CS60045 Artificial Intelligence Autumn 2023

Simple Search Techniques

Generic search algorithm (Best-first search)

Create an empty ***frontier***, and insert **start_node** to ***frontier***.
Create an empty ***reached*** table, and insert (**start_node**, **initial_cost**) to ***reached***.
While ***frontier*** is not empty, repeat:
    Extract the best **node** from ***frontier***.
    If **node** is a goal node, return **node**.
    Expand **node** to get the list of child nodes.
    For each **child**, repeat:
        Calculate **new_path_cost** from root to **child** (node_path_cost + transition_cost)
        If (**child** in not in ***reached***)
            Add **child** to ***frontier***.
            Record **node** as the parent of **child**.
            Add (**child**, **new_path_cost**) to ***reached***.
        Else if (**new_path_cost** to **child** is better than the earlier path cost to child)
            Add **child** to ***frontier***.
            Record **node** as the parent of **child**.
            Replace (**child**, **old_path_cost**) by (**child**, **new_path_cost**) in ***reached***.
Return ***failure***.

How to choose the best node for expanding

Governed by a function $f(n)$ of the nodes n.

A general data structure for ***frontier*** is a priority queue.

Breadth-first search: $f(n)$ is the depth $d(n)$ of n from the root node. ***frontier*** is a FIFO queue.

Depth-first search: $f(n)$ is the negative of the depth of n from the root node. ***frontier*** is a LIFO stack.

Uniform-cost search: $f(n)$ is the sum $g(n)$ of the edge costs from the root node to n.

Greedy best-first search: $f(n)$ is an estimate $h(n)$ of the remaining cost to reach a goal.

A* search: $f(n) = g(n) + h(n)$ (estimated best cost from root to goal via n).

Weighted A* search: $f(n) = g(n) + W \times h(n)$ with $1 < W < \infty$ (focus is on the remaining cost without completely ignoring the cost so far).

Necessity to remember reached states

Redundant paths

- Cycles
- Paths with larger costs (than discovered so far)

Without remembering reached states, the search is bound to loop.

Graph-search: Reached states are remembered.

- Useful when there are many redundant paths.
- Demands memory.

Tree-like search: Reached states are not remembered.

- Useful when there are no or not many redundant paths.
- Memory efficient.

Open and closed nodes

Nodes that are not generated are neither open nor closed.

A node that is generated but not expanded is called **open**.

An (generated and) expanded node is called **closed**.

The collection of all (currently) open nodes is stored in ***frontier***.

The collection of all open and closed nodes is stored in ***reached***.

In the “else” part of the best-first search algorithm, a reached child node is opened because a cheaper path to this already reached node is discovered.

- If that child node is already open, then its $f(\ )$ value may change. Record that in ***frontier***.
- If that child is closed, then it is **reopened**. This situation is undesirable, because reopening nodes many times may increase the running time significantly. A good search algorithm would aim at avoiding this as much as possible.

Termination of the algorithm

If the frontier becomes empty, the search fails.

The search succeeds (and the algorithm stops) as soon as a goal node is extracted from ***frontier***. But note that:

1. This is not necessarily an optimal goal node. Its extraction is based on *f*(), not on *g*().
2. Even if it is, continuing the search could discover better root-to-goal paths to this node.

If no closed node ever needs to be reopened, **Situation 2** will not happen. **Situation 1** may still happen.

Late goal test: A goal node stays in the open state in ***frontier*** until it is ready to be extracted. This gives the node time to improve its *g*() value.

Early goal test: A goal node is returned as soon as it is generated (that is, becomes open).

Properties of a search algorithm

Completeness: If a solution exists, can it be eventually found?

Cost optimality: Does the algorithm generate the goal of best cost?

Time complexity

Memory requirement

Uninformed search: No additional information how far a node is to a goal.

Informed search: Some heuristic estimates for $h(n)$ are available.

Breadth-first search (BFS)

Useful when all transitions have the same cost.

$f(n) = d(n)$

Non-cycle redundant paths can never be found.

Reached still needs to be maintained to avoid looping in cycles.

Early goal test is applicable to BFS.

Complete: If a goal node exists, it will be eventually found, even if the state space is infinite.

Cost-optimal: Yes if all transitions have the same cost, not necessarily otherwise.

Running time: $O(b^d)$, where b is the branching factor (assumed constant), and d is the depth of the goal node.

Space requirement: $O(b^d)$

Uniform-cost search (UCS, Dijkstra's algorithm)

$f(n) = g(n)$ (path cost from root to n)

The best cost to a node may reduce when new paths are found.

We have to use the late goal test.

Complete: Yes (in absence of 0-cost transitions).

Cost-optimal: Yes (once a node is taken out from frontier, its cost cannot reduce in future).

Time and space requirement: $O(b^{C^*/\varepsilon})$, where C^* is the optimal cost, and ε is a lower bound on the cost of each action. We assume that $\varepsilon > 0$. If 0-cost actions are allowed, then the search may explore an infinite tree with these transitions only, without making any progress toward the goal.

Depth-first search (DFS)

$f(n) = -d(n)$

Often implemented as a tree-like search (reached states are not remembered)

Completeness

- Yes for finite acyclic graphs
- Yes for finite cyclic graphs with reached states remembered
- Yes for finite cyclic graphs (even if reached states are not remembered, cycles can be detected using back edges)
- No for infinite graphs even if there are no cycles. (In the factorial-square_root-floor example, one may keep on taking factorials or square roots indefinitely without ever reaching the goal.)

Cost-optimal: No (the first goal node taken out of the frontier is reported as the solution).

Space requirement: $O(bd_{max})$ — efficient for finite trees (not an exponential function of d).

Backtracking

DFS keeps a single path with the internal nodes having all its siblings.

Backtracking search does not keep the siblings.

Children of each node are generated one by one.

Each node needs to remember the last child generated.

Size of the frontier is $O(d_{max})$ (the factor of b is gone).

Depth-limited Search

DFS is not cost-optimal and may be incomplete.

It has only one advantage of very small memory requirement (assuming that the reached states are not remembered).

If we know a bound D for a goal node, we restrict DFS to depth D.

D can be the diameter of the state-space graph (if known or computable a priori).

No need to check cycles.

Running time is $O(b^D)$ and memory requirement is $O(bD)$.

Incomplete if D is a poor choice.

Iterative Deepening Search (IDS)

Useful if D cannot be computed apriori.

Repeat running the depth-limited search for $D = 0, 1, 2, \ldots$ until a goal is found.

Explores small subtrees again and again, but the cost increase is tolerable. Total number of nodes generated is $(d + 1) + db + (d - 1)b^2 + (d - 2)b^3 + \ldots + b^d = O(b^d)$. This is the same as BFS.

Numerical example: Take $b = 4$ and $d = 15$. Suppose that each state requires 1 KB storage.

- BFS generates at most $1 + 4 + 4^2 + \ldots + 4^{15} = 1{,}163{,}236{,}693$ nodes.
- IDS generates at most $16 + 15 \times 4 + 14 \times 4^2 + 13 \times 4^3 + \ldots + 4^{15} = 1{,}372{,}036{,}204$ nodes.
- Memory requirement for BFS is about 1.16 TB.
- Memory requirement for IDS is about 61 KB.

Running time is $O(b^d)$ (success at depth d) or $O(b^{dmax})$ (goal node does not exist).

Hybrid approach: Run BFS as long as memory permits. Then run IDS.

Bidirectional search

Applicable when a/the solution is known beforehand. The task is only to find an optimal path.

Run one instance of best-first search from the initial state.

Run a second instance of best-first search from the goal state by reversing the transitions.

Two frontiers and two reached tables need to be maintained.

When the two frontiers meet, a solution (not necessarily the optimal one) is found. There may be need to find multiple intersections of the two frontiers to locate the optimum path.

If C^* is the optimal cost to reach the goal, neither of the two searches needs to expand a node of cost (much) greater than $C^* / 2$. This gives good speedup.

For example, if both the searches are BFS, and the goal is at depth d, then unidirectional BFS takes time and space of the order $O(b^d)$, whereas bidirectional BFS requires $O(b^{d/2})$ time and space.