Chapter 3

Solving problem by Searching
Outlines

• Problem-solving agents
• Example Problem
• Searching for Solution
• Uninformed Search Strategies
• Avoiding Repeated States
• Searching with partial Information
Problem Solving Agent

Problem Solving Agent:
- An agent with several options can first examine different possible sequences of actions to choose the best sequence.

Problem Solving Environment:
- static [learning]
- observable [logic]
- deterministic [uncertainty]
- discrete [uncertainty, logic]

This is an abstraction of a real problem.
What does a program that represents a problem-solving agent look like?
Search Problem

- State space
  - Initial state
  - Successor function
- Goal test
- Path cost
Romania

• What qualifies as a solution?
  – You can/cannot reach Bucharest by 1:00
  – You can reach Bucharest in \( x \) hours
  – The shortest path to Bucharest passes through these cities
  – The sequence of cities in the shortest path from Arad to Bucharest is ________
  – The actions one takes to travel from Arad to Bucharest along the shortest path
Romania

• What additional information does one need?
  – A map
More concrete problem definition

- A state space: Which cities could you be in
- An initial state: Which city do you start from
- A goal state: Which city do you aim to reach
- A function defining state transitions: When in city foo, the following cities can be reached
- A function defining the “cost” of a state sequence: How long does it take to travel through a city sequence
More concrete problem definition

A state space

An initial state

A goal state

A function defining state transitions

A function defining the “cost” of a state sequence

Choose a representation

Choose an element from the representation

Create

\[
\text{goal\_function}(\text{state}) \text{ such that } \text{TRUE is returned upon reaching goal}
\]

\[
\text{successor\_function}(\text{state}) = \{ \langle \text{action, state} \rangle, \langle \text{action, state} \rangle, \ldots \}
\]

\[
\text{cost (sequence)} = \text{number}
\]
State Space

• Real world is absurdly complex \( \Rightarrow \) state space must be abstracted for problem solving
  – (Abstract) state = set of real states
  – (Abstract) action = complex combination of real actions, e.g., “Arad\(\Rightarrow\)Zerind” represents a complex set of possible routes, detours, rest stops, etc.
  – (Abstract) solution = set of real paths that are solutions in the real world

• Each abstract action should be “easier” than the original problem and should permit expansion to a more detailed solution
Important notes about this example

- **Static environment** (available states, successor function, and cost functions don’t change)
- **Observable** (the agent knows where it is… percept == state)
- **Discrete** (the actions are discrete)
- **Deterministic** (successor function is always the same)
Problem formulation

A problem is defined by four items:

* initial state  
  e.g., “at Arad”  

* operators (or successor function $S(x)$)  
  e.g., Arad → Zerind  
  Arad → 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
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)
state ← 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: 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.

<table>
<thead>
<tr>
<th>Single-state, start in #5.</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Multiple-state, start in ${1, 2, 3, 4, 5, 6, 7, 8}$</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>e.g., Right goes to ${2, 4, 6, 8}$.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Contingency, start in #5</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Murphy's Law: Suck can dirty a clean carpet</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Local sensing: dirt, location only.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Solution??</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Solution??</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
Exploratory search is an old idea: The Labyrinth and the Ariadne Thread

According to Greek mythology, Theseus came to Crete to slay the Minotaur, a monster who lived in a Labyrinth. Ariadne gave Theseus a ball of yarn which he unwound as he entered the Labyrinth. After killing the Minotaur, Theseus traced the thread back to the entrance of the Labyrinth, rejoined Ariadne, and successfully escaped Crete.
Example: 8-Puzzle

Initial state

Goal state

Search is about the exploration of alternatives
15-Puzzle

Introduced in 1878 by Sam Loyd, who dubbed himself “America’s greatest puzzle-expert”
Sam Loyd offered $1,000 of his own money to the first person who would solve the following problem:
But no one ever won the prize!!
### 8-Puzzle: State Space

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

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

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

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>4</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
8-Puzzle: Successor Function

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

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

```
8 2 7
3 4 6
5 1
```
Stating a Problem as a Search Problem

- State space $S$
- Successor function:
  \[ x \in S \rightarrow \text{successors}(x) \in 2^S \]
- Arc cost
- Initial state $s_0$
- Goal test:
  \[ x \in S \rightarrow \text{goal?}(x) = \text{T or F} \]
State Graph

- It is defined as follows:
  - Each state is represented by a distinct node
  - An arc connects a node $s$ to a node $s'$ if $s' \in \text{SUCCESSORS}(s)$
- The state graph may contain more than one connected component
Solution to the Search Problem

- A solution is a path connecting the initial to a goal node (any one)
Solution to the Search Problem

- A solution is a path connecting the initial to a goal node (any one)
- The cost of a path is the sum of the edge costs along this path
- An optimal solution is a solution path of minimum cost
- There might be no solution!
How big is the state space of the \((n^2-1)\)-puzzle?

- 8-puzzle $\to 9! = 362,880$ states
- 15-puzzle $\to 16! \approx 1.3 \times 10^{12}$ states
- 24-puzzle $\to 25! \approx 10^{25}$ states

But only half of these states are reachable from any given state
Wlg, let the goal be:

Let \( n_i \) be the number of tiles \( j < i \) that appear after tile \( i \) (from left to right and top to bottom).

\[ N = n_2 + n_3 + \ldots + n_{15} + \text{row number of empty tile} \]

\[ n_2 = 0 \quad n_3 = 0 \quad n_4 = 0 \]
\[ n_5 = 0 \quad n_6 = 0 \quad n_7 = 1 \]
\[ n_8 = 1 \quad n_9 = 1 \quad n_{10} = 4 \]
\[ n_{11} = 0 \quad n_{12} = 0 \quad n_{13} = 0 \]
\[ n_{14} = 0 \quad n_{15} = 0 \]

\[ \Rightarrow N = 7 + 4 \]
Proposition: \((N \text{ mod } 2)\) is invariant under any legal move of the empty tile

Proof:

• Any horizontal move of the empty tile leaves \(N\) unchanged
• A vertical move of the empty tile changes \(N\) by an even increment

\[
s = \begin{array}{cccc}
1 & 2 & 3 & 4 \\
5 & 6 & \boxed{7} & \\
9 & 10 & 11 & 8 \\
13 & 14 & 15 & 12 \\
\end{array}
\]

\[
s' = \begin{array}{cccc}
1 & 2 & 3 & 4 \\
5 & 6 & 11 & 7 \\
9 & 10 & \boxed{8} & \\
13 & 14 & 15 & 12 \\
\end{array}
\]

\[N(s') = N(s) + 3 - 1\]
Proposition: \((N \mod 2)\) is invariant under any legal move of the empty tile

- For a goal state \(g\) to be reachable from a state \(s\), a necessary condition is that \(N(g)\) and \(N(s)\) have the same parity

- It can be shown that this is also a sufficient condition

- The state graph consists of two connected components of equal size
So, the second state is not reachable from the first, and Sam Loyd took no risk with his money ...
What is the Actual State Space?

a) The set of all states?
   [e.g., a set of 16! states for the 15-puzzle]

b) The set of all states from which a given goal state is reachable?
   [e.g., a set of 16!/2 states for the 15-puzzle]

c) The set of all states reachable from a given initial state?

In general, the answer is a)
What is the Actual State Space?

a) The set of all states?  
   [e.g., a set of 16! states for the 15-puzzle]

b) The set of all states from which a given goal state is reachable?  
   [e.g., a set of 16!/2 states for the 15-puzzle]

c) The set of all states reachable from a given initial state?  

In general, the answer is a)

But a fast test determining whether a state is reachable from another is very useful, as search-based problem solvers are often very inefficient when a problem has no solution.

More on this in future lectures ...
Stating a Problem as a Search Problem

- State space $S$
- Successor function: $x \in S \rightarrow \text{SUCCESSORS}(x) \in 2^S$
- Arc cost
- Initial state $s_0$
- Goal test:
  $x \in S \rightarrow \text{GOAL?}(x) = \text{T or F}$
- A solution is a path joining the initial to a goal node

$S$

Diagram:

- Nodes represent states
- Arrows represent transitions
- Red nodes indicate goal states
Searching the State Space

- Often it is not feasible to build a complete representation of the state graph.
8-, 15-, 24-Puzzles

8-puzzle $\rightarrow 362,880$ states

15-puzzle $\rightarrow 1.3 \times 10^{12}$ states

24-puzzle $\rightarrow 10^{25}$ states

0.036 sec

$< 4$ hours

$> 10^9$ years

100 millions states/sec
Searching the State Space

- Often it is not feasible to build a complete representation of the state graph.
- A problem solver must construct a solution by exploring a small portion of the graph.
Searching the State Space
Searching the State Space

Search tree
Searching the State Space
Searching the State Space

Search tree
Searching the State Space
Searching the State Space
Simple Problem-Solving-Agent Algorithm

1. $s_0 \leftarrow$ sense/read initial state
2. $GOAL? \leftarrow$ select/read goal test
3. Succ $\leftarrow$ select/read successor function
4. solution $\leftarrow$ search($s_0$, $GOAL?$, Succ)
5. perform(solution)
State Space

- Each state is an abstract representation of a collection of possible worlds sharing some crucial properties and differing on non-important details only.

  E.g.: In assembly planning, a state does not define exactly the absolute position of each part.

- The state space is discrete. It may be finite, or infinite.
Successor Function

- It implicitly represents all the actions that are feasible in each state
Successor Function

- It implicitly represents all the actions that are feasible in each state
- Only the results of the actions (the successor states) and their costs are returned by the function
- The successor function is a “black box”: its content is unknown

E.g., in assembly planning, the function does not say if it only allows two sub-assemblies to be merged or if it makes assumptions about subassembly stability
Path Cost

- An arc cost is a positive number measuring the “cost” of performing the action corresponding to the arc, e.g.:
  - 1 in the 8-puzzle example
  - expected time to merge two sub-assemblies

- We will assume that for any given problem the cost $c$ of an arc always verifies: $c \geq \varepsilon > 0$, where $\varepsilon$ is a constant
Path Cost

- An arc cost is a positive number measuring the “cost” of performing the action corresponding to the arc, e.g.:
  - 1 in the 8-puzzle example
  - expected time to merge two sub-assemblies

- We will assume that for any given problem the cost $c$ of an arc always verifies: $c \geq \varepsilon > 0$, where $\varepsilon$ is a constant

[This condition guarantees that, if path becomes arbitrarily long, its cost also becomes arbitrarily large]

Why is this needed?
Goal State

- It may be explicitly described:

- or partially described:

- or defined by a condition,
  e.g., the sum of every row, of every column, and of every diagonals equals 30

(“a” stands for “any”)
Other examples
8-Queens Problem

Place 8 queens in a chessboard so that no two queens are in the same row, column, or diagonal.

A solution

Not a solution
Formulation #1

- **States**: all arrangements of 0, 1, 2, ..., or 8 queens on the board
- **Initial state**: 0 queen on the board
- **Successor function**: each of the successors is obtained by adding one queen in an empty square
- **Arc cost**: irrelevant
- **Goal test**: 8 queens are on the board, with no two of them attacking each other

→ $64 \times 63 \times \ldots \times 53 \sim 3 \times 10^{14}$ states
Formulation #2

- **States**: all arrangements of $k = 0, 1, 2, ..., \text{or } 8$ queens in the $k$ leftmost columns with no two queens attacking each other
- **Initial state**: 0 queen on the board
- **Successor function**: each successor is obtained by adding one queen in any square that is not attacked by any queen already in the board, in the leftmost empty column
- **Arc cost**: irrelevant
- **Goal test**: 8 queens are on the board

→ 2,057 states
n-Queens Problem

- A solution is a goal node, not a path to this node (typical of design problem)
- Number of states in state space:
  - 8-queens $\rightarrow$ 2,057
  - 100-queens $\rightarrow$ $10^{52}$
- But techniques exist to solve n-queens problems efficiently for large values of $n$
  They exploit the fact that there are many solutions well distributed in the state space
Path Planning

What is the state space?
Formulation #1

Cost of one horizontal/vertical step = 1
Cost of one diagonal step = $\sqrt{2}$
Optimal Solution

This path is the shortest in the discretized state space, but not in the original continuous space.
Formulation #2

sweep-line
Formulation #2
States
Successor Function
A path-smoothing post-processing step is usually needed to shorten the path further.
Formulation #3

Cost of one step: length of segment
Formulation #3

Cost of one step: length of segment

Visibility graph
The shortest path in this state space is also the shortest in the original continuous space.
Assembly (Sequence) Planning
Possible Formulation

- **States**: All decompositions of the assembly into subassemblies (subsets of parts in their relative placements in the assembly)
- **Initial state**: All subassemblies are made of a single part
- **Goal state**: Un-decomposed assembly
- **Successor function**: Each successor of a state is obtained by merging two subassemblies (the successor function must check if the merging is feasible: collision, stability, grasping, ...)
- **Arc cost**: 1 or time to carry the merging
A Portion of State Space
But the formulation rules out “non-monotonic” assemblies
But the formulation rules out “non-monotonic” assemblies
But the formulation rules out “non-monotonic” assemblies
But the formulation rules out “non-monotonic” assemblies
But the formulation rules out "non-monotonic" assemblies
But the formulation rules out “non-monotonic” assemblies

This “subassembly” is not allowed in the definition of the state space: the 2 parts are not in their relative placements in the assembly.

Allowing any grouping of parts as a valid subassembly would make the state space much bigger and more difficult to search.
Assumptions in Basic Search

- The world is static
- The world is discretizable
- The world is observable
- The actions are deterministic

But many of these assumptions can be removed, and search still remains an important problem-solving tool.
Vacuum Cleaner Problem

- A vacuum robot lives in a two-room environment

- **States**: The robot is in one of the two rooms, and each room may or may not contain dirt → 8 states

- **Successor function**: the successors of a state correspond to trying 3 actions: Right, Left, Suck.

- **Initial state**: Unknown (not observable)

- **Goal state**: No dust in the rooms
Re-Formulation with “Belief States”

- **Belief states**: sets of states $\rightarrow 2^8 = 256$ belief states
- **Initial belief state**: set of 8 states
- **Successor function**: the successors of a belief state correspond to trying Right, Left, Suck.
- **Goal belief state**: any set of states with no dust in the rooms
Part Feeding
Part Feeding
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.
Enter schematics; do not worry about placement & wire crossing.

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.
Polynomial-time hierarchy

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

$AC^0$: can be solved using gates of constant depth
$NC^1$: 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
Search and AI

- Search methods are **ubiquitous** in AI systems. They often are the backbones of both core and peripheral modules.

- An **autonomous robot** uses search methods:
  - to decide which actions to take and which sensing operations to perform,
  - to quickly anticipate and prevent collision,
  - to plan trajectories,
  - to interpret large numerical datasets provided by sensors into compact symbolic representations,
  - to diagnose why something did not happen as expected,
  - etc...
Applications

Search plays a key role in many applications, e.g.:

- Route finding: airline travel, networks
- Package/mail distribution
- Pipe routing, VLSI routing
- Comparison and classification of protein folds
- Pharmaceutical drug design
- Design of protein-like molecules
- Inverse analysis for non-destructive testing
- Video games
Simple Problem-Solving-Agent Algorithm

1. $s_0 \leftarrow$ sense/read state
2. GOAL? $\leftarrow$ select/read goal test
3. SUCCESSORS $\leftarrow$ read successor function
4. solution $\leftarrow$ search($s_0$, G, Succ)
5. perform(solution)
Searching the State Space

Search tree

Note that some states are visited multiple times
Basic Search Concepts

- Search tree
- Search node
- Node expansion
- Fringe of search tree
- **Search strategy**: At each stage it determines which node to expand
Search Nodes ≠ States
If states are allowed to be revisited, the search tree may be infinite even when the state space is finite.
**Data Structure of a Node**

Depth of a node $N = \text{length of path from root to } N$

(Height of the root = 0)
Node expansion

The expansion of a node N of the search tree consists of:

1) Evaluating the successor function on \text{STATE}(N)
2) Generating a child of N for each state returned by the function
Fringe and Search Strategy

- The **fringe** is the set of all search nodes that haven’t been expanded yet

Is it identical to the set of leaves?
Fringe and Search Strategy

- The **fringe** is the set of all search nodes that haven’t been expanded yet.
- It is implemented as a *priority queue* FRINGE.
  - INSERT(node,FRINGE)
  - REMOVE(FRINGE)
- The ordering of the nodes in FRINGE defines the *search strategy*. 

106
Search Algorithm

1. If \texttt{GOAL?(initial-state)} then return initial-state
2. \texttt{INSERT(initial-node,FRINGE)}
3. Repeat:
   a. If empty(FRINGE) then return \texttt{failure}
   b. \texttt{n} \leftarrow \texttt{REMOVE(FRINGE)}
   c. \texttt{s} \leftarrow \texttt{STATE(n)}
   d. For every state \texttt{s'} in \texttt{SUCCESSORS(s)}
      i. Create a new node \texttt{n'} as a child of \texttt{n}
      ii. If \texttt{GOAL?(s')} then return \texttt{path or goal state}
      iii. \texttt{INSERT(n',FRINGE)}
Performance Measures

- **Completeness**
  A search algorithm is complete if it finds a solution whenever one exists
  [What about the case when no solution exists?]

- **Optimality**
  A search algorithm is optimal if it returns a minimum-cost path whenever a solution exists
  [Other optimality measures are possible]

- **Complexity**
  It measures the time and amount of memory required by the algorithm
Important Parameters

1) Maximum number of successors of any state
   → branching factor $b$ of the search tree

2) Minimal length of a path between the initial and a goal state
   → depth $d$ of the shallowest goal node in the search tree
Blind vs. Heuristic Strategies

- **Blind (or un-informed) strategies** do not exploit state descriptions to select which node to expand next.

- **Heuristic (or informed) strategies** exploit state descriptions to select the “most promising” node to expand.
For a blind strategy, $N_1$ and $N_2$ are just two nodes (at some depth in the search tree)
For a heuristic strategy counting the number of misplaced tiles, $N_2$ is more promising than $N_1$. 

Example
Some search problems, such as the \((n^2-1)\)-puzzle, are NP-hard

One can’t expect to solve all instances of such problems in less than exponential time

One may still strive to solve each instance as efficiently as possible
Blind Strategies

- **Breadth-first**
  - Bidirectional

- **Depth-first**
  - Depth-limited
  - Iterative deepening

- **Uniform-Cost**
  (variant of breadth-first)

\[
\text{Arc cost} = 1
\]

\[
\text{Arc cost} = c(\text{action}) \geq \varepsilon > 0
\]
Breadth-First Strategy

New nodes are inserted at the end of FRINGE

FRINGE = (1)
Breadth-First Strategy

New nodes are inserted at the end of FRINGE

FRINGE = (2, 3)
Breadth-First Strategy

New nodes are inserted at the end of FRINGE

FRINGE = (3, 4, 5)
Breadth-First Strategy

New nodes are inserted at the end of FRINGE

FRINGE = (4, 5, 6, 7)
Breadth-first search

Move downwards, level by level, until goal is reached.
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 $b^d$ nodes. (Thus: 4).

- In General: $b^d$
Evaluation

- $b$: branching factor
- $d$: depth of shallowest goal node
- Breadth-first search is:
  - Complete
  - Optimal if step cost is 1
- Number of nodes generated:
  $1 + b + b^2 + \ldots + b^d = \frac{(b^{d+1}-1)}{(b-1)} = O(b^d)$
- $\rightarrow$ Time and space complexity is $O(b^d)$
Big O Notation

g(n) = O(f(n)) if there exist two positive constants a and N such that:

for all n > N: \( g(n) \leq a \times f(n) \)
## Time and Memory Requirements

<table>
<thead>
<tr>
<th>d</th>
<th># Nodes</th>
<th>Time</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>111</td>
<td>.01 msec</td>
<td>11 Kbytes</td>
</tr>
<tr>
<td>4</td>
<td>11,111</td>
<td>1 msec</td>
<td>1 Mbyte</td>
</tr>
<tr>
<td>6</td>
<td>$\sim10^6$</td>
<td>1 sec</td>
<td>100 Mb</td>
</tr>
<tr>
<td>8</td>
<td>$\sim10^8$</td>
<td>100 sec</td>
<td>10 Gbytes</td>
</tr>
<tr>
<td>10</td>
<td>$\sim10^{10}$</td>
<td>2.8 hours</td>
<td>1 Tbyte</td>
</tr>
<tr>
<td>12</td>
<td>$\sim10^{12}$</td>
<td>11.6 days</td>
<td>100 Tbytes</td>
</tr>
<tr>
<td>14</td>
<td>$\sim10^{14}$</td>
<td>3.2 years</td>
<td>10,000 Tbytes</td>
</tr>
</tbody>
</table>

Assumptions: $b = 10$; 1,000,000 nodes/sec; 100 bytes/node
### Time and Memory Requirements

<table>
<thead>
<tr>
<th>d</th>
<th># Nodes</th>
<th>Time</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>111</td>
<td>.01 msec</td>
<td>11 Kbytes</td>
</tr>
<tr>
<td>4</td>
<td>11,111</td>
<td>1 msec</td>
<td>1 Mbyte</td>
</tr>
<tr>
<td>6</td>
<td>(\sim 10^6)</td>
<td>1 sec</td>
<td>100 Mb</td>
</tr>
<tr>
<td>8</td>
<td>(\sim 10^8)</td>
<td>100 sec</td>
<td>10 Gbytes</td>
</tr>
<tr>
<td>10</td>
<td>(\sim 10^{10})</td>
<td>2.8 hours</td>
<td>1 Tbyte</td>
</tr>
<tr>
<td>12</td>
<td>(\sim 10^{12})</td>
<td>11.6 days</td>
<td>100 Tbytes</td>
</tr>
<tr>
<td>14</td>
<td>(\sim 10^{14})</td>
<td>3.2 years</td>
<td>10,000 Tbytes</td>
</tr>
</tbody>
</table>

Assumptions: \(b = 10\); 1,000,000 nodes/sec; 100 bytes/node
Remark

If a problem has no solution, breadth-first may run for ever (if the state space is infinite or states can be revisited arbitrary many times)
Bidirectional Strategy

2 fringe queues: FRINGE1 and FRINGE2

Time and space complexity is $O(b^{d/2}) << O(b^d)$ if both trees have the same branching factor $b$

Question: What happens if the branching factor is different in each direction?
Depth-First Strategy

New nodes are inserted at the front of FRINGE

FRINGE = (1)
Depth-First Strategy

New nodes are inserted at the front of FRINGE
Depth-First Strategy

New nodes are inserted at the front of FRINGE

FRINGE = (4, 5, 3)
Depth-First Strategy

New nodes are inserted at the front of FRINGE
Depth-First Strategy

New nodes are inserted at the front of FRINGE
Depth-First Strategy

New nodes are inserted at the front of FRINGE
Depth-First Strategy

New nodes are inserted at the front of FRINGE
Depth-First Strategy

New nodes are inserted \textit{at the front} of FRINGE
Depth-First Strategy

New nodes are inserted at the front of FRINGE
Depth-First Strategy

New nodes are inserted at the front of FRINGE
Depth-First Strategy

New nodes are inserted at the front of FRINGE
Depth First Search
Time complexity of depth-first: details

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

- 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: $m = 3$, $b = 3$:
  - QUEUE contains all nodes. Thus: 7.
  - In General: $((b-1) \times m) + 1$
  - Order: $O(m \times b)$
Evaluation

- $b$: branching factor
- $d$: depth of shallowest goal node
- $m$: maximal depth of a leaf node
- Depth-first search is:
  - Complete only for finite search tree
  - Not optimal
- Number of nodes generated:
  $1 + b + b^2 + \ldots + b^m = O(b^m)$
- Time complexity is $O(b^m)$
- Space complexity is $O(bm)$ [or $O(m)$]

[Reminder: Breadth-first requires $O(b^d)$ time and space]
Depth-Limited Search

- Depth-first with depth cutoff \( k \) (depth below which nodes are not expanded)

- Three possible outcomes:
  - Solution
  - Failure (no solution)
  - Cutoff (no solution within cutoff)
Iterative Deepening Search

- Provides the best of both breadth-first and depth-first search

- Main idea: Totally horrifying!

- IDS
- For \( k = 0, 1, 2, \ldots \) do:
- Perform depth-first search with depth cutoff \( k \)
Iterative Deepening
Iterative Deepening
Iterative Deepening
Performance

- Iterative deepening search is:
  - Complete
  - Optimal if step cost = 1
- Time complexity is:
  \[(d+1)(1) + db + (d-1)b^2 + \ldots + (1) b^d = O(b^d)\]
- Space complexity is: \(O(bd)\) or \(O(d)\)
Calculation

db + (d-1)b^2 + ... + (1) b^d
= b^d + 2b^{d-1} + 3b^{d-2} + ... + db
= (1 + 2b^{-1} + 3b^{-2} + ... + db^{-d}) \times b^d
\leq \left( \sum_{i=1,\ldots,\infty} ib^{(1-i)} \right) \times b^d = b^d \left( b/(b-1) \right)^2
Number of Generated Nodes (Breadth-First & Iterative Deepening)

\( d = 5 \) and \( b = 2 \)

<table>
<thead>
<tr>
<th>BF</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 \times 6 = 6</td>
</tr>
<tr>
<td>2</td>
<td>2 \times 5 = 10</td>
</tr>
<tr>
<td>4</td>
<td>4 \times 4 = 16</td>
</tr>
<tr>
<td>8</td>
<td>8 \times 3 = 24</td>
</tr>
<tr>
<td>16</td>
<td>16 \times 2 = 32</td>
</tr>
<tr>
<td>32</td>
<td>32 \times 1 = 32</td>
</tr>
<tr>
<td>63</td>
<td>120</td>
</tr>
</tbody>
</table>
**Number of Generated Nodes**  
(Breadth-First & Iterative Deepening)

\[ d = 5 \text{ and } b = 10 \]

<table>
<thead>
<tr>
<th>BF</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>10</td>
<td>50</td>
</tr>
<tr>
<td>100</td>
<td>400</td>
</tr>
<tr>
<td>1,000</td>
<td>3,000</td>
</tr>
<tr>
<td>10,000</td>
<td>20,000</td>
</tr>
<tr>
<td>100,000</td>
<td>100,000</td>
</tr>
<tr>
<td>111,111</td>
<td>123,456</td>
</tr>
</tbody>
</table>

\[ \frac{123,456}{111,111} \approx 1.111 \]
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 $n = \text{all nodes that have } n \text{ as successor}$
  - 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: \(2 \cdot O(b^{d/2}) = O(b^{d/2})\)
- 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

Final State

\[ \frac{d}{2} \]

\[ d \]
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 Bidirectional</th>
<th>Breadth-first</th>
<th>Uniform Depth-first</th>
<th>Depth-limited deepening (if applicable)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Time</strong></td>
<td>$b^d$</td>
<td>$b^d$</td>
<td>$b^l$</td>
</tr>
<tr>
<td><strong>Space</strong></td>
<td>$b^d$</td>
<td>$b^d$</td>
<td>$b^d$</td>
</tr>
<tr>
<td><strong>Optimal?</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><strong>Complete?</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>No, if $l \geq d$</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
Comparison of Strategies

- Breadth-first is complete and optimal, but has high space complexity.
- Depth-first is space efficient, but is neither complete, nor optimal.
- Iterative deepening is complete and optimal, with the same space complexity as depth-first and almost the same time complexity as breadth-first.
Revisited States

No

Few

Many

search tree is finite

search tree is infinite

8-queens

assembly planning

8-puzzle and robot navigation
Avoiding Revisited States

- Requires comparing state descriptions
- Breadth-first search:
  - Store all states associated with *generated* nodes in VISITED
  - If the state of a new node is in VISITED, then discard the node
Avoiding Revisited States

- Requires comparing state descriptions
- Breadth-first search:
  - Store all states associated with *generated* nodes in \texttt{VISITED}
  - If the state of a new node is in \texttt{VISITED}, then discard the node

Implemented as hash-table or as explicit data structure with flags
Avoiding Revisited States

- Depth-first search:
  Solution 1:
  - Store all states associated with nodes in current path in VISITED
  - If the state of a new node is in VISITED, then discard the node
  → Only avoids loops

Solution 2:
  - Store all generated states in VISITED
  - If the state of a new node is in VISITED, then discard the node
  → Same space complexity as breadth-first!
Uniform-Cost Search

- Each arc has some cost \( c \geq \varepsilon > 0 \)
- The cost of the path to each fringe node \( N \) is \( g(N) = \sum \) costs of arcs
- The goal is to generate a solution path of minimal cost
- The queue FRINGE is sorted in increasing cost

- Need to modify search algorithm
Modified Search Algorithm

1. INSERT(initial-node,FRINGE)

2. Repeat:
   a. If empty(FRINGE) then return failure
   b. n ← REMOVE(FRINGE)
   c. s ← STATE(n)
   ➢ d. If GOAL?(s) then return path or goal state
   e. For every state s' in SUCCESSORS(s)
      i. Create a node n' as a successor of n
      ii. INSERT(n',FRINGE)
Avoiding Revisited States in Uniform-Cost Search

- When a node $N$ is expanded the path to $N$ is also the best path from the initial state to $\text{STATE}(N)$

- So:
  - When a node is **expanded**, store its state into CLOSED
  - When a new node $N$ is generated:
    - If $\text{STATE}(N)$ is in CLOSED, discard $N$
    - If there exits a node $N'$ in the fringe such that $\text{STATE}(N') = \text{STATE}(N)$, discard the node - $N$ or $N'$ - with the highest-cost path
Comments:
- Each edge represents two opposite arcs
- The cost of an arc is its length
- The animation turns an arc to green when the end node is expanded