AI1/AI&ML - Uninformed Search

Dr Leonardo Stella

Aims of the Session

This session aims to help you:

- Describe asymptotic analysis and why it is important
- Explain the steps to formulate a search problem
- Apply and compare the performance of Breadth-First Search, Depth-First Search and its variations

Overview

- **Asymptotic Analysis**
- Search Problem Formulation
- Breadth-First Search
- Depth-First Search
- Variations of Depth-First Search

Asymptotic Analysis

- Computer scientists are often asked to determine the quality of an algorithm by comparing it with other ones and measure the speed and memory required
- **Benchmarking** is one approach:
  - We run the algorithms and we measure speed (in seconds) and memory consumption (in bytes)
  - Problem: this approach measures the performance of a specific program written in a particular language, on a given computer, with particular input data

- **Asymptotic analysis** is the second approach:
  - It is a mathematical abstraction over both the exact number of operations (by ignoring constant factors) and exact content of the input (by considering the size of the input, only)
  - It is independent of the particular implementation and input

Asymptotic Analysis

- The first step in the analysis is to abstract over the input. In practice, we characterise the size of the input, which we call n
- The second step is to abstract over the implementation. The idea is to find some measure that reflects the running time of the algorithm
- For asymptotic analysis, we typically use 3 notations:
  - Big O notation: $O(\cdot)$
  - Big Omega notation: $\Omega(\cdot)$
  - Big Theta notation: $\Theta(\cdot)$

Asymptotic Analysis: Big O

- We say that $f(n) \in O(g(n))$ when the following condition holds:

$$\exists k > 0 \; \exists n_0 \forall n > n_0 \colon |f(n)| \leq k \cdot g(n)$$

- The above reads: “There exists a positive constant k, n_0 such that for all $n > n_0$, $|f(n)| \leq k \cdot g(n)$”
- In simple terms, this is equivalent to saying that $|f|$ is bounded above by a function g (up to a constant factor) asymptotically

Asymptotic Analysis: Big Theta and Big Omega

- We say that $f(n) \in \Omega(g(n))$ when the following condition holds:

$$\exists k > 0 \; \exists n_0 \forall n > n_0 \colon |f(n)| \geq k \cdot g(n)$$

- This is equivalent to saying that f is bounded below by g asymptotically
- We say that $f(n) \in \Theta(g(n))$ when the following condition holds:

$$\exists k_1, k_2 > 0 \; \exists n_0 \forall n > n_0 \colon k_1 \cdot g(n) \leq |f(n)| \leq k_2 \cdot g(n)$$

- Or f is bounded both above and below by g asymptotically

Asymptotic Analysis: Example

- Consider the following algorithm (pseudocode):

function SUMMATION(*sequence*) **returns** a number
  sum ← 0
  for $i = 1$ **to** LENGTH(*sequence*) **do**
    sum ← *sum* + *sequence*[i]
  return *sum*

- Step 1: abstract over input, e.g., the length of the *sequence*
- Step 2: abstract over the implementation, e.g., total number of steps. If we call this characterisation $T(n)$ and we count lines of code, we have $T(n) = 2n + 2$

Asymptotic Analysis: Example

- Consider the following algorithm (pseudocode):

function SUMMATION(*sequence*) **returns** a number
  sum ← 0
  for $i = 1$ **to** LENGTH(*sequence*) **do**
    sum ← *sum* + *sequence*[i]
  return *sum*

- We say that the SUMMATION algorithm is $O(n)$, meaning that its measure is at most of constant times n with few possible exceptions
- $T(n) \in O(f(n))$ if $T(n) \leq k \cdot f(n)$ for some k, for all $n > n_0$
- For $T(n) = 2n + 2$, an example would be: $k = 3, n_0 = 2$

Summary

- Asymptotic analysis is a powerful tool to describe the speed and memory consumption of an algorithm
- It is useful as it is independent of a particular implementation and input
- It is an approximation as the input n approaches infinity and over the number of steps required
- Convenient to compare algorithms, e.g., an $O(n)$ algorithm is better than an $O(n^2)$ algorithm
- Other notations exist, such as $\Omega(n)$ and $\Theta(n)$

Overview

- Asymptotic Analysis
- **Search Problem Formulation**
- Breadth-First Search
- Depth-First Search
- Variations of Depth-First Search

Problem-Solving Agents

- In this lecture, we introduce the concept of a goal-based agent called **problem-solving agent**
- An agent is something that perceives and acts in an environment
- A problem-solving agent
  - Uses atomic representations (each state of the world is perceived as indivisible)
  - Requires a precise definition of the problem and its goal/solution

Search Problem Formulation

- **Problem formulation** is the process of deciding what actions and states to consider, given a goal

- To this end, we make the following assumptions about the environment:
  - **Observable**, i.e., the agent knows the current state
  - **Discrete**, i.e., there are only finitely many actions at any state
  - **Known**, i.e., the agent knows which states are reached by each action
  - **Deterministic**, i.e., each action has exactly one outcome

- Under these assumptions, the solution to any problem is a fixed sequence of actions

Search Problem Formulation

- The agent's task is to find out how to act, now and in the future, in order to reach a goal state: namely to determine a sequence of actions
- The process of looking for a sequence of actions is called **search**
- A solution to a search problem is the sequence of actions from the initial state to the goal state

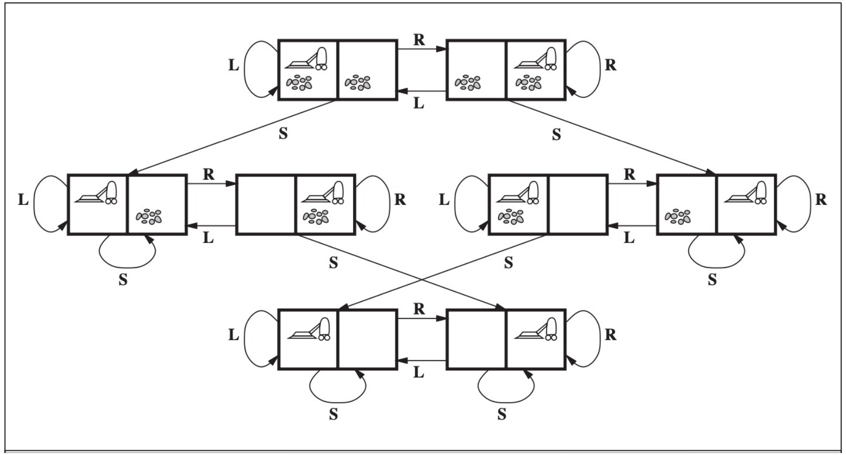
Search Problem Formulation

- A problem is defined formally by five components:
  - **Initial state**, i.e., the state that the agent starts in
  - **Actions**, i.e., a description of all possible actions that can be executed in a given state *s*
  - **Transition model**, i.e., the states resulting from executing each action *a* from every state *s* (a description of what each action does)
  - **Goal test** to determine if a state is a goal state
  - **Path cost** function that assigns a value (cost) to each path

- The first three components considered together define the **state space** of the problem, in the form of a directed graph or network
- A path in the state space is a sequence of states connected by a sequence of actions

Example: Vacuum World

- Let us consider the following example where the state is determined by the dirt location and agent location
  - **Initial state**: any state
  - **Actions**: L (left), R (right) and S (suction)
  - **Transition model**: see image
  - **Goal test**: checks if all squares are clean
  - **Path cost**: each step costs 1

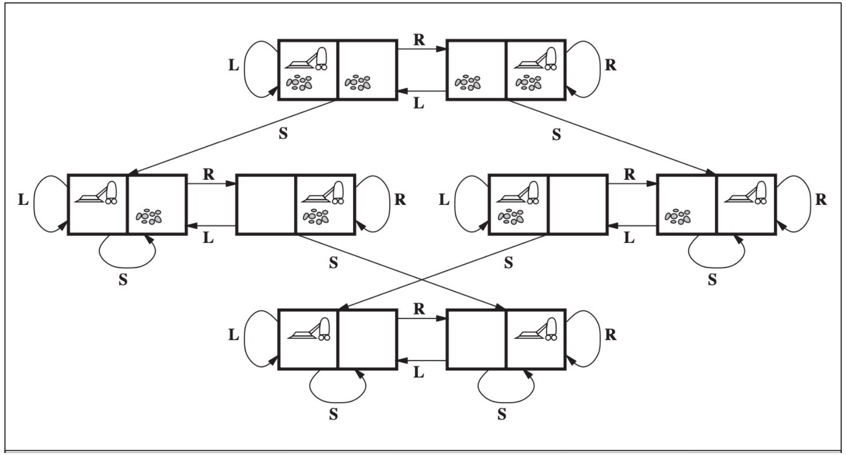


Example: Vacuum World

- Let’s find the solution when the initial state is the top-left state, namely the agent is in the left square, both squares are dirty

- **Example of solution**: S (suction), R (right), S (suction)

- **Cost of the solution**: 1 + 1 + 1 = 3

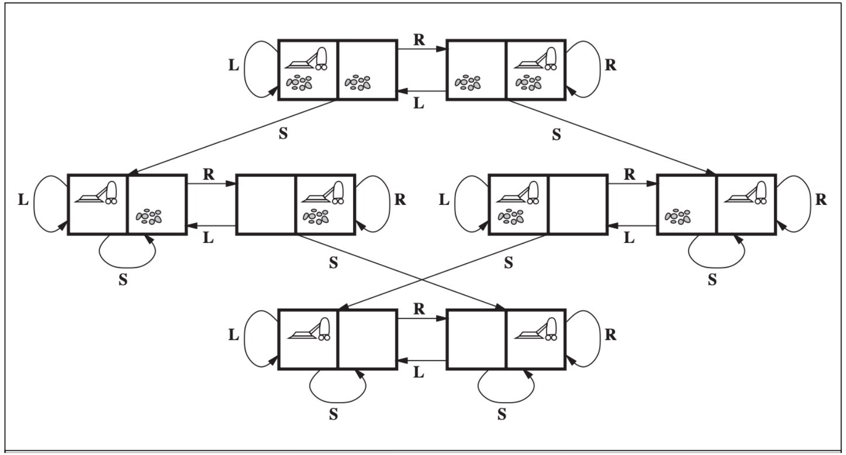


Discussion

- It is important to note that typical AI problems have a large number of states and it is virtually impossible to draw the state space graph

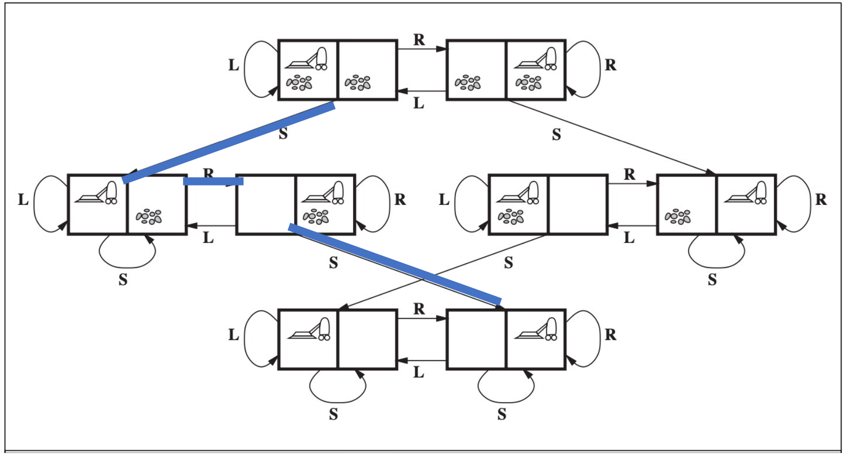
- For the state space graph for the vacuum world example has a small number of states

- The state space graph for chess would be very large



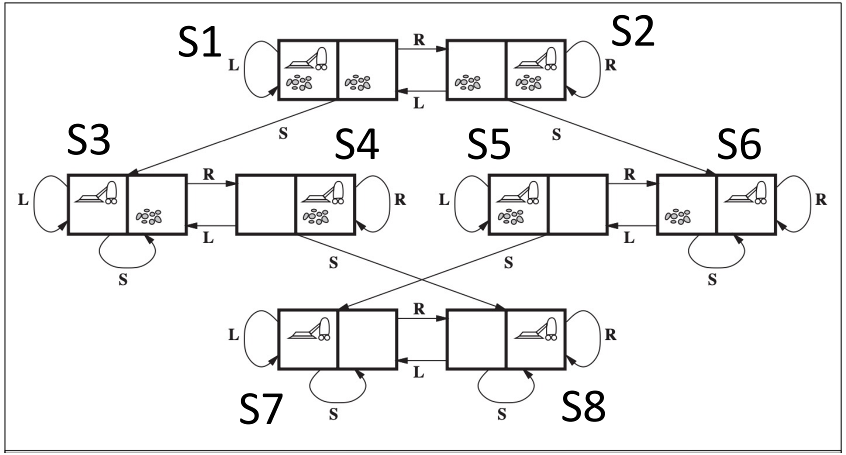
Notation

- A solution can be seen as a path in the state space graph



Notation

- A solution can be seen as a path in the state space graph
- Each state corresponds to a node in the state space graph



Summary

- A **problem-solving agent** is an agent that is able to search for a solution in a given problem

- **Problem formulation**, namely the process of deciding what actions and states to consider, given a goal

Overview

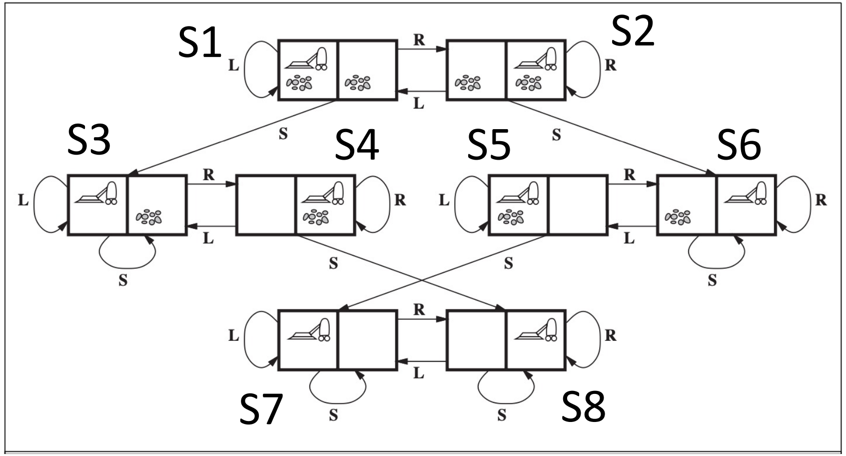
- Asymptotic Analysis
- Search Problem Formulation
- **Breadth-First Search**
- Depth-First Search
- Variations of Depth-First Search

Searching for Solutions

- A solution is an action sequence from an initial state to a goal state
- Possible action sequences form a **search tree** with initial state at the root; actions are the branches and nodes correspond to the state space
- The idea is to expand the current state by applying each possible action: this generates a new set of states

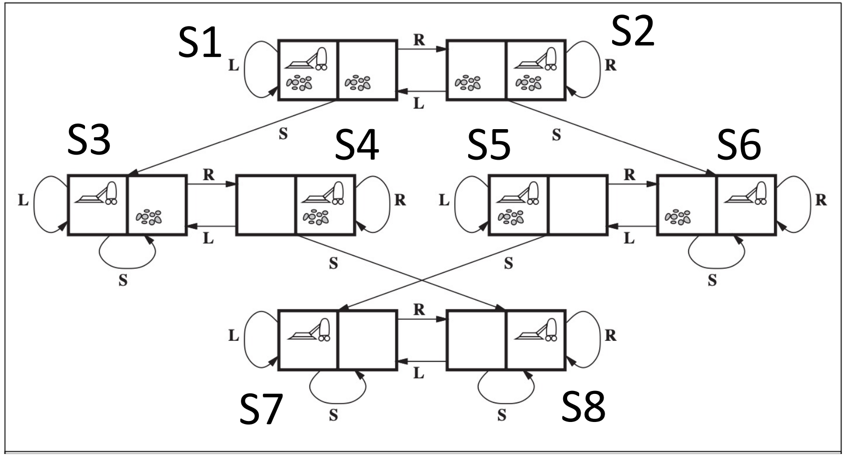
Searching for Solutions

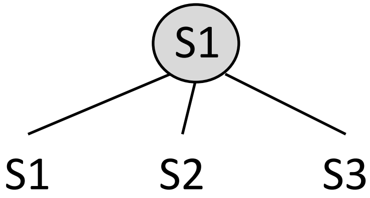
- Let us consider the example from before



Searching for Solutions

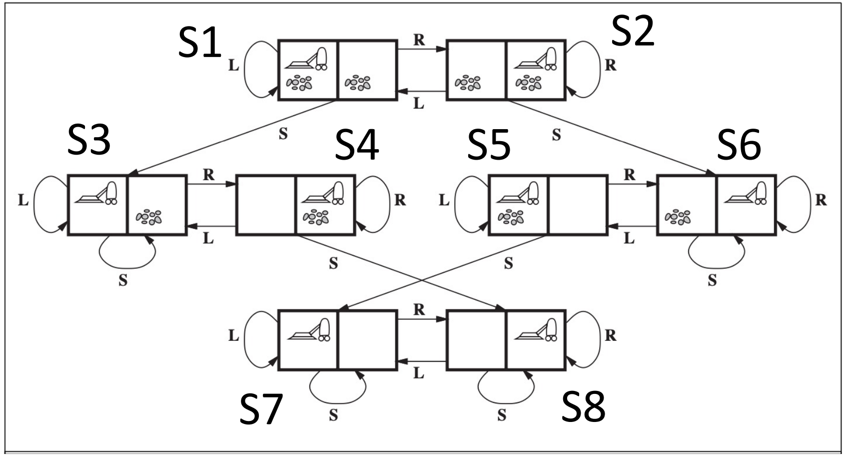
- Let us consider the example from before
- If S1 is the initial state and {S7, S8} is the set of goal states, the corresponding search tree after expanding the initial state is:

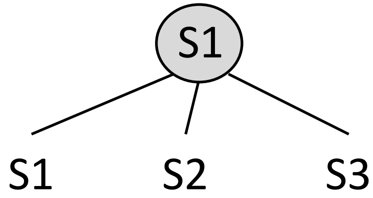




Searching for Solutions

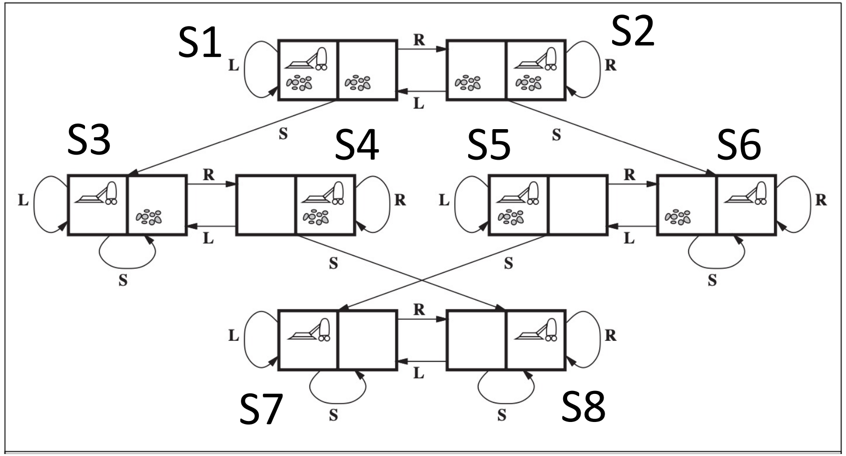
- Each of the three nodes resulting from the first expansion is a **leaf node**
- The set of all leaf nodes available for expansion at any given time is called the **frontier** (also sometimes called the **open list**)
- The path from S1 to S1 is a **loopy path** and in general is not considered

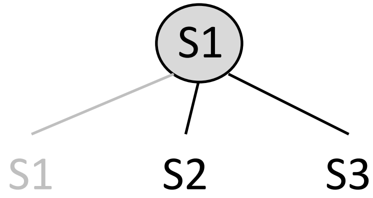




Searching for Solutions

- Each of the three nodes resulting from the first expansion is a **leaf node**
- The set of all leaf nodes available for expansion at any given time is called the **frontier** (also sometimes called the **open list**)
- The path from S1 to S1 is a **loopy path** and in general is not considered





Uninformed Search Strategies

- **Uninformed search** (also called **blind search**) means that the strategies have no additional information about states beyond that provided in the problem definition

- Uninformed search strategies can only generate successors and distinguish a goal state from a non-goal state

- The key difference between two uninformed search strategies is the **order** in which nodes are expanded

Breadth-First Search

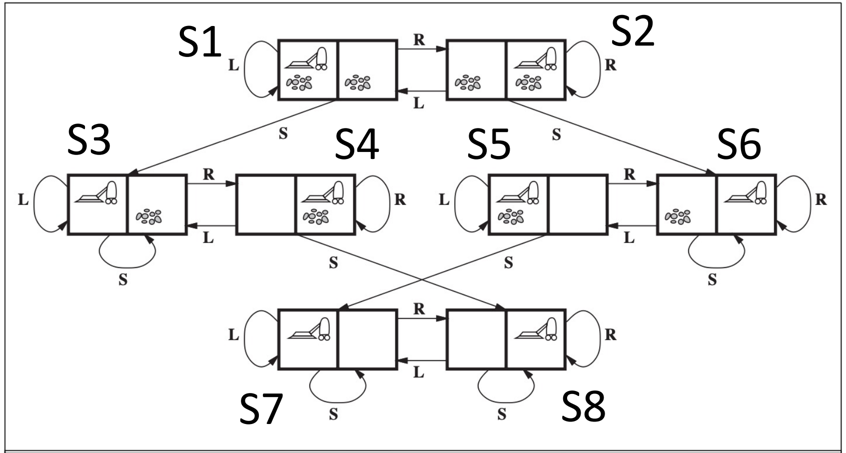
- Breadth-First search is one of the most common search strategies:
  - The root node is expanded first
  - Then, all the successors of the root node are expanded
  - Then, the successors of each of these nodes

- In general, the frontier nodes that are expanded belong to a given depth of the tree

- This is equivalent to expanding the shallowest unexpanded node in the frontier; simply use a **queue** (**FIFO**) for expansion

Breadth-First Search

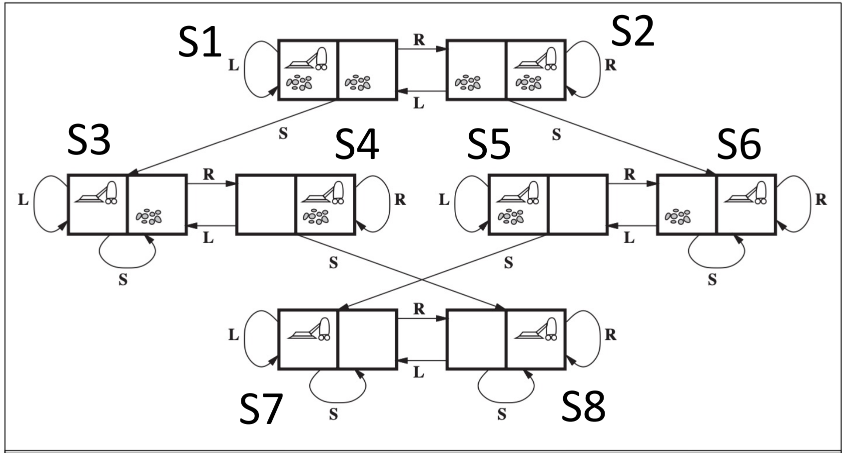
- Breadth-First search algorithm:
  - **Expand** the shallowest node in the frontier
  - **Do not add** children in the frontier if the node is already in the frontier or in the list of visited nodes (to avoid loopy paths)
  - **Stop** when a goal node is added to the frontier

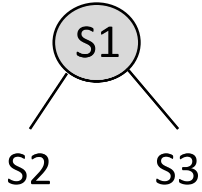


S1

Breadth-First Search

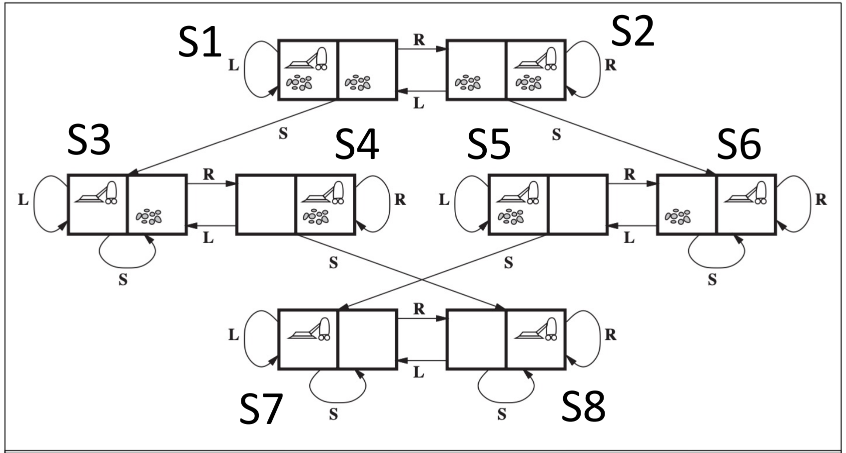
- Breadth-First search algorithm:
  - **Expand** the shallowest node in the frontier
  - **Do not add** children in the frontier if the node is already in the frontier or in the list of visited nodes (to avoid loopy paths)
  - **Stop** when a goal node is added to the frontier

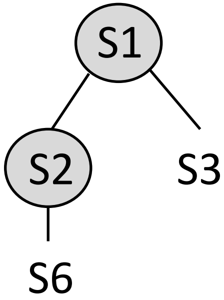




Breadth-First Search

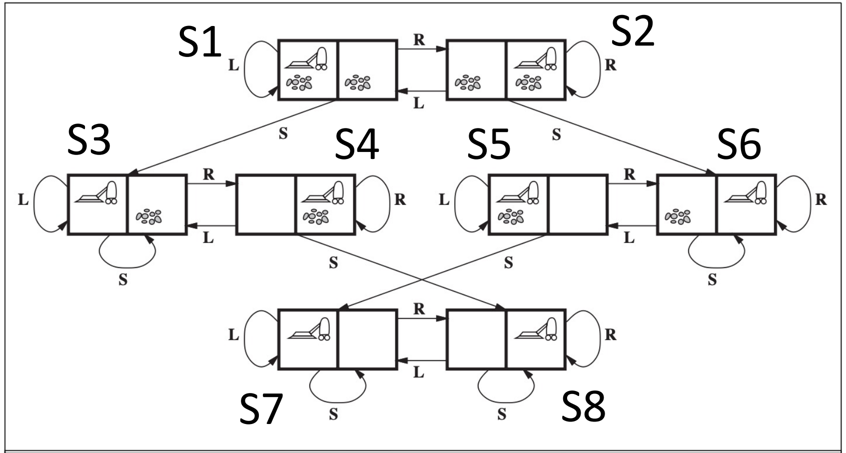
- Breadth-First search algorithm:
  - **Expand** the shallowest node in the frontier
  - **Do not add** children in the frontier if the node is already in the frontier or in the list of visited nodes (to avoid loopy paths)
  - **Stop** when a goal node is added to the frontier

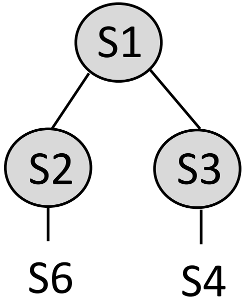




Breadth-First Search

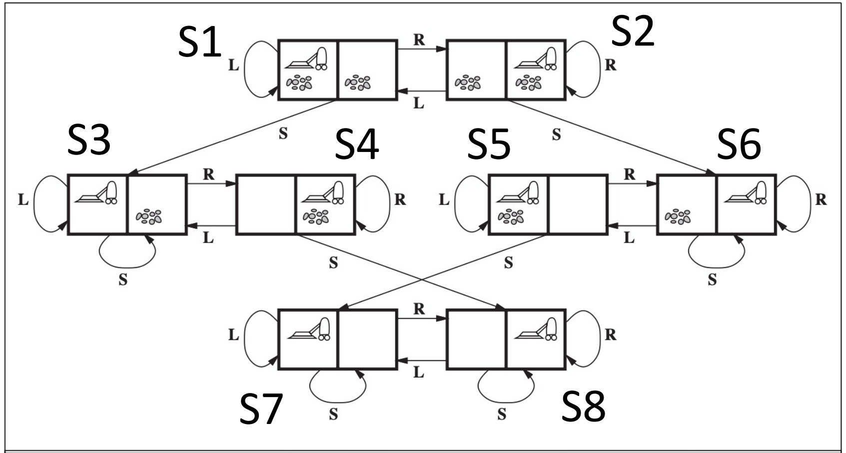
- Breadth-First search algorithm:
  - **Expand** the shallowest node in the frontier
  - **Do not add** children in the frontier if the node is already in the frontier or in the list of visited nodes (to avoid loopy paths)
  - **Stop** when a goal node is added to the frontier

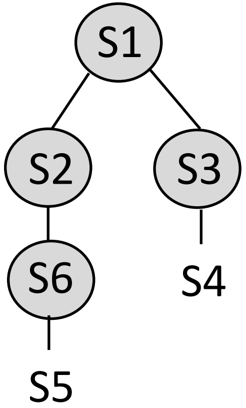




Breadth-First Search

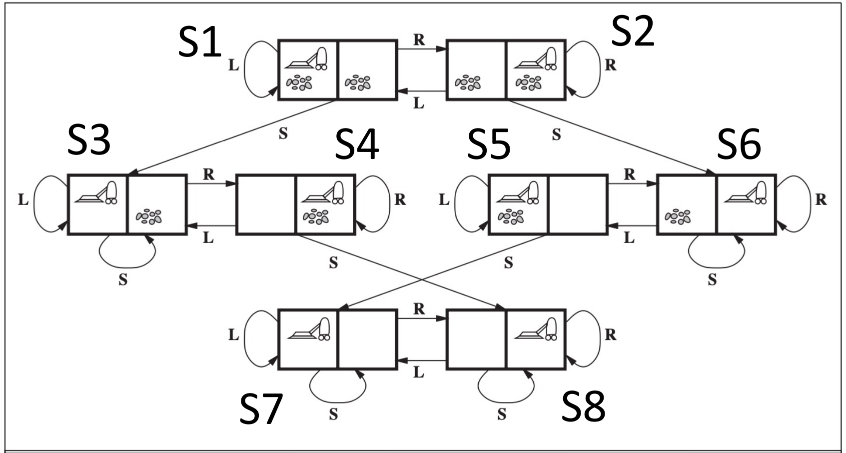
- Breadth-First search algorithm:
  - **Expand** the shallowest node in the frontier
  - **Do not add** children in the frontier if the node is already in the frontier or in the list of visited nodes (to avoid loopy paths)
  - **Stop** when a goal node is added to the frontier

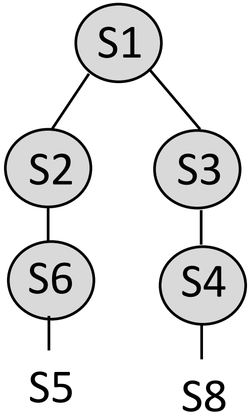




Breadth-First Search

- Breadth-First search algorithm:
  - **Expand** the shallowest node in the frontier
  - **Do not add** children in the frontier if the node is already in the frontier or in the list of visited nodes (to avoid loopy paths)
  - **Stop** when a goal node is added to the frontier





Breadth-First Search

- Solution:

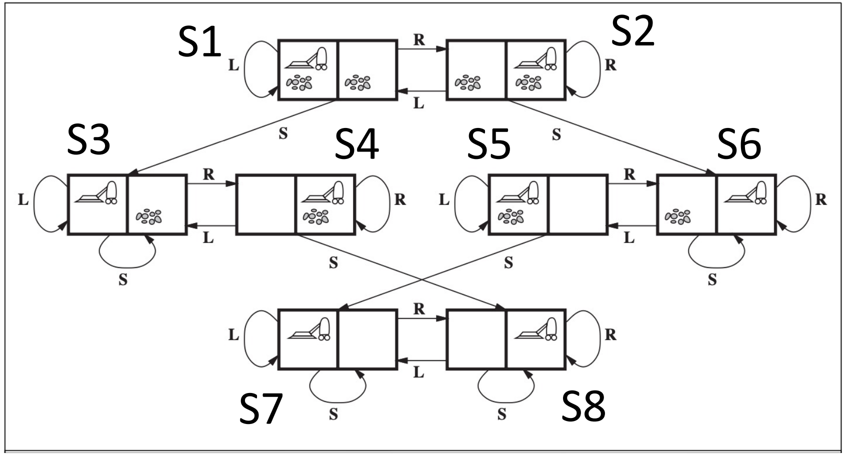
S, R, S

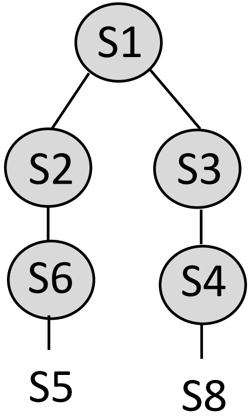
- Cost of the solution:

1 + 1 + 1 = 3

- Order of nodes visited

S1, S2, S3, S6, S4





Measuring Performance

We can evaluate the performance of an algorithm based on the following:

- **Completeness**, i.e., whether the algorithm is guaranteed to find a solution if there is one
- **Optimality**, i.e., whether the strategy is able to find the optimal solution
- **Time complexity**, i.e., the time the algorithm takes to find a solution
- **Space complexity**, i.e., the memory used to perform the search

Measuring Performance

We can evaluate the performance of an algorithm based on the following:

- **Completeness**, i.e., whether the algorithm is guaranteed to find a solution if there is one
- **Optimality**, i.e., whether the strategy is able to find the optimal solution
- **Time complexity**, i.e., the time the algorithm takes to find a solution
- **Space complexity**, i.e., the memory used to perform the search

- To measure the performance, the size of the space graph is typically used, i.e., $|\mathcal{V}| + |\mathcal{E}|$, the set of vertices and set of edges, respectively

Measuring Performance

- In AI, we use an implicit representation of the graph via the initial state, actions and transition model (also the graph could be infinite)

- Therefore, the following three quantities are used
  - **Branching factor**, the maximum number of successors of each node: b
  - **Depth** of the shallowest goal node (number of steps from the root): d
  - The maximum length of any path in the state space: m

BFS - Performance

Let us evaluate the performance of the breadth-first search algorithm

- **Completeness**: if the goal node is at some finite depth d, then the BFS algorithm is complete as it will find it (given that b is finite)
- **Optimality**: BFS is optimal if the path cost is a nondecreasing function of the depth of the node (e.g., all actions have the same cost)

BFS - Performance

Let us evaluate the performance of the breadth-first search algorithm

- **Completeness**: if the goal node is at some finite depth d, then the BFS algorithm is complete as it will find it (given that b is finite)
- **Optimality**: BFS is optimal if the path cost is a nondecreasing function of the depth of the node (e.g., all actions have the same cost)
- **Time complexity**: $O(b^d)$, assuming a uniform tree where each node has b successors, we generate $b + b^2 + \cdots + b^d = O(b^d)$

BFS - Performance

Let us evaluate the performance of the breadth-first search algorithm

- **Completeness**: if the goal node is at some finite depth d, then the BFS algorithm is complete as it will find it (given that b is finite)
- **Optimality**: BFS is optimal if the path cost is a nondecreasing function of the depth of the node (e.g., all actions have the same cost)
- **Time complexity**: $O(b^d)$, assuming a uniform tree where each node has b successors, we generate $b + b^2 + \cdots + b^d = O(b^d)$
- **Space complexity**: $O(b^d)$, if we store all expanded nodes, we have $O(b^{d-1})$ explored nodes in memory and $O(b^d)$ in the frontier

Summary

- Uninformed tree search strategies have no additional information

- Breadth-First Search is a search algorithm that expands the nodes in the frontier starting from the shallowest, similar to a queue (FIFO)

- This algorithm is complete (for finite b), optimal (if the path cost is nondecreasing), but it has high time and space complexity $O(b^d)$

Overview

- Asymptotic Analysis
- Search Problem Formulation
- Breadth-First Search
- **Depth-First Search**
- Variations of Depth-First Search

Depth-First Search

- Depth-First search is another common search strategy:
  - The root node is expanded first
  - Then, the first (or one at random) successor of the root node is expanded
  - Then, the deepest node in the current frontier is expanded

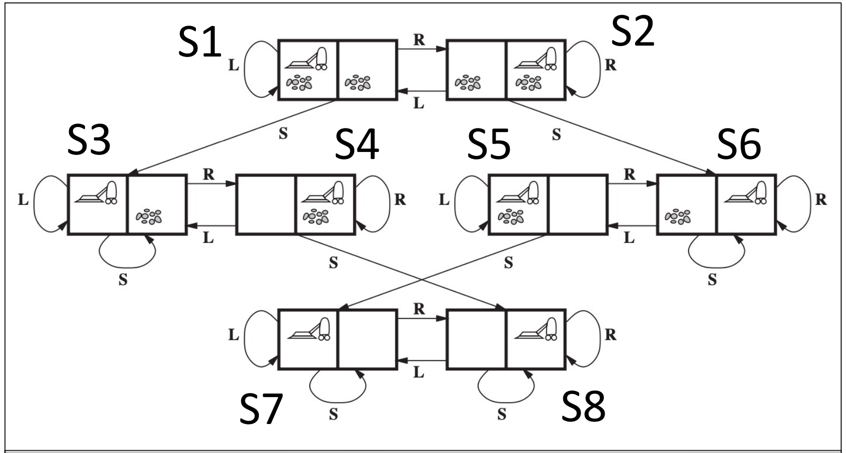
- This is equivalent to expanding the deepest unexpanded node in the frontier; simply use a **stack** (**LIFO**) for expansion

- Basically, the most recently generated node is chosen for expansion

Depth-First Search

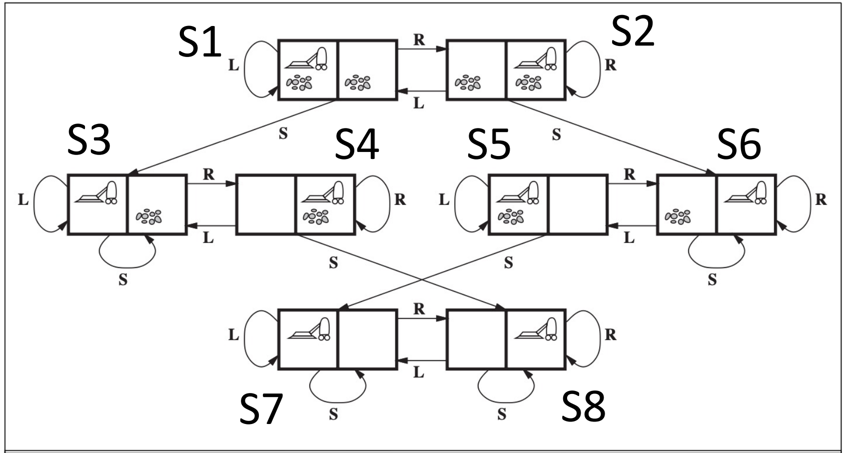
- Depth-First search algorithm:
  - **Expand** the deepest node in the frontier
  - **Do not add** children in the frontier if the node is already in the frontier or in the list of visited nodes (to avoid loopy paths)
  - **Stop** when a goal node is visited

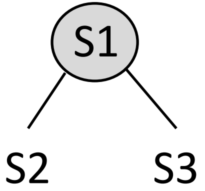
S1



Depth-First Search

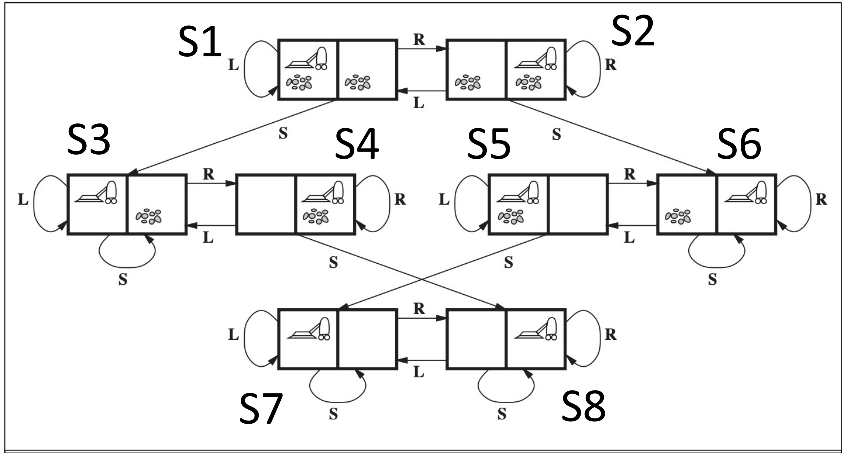
- Depth-First search algorithm:
  - **Expand** the deepest node in the frontier
  - **Do not add** children in the frontier if the node is already in the frontier or in the list of visited nodes (to avoid loopy paths)
  - **Stop** when a goal node is visited

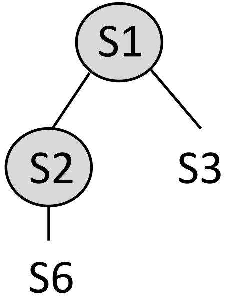




Depth-First Search

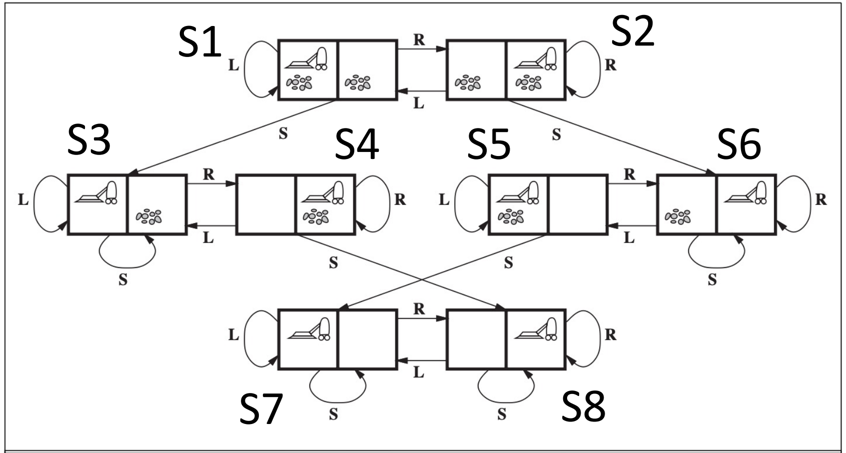
- Depth-First search algorithm:
  - **Expand** the deepest node in the frontier
  - **Do not add** children in the frontier if the node is already in the frontier or in the list of visited nodes (to avoid loopy paths)
  - **Stop** when a goal node is visited

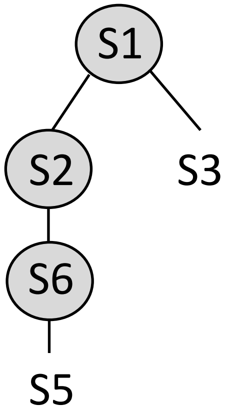




Depth-First Search

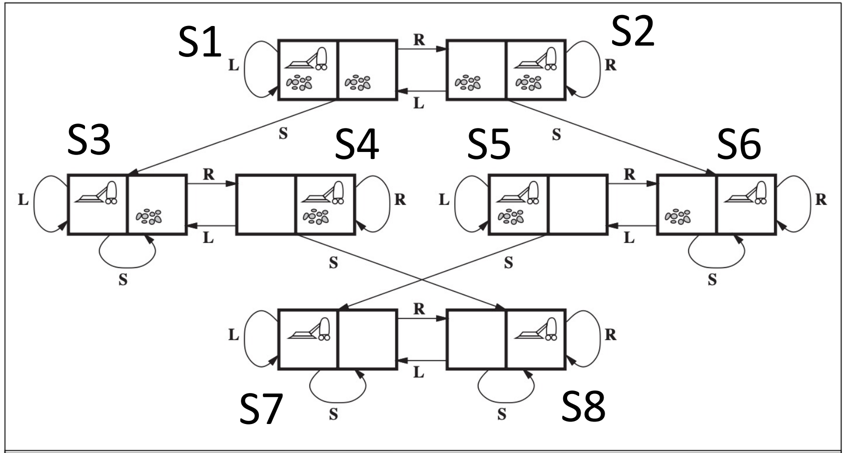
- Depth-First search algorithm:
  - **Expand** the deepest node in the frontier
  - **Do not add** children in the frontier if the node is already in the frontier or in the list of visited nodes (to avoid loopy paths)
  - **Stop** when a goal node is visited

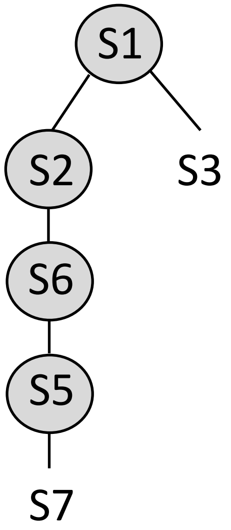




Depth-First Search

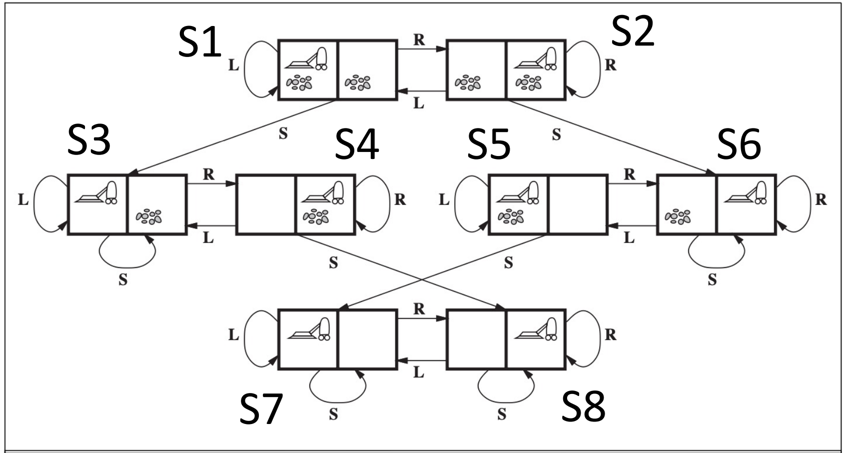
- Depth-First search algorithm:
  - **Expand** the deepest node in the frontier
  - **Do not add** children in the frontier if the node is already in the frontier or in the list of visited nodes (to avoid loopy paths)
  - **Stop** when a goal node is visited

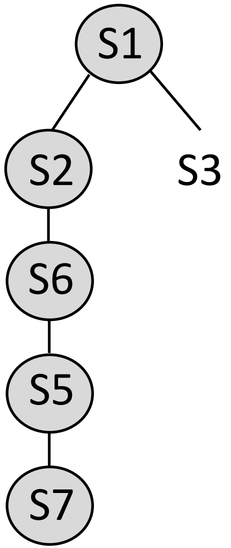




Depth-First Search

- Depth-First search algorithm:
  - **Expand** the deepest node in the frontier
  - **Do not add** children in the frontier if the node is already in the frontier or in the list of visited nodes (to avoid loopy paths)
  - **Stop** when a goal node is visited





Depth-First Search

- Solution:

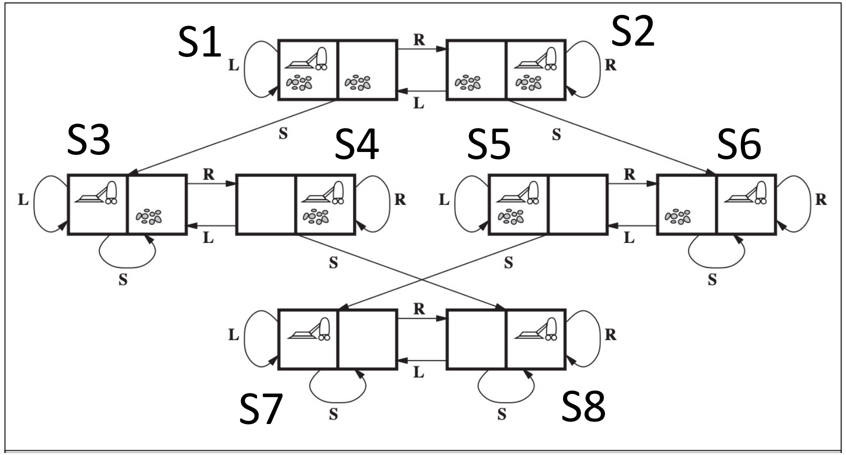
R, S, L, S

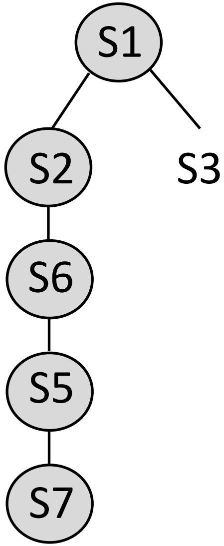
- Cost of the solution:

1 + 1 + 1 + 1 = 4

- Order of nodes visited

S1, S2, S6, S5, S7





DFS - Performance

Let us evaluate the performance of the depth-first search algorithm

- **Completeness**: DFS is not complete if the search space is infinite or if we do not check infinite loops; it is complete if the search space is finite
- **Optimality**: DFS is not optimal as it can expand a left subtree when the goal node is in the first level of the right subtree

DFS - Performance

Let us evaluate the performance of the depth-first search algorithm

- **Completeness**: DFS is not complete if the search space is infinite or if we do not check infinite loops; it is complete if the search space is finite
- **Optimality**: DFS is not optimal as it can expand a left subtree when the goal node is in the first level of the right subtree
- **Time complexity**: $O(b^m)$, as it depends on the maximum length of the path in the search space (in general m can be much larger than d)

DFS - Performance

Let us evaluate the performance of the depth-first search algorithm

- **Completeness**: DFS is not complete if the search space is infinite or if we do not check infinite loops; it is complete if the search space is finite
- **Optimality**: DFS is not optimal as it can expand a left subtree when the goal node is in the first level of the right subtree
- **Time complexity**: $O(b^m)$, as it depends on the maximum length of the path in the search space (in general m can be much larger than d)
- **Space complexity**: $O(b^m)$, as we store all the nodes from each path from the root node to the leaf node

Summary

- Depth-First Search is a search algorithm that expands the nodes in the frontier starting from the deepest, similar to a stack (LIFO)

- This algorithm is complete (for finite search space), but not optimal; also it has high time complexity and space complexity $O(b^m)$

Overview

- Asymptotic Analysis
- Search Problem Formulation
- Breadth-First Search
- Depth-First Search
- **Variations of Depth-First Search**

Depth-First Search - Variations

- Depth-First Search comes with several issues
  - Not optimal
  - High time complexity
  - High space complexity

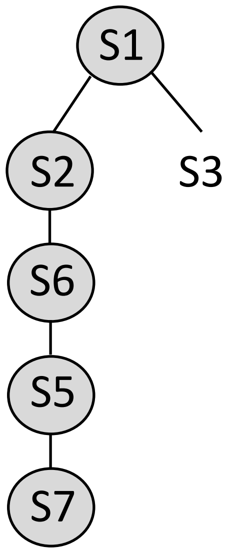
- DFS with less memory usage (saving space complexity)

- Depth-Limited Search

Depth-First Search – Less Memory Usage

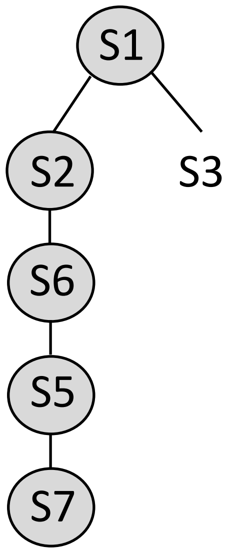
- Imagine we have a tree similar the one in the example

- Now, S7 is not a goal node and it has no children



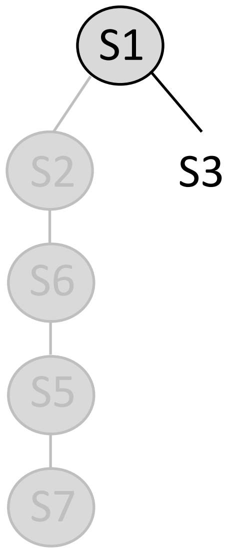
Depth-First Search – Less Memory Usage

- Imagine we have a tree similar the one in the example
- Now, S7 is not a goal node and it has no children
- The next step of the algorithm would be to expand S3



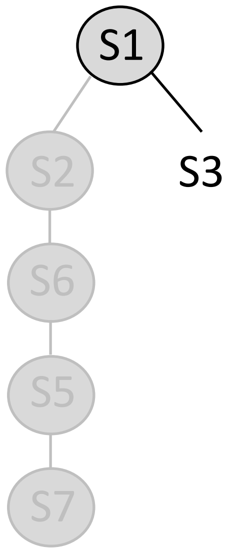
Depth-First Search – Less Memory Usage

- Imagine we have a tree similar the one in the example
- Now, S7 is not a goal node and it has no children
- The next step of the algorithm would be to expand S3
- Since we explored all the left subtree, we can remove it from memory



Depth-First Search – Less Memory Usage

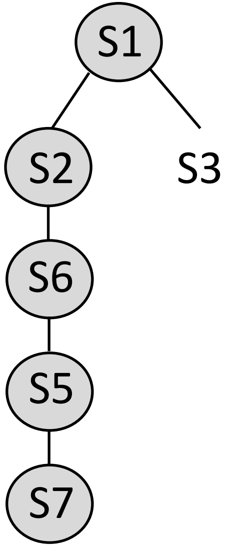
- This would reduce the space complexity to $O(bm)$
- We need to store a single path along with the siblings for each node on the path
- Recall that b is the branching factor and m is the maximum depth of the search tree



Depth-Limited Search

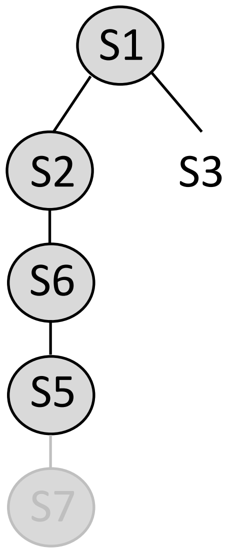
- The issue related to depth-first search in infinite state spaces can be mitigated by providing a depth limit ℓ

- This approach is called **depth-limited search**



Depth-Limited Search

- The issue related to depth-first search in infinite state spaces can be mitigated by providing a depth limit ℓ
- This approach is called **depth-limited search**
- For $\ell = 3$, we would have

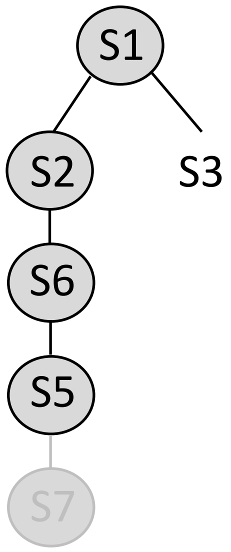


Depth-Limited Search

- This adds an additional source of incompleteness if we choose $\ell < d$, namely the shallowest goal is beyond the depth limit

- This approach is nonoptimal also in the case $\ell > d$

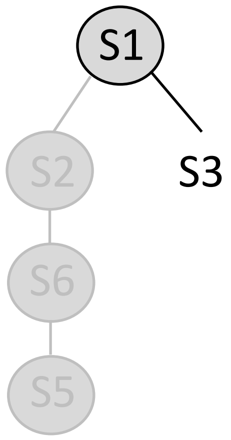
- Time complexity is $O(b^{\ell})$



Depth-Limited Search – Less Memory Usage

- As before, we can remove the explored paths from memory after we have reached the depth limit ℓ

- Space complexity is $O(b\ell)$



Comparing Uninformed Search Strategies

Criterion / Algorithm	Breadth-First	Depth-First	Depth-First (less memory)	Depth-Limited (less memory)
Completeness	Yes*	Yes***	Yes***	Yes if $\ell \geq d$
Optimality	Yes**	No	No	No
Time	$O(b^d)$	$O(b^m)$	$O(b^m)$	$O(b^\ell)$
Space	$O(b^d)$	$O(b^m)$	$O(bm)$	$O(b\ell)$

* If b is finite

** If the path cost is a nondecreasing function of the depth of the node (e.g., all actions have the same cost)

*** If the search space is finite (also, loopy paths are removed)

Summary

- Depth-First Search can be improved in terms of its time and space complexity through some modifications

- Depth-First Search with less memory usage only keeps in memory the current path and the siblings of the nodes

- Depth-Limited Search is another variation, where a depth limit is specified; this adds an additional source of incompleteness

Aims of the Session

You should now be able to:

- Describe asymptotic analysis and why it is important
- Explain the steps to formulate a search problem
- Apply and compare the performance of Breadth-First Search, Depth-First Search and its variations