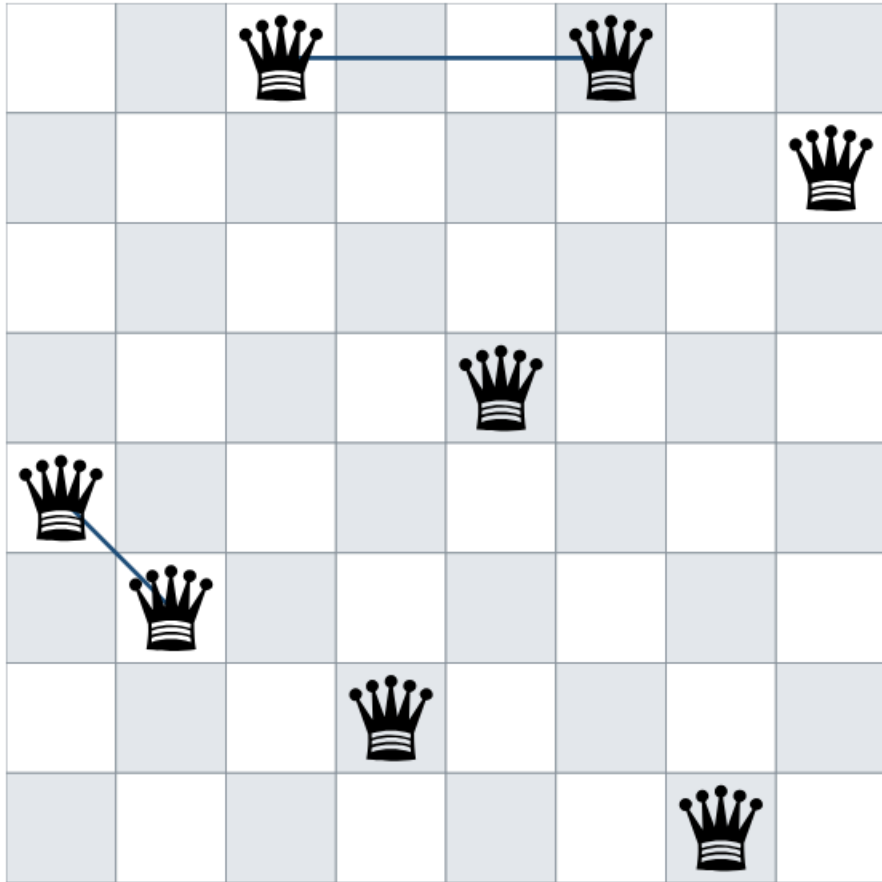
When Improvement Stops



Current cost: 2

Lowest neighbour cost: 2

When Improvement Stops



Current cost: 2

Lowest neighbour cost: 2

No strictly better move

Stop, but not solved

An equal-cost move would be a sideways move

```
1 from queens import experiment, format_mean as fmt
2 trials, seed = 1000, 1 # Change these and run again
3 result = experiment(trials, seed)
4 print(f"{trials} independent attempts; seed {seed}")
5 print(f"Solved: {result['success_rate']:.1%}")
6 print("Mean moves if solved:", fmt(result["mean_success_moves"]))
7 print("Mean moves if stopped:", fmt(result["mean_stopped_moves"]))
8
```

Does the Experiment Match?

Five seeds $\times$ 10,000 independent attempts

Solved

14.07%

Mean moves if solved

4.08

Mean moves if stopped

3.06

7,035 solutions in 50,000 attempts

Success across the five seeds: 13.23%–14.42%

One strict attempt, without sideways moves · Accepted moves only

Does the Experiment Match?

Five seeds $\times$ 10,000 independent attempts

Solved	Mean moves if solved	Mean moves if stopped
14.07%	4.08	3.06

7,035 solutions in 50,000 attempts

Success across the five seeds: 13.23%–14.42%

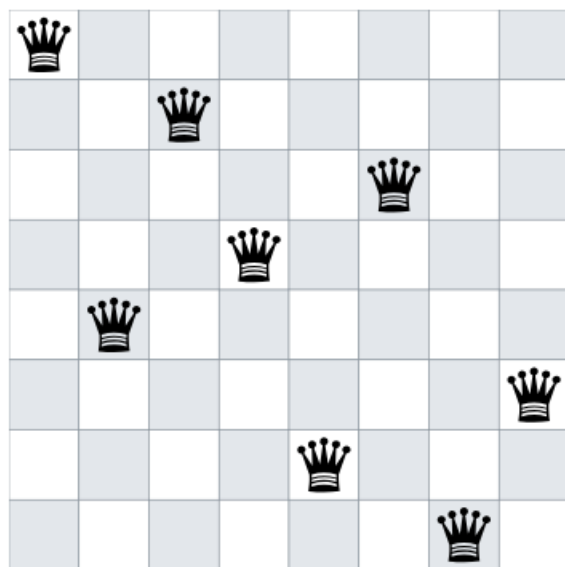
Close to AIMA's approximate 14% success and 4 / 3 moves

One strict attempt, without sideways moves · Accepted moves only

Try a New Starting State

Random restart: begin each attempt from a fresh random board

Attempt 1



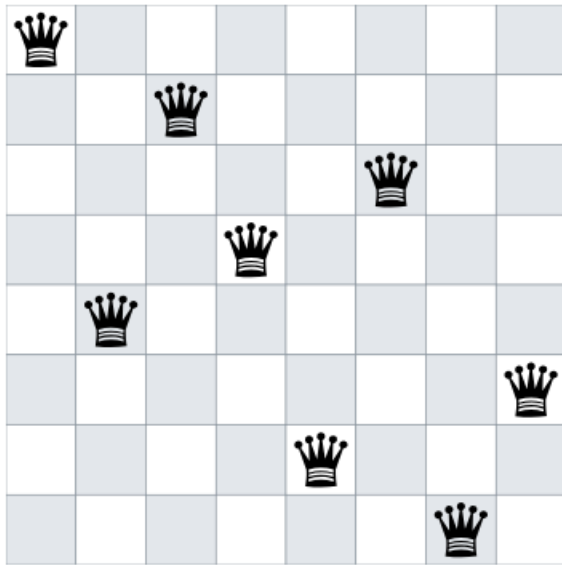
Stopped: cost 1

Repeat until solved, or a chosen attempt limit is reached

Try a New Starting State

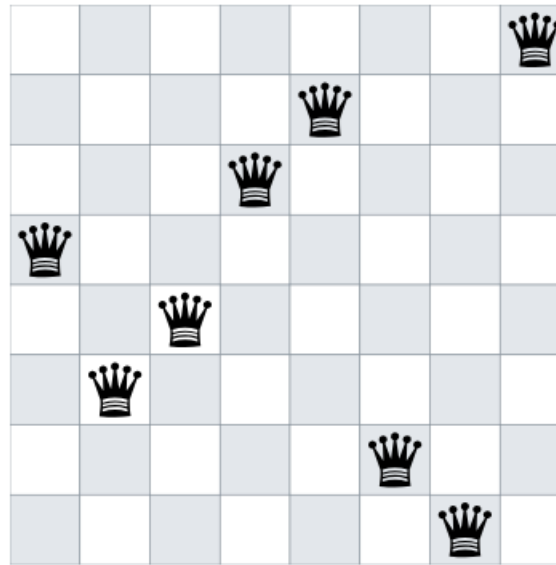
Random restart: begin each attempt from a fresh random board

Attempt 1



Stopped: cost 1

Attempt 2



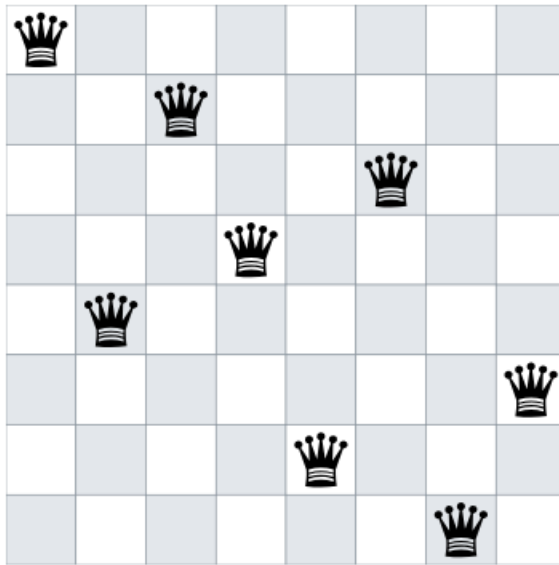
Stopped: cost 3

Repeat until solved, or a chosen attempt limit is reached

Try a New Starting State

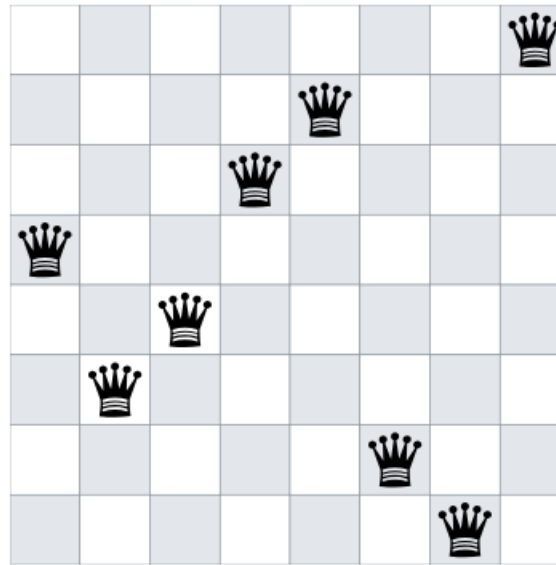
Random restart: begin each attempt from a fresh random board

Attempt 1



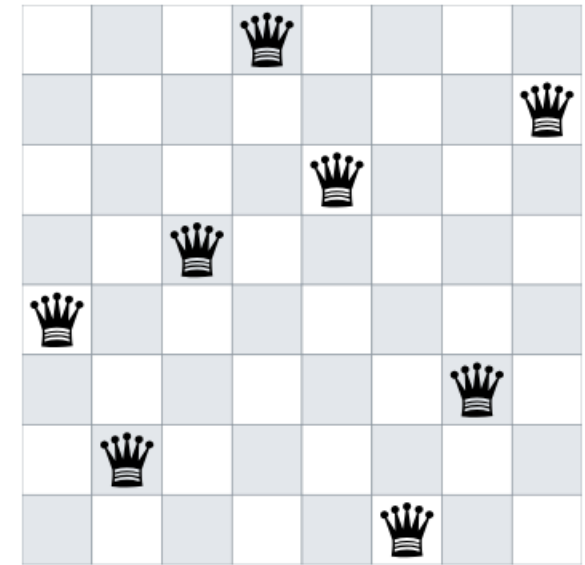
Stopped: cost 1

Attempt 2



Stopped: cost 3

Attempt 3



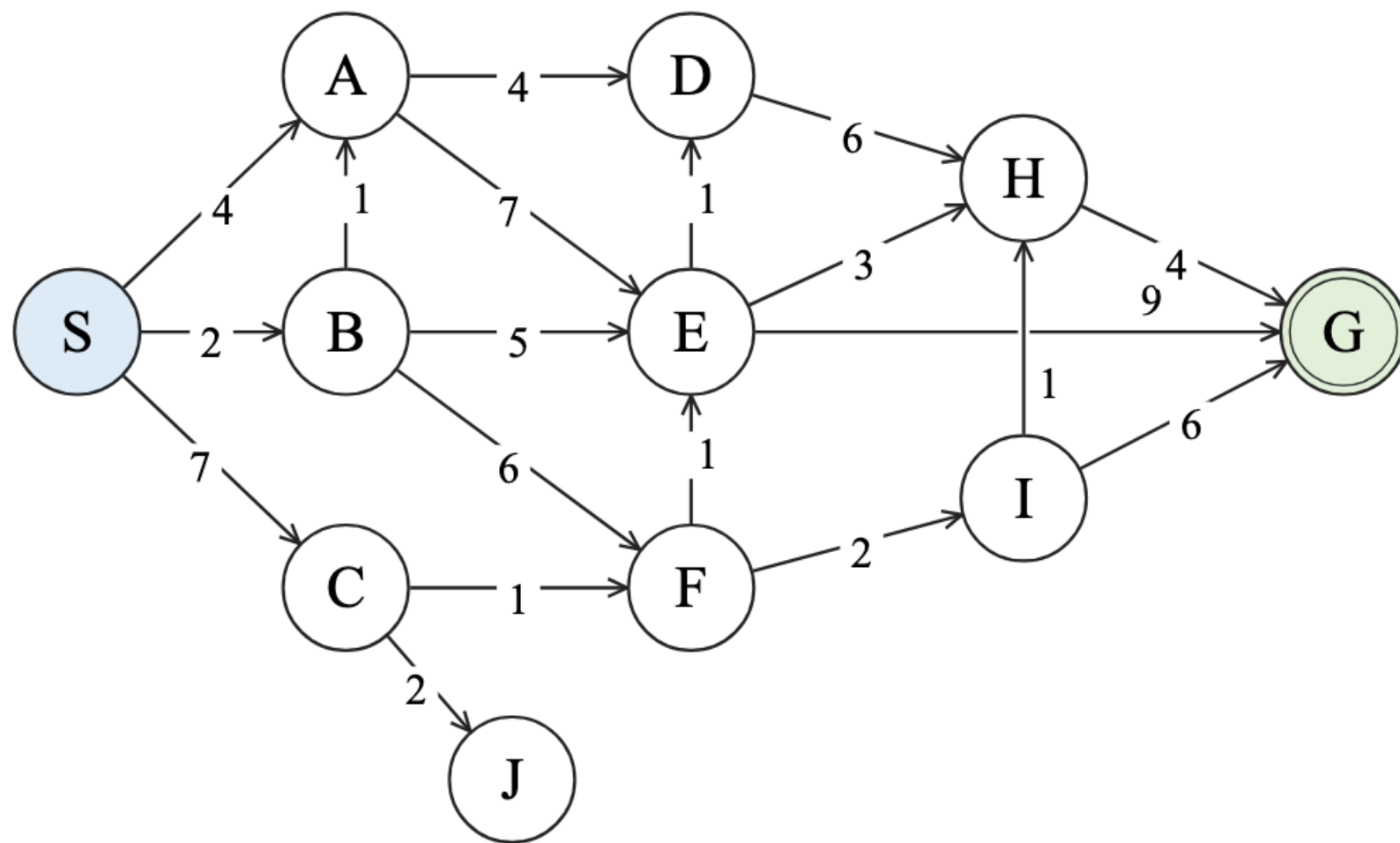
Solved: cost 0

Repeat until solved, or a chosen attempt limit is reached

Task Preparation

Log in to Moodle → Click Ed

Task: BFS-GSA



TASK

Start at **S**, goal **G**

Frontier: Add alphabetically

1. Exploration Order

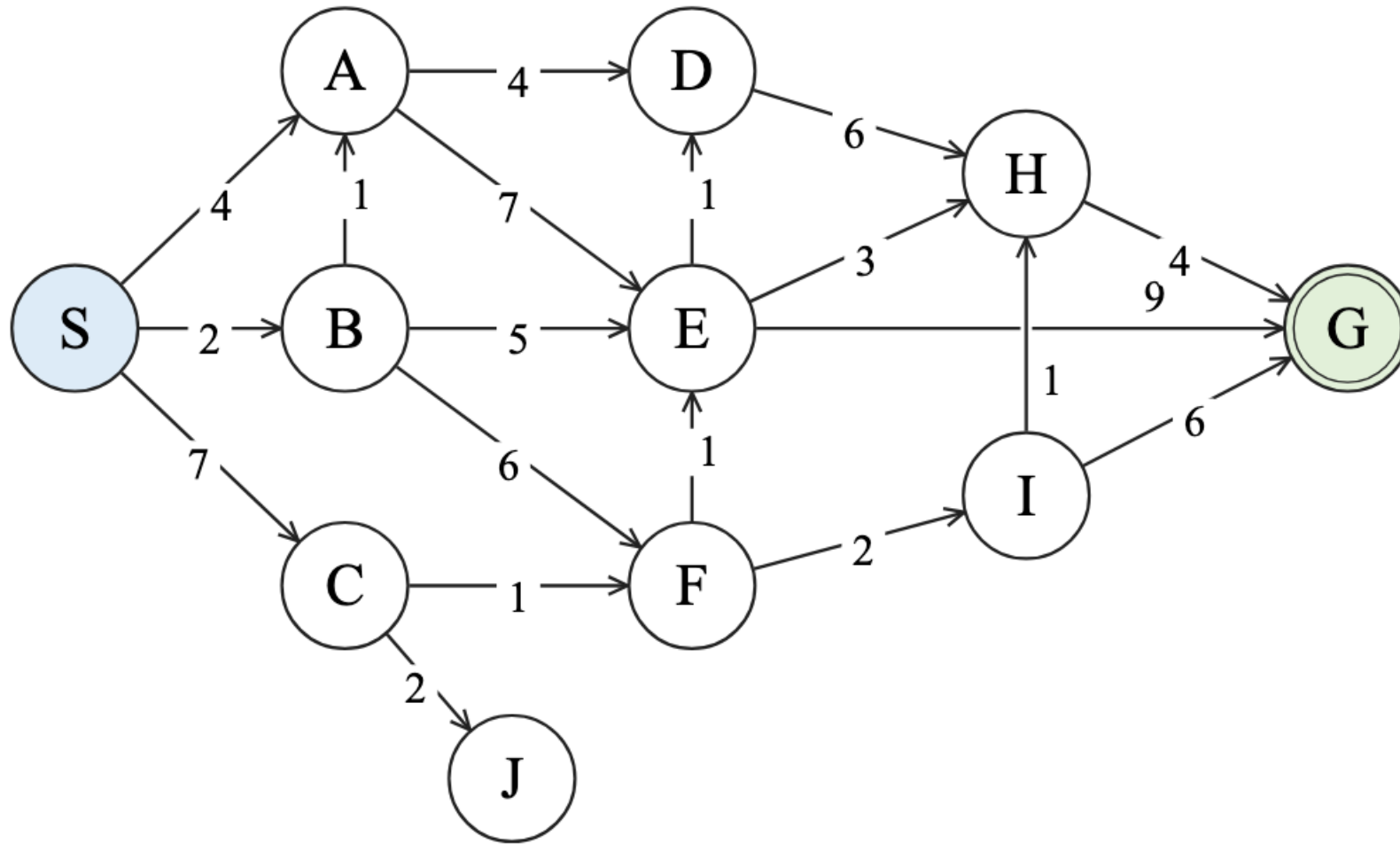
2. Solution path

Uppercase strings, no spaces

Example: SAG

[Submit on Ed](#)

Task: DFS-TSA



TASK

Start at **S**, goal **G**

Frontier: Add alphabetically

1. Exploration Order

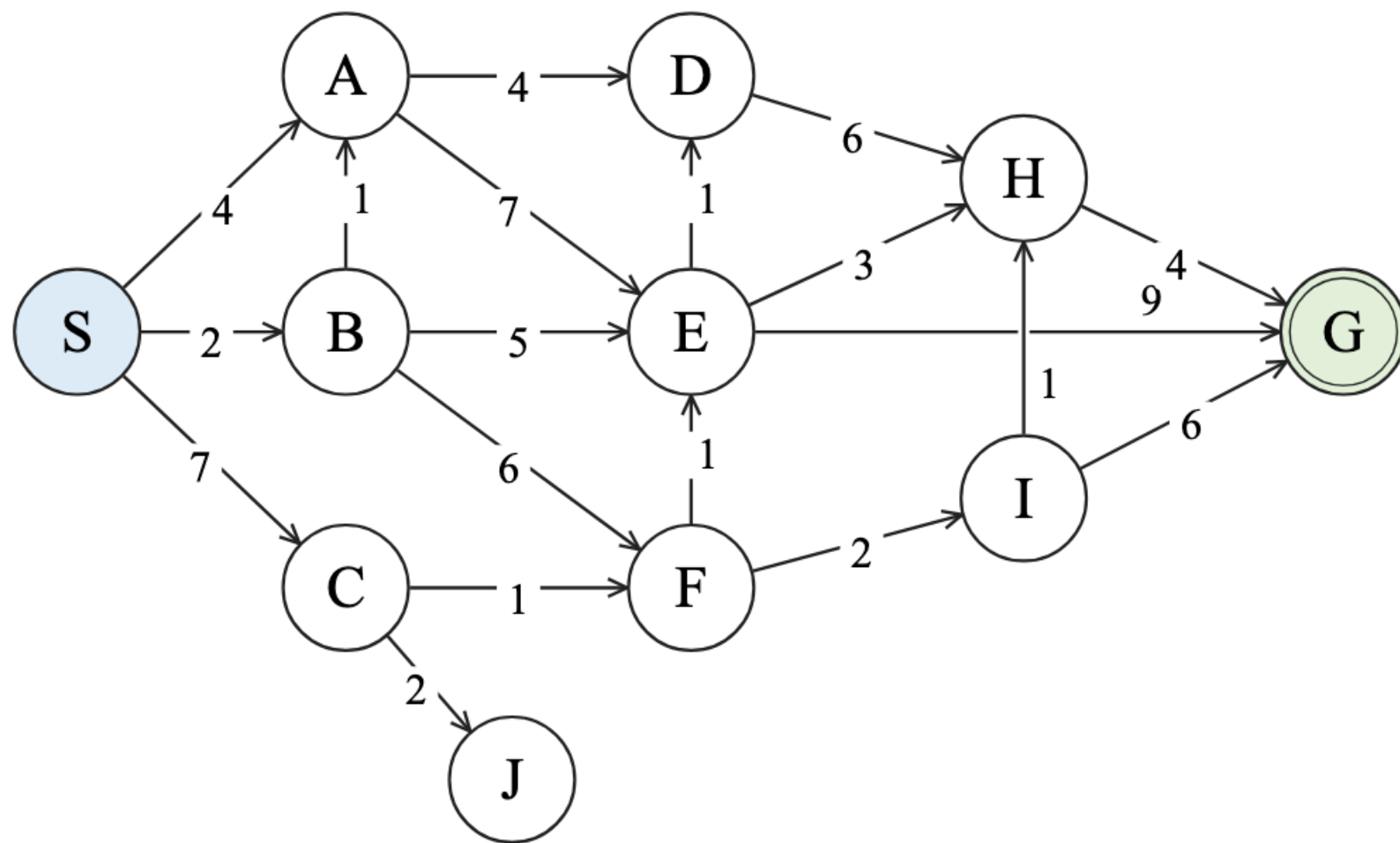
2. Solution path

Uppercase strings, no spaces

Example: SAG

[Submit on Ed](#)

Task: UCS-GSA



TASK

Start at **S**, goal **G**

Frontier: Add alphabetically

1. Exploration Order

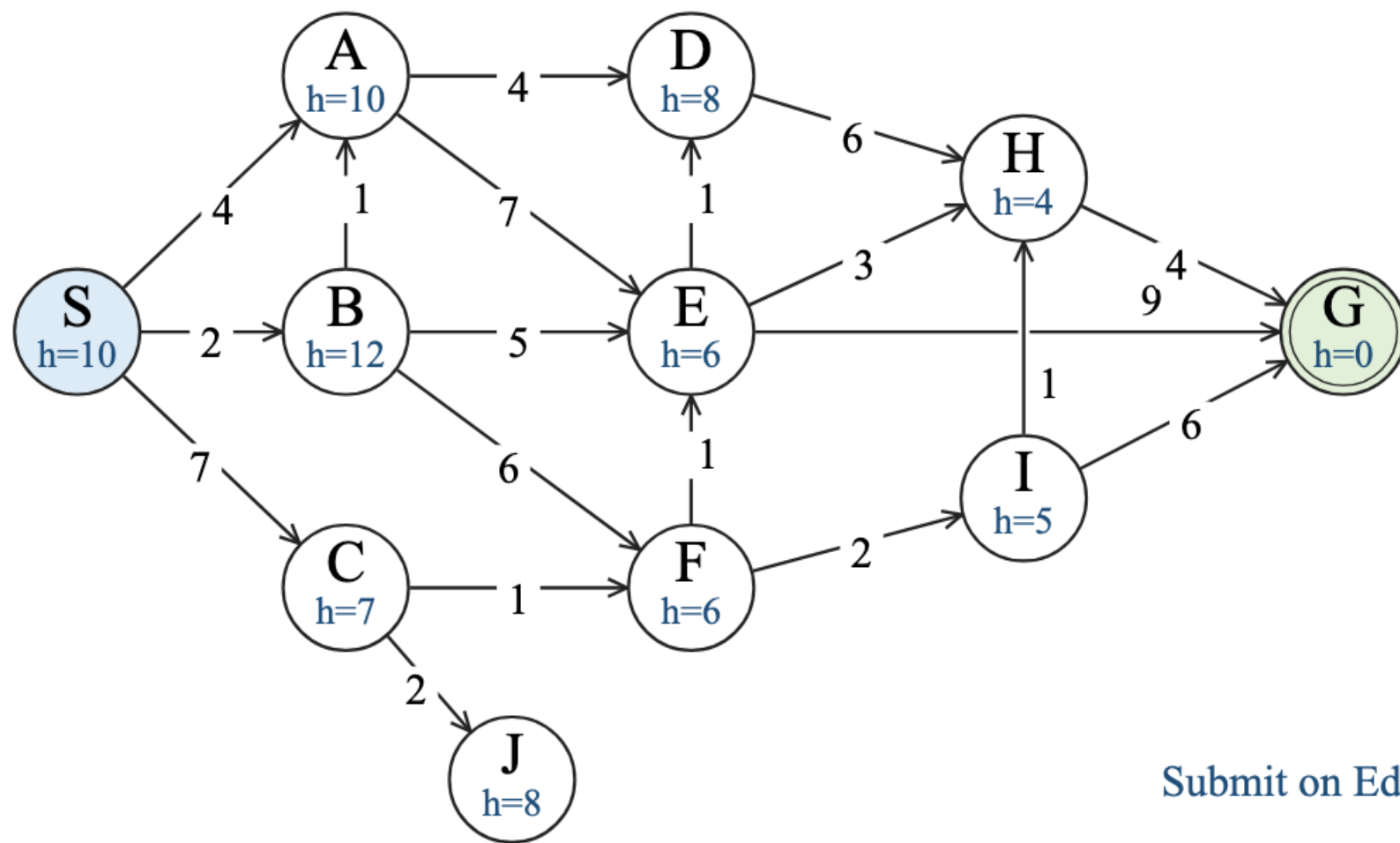
2. Solution path

Uppercase strings, no spaces

Example: SAG

[Submit on Ed](#)

Task: A*-GSA



TASK

Start at **S**, goal **G**

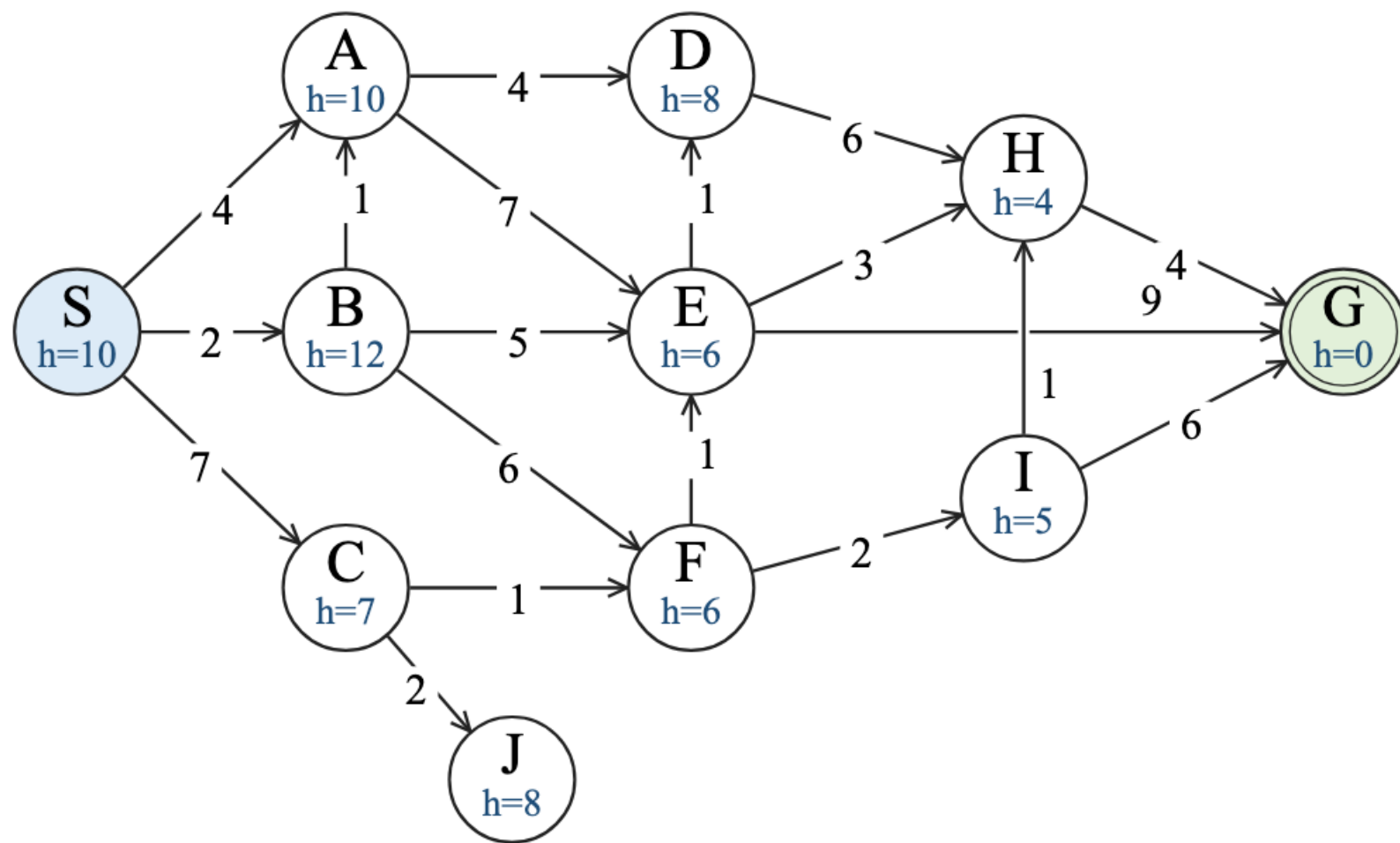
Frontier: Add alphabetically

1. Exploration Order
2. Solution path
3. Admissible?
4. Consistent?

Submit on Ed

Uppercase strings, no spaces
Example: SAG

Task: A*-GSA



TASK

Start at **S**, goal **G**

Frontier: Add alphabetically

1. Exploration Order
2. Solution path
3. Admissible?
4. Consistent?

Uppercase strings, no spaces
Example: SAG

Task: Local Search

TASK

	A	B	C	D	E
1	Q ₁	Q ₂			
2					
3				Q ₄	
4					
5			Q ₃		Q ₅

Minimise attacking pairs

No sideways moves

1. Sequence of moves

2. What is the cost after the last move?

Uppercase cells, separated by one space

Example: B4 E2

[Submit on Ed](#)

One queen per column · Move within its column

Constraint Satisfaction Problems

Constraint Satisfaction Problems

A **factored state** records a value for each variable

A solution assigns every variable a value that satisfies all constraints

Use constraints to rule out combinations before exploring them

Defining CSPs

Variables

$$X = \{X_1, \dots, X_n\}$$

Domains

D_i contains the allowed values of X_i

Constraints

Allowable combinations of values

A state is a partial or complete assignment

$$\{X_i = v_i, X_j = v_j, \dots\}$$

Solutions to CSPs

Consistent: no constraint is violated

Complete: every variable has a value

A **solution** is both consistent and complete

Australia: Map Colouring

Variables

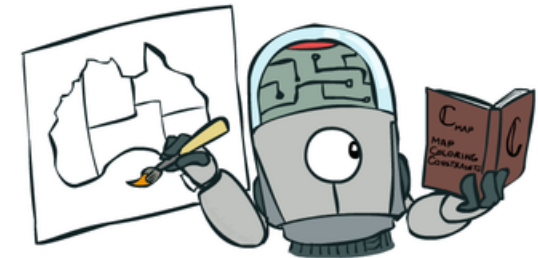
$$X = \{WA, NT, Q, NSW, V, SA, T\}$$

Domains

$$D_i = \{\text{red, green, blue}\}$$

Constraints

Neighbours have different colours



Australia: Map Colouring

Variables

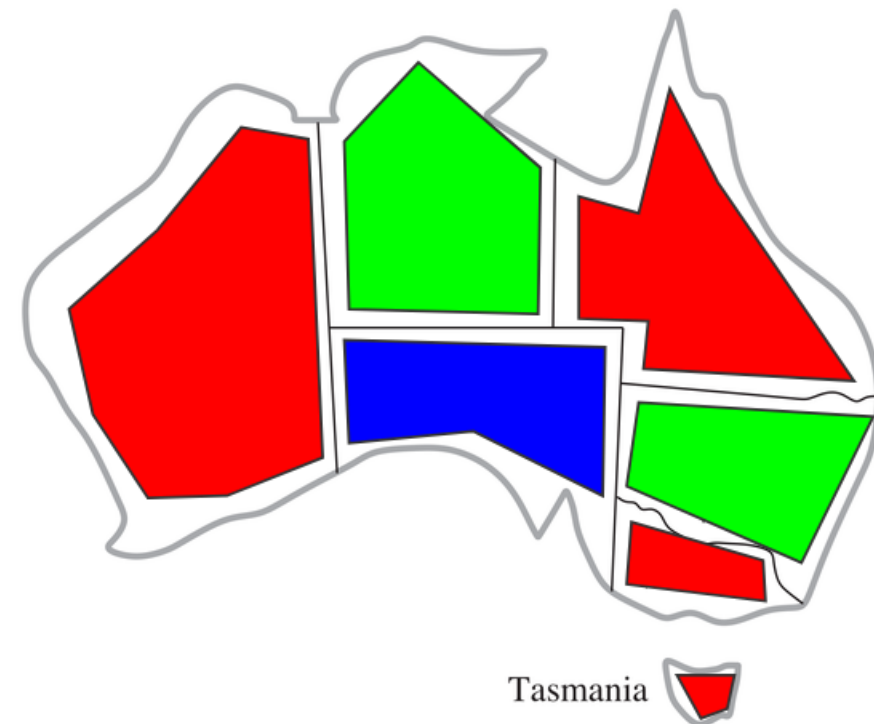
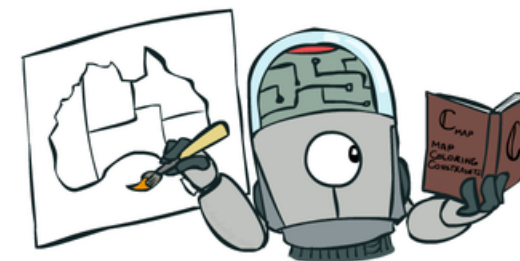
$$X = \{\text{WA, NT, Q, NSW, V, SA, T}\}$$

Domains

$$D_i = \{\text{red, green, blue}\}$$

Constraints

Neighbours have different colours



Constraint Graph

Nodes represent variables

Edges represent constraints

Tasmania is independent
of the mainland variables

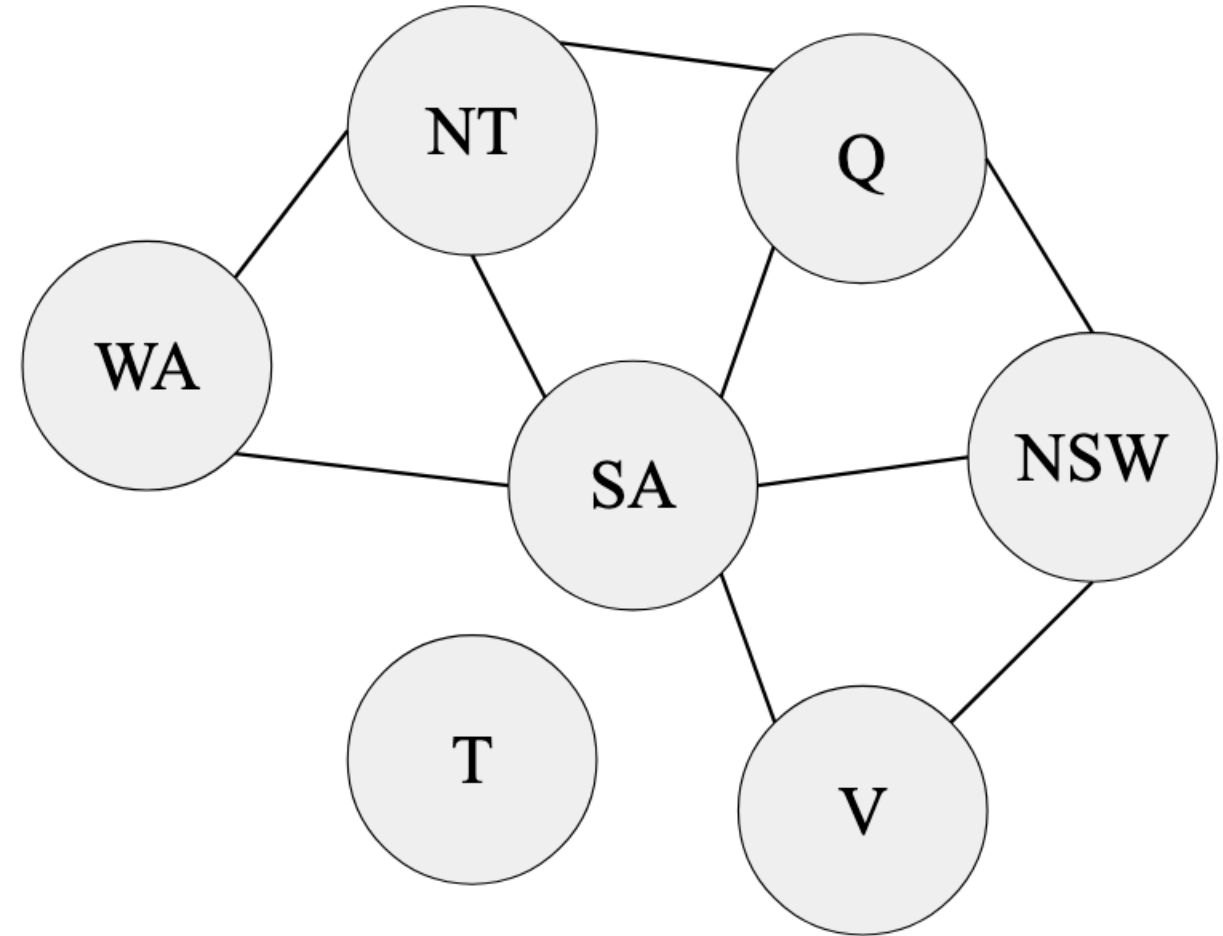


Constraint Graph

Nodes represent variables

Edges represent constraints

Tasmania is independent
of the mainland variables



Solving CSPs

State

The values assigned so far

Initial State

Empty assignment {}

Successor

Assign a value to an unassigned variable

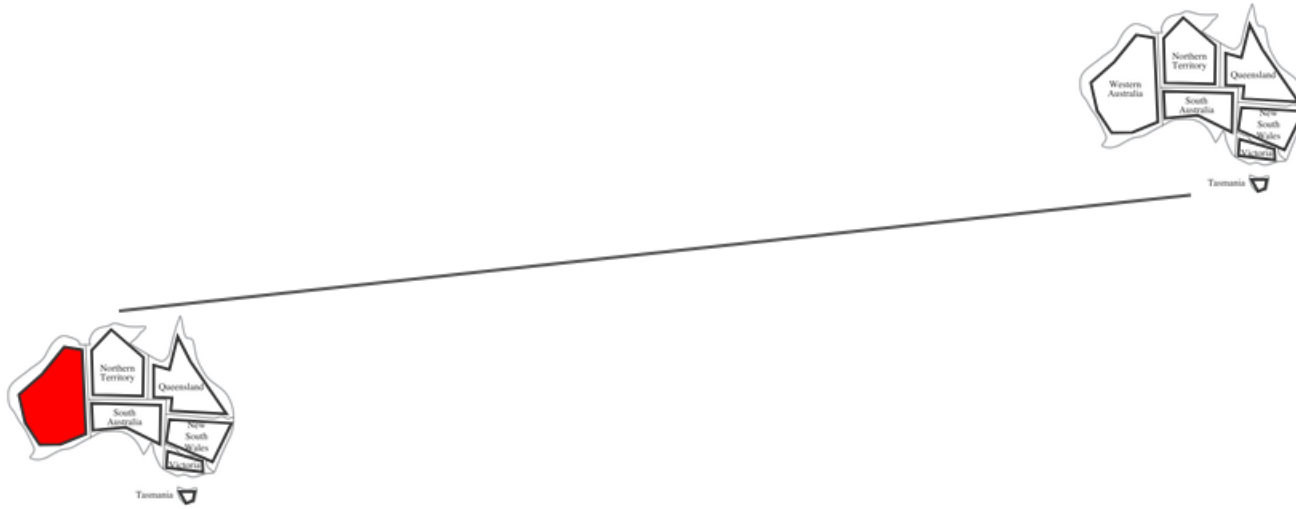
Goal Test

Complete and consistent

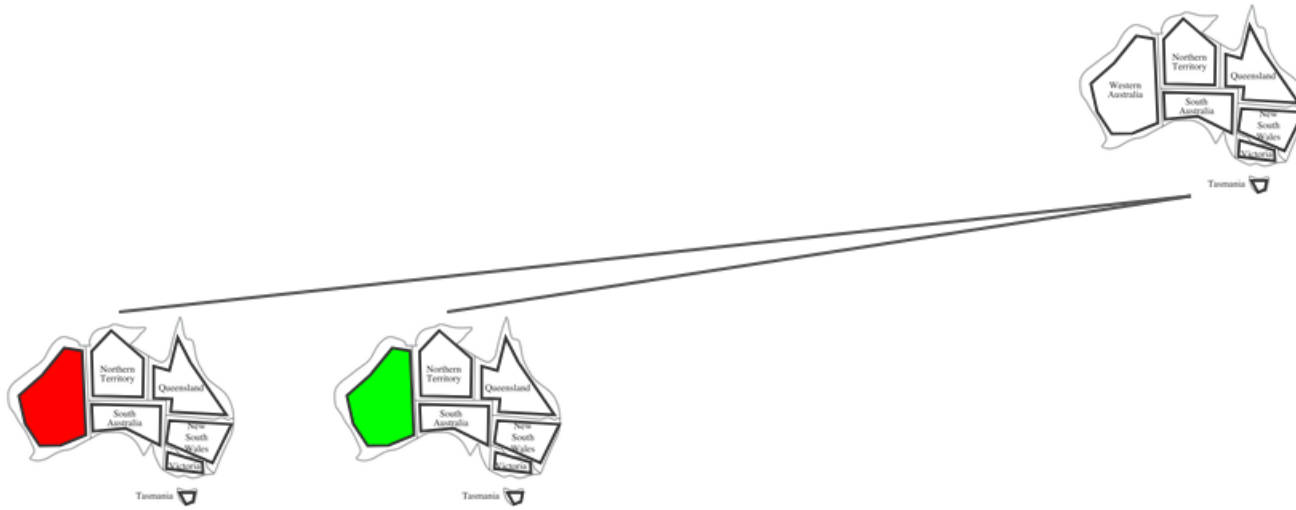
Breadth-First Search



Breadth-First Search



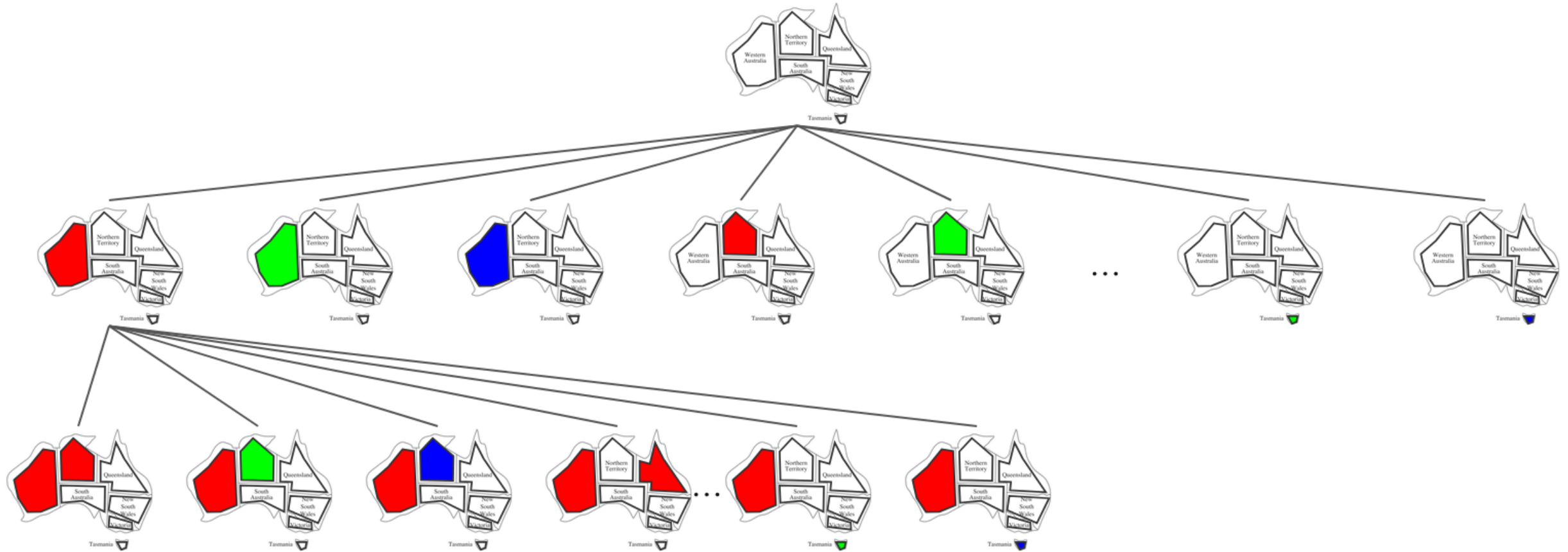
Breadth-First Search



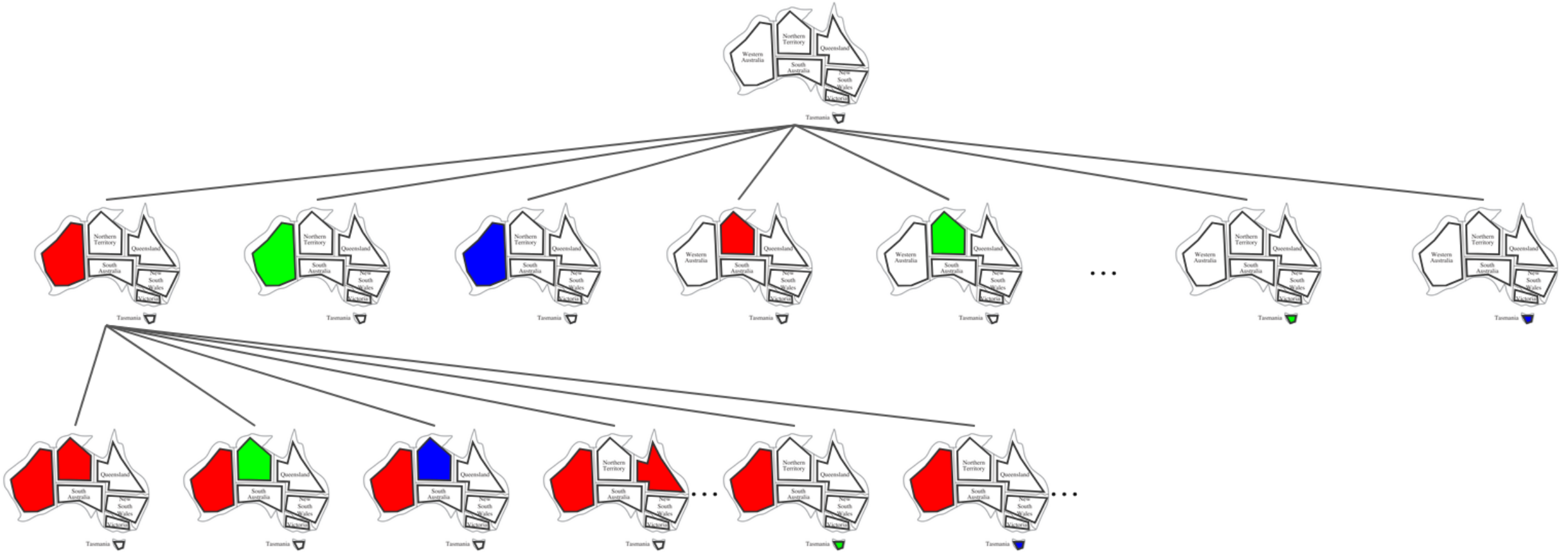
Breadth-First Search



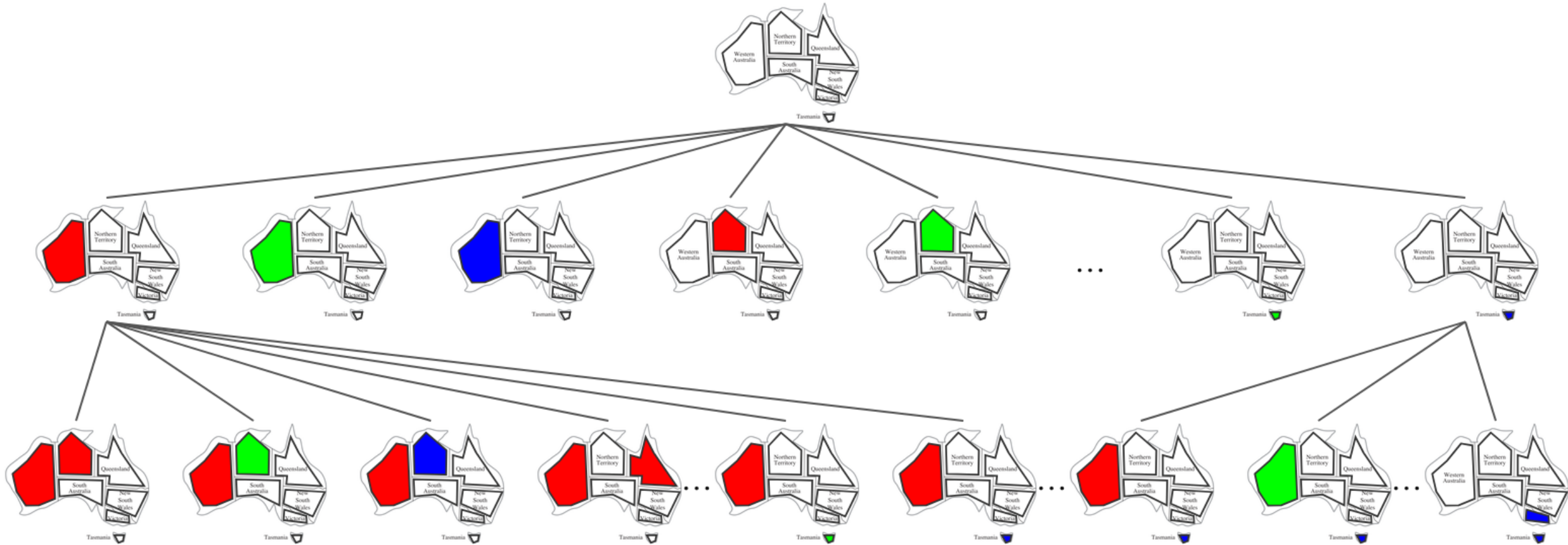
Breadth-First Search



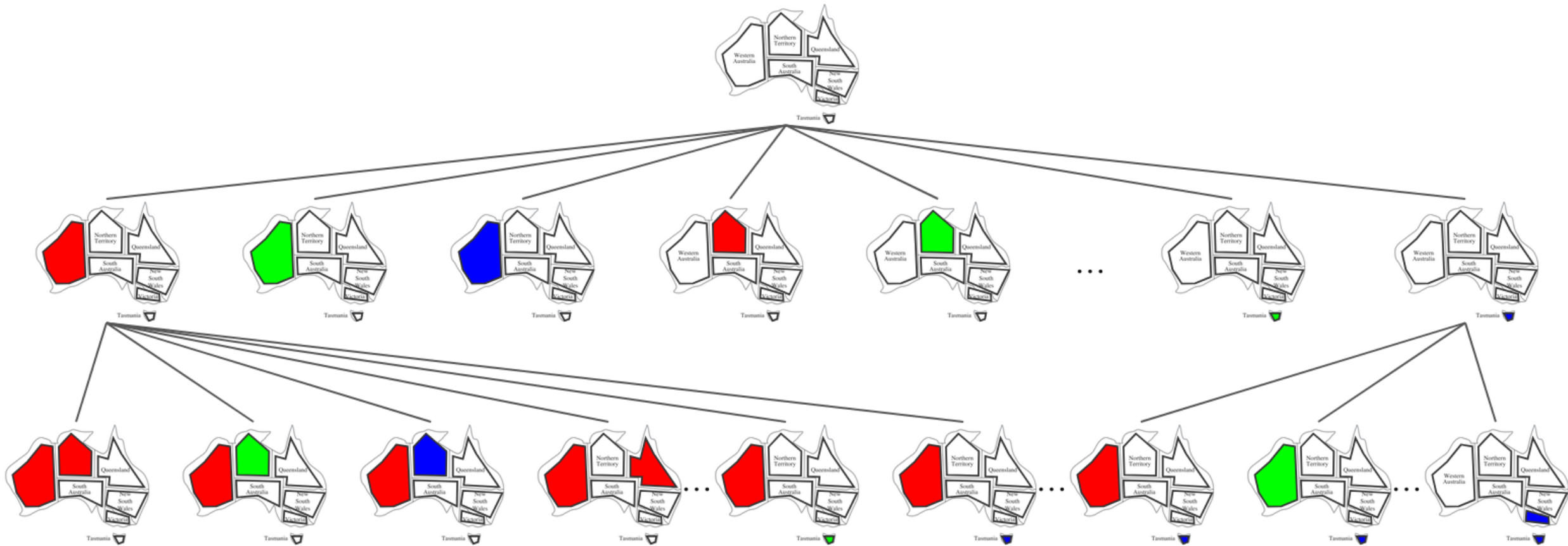
Breadth-First Search



Breadth-First Search

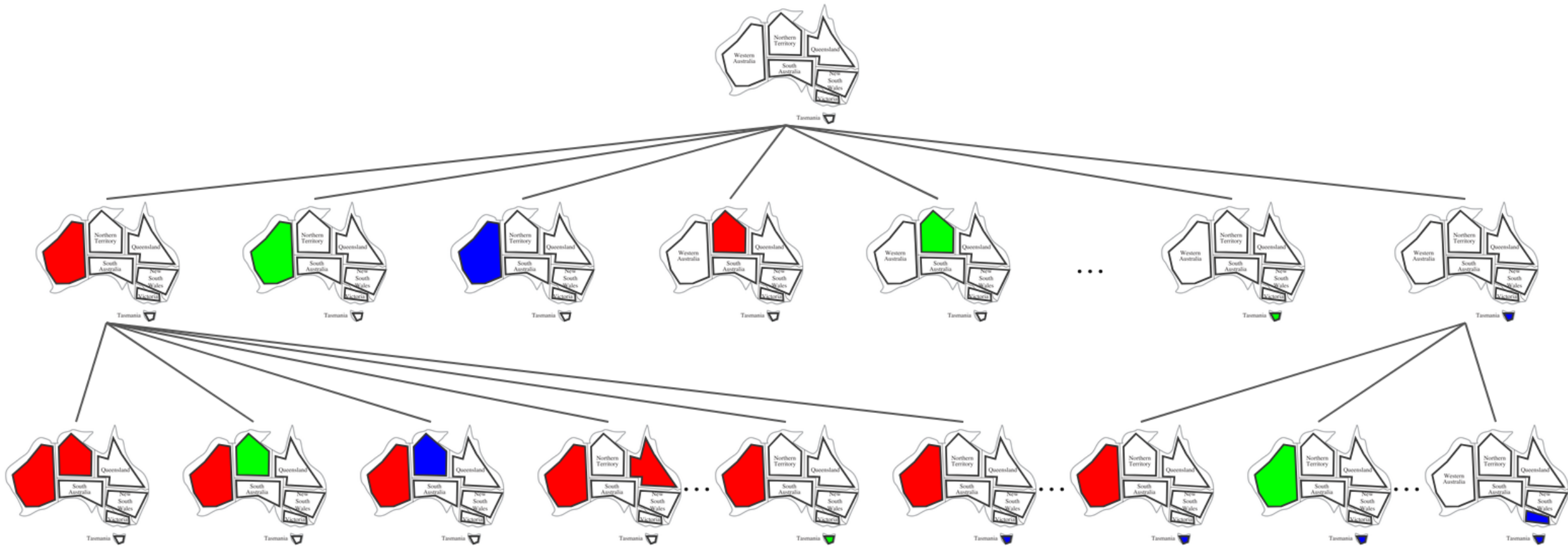


Breadth-First Search



All solutions are at depth 7

Breadth-First Search



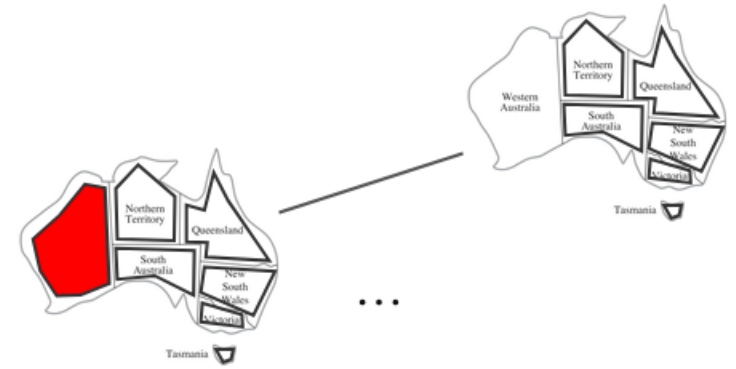
All solutions are at depth 7

Try DFS

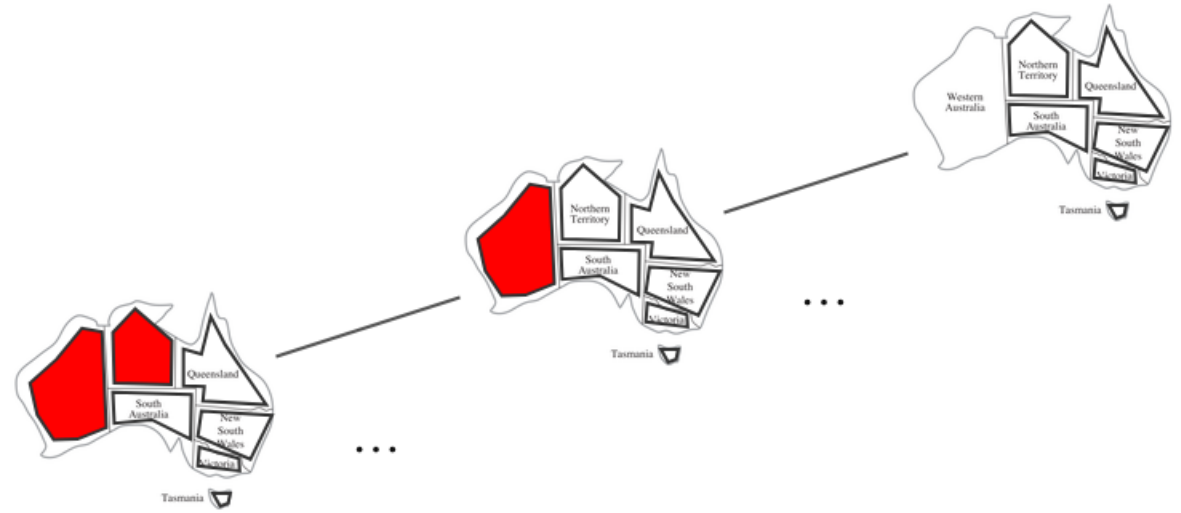
DFS



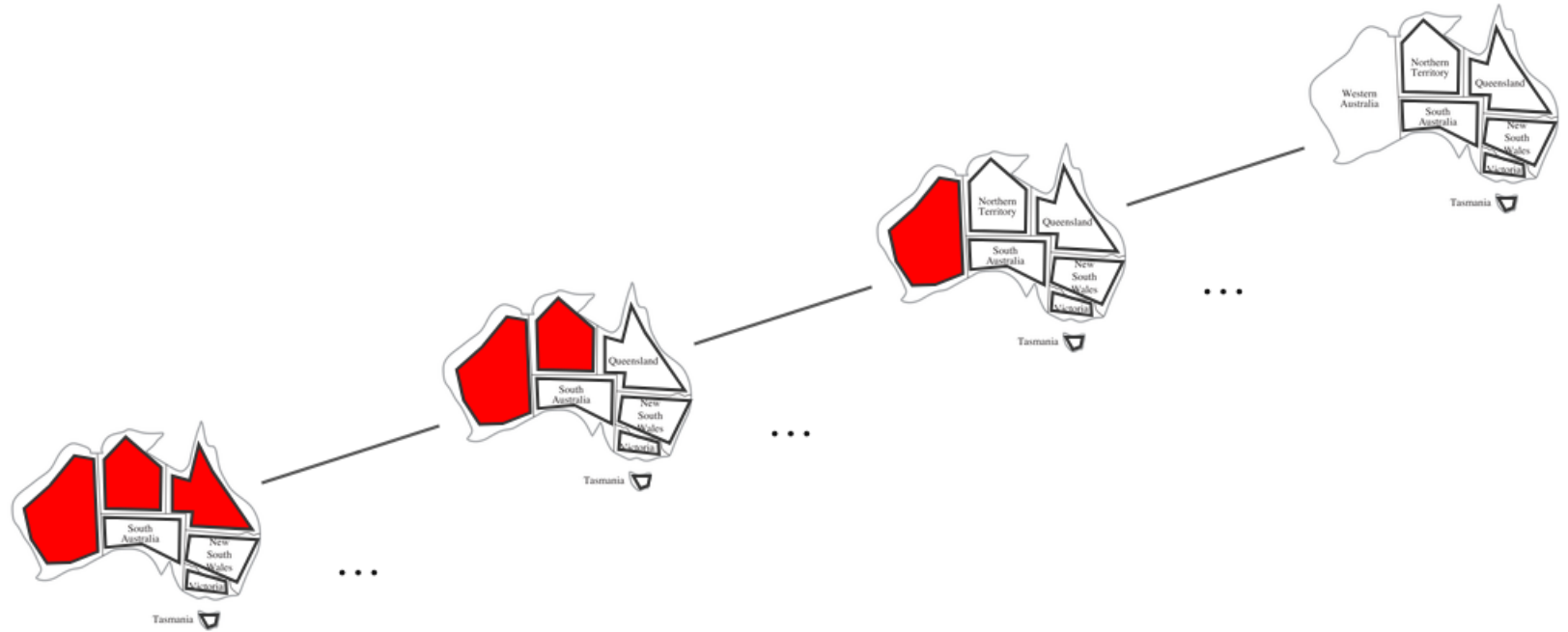
DFS



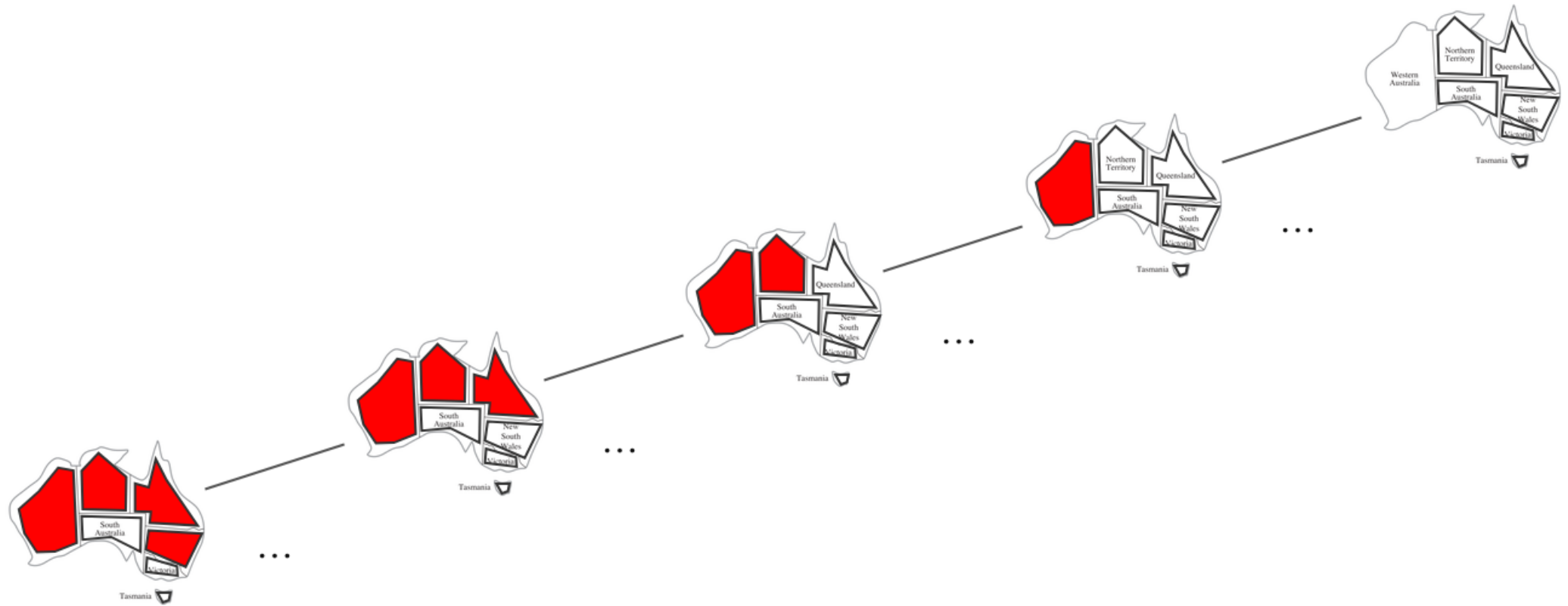
DFS



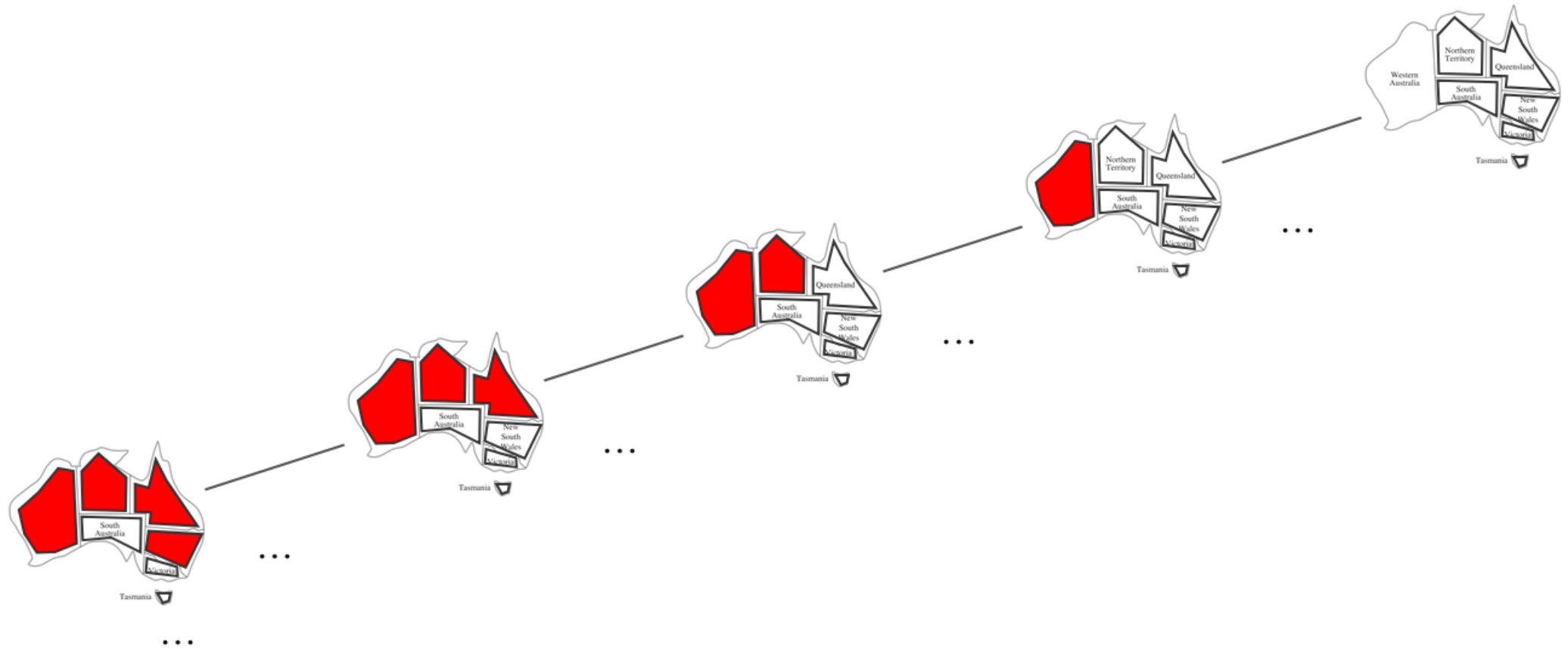
DFS



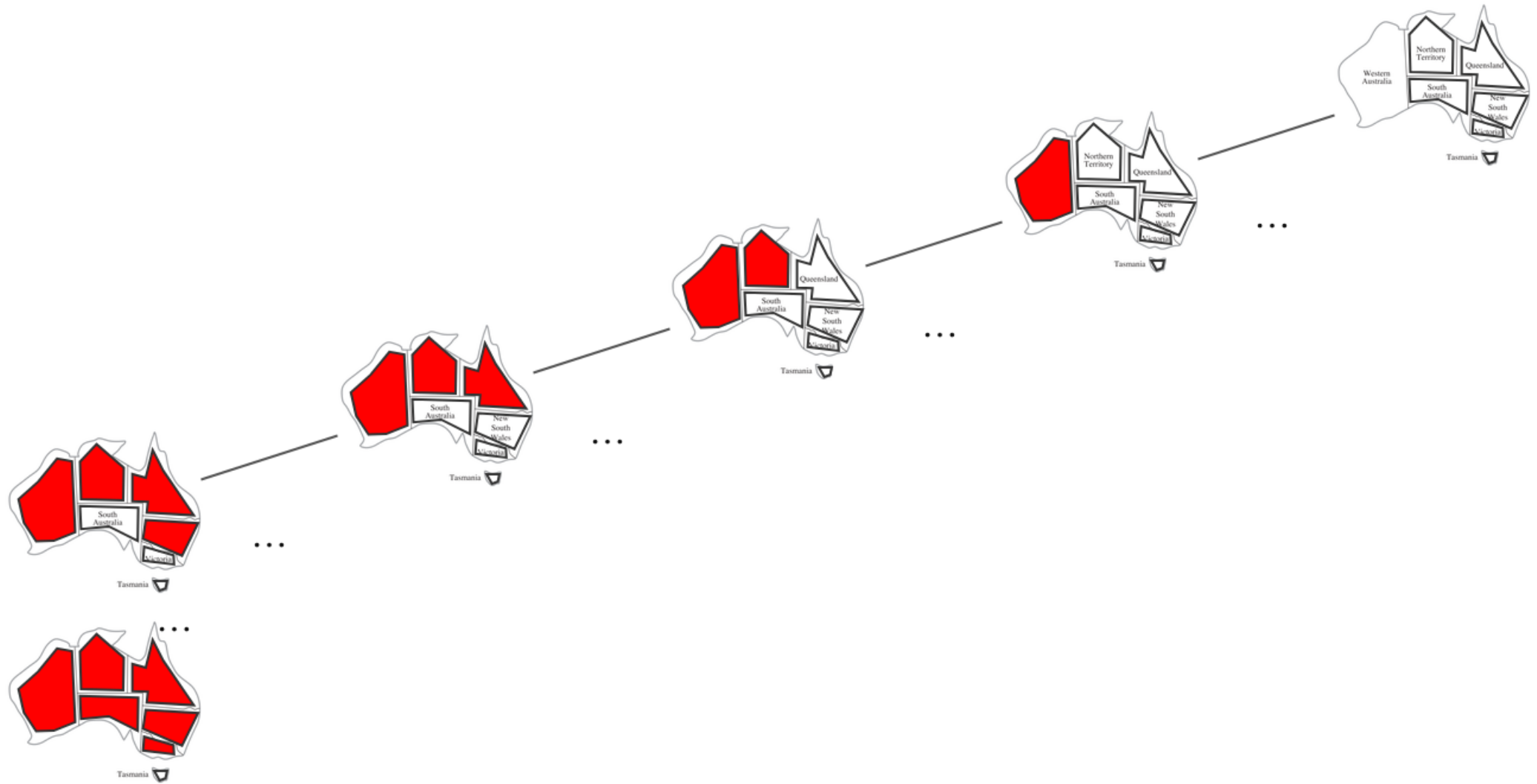
DFS



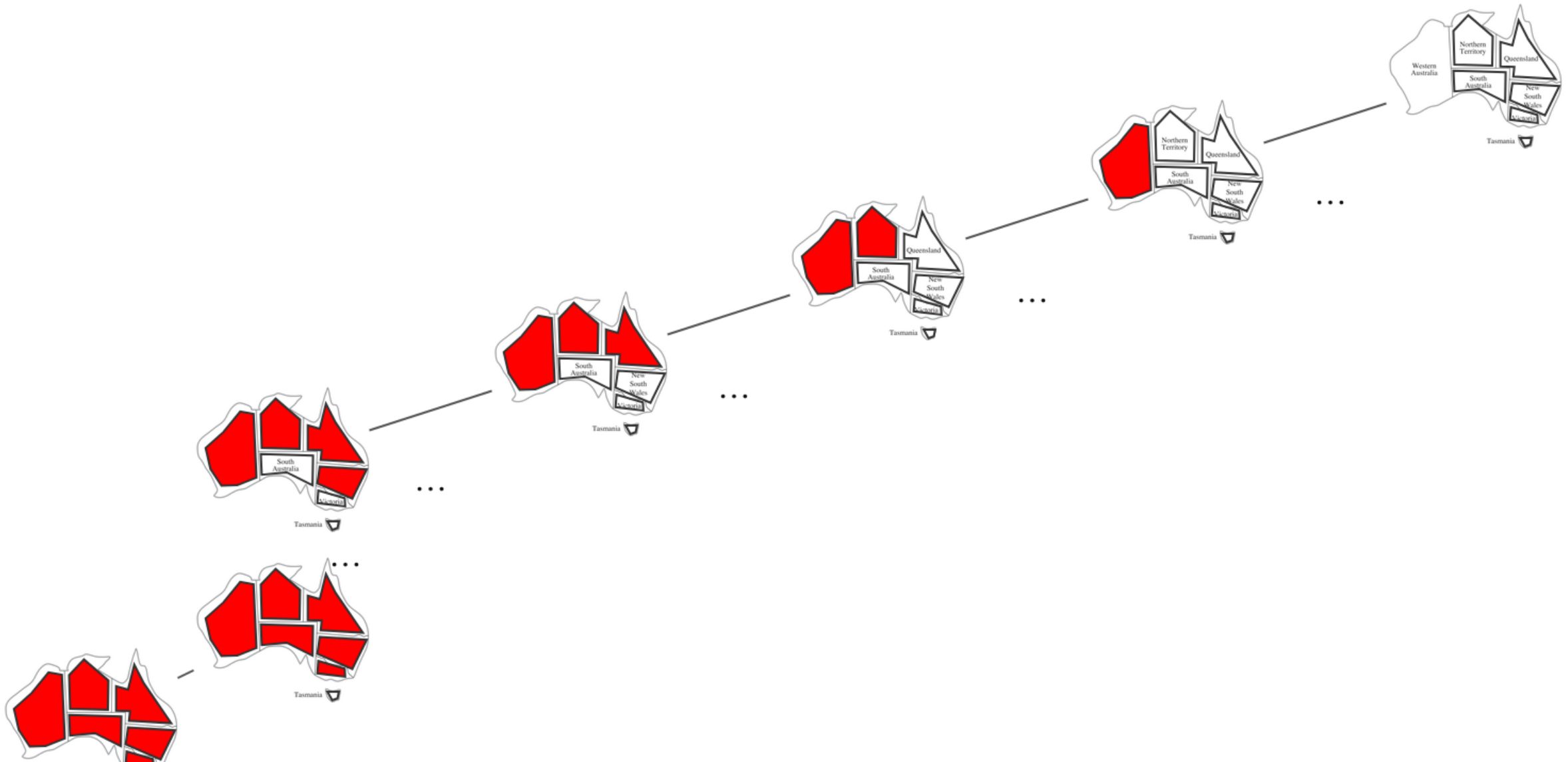
DFS



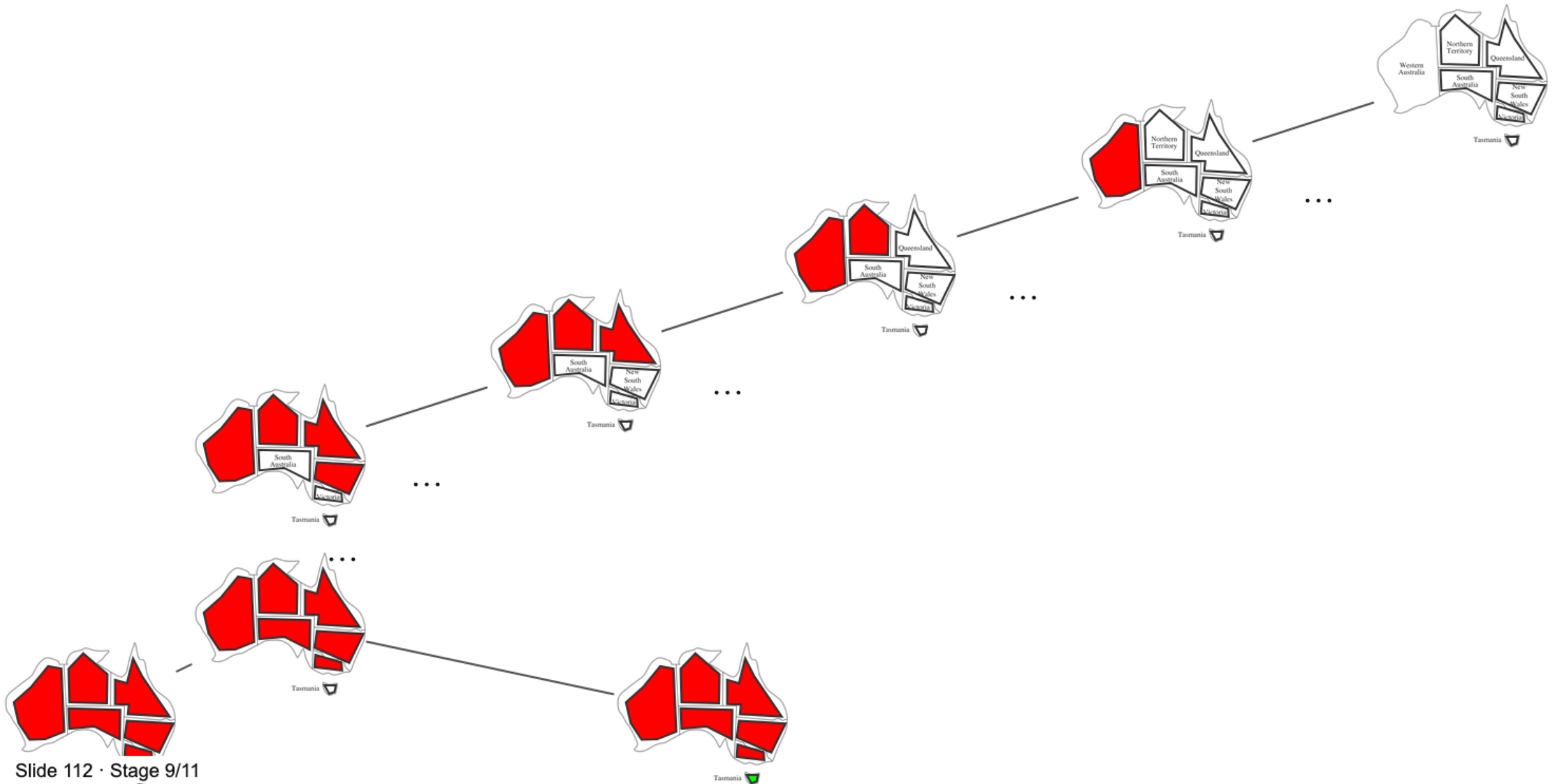
DFS



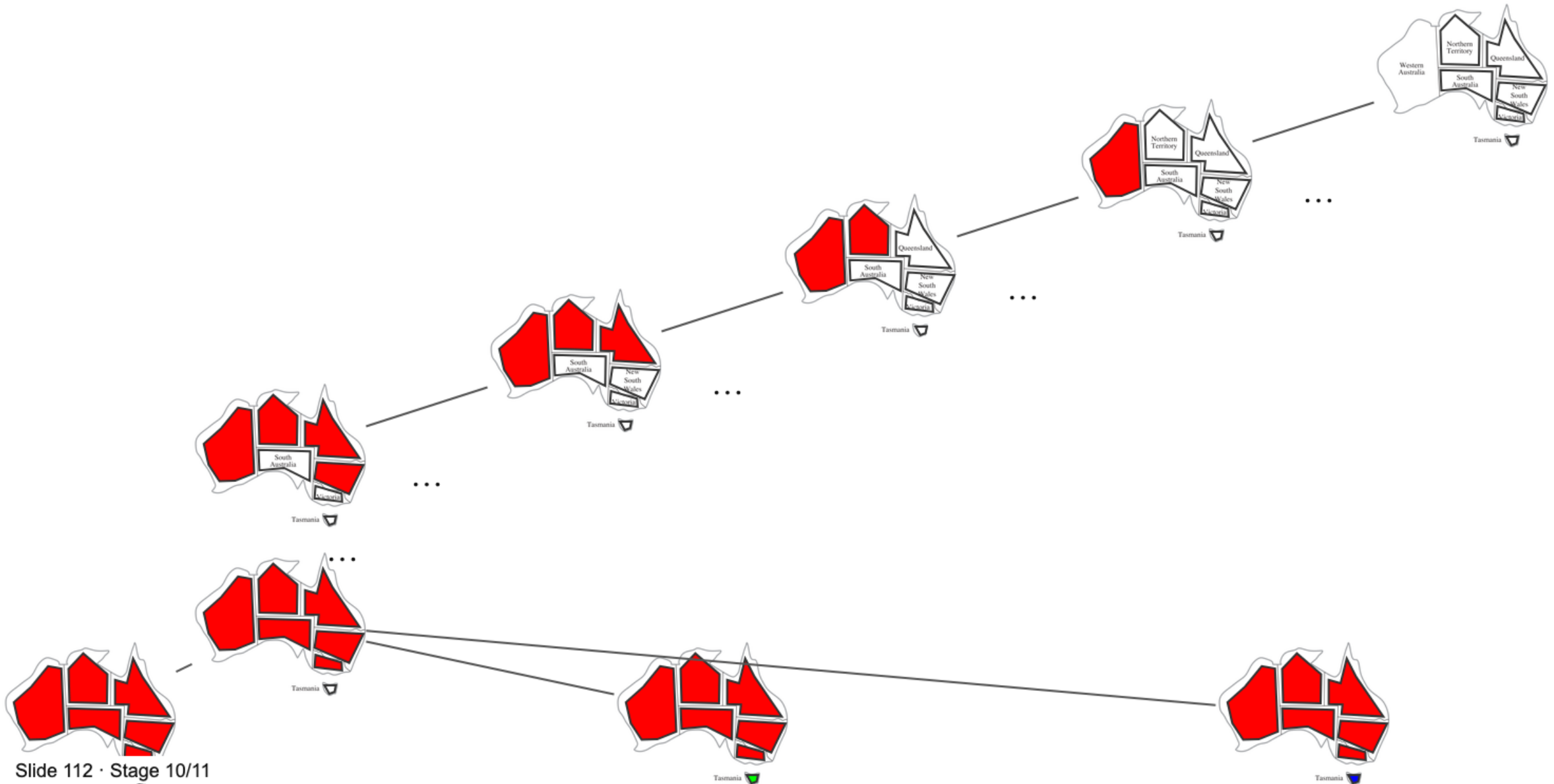
DFS



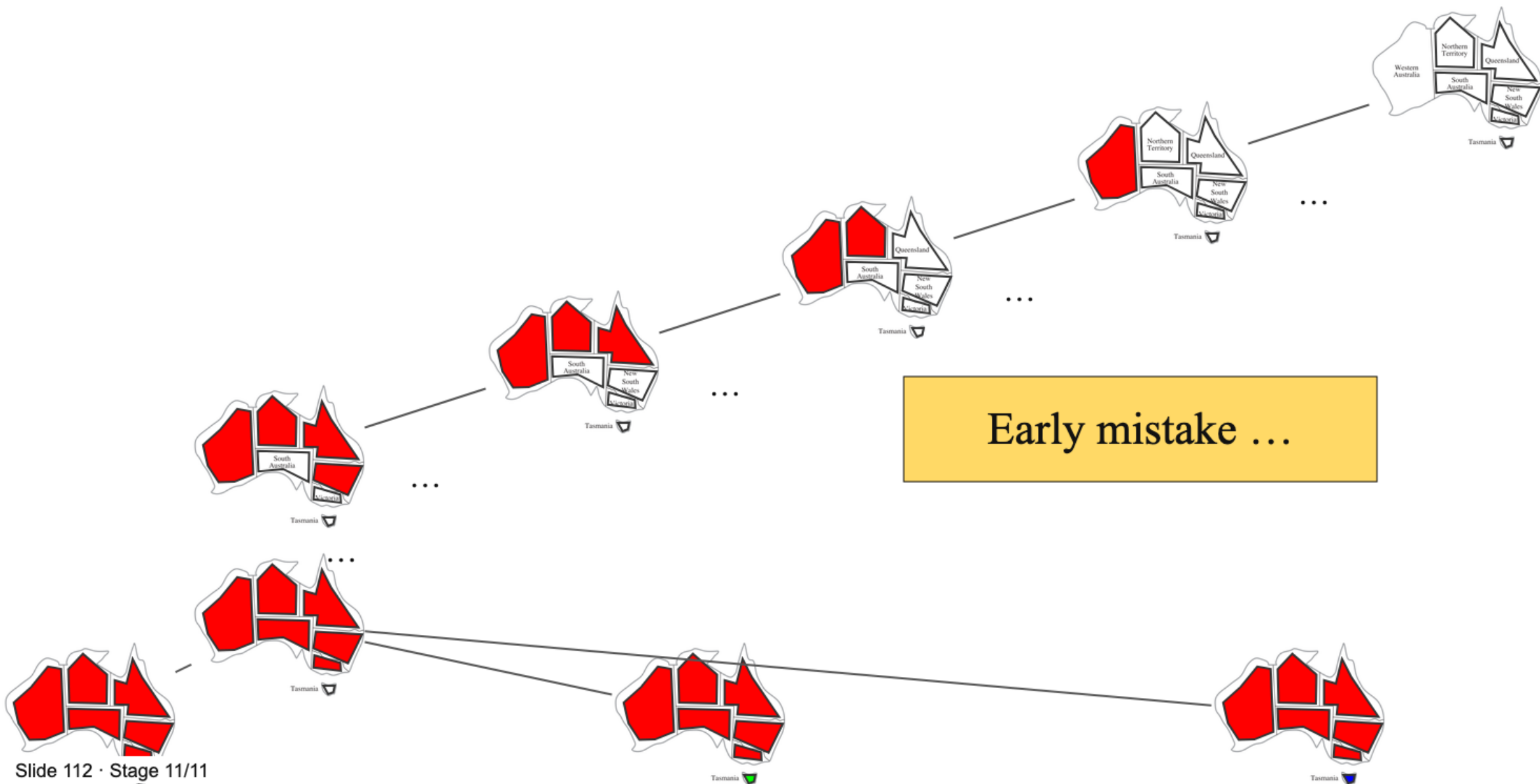
DFS



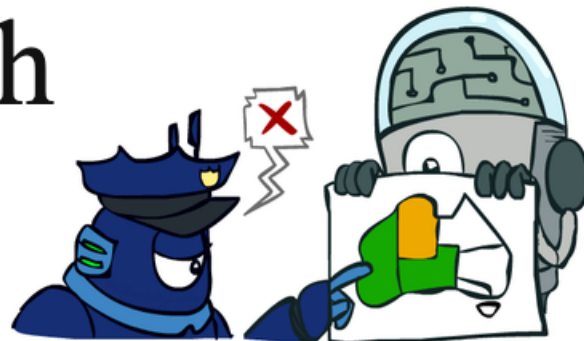
DFS



DFS



Backtracking Search



Choose **one unassigned variable** at each level

Branch over its legal values

Try only values consistent with earlier assignments

If no legal value remains, **backtrack**

Depth-first search with these two ideas is backtracking search

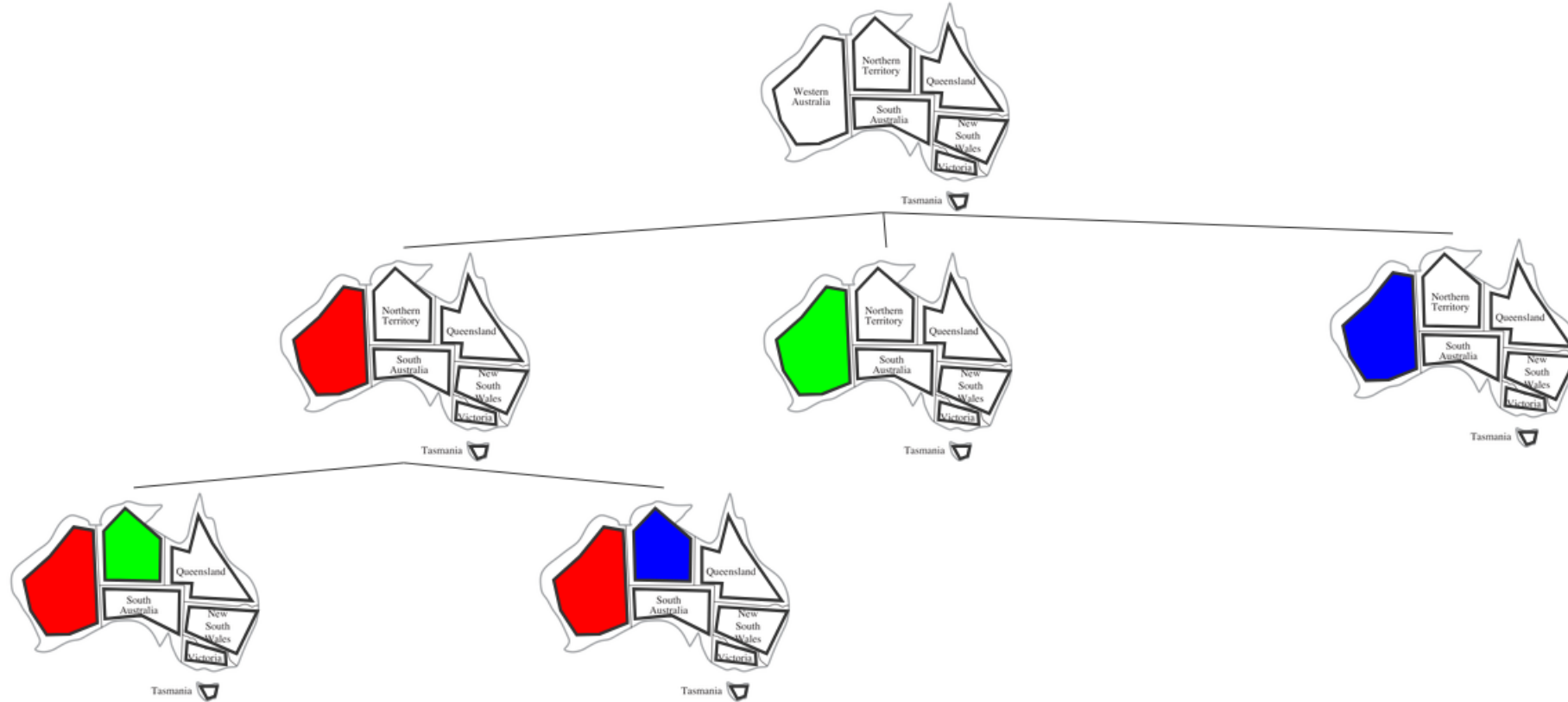
Backtracking



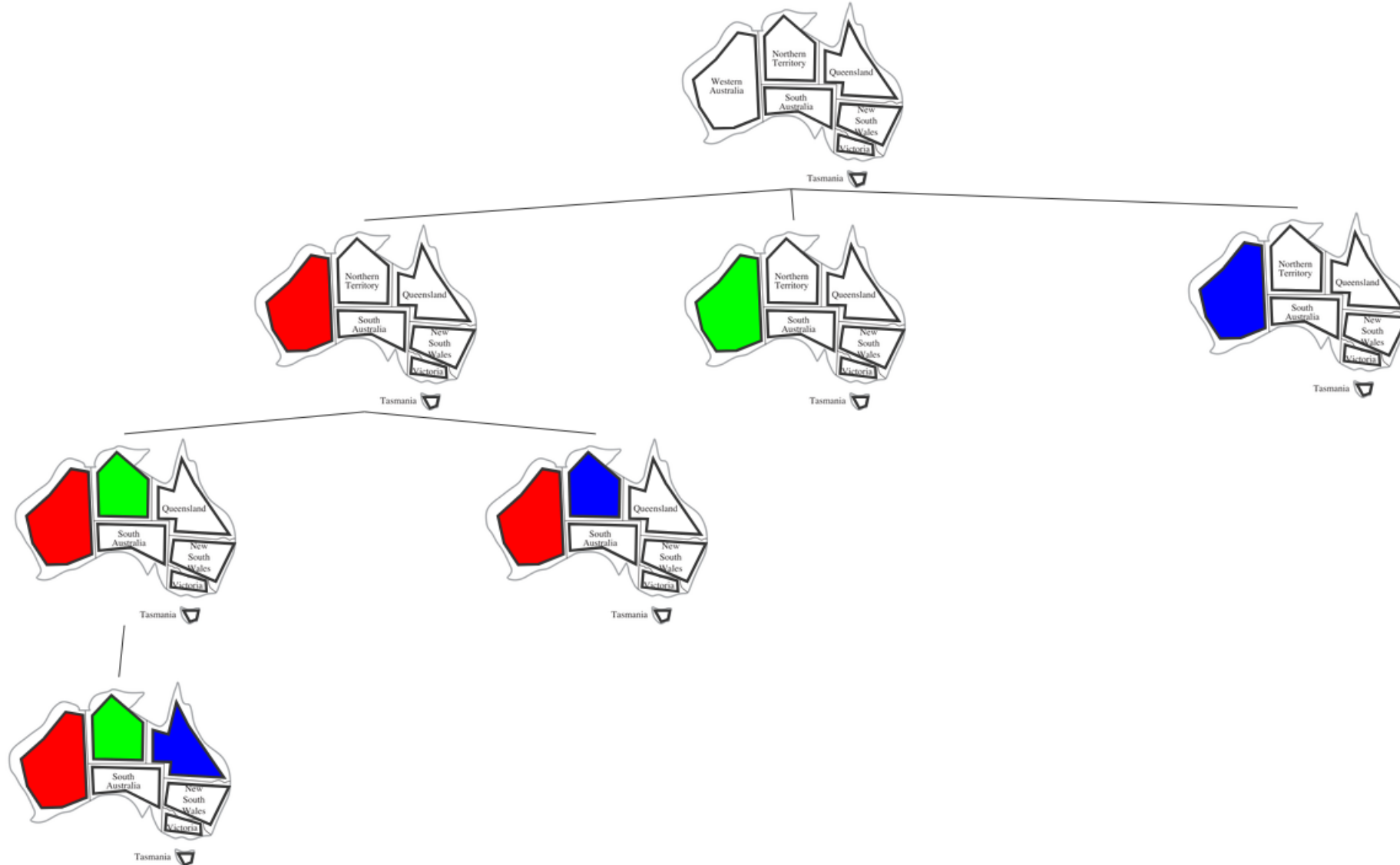
Backtracking



Backtracking



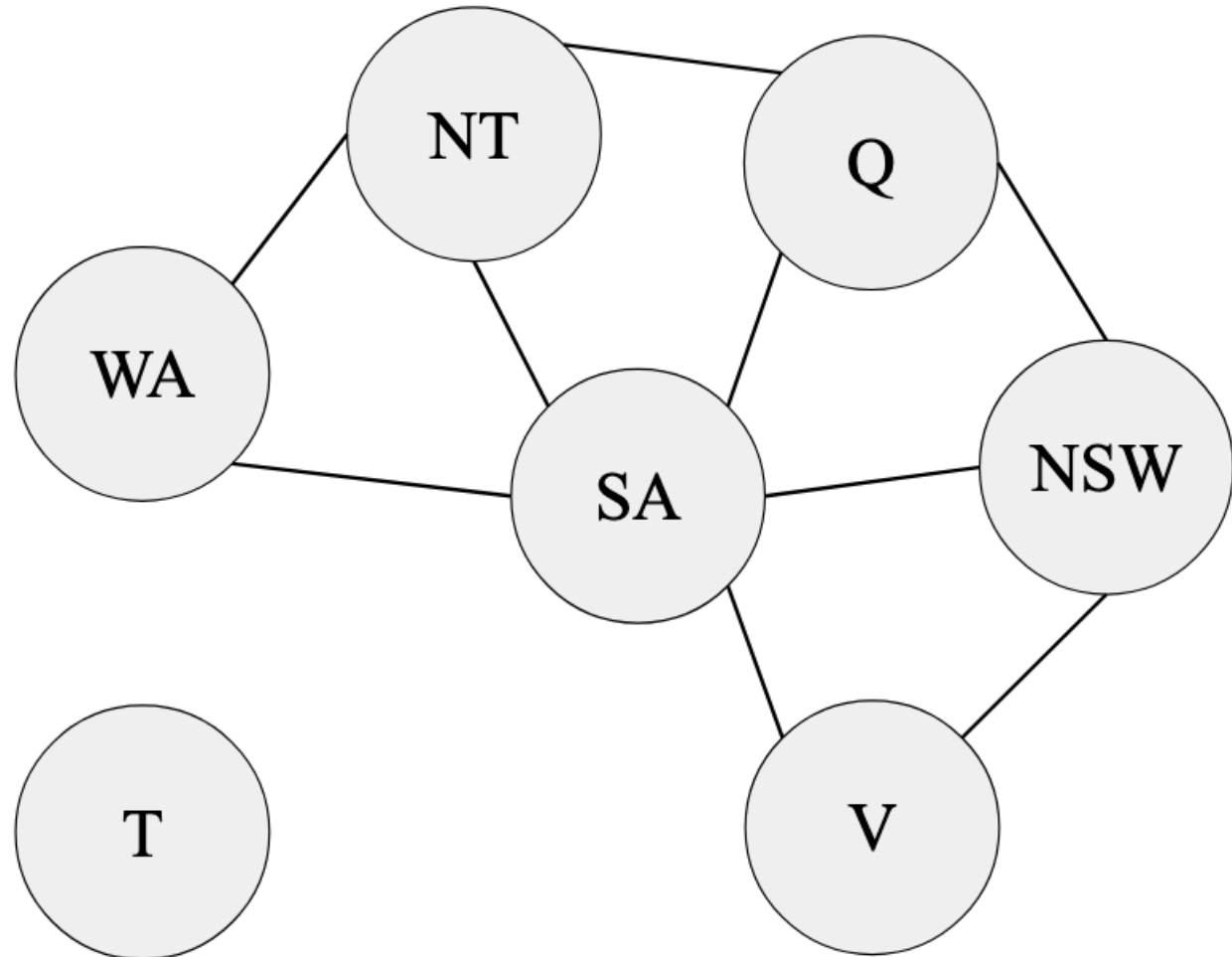
Backtracking



Backtracking

Variables: {NSW, WA, NT, Q, SA, V, T}

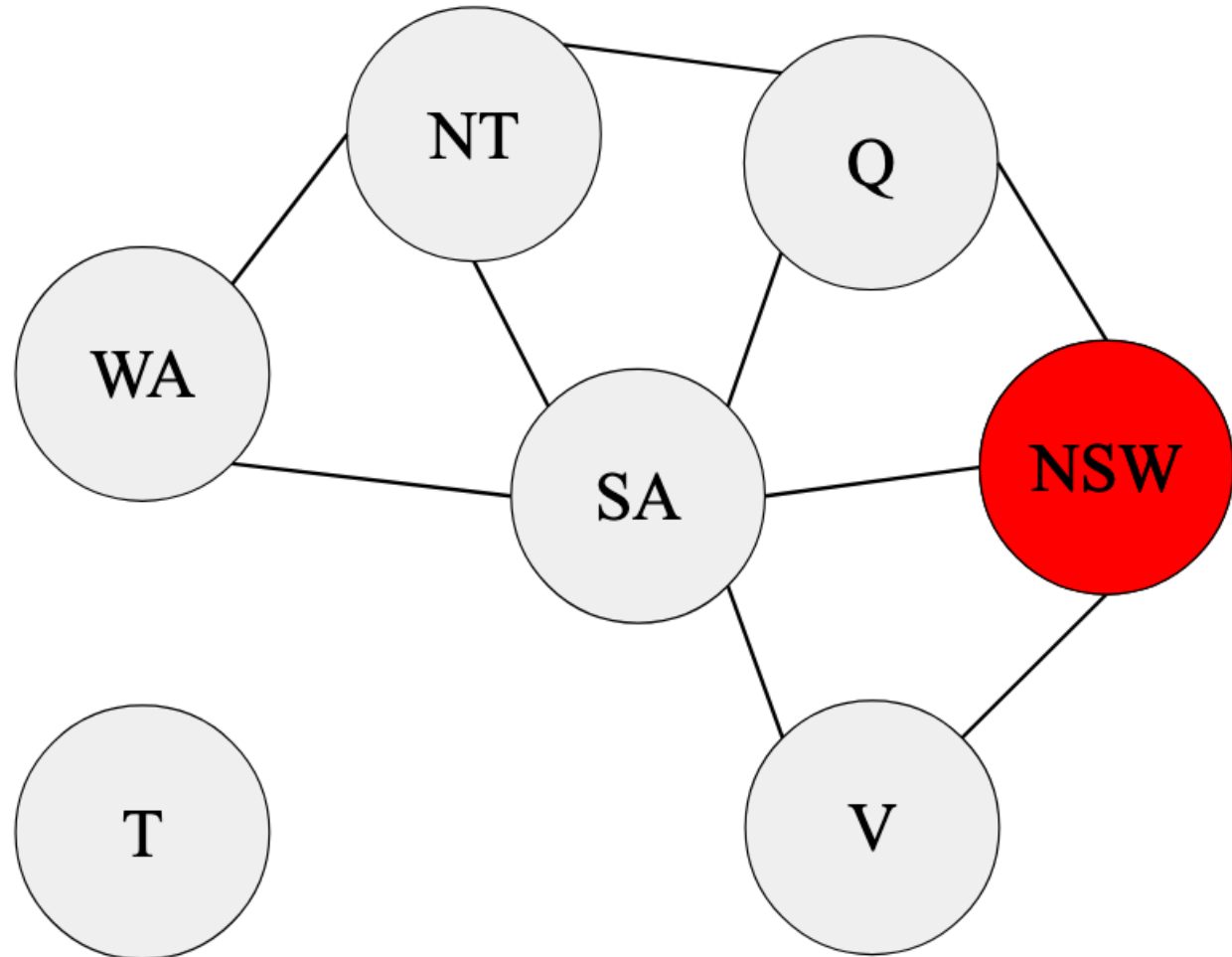
Colors: {red, green, blue}



Backtracking

Variables: {NSW, WA, NT, Q, SA, V, T}

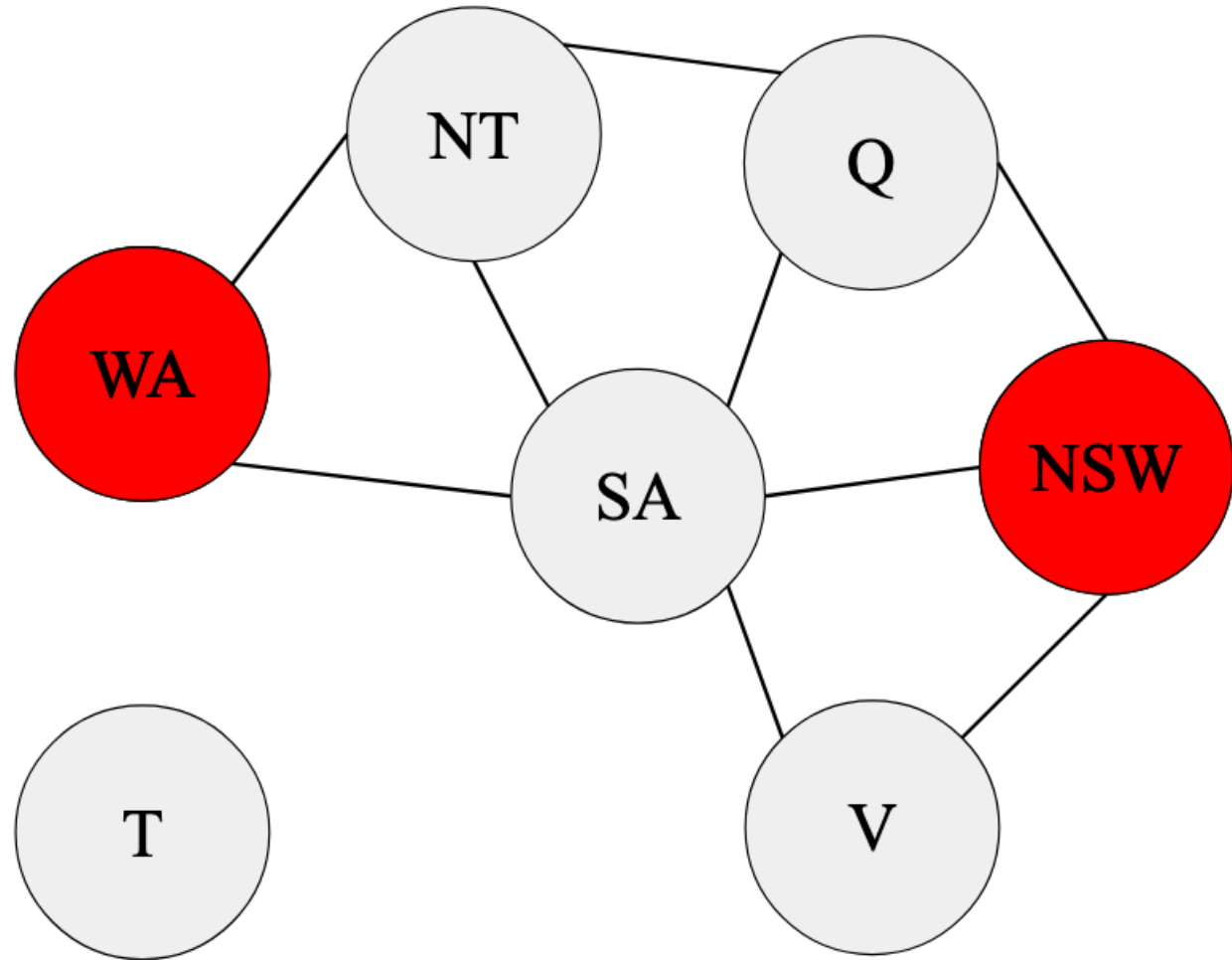
Colors: {red, green, blue}



Backtracking

Variables: {NSW, WA, NT, Q, SA, V, T}

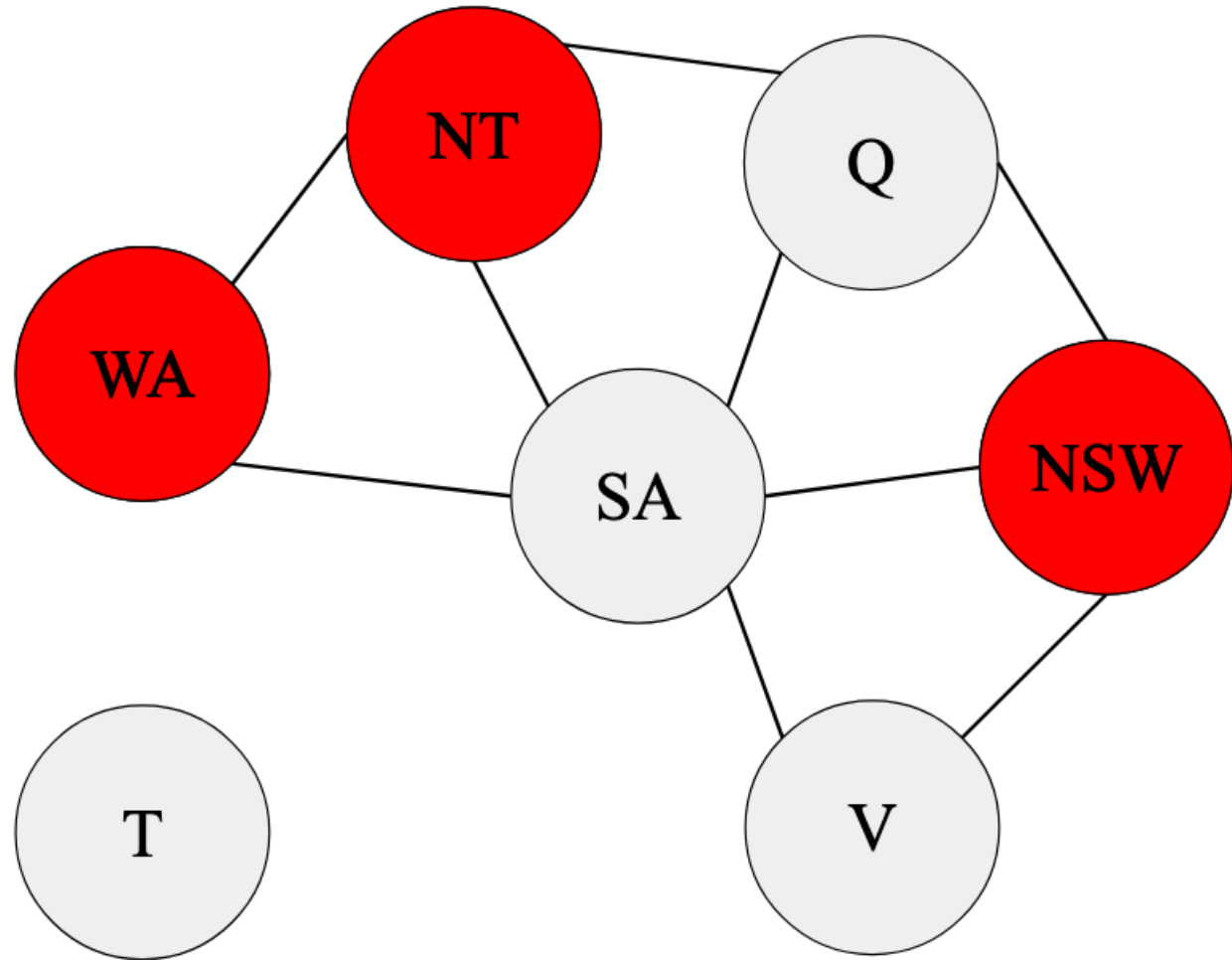
Colors: {red, green, blue}



Backtracking

Variables: {NSW, WA, NT, Q, SA, V, T}

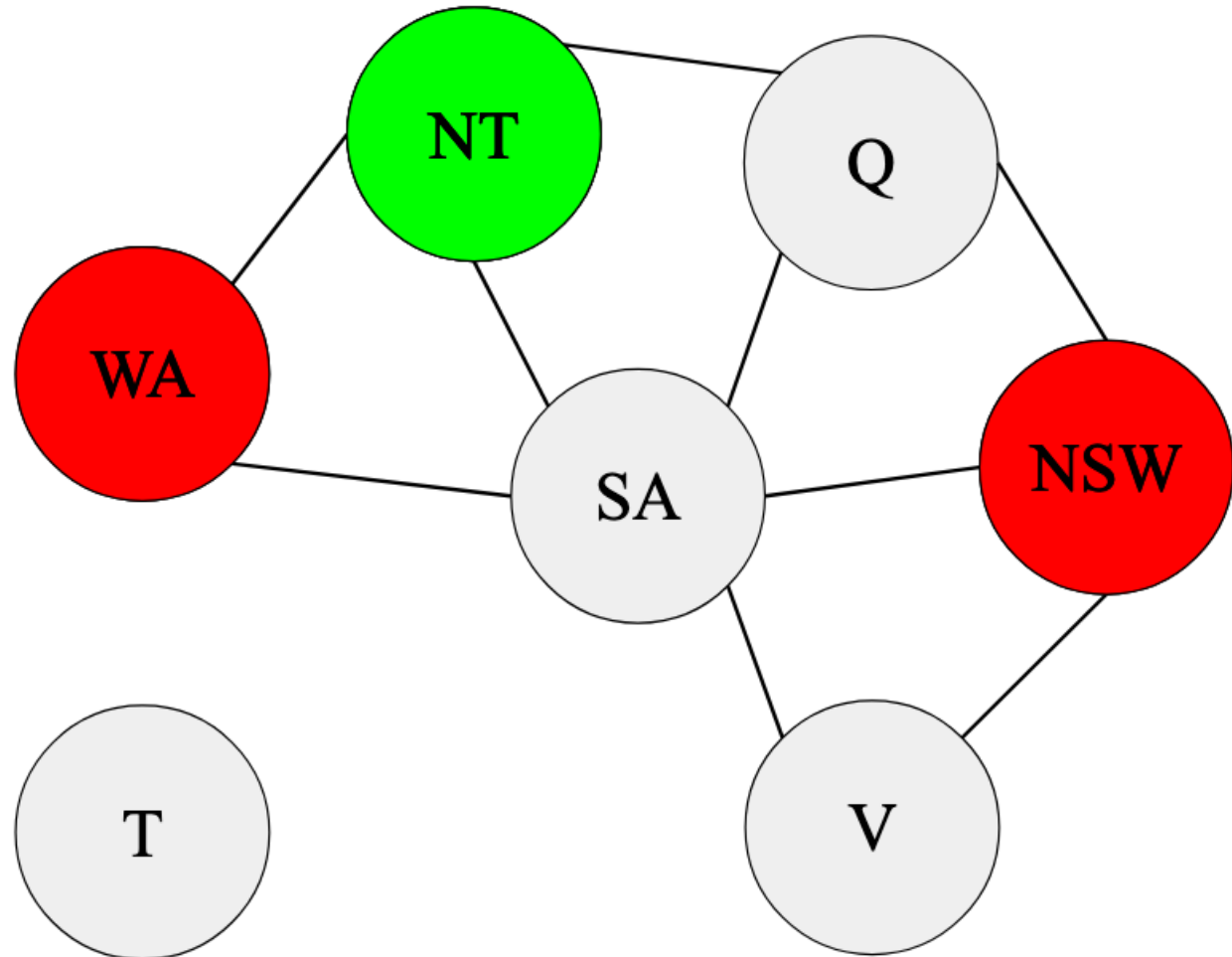
Colors: {red, green, blue}



Backtracking

Variables: {NSW, WA, NT, Q, SA, V, T}

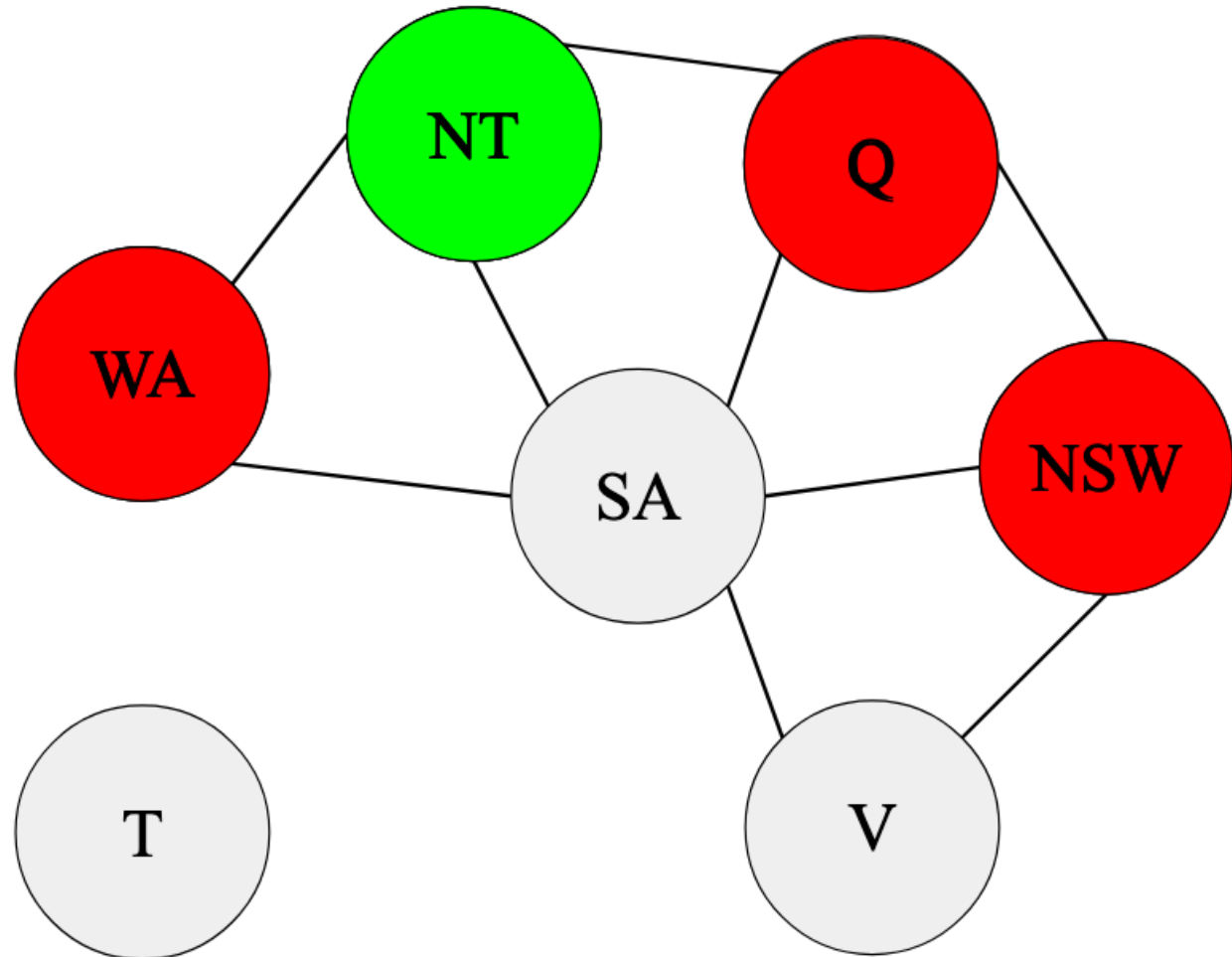
Colors: {red, green, blue}



Backtracking

Variables: {NSW, WA, NT, Q, SA, V, T}

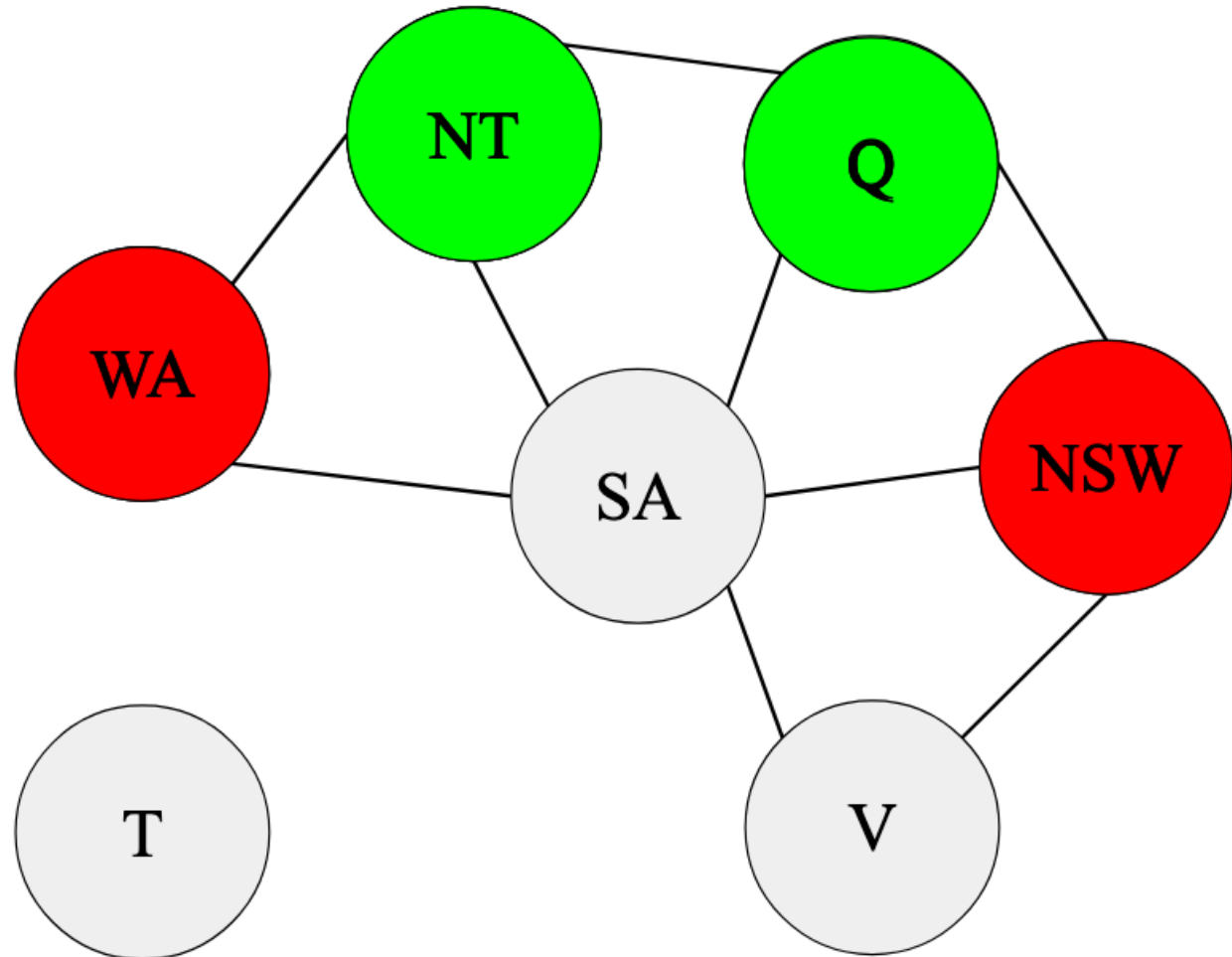
Colors: {red, green, blue}



Backtracking

Variables: {NSW, WA, NT, Q, SA, V, T}

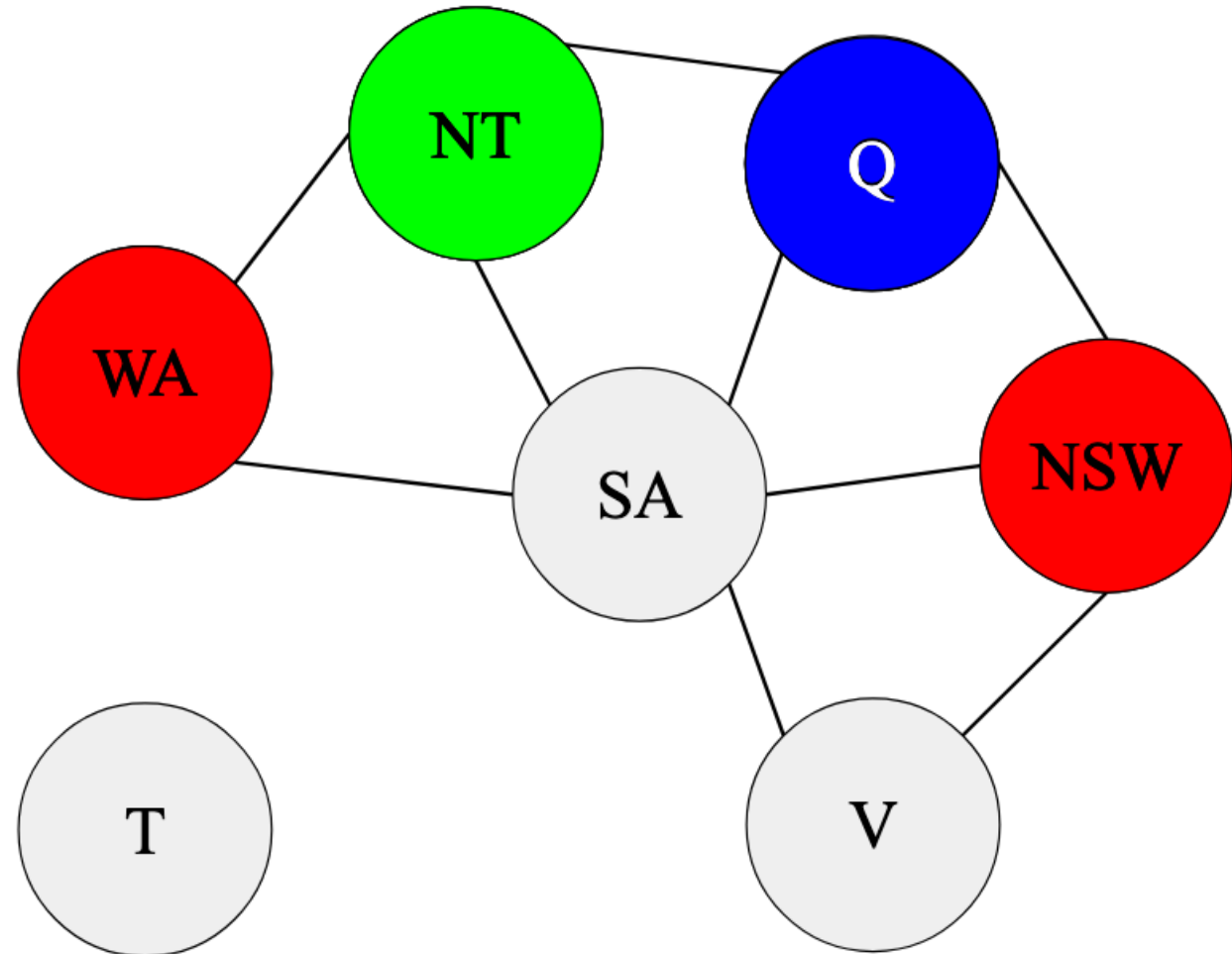
Colors: {red, green, blue}



Backtracking

Variables: {NSW, WA, NT, Q, SA, V, T}

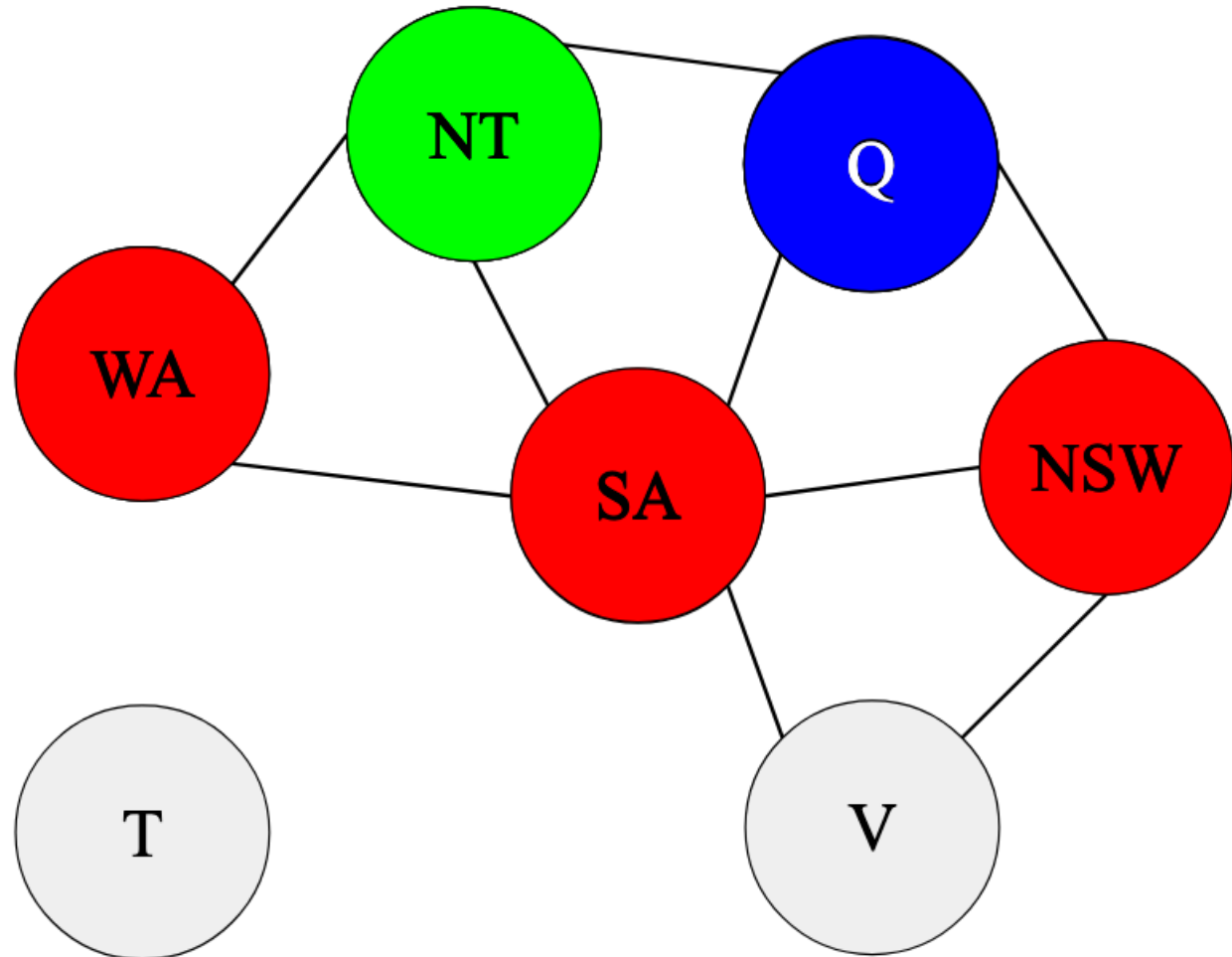
Colors: {red, green, blue}



Backtracking

Variables: {NSW, WA, NT, Q, SA, V, T}

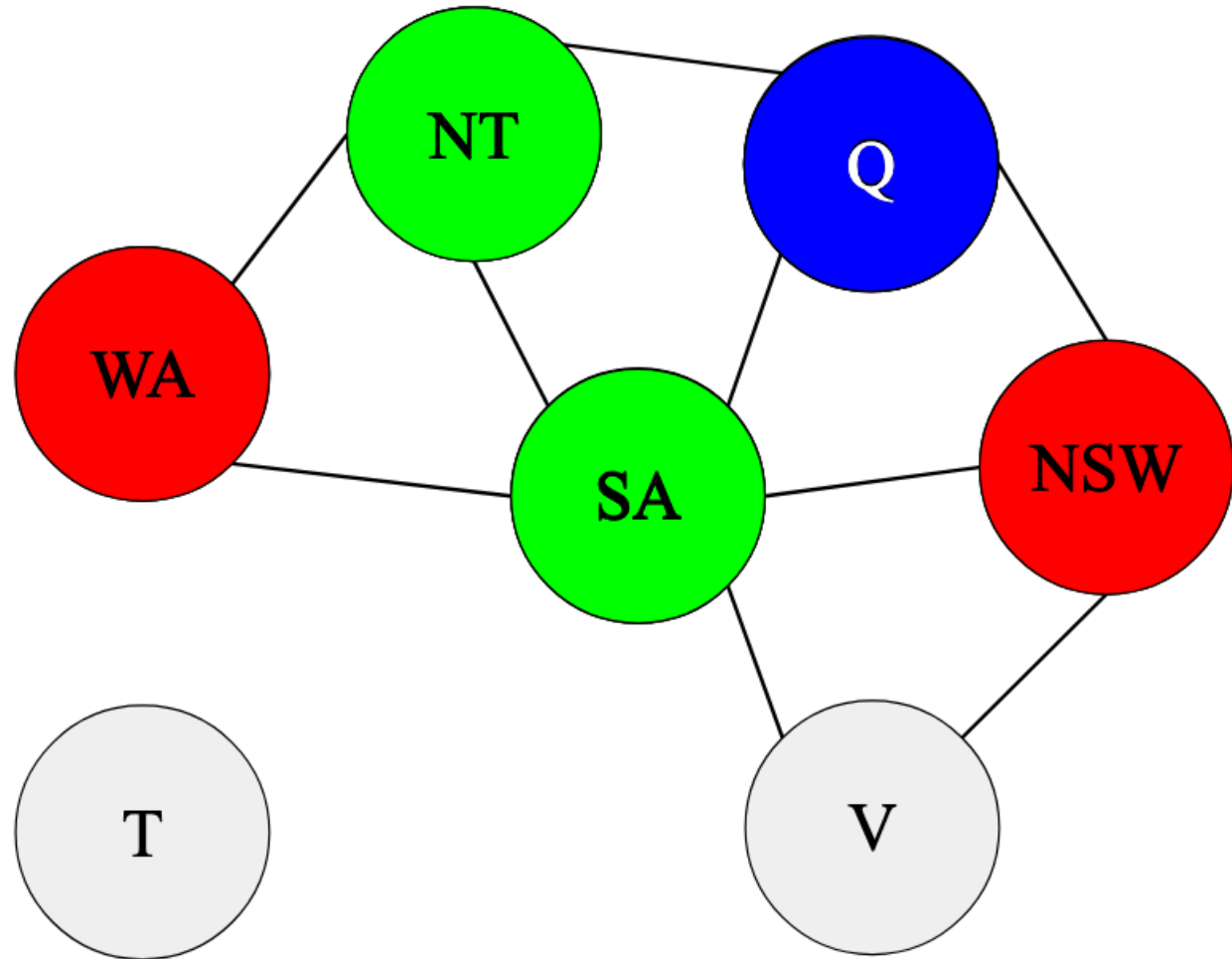
Colors: {red, green, blue}



Backtracking

Variables: {NSW, WA, NT, Q, SA, V, T}

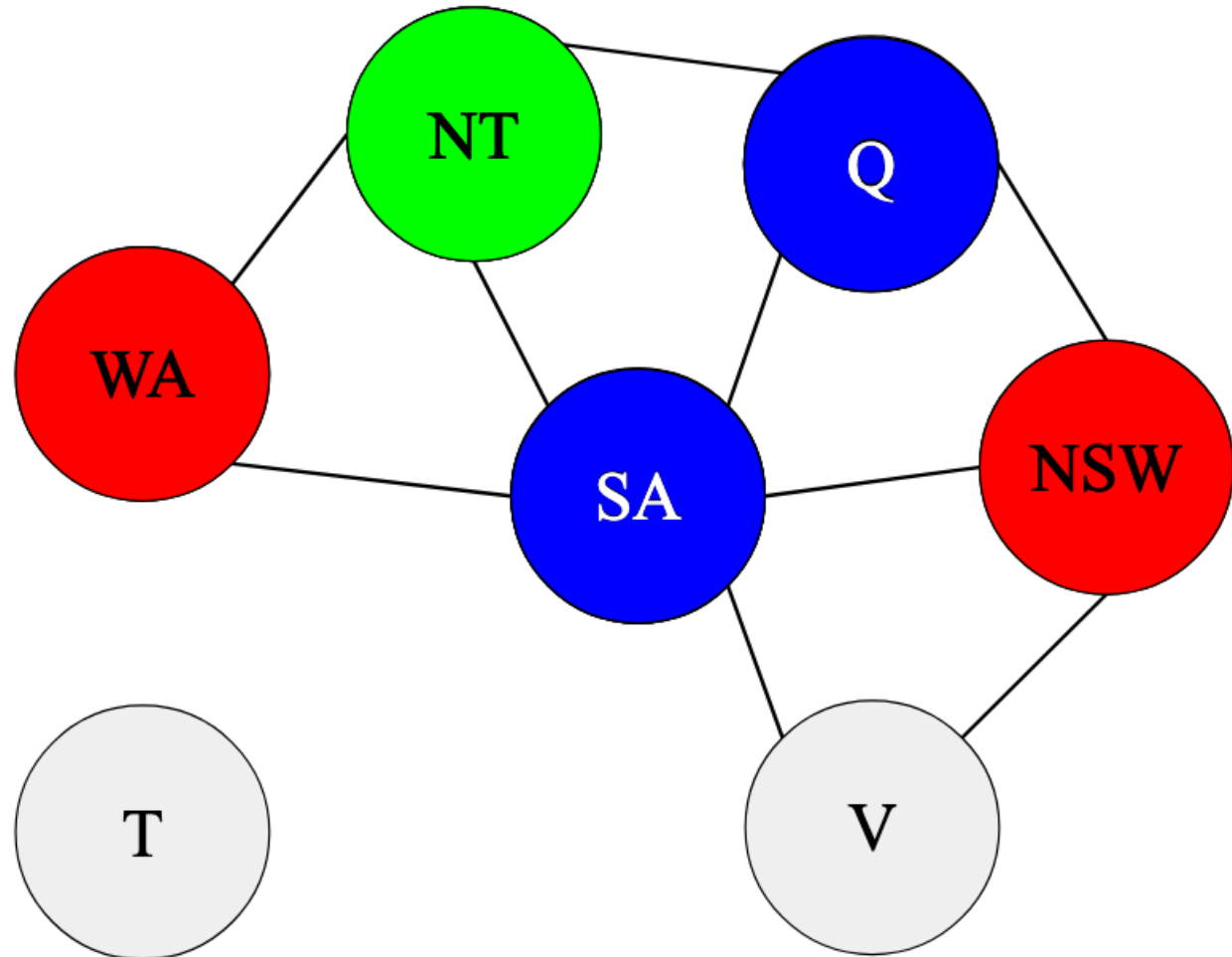
Colors: {red, green, blue}



Backtracking

Variables: {NSW, WA, NT, Q, SA, V, T}

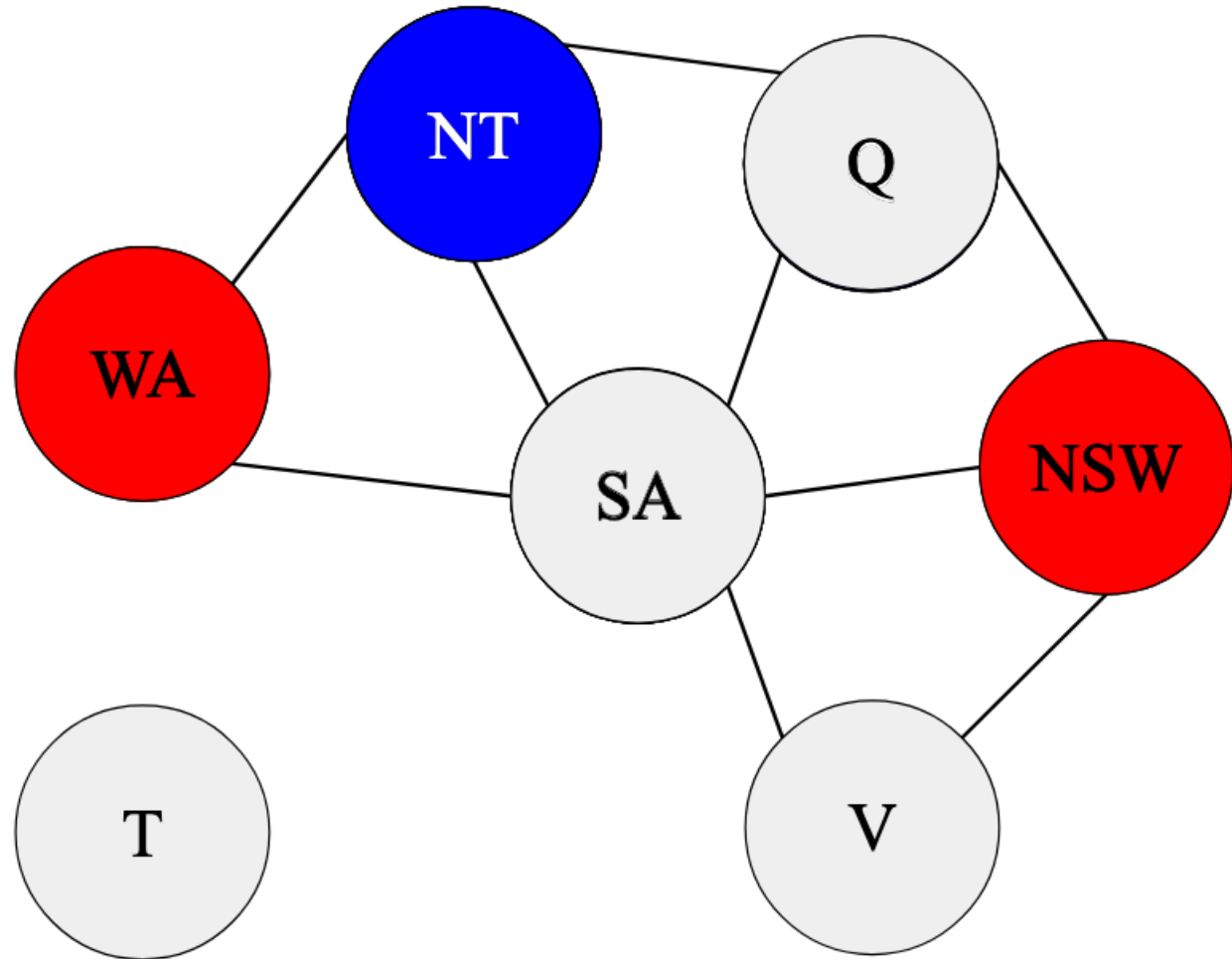
Colors: {red, green, blue}



Backtracking

Variables: {NSW, WA, NT, Q, SA, V, T}

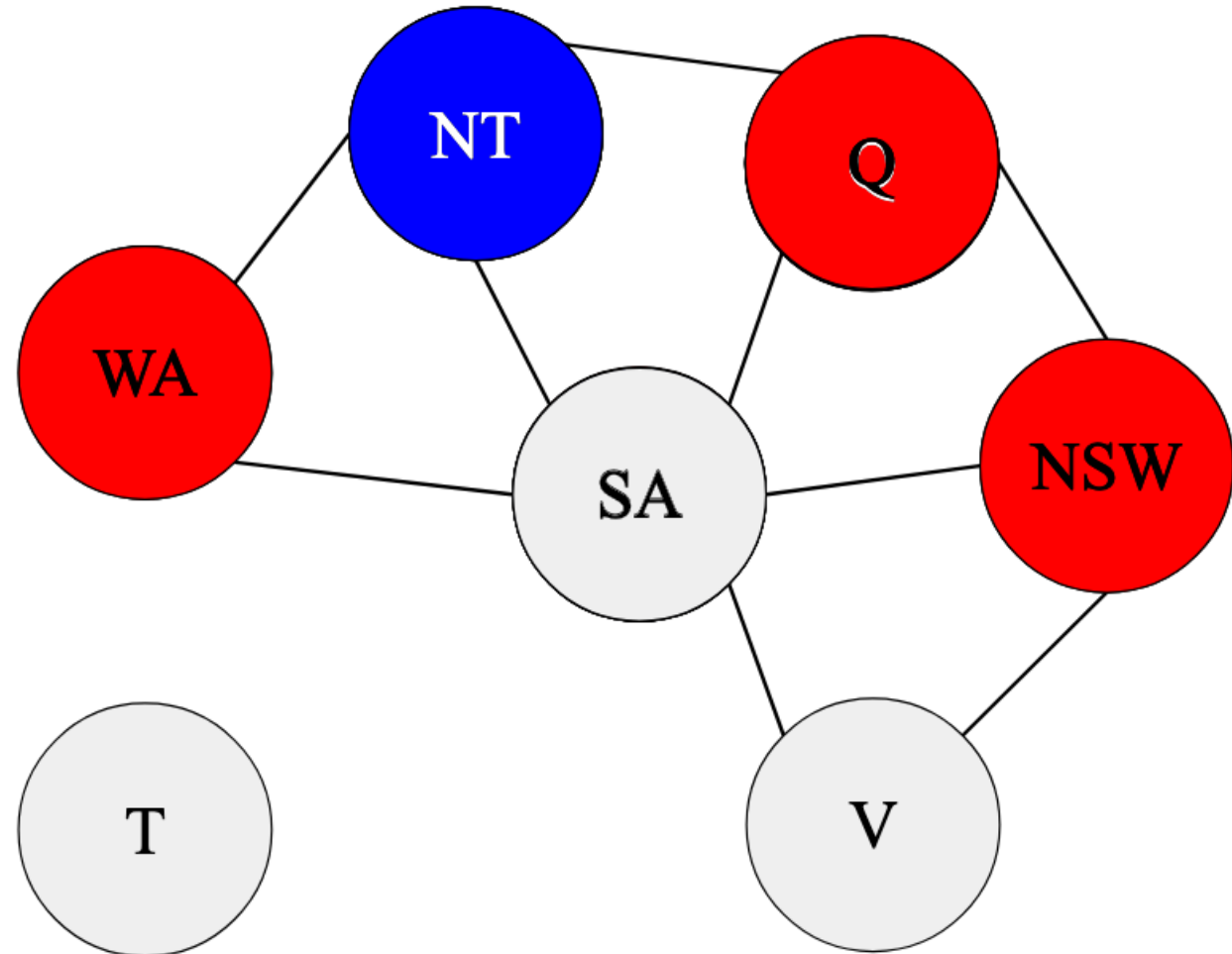
Colors: {red, green, blue}



Backtracking

Variables: {NSW, WA, NT, Q, SA, V, T}

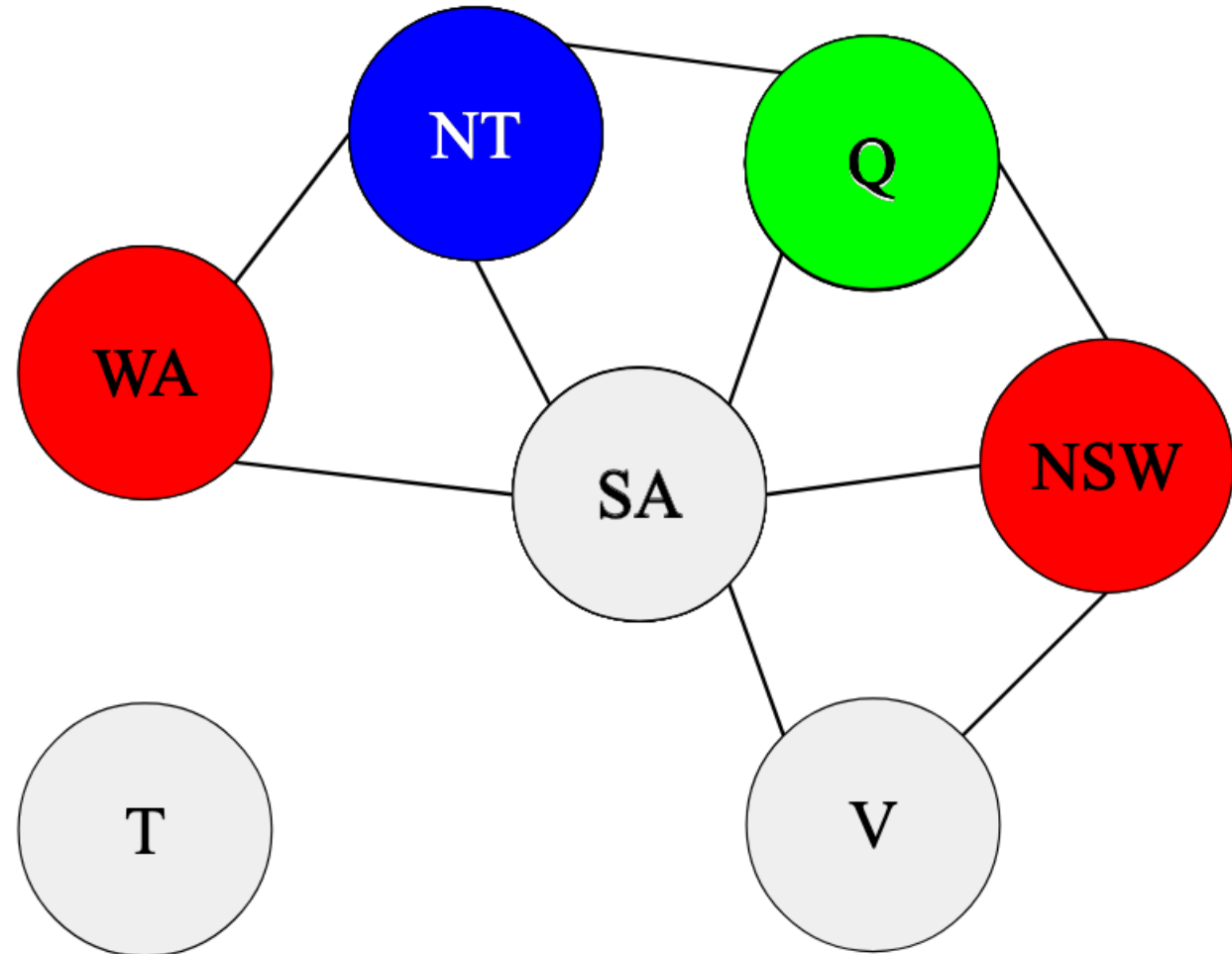
Colors: {red, green, blue}



Backtracking

Variables: {NSW, WA, NT, Q, SA, V, T}

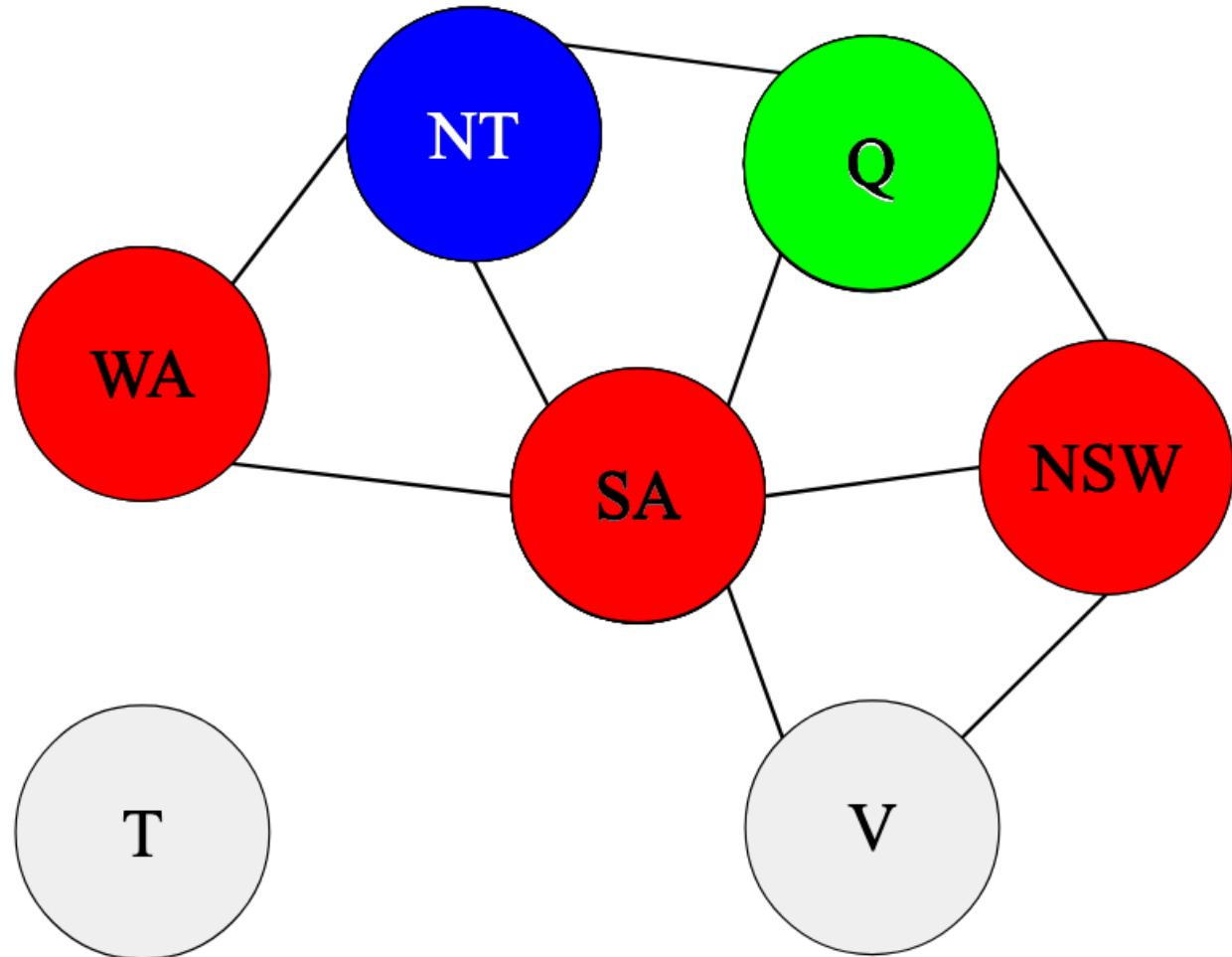
Colors: {red, green, blue}



Backtracking

Variables: {NSW, WA, NT, Q, SA, V, T}

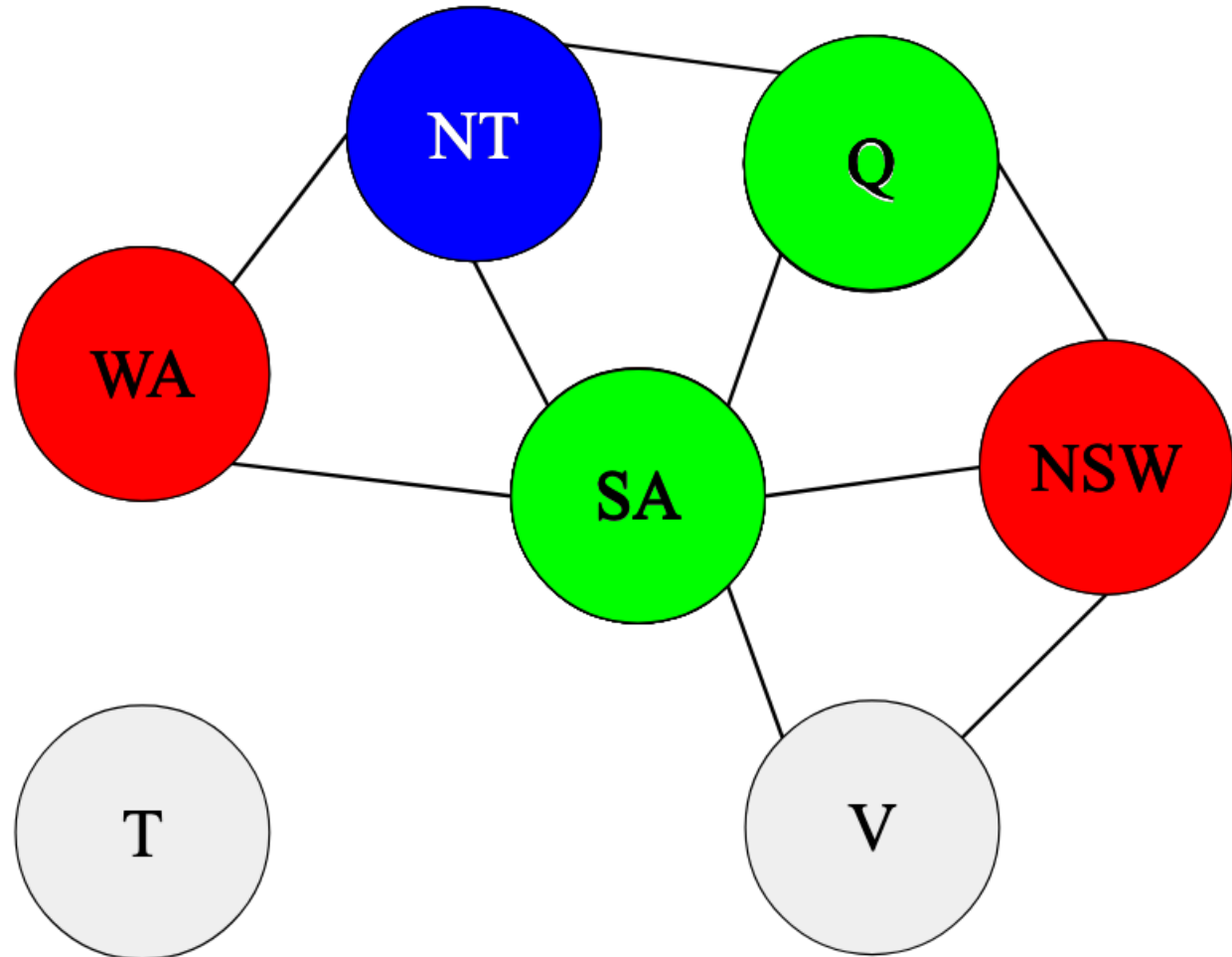
Colors: {red, green, blue}



Backtracking

Variables: {NSW, WA, NT, Q, SA, V, T}

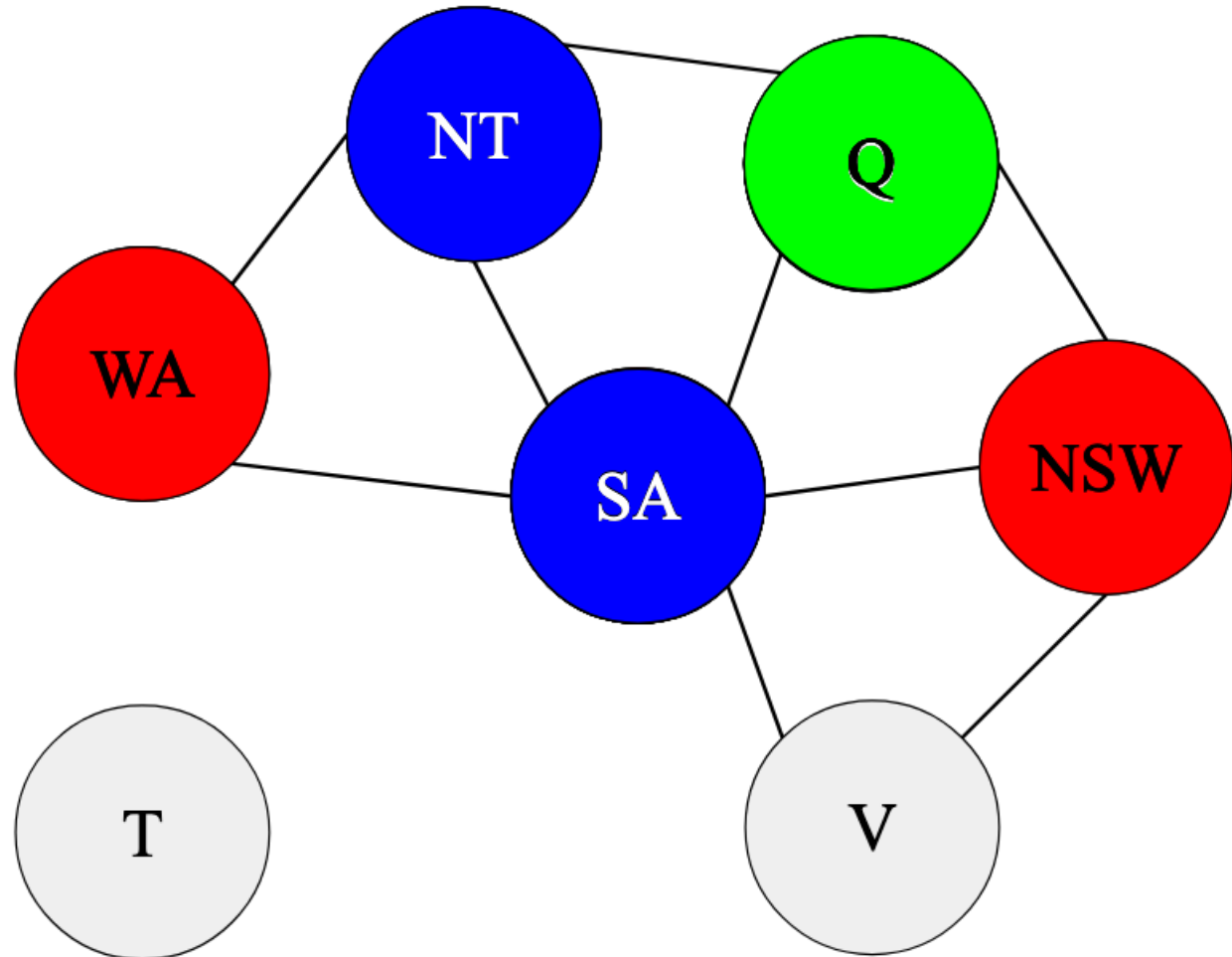
Colors: {red, green, blue}



Backtracking

Variables: {NSW, WA, NT, Q, SA, V, T}

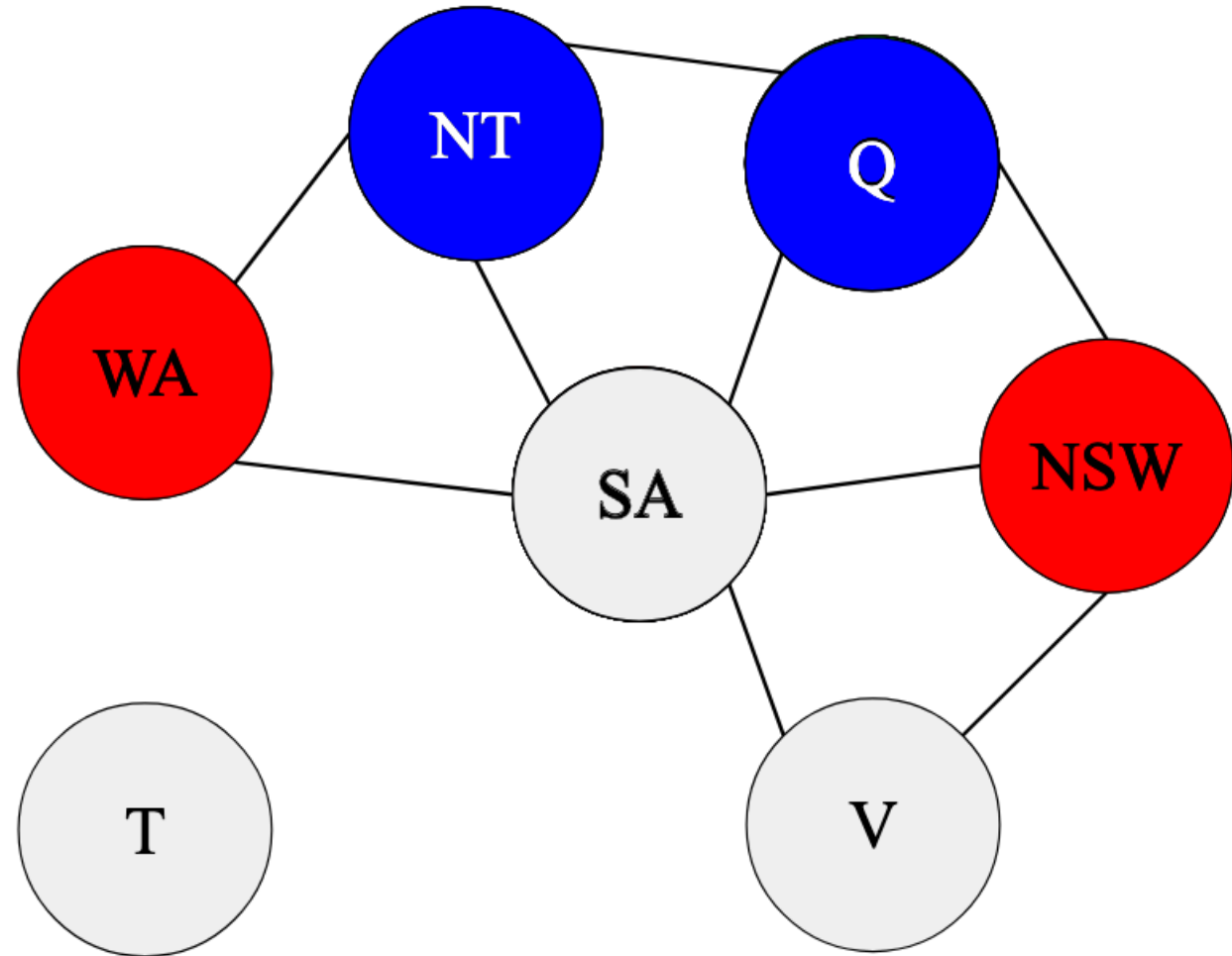
Colors: {red, green, blue}



Backtracking

Variables: {NSW, WA, NT, Q, SA, V, T}

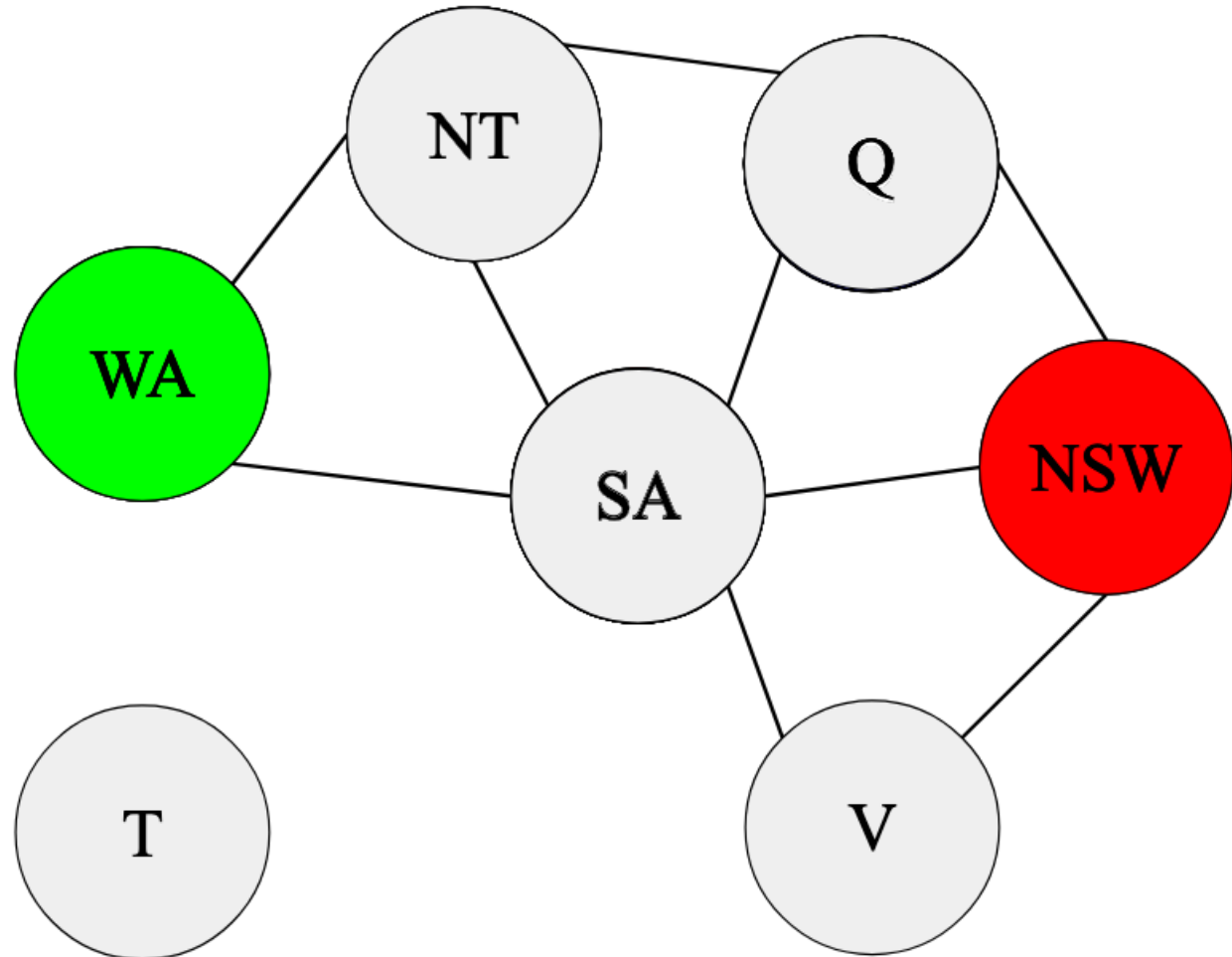
Colors: {red, green, blue}



Backtracking

Variables: {NSW, WA, NT, Q, SA, V, T}

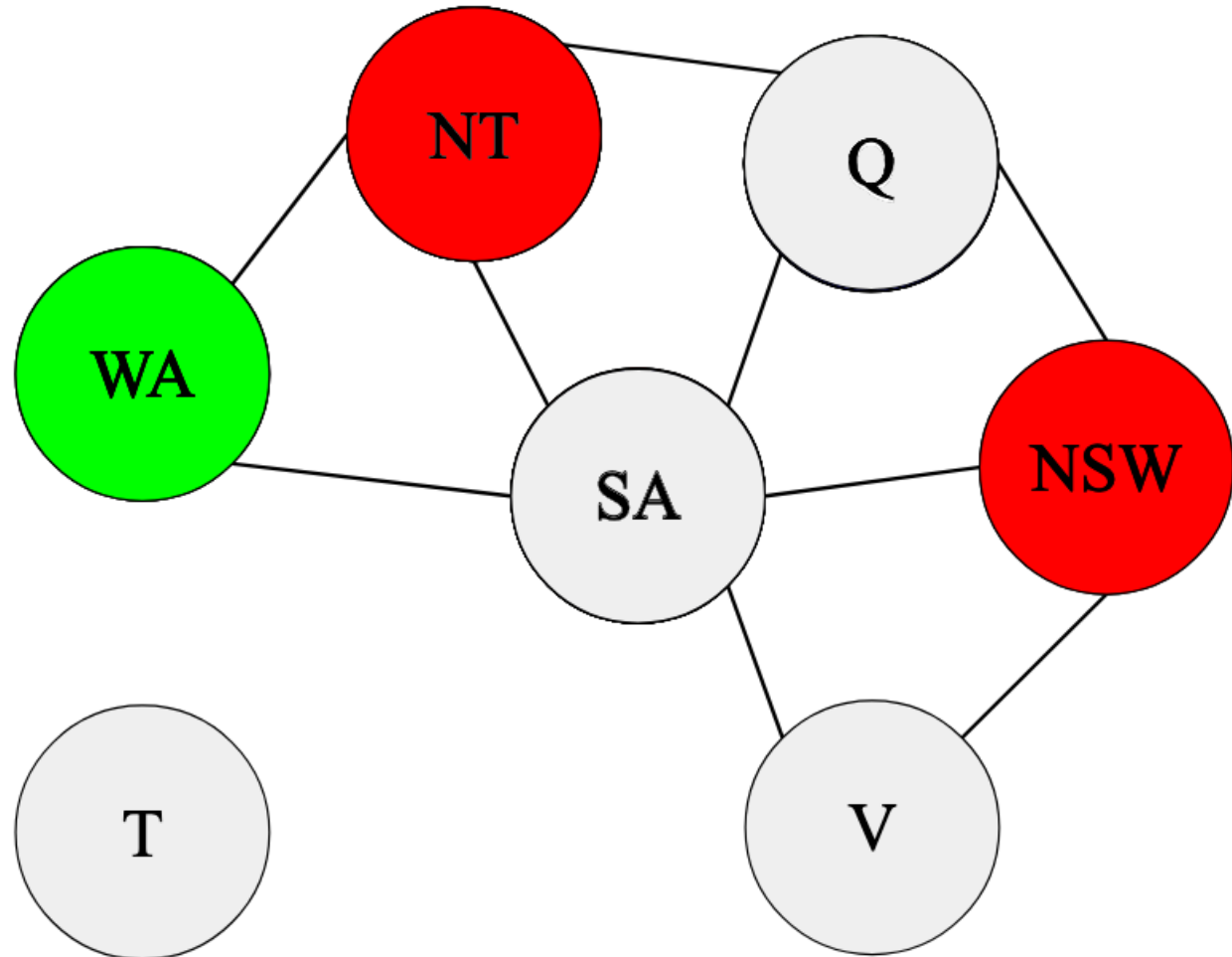
Colors: {red, green, blue}



Backtracking

Variables: {NSW, WA, NT, Q, SA, V, T}

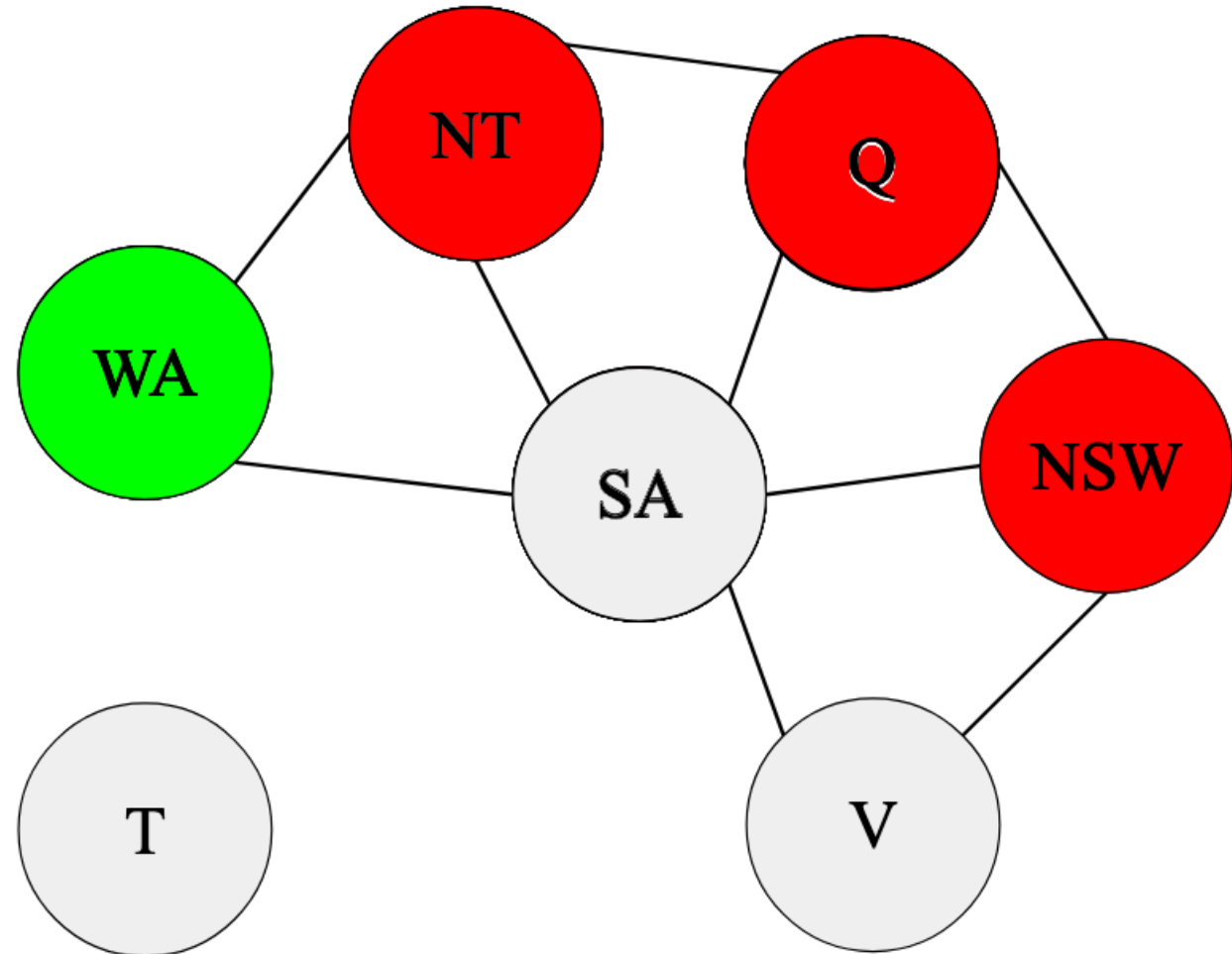
Colors: {red, green, blue}



Backtracking

Variables: {NSW, WA, NT, Q, SA, V, T}

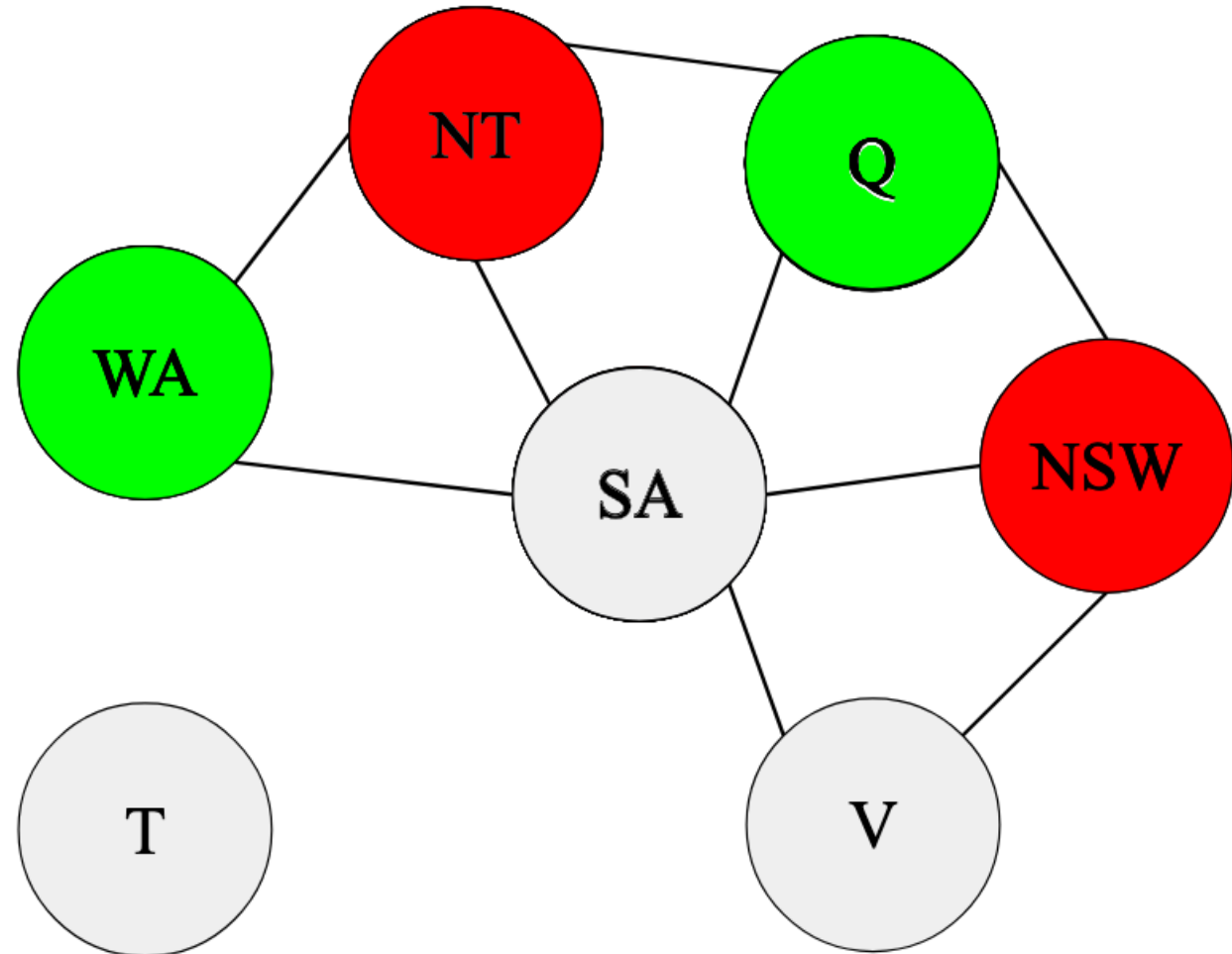
Colors: {red, green, blue}



Backtracking

Variables: {NSW, WA, NT, Q, SA, V, T}

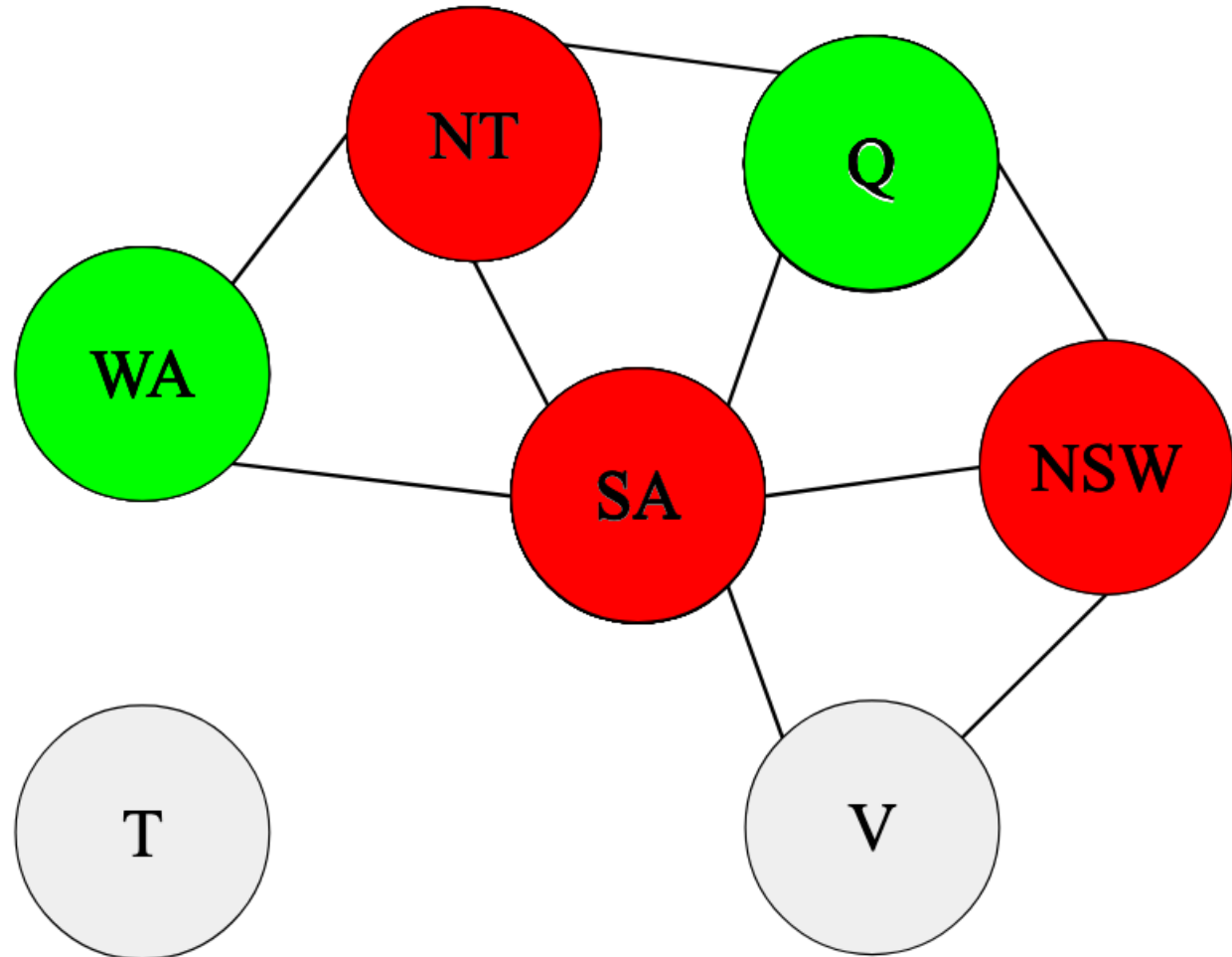
Colors: {red, green, blue}



Backtracking

Variables: {NSW, WA, NT, Q, SA, V, T}

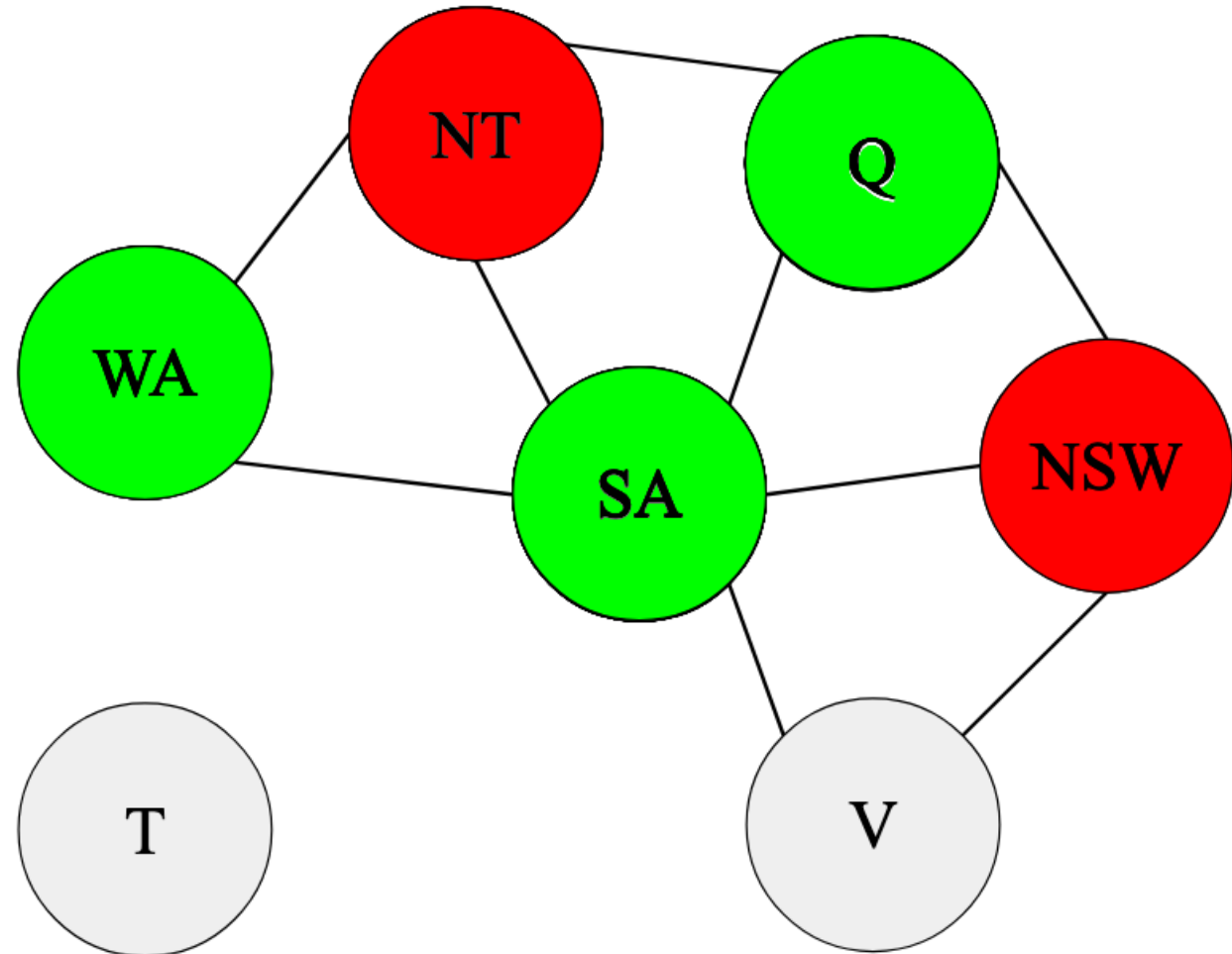
Colors: {red, green, blue}



Backtracking

Variables: {NSW, WA, NT, Q, SA, V, T}

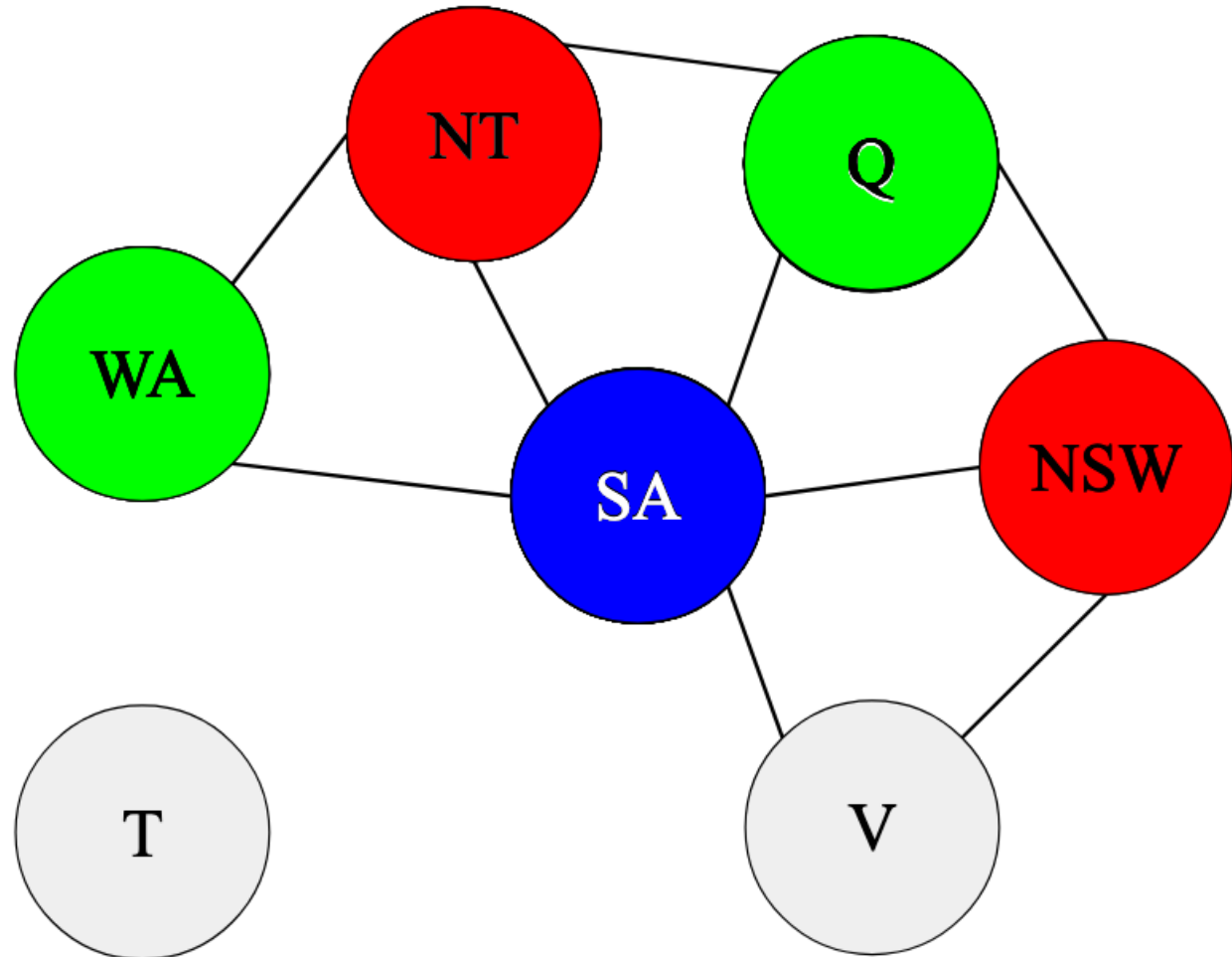
Colors: {red, green, blue}



Backtracking

Variables: {NSW, WA, NT, Q, SA, V, T}

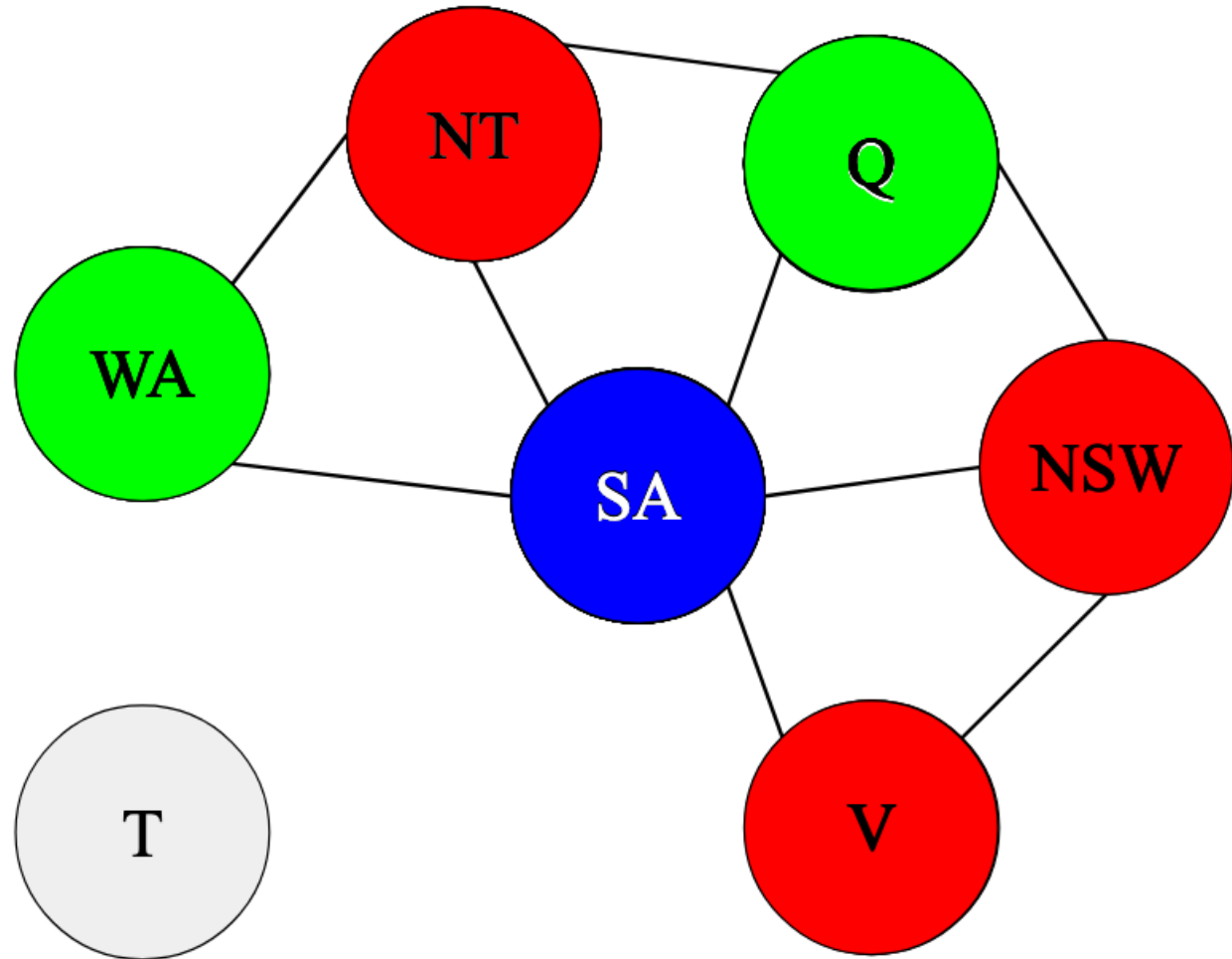
Colors: {red, green, blue}



Backtracking

Variables: {NSW, WA, NT, Q, SA, V, T}

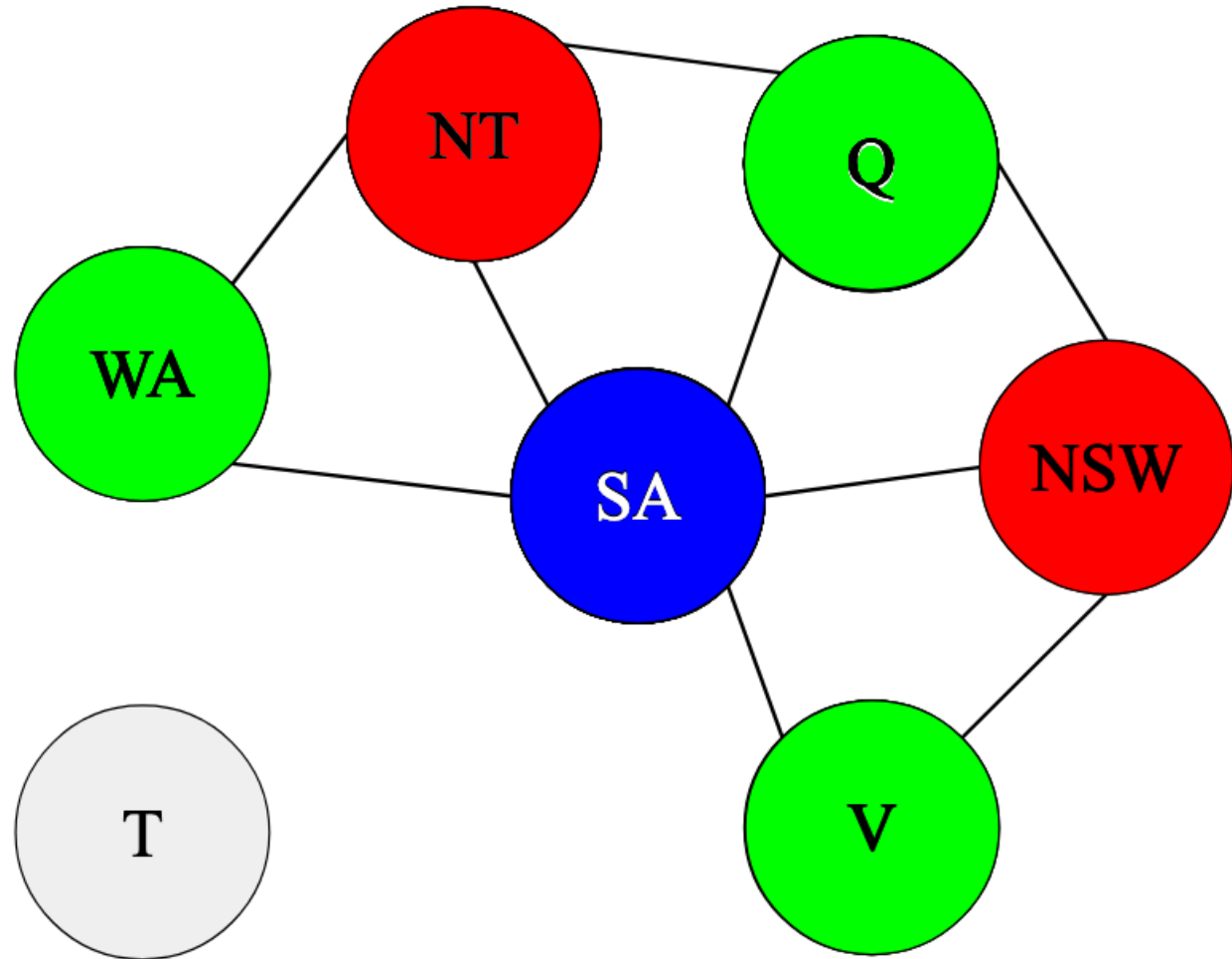
Colors: {red, green, blue}



Backtracking

Variables: {NSW, WA, NT, Q, SA, V, T}

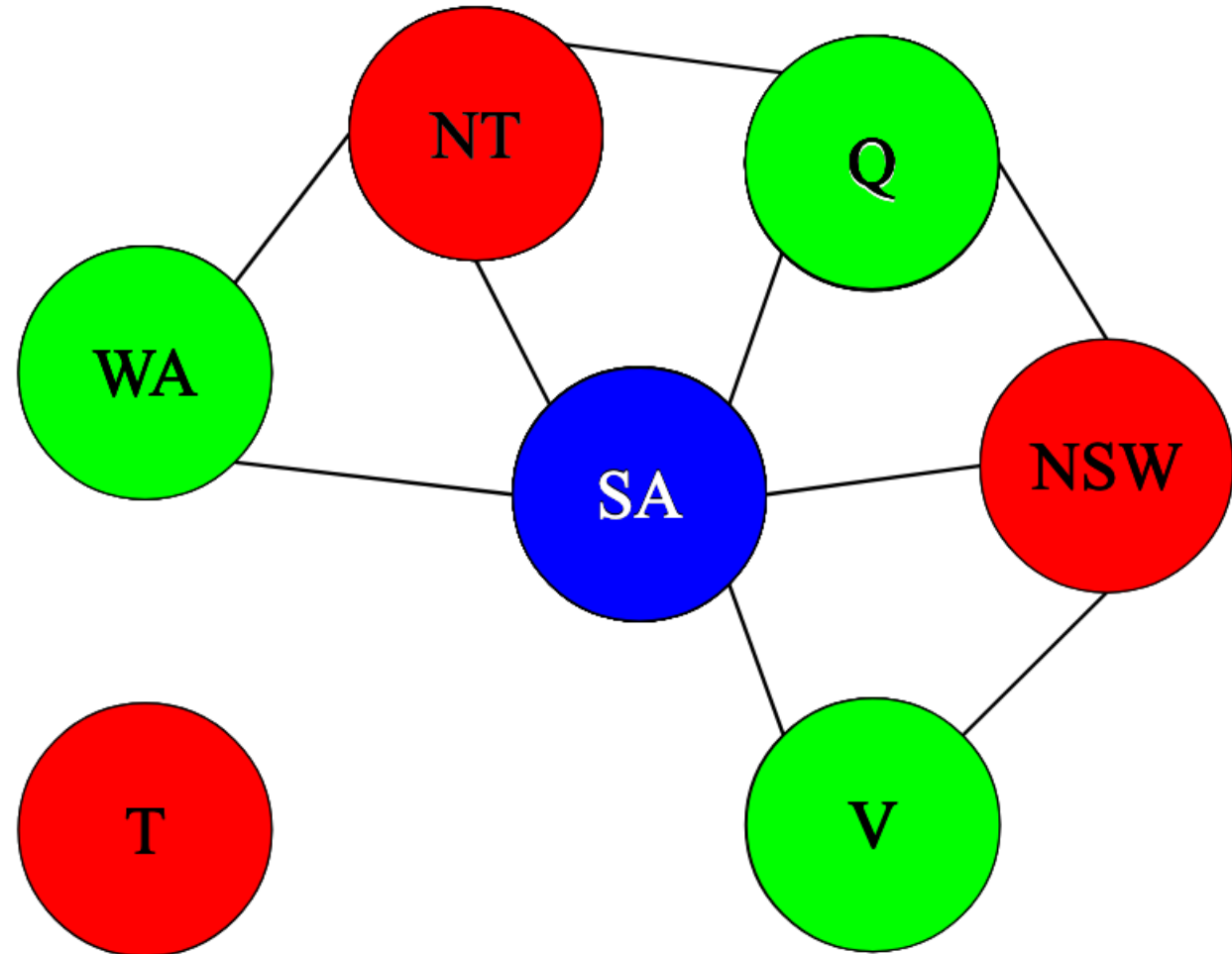
Colors: {red, green, blue}



Backtracking

Variables: {NSW, WA, NT, Q, SA, V, T}

Colors: {red, green, blue}



Backtracking Search

Choose a variable → Try a consistent value → Recurse → Undo

PY main.py

```
1 def solve(domains, neighbours, assignment=None):
2     if assignment is None: assignment = {}
3     if len(assignment) == len(domains): return assignment.copy()
4     x = next(x for x in domains if x not in assignment)
5     for colour in domains[x]:
6         if all(assignment.get(y) != colour for y in neighbours[x]):
7             assignment[x] = colour
8             result = solve(domains, neighbours, assignment)
9             if result is not None: return result
10            del assignment[x]
11    return None
12 neighbours = {
13     'WA': ('NT', 'SA'), 'NT': ('WA', 'SA', 'Q'),
14     'SA': ('WA', 'NT', 'Q', 'NSW', 'V'),
15     'Q': ('NT', 'SA', 'NSW'), 'NSW': ('SA', 'Q', 'V'),
16     'V': ('SA', 'NSW'), 'T': ()}
17 order = ('NSW', 'WA', 'NT', 'Q', 'SA', 'V', 'T')
18 domains = {x: ['red', 'green', 'blue'] for x in order}
```

Improving Backtracking

Can we detect inevitable failure earlier?

Forward checking: remove conflicting values from neighbours

Constraint propagation: repeat these deductions with AC-3

Forward Checking

After each assignment, remove conflicting values from the domains of unassigned neighbours



WA	NT	Q	NSW	V	SA	T
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>

Forward Checking

After each assignment, remove conflicting values from the domains of unassigned neighbours



WA	NT	Q	NSW	V	SA	T
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>

Forward Checking

After each assignment, remove conflicting values from the domains of unassigned neighbours



WA	NT	Q	NSW	V	SA	T
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div></div>						

Forward Checking

After each assignment, remove conflicting values from the domains of unassigned neighbours



WA	NT	Q	NSW	V	SA	T
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div></div>	<div><div></div><div></div></div>					

Forward Checking

After each assignment, remove conflicting values from the domains of unassigned neighbours



WA	NT	Q	NSW	V	SA	T
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div></div>	<div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>				

Forward Checking

After each assignment, remove conflicting values from the domains of unassigned neighbours



WA	NT	Q	NSW	V	SA	T
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div></div>	<div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>			

Forward Checking

After each assignment, remove conflicting values from the domains of unassigned neighbours



WA	NT	Q	NSW	V	SA	T
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div></div>	<div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>		

Forward Checking

After each assignment, remove conflicting values from the domains of unassigned neighbours



WA	NT	Q	NSW	V	SA	T
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div></div>	<div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div></div>	

Forward Checking

After each assignment, remove conflicting values from the domains of unassigned neighbours



WA	NT	Q	NSW	V	SA	T
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>

Forward Checking

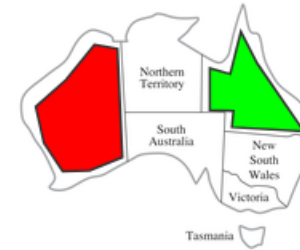
After each assignment, remove conflicting values from the domains of unassigned neighbours



WA	NT	Q	NSW	V	SA	T
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div></div>	<div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>

Forward Checking

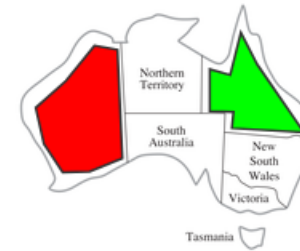
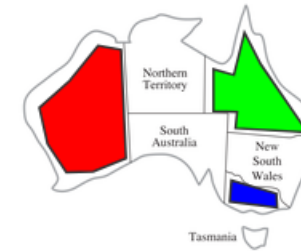
After each assignment, remove conflicting values from the domains of unassigned neighbours



WA	NT	Q	NSW	V	SA	T
<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>
<div><div>Red</div></div>	<div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>
<div><div>Red</div></div>	<div><div>Blue</div></div>	<div><div>Green</div></div>	<div><div>Red</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>

Forward Checking

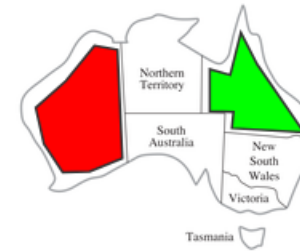
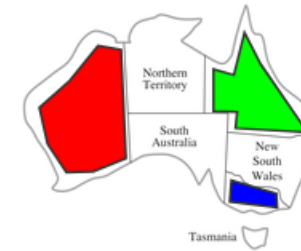
After each assignment, remove conflicting values from the domains of unassigned neighbours



WA	NT	Q	NSW	V	SA	T
<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>
<div><div>Red</div></div>	<div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>
<div><div>Red</div></div>	<div><div>Blue</div></div>	<div><div>Green</div></div>	<div><div>Red</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>

Forward Checking

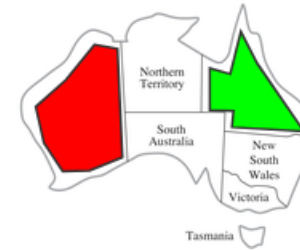
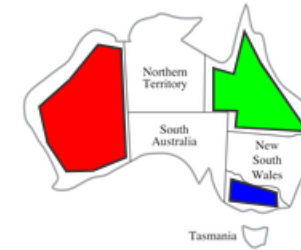
After each assignment, remove conflicting values from the domains of unassigned neighbours



WA	NT	Q	NSW	V	SA	T
<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>
<div><div>Red</div></div>	<div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>
<div><div>Red</div></div>	<div><div>Blue</div></div>	<div><div>Green</div></div>	<div><div>Red</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>
<div><div>Red</div></div>	<div><div>Blue</div></div>	<div><div>Green</div></div>	<div><div>Red</div></div>	<div><div>Blue</div></div>		<div><div>Red</div><div>Green</div><div>Blue</div></div>

Forward Checking

After each assignment, remove conflicting values from the domains of unassigned neighbours



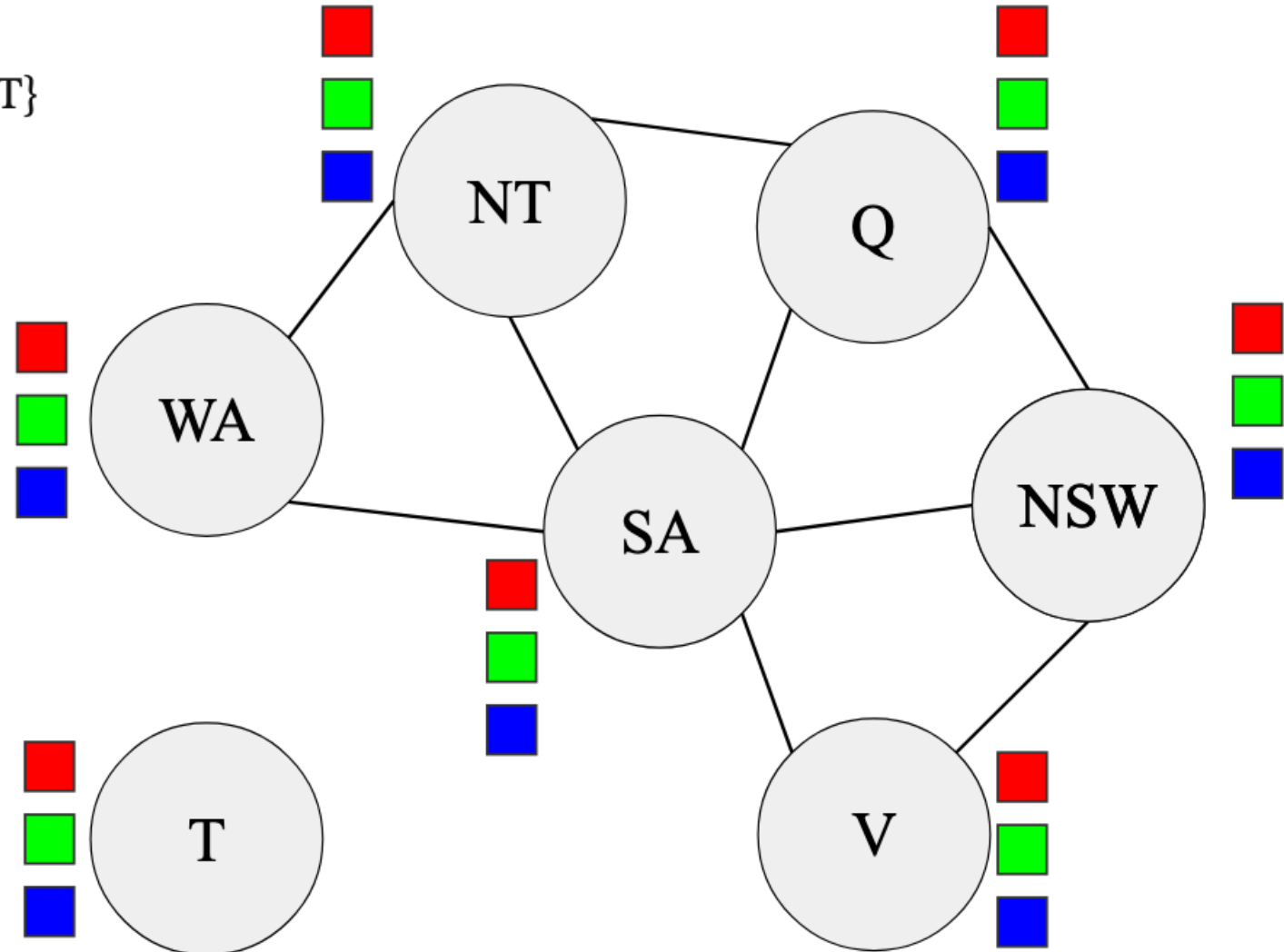
Detect failure
earlier

WA	NT	Q	NSW	V	SA	T
<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>
<div><div>Red</div></div>	<div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>
<div><div>Red</div></div>	<div><div>Blue</div></div>	<div><div>Green</div></div>	<div><div>Red</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>
<div><div>Red</div></div>	<div><div>Blue</div></div>	<div><div>Green</div></div>	<div><div>Red</div></div>	<div><div>Blue</div></div>		<div><div>Red</div><div>Green</div><div>Blue</div></div>

Backtracking with Forward Checking

Variables: {NSW, WA, NT, Q, SA, V, T}

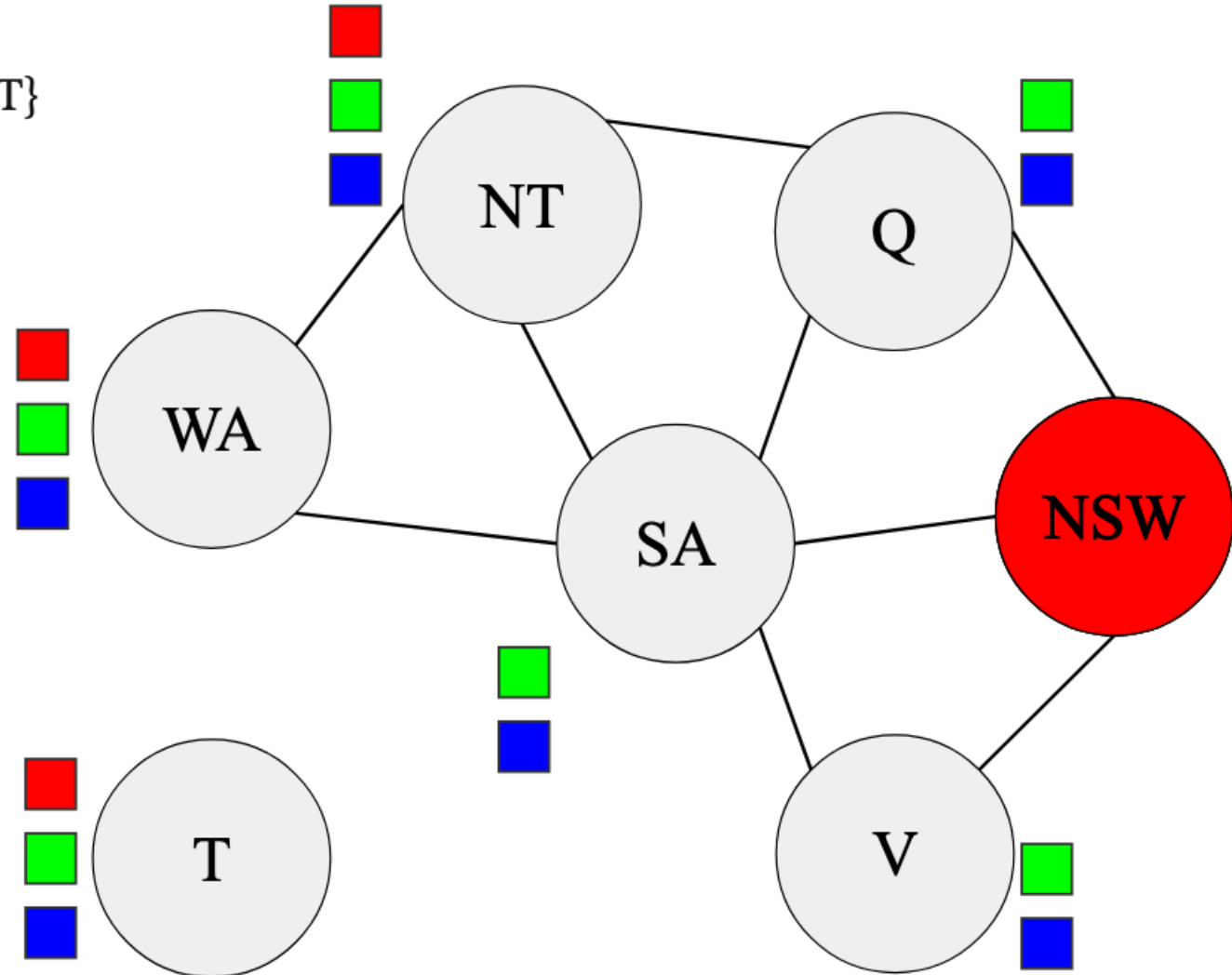
Colors: {red, green, blue}



Backtracking with Forward Checking

Variables: {NSW, WA, NT, Q, SA, V, T}

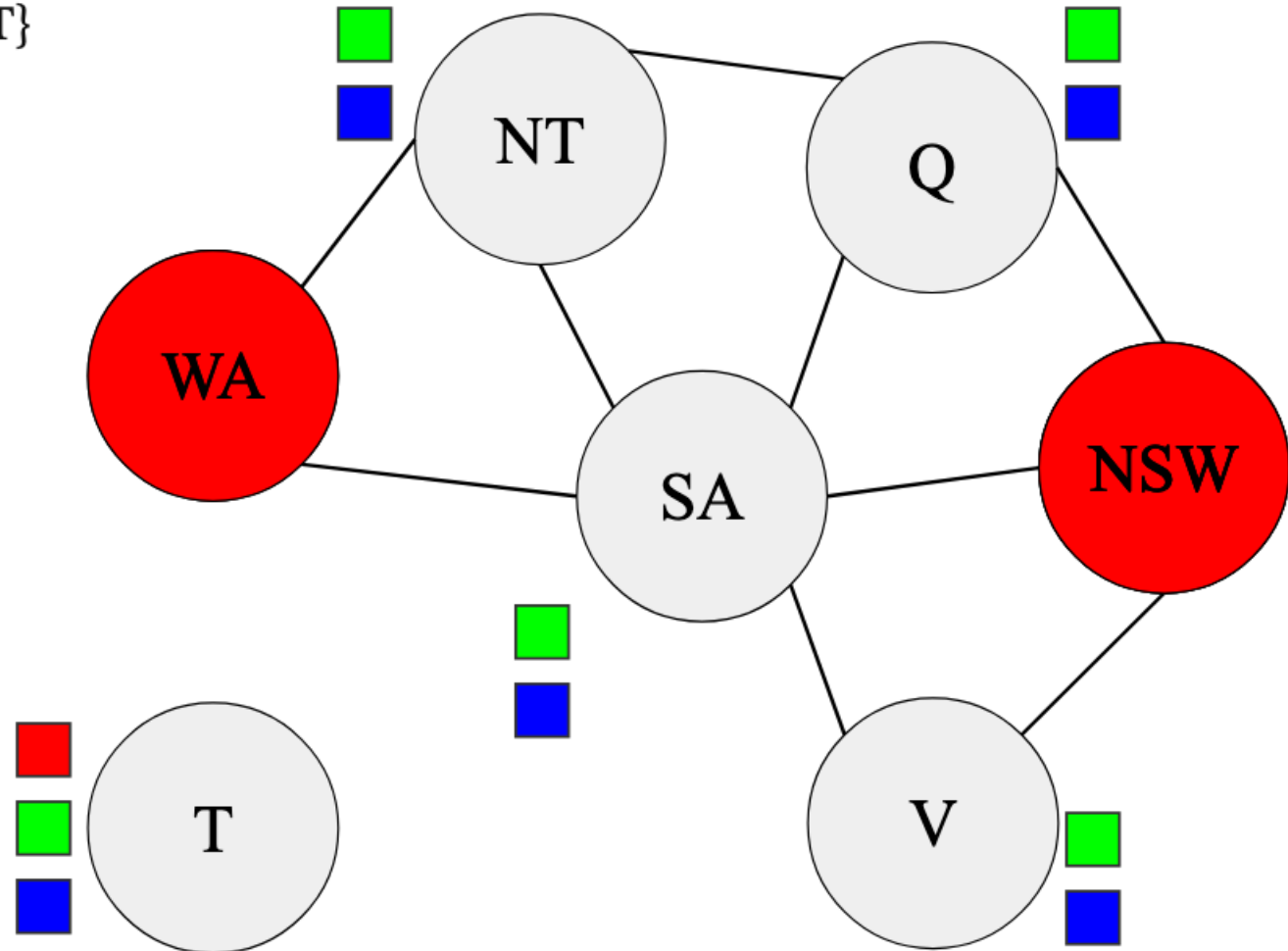
Colors: {red, green, blue}



Backtracking with Forward Checking

Variables: {NSW, WA, NT, Q, SA, V, T}

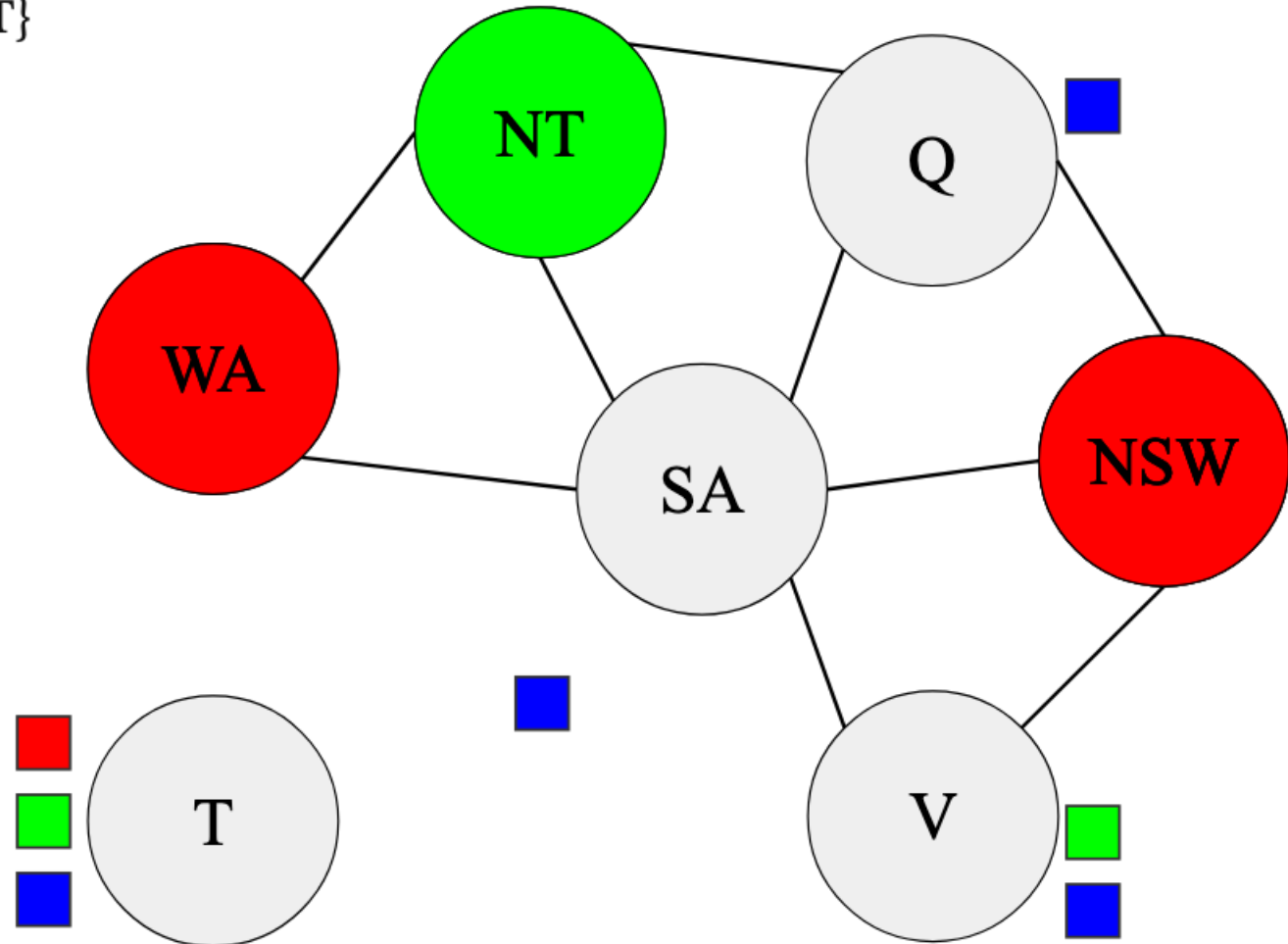
Colors: {red, green, blue}



Backtracking with Forward Checking

Variables: {NSW, WA, NT, Q, SA, V, T}

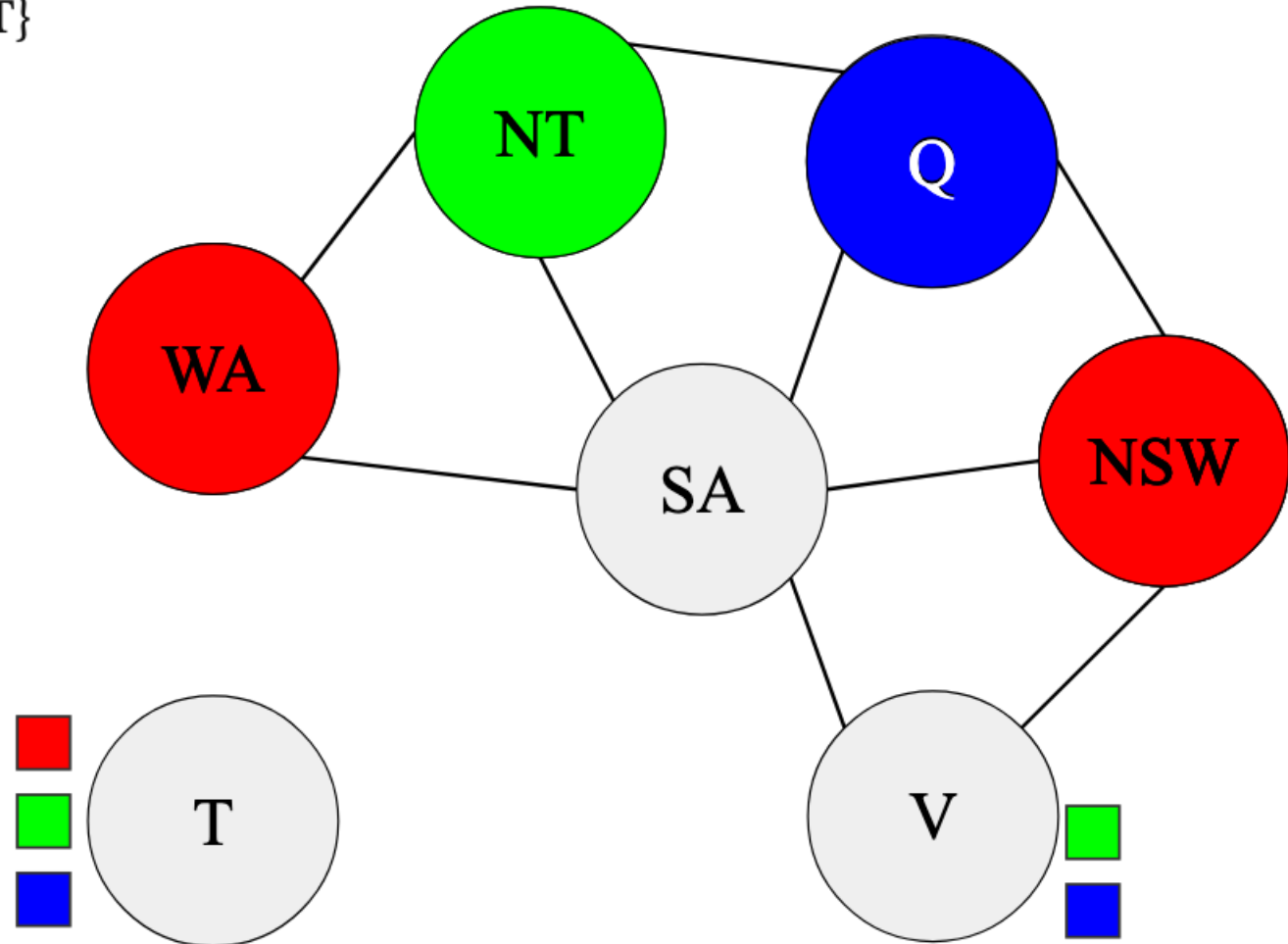
Colors: {red, green, blue}



Backtracking with Forward Checking

Variables: {NSW, WA, NT, Q, SA, V, T}

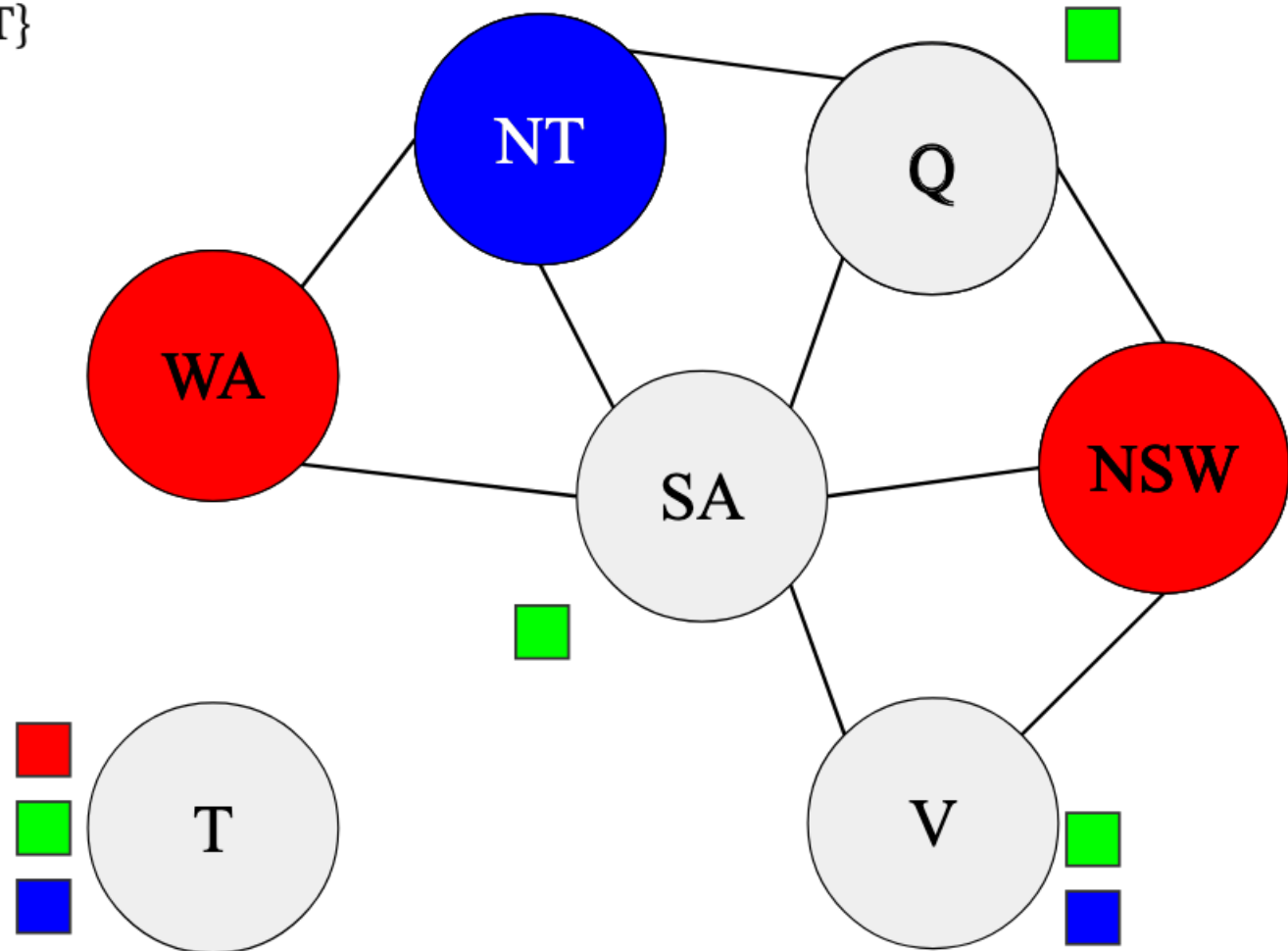
Colors: {red, green, blue}



Backtracking with Forward Checking

Variables: {NSW, WA, NT, Q, SA, V, T}

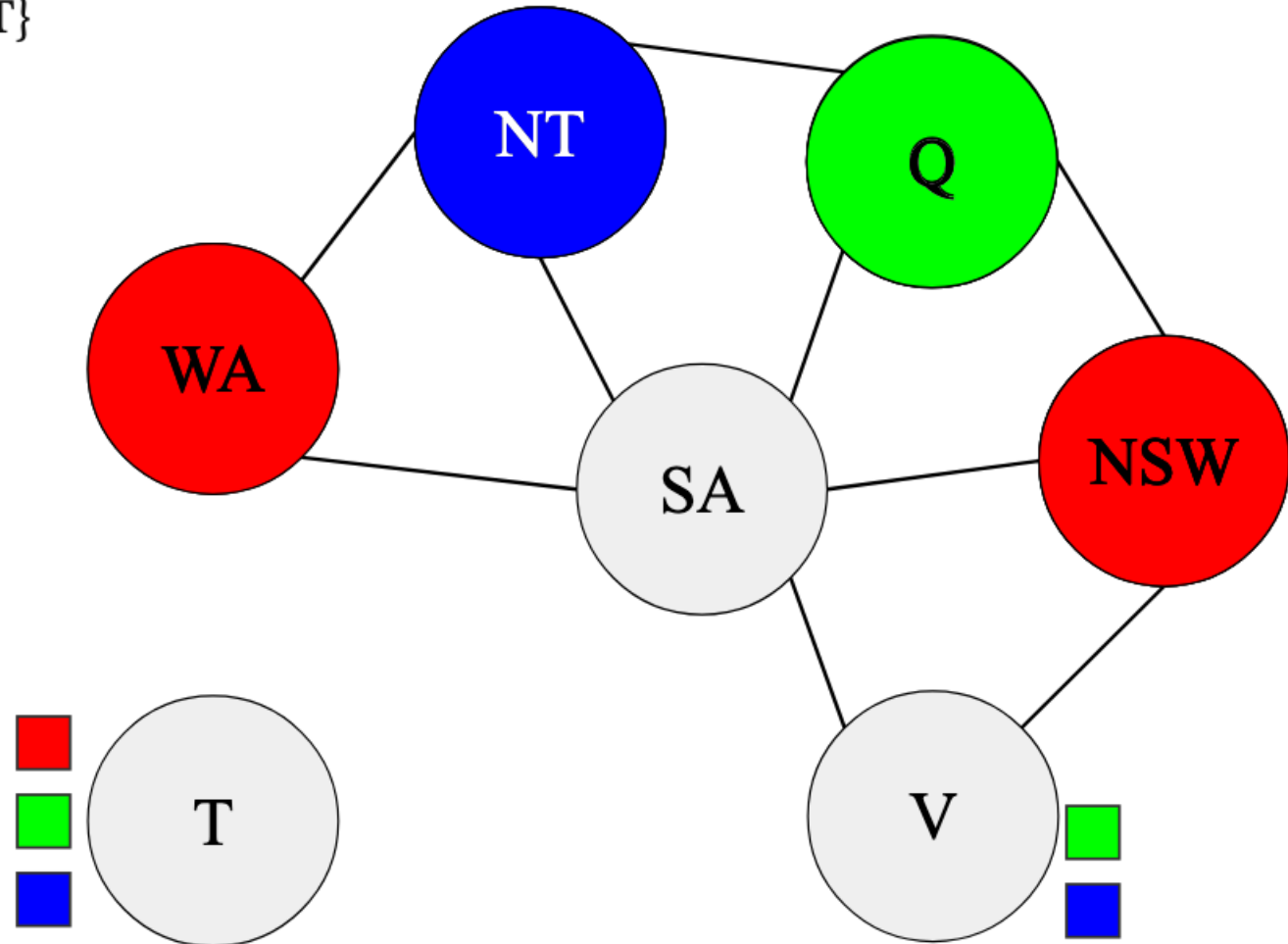
Colors: {red, green, blue}



Backtracking with Forward Checking

Variables: {NSW, WA, NT, Q, SA, V, T}

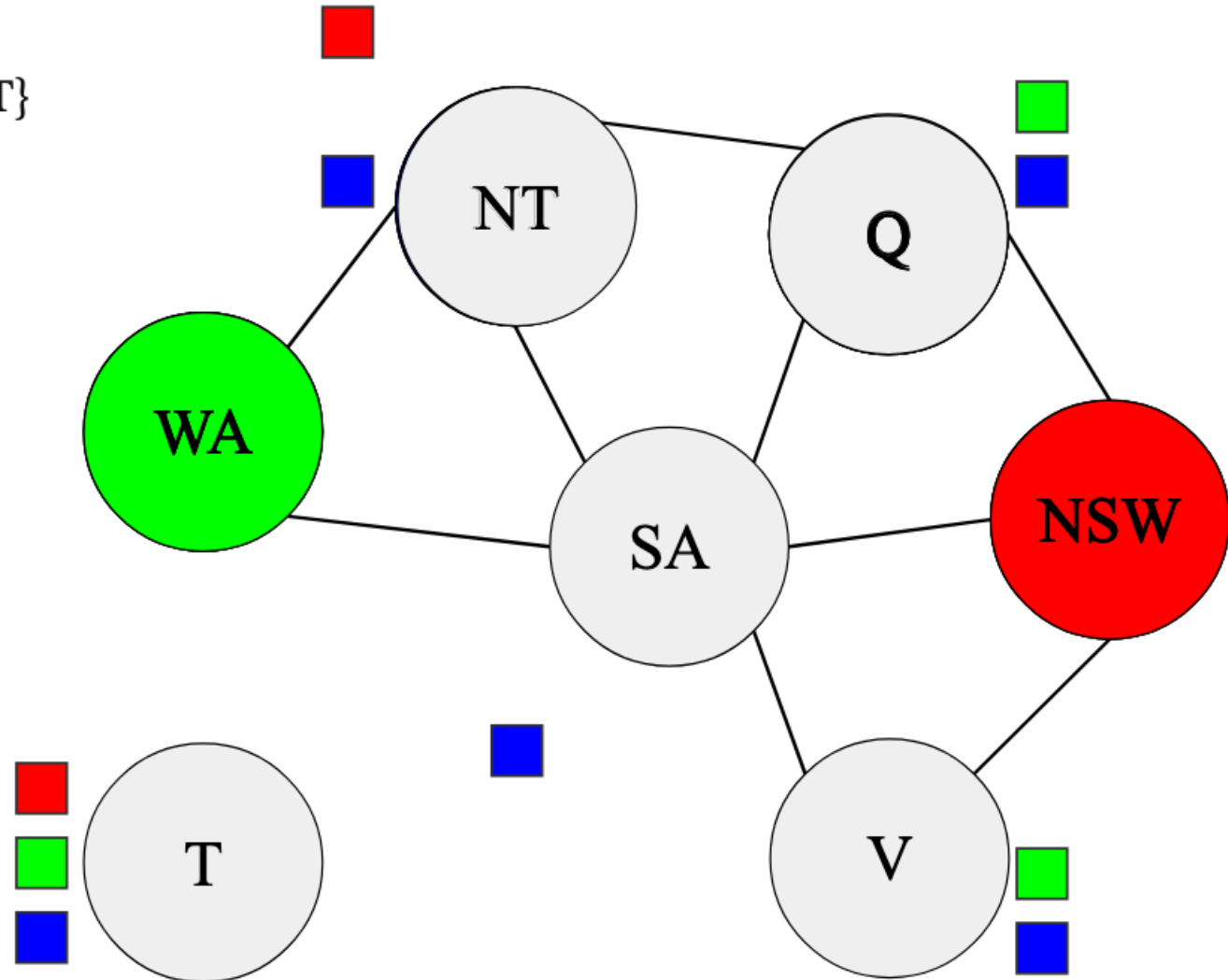
Colors: {red, green, blue}



Backtracking with Forward Checking

Variables: {NSW, WA, NT, Q, SA, V, T}

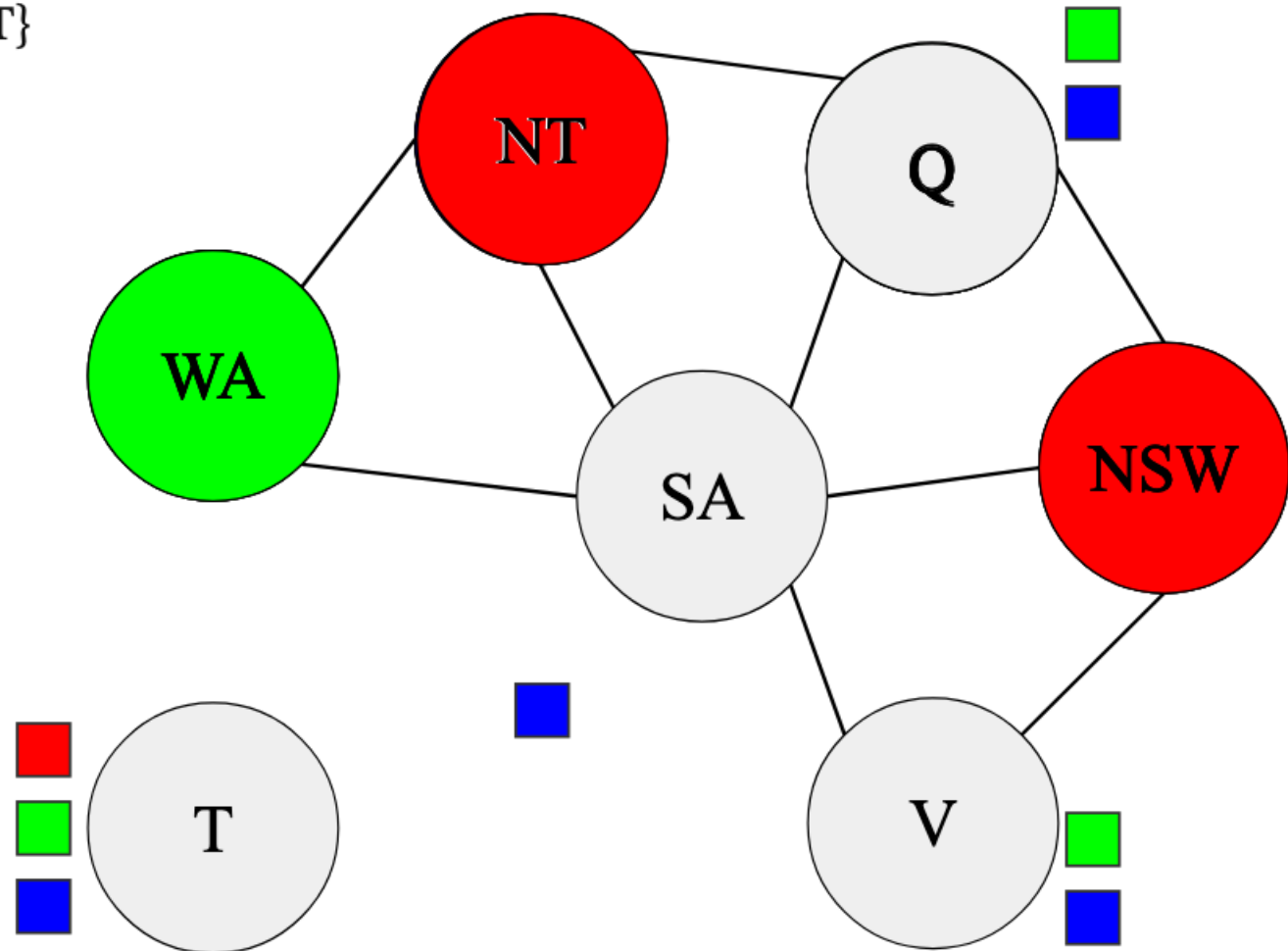
Colors: {red, green, blue}



Backtracking with Forward Checking

Variables: {NSW, WA, NT, Q, SA, V, T}

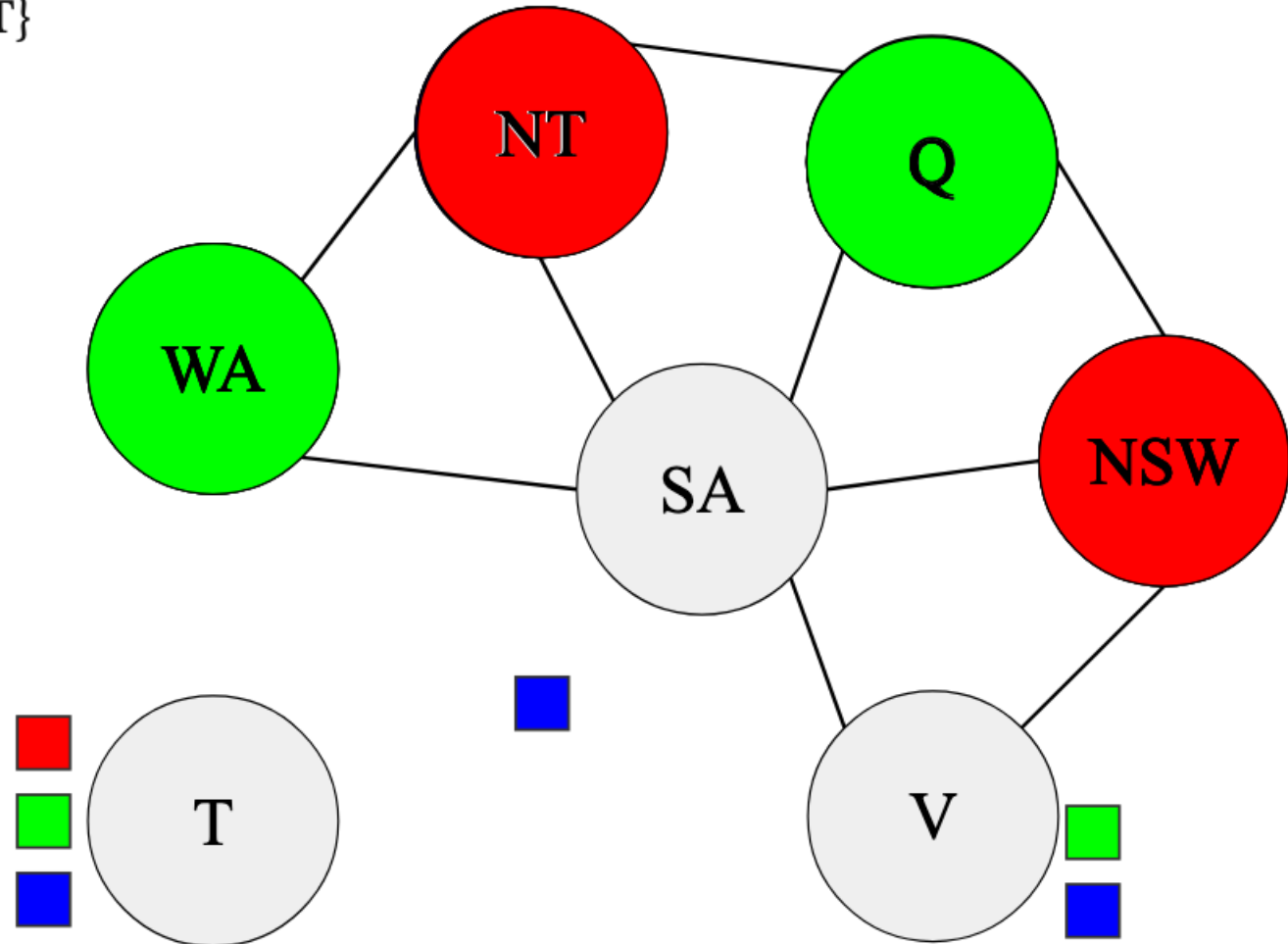
Colors: {red, green, blue}



Backtracking with Forward Checking

Variables: {NSW, WA, NT, Q, SA, V, T}

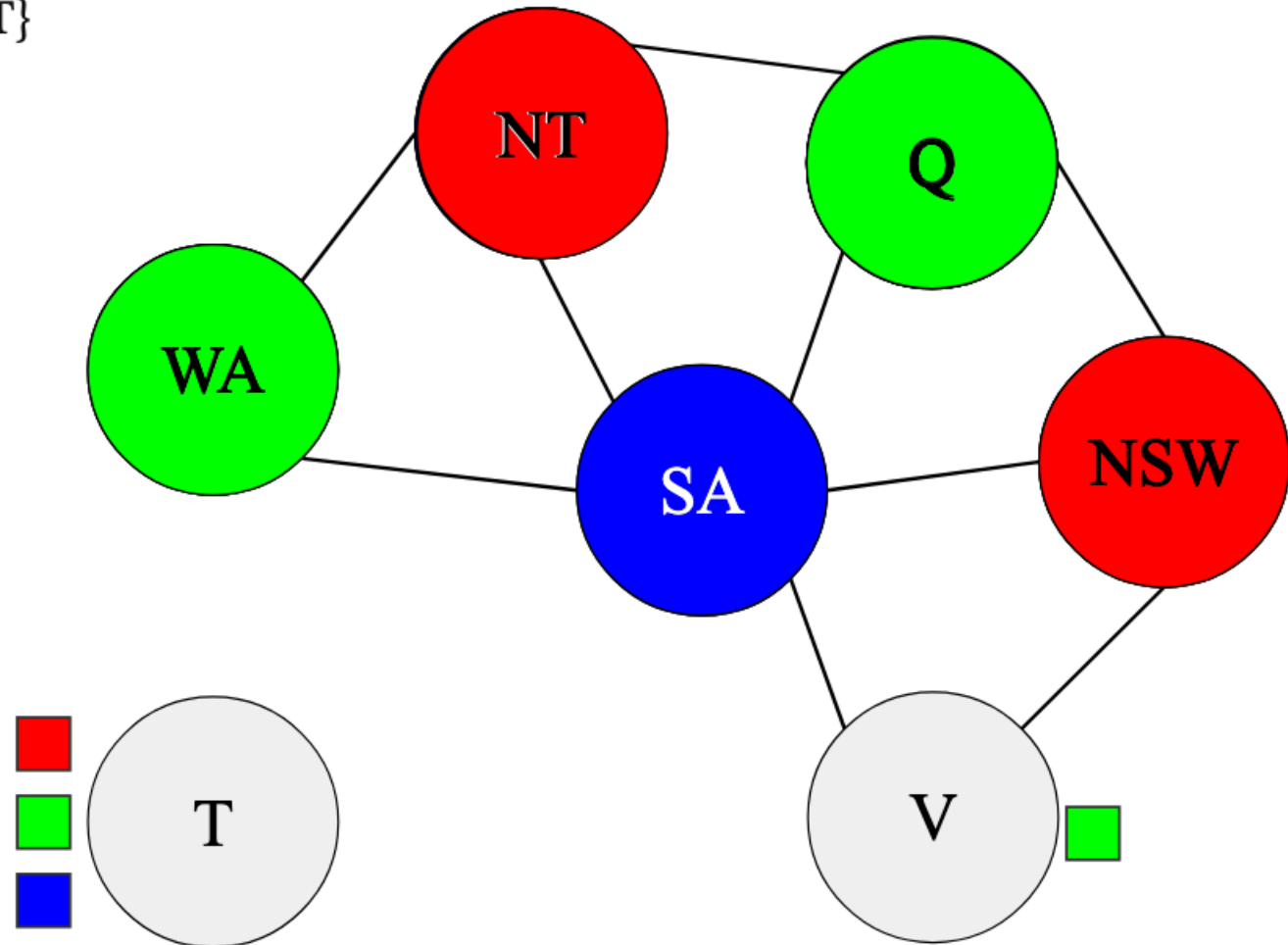
Colors: {red, green, blue}



Backtracking with Forward Checking

Variables: {NSW, WA, NT, Q, SA, V, T}

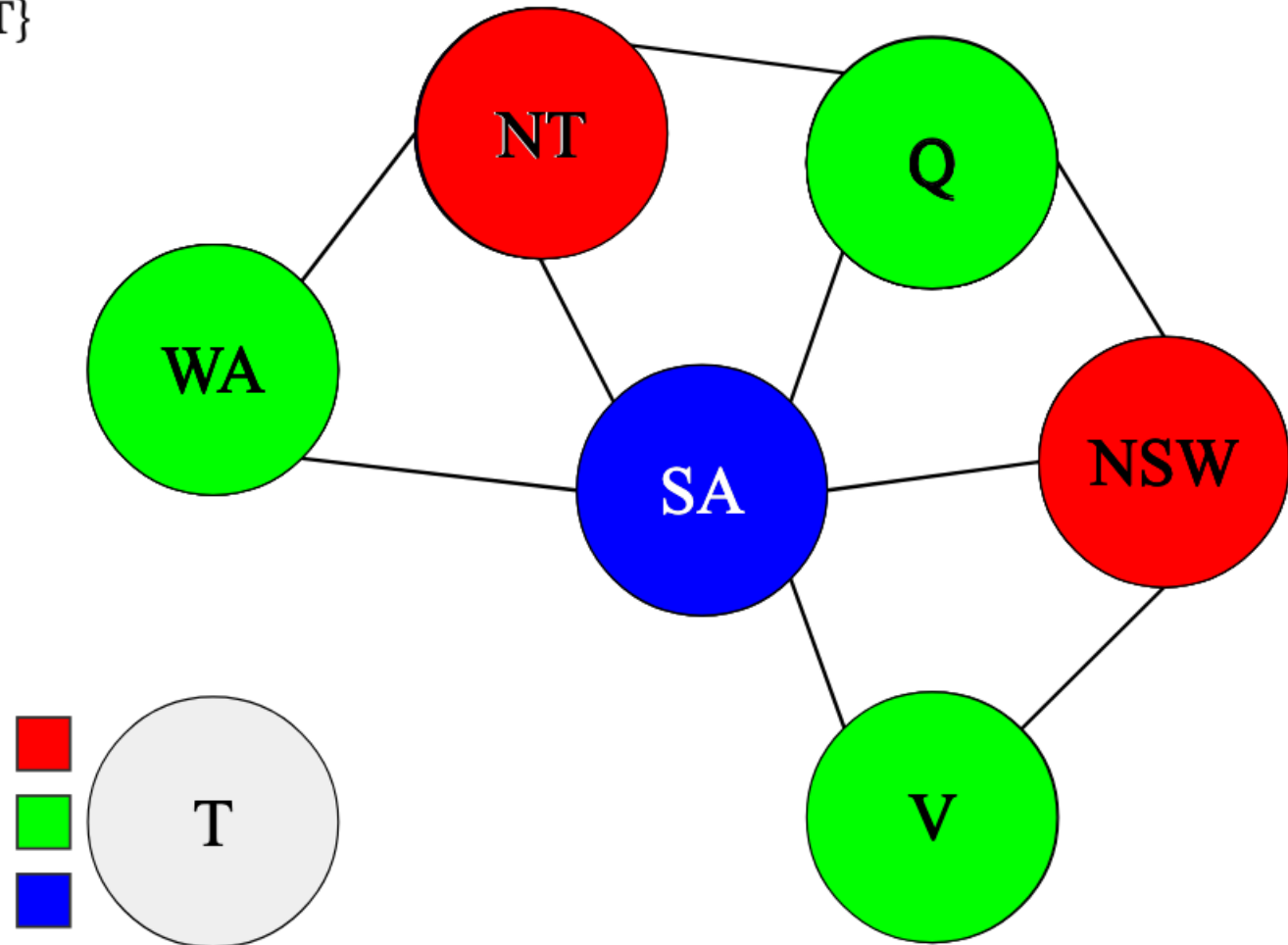
Colors: {red, green, blue}



Backtracking with Forward Checking

Variables: {NSW, WA, NT, Q, SA, V, T}

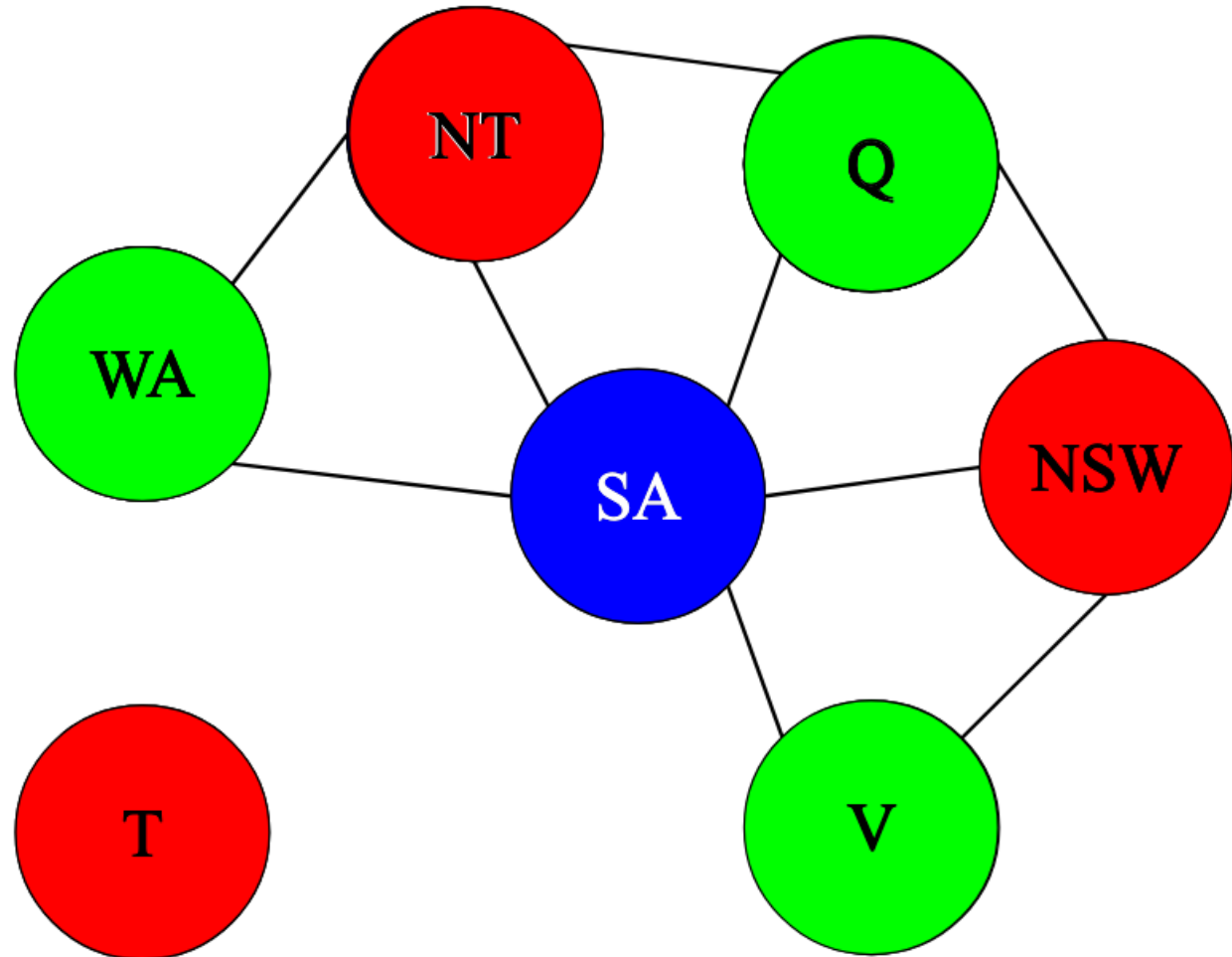
Colors: {red, green, blue}



Backtracking with Forward Checking

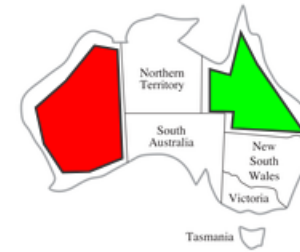
Variables: {NSW, WA, NT, Q, SA, V, T}

Colors: {red, green, blue}



When Forward Checking Misses Failure

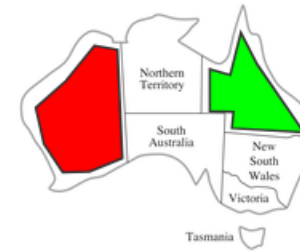
After each assignment, remove conflicting values from the domains of unassigned neighbours



WA	NT	Q	NSW	V	SA	T
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>

When Forward Checking Misses Failure

After each assignment, remove conflicting values from the domains of unassigned neighbours

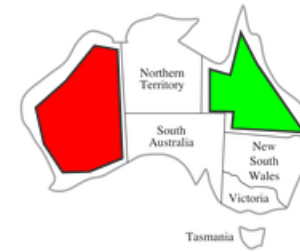


Doomed to fail

WA	NT	Q	NSW	V	SA	T
<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>
<div><div>Red</div></div>	<div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>
<div><div>Red</div></div>	<div><div>Blue</div></div>	<div><div>Green</div></div>	<div><div>Red</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>

When Forward Checking Misses Failure

After each assignment, remove conflicting values from the domains of unassigned neighbours



Doomed to fail

Can we detect this earlier?

WA	NT	Q	NSW	V	SA	T
<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>
<div><div>Red</div></div>	<div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Green</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>
<div><div>Red</div></div>	<div><div>Blue</div></div>	<div><div>Green</div></div>	<div><div>Red</div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>	<div><div>Blue</div></div>	<div><div>Red</div><div>Green</div><div>Blue</div></div>

Consistency of an Arc

An arc $X \rightarrow Y$ is consistent if every $x \in D_X$ has a compatible $y \in D_Y$

$$\forall x \in D_X, \exists y \in D_Y: C_{XY}(x, y)$$

WA	NT	Q	NSW	V	SA	T
	  	  	  	  	  	  

Consistency of an Arc

An arc $X \rightarrow Y$ is consistent if every $x \in D_X$ has a compatible $y \in D_Y$

$$\forall x \in D_X, \exists y \in D_Y: C_{XY}(x, y)$$

WA	NT	Q	NSW	V	SA	T
	  	  	  	  	  	  



Consistency of an Arc

An arc $X \rightarrow Y$ is consistent if every $x \in D_X$ has a compatible $y \in D_Y$

$$\forall x \in D_X, \exists y \in D_Y: C_{XY}(x, y)$$

WA	NT	Q	NSW	V	SA	T
	 	  	  	  	  	  



Constraint Propagation



WA	NT	Q	NSW	V	SA	T
			 	  		  

$X \rightarrow Y$: every value in D_X needs a compatible value in D_Y

Constraint Propagation



WA	NT	Q	NSW	V	SA	T
			 	  		  

$X \rightarrow Y$: every value in D_X needs a compatible value in D_Y

Constraint Propagation



WA	NT	Q	NSW	V	SA	T
			 	  		  

$X \rightarrow Y$: every value in D_X needs a compatible value in D_Y

Constraint Propagation



WA	NT	Q	NSW	V	SA	T
			 	  		  

$X \rightarrow Y$: every value in D_X needs a compatible value in D_Y

Constraint Propagation



WA	NT	Q	NSW	V	SA	T
			 	  		  

$X \rightarrow Y$: every value in D_X needs a compatible value in D_Y

Constraint Propagation



WA	NT	Q	NSW	V	SA	T
			 	  		  

$X \rightarrow Y$: every value in D_X needs a compatible value in D_Y

Constraint Propagation



WA	NT	Q	NSW	V	SA	T
				  		  

$X \rightarrow Y$: every value in D_X needs a compatible value in D_Y

Constraint Propagation



WA	NT	Q	NSW	V	SA	T
				  		  

$X \rightarrow Y$: every value in D_X needs a compatible value in D_Y

Constraint Propagation



WA	NT	Q	NSW	V	SA	T
				  		  

$X \rightarrow Y$: every value in D_X needs a compatible value in D_Y

Constraint Propagation



WA	NT	Q	NSW	V	SA	T
				 		  

$X \rightarrow Y$: every value in D_X needs a compatible value in D_Y

Constraint Propagation



WA	NT	Q	NSW	V	SA	T
				 		  

$X \rightarrow Y$: every value in D_X needs a compatible value in D_Y

Constraint Propagation



WA	NT	Q	NSW	V	SA	T
				 		  

$X \rightarrow Y$: every value in D_X needs a compatible value in D_Y

Constraint Propagation



WA	NT	Q	NSW	V	SA	T
				 		  

$X \rightarrow Y$: every value in D_X needs a compatible value in D_Y

Constraint Propagation



WA	NT	Q	NSW	V	SA	T
				 		  

$X \rightarrow Y$: every value in D_X needs a compatible value in D_Y

Constraint Propagation

When a domain shrinks, recheck arcs pointing into it
Repeat until no domain changes, or a domain is empty



WA	NT	Q	NSW	V	SA	T
				 		  

$X \rightarrow Y$: every value in D_X needs a compatible value in D_Y

AC-3

Revise an arc → Revisit affected incoming arcs

PY main.py

```
1 from collections import deque
2 def ac3(domains, neighbours):
3     if any(not values for values in domains.values()): return False
4     queue = deque((x, y) for x in neighbours for y in neighbours[x])
5     while queue:
6         x, y = queue.popleft()
7         if revise(domains, x, y):
8             if not domains[x]: return False
9             queue.extend((z, x) for z in neighbours[x] if z != y)
10    return True
11 def revise(domains, x, y):
12     kept = [a for a in domains[x]
13             if any(a != b for b in domains[y])]
14     changed = kept != domains[x]
15     domains[x] = kept
16     return changed
17 neighbours = {
18     'WA': ('NT', 'SA'), 'NT': ('WA', 'SA', 'Q'),
```

AC-3

Why exclude
the reverse arc?

Revise an arc → Revisit affected incoming arcs

PY main.py

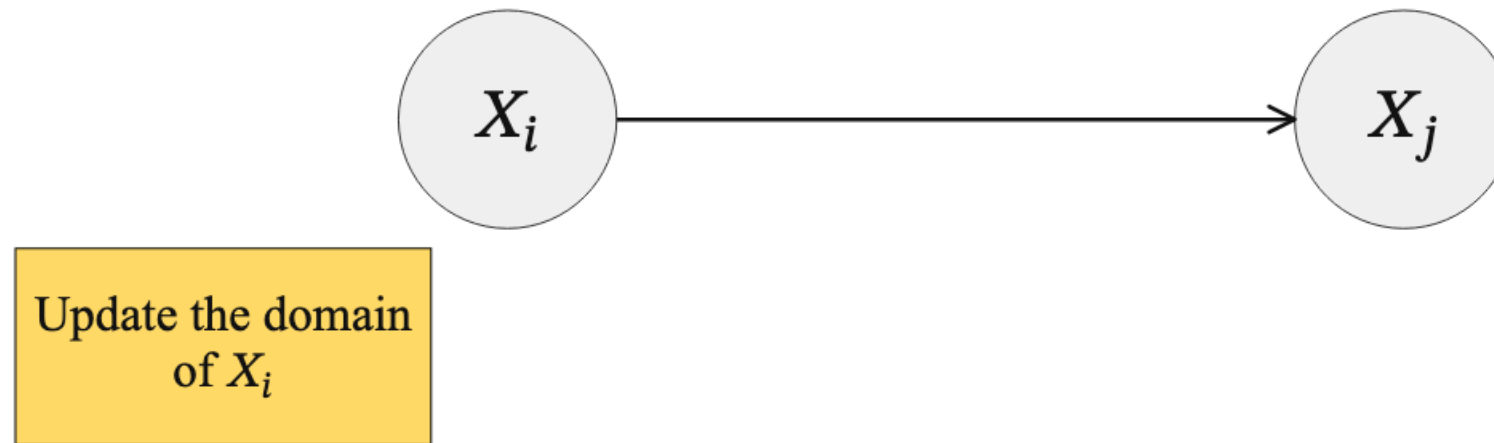
```
1 from collections import deque
2 def ac3(domains, neighbours):
3     if any(not values for values in domains.values()): return False
4     queue = deque((x, y) for x in neighbours for y in neighbours[x])
5     while queue:
6         x, y = queue.popleft()
7         if revise(domains, x, y):
8             if not domains[x]: return False
9             queue.extend((z, x) for z in neighbours[x] if z != y)
10    return True
11 def revise(domains, x, y):
12     kept = [a for a in domains[x]
13             if any(a != b for b in domains[y])]
14     changed = kept != domains[x]
15     domains[x] = kept
16     return changed
17 neighbours = {
18     'WA': ('NT', 'SA'), 'NT': ('WA', 'SA', 'Q'),
```

Why Exclude the Reverse Arc?

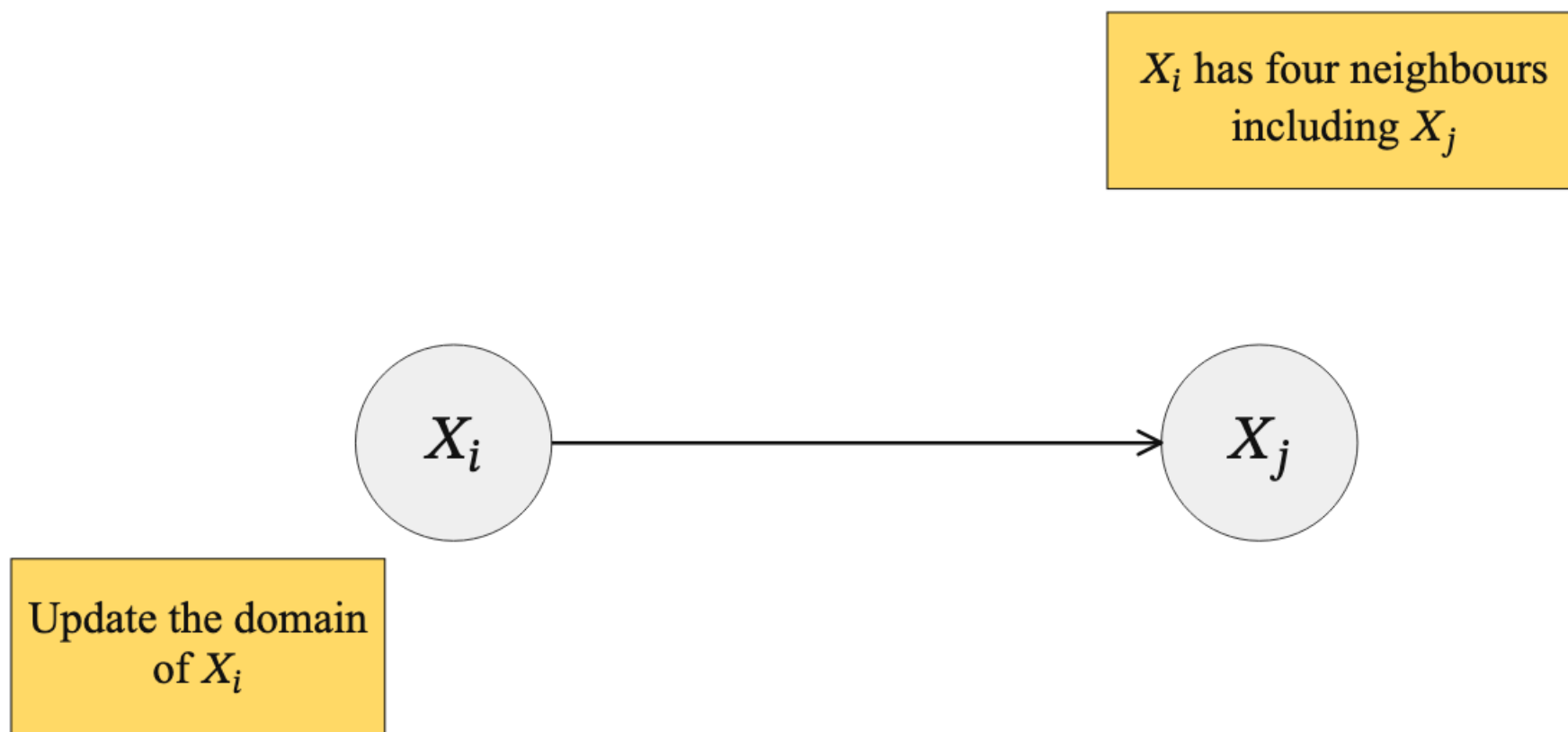
Why Exclude the Reverse Arc?



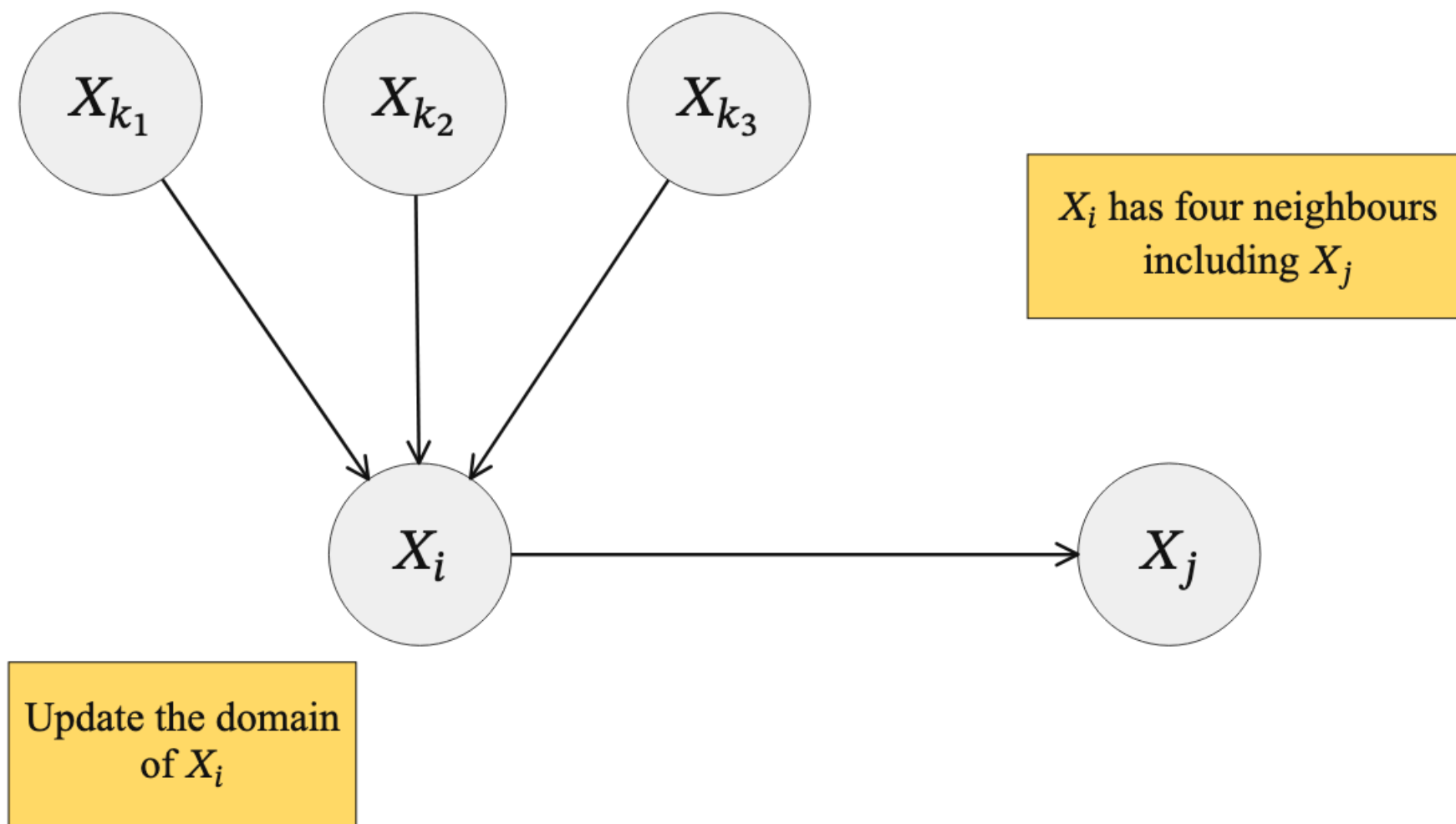
Why Exclude the Reverse Arc?



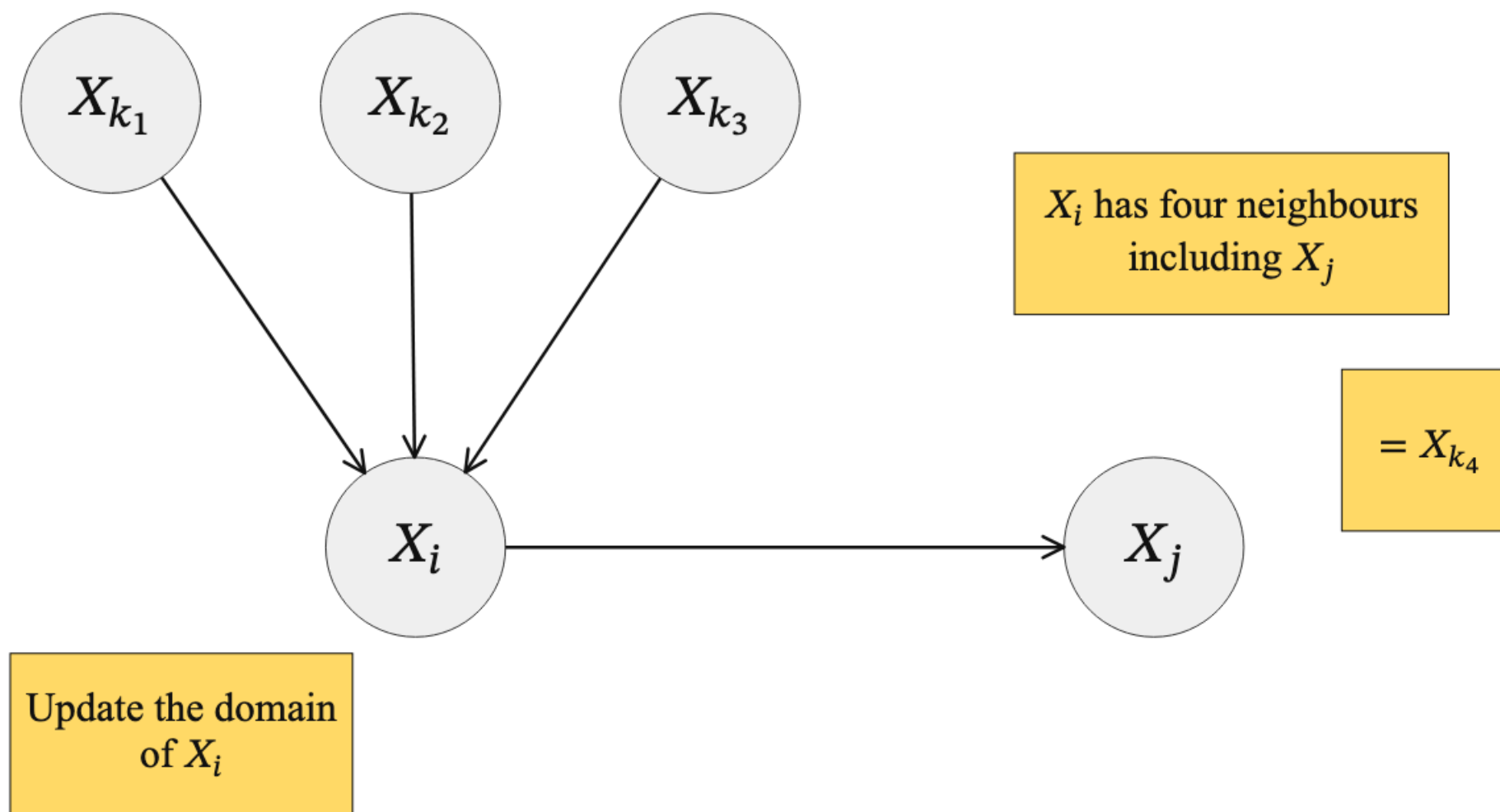
Why Exclude the Reverse Arc?



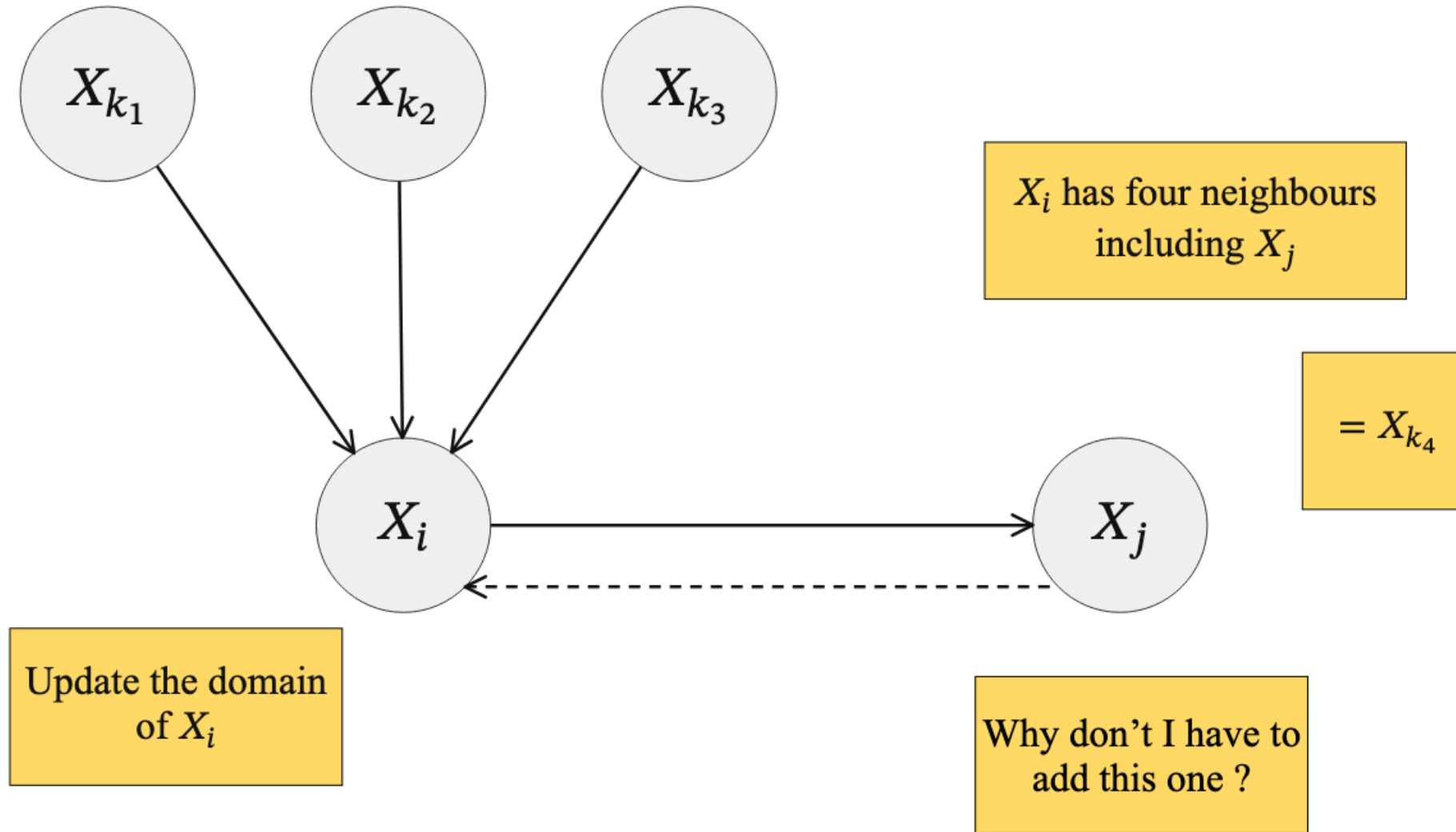
Why Exclude the Reverse Arc?



Why Exclude the Reverse Arc?



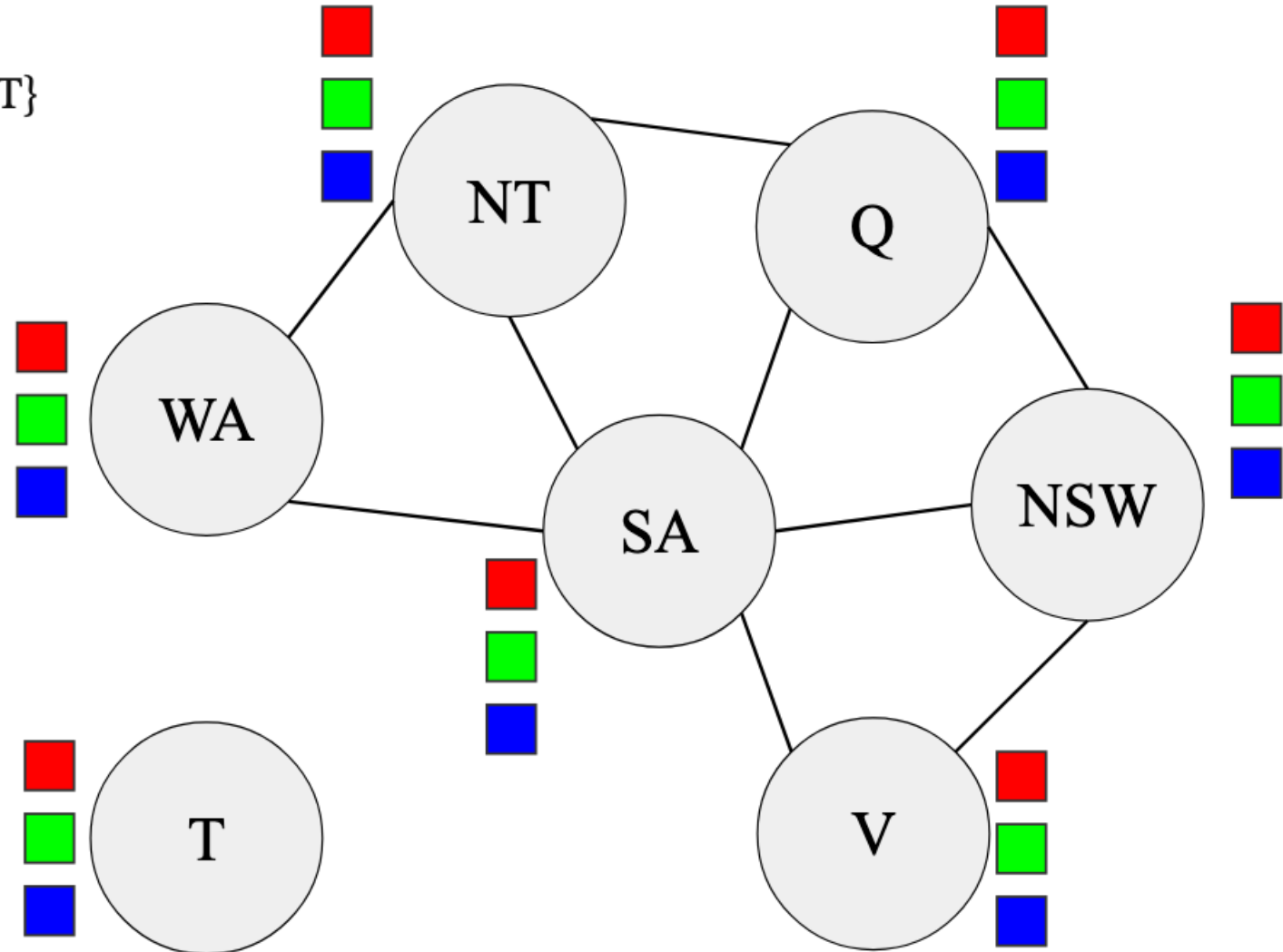
Why Exclude the Reverse Arc?



Backtracking with AC-3

Variables: {NSW, WA, NT, Q, SA, V, T}

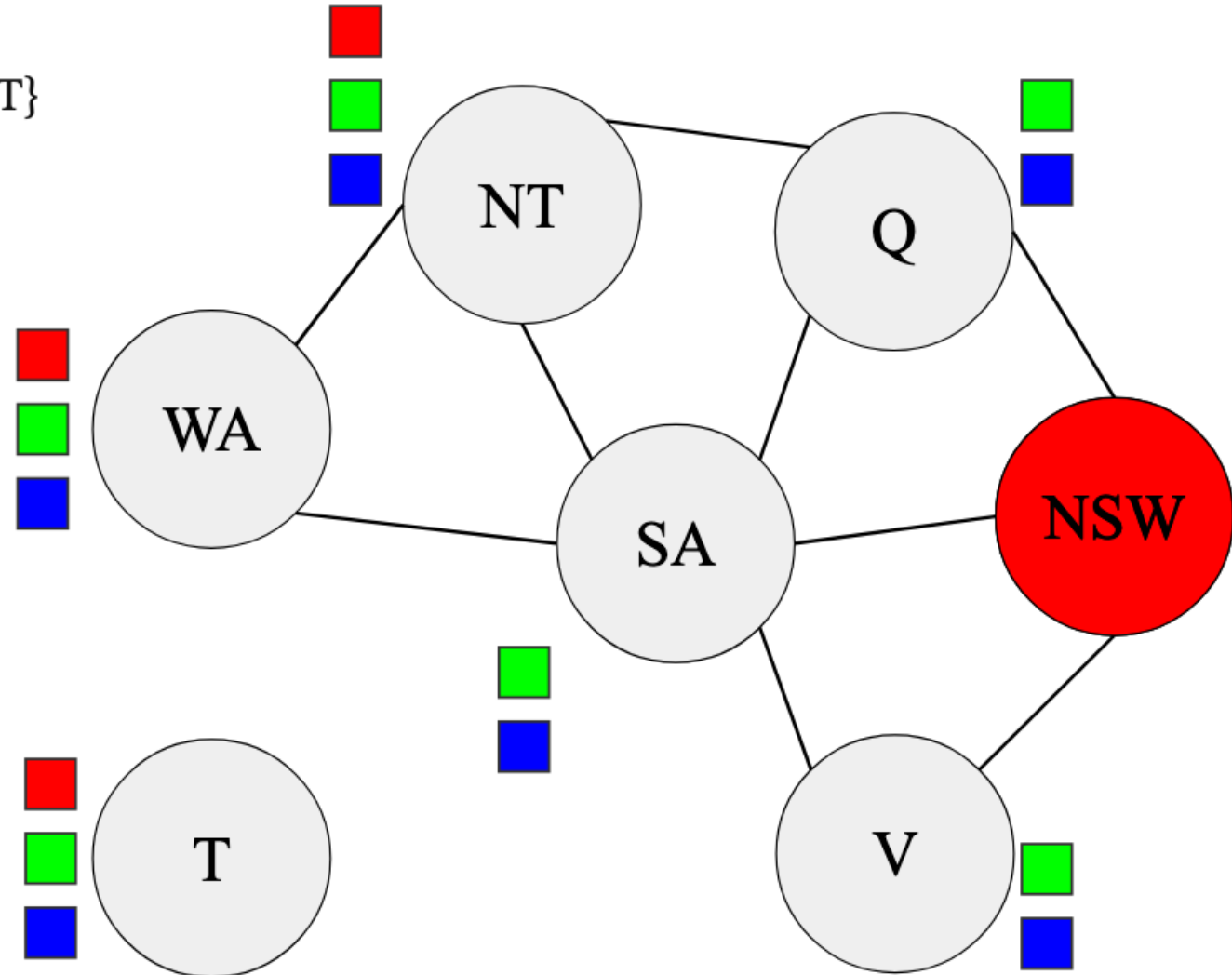
Colors: {red, green, blue}



Backtracking with AC-3

Variables: {NSW, WA, NT, Q, SA, V, T}

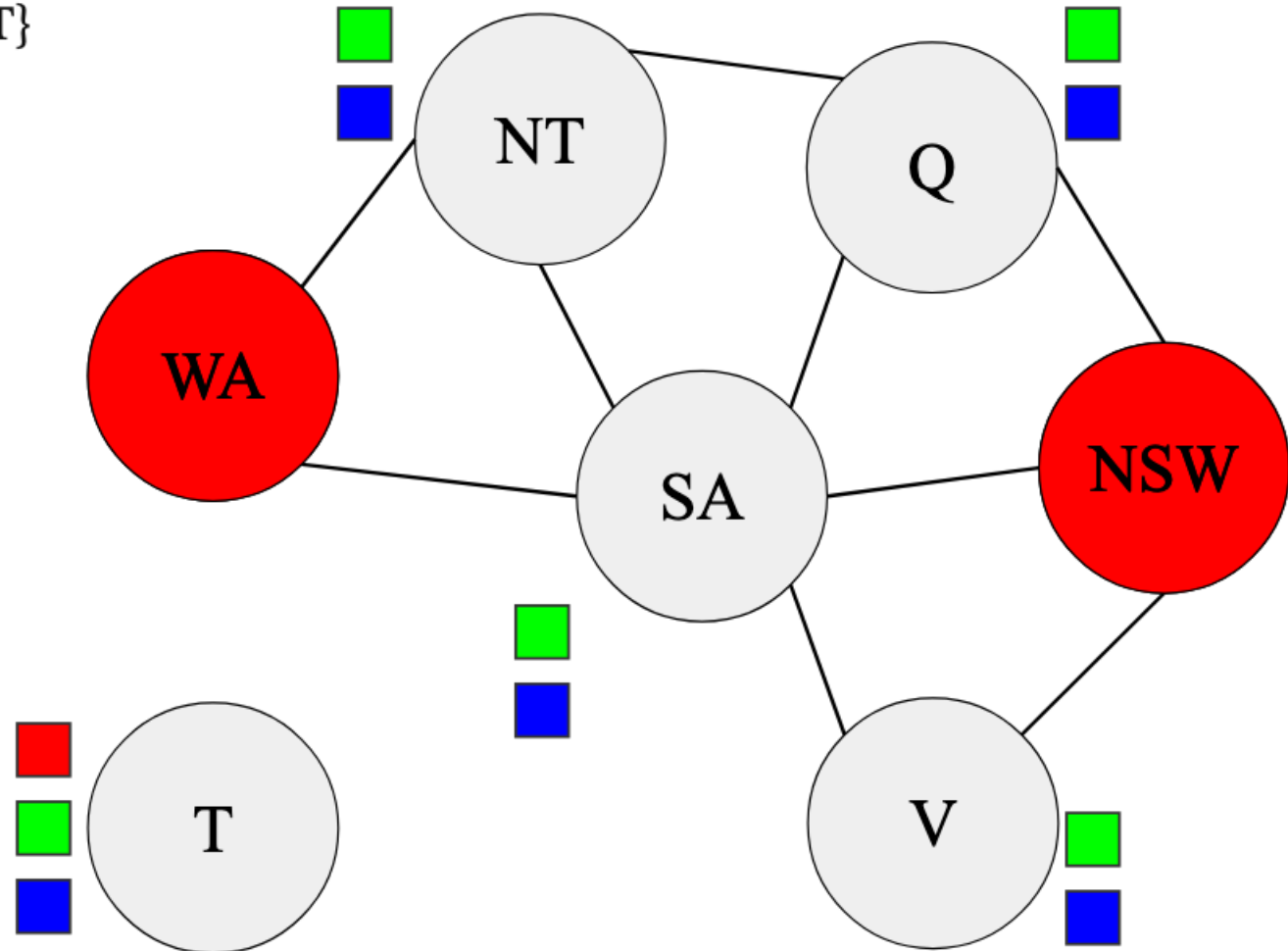
Colors: {red, green, blue}



Backtracking with AC-3

Variables: {NSW, WA, NT, Q, SA, V, T}

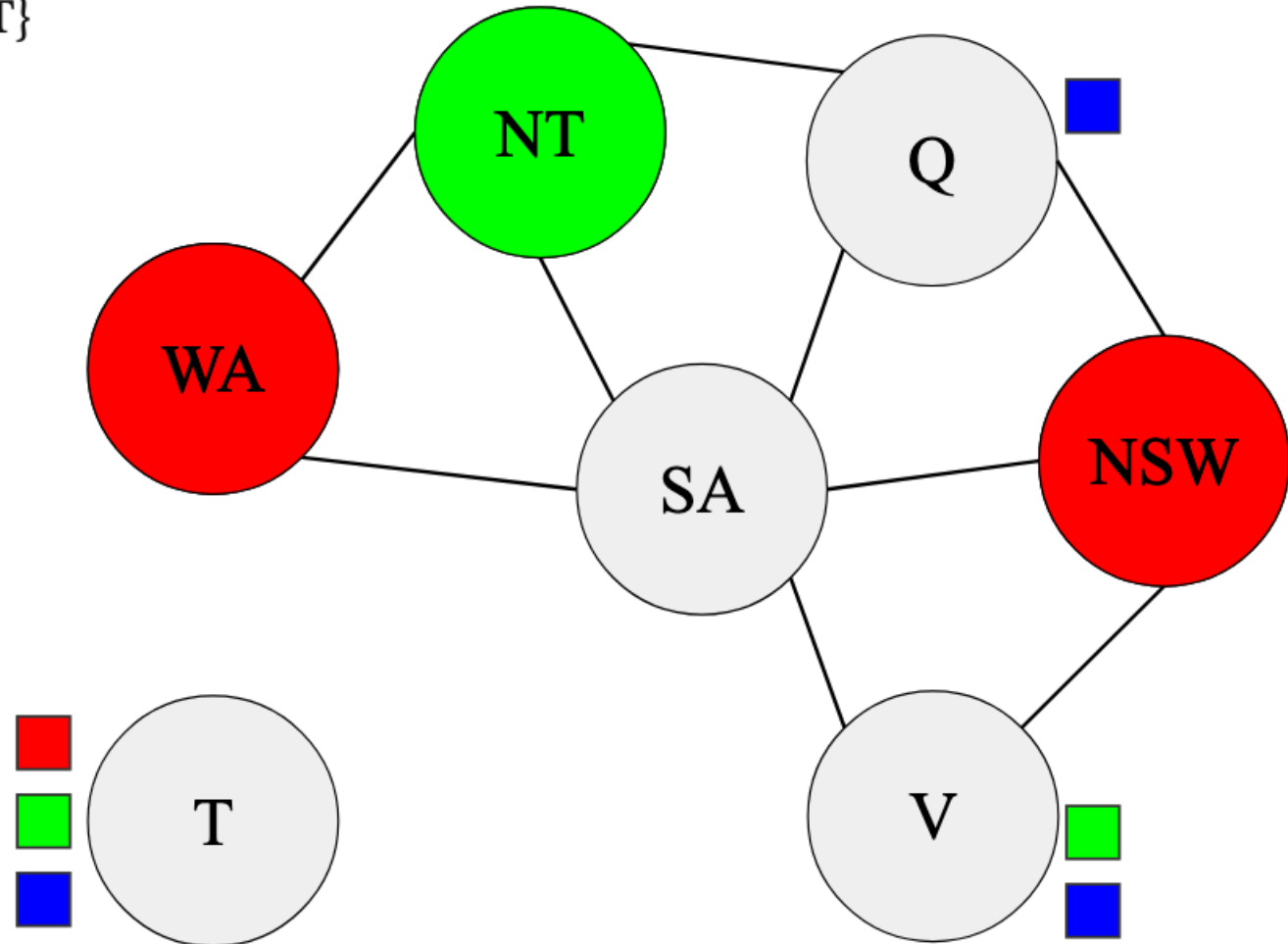
Colors: {red, green, blue}



Backtracking with AC-3

Variables: {NSW, WA, NT, Q, SA, V, T}

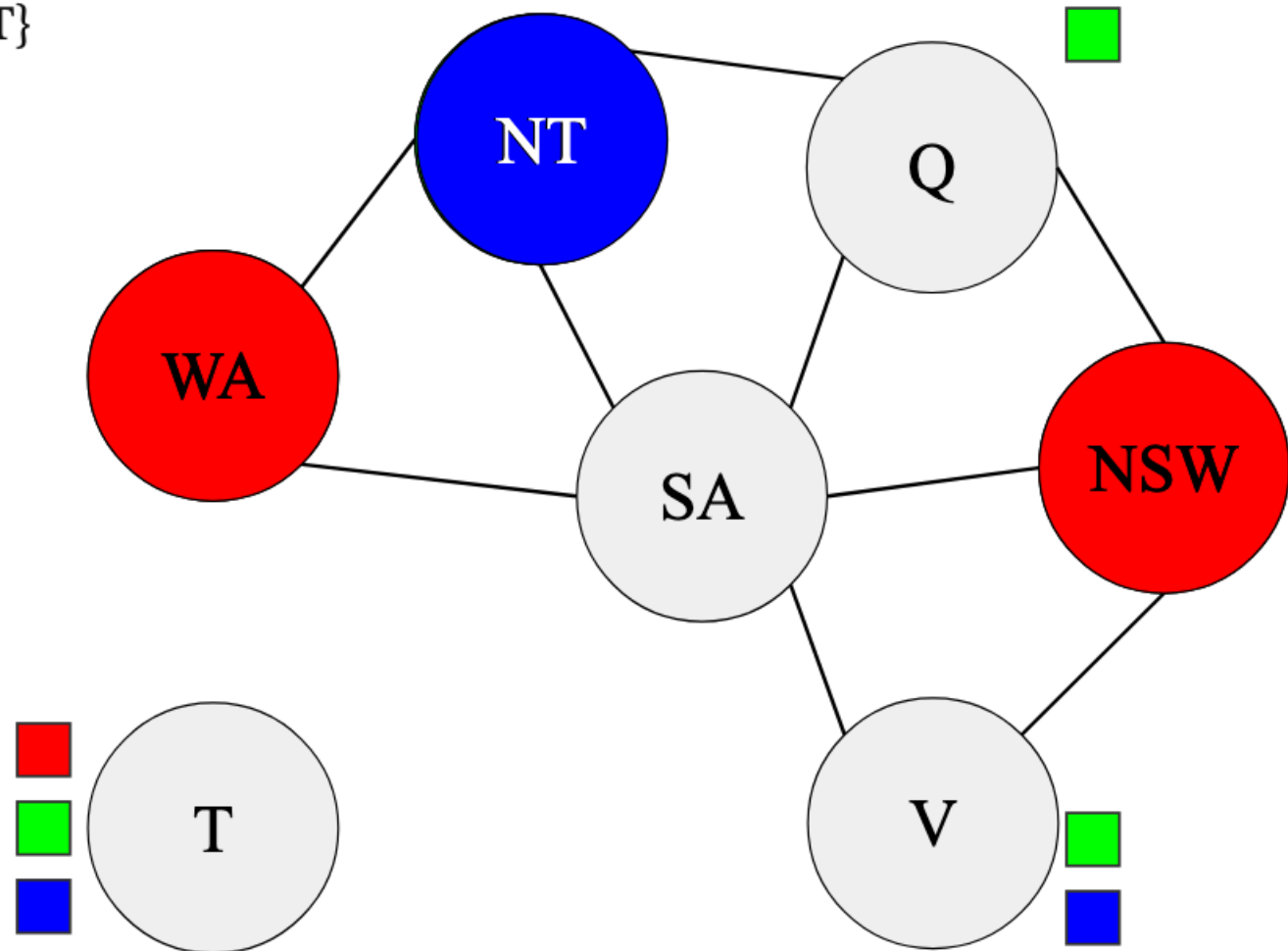
Colors: {red, green, blue}



Backtracking with AC-3

Variables: {NSW, WA, NT, Q, SA, V, T}

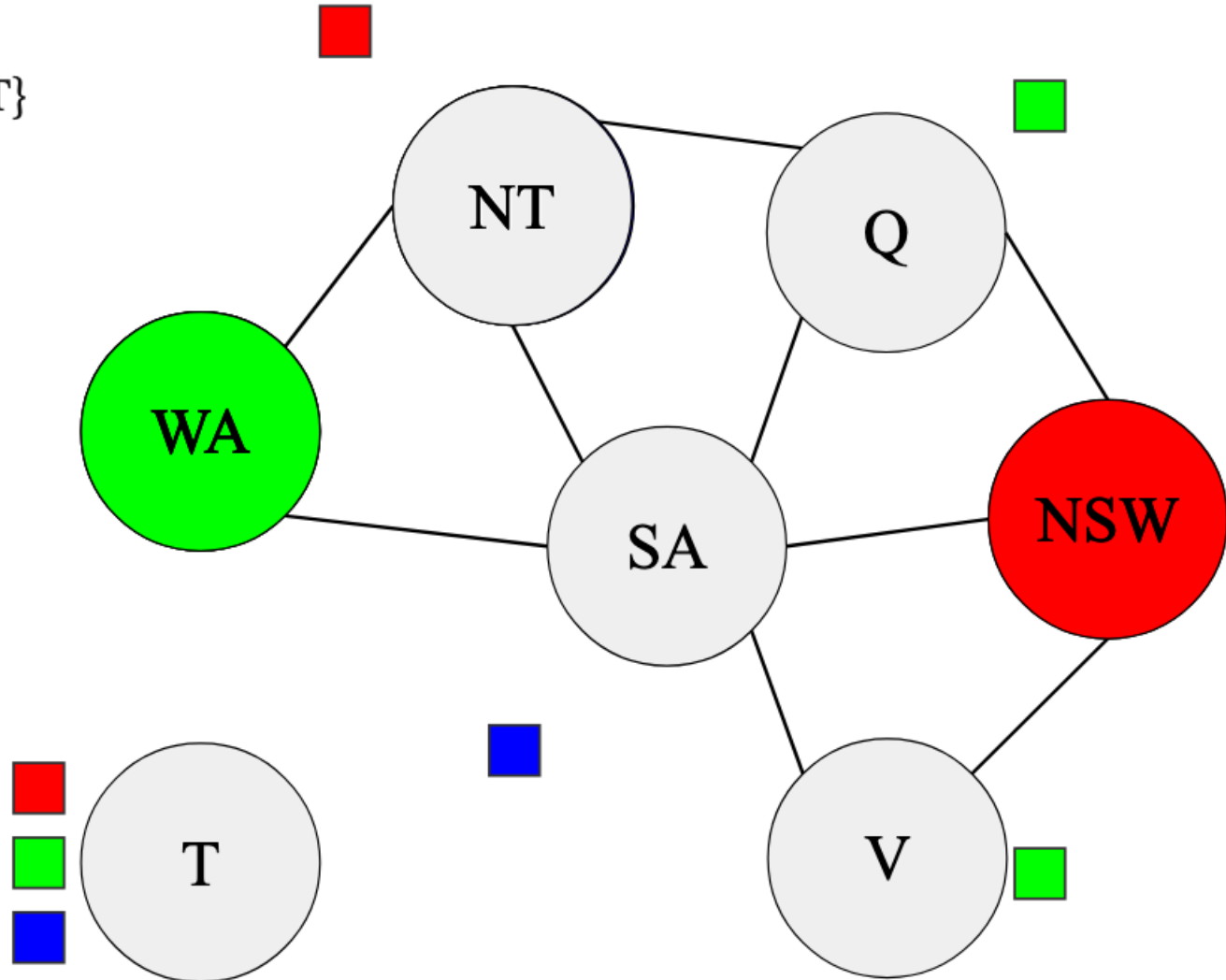
Colors: {red, green, blue}



Backtracking with AC-3

Variables: {NSW, WA, NT, Q, SA, V, T}

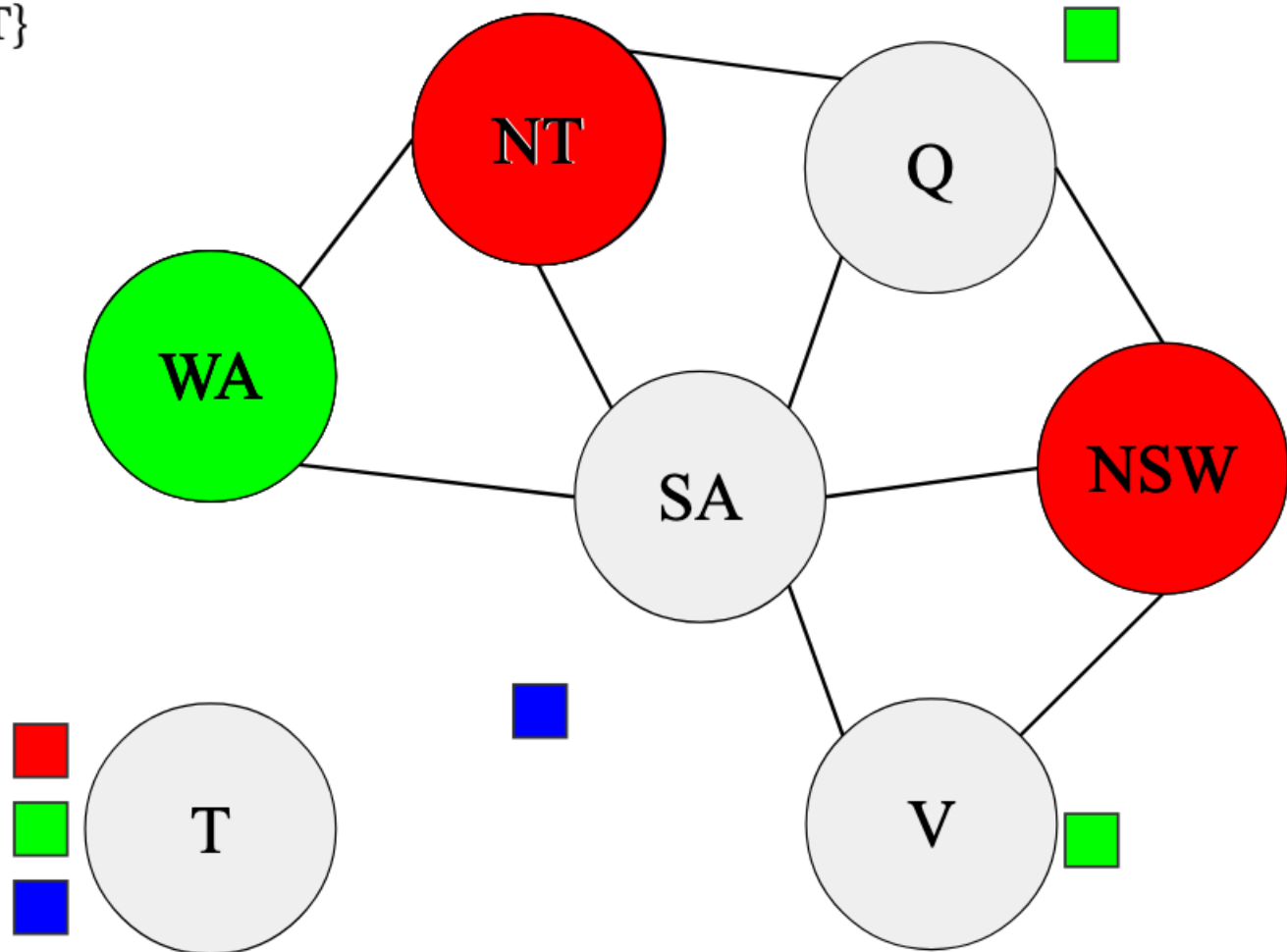
Colors: {red, green, blue}



Backtracking with AC-3

Variables: {NSW, WA, NT, Q, SA, V, T}

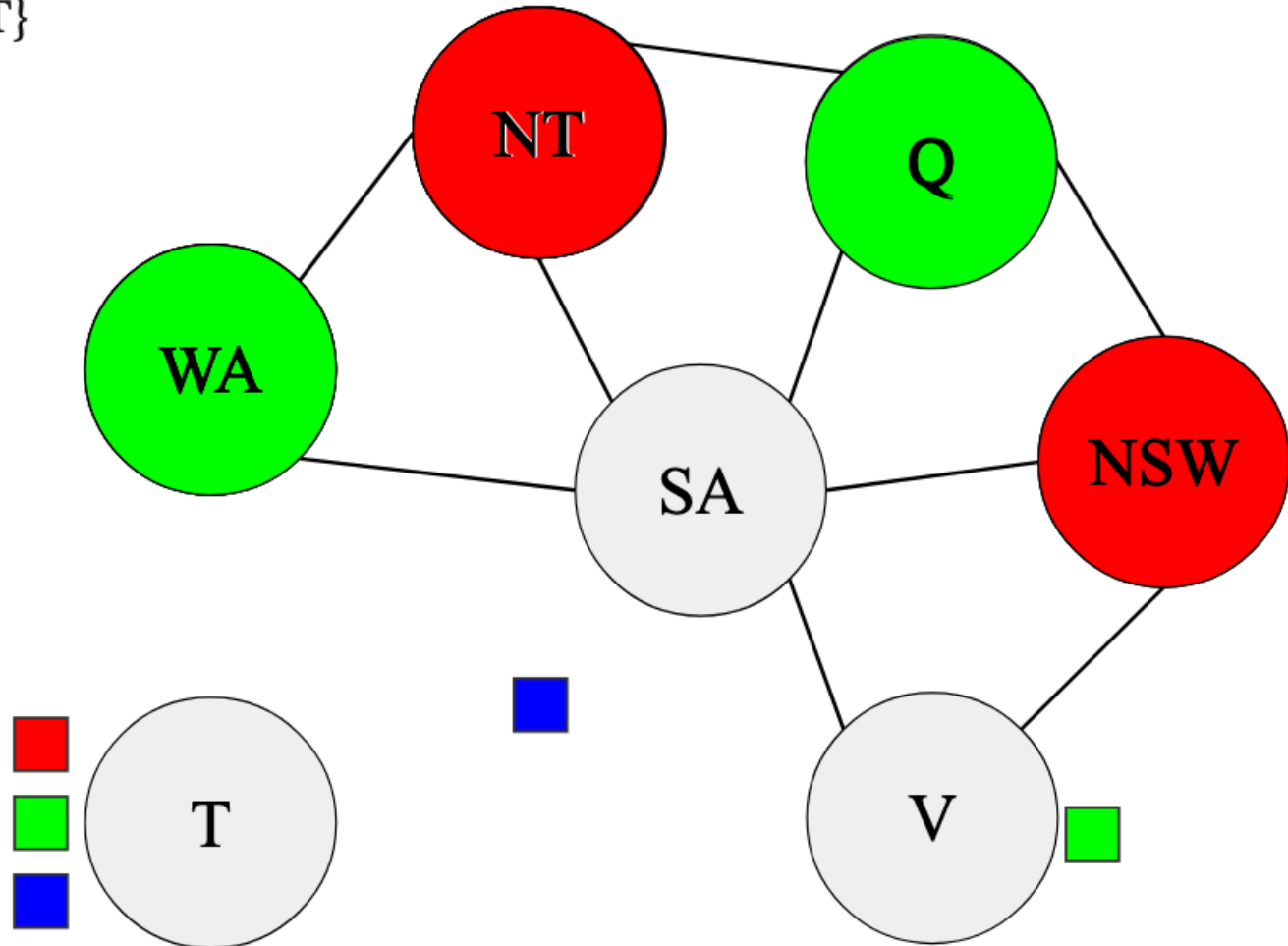
Colors: {red, green, blue}



Backtracking with AC-3

Variables: {NSW, WA, NT, Q, SA, V, T}

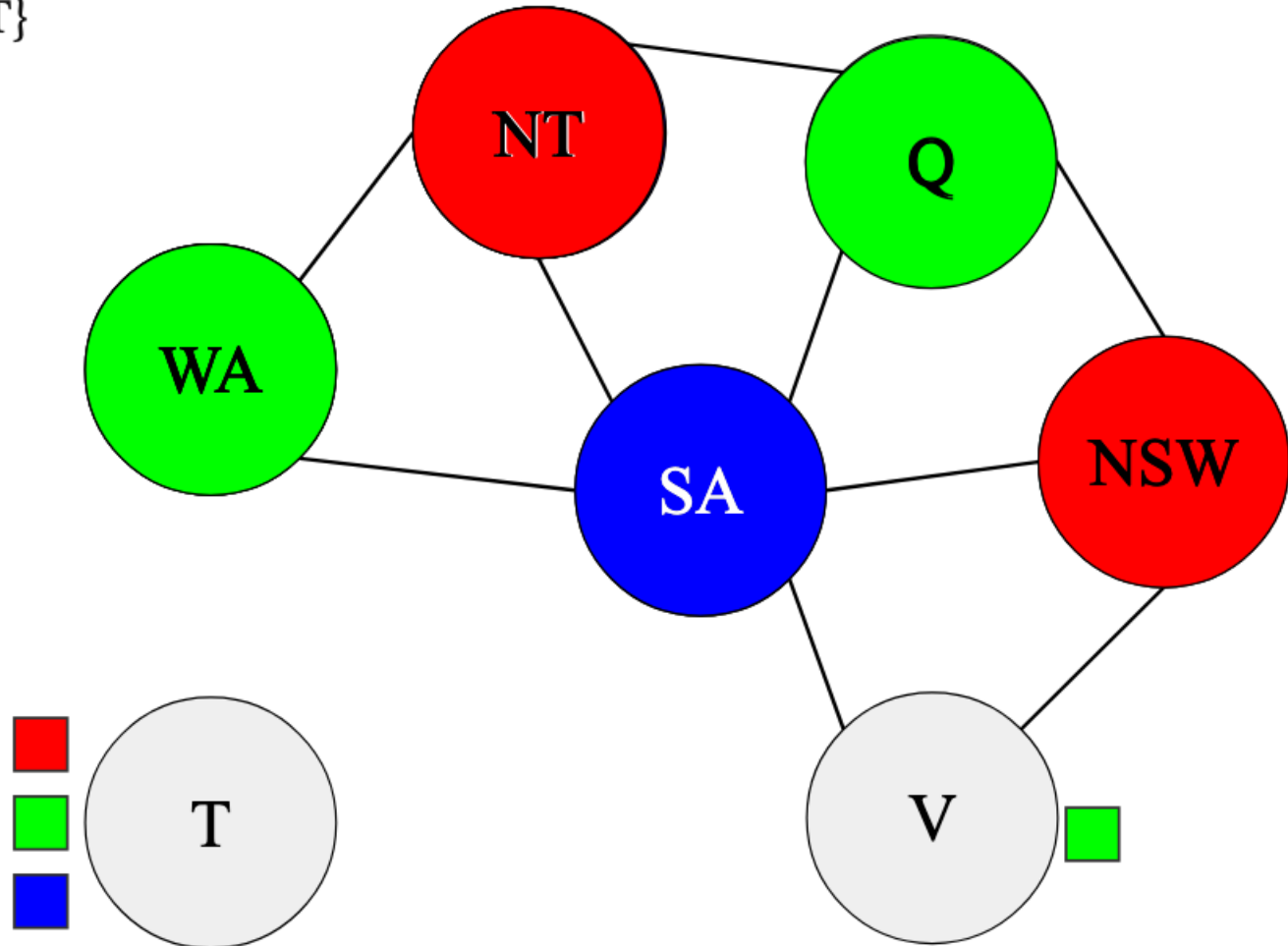
Colors: {red, green, blue}



Backtracking with AC-3

Variables: {NSW, WA, NT, Q, SA, V, T}

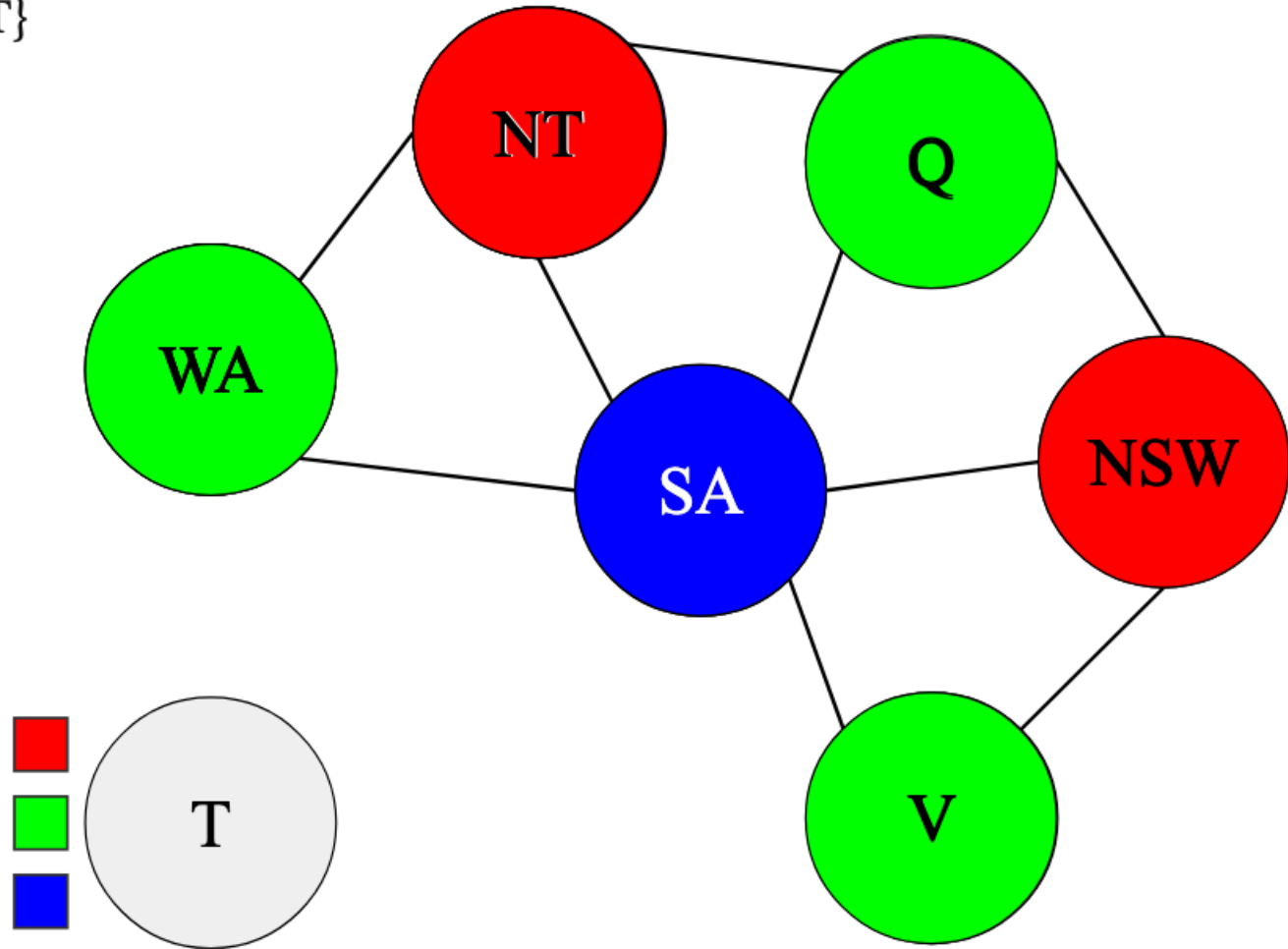
Colors: {red, green, blue}



Backtracking with AC-3

Variables: {NSW, WA, NT, Q, SA, V, T}

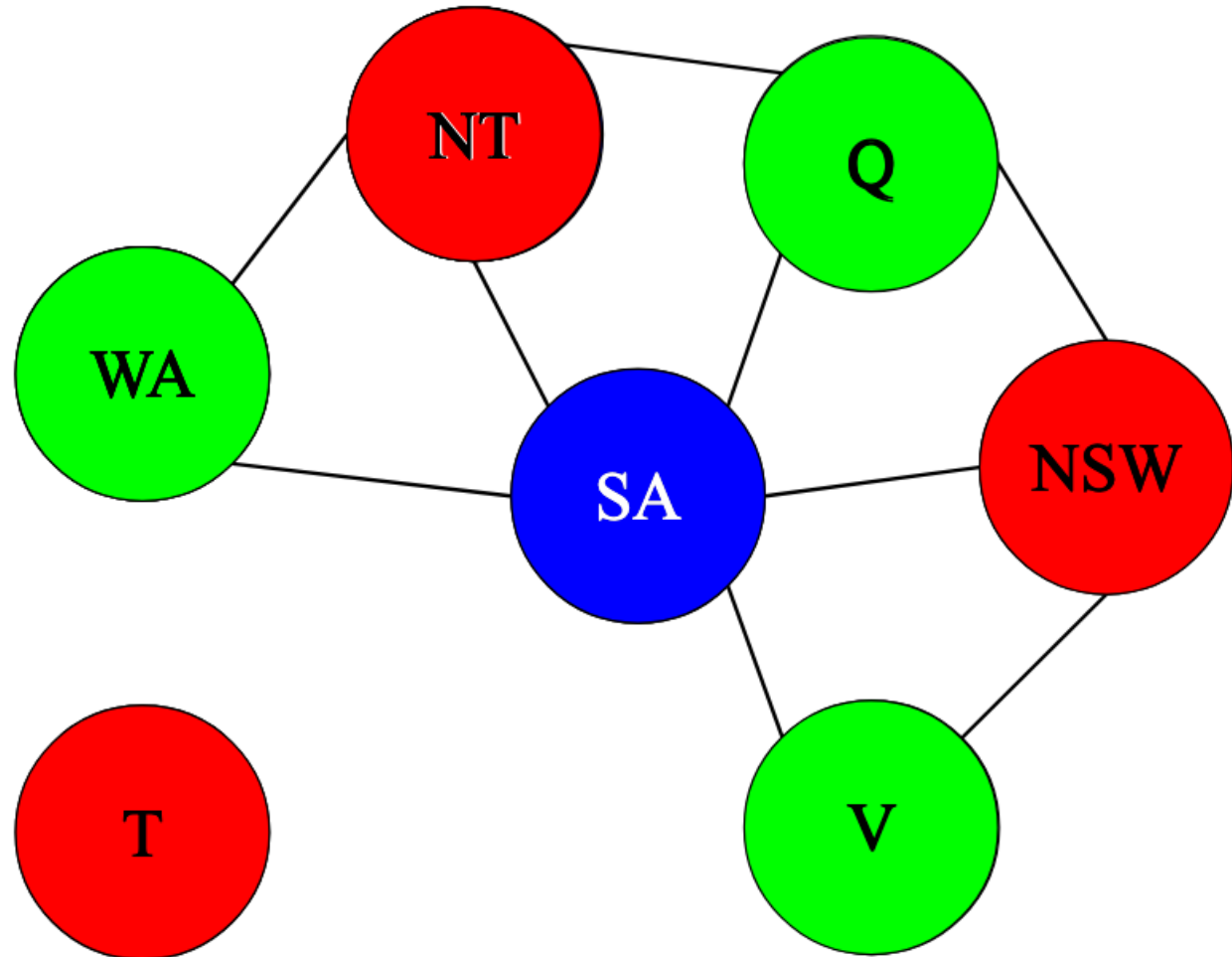
Colors: {red, green, blue}



Backtracking with AC-3

Variables: {NSW, WA, NT, Q, SA, V, T}

Colors: {red, green, blue}



Variable and Value Ordering

Variable and Value Ordering

Variable Ordering

Minimum Remaining Values (MRV)

Choose the variable with the fewest legal values

Most constrained variable

Fail-first heuristic

Variable and Value Ordering

Variable Ordering

Minimum Remaining Values (MRV)

Choose the variable with the fewest legal values

Most constrained variable

Fail-first heuristic

Break MRV Ties

Degree Heuristic

Most constraints on unassigned variables

Variable and Value Ordering

Variable Ordering

Minimum Remaining Values (MRV)

Choose the variable with the fewest legal values

Most constrained variable

Fail-first heuristic

Break MRV Ties

Degree Heuristic

Most constraints on unassigned variables

Value Ordering

Least Constraining Value (LCV)

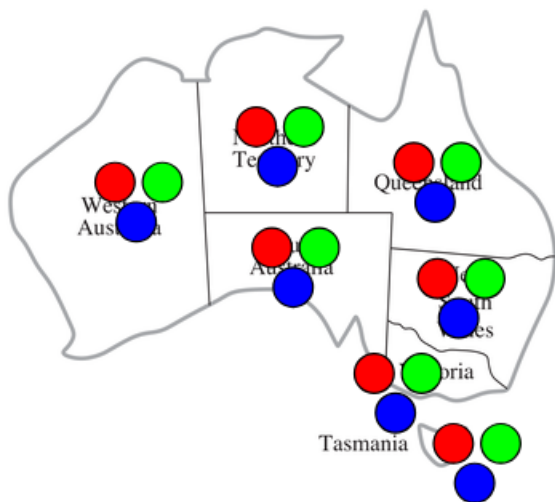
Rules out the fewest values in unassigned neighbours

Variable Ordering (MRV)

Minimum Remaining Values

Choose the unassigned variable with the fewest legal values

Most constrained variable · Fail-first heuristic



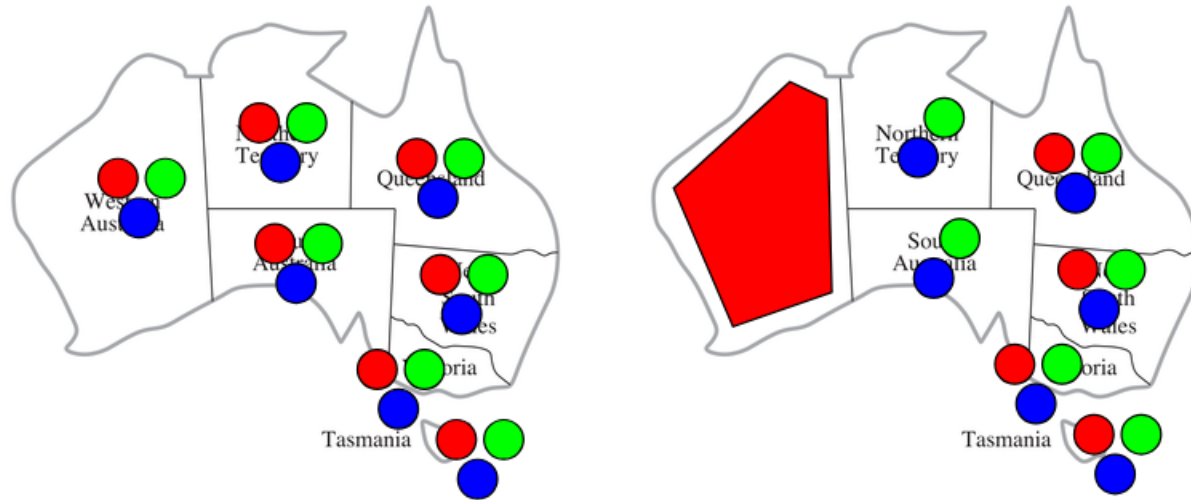
Using Forward Checking

Variable Ordering (MRV)

Minimum Remaining Values

Choose the unassigned variable with the fewest legal values

Most constrained variable · Fail-first heuristic



Using Forward Checking

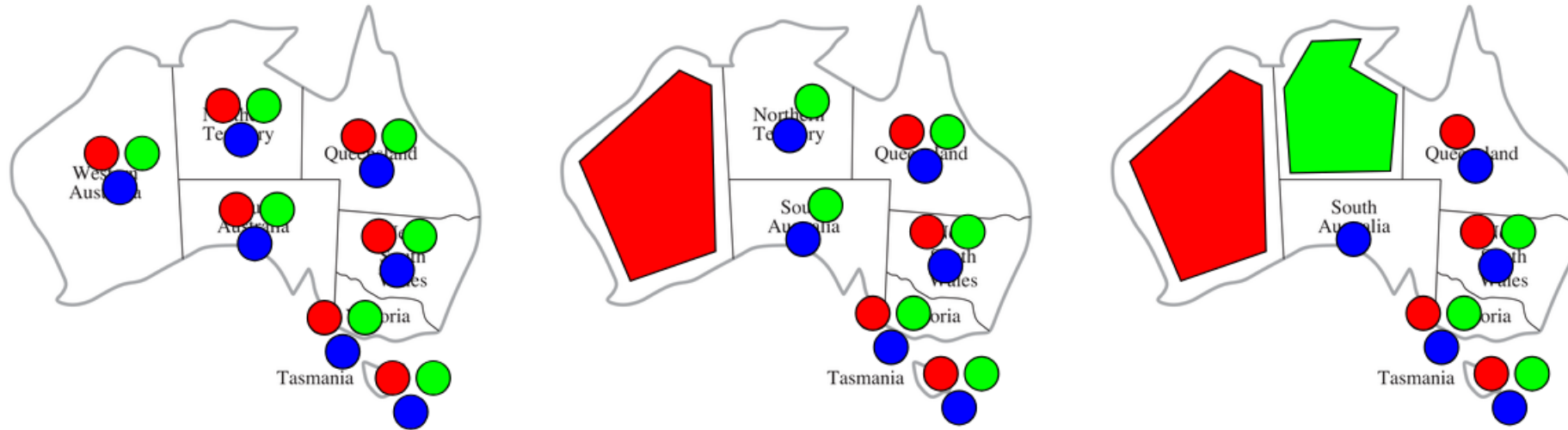
Slide 127 · Stage 2/4

Variable Ordering (MRV)

Minimum Remaining Values

Choose the unassigned variable with the fewest legal values

Most constrained variable · Fail-first heuristic



Using Forward Checking

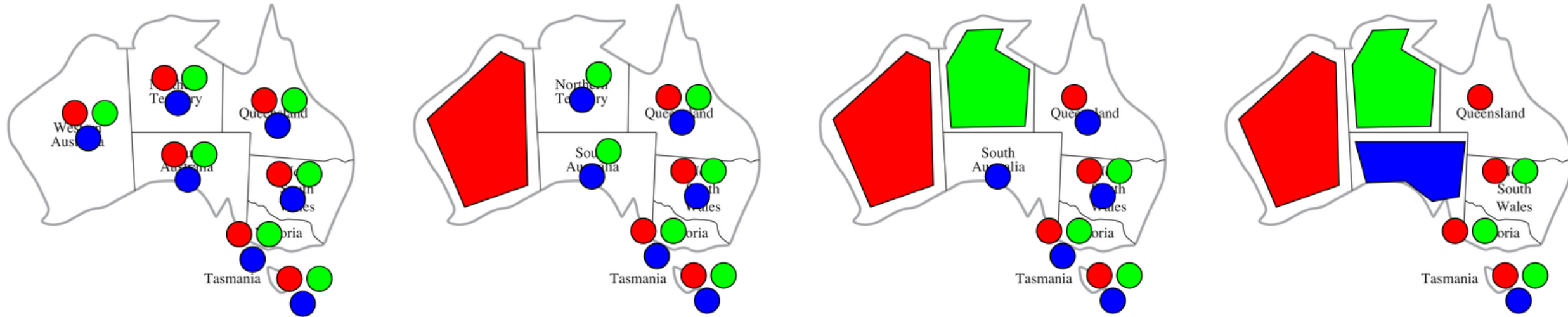
Slide 127 · Stage 3/4

Variable Ordering (MRV)

Minimum Remaining Values

Choose the unassigned variable with the fewest legal values

Most constrained variable · Fail-first heuristic



Using Forward Checking

Slide 127 · Stage 4/4

Variable Ordering (MRV + Deg)

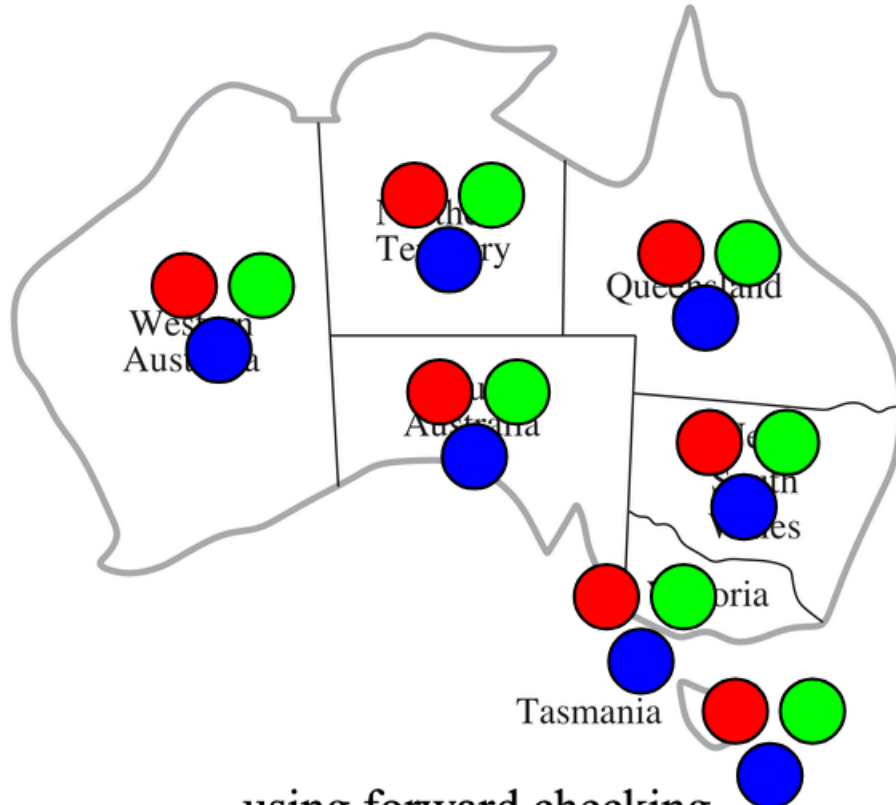
Degree Heuristic

Among MRV ties, choose the variable with the most constraints on unassigned variables

Variable Ordering (MRV + Deg)

Degree Heuristic

Among MRV ties, choose the variable with the most constraints on unassigned variables

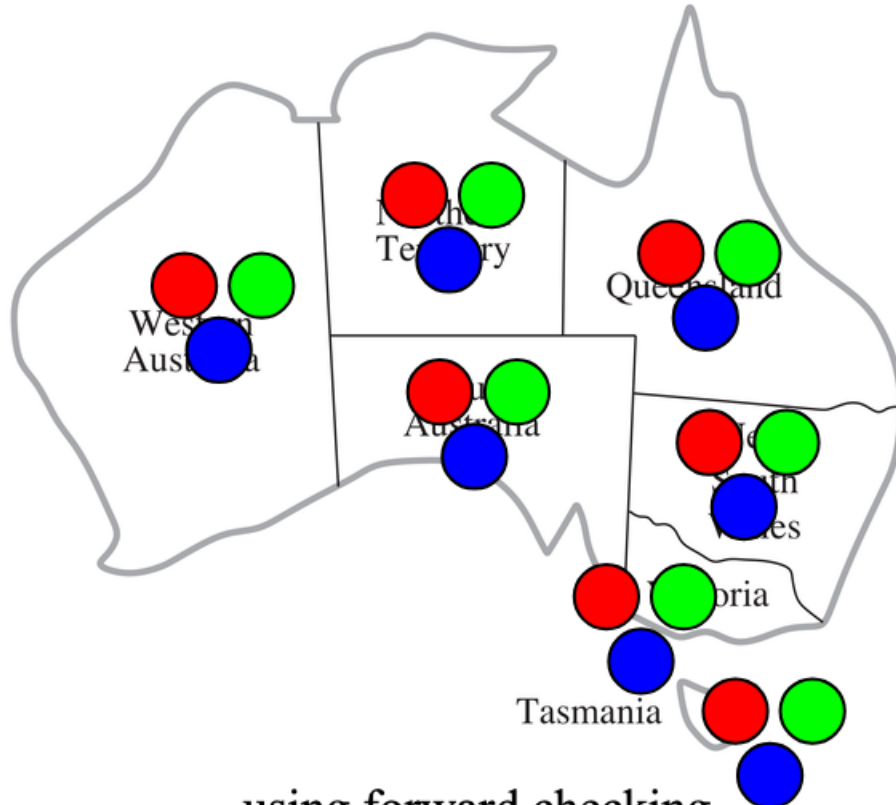


using forward checking

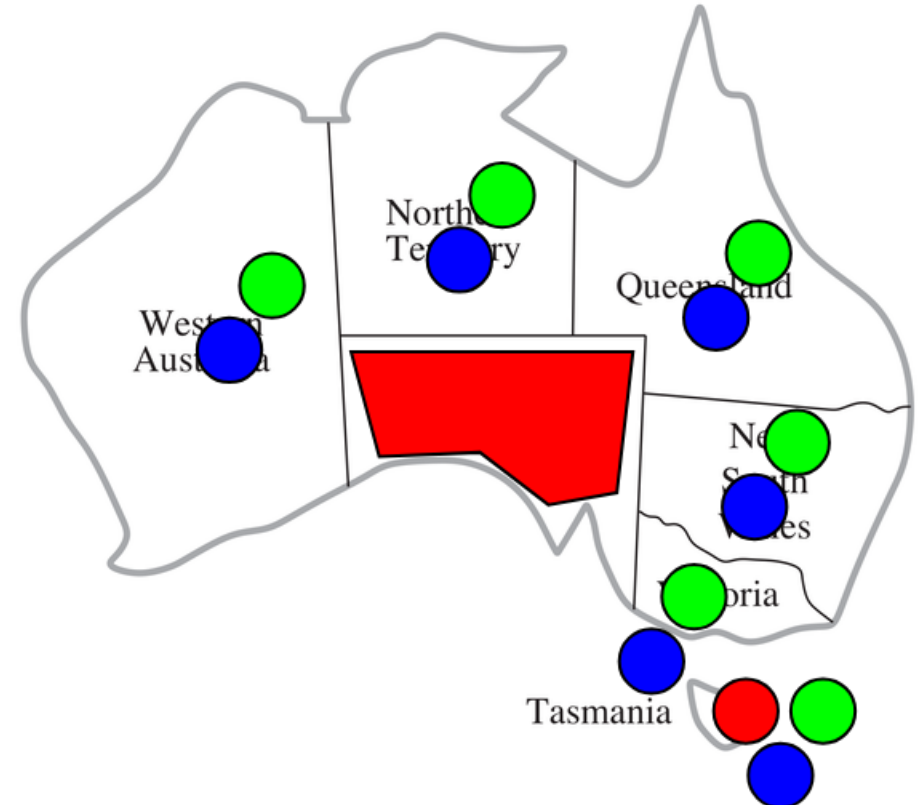
Variable Ordering (MRV + Deg)

Degree Heuristic

Among MRV ties, choose the variable with the most constraints on unassigned variables



using forward checking

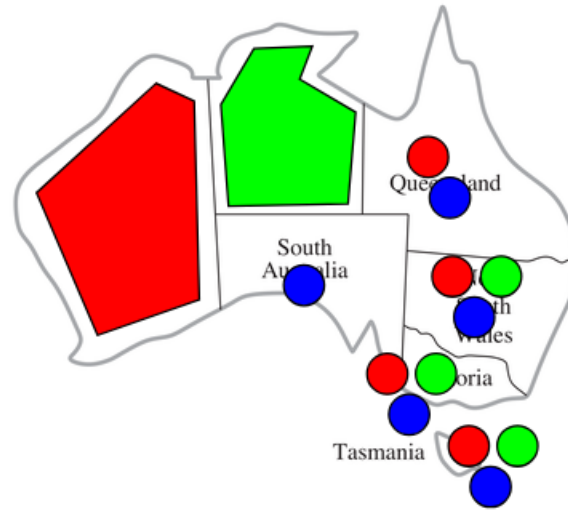


Value Ordering (LCV)

Least Constraining Value

Try the value that removes the fewest options from unassigned neighbours

Assign Q Next
Which Colour First?

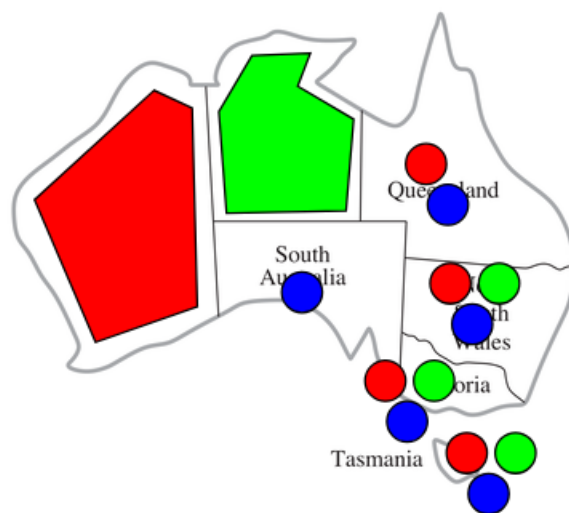
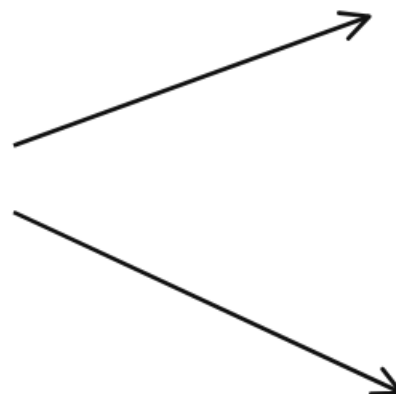
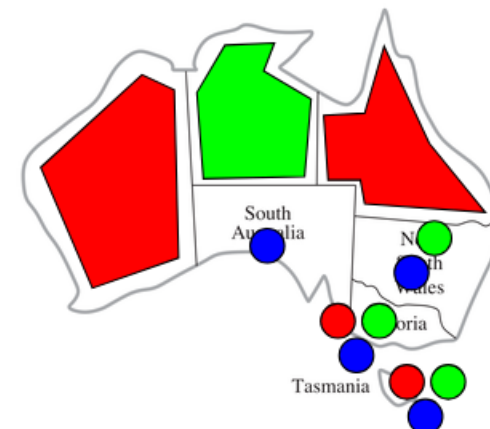
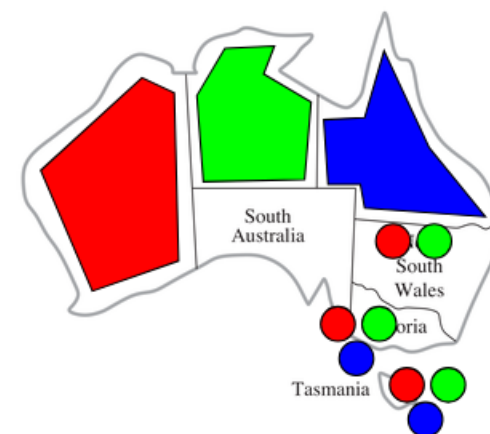


Value Ordering (LCV)

Least Constraining Value

Try the value that removes the fewest options from unassigned neighbours

Assign Q Next
Which Colour First?

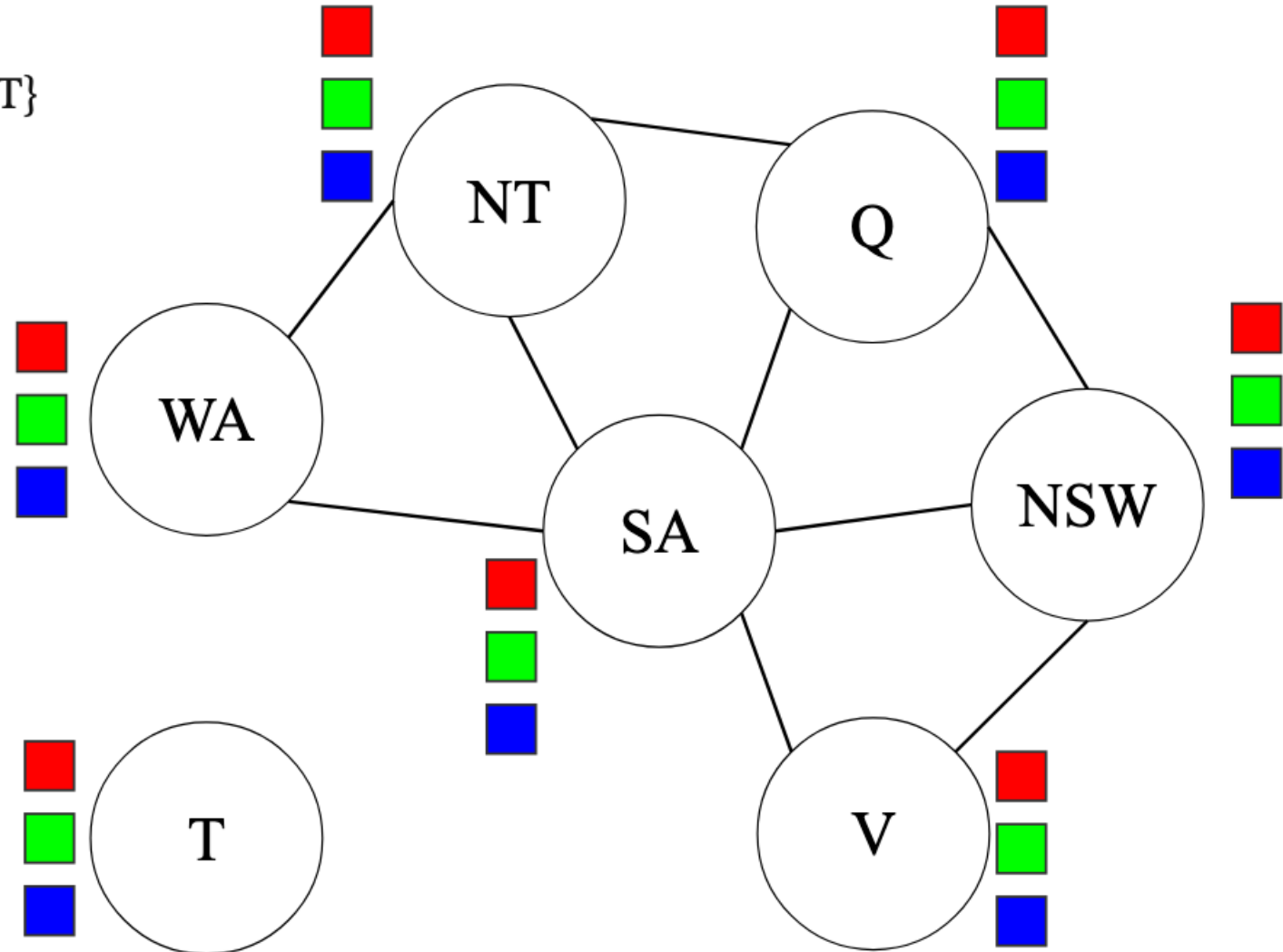


Backtracking with All Heuristics

AC-3 · MRV · Degree · LCV

Variables: {NSW, WA, NT, Q, SA, V, T}

Colors: {red, green, blue}

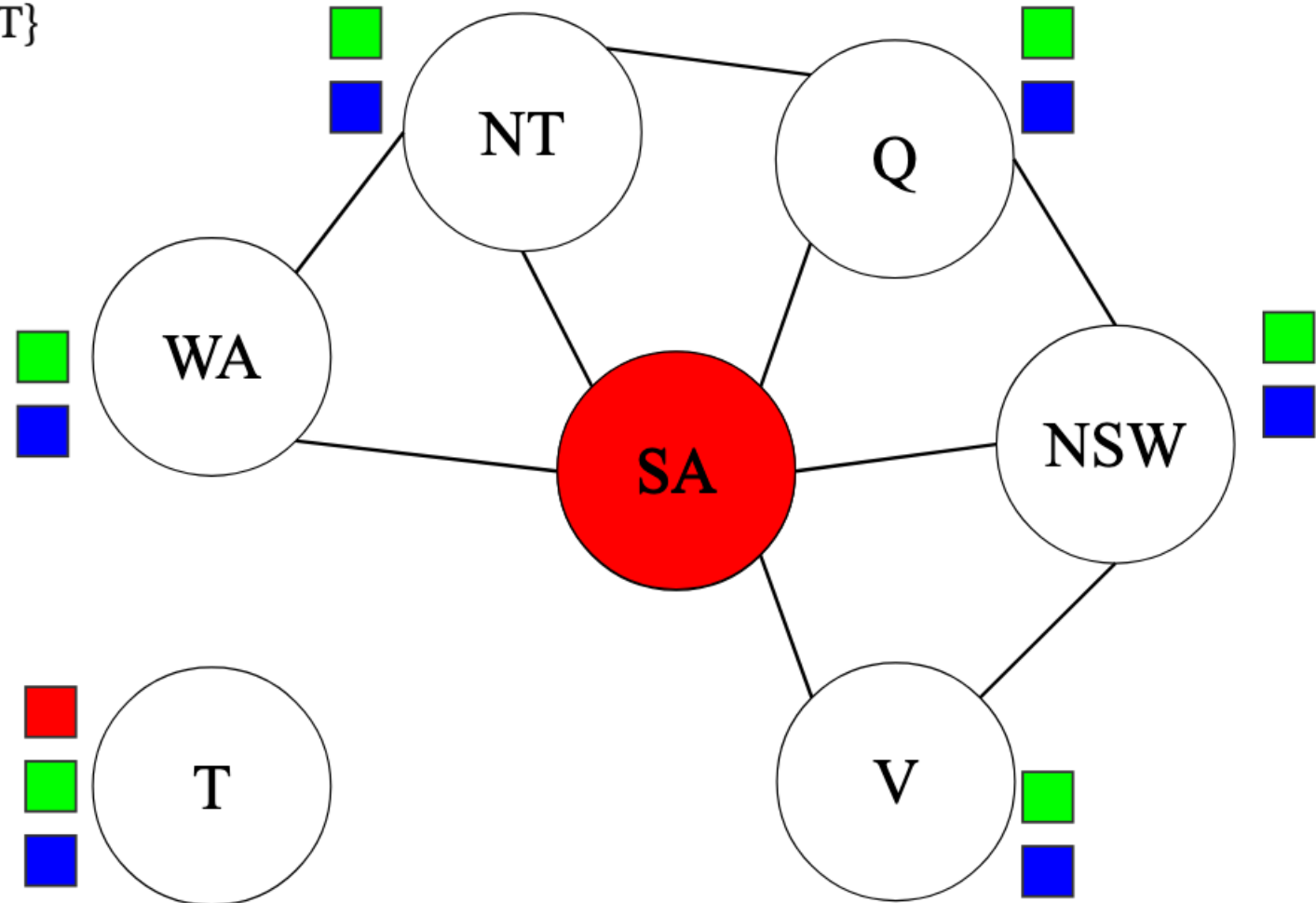


Backtracking with All Heuristics

AC-3 · MRV · Degree · LCV

Variables: {NSW, WA, NT, Q, SA, V, T}

Colors: {red, green, blue}

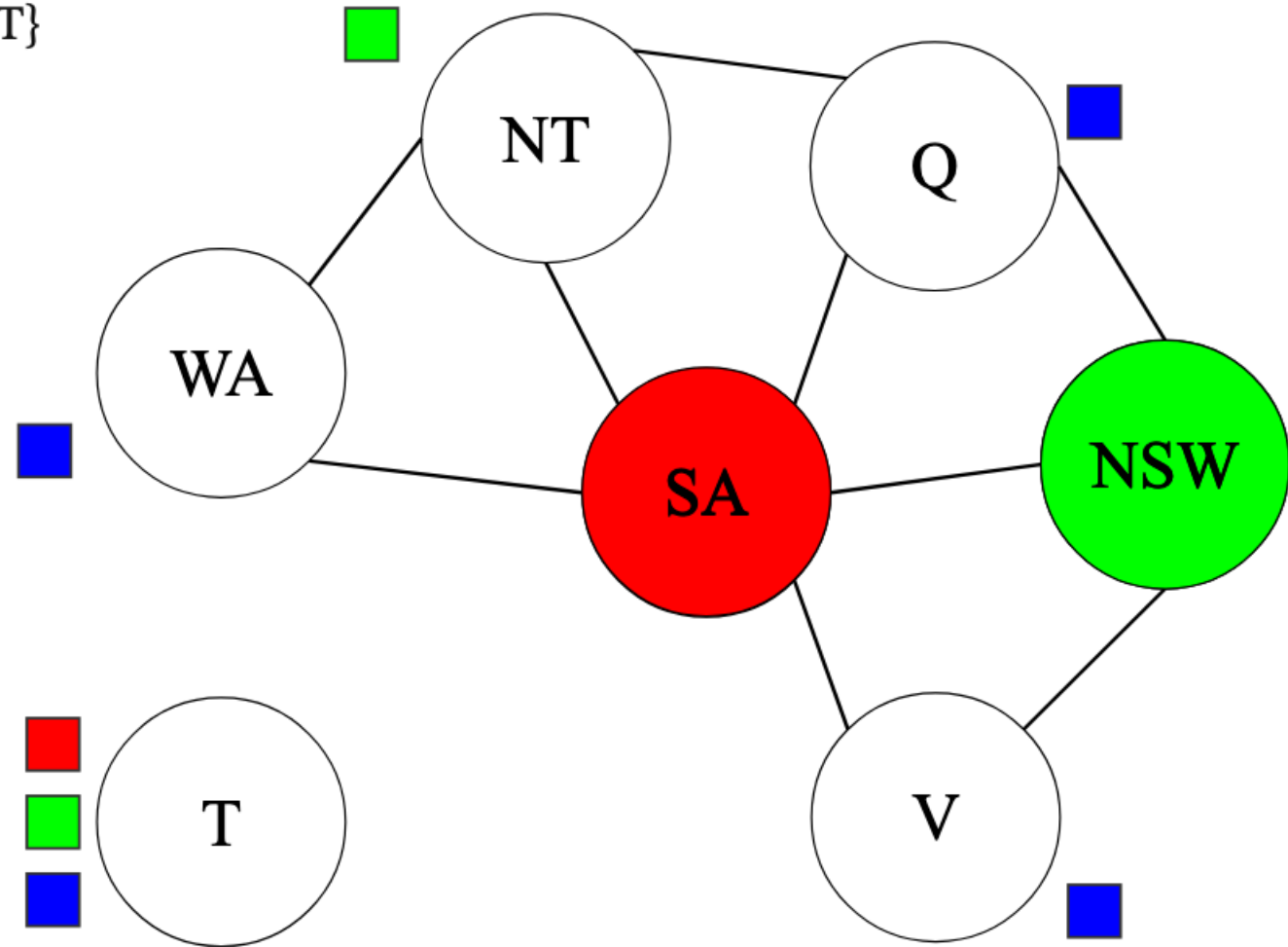


Backtracking with All Heuristics

AC-3 · MRV · Degree · LCV

Variables: {NSW, WA, NT, Q, SA, V, T}

Colors: {red, green, blue}

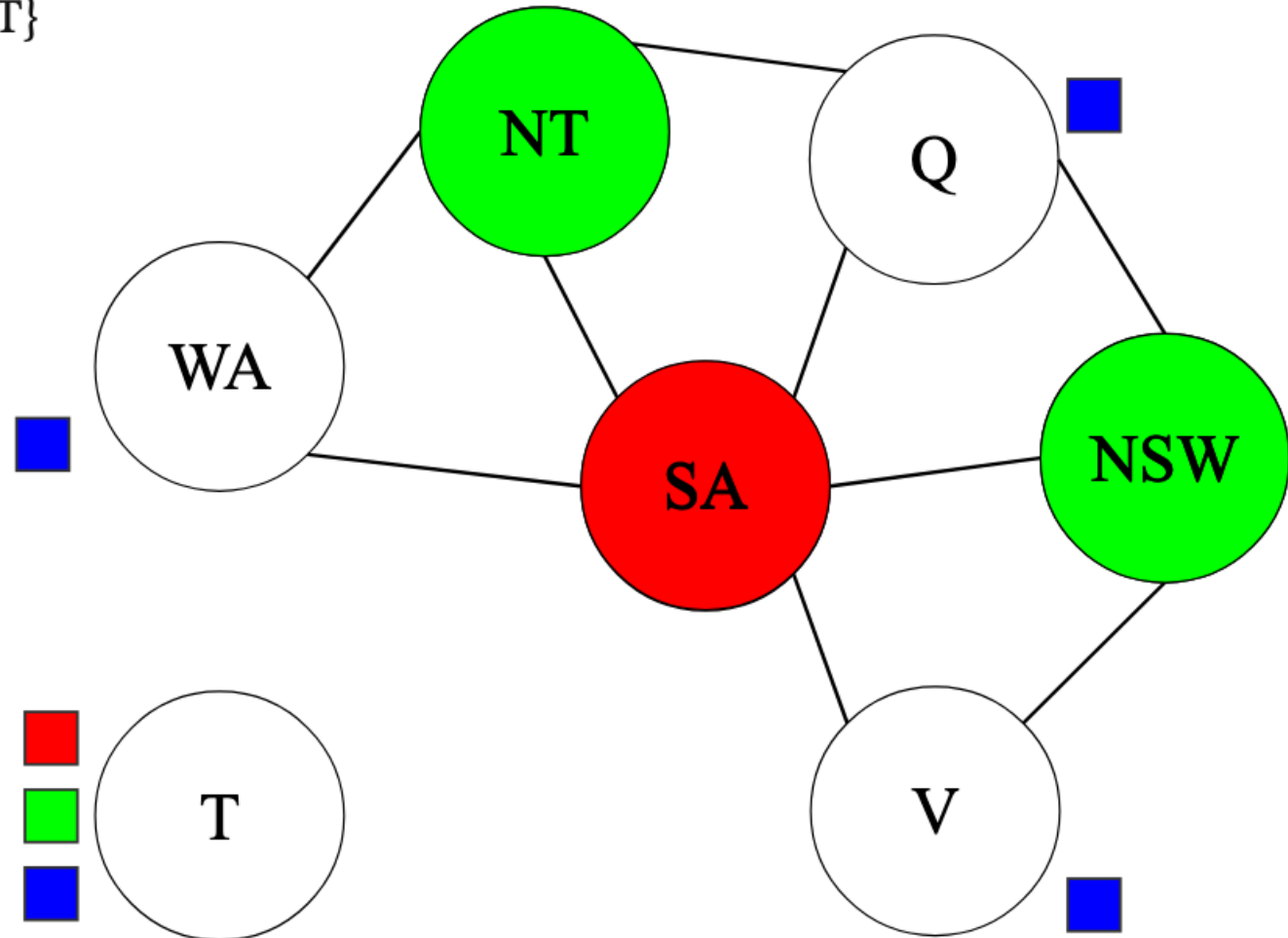


Backtracking with All Heuristics

AC-3 · MRV · Degree · LCV

Variables: {NSW, WA, NT, Q, SA, V, T}

Colors: {red, green, blue}

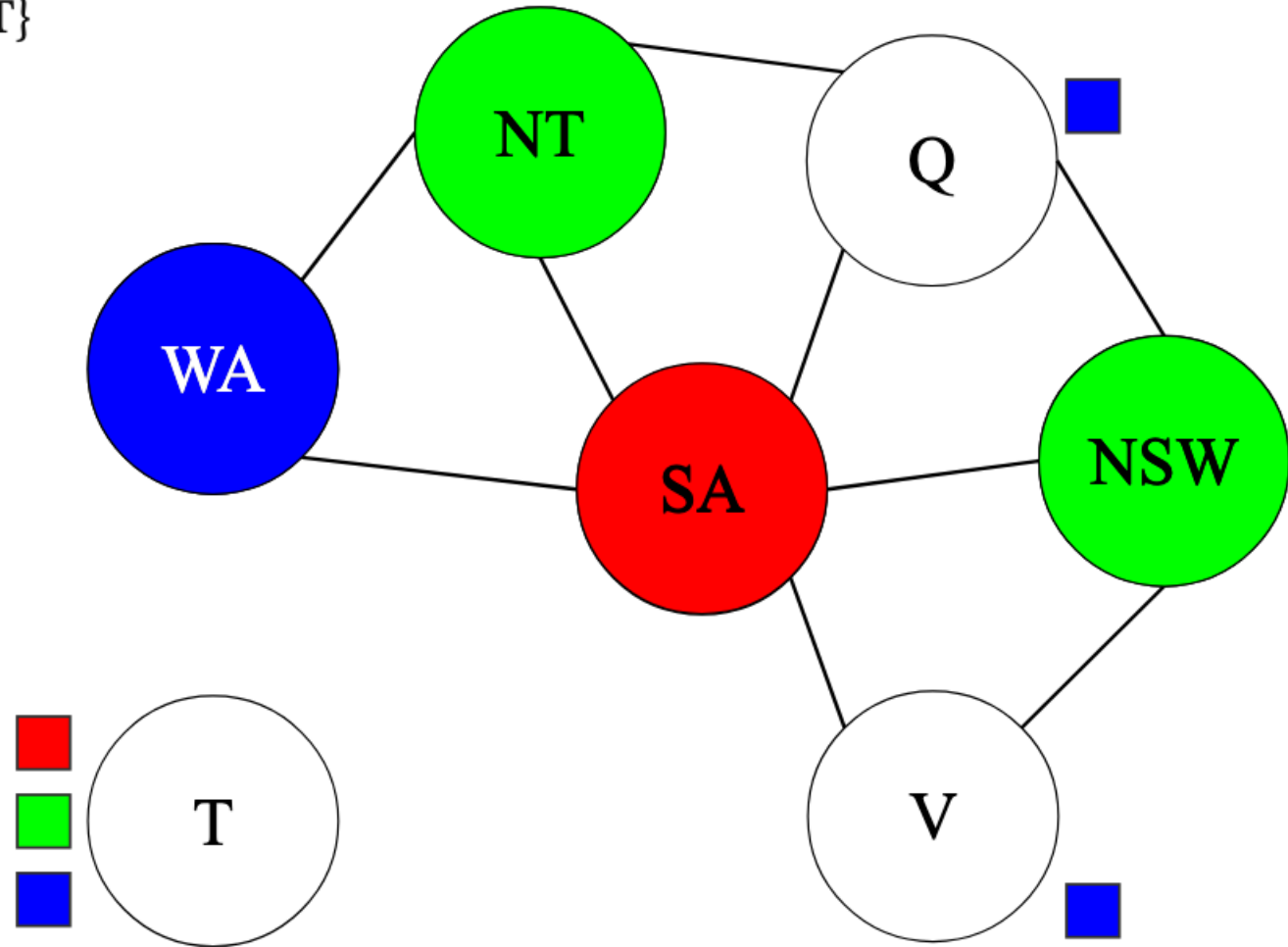


Backtracking with All Heuristics

AC-3 · MRV · Degree · LCV

Variables: {NSW, WA, NT, Q, SA, V, T}

Colors: {red, green, blue}

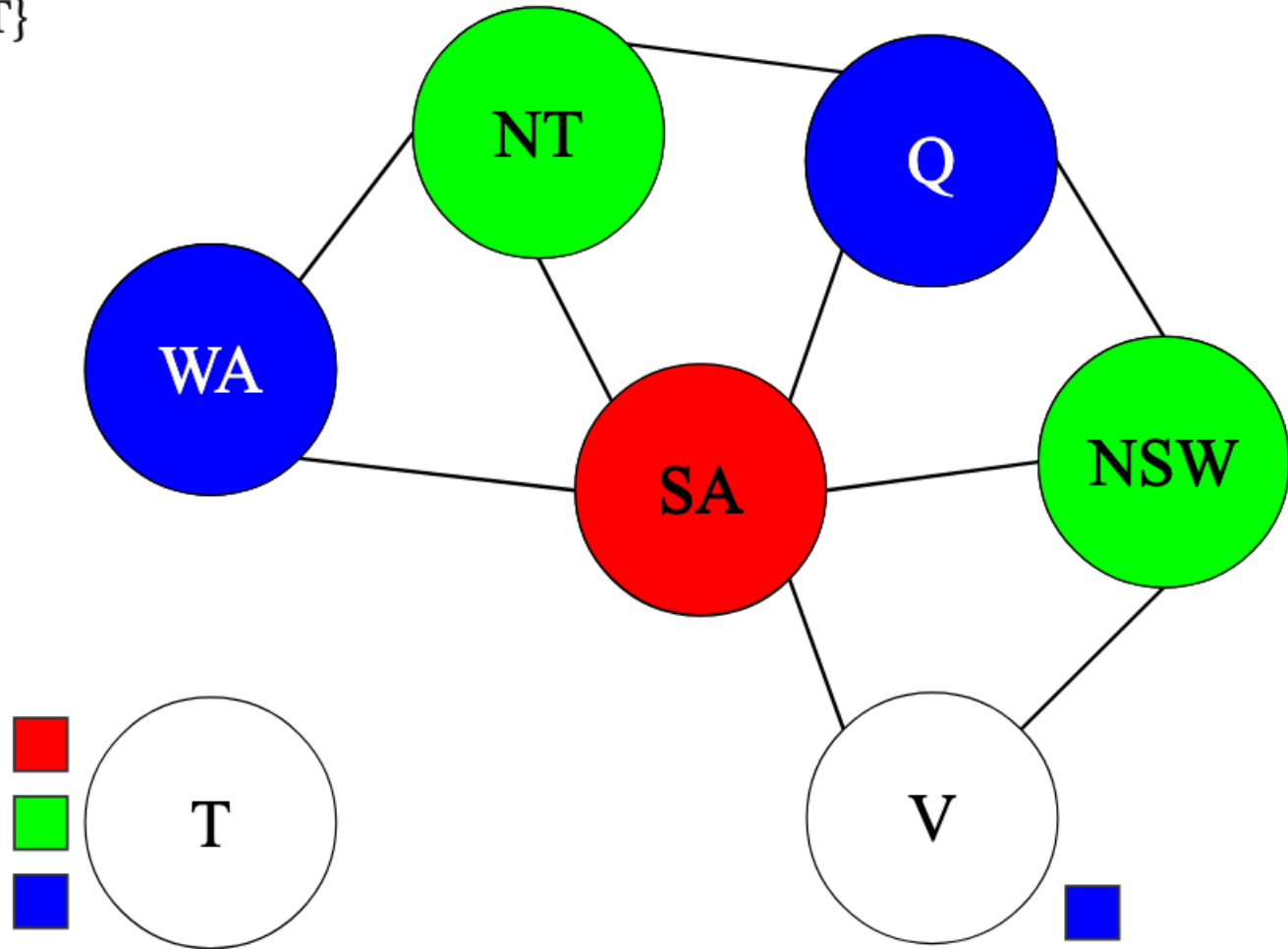


Backtracking with All Heuristics

AC-3 · MRV · Degree · LCV

Variables: {NSW, WA, NT, Q, SA, V, T}

Colors: {red, green, blue}

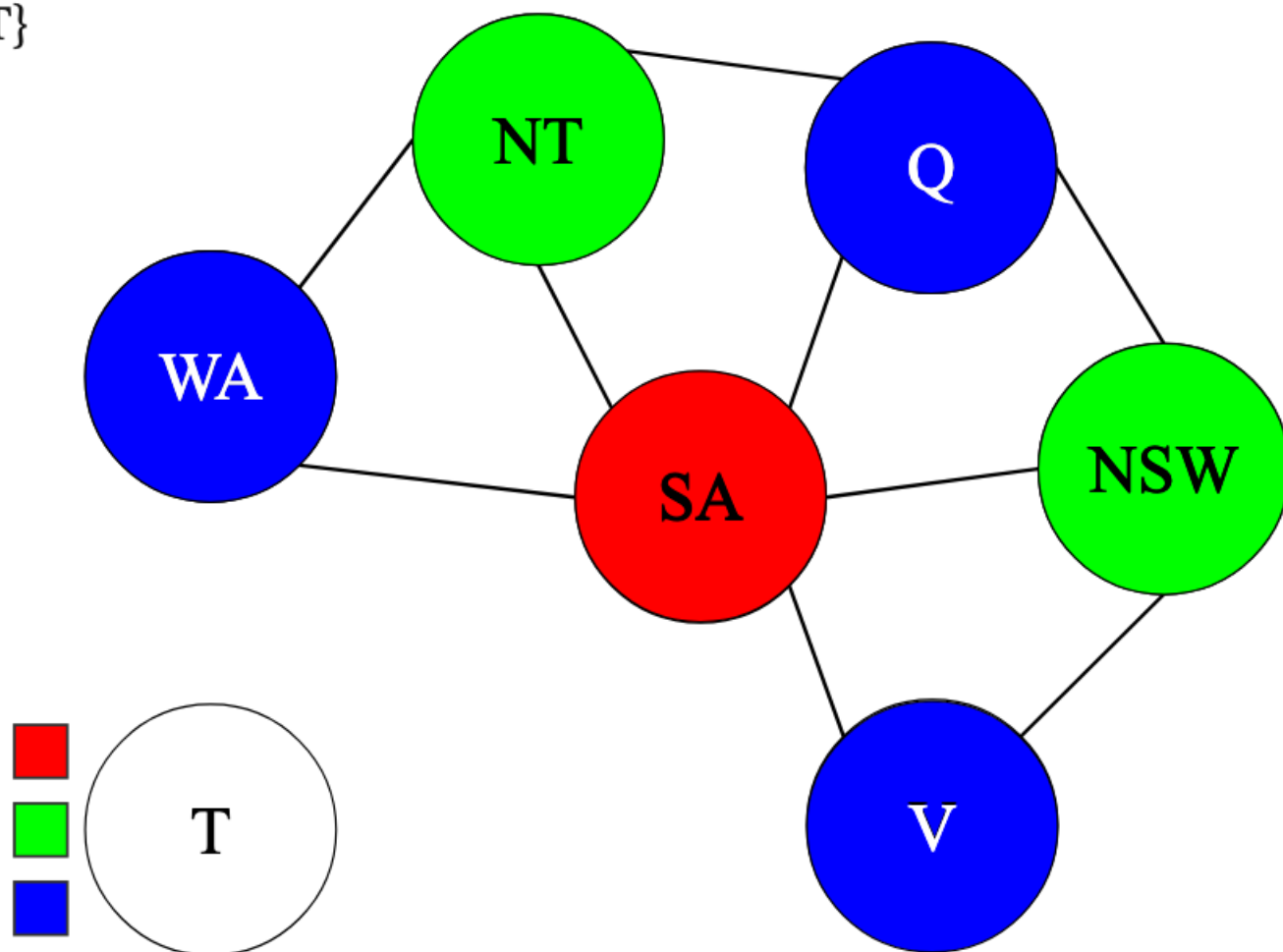


Backtracking with All Heuristics

AC-3 · MRV · Degree · LCV

Variables: {NSW, WA, NT, Q, SA, V, T}

Colors: {red, green, blue}

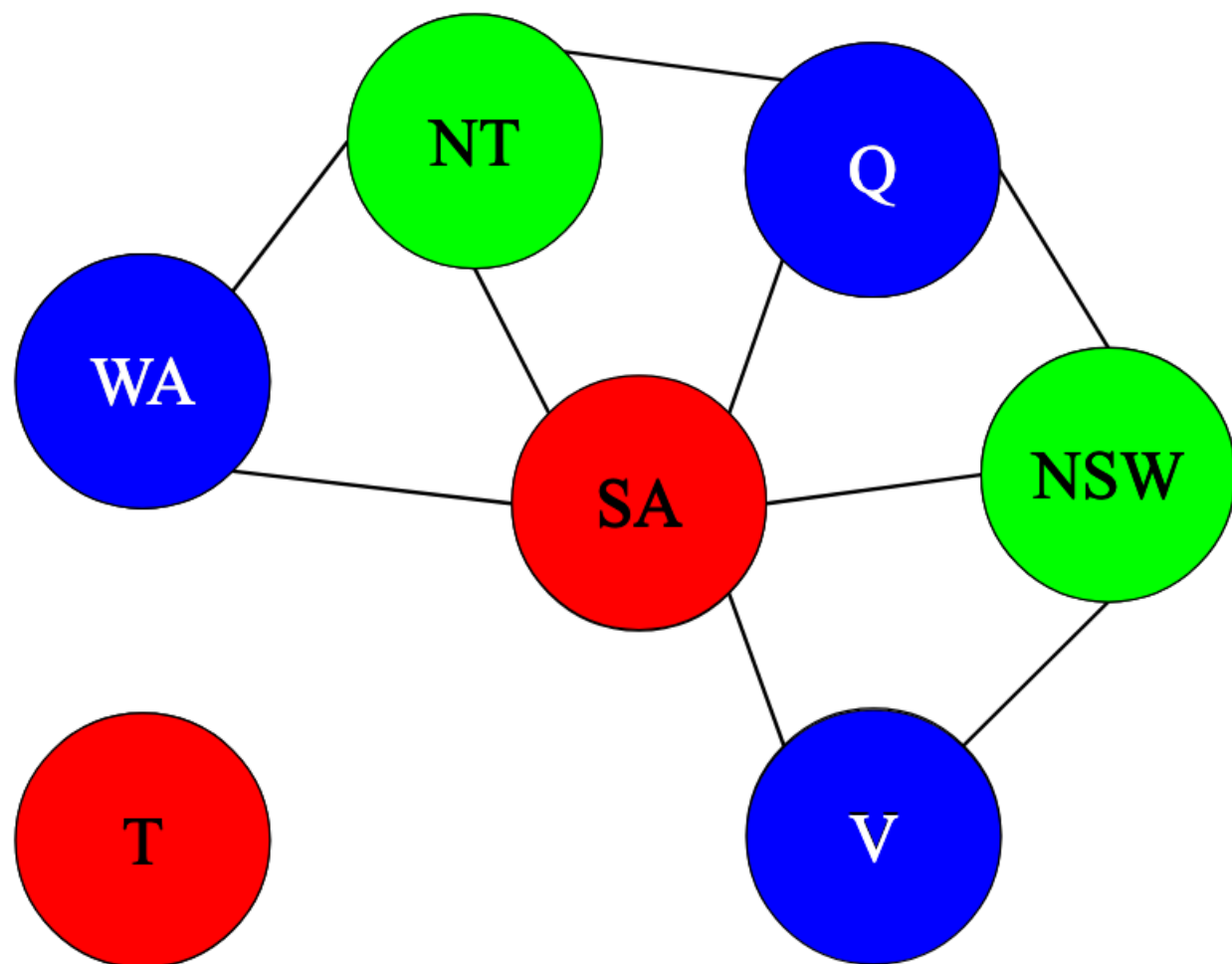


Backtracking with All Heuristics

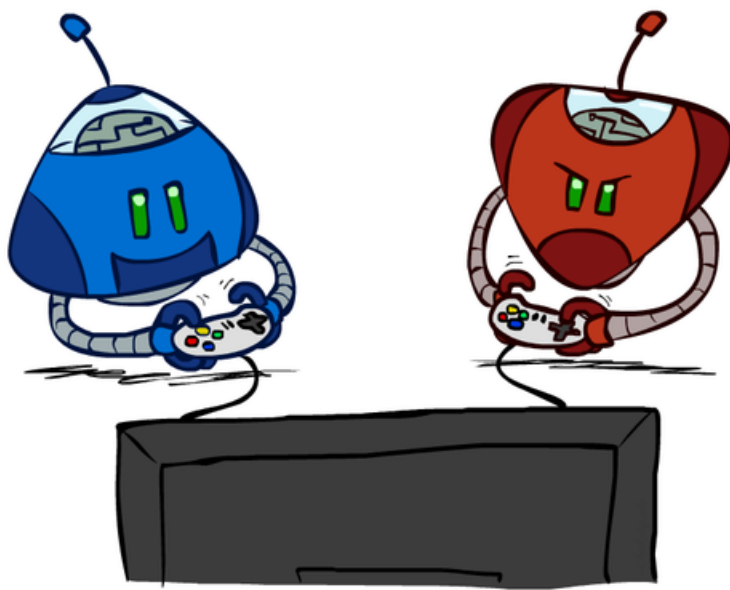
AC-3 · MRV · Degree · LCV

Variables: {NSW, WA, NT, Q, SA, V, T}

Colors: {red, green, blue}



Adversarial Search

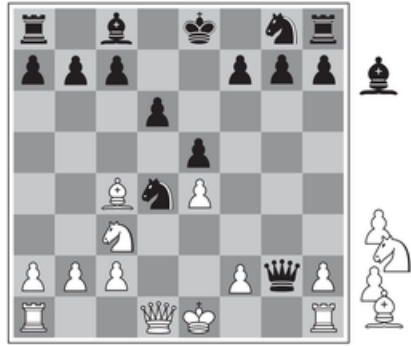


Other Agents Choose Their Own Moves

Our next state depends on our move *and their response*

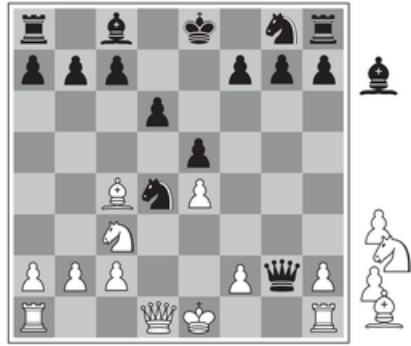
Other Agents Choose Their Own Moves

Our next state depends on our move *and their response*



Other Agents Choose Their Own Moves

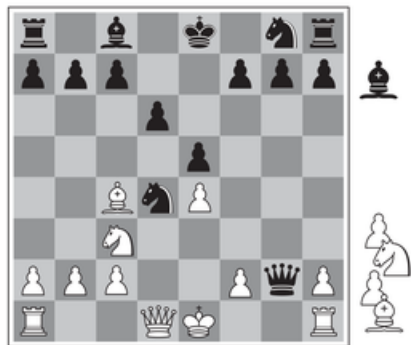
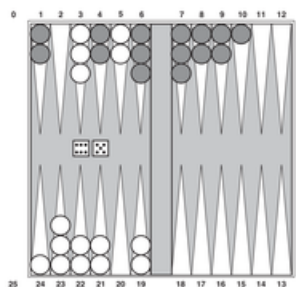
Our next state depends on our move *and their response*



X	O	
	X	
	O	

Other Agents Choose Their Own Moves

Our next state depends on our move *and their response*

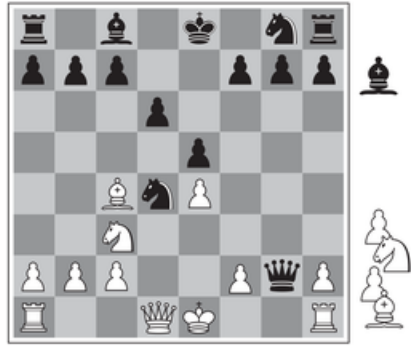


Chance: Later

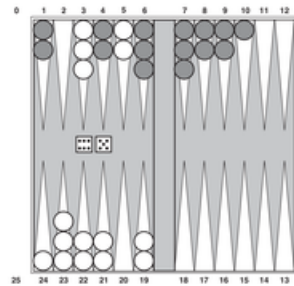
X	O	
	X	
	O	

Other Agents Choose Their Own Moves

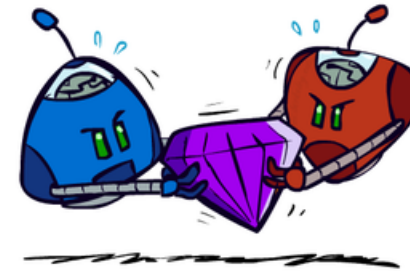
Our next state depends on our move *and their response*



X	O	
	X	
	O	

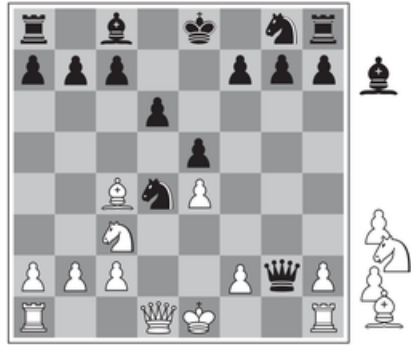


Chance: Later

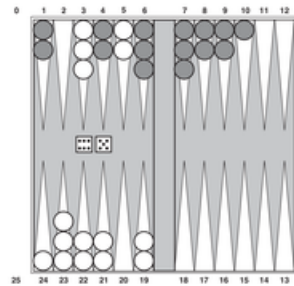


Other Agents Choose Their Own Moves

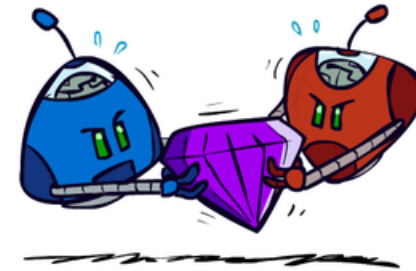
Our next state depends on our move *and their response*



X	O	
	X	
	O	



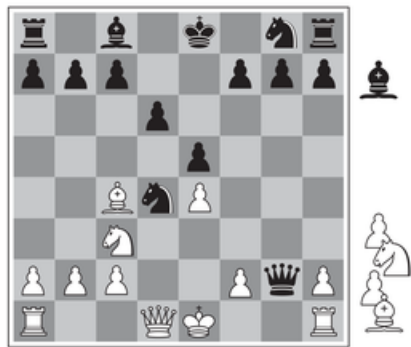
Chance: Later



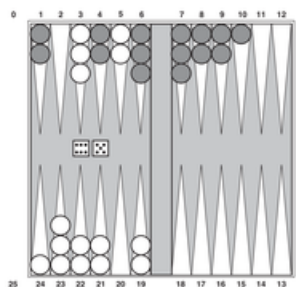
Here: Two Players · Alternating Turns · Deterministic · Fully Observable

Other Agents Choose Their Own Moves

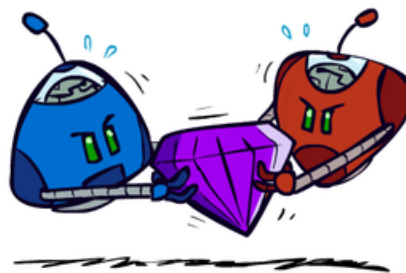
Our next state depends on our move *and their response*



X	O	
	X	
	O	



Chance: Later



Here: Two Players · Alternating Turns · Deterministic · Fully Observable

Zero-Sum: What Helps MAX Hurts MIN

Defining a Game

State s ; Initial State s_0

Board configuration *and player to move*

Player(s)

Whose turn is it?

Actions(s); Result(s, a)

Legal moves; state after a move

TerminalTest(s)

Has the game ended?

Utility(s, p)

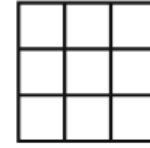
Final payoff for player p

Defining a Game

State s ; Initial State s_0	Board configuration <i>and player to move</i>
Player(s)	Whose turn is it?
Actions(s); Result(s, a)	Legal moves; state after a move
TerminalTest(s)	Has the game ended?
Utility(s, p)	Final payoff for player p

Game Tree: Nodes Are States; Edges Are Moves

Tic-Tac-Toe Game Tree



Players

X is MAX

O is MIN

Utility for X

Win: +1

Draw: 0

Loss: -1

Tic-Tac-Toe Game Tree

Players

X is MAX

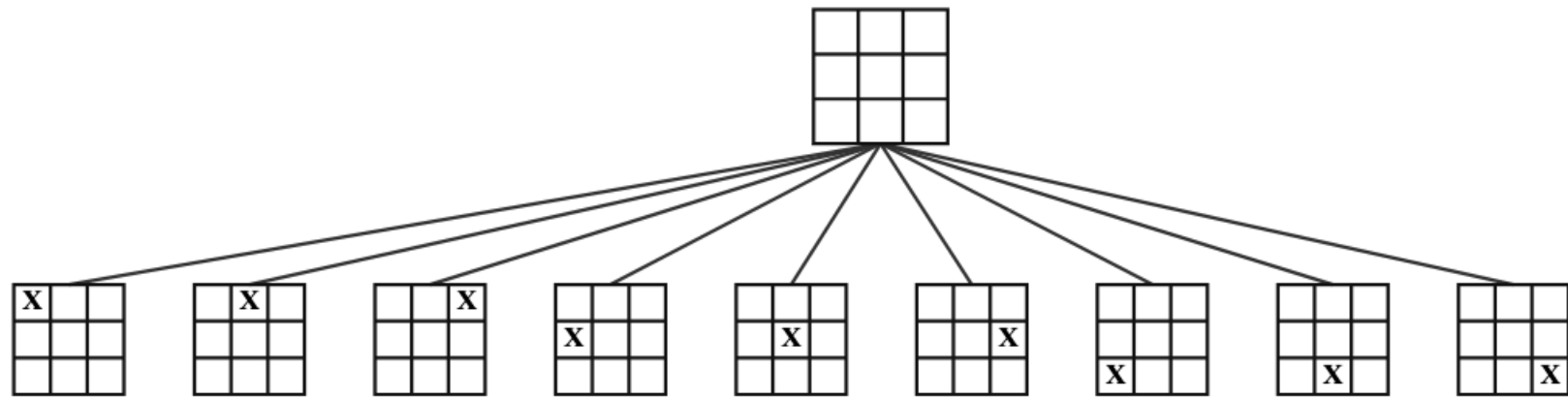
O is MIN

Utility for X

Win: +1

Draw: 0

Loss: -1



Tic-Tac-Toe Game Tree

Players

X is MAX

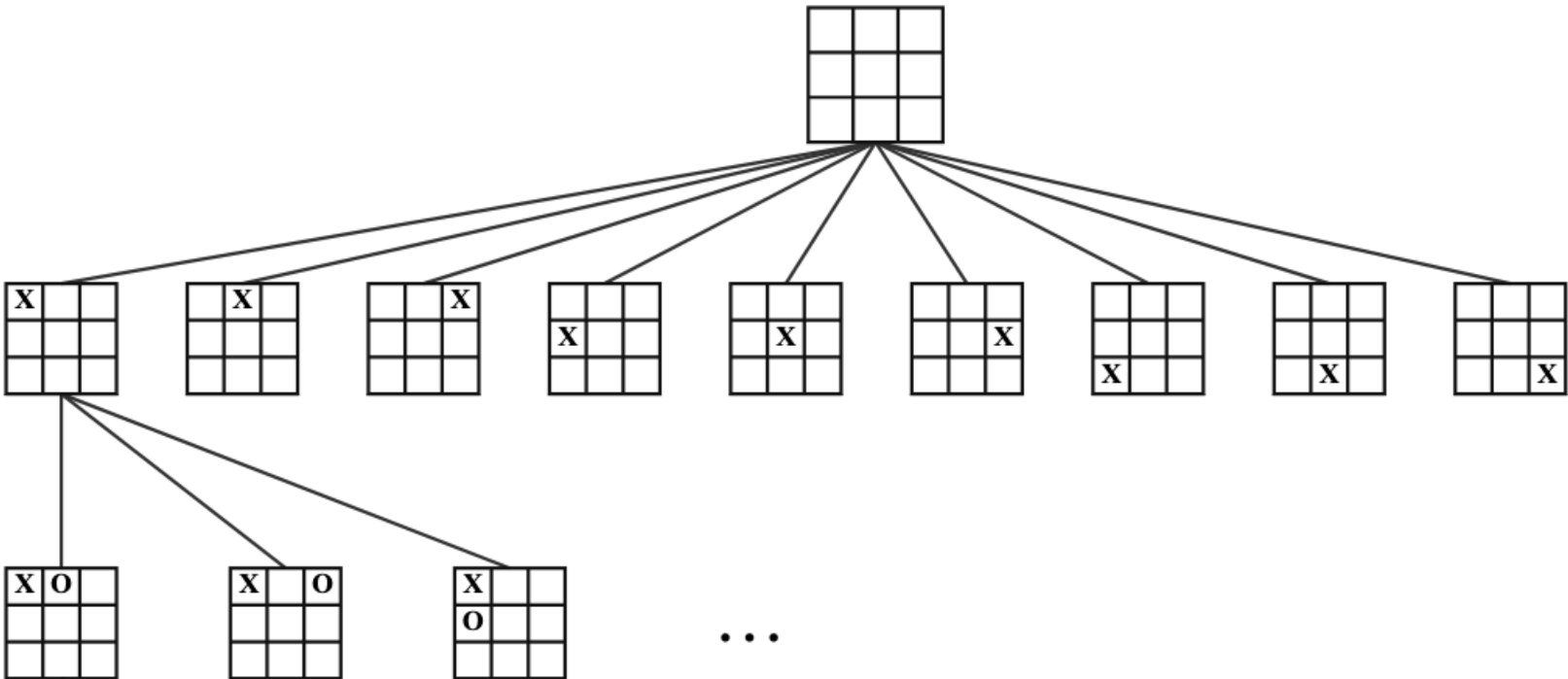
O is MIN

Utility for X

Win: +1

Draw: 0

Loss: -1



Tic-Tac-Toe Game Tree

Players

X is MAX

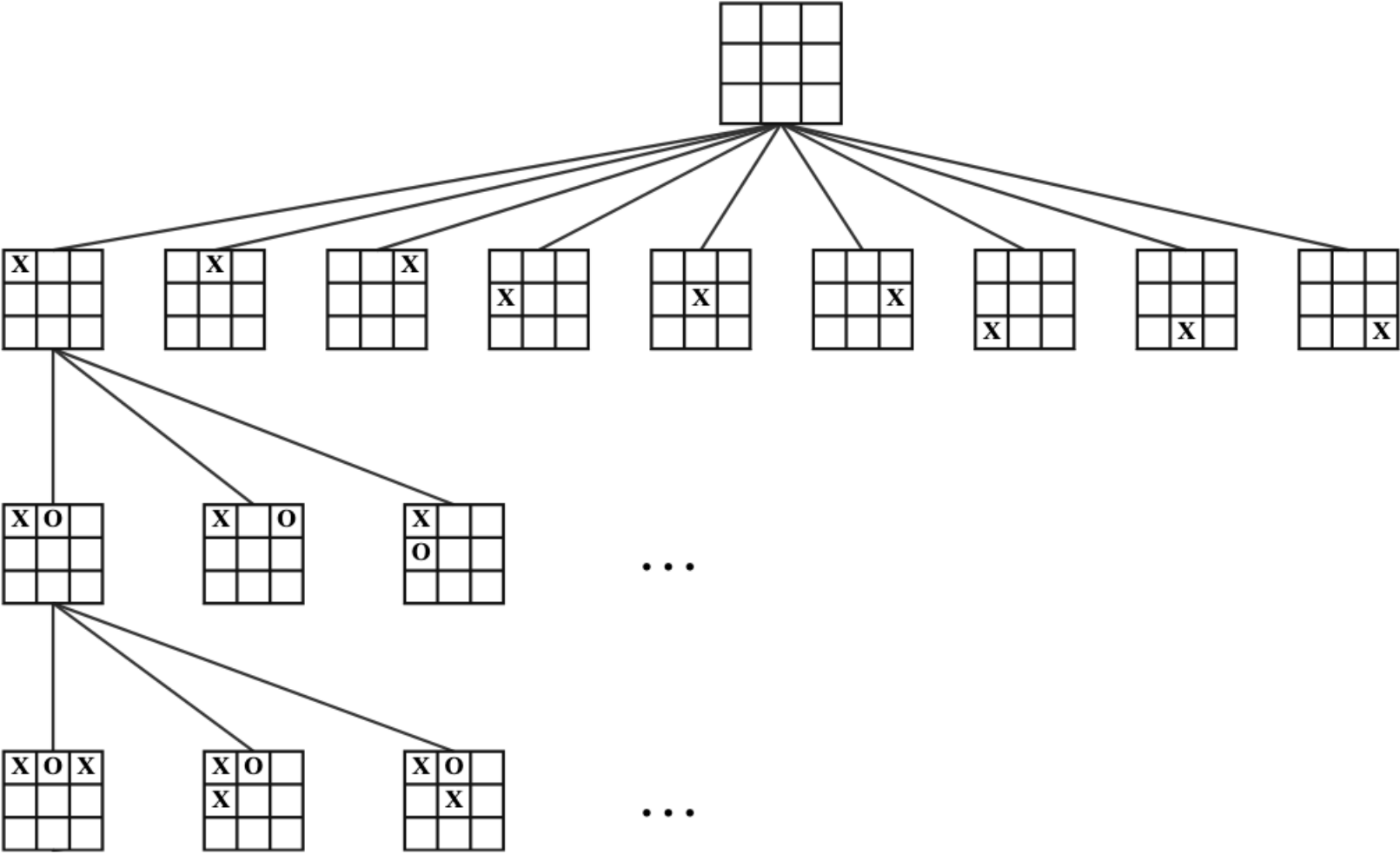
O is MIN

Utility for X

Win: +1

Draw: 0

Loss: -1



Tic-Tac-Toe Game Tree

Players

X is MAX

O is MIN

Utility for X

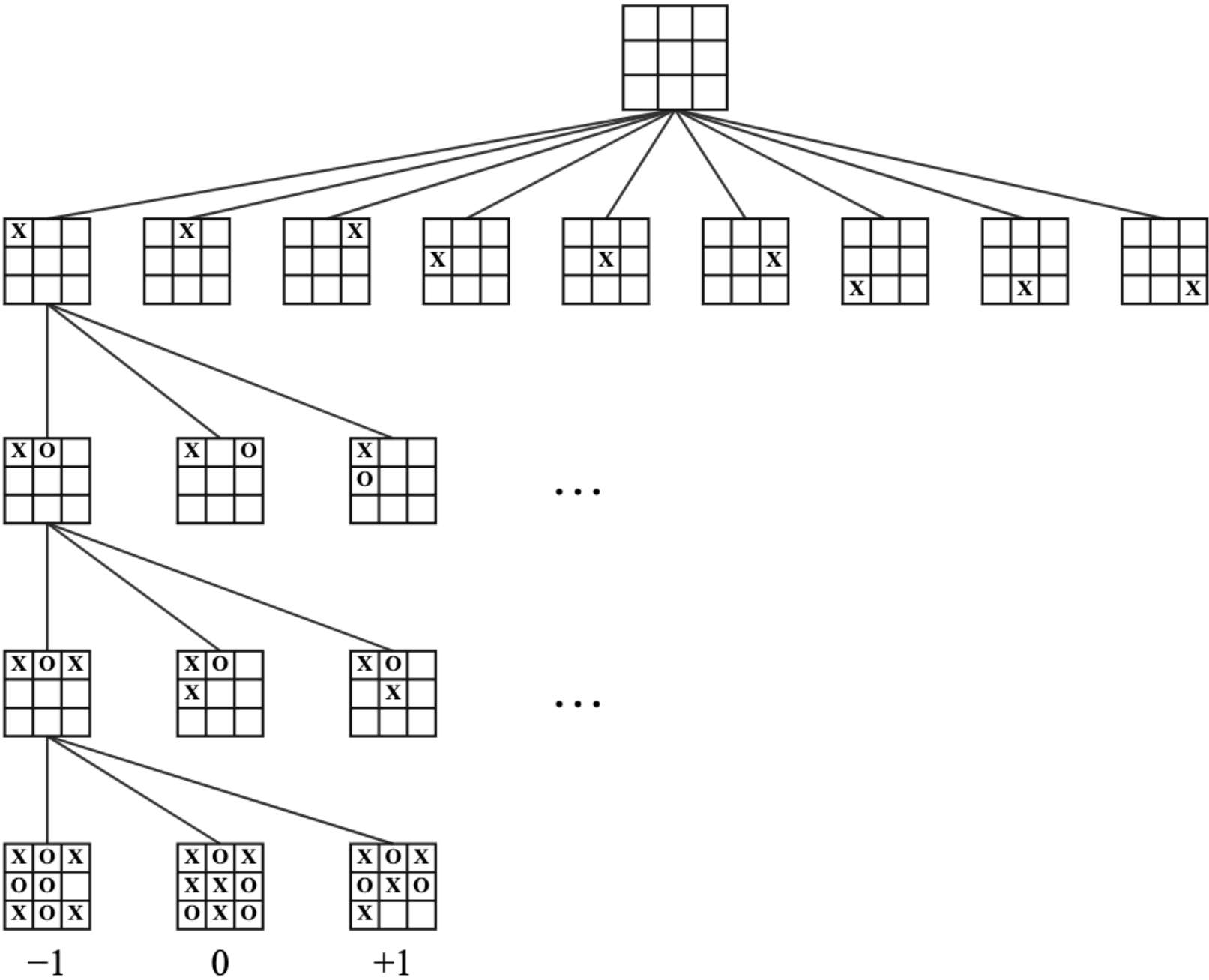
Win: +1

Draw: 0

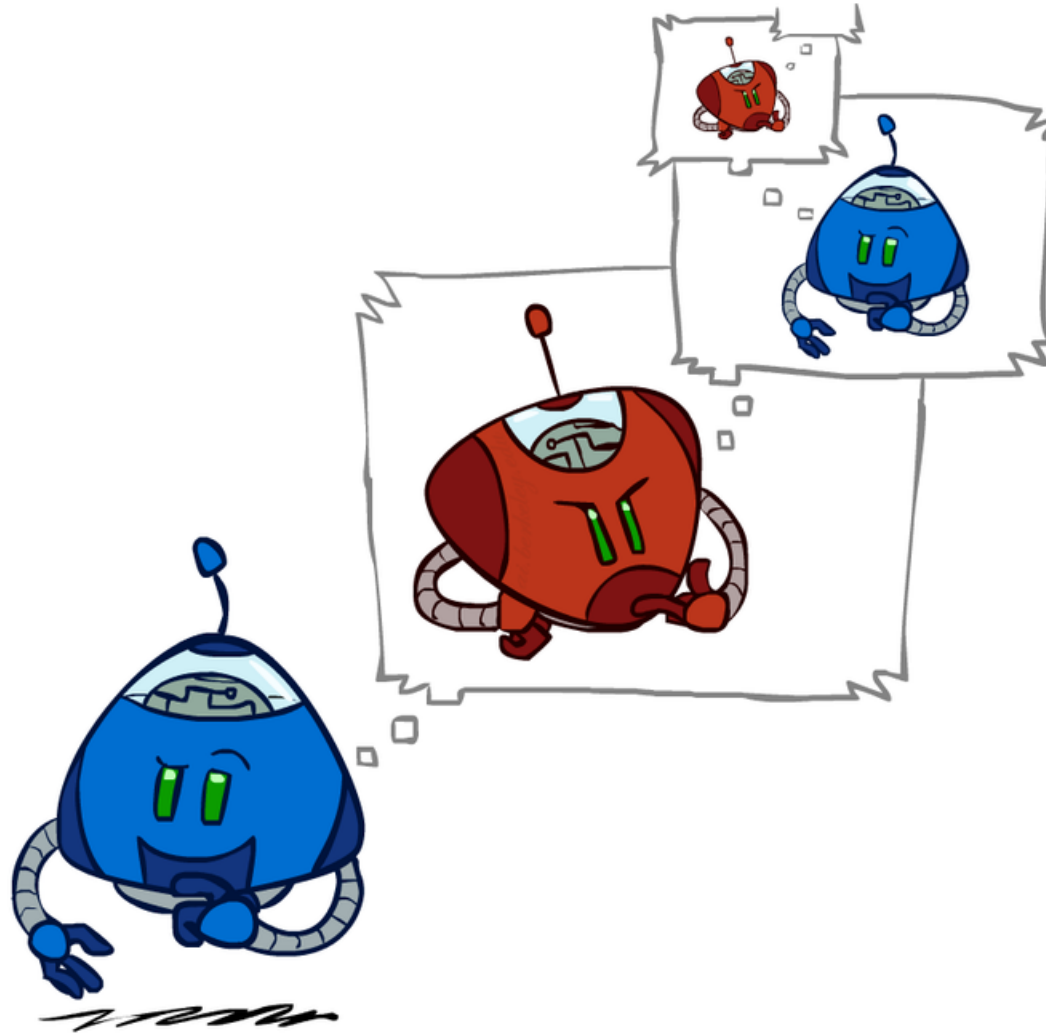
Loss: -1

State Includes
the Player to

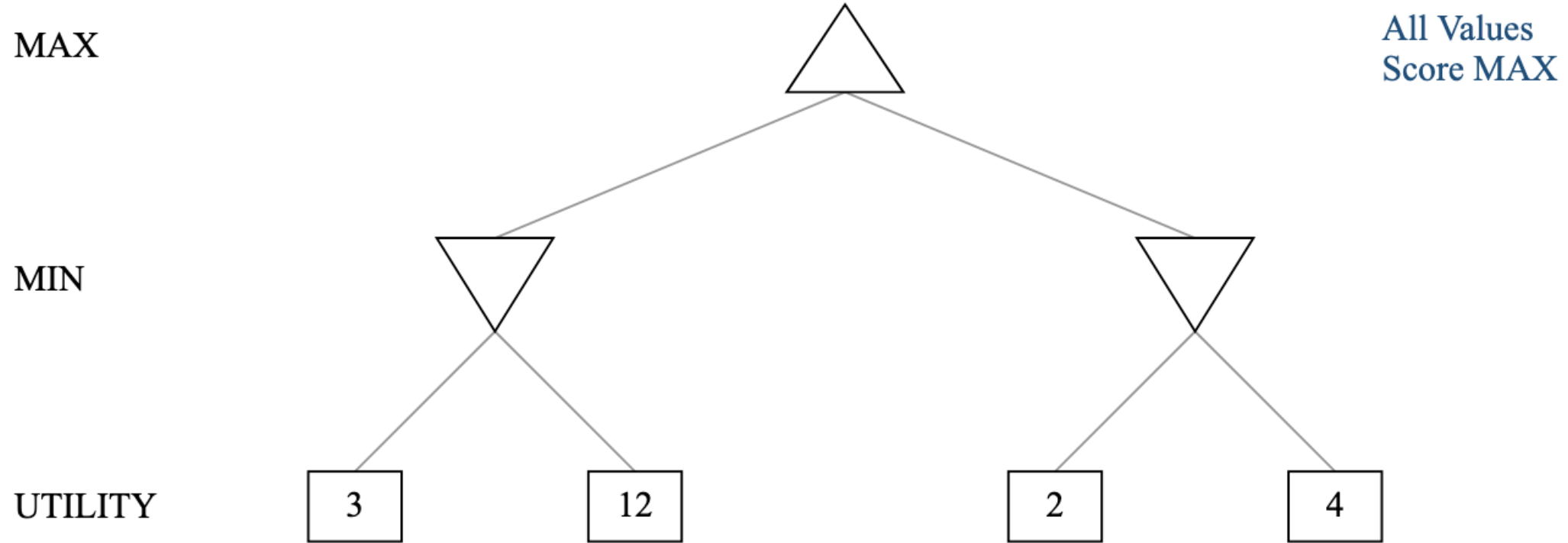
Move



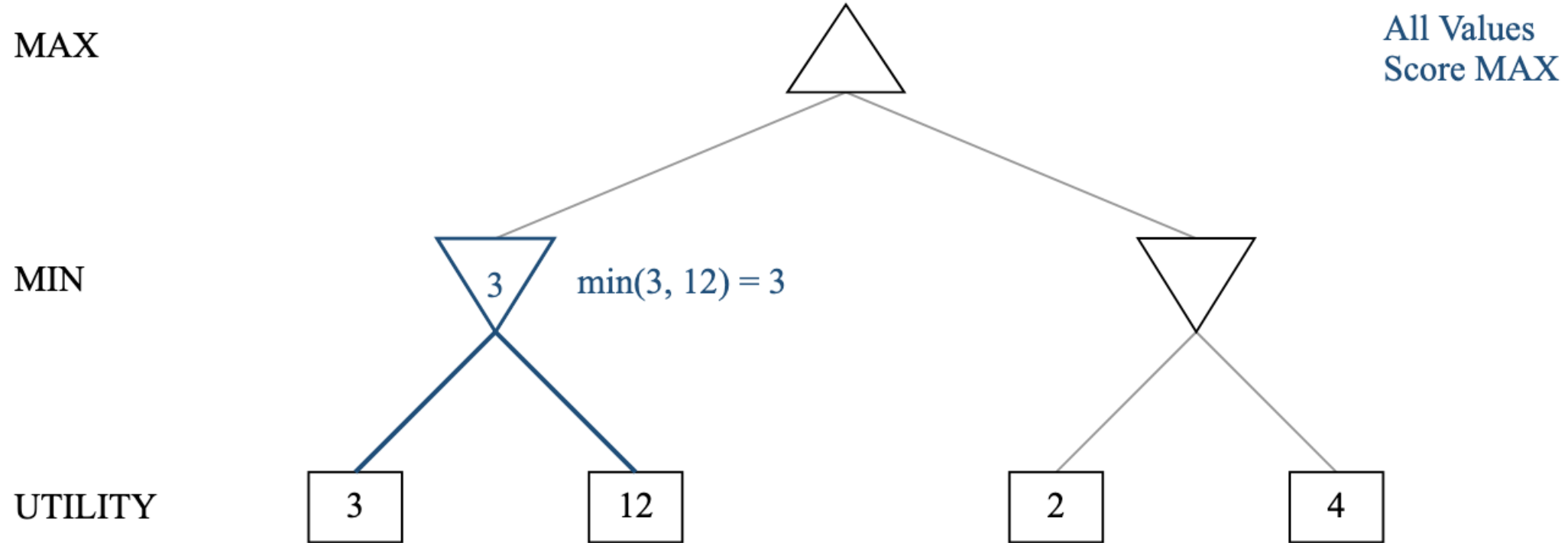
Think About Their Reply



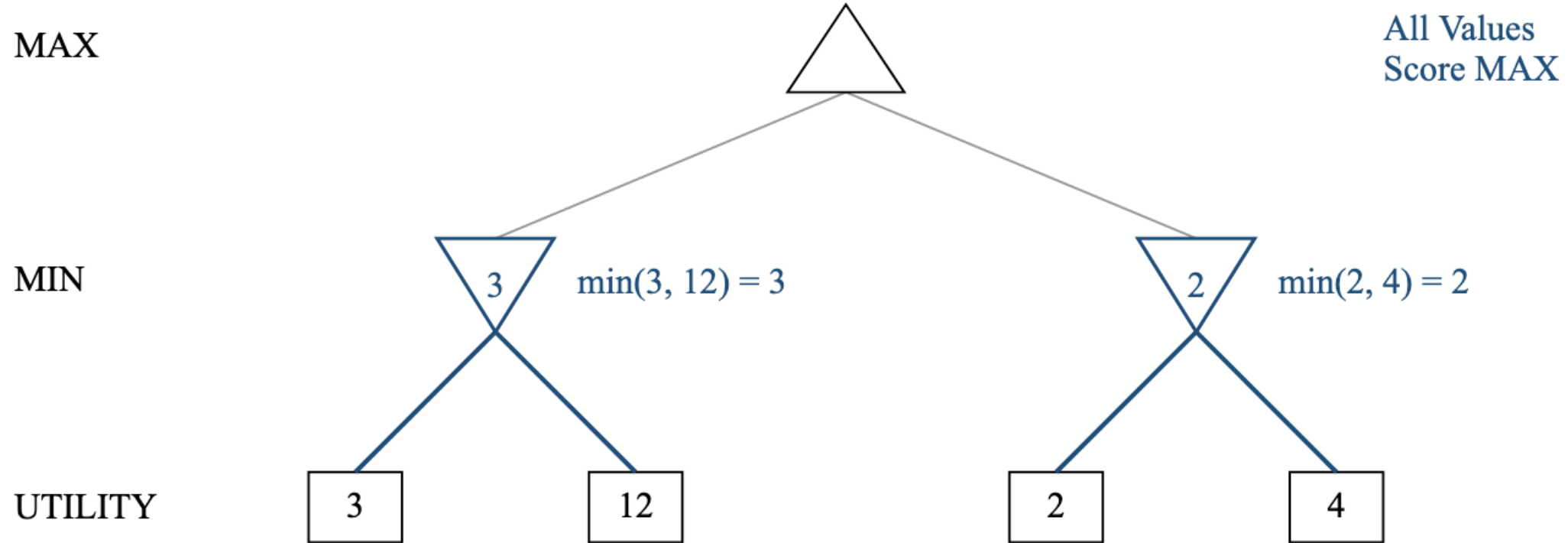
Back Up Values from the Leaves



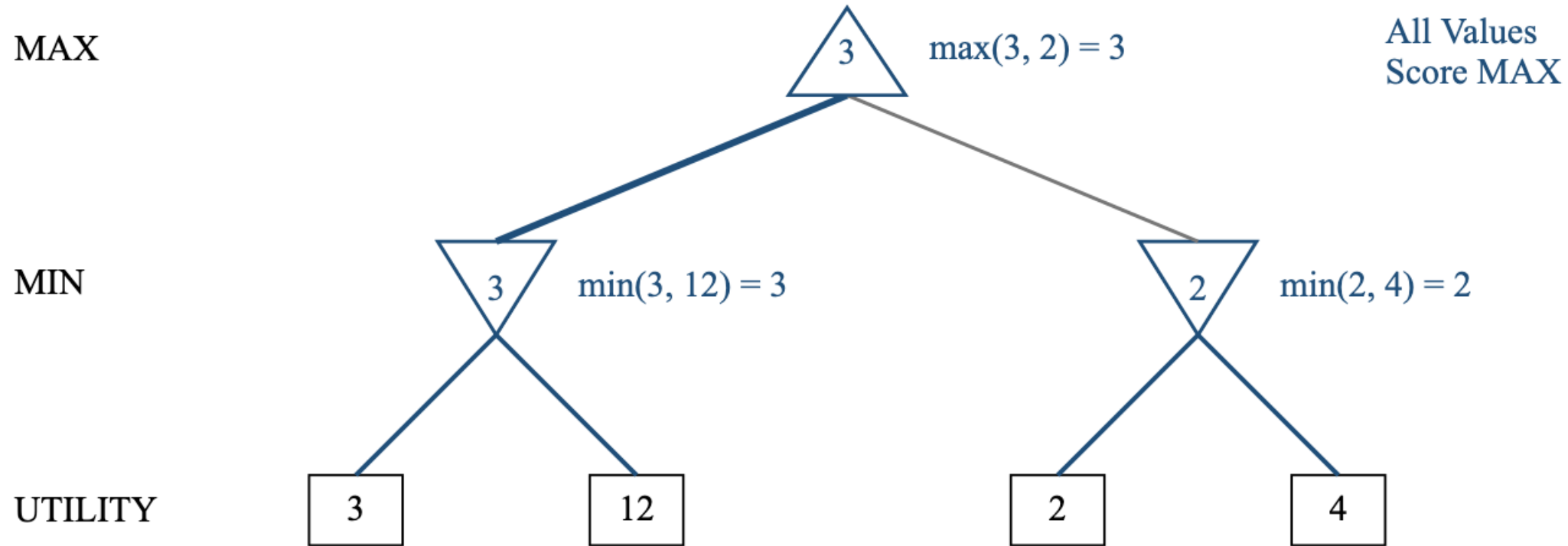
Back Up Values from the Leaves



Back Up Values from the Leaves



Back Up Values from the Leaves



MAX Chooses Left: 3 Is Larger Than 2

Minimax Follows the Tree Recursively

```
def minimax(tree, maximizing=True):
```

Minimax Follows the Tree Recursively

```
def minimax(tree, maximizing=True):  
  
    if not isinstance(tree, list): return tree
```

Base Case

A leaf returns its utility

Minimax Follows the Tree Recursively

```
def minimax(tree, maximizing=True):  
  
    if not isinstance(tree, list): return tree  
  
    values = [minimax(child, not maximizing) for child in tree]
```

Base Case

A leaf returns its utility

Recursive Step

Visit every child and switch the player

Minimax Follows the Tree Recursively

```
def minimax(tree, maximizing=True):  
  
    if not isinstance(tree, list): return tree  
  
    values = [minimax(child, not maximizing) for child in tree]  
  
    return max(values) if maximizing else min(values)
```

Base Case

A leaf returns its utility

Recursive Step

Visit every child and switch the
player

Back Up

MAX returns max; MIN returns min

One Recursive Call per State in the Game Tree

Run Minimax on a Deeper Tree

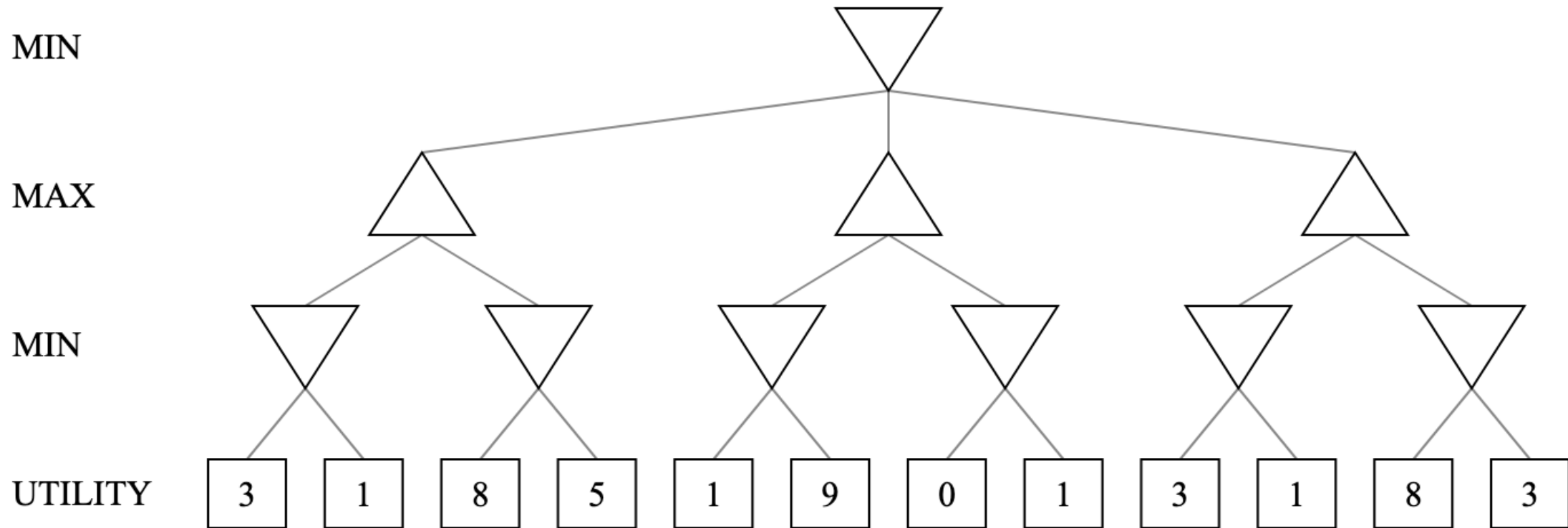
PY main.py

```
1 def minimax(tree, maximizing=True):
2     if not isinstance(tree, list): return tree
3     values = [minimax(child, not maximizing) for child in tree]
4     return max(values) if maximizing else min(values)
5 def node(*children): return list(children)
6 left = node(node(3, 5), node(6, 9))
7 middle = node(node(1, 2), node(0, -1))
8 right = node(node(7, 4), node(5, 8))
9 tree = node(left, middle, right)
10 values = [minimax(child, False) for child in tree]
11 best = max(range(len(tree)), key=lambda i: values[i])
12 print("Child values:", values, "Root value:", minimax(tree))
13 print("Best move:", ["Left", "Middle", "Right"][best])
14
```

[Download: minimax.py](#)

Minimax Practice

MIN Moves First: What Is the Root Value? Which Move?



TASK

TASK

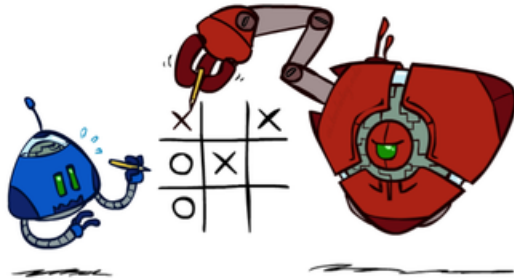
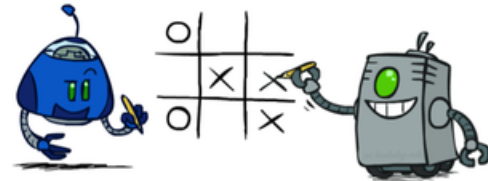


TASK

TASK

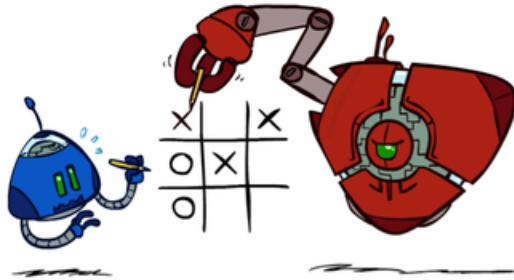
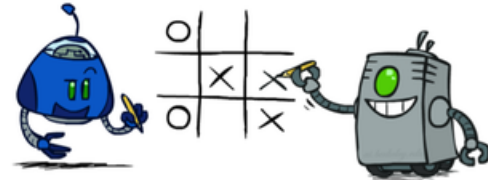
What Does Minimax Guarantee?

An Optimal Strategy Against Optimal Opposition



What Does Minimax Guarantee?

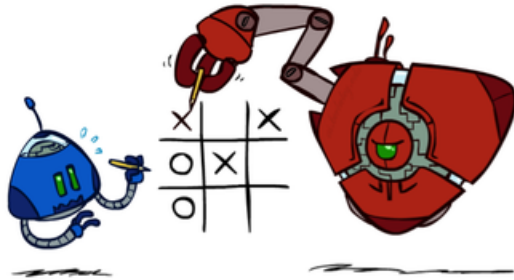
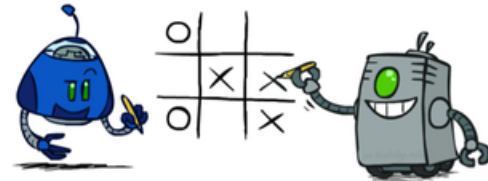
An Optimal Strategy Against Optimal Opposition



It Maximizes the Worst-Case Payoff

What Does Minimax Guarantee?

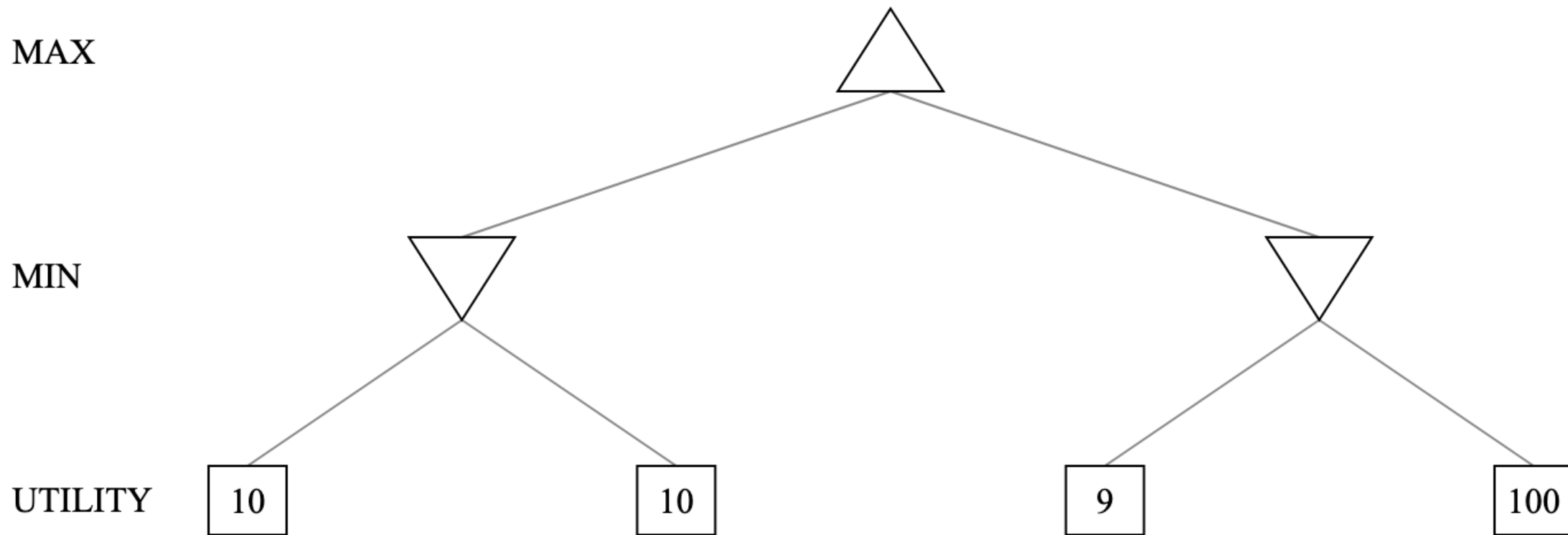
An Optimal Strategy Against Optimal Opposition



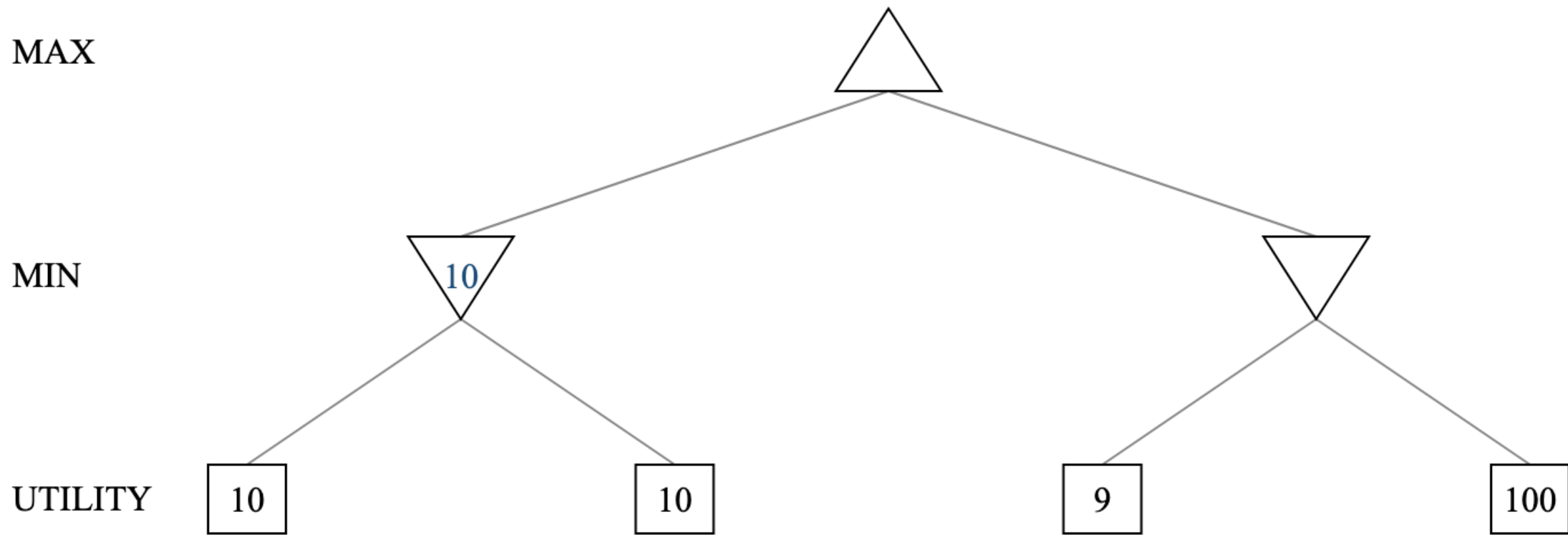
It Maximizes the Worst-Case Payoff

It Does Not Predict Every Opponent's Actual Move

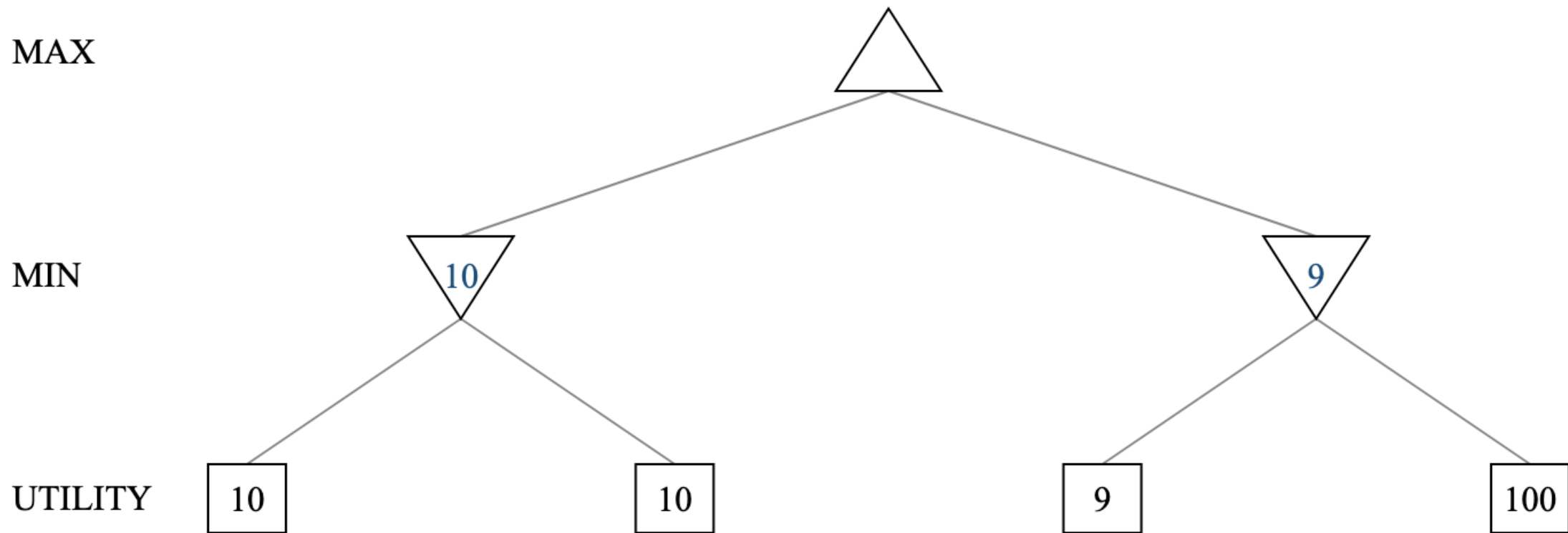
What If MIN Makes Mistakes?



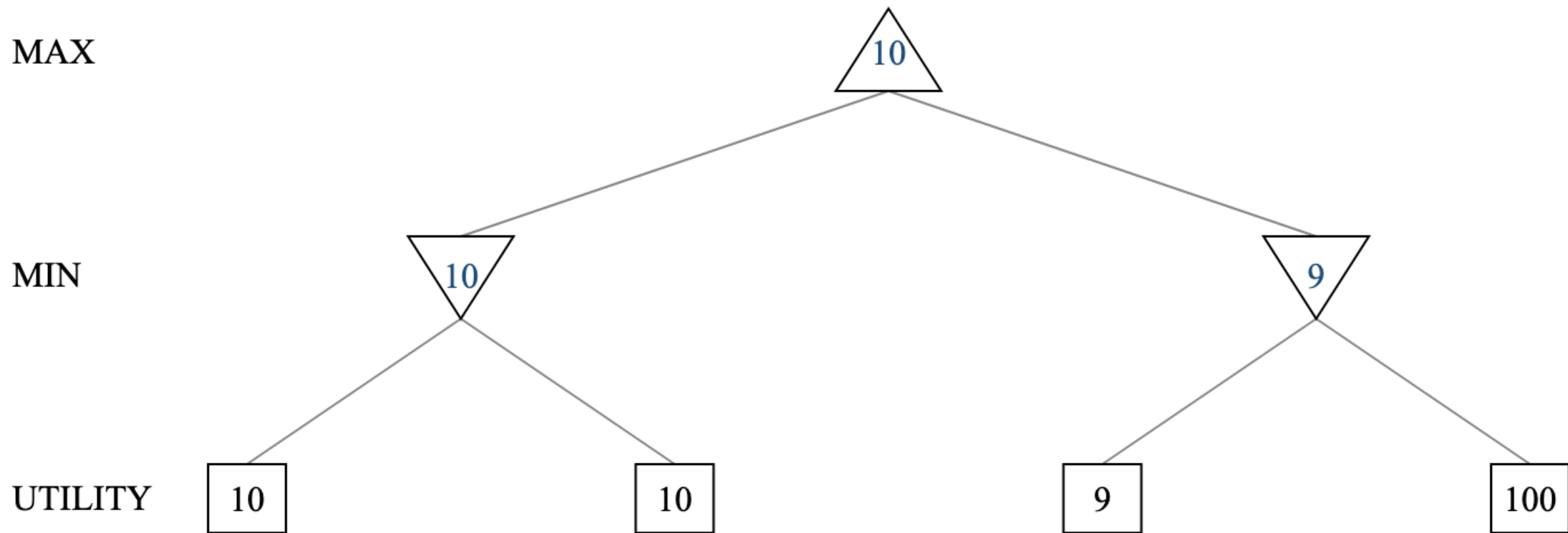
What If MIN Makes Mistakes?



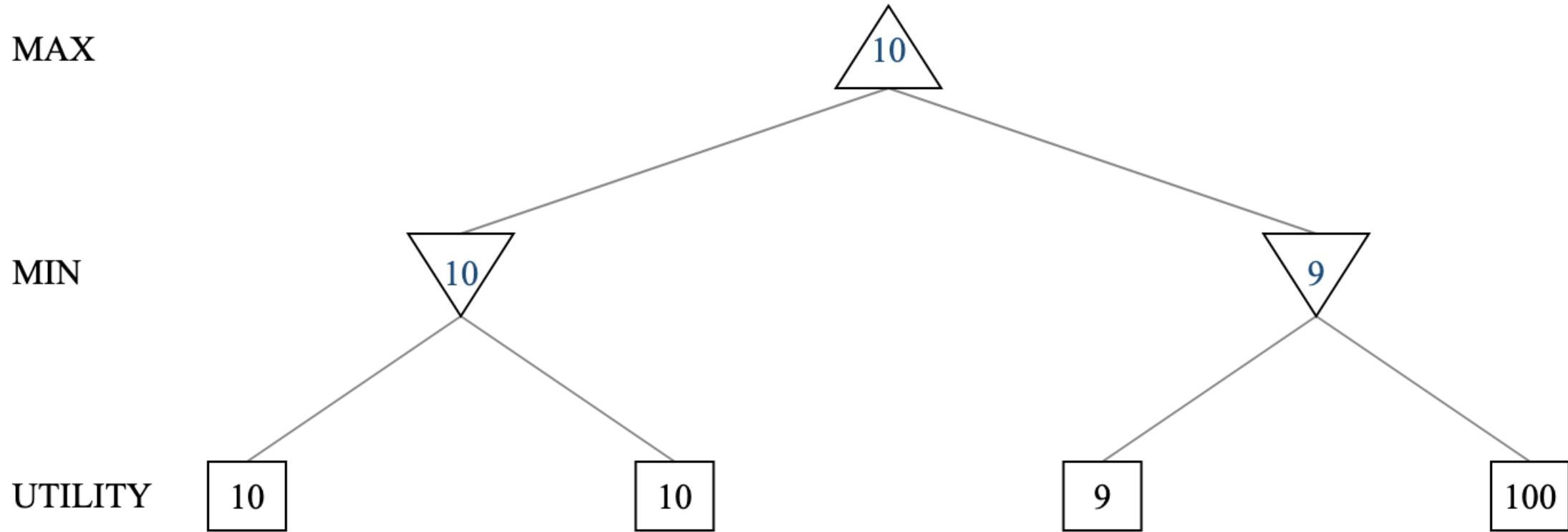
What If MIN Makes Mistakes?



What If MIN Makes Mistakes?

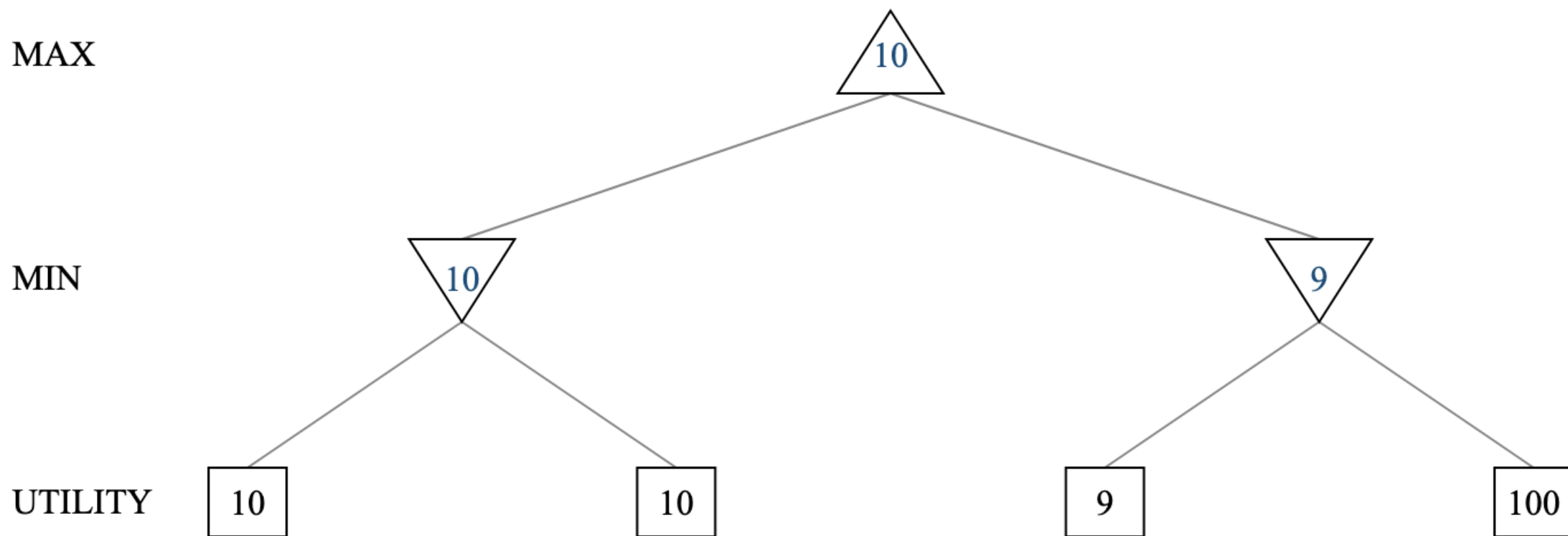


What If MIN Makes Mistakes?



Left Guarantees 10 · Right Guarantees 9

What If MIN Makes Mistakes?



Left Guarantees 10 · Right Guarantees 9

If MIN Chooses Randomly: Right Averages 54.5

Minimax Properties

Finite Game Tree · Branching Factor b · Maximum Depth m

Search All Replies, Depth First

Minimax Properties

Finite Game Tree · Branching Factor b · Maximum Depth m

Complete?

Yes, for a finite game tree

Search All Replies, Depth First

Minimax Properties

Finite Game Tree · Branching Factor b · Maximum Depth m

Complete?

Yes, for a finite game tree

Optimal?

Yes, against optimal opposition

Search All Replies, Depth First

Minimax Properties

Finite Game Tree · Branching Factor b · Maximum Depth m

Complete?

Yes, for a finite game tree

Optimal?

Yes, against optimal opposition

Time

$O(b^m)$

Search All Replies, Depth First

Minimax Properties

Finite Game Tree · Branching Factor b · Maximum Depth m

Complete?

Yes, for a finite game tree

Optimal?

Yes, against optimal opposition

Time

$O(b^m)$

Space

$O(bm)$, with DFS and stored successor lists

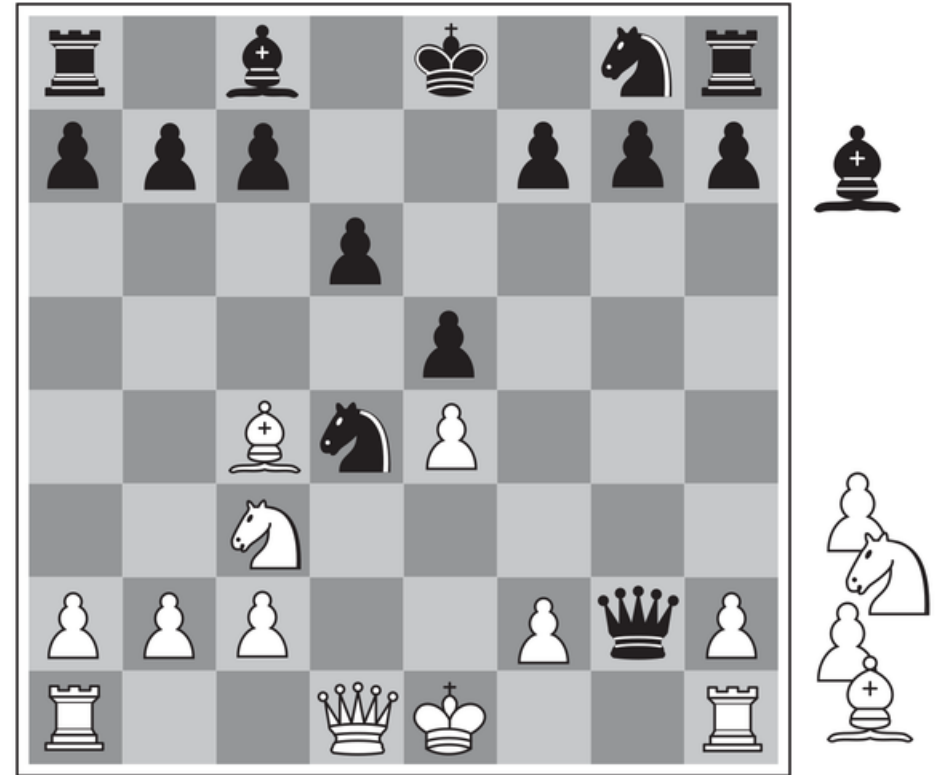
Search All Replies, Depth First

Full Search Is Too Large

Illustrative Chess Estimate

$$b \approx 35 \cdot m \approx 100$$

$$35^{100} \approx 2.6 \times 10^{154}$$



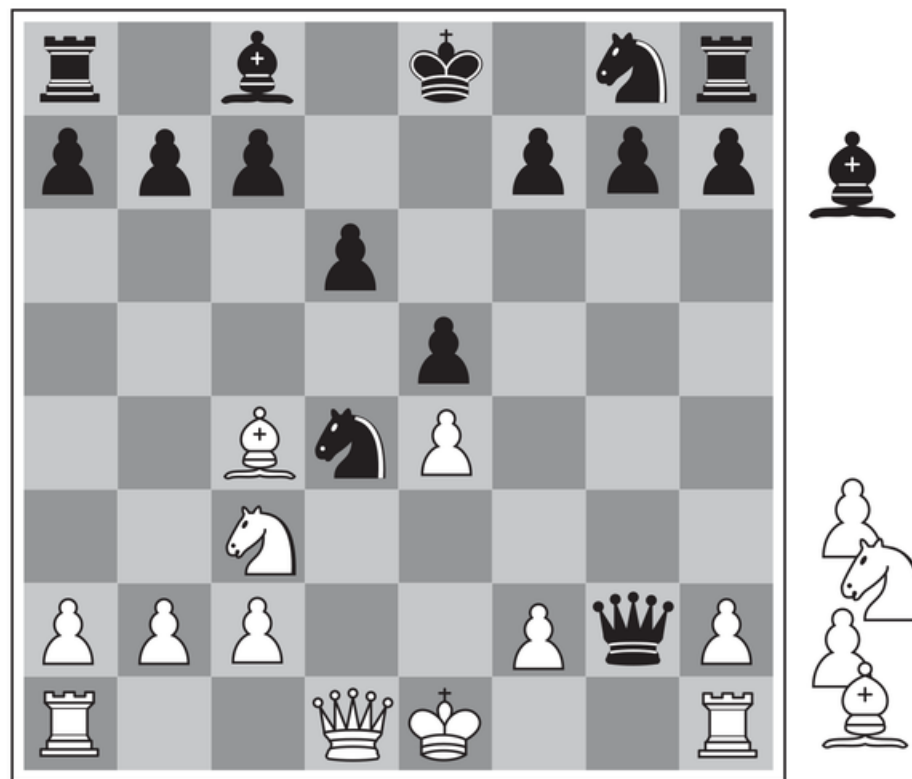
Full Search Is Too Large

Illustrative Chess Estimate

$$b \approx 35 \cdot m \approx 100$$

$$35^{100} \approx 2.6 \times 10^{154}$$

We Cannot Search to Every Terminal
Position

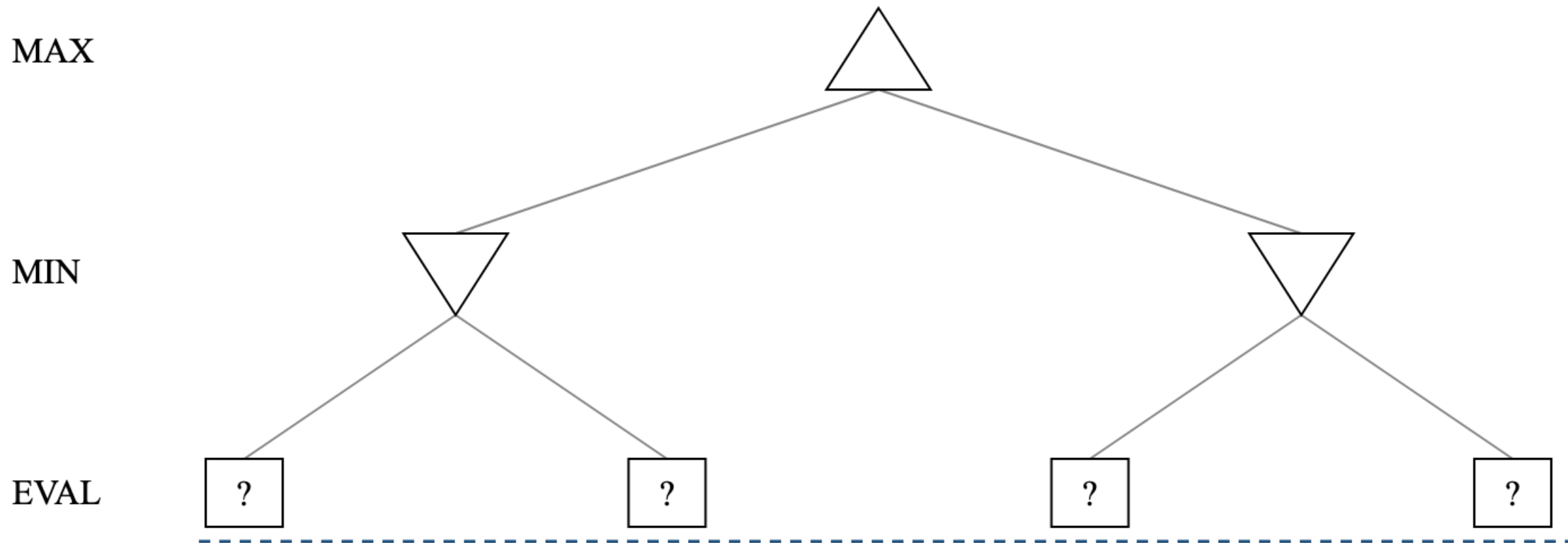


Resource Limits



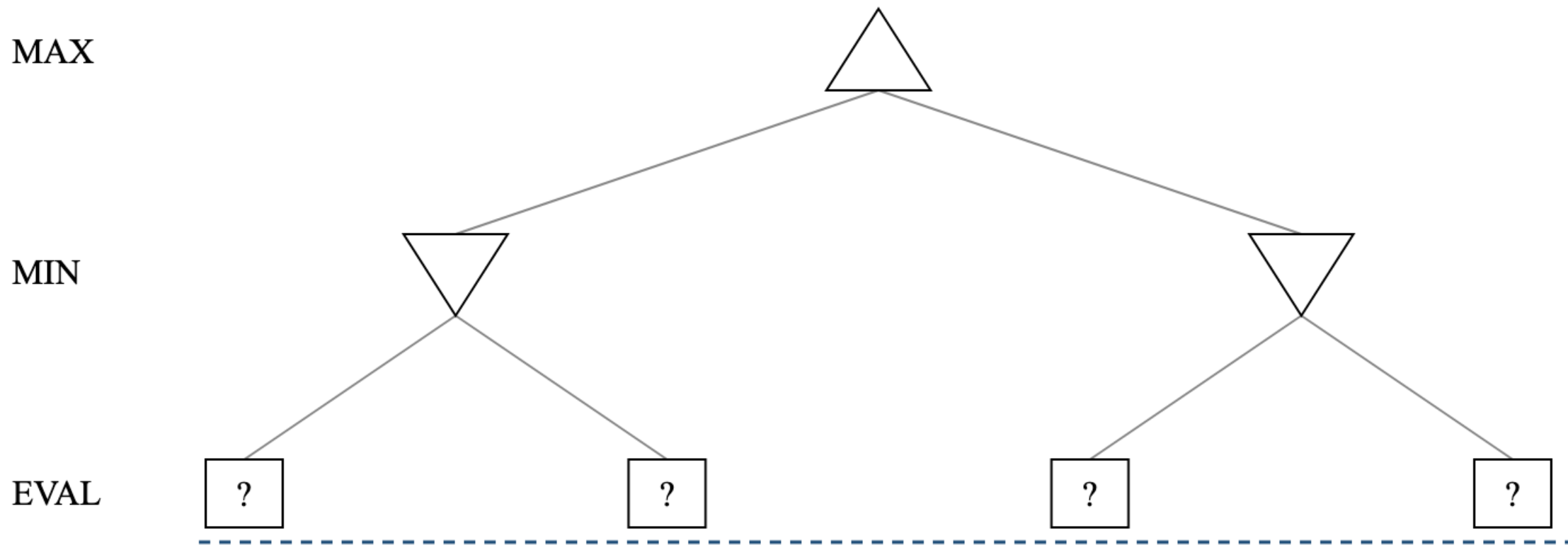
Stop at a Depth Limit

Depth Limit $\ell = 2 \cdot \text{MAX}$ and MIN Still Alternate



Stop at a Depth Limit

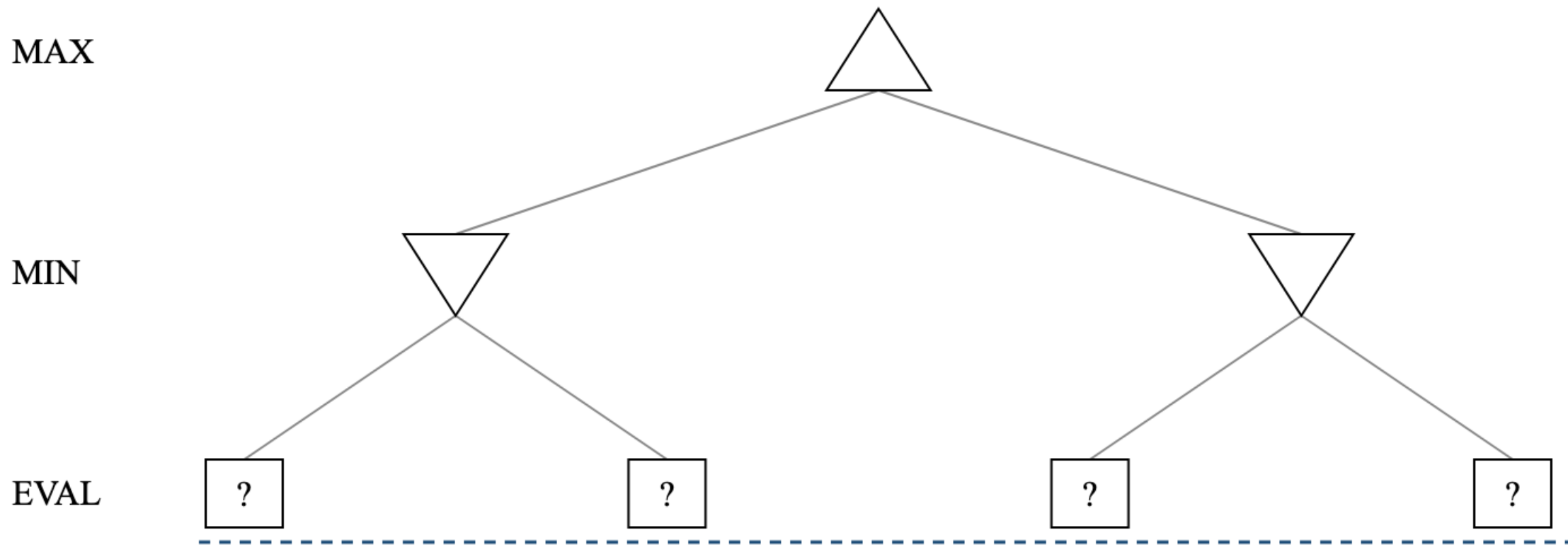
Depth Limit $\ell = 2 \cdot \text{MAX}$ and MIN Still Alternate



Terminal: Exact Utility · Cutoff: Estimate the Position

Stop at a Depth Limit

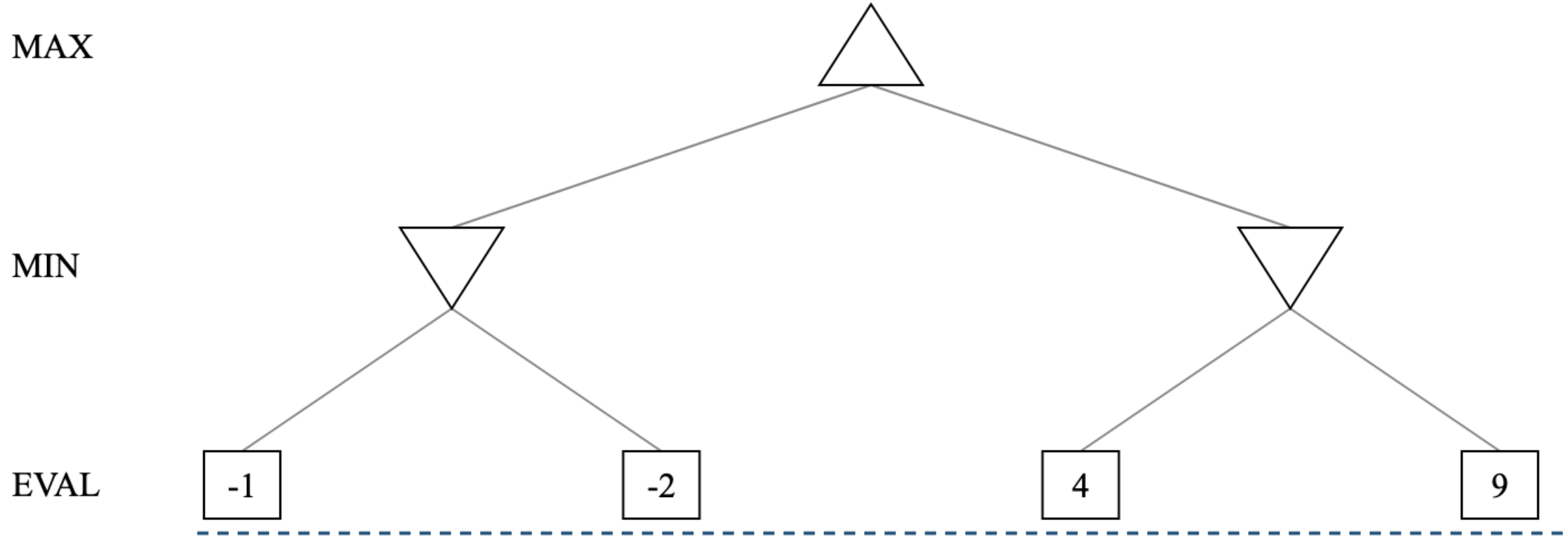
Depth Limit $\ell = 2 \cdot \text{MAX}$ and MIN Still Alternate



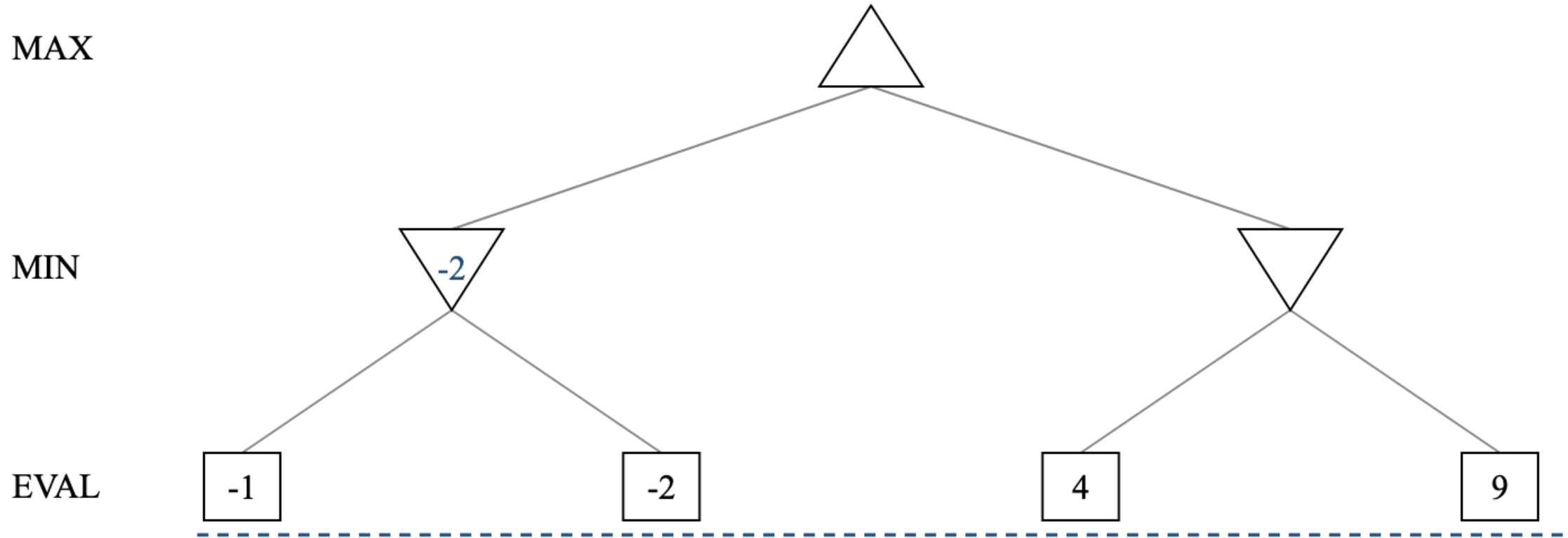
Terminal: Exact Utility · Cutoff: Estimate the Position

Time: $O(b^\ell)$ · Space: $O(b\ell)$

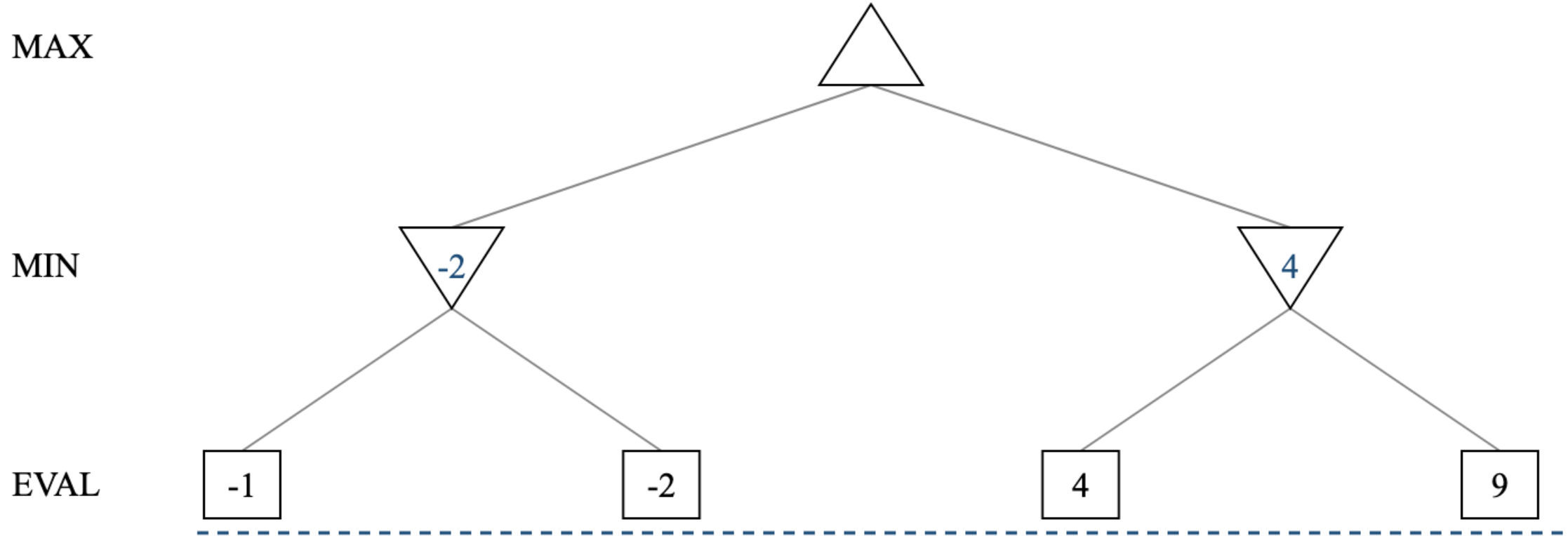
Back Up the Cutoff Estimates



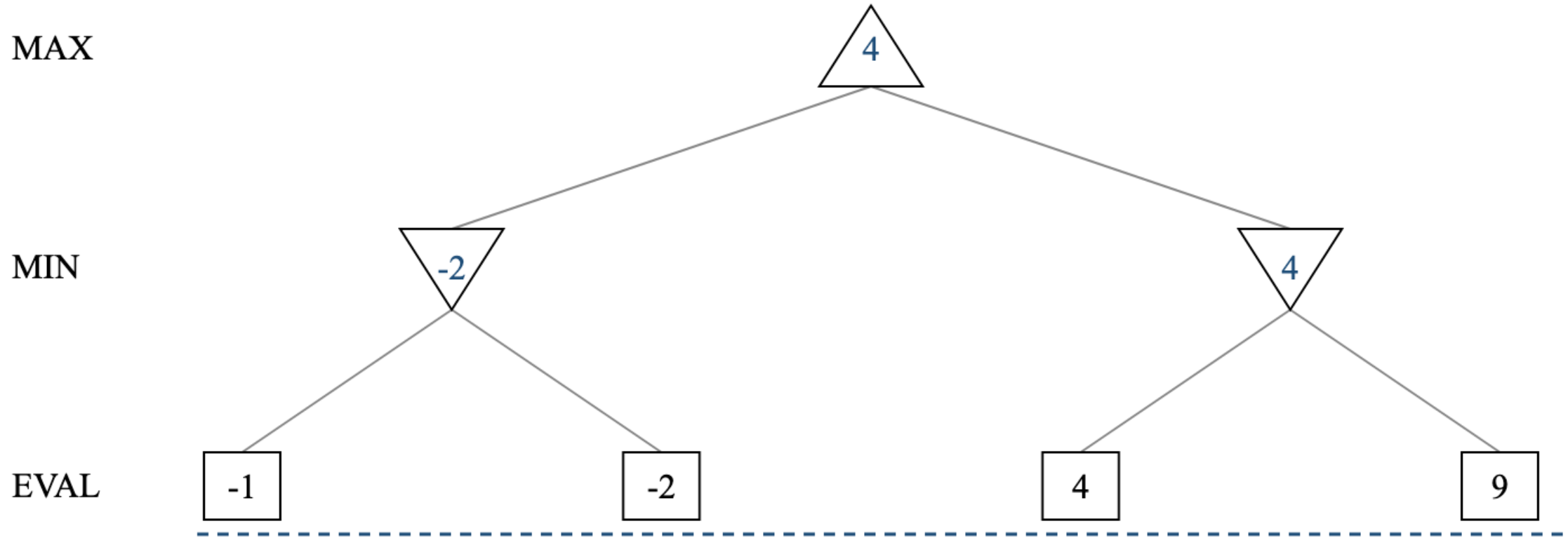
Back Up the Cutoff Estimates



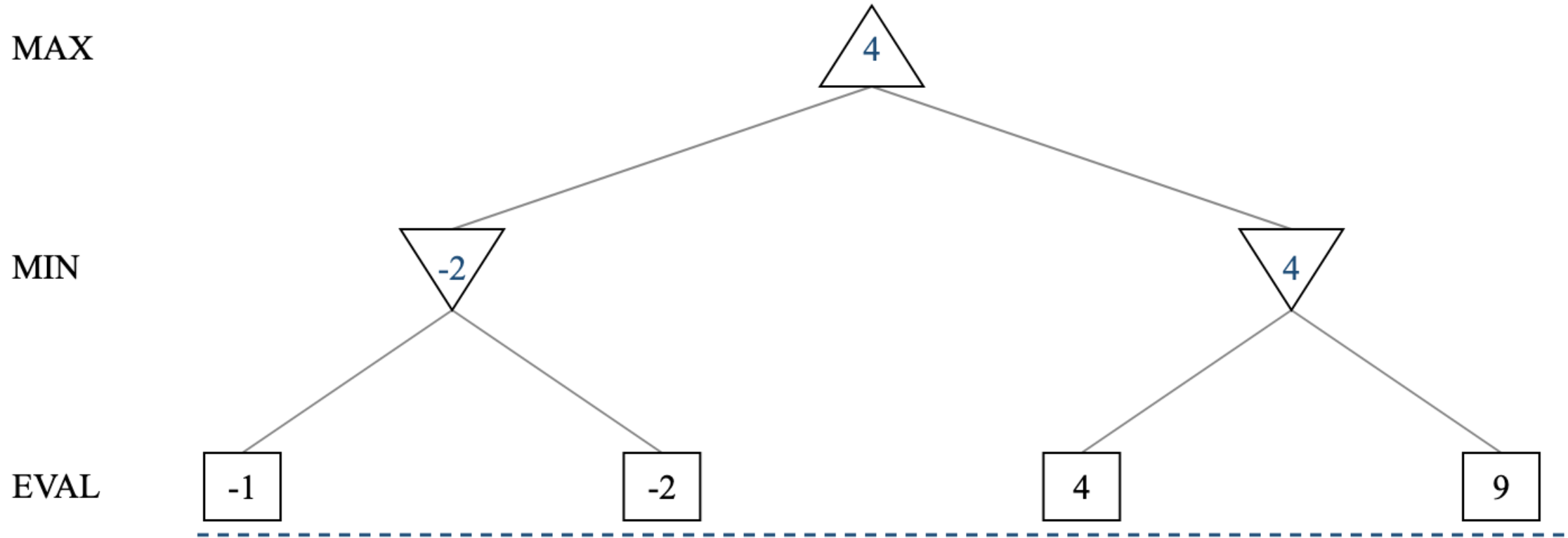
Back Up the Cutoff Estimates



Back Up the Cutoff Estimates

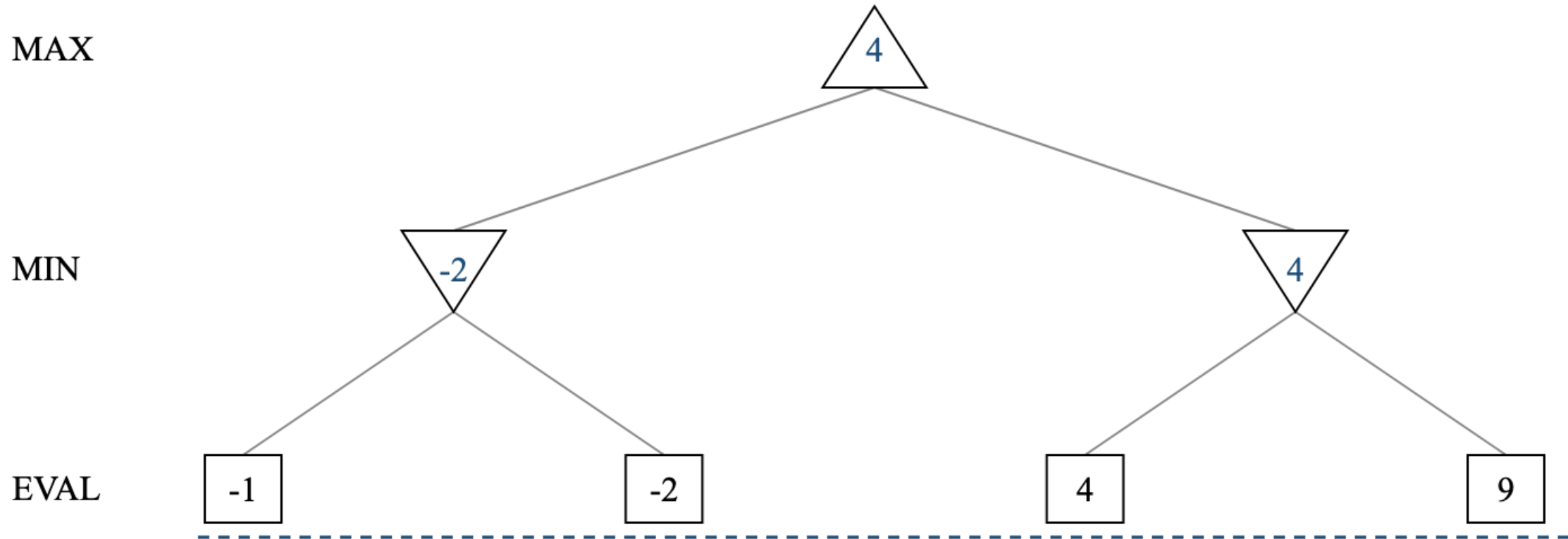


Back Up the Cutoff Estimates



Estimated Root Value: 4 · Choose Right

Back Up the Cutoff Estimates



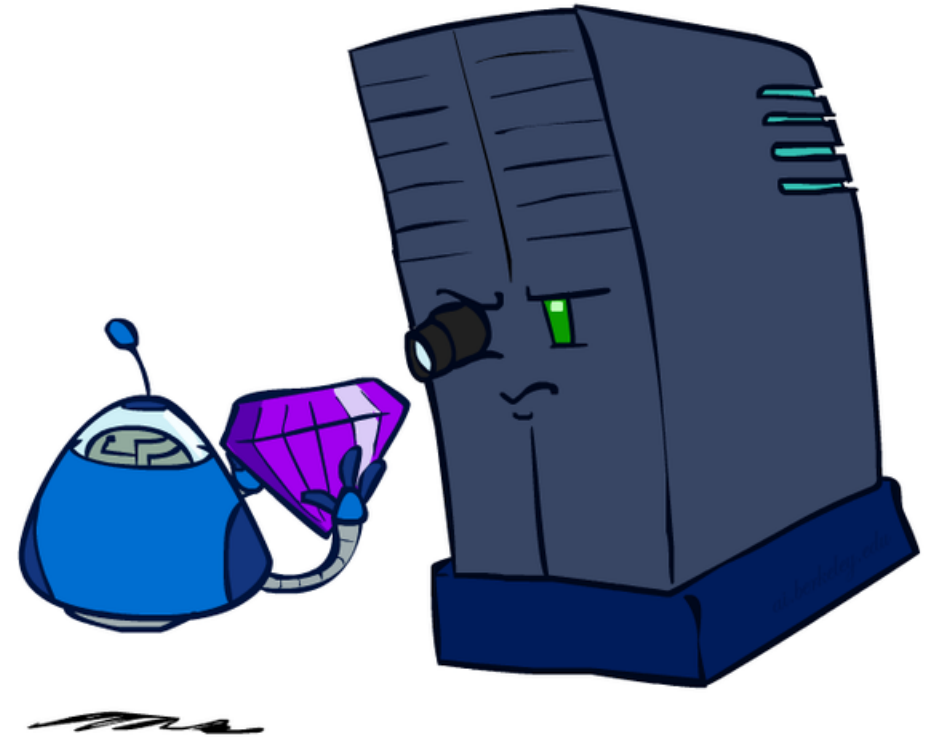
Estimated Root Value: 4 · Choose Right

The Guarantee Depends on the Estimates

Evaluate a Non-Terminal Position

Eval(s): Estimate the Position's Value
for MAX

Ideal: The True Minimax Value

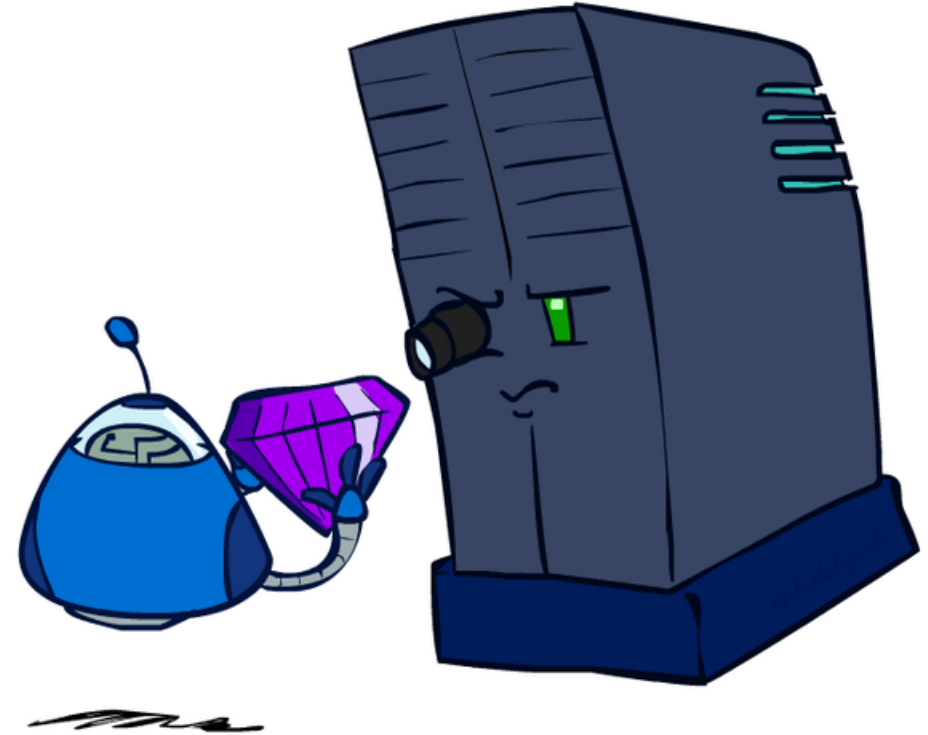


Evaluate a Non-Terminal Position

Eval(s): Estimate the Position's Value
for MAX

Ideal: The True Minimax Value

Useful: Fast Enough to Evaluate Many
Positions



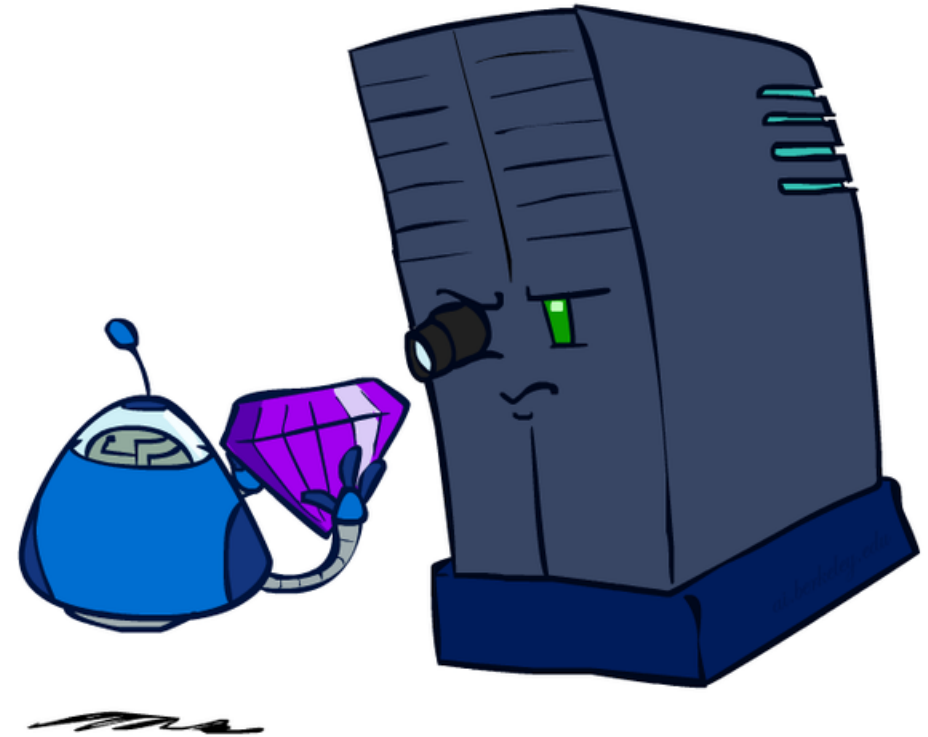
Evaluate a Non-Terminal Position

Eval(s): Estimate the Position's Value
for MAX

Ideal: The True Minimax Value

Useful: Fast Enough to Evaluate Many
Positions

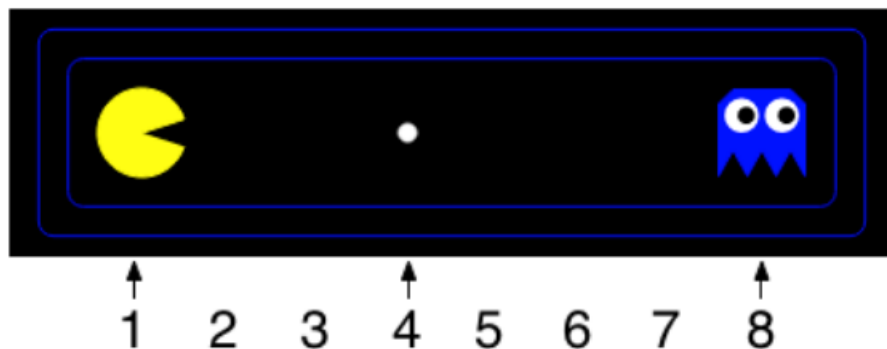
A Larger Number Should Mean a Better
Position for MAX



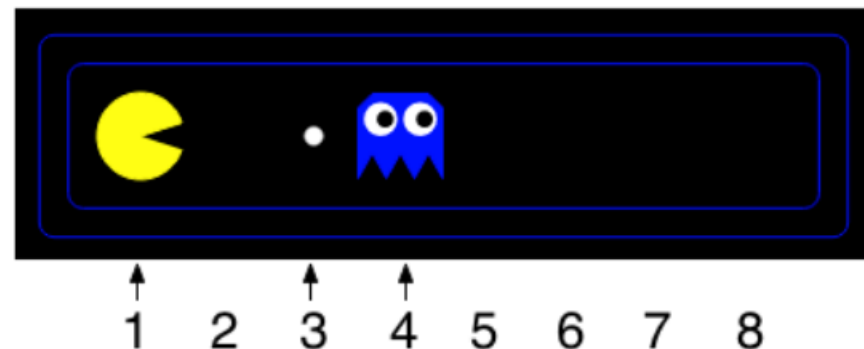
TASK

Task: Evaluation Functions

LEFT POSITION



RIGHT POSITION



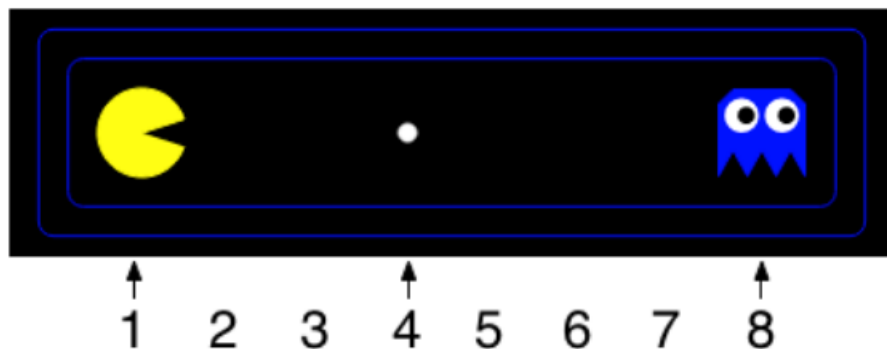
Which Functions Prefer the Left Position?

f = distance to food g = distance to ghost

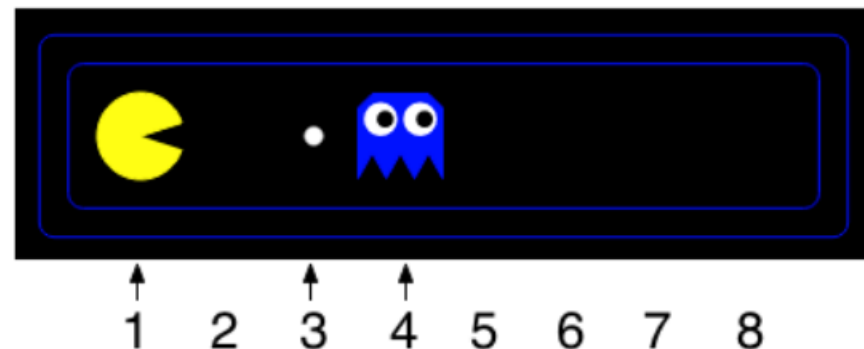
TASK

Task: Evaluation Functions

LEFT POSITION



RIGHT POSITION



Which Functions Prefer the Left Position?

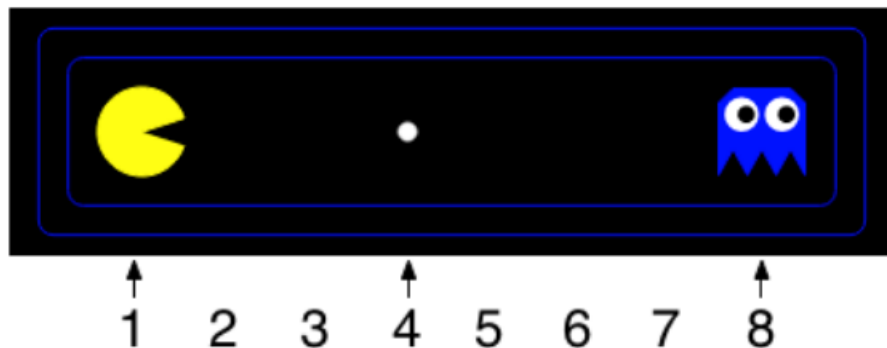
A. $\frac{1}{f}$

f = distance to food · g = distance to ghost

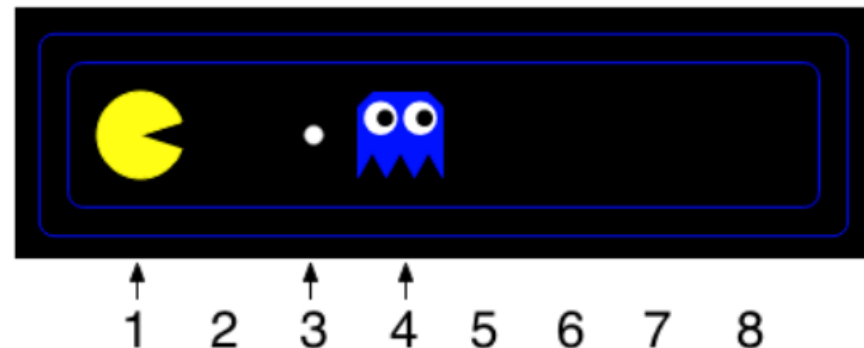
TASK

Task: Evaluation Functions

LEFT POSITION



RIGHT POSITION



Which Functions Prefer the Left Position?

A. $\frac{1}{f}$

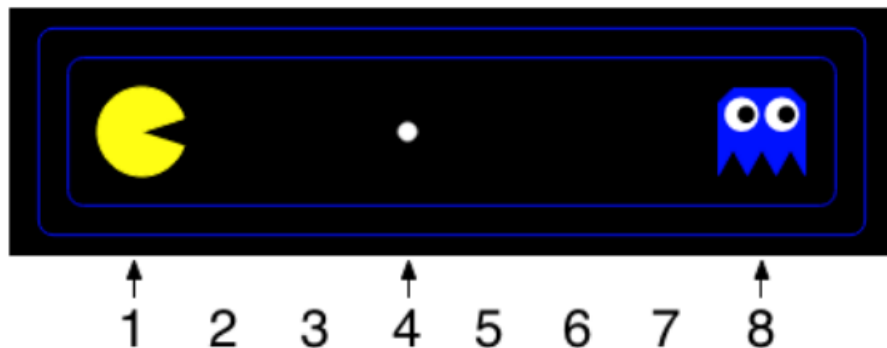
B. g

f = distance to food · g = distance to ghost

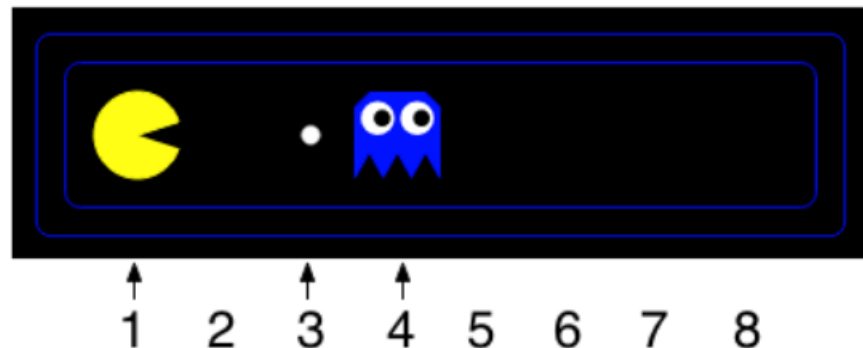
TASK

Task: Evaluation Functions

LEFT POSITION



RIGHT POSITION



Which Functions Prefer the Left Position?

A. $\frac{1}{f}$

B. g

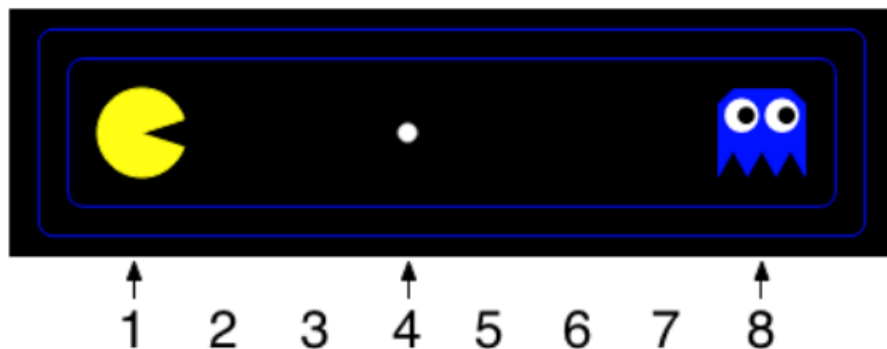
C. $g + \frac{1}{f}$

f = distance to food · g = distance to ghost

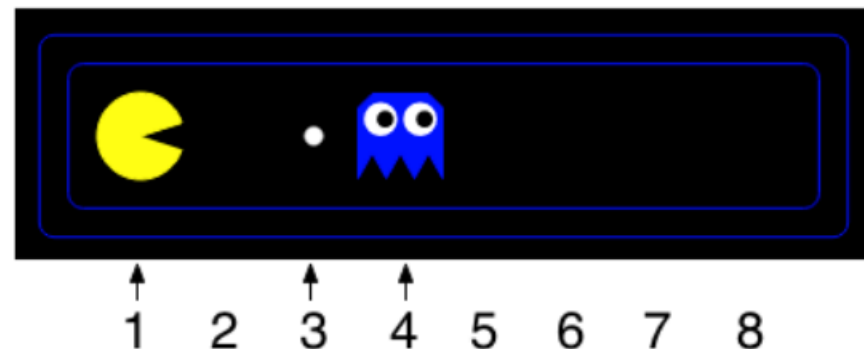
TASK

Task: Evaluation Functions

LEFT POSITION



RIGHT POSITION



Which Functions Prefer the Left Position?

A. $\frac{1}{f}$

B. g

C. $g + \frac{1}{f}$

D. $g + \frac{1000}{f}$

f = distance to food · g = distance to ghost

Combine Features into an Evaluation

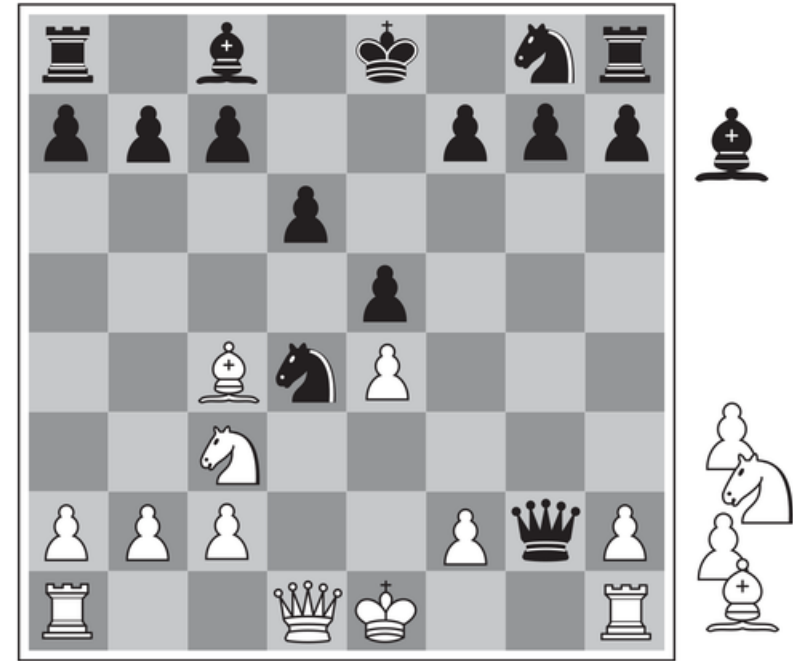
$$\text{Eval}(s) = w_1 f_1(s) + \cdots + w_n f_n(s)$$

Combine Features into an Evaluation

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

Pawn

1



Combine Features into an Evaluation

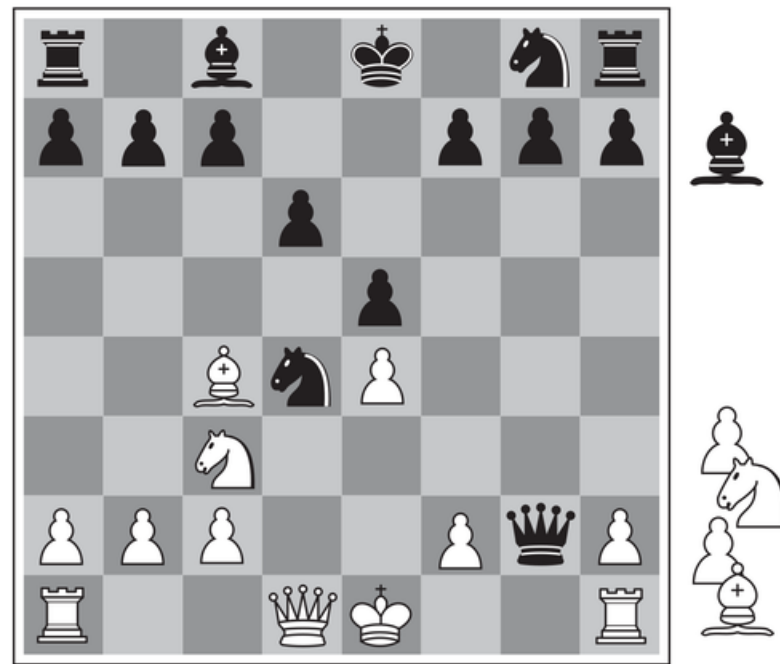
$$\text{Eval}(s) = w_1 f_1(s) + \dots + w_n f_n(s)$$

Pawn

1

Knight / Bishop

3



Combine Features into an Evaluation

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

Pawn

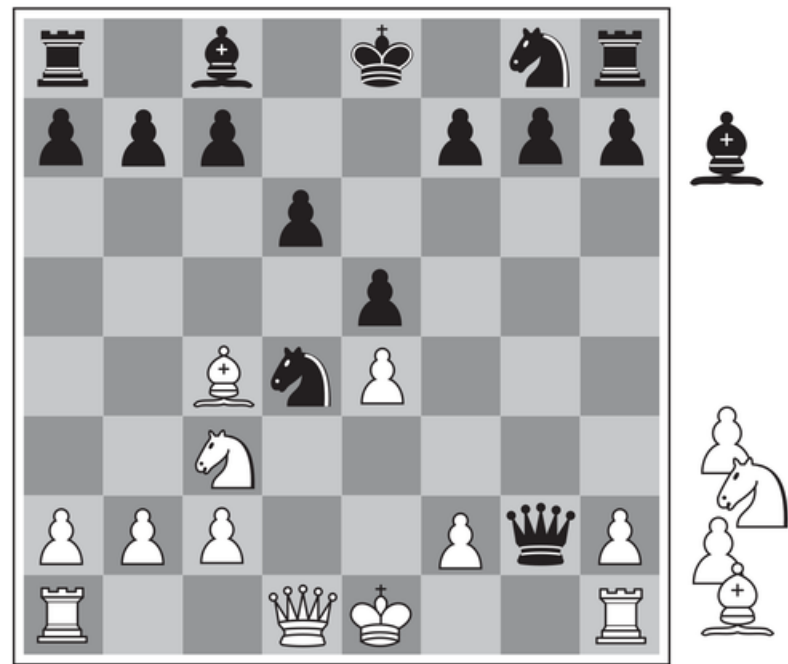
1

Knight / Bishop

3

Rook

5



Combine Features into an Evaluation

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

Pawn

1

Knight / Bishop

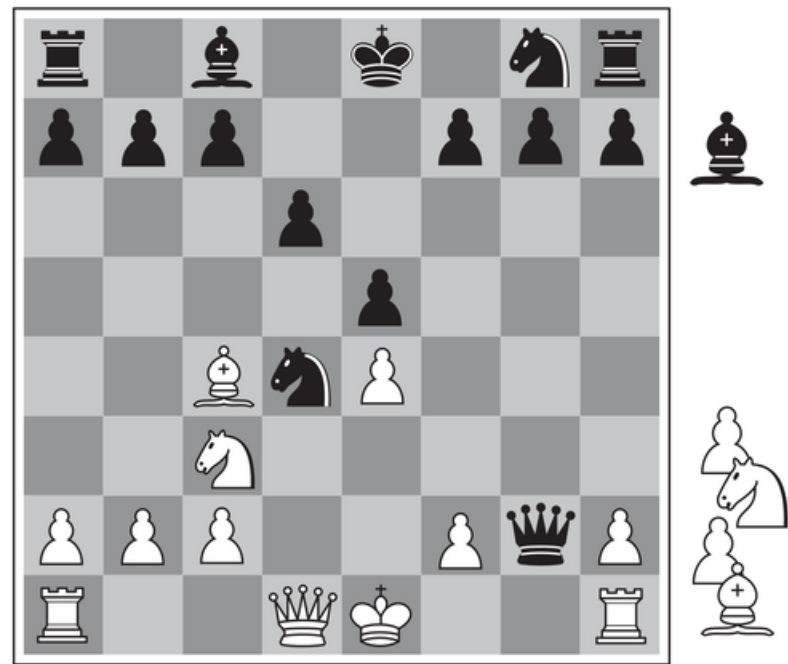
3

Rook

5

Queen

9



Combine Features into an Evaluation

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

Pawn

1

Knight / Bishop

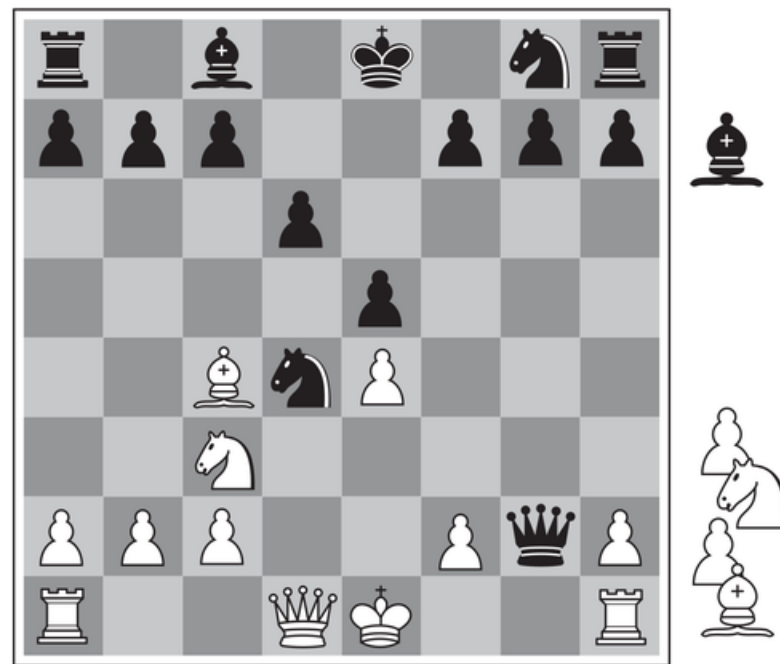
3

Rook

5

Queen

9



Feature: MAX's Material – MIN's Material

Combine Features into an Evaluation

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

Pawn

1

Knight / Bishop

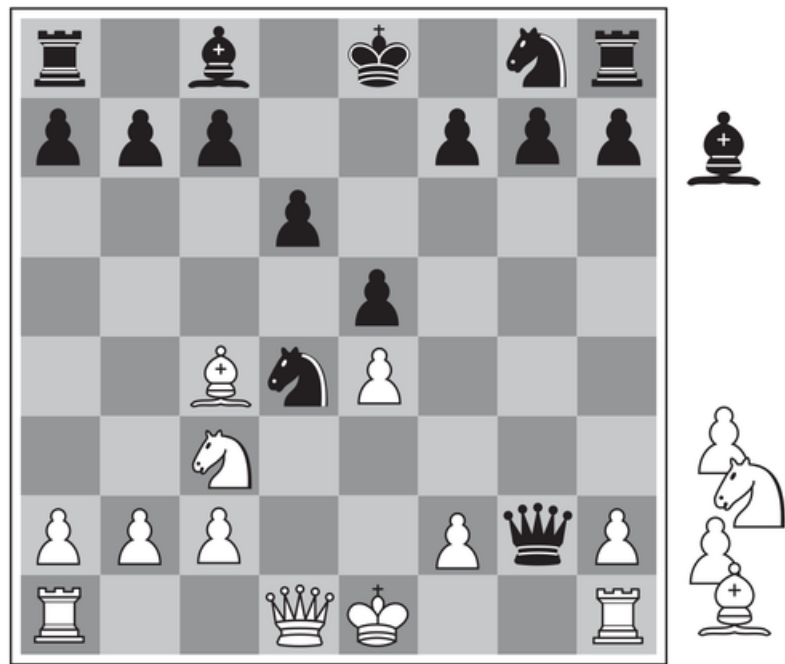
3

Rook

5

Queen

9

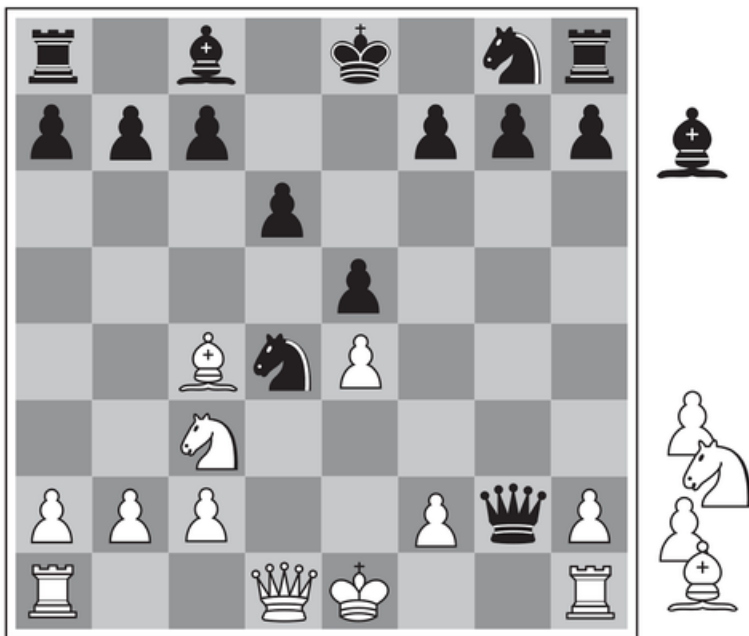


Feature: MAX's Material – MIN's Material

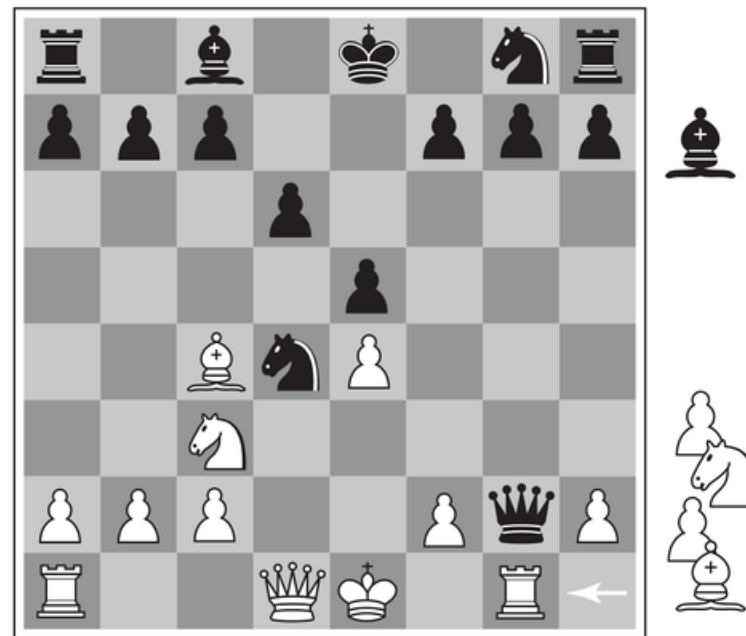
Also Consider King Safety and Pawn Structure

Material Is Not Enough

Who Has the Advantage in Each Position?



(a) White to move



(b) White to move

Depth and Evaluation Error

At the Cutoff, We Estimate a Non-Terminal State



Depth and Evaluation Error

At the Cutoff, We Estimate a Non-Terminal State

A Deeper Cutoff Is Often Closer to the Real Outcome



Depth and Evaluation Error

At the Cutoff, We Estimate a Non-Terminal State

A Deeper Cutoff Is Often Closer to the Real Outcome

Tradeoff: Better Features or More Search?



The Horizon Effect

A Bad Consequence Lies Just Beyond
the Cutoff



The Horizon Effect

A Bad Consequence Lies Just Beyond
the Cutoff

A Delaying Move Pushes It Out of
Sight



The Horizon Effect

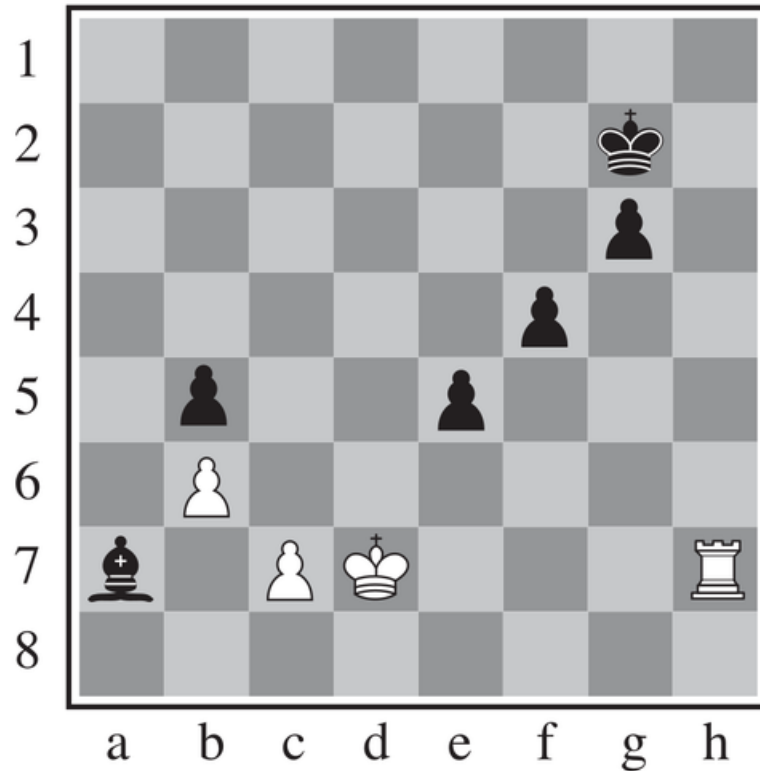
A Bad Consequence Lies Just Beyond
the Cutoff

A Delaying Move Pushes It Out of
Sight

The Delay May Make the Final Outcome Worse



Can Black Save the Bishop?

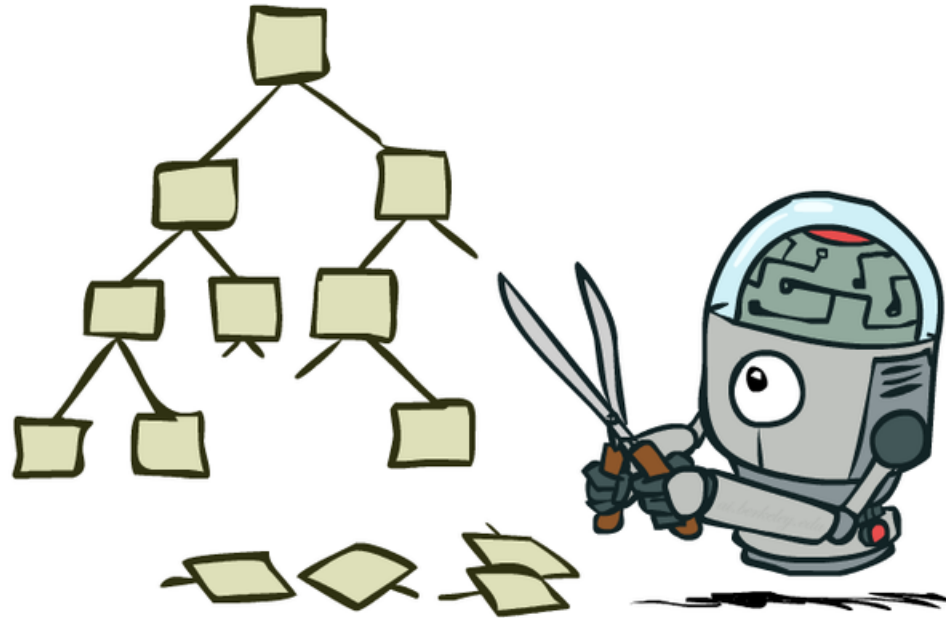


Black to Move

Can a Delaying Tactic Avoid the Loss?

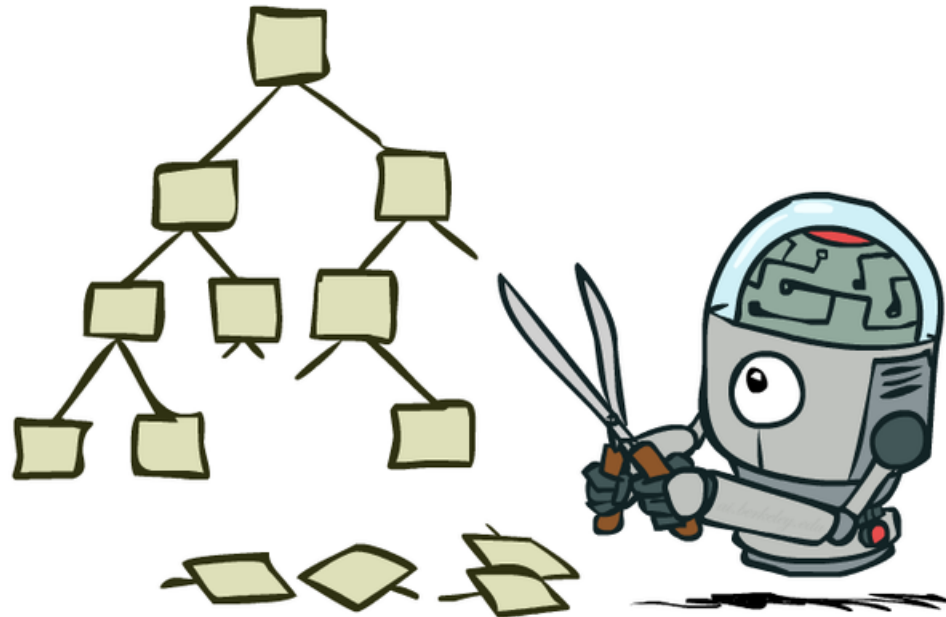
Can We Skip Part of the Tree?

Can We Compute the Exact Minimax Choice Without Visiting Every Node?



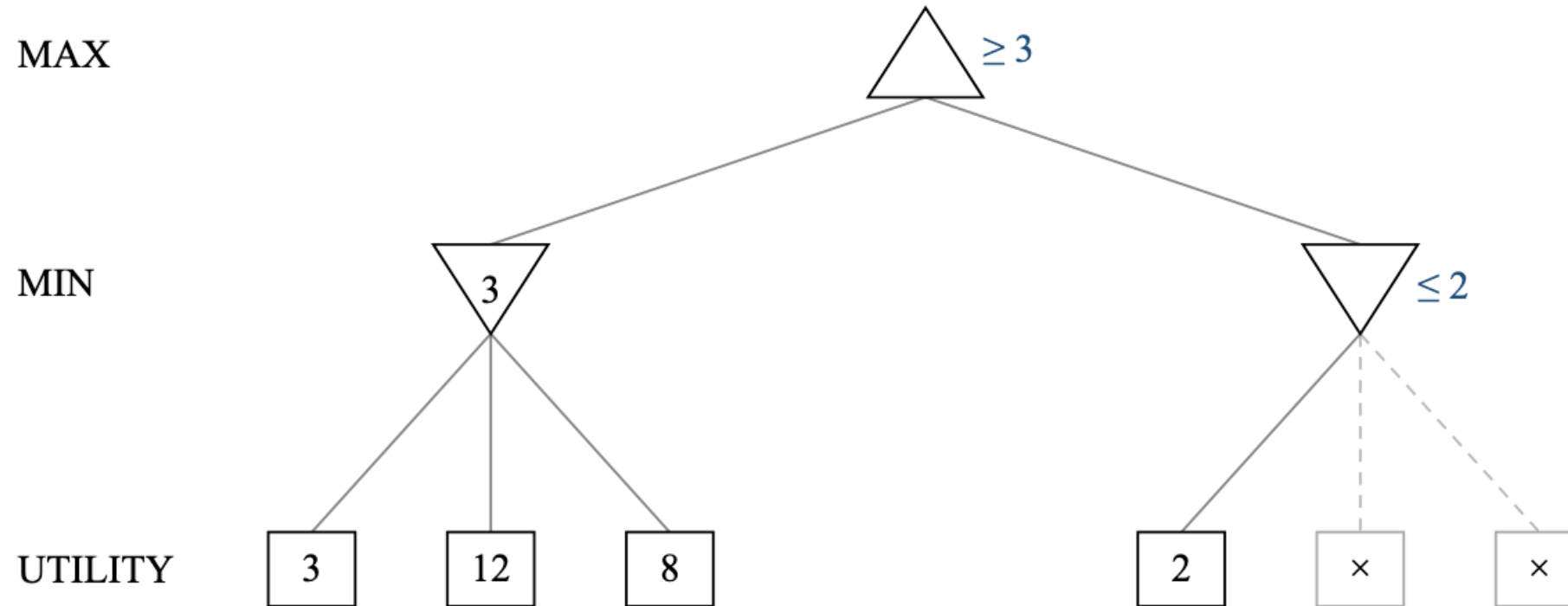
Can We Skip Part of the Tree?

Can We Compute the Exact Minimax Choice Without Visiting Every Node?

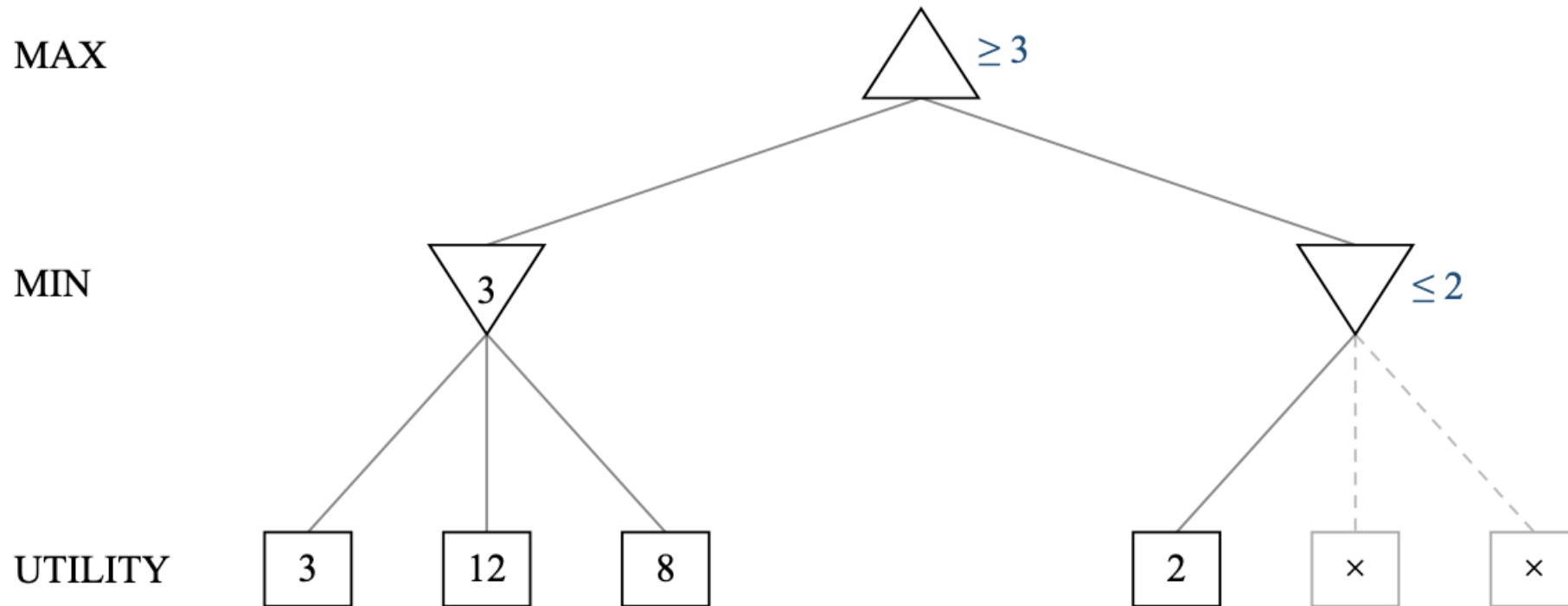


Yes · When a Branch Cannot Affect the Decision

A Branch Cannot Beat the Current Best

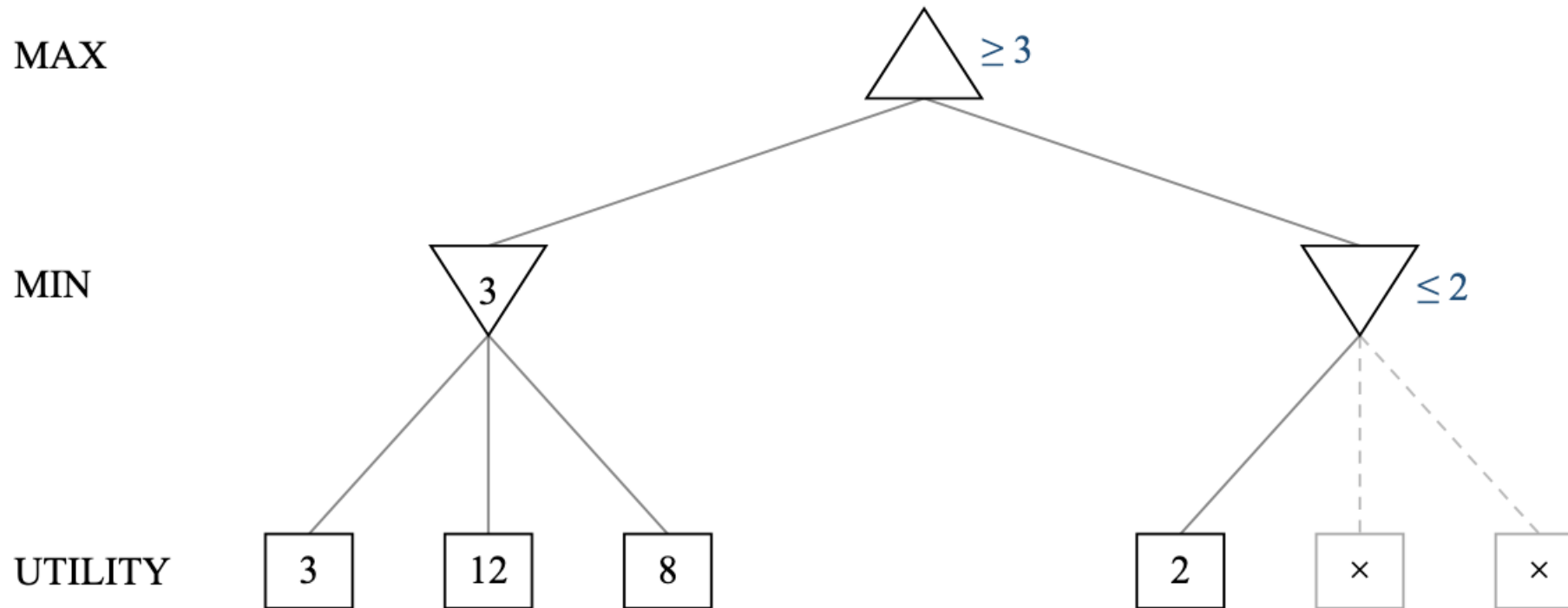


A Branch Cannot Beat the Current Best



Left Gives 3 · Right MIN Finds 2

A Branch Cannot Beat the Current Best



Left Gives 3 · Right MIN Finds 2

The Other Right Leaves Cannot Change MAX's Choice

Alpha–Beta Pruning

Skip Branches That Cannot Change the Root Decision

Alpha–Beta Pruning

Skip Branches That Cannot Change the Root Decision

Result

Same minimax value and optimal root action

Alpha–Beta Pruning

Skip Branches That Cannot Change the Root Decision

Result

Same minimax value and optimal root action

Benefit

Often visits far fewer nodes

Alpha–Beta Pruning

Skip Branches That Cannot Change the Root Decision

Result	Same minimax value and optimal root action
Benefit	Often visits far fewer nodes
Requirement	Track bounds while searching left to right

Alpha–Beta Pruning

Skip Branches That Cannot Change the Root Decision

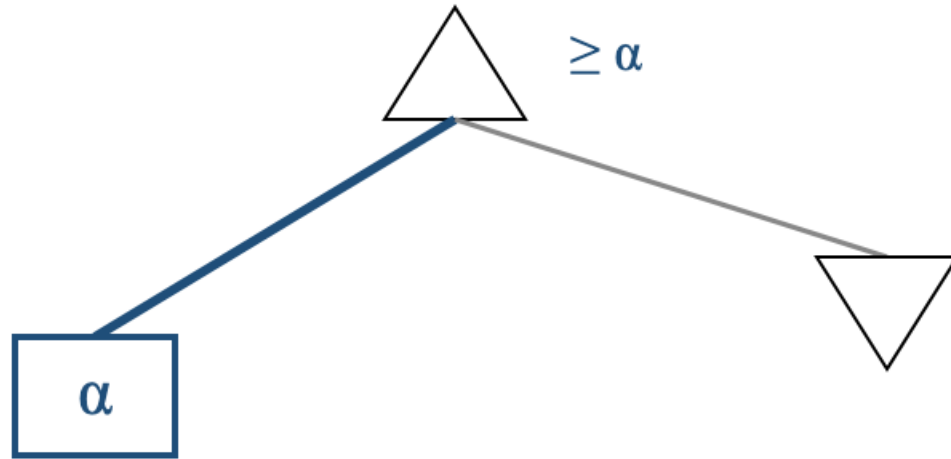
Result Same minimax value and optimal root action

Benefit Often visits far fewer nodes

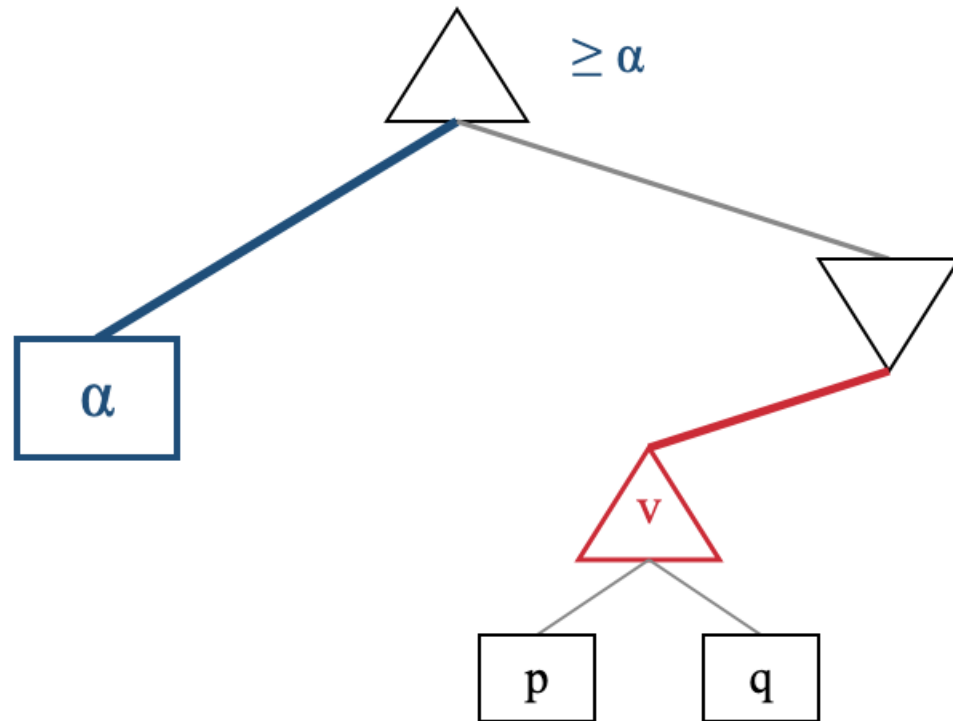
Requirement Track bounds while searching left to right

Pruning Changes Work, Not the Answer

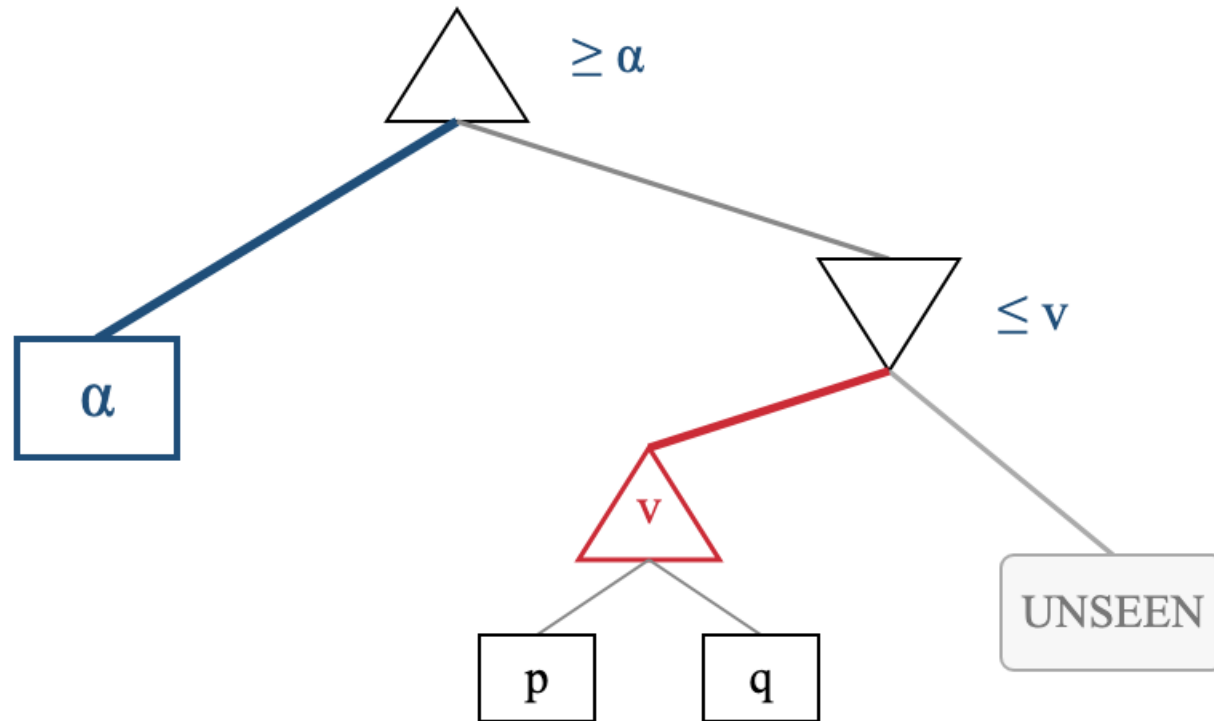
MIN Pruning



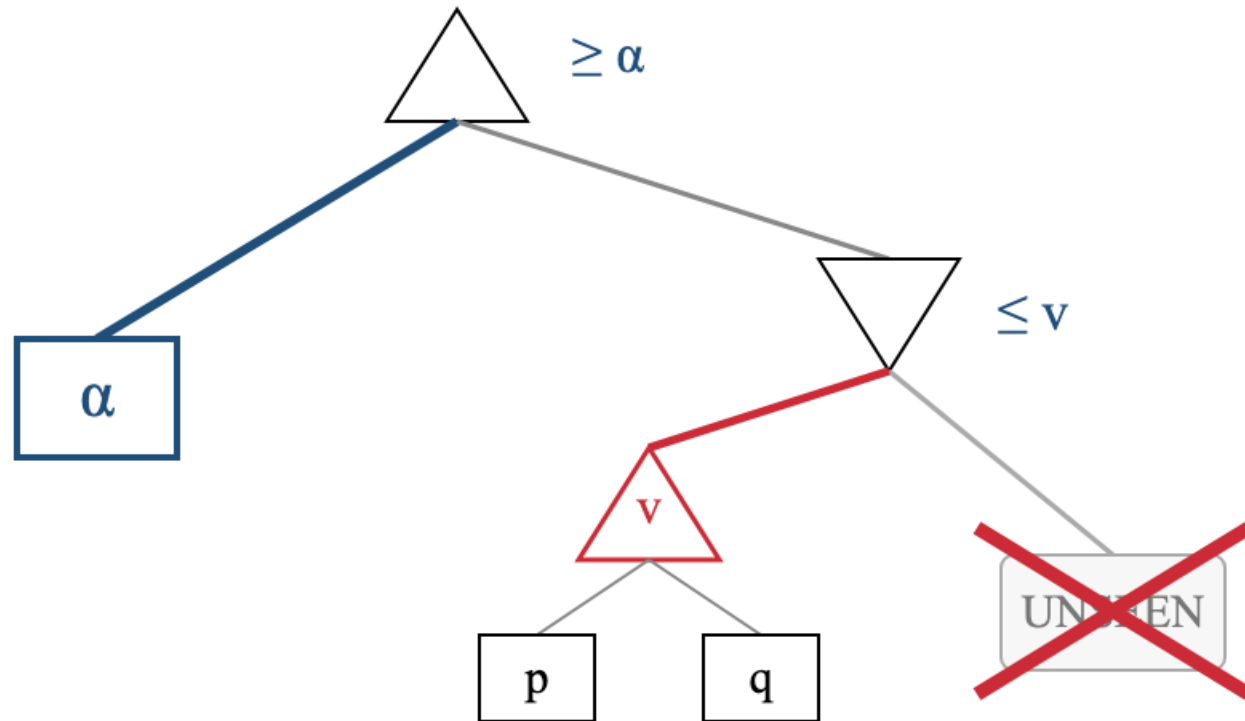
MIN Pruning



MIN Pruning

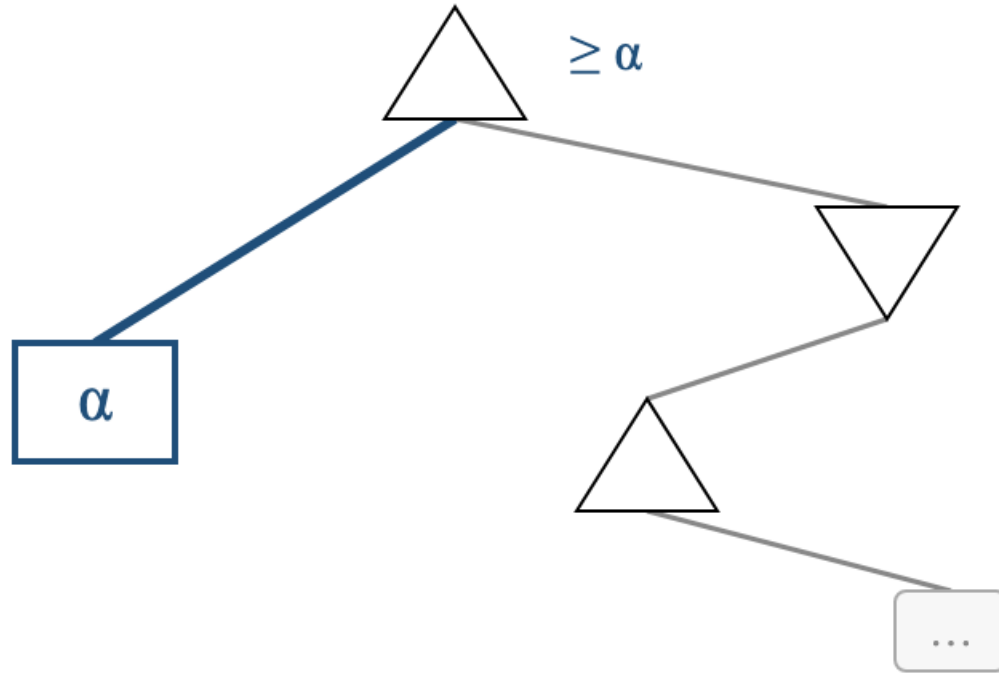


MIN Pruning

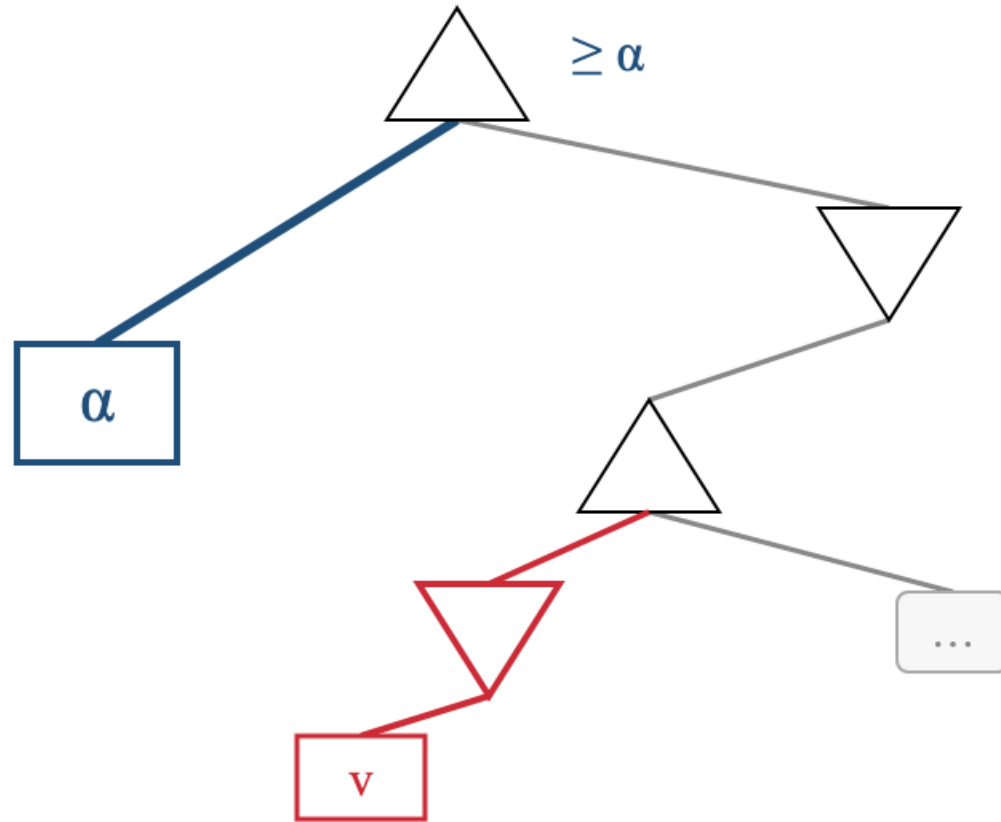


$v \leq \alpha \rightarrow \text{PRUNE}$

MIN Pruning at Greater Depth

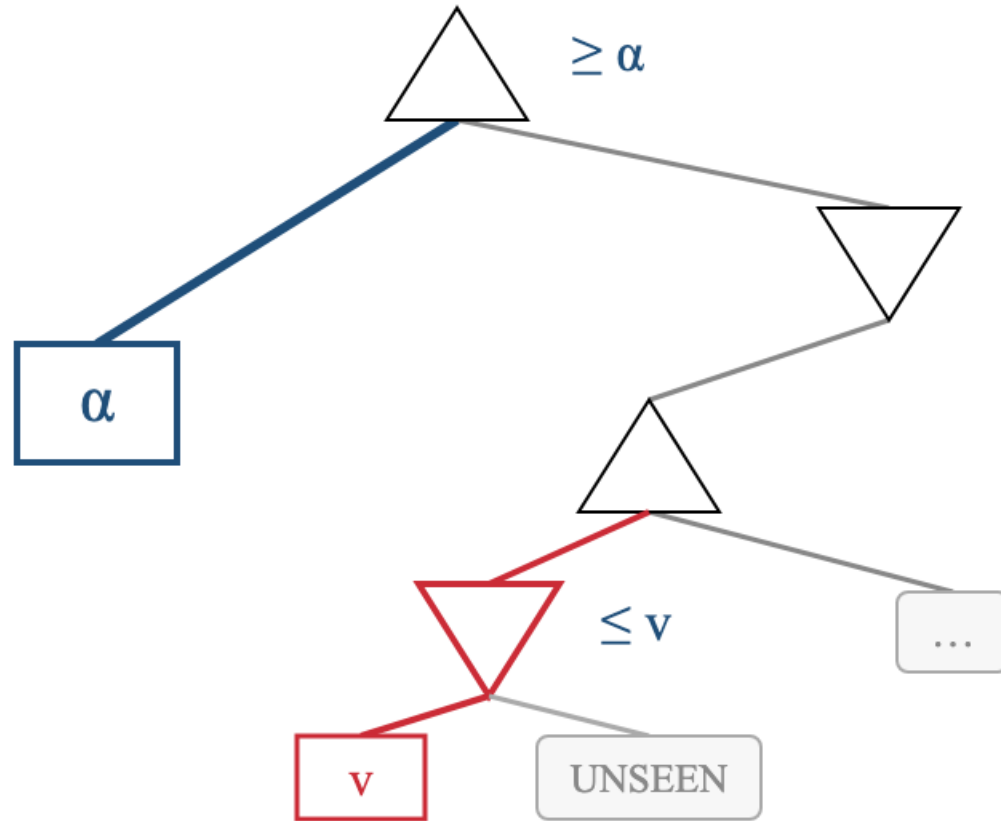


MIN Pruning at Greater Depth



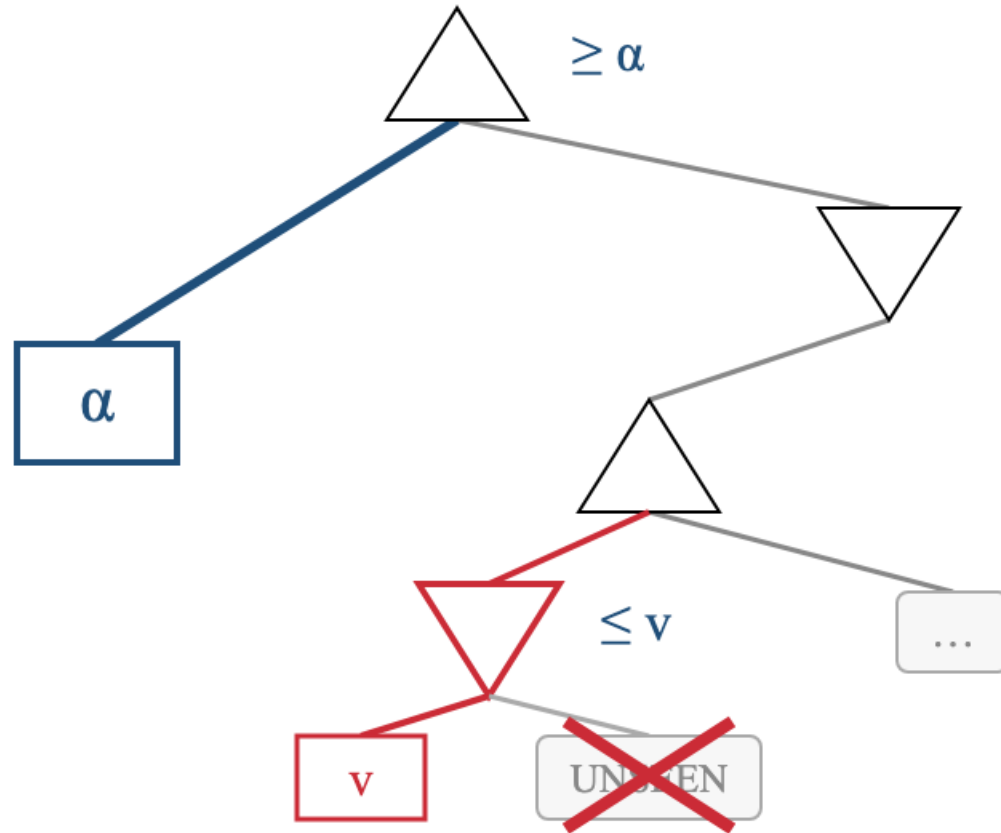
α : MAX's best explored option
on the path from the current node to the root

MIN Pruning at Greater Depth



α : MAX's best explored option
on the path from the current node to the root

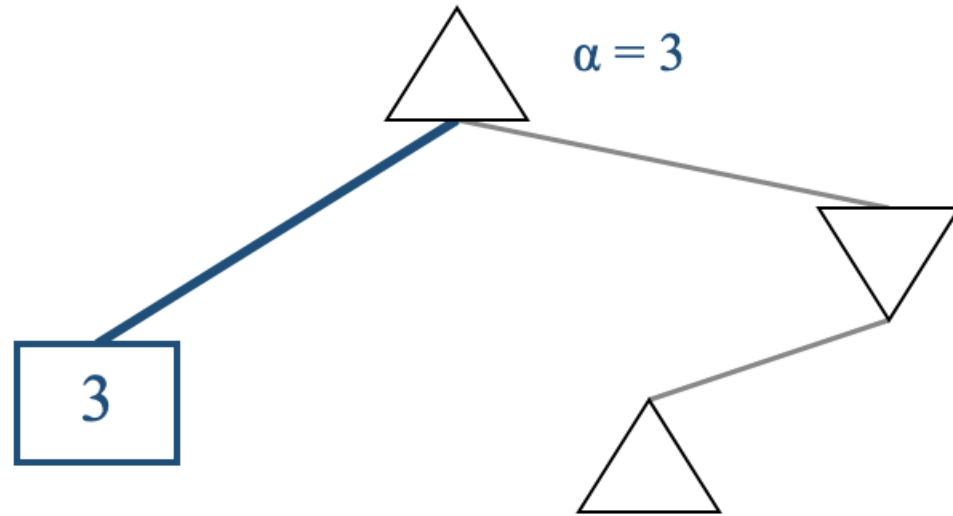
MIN Pruning at Greater Depth



α : MAX's best explored option
on the path from the current node to the root

$$v \leq \alpha \rightarrow \text{PRUNE}$$

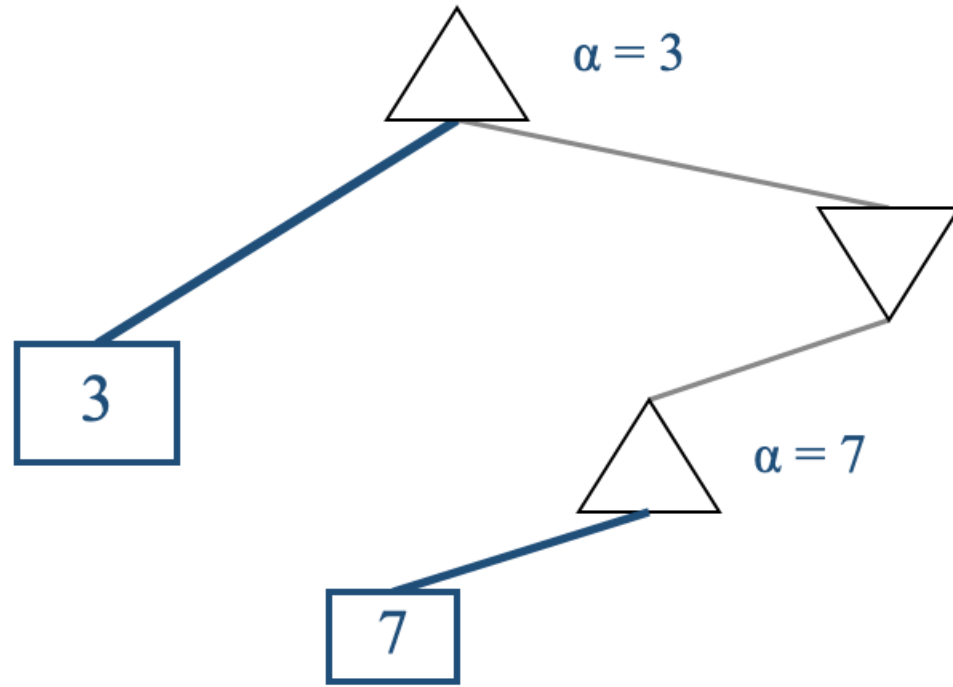
MIN Pruning at Greater Depth



α : MAX's best explored option
on the path from the current node to the root

$v \leq \alpha \rightarrow \text{PRUNE}$

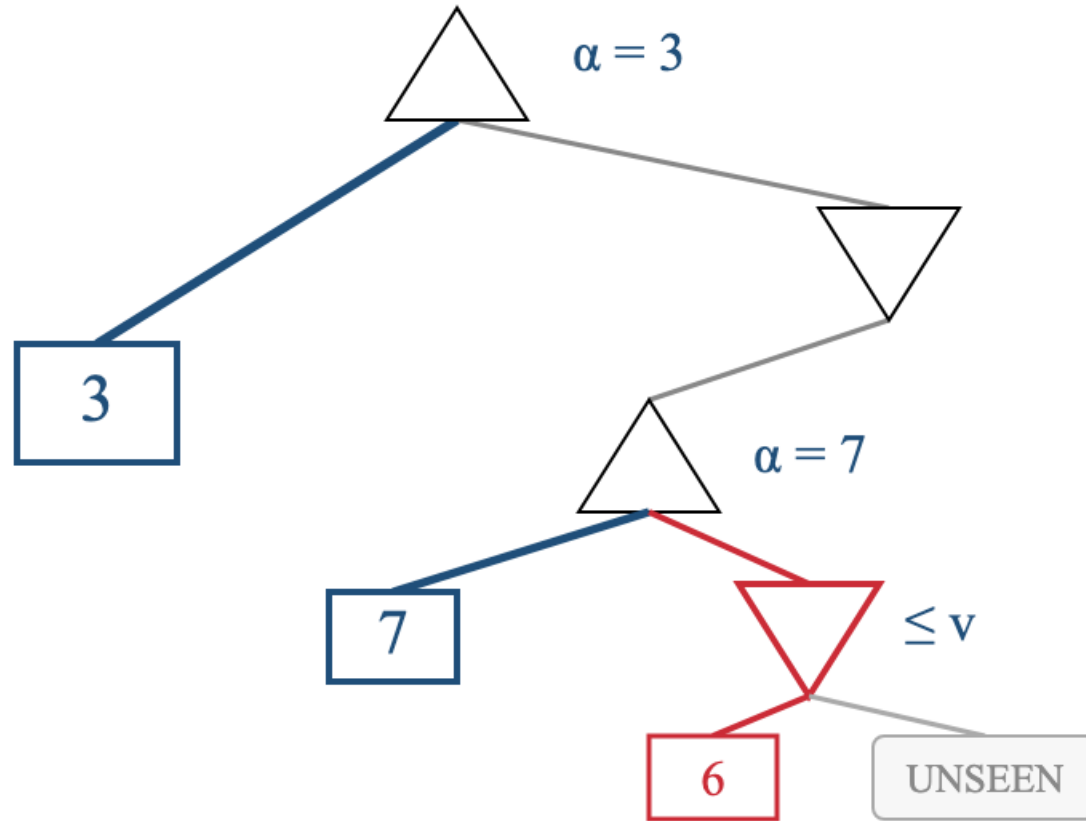
MIN Pruning at Greater Depth



α : MAX's best explored option
on the path from the current node to the root

$v \leq \alpha \rightarrow \text{PRUNE}$

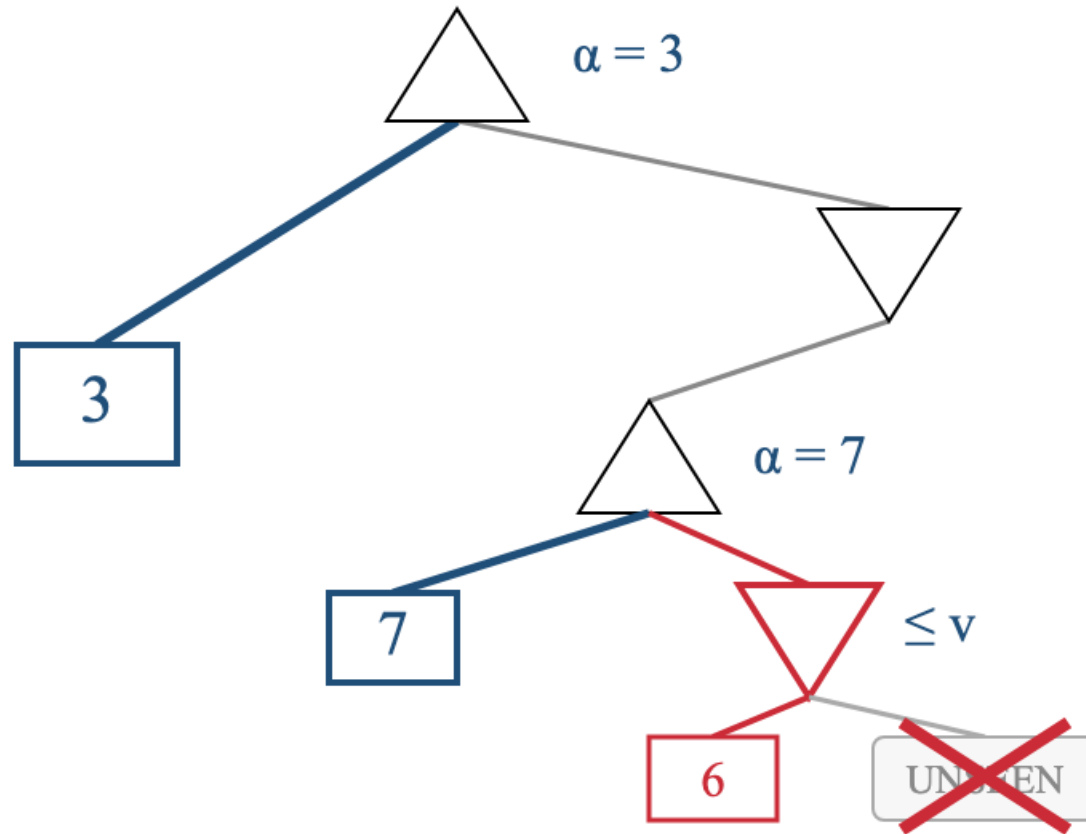
MIN Pruning at Greater Depth



α : MAX's best explored option
on the path from the current node to the root

$v \leq \alpha \rightarrow \text{PRUNE}$

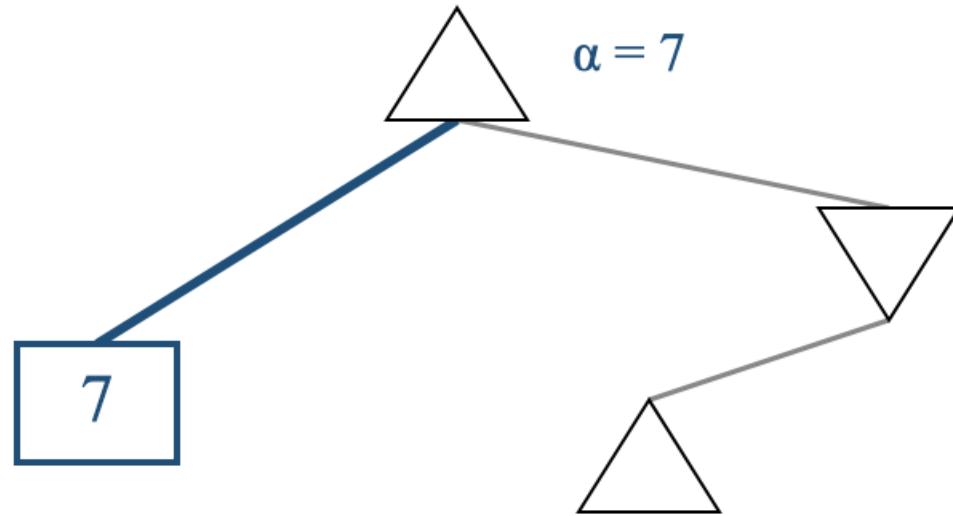
MIN Pruning at Greater Depth



α : MAX's best explored option
on the path from the current node to the root

$$v \leq \alpha \rightarrow \text{PRUNE}$$

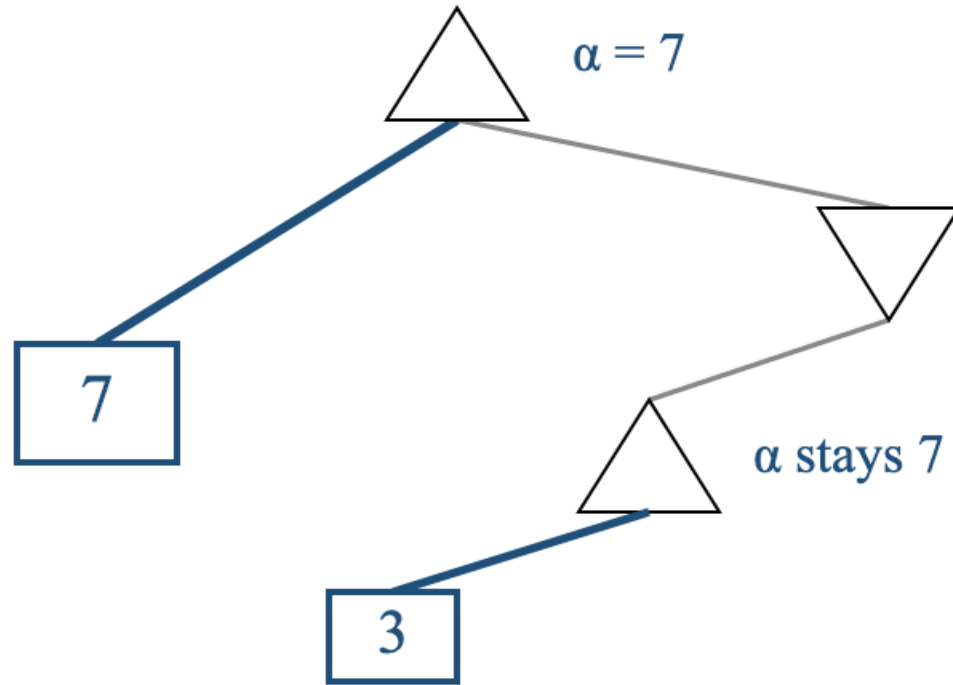
MIN Pruning at Greater Depth



α : MAX's best explored option
on the path from the current node to the root

$v \leq \alpha \rightarrow \text{PRUNE}$

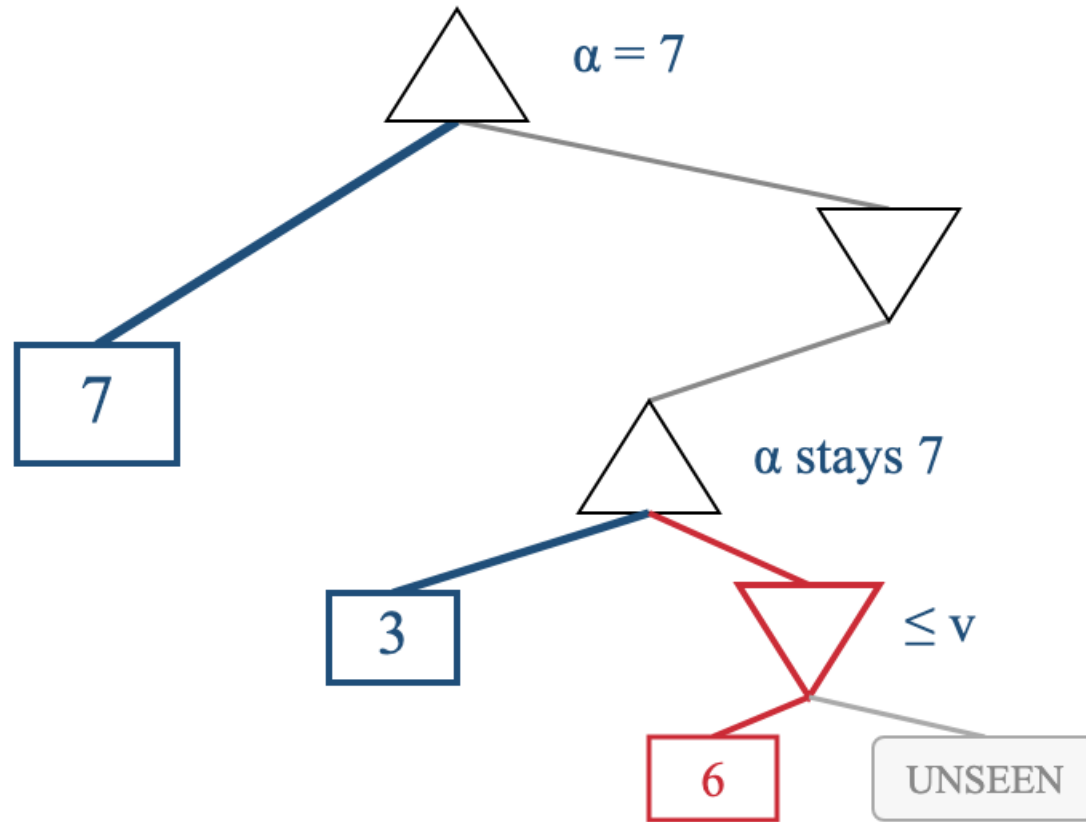
MIN Pruning at Greater Depth



α : MAX's best explored option
on the path from the current node to the root

$v \leq \alpha \rightarrow \text{PRUNE}$

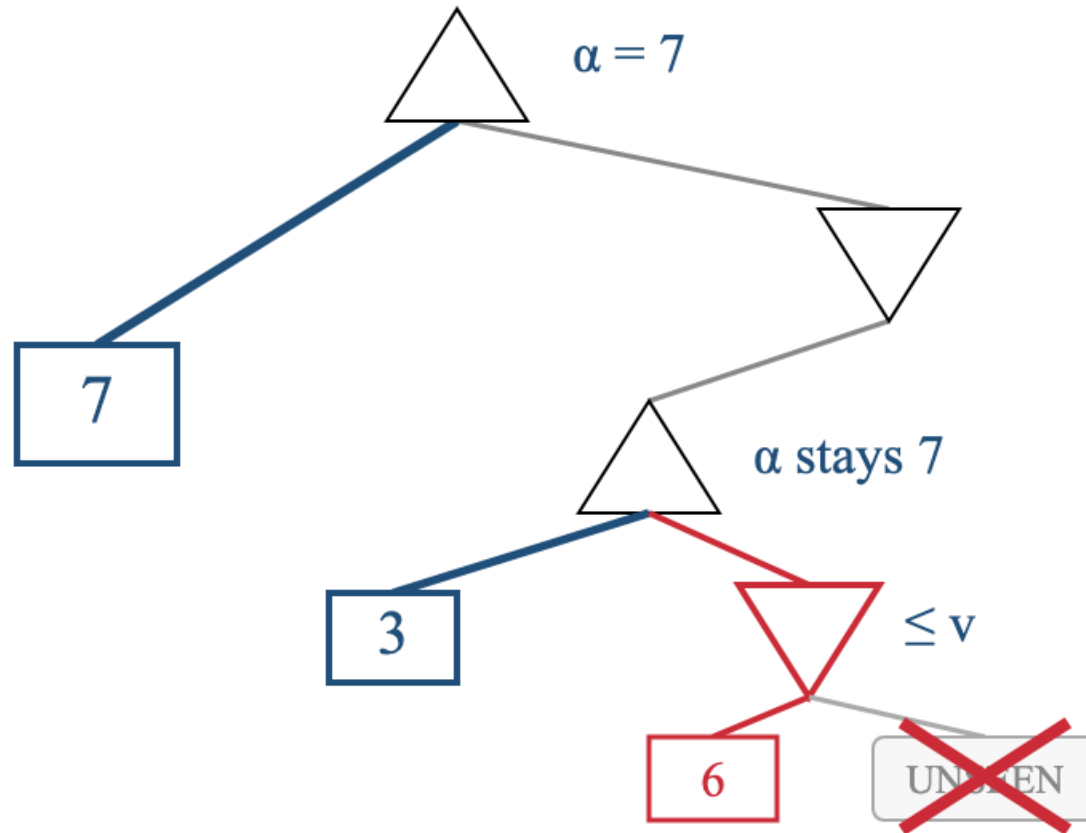
MIN Pruning at Greater Depth



α : MAX's best explored option
on the path from the current node to the root

$v \leq \alpha \rightarrow \text{PRUNE}$

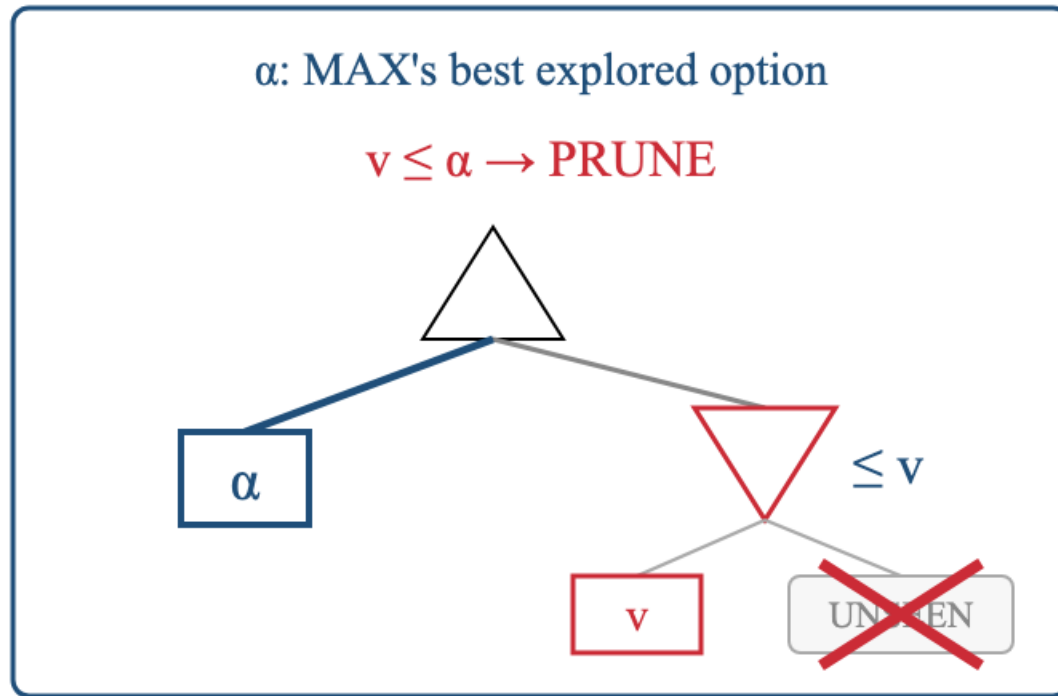
MIN Pruning at Greater Depth



α : MAX's best explored option
on the path from the current node to the root

$v \leq \alpha \rightarrow \text{PRUNE}$

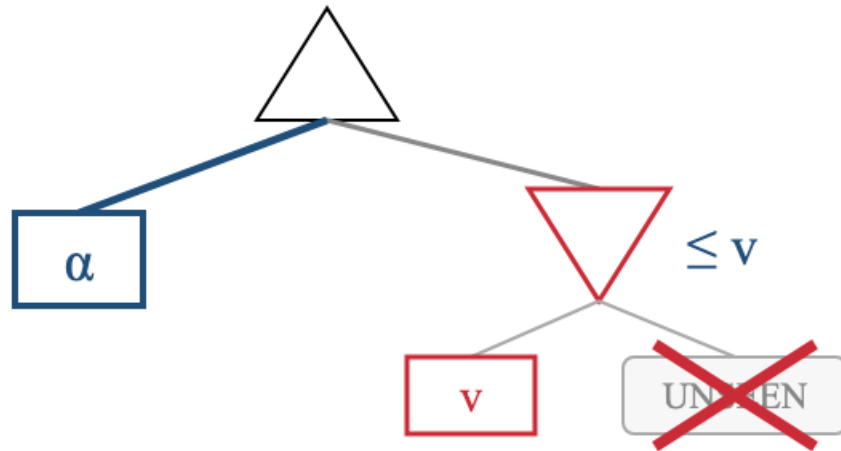
Pruning Conditions



Pruning Conditions

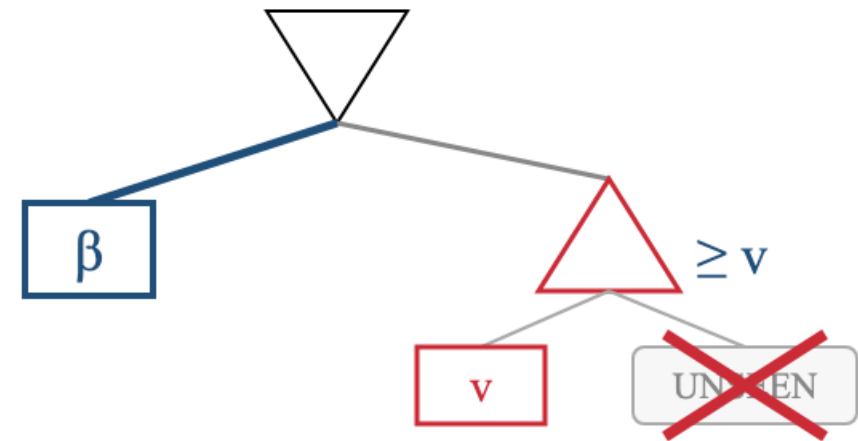
α : MAX's best explored option

$v \leq \alpha \rightarrow \text{PRUNE}$

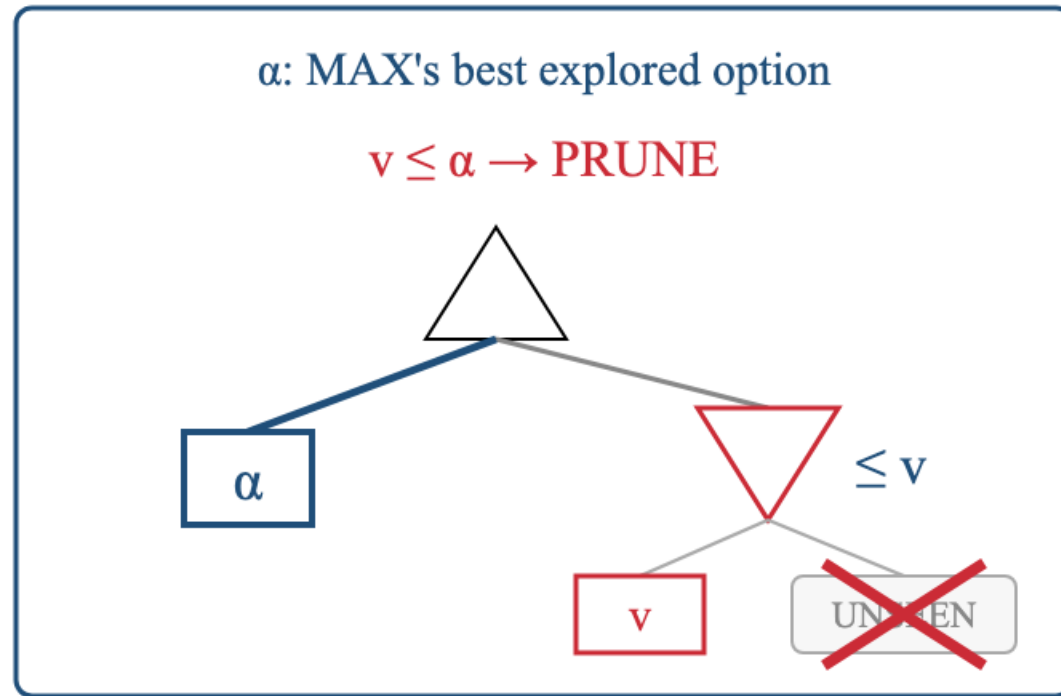
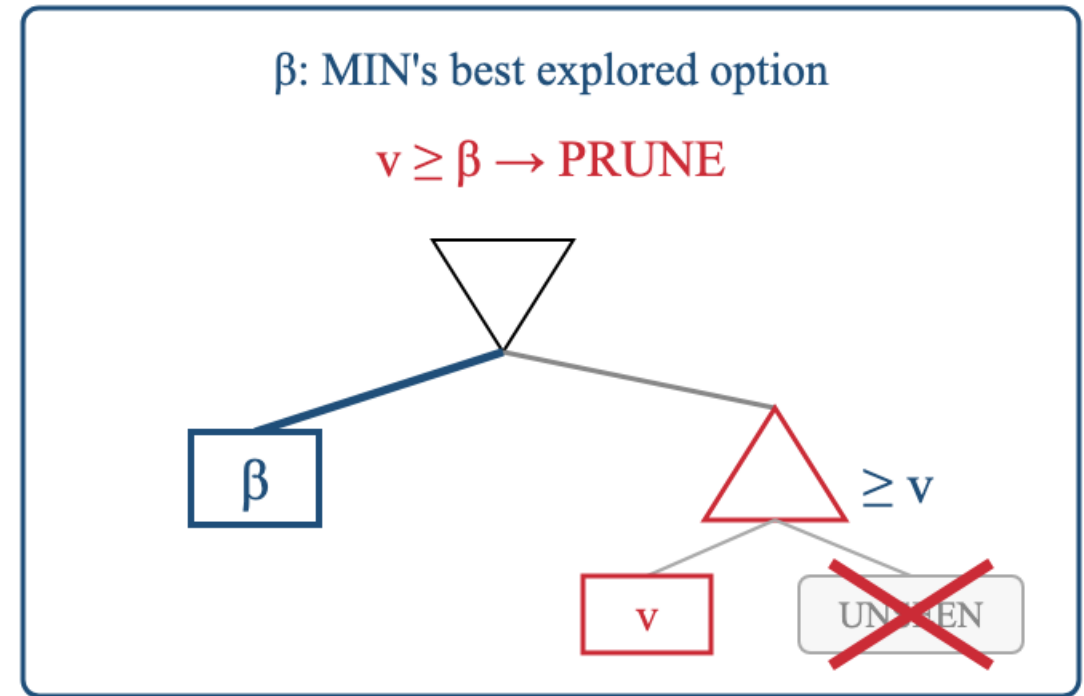


β : MIN's best explored option

$v \geq \beta \rightarrow \text{PRUNE}$



Pruning Conditions



MAX: $\alpha \leftarrow \max(\alpha, v)$

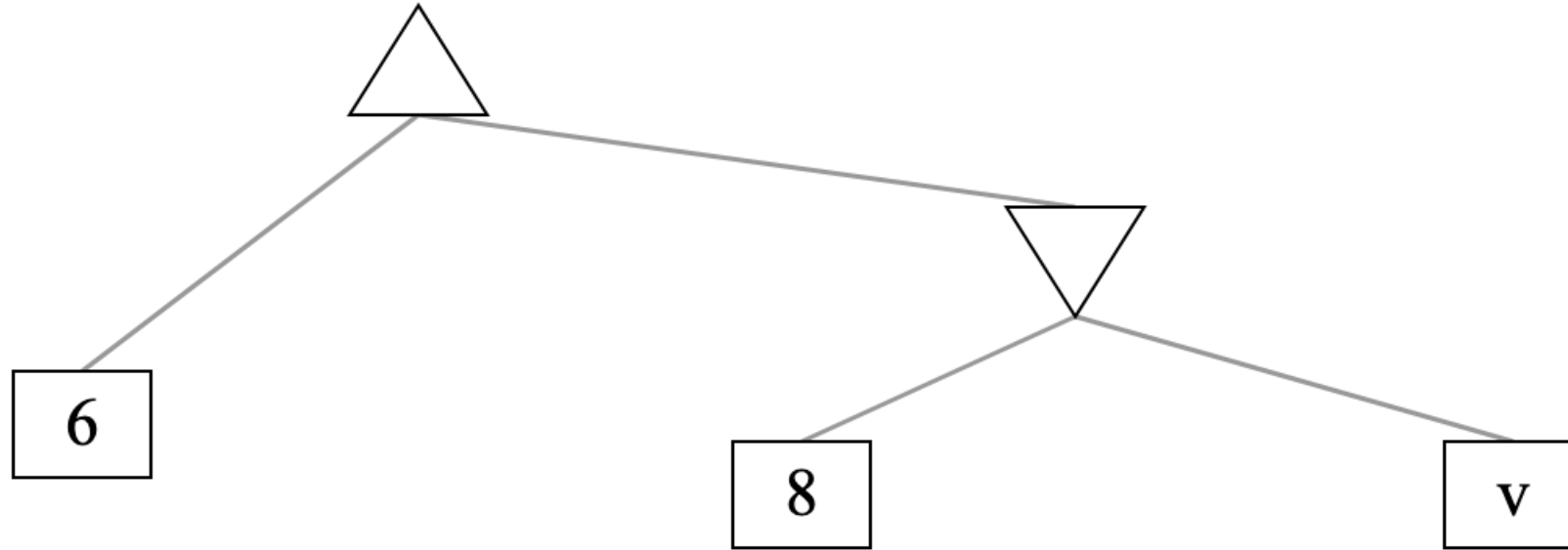
α starts at $-\infty$ and rises



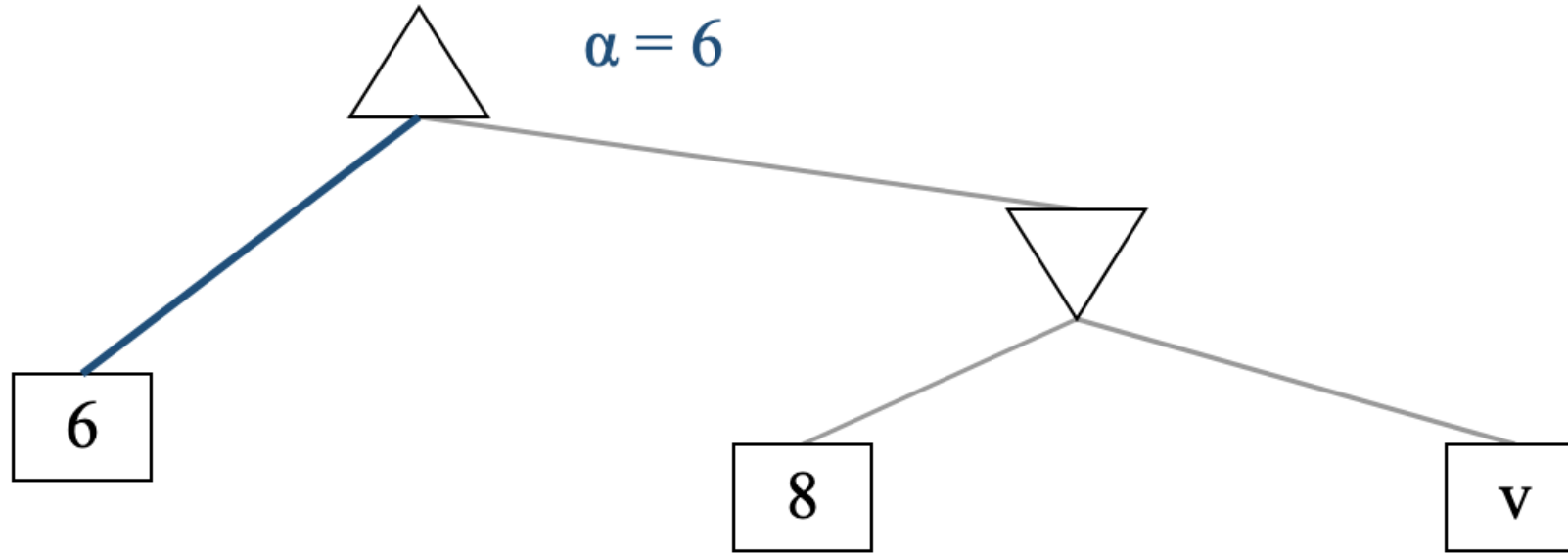
MIN: $\beta \leftarrow \min(\beta, v)$

β starts at $+\infty$ and falls

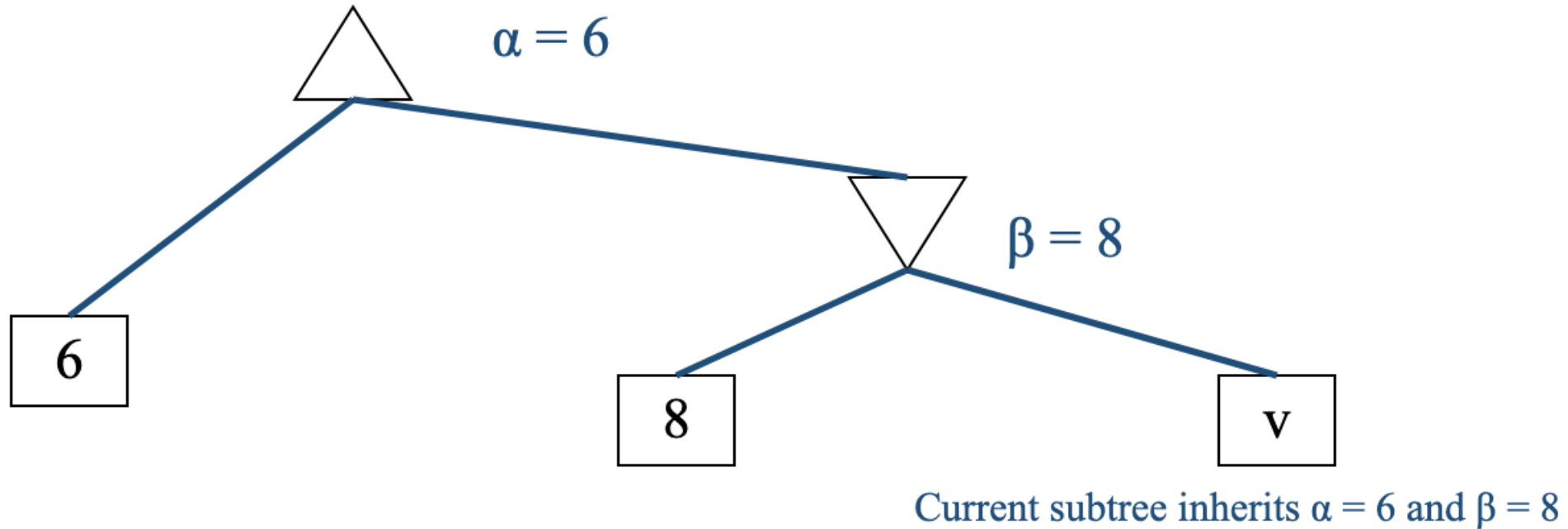
Which Values Can Change the Choice?



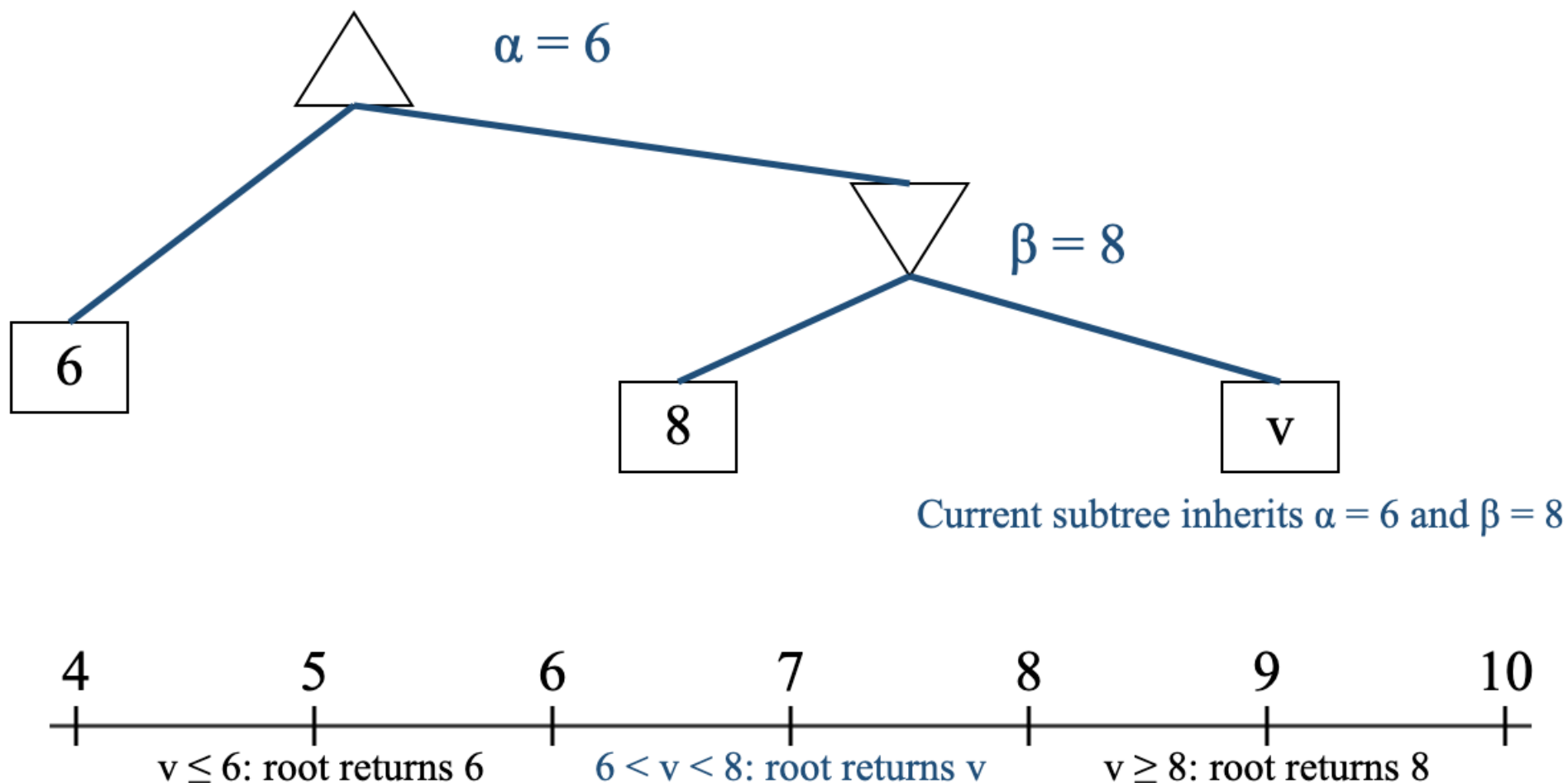
Which Values Can Change the Choice?



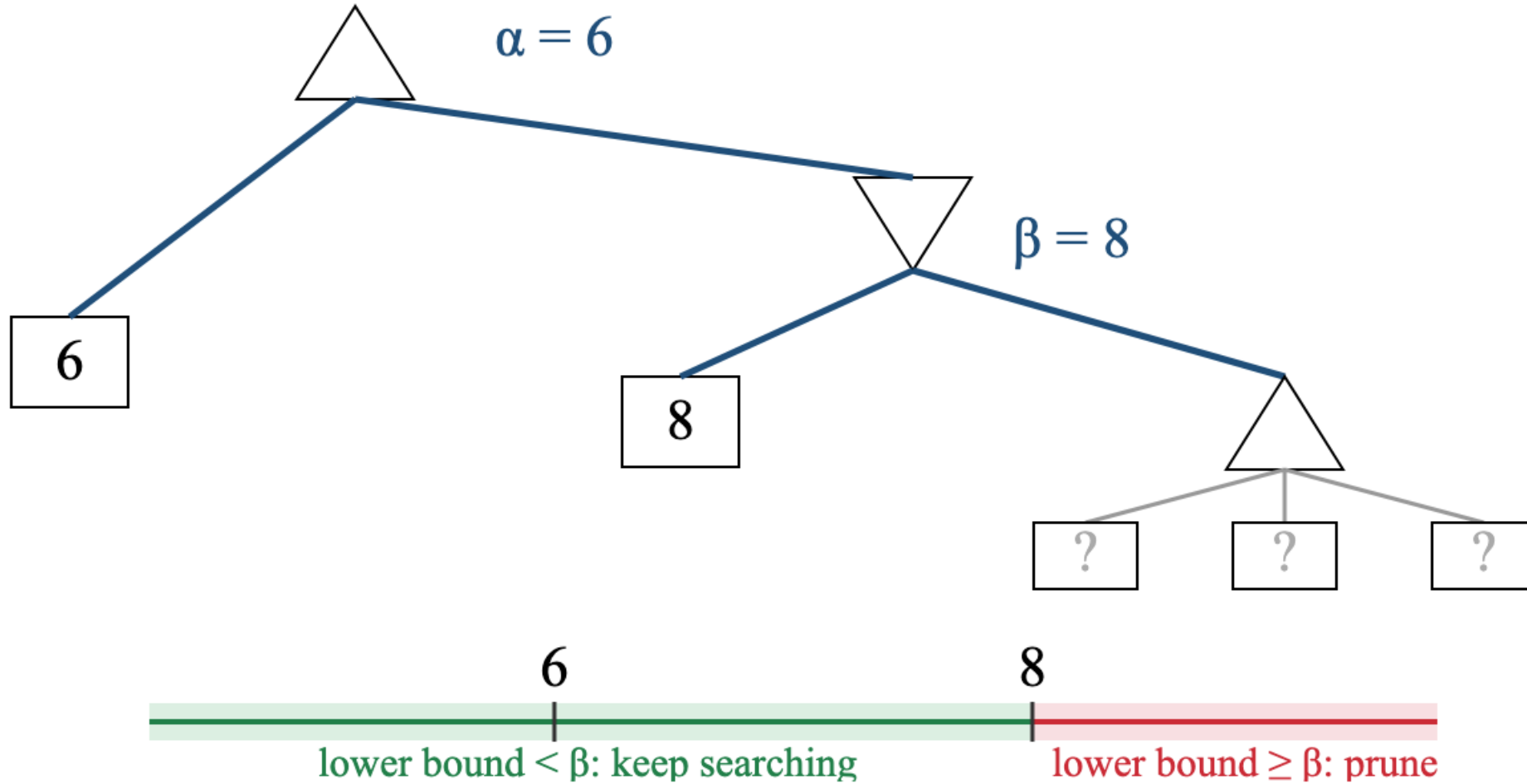
Which Values Can Change the Choice?



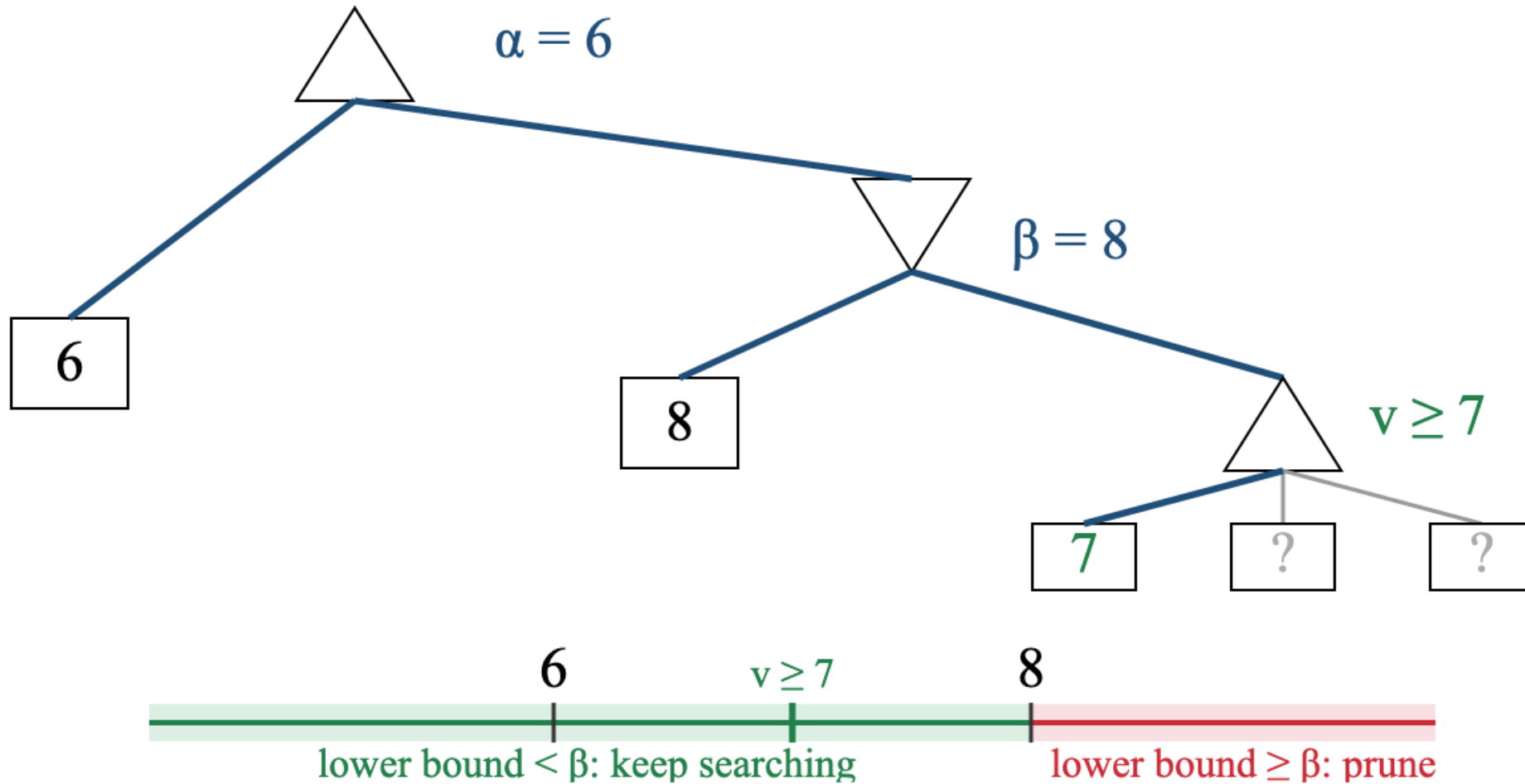
Which Values Can Change the Choice?



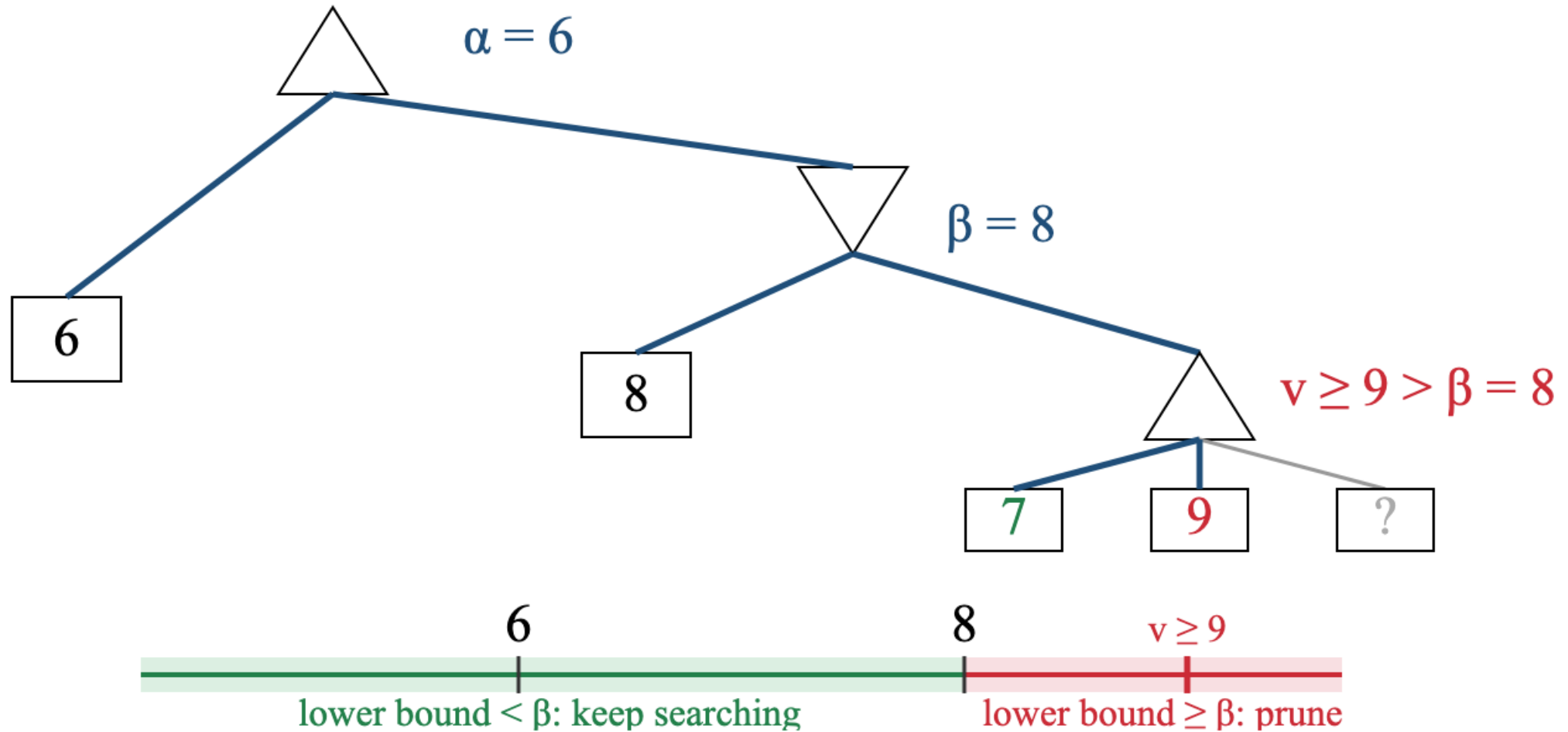
MIN Pruning



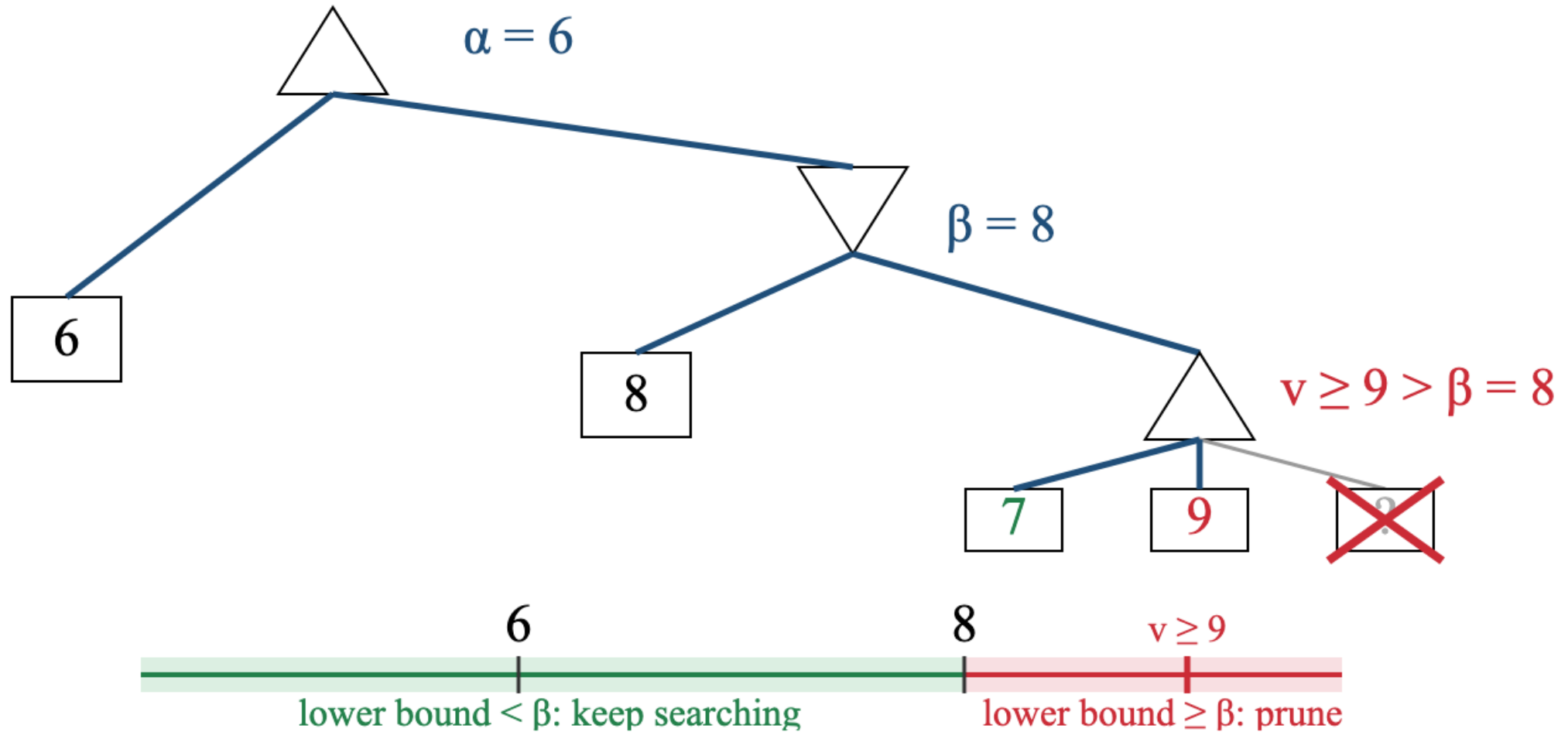
MIN Pruning



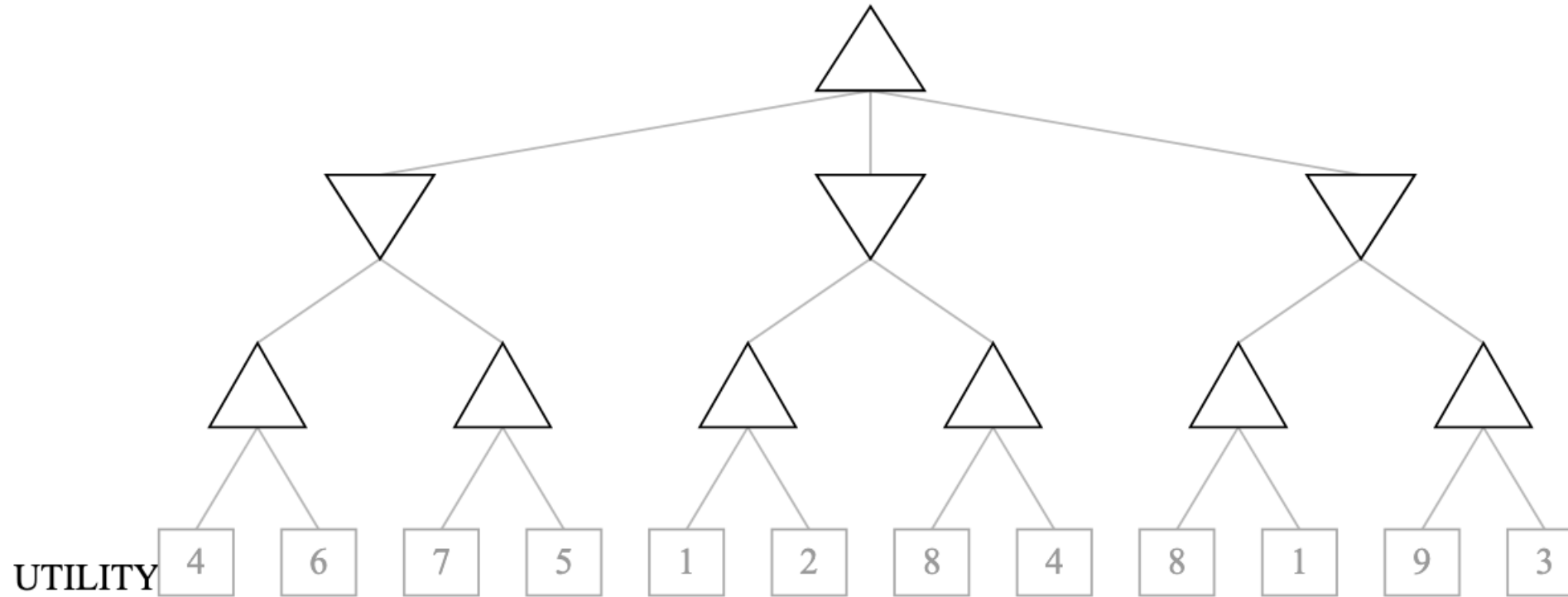
MIN Pruning



MIN Pruning

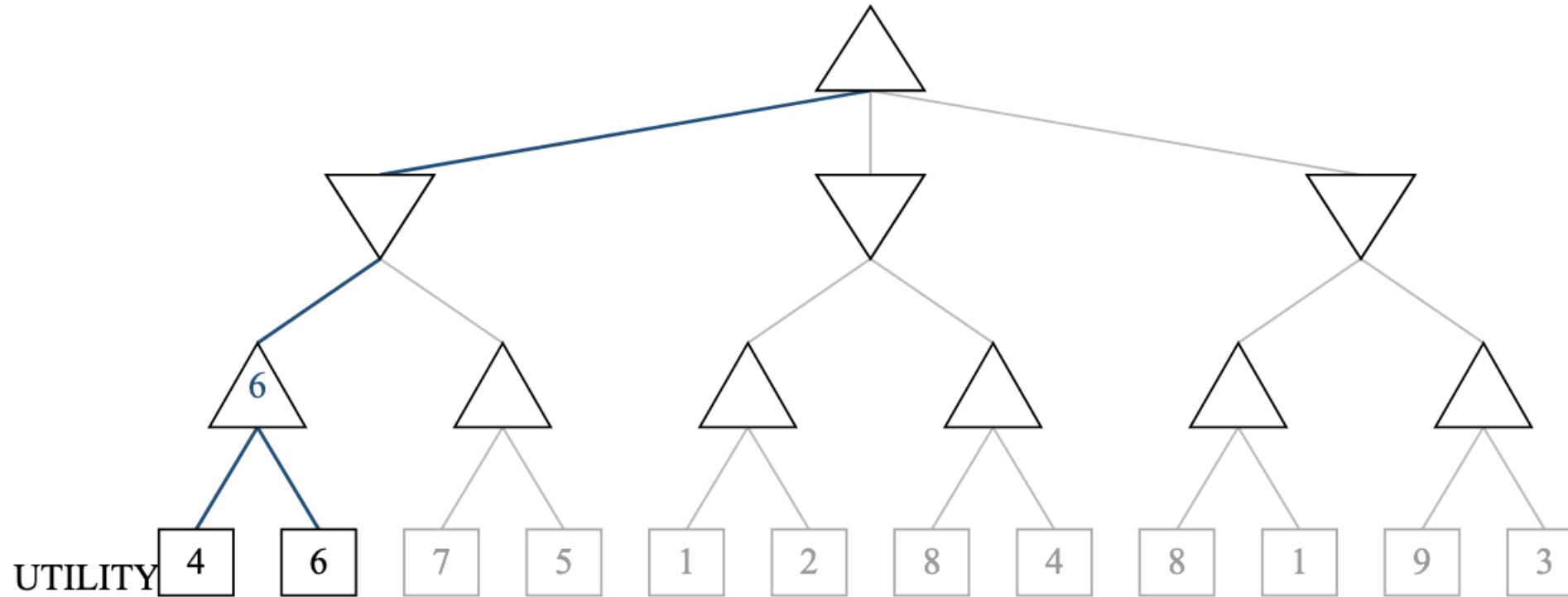


Initial Bounds



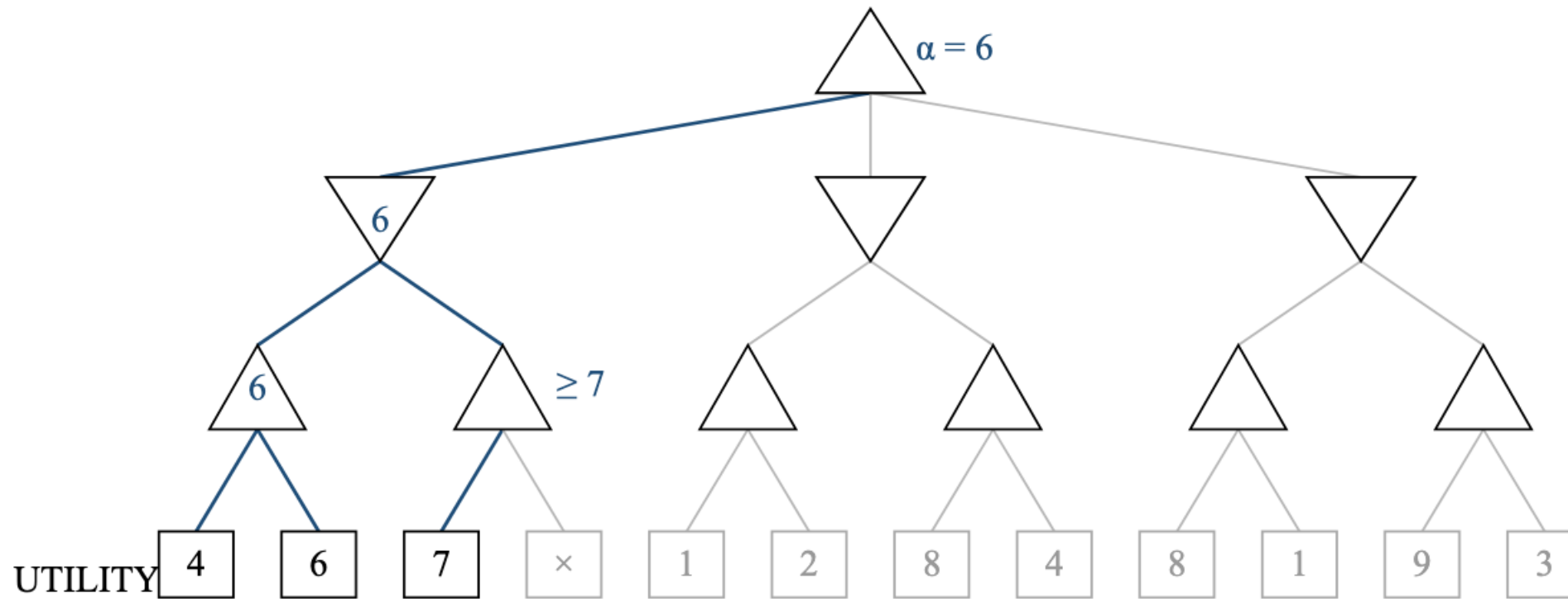
Root $\alpha = -\infty$ · Current $\beta = +\infty$

First MAX Returns 6



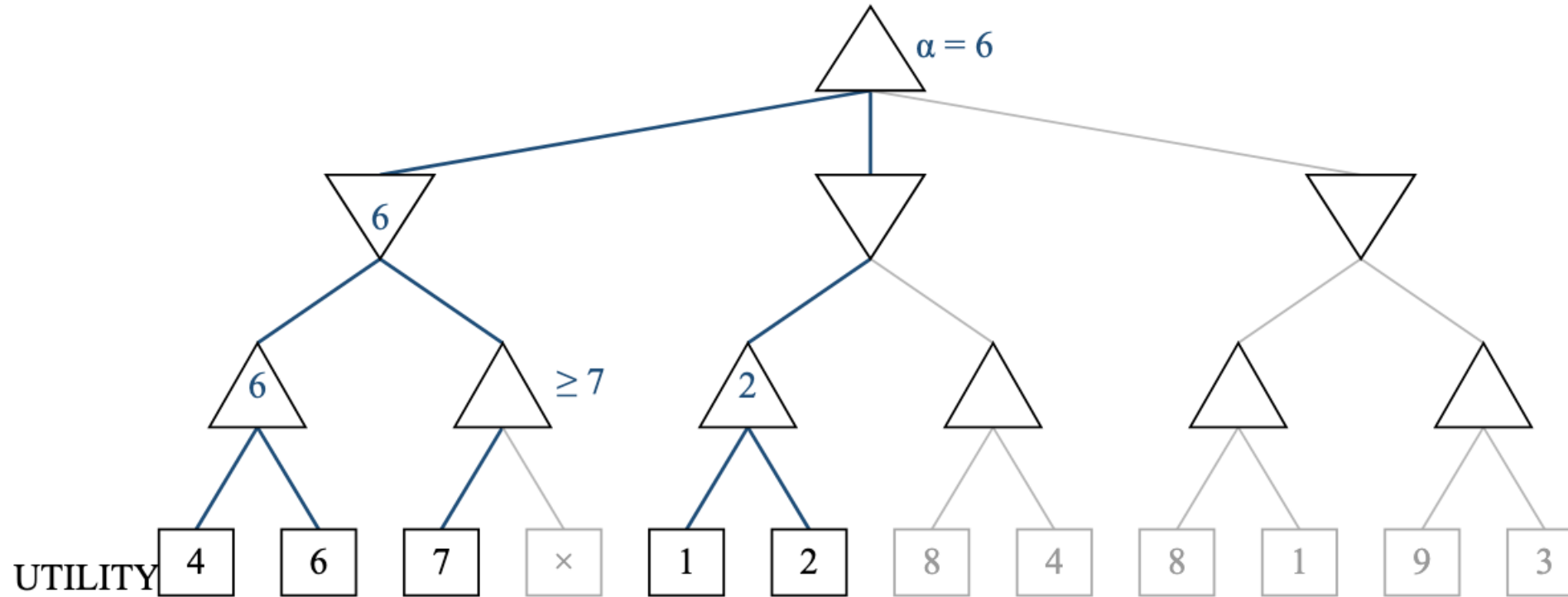
Left MIN updates β to 6

Left MIN Sets Alpha to 6



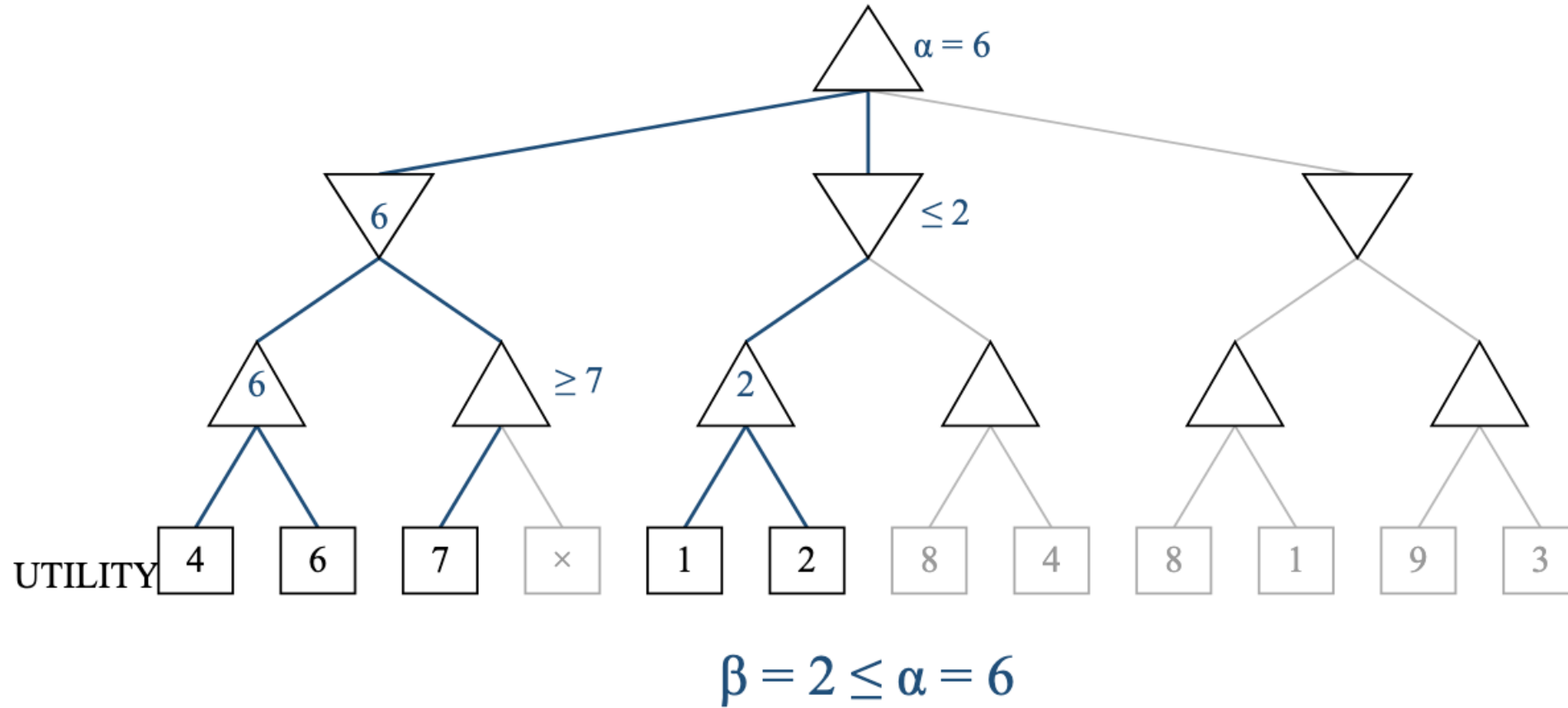
Root updates α to 6

Middle MAX Returns 2

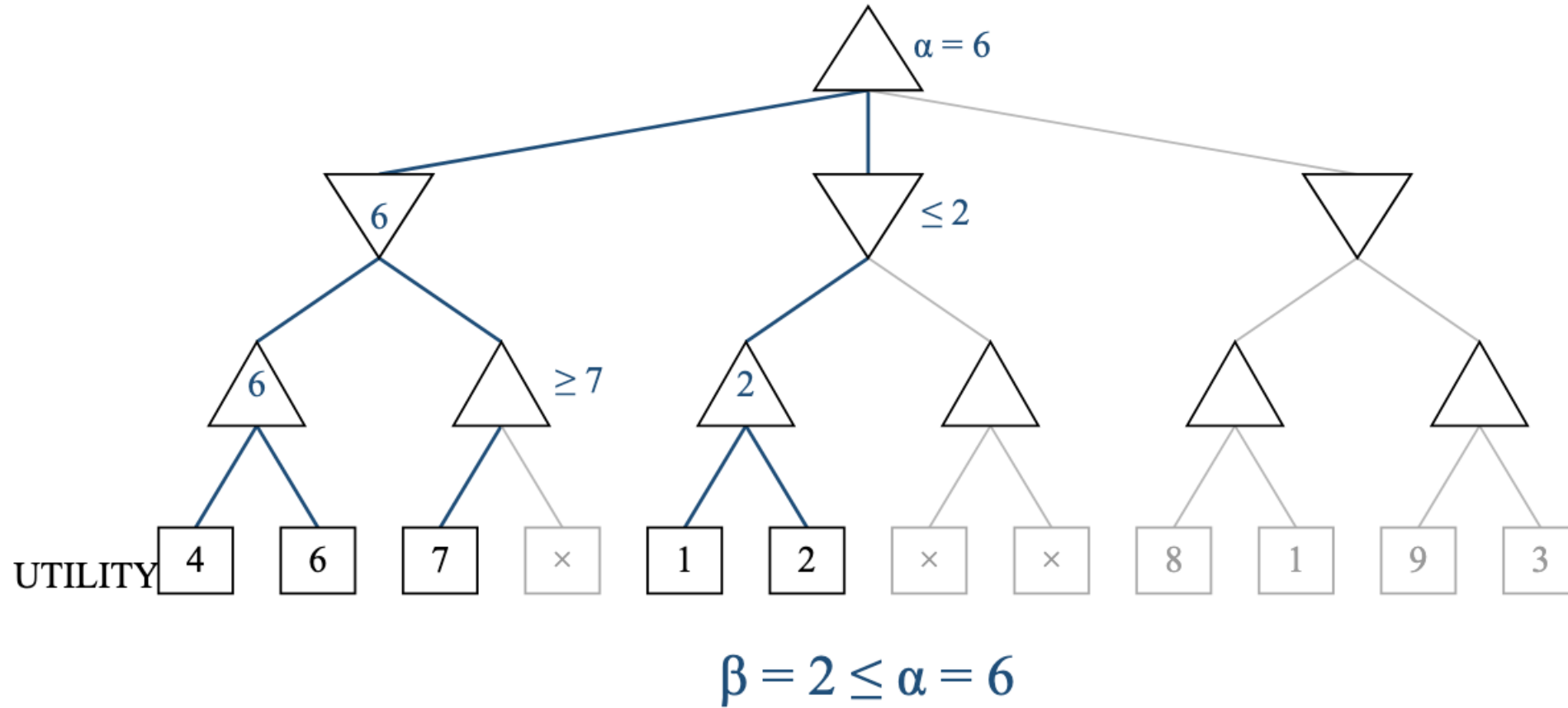


Middle MIN updates β to 2

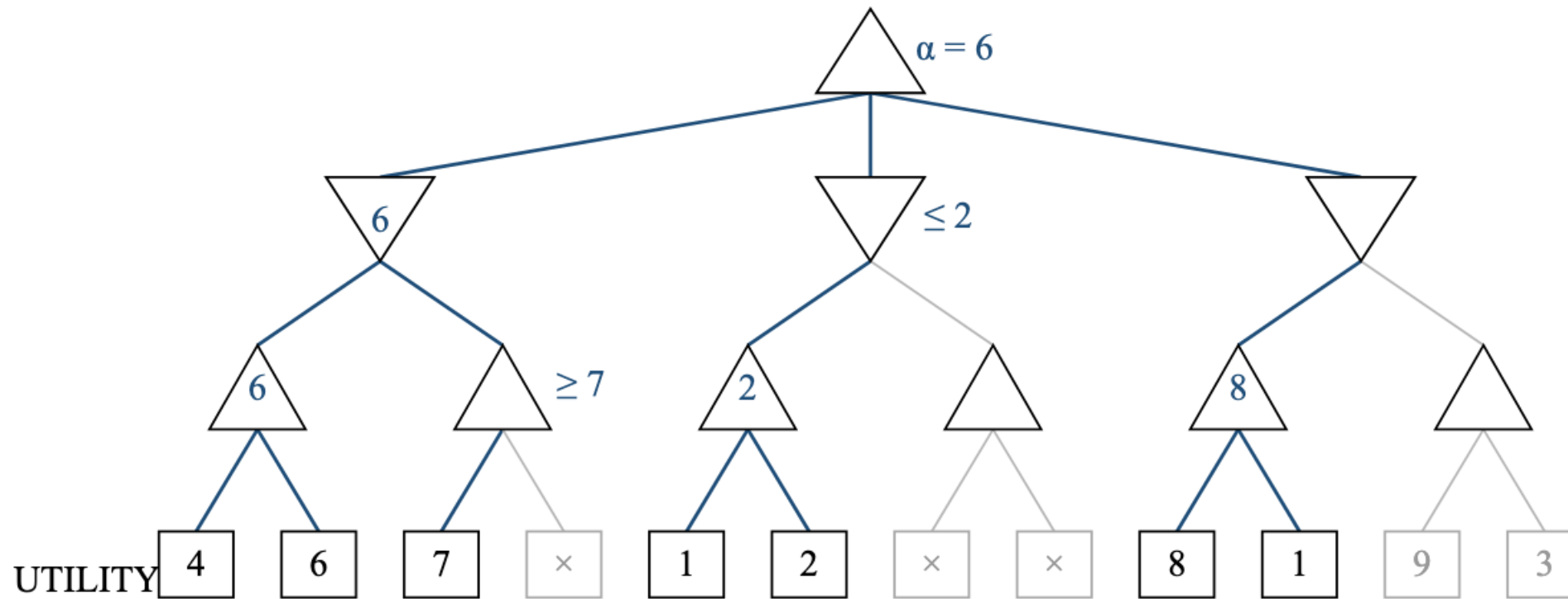
Middle MIN Reaches the Cutoff



Middle Branch Pruned

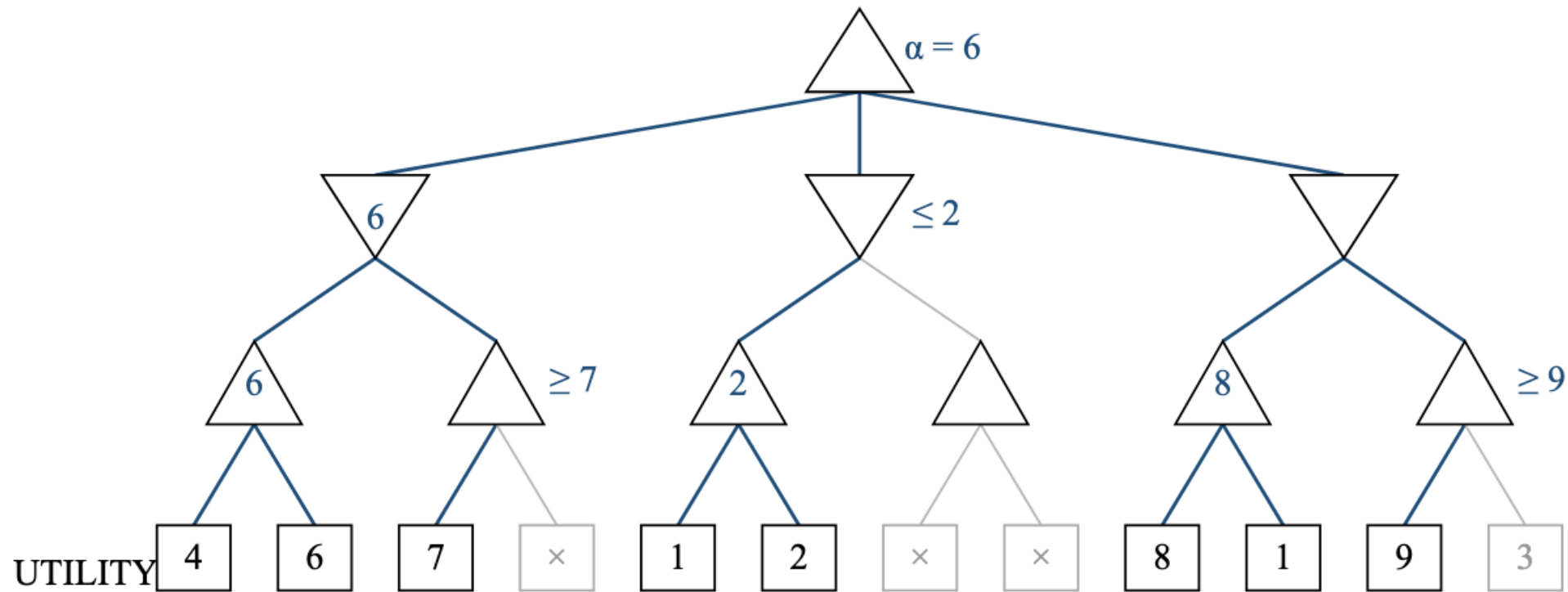


Right MAX Returns 8



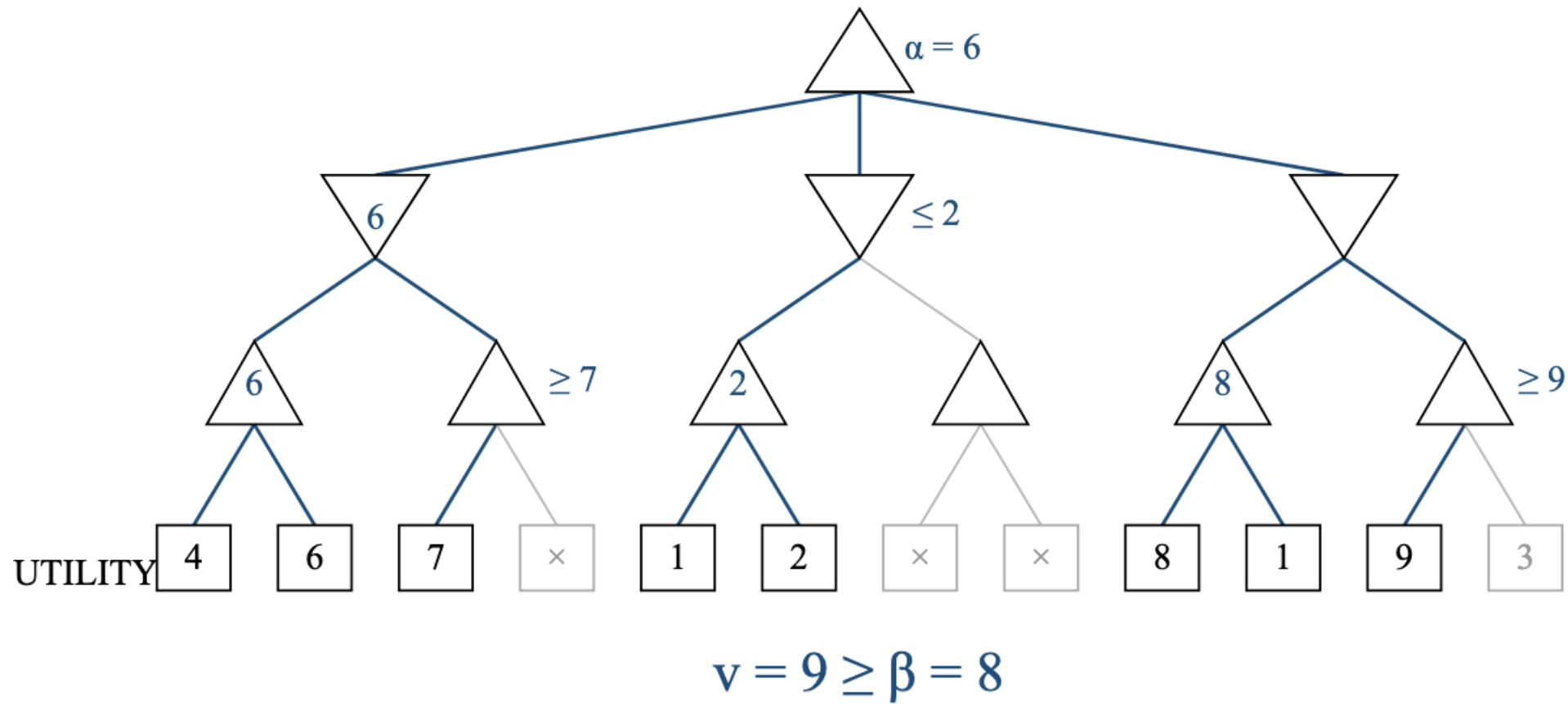
Right MIN updates β to 8

Next MAX Sees 9

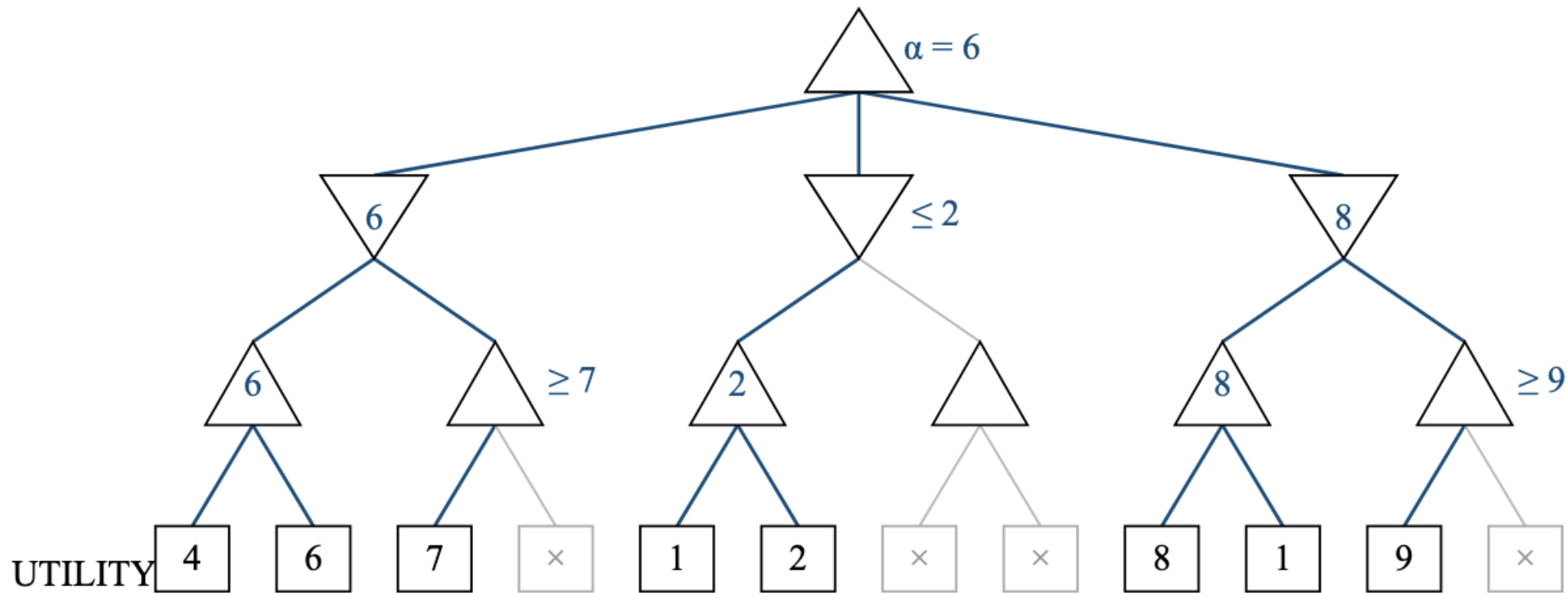


Current $v = 9$ · Right MIN $\beta = 8$

Right MAX Reaches the Cutoff

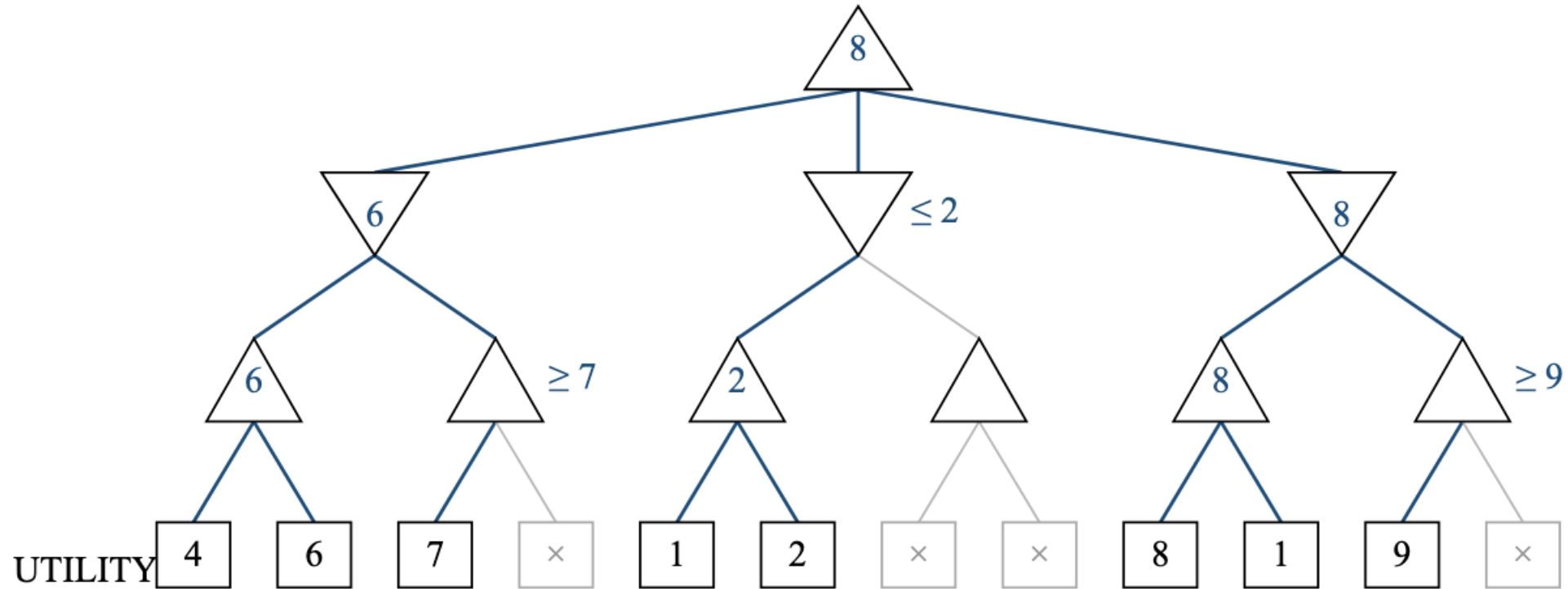


Right MIN Returns 8



Root $\alpha = 6$ · Right value = 8

Root Chooses 8



Final $\alpha = 8$

Alpha–Beta Search in Python

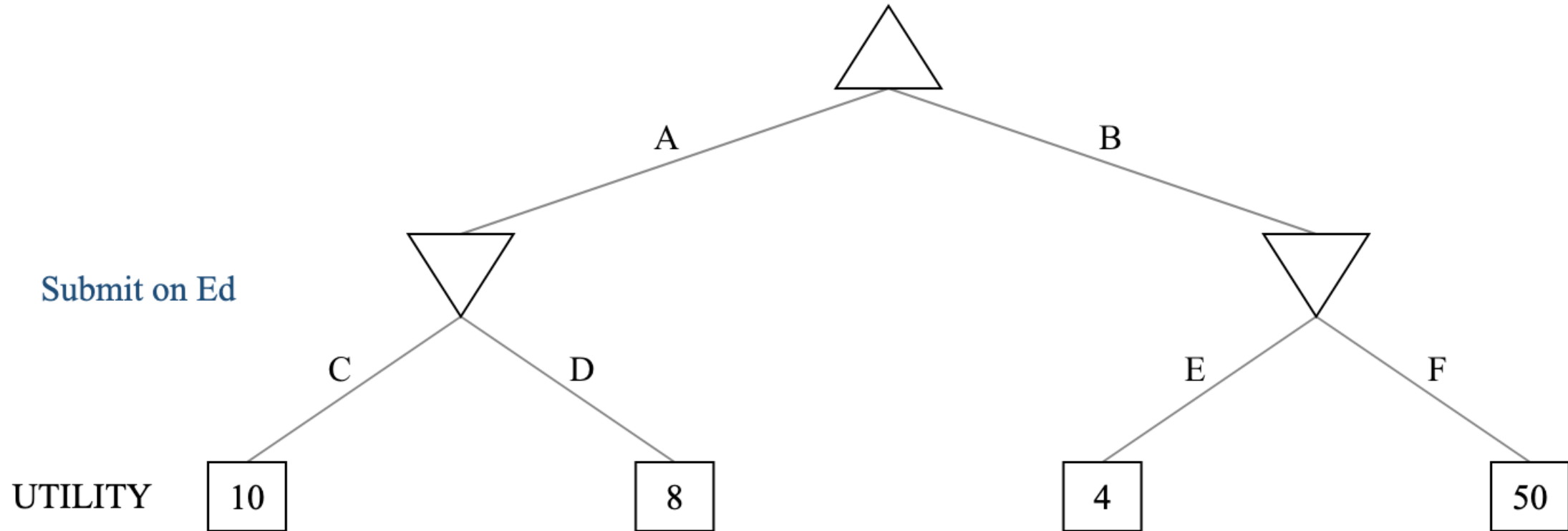
PY main.py

```
1 def alphabeta(tree, alpha, beta, max_turn):
2     if not isinstance(tree, list): return tree
3     value = -float("inf") if max_turn else float("inf")
4     for child in tree:
5         child_value = alphabeta(child, alpha, beta, not max_turn)
6         if max_turn:
7             value = max(value, child_value)
8             alpha = max(alpha, value)
9         else:
10            value = min(value, child_value)
11            beta = min(beta, value)
12            if alpha >= beta: break
13    return value
14 tree = [[[4, 6], [7, 5]], [[1, 2], [8, 4]], [[8, 1], [9, 3]]]
15 print(alphabeta(tree, -float("inf"), float("inf"), True)) # 8
16
```

Task: Alpha–Beta Pruning

TASK

Search Left to Right · Which Branch Is Pruned?

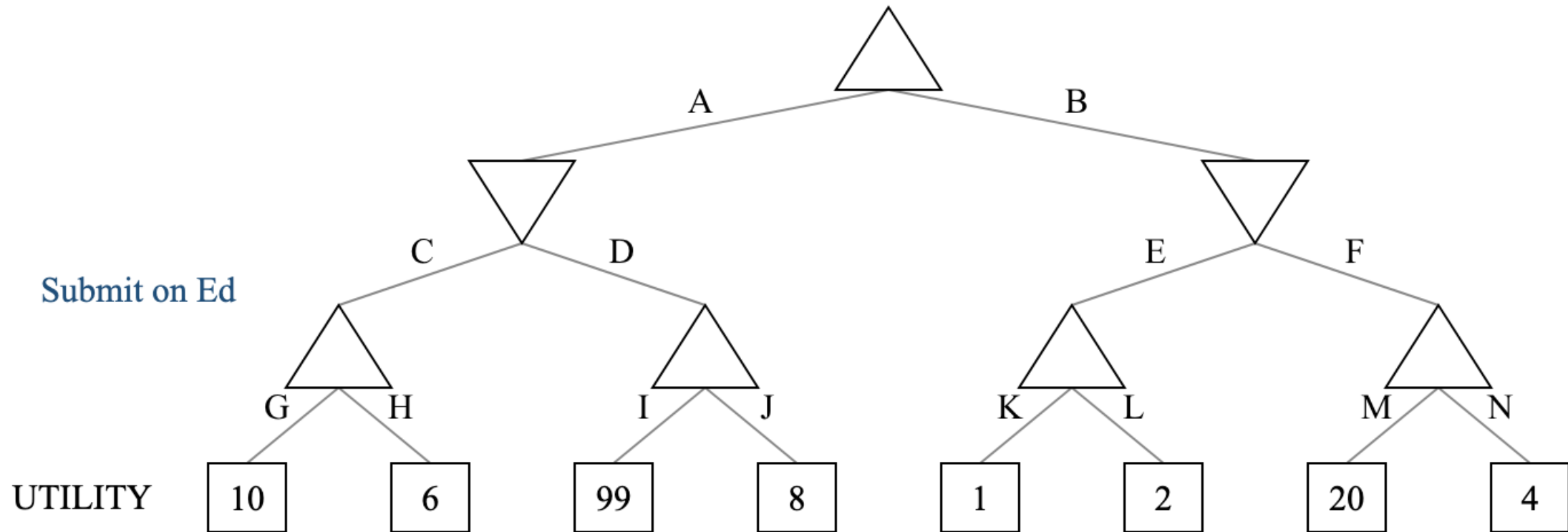


<https://edstem.org/au/courses/41129/lessons/119785/slides/821213>

Task: Deeper Alpha–Beta Pruning

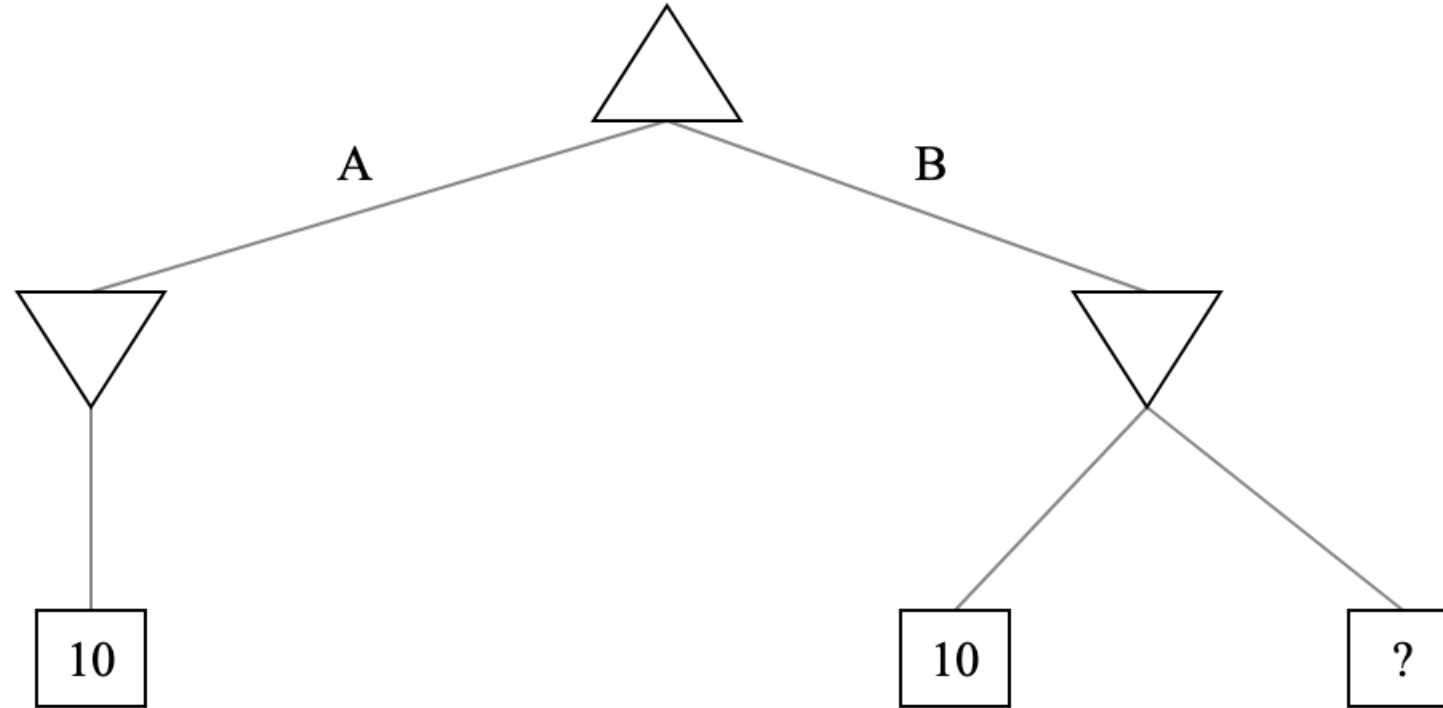
TASK

Search Left to Right · Which Branches Are Pruned?



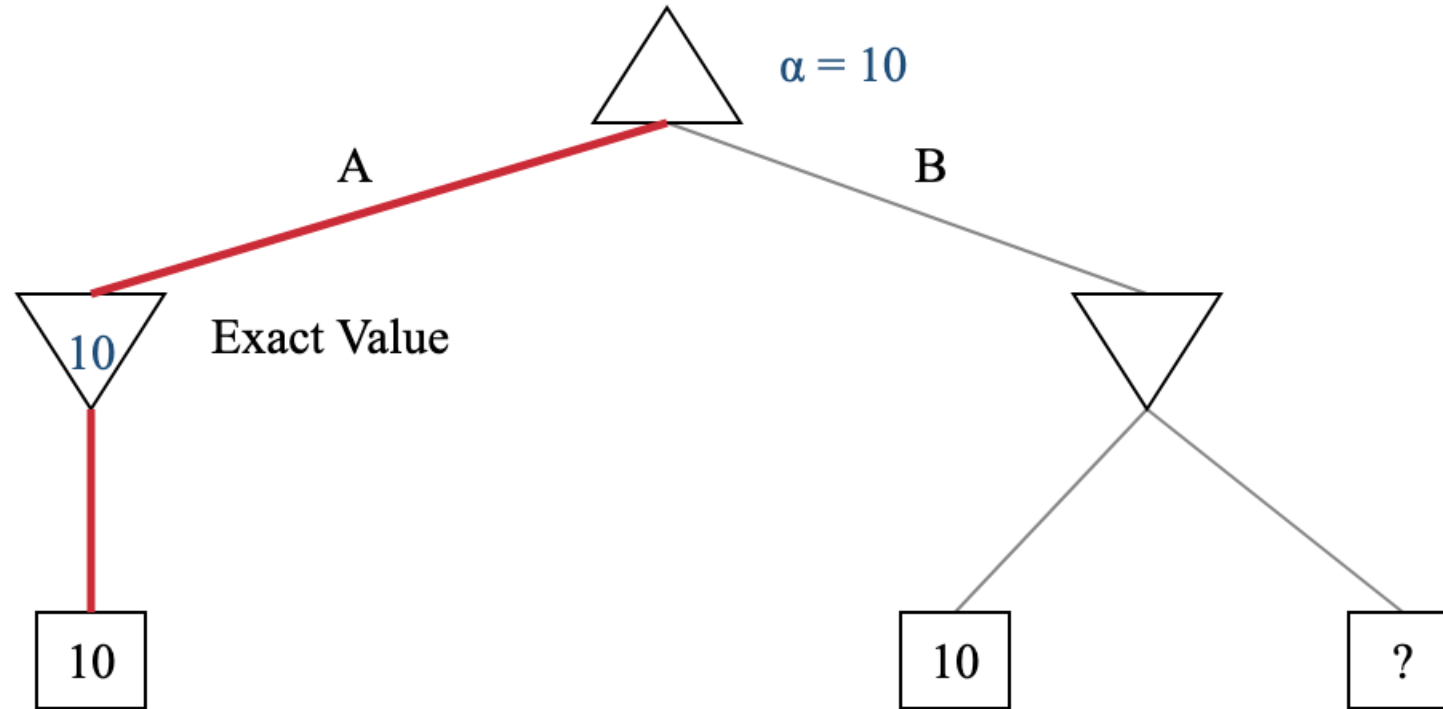
<https://edstem.org/au/courses/41129/lessons/119786/slides/821214>

A Bound Is Not an Exact Value



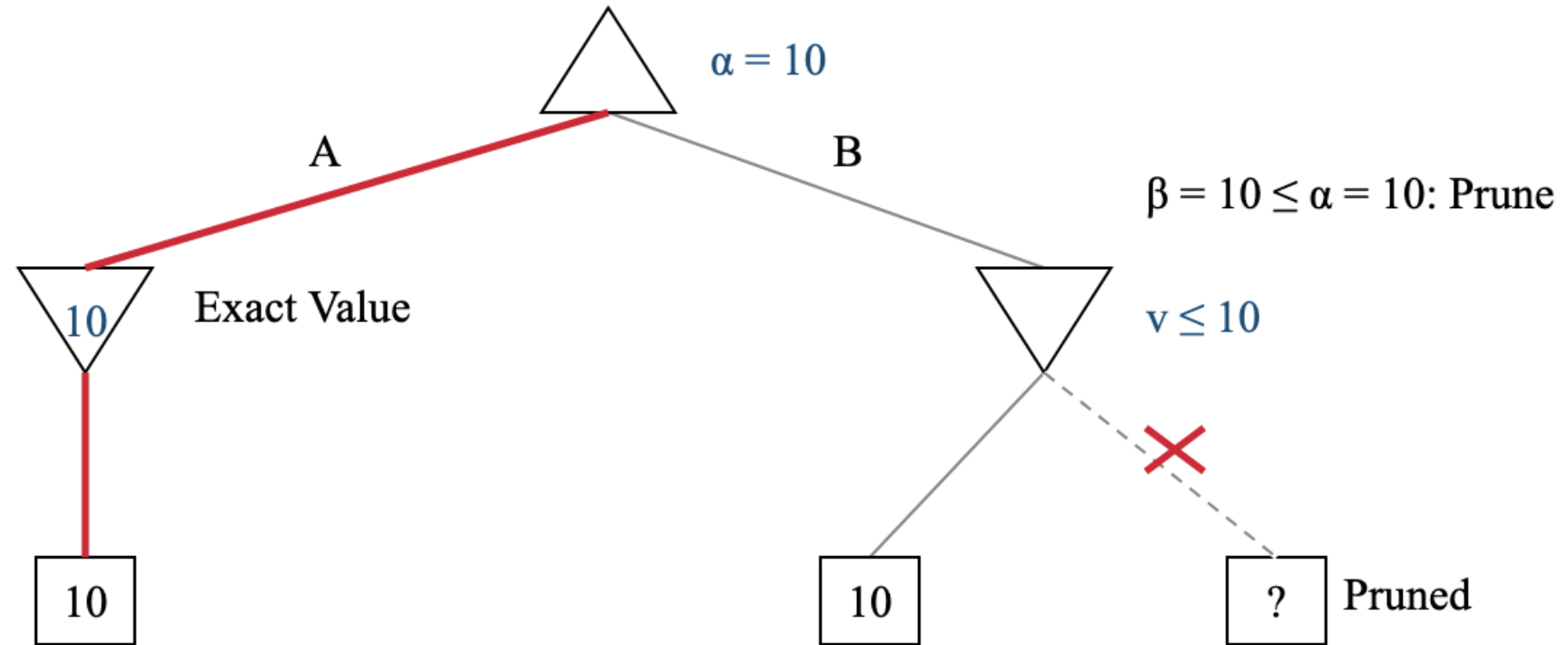
Search Left to Right · Which Move Should MAX Keep?

A Bound Is Not an Exact Value



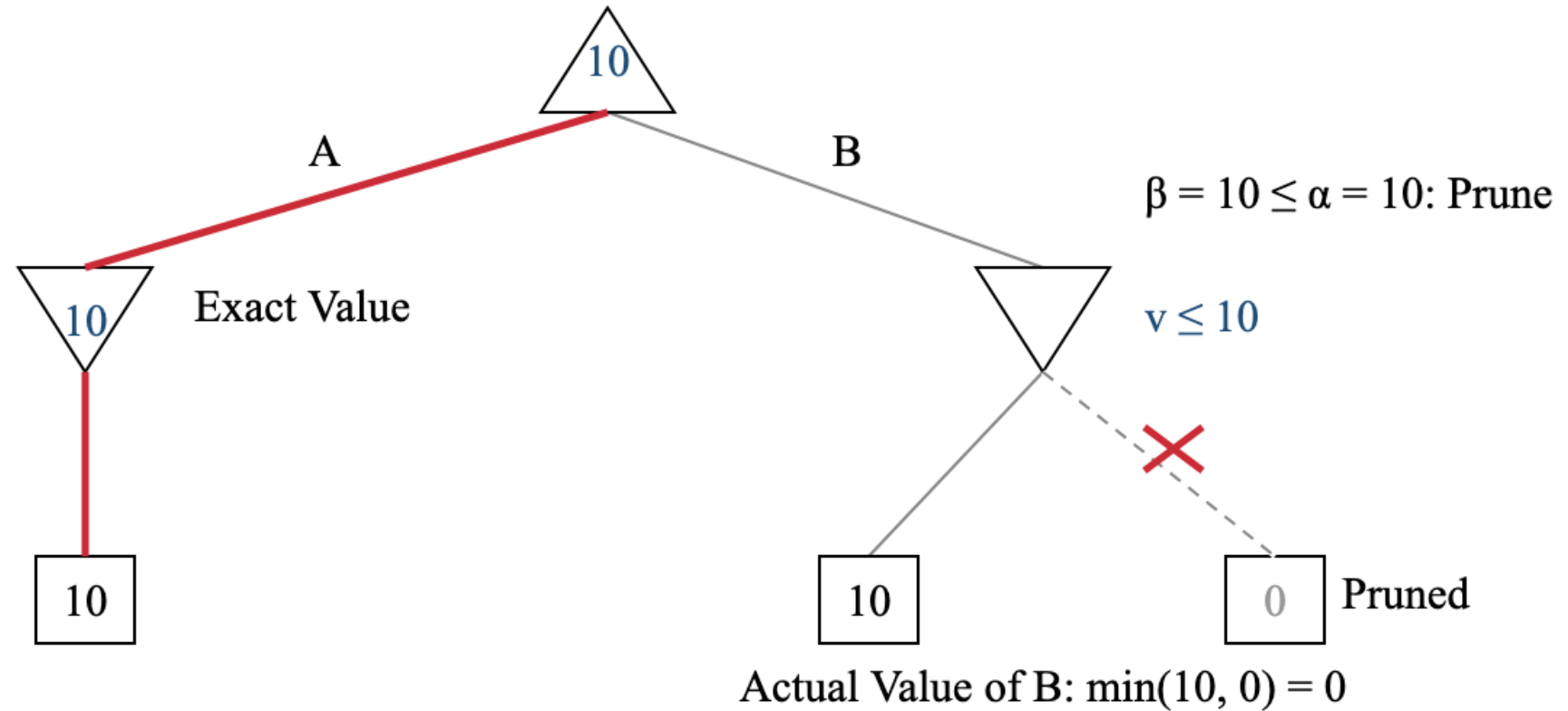
A Returns 10 · Keep A and Set $\alpha = 10$

A Bound Is Not an Exact Value



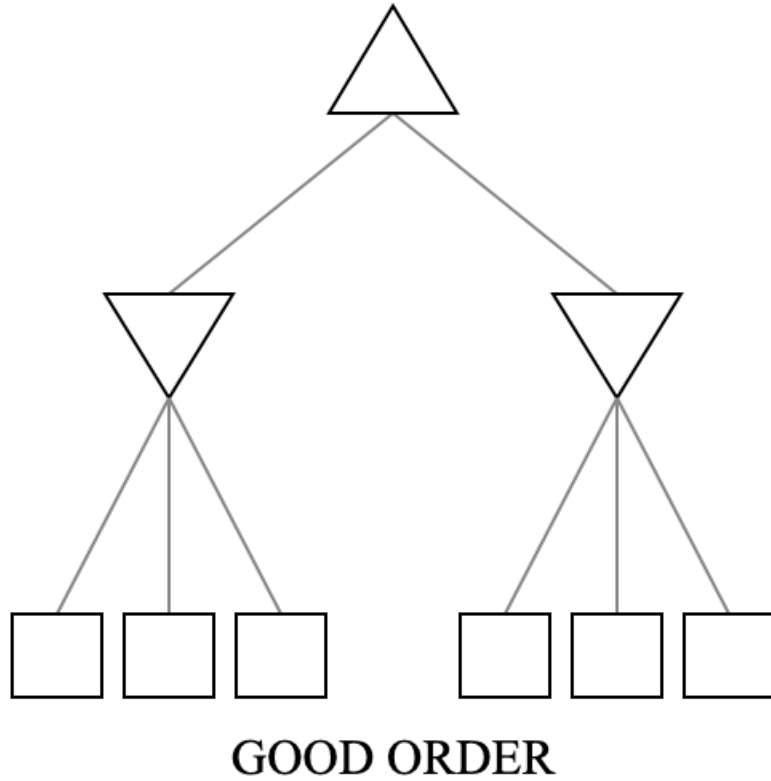
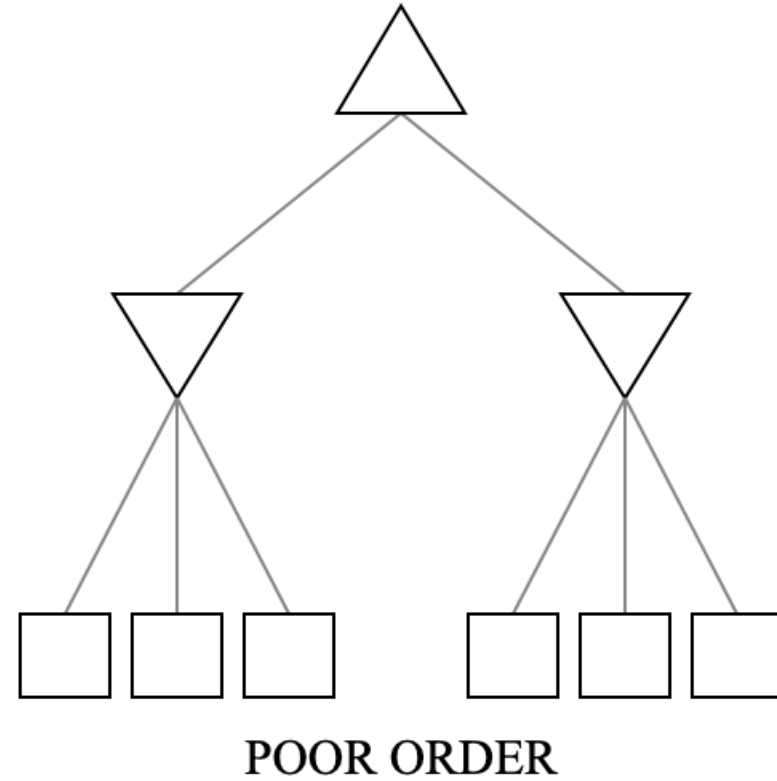
B Returns 10 as a Bound · This Does Not Prove a Tie

A Bound Is Not an Exact Value

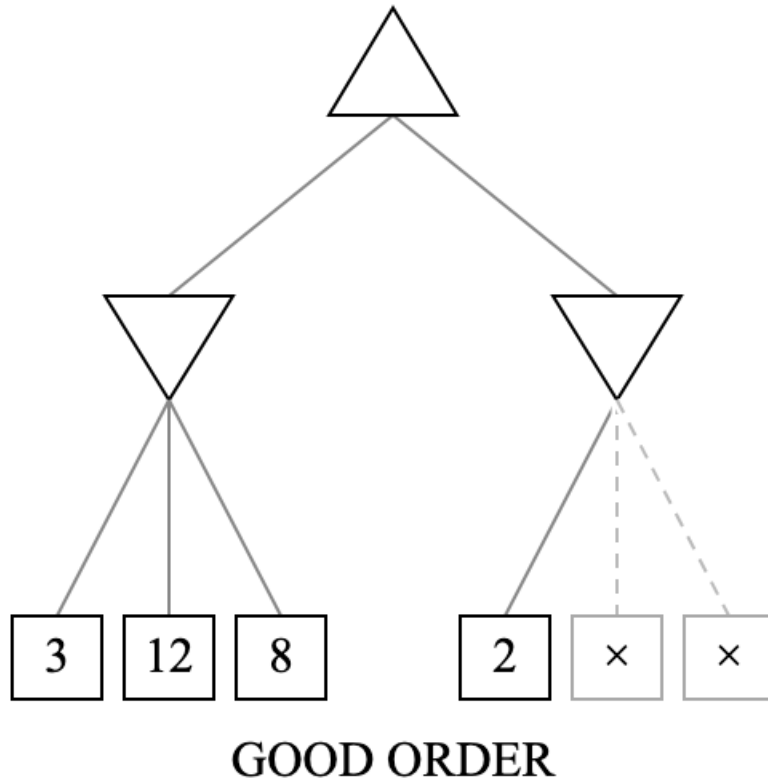
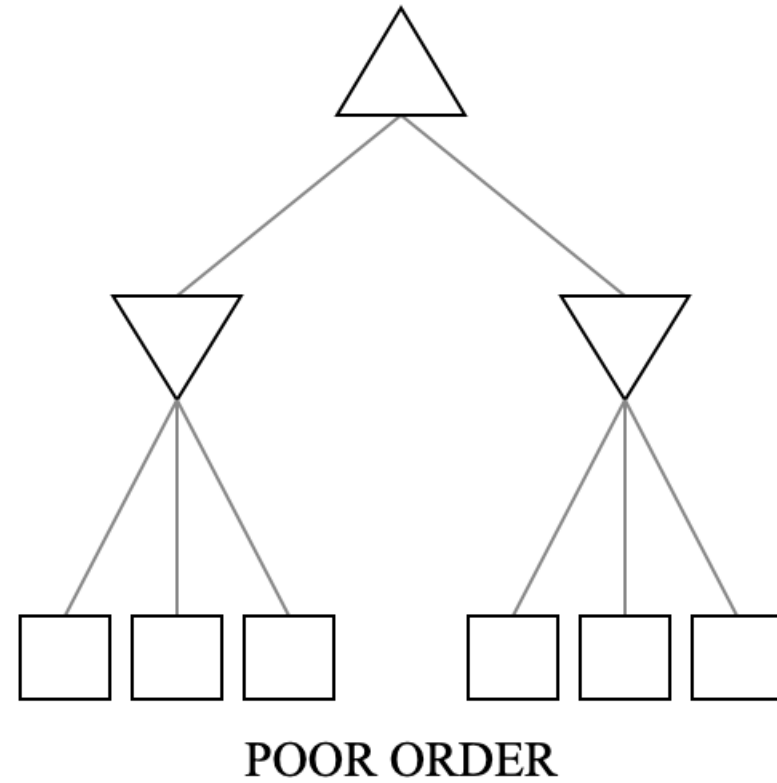


Keep A · Replace the Best Move Only on a Strict Improvement

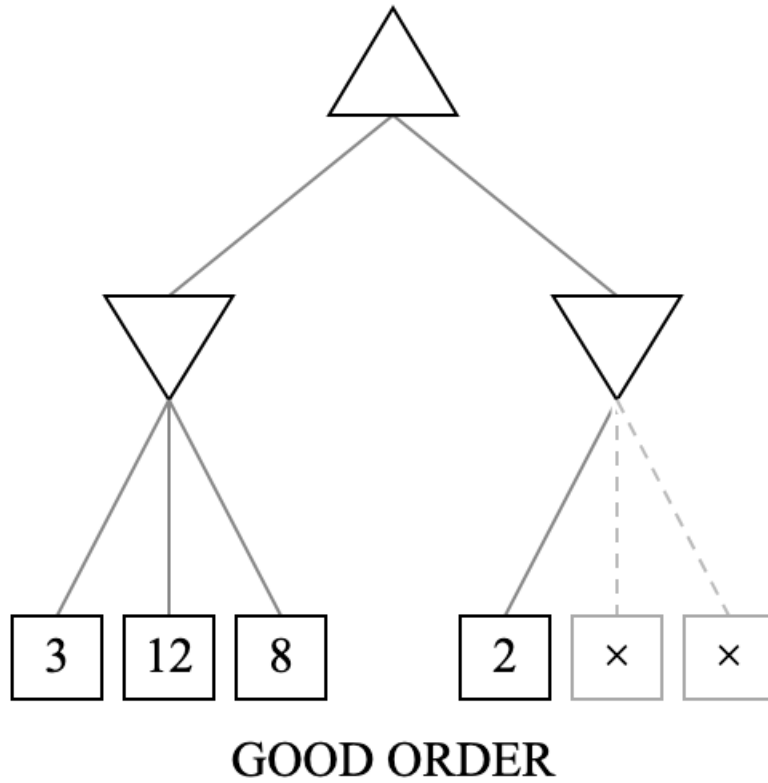
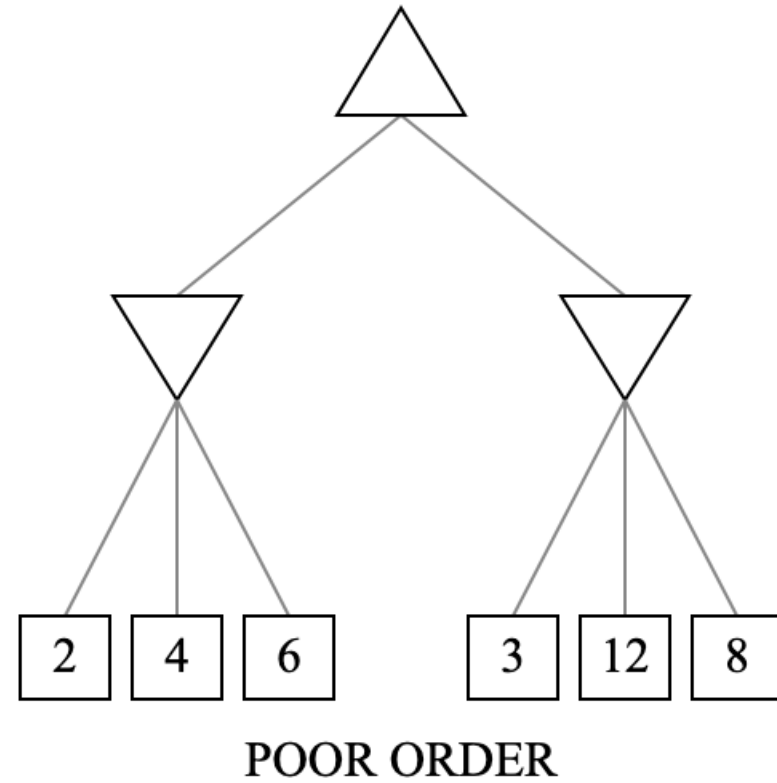
Move Ordering Controls Pruning



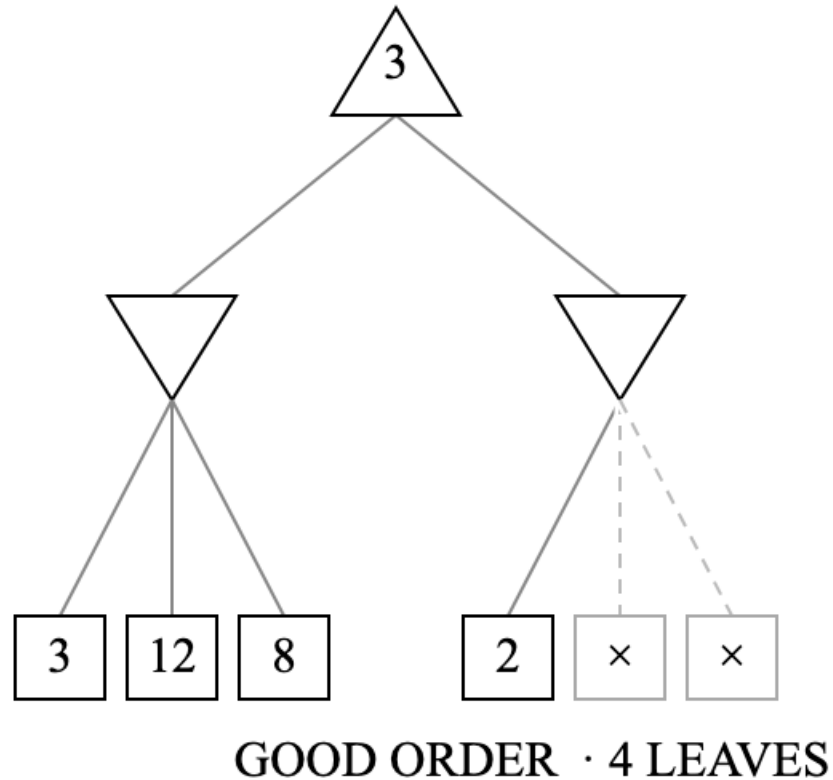
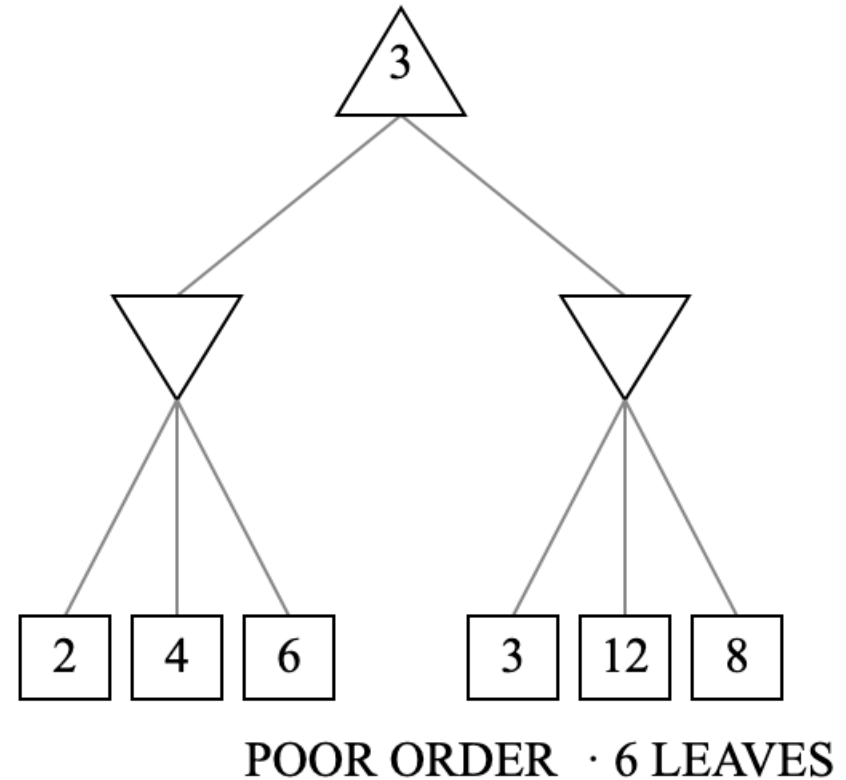
Move Ordering Controls Pruning



Move Ordering Controls Pruning



Move Ordering Controls Pruning



Same Minimax Value: 3 · Different Amount of Search

Good Ordering Searches Much Less

Minimax: $O(b^m)$

Knuth & Moore (1975), An Analysis of Alpha-Beta Pruning

Good Ordering Searches Much Less

Minimax: $O(b^m)$

Ideal Alpha–Beta Ordering: $O(b^{m/2})$

Knuth & Moore (1975), An Analysis of Alpha-Beta Pruning

Good Ordering Searches Much Less

Minimax: $O(b^m)$

Ideal Alpha–Beta Ordering: $O(b^{m/2})$

Roughly Twice the Search Depth for the Same Node Budget

Knuth & Moore (1975), An Analysis of Alpha-Beta Pruning

Good Ordering Searches Much Less

Minimax: $O(b^m)$

Ideal Alpha–Beta Ordering: $O(b^{m/2})$

Roughly Twice the Search Depth for the Same Node Budget

Example: $b = 10$ and $m = 6$ Gives 10^6 Leaves vs About 10^3

Knuth & Moore (1975), An Analysis of Alpha-Beta Pruning

Good Ordering Searches Much Less

Minimax: $O(b^m)$

Ideal Alpha–Beta Ordering: $O(b^{m/2})$

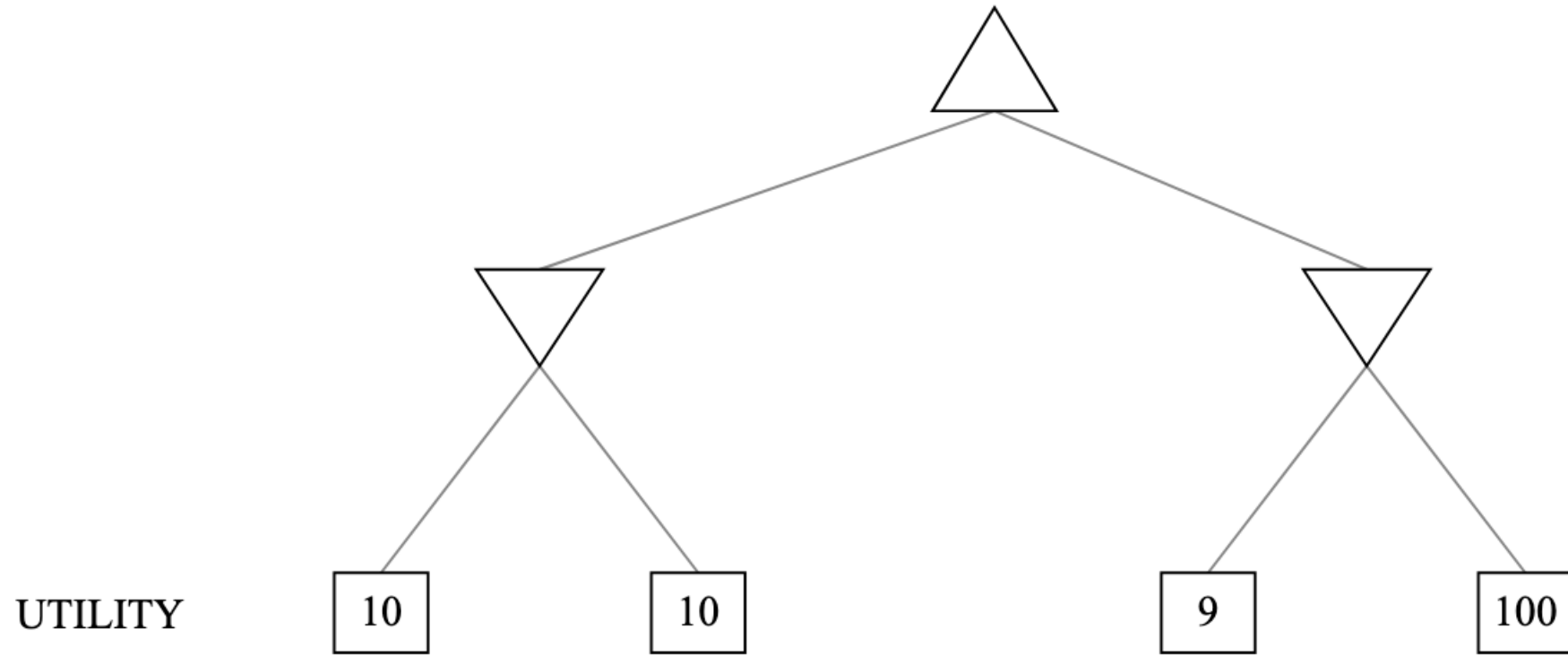
Roughly Twice the Search Depth for the Same Node Budget

Example: $b = 10$ and $m = 6$ Gives 10^6 Leaves vs About 10^3

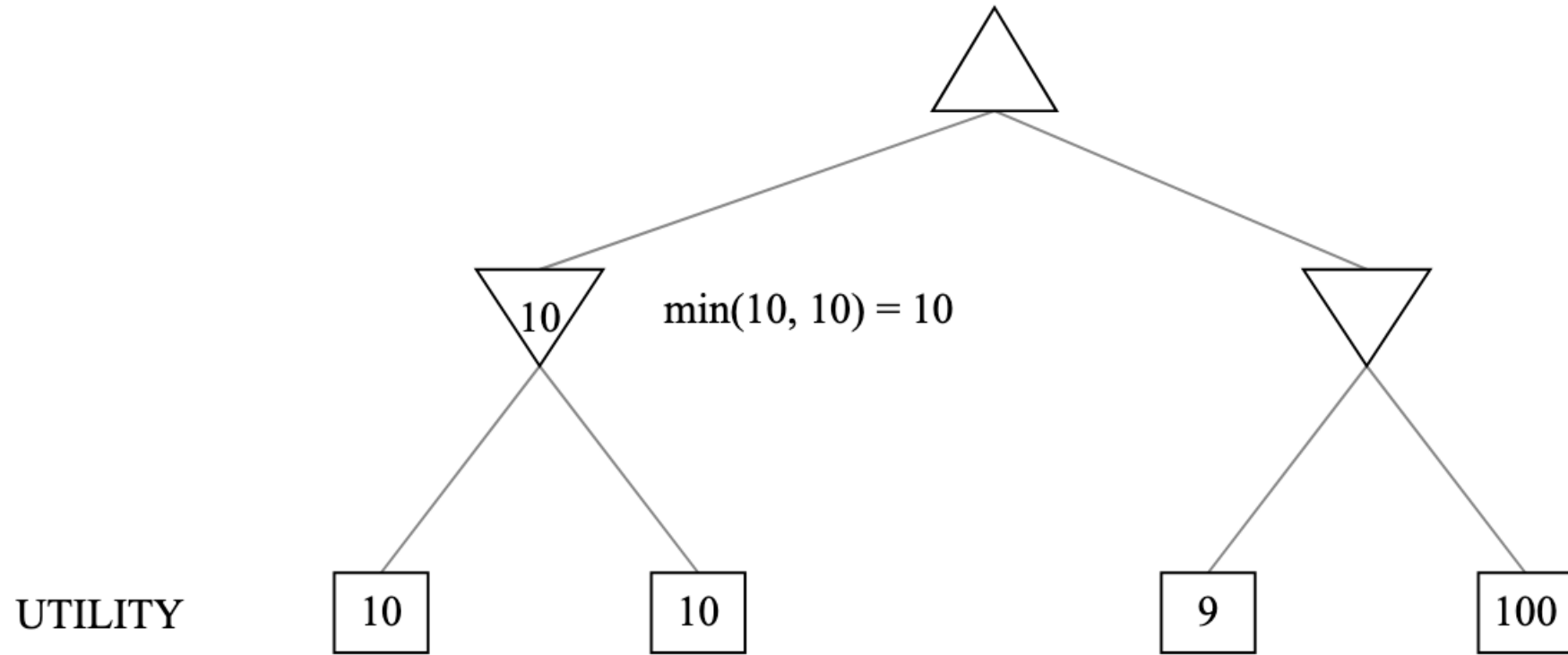
Worst Case Remains $O(b^m)$

Knuth & Moore (1975), An Analysis of Alpha-Beta Pruning

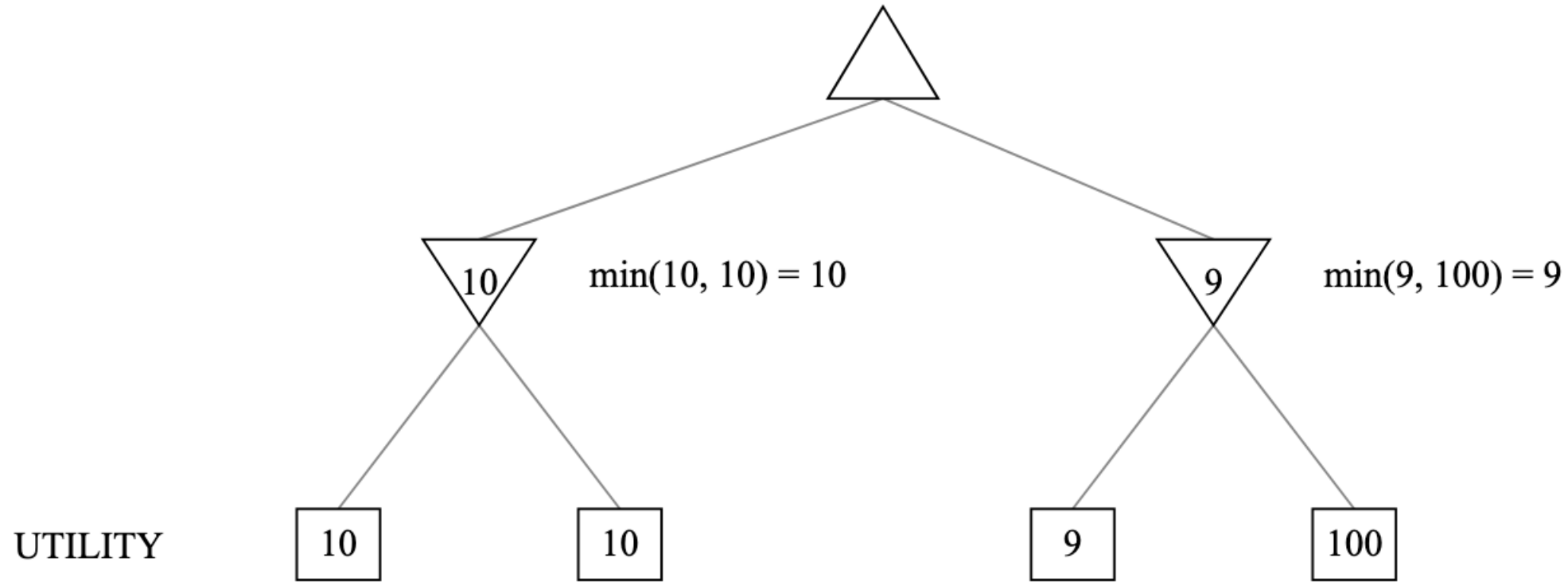
Worst Case Uses MIN's Choice



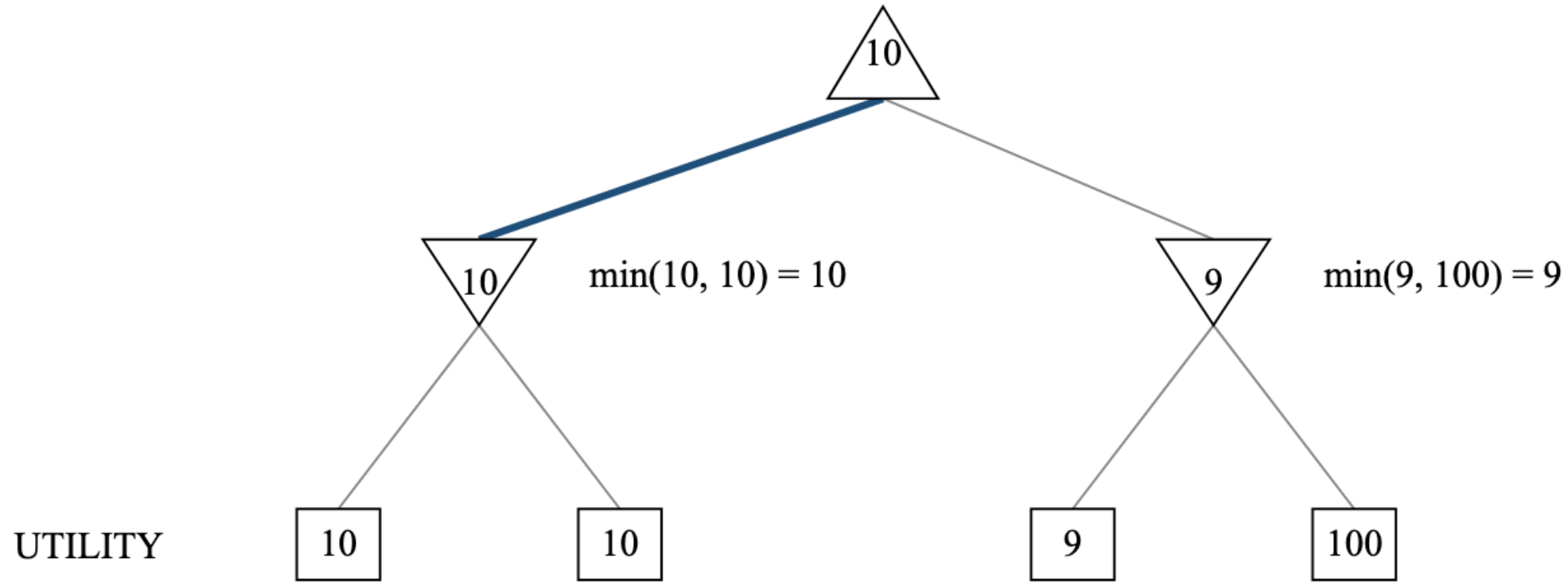
Worst Case Uses MIN's Choice



Worst Case Uses MIN's Choice

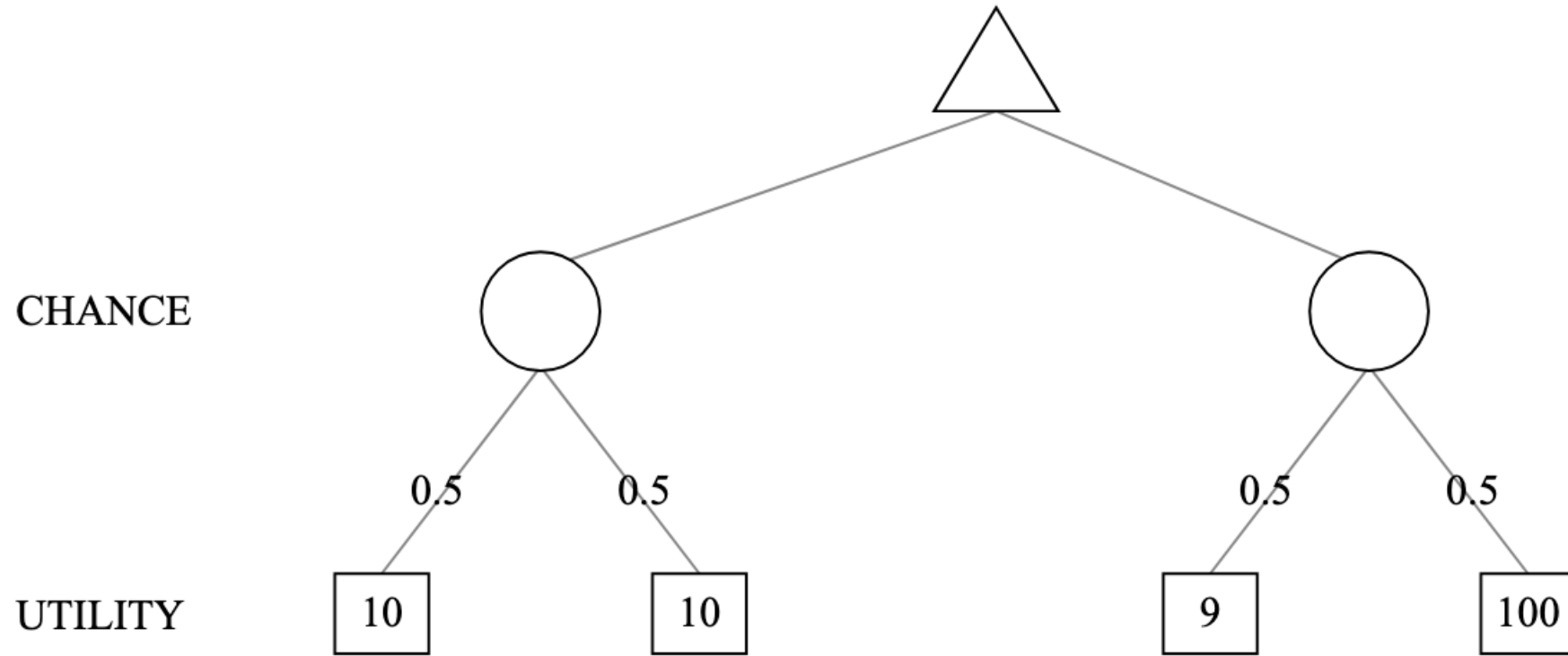


Worst Case Uses MIN's Choice

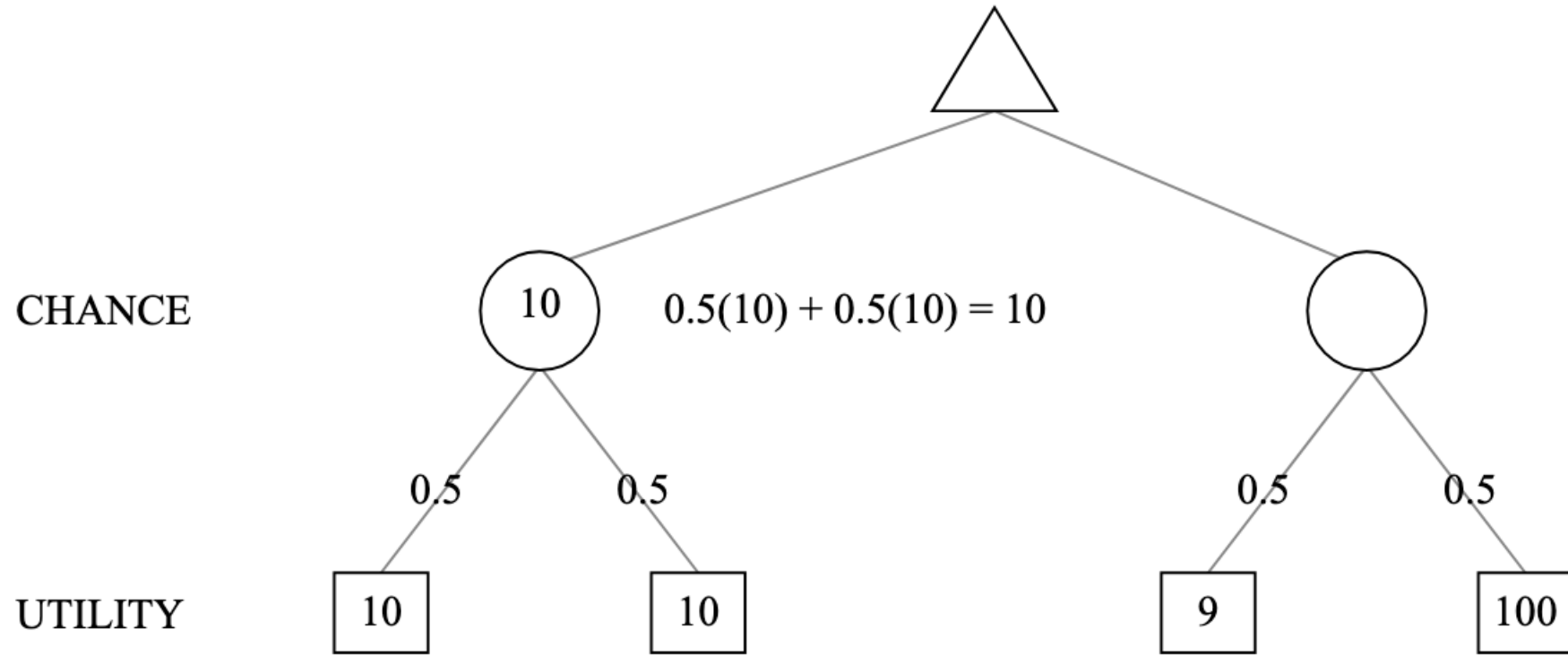


MAX Chooses Left: $10 > 9$

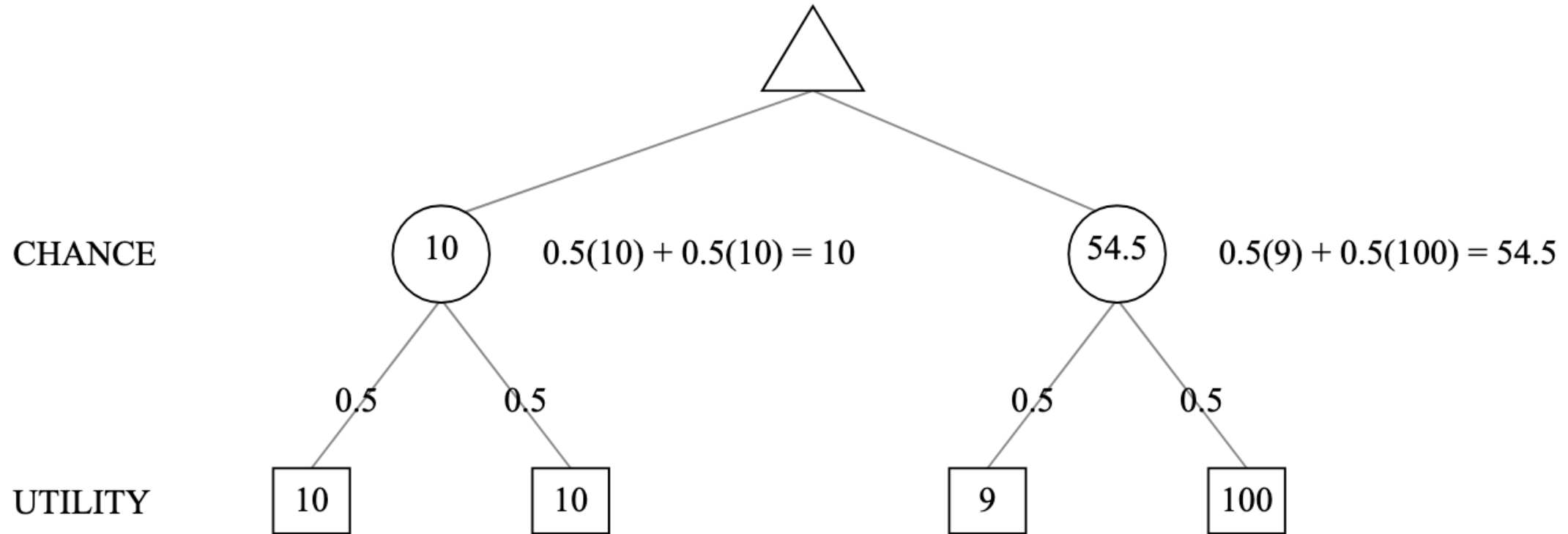
Chance Uses the Expected Value



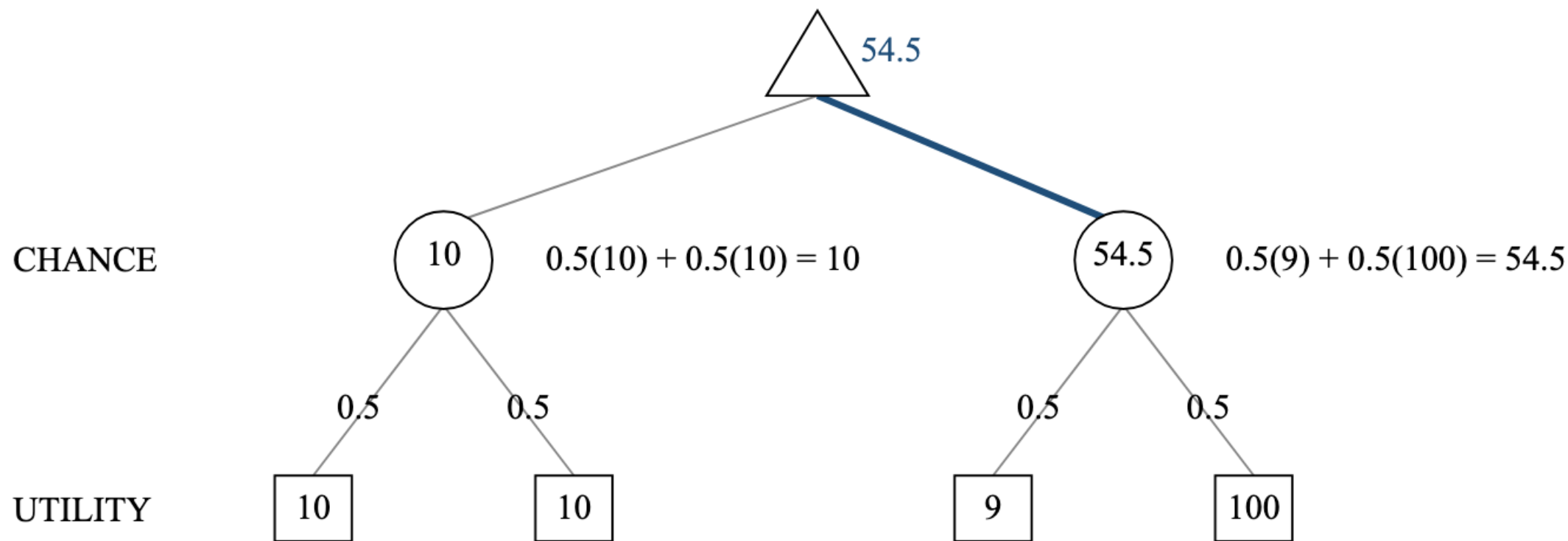
Chance Uses the Expected Value



Chance Uses the Expected Value



Chance Uses the Expected Value



MAX Chooses Right: $54.5 > 10$

Why Chance Nodes Appear

Explicit randomness

Dice, shuffled cards, noisy outcomes

Agent model

An opponent policy represented by probabilities

Action failure

The intended action may not occur

Why Chance Nodes Appear

Explicit randomness

Dice, shuffled cards, noisy outcomes

Agent model

An opponent policy represented by probabilities

Action failure

The intended action may not occur

A Chance Node Needs a Probability Model

Why Chance Nodes Appear

Explicit randomness

Dice, shuffled cards, noisy outcomes

Agent model

An opponent policy represented by probabilities

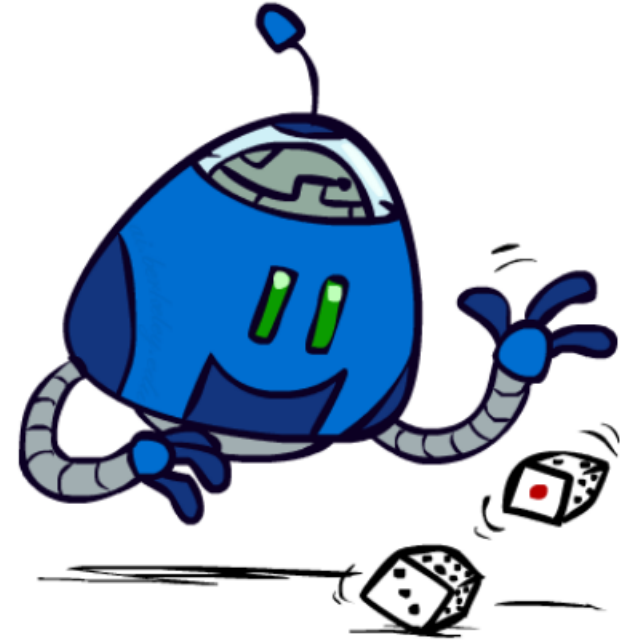
Action failure

The intended action may not occur

A Chance Node Needs a Probability Model

Expected Value Is Only as Good as That Model

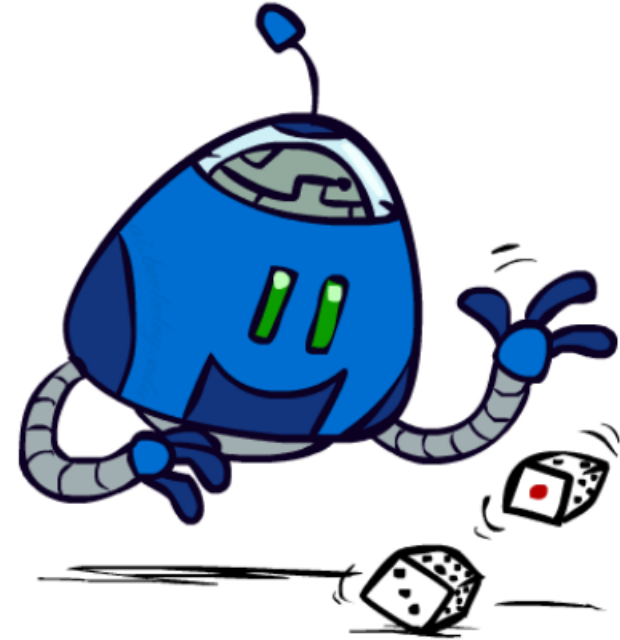
Expectimax Search



Expectimax Search

MAX Node

Take the Largest Child Value



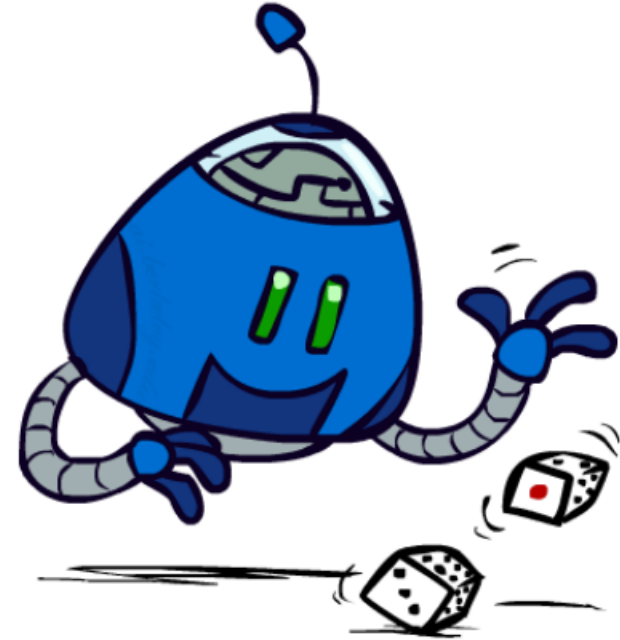
Expectimax Search

MAX Node

Take the Largest Child Value

CHANCE Node

Take the Probability-Weighted Average



Expectimax Search

MAX Node

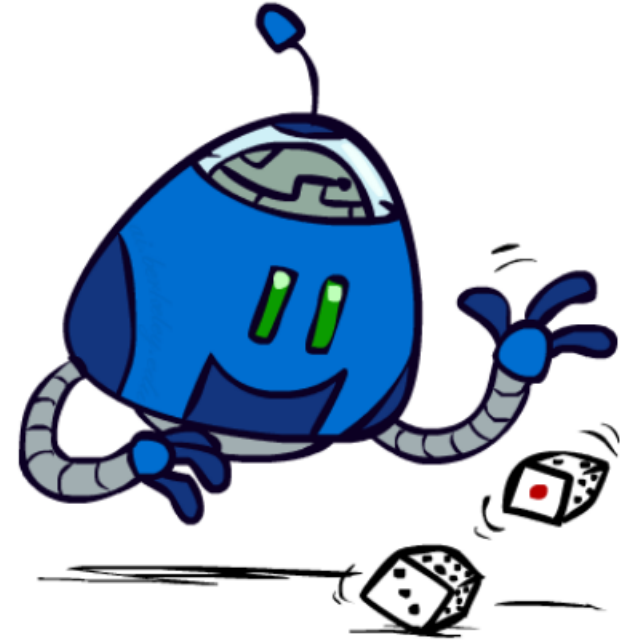
Take the Largest Child Value

CHANCE Node

Take the Probability-Weighted Average

Terminal Node

Return Utility



Expectimax Search

MAX Node

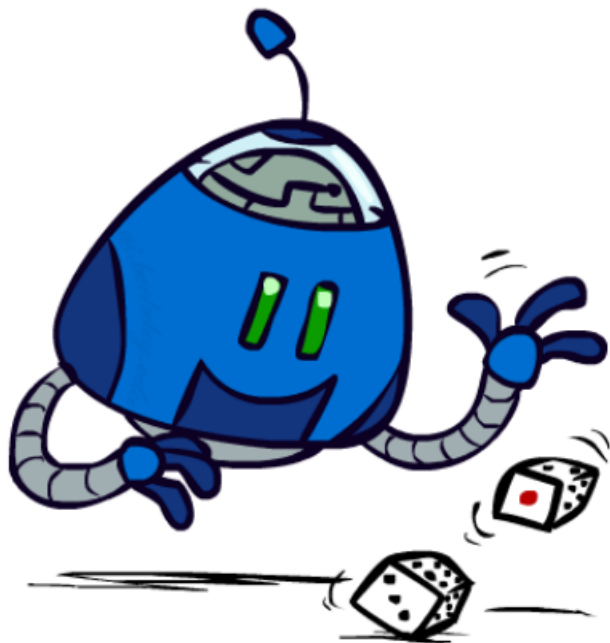
Take the Largest Child Value

CHANCE Node

Take the Probability-Weighted Average

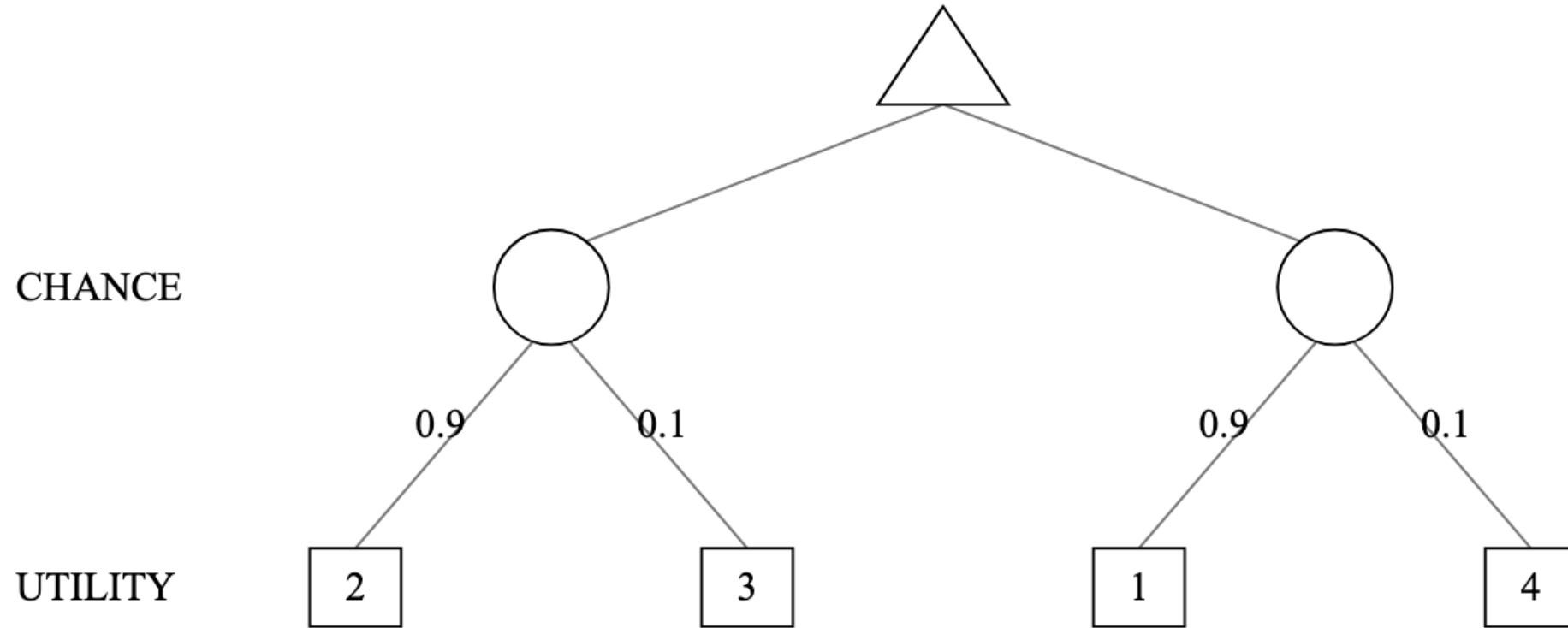
Terminal Node

Return Utility

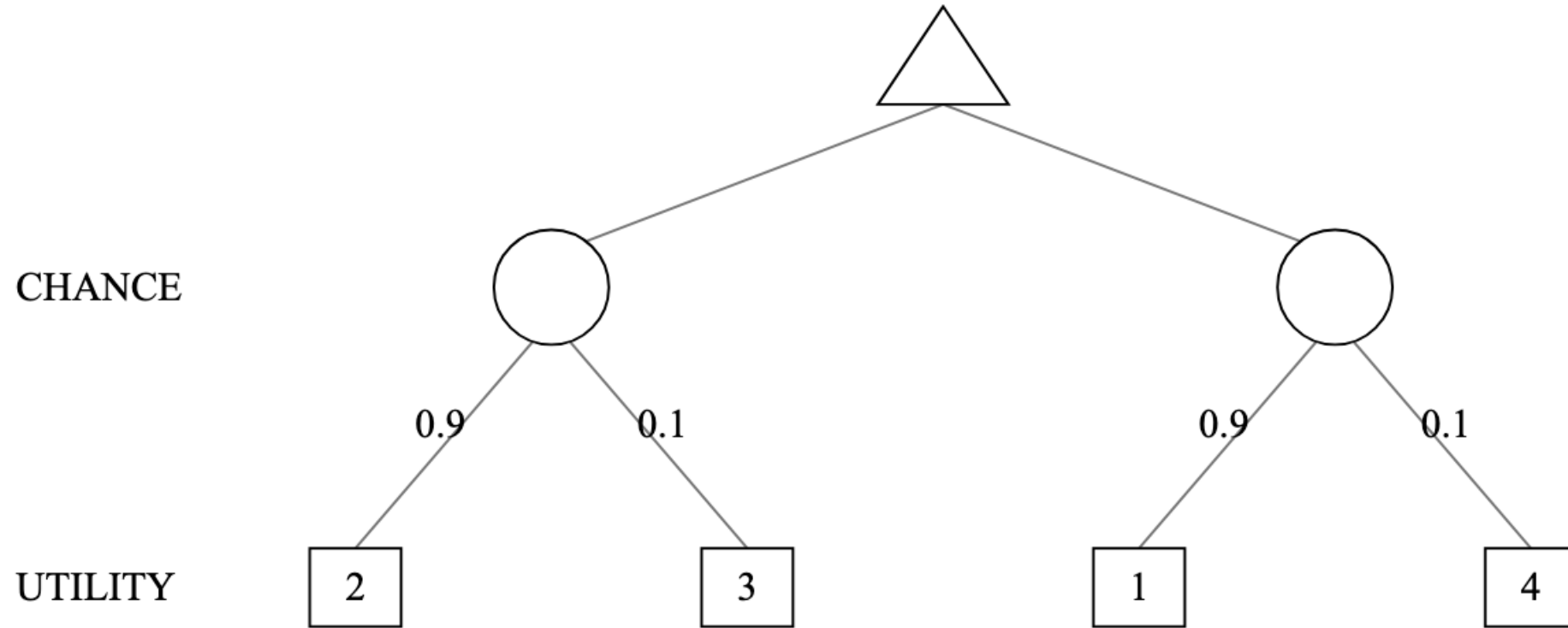


$$\text{CHANCE: } V(s) = \sum p_i V(s_i)$$

Expectimax Example

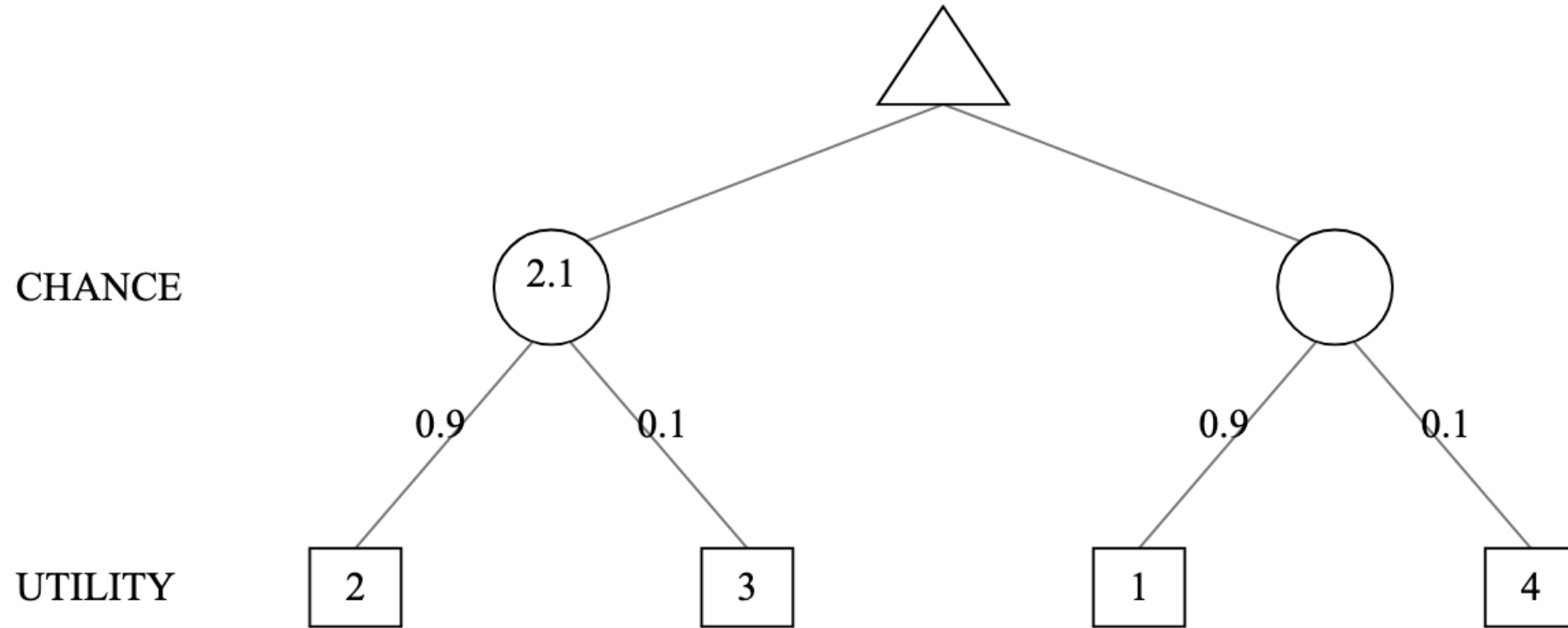


Expectimax Example



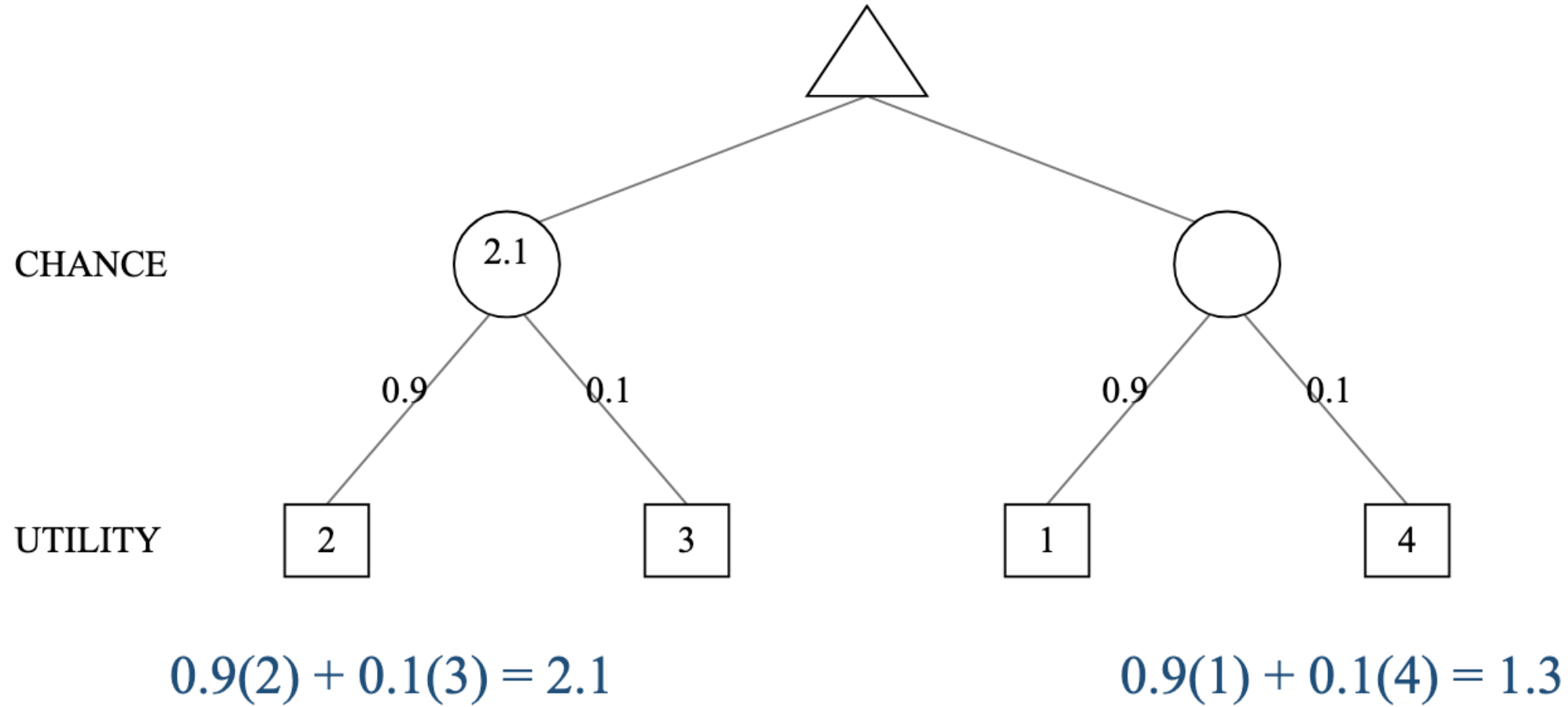
$$0.9(2) + 0.1(3) = 2.1$$

Expectimax Example

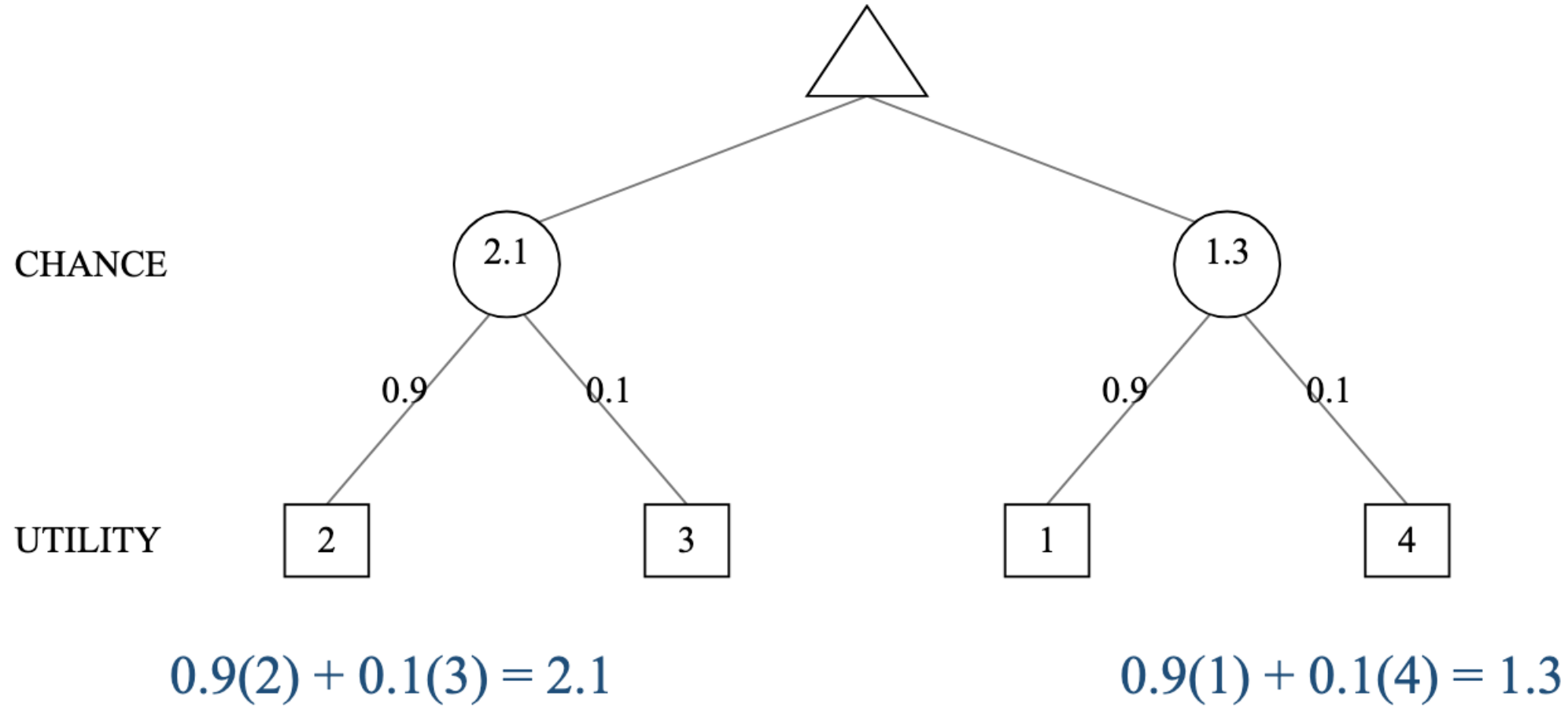


$$0.9(2) + 0.1(3) = 2.1$$

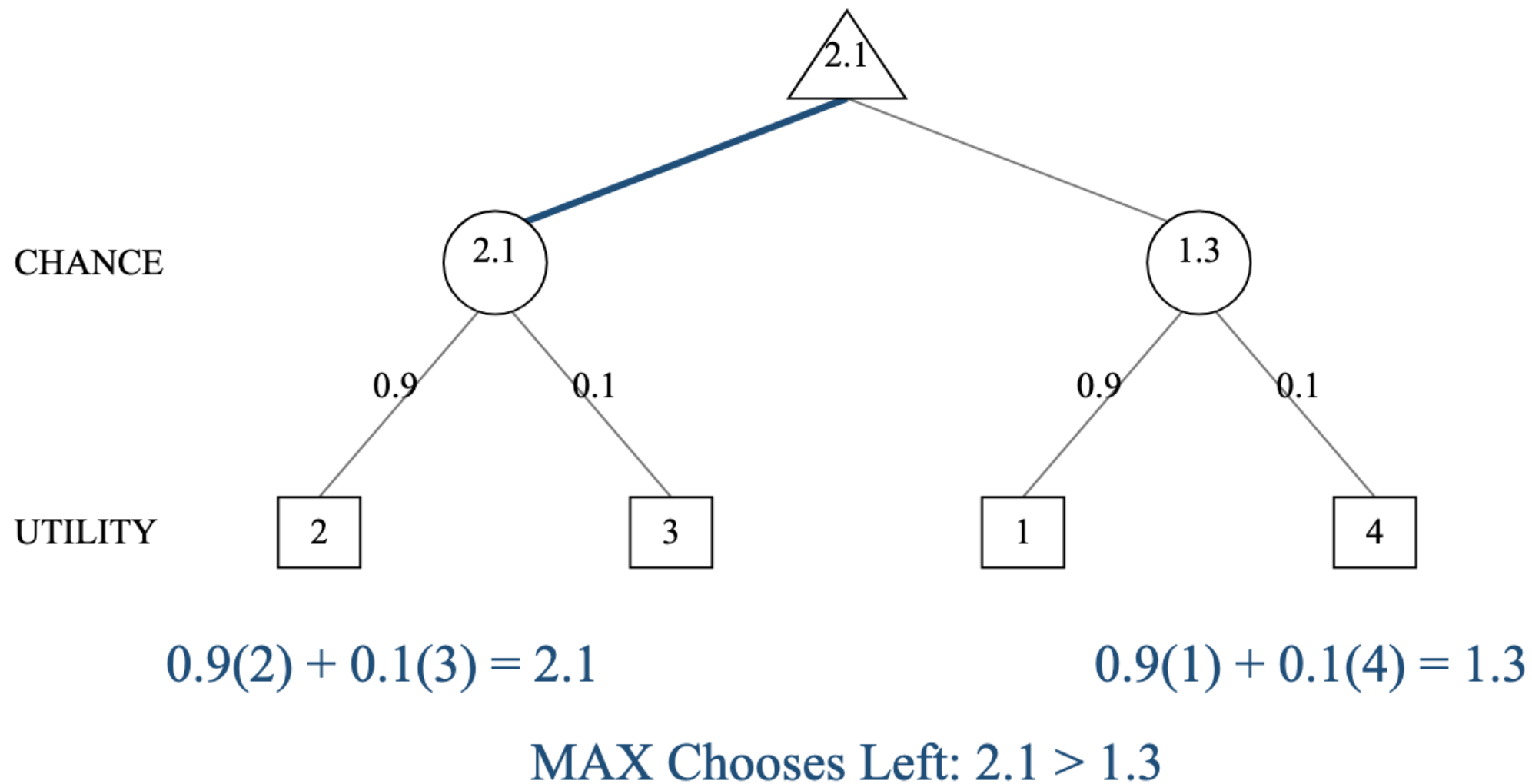
Expectimax Example



Expectimax Example

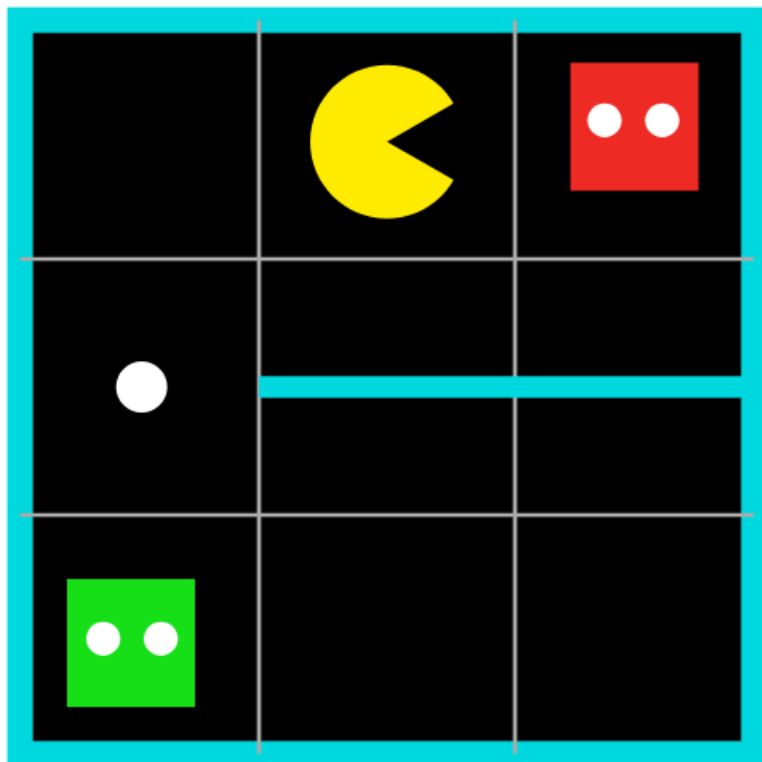


Expectimax Example



Task: Minimax or Expectimax?

TASK



Pacman First: WEST or EAST

After WEST and Both Ghosts, Pacman: EAST or SOUTH

Lower Ghost: NORTH or EAST · Upper Ghost: WEST only

Minimax: Worst Legal Ghost Move

Expectimax: Lower Ghost N/E with Probability 0.5 Each

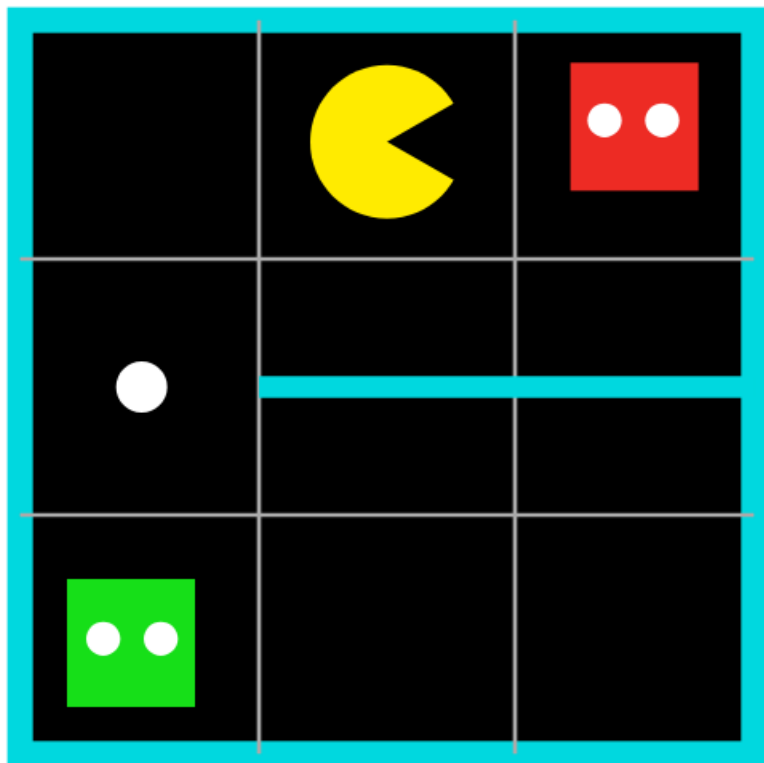
Only Food Eaten: Win · Ghost Collision: Loss

Score: -1 per Pacman action · $+10$ food · $+500$ win · -500 loss

Collision on Food: Loss, No Food Points

Task: Minimax or Expectimax?

TASK



Pacman First: WEST or EAST

After WEST and Both Ghosts, Pacman: EAST or SOUTH

Lower Ghost: NORTH or EAST · Upper Ghost: WEST only

Minimax: Worst Legal Ghost Move

Expectimax: Lower Ghost N/E with Probability 0.5 Each

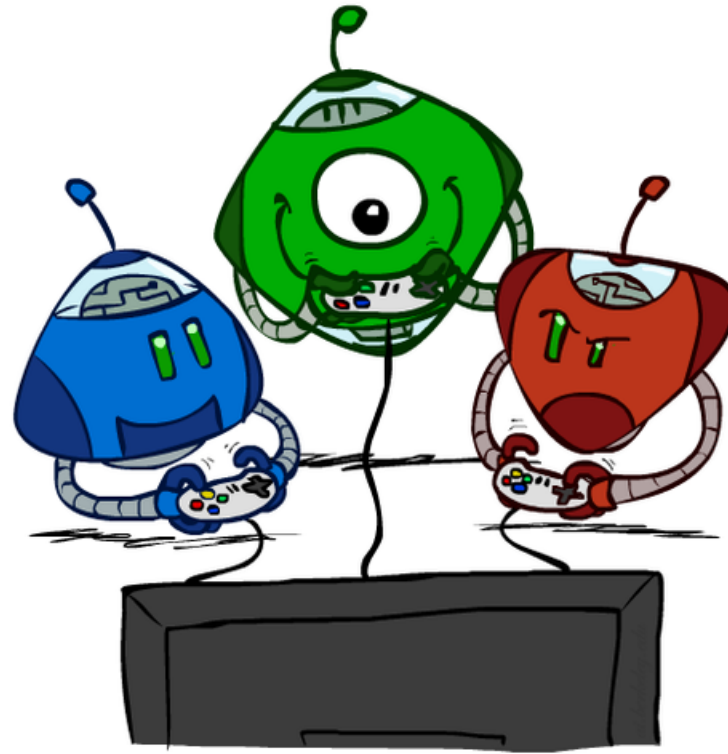
Only Food Eaten: Win · Ghost Collision: Loss

Score: -1 per Pacman action · $+10$ food · $+500$ win · -500 loss

Collision on Food: Loss, No Food Points

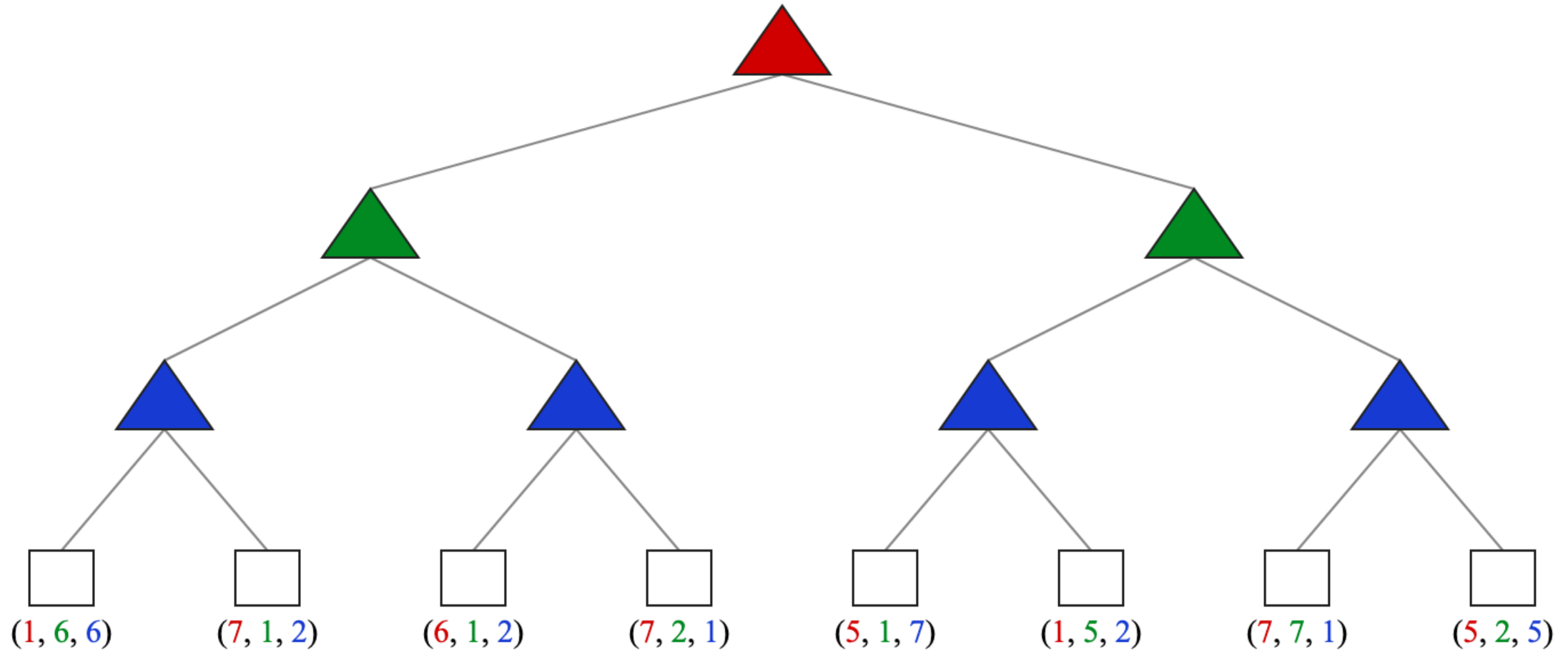
1. Under minimax, what is Pacman's optimal first action?
2. What value does minimax assign to the root?
3. Under expectimax, what is Pacman's optimal first action?
4. What value does expectimax assign to the root?

More Than Two Agents



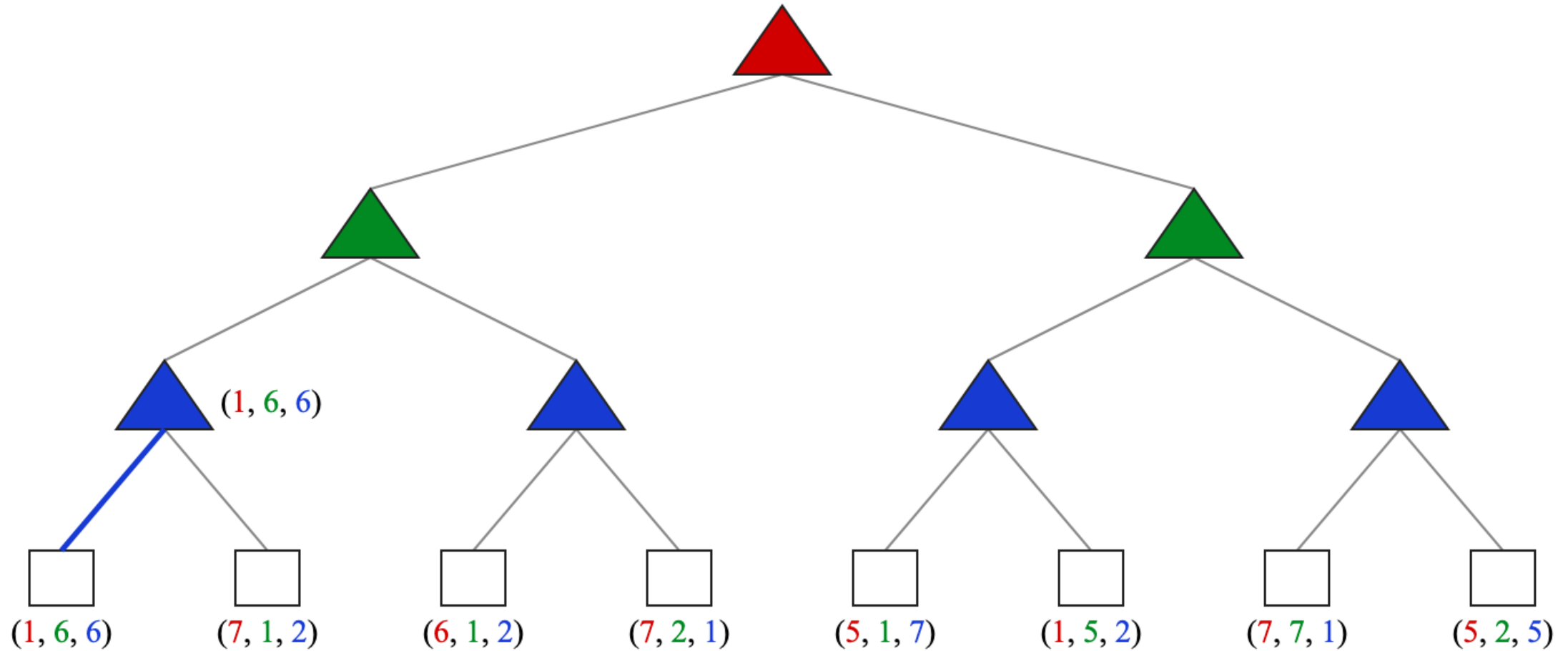
Utility Vectors

Red chooses the first value · Green the second · Blue the third



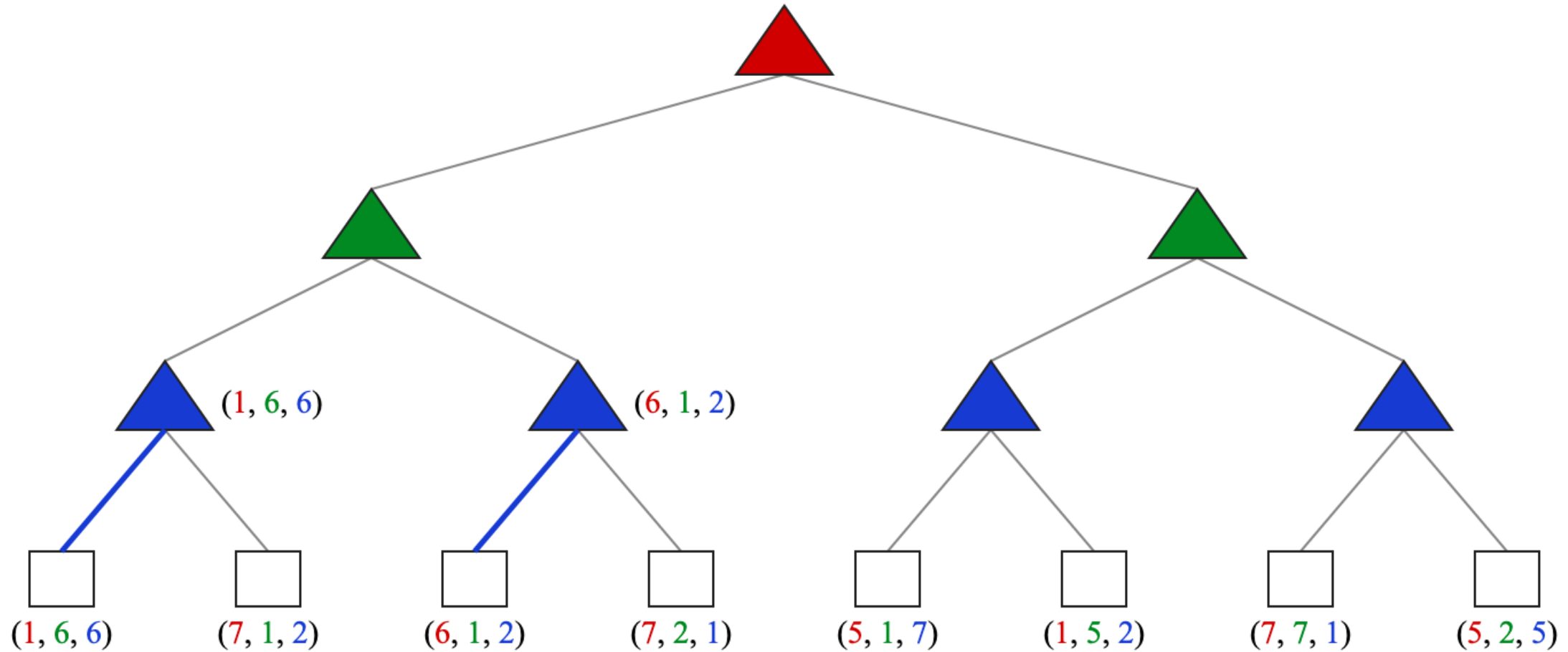
Utility Vectors

Red chooses the first value · Green the second · Blue the third



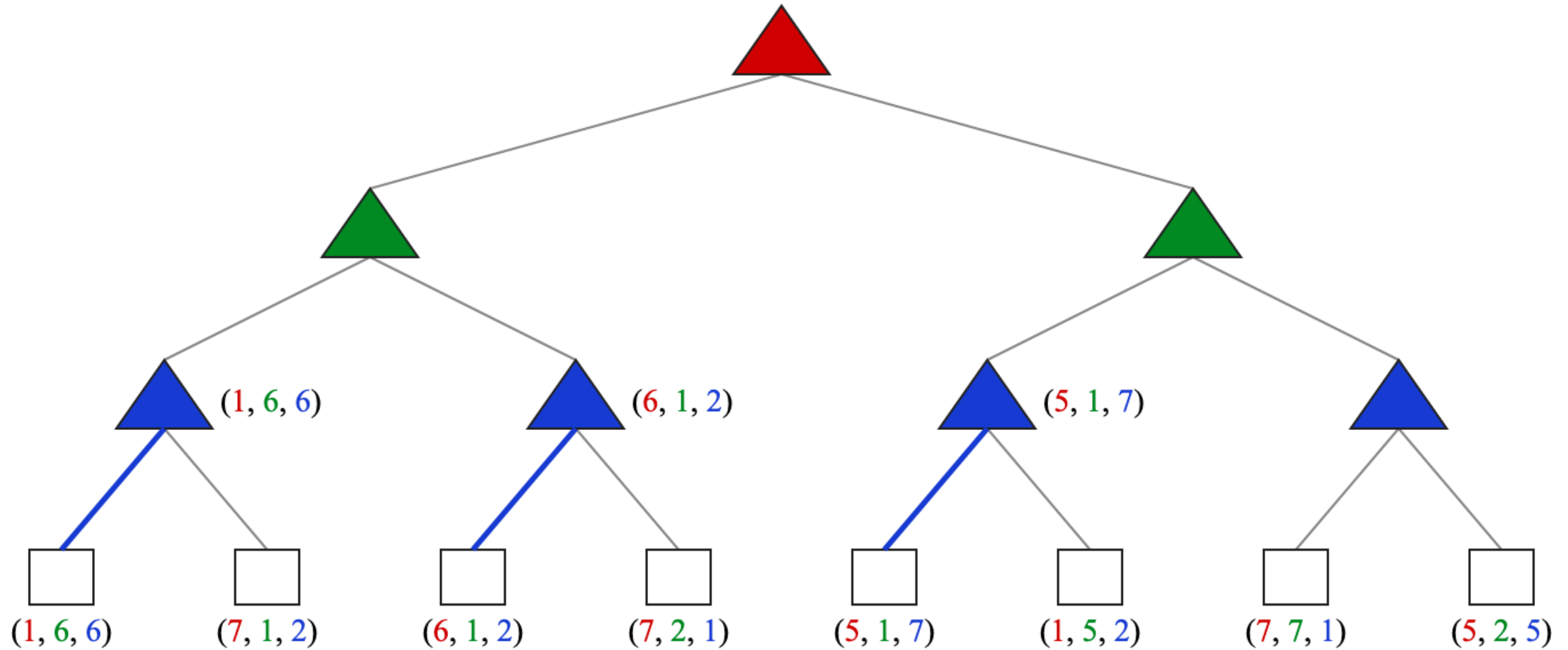
Utility Vectors

Red chooses the first value · Green the second · Blue the third



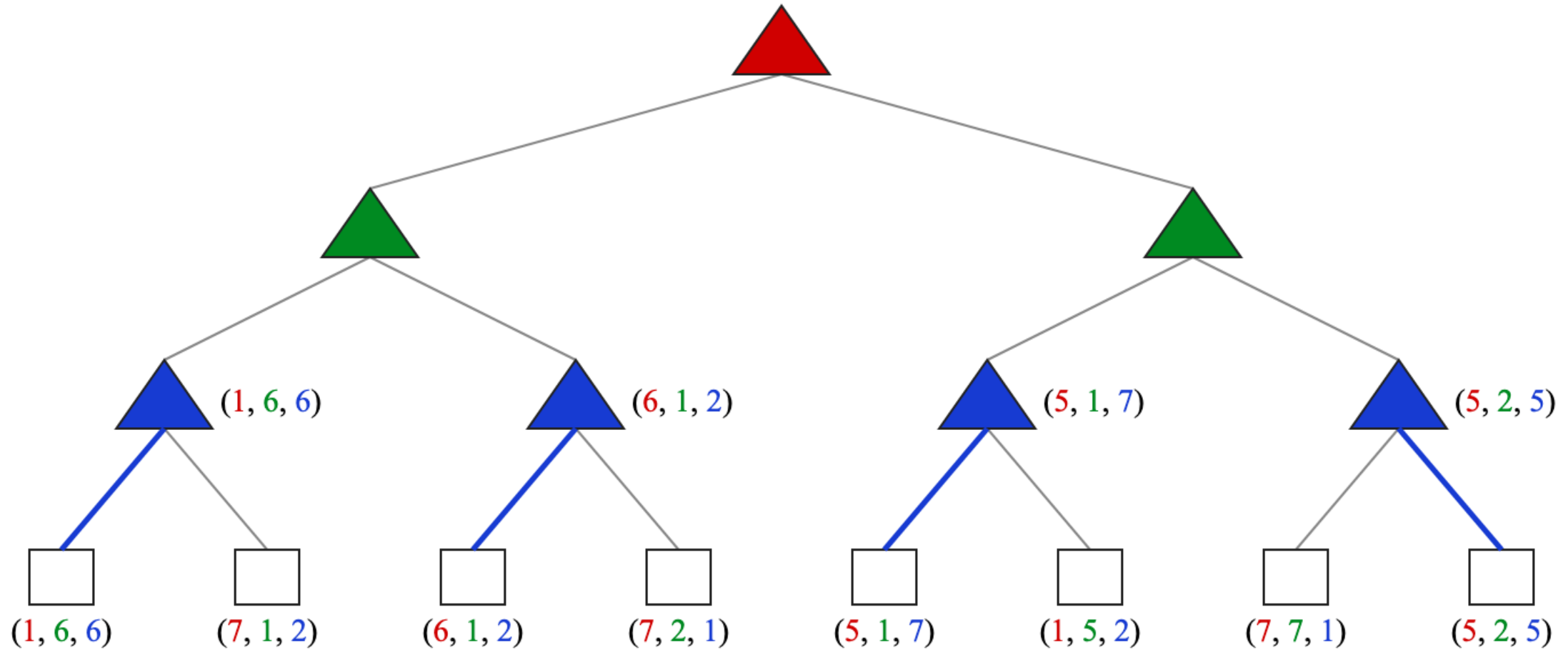
Utility Vectors

Red chooses the first value · Green the second · Blue the third



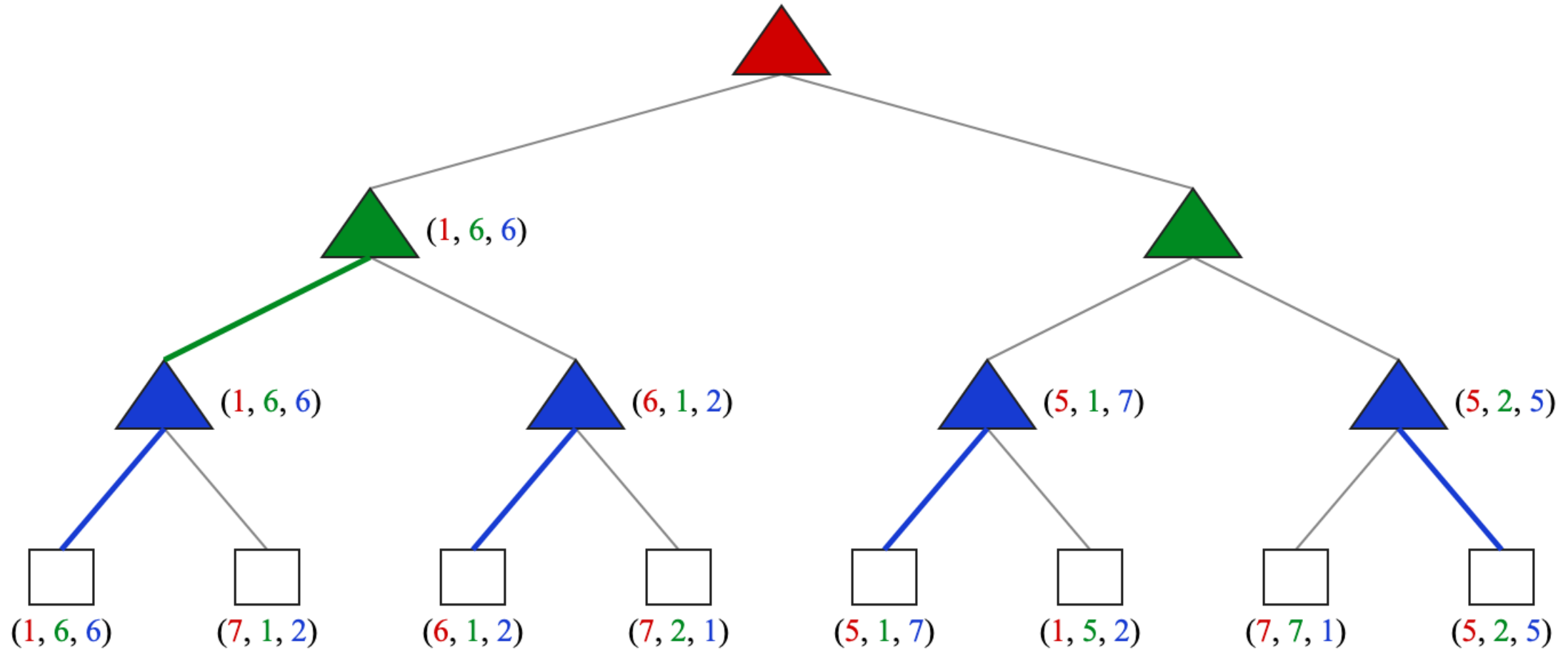
Utility Vectors

Red chooses the first value · Green the second · Blue the third



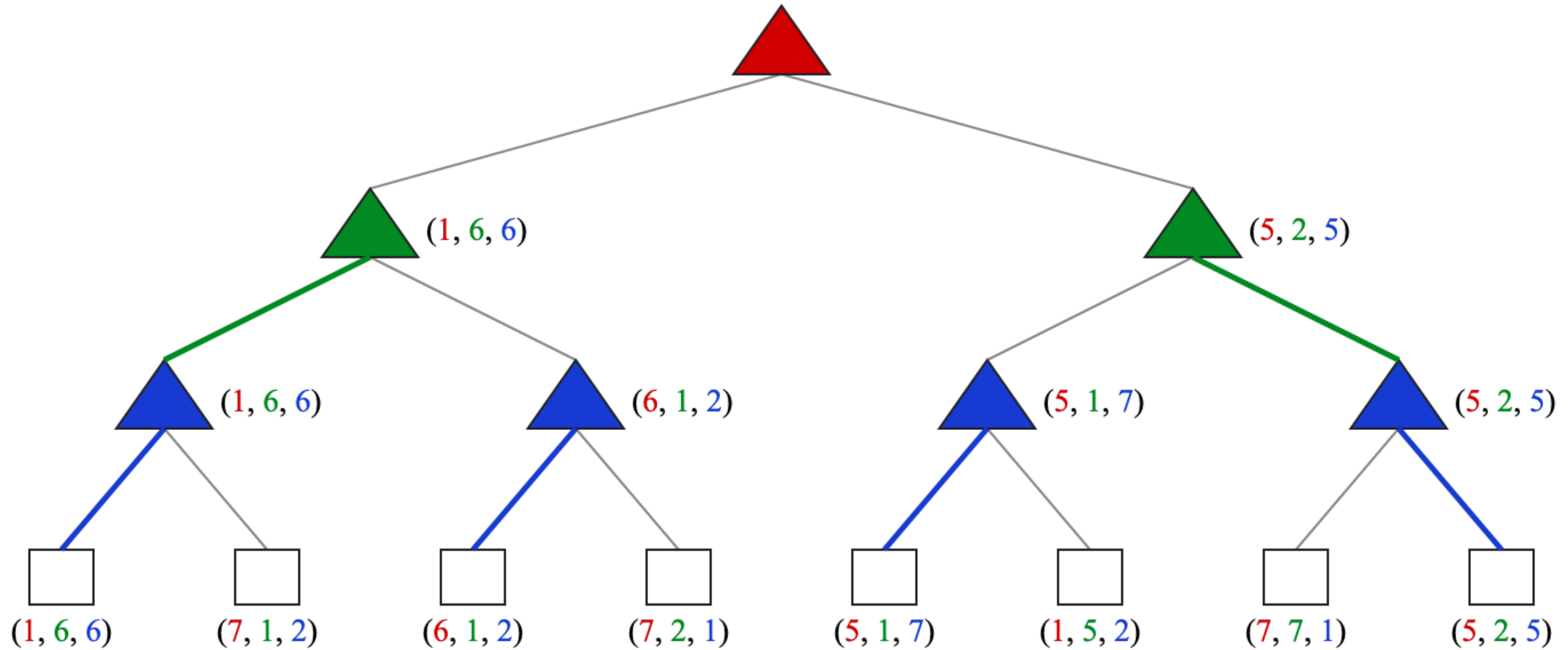
Utility Vectors

Red chooses the first value · Green the second · Blue the third



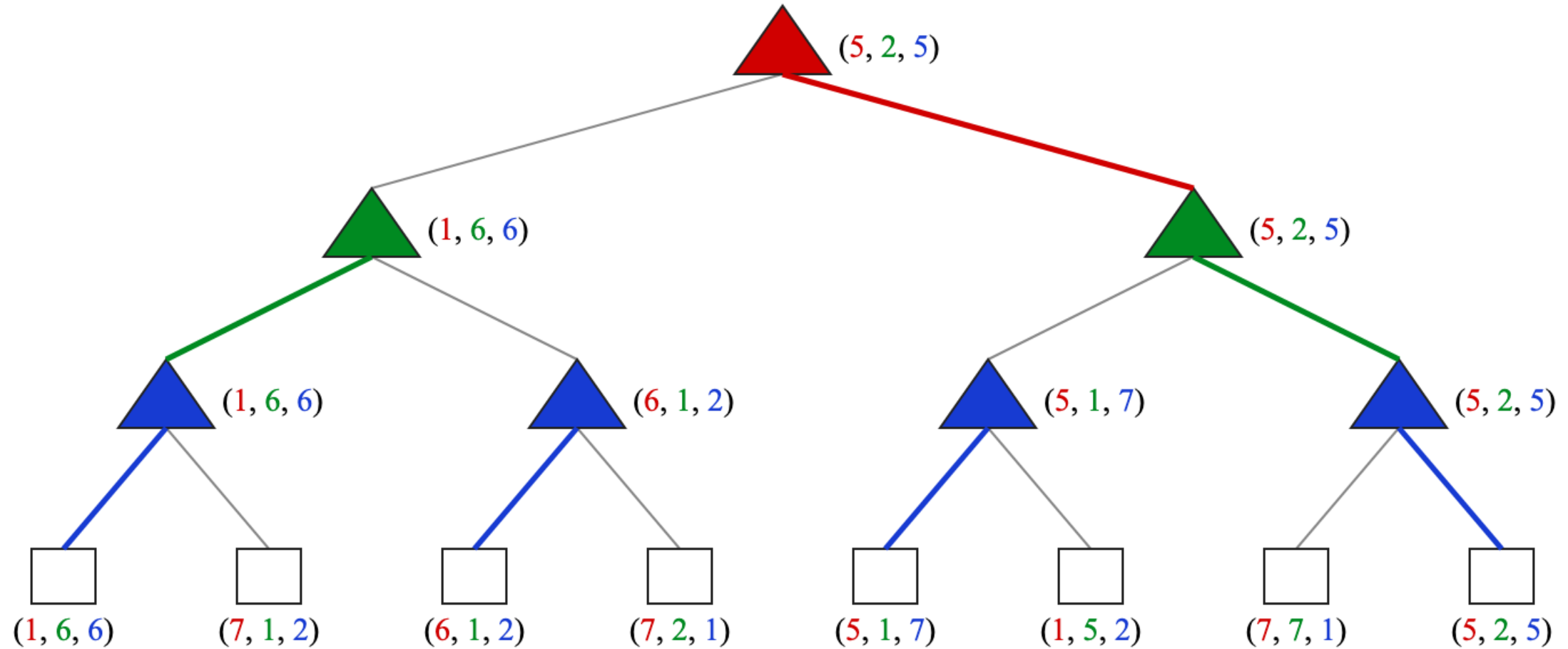
Utility Vectors

Red chooses the first value · Green the second · Blue the third



Utility Vectors

Red chooses the first value · Green the second · Blue the third



Reading

AIMA Chapter 3

Search · 88 pages

https://comp3270.dirk.hk/AIMA_Chapter_3.pdf

AIMA Chapter 4

Search in Complex Environments · 62 pages

https://comp3270.dirk.hk/AIMA_Chapter_4.pdf

AIMA Chapter 5

Adversarial Search · 63 pages

https://comp3270.dirk.hk/AIMA_Chapter_5.pdf

AIMA Chapter 6

Constraint Satisfaction · 50 pages

https://comp3270.dirk.hk/AIMA_Chapter_6.pdf

Password: Dirk