Uninformed Graph Search (for Paths)
CS 483 (001) - Spring 2017

Amarda Shehu

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

Spring 2017
1 Outline of Today’s Class

2 Example Problems

3 Elementary (Graph) Search Algorithms
   - Uninformed Search
     - Breadth-first Search (BFS)
     - Depth-first Search (DFS)
     - Depth-limited Search (DLS)
     - Iterative Deepening Search (IDS)
Problem: Traveling in 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

Outline of Today's Class
A problem is defined by five components:

1. **Initial state** e.g., “In(Arad)”

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

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

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**
- **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 (its too big), but its a useful idea.
Example: Vacuum World State Space Graph

states??:

How many states?

actions??:

Left, Right, Suck, NoOp

transition model??:

$([A, \text{dirt}], \text{Suck}) \rightarrow [A, \text{clean}], \ldots$

where is transition model in graph?

goal test??:

no dirt

path cost??:

1 per action (0 for NoOp)

Amarda Shehu (483)

Outline of Today’s Class
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??:**

**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**: 
  - \((A, \text{dirt}) \rightarrow (A, \text{clean})\) etc.
- **Goal Test**: no dirt
- **Path Cost**: 1 per action (0 for *NoOp*)

---

Amarda Shehu (483)    Outline of Today's Class
Example: Vacuum World State Space Graph

- **States:** Integer dirt and robot locations (ignore dirt amounts etc.)
- **Actions:** Left, Right, Suck, NoOp
- **Transition model:** 
- **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

transition model?: ([A, dirt], Suck) → [A, clean], ...
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.)
How many states?

actions??: Left, Right, Suck, NoOp

transition model??: ([A, dirt], Suck) → [A, clean], ...
where is transition model in graph?

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

**goal test??:** no dirt

**path cost??:**
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*) \(\rightarrow\) [A, clean], ...
- **Goal test**?: no dirt
- **Path cost**?: 1 per action (0 for *NoOp*)
Example: Vacuum World State Space Graph

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

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

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

**goal test??**: no dirt

**path cost??**: 1 per action (0 for *NoOp*)
Example: The 8-puzzle

Example Problems

Start State

Goal State

states??:
Example: The 8-puzzle

Example Problems

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

How many states?
Example: The 8-puzzle

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

[Image of a 3x3 grid with numbers 7, 2, 4, 5, 6, 8, 1, 3, and 8, with the blank space highlighted in the middle, labeled as Start State and 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

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
- **Goal Test**: = goal state (given)
- **Path Cost**: 1 per move

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

---

Start State

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

Goal State

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

Amarda Shehu (483)
Example: The 8-puzzle

**Example Problems**

- **Start State**
- **Goal State**

- **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?
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)

---

Start State

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

Goal State

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>
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**: 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!]

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)
- **Path Cost**: 1 per move

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

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

**actions??:**
Example: Robotic Assembly

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

**actions**: continuous motions of robot joints
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
Route-finding and Tour-finding Problems

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/neighbors
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.
Consider this 4-state space graph: How big is its search tree?

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?
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

Diagram showing a tree search with nodes Arad, Sibiu, Timisoara, and Zerind, connected by edges.
Tree Search Example
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.
Template of Graph Search with an Underlying Tree Search

**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 $\infty$)
Uninformed Graph Search

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 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 neighbors go at end
F is a queue
**Strategy:** Expand shallowest unexpanded node

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

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

**Strategy:** Expand shallowest unexpanded node

**Implementation:**
fringe = first-in first-out (FIFO), i.e., unvisited neighbors 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

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 \( m \)?
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

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
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

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

**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?

Time: $1 + b^1 + b^2 + b^3 + \ldots + b^d + b^{(b^d - 1)} = O(b^d + 1)$, 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

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

Space

\[ O(b^d + 1) \text{ (keeps every node in memory)} \]

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? 

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

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

Optimal?? Yes (if cost = 1 per step); not optimal in general
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.
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.
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.
Depth-first Search (DFS)
Depth-first Search (DFS)
**Strategy:** Expand deepest unexpanded node

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

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

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

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
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??
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 neighbors 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 \( O(b \cdot m) \).
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.
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 neighbors 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 neighbors 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 neighbors 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?
Another Advantage of DFS

RecursiveDFS(ν)
1: if ν is unmarked then
2:   mark ν
3:   for each edge ν, 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)

= DFS with depth limit \( d \) [i.e., nodes at depth \( d \) are not expanded]

**Recursive implementation:**

```plaintext
function \text{Depth-Limited-Search}( \text{problem}, \text{limit} ) \quad \text{returns} \quad \text{soln/fail/cutoff}

    \text{Recursive-DLS(\text{Make-Node(Initial-State[problem]), problem, limit})}

function \text{Recursive-DLS}(\text{node, problem, limit}) \quad \text{returns} \quad \text{soln/fail/cutoff}

    \text{cutoff-occurred?} \leftarrow \text{false}

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

    \text{else if } \text{Depth[node]} = \text{limit} \text{ then return } \text{cutoff}

    \text{else for each successor in Expand(node, problem) do}

        \text{result} \leftarrow \text{Recursive-DLS(successor, problem, limit)}

        \text{if result = cutoff then cutoff-occurred?} \leftarrow \text{true}

        \text{else if result \neq failure then return result}

    \text{if cutoff-occurred? then return cutoff else return failure}
```
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) @ $d_l = 0$

Limit = 0
Iterative Deepening Search (IDS) @ $d_i = 1$

Limit = 1

- Initial state
- Depth 1: Explore A, B, C
- Depth 2: Explore A, C
- Depth 3: Explore A, C
- Goal reached at depth 3
Iterative Deepening Search (IDS) @ $d_l = 2$

Limit = 2
Iterative Deepening Search (IDS) \( @ d_l = 3 \)

Limit = 3

```
Limit = 3
```

```
Amarda Shehu (483)
```
## Summary of Uninformed 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</td>
<td>$b^{d+1}$</td>
<td>$b^m$</td>
<td>$b^{d_l}$</td>
<td>$b^d$</td>
</tr>
<tr>
<td>Space</td>
<td>$b^{d+1}$</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>
Uninformed Search Summary

- 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
- What about least-cost paths on weighted graphs?
  - That is the subject of next lecture