Reminders

- Homework 2 has been released (due on Monday)
- Heeba has help hours Thursdays from 10am-12
- Lepei has help hours Tuesdays from 4-6pm
- Reading for Monday: Mitchell Chapter 8-9
Search Algorithms
Search Tree

Root node = start state

Expanded nodes

Frontier

Choose leaf node from frontier for expansion according to the search strategy

Determines the search process

Slides adapted from Chris Callison-Burch
States Versus Nodes

**State** is a representation of physical configuration.

**Node** is a data structure:
- parent
- children
- depth

transition(state, action)
returns state, succ

<state, parent, [children], action, path-cost, depth>

Expand: uses transition model to create new nodes
8-Puzzle **Search Tree**

(Nodes show state, parent, children - leaving *Action, Cost, Depth* Implicit)
Problem: Repeated states

Failure to detect *repeated states* can turn a linear problem into an *exponential* one!
Solution: Graph Search!

Graph search

• Simple Mod from tree search: *Check to see if a node has been visited before adding to search queue*
  • must keep track of all possible states (can use a lot of memory)
  • e.g., 8-puzzle problem, we have $9!/2 \approx 182K$ states
Graph Search vs Tree Search

function TREE-SEARCH(problem) returns a solution, or failure
initialize the frontier using the initial state of problem
loop do
  if the frontier is empty then return failure
  choose a leaf node and remove it from the frontier
  if the node contains a goal state then return the corresponding solution
  expand the chosen node, adding the resulting nodes to the frontier

function GRAPH-SEARCH(problem) returns a solution, or failure
initialize the frontier using the initial state of problem
initialize the explored set to be empty
loop do
  if the frontier is empty then return failure
  choose a leaf node and remove it from the frontier
  if the node contains a goal state then return the corresponding solution
  add node to the explored set
  expand the chosen node, adding the resulting nodes to the frontier only if not in the frontier of explored set
Uninformed Search
Uninformed Search

Uses only information available in problem definition

Informally:

*Uninformed search:* All non-goal nodes in frontier look equally good

*Informed search:* Some non-goal nodes can be ranked above others.
Breadth-First Search
8-Puzzle Search Tree

(Nodes show state, parent, children - leaving Action, Cost, Depth Implicit)
Breadth-first search

Idea:
  • Expand *shallowest* unexpanded node

Implementation:
  • *frontier* is FIFO (First-In-First-Out) Queue:
    • Put successors at the *end* of *frontier* successor list.
Breadth-first search

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
Breadth-first search

```python
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)
```

Node data structure contains variables like the state, a pointer to its parent node, the action that was used to create this state, and the path cost.

The Python yield keyword means that we don’t have to pre-compute a list of all successors.

Slides adapted from Chris Callison-Burch
Breadth-first search

```python
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
```

Subtle: Node inserted into queue only after testing to see if it is a goal state

Slides adapted from Chris Callison-Burch
bfs(x):
make a new queue called q
mark x visited
push x onto q
while q not empty:
  pop q into x
  for each y in x connections
    if y not visited:
      mark y visited
      push y onto q
Properties of breadth-first search

Complete? \( \sqrt{\text{Yes}} \)

Optimal? only if \( \text{cost} = 1 \) for every step

Time Complexity? \( 1 + b + b^2 + \ldots = O(b^d) \)

Space Complexity? \( O(b^d) \)

- \( b \): maximum branching factor
- \( d \): depth of the least cost solution

Goal at 1st level: \( b^1 \)
Exponential Space (and time) Is Not Good...

- Exponential complexity uninformed search problems *cannot* be solved for any but the smallest instances.
- *(Memory requirements are a bigger problem than execution time.)*

<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>0.11 milliseconds</td>
<td>107 kilobytes</td>
</tr>
<tr>
<td>4</td>
<td>11110</td>
<td>11 milliseconds</td>
<td>10.6 megabytes</td>
</tr>
<tr>
<td>6</td>
<td>$10^6$</td>
<td>1.1 seconds</td>
<td>1 gigabyte</td>
</tr>
<tr>
<td>8</td>
<td>$10^8$</td>
<td>2 minutes</td>
<td>103 gigabytes</td>
</tr>
<tr>
<td>10</td>
<td>$10^{10}$</td>
<td>3 hours</td>
<td>10 terabytes</td>
</tr>
<tr>
<td>12</td>
<td>$10^{12}$</td>
<td>13 days</td>
<td>1 petabyte</td>
</tr>
<tr>
<td>14</td>
<td>$10^{14}$</td>
<td>3.5 years</td>
<td>99 petabytes</td>
</tr>
</tbody>
</table>

Assumes $b=10$, 1M nodes/sec, 1000 bytes/node
Depth-First Search
Depth-first search

Idea:
  • Expand *deepest* unexpanded node

Implementation:
  • *frontier* is LIFO (Last-In-First-Out) Queue:
    • Put successors at the *front* of *frontier* successor list.
1  \( x = \text{start vertex}(1) \)
2  \( \text{dfs}(x) \)
3  
4  \text{def dfs}(x): 
5  \text{mark } x \text{ as visited} 
6  \text{for each } y \text{ in } x \text{ connections:} 
7  \text{if } y \text{ not visited then} 
8  \text{dfs}(y) 

Please subscribe @youtube.com/gjenkinslbcc or with icon in lower right  >>>>
8-Puzzle **Search Tree**

(Nodes show state, parent, children - leaving *Action, Cost, Depth* Implicit)

---

Slides adapted from Chris Callison-Burch
Properties of depth-first search

Complete? ✓ ▐
Optimal? No, even in cost step = 1 scenario
Time Complexity? $O(b^m)$
Space Complexity? $O(bm)$

$b$: maximum branching factor
$d$: depth of the solution
$m$: maximum depth of tree
Depth-first vs Breadth-first

Use depth-first if

- *Space is restricted*
- There are many possible solutions with long paths and wrong paths are usually terminated quickly
- Search can be fine-tuned quickly

Use breadth-first if

- *Possible infinite paths*
- Some solutions have short paths
- Can quickly discard unlikely paths

Slides adapted from Chris Callison-Burch
Search Conundrum

Breadth-first
- Complete,
- Optimal
- \*\*but uses $O(b^d)$ space\*

Depth-first
- Not complete \*unless m is bounded\*
- Not optimal
- Uses $O(b^m)$ time; terrible if $m \gg d$
- \*but only uses $O(b^*m)$ space\*
Depth-limited search: A building block

Depth-First search *but with depth limit* $l$.
  • i.e. nodes at depth $l$ *have no successors*.
  • No infinite-path problem!

If $l = d$ (by luck!), then optimal
  • But:
    • If $l < d$ then incomplete 😞
    • If $l > d$ then not optimal 😞

Time complexity: $O(b^l)$
Space complexity: $O(bl)$ 😊

Slides adapted from Chris Callison-Burch
Search Strategies

Review: \textit{Strategy} = order of tree expansion

- Implemented by different queue structures (LIFO, FIFO, priority)

Dimensions for evaluation

- \textit{Completeness}- always find the solution?
- \textit{Optimality} - finds a least cost solution (lowest path cost) first?
- \textit{Time complexity} - # of nodes generated (\textit{worst case})
- \textit{Space complexity} - # of nodes simultaneously in memory (\textit{worst case})

Time/space complexity variables

- $b$, \textit{maximum branching factor} of search tree
- $d$, \textit{depth} of the shallowest goal node
- $m$, maximum length of any path in the state space (potentially $\infty$)
Depth-first vs Breadth-first

Use depth-first if

- *Space is restricted*
- There are many possible solutions with long paths and wrong paths are usually terminated quickly
- Search can be fine-tuned quickly

Use breadth-first if

- *Possible infinite paths*
- Some solutions have short paths
- Can quickly discard unlikely paths
Search Conundrum

Breadth-first
- Complete,
- Optimal
- *but* uses $O(b^d)$ space

Depth-first
- Not complete *unless m is bounded*
- Not optimal
- Uses $O(b^m)$ time; terrible if $m >> d$
- *but* only uses $O(b^*m)$ space
Depth-limited search: A building block

Depth-First search *but with depth limit* $l$.
- i.e. nodes at depth $l$ *have no successors.*
- No infinite-path problem!

If $l = d$ (by luck!), then optimal
- But:
  - If $l < d$ then incomplete 😞
  - If $l > d$ then not optimal 😞

Time complexity: $O(b^l)$
Space complexity: $O(bl)$ 😊

Slides adapted from Chris Callison-Burch
Iterative Deepening
Iterative deepening search

A general strategy to find best depth limit $l$.

- **Key idea**: use *Depth-limited search* as subroutine, with increasing $l$.

  For $l = 0$ to $\infty$ do
  
  `depth-limited-search to level l`
  
  if it succeeds
  
  then return solution

- **Complete & optimal**: Goal is always found at depth $d$, the depth of the shallowest goal-node.
ID search, Evaluation

Complete: YES (no infinite paths)

Time complexity: \( O(bd) \)

Space complexity: \( O(bd) \)

Optimal: YES if step cost is 1.
## Summary of algorithms

<table>
<thead>
<tr>
<th></th>
<th>BFS</th>
<th>DFS</th>
<th>Depth-limited</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Complete?</strong></td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Time</strong></td>
<td>$b^d$</td>
<td>$b^m$</td>
<td>$b^l$</td>
<td>$b^d$</td>
</tr>
<tr>
<td><strong>Space</strong></td>
<td>$b^d$</td>
<td>$b^m$</td>
<td>$b^l$</td>
<td>$b^d$</td>
</tr>
<tr>
<td><strong>Optimal?</strong></td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>✓</td>
</tr>
</tbody>
</table>
Search Strategies

Review: **Strategy** = order of tree expansion

- Implemented by different queue structures (LIFO, FIFO, priority)

Dimensions for evaluation

- **Completeness** - always find the solution?
- **Optimality** - finds a least cost solution (lowest path cost) first?
- **Time complexity** - # of nodes generated (**worst case**)
- **Space complexity** - # of nodes simultaneously in memory (**worst case**)

Time/space complexity variables

- $b$, **maximum branching factor** of search tree
- $d$, **depth** of the shallowest goal node
- $m$, maximum length of any path in the state space (potentially $\infty$)