Artificial Intelligence

Uninformed search strategies

Uninformed Search Strategies

• *Uninformed* strategies use only the information available in the problem definition
  – Also called “Blind Search”
  – No info on number of steps or path cost from “current state to goal state”
  – Only distinguish goal state from non-goal state
Uninformed Search Strategies

- Consider route-finding problem (Arad to Bucharest)
  - Uninformed search has no preference of which to choose first between the three states connected to Arad
  - Informed (heuristic) search might notice that only Sibu is in the “direction of” Arad to Bucharest

Uninformed Search Methods

- Types
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Depth-limited search
  - Iterative deepening search
- Strategies differ by the order in which nodes are expanded
Breadth-First Search

- Expand root node, then expand all successors, then their successors, and so on
- Expand shallowest unexpanded node first
  - Left-to-right or right-to-left ordering
- All nodes at level \( d \) are expanded before level \( d+1 \)
  - Root node is level \( d = 0 \)
- **Method finds the shallowest goal state**
- Queue implementation
  - Put expanded nodes at end of “queue” (\( Q \) in the following slides)

BFS Example: Romania

\[ Q = [\text{Arad}] \]
\[ \text{initial } Q \text{ is start state} \]
\[ \text{Arad } \leftarrow Q = [-] \]
\[ \text{"pop" } Q \]
\[ \text{Is it the goal? NO} \]
BFS Example: Romania

$Q = \{\text{Zerind, Sibiu, Timisoara}\}$

*add children of Arad to end of $Q$*

**BFS Example: Romania**

Zerind ← $Q = \{\text{Sibiu, Timisoara}\}$

*Is it the goal? NO*

$Q = \{\text{Sibiu, Timisoara, Arad, Oradea}\}$

*add children of Zerind to end of $Q$*

*More on this issue later!*
BFS Example: Romania

“pop” $\mathcal{Q}$

Sibiu $\leftarrow \mathcal{Q} = \{\text{Timisoara, Arad, Oradea}\}$

Is it the goal? NO

$\mathcal{Q} = \{\text{Timisoara, Arad, Oradea, Arad, Oradea, Fagaras, Riminiciu Vilcea}\}$

add children of Sibiu to end of $\mathcal{Q}$

Keep going until hit a goal state (Bucharest) when “pop” $\mathcal{Q}$…
Properties of Breadth-First Search

- **Complete (good)**
  - If branching fact \( b \) is finite
- **Space** – nodes generated (exponential - bad)
  - \( O(b^{d+1}) = b + b^2 + \ldots + b^d + (b^{d+1} - b) \), \( d = \) goal depth
  - Assume goal is last node (e.g., rightmost) at depth \( d \)
  - Goal state is not expanded, but still have all the children stored of other nodes at depth \( d \!\)!
  - Big limitation (need lots of space)
    - Depth=10, branching=10, space=1000 bytes/node (101 terabytes!)
- **Time (bad)**
  - Same as space
- **Optimality (good)**
  - Not in general, shallowest may not be optimal path cost
  - Optimal if path cost non-decreasing function of node depth

Uniform-Cost Search

- *Modified* breadth-first strategy
- Expand least-cost unexpanded leaf node first (rather than lowest-depth as in BFS)
  - General additive cost function (not cost from current state to goal! – not “informed” search!!)
- Guaranteed to be the cheapest solution
  - Otherwise it would have been expanded earlier
- **Queue implementation**
  - Insert nodes in order of increasing path cost (cheapest first)
Romania Step Costs in km

UCS Example: Romania

$Q = \{\text{Arad}\}$ initial $Q$ is start state

Arad $\leftarrow Q = \{\}$ "pop" $Q$

Is it the goal? NO
UCS Example: Romania

Q = [Zerind, Timisoara, Sibiu]

add children of Arad IN ORDER OF COST to Q

“pop” Q
Zerind ← Q = [Timisoara, Sibiu]
Is it the goal? NO

add children of Zerind IN ORDER OF COST to Q

More on this issue later!
Properties of Uniform Cost Search

- Complete (good)
- Time and space (can be bad)
  - Can be much greater than $b^d$
    - Can explore large subtrees of small steps before exploring large (and perhaps useful) steps
- Optimal (good)
Depth-First Search

- Always expand **deepest** unexpanded node (on the fringe) first
  - Left-to-right or right-to-left ordering
- Only when hit "dead-end" (leaf) does search go back and expand nodes at next shallower level
- **Queue implementation**
  - Insert expanded nodes at **front** of queue

DFS Example: Romania

\[ Q = [\text{Arad}] \]

*initial \( Q \) is start state*

\[ \text{Arad} \leftarrow Q = [-] \]

*"pop" \( Q \)*

Is it the goal? **NO**
DFS Example: Romania

Q = [Zerind, Sibiu, Timisoara]

add children of Arad to FRONT of Q

DFS Example: Romania

“pop” Q
Zerind ← Q = [Sibiu, Timisoara]

Is it the goal? NO

Q = [Arad, Oradea, Sibiu, Timisoara]

add children of Zerind to FRONT of Q
DFS Example: Romania

```
Arad ← Q = [Oradea, Sibiu, Timisoara]
```

CYCLE!!!!!

Depth-first search can perform infinite cyclic excursions if not check for repeated-states along upward path in tree!

DFS Example: Binary Tree

○
DFS Example: Binary Tree

DFS Example: Binary Tree
DFS Example: Binary Tree

DFS Example: Binary Tree
DFS Example: Binary Tree
Properties of Depth-First Search

- Potentially not complete (can be bad)
  - Fails in infinite-depth spaces or with loops
- Time (bad)
  - $O(b^m)$, $m=$ maximum depth
  - Bad if $m$ is larger than depth of goal ($d$)
  - Good if multiple solutions (hit one)
- Space (better)
  - $O(mb)$, linear space (keep only leaf nodes “as you go”)
    - Need only store a single path from the root to the leaf node, along with remaining unexpanded sibling nodes for each node on path.
- Optimality (bad)
  - No, it returns the first deepest solution, so it could miss a shallower solution it has not yet seen (even at low depth)
“Depth-Limited” Search

- Depth-first search, with depth limit of \( l \)
- Avoids pitfalls of depth-first search by imposing a cutoff (stop) depth
- Implementation
  - Depth-first queue with nodes at depth \( l \) having no successors
- With 20 cities in the Arad to Bucharest example, know that a solution length < 20 exists
- Guaranteed to find solution (if exists), but not guaranteed to find shortest solution
  - Complete (if depth limit big enough), but not optimal
- Time and space complexity of depth-first search

Iterative Deepening Search

- Sidesteps issue of choosing best depth limit
  - Not really know a good depth limit for most problems
- Try all possible depth limits
  - First depth 0, then depth 1, then depth 2, …
- May seem wasteful, but overhead is not very costly
  - Because most of the nodes are toward bottom of tree
  - Not costly to generate upper nodes multiple times
- Preferred method with large search space and depth of solution not known
Iterative Deepening Search: $l=0$

[Stop]

Iterative Deepening Search: $l=1$
Iterative Deepening Search: $l=1$

[Stop]

Iterative Deepening Search: $l=2$
Iterative Deepening Search: $l=2$

- Arad
  - Zerind
  - Sibiu
  - Timisoara

- Orașel
  - Arad
  - Oradea
Iterative Deepening Search: $l=2$

[Diagram showing a tree structure with nodes and edges labeled with cities including Arad, Orașel, Siliva, Timișoara, Zerind, Făgăraș, and Vâlcea.]

[Stop]
## Properties of Iterative Deepening Search

- **Complete** (good)
  - Finds shallowest solution – as in breadth-first
- **Time** (not too bad)
  - $O(b^d)$, where $d$ is depth of shallowest solution
    - Breadth-first is $O(b^{d+1})$!
- **Space** (good)
  - $O(bd)$ – as in depth-first
- **Optimality** (good)
  - Yes $IDS$ combines benefits of depth-first and breadth-first search

### Nodes generated when goal at depth $d$

- (bottom level; root is depth 0) with branching factor $b$

\[
(d)b + (d-1)b^2 + \ldots + (1)b^d \rightarrow O(b^d)
\]

- Children of root generated $d$ times
- Nodes at bottom generated once

*Note: root node always generated/available*
Properties of Iterative Deepening Search

• Compare to Breadth-first search
  \[ b + b^2 + \ldots + b^d + (b^{d+1} - b) \rightarrow O(b^{d+1}) \]
  Assume goal is last node (e.g., rightmost) at depth \( d \)
  These are nodes already put in the queue/stack from the non-goal nodes at depth \( d \) (though not examined later)
  Depth-first search does not have these extra nodes

• Comparison for \( b = 10 \) and \( d = 5 \)
  – \#nodes(IDS) = 123,450
  – \#nodes(BFS) = 1,111,100

When To Use Iterative Deepening Search?

“In general, Iterative Deepening is the preferred uniformed search method when there is a large search space and the depth of the solution is not known.”
In-Class Exercise

Search Tree

Order child nodes “alphabetically” left to right
Search Tree
Summary

- Breadth-first search
  - Expands shallowest node in tree first
- Uniform-cost search
  - Expands least-cost (additive) leaf node first
- Depth-first search
  - Expands deepest node first
- Depth-limited search
  - Limits how deep a depth-first search can go
- Iterative deepening search ***
  - Depth-limited search with increasing limits until goal is found
  - Combines benefits of BFS and DFS