Lecture 6
Analysis of A*,
Search Refinements
(Ch 3.7.1 - 3.7.4)
Lecture Overview

Recap of previous lecture
- Analysis of A*
- Branch-and-Bound
- Cycle checking, multiple path pruning
How to Make Search More Informed?

Def.: A search heuristic \( h(n) \) is an estimate of the cost of the optimal (cheapest) path from node \( n \) to a goal node.

- \( h \) can be extended to paths: \( h(\langle n_0, ..., n_k \rangle) = h(n_k) \)
- \( h(n) \) should leverage readily obtainable information (easy to compute) about a node.
Always choose the path on the frontier with the smallest $h$ value.

- BestFS treats the frontier as a priority queue ordered by $h$.
- Can get to the goal pretty fast if it has a good $h$ but...

It is not complete, nor optimal.

still has time and space worst-case complexity of $O(b^m)$
## Learning Goal for Search

Apply basic properties of search algorithms:

- completeness, optimality, time and space complexity

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complete</th>
<th>Optimal</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFS</td>
<td>N (Y if no cycles)</td>
<td>N</td>
<td>$O(b^m)$</td>
<td>$O(mb)$</td>
</tr>
<tr>
<td>BFS</td>
<td>Y</td>
<td>Y</td>
<td>$O(b^m)$</td>
<td>$O(b^m)$</td>
</tr>
<tr>
<td>IDS</td>
<td>Y</td>
<td>Y</td>
<td>$O(b^m)$</td>
<td>$O(mb)$</td>
</tr>
<tr>
<td>LCFS (when arc costs available)</td>
<td>Y Costs &gt; $\epsilon$</td>
<td>Y Costs &gt;=0</td>
<td>$O(b^m)$</td>
<td>$O(b^m)$</td>
</tr>
<tr>
<td>Best First (when $h$ available)</td>
<td>N</td>
<td>N</td>
<td>$O(b^m)$</td>
<td>$O(b^m)$</td>
</tr>
</tbody>
</table>

Legend:
- **uninformed**
- **Uninformed but using arc cost**
- **Informed (goal directed)**
A* Search

- A* search takes into account both
  - the cost of the path to a node \( c(p) \)
  - the heuristic value of that path \( h(p) \).

- Let \( f(p) = c(p) + h(p) \).
  - \( f(p) \) is an estimate of the cost of a path from the start to a goal via \( p \).

A* always chooses the path on the frontier with the lowest estimated distance from the start to a goal node constrained to go via that path.

Compare A* and LCFS on the Vancouver graph.
Computing f-values

F-value of ubc → kd → jb?
Optimality of A*

A* is complete (finds a solution, if one exists) and optimal (finds the optimal path to a goal) if

- arc costs are \( \varepsilon > 0 \)
- \( h(n) \) is admissible

The book also mentions explicitly that the branching factor \( b \) has to be finite, which we have been assuming by default (without this condition even BFS would not be complete)
Def.:
Let \( c(n) \) denote the cost of the optimal path from node \( n \) to any goal node. A search heuristic \( h(n) \) is called **admissible** if \( h(n) \leq c(n) \) for all nodes \( n \), i.e. if for all nodes it is an **underestimate** of the cost to any goal.

- **Example**: is the straight-line distance admissible?
  - **YES**
  - The shortest distance between two points is a line.
Admissibility of a heuristic

Def.: Let \( c(n) \) denote the cost of the optimal path from node \( n \) to any goal node. A search heuristic \( h(n) \) is called **admissible** if \( h(n) \leq c(n) \) for all nodes \( n \), i.e. if for all nodes it is an underestimate of the cost to any goal.

**Example:**
the goal is Urzieni (red box), but all we know is the straight-line distances to Bucharest (green box)

- Possible \( h(n) = \text{sld}(n, \text{Bucharest}) + \text{cost(\text{Bucharest}, Urzineni)} \)
- Admissible? **NO**
  - Actual cost of going from **Vastul** to **Urzineni** is shorter than this estimate
Lecture Overview

• Recap of previous lecture
• Analysis of A*
• Branch-and-Bound
• Cycle checking, multiple path pruning
Optimality of A*

To prove optimality, we will start by proving the following:

Let
- \( p \) be a subpath of an optimal path \( p^* \).
- \( c(\cdot) \) be the actual path cost of a path (> 0)
- \( f(\cdot) \) be the f-values of a path
- \( h() \) be the heuristic value of a path (≥ 0)

We can show that if \( h() \) is admissible, \( f(p) \leq f(p^*) \)
Why A* is optimal

- Let $p^*$ be the optimal solution path, with cost $c^*$.
- Let $p'$ be a suboptimal solution path. That is $c(p') > c^*$.

We are going to show that any sub-path $p''$ of $p^*$ on the frontier will be expanded before $p'$.

Therefore, A* will find $p^*$ before $p'$.
Why A* is optimal

- Let $p^*$ be the optimal solution path, with cost $c^*$.
- Let $p'$ be a suboptimal solution path. That is $c(p') > c^*$. $p^*$
- Let $p''$ be a sub-path of $p^*$ on the frontier.

we know that because at a goal node \[ f(\text{goal}) = c(\text{goal}) \]

and because $h$ is admissible (see previous proof)

thus \[ f(p'') < f(p') \]

Any sup-path of the optimal solution path will be expanded before $p'$
Run A* on this example to see how A* starts off going down the suboptimal path (through N5) but then recovers and never expands it, because there are always subpaths of the optimal path through N2 on the frontier with lower f value.
Why is A* complete

It does not get caught in cycles

- Let \( f^* \) be the cost of the (an) optimal solution path \( p^* \)
  (unknown but finite if there exists a solution)
- Each sub-path \( p \) of \( p^* \) will be expanded before \( p^* \)
  - See previous proof
- With positive (and \( \epsilon \)) arc costs, the cost of any other path \( p \) on the frontier
  would eventually exceed \( f^* \)
  - This happens at depth no greater than \( \frac{f^*}{c_{\text{min}}} \), where \( c_{\text{min}} \) is the
  minimal arc cost in the search graph

See how it works on the “misleading heuristic” problem in AI space:
A* does not get caught into the cycle because f(n) of sub paths in the cycle eventually (at depth <= 55.4/6.9) exceed the cost of the optimal solution 55.4 (N0->N6->N7->N8)
If fact, we can say something even stronger about A* (when it is admissible)

A* is optimally efficient.

It finds the goal with the minimum # of expansions compared to other optimal algorithms (given the same heuristic function)

This is because any algorithm that does not expand every node with $f(n) < f^*$ risks missing the optimal solution.
Time Space Complexity of $A^*$

- **Time complexity** is $O(b^m)$
  - the heuristic could be completely uninformative and the edge costs could all be the same, meaning that $A^*$ does the same thing as BFS

- **Space complexity** is $O(b^m)$ like BFS, $A^*$ maintains a frontier which grows with the size of the tree
**Effect of Search Heuristic**

- A search heuristic that is a better approximation on the actual cost reduces the number of nodes expanded by A*.

Example: 8puzzle:

1. tiles can move anywhere
   - \( h_1 \) : number of tiles that are out of place
2. tiles can move to any adjacent square
   - \( h_2 \) : sum of number of squares that separate each tile from its correct position

average number of paths expanded: \( d = \text{depth of the solution} \)

\[ \begin{align*}
  d=12 & \quad \text{IDS} = 3,644,035 \text{ paths} \\
         & \quad A^*(h_1) = 227 \text{ paths} \\
         & \quad A^*(h_2) = 73 \text{ paths} \\
  d=24 & \quad \text{IDS} = \text{too many paths} \\
         & \quad A^*(h_1) = 39,135 \text{ paths} \\
         & \quad A^*(h_2) = 1,641 \text{ paths}
\]
## Learning Goal for Search

Apply basic properties of search algorithms:

- completeness, optimality, time and space complexity

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complete</th>
<th>Optimal</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFS</td>
<td>N</td>
<td>Y</td>
<td>$O(b^m)$</td>
<td>$O(mb)$</td>
</tr>
<tr>
<td>(Y if no cycles)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFS</td>
<td>Y</td>
<td>Y</td>
<td>$O(b^m)$</td>
<td>$O(b^m)$</td>
</tr>
<tr>
<td>IDS</td>
<td>Y</td>
<td>Y</td>
<td>$O(b^m)$</td>
<td>$O(mb)$</td>
</tr>
<tr>
<td>LCFS</td>
<td>Y</td>
<td>Y</td>
<td>$O(b^m)$</td>
<td>$O(b^m)$</td>
</tr>
<tr>
<td>(when arc costs available)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Best First</td>
<td>N</td>
<td>N</td>
<td>$O(b^m)$</td>
<td>$O(b^m)$</td>
</tr>
<tr>
<td>(when $h$ available)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(when arc costs &gt; $\varepsilon$ and $h$ admissible)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Learning Goal for Search

Apply basic properties of search algorithms:
- completeness, optimality, time and space complexity

<table>
<thead>
<tr>
<th>Search Algorithm</th>
<th>Complete</th>
<th>Optimal</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFS</td>
<td>N (Y if no cycles)</td>
<td>N</td>
<td>$O(b^m)$</td>
<td>$O(mb)$</td>
</tr>
<tr>
<td>BFS</td>
<td>Y</td>
<td>Y</td>
<td>$O(b^m)$</td>
<td>$O(b^m)$</td>
</tr>
<tr>
<td>IDS</td>
<td>Y</td>
<td>Y</td>
<td>$O(b^m)$</td>
<td>$O(mb)$</td>
</tr>
<tr>
<td>LCFS</td>
<td>Y</td>
<td>Y</td>
<td>$O(b^m)$</td>
<td>$O(b^m)$</td>
</tr>
<tr>
<td>(when arc costs available)</td>
<td>Costs $&gt; \varepsilon$</td>
<td>Costs $\geq 0$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Best First</td>
<td>N</td>
<td>N</td>
<td>$O(b^m)$</td>
<td>$O(b^m)$</td>
</tr>
<tr>
<td>(when $h$ available)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A*</td>
<td>Y</td>
<td>Y</td>
<td>$O(b^m)$</td>
<td>$O(b^m)$</td>
</tr>
<tr>
<td>(when arc costs $&gt; \varepsilon$ and $h$ admissible)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Lecture Overview

- Recap of previous lecture
- Analysis of A*

Branch-and-Bound

- Cycle checking, multiple path pruning
Branch-and-Bound Search

• What does allow A* to do better than the other search algorithms we have seen?

• What is the biggest problem with A*?

• Possible Solution:
Branch-and-Bound Search

One way to combine DFS with heuristic guidance

- Follows exactly the same search path as DFS
  - But to ensure optimality, it does not stop at the first solution found

- It continues, after recording upper bound on solution cost
  - upper bound: \( UB = \) cost of the best solution found so far
  - Initialized to \( \infty \) or any overestimate of optimal solution cost

- When a path \( p \) is selected for expansion:
  - Compute lower bound \( LB(p) = f(p) = \text{cost}(p) + h(p) \)
    - If \( LB(p) \geq UB \), remove \( p \) from frontier without expanding it (pruning)
    - Else expand \( p \), adding all of its neighbors to the frontier
• Arc cost = 1
• \( h(n) = 0 \) for every \( n \)
• \( UB = \infty \)

Before expanding a path \( p \), check its \( f \) value \( f(p) \):
Expand only if \( f(p) < UB \)

\[ f(p) = 1 \]
\[ f(p) = 2 \]

Solution!
\[ UB = ? \]
• Arc cost = 1
• \( h(n) = 0 \) for every \( n \)
• \( UB = 5 \)

Before expanding a path \( p \), check its \( f \) value \( f(p) \):
Expand only if \( f(p) < UB \)
• Arc cost = 1
• $h(n) = 0$ for every $n$
• $UB = \infty$

Before expanding a path $p$, check its $f$ value $f(p)$:
Expand only if $f(p) < UB$
Before expanding a path $p$, check its f value $f(p)$:
Expand only if $f(p) < \text{UB}$

- Arc cost = 1
- $h(n) = 0$ for every $n$
- UB = 3

Prune!
Before expanding a path \( p \), check its \( f \) value \( f(p) \):
Expand only if \( f(p) < UB \)

- Arc cost = 1
- \( h(n) = 0 \) for every \( n \)
- UB = 3

Before expanding a path, check its \( f \) value. If the \( f \) value is greater than the UB, prune the path.
<table>
<thead>
<tr>
<th>Search Method</th>
<th>Complete</th>
<th>Optimal</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFS</td>
<td>N</td>
<td>N</td>
<td>$O(b^m)$</td>
<td>$O(mb)$</td>
</tr>
<tr>
<td>BFS</td>
<td>Y</td>
<td>Y</td>
<td>$O(b^m)$</td>
<td>$O(b^m)$</td>
</tr>
<tr>
<td>IDS</td>
<td>Y</td>
<td>Y</td>
<td>$O(b^m)$</td>
<td>$O(mb)$</td>
</tr>
<tr>
<td>LCFS (when arc costs available)</td>
<td>Y Costs $&gt; \varepsilon$</td>
<td>Y Costs $\geq 0$</td>
<td>$O(b^m)$</td>
<td>$O(b^m)$</td>
</tr>
<tr>
<td>Best First (when $h$ available)</td>
<td>N</td>
<td>N</td>
<td>$O(b^m)$</td>
<td>$O(b^m)$</td>
</tr>
<tr>
<td>A* (when arc costs $&gt; \varepsilon$ and $h$ admissible)</td>
<td>Y</td>
<td>Y</td>
<td>$O(b^m)$</td>
<td>Optimally Efficient</td>
</tr>
<tr>
<td>Branch-and-Bound</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Lecture Overview

• Recap of previous lecture
• Analysis of A*
• Branch-and-Bound
→ Cycle checking, multiple path pruning
Clarification: state space graph vs search tree

State space graph represents the states in a search problem, and how they are connected by the available operators.

Search Tree: Shows how the search space is traversed by a given search algorithm: explicitly “unfolds” the paths that are expanded.

If there are no cycles or multiple paths, the two look the same.
Clarification: state space graph vs search tree

State space graph

If there are cycles or multiple paths, the two look very different
Size of state space vs. search tree

If there are cycles or multiple paths, the two look very different

- E.g. state space with \( d \) states and 2 actions from each state to next
  - With \( d + 1 \) states, search tree has depth \( d \)
  - \( 2^d \) possible paths through the search space => exponentially larger search tree!
- With cycles or multiple parents, the search tree can be exponential in the state space
Cycle Checking and Multiple Path Pruning

- **Cycle checking**: good when we want to avoid infinite loops, but also want to find more than one solution, if they exist.

- **Multiple path pruning**: good when we only care about finding one solution.
  - Subsumes cycle checking.
Cycle Checking

Good when we want to avoid infinite loops, but also want to find more than one solution, if they exist.

- You can prune a path that ends in a node already on the path.
- This pruning cannot remove an optimal solution => cycle check

- What is the computational cost of cycle checking?
Cycle Checking

- See how DFS and BFS behave on Cyclic Graph Example in Aispace, when Search Options-> Pruning -> Loop detection is selected

- Set N1 to be a normal node so that there is only one start node.

- Check, for each algorithm, what happens during the first expansion from node 3 to node 2
Depth First Search

Since DFS looks at one path at a time, when a node is encountered for the second time (e.g. Node 2 while expanding N0, N2, N5, N3) it is guaranteed to be part of a cycle.
Breadth First Search

Since BFS keeps multiple subpaths going, when a node is encountered for the second time, it could be as part of expanding a different path (e.g. Node 2 while expanding N0-> N3). Not necessarily a cycle.
Breadth First Search

The cycle for BFS happens when N2 is encountered for the second time while expanding the path N0->N2->N5->N3.
Cycle Checking

• You can prune a path that ends in a node already on the path.
• This pruning cannot remove an optimal solution => cycle check

• What is the computational cost of cycle checking?
• Using depth-first methods, with the graph explicitly stored, this can be done in as low as constant time (i.e., not dependent on path length)
  - Only one path being explored at a time
• Other methods: cost is linear in path length
  - Must check each node in the current path, since a node could have been expanded as part of a different path on the frontier
• If we only want one path to the solution
• Can prune path to a node $n$ that has already been reached via a previous path
  - Subsumes cycle check
Multiple Path Pruning

- Can see how it works by
  - Running BFS on the Cyclic Graph Example in AISpace
  - See how it handles the multiple paths from N0 to N2
  - You can erase start node N1 to simplify things
Example with BSF

Handling the multiple paths from N0 to N2
(<N0, N2>, <N0, N3, N2>)

Path Node 0 -> Node 3 expanded.
Neighbours of Node 3: Node 2, Node 4,
Multiple-Path Pruning & Optimal Solutions

Problem: what if a subsequent path $p_2$ to $n$ is better (shorter or less costly) than the first path $p_1$ to a node $n$, and we want an optimal solution?

- Can remove all paths from the frontier that use the longer path: these cannot be optimal.
- Can change the initial segment of the paths on the frontier to use the shorter

or

- Could prove that this problem cannot happen for a given algorithm, because the first path found to any node is the best (shortest)
Learning Goals for Search (up to today)

• Apply basic properties of search algorithms:
  • completeness, optimality
  • time and space complexity

• Select the most appropriate search algorithms for specific problems.
  • Depth-First Search vs. Breadth-First Search vs. Iterative Deepening vs. Least-Cost-First Search, Best First Search, A*, Branch and Bound

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

• Construct heuristic functions and discuss their admissibility for specific search problems

• Prove optimality of A*

• Pruning cycles and multiple paths
To Do for Next Class

• Read
  • Chp 4.1-4.2 (Intro to Constraint Satisfaction Problems)

• Do Practice Exercise 3E

• Keep working on assignment-1!
Lecture Overview

• Recap of previous lecture
• Analysis of A*
• Branch-and-Bound
• Cycle checking, multiple path pruning

Dynamic programming
Dynamic Programming

- Idea: for statically stored graphs, build a table of dist(n):
  - The actual distance of the shortest path from node n to a goal g
  - This is the perfect ..........

- \( \text{dist}(g) = \)
- \( \text{dist}(z) = \)
- \( \text{dist}(c) = \)
- \( \text{dist}(b) = \)
Dynamic Programming

Dist(n) can be built backwards from the goal:

\[
\text{dist}(n) = \begin{cases} 
  0 & \text{if } is\_goal(n), \\
  \min_{(n,m) \in A} (\text{cost}(n,m) + \text{dist}(m)) & \text{otherwise} 
\end{cases}
\]

over all the neighbors m of n
Dynamic Programming

Dist(n) can be built **backwards** from the goal:

\[
dist(n) = \begin{cases} 
0 & \text{if } \text{is\_goal}(n), \\
\min_{(n,m) \in A} (\text{cost}(n,m) + dist(m)) & \text{otherwise} 
\end{cases}
\]

over all the neighbors m of n

Dist(KD) =
Dynamic Programming

Dist(n) can be built **backwards** from the goal:

\[
dist(n) = \begin{cases} 
0 & \text{if is_goal(n),} \\
\min_{(n,m) \in A} \left( \text{cost}(n,m) + dist(m) \right) & \text{otherwise}
\end{cases}
\]

over all the neighbors m of n

\[
\text{Dist(KD)} = \min \left[ (4+8), (3+8) \right] = 11
\]

Requires **explicit goal nodes** (the previous methods only assumed a function that recognizes goal nodes);
Dynamic Programming

- Idea: for **statically stored graphs**, build a table of dist(n):
  - The **actual distance** of the shortest path from node n to a goal g
  - This is the perfect h

- How could we implement that?
  - Run LCFS with multiple path pruning in the **backwards** graph (arcs reversed), starting from the goal
  - Can do this in Aispace by using *invert graph*, in *create* mode
LCSF on Inverted Graph (sample steps)
Dynamic Programming

- Idea: for statically stored graphs, build a table of dist(n):
  - The actual distance of the shortest path from node n to a goal g
  - This is the perfect h

- How could we implement that?
  - Run LCFS with multiple path pruning in the backwards graph (arcs reversed), starting from the goal
  - Can do this in Aispace by using invert graph, in create mode

- When it’s time to act (forward): for each node n always pick neighbor m that minimizes
  \[
  \min_{(n,m) \in A} \left( \text{cost}(n, m) + \text{dist}(m) \right)
  \]

- Problems?
Dynamic Programming

• Idea: for statically stored graphs, build a table of dist(n):
  • The actual distance of the shortest path from node n to a goal g
  • This is the perfect h

• How could we implement that?
  • Run LCFS with multiple path pruning in the backwards graph (arcs reversed), starting from the goal
  • Can do this in Aispace by using invert graph, in create mode

• When it’s time to act (forward): for each node n always pick neighbor m that minimizes

\[
\min_{(n,m) \in A} (\text{cost}(n,m) + \text{dist}(m))
\]

• Problems?
  • Needs space to explicitly store the full search graph
  • The dist function needs to be recomputed for each goal: method not suitable when goals vary often
Learning Goals for Search (up to today)

- Apply basic properties of search algorithms:
  - completeness, optimality
  - time and space complexity
- Select the most appropriate search algorithms for specific problems.
  - Depth-First Search vs. Breadth-First Search vs. Iterative Deepening vs. Least-Cost-First Search, Best First Search, A*, Branch and Bound
- Define/read/write/trace/debug different search algorithms
- Construct heuristic functions and discuss their admissibility for specific search problems
- Prove optimality of A*
- Pruning cycles and multiple paths
To Do for Next Class

• Read
  • Chp 4.1-4.2 (Intro to Constraint Satisfaction Problems)

• Do Practice Exercise 3E

• Keep working on assignment-1!