Outline

I. Problem formulation

II. Example problems

III. Search algorithms

IV. Breadth-first search

1975 ACM Turing Award Lecture:

* Computer science as empirical inquiry: symbols and search
  Allen Newell and Herbert Simon

* Figures are from the textbook site (or drawn by the instructor) unless the source is specifically cited.
I. Problem-Solving Agent

- A reflex agent cannot operate when the mapping from states to actions becomes too large to store.

```plaintext
if current percepts then action
```

- A problem-solving agent is one kind of goal-based agent:
  - It considers states with no internal structure, i.e., in atomic representations.

- Problems are solved by general-purpose search algorithms.
Sightseeing Trip in Romania
Sightseeing Trip in Romania

How to get to Bucharest from Arad?
How to get to Bucharest from Arad?
Four-Phase Problem Solving

- **Goal formulation**

- **Problem formulation**
  - states: cities
  - action: travel from one city to an adjacent city

- **Search**
  - Find a solution.
    - A sequence of actions to reach the goal

- **Execution**
Search Problem

♦ State space (as a graph)
Search Problem

- **State space** (as a graph)
- **Initial state** (e.g., Arad)
Search Problem

- **State space** (as a graph)
- **Initial state** (e.g., Arad)
- **Goal state(s)** (e.g., Bucharest)

\[ \text{IS-GOAL(Fagaras)} \]
Search Problem

- **State space** (as a graph)
- **Initial state** (e.g., Arad)
- **Goal state(s)** (e.g., Bucharest): IS-GOAL(Fagaras)
- **Actions**
Search Problem

- **State space** (as a graph)
- **Initial state** (e.g., Arad)
- **Goal state(s)** (e.g., Bucharest)
- **Actions**

\[ \text{ACTIONS}(s) : \text{a finite set of actions executable at state } s. \]

\[ \text{ACTIONS}(\text{Arad}) = \{ \text{ToSibiu, ToTimisoara, ToZerind} \} \]
Search Problem

- **State space (as a graph)**
- **Initial state** (e.g., Arad)
- **Goal state(s)** (e.g., Bucharest)
- **Actions**

\[
\text{ACTIONS}(s) \text{: a finite set of actions executable at state } s.
\]

\[
\text{ACTIONS}(Arad) = \{ \text{ToSibiu, ToTimisoara, ToZerind} \}
\]

- **Transition model**

\[
\text{RESULT}(Arad, ToZerind) = \text{Zerind}
\]
Search Problem

- **State space** (as a graph)
- **Initial state** (e.g., Arad)
- **Goal state(s)** (e.g., Bucharest)
- **Actions**

\[ \text{ACTION}(s) \]: a finite set of actions executable at state \( s \).

\[ \text{ACTION}(\text{Arad}) = \{ \text{ToSibiu}, \text{ToTimisoara}, \text{ToZerind} \} \]

- **Transition model**

\[ \text{RESULT}(\text{Arad, ToZerind}) = \text{Zerind} \]

- **Action cost function**

\[ c(s, a, s') \]: cost of applying action \( a \) in state \( s \) to reach state \( s' \).
Search Problem

- **State space** (as a graph)
- **Initial state** (e.g., Arad)
- **Goal state(s)** (e.g., Bucharest)
- **Actions**

**ACTIONS**($s$): a finite set of actions executable at state $s$.

**ACTIONS**(Arad) = \{ToSibiu, ToTimisoara, ToZerind\}

- **Transition model**

**RESULT**(Arad, ToZerind) = Zerind

- **Action cost function**

$c(s, a, s')$: cost of applying action $a$ in state $s$ to reach state $s'$.

- **Solution**: initial state $\leadsto$ goal state (e.g., Arad – Sibiu – Fagaras – Bucharest)
II. Example 1: Vacuum World

State-space graph:

- Actions: Suck, Lft, Right
- Goal: every cell is clean.
II. Example 1: Vacuum World

State-space graph:

- Actions: Suck, Lft, Right
- Goal: every cell is clean.
II. Example 1: Vacuum World

State-space graph:

- Actions: *Suck, Lft, Right*
- Goal: every cell is clean.

Agent in either cell

\[ 2 \cdot 2 \]

Left cell has dirt or not
II. Example 1: Vacuum World

State-space graph:

- Actions: Suck, Lft, Right
- Goal: every cell is clean.

Agent in either cell

\[ 2 \cdot 2 \cdot 2 = 8 \text{ states} \]

Left cell has dirt or not
Right cell has dirt or not
II. Example 1: Vacuum World

State-space graph:

- Actions: Suck, Lft, Right
- Goal: every cell is clean.
- Cost: 1 for each action

\[ 2 \cdot 2 \cdot 2 = 8 \text{ states} \]
Example 2: 8-Puzzle

![Initial state]

![Goal state]

Actions: ways of sliding a tile (adjacent to the blank space).

Left, Right, Up, Down

Applied to the right neighbor of the blank space
Example 2: 8-Puzzle

Actions: ways of sliding a tile (adjacent to the blank space).

Only two possible actions at the initial state: Right and Down.

Left, Right, Up, Down

Applied to the right neighbor of the blank space
Example 2: 8-Puzzle

Initial state

Goal state

Actions: ways of sliding a tile (adjacent to the blank space).

Cost 1 for each action.

Only two possible actions at the initial state: Right and Down.

Left, Right, Up, Down

Applied to the right neighbor of the blank space
Solution to an 8-Puzzle

Initial state

Goal state
Solution to an 8-Puzzle

Initial state

Goal state
Inversions in an 8-Puzzle

State in which all the tiles appear in the correct order.

Every tile with a smaller number should appear either above or to the left of any tile with a larger number.

Every violation of the order is called an inversion.
Inversions in an 8-Puzzle

State in which all the tiles appear in the correct order.

Every tile with a smaller number should appear either above or to the left of any tile with a larger number.

0 inversion

Every violation of the order is called an inversion.
Inversions in an 8-Puzzle

State in which all the tiles appear in the correct order.

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

Every tile with a smaller number should appear either above or to the left of any tile with a larger number.

Every violation of the order is called an inversion.

How many inversions are in the goal state?

```
1 2 3
8 4
7 6 5
```
Inversions in an 8-Puzzle

State in which all the tiles appear in the correct order.

Every tile with a smaller number should appear either above or to the left of any tile with a larger number.

Every violation of the order is called an inversion.

How many inversions are in the goal state?

0 inversion

6 comes before 5
7 comes before 5
7 comes before 6
8 comes before 4
8 comes before 5
8 comes before 6
8 comes before 7
Inversions in an 8-Puzzle

State in which all the tiles appear in the correct order.

Every tile with a smaller number should appear either above or to the left of any tile with a larger number.

Every violation of the order is called an inversion.

How many inversions are in the goal state?

0 inversion

7 inversions!
Solvability of an 8-Puzzle

**Theorem** An 8-puzzle with an initial state and a goal state is solvable if and only if the two states differ by an *even* number of inversions.
Theorem  An 8-puzzle with an initial state and a goal state is solvable if and only if the two states differ by an even number of inversions.

If we fix the goal state at

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

(7 inversions)

the puzzle is solvable if and only if the initial state has an odd number of inversions.
Theorem  An 8-puzzle with an initial state and a goal state is solvable if and only if the two states differ by an \textit{even} number of inversions.

If we fix the goal state at \[
\begin{array}{ccc}
1 & 2 & 3 \\
8 & 4 & 7 \\
6 & 5 & \end{array}
\] (7 inversions), the puzzle is solvable if and only if the initial state has an \textit{odd} number of inversions.

Initial state 1:

\[
\begin{array}{ccc}
4 & 1 & 2 \\
3 & 5 & 6 \\
8 & 7 & \end{array}
\]
Theorem  An 8-puzzle with an initial state and a goal state is solvable if and only if the two states differ by an \textit{even} number of inversions.

If we fix the goal state at \[
1 \quad 2 \quad 3 \\
8 \quad 4 \\
7 \quad 6 \quad 5
\] the puzzle is solvable if and only if the initial state has an \textit{odd} number of inversions.

Initial state 1:
\[
4 \quad 1 \quad 2 \\
3 \quad 5 \\
8 \quad 6 \quad 7
\] 5 inversions:
- 4 comes before 1, 2, 3
- 8 comes before 6, 7
Solvability of an 8-Puzzle

**Theorem** An 8-puzzle with an initial state and a goal state is solvable if and only if the two states differ by an *even* number of inversions.

If we fix the goal state at  

\[
\begin{array}{ccc}
1 & 2 & 3 \\
8 & 4 & \\
7 & 6 & 5 \\
\end{array}
\]

(7 inversions)

the puzzle is solvable if and only if the initial state has an *odd* number of inversions.

**Initial state 1:**

\[
\begin{array}{ccc}
4 & 1 & 2 \\
3 & 5 & \\
8 & 6 & 7 \\
\end{array}
\]

5 inversions:  

- 4 comes before 1, 2, 3
- 8 comes before 6, 7

Solvable
Solvability of an 8-Puzzle

**Theorem** An 8-puzzle with an initial state and a goal state is solvable if and only if the two states differ by an *even* number of inversions.

Initial state 1:

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

5 inversions: 4 comes before 1, 2, 3
8 comes before 6, 7

Solvable

Initial state 2:

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

(7 inversions)

If we fix the goal state at the puzzle is solvable if and only if the initial state has an *odd* number of inversions.
Solvability of an 8-Puzzle

**Theorem** An 8-puzzle with an initial state and a goal state is solvable if and only if the two states differ by an *even* number of inversions.

If we fix the goal state at the puzzle is solvable if and only if the initial state has an *odd* number of inversions.

**Initial state 1:**

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

5 inversions:
- 4 comes before 1, 2, 3
- 8 comes before 6, 7

**Initial state 2:**

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

6 inversions:
- 4 comes before 1, 2, 3
- 5 comes before 3
- 8 comes before 6, 7
Solvability of an 8-Puzzle

**Theorem**  An 8-puzzle with an initial state and a goal state is solvable if and only if the two states differ by an *even* number of inversions.

If we fix the goal state at the puzzle is solvable if and only if the initial state has an *odd* number of inversions.

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

5 inversions:  $\iff$ Solvable

4 comes before 1, 2, 3
8 comes before 6, 7

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

6 inversions:  $\iff$ Not solvable

4 comes before 1, 2, 3
5 comes before 3
8 comes before 6, 7

(7 inversions)
The 8-Queens Problem

- **Goal:** A placement of 8 queens on the chess board in which no queen *attacks* another.
  
  Same row, column, or diagonal

- **Initial state:** no queen on the board.
The 8-Queens Problem

- Goal: A placement of 8 queens on the chess board in which no queen attacks another.
  - Same row, column, or diagonal
- Initial state: no queen on the board.

Close to a solution
The 8-Queens Problem

Goal: A placement of 8 queens on the chess board in which no queen attacks another.

- Initial state: no queen on the board.

Close to a solution
The 8-Queens Problem

- Goal: A placement of 8 queens on the chess board in which no queen attacks another.
  - Same row, column, or diagonal

- Initial state: no queen on the board.

Constraint satisfaction!

Close to a solution
The 8-Queens Problem

- Goal: A placement of 8 queens on the chess board in which no queen attacks another.
  - Same row, column, or diagonal

- Initial state: no queen on the board.

Constrain satisfaction!

The $n$-queens problem
Place $n$ queens on an $n \times n$ chess board so that no queen attacks another.
Knuth’s Conjecture (1964)

Any integer > 4 can be reached from 4 via a sequence of square root, floor, and factorial operations.

\[ \sqrt{\sqrt{\sqrt{\sqrt{(4!)!}}}} = 5 \]

Donald Knuth (Stanford)
“father of the analysis of algorithms”
ACM Turing Award (1974)
National Medal of Science (1979)

* Photo from https://amturing.acm.org/byyear.cfm.
Knuth’s Conjecture (1964)

Any integer \( > 4 \) can be reached from 4 via a sequence of square root, floor, and factorial operations.

\[
\sqrt{\sqrt{\sqrt{(4!)!}}} = 5
\]

- States: positive real numbers.
- Initial state: 4.
- Goal state: the desired integer \( > 4 \).
- Actions: square root, floor, or factorial operation.
- Action cost: 1.

Donna Knuth (Stanford)
“father of the analysis of algorithms”
ACM Turing Award (1974)
National Medal of Science (1979)
Real-World Problems

- Route finding (e.g., from Ames, IA to Mountain View, CA)
- Traveling salesman problem

* Figure from [http://www.crpc.rice.edu/CRPC/newsletters/sum98/news_tsp.html](http://www.crpc.rice.edu/CRPC/newsletters/sum98/news_tsp.html).
Real-World Problems

- Route finding (e.g., from Ames, IA to Mountain View, CA)
- Traveling salesman problem
- VLSI layout
- Robot navigation
- Autonomous assembly sequencing (e.g., protein design)

* Figure from [http://www.crpc.rice.edu/CRPC/newsletters/sum98/news_tsp.html](http://www.crpc.rice.edu/CRPC/newsletters/sum98/news_tsp.html).
III. Tree Search

Superimpose a search tree over the state space graph and find a path from the initial state to the goal state.
III. Tree Search

Superimpose a search tree over the state space graph and find a path from the initial state to the goal state.
III. Tree Search

Superimpose a search tree over the state space graph and find a path from the initial state to the goal state.
III. Tree Search

Superimpose a search tree over the state space graph and find a path from the initial state to the goal state.
III. Tree Search

Superimpose a search tree over the state space graph and find a path from the initial state to the goal state.
III. Tree Search

Superimpose a search tree over the state space graph and find a path from the initial state to the goal state.
Generated Search Trees

Arad
Bucharest
Which node on the frontier to expand next?
Best-First Search

Choose a node $n$ with minimum value of some evaluation function $f(n)$.

- Return it if its state is a goal state.
- Otherwise generate child nodes and add them to the frontier.
Best-First Search

Choose a node \( n \) with minimum value of some \textit{evaluation function} \( f(n) \).

- Return it if its state is a goal state.
- Otherwise generate child nodes and add them to the frontier.

Node structure:

<table>
<thead>
<tr>
<th>STATE</th>
<th>PARENT</th>
<th>ACTION</th>
<th>PATH-COST</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>( UP )</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Node:

- STATE: 6 7 4
- ACTION = \( UP \)
- PATH-COST = 4
Best-First Search

Choose a node $n$ with minimum value of some *evaluation function* $f(n)$.
- Return it if its state is a goal state.
- Otherwise generate child nodes and add them to the frontier.

Node structure:
- `node.STATE`
- `node.PARENT`
- `node.ACTION`: action applied at the parent node to generate this node
- `node.PATH-COST`: total cost of the past from the initial state to this node.
Best-First Search

Choose a node $n$ with minimum value of some \textit{evaluation function} $f(n)$.

- Return it if its state is a goal state.
- Otherwise generate child nodes and add them to the frontier.

Node structure:

- $node$.\texttt{STATE}
- $node$.\texttt{PARENT}
- $node$.\texttt{ACTION}: action applied at the parent node to generate this node
- $node$.\texttt{PATH-COST}: total cost of the past from the initial state to this node.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|}
\hline
1 & 2 & 3 \\
\hline
6 & 7 & 4 \\
\hline
8 & 5 &   \\
\hline
\end{tabular}
\end{table}
More on Data Structure

Frontier:

- IS-EMPTY(frontier)
- POP(frontier)
- TOP(frontier)
- ADD(node, frontier)

Three kinds of queues:

- A priority queue pops the node with the minimum cost.
- A FIFO queue pops the first added node (used in BFS).
- A LIFO queue pops the most recently added node (used in DFS).
Best-First Search Algorithm

function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
node ← NODE(STATE=problem.INITIAL)
frontier ← a priority queue ordered by f, with node as an element
reached ← a lookup table, with one entry with key problem.INITIAL and value node
while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
        s ← child.STATE
        if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
            reached[s] ← child
            add child to frontier
    return failure

function EXPAND(problem, node) yields nodes
s ← node.STATE
for each action in problem.ACTIONS(s) do
    s' ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s')
yield NODE(STATE=s', PARENT=node, ACTION=action, PATH-COST=cost)
Best-First Search Algorithm

function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
node ← NODE(STATE=problem.INITIAL)
frontier ← a priority queue ordered by f, with node as an element // states on the frontier
reached ← a lookup table, with one entry with key problem.INITIAL and value node // states that have been reached
while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
        s ← child.STATE
        if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
            reached[s] ← child
            add child to frontier
    return failure

function EXPAND(problem, node) yields nodes
s ← node.STATE
for each action in problem.ACTIONS(s) do
    s' ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s')
yield NODE(STATE=s', PARENT=node, ACTION=action, PATH-COST=cost)
Best-First Search Algorithm

function \textsc{Best-First-Search}(problem, f) \textbf{returns} a solution node or \texttt{failure}
\begin{itemize}
  \item node \leftarrow \textsc{Node}(\texttt{State}=problem.\texttt{Initial})
  \item frontier \leftarrow \texttt{a priority queue} ordered by \(f\), with \texttt{node} as an element \hspace{1cm} // states on the frontier
  \item reached \leftarrow \texttt{a lookup table}, with one entry with key \texttt{problem.\texttt{Initial}} and value \texttt{node} \hspace{1cm} // states \hspace{1cm} // that have been reached
\end{itemize}
\begin{algorithmic}
  \While{\not \textsc{Is-Empty(frontier)}}
    \State \texttt{node} \leftarrow \textsc{Pop(frontier)}
    \If{\texttt{problem.Is-Goal(node.\texttt{State})}}
      \State \textbf{return} \texttt{node}
    \EndIf
    \For{\texttt{each} \texttt{child} \texttt{in} \textsc{Expand}(\texttt{problem}, \texttt{node})}
      \State \texttt{s} \leftarrow \texttt{child.\texttt{State}}
      \If{\texttt{s} \texttt{is not in} \texttt{reached} \texttt{or} \texttt{child.\texttt{Path-Cost} < reached[s].\texttt{Path-Cost}}}
        \State \texttt{reached}[\texttt{s}] \leftarrow \texttt{child}
        \State \texttt{add} \texttt{child} \texttt{to} \texttt{frontier}
      \EndIf
    \EndFor
  \EndWhile
  \State \textbf{return} \texttt{failure}
\end{algorithmic}

function \textsc{Expand}(problem, node) \textbf{yields} nodes
\begin{itemize}
  \item \texttt{s} \leftarrow \texttt{node.\texttt{State}}
  \item \textbf{for each} \texttt{action} \texttt{in} \texttt{problem.\texttt{Actions}(s)} \texttt{do}
    \State \texttt{s}' \leftarrow \texttt{problem.\texttt{Result}(s, action)}
    \State \texttt{cost} \leftarrow \texttt{node.\texttt{Path-Cost} + problem.\texttt{Action-Cost}(s, action, s')}
  \EndFor
  \State \textbf{yield} \texttt{Node(\texttt{State}=s', \texttt{Parent}=node, \texttt{Action}=action, \texttt{Path-Cost}=cost)}
\end{itemize}

Can implement BFS and DFS.
Best-First Search Algorithm

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
    node ← NODE(STATE=problem.INITIAL)
    frontier ← a priority queue ordered by f, with node as an element // states on the frontier
    reached ← a lookup table, with one entry with key problem.INITIAL and value node // states that have been reached
    while not IS-EMPTY(frontier) do
        node ← POP(frontier);
        if problem.IS-GOAL(node.STATE) then return node
        for each child in EXPAND(problem, node) do
            s ← child.STATE
            if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
                reached[s] ← child // state s is reached at the node child.
                add child to frontier
        return failure

function EXPAND(problem, node) yields nodes
    s ← node.STATE
    for each action in problem.ACTIONS(s) do
        s' ← problem.RESULT(s, action)
        cost ← node.PATH-COST + problem.ACTION-COST(s, action, s')
        yield NODE(STATE=s', PARENT=node, ACTION=action, PATH-COST=cost)
```

Can implement BFS and DFS.
Repeated State
Repeated State
Repeated State
Repeated State

- Failure to detect repeated states can turn a solvable problem into an unsolvable one.
Repeated State

- Failure to detect repeated states can turn a solvable problem into an unsolvable one.
- Keep only the best path to each state.
Performance Measures

- **Completeness**: Is the algorithm guaranteed to find a solution whenever one exists, and to report failure otherwise?

  The state space may be infinite!

- **Cost optimality**: Does it find a solution with the lowest path cost of all solutions?

- **Time complexity**: Physical time or the number of states and actions.

- **Space complexity**: Memory needed for the search.

  \[ |V| + |E| \]
Depth and Branching Factor

- The measure in terms of $|V| + |E|$ is appropriate when the graph is *explicit*. 
Depth and Branching Factor

- The measure in terms of $|V| + |E|$ is appropriate when the graph is explicit.

- But often *implicit* graph representation of a state space in AI:
Depth and Branching Factor

- The measure in terms of $|V| + |E|$ is appropriate when the graph is explicit.

- But often *implicit* graph representation of a state space in AI:
  - the initial state
  - actions
  - a transition model
Depth and Branching Factor

- The measure in terms of $|V| + |E|$ is appropriate when the graph is explicit.

- But often implicit graph representation of a state space in AI:
  - the initial state
  - actions
  - a transition model

- Accordingly, complexity is measured in terms of
  - $d$: depth (number of actions in the optimal solution)
Depth and Branching Factor

- The measure in terms of $|V| + |E|$ is appropriate when the graph is explicit.

- But often implicit graph representation of a state space in AI:
  - the initial state
  - actions
  - a transition model

- Accordingly, complexity is measured in terms of
  - $d$: depth (number of actions in the optimal solution)
  - $m$: maximum number of actions in any path
Depth and Branching Factor

• The measure in terms of $|V| + |E|$ is appropriate when the graph is \textit{explicit}.

• But often \textit{implicit} graph representation of a state space in AI:
  - the initial state
  - actions
  - a transition model

• Accordingly, complexity is measured in terms of
  - $d$: \textit{depth} (number of actions in the optimal solution)
  - $m$: \textit{maximum number of actions} in any path
  - $b$: \textit{branching factor} (number of successors of a node).
Infinite State Space

Knuth’s conjecture: Any integer $> 4$ can be reached from 4 via a sequence of square root, floor, and factorial operations.

$\sqrt[\sqrt[\sqrt[\sqrt{4}]]]{} = 5$
Infinite State Space

Knuth’s conjecture: Any integer \( > 4 \) can be reached from 4 via a sequence of square root, floor, and factorial operations.

\[
\sqrt[3]{\sqrt[4]{\sqrt{4!}}} = 5
\]

**Algorithm** Repeatedly applies the “factorial operator”.

* Photo from [https://amturing.acm.org/byyear.cfm](https://amturing.acm.org/byyear.cfm).
Knuth’s conjecture: Any integer > 4 can be reached from 4 via a sequence of square root, floor, and factorial operations.

Algorithm
Repeatedly applies the “factorial operator”.

\[
\sqrt{\sqrt{\sqrt{\sqrt{(4!)!}}}} = 5
\]
Breadth-First Search

Uninformed search: No clue about how close a state is to the goal.
Breadth-First Search

*Uninformed* search: No clue about how close a state is to the goal.

Expand the root first, then all its successors, next their successors, and so on.
Breadth-First Search

*Uninformed* search: No clue about how close a state is to the goal.

Expand the root first, then all its successors, next their successors, and so on.
Breadth-First Search

*Uninformed* search: No clue about how close a state is to the goal.

Expand the root first, then all its successors, next their successors, and so on.
**Breadth-First Search**

*Uninformed search*: No clue about how close a state is to the goal.

Expand the root first, then all its successors, next their successors, and so on.
Breadth-First Search

*Uninformed* search: No clue about how close a state is to the goal.

Expand the root first, then all its successors, next their successors, and so on.
Breadth-First Search

*Uninformed* search: No clue about how close a state is to the goal.

Expand the root first, then all its successors, next their successors, and so on.

- Systematic search.
- Complete even when the state space is infinite.
- Always finds a solution with a minimum number of actions.
BFS Algorithm

- Can call BEST-FIRST-SEARCH by letting the evaluation function \( f(n) = \text{node depth} \).
- ♠ Not efficient: Use a FIFO queue and adopt early goal test (when a node is generated).

```plaintext
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
node ← NODE(problem.INITIAL)
if problem.IS-GOAL(node.STATE) then return node
frontier ← a FIFO queue, with node as an element
reached ← \{problem.INITIAL\}
while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    for each child in EXPAND(problem, node) do
        s ← child.STATE
        if problem.IS-GOAL(s) then return child
        if s is not in reached then
            add s to reached
            add child to frontier
    return failure
```
BFS on a uniform tree where every node has $b$ successors.

• $b$ nodes at depth 1 generated by the root.

• Each node at depth 1 generates $b$ nodes. $\Rightarrow b^2$ nodes at depth 2.

• And so on.
BFS on a uniform tree where every node has $b$ successors.

- $b$ nodes at depth 1 generated by the root.
- Each node at depth 1 generates $b$ nodes. $\Rightarrow b^2$ nodes at depth 2.
- And so on.

Solution at depth $d$
BFS on a uniform tree where every node has $b$ successors.

- $b$ nodes at depth 1 generated by the root.
- Each node at depth 1 generates $b$ nodes. \( \Rightarrow b^2 \) nodes at depth 2.
- And so on.

Solution at depth $d$ \( \iff \) \#nodes = $1 + b + \cdots + b^d$
Time and Space Complexities

BFS on a uniform tree where every node has $b$ successors.

- $b$ nodes at depth 1 generated by the root.
- Each node at depth 1 generates $b$ nodes. $\Rightarrow b^2$ nodes at depth 2.
- And so on.

Solution at depth $d$ $\iff$ #nodes $= 1 + b + \cdots + b^d$

$$= \frac{b^{d+1} - 1}{b - 1} \quad (\text{assuming } b > 1)$$

branching factor
Time and Space Complexities

BFS on a uniform tree where every node has $b$ successors.

- $b$ nodes at depth 1 generated by the root.
- Each node at depth 1 generates $b$ nodes. $\Rightarrow b^2$ nodes at depth 2.
- And so on.

Solution at depth $d \Longleftrightarrow \text{#nodes} = 1 + b + \cdots + b^d = O(b^d)$

$$= \frac{b^{d+1}-1}{b-1} \text{ (assuming } b > 1)$$
Time and Space Complexities

BFS on a uniform tree where every node has $b$ successors.

- $b$ nodes at depth 1 generated by the root.
- Each node at depth 1 generates $b$ nodes. $\Rightarrow b^2$ nodes at depth 2.
- And so on.

Solution at depth $d$ $\iff$ 
$\# nodes = 1 + b + \cdots + b^d = O(b^d)$

$$= \frac{b^{d+1}-1}{b-1} \quad \text{(assuming } b > 1)$$

Time & space complexities
(since every node remains in memory)
Time and Space Complexities

BFS on a uniform tree where every node has $b$ successors.

- $b$ nodes at depth 1 generated by the root.
- Each node at depth 1 generates $b$ nodes. $\Rightarrow b^2$ nodes at depth 2.
- And so on.

Solution at depth $d$ $\quad \#\text{nodes} = 1 + b + \cdots + b^d = O(b^d)$

$$= \frac{b^{d+1}-1}{b-1} \quad \text{(assuming } b > 1)$$

Time & space complexities
(since every node remains in memory)

• Only small search problems are solvable due to exponential time complexity.
Time and Space Complexities

BFS on a uniform tree where every node has $b$ successors.

- $b$ nodes at depth 1 generated by the root.
- Each node at depth 1 generates $b$ nodes. ⇒ $b^2$ nodes at depth 2.
- And so on.

Solution at depth $d$ $\iff$ \[ \text{#nodes} = 1 + b + \cdots + b^d = O(b^d) \]
\[ = \frac{b^{d+1} - 1}{b - 1} \quad (\text{assuming } b > 1) \]

Time & space complexities
(since every node remains in memory)

- Only small search problems are solvable due to exponential time complexity.
- Memory is a bigger issue than time.
## Time and Memory Requirements for BFS

Tree search: $O(b^d)$

<table>
<thead>
<tr>
<th>Depth</th>
<th>Nodes</th>
<th>Time</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>110</td>
<td>.11 milliseconds</td>
<td>107 KB</td>
</tr>
<tr>
<td>4</td>
<td>11,110</td>
<td>11 milliseconds</td>
<td>10.6 MB</td>
</tr>
<tr>
<td>6</td>
<td>$10^6$</td>
<td>1.1 seconds</td>
<td>1 GB</td>
</tr>
<tr>
<td>8</td>
<td>$10^8$</td>
<td>2 minutes</td>
<td>103 GB</td>
</tr>
<tr>
<td>10</td>
<td>$10^{10}$</td>
<td>3 hours</td>
<td>10 TB</td>
</tr>
<tr>
<td>12</td>
<td>$10^{12}$</td>
<td>13 days</td>
<td>1 PB</td>
</tr>
<tr>
<td>14</td>
<td>$10^{14}$</td>
<td>3.5 years</td>
<td>99 PB</td>
</tr>
<tr>
<td>16</td>
<td>$10^{16}$</td>
<td>350 years</td>
<td>10 EB</td>
</tr>
</tbody>
</table>

$b = 10$
# Time and Memory Requirements for BFS

Tree search: $O(b^d)$

<table>
<thead>
<tr>
<th>Depth</th>
<th>Nodes</th>
<th>Time</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>110</td>
<td>.11 milliseconds</td>
<td>107 KB</td>
</tr>
<tr>
<td>4</td>
<td>11,110</td>
<td>11 milliseconds</td>
<td>10.6 MB</td>
</tr>
<tr>
<td>6</td>
<td>$10^6$</td>
<td>1.1 seconds</td>
<td>1 GB</td>
</tr>
<tr>
<td>8</td>
<td>$10^8$</td>
<td>2 minutes</td>
<td>103 GB</td>
</tr>
<tr>
<td>10</td>
<td>$10^{10}$</td>
<td>3 hours</td>
<td>10 TB</td>
</tr>
<tr>
<td>12</td>
<td>$10^{12}$</td>
<td>13 days</td>
<td>1 PB</td>
</tr>
<tr>
<td>14</td>
<td>$10^{14}$</td>
<td>3.5 years</td>
<td>99 PB</td>
</tr>
<tr>
<td>16</td>
<td>$10^{16}$</td>
<td>350 years</td>
<td>10 EB</td>
</tr>
</tbody>
</table>

$b = 10$

Graph search is preferred since its time $b d$.)

d and space are proportional to the size of the state space (often less than $O(b^d)$).
Uniform-Cost Search (Dijkstra’s Algorithm)

function UNIFORM-COST-SEARCH(problem) returns a solution node, or failure
return BEST-FIRST-SEARCH(problem, PATH-COST)

- Spreads out in waves of uniform path-cost.
Uniform-Cost Search (Dijkstra’s Algorithm)

function UNIFORM-COST-SEARCH(problem) returns a solution node, or failure
return BEST-FIRST-SEARCH(problem, PATH-COST)

- Spreads out in waves of uniform path-cost.
Uniform-Cost Search (Dijkstra’s Algorithm)

function UNIFORM-COST-SEARCH(problem) returns a solution node, or failure
return BEST-FIRST-SEARCH(problem, PATH-COST)

- Spreads out in waves of uniform path-cost.

Sibiu 99 Fagaras
80 Rimnicu Vilcea
Rimnicu Vilcea 97 Pitesti
211 Bucharest

Rimnicu Vilcea (80)
Fagaras (99)
Uniform-Cost Search (Dijkstra’s Algorithm)

function UNIFORM-COST-SEARCH(problem) returns a solution node, or failure
return BEST-FIRST-SEARCH(problem, PATH-COST)

• Spreads out in waves of uniform path-cost.
Uniform-Cost Search (Dijkstra’s Algorithm)

```plaintext
function UNIFORM-COST-SEARCH(problem) returns a solution node, or failure
return BEST-FIRST-SEARCH(problem, PATH-COST)
```

- Spreads out in waves of uniform path-cost.
Uniform-Cost Search (Dijkstra’s Algorithm)

function UNIFORM-COST-SEARCH(problem) returns a solution node, or failure
return BEST-FIRST-SEARCH(problem, PATH-COST)

- Spreads out in waves of uniform path-cost.
Uninformed Search: Completeness and Complexity

♦ **Completeness**: systematic exploration of all paths – no chance of being trapped in one.
Uninformed Search: Completeness and Complexity

- **Completeness**: systematic exploration of all paths – no chance of being trapped in one.

- **Optimality**: following from that of Dijkstra’s algorithm.
Uninformed Search: Completeness and Complexity

- **Completeness**: systematic exploration of all paths – no chance of being trapped in one.

- **Optimality**: following from that of Dijkstra’s algorithm.

- **Complexity**

  \[ O\left(b^{1+\lfloor C^*/\epsilon \rfloor}\right) \]

  \(C^*\): cost of the optimal solution

  \(\epsilon\): lower bound on the costs of all actions
Uninformed Search: Completeness and Complexity

- **Completeness**: systematic exploration of all paths – no chance of being trapped in one.

- **Optimality**: following from that of Dijkstra’s algorithm.

- **Complexity**

  $$O\left(b^{1+\left\lfloor \frac{C^*}{\epsilon} \right\rfloor}\right)$$

  $C^*$: cost of the optimal solution

  $\epsilon$: lower bound on the costs of all actions

  $\gg b^d$ possible.
Uninformed Search: Completeness and Complexity

♦ Completeness: systematic exploration of all paths – no chance of being trapped in one.

♦ Optimality: following from that of Dijkstra’s algorithm.

♦ Complexity

\[ O\left(b^{1+\left\lfloor C^*/\epsilon\right\rfloor}\right) \]

\( C^* \): cost of the optimal solution
\( \epsilon \): lower bound on the costs of all actions

\[ b^d \] possible.

\[ = b^{d+1} \] if all actions have the same cost.