Review: Search

Tree and graph search: strategy depends on **AddToQueue**

input: $S, x_i, G(\cdot)$

```plaintext
AddToQueue($x_i, Queue$); // Add $x_i$ to a queue of nodes to be expanded

while(!IsEmpty($Queue$))
    $x \leftarrow$ Front($Queue$); // Retrieve the front of the queue
    if($x.expanded == true$) continue; // Do not expand a node twice
    if($G(x)$) return solution; // Return if goal is reached
    $x.expanded = true$; // Mark $x$ as expanded

    for each successor $n_i$ of $x$ // Add all neighbors of $x$ to the queue
        if($n_i.expanded == false$) AddToQueue($n_i, Queue$)
```

**Different search algorithms use different AddToQueue**

- Breadth first search (BFS): FIFO queue – first in first out
- Depth first search (DFS): LIFO queue – last in first out
- Uniform-cost: Priority queue – node with smallest cost in the front

Tree search CANNOT tell whether a node/state has been seen before
Review: Tree/Graph Search Example

 défini le problème graphique

 Produit BFS arbre et graph search

 Produit BFS arbre et graph search

 Adjacency list
 S: A, B
 A: B, C, G
 B: C
 C: G
 G:

 Tree search

 Graph search

 Nodes may be expanded again

 More compact search tree!
## Review: Summary on Uninformed (Tree) Search

<table>
<thead>
<tr>
<th>Method</th>
<th>Complete?</th>
<th>Optimal?</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BFS</strong></td>
<td>Yes, for ( b &lt; \infty )</td>
<td>Yes, for uniform edge costs</td>
<td>( O(b^d) )</td>
<td>( O(b^d) )</td>
</tr>
<tr>
<td>Uniform cost</td>
<td>Yes, for ( b &lt; \infty ) and ( \epsilon &gt; \text{fixed } \delta &gt; 0 )</td>
<td>Yes</td>
<td>( O(b^{1+\frac{C^*}{\epsilon}}) )</td>
<td>( O(b^{1+\frac{C^*}{\epsilon}}) )</td>
</tr>
<tr>
<td><strong>DFS</strong></td>
<td>No</td>
<td>No</td>
<td>( O(b^m) )</td>
<td>( O(bm) )</td>
</tr>
<tr>
<td>Depth limited DFS</td>
<td>No</td>
<td>No</td>
<td>( O(b^\ell) )</td>
<td>( O(b\ell) )</td>
</tr>
<tr>
<td>Iterative deepening DFS</td>
<td>Yes, for ( b &lt; \infty )</td>
<td>Yes, for uniform edge costs</td>
<td>( O(b^d) )</td>
<td>( O(bd) )</td>
</tr>
<tr>
<td>Bidirectional</td>
<td>Yes, for BFS with ( b &lt; \infty )</td>
<td>Yes, for uniform edge costs</td>
<td>( O(b^{\frac{d}{2}}) )</td>
<td>( O(b^{\frac{d}{2}}) )</td>
</tr>
</tbody>
</table>

- \( b \): branching factor (average)
- \( d \): max. solution depth (i.e., depth of \( x_G \))
- \( m \): max. depth of any node from \( x_I \)
- \( \ell \): depth limit
- \( C^* \): optimal solution cost
- \( \epsilon \): smallest step (edge) cost, must be finite

\[ \Rightarrow \text{For graph search on a finite state space, DFS is complete and depth limited DFS is complete when } \ell \geq d \]
Lecture 3
Informed Search

CS 440: Intro AI
Jingjin Yu | Rutgers
Informed (Heuristic) Search

⇒ Uses domain knowledge relevant to the problem
⇒ The goal is to **limit the search space** that will be explored
⇒ For this lecture
  ⇒ Evaluation function $f(x)$ and heuristic function $h(x)$
  ⇒ Greedy best-first search
  ⇒ A* Best-first search
  ⇒ Selection of heuristics
Uniform-Cost Search, Revisited

UC maintains queue order based on current cost

In carrying out UC search, we also compute cost-to-come

⇒ S: 0, A: 1, B: 3, C: 5, G:8

⇒ Cost-to-come for a node \( x \) is often denoted \( g(x) \)
For uniform-cost search, node with the smallest cost-to-come (usually denoted $g(x)$) is placed in the front of the queue.

We say that UC uses the cost-to-come as the evaluation function, that is, $f(x) = g(x)$.

Used by AddToQueue.

In tree/graph search, the heuristic function is often an estimated value of the cost-to-go function, i.e.,

$h(x) = \text{estimated cost of the cheapest path from the state at node } x \text{ to a goal}$

We can construct new $f(x)$ using $g(x)$ and $h(x)$. 

$h(x)$ = estimated cost of the cheapest path from the state at node $x$ to a goal
Admissible and Consistent Heuristic

⇒ Assume the cheapest path from \( x \) to a goal is \( C_x^* \), an **admissible heuristic** satisfies

\[
h(x) \leq C_x^*
\]

⇒ A **consistent** heuristic is defined as (a form of triangle inequality)

\[
h(n) \leq c(n, a, n') + h(n') \text{ and } h(x_G) = 0
\]

⇒ A consistent heuristic is always admissible (homework problem)

⇒ The reverse is not always true (we will see examples)

⇒ Example of heuristic functions

⇒ Manhattan distance

⇒ Straight-line distance

⇒ Admissible?
Greedy Best-First Search

Greedy best-first search uses $f(x) = h(x)$

Example: path from S to G, tree/graph search (same for this example)

<table>
<thead>
<tr>
<th>State</th>
<th>$h(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>7</td>
</tr>
<tr>
<td>A</td>
<td>6</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>G</td>
<td>0</td>
</tr>
</tbody>
</table>

Optimal?
Properties of Greedy Best-First Search

- Completeness: Yes, only for graph search in a finite state space
  - Tree search is not complete even for finite state space

For the example, tree search oscillates between Iasi and Neamț.
Properties of Greedy Best-First Search, Cont.

⇒ **Optimality**: No

⇒ **Time complexity**: $O(b^d)$ in worst case – exploring the entire tree, similar to running a BFS, but can potentially be much better

⇒ **Space complexity**: $O(b^d)$ – may need to remember all search tree nodes
A* Best-First Search

- A* uses $f(x) = g(x) + h(x)$ (cost-to-come + estimated cost-to-go)
- Example: path from S to G, tree search

<table>
<thead>
<tr>
<th>State</th>
<th>$h(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>7</td>
</tr>
<tr>
<td>A</td>
<td>6</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>G</td>
<td>0</td>
</tr>
</tbody>
</table>
Properties of A*

- **Completeness**: Yes, for finite state space
  - Tree search complete even for infinite state space when cost/step is positive
    - This is the case because repeated edges increase $g(x)$
    - Branching factor must be finite though

- **Optimality**: Yes, for tree search with admissible heuristic
  - Proof to follow
  - Graph search requires a stronger heuristic to guarantee optimality
    - Example to follow

- **Space and Time complexity**:
  - Highly dependent on the state space, can be exponential
  - No search node with cost higher than $C^*$ is expanded
  - Still, we may encounter a large number of search nodes
    - Can be exponential in the number of state space states
Proof of Optimality of A* (Admissible Heuristic)

⇒ Assume an optimal solution has cost $C^*$

⇒ If A* is non-optimal, a non-optimal path to $x_G$ with cost $C'$ must be expanded at $x_G$. Let the evaluation function for this search node be $f_1(x_G)$.

⇒ We know $f_1(x_G) = C' + 0 = C' > C^*$

⇒ Assume $n$ is the last node on an optimal path that is on the queue (i.e., $n$ is not expanded)
   ⇒ It is possible that $n = x_G$
   ⇒ There must be such an $n$, otherwise the optimal path has been found
   ⇒ This assumes tree search, which explores all branches

⇒ Then $f(n) = g(n) + h(n) \leq C^* < C' = f_1(x_G)$

⇒ This means $n$ should be expanded first

⇒ A contradiction ⇒ A* is optimal
A* Graph Search with Admissible Heuristic

Same example: path from S to G, graph search

\[
\begin{array}{c|c}
\text{State} & h(x) \\
\hline
S & 7 \\
A & 6 \\
B & 2 \\
C & 1 \\
G & 0 \\
\end{array}
\]

Not optimal!

\[ h(x) \text{ is inconsistent, e.g., } h(S) > c(S,B) + h(B), h(A) > c(A,B) + h(B) \]
A* Graph Search with Consistent Heuristic

⇒ Updated example: path from S to G, graph search

See Russell and Norvig for the short proof using consistent heuristic
Advantage of A* Search

Both A* and uniform-cost are optimal. Why A*?

- Because A* **biases** the search toward the goal
- A* may visit much fewer nodes
- Similarly, better heuristic $\rightarrow$ smaller explored area
Heuristic Function Design

⇒ For route finding problems, Euclidean distance is consistent
  ⇒ Also very efficient!

⇒ Designing heuristic function can be non-trivial

⇒ Consider two heuristics for the 8-puzzle

\[ h_1 : \text{number of misplaced pieces} \]

\[ h_2 : \text{sum of Manhattan distances to goal for all pieces} \]
Heuristic Function Design

\[ h_1: \text{#misplaced game pieces} = 8 \]

\[ h_2: \text{sum Manhattan distances} = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18 \]

⇒ Both heuristics are admissible (homework problem)

⇒ They are also consistent (also homework problem)

⇒ They are like Euclidean distances for grids
Evaluating Heuristic Functions

- We prefer heuristics that will make A* expand fewer nodes
- One method to evaluate the number of nodes expanded is through **effective branching factor**, $b^*$, defined implicitly as

  \[ N + 1 = 1 + b^* + (b^*)^2 + \cdots + (b^*)^d \]

- E.g., for $d = 5$ and $N = 52$, $b^* \approx 1.92$.
- Suggests the method is almost like a binary tree with roughly 2 branches per node
- The smaller $b^*$ is, the better.
- For $h_1$ and $h_2$, $h_2$ has lower $b^*$ (empirically)
- This is expected because $h_2 \geq h_1$ always holds
- We say $h_2$ **dominates** $h_1$ when $h_2 \geq h_1$ always holds
Automatic Generation of Heuristics

⇒ Both $h_1$ and $h_2$ are optimal solutions to a simpler version of the 8-puzzle
  
  ⇒ $h_1$: taking out all pieces has no cost, placing a piece has a cost of 1
  ⇒ $h_2$: each cell can hold multiple game pieces

⇒ In general, “relaxing” the problem can yield admissible heuristics

⇒ For example, if we do not care the label of some pieces, e.g.,
Combining Heuristics

⇒ Suppose we have multiple heuristics, $h_1, h_2, ..., h_n$
⇒ There is no dominance between any $h_i, h_j$.
⇒ We may construct a new heuristic

\[ h_{\text{max}}(x) = \max\{h_1(x), h_2(x), ..., h_n(x)\} \]

⇒ $h_i$ admissible for all $i \Rightarrow h_{\text{max}}$ is admissible
⇒ $h_i$ consistent for all $i \Rightarrow h_{\text{max}}$ is consistent
⇒ Clearly, $h_{\text{max}}$ dominates $h_1, h_2, ..., h_n$