Today’s lecture

- Overview of Search Strategies
- Blind Search (Most slides will be skipped)
- Informed Search
  - How to use heuristics (domain knowledge) in order to accelerate search?
  - A* and IDA*
- Reading: Sections 4.1-4.2.

General Tree Search

```
function TREE-SEARCH(problem, fringe) returns a solution, or failure
fringe = INSERT(MAKE-NODE(InitialState[problem]), fringe)
loop do
  if fringe is empty then return failure
  node = REMOVE-FRONT(fringe)
  if GOAL-TEST[problem] applied to STATE(node) succeeds: return node
  fringe = INSERT-ALL(EXPAND(node, problem), fringe)
```

```
function EXPAND(node, problem) returns a set of nodes
successors = the empty set
for each action, result in SUCCESSOR-Fs(problem)[STATE[node]] do
  s = a new node
  PARENT[NODE[s]] = node; ACTION[s] = action; STATE[s] = result
  Path-Cost[s] = Path-Cost[node] + Step-Cost(node, action, s)
  Depth[s] = Depth[node] + 1
  add s to successors
return successors
```

Set up state of node
Repeated states

- Failure to detect repeated states can turn a linear problem into an exponential one!

Avoiding Repeated States

- Do not re-generate the state you just came from.
- Do not create paths with cycles.
- Do not generate any state that was generated before (using a hash table to store all generated nodes)
  - Markov Assumption
- Add **Close list** to search algorithm or cleverly construct search space

Graph Search

```python
function Graphs-Search(problem, fringe) returns a solution, or failure
    closed ← an empty set
    fringe ← INSERT(Make-Node(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(node) then return node
        if node is not in closed then
            add STATE[node] to closed
            fringe ← INSERT-ALL(Examine(node, problem), fringe)
        end
    end
    Don't expand node if already on closed list
```

Search Strategies

- A key issue in search is limiting the portion of the state space that must be explored to find a solution.
- The portion of the search space explored can be affected by the order in which states (and thus solutions) are examined.
- The search strategy determines this order by determining which node (partial solution) will be expanded next.
A key issue in search is limiting the portion of the state space that must be explored to find a solution.

The portion of the search space explored can be affected by the order in which states (and thus solutions) are examined.

The search strategy determines this order by determining which node (partial solution) will be expanded next.

A strategy: picking the order of node expansion

Evaluation criteria:

- **Completeness**: is the strategy guaranteed to find a solution when there is one?
- **Time complexity**: how long does it take to find a solution? (number of nodes generated)
- **Space complexity**: how much memory does it need to perform the search? (maximum number of nodes in memory)
- **Optimality**: does the strategy find the highest-quality solution when there are several solutions?

Time and space complexity are measured:

- $b$ – maximum branching factor of the search tree
- $d$ – depth of the least-cost solution
- $m$ – maximum depth of the state space (may be infinity)

The extent to which partial solutions are kept and used to guide the search process

The context used in making decisions

The degree to which the search process is guided by domain knowledge

The degree to which control decisions are made dynamically at run-time

Search strategies can be classified in the following general way:

- Uninformed/blind search;
- Informed/heuristic search;
  - Relationship of nodes to goal state, and intra-node relationships
  - Multi-level/multi-dimensional/multi-direction;
- Systematic versus Stochastic
- Game/Adversarial search
  - Game search deals with the presence of an opponent that takes actions that diminish an agent’s performance (see AIMA Chapter 6)
Uninformed/Blind Search Strategies

- Uninformed strategies do not use any information about how close (distance, cost) a node might be to a goal (additional cost to reach goal).

- They differ in the order that nodes are expanded (and operator cost assumptions).

Examples of Blind Search Strategies

- Breadth-first search (open list is FIFO queue)
- Uniform-cost search (shallowest node first)
- Depth-first search (open list is a LIFO queue)
- Depth-limited search (DFS with cutoff)
- Iterative-deepening search (incrementing cutoff)
- Bi-directional search (forward and backward)

Will Skip Following Slides in Class Discussion

- Breadth-first Search
- Uniform-cost Search
- Depth-first Search
- Depth-limited Search

Breadth-First Search

Expand shallowest unexpanded node

Fringe: is a FIFO queue, new successors go to the end
Breadth-First Search

(b - branching factor, d -depth)

- Completeness:
  - Yes
- Time complexity:
  - $1 + b + b^2 + b^3 + \ldots + b^d = O(b^{d+1})$
- Space complexity:
  - $O(b^{d+1})$, keep every node in memory.
- Optimality:
  - Yes (only if step costs are identical)

Time and Memory requirements for a breadth-first search.
- The figures shown assume (1) branching factor $b=10$; (2) 1000 nodes/second; (3) 100 bytes/node
- Time and Space complex Cannot be use to solve any but the smallest problem

Uniform Cost Search

- BFS finds the shallowest goal state.
- Uniform cost search modifies the BFS by expanding ONLY the lowest cost node (as measured by the path cost $g(n)$)
- The cost of a path must never decrease as we traverse the path, i.e. no negative cost should in the problem domain

Time complexity: $b^{C^*}$
- Optimality: Yes
- Space complexity: $b^{C^*}$

$C^*$: optimal cost, $c$: minimum cost of each step
Depth-First Search (Cont’d)

Expand deepest unexpanded node

Fringe: is a LIFO queue, new successors go to the front

- DFS always expands one of the nodes at the deepest level of the tree.
- The search only go back once it hits a dead end (a non-goal node with no expansion)
- DFS have modest memory requirements, it only needs to store a single path from root to a leaf node.
- For problems that have many solutions, DFS may actually be faster than BFS, because it has a good chance of finding a solution after exploring only a small portion of the whole space.

Properties of Depth-first Search

- Complete: ??
- Time: ??
- Space: ??
- Optimal: ??

One problem with DFS is that it can get stuck going down the wrong path.

Many problems have very deep or even infinite search trees.

DFS should be avoided for search trees with large or infinite maximum depths.

It is common to implement a DFS with a recursive function that calls itself on each of its children in turn.
Properties of Depth-first Search

- Complete: No, fails in infinite-depth spaces, spaces with loops. Modify to avoid repeated states along path: complete in finite spaces
- Time: $b^m$: terrible if $m$ is much larger than $d$ (depth of solution), but if solutions are dense, may be much faster than breadth-first
- Space: $b^m$, i.e., linear space!
- Optimal: no
  - $b$ – maximum branching factor of the search tree
  - $m$ – maximum depth of the state space

Depth-Limited Search

“Practical” DFS
- DLS avoids the pitfalls of DFS by imposing a cutoff on the maximum depth of a path.
- However, if we choose a depth limit that is too small, then DLS is not even complete.
- The time and space complexity of DLS is similar to DFS.

Depth-Limited Search (cont)

Completeness:
- Yes, only if $l \geq d$

Time complexity:
- $b^l$

Space complexity:
- $bl$

Optimality:
- No

Iterative Deepening Search

The hard part about DLS is picking a good limit.

IDS is a strategy that sidesteps the issue of choosing the best depth limit by trying all possible depth limits: first depth 0, then depth 1, the depth 2, and so on.
Iterative Deepening Search (cont)

Repeated Depth-limited search where depth-limit is increased incrementally until solution is found

Properties of iterative deepening search

- Complete: Yes
- Time: $O(bd)$
- Space: $O(bd)$
- Optimal: Yes, if step cost = 1
Can be modified to explore uniform-cost tree

Iterative Deepening Search (cont)

IDS may seem wasteful, because so many states are expanded multiple times.
For most problems, however, the overhead of this multiple expansion is actually rather small.
- Major cost is at fringe where solution is; this last fringe only occurs once
- In effect, it combines the benefits of DFS and BFS. It is optimal and complete, like BFS and has modest memory requirements of DFS.
- IDS is the preferred search method when there is a large search space and the depth of the solution is not known.
**Bi-directional Search**

A schematics view of a bi-directional BFS that is about to succeed, when a branch from the start node meets a branch from the goal node.

**Comparing Blind Search Strategies**

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Uniform-First</th>
<th>Breadth-First</th>
<th>Depth-Limited</th>
<th>Depth-First</th>
<th>Informed</th>
<th>Bi-directional</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Space</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Optimal?</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

- **b** - branching factor; **d** is the depth of the solution; **m** is the maximum depth of the search tree; **l** is the depth limit.
- BFS, IDS, and BDS are optimal only if step costs are all identical.
- Iterative deepening search uses only linear space and not much more time.

**Informed/Heuristic Search**

- While uninformed search methods can in principle find solutions to any state space problem, they are typically too inefficient to do so in practice.
- Informed search methods use *problem-specific knowledge* to improve average search performance.
What are heuristics?

- **Heuristic**: problem-specific knowledge that reduces expected search effort.
- Informed search uses a heuristic evaluation function that denotes the relative desirability of expanding a node/state.
  - often include some estimate of the cost to reach the nearest goal state from the current state.
  - How much of the state of the search does it take into account in making this decision?
- In blind search techniques, such knowledge can be encoded only via state space and operator representation.

Examples of heuristics

- Travel planning
  - Euclidean distance
  - 8-puzzle
    - **Manhattan distance**
    - **Number of misplaced tiles**
  - Traveling salesman problem
  - Minimum spanning tree

Where do heuristics come from?

Heuristics from relaxed models

- Heuristics can be generated via simplified models of the problem
- Simplification can be modeled as deleting constraints on operators
- Key property: Heuristic can be calculated efficiently (low overhead -- why important?)

Informed Search Strategies

- Best-first search (a.k.a. ordered search):
  - **greedy** (a.k.a. best-first)
  - **A***
  - ordered depth-first (a.k.a. hill-climbing)
- Memory-bounded search:
  - Iterative deepening A*** (IDA***)
  - Simplified memory-bounded A*** (SMA***)
- Recursive Best First Search (RBFS)
- Time-bounded search:
  - Anytime A***
  - RTA*** (searching and acting)
- Iterative improvement algorithms (generate-and-test approaches):
  - Steepest ascent hill-climbing
  - Random-restart hill-climbing
  - Simulated annealing
- Multi-Level/Multi-Dimensional Search
  - Hierarchical A***
  - Blackboard
Informed/Heuristic Search

- While uninformed search methods can in principle find solutions to any state space problem, they are typically too inefficient to do so in practice.
- Informed search methods use problem-specific knowledge to improve average search performance.

What are heuristics?

- Heuristic: problem-specific knowledge that reduces expected search effort.
  - In blind search techniques, such knowledge can be encoded only via state space and operator representation.
- Informed search uses a heuristic evaluation function that denotes the relative desirability of expanding a node/state.
  - Often include some estimate of the cost to reach the nearest goal state from the current state.

One Way of Introducing Heuristic Knowledge into Search – Heuristic Evaluation Function

- Heuristic evaluation function \( h : \Psi \rightarrow R \), where \( \Psi \) is a set of all states and \( R \) is a set of real numbers, maps each state \( s \) in the state space \( \Psi \) into a measurement \( h(s) \) which is an estimate of the cost extending of the cheapest path from \( s \) to a goal node.

Node A has 3 children.

- \( h(s1)=0.8, h(s2)=2.0, h(s3)=1.6 \)
- The value refers to the cost involved for an action. A continued search at \( s1 \) based on \( h(s1) \) being the smallest is ‘heuristically’ the best.

Best-first search

- Idea: use an evaluation function for each node, which estimates its “desirability”
- Expand most desirable unexpanded node
- Implementation: open list is sorted in decreasing order of desirability
- A combination of depth first (DFS) and breadth first search (BFS).
  - Go depth-first until node path is no longer the most promising one (lowest expected cost) then backup and look at other paths that were previously promising (and now are the most promising) but not pursued. At each search step pursuing in a breath-first manner the paths that have lowest expected cost.
Best-First Search*

1) Start with OPEN containing just the initial state.
2) Until a goal is found or there are no nodes left on OPEN do:
   (a) Pick the best node (based on the heuristic function) on OPEN.
   (b) If it is a goal node, return the solution otherwise place node on the CLOSED list.
   (c) Generate its successors.
   (d) For each successor node do:
      i. If it has not been generated before (i.e., not on CLOSED list), evaluate it, add it to OPEN, and record its parent.
      ii. If it has been generated before, change the parent if this new path is better than the previous one. In that case, update the cost of getting to this node and to any successors that this node may already have.

Greedy search

- Simple form of best-first search
- Heuristic evaluation function $h(n)$ estimates the cost from $n$ to the closest goal
- Example: straight-line distance from city $n$ to goal city (Bucharest)
- Greedy search expands the node (on OPEN list) that appears to be closest to the goal
- Properties of greedy search?
Problems with Greedy Search (cont)

- Complete?
  - No, can get stuck in loops if not maintaining Closed list.
    - e.g. (Lasi to Fagaras) Lasi to Neamt to Lasi to Neamt
    - Neamt dead-end need to turn back

- Time??
  - $O(b^m)$, but a good heuristic can give dramatic improvement
    where $m$ is the maximum depth of the search space

- Space??
  - $O(b^m)$, keeps all nodes in memory

- Optimal??
  - No (minimum cost path in example is 418 rather than 450)

Minimizing total path cost: A*

- $G($greedy$)$ minimizes the estimate cost to the goal, $h(n)$, - not optimal and incomplete.
- $U($uniform$)$ minimizes the cost of the path so far, $g(n)$ and is optimal and complete but can be very inefficient.
- A* Search combines both GS $h(n)$ and UCS $g(n)$ to give $f(n)$ which estimates the cost of the cheapest solution through $n$.
- A* is similar to best-first search except that the evaluation is based on total path (solution) cost:
  \[ f(n) = g(n) + h(n) \]
  
  where:
  
  $g(n)$ = cost of path from the initial state to $n$
  
  $h(n)$ = estimate of the remaining distance

Greedy -- Complete, but not optimal…

- When $h$ is admissible, monotonicity can be maintained when combined with pathmax:
  \[ f(n') = \max(f(n), g(n') + h(n')) \]

  Equivalent to defining a new heuristic function $h'$ and save $f$ of parent as part of state of node

  \[ f(n') = g(n') + h'(n'), h'(n') = \max(f(n) - g(n'), h(n')) \]
f = g + h
Admissibility and Monotonicity∗

- Admissible heuristic $h(n) = \text{never overestimates the actual cost } h^\ast(n) \text{ to reach a goal } \& \ h(n) \geq 0 \& h(\text{goal}) = 0$
- Monotone (Consistency) heuristic
  - For every node $n$ and every successor $n'$ reached from $n$ by action $a$:
    - $h(n) \leq \text{cost of } (n,a,n') + h(n')$ & $h(\text{goal}) = 0$
  - The deeper you go along a path the better (or as good) the estimate of the distance to the goal state.
  - the $f$-value (which is $g+h$) never decreases along any path.
  - Implies $\implies$ Each state reached has the minimal $g(n)$

- When $h$ is admissible, monotonicity can be maintained when combined with pathmax: $f(n') = \max(f(n), g(n') + h(n'))$
  - Create new $h^\ast \leftarrow h^\ast(n') := \max(f(n), g(n') + h(n'))$
  - Force $f$ to never decrease along path since $f(n') \geq f(n)$

  Does monotonicity in $f$ imply admissibility?

Next lecture

- Finish off Discussion of A*
- IDA*
- Other Time and Space Variations of A*
  - RBFS
  - SMA*
  - RTA*