Lecture 2: Problem Solving and (Uninformed) Search
CS 580 (001) - Spring 2016

Amarda Shehu

Department of Computer Science
George Mason University, Fairfax, VA, USA

January 27, 2016
1 Outline of Today’s Class

2 Problem-solving Agents

3 Problem Types

4 Problem Formulation

5 Example Problems

6 Elementary (Graph) Search Algorithms
   - Uninformed Search
     - Breadth-first Search (BFS)
     - Depth-first Search (DFS)
     - Depth-limited Search (DLS)
     - Iterative Deepening Search (IDS)
function SIMPLE-PROBLEM-SOLVING-AGENT( percept) returns an action

static: seq, an action sequence, initially empty
state, some description of the current world state
goal, a goal, initially null
problem, a problem formulation

state ← UPDATE-STATE(state, percept)
if seq is empty then
  goal ← FORMULATE-GOAL(state)
  problem ← FORMULATE-PROBLEM(state, goal)
  seq ← SEARCH(problem)
action ← RECOMMENDATION(seq, state)
seq ← REMAINDER(seq, state)
return action

Note: this is offline problem solving; solution executed “eyes closed.”

Online problem solving involves acting without complete knowledge.
Example: Romania

On holiday in Romania; currently in Arad.
Flight leaves tomorrow from Bucharest.

**Formulate goal:**
be in Bucharest

**Formulate problem:**
states: various cities
actions: drive between cities

**Find solution:**
sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest
Example: Romania
Problem Types

- **Fully-observable, Known, Deterministic** → single-state problem  
  Agent knows exactly which state it will be in; solution is a sequence of actions that can be executed *eyes closed*  
  **open loop**: no need to sense environment during execution

- **Non-observable** → conformant problem  
  Agent may have no idea where it is; solution (if any) is a sequence  
  Also known as multi-state problem: agent knows which states it might be in

- **Nondeterministic** and/or **Partially observable** → contingency problem  
  Percepts provide *new* information about current state  
  Solution is a contingent plan or a policy  
  Often **interleave** search, execution  
  plans contain conditional parts based on sensors

- **Unknown environment** → exploration problem ("online")  
  Agent must learn the effect of its actions
Example: Vacuum World

Single-state, start in #5.
Solution??

1
2
3
4
5
6
7
8
**Example: Vacuum World**

**Single-state**, start in #5.
Solution??

\[\text{Right, Suck}\]

**Murphy's Law:** Suck can dirty a clean carpet

Local sensing: dirt, location only.
Solution??

\[\text{Right, if dirt then Suck}\]
Example: Vacuum World

**Single-state**, start in #5.
Solution??

\[ \text{[Right, Suck]} \]

**Conformant**, start in
\{1, 2, 3, 4, 5, 6, 7, 8\}
**Example: Vacuum World**

**Single-state**, start in #5.  
Solution??

\[ \text{[Right, Suck]} \]

**Conformant**, start in 
\{1, 2, 3, 4, 5, 6, 7, 8\}  
e.g., **Right** goes to \{2, 4, 6, 8\}.  
Solution??
Example: Vacuum World

**Single-state**, start in #5.
Solution??

\[ \text{[Right, Suck]} \]

**Conformant**, start in \{1, 2, 3, 4, 5, 6, 7, 8\}
e.g., Right goes to \{2, 4, 6, 8\}.
Solution??

\[ \text{[Right, Suck, Left, Suck]} \]
Example: Vacuum World

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Single-state**, start in #5.
Solution??
[Right, Suck]

**Conformant**, start in
{1, 2, 3, 4, 5, 6, 7, 8}
e.g., Right goes to {2, 4, 6, 8}.
Solution??
[Right, Suck, Left, Suck]

**Contingency**, start in #5
Murphy’s Law: Suck can dirty a clean carpet
Local sensing: dirt, location only.
Example: Vacuum World

1. **Single-state**, start in #5.
Solution??

2. [Right, Suck]

3. **Conformant**, start in
{1, 2, 3, 4, 5, 6, 7, 8}
e.g., Right goes to {2, 4, 6, 8}.
Solution??

4. [Right, Suck, Left, Suck]

5. **Contingency**, start in #5
Murphy’s Law: Suck can dirty a clean carpet
Local sensing: dirt, location only.
Solution??

6. [Right, if dirt then Suck]
Example: Vacuum World

**Single-state**, start in #5.
Solution??

\[ \text{[Right, Suck]} \]

**Conformant**, start in
\{1, 2, 3, 4, 5, 6, 7, 8\}
e.g., Right goes to \{2, 4, 6, 8\}.
Solution??

\[ \text{[Right, Suck, Left, Suck]} \]

**Contingency**, start in #5
Murphy’s Law: Suck can dirty a clean carpet
Local sensing: dirt, location only.
Solution??

\[ \text{[Right, if dirt then Suck]} \]
Formulation of a Problem via Five Components

1. **Initial state(s):** the state(s) the agent starts in
Formulation of a Problem via Five Components

1. **Initial state(s)**: the state(s) the agent starts in

2. **Actions/operators**: given any state \( s \), \( \text{ACTION}(s) \) returns set of actions that can be executed from \( s \)
Formulation of a Problem via Five Components

1. **Initial state(s):** the state(s) the agent starts in

2. **Actions/operators:** given any state $s$, $\text{ACTION}(s)$ returns set of actions that can be executed from $s$

3. **Transition model:** maps state-action pairs to states; given a state $s$ and action $a$, $\text{RESULT}(s, a)$ returns the state that results from carrying out action $a$ on $s$
**Formulation of a Problem via Five Components**

1. **Initial state(s):** the state(s) the agent starts in

2. **Actions/operators:** given any state $s$, \( \text{ACTION}(s) \) returns set of actions that can be executed from $s$

3. **Transition model:** maps state-action pairs to states; given a state $s$ and action $a$, \( \text{RESULT}(s, a) \) returns the state that results from carrying out action $a$ on $s$

   1.-3. implicitly define state space: set of all states reachable from initial state and any sequence of actions.
Formulation of a Problem via Five Components

1. **Initial state(s):** the state(s) the agent starts in

2. **Actions/operators:** given any state \( s \), \( \text{ACTION}(s) \) returns set of actions that can be executed from \( s \)

3. **Transition model:** maps state-action pairs to states; given a state \( s \) and action \( a \), \( \text{RESULT}(s, a) \) returns the state that results from carrying out action \( a \) on \( s \)

   1.-3. implicitly define **state space:** set of all states reachable from initial state and any sequence of actions.

   encoded as a directed graph: nodes are states and edges are actions.
Formulation of a Problem via Five Components

1. **Initial state(s):** the state(s) the agent starts in

2. **Actions/operators:** given any state $s$, $\text{ACTION}(s)$ returns set of actions that can be executed from $s$

3. **Transition model:** maps state-action pairs to states; given a state $s$ and action $a$, $\text{RESULT}(s, a)$ returns the state that results from carrying out action $a$ on $s$

1.-3. implicitly define **state space:** set of all states reachable from initial state and any sequence of actions.

   encoded as a directed graph: nodes are states and edges are actions.

   what is a path in this graph?
**Formulation of a Problem via Five Components**

1. **Initial state(s)**: the state(s) the agent starts in

2. **Actions/operators**: given any state $s$, $\text{ACTION}(s)$ returns set of actions that can be executed from $s$

3. **Transition model**: maps state-action pairs to states; given a state $s$ and action $a$, $\text{RESULT}(s, a)$ returns the state that results from carrying out action $a$ on $s$

   1.-3. implicitly define **state space**: set of all states reachable from initial state and any sequence of actions.

   encoded as a directed graph: nodes are states and edges are actions.

   what is a path in this graph?

4. **Goal test**: determines whether a given state is a goal state defined explicitly or via a property
Formulation of a Problem via Five Components

1. **Initial state(s):** the state(s) the agent starts in

2. **Actions/operators:** given any state $s$, $\text{ACTION}(s)$ returns set of actions that can be executed from $s$

3. **Transition model:** maps state-action pairs to states; given a state $s$ and action $a$, $\text{RESULT}(s, a)$ returns the state that results from carrying out action $a$ on $s$

   1.-3. implicitly define **state space:** set of all states reachable from initial state and any sequence of actions.

   encoded as a directed graph: nodes are states and edges are actions.

   what is a path in this graph?

4. **Goal test:** determines whether a given state is a goal state defined explicitly or via a property

5. **Path cost:** computational cost of the execution of the path/plan is
Formulation of a Problem via Five Components

1. **Initial state(s):** the state(s) the agent starts in

2. **Actions/operators:** given any state \( s \), \( \text{ACTION}(s) \) returns set of actions that can be executed from \( s \)

3. **Transition model:** maps state-action pairs to states; given a state \( s \) and action \( a \), \( \text{RESULT}(s, a) \) returns the state that results from carrying out action \( a \) on \( s \)
   
   1.-3. implicitly define state space: set of all states reachable from initial state and any sequence of actions.
   
   encoded as a directed graph: nodes are states and edges are actions.
   
   what is a path in this graph?

4. **Goal test:** determines whether a given state is a goal state
   
   defined explicitly or via a property

5. **Path cost:** computational cost of the execution of the path/plan is
A problem is defined by five components:

1. **Initial state**  
   e.g., “In(Arad)”

2. **Actions**  
   e.g.  
   \[
   \text{ACTION(Arad)} = \{ \text{Arad} \rightarrow \text{Timisoara}, \text{Arad} \rightarrow \text{Sibiu}, \ldots, \text{Arad} \rightarrow \text{Zerind} \} 
   \]

3. **Transition model**  
   e.g.  
   \[
   \text{RESULT(Arad, Arad} \rightarrow \text{Zerind}) = \text{Zerind} 
   \]

4. **Goal test**, can be:  
   - **explicit**  
     e.g., “In(Bucharest)”  
   - **implicit**  
     e.g., \text{NoDirt}(s)

5. **Path cost** (additive)  
   e.g. sum of distances, number of actions executed, etc.  
   - \(c(x, a, y)\) is the step cost, assumed to be \(\geq 0\)

**Solution:**

A **solution** is a sequence of actions leading from the initial state to a goal state. 
the process of looking for a solution is called **search**
Abstraction: Selecting a State Space

Real world is absurdly complex

⇒ state space must be **abstracted** for problem solving

(Abstract) state = set of real states

(Abstract) action = complex combination of real actions

e.g., “Arad → Zerind” represents a complex set of possible routes, detours, rest stops, etc.

For guaranteed realizability, any real state “in Arad” must get to some real state “in Zerind”

(Abstract) solution =
set of real paths that are solutions in the real world

Each abstract action should be “easier” than the original problem!
- **State space graph**: A mathematical representation of a search problem.
- **Nodes** are (abstracted) world configurations.
- **Arcs/edges/links** represent successors (action results).
- **Goal test** is a set of goal nodes (maybe only one).
- In a state space graph, each state occurs only once!
- We can rarely build this full graph in memory (it's too big), but it's a useful idea.
Example: Vacuum World State Space Graph

**States:**
- Integer dirt and robot locations (ignore dirt amounts, etc.)

**Actions:**
- Left, Right, Suck, NoOp

**Transition Model:**
- \(((\text{A, dirt}), \text{Suck}) \rightarrow (\text{A, clean}), . . .)\)

**Goal Test:**
- No dirt

**Path Cost:**
- 1 per action (0 for NoOp)

---

Amarda Shehu (580)
Example: Vacuum World State Space Graph

**.states??:** integer dirt and robot locations (ignore dirt *amounts* etc.)
Example: Vacuum World State Space Graph

states??: integer dirt and robot locations (ignore dirt amounts etc.)

How many states?
Example: Vacuum World State Space Graph

- **states**: integer dirt and robot locations (ignore dirt amounts etc.)
- **actions**:
- **transition model**

```
(A, dirt) \rightarrow (A, clean)
```

- **goal test**: no dirt
- **path cost**: 1 per action (0 for NoOp)

How many states?
Example: Vacuum World State Space Graph

states: integer dirt and robot locations (ignore dirt amounts etc.)

actions: Left, Right, Suck, NoOp

How many states?
Example: Vacuum World State Space Graph

states??: integer dirt and robot locations (ignore dirt amounts etc.)

actions??: Left, Right, Suck, NoOp

transition model??:
Example: Vacuum World State Space Graph

- **states**??: integer dirt and robot locations (ignore dirt amounts etc.)
- **actions**??: *Left, Right, Suck, NoOp*
- **transition model**??: $([A, \text{dirt}], \text{Suck}) \rightarrow [A, \text{clean}], \ldots$
Example: Vacuum World State Space Graph

**states**: integer dirt and robot locations (ignore dirt amounts etc.)

**actions**: Left, Right, Suck, NoOp

**transition model**: ([A, dirt], Suck) → [A, clean], ...

How many states?

where is transition model in graph?
Example: Vacuum World State Space Graph

states??: integer dirt and robot locations (ignore dirt amounts etc.)

actions??: \textit{Left, Right, Suck, NoOp}

transition model??: \(([A, \text{dirt}], \text{Suck}) \rightarrow [A, \text{clean}], \ldots\)

goal test??:
Example: Vacuum World State Space Graph

states: integer dirt and robot locations (ignore dirt amounts etc.)
actions: Left, Right, Suck, NoOp
transition model: ([A, dirt], Suck) → [A, clean], ...
goal test: no dirt

How many states?
where is transition model in graph?
Example: Vacuum World State Space Graph

states??: integer dirt and robot locations (ignore dirt amounts etc.)

actions??: Left, Right, Suck, NoOp

transition model??: ([A, dirt], Suck) → [A, clean], …

goal test??: no dirt

path cost??:
Example: Vacuum World State Space Graph

- **States??**: integer dirt and robot locations (ignore dirt amounts etc.)
- **How many states??**
- **Actions??**: *Left*, *Right*, *Suck*, *NoOp*
- **Transition model??**: \([A, \text{dirt}], \text{Suck} \) → \([A, \text{clean}]\), ...
  - **Where is transition model in graph??**
- **Goal test??**: no dirt
- **Path cost??**: 1 per action (0 for *NoOp*)
Example: Vacuum World State Space Graph

states: \[ \text{integer dirt and robot locations (ignore dirt amounts etc.)} \]

actions: \( \text{Left, Right, Suck, NoOp} \)

transition model: \( ([\text{A}, \text{dirt}], \text{Suck}) \rightarrow [\text{A}, \text{clean}], \ldots \)  

where is transition model in graph?

goal test: \( \text{no dirt} \)

path cost: \( 1 \) per action (0 for \( \text{NoOp} \))
Example: The 8-puzzle

The 8-puzzle is a classic problem in computer science and artificial intelligence. It consists of a 3x3 grid with eight numbered tiles and one blank space. The goal is to rearrange the tiles from a given start state to a specified goal state by sliding the tiles into the blank space. There are 8! = 40,320 possible states, but only half are reachable due to the movement constraints.

States:
- Start State:
  - 7 2 4
  - 5 6
  - 8 3 1

- Goal State:
  - 1 2 3
  - 4 5 6
  - 7 8

The number of states can be calculated using the formula for permutations of a restricted set, which in this case is $\frac{8!}{2} = 18,144$. The actions are based on the blank space's movement: Left, Right, Up, Down. The transition model is defined as given a state and an action, it returns the resulting state. The goal test checks if the current state is the goal state. The path cost is unity per move, indicating the cost of each move is 1. Note that the optimal solution of the n-Puzzle family is NP-hard.
Example: The 8-puzzle

- **states**: integer locations of tiles (ignore intermediate positions)

---

Example Problems

Amarda Shehu (580)
Example: The 8-puzzle

states??: integer locations of tiles (ignore intermediate positions)

How many states?
Example: The 8-puzzle

**states**: integer locations of tiles (ignore intermediate positions)

**actions**: blank space "moves" Left, Right, Up, Down

**transition model**: Given state and action, returns resulting state

**goal test**: = goal state (given)

**path cost**: 1 per move

[Note: optimal solution of n-Puzzle family is NP-hard!]

Amarda Shehu (580)

Example Problems

How many states?
Example: The 8-puzzle

states: integer locations of tiles (ignore intermediate positions)
actions: blank space “moves” Left, Right, Up, Down

How many states?
Example: The 8-puzzle

states: integer locations of tiles (ignore intermediate positions)
actions: blank space “moves” Left, Right, Up, Down
transition model:

Start State

Goal State

How many states?
Example: The 8-puzzle

states??: integer locations of tiles (ignore intermediate positions)
actions??: blank space “moves” Left, Right, Up, Down
transition model??: Given state and action, returns resulting state

Start State

Goal State

How many states?

[Image of 8-puzzle states and goal state]
Example: The 8-puzzle

- **states**: integer locations of tiles (ignore intermediate positions)
- **actions**: blank space “moves” Left, Right, Up, Down
- **transition model**: Given state and action, returns resulting state
- **goal test**: How many states?

Start State

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

Goal State

```
1 2 3
4 5 6
7 8
```
Example: The 8-puzzle

states: integer locations of tiles (ignore intermediate positions)

actions: blank space “moves” Left, Right, Up, Down

transition model: Given state and action, returns resulting state

goal test: = goal state (given)

How many states?
Example: The 8-puzzle

**States**: integer locations of tiles (ignore intermediate positions)

**Actions**: blank space “moves” Left, Right, Up, Down

**Transition Model**: Given state and action, returns resulting state

**Goal Test**: = goal state (given)

**Path Cost**: 1 per move

---

[Note: optimal solution of $n$-Puzzle family is NP-hard!]
Example: The 8-puzzle

- **states??:** integer locations of tiles (ignore intermediate positions)
- **actions??:** blank space “moves” Left, Right, Up, Down
- **transition model??:** Given state and action, returns resulting state
- **goal test??:** = goal state (given)
- **path cost??:** 1 per move

[Note: optimal solution of $n$-Puzzle family is NP-hard!]

---

**Start State**

- 7
- 2
- 4
- 5
- 6
- 8

**Goal State**

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
Example: The 8-puzzle

- **states**: integer locations of tiles (ignore intermediate positions)
- **actions**: blank space “moves” Left, Right, Up, Down
- **transition model**: Given state and action, returns resulting state
- **goal test**: = goal state (given)
- **path cost**: 1 per move

[Note: optimal solution of $n$-Puzzle family is NP-hard!]

---

**Start State**

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

**Goal State**

```
1  2  3
4  5  6
7  8
```
Example: Robotic Assembly

states??:

Example Problems

Amarda Shehu (580)
Example: Robotic Assembly

states: real-valued coordinates of robot joint angles + parts of the object to be assembled
Example: Robotic Assembly

- **states**: real-valued coordinates of robot joint angles + parts of the object to be assembled
- **actions**: continuous motions of robot joints
- **transition model**: state + action yields new state
- **goal test**: complete assembly with no robot included!
- **path cost**: time to execute
Example: Robotic Assembly

- **states**: real-valued coordinates of robot joint angles + parts of the object to be assembled
- **actions**: continuous motions of robot joints

---

Amarda Shehu (580)  
Example Problems  
14
Example: Robotic Assembly

states: real-valued coordinates of robot joint angles + parts of the object to be assembled
actions: continuous motions of robot joints
transition model:
Example: Robotic Assembly

states?: real-valued coordinates of robot joint angles + parts of the object to be assembled
actions?: continuous motions of robot joints
transition model?: state + action yields new state
Example: Robotic Assembly

states: real-valued coordinates of robot joint angles + parts of the object to be assembled

actions: continuous motions of robot joints

transition model: state + action yields new state

goal test:
Example: Robotic Assembly

states: real-valued coordinates of robot joint angles + parts of the object to be assembled
actions: continuous motions of robot joints
transition model: state+action yields new state
goal test: complete assembly with no robot included!
Example: Robotic Assembly

states:
real-valued coordinates of robot joint angles + parts of the object to be assembled

actions:
continuous motions of robot joints

transition model:
state + action yields new state

goal test:
complete assembly with no robot included!

path cost:
Example: Robotic Assembly

- states**: real-valued coordinates of robot joint angles + parts of the object to be assembled
- actions**: continuous motions of robot joints
- transition model**: state + action yields new state
- goal test**: complete assembly with no robot included!
- path cost**: time to execute
**Example: Robotic Assembly**

- **states**: real-valued coordinates of robot joint angles + parts of the object to be assembled
- **actions**: continuous motions of robot joints
- **transition model**: state + action yields new state
- **goal test**: complete assembly with no robot included!
- **path cost**: time to execute
The vacuum cleaner problem, 8-puzzle (block sliding), 8-queens, and others are examples of toy, route-finding problems.

Real-world route-finding problems can be found in robot navigation, manipulation, assembly, airline travel web-planning, and more.

Tour-finding problems are slightly different: “visit every city at least once, starting and ending in Bucharest.”

Traveling salesperson problem (TSP): find shortest tour that visits each city exactly once, NP-hard.

Other related, complex problems: packing, scheduling, VLSI layout, protein folding, protein design.
Choosing states and actions:

- **abstraction**: remove unnecessary information from representation; makes it cheaper to find a solution

Searching for Solutions:

- **operators expand a state**: generate new states from present ones
- **fringe or frontier**: discovered states to be expanded
- **search strategy**: tells which state in fringe set to expand next

Measuring Performance:

- does it find a solution?
- what is the search cost?
- what is the total cost (path cost + search cost)
A Search Tree:

A “what if” tree of plans and their outcomes

The start state is the root node

Children correspond to successors

Nodes show states, but correspond to PLANS that achieve those states

For most problems, we can never actually build the whole tree
State Space Graphs vs. Search Trees

We construct both on demand and we construct as little as possible.

State Space Graph

Search Tree

We construct both on demand and we construct as little as possible.
Consider this 4-state space graph:

Lots of repeated structure in the search tree!
Repeated States

Failure to detect repeated states can turn a linear problem into an exponential one!

Repeated structure can be easily avoided:
Repeated States

Failure to detect repeated states can turn a linear problem into an exponential one!

Repeated structure can be easily avoided: How?
Repeated States

Failure to detect repeated states can turn a linear problem into an exponential one!

Repeated structure can be easily avoided: How?
function **Graph-Search**\( (\text{problem}, \text{fringe}) \) **returns** a solution, or failure

\[
closed \leftarrow \text{an empty set}
\]

\[
\text{fringe} \leftarrow \text{Insert} (\text{Make-Node}(\text{Initial-State}[\text{problem}]), \text{fringe})
\]

**loop do**

\[
\text{if fringe is empty then return failure}
\]

\[
\text{node} \leftarrow \text{Remove-Front}(\text{fringe})
\]

\[
\text{if Goal-Test}(\text{problem}, \text{State}[\text{node}]) \text{ then return node}
\]

\[
\text{if State}[\text{node}] \text{ is not in closed then}
\]

\[
\text{add State}[\text{node}] \text{ to closed}
\]

\[
\text{fringe} \leftarrow \text{InsertAll} (\text{Expand}(\text{node}, \text{problem}), \text{fringe})
\]

**end**
Searching with a Search Tree

**Basic idea:**

Expand out potential plans (tree nodes)

Maintain a *fringe* of partial plans under consideration

Try to expand as few tree nodes as possible *(Why?)*
Searching with a Search Tree

Basic idea:

- Expand out potential plans (tree nodes)
- Maintain a fringe of partial plans under consideration
- Try to expand as few tree nodes as possible (Why?)
Basic idea:
offline, simulated exploration of state space
by generating successors of already-explored states
(a.k.a. expanding states)

function Tree-Search( problem, strategy) returns a solution, or failure
initialize the search tree using the initial state of problem
loop do
  if there are no candidates for expansion then return failure
  choose a leaf node for expansion according to strategy
  if the node contains a goal state then return the corresponding solution
  else expand the node and add the resulting nodes to the search tree
end
Fundamental to Graph Search/Traversal Algorithms:

- Successor function: generate successors/neighbors and distinguish a goal state from a non-goal state.

**Completeness** Goal should not be missed if a path exists.

**Efficiency** No edge should be traversed more than twice.
Tree Search Example

```
Arad
  /   \
Sibiu
  /   \  /   \
Fagaras Oradea Rimnicu Vilcea
  /       \
Arad
  /   \
Zerind
  /   \
Timisoara
  /   \
Lugoj
  /   \
Arad
  /   \
Oradea
```
Tree Search Example
Tree Search Example

```
Arad
  /   \
Sibiu /     \
|    |     |
Arad | Fagaras | Oradea
     |        |      
     |        |      
     Himnicu Vilcea
```

```
Arad
  /   \
Timisoara
  /     \
Arad | Lugoj
     |      |
     |      |
     Arad | Oradea
```

```
Arad
  /   \
Zerind
  /     \
Arad | Oradea
     |      |
     |      |
     Arad | Oradea
```
A state is a (representation of) a physical configuration
A node is a data structure constituting part of a search tree

includes parent, children, depth, path cost \( g(x) \)
States do not have parents, children, depth, or path cost!

The `Expand` function creates new nodes, filling in the various fields and using the `SuccessorFn` of the problem to create the corresponding states.
Important insight:
- Any search algorithm constructs a tree, adding to it vertices from state-space graph $G$ in some order.
- $G = (V, E)$ — look at it as split in two: set $S$ on one side and $V - S$ on the other.
- Search proceeds as vertices are taken from $V - S$ and added to $S$.
- Search ends when $V - S$ is empty or goal found.
- First vertex to be taken from $V - S$ and added to $S$?
- Next vertex? (... expansion ...)
- Where to keep track of these vertices? (... fringe/frontier ...)

Important ideas:
- Fringe (frontier into $V - S$/border between $S$ and $V - S$).
- Expansion (neighbor generation so can add to fringe).
- Exploration strategy (what order to grow $S$?)

Main question:
- Which fringe/frontier nodes to explore/expand next?
- Strategy distinguishes search algorithms from one another.
function Tree-Search( problem, fringe) returns a solution, or failure
fringe ← Insert(Make-Node(Initial-State[problem]), fringe)
loop do
  if fringe is empty then return failure
  node ← Remove-Front(fringe)
  if Goal-Test(problem, State(node)) then return node
  fringe ← InsertAll(Expand(node, problem), fringe)

function Expand(node, problem) returns a set of nodes
  successors ← the empty set
  for each action, result in Successor-Fn(problem, State[node]) do
    s ← a new Node
    Parent-Node[s] ← node; Action[s] ← action; State[s] ← result
    Path-Cost[s] ← Path-Cost[node] + Step-Cost(State[node], action, result)
    Depth[s] ← Depth[node] + 1
    add s to successors
  return successors
A strategy is defined by picking the **order of node expansion**

Strategies are evaluated along the following dimensions:

- **completeness**—does it always find a solution if one exists?
- **time complexity**—number of nodes generated/expanded
- **space complexity**—maximum number of nodes in memory
- **optimality**—does it always find a least-cost solution?

Time and space complexity are measured in terms of:

- $b$—maximum branching factor of the search tree
- $d$—depth of the least-cost solution
- $m$—maximum depth of the state space (may be $\infty$)
Characteristics of Uninformed Graph Search/Traversal:

- There is no additional information about states/vertices beyond what is provided in the problem definition.
- All that the search does is generate successors/neighbors and distinguish a **goal** state from a **non-goal state**.

The systematic search “lays out” all paths from initial vertex; it traverses the search tree of the graph.
Uninformed Graph Search

F: search data structure (fringe)
parent array: stores “edge comes from” to record visited states

1: F.insert(v)
2: parent[v] ← true
3: while not F.isEmpty do
4:  u ← F.extract()
5:  if isGoal(u) then
6:     return true
7:  for each v in outEdges(u) do
8:     if no parent[v] then
9:        F.insert(v)
10:    parent[v] ← u

Figure : Graph

Figure : Search Tree of Graph
Uninformed Search Algorithms

- Breadth-first Search (BFS)
- Depth-first Search (DFS)
- Depth-limited search (DLS)
- Iterative Deepening Search (IDS)
Breadth-first Search (BFS)
**Breadth-first Search (BFS)**

**Strategy:** Expand shallowest unexpanded node

**Implementation:**
fringe = first-in first-out (FIFO), i.e., unvisited successors go at end
F is a queue
Breadth-first Search (BFS)

**Strategy:** Expand shallowest unexpanded node

**Implementation:**
fringe = first-in first-out (FIFO), i.e., unvisited successors go at end
F is a queue
Breadth-first Search (BFS)

**Strategy:** Expand shallowest unexpanded node

**Implementation:**
fringe = first-in first-out (FIFO), i.e., unvisited successors go at end
F is a queue
**Strategy:** Expand shallowest unexpanded node

**Implementation:**
fringe = first-in first-out (FIFO), i.e., unvisited successors go at end
F is a queue
Breadth-first Search (BFS)

**F**: search data structure (fringe)

**F is a queue (FIFO) in BFS!**

parent array: stores “edge comes from” to record visited states

```plaintext
1: F.insert(v)
2: parent[v] ← true
3: while not F.isEmpty do
4:   u ← F.extract()
5:   if isGoal(u) then
6:     return true
7:   for each v in outEdges(u) do
8:     if no parent[v] then
9:       F.insert(v)
10:      parent[v] ← u
```

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

$O(|V| + |E|)$

What about in terms of $b$ and $m$?

Amarda Shehu (580) Elementary (Graph) Search Algorithms
Breadth-first Search (BFS)

F: search data structure (fringe)

**F is a queue (FIFO) in BFS!**

parent array: stores “edge comes from” to record visited states

1. F.insert(v)
2. parent[v] ← true
3. **while** not F.isEmpty do
4.   u ← F.extract()
5.   **if** isGoal(u) **then**
6.     **return** true
7.   **for** each v in outEdges(u) do
8.     **if** no parent[v] **then**
9.       F.insert(v)
10.      parent[v] ← u

Running Time?

Let V and E be vertices and edges in search tree
Breadth-first Search (BFS)

F: search data structure (fringe)

**F is a queue (FIFO) in BFS!**

parent array: stores “edge comes from” to record visited states

1. F.insert(v)
2. parent[v] ← true
3. while not F.isEmpty do
4.   u ← F.extract()
5.   if isGoal(u) then
6.     return true
7.   for each v in outEdges(u) do
8.     if no parent[v] then
9.       F.insert(v)
10.      parent[v] ← u

**Running Time?**

Let V and E be vertices and edges in search tree

$O(|V| + |E|)$
Breadth-first Search (BFS)

**F**: search data structure (fringe)

**F is a queue (FIFO) in BFS!**

parent array: stores “edge comes from” to record visited states

```plaintext
1: F.insert(v)
2: parent[v] ← true
3: while not F.isEmpty do
4:   u ← F.extract()
5:   if isGoal(u) then
6:     return true
7:   for each v in outEdges(u) do
8:     if no parent[v] then
9:       F.insert(v)
10:      parent[v] ← u
```

**Running Time?**

Let $V$ and $E$ be vertices and edges in search tree

$O(|V| + |E|)$

What about in terms of $b$ and $m$?
Breadth-first Search (BFS)

F: search data structure (fringe)

**F is a queue (FIFO) in BFS!**

parent array: stores “edge comes from” to record visited states

1. F.insert(v)
2. parent[v] ← true
3. while not F.isEmpty do
4.   u ← F.extract()
5.   if isGoal(u) then
6.     return true
7.   for each v in outEdges(u) do
8.     if no parent[v] then
9.       F.insert(v)
10.      parent[v] ← u

Running Time?
Let V and E be vertices and edges in search tree
\[ O(|V| + |E|) \]

What about in terms of \( b \) and \( m \)?
Properties of Breadth-first Search (BFS)

Complete:
Yes (if \( b \) is finite)

Time:
\[ 1 + b + b^2 + b^3 + \ldots + b^d + b^{(b^d - 1)} = O(b^d + 1), \text{i.e., exp. in } d \]

Space:
\[ O(b^d + 1) \] (keeps every node in memory)

Optimal:
Yes (if cost = 1 per step); not optimal in general

Space is the big problem; can easily generate nodes at 100MB/sec so 24hrs = 8640GB.
Properties of Breadth-first Search (BFS)

Complete? Yes (if $b$ is finite)

Time

$$1 + b + b^2 + b^3 + \ldots + b^d = O(b^d + 1),$$

i.e., exponential in $d$.

Space

$$O(b^d + 1)$$

(keeps every node in memory).

Optimal?

Yes (if cost = 1 per step); not optimal in general.

Space is the big problem; can easily generate nodes at 100MB/sec

so 24hrs = 8640GB.
Properties of Breadth-first Search (BFS)

Complete: Yes (if \( b \) is finite)

Time: \( 1 + b + b^2 + b^3 + \ldots + b^d + b^{(b^d - 1)} = O(b^{d+1}) \), i.e., exponential in \( d \)

Space: \( O(b^{d+1}) \) (keeps every node in memory)

Optimal: Yes (if cost = 1 per step); not optimal in general

Space is the big problem; can easily generate nodes at 100MB/sec so 24hrs = 8640GB.
Properties of Breadth-first Search (BFS)

**Complete??** Yes (if $b$ is finite)

**Time??** $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp. in $d$. 

**Space??** $O(b^{d+1})$ (keeps every node in memory)
Properties of Breadth-first Search (BFS)

Complete? Yes (if $b$ is finite)

Time? $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp. in $d$

Space?
Properties of Breadth-first Search (BFS)

- **Complete?** Yes (if \( b \) is finite)

- **Time?** \( 1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1}), \) i.e., exp. in \( d \)

- **Space?** \( O(b^{d+1}) \) (keeps every node in memory)
Properties of Breadth-first Search (BFS)

Complete?? Yes (if \( b \) is finite)

Time?? \( 1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1}) \), i.e., exp. in \( d \)

Space?? \( O(b^{d+1}) \) (keeps every node in memory)

Optimal??
Properties of Breadth-first Search (BFS)

Complete?? Yes (if \( b \) is finite)

Time?? \( 1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1}), \) i.e., exp. in \( d \)

Space?? \( O(b^{d+1}) \) (keeps every node in memory)

Optimal?? Yes (if cost = 1 per step); not optimal in general
Properties of Breadth-first Search (BFS)

Complete?? Yes (if $b$ is finite)

Time?? $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp. in $d$

Space?? $O(b^{d+1})$ (keeps every node in memory)

Optimal?? Yes (if cost = 1 per step); not optimal in general

Space
Properties of Breadth-first Search (BFS)

Complete?? Yes (if $b$ is finite)

Time?? $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp. in $d$

Space?? $O(b^{d+1})$ (keeps every node in memory)

Optimal?? Yes (if cost = 1 per step); not optimal in general

Space is the big problem; can easily generate nodes at 100MB/sec

so 24hrs = 8640GB.
Properties of Breadth-first Search (BFS)

Complete?? Yes (if \( b \) is finite)

Time?? \( 1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1}) \), i.e., exp. in \( d \)

Space?? \( O(b^{d+1}) \) (keeps every node in memory)

Optimal?? Yes (if cost = 1 per step); not optimal in general

**Space** is the big problem; can easily generate nodes at 100MB/sec

so 24hrs = 8640GB.
BFS Summary

Basic Behavior:
- Expands all nodes at depth $d$ before those at depth $d + 1$
- The sequence is root, then children, then grandchildren in the search tree.

Problems:
- If the path cost is a non-decreasing function of the depth of the goal node, then BFS is optimal (uniform cost search a generalization)
BFS Summary

Basic Behavior:

- Expands all nodes at depth $d$ before those at depth $d + 1$
- The sequence is root, then children, then grandchildren in the search tree.

Problems:

- If the path cost is a non-decreasing function of the depth of the goal node, then BFS is optimal (uniform cost search a generalization)
- A graph with no weights can be considered to have edges of weight 1. In this case, BFS is optimal.
BFS Summary

Basic Behavior:
- Expands all nodes at depth \( d \) before those at depth \( d + 1 \)
- The sequence is root, then children, then grandchildren in the search tree.

Problems:
- If the path cost is a non-decreasing function of the depth of the goal node, then BFS is optimal (uniform cost search a generalization)
- A graph with no weights can be considered to have edges of weight 1. In this case, BFS is optimal.
- BFS will find shallowest goal after expanding all shallower nodes (if branching factor is finite). Hence, BFS is complete.
BFS Summary

Basic Behavior:
- Expands all nodes at depth $d$ before those at depth $d + 1$
- The sequence is root, then children, then grandchildren in the search tree.

Problems:
- If the path cost is a non-decreasing function of the depth of the goal node, then BFS is optimal (uniform cost search a generalization)
- A graph with no weights can be considered to have edges of weight 1. In this case, BFS is optimal.
- BFS will find shallowest goal after expanding all shallower nodes (if branching factor is finite). Hence, BFS is complete.
- BFS is not very popular because time and space complexity are exponential: $O(b^{d+1})$ and $O(b^{d+1})$, respectively.
- Memory requirements of BFS are a bigger problem.
Basic Behavior:

- Expands all nodes at depth $d$ before those at depth $d + 1$
- The sequence is root, then children, then grandchildren in the search tree.

Problems:

- If the path cost is a non-decreasing function of the depth of the goal node, then BFS is optimal (uniform cost search a generalization)
- A graph with no weights can be considered to have edges of weight 1. In this case, BFS is optimal.
- BFS will find shallowest goal after expanding all shallower nodes (if branching factor is finite). Hence, BFS is complete.
- BFS is not very popular because time and space complexity are exponential: $O(b^{d+1})$ and $O(b^{d+1})$, respectively.
- Memory requirements of BFS are a bigger problem.
Depth-first Search (DFS)
Depth-first Search (DFS)
**Depth-first Search (DFS)**

**Strategy:** Expand deepest unexpanded node

**Implementation:**
fringe = last-in first-out (LIFO), i.e., unvisited successors at front
F is a stack
**Strategy:** Expand deepest unexpanded node

**Implementation:**
fringe = last-in first-out (LIFO), i.e., unvisited successors at front
F is a stack
**Strategy:** Expand deepest unexpanded node

**Implementation:**
fringe = last-in first-out (LIFO), i.e., unvisited successors at front
F is a stack
**Strategy:** Expand deepest unexpanded node

**Implementation:**
fringe = last-in first-out (LIFO), i.e., unvisited successors at front  
F is a stack
**Strategy:** Expand deepest unexpanded node

**Implementation:**
\( \text{fringe} = \text{last-in first-out (LIFO)}, \text{i.e., unvisited successors at front} \)

\( F \) is a stack
Depth-first Search (DFS)

**Strategy:** Expand deepest unexpanded node

**Implementation:**
fringe = last-in first-out (LIFO), i.e., unvisited successors at front
F is a stack
**Strategy:** Expand deepest unexpanded node

**Implementation:**
fringe = last-in first-out (LIFO), i.e., unvisited successors at front
F is a stack
Depth-first Search (DFS)

**Strategy:** Expand deepest unexpanded node

**Implementation:**
fringe = last-in first-out (LIFO), i.e., unvisited successors at front
F is a stack
**Depth-first Search (DFS)**

**Strategy:** Expand deepest unexpanded node

**Implementation:**
- fringe = last-in first-out (LIFO), i.e., unvisited successors at front
- F is a stack
**Strategy:** Expand deepest unexpanded node

**Implementation:**
fringe = last-in first-out (LIFO), i.e., unvisited successors at front
F is a stack
**Strategy:** Expand deepest unexpanded node

**Implementation:**
fringe = last-in first-out (LIFO), i.e., unvisited successors at front
F is a stack
**Depth-first Search (DFS)**

**Strategy:** Expand deepest unexpanded node

**Implementation:**
- **fringe** = last-in first-out (LIFO), i.e., unvisited successors at front
- F is a stack
**Depth-first Search (DFS)**

F: search data structure (**fringe**)

**F is a stack (LIFO) in DFS!**

parent array: stores “edge comes from” to record visited states

1. F.insert(v)
2. parent[v] ← true
3. while not F.isEmpty do
4.   u ← F.extract()
5.   if isGoal(u) then
6.     return true
7.   for each v in outEdges(u) do
8.     if no parent[v] then
9.     F.insert(v)
10.    parent[v] ← u

**Running Time?**

Let V and E be vertices and edges in search tree

$O(|V| + |E|)$

What about in terms of $b$ and $m$?
Depth-first Search (DFS)

F: search data structure (fringe)

**F is a stack (LIFO) in DFS!**

parent array: stores “edge comes from” to record visited states

1. F.insert(v)
2. parent[v] ← true
3. while not F.isEmpty do
4.   u ← F.extract()
5.   if isGoal(u) then
6.     return true
7.   for each v in outEdges(u) do
8.     if no parent[v] then
9.       F.insert(v)
10.      parent[v] ← u

Running Time?
Let V and E be vertices and edges in search tree

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?

Let $V$ and $E$ be vertices and edges in search tree

Running Time?
Depth-first Search (DFS)

F: search data structure (fringe)

F is a stack (LIFO) in DFS!
parent array: stores “edge comes from” to record visited states

1: F.insert(v)
2: parent[v] ← true
3: while not F.isEmpty do
4: u ← F.extract()
5: if isGoal(u) then
6: return true
7: for each v in outEdges(u) do
8: if no parent[v] then
9: F.insert(v)
10: parent[v] ← u

Running Time?
Let V and E be vertices and edges in search tree
$O(|V| + |E|)$
Depth-first Search (DFS)

F: search data structure (fringe)

F is a stack (LIFO) in DFS!

parent array: stores “edge comes from” to record visited states

1: F.insert(v)
2: parent[v] ← true
3: while not F.isEmpty do
4: u ← F.extract()
5: if isGoal(u) then
6: return true
7: for each v in outEdges(u) do
8: if no parent[v] then
9: F.insert(v)
10: parent[v] ← u

Running Time?

Let V and E be vertices and edges in search tree

$O(|V| + |E|)$

What about in terms of $b$ and $m$?
Depth-first Search (DFS)

F: search data structure (fringe)

**F is a stack (LIFO) in DFS!**

parent array: stores “edge comes from” to record visited states

1: F.insert(v)
2: parent[v] ← true
3: while not F.isEmpty do
4:   u ← F.extract()
5:   if isGoal(u) then
6:     return true
7:   for each v in outEdges(u) do
8:     if no parent[v] then
9:       F.insert(v)
10:      parent[v] ← u

**Running Time?**

Let V and E be vertices and edges in search tree

$O(|V| + |E|)$

What about in terms of $b$ and $m$?
Properties of Depth-first Search (DFS)

Complete?

Time: \(O(b^m)\): terrible if \(m\) is much larger than \(d\) but if solutions are dense, may be much faster than BFS

Space: \(O(bm)\), i.e., linear space!

Optimal: No
Why?
Properties of Depth-first Search (DFS)

**Complete??** No: fails in infinite-depth spaces, spaces with loops

Modify to avoid repeated states along path

⇒ complete in finite spaces
Properties of Depth-first Search (DFS)

**Complete?? No:** fails in infinite-depth spaces, spaces with loops

- Modify to avoid repeated states along path
  ⇒ complete in finite spaces

**Time??**

- $O(b^m)$: terrible if $m$ is much larger than $d$ but if solutions are dense, may be much faster than BFS

**Space??**

- $O(bm)$, i.e., linear space!
Properties of Depth-first Search (DFS)

Complete?? No: fails in infinite-depth spaces, spaces with loops
  Modify to avoid repeated states along path
  ⇒ complete in finite spaces

Time?? $O(b^m)$: terrible if $m$ is much larger than $d$
  but if solutions are dense, may be much faster than BFS
Properties of Depth-first Search (DFS)

Complete?? No: fails in infinite-depth spaces, spaces with loops
  Modify to avoid repeated states along path
  ⇒ complete in finite spaces

Time?? $O(b^m)$: terrible if $m$ is much larger than $d$
  but if solutions are dense, may be much faster than BFS

Space??
Properties of Depth-first Search (DFS)

Complete?? No: fails in infinite-depth spaces, spaces with loops
   Modify to avoid repeated states along path
   ⇒ complete in finite spaces

Time?? $O(b^m)$: terrible if $m$ is much larger than $d$
   but if solutions are dense, may be much faster than BFS

Space?? $O(bm)$, i.e., linear space!
Properties of Depth-first Search (DFS)

Complete?? No: fails in infinite-depth spaces, spaces with loops
  Modify to avoid repeated states along path
  ⇒ complete in finite spaces

Time?? \( O(b^m) \): terrible if \( m \) is much larger than \( d \)
  but if solutions are dense, may be much faster than BFS

Space?? \( O(bm) \), i.e., linear space!

Optimal??
Properties of Depth-first Search (DFS)

Complete? No: fails in infinite-depth spaces, spaces with loops
  Modify to avoid repeated states along path
  ⇒ complete in finite spaces

Time? \( O(b^m) \): terrible if \( m \) is much larger than \( d \)
  but if solutions are dense, may be much faster than BFS

Space? \( O(bm) \), i.e., linear space!

Optimal? No
Properties of Depth-first Search (DFS)

Complete? No: fails in infinite-depth spaces, spaces with loops
  Modify to avoid repeated states along path
  ⇒ complete in finite spaces

Time? $O(b^m)$: terrible if $m$ is much larger than $d$
  but if solutions are dense, may be much faster than BFS

Space? $O(bm)$, i.e., linear space!

Optimal? No

Why?
Properties of Depth-first Search (DFS)

Complete?? No: fails in infinite-depth spaces, spaces with loops
  Modify to avoid repeated states along path
  ⇒ complete in finite spaces

Time?? $O(b^m)$: terrible if $m$ is much larger than $d$
  but if solutions are dense, may be much faster than BFS

Space?? $O(bm)$, i.e., linear space!

Optimal?? No | Why?
DFS Summary

Basic Behavior:

- Expands the deepest node in the tree
- Backtracks when reaches a non-goal node with no descendants

Problems:

- Make a wrong choice and can go down along an infinite path even though the solution may be very close to initial vertex
- Hence, DFS is not optimal

Let $b$ be the maximum number of successors of any node (known as branching factor), $d$ be depth of shallowest goal, and $m$ be maximum length of any path in the search tree.

Time complexity is $O(b^m)$ and space complexity is $O(b \cdot m)$. 
DFS Summary

Basic Behavior:
- Expands the deepest node in the tree
- Backtracks when reaches a non-goal node with no descendants

Problems:
- Make a wrong choice and can go down along an infinite path even though the solution may be very close to initial vertex
- Hence, DFS is not optimal
  - If subtree is of unbounded depth and contains no solutions, DFS will never terminate.

Let $b$ be the maximum number of successors of any node (known as branching factor), $d$ be depth of shallowest goal, and $m$ be maximum length of any path in the search tree.

Time complexity is $O(b^m)$ and space complexity is $O(b \cdot m)$. 

Amarda Shehu (580)
DFS Summary

Basic Behavior:
- Expands the deepest node in the tree
- Backtracks when reaches a non-goal node with no descendants

Problems:
- Make a wrong choice and can go down along an infinite path even though the solution may be very close to initial vertex
- Hence, DFS is not optimal
- If subtree is of unbounded depth and contains no solutions, DFS will never terminate.
- Hence, DFS is not complete
DFS Summary

Basic Behavior:
- Expands the deepest node in the tree
- Backtracks when reaches a non-goal node with no descendants

Problems:
- Make a wrong choice and can go down along an infinite path even though the solution may be very close to initial vertex
- Hence, DFS is not optimal
- If subtree is of unbounded depth and contains no solutions, DFS will never terminate.
- Hence, DFS is not complete

Let $b$ be the maximum number of successors of any node (known as branching factor), $d$ be depth of shallowest goal, and $m$ be maximum length of any path in the search tree.
DFS Summary

Basic Behavior:
- Expands the deepest node in the tree
- Backtracks when reaches a non-goal node with no descendants

Problems:
- Make a wrong choice and can go down along an infinite path even though the solution may be very close to initial vertex
- Hence, DFS is not optimal
- If subtree is of unbounded depth and contains no solutions, DFS will never terminate.
- Hence, DFS is not complete
- Let $b$ be the maximum number of successors of any node (known as branching factor), $d$ be depth of shallowest goal, and $m$ be maximum length of any path in the search tree
- Time complexity is $O(b^m)$ and space complexity is $O(b \cdot m)$
**DFS Summary**

**Basic Behavior:**
- Expands the deepest node in the tree
- Backtracks when reaches a non-goal node with no descendants

**Problems:**
- Make a wrong choice and can go down along an infinite path even though the solution may be very close to initial vertex
- Hence, DFS is not optimal
- If subtree is of unbounded depth and contains no solutions, DFS will never terminate.
- Hence, DFS is not complete
- Let $b$ be the maximum number of successors of any node (known as branching factor), $d$ be depth of shallowest goal, and $m$ be maximum length of any path in the search tree
- Time complexity is $O(b^m)$ and space complexity is $O(b \cdot m)$
BFS vs. DFS

- When will BFS outperform DFS?
- When will DFS outperform BFS?
RecursiveDFS($v$)
1: if $v$ is unmarked then
2: mark $v$
3: for each edge $v, u$ do
4: RecursiveDFS($u$)

Color arrays can be kept to indicate that a vertex is undiscovered, the first time it is discovered, when its neighbors are in the process of being considered, and when all its neighbors have been considered.

DFS can be used to timestamp vertices with when they are discovered and when they are finished. These start and finish times are useful in various applications of DFS regarding constraint satisfaction.
Depth-limited Search (DLS)

- Problem with DFS is presence of infinite paths
- DLS limits the depth of a path in search tree of DFS
- Modifies DFS by using a predetermined depth limit $d_l$
- DLS is incomplete if the shallowest goal is beyond the depth limit $d_l$
- DLS is not optimal if $d < d_l$
- Time complexity is $O(b^{d_l})$ and space complexity is $O(b \cdot d_l)$
Depth-limited Search (DLS)

\(= \text{DFS with depth limit } d_l [\text{i.e., nodes at depth } d_l \text{ are not expanded}]\)

**Recursive implementation:**

```plaintext
function Depth-Limited-Search( problem, limit) returns soln/fail/cutoff
    Recursive-DLS(Make-Node(Initial-State[problem]), problem, limit)

function Recursive-DLS(node, problem, limit) returns soln/fail/cutoff
    cutoff-occurred? ← false
    if Goal-Test(problem, State[node]) then return node
    else if Depth[node] = limit then return cutoff
    else for each successor in Expand(node, problem) do
        result ← Recursive-DLS(successor, problem, limit)
        if result = cutoff then cutoff-occurred? ← true
        else if result ≠ failure then return result
    if cutoff-occurred? then return cutoff else return failure
```

Amarda Shehu (580)  
Elementary (Graph) Search Algorithms  
63
Iterative Deepening Search (IDS)

- Finds the best depth limit by incrementing $d_l$ until goal is found at $d_l = d$
- Can be viewed as running DLS with consecutive values of $d_l$
- IDS combines the benefits of both DFS and BFS
- Like DFS, its space complexity is $O(b \cdot d)$
- Like BFS, it is complete when the branching factor is finite, and it is optimal if the path cost is a non-decreasing function of the depth of the goal node
- Its time complexity is $O(b^d)$
- IDS is the preferred uninformed search when the state space is large, and the depth of the solution is not known
function Iterative-Deepening-Search(problem) returns a solution
inputs: problem, a problem

for depth ← 0 to ∞ do
    result ← Depth-Limited-Search(problem, depth)
    if result ≠ cutoff then return result
end
Iterative Deepening Search (IDS) @ $d_i = 0$

Limit = 0

Diagram showing the initial state of Iterative Deepening Search at $d_i = 0$. The start node is marked, and the search boundary is set to the limit of 0.
Iterative Deepening Search (IDS) @ $d_l = 1$

Limit = 1

A

A

A

A

C

C

C

C
Iterative Deepening Search (IDS) @ $d_l = 2$

Limit = 2

Diagram showing the iterative deepening search process with a limit of 2.
Iterative Deepening Search (IDS) @ $d_l = 3$

Limit = 3

- $A$
- $B$
- $C$
- $D$
- $E$
- $F$
- $G$
- $H$
- $I$
- $J$
- $K$
- $L$
- $M$
- $N$
- $O$

Amarda Shehu (580)
Elementary (Graph) Search Algorithms
<table>
<thead>
<tr>
<th>Criterion</th>
<th>Breadth-First</th>
<th>Depth-First</th>
<th>Depth-Limited</th>
<th>Iterative Deepening</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete?</td>
<td>Yes*</td>
<td>No</td>
<td>Yes, if (d_l \geq d)</td>
<td>Yes</td>
</tr>
<tr>
<td>Time (b^{d+1})</td>
<td></td>
<td>(b^m)</td>
<td>(b^{d_l})</td>
<td>(b^d)</td>
</tr>
<tr>
<td>Space (b^{d+1})</td>
<td></td>
<td>(bm)</td>
<td>(bd_l)</td>
<td>(bd)</td>
</tr>
<tr>
<td>Optimal?</td>
<td>Yes*</td>
<td>No</td>
<td>No</td>
<td>Yes*</td>
</tr>
</tbody>
</table>
- Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored
- Variety of uninformed search strategies
- IDS uses only linear space and not much more time than other uninformed algorithms
- Graph search can be exponentially more efficient than tree search
- What about least-cost paths with non-uniform state-state costs?
  - That is the subject of next lecture