Last Time: Graph Search

Find path between 2 points in an arbitrary graph.

Represent all possible paths from A with a tree:

Labs

Last Week: Robots in Mazes
This Week: Uniform Cost Search, MapQuest
Graph Search Algorithm

Basic Algorithm:
- Initialize agenda (list of nodes to consider)
- Repeat the following:
  - Remove one node from the agenda
  - Add its children to the agenda
  - until goal is found or agenda is empty
- Return resulting path

Order Matters!

Strategy: Replace last node in agenda by its successors

Order Matters!

Strategy: Remove first node and add its successors to end

Notes
Dynamic Programming

As applies to search:
(Depends slightly on which algorithm we're using)

BFS: The shortest path \( S \rightarrow X \rightarrow G \) is made up of the shortest path \( S \rightarrow X \) and the shortest path \( X \rightarrow G \).

DFS: A path \( S \rightarrow X \rightarrow G \) is made up of a path \( S \rightarrow X \) and a path \( X \rightarrow G \).

The moral: once we have found a path \( S \rightarrow X \), we don’t need to spend time looking for other paths through \( X \).

Algorithm (including dynamic programming):

- Initialize agenda (list of nodes to consider)
- Initialize visited set (set of states visited)
- Repeat the following:
  - Remove one node from the agenda
  - For each of that node’s children:
    - If its state is in the visited list, skip it
    - Otherwise, add it to agenda and add its state to visited list
  - Until goal is found or agenda is empty
- Return resulting path

Python Framework

- **SearchNode class**:
  - Attributes:
    - state (arbitrary)
    - parent (instance of SearchNode, or None)
  - Methods:
    - path (returns list of states representing path from root)

- **search function**:
  - Arguments:
    - successor function (function state→list of states)
    - starting state
    - goal test (function state→bool)
    - dfs (True for DFS, False for BFS)
def search(successors, start_state, goal_test, dfs = False):
    if goal_test(start_state):
        return [start_state]
    else:
        agenda = [SearchNode(start_state, None)]
        visited = {start_state}
        while len(agenda) > 0:
            parent = agenda.pop(-1 if dfs else 0)
            for child_state in successors(parent.state):
                child = SearchNode(child_state, parent)
                if goal_test(child_state):
                    return child.path()
                if child_state not in visited:
                    agenda.append(child)
                    visited.add(child.state)
        return None

Example

Find path $A \rightarrow I$, BFS w/ DP

What is a Graph?

Set $V$ of vertices
Set $E$ of edges connecting vertices
Set $W$ of edge costs (or "weights")
Uniform-Cost Search

Consider searching for least-cost paths instead of shortest paths. Instead of popping from agenda based on when nodes were added, pop based on cost of the paths they represent.

Slight change to framework:
- SearchNode class:
  - Attributes:
    - state (arbitrary)
    - parent (instance of SearchNode, or None)
    - cost of whole path from start
  - Methods:
    - path (returns list of states representing path from root)
- uniform_cost_search function:
  - Arguments:
    - successor function (state→list of (state,cost) tuples)
    - starting state
    - goal test (function state→bool)

```python
def uniform_cost_search(successors, start_state, goal_test):
    if goal_test(start_state):
        return [start_state]
    agenda = [(0, SearchNode(start_state, None, cost=0))]
    expanded = set()
    while len(agenda) > 0:
        agenda.sort()
        priority, parent = agenda.pop(0)
        if parent.state not in expanded:
            expanded.add(parent.state)
            if goal_test(parent.state):
                return parent.path()
            for child_state, child_cost in successors(parent.state):
                child = SearchNode(child_state, parent, parent.cost+child_cost)
                if child.state not in expanded:
                    agenda.append((child.cost, child))
    return None
```

Testing for goal condition must be done at expansion time, not at visit time. Similarly for dynamic programming.
Problem?

So far, searches have radiated outward from the starting point. We only notice the goal when we stumble upon it.

Too much time spent searching on the **wrong side of the goal**.

Example

Find path $A \rightarrow I$, Uniform Cost Search
Heuristics

So far, our searches only consider start-to-current. We can add heuristics to consider an estimate of current-to-goal as well.

\[ h(x) \text{ estimate of cost of lowest-cost path } X \rightarrow \text{goal} \]

- **SearchNode class:**
  - **Attributes:**
    - state (arbitrary)
    - parent (instance of SearchNode, or None)
    - cost of whole path from start
  - **Methods:**
    - path (returns list of states representing path from root)

- **uniform_cost_search function:**
  - **Arguments:**
    - successor function (state \( \rightarrow \) list of (state, cost) tuples)
    - starting state
    - goal test (function state \( \rightarrow \) bool)
    - heuristic (function state \( \rightarrow \) estimated cost)

```python
def uniform_cost_search(successors, start_state, goal_test, heuristic=lambda s: 0):
    if goal_test(start_state):
        return [start_state]
    agenda = [(heuristic(start_state), SearchNode(start_state, None, cost=0))]
    expanded = set()  
    while len(agenda) > 0:
        agenda.sort()  
        priority, parent = agenda.pop(0)
        if parent.state not in expanded:
            expanded.add(parent.state)
            if goal_test(parent.state):
                return parent.path()
            for child_state, cost in successors(parent.state):
                child = SearchNode(child_state, parent, parent.cost+cost)
                if child.state not in expanded:
                    agenda.append((child.cost+heuristic(child.state), child))
    return None
```

**Example**

Find path \( E \rightarrow I \), A*, heuristic: \( h(s) = M(s, I)/2 \)

![Diagram of a search graph with states and edges showing distances and directions leading from E to I, including one possible path labeled E → D → F → I.](image)
Admissible Heuristics

An admissible heuristic does not overestimate the actual cost of the shortest cost path.

If the heuristic \( h(s) \) is larger than the actual cost from \( s \) to goal, then the “best” solution may be missed!

If the heuristic is an underestimate, the search space will be larger than necessary, but we are guaranteed the shortest path.

The ideal heuristic should be:
- as close as possible to actual cost (without overestimating)
- easy to calculate

A* (without DP) is guaranteed to find least-cost path if heuristic is admissible.

With DP, heuristic must also be consistent.

Check Yourself!

Consider searching in a four-action grid (up, down, left, right), where all actions have cost 1. Let \((r_0, c_0)\) represent the current location, and \((r_1, c_1)\) represent the goal.

Which of the following heuristics are admissible?

1. \(\text{abs}(r_0-r_1) + \text{abs}(c_0-c_1)\)
2. \(\min(\text{abs}(r_0-r_1), \text{abs}(c_0-c_1))\)
3. \(\max(\text{abs}(r_0-r_1), \text{abs}(c_0-c_1))\)
4. \(2\times\min(\text{abs}(r_0-r_1), \text{abs}(c_0-c_1))\)
5. \(2\times\max(\text{abs}(r_0-r_1), \text{abs}(c_0-c_1))\)

Check Yourself!

Which of the admissible heuristics minimizes the number of nodes expanded?

1. \(\text{abs}(r_0-r_1) + \text{abs}(c_0-c_1)\)
2. \(\min(\text{abs}(r_0-r_1), \text{abs}(c_0-c_1))\)
3. \(\max(\text{abs}(r_0-r_1), \text{abs}(c_0-c_1))\)
4. \(2\times\min(\text{abs}(r_0-r_1), \text{abs}(c_0-c_1))\)
5. \(2\times\max(\text{abs}(r_0-r_1), \text{abs}(c_0-c_1))\)
Heuristics in Other Domains

Consider the "word ladder" problem from EX12 (and last lecture). Searching from "quiz" to "best".

Example: 8-Puzzle

Start

1 2 3
4 5 6
7 8

Goal

1 2
3 4 5
6 7 8

Large number of board configurations (states):
- $9! = 362,880$ (if you count all)
- $9!/2 = 181,440$ accessible from start state

Almost half of accessible states (84,516) are expanded by UC.
Check Yourself!

Consider three heuristics for the “eight puzzle”:
1. 0
2. number of tiles out of place
3. sum over tiles of Manhattan distances to their goals

Let $M_i =$ num. moves in the best solution using heuristic $i$.
Let $E_i =$ num. states expanded using heuristic $i$.

Which of the following are true?
1. $M_1 = M_2 = M_3$
2. $M_1 > M_2 > M_3$
3. $E_1 = E_2 = E_3$
4. $E_1 \geq E_2 \geq E_3$
5. the same “best” solution results from all three heuristics

Recap

Developed a new class of search algorithms: uniform cost
Developed a new class of optimizations: heuristics

Summary of Search Algorithms:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Agenda</th>
<th>Goal Test</th>
<th>GP</th>
<th>Guarantees</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFS</td>
<td>Stack (LIFO)</td>
<td>Visit</td>
<td>Visited Set</td>
<td>Some Path</td>
</tr>
<tr>
<td>BFS</td>
<td>Queue (FIFO)</td>
<td>Visit</td>
<td>Visited Set</td>
<td>Shortest Path</td>
</tr>
<tr>
<td>A*</td>
<td>Priority Queue</td>
<td>Expand</td>
<td>Expanded Set</td>
<td>Least-cost Path</td>
</tr>
</tbody>
</table>

† Provided a path exists
* In a finite search domain