Lecture 4
Uninformed Search Strategies,
Search with Costs
(Ch 3.1-3.5, 3.7.3)
Announcement

• Assignment 1
  • out tomorrow (Friday)
  • due Friday Jan 27

• Office hours (will be posted on course website)

Cristina: Thursday, 2pm – 3pm

TAs

<table>
<thead>
<tr>
<th>Day</th>
<th>Time</th>
<th>Location</th>
<th>TA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mondays</td>
<td>10-11am</td>
<td>TBD</td>
<td>Michael</td>
</tr>
<tr>
<td>Tuesdays</td>
<td>4-5pm</td>
<td>DLC table 1</td>
<td>Rui</td>
</tr>
<tr>
<td>Wednesdays</td>
<td>12-1pm</td>
<td>ICCS X151</td>
<td>Abed</td>
</tr>
<tr>
<td>Thursdays</td>
<td>1-2pm</td>
<td>ICCS X341</td>
<td>Mario</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fridays</td>
<td>2-3pm</td>
<td>ICCS X237</td>
<td>Jordon</td>
</tr>
</tbody>
</table>
Today’s Lecture

Recap from Previous lectures

• Depth first search - analysis
• Breadth first search
• Iterative deepening
• Search with costs
• Intro to heuristic search (time permitting)
Bogus Version of Generic Search Algorithm

Input: a graph
  - a set of start nodes
  - Boolean procedure $goal(n)$ that tests if $n$ is a goal node

$frontier := [g : g$ is a goal node]$;

While $frontier$ is not empty:
  - select and remove path $<n_o, ..., n_k>$ from $frontier$;
  - If $goal(n_k)$
    - return $<n_o, ..., n_k>$;
    - Find a neighbor $n$ of $n_k$
      - add $n$ to $frontier$;

end

• How many bugs?

A. One  B. Two  C. Three  D. Four
Bogus Version of Generic Search Algorithm

- Start at the **start** node(s), NOT at the goal
- Add **all** neighbours of $n_k$ to the frontier, NOT just one
- Add **path(s)** to frontier, NOT just the node(s)
Comparing Searching Algorithms: Will it find a solution? the best one?

Def. : A search algorithm is **complete** if whenever there is at least one solution, the algorithm **is guaranteed to find it** within a finite amount of time.

Def.: A search algorithm is **optimal** if when it finds a solution, it is **the best one**.

Def.: The **time complexity** of a search algorithm is the **worst-case** amount of time it will take to run, expressed in terms of 
- maximum path length \( m \)
- maximum branching factor \( b \).

Def.: The **space complexity** of a search algorithm is the **worst-case** amount of memory that the algorithm will use (i.e., the maximum number of nodes on the frontier), also expressed in terms of \( m \) and \( b \).
The branching factor of a node is the number of arcs going out of the node.

If the branching factor of a node is $b$ and the graph is a tree, how many nodes are $n$ steps away from that node?

- A. $nb$
- B. $b^n$
- C. $n^b$
- D. $n/b$
The branching factor of a node is the number of arcs going out of the node.

If the branching factor of a node is \( b \) and the graph is a tree, how many nodes are \( n \) steps away from that node?

\[ B. \ b^n \]
Depth-First Search, DFS

- explores each path on the frontier until its end (or until a goal is found) before considering any other path.
- the frontier is a last-in-first-out stack
Today’s Lecture

• Recap from Previous lectures
  ➡ Depth first search - analysis
  • Breadth first search
  • Iterative deepening
  • Search with costs
  • Intro to heuristic search (time permitting)
Def. : A search algorithm is **complete** if whenever there is at least one solution, the algorithm is guaranteed to find it within a finite amount of time.

Is DFS **complete**?
Analysis of DFS

Def.: A search algorithm is **complete** if whenever there is at least one solution, the algorithm is guaranteed to find it within a finite amount of time.

Is DFS complete? **No**

- If there are cycles in the graph, DFS may get “stuck” in one of them
- See this in AISpace by loading “Cyclic Graph Examples” or by adding a cycle to “Simple Tree”
  - E.g., click on “Create” tab, create a new edge from N7 to N1, go back to “Solve” and see what happens
Def.: A search algorithm is **optimal** if when it finds a solution, it is the best one (e.g., the shortest)

Is DFS optimal?

- E.g., goal nodes: red boxes
Analysis of DFS

Def.: A search algorithm is **optimal** if when it finds a solution, it is the **best one** (e.g., the shortest)

Is DFS optimal?

- A. Yes
- B. No

• E.g., goal nodes: red boxes
Analysis of DFS

Def.: A search algorithm is **optimal** if when it finds a solution, it is the best one (e.g., the shortest)

Is DFS **optimal**? No

- It can “stumble” on longer solution paths before it gets to shorter ones.
  - E.g., goal nodes: red boxes

- see this in AISpace by loading “Extended Tree Graph” and set N6 as a goal
  - e.g., click on “Create” tab, right-click on N6 and select “set as a goal node”
Analysis of DFS

Def.: The **time complexity** of a search algorithm is the worst-case amount of time it will take to run, expressed in terms of
- maximum path length \( m \)
- maximum forward branching factor \( b \).

• What is DFS’s **time complexity**, in terms of \( m \) and \( b \)?

• E.g., single goal node -> red box
Analysis of DFS

Def.: The time complexity of a search algorithm is the worst-case amount of time it will take to run, expressed in terms of
- maximum path length $m$
- maximum forward branching factor $b$.

• What is DFS’s time complexity, in terms of $m$ and $b$?

A. $O(b^m)$
B. $O(m^b)$
C. $O(bm)$
D. $O(b+m)$

• E.g., single goal node -> red box
• Hint: think about how many nodes are in a search tree $m$ steps away from the start
Analysis of DFS

Def.: The **time complexity** of a search algorithm is the **worst-case** amount of time it will take to run, expressed in terms of
- maximum path length \( m \)
- maximum forward branching factor \( b \).

**O(\( b^m \))**

- What is DFS’s **time complexity**, in terms of \( m \) and \( b \)?

- In the worst case, must examine every node in the tree
  - E.g., single goal node -> red box
Analysis of DFS

Def.: The **space complexity** of a search algorithm is the **worst-case** amount of memory that the algorithm will use (i.e., the maximal number of nodes on the frontier), expressed in terms of
- maximum path length \( m \)
- maximum forward branching factor \( b \).

• What is DFS’s **space complexity**, in terms of \( m \) and \( b \)?

See how this works in...
Analysis of DFS

Def.: The space complexity of a search algorithm is the worst-case amount of memory that the algorithm will use (i.e., the maximal number of nodes on the frontier), expressed in terms of:
- maximum path length \( m \)
- maximum forward branching factor \( b \).

- What is DFS’s space complexity, in terms of \( m \) and \( b \)?

\[ O(bm) \]

- for every node in the path currently explored, DFS maintains a path to its unexplored siblings in the search tree
  - Alternative paths that DFS needs to explore
- The longest possible path is \( m \), with a maximum of \( b-1 \) alternative paths per node along the path
To Summarize

<table>
<thead>
<tr>
<th>Complete</th>
<th>Optimal</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFS</td>
<td>NO</td>
<td>NO</td>
<td>$O(b^m)$</td>
</tr>
</tbody>
</table>
Analysis of DFS: Summary

• **Is DFS complete?** NO
  • Depth-first search isn't guaranteed to halt on graphs with cycles.
  • However, DFS is complete for **finite acyclic graphs**.

• **Is DFS optimal?** NO
  • It can “stumble” on longer solution paths before it gets to shorter ones.

• What is the worst-case **time complexity**, if the maximum path length is $m$ and the maximum branching factor is $b$?
  • $O(b^m)$: must examine every node in the tree.
  • Search is unconstrained by the goal until it happens to stumble on the goal.

• What is the worst-case **space complexity**?
  • $O(bm)$
  • the longest possible path is $m$, and for every node in that path must maintain a fringe of size $b$. 
Analysis of DFS (cont.)

DFS is appropriate when

- Space is restricted
- Many solutions, with long path length

It is a poor method when

- There are cycles in the graph
- There are sparse solutions at shallow depth
Why DFS need to be studied and understood?

• It is simple enough to allow you to learn the basic aspects of searching

• It is the basis for a number of more sophisticated and useful search algorithms (e.g. Iterative Deepening)
Today’s Lecture

• Recap from Previous lectures
• Depth first search - analysis
  ➢ Breadth first search
  • Iterative deepening
  • Search with costs
  • Intro to heuristic search (time permitting)
Breadth-first search (BFS)

- BFS explores all paths of length /on the frontier, before looking at path of length / + 1
**BFS as an instantiation of the Generic Search Algorithm**

**Input:** a graph
- a set of start nodes
- Boolean procedure `goal(n)`
  - testing if n is a goal node

**frontier**: `=<s>: s is a start node>;`

**While** `frontier` is not empty:
- **select** and **remove** path `<n₀,...,nₖ>` from `frontier`;
- **If** `goal(nₖ)`
  - **return** `<n₀,...,nₖ>``;
- **Else**
  - **For every** neighbor n of `nₖ`,
    - **add** `<n₀,...,nₖ, n>` to `frontier`;

**In BFS, the frontier is a first-in-first-out queue**

Let’s see how this works in AIspace in the Search Applet toolbar, set “Search Options -> Search Algorithms” to “Breadth-First Search”.

27
Breadth-first Search: BFS

Example:

- the frontier is \([p_1, p_2, \ldots, p_r]\)
- neighbors of the last node of \(p_1\) are \(\{n_1, \ldots, n_k\}\)

What happens?
- \(p_1\) is selected, and its end node is tested for being a goal. If not
- New \(k\) paths are created attaching each of \(\{n_1, \ldots, n_k\}\) to \(p_1\)
- These follow \(p_r\) at the end of the frontier.
- Thus, the frontier is now \([p_2, \ldots, p_r, (p_1, n_1), \ldots, (p_1, n_k)]\).
- \(p_2\) is selected next.

As for DFS, you can get a much better sense of how BFS works by looking at the Search Applet in AI Space
Def.: A search algorithm is **complete** if whenever there is at least one solution, the algorithm is **guaranteed to find it within a finite amount of time**.

Is BFS **complete**?
Analysis of BFS

Def.: A search algorithm is **complete** if whenever there is at least one solution, the algorithm is guaranteed to find it within a finite amount of time.

Is BFS **complete**?

A. Yes  
B. No  
C. It depends
Is BFS complete? Yes

- If a solution exists at level \( l \), the path to it will be explored before any other path of length \( l + 1 \)
  - impossible to fall into an infinite cycle
- see this in AISpace by loading “Cyclic Graph Examples” or by adding a cycle to “Simple Tree”
Def.: A search algorithm is **optimal** if when it finds a solution, it is the best one

Is BFS optimal?

- E.g., two goal nodes: red boxes
Def.: A search algorithm is **optimal** if when it finds a solution, it is the best one

Is BFS optimal? • E.g., two goal nodes: red boxes

A. Yes
B. No
C. It depends
Def.: A search algorithm is **optimal** if when it finds a solution, it is the best one.

Is BFS **optimal**? Yes

- E.g., two goal nodes: red boxes
- Any goal at level / (e.g. red box N7) will be reached before goals at lower levels
Analysis of BFS

Def.: The time complexity of a search algorithm is the worst-case amount of time it will take to run, expressed in terms of
- maximum path length \( m \)
- maximum forward branching factor \( b \).

• What is BFS’s time complexity, in terms of \( m \) and \( b \)?

A. \( O(b^m) \)
B. \( O(m^b) \)
C. \( O(bm) \)
D. \( O(b+m) \)
Analysis of BFS

Def.: The **time complexity** of a search algorithm is the **worst-case** amount of time it will take to run, expressed in terms of
- maximum path length \( m \)
- maximum forward branching factor \( b \).

\[ O(b^m) \]

- What is BFS’s **time complexity**, in terms of \( m \) and \( b \) ?

- Like DFS, in the worst case BFS must examine every node in the tree
  - E.g., single goal node -> red box
Def.: The **space complexity** of a search algorithm is the **worst case** amount of memory that the algorithm will use (i.e., the maximal number of nodes on the frontier), expressed in terms of
- maximum path length \textit{m}
- maximum forward branching factor \textit{b}.

- What is BFS’s **space complexity**, in terms of \textit{m} and \textit{b} ?

A. \(O(b^m)\)
B. \(O(m^b)\)
C. \(O(bm)\)
D. \(O(b+m)\)
Analysis of BFS

Def.: The space complexity of a search algorithm is the worst case amount of memory that the algorithm will use (i.e., the maximal number of nodes on the frontier), expressed in terms of
- maximum path length \( m \)
- maximum forward branching factor \( b \).

- What is BFS’s space complexity, in terms of \( m \) and \( b \) ?

\[ O(b^m) \]

- BFS must keep paths to all the nodes at level
When to use BFS vs. DFS?

- The search graph has cycles or is infinite
- We need the shortest path to a solution
- There are only solutions at great depth
- There are some solutions at shallow depth and others deeper
- No way the search graph will fit into memory
When to use BFS vs. DFS?

• The search graph has cycles or is infinite
  **BFS**

• We need the shortest path to a solution
  **BFS**

• There are only solutions at great depth
  **DFS**

• There are some solutions at shallow depth and others deeper
  **BFS**

• No way the search graph will fit into memory
  **DFS**
To Summarize

<table>
<thead>
<tr>
<th></th>
<th>Complete</th>
<th>Optimal</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFS</td>
<td>NO</td>
<td>NO</td>
<td>$O(b^m)$</td>
<td>$O(bm)$</td>
</tr>
<tr>
<td>BFS</td>
<td>YES</td>
<td>YES</td>
<td>$O(b^m)$</td>
<td>$O(b^m)$</td>
</tr>
</tbody>
</table>

How can we achieve an acceptable (linear) space complexity while maintaining completeness and optimality?

**Key Idea:** re-compute elements of the frontier rather than saving them.
Iterative Deepening DFS (IDS) in a Nutshell

- Use DFS to look for solutions at depth 1, then 2, then 3, etc
  - For depth D, ignore any paths with longer length
  - Depth-bounded depth-first search

If no goal re-start from scratch and get to depth 2

If no goal re-start from scratch and get to depth 3

If no goal re-start from scratch and get to depth 4
(Time) Complexity of IDS

- That sounds wasteful!
- Let’s analyze the time complexity
- For a solution at depth $m$ with branching factor $b$

<table>
<thead>
<tr>
<th>Depth</th>
<th>Total # of paths at that level</th>
<th>#times created by BFS (or DFS)</th>
<th>#times created by IDS</th>
<th>Total #paths for IDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\cdot$</td>
<td>$\cdot$</td>
<td>$\cdot$</td>
<td>$\cdot$</td>
<td>$\cdot$</td>
</tr>
<tr>
<td>$\cdot$</td>
<td>$\cdot$</td>
<td>$\cdot$</td>
<td>$\cdot$</td>
<td>$\cdot$</td>
</tr>
<tr>
<td>$\cdot$</td>
<td>$\cdot$</td>
<td>$\cdot$</td>
<td>$\cdot$</td>
<td>$\cdot$</td>
</tr>
<tr>
<td>$m-1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$m$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
(Time) Complexity of IDS

- That sounds wasteful!
- Let’s analyze the time complexity
- For a solution at depth $m$ with branching factor $b$

<table>
<thead>
<tr>
<th>Depth</th>
<th>Total # of paths at that level</th>
<th>#times created by BFS (or DFS)</th>
<th>#times created by IDS</th>
<th>Total #paths for IDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$b$</td>
<td>1</td>
<td>$m$</td>
<td>$mb$</td>
</tr>
<tr>
<td>2</td>
<td>$b^2$</td>
<td>1</td>
<td>$m-1$</td>
<td>$(m-1) b^2$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$m-1$</td>
<td>$b^{m-1}$</td>
<td>1</td>
<td>2</td>
<td>$2 b^{m-1}$</td>
</tr>
<tr>
<td>$m$</td>
<td>$b^m$</td>
<td>1</td>
<td>1</td>
<td>$b^m$</td>
</tr>
</tbody>
</table>
(Time) Complexity of IDS

Solution at depth \( m \), branching factor \( b \)

Total # of paths generated:

\[
b^m + 2b^{m-1} + 3b^{m-2} + \ldots + mb
\]

\[
= b^m (1 + b^{-1} + b^{-2} + \ldots + m b^{-m})
\]

\[
= b^m \left( \sum_{i=1}^{m} ib^{1-i} \right) \leq b^m \left( \sum_{i=1}^{\infty} ib^{1-i} \right) = b^m \left( \frac{b}{b-1} \right)^2 \in O(b^m)
\]

- For \( b = 10 \), \( m = 5 \). BSF 111,111 and ID = 123,456 (only 11% more nodes)

- The larger \( b \) the better, but even with \( b = 2 \) the search ID will take only 2 times as much as BFS
Further Analysis of Iterative Deepening DFS (IDS)

- Space complexity

<table>
<thead>
<tr>
<th>Depth</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>If no goal re-start from scratch and get to depth 2</td>
</tr>
<tr>
<td>3</td>
<td>If no goal re-start from scratch and get to depth 3</td>
</tr>
</tbody>
</table>

- Complete?

- Optimal?
### Summary of Uninformed Search

<table>
<thead>
<tr>
<th></th>
<th>Complete</th>
<th>Optimal</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DFS</strong></td>
<td>N</td>
<td>N</td>
<td>$O(b^m)$</td>
<td>$O(mb)$</td>
</tr>
<tr>
<td><strong>BFS</strong></td>
<td>Y</td>
<td>Y</td>
<td>$O(b^m)$</td>
<td>$O(b^m)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(shortest)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>IDS</strong></td>
<td>Y</td>
<td>Y</td>
<td>$O(b^m)$</td>
<td>$O(mb)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(shortest)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>LCFS</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Today’s Lecture

• Recap from Previous lectures
• Depth first search - analysis
• Breadth first search
• Iterative deepening

Search with costs

• Intro to heuristic search (time permitting)
Search with Costs

Sometimes there are costs associated with arcs.

Def.: The cost of a path is the sum of the costs of its arcs

\[
\text{cost}(\langle n_0, \ldots, n_k \rangle) = \sum_{i=1}^{k} \text{cost}(\langle n_{i-1}, n_i \rangle)
\]
Example: Traveling in Romania
**Search with Costs**

Sometimes there are costs associated with arcs.

**Def.:** The cost of a path is the sum of the costs of its arcs

\[
\text{cost}\left(\langle n_0, \ldots, n_k \rangle \right) = \sum_{i=1}^{k} \text{cost}\left(\langle n_{i-1}, n_i \rangle \right)
\]

In this setting we often don't just want to find any solution

- we usually want to find the solution that minimizes cost

**Def.:** A search algorithm is optimal if when it finds a solution, it is the best one: it has the lowest path cost
Lowest-Cost-First Search (LCFS)

- **Lowest-cost-first search** finds the path with the lowest cost to a goal node.
- At each stage, it **selects** the path with the lowest cost on the frontier.
- The **frontier** is implemented as a priority queue ordered by path cost.

Let's see how this works in Alspace: in the Search Applet toolbar
- select the “Vancouver Neighborhood Graph” problem
- set “Search Options -> Search Algorithms” to “Lowest-Cost-First ”.
- select “Show Edge Costs” under “View”
- Create a new arc from UBC to SP with cost 20 and run LCFS
Example of one step for LCFS:

- Let’s use \((p_i, c_i)\) to indicate a path \(p_i\) and its cost
- the frontier is \([(p_2, 5), (p_3, 7), (p_1, 11)]\)
- \(p_2\) is the lowest-cost node in the frontier:
- Paths obtained by adding neighbor nodes to the end node of \(p_2\) are: \(\{(p_9, 10), (p_{10}, 15)\}\)
- What happens?
  - \(p_2\) is selected, and tested for being a goal: \textit{false}
  - Neighbor paths of \(p_2\) are inserted into the frontier, which is then sorted by cost
  - Thus, the frontier is now \([(p_3, 7), (p_9, 10), (p_1, 11), (p_{10}, 15)]\).
  - \((p_3, 7)\) is selected next.
When arc costs are equal LCFS is equivalent to:

A. DFS
B. BFS
C. IDS
D. None of the Above
• When arc costs are equal LCFS is equivalent to.. 

A. DFS  

B. BFS  

C. IDS  

D. None of the Above
Analysis of Lowest-Cost Search (1)

• Is LCFS complete?
  • not in general: for instance, a cycle with zero or negative arc costs could be followed forever.

see how this works in Alspace:
  • e.g., add arc with cost -20 to the simple search graph from N4 to S in Simple Search Tree

  • yes, as long as arc costs are strictly positive, greater than a given constant $\varepsilon$*

*If costs along an infinite path can become infinitively small, their sum can be finite (e.g. series $\sum_{i=1}^{\infty} \frac{1}{2^i} < 1$)
Analysis of Lowest-Cost Search (1)

• Is LCFS complete?
  • not in general: for instance, a cycle with zero or negative arc costs could be followed forever.

  see how this works in Alspace:
  • e.g, add arc with cost -20 to the simple search graph from N4 to S

  • yes, as long as arc costs are strictly positive yes, greater than a given constant \( \varepsilon \) *

• Is LCFS optimal?

*If costs along an infinite path can become infinitively small, their sum can be finite (e.g. series \( \sum_{i=1}^{\infty} \frac{1}{2^i} < 1 \))
Analysis of Lowest-Cost Search (1)

• Is LCFS complete?
  • not in general: for instance, a cycle with zero or negative arc costs could be followed forever.
  • yes, as long as arc costs are strictly positive yes, greater than a given constant $\varepsilon$ *

see how this works in Alspace:
  • e.g, add arc with cost -20 to the simple search graph from N4 to S

• Is LCFS optimal?
  • Not in general.
  • Arc costs could be negative: a path that initially looks high-cost could end up getting a ``refund''.
  • However, LCFS is optimal if arc costs are guaranteed to be $\geq 0$

*If costs along an infinite path can become infinitively small, their sum can be finite (e.g. series $\sum_{i=1}^{\infty} \frac{1}{2^i} < 1$)
Learning Goals for today’s class

• Select the most appropriate search algorithms for specific problems.
  • Depth-First Search vs. Breadth-First Search vs. Iterative Deepening vs LCFS

• Define/read/write/trace/debug different search algorithms

• Define search heuristic and provide examples
• Heuristic Search and A*: Ch 3.6, 3.6.1