Last time: Summary

• Definition of AI?
• Turing Test?
• Intelligent Agents:
  • Anything that can be viewed as perceiving its environment through sensors and acting upon that environment through its effectors to maximize progress towards its goals.
  • PAGE (Percepts, Actions, Goals, Environment)
  • Described as a Perception (sequence) to Action Mapping: $f: P^* \rightarrow A$
  • Using look-up-table, closed form, etc.

• Agent Types: Reflex, state-based, goal-based, utility-based

• Rational Action: The action that maximizes the expected value of the performance measure given the percept sequence to date
Outline: Problem solving and search

• Introduction to Problem Solving

• Complexity

• Uninformed search
  • Problem formulation
  • Search strategies: depth-first, breadth-first

• Informed search
  • Search strategies: best-first, A*
  • Heuristic functions
Example: Measuring problem!

**Problem:** Using these three buckets, measure 7 liters of water.
Example: Measuring problem!

- (one possible) Solution:

```
\[
\text{start} \quad 3 \quad 5 \quad 9
\]
```

```
\text{a} \quad \text{b} \quad \text{c}
```

```text
\[
0 \quad 0 \quad 0
\]
```
Example: Measuring problem!

• (one possible) Solution:

```
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>start</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

```
  a   b   c
  3 l  5 l  9 l
```
Example: Measuring problem!

- (one possible) Solution:

```plaintext
\begin{array}{ccc}
\text{start} & \text{a} & \text{b} & \text{c} \\
0 & 0 & 0 & 0 \\
3 & 0 & 0 & 0 \\
0 & 0 & 0 & 3 \\
\end{array}
```

![Bar and Cylinders](image-url)
Example: Measuring problem!

- (one possible) Solution:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

\[ \text{start} \]
Example: Measuring problem!

• (one possible) Solution:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>6</td>
</tr>
</tbody>
</table>

[start]

![Cylinders](3 l, 5 l, 9 l) a b c
Example: Measuring problem!

- (one possible) Solution:

```
<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>6</td>
</tr>
</tbody>
</table>
```

![Diagram showing three containers: a (3 l), b (5 l), and c (9 l)]
Example: Measuring problem!

• (one possible) Solution:

```
  a  b  c
  0  0  0
  3  0  0
  0  0  3
  3  0  3
  0  0  6
  3  0  6
  0  3  6
```

![Cylinders](image)
Example: Measuring problem!

- (one possible) Solution:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>start</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>
Example: Measuring problem!

• (one possible) Solution:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

start

3 l
5 l
9 l
Example: Measuring problem!

- (one possible) Solution:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>goal</td>
<td>0</td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>

(a) 3 l  
(b) 5 l  
(c) 9 l
Example: Measuring problem!

• Another Solution:

```
0 0 0
0 5 0
```

```
3 l
5 l
9 l
```
Example: Measuring problem!

• Another Solution:

```
\[\begin{array}{ccc}
  a & b & c \\
  0 & 0 & 0 \\
  0 & 5 & 0 \\
  3 & 2 & 0 \\
\end{array}\]

\begin{align*}
\text{start} & \\
3l & \quad 5l & \quad 9l \\
\text{a} & \quad \text{b} & \quad \text{c}
\end{align*}
```
Example: Measuring problem!

• Another Solution:

\[
\begin{array}{ccc}
\text{a} & \text{b} & \text{c} \\
0 & 0 & 0 \\
0 & 5 & 0 \\
3 & 2 & 0 \\
3 & 0 & 2 \\
\end{array}
\]

\[
\begin{array}{lll}
& \text{3 l} & \text{5 l} & \text{9 l} \\
\text{a} & \text{b} & \text{c} \\
\end{array}
\]
Example: Measuring problem!

- Another Solution:

```
  a  b  c

0  0  0  start
0  5  0
3  2  0
3  0  2
3  5  2
```
Example: Measuring problem!

- Another Solution:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
</table>
| 0 | 0 | 0 | 0 | start
| 0 | 5 | 0 |   |
| 3 | 2 | 0 |   |
| 3 | 0 | 2 |   |
| 3 | 5 | 2 |   |
| 3 | 0 | 7 | goal

![Cylinders](image)
Which solution do we prefer?

<table>
<thead>
<tr>
<th>Solution 1:</th>
<th>Solution 2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>a b c</td>
<td>a b c</td>
</tr>
<tr>
<td>0 0 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>3 0 0</td>
<td>0 5 0</td>
</tr>
<tr>
<td>0 0 3</td>
<td>3 2 0</td>
</tr>
<tr>
<td>3 0 3</td>
<td>3 0 2</td>
</tr>
<tr>
<td>0 0 6</td>
<td>3 5 2</td>
</tr>
<tr>
<td>3 0 6</td>
<td>3 0 7</td>
</tr>
<tr>
<td>0 3 6</td>
<td>0 5 6</td>
</tr>
<tr>
<td>3 3 6</td>
<td>3 0 2</td>
</tr>
<tr>
<td>1 5 6</td>
<td>0 5 7</td>
</tr>
<tr>
<td>0 5 7</td>
<td>3 0 7</td>
</tr>
</tbody>
</table>

start

goal
Problem-Solving Agent

function SIMPLE-PROBLEM-SOLVING-AGENT( p) returns an action
inputs:  p, a percept
static:  s, an action sequence, initially empty
        state, some description of the current world state
        g, a goal, initially null
        problem, a problem formulation

state ← UPDATE-STATE(state, p)  // What is the current state?
if s is empty then
    g ← FORMULATE-GOAL(state)  // From LA to San Diego (given curr. state)
    problem ← FORMULATE-PROBLEM(state, g)  // e.g., Gas usage
    s ← SEARCH( problem)
action ← RECOMMENDATION( s, state)
    s ← REMAINDER( s, state)  // If fails to reach goal, update
return action

Note: This is offline problem-solving. Online problem-solving involves acting w/o complete knowledge of the problem and environment.
Example: Buckets

Measure 7 liters of water using a 3-liter, a 5-liter, and a 9-liter buckets.

• **Formulate goal:** Have 7 liters of water in 9-liter bucket

• **Formulate problem:**
  - States: amount of water in the buckets
  - Operators: Fill bucket from source, empty bucket

• **Find solution:** sequence of operators that bring you from current state to the goal state
The environment types largely determine the agent design.
Problem types

- **Single-state problem:** deterministic, accessible
  
  *Agent knows everything about world, thus can calculate optimal action sequence to reach goal state.*

- **Multiple-state problem:** deterministic, inaccessible
  
  *Agent must reason about sequences of actions and states assumed while working towards goal state.*

- **Contingency problem:** nondeterministic, inaccessible
  - Must use sensors during execution
  - Solution is a tree or policy
  - Often interleave search and execution

- **Exploration problem:** unknown state space
  
  *Discover and learn about environment while taking actions.*
Problem types

- **Single-state problem:** deterministic, accessible

  - Agent knows everything about world (the exact state),
  
  - Can calculate optimal action sequence to reach goal state.
    
  - E.g., playing chess. Any action will result in an exact state
Problem types

• **Multiple-state problem:** deterministic, inaccessible

  • Agent does not know the exact state (could be in any of the possible states)
    • May not have sensor at all

  • Assume states while working towards goal state.

• E.g., walking in a dark room
  • If you are at the door, going straight will lead you to the kitchen
  • If you are at the kitchen, turning left leads you to the bedroom
  • ...
Problem types

- **Contingency problem:** nondeterministic, inaccessible
  - Must use sensors during execution
  - Solution is a tree or policy
  - Often interleave search and execution

- E.g., a new skater in an arena
  - Sliding problem.
  - Many skaters around
Problem types

- **Exploration problem**: unknown state space

  Discover and learn about environment while taking actions.

  - *E.g., Maze*
Example: Vacuum world

**Simplified world:** 2 locations, each may or not contain dirt, each may or not contain vacuuming agent.

**Goal of agent:** clean up the dirt.

---

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

**Multiple-state, start in \{1, 2, 3, 4, 5, 6, 7, 8\}**

* Например, Right goes to \{2, 4, 6, 8\}. Solution??

**Contingency, start in #5**

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

**Local sensing:** dirt, location only.

**Solution??**
Example: Romania

• In Romania, on vacation. Currently in Arad.
• Flight leaves tomorrow from Bucharest.

• **Formulate goal:**
  ➢ be in Bucharest

• **Formulate problem:**
  ➢ states: various cities
  ➢ operators: drive between cities

• **Find solution:**
  ➢ sequence of cities, such that total driving distance is minimized.
Example: Traveling from Arad To Bucharest
Problem formulation

A problem is defined by four items:

initial state   e.g., “at Arad”

operators (or successor function $S(x)$)
   e.g., Arad $\rightarrow$ Zerind   Arad $\rightarrow$ Sibiu   etc.

goal test, can be
   explicit, e.g., $x = “at Bucharest”$
   implicit, e.g., $NoDirt(x)$

path cost (additive)
   e.g., sum of distances, number of operators executed, etc.

A solution is a sequence of operators
leading from the initial state to a goal state
Selecting a state space

- Real world is absurdly complex; some abstraction is necessary to allow us to reason on it...

- Selecting the correct abstraction and resulting state space is a difficult problem!

- Abstract states $\leftrightarrow$ real-world states

- Abstract operators $\leftrightarrow$ sequences or real-world actions
  (e.g., going from city i to city j costs $L_{ij} \leftrightarrow$ actually drive from city i to j)

- Abstract solution $\leftrightarrow$ set of real actions to take in the real world such as to solve problem
Example: 8-puzzle

- State:
- Operators:
- Goal test:
- Path cost:
Example: 8-puzzle

- State: integer location of tiles (ignore intermediate locations)
- Operators: moving blank left, right, up, down (ignore jamming)
- Goal test: does state match goal state?
- Path cost: 1 per move

start state

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

goal state

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

Why search algorithms?

- 8-puzzle has 362,800 states
- 15-puzzle has $10^{12}$ states
- 24-puzzle has $10^{25}$ states

So, we need a principled way to look for a solution in these huge search spaces...
Back to Vacuum World

states??
operators??
goal test??
path cost??
Back to Vacuum World

states???: integer dirt and robot locations (ignore dirt *amounts*)
operators???: *Left*, *Right*, *Suck*
goal test???: no dirt
path cost???: 1 per operator
Example: Robotic Assembly

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

operators??: continuous motions of robot joints

goal test??: complete assembly with no robot included!

path cost??: time to execute
Real-life example: VLSI Layout

• Given schematic diagram comprising components (chips, resistors, capacitors, etc) and interconnections (wires), find optimal way to place components on a printed circuit board, under the constraint that only a small number of wire layers are available (and wires on a given layer cannot cross!)

• “optimal way”??

- minimize surface area
- minimize number of signal layers
- minimize number of vias (connections from one layer to another)
- minimize length of some signal lines (e.g., clock line)
- distribute heat throughout board
- etc.
Protel's hierarchical schematic design features let you take a "bottom up" or "top down" approach, depending on your preferred methodology. Protel can automatically generate sub-sheets based on higher-level sheet symbols, or create sheet symbols based on existing sheets.
Use automated tools to place components and route wiring.

Protel 99 SE’s unique 3D visualization feature lets you see your finished board before it leaves your desktop. Sophisticated 3D modeling and extrusion techniques render your board in stunning 3D without the need for additional height information. Rotate and zoom to examine every aspect of your board.
Search algorithms

Basic idea:
offline, systematic exploration of simulated state-space by generating successors of explored states (expanding)

Function General-Search(problem, strategy) returns a solution, or failure
initialize the search tree using the initial state 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 resulting nodes to the search tree
end
Last time: Problem-Solving

• **Problem solving:**
  • Goal formulation
  • Problem formulation (states, operators)
  • Search for solution

• **Problem formulation:**
  • Initial state
  • ?
  • ?
  • ?

• **Problem types:**
  • single state: accessible and deterministic environment
  • multiple state: ?
  • contingency: ?
  • exploration: ?
Last time: Problem-Solving

- **Problem solving:**
  - Goal formulation
  - Problem formulation (states, operators)
  - Search for solution

- **Problem formulation:**
  - Initial state
  - Operators
  - Goal test
  - Path cost

- **Problem types:**
  - single state: accessible and deterministic environment
  - multiple state: ?
  - contingency: ?
  - exploration: ?
Last time: Problem-Solving

- **Problem solving:**
  - Goal formulation
  - Problem formulation (states, operators)
  - Search for solution

- **Problem formulation:**
  - Initial state
  - Operators
  - Goal test
  - Path cost

- **Problem types:**
  - single state: accessible and deterministic environment
  - multiple state: inaccessible and deterministic environment
  - contingency: inaccessible and nondeterministic environment
  - exploration: unknown state-space
Last time: Finding a solution

Solution: is ???

Basic idea: offline, systematic exploration of simulated state-space by generating successors of explored states (expanding)

<table>
<thead>
<tr>
<th>Function</th>
<th>General-Search(\textit{problem, strategy}) returns a \textit{solution}, or failure</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>initialize the search tree using the initial state problem</td>
</tr>
<tr>
<td>loop do</td>
<td>if there are no candidates for expansion then \textbf{return} failure</td>
</tr>
<tr>
<td></td>
<td>choose a leaf node for expansion according to strategy</td>
</tr>
<tr>
<td></td>
<td>if the node contains a goal state then \textbf{return} the corresponding solution</td>
</tr>
<tr>
<td></td>
<td>else expand the node and add resulting nodes to the search tree</td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
</tbody>
</table>
Last time: Finding a solution

**Solution:** is a sequence of operators that bring you from current state to the goal state.

**Basic idea:** offline, systematic exploration of simulated state-space by generating successors of explored states (expanding).

**Function** General-Search(_problem, _strategy_) returns a _solution_, or failure

initialize the search tree using the initial state 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 resulting nodes to the search tree
end

**Strategy:** The search strategy is determined by ???
Last time: Finding a solution

**Solution:** is a sequence of operators that bring you from current state to the goal state

**Basic idea:** offline, systematic exploration of simulated state-space by generating successors of explored states (expanding)

```plaintext
Function General-Search(problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state 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 resulting nodes to the search tree
    end

**Strategy:** The search strategy is determined by the order in which the nodes are expanded.
```
Example: Traveling from Arad To Bucharest
From problem space to search tree

- Some material in this and following slides is from http://www.cs.kuleuven.ac.be/~dannyd/FAI/ check it out!

Problem space

Associated loop-free search tree
Paths in search trees

Denotes: SA

Denotes: SDA

Denotes: SDEBA
General search example
General search example
General search example
General search example
Implementation of search algorithms

Function General-Search(problem, Queuing-Fn) returns a solution, or failure

\begin{verbatim}
    nodes ← make-queue(make-node(initial-state[problem]))
    loop do
        if nodes is empty then return failure
        node ← Remove-Front(nodes)
        if Goal-Test[problem] applied to State(node) succeeds then return node
        nodes ← Queuing-Fn(nodes, Expand(node, Operators[problem]))
    end
\end{verbatim}

Queuing-Fn(queue, elements) is a queuing function that inserts a set of elements into the queue and determines the order of node expansion. Varieties of the queuing function produce varieties of the search algorithm.
Encapsulating *state* information in *nodes*

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 **Operators** (or **Successor**F**N**) of the problem to create the corresponding states.
Evaluation of search strategies

- A search strategy is defined by picking the order of node expansion.

- Search algorithms are commonly evaluated according to the following four criteria:
  - **Completeness**: does it always find a solution if one exists?
  - **Time complexity**: how long does it take as function of num. of nodes?
  - **Space complexity**: how much memory does it require?
  - **Optimality**: does it guarantee the least-cost solution?

- Time and space complexity are measured in terms of:
  - \( b \) – max branching factor of the search tree
  - \( d \) – depth of the least-cost solution
  - \( m \) – max depth of the search tree (may be infinity)
Binary Tree Example

Depth = 0

Depth = 1

Depth = 2

Number of nodes: \( n = 2 \) max depth
Number of levels (max depth) = \( \log(n) \) (could be \( n \))
Complexity

- Why worry about complexity of algorithms?

  - because a problem may be solvable in principle but may take too long to solve in practice
Complexity: Tower of Hanoi

Figure 11-6  Tower of Hanoi problem with three disks
Complexity:
Tower of Hanoi
Complexity: Tower of Hanoi

- 3-disk problem: $2^3 - 1 = 7$ moves

- 64-disk problem: $2^{64} - 1$.
  - $2^{10} = 1024 \approx 1000 = 10^3$, 
  - $2^{64} = 2^4 \times 2^{60} \approx 2^4 \times 10^{18} = 1.6 \times 10^{19}$

- One year $\approx 3.2 \times 10^7$ seconds
Complexity: Tower of Hanoi

- The wizard’s speed = one disk / second

\[
1.6 \times 10^{19} = 5 \times 3.2 \times 10^{18} = \\
5 \times (3.2 \times 10^7) \times 10^{11} = \\
(3.2 \times 10^7) \times (5 \times 10^{11})
\]
Complexity: Tower of Hanoi

- The time required to move all 64 disks from needle 1 to needle 3 is roughly $5 \times 10^{11}$ years.

- It is estimated that our universe is about 15 billion years old, or $1.5 \times 10^{10}$ years old.

$$5 \times 10^{11} = 50 \times 10^{10} \approx 33 \times (1.5 \times 10^{10}).$$
Complexity: Tower of Hanoi

• Assume: a computer with 1 billion \( = 10^9 \) moves/second.
  • \( \text{Moves/year}=(3.2 \times 10^7) \times 10^9 = 3.2 \times 10^{16} \)

• To solve the problem for 64 disks:
  • \( 2^{64} \approx 1.6 \times 10^{19} = 1.6 \times 10^{16} \times 10^3 = (3.2 \times 10^{16}) \times 500 \)

• 500 years for the computer to generate \( 2^{64} \) moves at the rate of 1 billion moves per second.
Complexity

• Why worry about complexity of algorithms?
  ➢ because a problem may be solvable in principle but may take too long to solve in practice

• How can we evaluate the complexity of algorithms?
  ➢ through asymptotic analysis, i.e., estimate time (or number of operations) necessary to solve an instance of size \( n \) of a problem when \( n \) tends towards infinity

  ➢ See AIMA, Appendix A.
Complexity example: Traveling Salesman Problem

- There are n cities, with a road of length $L_{ij}$ joining city i to city j.
- The salesman wishes to find a way to visit all cities that is optimal in two ways:
  - each city is visited only once, and
  - the total route is as short as possible.
This is a *hard* problem: the only known algorithms (so far) to solve it have exponential complexity, that is, the number of operations required to solve it grows as \( \exp(n) \) for \( n \) cities.
Why is exponential complexity “hard”? It means that the number of operations necessary to compute the exact solution of the problem grows exponentially with the size of the problem (here, the number of cities).

- \( \exp(1) = 2.72 \)
- \( \exp(10) = 2.20 \times 10^4 \) (daily salesman trip)
- \( \exp(100) = 2.69 \times 10^{43} \) (monthly salesman planning)
- \( \exp(500) = 1.40 \times 10^{217} \) (music band worldwide tour)
- \( \exp(250,000) = 10^{108,573} \) (fedex, postal services)
- Fastest computer = \( 10^{12} \) operations/second
In general, exponential-complexity problems cannot be solved for any but the smallest instances!
Complexity

• **Polynomial-time (P) problems:** we can find algorithms that will solve them in a time (=number of operations) that grows polynomially with the size of the input.

➤ **for example:** sort n numbers into increasing order: poor algorithms have \(n^2\) complexity, better ones have \(n \log(n)\) complexity.
Complexity

- Since we did not state what the order of the polynomial is, it could be very large! Are there algorithms that require more than polynomial time?

- Yes (until proof of the contrary); for some algorithms, we do not know of any polynomial-time algorithm to solve them. These are referred to as nondeterministic-polynomial-time (NP) algorithms.

- For example: traveling salesman problem.

- In particular, exponential-time algorithms are believed to be NP.
Note on NP-hard problems

• The formal definition of NP problems is:

A problem is **nondeterministic polynomial** if there exists some algorithm that can guess a solution and then verify whether or not the guess is correct in polynomial time.

(one can also state this as these problems being solvable in polynomial time on a nondeterministic Turing machine.)

In practice, until proof of the contrary, this means that known algorithms that run on known computer architectures will take more than polynomial time to solve the problem.
Complexity: $O()$ and $o()$ measures (Landau symbols)

• How can we represent the complexity of an algorithm?

• Given: Problem input (or instance) size: $n$
  Number of operations to solve problem: $f(n)$

• If, for a given function $g(n)$, we have:
  \[ \exists k \in \mathbb{R}, \exists n_0 \in \mathbb{N}, \forall n \in \mathbb{N}, n \geq n_0, f(n) \leq kg(n) \]
  then $f \in O(g)$ \hspace{1cm} $f$ is dominated by $g$

• If, for a given function $g(n)$, we have:
  \[ \forall k \in \mathbb{R}, \exists n_0 \in \mathbb{N}, \forall n \in \mathbb{N}, n \geq n_0, f(n) \leq kg(n) \]
  then $f \in o(g)$ \hspace{1cm} $f$ is negligible compared to $g$
Landau symbols

\[ f \in O(g) \iff \exists k, \quad f(n) \leq k g(n) \iff \lim_{n \to \infty} \frac{f}{g} \text{ is bounded} \]

\[ f \in o(g) \iff \forall k, \quad f(n) \leq k g(n) \iff \lim_{n \to \infty} \frac{f(n)}{g(n)} \to 0 \]
Examples, properties

- \( f(n) = n, \ g(n) = n^2: \)
  
  \( n \) is \( o(n^2) \), because \( n/n^2 = 1/n \to 0 \) as \( n \to \infty \)

  similarly, \( \log(n) \) is \( o(n) \)

  \( n^C \) is \( o(\exp(n)) \) for any \( C \)

- if \( f \) is \( O(g) \), then for any \( K, K.f \) is also \( O(g) \); idem for \( o() \)

- if \( f \) is \( O(h) \) and \( g \) is \( O(h) \), then for any \( K, L: K.f + L.g \) is \( O(h) \)

  idem for \( o() \)

- if \( f \) is \( O(g) \) and \( g \) is \( O(h) \), then \( f \) is \( O(h) \)

- if \( f \) is \( O(g) \) and \( g \) is \( o(h) \), then \( f \) is \( o(h) \)

- if \( f \) is \( o(g) \) and \( g \) is \( O(h) \), then \( f \) is \( o(h) \)
Polynomial-time hierarchy

- From Handbook of Brain Theory & Neural Networks (Arbib, ed.; MIT Press 1995).

**AC°**: can be solved using gates of constant depth

**NC¹**: can be solved in logarithmic depth using 2-input gates

**NC**: can be solved by small, fast parallel computer

**P**: can be solved in polynomial time

**P-complete**: hardest problems in P; if one of them can be proven to be NC, then P = NC

**NP**: nondeterministic-polynomial algorithms

**NP-complete**: hardest NP problems; if one of them can be proven to be P, then NP = P

**PH**: polynomial-time hierarchy
Complexity and the human brain

- Are computers close to human brain power?

- Current computer chip (CPU):
  - $10^3$ inputs (pins)
  - $10^7$ processing elements (gates)
  - 2 inputs per processing element (fan-in = 2)
  - processing elements compute boolean logic (OR, AND, NOT, etc)

- Typical human brain:
  - $10^7$ inputs (sensors)
  - $10^{10}$ processing elements (neurons)
  - fan-in = $10^3$
  - processing elements compute complicated functions

Still a lot of improvement needed for computers; but computer clusters come close!
Remember: Implementation of search algorithms

**Function** General-Search(problem, Queuing-Fn) **returns** a solution, or failure

nodes \(\leftarrow\) make-queue(make-node(initial-state[problem]))

**loop do**

- **if** nodes is empty **then return** failure
- node \(\leftarrow\) Remove-Front(nodes)
- **if** Goal-Test[problem] applied to State(node) succeeds **then return** node
- nodes \(\leftarrow\) Queuing-Fn(nodes, Expand(node, Operators[problem]))

**end**

**Queuing-Fn**(*queue, elements*) is a queuing function that inserts a set of elements into the queue and determines the order of node expansion. Varieties of the queuing function produce varieties of the search algorithm.
Encapsulating *state* information in *nodes*

A *state* is a (representation of) a physical configuration.

A *node* is a data structure constituting part of a search tree that 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 *operators* (or *successorFn*) of the problem to create the corresponding states.
Evaluation of search strategies

• A search strategy is defined by picking the order of node expansion.

• Search algorithms are commonly evaluated according to the following four criteria:
  • **Completeness**: does it always find a solution if one exists?
  • **Time complexity**: how long does it take as function of num. of nodes?
  • **Space complexity**: how much memory does it require?
  • **Optimality**: does it guarantee the least-cost solution?

• Time and space complexity are measured in terms of:
  • $b$ – max branching factor of the search tree
  • $d$ – depth of the least-cost solution
  • $m$ – max depth of the search tree (may be infinity)
Note: Approximations

• In our complexity analysis, we do not take the built-in loop-detection into account.

• The results only ‘formally’ apply to the variants of our algorithms WITHOUT loop-checks.

• Studying the effect of the loop-checking on the complexity is hard:
  • overhead of the checking MAY or MAY NOT be compensated by the reduction of the size of the tree.

• Also: our analysis DOES NOT take the length (space) of representing paths into account !!

http://www.cs.kuleuven.ac.be/~dannyd/FAI/
Uninformed search strategies

Use only information available in the problem formulation

• Breadth-first
• Uniform-cost
• Depth-first
• Depth-limited
• Iterative deepening
Breadth-first search

Expand shallowest unexpanded node

Implementation:

\[ \text{QUEUEING FN} = \text{put successors at end of queue} \]

Arad
Breadth-first search

Move downwards, level by level, until goal is reached.
Example: Traveling from Arad To Bucharest
Breadth-first search
Breadth-first search

Diagram: A graph with cities Arad, Oradea, Sibiu, and Timisoara connected. Arad is the root node, and the graph is expanded to show connected cities.
Breadth-first search
Properties of breadth-first search

• Completeness:
• Time complexity:
• Space complexity:
• Optimality:

Search algorithms are commonly evaluated according to the following four criteria:
• **Completeness**: does it always find a solution if one exists?
• **Time complexity**: how long does it take as function of num. of nodes?
• **Space complexity**: how much memory does it require?
• **Optimality**: does it guarantee the least-cost solution?

Time and space complexity are measured in terms of:
• $b$ – max branching factor of the search tree
• $d$ – depth of the least-cost solution
• $m$ – max depth of the search tree (may be infinity)
Properties of breadth-first search

- Completeness: Yes, if $b$ is finite
- Time complexity: $1+b+b^2+\ldots+b^d = O(b^d)$, i.e., exponential in $d$
- Space complexity: $O(b^d)$, keeps every node in memory
- Optimality: Yes (assuming cost = 1 per step)

Why keep every node in memory? To avoid revisiting already-visited nodes, which may easily yield infinite loops.
Time complexity of breadth-first search

- If a goal node is found on depth $d$ of the tree, all nodes up till that depth are created.

- Thus: $O(b^d)$
Space complexity of breadth-first

- Largest number of nodes in QUEUE is reached on the level $d$ of the goal node.

- QUEUE contains all red and yellow nodes. (Thus: 4).

- In General: $b^d$
Uniform-cost search

Expand least-cost unexpanded node

Implementation:

\[ \text{QueueingFn} = \text{insert in order of increasing path cost} \]

So, the queueing function keeps the node list sorted by increasing path cost, and we expand the first unexpanded node (hence with smallest path cost)

A refinement of the breadth-first strategy:

Breadth-first = uniform-cost with path cost = node depth
Romania with step costs in km

Straight-line distance to Bucharest:

- Arad: 366
- Bucharest: 0
- Craiova: 160
- Dobrota: 242
- Eforie: 161
- Fagaras: 178
- Giurgiu: 77
- Hirsova: 151
- Iasi: 226
- Lugoj: 244
- Mehadia: 241
- Neamt: 234
- Oradea: 380
- Pitesti: 98
- Rimnic Vîlcea: 193
- Sibiu: 253
- Timisoara: 329
- Urziceni: 80
- Vaslui: 199
- Zerind: 374
Uniform-cost search
Uniform-cost search
Uniform-cost search
Properties of uniform-cost search

- Completeness: Yes, if step cost \( \geq \varepsilon > 0 \)
- Time complexity: \# nodes with \( g \leq \) cost of optimal solution, \( \leq O(b^d) \)
- Space complexity: \# nodes with \( g \leq \) cost of optimal solution, \( \leq O(b^d) \)
- Optimality: Yes, as long as path cost never decreases

\( g(n) \) is the path cost to node \( n \)

Remember:

- \( b = \) branching factor
- \( d = \) depth of least-cost solution
Implementation of uniform-cost search

- Initialize \textbf{Queue} with root node (built from start state)

- Repeat until (\textbf{Queue} empty) or (first node has \textbf{Goal state}):
  - Remove first node from front of Queue
  - Expand node (find its children)
  - Reject those children that have already been considered, to avoid loops
  - Add remaining children to Queue, \textit{in a way that keeps entire queue sorted by increasing path cost}

- If Goal was reached, return success, otherwise failure
Caution!

- Uniform-cost search not optimal if it is terminated when \textit{any} node in the queue has goal state.

- Uniform cost returns the path with cost 102 (if any goal node is considered a solution), while there is a path with cost 25.
Note: Loop Detection

- In class, we saw that the search may fail or be sub-optimal if:
  - no loop detection: then algorithm runs into infinite cycles
    \((A \rightarrow B \rightarrow A \rightarrow B \rightarrow \ldots)\)
  - not queuing-up a node that has a state which we have already visited: may yield suboptimal solution
  - simply avoiding to go back to our parent: looks promising, but we have not proven that it works

Solution? do not enqueue a node if its state matches the state of any of its parents (assuming path costs > 0).

Indeed, if path costs > 0, it will always cost us more to consider a node with that state again than it had already cost us the first time.

Is that enough??
Example

Example Illustrating Uninformed Search Strategies

From: http://www.csee.umbc.edu/471/current/notes/uninformed-search/
Breadth-First Search Solution

Breadth-First Search

return GENERAL-SEARCH(problem, ENQUEUE-AT-END)
exp. node nodes list
  { S }  
S  { A B C }  
A  { B C D E G }  
B  { C D E G G' }  
C  { D E G G' G" }  
D  { E G G' G" }  
E  { G G' G" }  
G  { G' G" }  

Solution path found is S A G  ← this G also has cost 10
Number of nodes expanded (including goal node) = 7
Uniform-Cost Search Solution

From: http://www.csee.umbc.edu/471/current/notes/uninformed-search/

Uniform-Cost Search

GENERAL-SEARCH(problem, ENQUEUE-BY-PATH-COST)

exp. node   nodes list

( S )

S  { A(1) B(5) C(8) }

A  { D(4) B(5) C(8) E(8) G(10) }  (NB, we don't return G)

D  { B(5) C(8) E(8) G(10) }

B  { C(8) E(8) G(9) G(10) }

C  { E(8) G(9) G(10) G(13) }

E  { G(9) G(10) G(13) }

G  { }  

Solution path found is S B G  <-- this G has cost 9, not 10
Number of nodes expanded (including goal node) = 7
Note: Queueing in Uniform-Cost Search

In the previous example, it is wasteful (but not incorrect) to queue-up three nodes with G state, if our goal is to find the least-cost solution:

Although they represent different paths, we know for sure that the one with smallest path cost (9 in the example) will yield a solution with smaller total path cost than the others.

So we can refine the queueing function by:

- queue-up node if

  1) its state does not match the state of any parent

  and

  2) path cost smaller than path cost of any unexpanded node with same state in the queue (and in this case, replace old node with same state by our new node)

Is that it??
A Clean Robust Algorithm

**Function** UniformCost-Search(problem, Queuing-Fn) **returns** a solution, or failure

open $\leftarrow$ make-queue(make-node(initial-state[problem]))
closed $\leftarrow$ [empty]

**loop do**

if open is empty then **return** failure
currnode $\leftarrow$ Remove-Front(open)
if Goal-Test[problem] applied to State(currnode) then **return** currnode
children $\leftarrow$ Expand(currnode, Operators[problem])
while children not empty

[... see next slide ...]

end

closed $\leftarrow$ Insert(closed, currnode)
open $\leftarrow$ Sort-By-PathCost(open)
end
A Clean Robust Algorithm

[... see previous slide ...]

```
children ← Expand(currnode, Operators[problem])
while children not empty
    child ← Remove-Front(children)
    if no node in open or closed has child’s state
        open ← Queuing-Fn(open, child)
    else if there exists node in open that has child’s state
        if PathCost(child) < PathCost(node)
            open ← Delete-Node(open, node)
            open ← Queuing-Fn(open, child)
    else if there exists node in closed that has child’s state
        if PathCost(child) < PathCost(node)
            closed ← Delete-Node(closed, node)
            open ← Queuing-Fn(open, child)
end
[... see previous slide ...]
```
Example

<table>
<thead>
<tr>
<th>#</th>
<th>State</th>
<th>Depth</th>
<th>Cost</th>
<th>Parent</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
</tbody>
</table>
Example

<table>
<thead>
<tr>
<th>#</th>
<th>State</th>
<th>Depth</th>
<th>Cost</th>
<th>Parent</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>1</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

Black = open queue
Grey = closed queue

Insert expanded nodes
Such as to keep open queue sorted
Node 2 has 2 successors: one with state B and one with state S.

We have node #1 in *closed* with state S; but its path cost 0 is smaller than the path cost obtained by expanding from A to S. So we do not queue-up the successor of node 2 that has state S.
Node 4 has a successor with state C and Cost smaller than node #3 in open that Also had state C; so we update open To reflect the shortest path.
Example

<table>
<thead>
<tr>
<th>#</th>
<th>State</th>
<th>Depth</th>
<th>Cost</th>
<th>Parent</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>B</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>C</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>D</td>
<td>4</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>G</td>
<td>3</td>
<td>102</td>
<td>4</td>
</tr>
</tbody>
</table>
Example

<table>
<thead>
<tr>
<th>#</th>
<th>State</th>
<th>Depth</th>
<th>Cost</th>
<th>Parent</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>B</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>C</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>D</td>
<td>4</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>E</td>
<td>5</td>
<td>13</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>G</td>
<td>3</td>
<td>102</td>
<td>4</td>
</tr>
</tbody>
</table>
Example

<table>
<thead>
<tr>
<th>#</th>
<th>State</th>
<th>Depth</th>
<th>Cost</th>
<th>Parent</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>B</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>C</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>D</td>
<td>4</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>E</td>
<td>5</td>
<td>13</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>F</td>
<td>6</td>
<td>18</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>G</td>
<td>3</td>
<td>102</td>
<td>4</td>
</tr>
</tbody>
</table>
Example

<table>
<thead>
<tr>
<th>#</th>
<th>State</th>
<th>Depth</th>
<th>Cost</th>
<th>Parent</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>B</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>C</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>D</td>
<td>4</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>E</td>
<td>5</td>
<td>13</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>F</td>
<td>6</td>
<td>18</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>G</td>
<td>7</td>
<td>23</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>G</td>
<td>3</td>
<td>102</td>
<td>4</td>
</tr>
</tbody>
</table>
Example

<table>
<thead>
<tr>
<th>#</th>
<th>State</th>
<th>Depth</th>
<th>Cost</th>
<th>Parent</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>B</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>C</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>D</td>
<td>4</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>E</td>
<td>5</td>
<td>13</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>F</td>
<td>6</td>
<td>18</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>G</td>
<td>7</td>
<td>23</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>G</td>
<td>3</td>
<td>102</td>
<td>4</td>
</tr>
</tbody>
</table>

Goal reached
More examples...

- See the great demos at:

Depth-first search

Expand deepest unexpanded node

**Implementation:**

\[ \text{QUEUEINGFN} = \text{insert successors at front of queue} \]
Depth First Search
Romania with step costs in km

<table>
<thead>
<tr>
<th>City</th>
<th>Straight-line distance to Bucharest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arad</td>
<td>366</td>
</tr>
<tr>
<td>Bucharest</td>
<td>0</td>
</tr>
<tr>
<td>Craiova</td>
<td>160</td>
</tr>
<tr>
<td>Dobrota</td>
<td>242</td>
</tr>
<tr>
<td>Eforie</td>
<td>161</td>
</tr>
<tr>
<td>Fagaras</td>
<td>178</td>
</tr>
<tr>
<td>Giurgiu</td>
<td>77</td>
</tr>
<tr>
<td>Hirsova</td>
<td>151</td>
</tr>
<tr>
<td>Iasi</td>
<td>226</td>
</tr>
<tr>
<td>Lugoj</td>
<td>244</td>
</tr>
<tr>
<td>Mehadia</td>
<td>241</td>
</tr>
<tr>
<td>Neamt</td>
<td>234</td>
</tr>
<tr>
<td>Oradea</td>
<td>380</td>
</tr>
<tr>
<td>Pitesti</td>
<td>98</td>
</tr>
<tr>
<td>Rimnicu Vilea</td>
<td>193</td>
</tr>
<tr>
<td>Sibiu</td>
<td>253</td>
</tr>
<tr>
<td>Timisoara</td>
<td>329</td>
</tr>
<tr>
<td>Urziceni</td>
<td>80</td>
</tr>
<tr>
<td>Vaslui</td>
<td>199</td>
</tr>
<tr>
<td>Zerind</td>
<td>374</td>
</tr>
</tbody>
</table>
Depth-first search
Depth-first search
Depth-first search

I.e., depth-first search can perform infinite cyclic excursions
Need a finite, non-cyclic search space (or repeated-state checking)
Properties of depth-first search

- Completeness: No, fails in infinite state-space (yes if finite state space)
- Time complexity: \(O(b^m)\)
- Space complexity: \(O(bm)\)
- Optimality: No

Remember:

\[ b = \text{branching factor} \]
\[ m = \text{max depth of search tree} \]
Time complexity of depth-first: details

- In the worst case:
  - the (only) goal node may be on the right-most branch,

\[ \text{Time complexity} = b^m + b^{m-1} + \ldots + 1 = \frac{b^{m+1} - 1}{b - 1} \]

Thus: \( O(b^m) \)
Space complexity of depth-first

- Largest number of nodes in QUEUE is reached in bottom left-most node.
- Example: \( m = 3, \ b = 3 \):

\[
\text{QUEUE contains all } \begin{array}{c}
\text{nodes. Thus: 7.}
\end{array}
\]

- In General: \((b-1) \times m + 1\)
- Order: \(O(m \times b)\)
Avoiding repeated states

In increasing order of effectiveness and computational overhead:

• do not return to state we come from, i.e., expand function will skip possible successors that are in same state as node’s parent.

• do not create paths with cycles, i.e., expand function will skip possible successors that are in same state as any of node’s ancestors.

• do not generate any state that was ever generated before, by keeping track (in memory) of every state generated, unless the cost of reaching that state is lower than last time we reached it.
Depth-limited search

Is a depth-first search with depth limit $l$

**Implementation:**
Nodes at depth $l$ have no successors.

**Complete:** if cutoff chosen appropriately then it is guaranteed to find a solution.

**Optimal:** it does not guarantee to find the least-cost solution
Iterative deepening search

**Function** Iterative-deepening-Search(*problem*) *returns* a solution, or failure

for *depth* = 0 to ∞ do
    *result* ← Depth-Limited-Search(*problem, depth*)
    if *result* succeeds then return *result*
end
return failure

Combines the best of breadth-first and depth-first search strategies.

- Completeness: Yes,
- Time complexity: \( O(b^d) \)
- Space complexity: \( O(bd) \)
- Optimality: Yes, if step cost = 1
### Romania with step costs in km

<table>
<thead>
<tr>
<th>Location</th>
<th>Distance (km)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arad</td>
<td>75</td>
</tr>
<tr>
<td>Bucharest</td>
<td>0</td>
</tr>
<tr>
<td>Craiova</td>
<td>151</td>
</tr>
<tr>
<td>Dobroa</td>
<td>242</td>
</tr>
<tr>
<td>Eforie</td>
<td>92</td>
</tr>
<tr>
<td>Fagaras</td>
<td>87</td>
</tr>
<tr>
<td>Giurgiu</td>
<td>77</td>
</tr>
<tr>
<td>Hirsova</td>
<td>142</td>
</tr>
<tr>
<td>Iasi</td>
<td>98</td>
</tr>
<tr>
<td>Lisa</td>
<td>87</td>
</tr>
<tr>
<td>Oradea</td>
<td>140</td>
</tr>
<tr>
<td>Pitești</td>
<td>211</td>
</tr>
<tr>
<td>Rimnicu Vâlcea</td>
<td>138</td>
</tr>
<tr>
<td>Sibiu</td>
<td>111</td>
</tr>
<tr>
<td>Timișoara</td>
<td>146</td>
</tr>
<tr>
<td>Lugoj</td>
<td>70</td>
</tr>
<tr>
<td>Mehadia</td>
<td>75</td>
</tr>
<tr>
<td>Dobroa</td>
<td>120</td>
</tr>
<tr>
<td>Craiova</td>
<td>101</td>
</tr>
<tr>
<td>Giurgiu</td>
<td>90</td>
</tr>
<tr>
<td>Vâlcea</td>
<td>85</td>
</tr>
<tr>
<td>Bucharest</td>
<td>98</td>
</tr>
<tr>
<td>Sibiu</td>
<td>193</td>
</tr>
<tr>
<td>Timișoara</td>
<td>253</td>
</tr>
<tr>
<td>Urziceni</td>
<td>329</td>
</tr>
<tr>
<td>Vâlcea</td>
<td>80</td>
</tr>
<tr>
<td>Eforie</td>
<td>199</td>
</tr>
<tr>
<td>Zerind</td>
<td>374</td>
</tr>
</tbody>
</table>

The straight-line distance to Bucharest:
- Arad: 366 km
- Bucharest: 0 km
- Craiova: 160 km
- Dobroa: 242 km
- Eforie: 161 km
- Fagaras: 178 km
- Giurgiu: 77 km
- Hirsova: 151 km
- Iasi: 226 km
- Lugoj: 244 km
- Mehadia: 241 km
- Neamț: 234 km
- Oradea: 380 km
- Pitești: 98 km
- Rimnicu Vâlcea: 193 km
- Sibiu: 253 km
- Timișoara: 329 km
- Urziceni: 80 km
- Vâlcea: 199 km
- Eforie: 199 km
- Zerind: 374 km
Iterative deepening search $l = 0$
Iterative deepening search $l = 1$
Iterative deepening search $l = 2$
Iterative deepening complexity

- Iterative deepening search may seem wasteful because so many states are expanded multiple times.

- In practice, however, the overhead of these multiple expansions is small, because most of the nodes are towards leaves (bottom) of the search tree:

  thus, the nodes that are evaluated several times (towards top of tree) are in relatively small number.
Iterative deepening complexity

- In iterative deepening, nodes at bottom level are expanded once, level above twice, etc. up to root (expanded $d+1$ times) so total number of expansions is:
  \[(d+1)1 + (d)b + (d-1)b^2 + \ldots + 3b^{(d-2)} + 2b^{(d-1)} + 1b^d = O(b^d)\]

- In general, iterative deepening is preferred to depth-first or breadth-first when search space large and depth of solution not known.
Bidirectional search

- Both search forward from initial state, and backwards from goal.
- Stop when the two searches meet in the middle.

**Problem:** how do we search backwards from goal??
- predecessor of node n = all nodes that have n as successor
- this may not always be easy to compute!
- if several goal states, apply predecessor function to them just as we applied successor (only works well if goals are explicitly known; may be difficult if goals only characterized implicitly).
Bidirectional search

- **Problem:** how do we search backwards from goal?? (cont.)
  - ...
  - for bidirectional search to work well, there must be an efficient way to check whether a given node belongs to the other search tree.
  - select a given search algorithm for each half.
Bidirectional search

1. QUEUE1 <-- path only containing the root;
QUEUE2 <-- path only containing the goal;

2. WHILE both QUEUEs are not empty
   AND QUEUE1 and QUEUE2 do NOT share a state
   DO remove their first paths;
      create their new paths (to all children);
      reject their new paths with loops;
      add their new paths to back;

3. IF QUEUE1 and QUEUE2 share a state
   THEN success;
   ELSE failure;
Bidirectional search

- Completeness: Yes
- Time complexity: \(2 \times O(b^{d/2}) = O(b^{d/2})\)
- Space complexity: \(O(b^{m/2})\)
- Optimality: Yes

- To avoid one by one comparison, we need a hash table of size \(O(b^{m/2})\)
- *If hash table is used, the cost of comparison is \(O(1)\)*
Bidirectional Search

Initial State \hspace{2cm} Final State

\hspace{2cm} d / 2 \hspace{2cm} d
Bidirectional search

- **Bidirectional search merits:**
  - Big difference for problems with branching factor $b$ in both directions
    - A solution of length $d$ will be found in $O(2b^{d/2}) = O(b^{d/2})$
    - For $b = 10$ and $d = 6$, only 2,222 nodes are needed instead of 1,111,111 for breadth-first search
Bidirectional search

- **Bidirectional search issues**
  - *Predecessors* of a node need to be generated
    - Difficult when operators are not reversible
  - What to do if there is no *explicit list of goal* states?
  - For each node: *check if it appeared in the other search*
    - Needs a hash table of $O(b^{d/2})$
  - What is the *best search strategy* for the two searches?
### Comparing uninformed search strategies

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Breadth-first</th>
<th>Uniform cost</th>
<th>Depth-first</th>
<th>Depth-limited</th>
<th>Iterative deepening</th>
<th>Bidirectional (if applicable)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Time</strong></td>
<td>$b^d$</td>
<td>$b^d$</td>
<td>$b^m$</td>
<td>$b^l$</td>
<td>$b^d$</td>
<td>$b^{(d/2)}$</td>
</tr>
<tr>
<td><strong>Space</strong></td>
<td>$b^d$</td>
<td>$b^d$</td>
<td>$b^m$</td>
<td>$b^l$</td>
<td>$b^d$</td>
<td>$b^{(d/2)}$</td>
</tr>
<tr>
<td><strong>Optimal?</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Complete?</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes, if $l \geq d$</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

- $b$ – max branching factor of the search tree
- $d$ – depth of the least-cost solution
- $m$ – max depth of the state-space (may be infinity)
- $l$ – depth cutoff
Summary

• Problem formulation usually requires abstracting away real-world details to define a state space that can be explored using computer algorithms.

• Once problem is formulated in abstract form, complexity analysis helps us picking out best algorithm to solve problem.

• Variety of uninformed search strategies; difference lies in method used to pick node that will be further expanded.

• Iterative deepening search only uses linear space and not much more time than other uniformed search strategies.