Artificial Intelligence

CS 165A

Jan 30, 2020

Instructor: Prof. Yu-Xiang Wang

→ Basic search
→ Informed search
Today

- Recap of Lecture 7
- Uninformed search
- Informed search
Example: Missionaries and Cannibals
(3 min discussion)

Problem: Three missionaries and three cannibals are on one side of a river, along with a boat that can hold one or two people. Find a way to get everyone to the other side, without ever leaving a group of missionaries in one place outnumbered by the cannibals in that place.

- States, operators, goal test, path cost?
M&C (cont.)

- Initial state

- Goal state

\[(3 \ 3 \ 1)\]

\[(M_L \ C_L \ B_L)\]

\[(0 \ 0 \ 0)\]
M&C (cont.)

(2 2 0)
M&C (cont.)

• Problem description \(<\{S\}, S_0, \{S_{G_j}\}, \{O_i\}, \{g_i\}>\)
• \(\{S\} : \{ (\{0,1,2,3\} \{0,1,2,3\} \{0,1\}) \} \)
• \(S_0 : (3 \ 3 \ 1)\)
• \(S_{G} : (0 \ 0 \ 0)\)
• \(g = 1\)
• \(\{O\} : \{ (x \ y \ b) \rightarrow (x' \ y' \ b') \} \)
• Safe state: \((x \ y \ b)\) is safe iff
  – \(x > 0\) implies \(x \geq y\) and
    \(x < 3\) implies \(y \geq x\)
  – Can be restated as
    \((x = 1 \ or \ x = 2)\) implies \(x = y\)

Operators:

\[(x \ y \ 1) \rightarrow (x-2 \ y \ 0)\]
\[(x \ y \ 1) \rightarrow (x-1 \ y-1 \ 0)\]
\[(x \ y \ 1) \rightarrow (x \ y-2 \ 0)\]
\[(x \ y \ 1) \rightarrow (x-1 \ y \ 0)\]
\[(x \ y \ 1) \rightarrow (x \ y-1 \ 0)\]
\[(x \ y \ 0) \rightarrow (x+2 \ y \ 1)\]
\[(x \ y \ 0) \rightarrow (x+1 \ y+1 \ 1)\]
\[(x \ y \ 0) \rightarrow (x+1 \ y \ 1)\]
\[(x \ y \ 0) \rightarrow (x \ y+1 \ 1)\]
M&C (cont.)

- 11 steps
- $5^{11} = 48$ million states to explore

One solution path:

(3 3 1)
(2 2 0)
(3 2 1)
(3 0 0)
(3 1 1)
(1 1 0)
(2 2 1)
(0 2 0)
(0 3 1)
(0 1 0)
(0 2 1)
(0 0 0)
More quizzes: PACMAN

• The goal of a simplified PACMAN is to get to the pellet as quick as possible.
  – For a grid of size 30*30. Everything static.
  – What is a reasonable representation of the State, Operators, Goal test and Path cost?
More quizzes: PACMAN with static ghosts

- The goal is to eat all pellets as quickly as possible while staying alive. Eating the “Power pellet” will allow the pacman to eat the ghost.

- Think about how to formulate this problem. We will revisit it in the next lecture.
Quick summary on problem formulation

• Formulate problems as a search problem
  – Decide your level of abstraction. State, Action, Goal, Cost.
  – Represented by a state-diagram
  – Required solution: A sequence of actions
  – Optimal solution: A sequence of actions with minimum cost.

• Caveats:
  – Might not be a finite graph
  – Might not have a solution
  – Often takes exponential time to find the optimal solution

Let’s try solving it anyways!
  - Do we need an exact optimal solution?
  - Are problems in practice worst case?
Searching for Solutions

• Finding a solution is done by searching through the state space
  – While maintaining a set of partial solution sequences
• The search strategy determines which states should be expanded first
  – Expand a state = Applying the operators to the current state and thereby generating a new set of successor states
• Conceptually, the search process builds up a search tree that is superimposed over the state space
  – Root node of the tree ↔ Initial state
  – Leaves of the tree ↔ States to be expanded (or expanded to null)
  – At each step, the search algorithm chooses a leaf to expand
State Space vs. Search Tree

• The **state space** and the **search tree** are not the same thing!
  – A *state* represents a (possibly physical) configuration
  – A *search tree node* is a **data structure** which includes:
    • \{ parent, children, depth, path cost \}
  – States do not have parents, children, depths, path costs
  – Number of states ≠ number of nodes
State Space vs. Search Tree (cont.)

State space: 8 states
State Space vs. Search Tree (cont.)

Search tree (partially expanded)
Search Strategies

• Uninformed (blind) search
  – Can only distinguish goal state from non-goal state

• Informed (heuristic) search
  – Can evaluate states
Uninformed (“Blind”) Search Strategies

• No information is available other than
  – The current state
    • Its parent (perhaps complete path from initial state)
    • Its operators (to produce successors)
  – The goal test
  – The current path cost (cost from start state to current state)

• Blind search strategies
  – Breadth-first search
  – Uniform cost search
  – Depth-first search
  – Depth-limited search
  – Iterative deepening search
  – Bidirectional search
General Search Algorithm (Version 1)

• Various strategies are merely variations of the following function:

```plaintext
function GENERAL-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
end
```

(Called “Tree-Search” in the textbook)
General Search Algorithm (Version 2)

- Uses a queue (a list) and a queuing function to implement a search strategy
  - Queuing-Fn(queue, elements) inserts a set of elements into the queue and determines the order of node expansion

```plaintext
function GENERAL-SEARCH(problem, QUEUING-FN) returns a solution or failure

nodes ← MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem]))
loop do
  if nodes is empty then return failure
  node ← REMOVE-FRONT(nodes)
  if GOAL-TEST[problem] applied to STATE(node) succeeds then return node
  nodes ← QUEUING-FN(nodes, EXPAND(node, OPERATORS[problem]))
end
```

Uses a queue (a list) and a queuing function to implement a search strategy.

Queuing-Fn(queue, elements) inserts a set of elements into the queue and determines the order of node expansion.
How do we evaluate a search algorithm?

• Primary criteria to evaluate search strategies
  – **Completeness**
    • Is it guaranteed to find a solution (if one exists)?
  – **Optimality**
    *Note that this is not saying it’s space/time complexity is optimal.*
    • Does it find the “best” solution (if there are more than one)?
  – **Time complexity**
    • Number of nodes generated/expanded
    • (How long does it take to find a solution?)
  – **Space complexity**
    • How much memory does it require?

• Some performance measures
  – Best case
  – Worst case
  – Average case
  – Real-world case
How do we evaluate a search algorithm?

- Complexity analysis and \( O(\ ) \) notation (see Appendix A)
  - \( b = \) Maximum branching factor of the search tree
  - \( d = \) Depth of an optimal solution (may be more than one)
  - \( m = \) maximum depth of the search tree (may be infinite)

- Examples
  - \( O( b^3 d^2 ) \) – polynomial time
  - \( O( b^d ) \) – exponential time

For chess, \( b_{ave} = 35 \)

\( b = 2, \ d = 2, \ m = 3 \)
Breadth-First Search

- All nodes at depth $d$ in the search tree are expanded before any nodes at depth $d+1$
  - First consider all paths of length $N$, then all paths of length $N+1$, etc.
- Doesn’t consider path cost – finds the solution with the shortest path
- Uses FIFO queue

function **BREADTH-FIRST-SEARCH**(*problem*) returns a solution or failure
return **GENERAL-SEARCH**(*problem*, **ENQUEUE-AT-END**)
Example

State space graph

Search tree

Queue

(A)
(B C)
(C D)
(D B D E)
(B D E)
(D E D)
(E D)
(D F)
(F)
( )
Breadth-First Search

- Complete? Yes
- Optimal? If shallowest goal is optimal
- Time complexity? Exponential: $O(b^{d+1})$
- Space complexity? Exponential: $O(b^{d+1})$

In practice, the memory requirements are typically worse than the time requirements

$b = \text{branching factor (require finite } b)$
$d = \text{depth of shallowest solution}$
Depth-First Search

• Always expands one of the nodes at the deepest level of the tree
  – Low memory requirements
  – Problem: depth could be infinite
• Uses a stack (LIFO)

function \textsc{Depth-First-Search}(\textit{problem}) \textbf{returns} a solution or failure
return \textsc{General-Search}(\textit{problem, Enqueue-At-Front})
Example

State space graph

Search tree

Queue

(A)
(B C)
(D C)
(C)
(B D E)
(D D E)
(D E)
(E)
(F)
Depth-First Search

- **Complete?** No
- **Optimal?** No
- **Time complexity?** Exponential: \( O(b^m) \)
- **Space complexity?** Polynomial: \( O(bm) \)

\[ m = \text{maximum depth of the search tree} \]

(may be infinite)
**Space complexity of DFS**

- Why is the *space* complexity (memory usage) of depth-first search $O(bm)$?
  - Remove expanded node when all descendents evaluated
  - At each of the $m$ levels, you have to keep $b$ nodes in memory

**Example:**

$b = 3$

$m = 6$

Nodes in memory: $bm + 1 = 19$

Actually, $(b-1)m + 1 = 13$ nodes, the way we have been keeping our node list
Depth-Limited Search

- Like depth-first search, but uses a depth cutoff to avoid long (possibly infinite), unfruitful paths
  - Do depth-first search up to depth limit $l$
  - Depth-first is special case with limit = $inf$

- Problem: How to choose the depth limit $l$?
  - Some problem statements make it obvious (e.g., TSP), but others don’t (e.g., MU-puzzle)

```plaintext
function DEPTH-LIMITED-SEARCH(problem, depth-limit) returns a solution or failure

return GENERAL-SEARCH(problem, ENQUEUE-AT-FRONT-IF-UNDER-DEPTH-LIMIT)
```

Must explicitly represent node depth
Depth-Limited Search

- Complete?  No, unless $d \leq l$
- Optimal?  No
- Time complexity?  Exponential: $O(b^l)$
- Space complexity?  Exponential: $O(bl)$

$l = \text{depth limit}$
Iterative-Deepening Search

- Since the depth limit is difficult to choose in depth-limited search, use depth limits of \( l = 0, 1, 2, 3, \ldots \)
  - Do depth-limited search at each level

```plaintext
function **ITERATIVE-DEEPPING-SEARCH**(problem) returns a solution or failure
for depth \( \leftarrow 0 \) to \( \infty \) do
    if **DEPTH-LIMITED-SEARCH**(problem, depth) succeeds then return result
end
return failure
```
Iterative-Deepening Search

- IDS has advantages of
  - Breadth-first search – similar optimality and completeness guarantees
  - Depth-first search – Modest memory requirements

- This is the preferred blind search method when the search space is *large* and the solution depth is *unknown*

- Many states are expanded multiple times
  - Is this terribly inefficient?
    - No… and it’s great for memory (compared with breadth-first)
    - Why is it not particularly inefficient?
Iterative-Deepening Search: Efficiency

- Complete? Yes
- Optimal? Same as BFS
- Time complexity? Exponential: $O(b^d)$
- Space complexity? Polynomial: $O(bd)$
Bidirectional Search

Forward search only:
Bidirectional Search

Simultaneously search forward from the initial state and backward from the goal state

Much more efficient!
Bidirectional Search

- $O(b^{d/2})$ rather than $O(b^d)$ – hopefully
- Both actions and predecessors (inverse actions) must be defined
- Must test for intersection between the two searches
  - Constant time for test?
- Really a search strategy, not a specific search method
  - Often not practical….

Example:

$4^{10} \approx 1,000,000$

$2 \times 4^5 \approx 2,000$
Bidirectional Search

- Complete?    Yes
- Optimal?     Same as BFS
- Time complexity?  Exponential: $O(b^{d/2})$
- Space complexity? Exponential: $O(b^{d/2})$

* Assuming breadth-first search used from both ends
Uniform Cost Search

• Similar to breadth-first search, but always expands the lowest-cost node, as measured by the path cost function, $g(n)$
  – $g(n)$ is (actual) cost of getting to node $n$
  – Breadth-first search is actually a special case of uniform cost search, where $g(n) = \text{DEPTH}(n)$
  – If the path cost is monotonically increasing, uniform cost search will find the optimal solution

function **UNIFORM-COST-SEARCH**(problem) returns a solution or failure
return **GENERAL-SEARCH**(problem, **ENQUEUE-IN-COST-ORDER**)  

(Dijkstra’s algorithm of an potentially infinite graph)
Example

Try breadth-first and uniform cost
Uniform-Cost Search

• Complete? Yes if \(\varepsilon > 0\)

• Optimal? If minimum step cost > 0

• Time complexity? Exponential: \(O(b^{\left\lfloor \frac{C}{\varepsilon} \right\rfloor})\)

• Space complexity? Exponential: \(O(b^{\left\lfloor \frac{C}{\varepsilon} \right\rfloor})\)

C = optimal cost
\(\varepsilon = \) minimum step cost > 0

Same as breadth-first if all edge costs are equal
Can we do better than Tree Search?

• Sometimes.

• When the number of states are small
  – Dynamic programming (smart way of doing exhaustive search)
State Space vs. Search Tree (cont.)

Search tree (partially expanded)
Search Tree => Search Graph

Dynamic programming (with book keeping)

\[ O(b^d) \Rightarrow O(\# \text{ of states}) \]
Graph Search vs Tree Search

• Tree Search
  – We might repeat some states
  – But we do not need to remember states

• Graph Search
  - We remember all the states that have been explored
  - But we do not repeat some states
## Summary table of uninformed search

<table>
<thead>
<tr>
<th>Criteria</th>
<th>BFS</th>
<th>Uniform-cost</th>
<th>DFS</th>
<th>Depth-limited</th>
<th>IDS</th>
<th>Bidirectional</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete?</td>
<td>Yes#</td>
<td>Yes#&amp;</td>
<td>No</td>
<td>No</td>
<td>Yes#</td>
<td>Yes#+</td>
</tr>
<tr>
<td>Time</td>
<td>$O(b^d)$</td>
<td>$O(b^{1+\frac{C}{e}})$</td>
<td>$O(b^m)$</td>
<td>$O(b^l)$</td>
<td>$O(b^d)$</td>
<td>$O(b^{d/2})$</td>
</tr>
<tr>
<td>Space</td>
<td>$O(b^d)$</td>
<td>$O(b^{1+\frac{C}{e}})$</td>
<td>$O(bm)$</td>
<td>$O(bl)$</td>
<td>$O(bd)$</td>
<td>$O(b^{d/2})$</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>
</tbody>
</table>

\(b\): Branching factor
\(d\): Depth of the shallowest goal
\(l\): Depth limit
\(m\): Maximum depth of search tree
\(e\): The lower bound of the step cost

\#: Complete if \(b\) is finite
\&: Complete if step cost \(\geq e\)
\$: Optimal if all step costs are identical
\+: If both direction use BFS

(Section 3.4.7 in the AIMA book.)
The computer can’t “see” the search graph like we can
– No “bird’s eye view” – make relevant information explicit!

What information should you keep for a node in the search tree?
– State
  • (1 2 0)
– Parent node (or perhaps complete ancestry)
  • Node #3 (or, nodes 0, 2, 5, 11, 14)
– Depth of the node
  • $d = 4$
– Path cost up to (and including) the node
  • $g(node) = 12$
– Operator that produced this node
  • Operator #1
Upcoming next

• Informed search

• Some questions / desiderata
  1. Can we do better with some side information?
  2. We do not wish to make strong assumptions on the side information.
  3. If the side information is good, we hope to do better.
  4. If the side information is useless, we perform as well as an uninformed search method.
Best-First Search (with an Eval-Fn)

function **Best-First-Search**(problem, Eval-Fn) returns a solution or failure
 QUEUING-FN ← a function that orders nodes by Eval-Fn
 return **General-Search**(problem, QUEUING-FN)

- Uses a heuristic function, $h(n)$, as the Eval-Fn
- $h(n)$ estimates the cost of the best path from state $n$ to a goal state
  - $h(goal) = 0$
Greedy Best-First Search

- Greedy search – always expand the node that appears to be the closest to the goal (i.e., with the smallest $h$)
  - Instant gratification, hence “greedy”

```plaintext
function Greedy-Search(problem, h) returns a solution or failure
return Best-First-Search(problem, h)
```

- Greedy search often performs well, but:
  - It doesn’t always find the best solution / or any solution
  - It may get stuck
  - It performance completely depends on the particular $h$ function
A* Search (Pronounced “A-Star”)  

- Uniform-cost search minimizes $g(n)$ (“past” cost)  
- Greedy search minimizes $h(n)$ (“expected” or “future” cost)  
- “A* Search” combines the two:  
  - Minimize $f(n) = g(n) + h(n)$  
  - Accounts for the “past” and the “future”  
  - Estimates the cheapest solution (complete path) through node $n$

function **A*-SEARCH**(problem, $h$) returns a solution or failure  
return **BEST-FIRST-SEARCH**(problem, $f$)
\[ f(n) = g(n) + h(n) \]
A* Example

\[ f = 0 + 366 = 366 \]

\[ f = 75 + 374 = 449 \]

\[ 140 + 253 = 393 \]

\[ 118 + 329 = 447 \]

\[ 291 + 380 = 671 \]

\[ 280 + 366 = 506 \]

\[ 239 + 178 = 417 \]

\[ 220 + 193 = 413 \]
When does A* search “work”?

- Focus on optimality (finding the optimal solution)

- “A* Search” is optimal if $h$ is **admissible**
  - $h$ is optimistic – it never overestimates the cost to the goal
    - $h(n) \leq$ true cost to reach the goal
  - So $f(n)$ never overestimates the actual cost of the best solution passing through node $n$
Visualizing A* search

- A* expands nodes in order of increasing $f$ value
- Gradually adds "$f$-contours" of nodes
- Contour $i$ has all nodes with $f = f_i$, where $f_i < f_{i+1}$
Optimality of A* with an Admissible h

• Let OPT be the optimal path cost.
  – All non-goal nodes on this path have \( f \leq \text{OPT} \).
    • Positive costs on edges
      – The goal node on this path has \( f = \text{OPT} \).

• A* search does not stop until an f-value of OPT is reached.
  – All other goal nodes have an f cost higher than OPT.

• All non-goal nodes on the optimal path are eventually expanded.
  – The optimal goal node is eventually placed on the priority queue, and reaches the front of the queue.
Optimal Efficiency of A*

A* is **optimally efficient** for any particular $h(n)$

That is, no other optimal algorithm is guaranteed to expand fewer nodes with the same $h(n)$.

- Need to find a good and efficiently evaluable $h(n)$. 
A* Search with an Admissible $h$

- Optimal? Yes
- Complete? Yes
- Time complexity? Exponential; better under some conditions
- Space complexity? Exponential; keeps all nodes in memory
Recall: Graph Search vs Tree Search

• Tree Search
  – We might repeat some states
  – But we do not need to remember states

• Graph Search
  - We remember all the states that have been explored
  - But we do not repeat some states
Avoiding Repeated States using A* Search

• Is GRAPH-SEARCH optimal with A*?

Graph Search
Step 1: Among B, C, E, Choose C
Step 2: Among B, E, D, Choose B
Step 3: Among D, E, Choose E. (you are not going to select C again)

Try with TREE-SEARCH and GRAPH-SEARCH
Avoiding Repeated States using A* Search

• Is GRAPH-SEARCH optimal with A*?

Solution 1: Remember all paths: Need extra bookkeeping

Solution 2: Ensure that the first path to a node is the best!

Try with TREE-SEARCH and GRAPH-SEARCH
Consistency (Monotonicity) of heuristic $h$

- A heuristic is consistent (or monotonic) provided
  - for any node $n$, for any successor $n'$ generated by action $a$ with cost $c(n,a,n')$
    - $h(n) \leq c(n,a,n') + h(n')$
  - akin to triangle inequality.
  - guarantees admissibility (proof?).
  - values of $f(n)$ along any path are non-decreasing (proof?).
    - Contours of constant $f$ in the state space
- GRAPH-SEARCH using consistent $f(n)$ is optimal.
- Note that $h(n) = 0$ is consistent and admissible.