Searching

Often we are not given an algorithm to solve a problem, but only a specification of what is a solution — we have to search for a solution.

A typical problem is when the agent is in one state, it has a set of deterministic actions it can carry out, and wants to get to a goal state.

Many AI problems can be abstracted into the problem of finding a path in a directed graph.

Often there is more than one way to represent a problem as a graph.

Directed Graphs

A graph consists of a set $N$ of nodes and a set $A$ of ordered pairs of nodes, called arcs.

Node $n_2$ is a neighbor of $n_1$ if there is an arc from $n_1$ to $n_2$. That is, if $\langle n_1, n_2 \rangle \in A$.

A path is a sequence of nodes $\langle n_0, n_1, \ldots, n_k \rangle$ such that $\langle n_{i-1}, n_i \rangle \in A$.

Given a set of start nodes and goal nodes, a solution is a path from a start node to a goal node.

Often there is a cost associated with arcs and the cost of a path is the sum of the costs of the arcs in the path.

Example Problem for Delivery Robot

The robot wants to get from outside room 103 to the inside of room 123.

Graph for the Delivery Robot

Topological Map of DC/MC

cost = distance travelled

cost = number of doors + distance (1)
Partial Search Space for a Video Game

Grid game: collect coins \( C_1, C_2, C_3, C_4 \), don’t run out of fuel, and end up at location \((1, 1)\):

Problem Solving by Graph Searching

- Generic search algorithm: given a graph, start nodes, and goal nodes, incrementally explore paths from the start nodes.
- Maintain a frontier of paths from the start node that have been explored.
- As search proceeds, the frontier expands into the unexplored nodes until a goal node is encountered.
- The way in which the frontier is expanded defines the search strategy.

Graph Search Algorithm

**Input:** a graph, a set of start nodes, Boolean procedure goal\((n)\) that tests if \(n\) is a goal node.

\[
\text{frontier} := \{(s) : s \text{ is a start node}\};
\]

**while** frontier is not empty:

**select** and **remove** path \((n_0, \ldots, n_k)\) from frontier;

if \(\text{goal}(n_k)\)

**return** \((n_0, \ldots, n_k)\);

**for** every neighbor \(n\) of \(n_k\)

add \((n_0, \ldots, n_k, n)\) to frontier;

**end while**
### Types of Search

- Uninformed (blind)
- Heuristic
- More sophisticated “hacks”

### Depth-first Search

- **Depth-first search** treats the frontier as a stack.
- It always selects one of the last elements added to the frontier.
- If the list of paths on the frontier is $[p_1, p_2, \ldots]$:
  - $p_1$ is selected. Paths that extend $p_1$ are added to the front of the stack (in front of $p_2$).
  - $p_2$ is only selected when all paths from $p_1$ have been explored.

### Illustrative Graph — Depth-first Search

![Illustrative Graph — Depth-first Search](image)

### Complexity of Depth-first Search

- Depth-first search isn’t guaranteed to halt on infinite graphs or on graphs with cycles.
- The space complexity is linear in the size of the path being explored.
- Search is unconstrained by the goal until it happens to stumble on the goal (uninformed or blind).
- What is the worst-case time complexity of depth-first search?

### Graph Search Algorithm - with Cycle Check

**Input:** a graph, a set of start nodes, Boolean procedure $\text{goal}(n)$ that tests if $n$ is a goal node.

1. $\text{frontier} := \{ (s) : s \text{ is a start node} \}$;
2. while $\text{frontier}$ is not empty:
   1. select and remove path $\langle n_0, \ldots, n_k \rangle$ from $\text{frontier}$;
   2. if $\text{goal}(n_k)$
      - return $\langle n_0, \ldots, n_k \rangle$;
   3. for every neighbor $n$ of $n_k$
      - if $n \notin \langle n_0, \ldots, n_k \rangle$
         - add $\langle n_0, \ldots, n_k, n \rangle$ to $\text{frontier}$;
3. end while

### Breadth-first Search

- **Breadth-first search** treats the frontier as a queue.
- It always selects one of the earliest elements added to the frontier.
- If the list of paths on the frontier is $[p_1, p_2, \ldots, p_r]$:
  - $p_1$ is selected. Its neighbors are added to the end of the queue, after $p_r$.
  - $p_2$ is selected next.
Illustrative Graph — Breadth-first Search

Complexity of Breadth-first Search

- The **branching factor** of a node is the number of its neighbors.
- If the branching factor for all nodes is finite, breadth-first search is guaranteed to find a solution if one exists. It is guaranteed to find the path with fewest arcs.
- Time complexity is exponential in the path length: $b^n$, where $b$ is branching factor, $n$ is path length.
- The space complexity is exponential in path length: $b^n$.
- Search is unconstrained by the goal.
- Is it affected by cycles?

Graph Search Algorithm - with Multiple Path Pruning

```
Input: a graph,
      a set of start nodes,
      Boolean procedure goal(n) that tests if n is a goal node.

frontier := \{⟨s⟩ : s is a start node\};
has_path := {};

while frontier is not empty:
    select and remove path ⟨n_0, ..., n_k⟩ from frontier;
    if n_k ∉ has_path:
        add n_k to has_path;
        if goal(n_k)
            return ⟨n_0, ..., n_k⟩;
        for every neighbor n of n_k
            add ⟨n_0, ..., n_k, n⟩ to frontier;
    end while
```

Lowest-cost-first Search

- Sometimes there are **costs** associated with arcs. The cost of a path is the sum of the costs of its arcs.

$$\text{cost}(⟨n_0, ..., n_k⟩) = \sum_{i=1}^{k} |⟨n_{i-1}, n_i⟩|$$

- At each stage, lowest-cost-first search selects a path on the frontier with lowest cost.
- The frontier is a priority queue ordered by path cost.
- It finds a least-cost path to a goal node.
- When arc costs are equal $\Rightarrow$ breadth-first search.
- Uniformed/Blind search (in that it does not take the goal into account)
- Complexity?

Heuristic Search

- **Idea:** don’t ignore the goal when selecting paths.
- Often there is extra knowledge that can be used to guide the search: **heuristics**.
- $h(n)$ is an estimate of the cost of the shortest path from node $n$ to a goal node.
- $h(n)$ uses only readily obtainable information (that is easy to compute) about a node.
- $h$ can be extended to paths: $h(⟨n_0, ..., n_k⟩) = h(n_k)$.
- $h(n)$ is an underestimate if there is no path from $n$ to a goal that has path length less than $h(n)$.

Example Heuristic Functions

- If the nodes are points on a Euclidean plane and the cost is the distance, we can use the straight-line distance from $n$ to the closest goal as the value of $h(n)$.
- If the nodes are locations and cost is time, we can use the distance to a goal divided by the maximum speed.
- What about Chess?
Best-first Search

Idea: select the path whose end is closest to a goal according to the heuristic function.

- Best-first search selects a path on the frontier with minimal $h$-value.
- It treats the frontier as a priority queue ordered by $h$.

Illustrative Example — Best First Search

```
2  C  G  A  S
4
4  3  2  1  0
2  1  1
```

best first: S-A-C-G (not optimal)

Graph Search Algorithm - with Multiple Path Pruning

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

```
frontier := \{ (s) : s is a start node \};
has_path := \{ \};
```

while frontier is not empty:

- select and remove path \( <n_0, \ldots, n_k> \) from frontier;
- if \( n_k \notin has_path \):
  - add \( n_k \) to has_path;
  - if goal($n_k$)
    - return \( <n_0, \ldots, n_k> \);
  - for every neighbor \( n \) of \( n_k \)
    - add \( <n_0, \ldots, n_k, n> \) to frontier;

end while

Heuristic Depth-first Search

Idea: Do a depth-first search, but add paths to the stack ordered according to $h$

- Locally does a best-first search, but aggressively pursues the best looking path (even if it ends up being worse than one higher up).
- Suffers from the same problems as depth-first search
- Is often used in practice

Illustrative Graph — Heuristic Search

- cost of an arc is its length
- heuristic: euclidean distance
- red nodes all look better than green nodes

Graph Search Algorithm - with Multiple Path Pruning

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

```
frontier := \{ (s) : s is a start node \};
has_path := \{ \};
```

while frontier is not empty:

- select and remove path \( <n_0, \ldots, n_k> \) from frontier;
- if \( n_k \notin has_path \):
  - add \( n_k \) to has_path;
  - if goal($n_k$)
    - return \( <n_0, \ldots, n_k> \);
  - for every neighbor \( n \) of \( n_k \)
    - add \( <n_0, \ldots, n_k, n> \) to frontier;

end while
A* Search

- A* search uses both path cost and heuristic values.
- \( \text{cost}(p) \) is the cost of path \( p \).
- \( h(p) \) estimates the cost from the end of \( p \) to a goal.

Let \( f(p) = \text{cost}(p) + h(p) \). \( f(p) \) estimates the total path cost of going from a start node to a goal via \( p \).

Illustrative Example — Best First Search

[Diagram showing a graph with nodes S, A, B, C, G and edges with costs: S-A (2), A-B (1), B-C (1), C-G (2), S-C (4)]

Recall best first: S-A-C-G (not optimal)
A* : S-A-B-C-G (optimal)

Admissibility of A*

If there is a solution, A* always finds an optimal solution — the first path to a goal selected — if:
- the branching factor is finite
- arc costs are bounded above zero (there is some \( \epsilon > 0 \) such that all of the arc costs are greater than \( \epsilon \)), and
- \( h(n) \) is a lower bound on the length (cost) of the shortest path from \( n \) to a goal node.

Admissible heuristics never overestimate the cost to the goal.

Why is A* with admissible \( h \) optimal?

- Assume \( \text{path} \) is the optimal
- \( f(p) = \text{cost}(s, p) + h(p) < \text{cost}(\text{path}) \) due to \( h \) being a lower bound
- \( \text{cost}(s, g) < \text{cost}(s, p') + \text{cost}(p', g) \) due to optimality of \( \text{path} \)
- Therefore \( \text{cost}(s, p) + h(p) = f(p) < \text{cost}(s, p') + \text{cost}(p', g) \)
- Therefore, we will never choose \( \text{path}' \) while \( \text{path} \) is unexplored.
- A* halts, as the costs of the paths on the frontier keeps increasing, and will eventually exceed any finite number.

Graph Search Algorithm - with Multiple Path Pruning

Input: a graph, a set of start nodes, Boolean procedure \( \text{goal}(n) \) that tests if \( n \) is a goal node.

\( \text{frontier} := \{ (s) : s \text{ is a start node} \}; \)
\( \text{has}_\text{path} := \{ \}; \)
while \( \text{frontier} \) is not empty:
- select and remove path \( \langle n_0, \ldots, n_k \rangle \) from \( \text{frontier} \);
- if \( n_k \notin \text{has}_\text{path} \):
  - add \( n_k \) to \( \text{has}_\text{path} \);
  - if \( \text{goal}(n_k) \)
    - return \( \langle n_0, \ldots, n_k \rangle \);
  - for every neighbor \( n \) of \( n_k \)
    - add \( \langle n_0, \ldots, n_k, n \rangle \) to \( \text{frontier} \);
end while
How do we construct a heuristic?

Relax the game (make it simpler, easier)
1. Can move tile from position A to position B if A is next to B
   (ignore whether or not position is blank)
2. Can move tile from position A to position B if B is blank
   (ignore adjacency)
3. Can move tile from position A to position B
   ▶ leads to manhattan distance heuristic
   ▶ To solve the puzzle need to slide each tile into its final position
   ▶ Admissible

How do we construct a heuristic?

Relax the game (make it simpler, easier)
1. Can move tile from position A to position B if A is next to B
   (ignore whether or not position is blank)

Graph for the Delivery Robot

cost = distance travelled
heuristic = euclidean distance

Topological Map of DC/MC

cost = number of doors
heuristic? = euclidean distance

Summary of Search Strategies

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Frontier Selection</th>
<th>Halts?</th>
<th>Space</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depth-first</td>
<td>Last node added</td>
<td>No</td>
<td>Linear</td>
<td>Exp</td>
</tr>
<tr>
<td>Breadth-first</td>
<td>First node added</td>
<td>Yes</td>
<td>Exp</td>
<td>Exp</td>
</tr>
<tr>
<td>Heuristic depth-first</td>
<td>Local(^1) min (h(n))</td>
<td>No</td>
<td>Linear</td>
<td>Exp</td>
</tr>
<tr>
<td>Best-first</td>
<td>Global(^2) min (h(n))</td>
<td>No</td>
<td>Exp</td>
<td>Exp</td>
</tr>
<tr>
<td>Lowest-cost-first</td>
<td>Minimal cost((n))</td>
<td>Yes</td>
<td>Exp</td>
<td>Exp</td>
</tr>
<tr>
<td>(A^*)</td>
<td>Minimal (f(n))</td>
<td>Yes</td>
<td>Exp</td>
<td>Exp</td>
</tr>
</tbody>
</table>

\(^1\)Locally in some region of the frontier
\(^2\)Globally for all nodes on the frontier
A searcher can prune a path that ends in a node already on the path, without removing an optimal solution.

Using depth-first methods, with the graph explicitly stored, this can be done in constant time (add a flag to each node).

For other methods, the cost is linear in path length, since we have to check for cycles in the current path.

Multiple path pruning: prune a path to node $n$ that the searcher has already found a path to.

Multiple-path pruning subsumes a cycle check.

This entails storing all nodes it has found paths to.

Want to guarantee that an optimal solution can still be found.

Problem: what if a subsequent path to $n$ is shorter than the first path to $n$?

- remove all paths from the frontier that use the longer path.
- change the initial segment of the paths on the frontier to use the shorter path.
- ensure this doesn’t happen. Make sure that the shortest path to a node is found first (lowest-cost-first search)

Suppose path $p$ to $n$ was selected, but there is a shorter path to $n$. Suppose this shorter path is via path $p'$ on the frontier.

- Suppose path $p'$ ends at node $n'$.
- $\text{cost}(p) + h(n) \leq \text{cost}(p') + h(n')$ because $p$ was selected before $p'$.
- $\text{cost}(p') + \text{cost}(n', n) < \text{cost}(p)$ because the path to $n$ via $p'$ is shorter.

\[ \text{cost}(n', n) < \text{cost}(p) - \text{cost}(p') \leq h(n') - h(n). \]

You can ensure this doesn’t occur if $h(n') - h(n) \leq \text{cost}(n', n)$.

Heuristic function $h$ satisfies the **monotone restriction** if $h(m) - h(n) \leq \text{cost}(m, n)$ for every arc $(m, n)$.

$h(m) - h(n)$ is the heuristic estimate of the path cost from $m$ to $n$.

The heuristic estimate of the path cost is always less than the actual cost.

If $h$ satisfies the monotone restriction, $A^*$ with multiple path pruning always finds the shortest path to a goal.

This is a strengthening of the admissibility criterion.

So far all search strategies that are guaranteed to halt use exponential space.

**Idea:** let’s recompute elements of the frontier rather than saving them.

Look for paths of depth 0, then 1, then 2, then 3, etc.

You need a depth-bounded depth-first searcher.

If a path cannot be found at depth $B$, look for a path at depth $B + 1$. Increase the depth-bound when the search fails unnaturally (depth-bound was reached).
Iterative Deepening Complexity

Complexity with solution at depth $k$ & branching factor $b$:

<table>
<thead>
<tr>
<th>level</th>
<th># times each node is expanded</th>
<th>iterative deepening</th>
<th># nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>$k$</td>
<td>$b$</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>$k - 1$</td>
<td>$b^2$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$k - 1$</td>
<td>1</td>
<td>2</td>
<td>$b^{k-1}$</td>
</tr>
<tr>
<td>$k$</td>
<td>1</td>
<td>1</td>
<td>$b^k$</td>
</tr>
</tbody>
</table>

$\geq b^k \leq b^k \left(\frac{b}{b-1}\right)^2$

Depth-first Branch-and-Bound

- Way to combine depth-first search with heuristic information.
- Finds optimal solution.
- Most useful when there are multiple solutions, and we want an optimal one.
- Uses the space of depth-first search.

Direction of Search

- The definition of searching is symmetric: find path from start nodes to goal node or from goal node to start nodes.
- **Forward branching factor**: number of arcs out of a node.
- **Backward branching factor**: number of arcs into a node.
- Search complexity is $b^n$. Should use forward search if forward branching factor is less than backward branching factor, and vice versa.
- Note: sometimes when graph is dynamically constructed, you may not be able to construct the backwards graph.

Bidirectional Search

- You can search backward from the goal and forward from the start simultaneously.
- This wins as $2b^{k/2} \ll b^k$. This can result in an exponential saving in time and space.
- The main problem is making sure the frontiers meet.
- This is often used with one breadth-first method that builds a set of locations that can lead to the goal. In the other direction another method can be used to find a path to these interesting locations.

Island Driven Search

- **Idea**: find a set of islands between $s$ and $g$.
  
  \[ s \rightarrow i_1 \rightarrow i_2 \rightarrow \ldots \rightarrow i_{m-1} \rightarrow g \]

  There are $m$ smaller problems rather than 1 big problem.
  - This can win as $mb^{k/m} \ll b^k$.
  - The problem is to identify the islands that the path must pass through. It is difficult to guarantee optimality.
  - You can solve the subproblems using islands $\Rightarrow$ hierarchy of abstractions.
Dynamic Programming

- Start from goal and work backwards
- Compute the cost-to-goal at each node recursively
- Cost from \( n \rightarrow \text{goal} \) is
  \[
  \text{Cost from } m \rightarrow \text{goal} + \text{cost from } n \text{ to } m
  \]
- \( \text{dist}(n) \) is cost-to-goal from node \( n \), and \( \text{cost}(n,m) \) is cost to go from \( n \) to \( m \)

\[
\text{dist}(n) = \begin{cases} 
0 & \text{if } n \text{ is goal} \\
\min_m (\text{cost}(n,m) + \text{dist}(m)) & \text{otherwise}
\end{cases}
\]

- \( \text{dist}(n) \) is a value function over nodes
- \( \text{policy}(n) \) is best \( m \) for each \( n \), so best path is
  \[
  \text{path}(n, \text{goal}) = \arg \min_m (\text{cost}(n,m) + \text{dist}(m)), \text{path}(m, \text{goal})
  \]

- problem: space needed to store entire graph

Dynamic Programming - Example
Constraints (Poole & Mackworth chapter 4)