CS4725/CS6705

Chapter 3: Solving Problems by Searching

Problem-solving agents

- Problem-solving agent: a type of goal-based agent
- Task: to find a sequence of actions that will lead to a desirable state
- We will look at various search algorithms and discuss their advantages and disadvantages.
- For now: assume an environment that is static, fully observable, discrete, deterministic

Example problem: 8-puzzle

- Idea: Given the puzzle in some configuration, try to slide the tiles around to achieve some other configuration.
- [Example on the board]

Defining a search problem

- A **state representation**: how do we represent the information that is relevant for each possible state that we could be in?
- The **initial state** that the agent starts in

(continued on next slide)

Defining a search problem (cont'd)

- A description of the **actions** available to the agent
  - **Successor function or transition model**: for any state *s* and action *a*, Result(*s,a*) is the state that results from performing action *a* in state *s*.
  - [Note: The initial state and successor function define the entire state space (all states that are reachable). A **path** in the state space is a sequence of states connected by a sequence of actions.]

Defining a search problem (cont'd)

- **Goal test**: For any state, does it satisfy our goal?

  [Goal could be a specific state, any state that belongs to some set, any state that satisfies some property]

- **Path cost function**: assigns a numeric cost to each path

  [For now, assume that the cost of a path is the sum of the costs of the actions on the path.]

Defining a search problem (cont'd)

- A **solution** is a path from the initial state to a goal state.
- An **optimal solution** is one with minimum path cost.
- [Specification of the 8-puzzle as a search problem on the board – initial state, actions, successor function, etc.]

Another example: 8-queens

- Idea: place 8 queens on a chess board so that no queen attacks any other. [A queen attacks any piece in the same row, column or diagonal (with no other pieces in between).]

- [Example of a solution on the board]

8-queens: 2 problem formulations

- Incremental formulation: start with an empty board; each action adds a new queen somewhere

- Complete-state formulation: start with 8 queens on the board; each action moves a queen (to fix some existing problem)

8-queens:
one possible incremental fomulation

- **States**: any arrangement of 0-8 queens
- **Initial state:** empty board
- **Successor function:** add a queen to any empty square
- **Goal test:** 8 queens on board, none attacked?
- **Path cost:** not important; we only care about whether we have a solution or not

8-queens (cont'd)

- Problem with this previous formulation:
  - State space is too huge: 3×10^{14} states
- Better plan: prevent placement of a queen in a square that is already attacked...

8-queens: A better formulation

- **States:** arrangement of n queens ($0 \leq n \leq 8$), one per column in the leftmost n columns, with no queen attacking another
- **Successor function:** add a queen somewhere in the leftmost empty column such that it is not attacked by any other queen
- [If no such move exists, then we've hit a dead end in our graph.]

8-queens (cont'd)

- This improved formulation reduces the state space from 3 x 10^{14} to 2057.
- 100 queens instead of 8: state space reduced from 10^{400} to 10^{52} (huge reduction, but this is still too big for the algorithms we'll talk about later)
- Chapter 4: complete-state formulation; Chapter 6: algorithm that makes even the million-queens problem easy to solve

Real-world problems

- Route-finding problems (travel planning, military operations, computer networks, etc.)
- Touring problems
- Traveling salesperson problem
- VLSI layout
- Robot navigation
- Automatic assembly sequencing

Real-world problems

- One example: map showing several cities; some pairs are connected by arcs, labelled by a distance/cost. Goal: find the shortest path from point A to point B
  - **States:** current location
  - **Initial state:** initial location
  - **Successor function:** travel from current city to any connected city
  - **Goal test:** are we in the goal city?
  - **Path cost:** total distance of all arcs travelled

Searching

- Now that we know how to formulate a problem, we will look at different techniques for search through the state space to find a solution.
- [For now: techniques might seem less than intelligent – basically brute-force search – but we'll soon see how we can improve on this.. Also, even the naive algorithms can be quite effective for small problems.]

Searching (cont'd)

- Build a **search tree**, starting from the initial state.
  - Check to see if current state is a goal state.
  - If not, expand the current state, by applying the successor function (i.e., what actions can we legally apply and what states would those lead us to?)
  - Now, which unexpanded state will we expand next? This is determined by our search strategy.
  - [For now: ignore the fact that we might see the same state pop up more than once. More on this later.]

Search nodes

- Think of a node n in our tree as a data structure with 5 components:
  - **State:** the state to which the node corresponds
  - **Parent-node:** the node that generated this node
  - **Action:** the action that was applied to the parent
  - **Path-cost:** cost of the path from initial state to this node, usually denoted g(n)
  - **Depth:** number of steps on the path from initial state to n
- [8-puzzle example on the board]

Search process

- Look at the nodes that have been generated, but not yet expanded (those on “the frontier”) and decide which one to expand next.
- Assume that we keep track of frontier nodes on a queue-like structure. The order in which nodes are inserted into the queue depends on the search algorithm.
- Once we select a node to expand, we check if it is a goal node. If not, expand it.

Graph search

- **Graph search** (as opposed to tree search):
  - Keep track of a list of nodes that have already been explored.
  - Whenever a node is generated that has already been explored, we do not add it to the list of frontier nodes.
  - This requires extra storage (to keep track of the list of explored nodes), but it eliminates a lot of unnecessary repeated work.

Evaluating search algorithms

- **Completeness:** If there is a solution, is it guaranteed to find one?
- **Optimality:** Does it find an optimal solution?
- **Time complexity:** How long does it take to find a solution? (How many nodes are expanded?)
- **Space complexity:** How much memory is needed to perform the search?

Evaluating search algorithms

- When we consider time and space complexity, we will look at them as functions of:
  - b: the branching factor: the maximum number of successors of any node
  - d: depth of the shallowest goal node
  - m: maximum length of any path in the state space

Uninformed search strategies (blind search)

- No information about states other than what is in the problem definition
- Can only generate successors and check for goal states

Breadth-first search

- Root is expanded first, then all successors of the root, then all of their successors, etc.
- Queue: newly generated successors are added to the end of the queue
- [Example on the board]

Breadth-first search

- Complete (as long as branching factor is finite)
- Not necessarily optimal, but it **is** optimal **if** the path cost is a nondecreasing function of node depth (e.g., if all actions have equal cost)
- Time complexity: $O(b^d)$ = bad news
- Space complexity: $O(b^d)$ = very bad news

Uniform-cost search

- Instead of expanding the shallowest node, expand the node with the lowest path cost. (Note: identical to BFS if all action costs are equal)
- [Example on the board]

Uniform cost search

- Complete and optimal, as long as the branching factor is finite and all actions have cost of at least some positive constant ε
- Time and space complexity: $O(b^{1 + \lfloor C^*/\varepsilon \rfloor})$, where C* is the cost of the optimal solution
- This *can* be much greater than b^d, but it is just equal to b^{d+1} if all step costs are equal

Depth-first search

- Always expands the deepest node in the current fringe
- Goes straight down to the deepest level of the search tree (to nodes that have no successors), then “backs up” to the previous level.
- Queue: really a stack – newly generated nodes are added to the front
- [Example on the board]

Depth-first search

- Very low space requirements for depth-first ***tree*** search: O(bm)
- However:
  - Can make bad choices and end up in really long (even infinite) paths
  - Not optimal (neither ***tree*** nor ***graph*** search)
  - Not complete (***tree*** search)
- Time complexity: $O(b^m)$, where m is the maximum length of any path... but what if there is no maximum?

Depth-limited search

- DFS with a pre-defined depth limit *l*, so nodes at depth *l* are treated as if they have no successors (even though they might)
- [Example on the board]

Depth-limited search

- Solves the infinite-path problems
- But how do we choose l?
  - l < d: DLS is not complete
  - l > d: DLS is not optimal
- Time complexity: $O(b^l)$
- Space complexity: O(bl)
- Good choice if we know the length of solutions

Iterative deepening search

- DLS with l=0, then l=1, then l=2, etc.
- Complete as long as branching factor is finite
- Optimal if path cost is a nondecreasing function of node depth (e.g., if all actions have equal cost)
- Time complexity: $O(b^d)$
- Space complexity: O(bd)
- Disadvantage? Certain nodes are visited multiple times. In practice, not a big problem
- In general: IDS is the preferred method of uninformed search if the search space is large and we do not know the depth of solutions.

Bidirectional search

- Run two simultaneous searches, one working forward from the initial state and one working backward from the goal.
- Stop when they meet in the middle.
- See book for more details.

- Summary of uninformed search techniques: [Figure 3.21 on overhead]

Informed search

- So far: uninformed (blind) search
  - Methods that just went through the state space in a pre-determined, systematic way, looking for goal states
  - Very inefficient for anything but very small problems
- Next: informed search strategies
  - More efficient methods that take advantage of problem-specific knowledge

“Best”-first search

- Nodes are selected for expansion based on some evaluation function, f(n)
- Many of these algorithms rely on a *heuristic function*:

  h(n) = estimated cost of the cheapest path from node n to any goal node

  If n is a goal node, then h(n) = 0.

Heuristic functions

- Example: route-finding in Romania (from book)
- Heuristic:
  - Estimate of the cheapest path from city n to Bucharest = straight-line distance from city n to Bucharest

Greedy best-first search

- Expands the node that appears to be the closest to the goal
- Evaluation function f(n) = h(n), heuristic fn.
- Simplified map of Romania (Figure 3.2)
- Straight-line distances to Bucharest (Fig. 3.22)
- Example search on board (Fig. 3.23)
  - Goes straight to a solution, but is not optimal

Greedy best-first search

- Similar to DFS: prefers to follow a single path all the way to the goal, backtracking when it hits a dead end
- Not optimal, as shown in last example
- Also incomplete: vulnerable to infinite loops, if we do not detect repeated states

[Example: In trying to reach Fagaras…

Iasi → Neamt → Iasi → Neamt → ...]

Greedy best-first search

- Worst-case time and space complexity: $O(b^m)$, where m is the maximum depth of any node
- However, a good heuristic function can greatly reduce complexity

A* search

- Best-known form of best-first search

- Uses the evaluation function f(n) = g(n) + h(n), where

  g(n) = cost to reach node n from the root

  h(n) = estimated cost of cheapest path from n to goal

A* search

- With certain restrictions on the heuristic function, A* search is complete and optimal.
- It is also very efficient.
- (Tree search) A* is optimal if h(n) is an **admissible** heuristic, one that never overestimates the cost to reach the goal.
  - Example from route-finding: straight-line distance (shortest path from A to B is a straight line)
  - [Romania example on board]

Optimality of A*

- (Graph search) A* is optimal if h(n) is a **consistent** heuristic:
  - For every node n and every successor n' of n generated by any action a, the estimated cost of reaching the goal from n is no greater than the step cost of getting to n' plus the estimated cost of reaching the goal from n':

$$h(n) \leq c(n,a,n') + h(n')$$

- See the textbook for a proof that A* is optimal under these conditions.

More on A*

- Note: if the heuristic h(n) is consistent, the nodes expanded by A* using graph search are guaranteed to have nondecreasing f(n) values. [See book for proof.]
  - Therefore, the first goal node selected will be optimal.
- A* is **optimally efficient** for any given heuristic function: no other optimal algorithm is guaranteed to expand fewer nodes than A*.

A*: is there a catch?

- Complete, optimal, optimally efficient: everything seems good!
- One catch: for most problems, the number of nodes expanded is still exponential in the length of the solution.
- Large memory requirements
- It turns out that A* is not practical for many large-scale problems.

Variations on A*

- See book for some variations on A* search:
- Iterative deepening A*
- Recursive best-first search
- Memory-bounded A* + simplified MA*

Heuristic functions

- Example: 8-puzzle
- We want admissible functions: never overestimate the number of steps to the goal
  - h_1 = number of misplaced tiles
  - h_2 = sum of distances of tiles from their goal positions (Manhattan distance)
  - Comparison on overhead [Figure 3.29]

Heuristics for 8-puzzle

- Note that h_2 **dominates** h_1:

  $h_2(n) \geq h_1(n)$ for all nodes n.

- This translates directly into efficiency.

Inventing admissible heuristics

- One approach:
  - Come up with a **relaxed** version of the original search problem, one with fewer restrictions on the actions. (8-puzzle example: ability to move a tile anywhere or the ability to slide a tile in any direction, even onto an occupied square)
  - The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem.

More on heuristics

- More information on designing heuristics and learning heuristics in the book