Optimality & Simple Heuristic Search

CMPUT 366: Intelligent Systems

P&M §3.6

Logistics

- TA office hours begin this week
  - See eClass page for times and meeting links
- Assignment #1 released next week

Lecture Outline

1. Logistics
2. Optimality & Least Cost First Search
3. Heuristics

Recap: Uninformed Search

Different **search strategies** have different properties and behaviour

- **Depth first search** is space-efficient but not always complete or time-efficient
- **Breadth first search** is complete and always finds the shortest path to a goal, but is not space-efficient
- **Iterative deepening search** can provide the benefits of both, at the expense of some time-efficiency
- All three strategies must potentially expand **every node**

Updated Iterative Deepening Search

Input: a *graph*; a set of *start nodes*; a $goal$ function

for *max_depth* from 1 to ∞:
    more_nodes := False
    $frontier := \{\langle s \rangle \mid s$ *is a start node*$\}$
    while $frontier$ is not empty:
        select the newest path $\langle n_0, \ldots, n_k \rangle$ from $frontier$
        remove $\langle n_0, \ldots, n_k \rangle$ from $frontier$
        if $goal(n_k)$:
            return $\langle n_0, \ldots, n_k \rangle$
        if *k < max_depth:*
            for each neighbour n of n_k:
                add $\langle n_0, \ldots, n_k, n \rangle$ to *frontier*
        else if n_k has neighbours:
            more_nodes := True
    end-while
    if *more_nodes* = False:
        return None

Optimality

Definition:
An algorithm is **optimal** if it is guaranteed to return an optimal (i.e., **minimal-cost**) solution **first** (i.e., before any other solution).

Question: Which of the three algorithms presented so far is optimal? Why?

Least Cost First Search

- *None* of the algorithms described so far is guided by **arc costs**
  - BFS and IDS are implicitly guided by **path length**, which can be the same for uniform-cost arcs
- They return a path to a goal node as soon as they happen to blunder across one, but it may not be the optimal one
- **Least Cost First Search** is a search strategy that is **guided by arc costs**

Least Cost First Search

Input: a *graph*; a set of *start nodes*; a *goal* function

$frontier := \{\langle s \rangle \mid s$ *is a start node*$\}$
while $frontier$ is not empty:
    select the cheapest path $\langle n_0, \ldots, n_k \rangle$ from *frontier*
    remove $\langle n_0, \ldots, n_k \rangle$ from $frontier$
    if $goal(n_k)$:
        return $\langle n_0, \ldots, n_k \rangle$
    for each neighbour n of n_k:
        add $\langle n_0, \ldots, n_k, n \rangle$ to *frontier*
end while

i.e., $cost(\langle n_0, \ldots, n_k \rangle) \leq cost(p)$ for all other paths $p \in frontier$

Question:

What **data structure** for the frontier implements this search strategy?

Least Cost First Search Analysis

- Least Cost First Search is **complete** and **optimal** if there is $\epsilon > 0$ with $cost(\langle n_1, n_2 \rangle) > \epsilon$ for every arc $\langle n_1, n_2 \rangle$:

    1. Suppose $\langle n_0, \ldots, n_k \rangle$ is the optimal solution
    2. Suppose that p is any non-optimal solution
       So, $cost(p) > cost(\langle n_0, \ldots, n_k \rangle)$
    3. For every $0 \leq \ell \leq k$, $cost(\langle n_0, \ldots, n_\ell \rangle) < cost(p)$
    4. So p will never be removed from the frontier before $\langle n_0, \ldots, n_k \rangle$

- What is the worst-case **space complexity** of Least Cost First Search?
  [A: $O(m)$] [B: $O(mb)$] [C: $O(b^m)$] [D: it depends]
- When does Least Cost First Search have to expand **every node** of the graph?

Summary: Search Strategies

	Depth First	Breadth First	Iterative Deepening	Least Cost First
Selection	Newest	Oldest	Newest, multiple	Cheapest
Data structure	Stack	Queue	Stack, counter	Priority queue
Complete?	Finite graphs only	Complete	Complete	Complete if $cost(p) > \varepsilon$
Space complexity	$O(mb)$	$O(b^m)$	$O(mb)$	$O(b^m)$
Time complexity	$O(b^m)$	$O(b^m)$	$O(mb^m)$ **	$O(b^m)$
Optimal?	No	No	No	Optimal

Domain Knowledge

- Domain-specific knowledge can help speed up search by identifying **promising directions** to explore
- We will encode this knowledge in a function called a **heuristic function** which **estimates** the cost to get from a node to a goal node
- The search algorithms in this lecture take account of this heuristic knowledge when **selecting** a path from the frontier

Heuristic Function

Definition:
A **heuristic function** is a function $h(n)$ that returns a non-negative estimate of the cost of the cheapest path from node n to a goal node.

- For paths: $h(\langle n_0, \ldots, n_k \rangle) = h(n_k)$
- Uses only **readily-available** information about a node (i.e., easy to compute)
- **Problem-specific**

Admissible Heuristic

Definition:
A heuristic function is **admissible** if $h(n)$ is **always less than or equal** to the cost of the cheapest path from n to any goal node.

- i.e., $h(n)$ is a **lower bound** on cost$(\langle n, \ldots, g \rangle)$ for any **goal node** g

Example Heuristics

- **Euclidean distance** for DeliveryBot
  (ignores that it can't go through walls)
- **Number of dirty rooms** for VacuumBot
  (ignores the need to move between rooms)
- **Points** for chess pieces
  (ignores positional strength)

Constructing Admissible Heuristics

- Search problems try to find a cost-minimizing path, subject to **constraints** encoded in the search graph
- How to construct an easier problem? **Drop** some constraints.
    - This is called a **relaxation** of the original problem
- The cost of the optimal solution to the relaxation will always be an **admissible heuristic** for the original problem (**Why?)**
- **Neat trick**: If you have two admissible heuristics h_1 and h_2, then $h_3(n) = \max\{h_1(n), h_2(n)\}$ is admissible too! (**Why?**)

Simple Uses of Heuristics

- **Heuristic depth first search:** Add neighbours to the frontier in **decreasing order** of their heuristic values, then run depth first search as usual
  - Will explore most promising successors first, but
  - Still explores **all paths** through a successor before considering other successors
  - Not complete, not optimal
- **Greedy best first search:** Select path from the frontier with the **lowest heuristic** value
  - Not guaranteed to work any better than breadth first search (**why?**)

Summary

- **Domain knowledge** can help speed up graph search
- Domain knowledge can be expressed by a **heuristic function**, which **estimates** the cost of a path to the goal from a node
- **Admissible** heuristics can be built from **relaxations** of the original problem
- Surprisingly, simple uses of heuristics do not guarantee improved performance
- Next time: **A* algorithm** for provably optimal use of admissible heuristics