Set 2: State-spaces and Uninformed Search

ICS 271 Fall 2014
Kalev Kask
Problem-Solving Agents

• Intelligent agents can solve problems by searching a state-space

• State-space Model
  – the agent’s model of the world
  – usually a set of discrete states
  – e.g., in driving, the states in the model could be towns/cities

• Goal State(s)
  – a goal is defined as a desirable state for an agent
  – there may be many states which satisfy the goal
    • e.g., drive to a town with a ski-resort
  – or just one state which satisfies the goal
    • e.g., drive to Mammoth

• Operators
  – operators are legal actions which the agent can take to move from one state to another
Example: Romania
Example: Romania

• On holiday in Romania; currently in Arad.
• Flight leaves tomorrow from Bucharest
• Formulate goal:
  – be in Bucharest
• Formulate problem:
  – states: various cities
  – actions: drive between cities
• Find solution:
  – sequence of actions (cities), e.g., Arad, Sibiu, Fagaras, Bucharest
Problem Types

- **Static / Dynamic**
  Previous problem was static: no attention to changes in environment

- **Observable / Partially Observable / Unobservable**
  Previous problem was observable: it knew its initial state.

- **Deterministic / Stochastic**
  Previous problem was deterministic: no new percepts were necessary, we can predict the future perfectly

- **Discrete / continuous**
  Previous problem was discrete: we can enumerate all possibilities
A **problem** is defined by five items:

- **initial state** e.g., "at Arad"
- **actions** or successor function $S(x) = \text{set of action-state pairs}$
  - e.g., $S(\text{Arad}) = \{<\text{Arad} \rightarrow \text{Zerind}, \text{Zerind}>, \ldots \}$
- **transition function** – maps action $\times$ state $\rightarrow$ state
- **goal test**, (or goal state)
  e.g., $x = \text{"at Bucharest"}, \text{Checkmate}(x)$
- **path cost** (additive)
  - e.g., sum of distances, number of actions executed, etc.
  - $c(x,a,y)$ is the **step cost**, assumed to be $\geq 0$

A **solution** is a sequence of actions leading from the initial state to a goal state.
State-Space Problem Formulation

• A statement of a Search problem has 5 components
  – 1. A start state S
  – 2. A set of operators/actions which allow one to get from one state to another
  – 3. transition function
  – 4. A set of possible goal states G, or ways to test for goal states
  – 5. Cost path

• A solution consists of
  – a sequence of operators which transform S into a goal state G

• Representing real problems in a State-Space search framework
  – may be many ways to represent states and operators
  – key idea: represent only the relevant aspects of the problem (abstraction)
Abstraction/Modeling

Process of removing irrelevant detail to create an abstract representation: "high-level", ignores irrelevant details

• Definition of Abstraction:

• Navigation Example: how do we define states and operators?
  – First step is to abstract “the big picture”
    • i.e., solve a map problem
    • nodes = cities, links = freeways/roads (a high-level description)
    • this description is an abstraction of the real problem
  – Can later worry about details like freeway onramps, refueling, etc

• Abstraction is critical for automated problem solving
  – must create an approximate, simplified, model of the world for the computer to deal with: real-world is too detailed to model exactly
  – good abstractions retain all important details
Robot block world

• Given a set of blocks in a certain configuration,
• Move the blocks into a goal configuration.
• Example:
  – (c,b,a) → (b,c,a)
Operator Description

Effects of Moving a Block
The State-Space Graph

• **Problem formulation:**
  – Give an abstract description of states, operators, initial state and goal state.

• **Graphs:**
  – vertices, edges (arcs), directed arcs, paths

• **State-space graphs:**
  – States are vertices
  – operators are directed arcs
  – solution is a path from start to goal

• **Problem solving activity:**
  – Generate a part of the search space that contains a solution

**State-space:**
1. A set of states
2. A set of “operators”/transitions
3. A start state S
4. A set of possible goal states
5. Cost path
The Traveling Salesperson Problem

- Find the shortest tour that visits all cities without visiting any city twice and return to starting point.
- State:
  - sequence of cities visited
- $S_0 = A$
The Traveling Salesperson Problem

- Find the shortest tour that visits all cities without visiting any city twice and return to starting point.
- State: sequence of cities visited
- $S_0 = A$

- Solution = a complete tour

Transition model

$\{a, c, d\} \iff \{(a, c, d, x) \mid X \notin a, c, d\}$
Example: 8-queen problem
Example: 8-Queens

- **states?** any arrangement of $n \leq 8$ queens
  - or arrangements of $n \leq 8$ queens in leftmost $n$ columns, 1 per column, such that no queen attacks any other.

- **initial state?** no queens on the board

- **actions?** add queen to any empty column
  - or add queen to leftmost empty column such that it is not attacked by other queens.

- **goal test?** 8 queens on the board, none attacked.

- **path cost?** 1 per move
The Sliding Tile Problem

**Figure 8.1**
Start and Goal Configurations for the Eight-Puzzle

\[ \text{move}(x, \text{loc } y, \text{loc } z) \]

- Up
- Down
- Left
- Right
The “8-Puzzle” Problem

Start State

1 2 3
4 6
7 5 8

Goal State

1 2 3
4 5 6
7 8
Example: robotic assembly

- **states?**: real-valued coordinates of robot joint angles
- **actions?**: continuous motions of robot joints
- **goal test?**: complete assembly
- **path cost?**: time to execute
Formulating Problems; Another Angle

• **Problem types**
  – Satisfying: 8-queen
  – Optimizing: Traveling salesperson

• **Goal types**
  – board configuration
  – sequence of moves
  – A strategy (contingency plan)

• **Satisfying leads to optimizing since “small is quick”**

• For traveling salesperson
  – satisfying easy, optimizing hard

• **Semi-optimizing:**
  – Find a good solution

• **In Russel and Norvig:**
  – single-state, multiple states, contingency plans, exploration problems
Searching the State Space

• States, operators, control strategies

• The search space graph is implicit

• The control strategy generates a small search tree.

• Systematic search
  – Do not leave any stone unturned

• Efficiency
  – Do not turn any stone more than once
Tree search example
Tree search example
function TREE-SEARCH(problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
State-Space Graph of the 8 Puzzle Problem

Figure 3.6 State space of the 8-puzzle generated by “move blank” operations.
Implementation

- **States vs Nodes**
  - A **state** is a (representation of) a physical configuration
  - A **node** is a data structure constituting part of a search tree contains info such as: state, parent node, action, path cost $g(x)$, depth

- The **Expand function** creates new nodes, filling in the various fields and using the **SuccessorFn** of the problem to create the corresponding states.

- **Queue managing frontier**: 
  - FIFO
  - LIFO
  - priority
Tree-Search vs Graph-Search

- **Tree-search**(problem), returns a solution or failure
  - Frontier ← initial state
  - Loop do
    - If frontier is empty return failure
    - Choose a leaf node and remove from frontier
    - If the node is a goal, return the corresponding solution
    - Expand the chosen node, adding its children to the frontier
  - -----------------------------------------------------------------------------------------------

- **Graph-search**(problem), returns a solution or failure
  - Frontier ← initial state, explored ← empty
  - Loop do
    - If frontier is empty return failure
    - Choose a leaf node and remove from frontier
    - If the node is a goal, return the corresponding solution.
    - Add the node to the explored.
    - Expand the chosen node, adding its children to the frontier, only if not in frontier or explored set
Tree-Search vs. Graph-Search

• Example: Assemble 5 objects \{a, b, c, d, e\}
• A state is a bit-vector (length 5), 1=object in assembly
• 11010 = a, b, d in assembly, c, e not
• State space
  – number of states \(2^5 = 32\)
  – number of edges \((2^5) \cdot 5 \cdot \frac{1}{2} = 80\)
• Tree-search space
  – number of nodes 5! = 120
• State can be reached in multiple ways
  – 11010 can be reached a+b+d or a+d+b etc.
• Graph-search:
  – three kinds of nodes: unexplored, frontier, explored
  – before adding a node, check if a state is in frontier or explored set
Graph-Search

271-fall 2014
Why Search Can be Difficult

• At the start of the search, the search algorithm does not know
  – the size of the tree
  – the shape of the tree
  – the depth of the goal states

• How big can a search tree be?
  – say there is a constant branching factor $b$
  – and one goal exists at depth $d$
  – search tree which includes a goal can have
    $b^d$ different branches in the tree (worst case)

• Examples:
  – $b = 2$, $d = 10$: $b^d = 2^{10} = 1024$
  – $b = 10$, $d = 10$: $b^d = 10^{10} = 10,000,000,000$
Searching the Search Space

• Uninformed (Blind) search
  – Breadth-first
  – Uniform-Cost first
  – Depth-first
  – Iterative deepening depth-first
  – Bidirectional
  – Depth-First Branch and Bound

• Informed Heuristic search
  – Greedy search, hill climbing, Heuristics

• Important concepts:
  – Completeness
  – Time complexity
  – Space complexity
  – Quality of solution
Breadth-First Search

• Expand shallowest unexpanded node
• Frontier: nodes waiting in a queue to be explored, also called OPEN

• Implementation:
  – *frontier* is a first-in-first-out (FIFO) queue, i.e., new successors go at end of the queue.

Is A a goal state?
Breadth-First Search

• Expand shallowest unexpanded node

• **Implementation:**
  
  – *frontier* is a FIFO queue, i.e., new successors go at end

Expand:
frontier = [B,C]

Is B a goal state?
Breadth-First Search

• Expand shallowest unexpanded node
• Implementation:
  – *frontier* is a FIFO queue, i.e., new successors go at end

Expand:
frontier=[C,D,E]

Is C a goal state?
Breadth-First Search

• Expand shallowest unexpanded node

• **Implementation:**
  
  – *frontier* is a FIFO queue, i.e., new successors go at end

Expand:
frontier=[D,E,F,G]

Is D a goal state?
Actually, in BFS we can check if a node is a goal node when it is generated (rather than expanded)
Breadth-First-Search (*)

OPEN = frontier, CLOSED = explored

1. Put the start node $s$ on OPEN
2. If OPEN is empty exit with failure.
3. Remove the first node $n$ from OPEN and place it on CLOSED.
4. Expand $n$, generating all its successors.
   - If child is not in CLOSED or OPEN, then
   - If child is not a goal, then put them at the end of OPEN in some order.
5. If $n$ is a goal node, exit successfully with the solution obtained by tracing back pointers from $n$ to $s$.
6. Go to step 2.

* This is graph-search
Example: Map Navigation

S = start,  G = goal,  other nodes = intermediate states, links = legal transitions
Initial BFS Search Tree

Note: this is the search tree at some particular point in the search.
Complexity of Breadth-First Search

• **Time Complexity**
  – assume (worst case) that there is 1 goal leaf at the RHS
  – so BFS will expand all nodes

\[
= 1 + b + b^2 + \ldots + b^d
\]

\[= O (b^d)\]

• **Space Complexity**
  – how many nodes can be in the queue (worst-case)?
  – at depth \(d\) there are \(b^d\) unexpanded nodes in the \(Q = O (b^d)\)
Examples of Time and Memory Requirements for Breadth-First Search

<table>
<thead>
<tr>
<th>Depth of Solution</th>
<th>Nodes Expanded</th>
<th>Time</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1 millisecond</td>
<td>100 bytes</td>
</tr>
<tr>
<td>2</td>
<td>111</td>
<td>0.1 seconds</td>
<td>11 kbytes</td>
</tr>
<tr>
<td>4</td>
<td>11,111</td>
<td>11 seconds</td>
<td>1 megabyte</td>
</tr>
<tr>
<td>8</td>
<td>$10^8$</td>
<td>31 hours</td>
<td>11 giabytes</td>
</tr>
<tr>
<td>12</td>
<td>$10^{12}$</td>
<td>35 years</td>
<td>111 terabytes</td>
</tr>
</tbody>
</table>

Assuming $b=10$, 1000 nodes/sec, 100 bytes/node
Breadth-First Search (BFS) Properties

- Solution Length: optimal
- Expand each node once (can check for duplicates, performs graph-search)
- Search Time: $O(b^d)$
- Memory Required: $O(b^d)$
- Drawback: requires exponential space
Uniform Cost Search

- Expand lowest-cost OPEN node \( g(n) \)
- In BFS \( g(n) = \text{depth}(n) \)

**Requirement**

- \( g(\text{successor})(n)) \geq g(n) \)
Uniform cost search

1. Put the start node $s$ on OPEN
2. If OPEN is empty exit with failure.
3. Remove the first node $n$ from OPEN and place it on CLOSED.
4. If $n$ is a goal node, exit successfully with the solution obtained by tracing back pointers from $n$ to $s$.
5. Otherwise, expand $n$, generating all its successors attach to them pointers back to $n$, and put them in OPEN in order of shortest cost.
6. Go to step 2.

DFS Branch and Bound

At step 4: compute the cost of the solution found and update the upper bound $U$.

At step 5: expand $n$, generating all its successors attach to them pointers back to $n$, and put on top of OPEN.

Compute cost of partial path to node and prune if larger than $U$. 

271-fall 2014
Depth-First Search

• Expand *deepest* unexpanded node
• **Implementation:**
  – *frontier* = Last In First Out (LIFO) queue, i.e., put successors at front

Is A a goal state?
Depth-first search

• Expand deepest unexpanded node
• Implementation:
   – \textit{frontier} = LIFO queue, i.e., put successors at front

queue=[B,C]

Is B a goal state?
Depth-first search

• Expand deepest unexpanded node
• **Implementation:**
  – *frontier* = LIFO queue, i.e., put successors at front

queue=[D,E,C]

Is D = goal state?
Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *frontier* = LIFO queue, i.e., put successors at front

queue=[H,I,E,C]

Is H = goal state?
Depth-first search

- Expand deepest unexpanded node
- **Implementation:**
  - *frontier* = LIFO queue, i.e., put successors at front

queue=[I,E,C]

Is I = goal state?
Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - \textit{frontier} = LIFO queue, i.e., put successors at front

\text{queue}=[E,C]

Is \text{E} = goal state?
Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - \textit{frontier} = LIFO queue, i.e., put successors at front

\[ \text{queue=}[J,K,C] \]

Is J = goal state?
Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - \textit{frontier} = LIFO queue, i.e., put successors at front

queue=[K,C]

Is K = goal state?
Depth-first search

- Expand deepest unexpanded node
- Implementation: 
  - *frontier* = LIFO queue, i.e., put successors at front

```
queue=[C]
```

Is C = goal state?
Depth-first search

• Expand deepest unexpanded node

• Implementation:
  – *frontier* = LIFO queue, i.e., put successors at front

```
queue=[F,G]
```

Is F = goal state?
Depth-first search

• Expand deepest unexpanded node
• Implementation:
  – *frontier* = LIFO queue, i.e., put successors at front

queue=[L,M,G]

Is L = goal state?
Depth-first search

• Expand deepest unexpanded node
• Implementation:
  – *frontier* = LIFO queue, i.e., put successors at front

queue=[M,G]

Is M = goal state?
Here, (if tree-search) then to avoid infinite depth (in case of finite state-space graph) assume we don’t expand any child node which appears already in the path from the root S to the parent. (Again, one could use other strategies)
Depth-First Search

Generation of the First Few Nodes in a Depth-First Search
The Graph When the Goal Is Reached in Depth-First Search
Depth-First-Search (*)

1. Put the start node s on OPEN

2. If OPEN is empty exit with failure.

3. Remove the first node n from OPEN.

4. If n is a goal node, exit successfully with the solution obtained by tracing back pointers from n to s.

5. Otherwise, expand n, generating all its successors (check for self-loops) attach to them pointers back to n, and put them at the top of OPEN in some order.

6. Go to step 2.

*search the tree search-space (but avoid self-loops)
** the default assumption is that DFS searches the underlying search-tree
Complexity of Depth-First Search?

• Time Complexity
  – assume d is deepest path in the search space
  – assume (worst case) that there is 1 goal leaf at the RHS
  – so DFS will expand all nodes

\[= 1 + b + b^2 + \ldots + b^d\]

\[= O(b^d)\]

• Space Complexity (for tree-search)
  – how many nodes can be in the queue (worst-case)?
  – \(O(bd)\) if deepest node at depth \(d\)
Example, Diamond Networks

graph-search vs tree-search (BFS vs DFS)

• Graph-search & BFS
• Tree-search & DFS
Depth-First tree-search Properties

• Non-optimal solution path
• Incomplete unless there is a depth bound
• (we will assume depth-limited DF-search)
• Re-expansion of nodes (when the state-space is a graph)
• Exponential time
• Linear space (for tree-search)
Comparing DFS and BFS

• BFS optimal, DFS is not
• Time Complexity worse-case is the same, but
  – In the worst-case BFS is always better than DFS
  – Sometime, on the average DFS is better if:
    • many goals, no loops and no infinite paths
• BFS is much worse memory-wise
  • DFS can be linear space
  • BFS may store the whole search space.
• In general
  • BFS is better if goal is not deep, if long paths, if many loops, if small search space
  • DFS is better if many goals, not many loops
  • DFS is much better in terms of memory
Iterative-Deepening Search (DFS)

- Every iteration is a DFS with a depth cutoff.

Iterative deepening (ID)

1. \(i = 1\)
2. While no solution, do
3. DFS from initial state \(S_0\) with cutoff \(i\)
4. If found goal, stop and return solution, else, increment cutoff

Comments:
- IDS implements BFS with DFS
- Only one path in memory
- BFS at step \(i\) may need to keep \(2^i\) nodes in OPEN
Iterative deepening search $L=0$
Iterative deepening search $L=1$
Iterative deepening search $L=2$
Iterative Deepening Search $L=3$
Iterative deepening search

Stages in Iterative-Deepening Search

Depth bound = 1
Depth bound = 2
Depth bound = 3
Depth bound = 4
Iterative Deepening (DFS)

- Time:
  \[
  T(n) = \sum_{j=1}^{n} \frac{b^{j+1} - 1}{b-1} = \frac{b^{n+2}}{(b-1)^2} = O(b^n)
  \]

- BFS time is \(O(b^n)\), \(b\) is the branching degree
- IDS is asymptotically like BFS,
- For \(b=10\quad d=5\quad d=\text{cut-off}\)
- DFS = 1+10+100,….=111,111
- IDS = 123,456
- Ratio is \(\frac{b}{b-1}\)
Summary on IDS

- A useful practical method
  - combines
    - guarantee of finding an optimal solution if one exists (as in BFS)
    - space efficiency, \(O(bd)\) of DFS
    - But still has problems with loops like DFS
Bidirectional Search

• **Idea**
  – simultaneously search forward from S and backwards from G
  – stop when both “meet in the middle”
  – need to keep track of the intersection of 2 open sets of nodes

• **What does searching backwards from G mean**
  – need a way to specify the predecessors of G
    • this can be difficult,
    • e.g., predecessors of checkmate in chess?
  – what if there are multiple goal states?
  – what if there is only a goal test, no explicit list?

• **Complexity**
  – time complexity is best: $O(2 \, b^{(d/2)}) = O(b^{(d/2)})$
  – memory complexity is the same
Bi-Directional Search

Fig. 2.10 Bidirectional and unidirectional breadth-first searches.
Comparison of Algorithms

<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>Time</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>Space</td>
<td>$b^d$</td>
<td>$b^d$</td>
<td>$b^m$</td>
<td>$b^l$</td>
<td>$bd$</td>
<td>$b^{d/2}$</td>
</tr>
<tr>
<td>Optimal?</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Complete?</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>

Figure 3.18 Evaluation of search strategies. $b$ is the branching factor; $d$ is the depth of solution; $m$ is the maximum depth of the search tree; $l$ is the depth limit.
Summary

• A review of search
  – a search space consists of nodes and operators: it is a tree/graph

• There are various strategies for “uninformed search”
  – breadth-first
  – depth-first
  – iterative deepening
  – bidirectional search
  – Uniform cost search
  – Depth-first branch and bound

• Repeated states can lead to infinitely large search trees
  – we looked at methods for detecting repeated states

• All of the search techniques so far are “blind” in that they do not look at how far away the goal may be: next we will look at informed or heuristic search, which directly tries to minimize the distance to the goal.