

Search

- Chapter 3 of R&N* 3rd edition is very useful reading.
- Chapter 4 of R&N 3rd edition is worth reading for enrichment. We'll touch upon some of the ideas again when we look at planning.

* R&N = Russell and Norvig, *Artificial Intelligence: a Modern Approach*

Acknowledgements

These CSC384 slides have been shared and updated by a number of people including (but not limited to):

- Sheila McIlraith
- Fahiem Bacchus
- Sonya Allin
- Craig Boutilier
- Hojjat Ghaderi
- Rich Zemel

In turn, some of these people have drawn materials from

- Russell & Norvig
- Andrew Moore
- Shaul Markovitch

If you see your slides in this deck, let me know! I'd like to acknowledge you.
Thanks to all for sharing!
- Sheila

What is Search

- One of the most fundamental techniques in AI
 - Underlying sub-module in many AI systems
- Solves many problems that humans are not good at.
- Can achieving super-human performance on other problems (Chess, Go)
- Very useful as a general algorithmic technique for solving problems (both in AI and in other areas)

Example: Holiday Planning

We must take into account various preferences and constraints to develop a plan.

An important technique in developing such a plan is “hypothetical” reasoning.

Example: a holiday in British Columbia

- e.g., if I fly into Vancouver at 8pm and drive a car to Whistler, I’ll have to drive on the roads at night. How desirable is this?
- If I’m in Whistler and leave at 7:30am, I can arrive in Kamloops in time for lunch.

How do we plan our holiday?

This kind of hypothetical reasoning involves asking

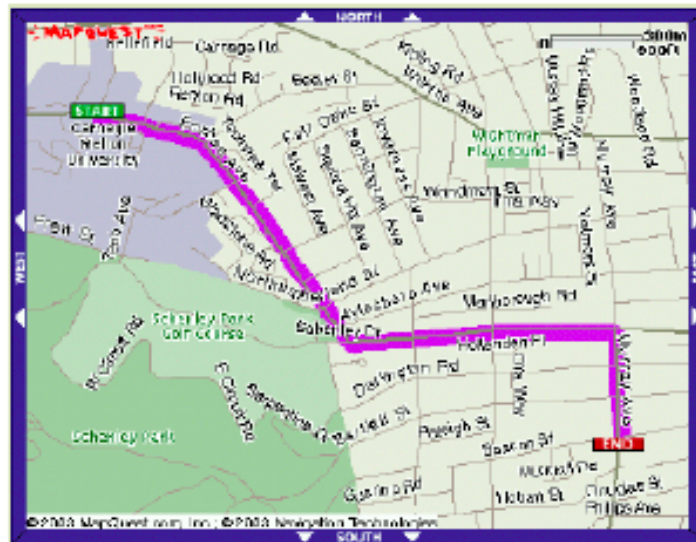
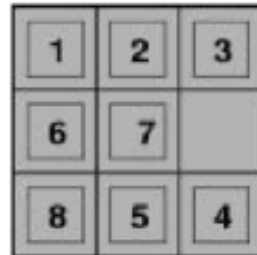
“what state will I be in after the following sequence of events?”

From this we can reason about what sequence of events one should try to bring about to achieve a desirable state.

Search is a computational method for capturing a particular version of this kind of reasoning.

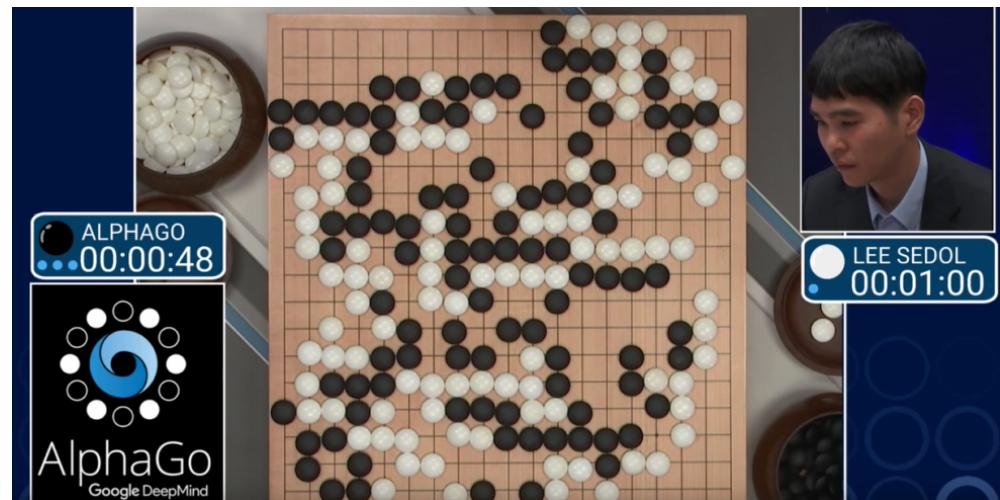
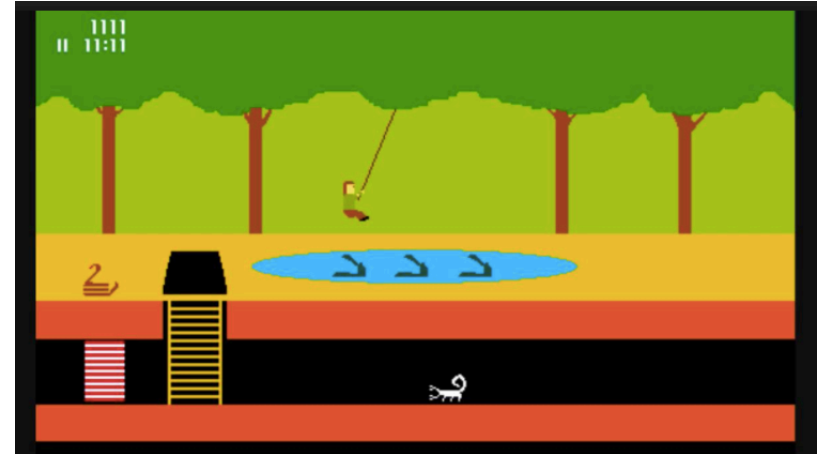
Diversity of Applications

Search Problems



Diversity of Applications

Search Problems



Why Search

Successful

- Many AI problems can be successfully solved by search
- Many AI search programs outperform humans (e.g. games)

Practical

- Many problems don't have a simple algorithmic solution. Casting these problems as search problems is often the easiest way of solving them.
- Search can also be useful in approximation (e.g., local search in optimization problems).
- Problem-specific knowledge can be exploited via heuristics.

Some critical aspects of intelligent behaviour, e.g., planning, can be naturally cast as search.

Challenge

Problem → Search Problem

Search only shows how to solve the problem once we have it correctly formulated.

The Formalism

To formulate a problem as a search problem we need the following components:

1. a **state space** over which to search. The state space necessarily involves **abstracting** the real problem.
2. **actions (successor functions, state space transitions)** that allow one to move between different states. The actions may be abstractions of actions you could actually perform.
3. the **initial state** that best represents the current state .
4. the **goal** or **desired condition** one wants to achieve.
5. (optionally) **costs** that are associated with actions.
6. (optionally) **heuristics** to help guide the search process.



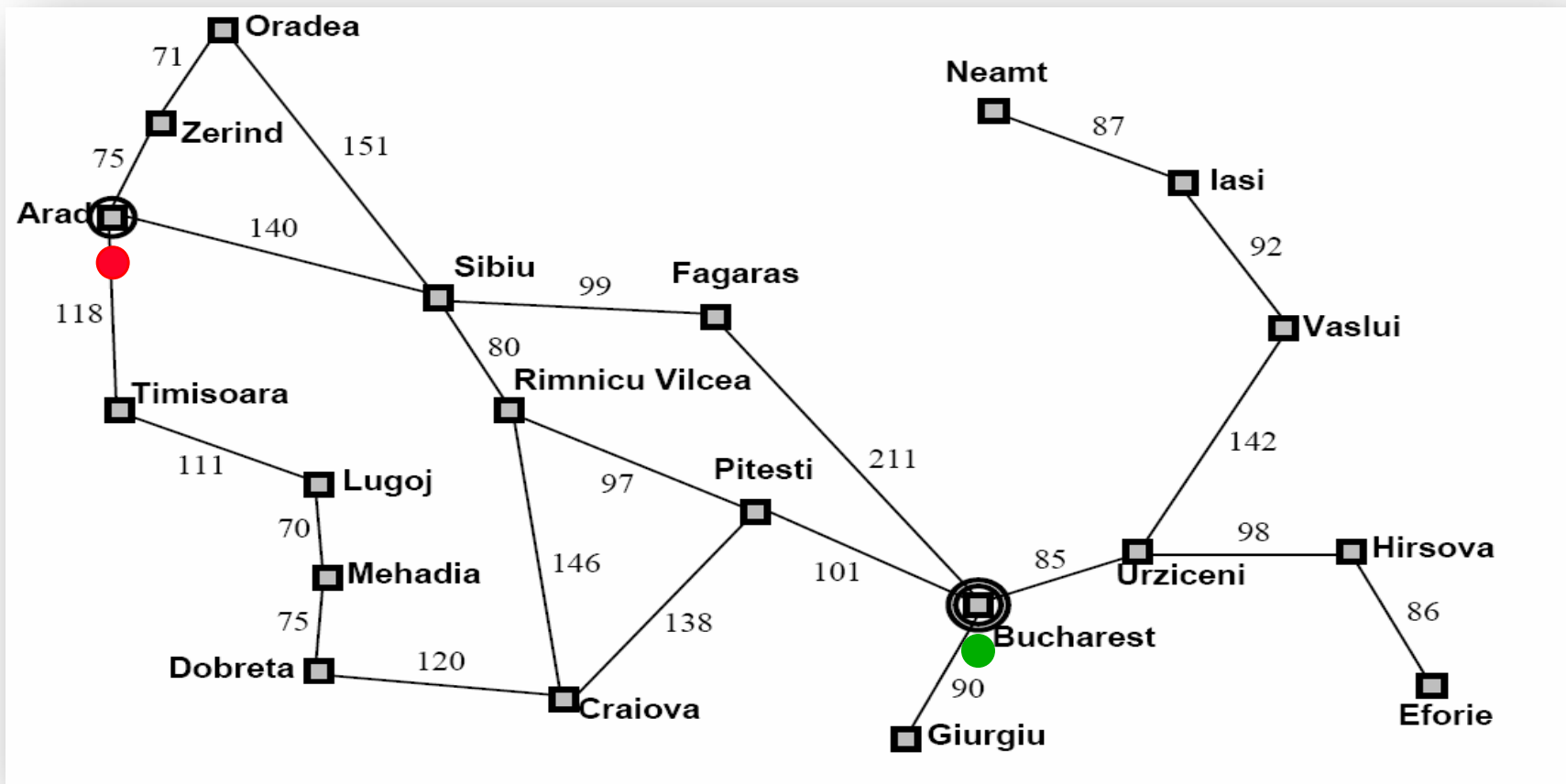
Utilizing the formalism to search

Once the problem has been formulated as a state space search, various algorithms can be utilized to solve the problem.

A **solution** to the problem will be a **sequence of actions/moves** that can transform your current state into state where your desired (goal) condition holds.

Example 1: Romania Travel.

Currently in **Arad**, need to get to **Bucharest** by tomorrow to catch a flight. What is the **State Space**? 🗨️



Example 1

State Space

States:

the cities you could be located in.

Actions:

drive between neighboring cities.

Initial state:

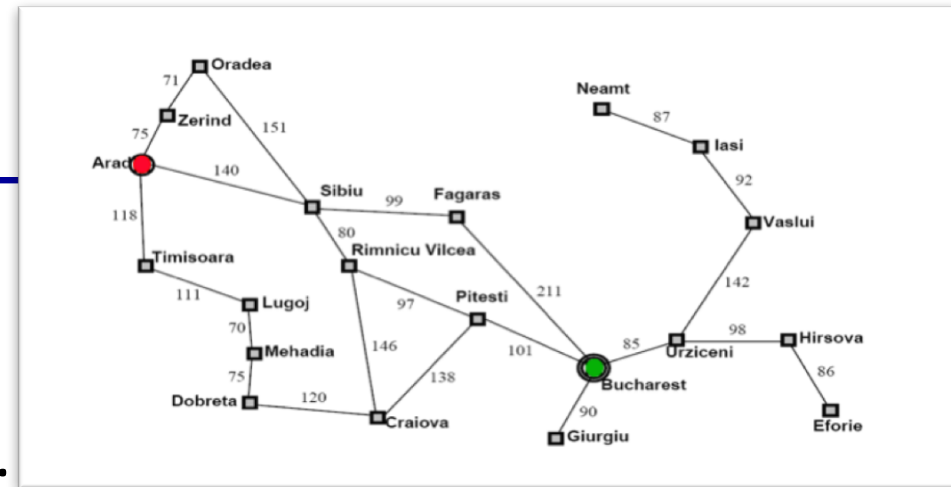
in Arad

Desired condition (Goal):

be in a state where you are in Bucharest. (How many states satisfy this condition?)

Solution

route, the sequence of cities to travel through to reach Bucharest.



Example 2. The 8-Puzzle

7	2	4
5		6
8	3	1

Start State

1	2	3
4	5	6
7	8	

Goal State

Rule: Can slide a tile into the blank spot.

(Equivalently, can think of it as moving the blank around).

Example 2. The 8-Puzzle

State space

- **States**: the different configurations of the tiles.
How many different states?
- **Actions (successor function)**: moving the blank up, down, left, right. Can every action be performed in every state?
- **Initial state**: as shown on previous slide.
- **Desired condition (Goal)**: a state where the tiles are all in the positions shown on the previous slide.

Solution will be a sequence of moves of the blank that transform the initial state to a goal state.

Example Search Space

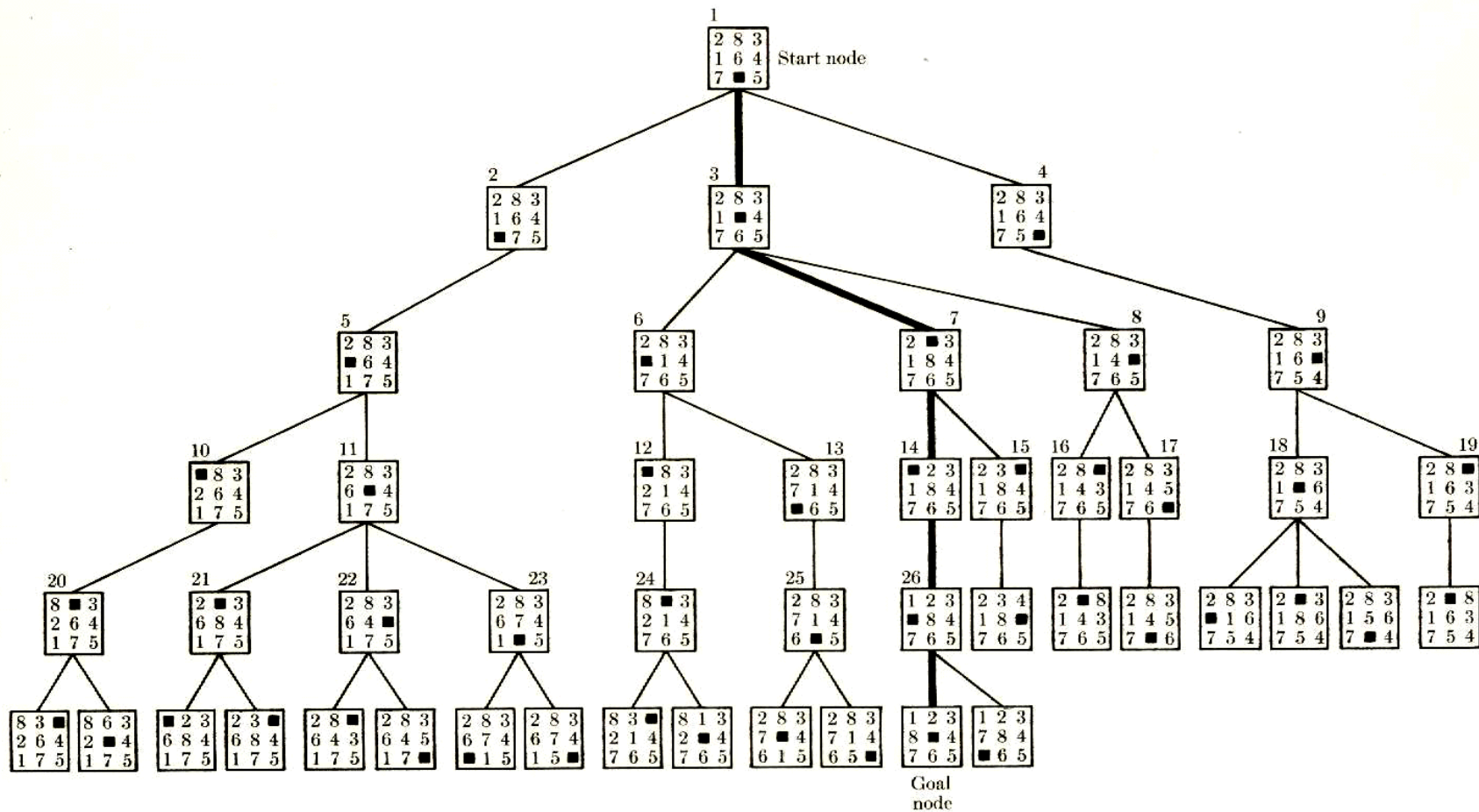


FIG. 3-2 The tree produced by a breadth-first search.

Example 2. The 8-Puzzle

Although there are $9!$ different configurations of the tiles (362,880) in fact the state space is divided into two disjoint parts.

Only when the blank is in the middle are all four actions possible.

Our goal condition is satisfied by only a single state. But one could easily have a goal condition like

- The 8 is in the upper left hand corner.
- How many different states satisfy this goal?

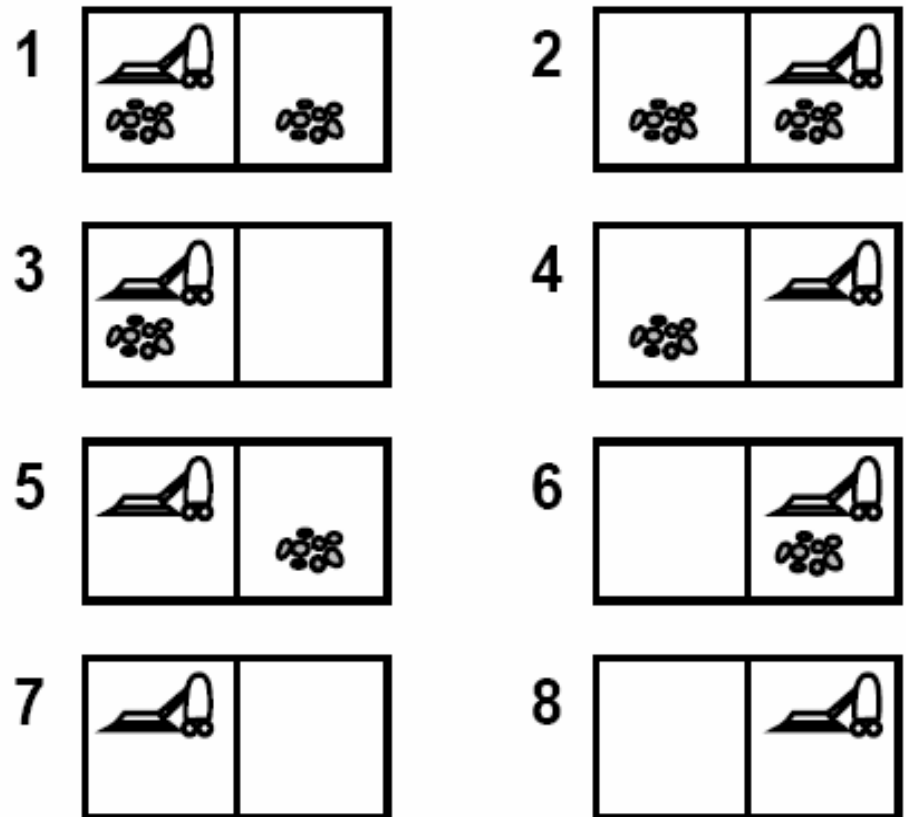
Example 3. Vacuum World

In the previous two examples, a state in the search space corresponded to a unique state of the world (modulo details we have abstracted away).

However, states need not map directly to world configurations. Instead, a state could map to the agent's **mental** conception of how the world is configured: **the agent's knowledge state.**

Example 3. Vacuum World

- We have a vacuum cleaner and two rooms.
- Each room may or may not be dirty.
- The vacuum cleaner can move **left** or **right** (*the action has no effect if there is no room to the right/left*).
- The vacuum cleaner can **suck**; this cleans the room (*even if the room was already clean*).

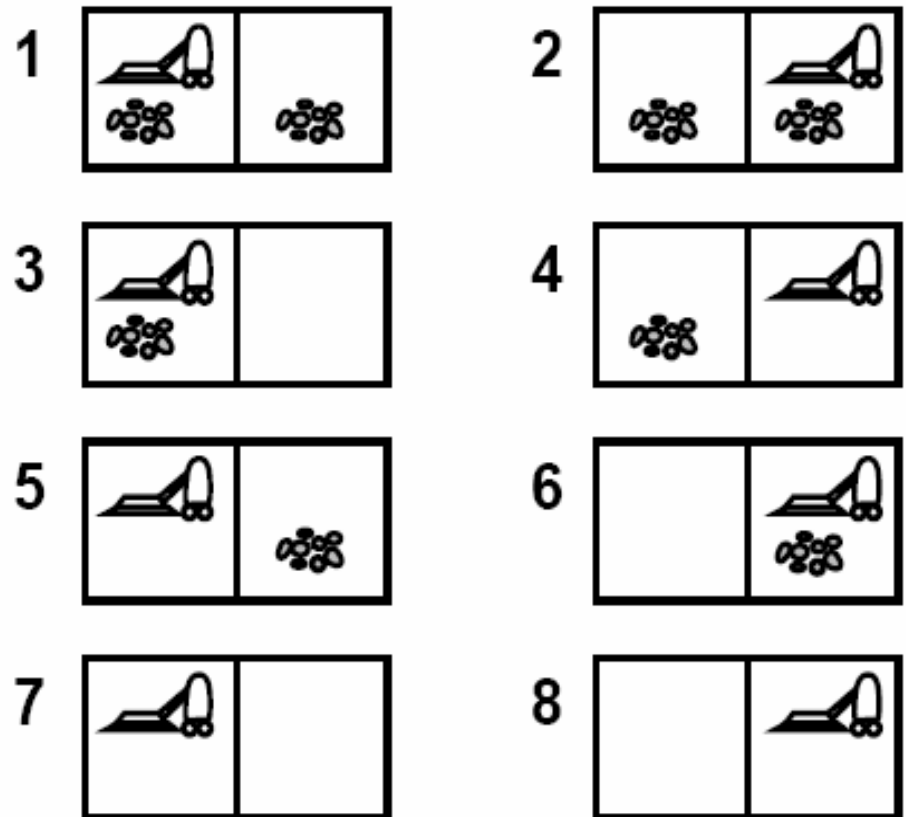


Physical states

Example 3. Vacuum World

Knowledge-level State Space

- The state space can consist of a set of states. The agent knows that it is in one of these states, but doesn't know which.

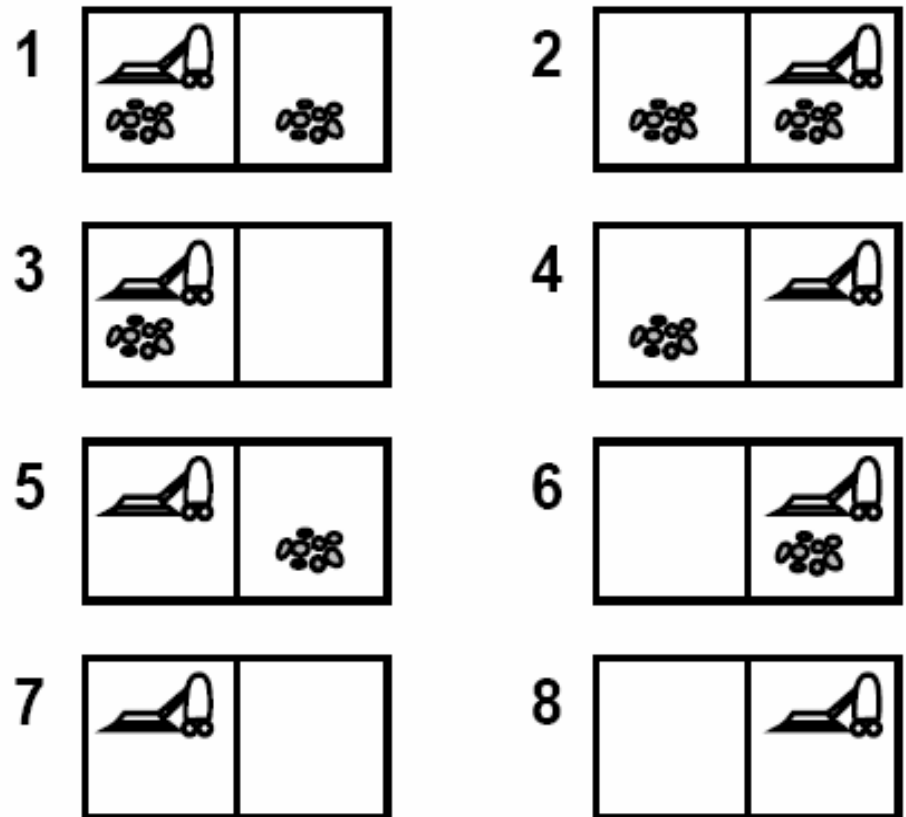


Goal is to have all rooms clean.

Example 3. Vacuum World

Knowledge-level State Space

- Complete knowledge of the world: agent knows exactly which state it is in. State space states consist of single physical states:
- Start in {5}:
<right, suck>

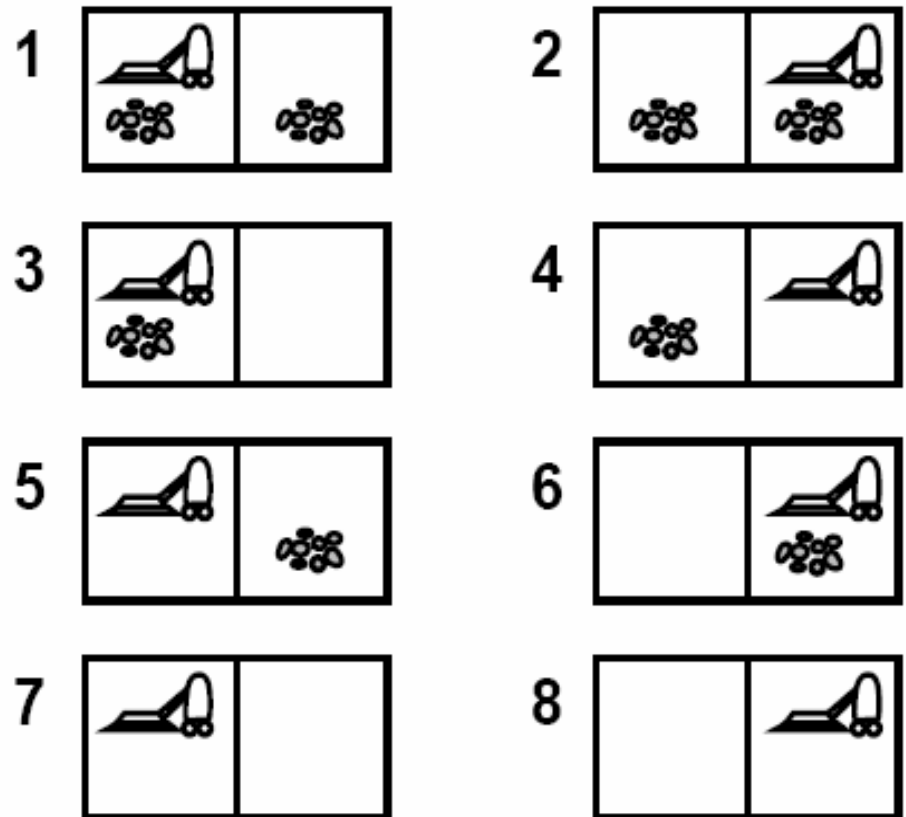


Goal is to have all rooms clean.

Example 3. Vacuum World

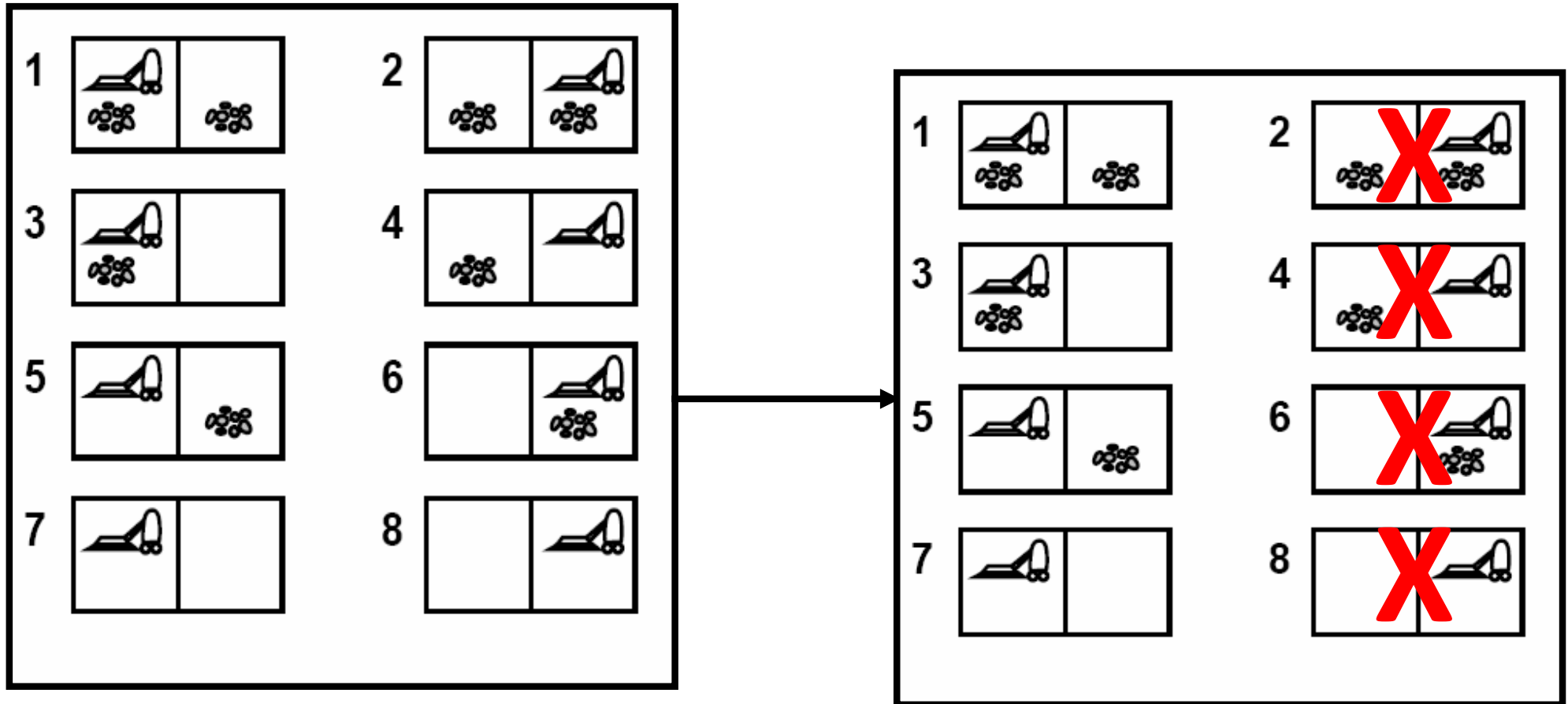
Knowledge-level State Space

- **No knowledge of the world.**
States consist of sets of physical states.
- **Start in {1,2,3,4,5,6,7,8},**
agent doesn't have any knowledge of where it is.
- Nevertheless, the actions **<left, suck, right, suck>** achieves the goal.



Goal is to have all rooms clean.

Example 3. Vacuum World

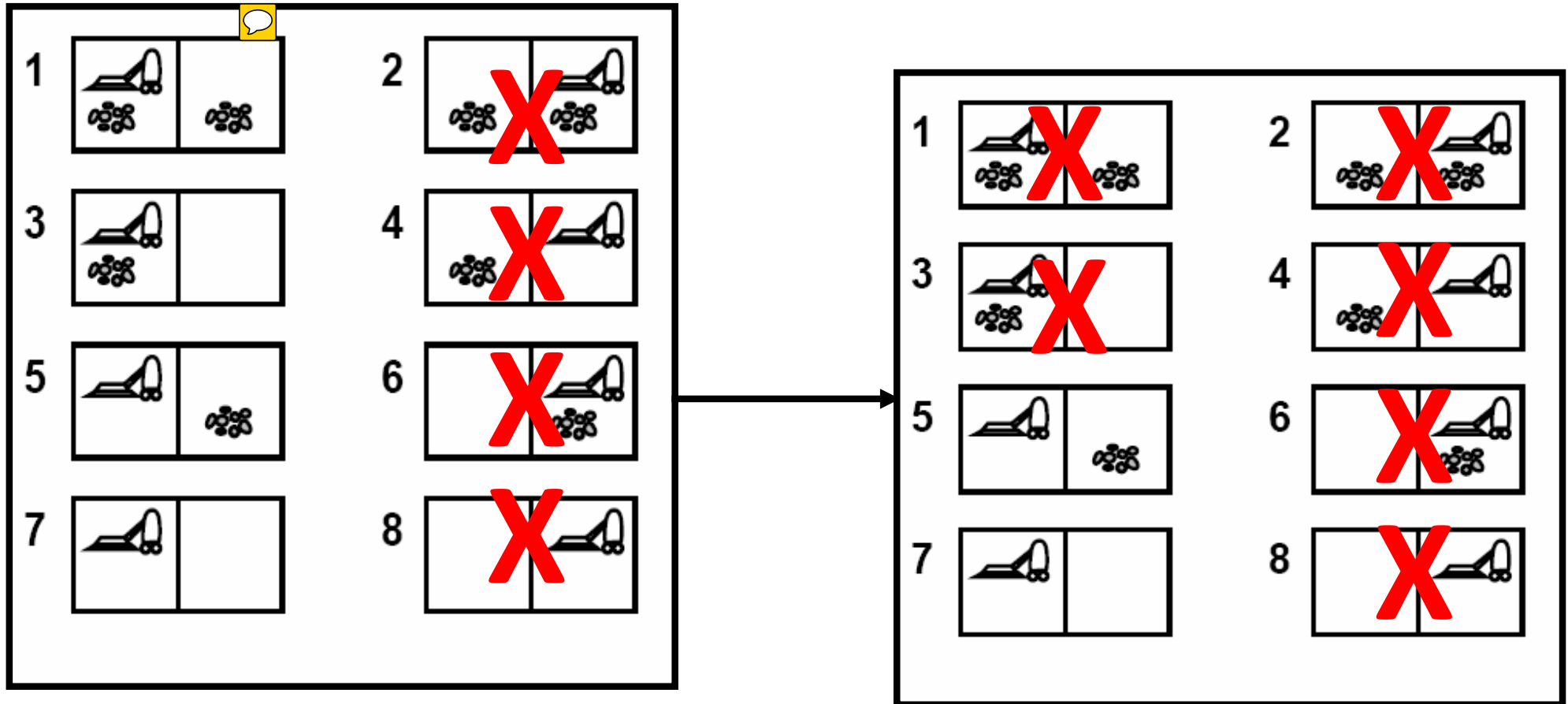


Initial state.

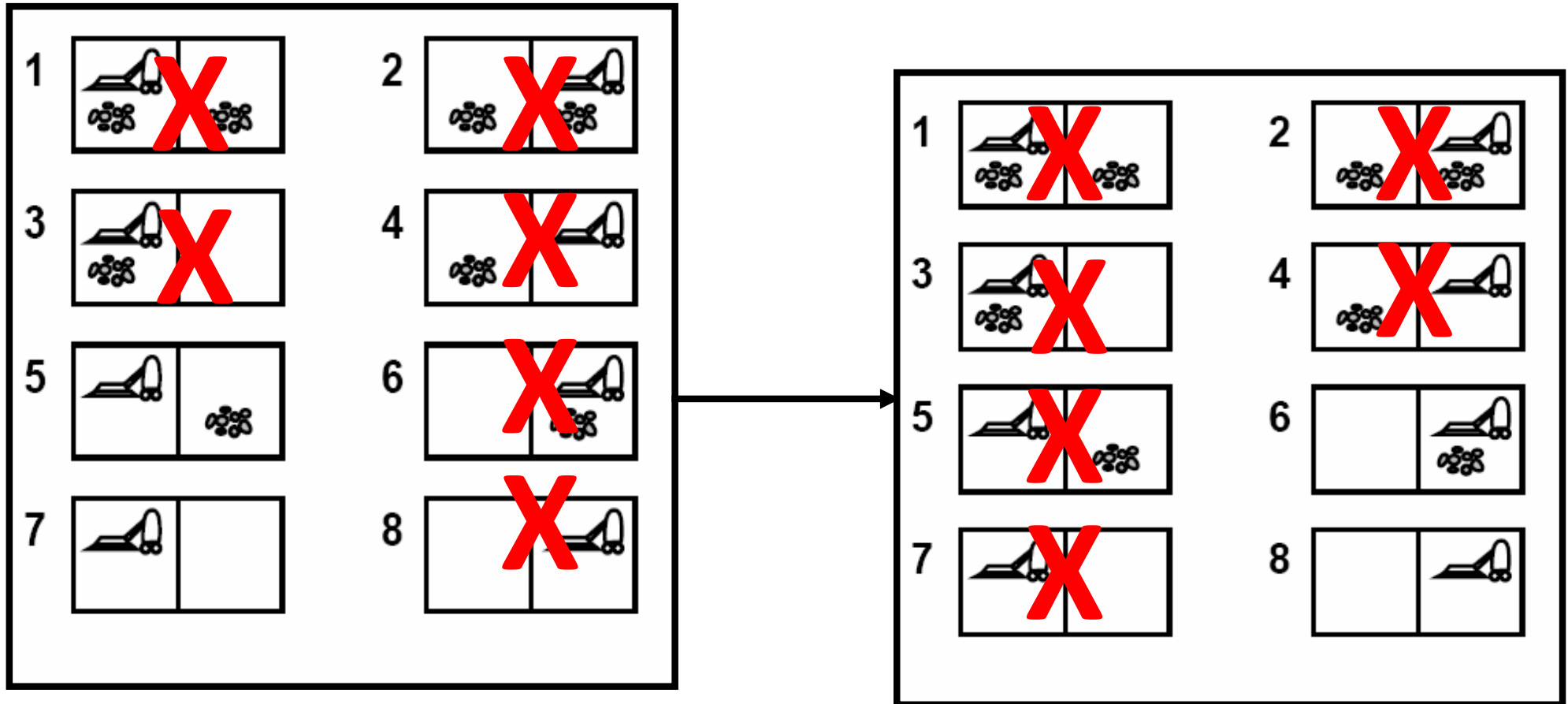
{1,2,3,4,5,6,7,8}

Left

Example 3. Vacuum World

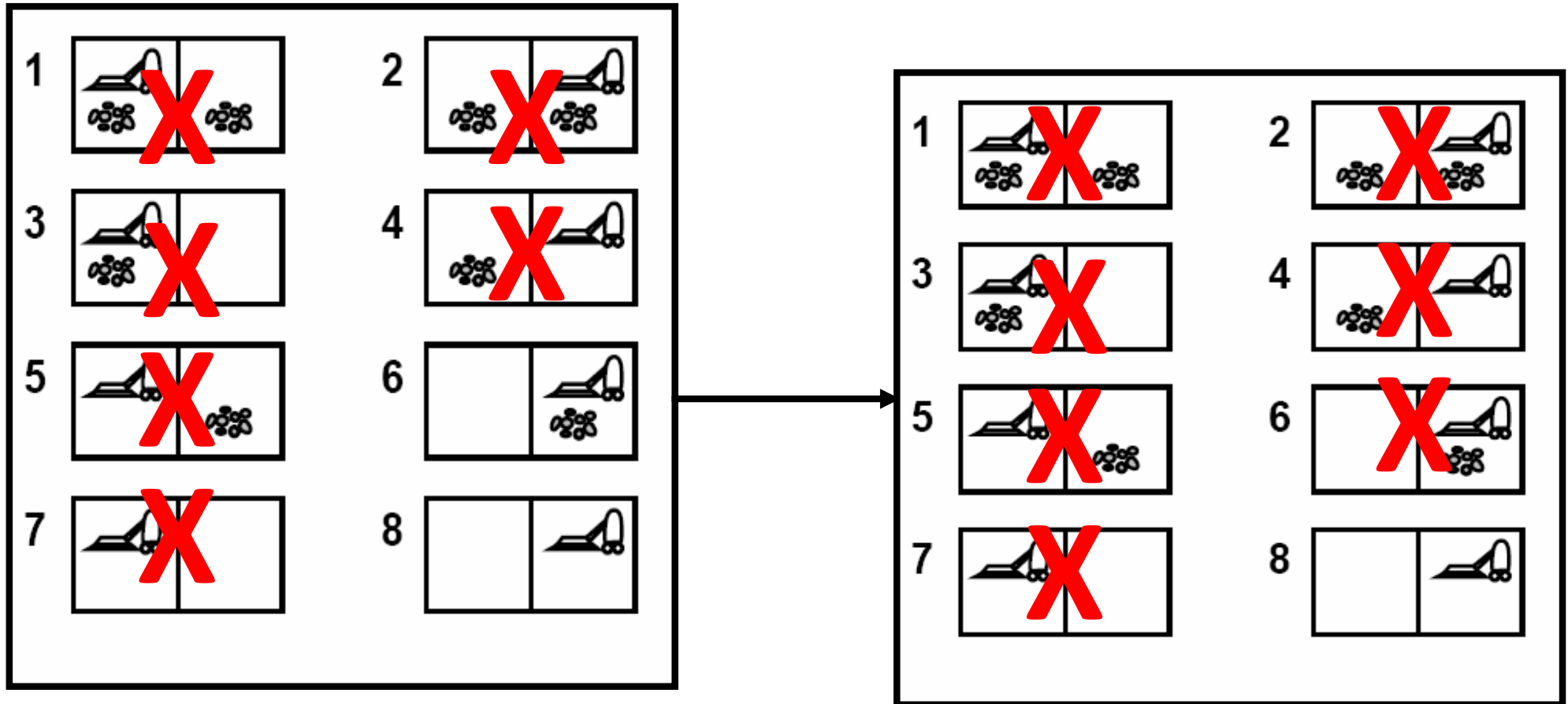


Example 3. Vacuum World



Right

Example 3. Vacuum World



Suck

More complex situations

The agent might be able to perform some sensing actions. These actions change the agent's mental state, not the world configuration.



With sensing can search for a **contingent** solution: a solution that is contingent on the outcome of the sensing actions

- **<right, if dirt then suck>**

Now the issue of interleaving execution and search comes into play.

More complex situations

Instead of complete lack of knowledge, the agent might think that some states of the world are more **likely** than others.

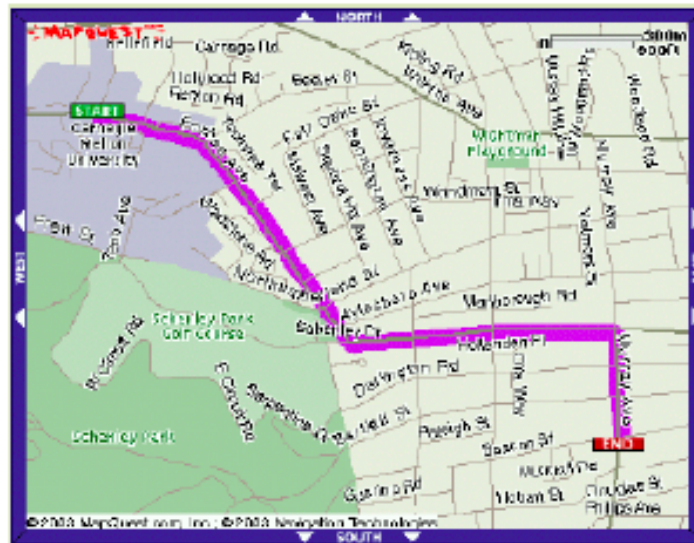
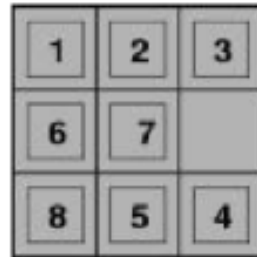
This leads to **probabilistic models** of the search space and different algorithms for solving the problem.

Later we will see some techniques for reasoning and making decisions under uncertainty.

We can also consider the outcome of an action to be nondeterministic, perhaps associating a probability with each outcome.

Diversity of Applications

Search Problems



ALGORITHMS FOR SEARCH

Algorithms for Search

Inputs:

- a specified **initial state** (a specific world state or a set of world states representing the agent's knowledge, etc.)
- a **successor** function $S(x) = \{\text{set of states that can be reached from state } x \text{ via a single action}\}$.
- a **goal test** a function that can be applied to a state and returns true if the state satisfies the goal condition.
- (optionally) a **step cost** function $C(x,a,y)$ which determines the cost of moving from state x to state y using action a . ($C(x,a,y) = \infty$ if a does not yield y from x). By default this is "1" for all actions (uniform cost).

Algorithms for Search

Output:

- a sequence of states leading from the initial state to a state satisfying the goal test.
- The sequence might be
 - annotated by the name of the action used.
 - optimal in cost for some algorithms.

Algorithms for Search

- To explore the state space during a search, we will iteratively apply the successor function to the states we discover.
- Each time, the successor function $S(x)$ yields a set of states that can be reached from x via any single action.
- Rather than just return a set of states, we might annotate these states by the action used to obtain them:
 - $S(x) = \{ \langle y, a \rangle, \langle z, b \rangle \}$
arrive at y via action a , arrive at z via action b .
 - $S(x) = \{ \langle y, a \rangle, \langle y, b \rangle \}$
arrive at y via action a , also y via alternative action b .
- It may also be important to reference the state of origin (i.e., the preceding state).

Tree search

- We put nodes (or states) we haven't explored or "expanded", but wish to consider, in a list called the Frontier (or Open list)
- Initially, the Frontier contains the initial state
- At each search iteration, we pop a node from the Frontier, apply Successors(x), and insert those children into the Frontier

```
TreeSearch(Frontier, Successors, Goal? )
```

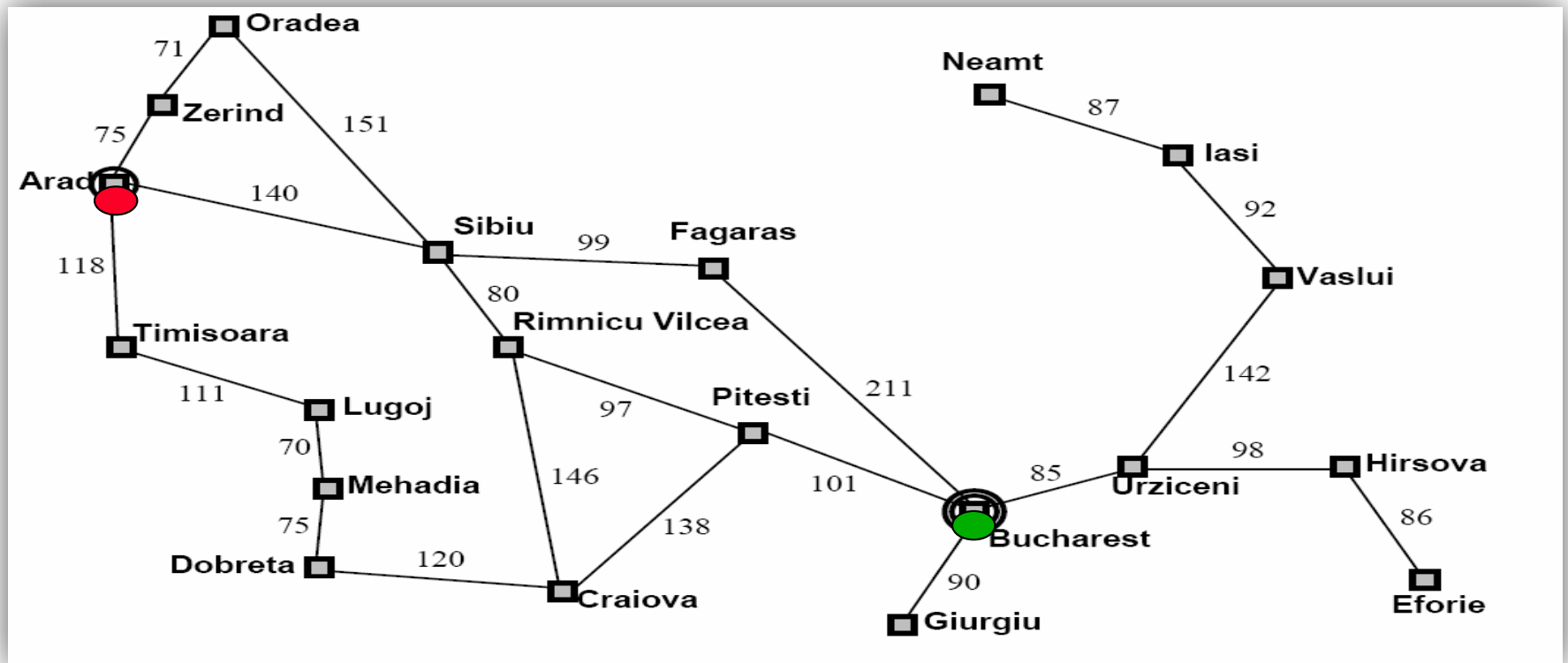
```
  If Frontier is empty return failure
```

```
  Curr = select state from Frontier
```

```
  If (Goal?(Curr)) return Curr.
```

```
  Frontier' = (Frontier - {Curr}) U Successors(Curr)
```

```
  return TreeSearch(Frontier', Successors, Goal?)
```



{Arad},

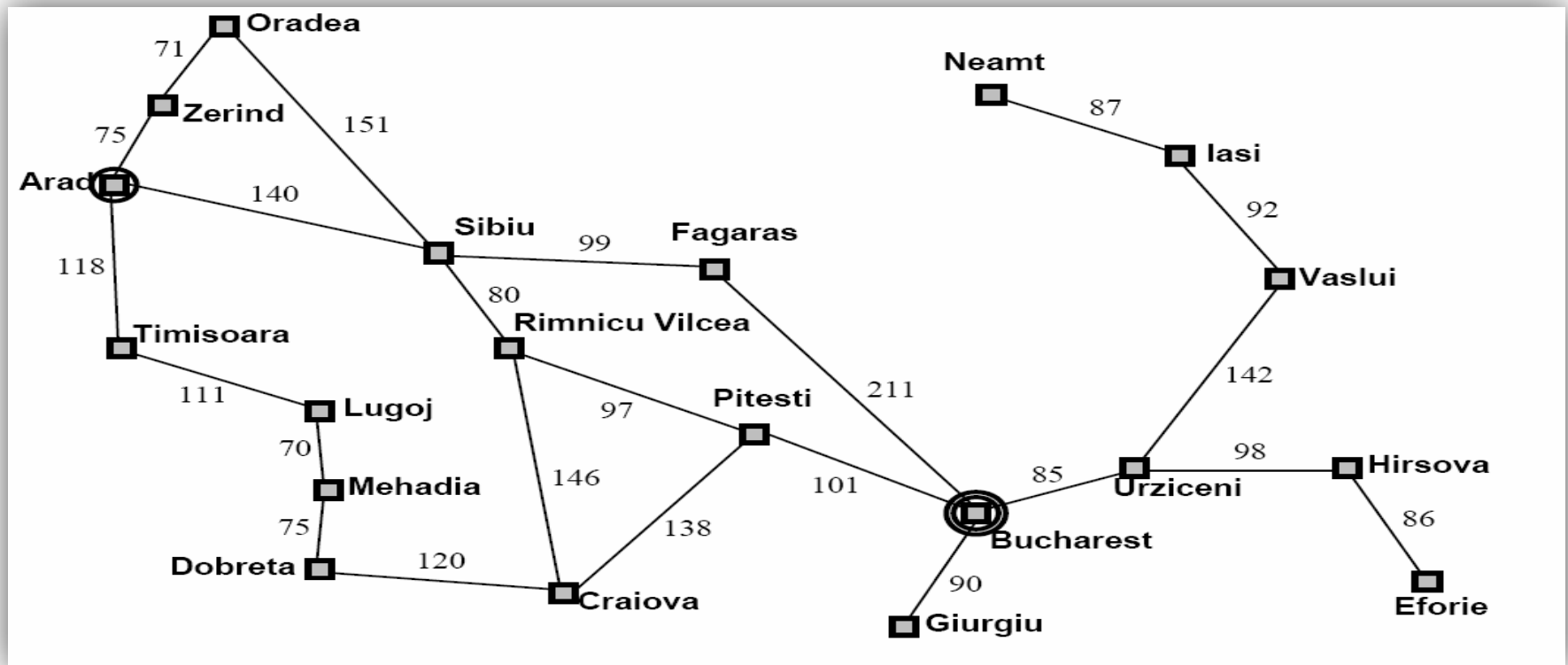
{Z<A>, T<A>, S<A>},

{Z<A>, T<A>, A<S;A>, O<S;A>, F<S;A>, R<S;A>}

{Z<A>, T<A>, A<S;A>, O<S;A>, R<S;A>, S<F;S;A>, B<F;S;A>}

Solution: Arad -> Sibiu -> Fagaras -> Bucharest

Cost: 140 + 99 + 211 = 450



{Arad}

{Z<A>, T<A>, S<A>},

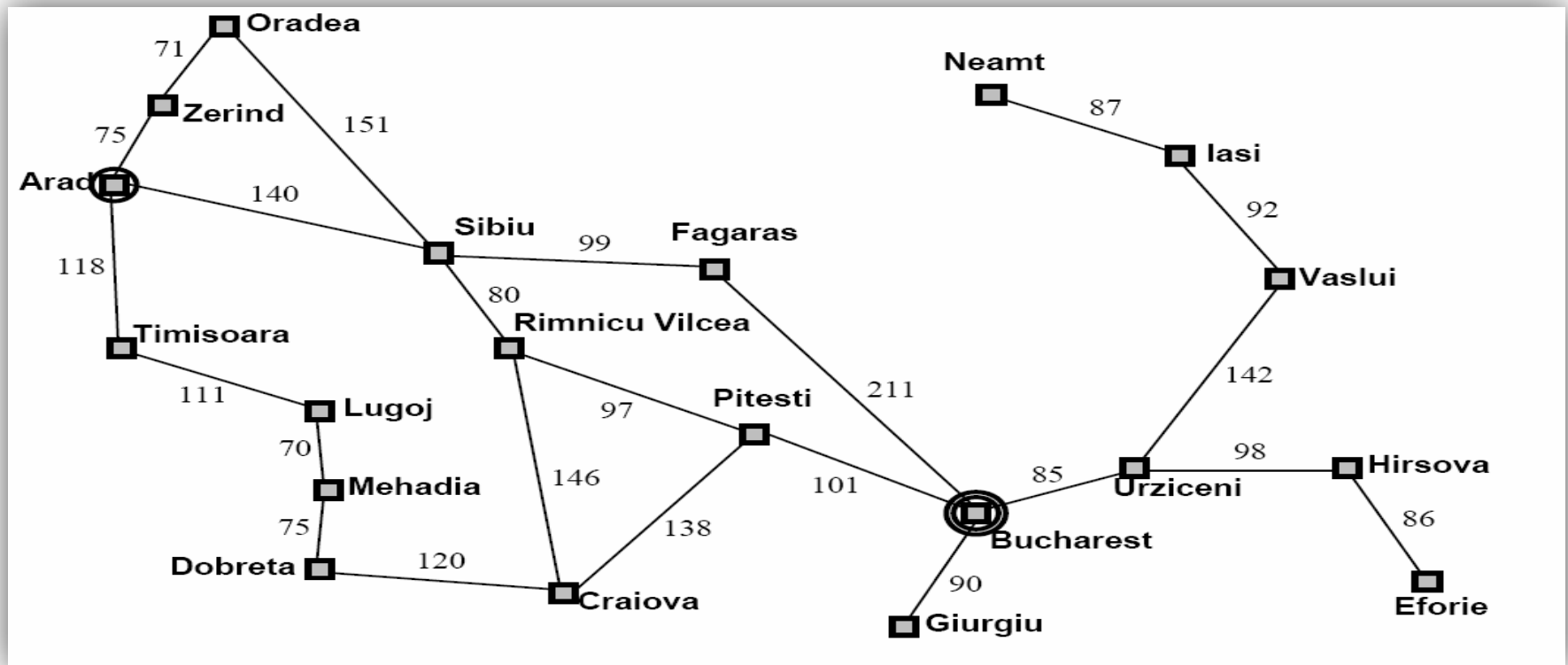
{Z<A>, T<A>, A<S,A>, O<S,A>, F<S,A>, R<S,A>}

{Z<A>, T<A>, A<S,A>, O<S,A>, F<S,A>, S<R,S,A>, P<R,S,A>, C<R,S,A>}

{Z<A>, T<A>, A<S,A>, O<S,A>, F<S,A>, S<R,S,A>, C<R,S,A>, R<P,R,S,A>, C<P,R,S,A>, B<P,R,S,A>}

Solution: Arad -> Sibiu -> Rimnicu Vilcea -> Pitesti -> Bucharest

Cost: 140 + 80 + 97 + 101 = 418



{Arad}

{Z<A>, T<A>, S<A>},

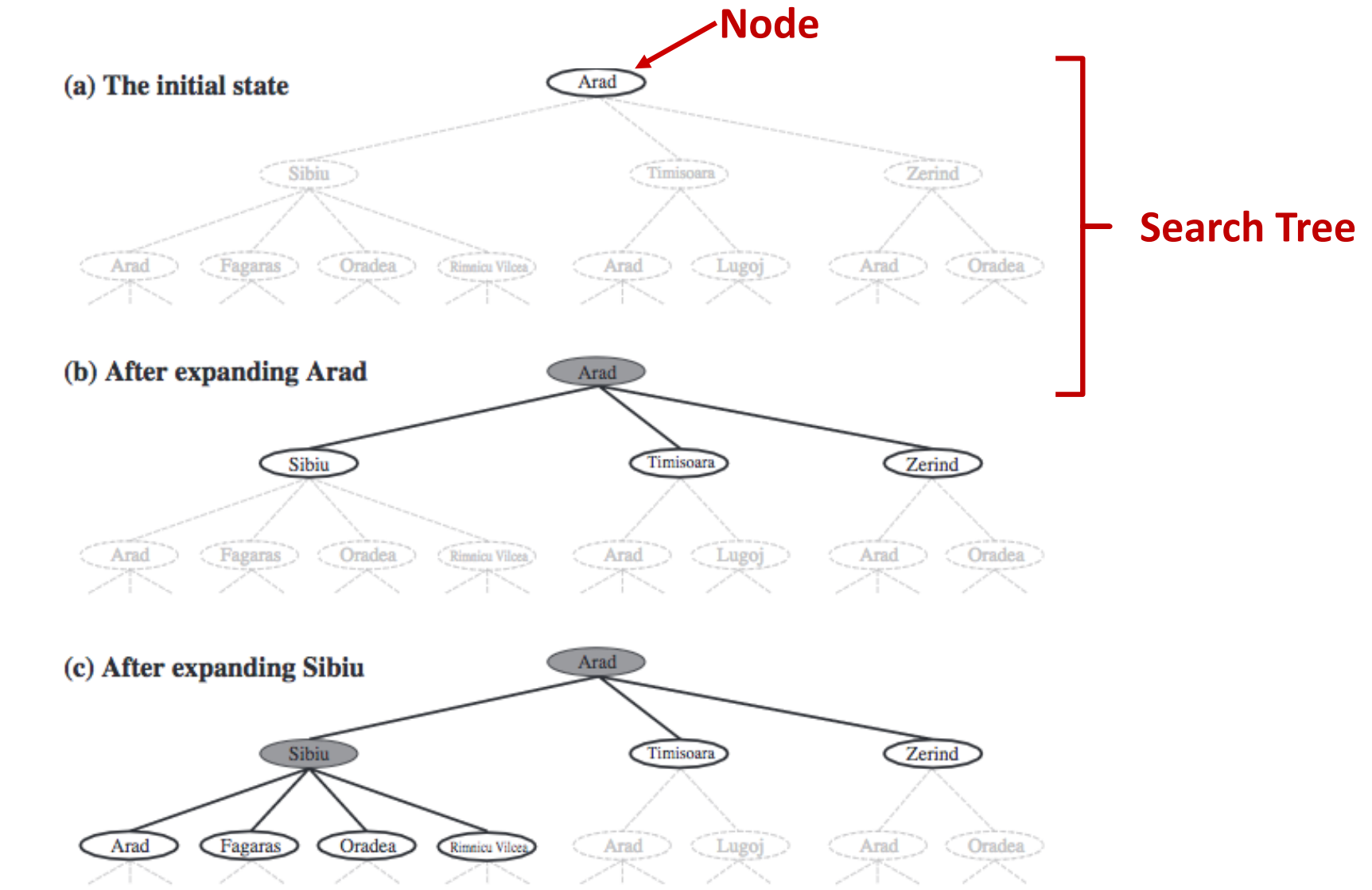
{Z<A>, T<A>, O<S;A>, F<S;A>, A<S;A>, R<S;A>}

{Z<A>, T<A>, O<S;A>, F<S;A>, R<S;A>, Z<A;S;A>, T<A;S;A>, S<A,S,A>}

.....

Frontier is a set of paths not a set of states: cycles become an issue.

Partial Search Space



Selection Rule

The example shows that the order states are selected from the frontier has a critical effect on the operation of the search:

- Whether or not a solution is found
- The cost of the solution found.
- The time and space required by the search.

Critical Properties of Search

Completeness: will the search always find a solution if a solution exists?

Optimality: will the search always find the least cost solution? (when actions have costs)

Time complexity: what is the maximum number of nodes that can be expanded or generated?

Space complexity: what is the maximum number of nodes that have to be stored in memory?

UNINFORMED SEARCH STRATEGIES

Uninformed Search Strategies

These are strategies that **adopt a fixed rule** for selecting the next state to be expanded.

The rule does not change irrespective of the search problem being solved.

These strategies **do not take into account any domain specific information** about the particular search problem.

Popular uninformed search techniques:

- Breadth-First,
- Uniform-Cost,
- Depth-First,
- Depth-Limited, and
- Iterative-Deepening search

Selecting vs. Sorting

Any selection rule can be achieved by employing an appropriate ordering of the frontier set.

A simple equivalence we will exploit

- Order the elements on the frontier.
- Always select the first element.

Breadth First

Place the successors of the current state at the **end** of the frontier.

Example:

- let the states be the positive integers $\{0,1,2,\dots\}$
- let each state n have as successors $n+1$ and $n+2$
 - E.g. $S(1) = \{2, 3\}$; $S(10) = \{11, 12\}$
- Start state 0
- Goal state 5

Breadth First Example.

{0<>}

{1,2} 

{2,2,3}

{2,3,3,4}

{3,3,4,3,4}

{3,4,3,4,4,5}

...

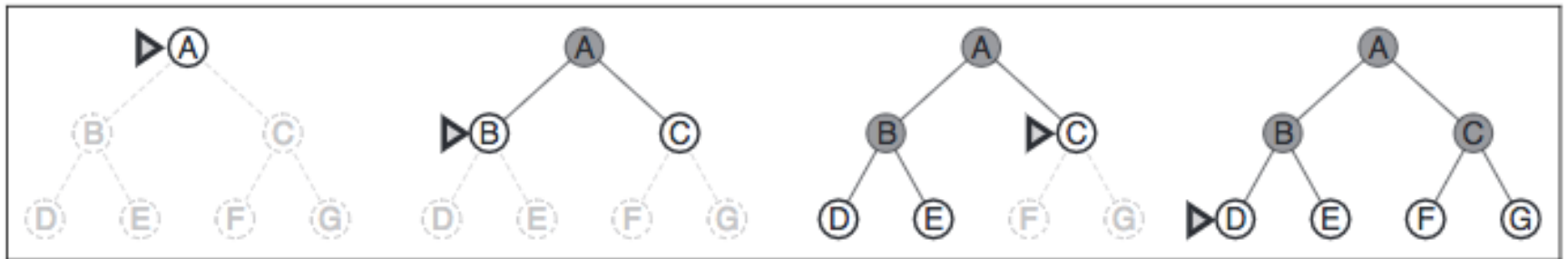
Breadth First Properties

Measuring time and space complexity.

- b = maximum number of successors of any state.
- d = number of actions in the shortest solution.

Breadth-first search

Breadth-first search on a single binary tree



Breadth First Properties

Completeness?

- The length of the path from the initial state to the expanded state must increase monotonically.
- we replace each expanded state with states on longer paths.
- All shorter paths are expanded prior before any longer path.
- Hence, eventually we must examine all paths of length d , and thus find the shortest solution.

Breadth First Properties

Time Complexity?

$$1 + b + b^2 + b^3 + \dots + b^{d-1} + b^d + b(b^d - 1) = O(b^{d+1})$$

Breadth First Properties

Space Complexity?

- $O(b^{d+1})$: If goal node is last node at level d , all of the successors of the other nodes will be on the frontier when the goal node is expanded $b(b^d - 1)$

Optimality?

- Will find shortest length solution
 - least cost solution?

Breadth First Properties

Space complexity is a real problem.

- E.g., let $b = 10$, and say 1000 nodes can be expanded per second and each node requires 100 bytes of storage:

Depth	Nodes	Time	Memory
1	1	1 millisec.	100 bytes
6	10^6	18 mins.	111 MB
8	10^8	31 hrs.	11 GB

- Run out of space long before we run out of time in most applications.

Uniform-Cost Search

Uniform-Cost Search

Keep “Frontier” or “OPEN” ordered by **increasing cost of the path.**

Always **expand the least cost** path. 

Identical to Breadth first if each action has the same cost.

Uniform-Cost Properties

Completeness?

- If each transition has costs $\geq \epsilon > 0$.
- The previous argument used for breadth first search holds: the cost of the path represented by each node n chosen to be expanded must be non-decreasing.

Optimality?

- Finds optimal solution if each transition has cost $\geq \epsilon > 0$.
- Explores paths in the search space in increasing order of cost. So must find minimum cost path to a goal before finding any higher costs paths.

Uniform-Cost Search. Proof of Optimality

Let us prove Optimality more formally. We will reuse this argument later on when we examine Heuristic Search

Uniform-Cost Search. Proof of Optimality

Lemma 1.

Let $c(n)$ be the cost of node n on OPEN (cost of the path represented by n). If n_2 is expanded IMMEDIATELY after n_1 then
 $c(n_1) \leq c(n_2)$.

Proof: there are 2 cases:

Uniform-Cost Search. Proof of Optimality

Lemma 1.

Let $c(n)$ be the cost of node n on OPEN (cost of the path represented by n). If n_2 is expanded IMMEDIATELY after n_1 then
 $c(n_1) \leq c(n_2)$.

Proof: there are 2 cases:

- a. n_2 was on OPEN when n_1 was expanded:

We must have $c(n_1) \leq c(n_2)$ otherwise n_2 would have been selected for expansion rather than n_1

- b. n_2 was added to OPEN when n_1 was expanded

Now $c(n_1) < c(n_2)$ since the path represented by n_2 extends the path represented by n_1 and thus cost at least ϵ more.

Uniform-Cost Search. Proof of Optimality

Lemma 2.

When node n is expanded every path in the search space with cost strictly less than $c(n)$ has already been expanded.

Proof:

- Let $n_0 = \langle \text{Start} \rangle$.
- Let $n_k = \langle \text{Start}, s_1, \dots, s_k \rangle$ be a path with cost less than $c(n)$, i.e., $c(n_k) < c(n)$
- Let n_i be the last node on this path that has been expanded:
 $\langle \text{Start}, s_1, \dots, s_{i-1}, s_i, s_{i+1}, \dots, s_k \rangle$
- So, n_{i+1} must still be on the frontier. Also $c(n_{i+1}) < c(n)$ since the cost of the entire path to n_k , $c(n_k) < c(n)$.
- But then uniform-cost would have expanded n_{i+1} not n .
- So when node n is expanded, every node on this path must already be expanded, i.e., the path represented by n_k has already been expanded. QED

Uniform-Cost Search. Proof of Optimality

Lemma 3.

The first time uniform-cost expands a node n terminating at state S , it has found the minimal cost path to S (it might later find other paths to S but none of them can be cheaper).

Proof:

- All cheaper paths have already been expanded, none of them terminated at S .
- All paths expanded after n will be at least as expensive, so no cheaper path to S can be found later.

So, when a path to a goal state is expanded the path must be optimal (lowest cost).

Uniform-Cost Properties

Time and Space Complexity?

- $O(b^{C^*/\epsilon} + 1)$ where
 - C^* is the cost of the optimal solution
 - ϵ is the minimal cost of an action
 - each transition has costs $\geq \epsilon > 0$.
- There may be many paths with cost $\leq C^*$: there can be as many as b^d paths of length d in the worst case.

Paths with cost lower than C^* can be as long as C^*/ϵ (why no longer?), so might have $b^{C^*/\epsilon}$ paths to explore before finding an optimal cost path.

Uniform Cost Search Overview

Paths in the State Space ordered by cost

Path	Cost
<S>	0
<S, a>	1.0
<S,b>	1.0
<S,a,c>	1.5
<S,d>	2.0
...	...

Note cost is non-decreasing

Uniform Cost Search Overview

Uniform Cost Search expands paths in non-decreasing order of cost. So it only goes down this list.

LEMMA 1

It does not miss any paths on this list. LEMMA 2

Path	Cost
<S>	0
<S, a>	1.0
<S,a,b>	1.5
<S,a,b,c>	3.0
<S,a,b,d,e>	4.0
...	...

Uniform Cost Search Overview

It does not miss any paths on this list. LEMMA 2

Path	Cost
<S>	0
<S, a>	1.0
<S,a,b>	1.5
<S,a,b,c>	3.0
<S,a,b,d,e>	4.0
...	...

- If <S,a,b,d,e> is expanded next, <S,a,b,c> must already been expanded. If not it or <S,a,b,c> or <S,a,b> or <S,a> or <S> must be available for expansion (on OPEN) and would be expanded next as they all have lower cost than <S,a,b,d,e>

Uniform Cost Search Overview

Thus working its way down such a list of paths in this order the first path achieving the goal that Uniform cost search finds will be the cheapest way of achieving the goal.

Critical Properties of Search

REVIEW FROM LAST DAY

Completeness: will the search always find a solution if a solution exists?

Optimality: will the search always find the least cost solution? (when actions have costs)

Time complexity: what is the maximum number of nodes that can be expanded or generated?

Space complexity: what is the maximum number of nodes that have to be stored in memory?

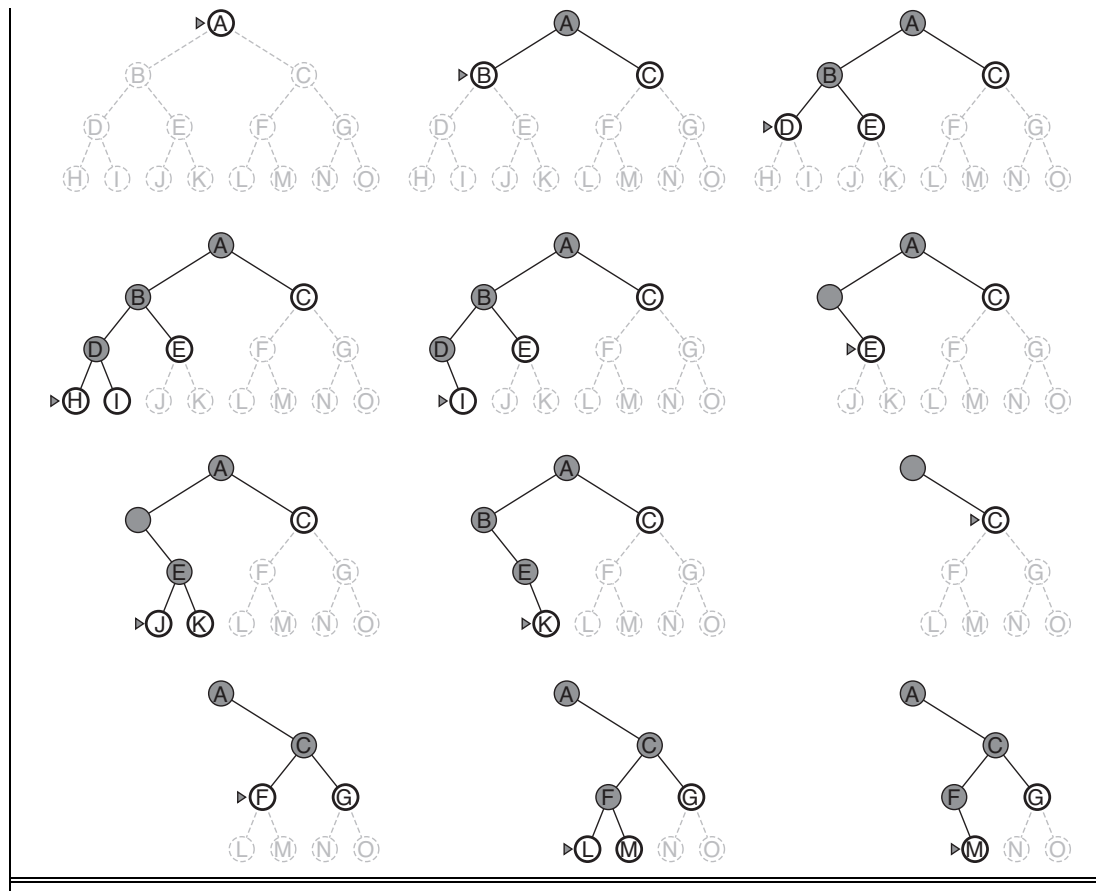
Depth-First Search

Depth First Search

Place the successors of the current state at the **front** of the frontier.

Therefore always expands the **deepest node** in the frontier/OPEN

Depth First Search



Depth First Search Example

(Applied to the example of Breadth First Search)



{0}

{1,2}

{2,3,2}

{3,4,3,2}

{4,5,4,3,2}

{5,6 5,4,3,2}

...

Depth First Properties

Completeness?

- Infinite paths? Cause incompleteness!
- Prune paths with duplicate states?

We get completeness if state space is finite

Optimality?

No!

Depth First Properties

Time Complexity?

- $O(b^m)$ where m is the length of the longest path in the state space.
- Very bad if m is much larger than d (shortest path to a goal state), but if there are many solution paths it can be much faster than breadth first. (Can by good luck bump into a solution quickly).

Depth-First Properties

Depth-First Backtrack Points = unexplored siblings of nodes along current path.

- These are the nodes that remain on open after we extract a node to expand.

Space Complexity?

- $O(bm)$, linear space!
 - Only explore a single path at a time.
 - The Frontier/OPEN only contains the deepest node on the current path along with the **backtrack** points.
- A significant advantage of DFS

Depth-Limited Search

Depth Limited Search

Breadth first has computational, especially, space problems. Depth first can run off down a very long (or infinite) path.

Depth limited search

- Perform depth first search but only to a pre-specified depth limit L .
- No node on a path that is more than L steps from the initial state is placed on the Frontier.
- We “truncate” the search by looking only at paths of length L or less.

Now infinite length paths are not a problem.

But will only find a solution if a solution of length $\leq L$ exists.

Depth Limited Search

DLS(Frontier, Successors, Goal?) 

If Frontier is empty return failure

Curr = select state from Frontier

If(Goal?(Curr)) return Curr.

If Depth(Curr) < L

Frontier' = (Frontier - {Curr}) U Successors(state)

Else

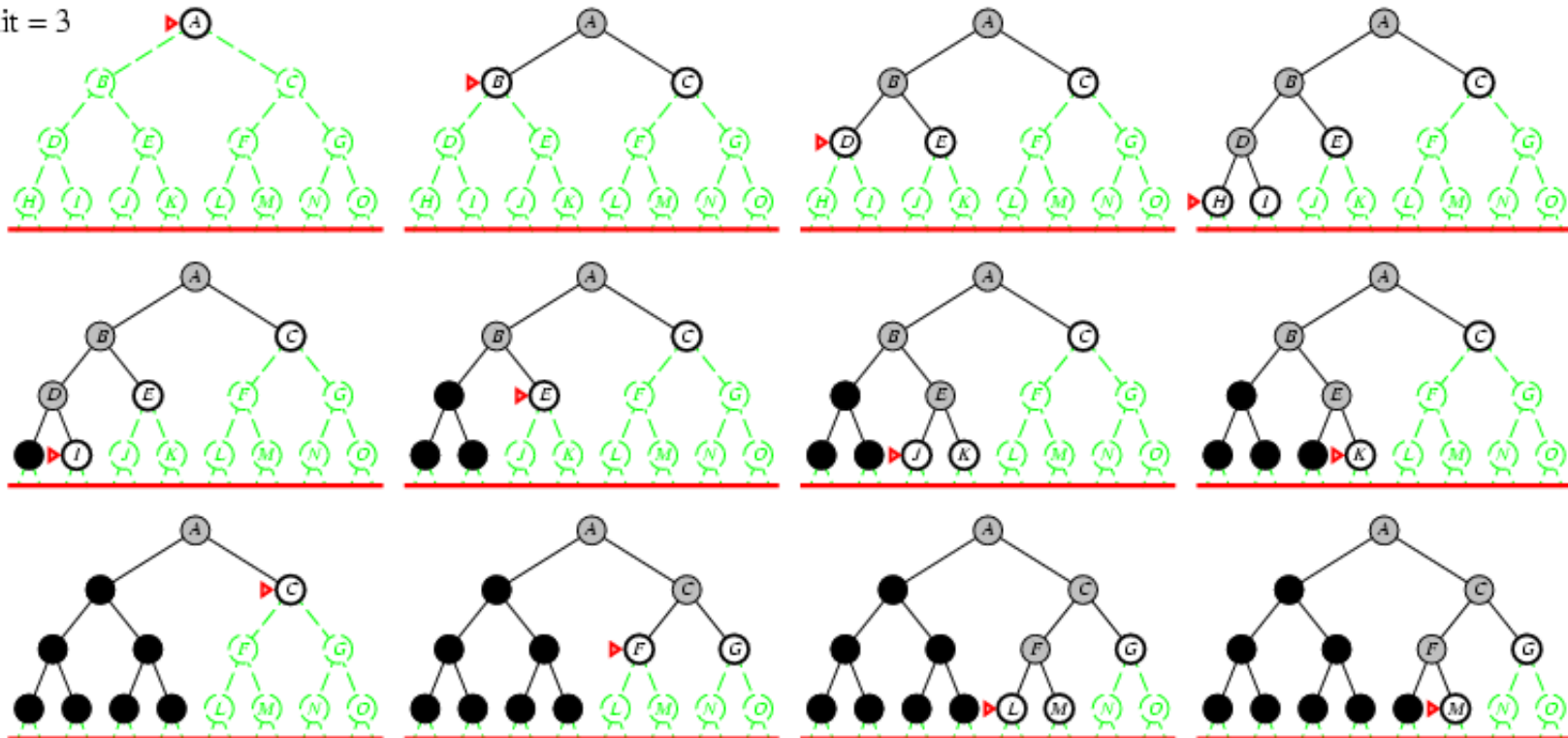
Frontier' = Frontier - {Curr}

CutOffOccured = TRUE.

return DLS(Frontier', Successors, Goal?)

Depth Limited Search Example

Limit = 3



Iterative Deepening Search

Iterative Deepening Search

Take the idea of depth limited search one step further.

Solve the problems of depth-first and breadth-first by extending depth limited search

Starting at depth limit $L = 0$, we iteratively increase the depth limit, performing a depth limited search for each depth limit.

Stop if a solution is found, or if the depth limited search failed without cutting off any nodes because of the depth limit.

- If no nodes were cut off, the search examined all paths in the state space and found no solution → no solution exists.

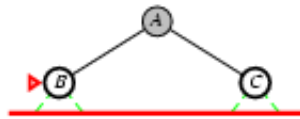
Iterative Deepening Search Example

Limit = 0

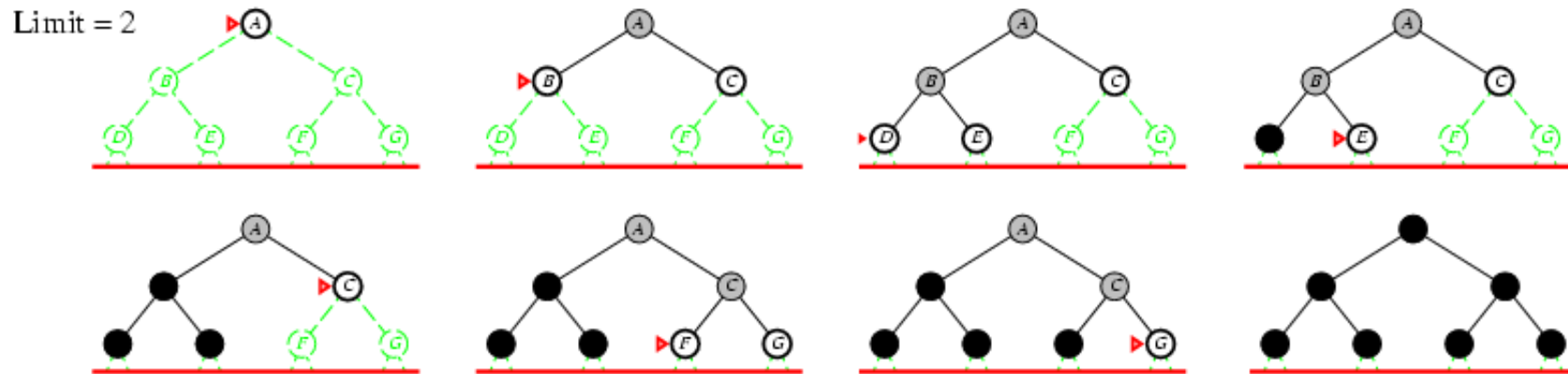


Iterative Deepening Search Example

Limit = 1

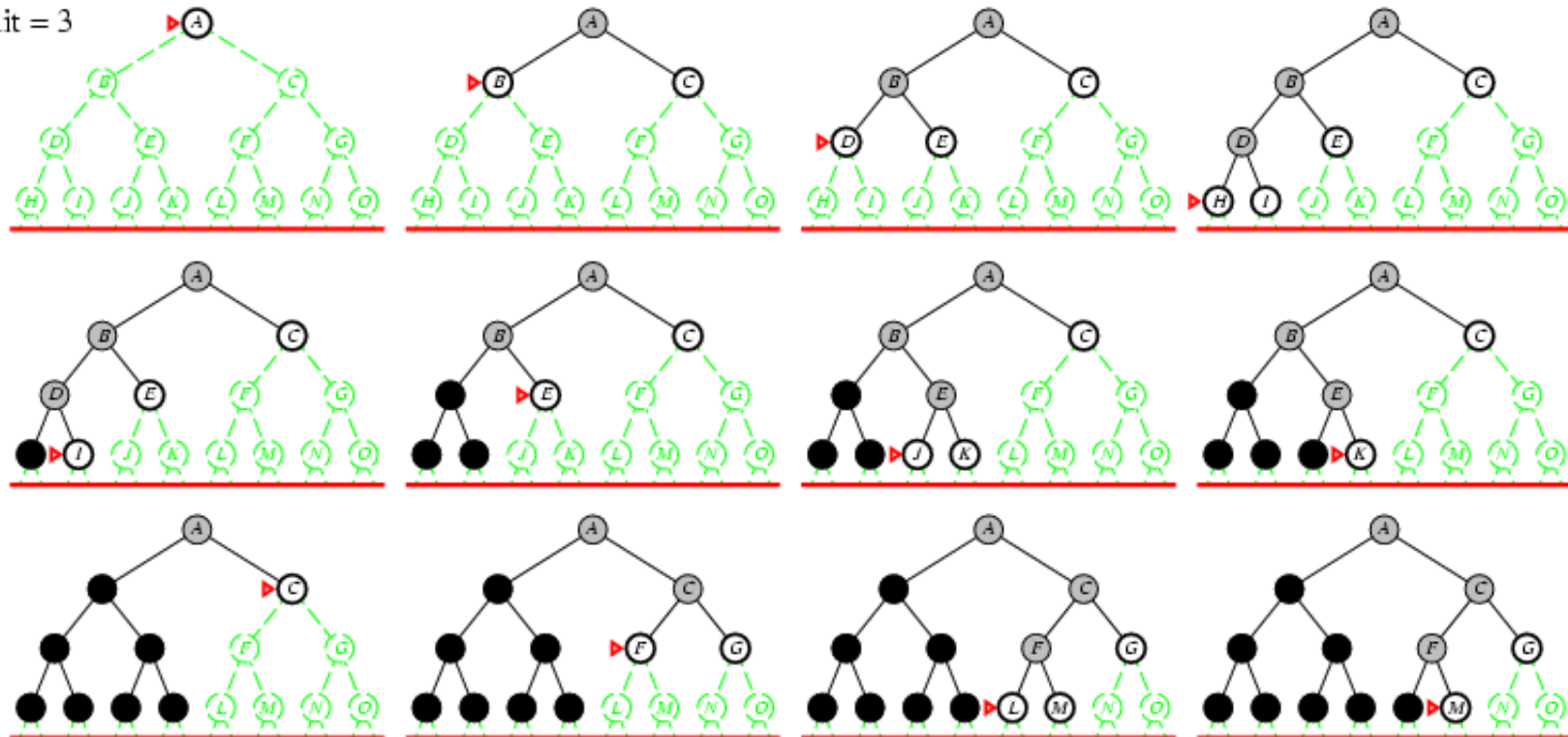


Iterative Deepening Search Example



Iterative Deepening Search Example

Limit = 3



Iterative Deepening Search Example

Iterative Deepening Search Properties

Completeness?

- Yes, if a minimal length solution of length d exists. What happens when the depth limit $L=d$?

What happens when the depth limit $L < d$?

Time Complexity?

Iterative Deepening Search Properties

Time Complexity

- $(d+1)b^0 + db^1 + (d-1)b^2 + \dots + (1)b^d = O(b^d)$
- E.g. $b=4, d=10$
 - $(11)*4^0 + 10*4^1 + 9*4^2 + \dots + 2*4^9 + 1*4^{10} = 1,864,131$
 - $4^{10} = 1,048,576$ (time to explore bottom level)
 - Most nodes (1 of the 1.8 million) lie on bottom level
 - In fact IDS can be more efficient than breadth first search: nodes at limit are not expanded. BFS must expand all nodes until it expand a goal node.

Breadth first can explore more nodes than IDS

- Recall for IDS, the time complexity is
 $(d+1)b^0 + db^1 + (d-1)b^2 + \dots + b^d = O(b^d)$
- Recall for BFS, the time complexity is
 - $1 + b + b^2 + b^3 + \dots + b^{d-1} + b^d + b(b^d - 1) = O(b^{d+1})$

E.g. $b=4, d=10$

- For IDS
 - $(11)*4^0 + 10*4^1 + 9*4^2 + \dots + 4^{10} = 1,864,131$ (nodes generated)
 - $4^{10} = 1,048,576$ (nodes in bottom level)
 - Most nodes lie on bottom layer.
- For BFS
 - $1 + 4 + 4^2 + \dots + 4^{10} + 4(4^{10} - 1) = 5,592,401$ (states generated)
 - In fact IDS can be more efficient than breadth first search: nodes at limit are not expanded. BFS must expand all nodes until it expand a goal node.

Breadth first can explore more nodes than IDS

Iterative Deepening Search Properties

Space Complexity

- $O(bd)$ Still linear!

Optimal?

- Will find shortest length solution which is optimal if costs are uniform.
- If costs are not uniform, we can use a “cost” bound instead.
 - Only expand paths of cost less than the cost bound.
 - Keep track of the minimum cost unexpanded path in each depth first iteration, increase the cost bound to this on the next iteration.
 - This can be very expensive. Need as many iterations of the search as there are distinct path costs.

Path/Cycle Checking

Path Checking

Recall paths are stored on the frontier (this allows us to output the solution path).

If $\langle S, n_1, \dots, n_k \rangle$ is a path to node n_k , and we expand n_k to obtain child c , we have

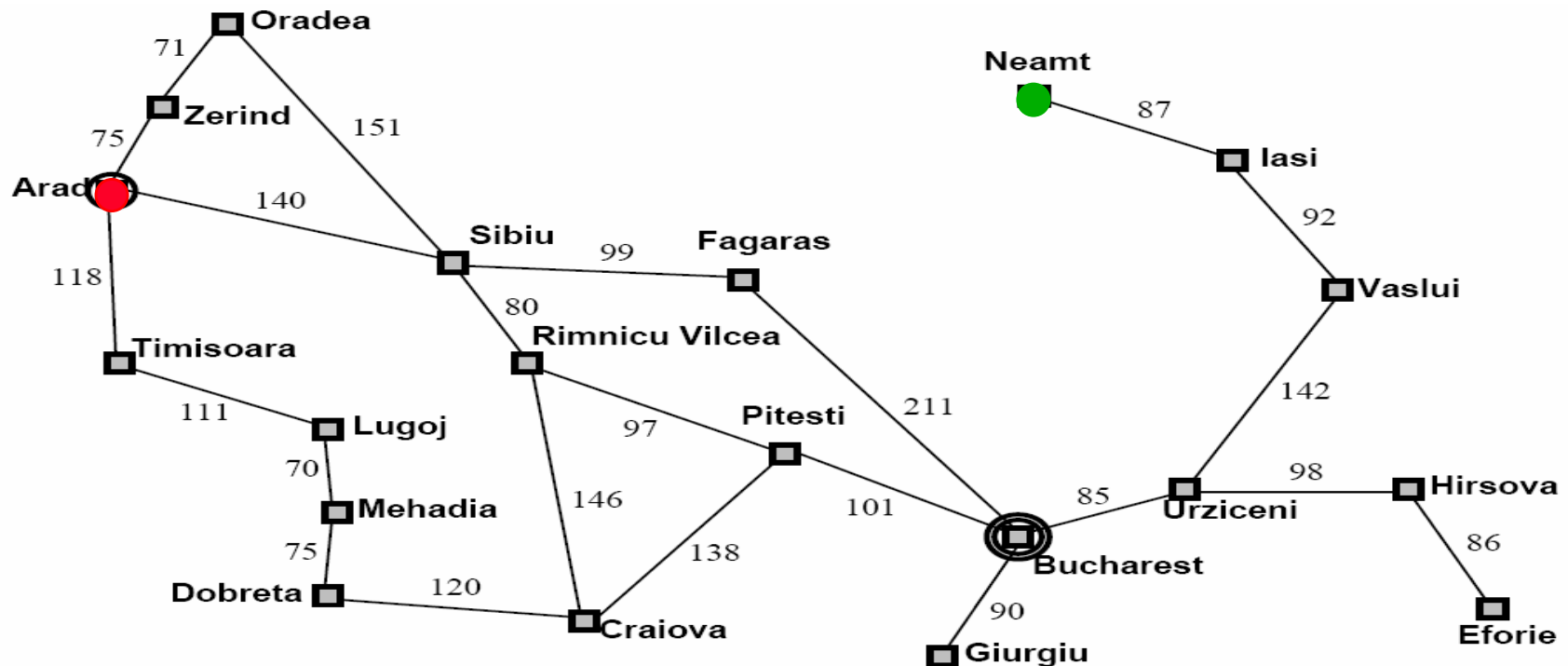
$\langle S, n_1, \dots, n_k, c \rangle$

As the path to “ c ”.

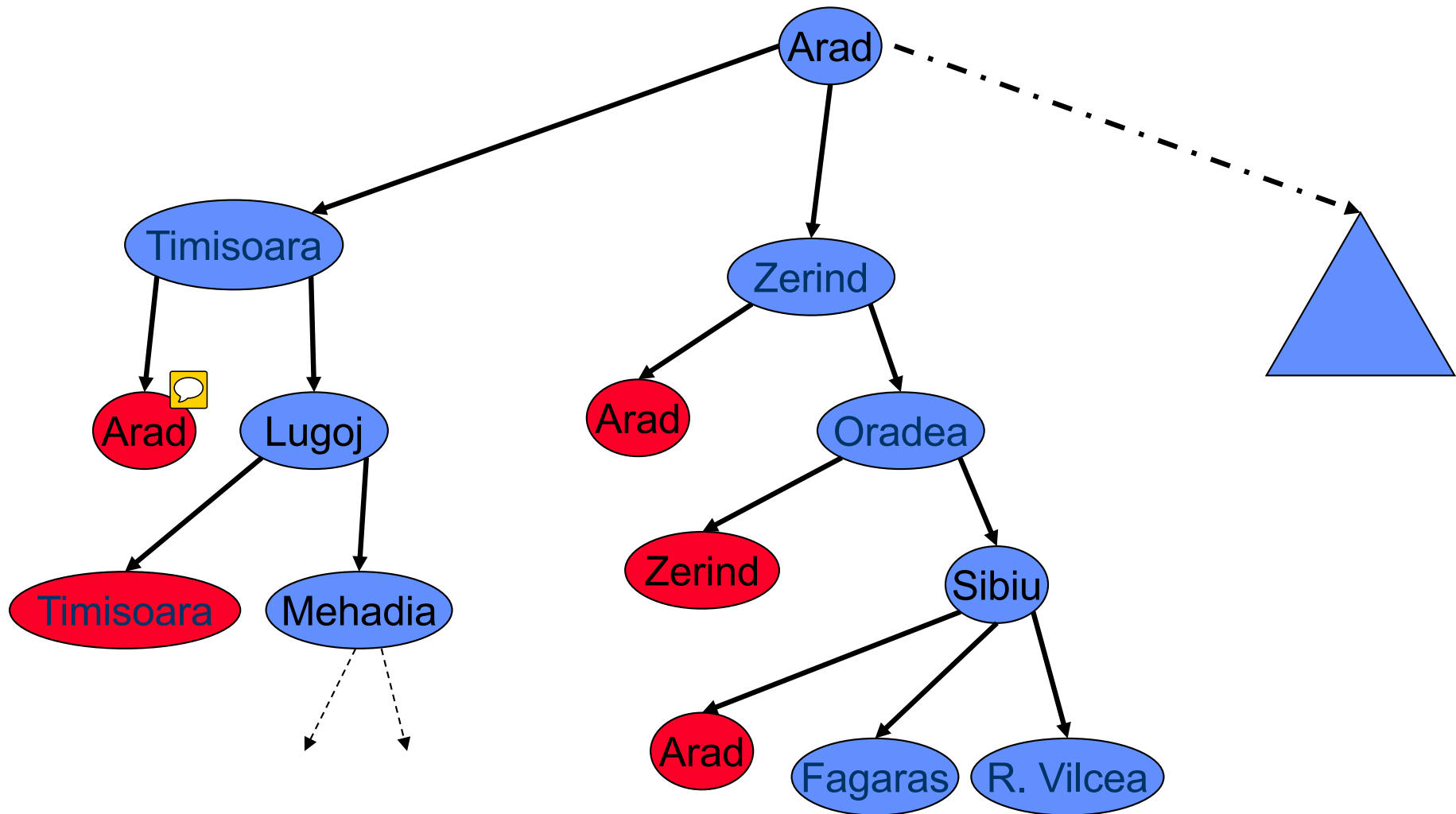
Path checking:

- Ensure that the state c is not equal to the state reached by any ancestor of c along this path.
- That is, paths are checked in isolation!

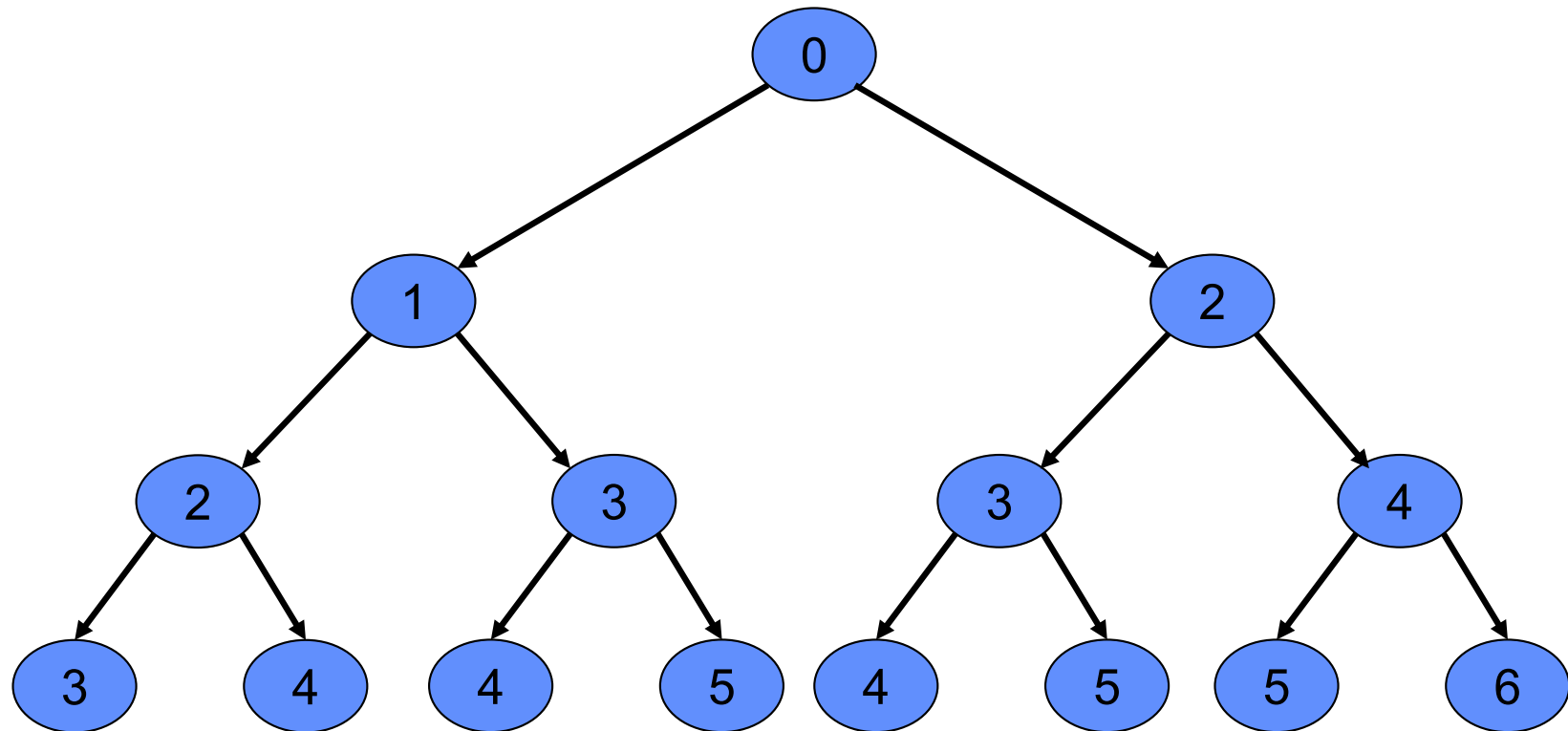
Example: Arad to Neamt



Path Checking Example





Path Checking Example

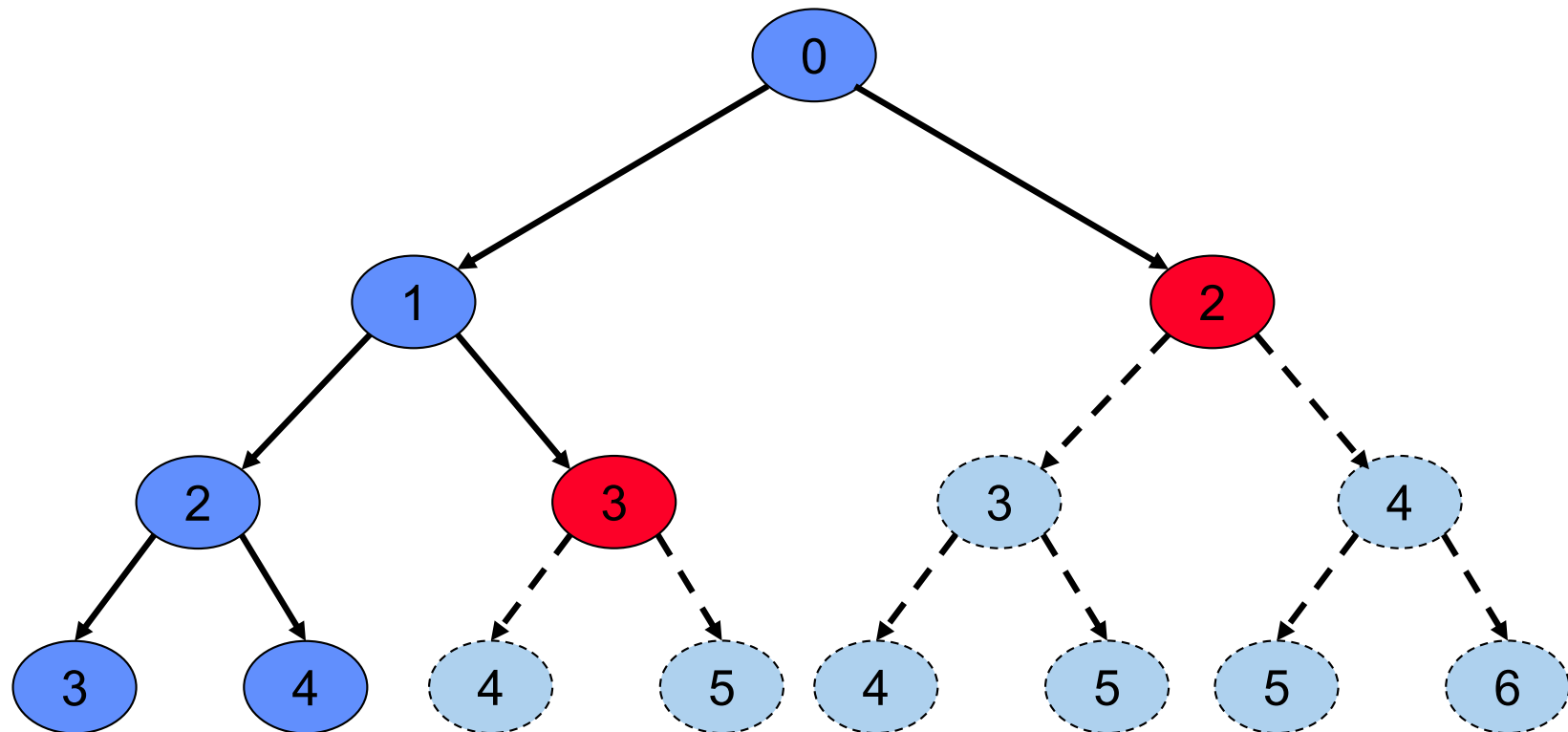


Cycle Checking

Cycle Checking (or “multiple path checking”)

- Keep track of **all states** previously expanded during the search using a **closed list**.
- When we expand n_k to obtain child c
 - ensure that c is not equal to any previously expanded state.
- This is called **cycle checking**, or **multiple path checking**.
- Why can't we utilize this technique with depth-first search?
 - If we modify depth-first search to do cycle checking what happens to space complexity?

Cycle Checking Example



Cycle Checking

High space complexity, only useful with breadth first search.

There is an additional issue when we are looking for an optimal solution

- With uniform-cost search, we still find an optimal solution
 - The first time uniform-cost expands a state it has found the minimal cost path to it.
- This means that the nodes rejected by cycle checking can't have better paths.
- We will see later that we don't always have this property when we do heuristic search.

Cycle Checking Example (BFS)

