The UNIVERSITY of VERMONT

Benjamin Fenster
CS-251 (Wu)
15 December 2005

Notes – Searching, Game-Playing, and Planning

I. Romania Example
   a. You're in Arad. Want to get to Bucharest by the fastest distance given existing roads
   b. Goal: Get to Bucharest
   c. Problem Types
      i. Deterministic, fully- observable. "Single-State Problem" We know the distances between cities in advance
      ii. Conformant: Can't see the whole world at a time
      iii. Contingency: Non-deterministic, partially observable (once we reach a new state we'll know more about it)
   d. This particular problem is single-state.

II. Eight-Puzzle Example
   a. Have a three-by-three grid with sliding pieces for each number 1 through 8.
   b. States: Don't care about the number of states but the way to describe them. In this case, where is the empty tile?
   c. Actions: Move a tile
   d. Path cost: 1 move / action
   e. Goal Test: All numbers are in the right place

III. Tree Search Algorithms
   a. Offline, simulated exploration of the state space
   b. States vs. Nodes
      i. A state is just a representation of the physical configuration
      ii. A node constitutes part of the search tree
   c. General strategy: Go through "fringe" nodes looking for a solution match.

IV. Uninformed Search Strategies
   a. Use only the information available in the problem definition
   b. BFS
      i. FIFO queue??
      ii. Complete (if there's a solution, BFS will find it)
      iii. $O(b^{d+1})$
   c. Uniform-Cost
      i. Have a queue ordered by the path cost (like BFS in that sense)
      ii. Complete
   d. DFS
      i. Depth-Limited search
      ii. Set depth limit l
      iii. Treat is a though nodes at depth l have no successors
      iv.
   e. Iterative Deepening (go by levels)
   f. Problem formulation usually requires abstracting away details to get a space that can be explored easily

V. Informed Search
   a. Tree Search Review
      i. Use "fringe" to store the next nodes to explore (adding them based on which algorithm you're following)
      ii. Start at some state
      iii. Take the first node from the fringe and explore it. If not done, expand the fringe by some strategy.
      iv. The strategy is defined by picking the order of nodes expanded
   b. Best-First Search
      i. Use an evaluation function on each node
      ii. Measure *desirability* (How far apart are cities, for example)
      iii. Keep the fringe sorted by decreasing desirability

iv. But: We don't just want to go to the nearest city. We want the city nearest the *destination*. We could keep a database of the straight-line distance between pairs of cities and use that to approximate.

c. Greedy
   i. Evaluation function is an estimate of cost from node to the closest goal.
   ii. Example: Straight-line distance to the destination city
   iii. It's greedy in that you may end up moving away from the goal in order to avoid a roundabout route later, in which case this algorithm doesn't yield the optimal results
   iv. Complete? No, you could get caught in a loop if a city that's closer to the destination turns out to be a dead end. Go there, come back; loop. Without loops (i.e. checking for repeated states), it's complete.
   v. Optimal? No, it may not be optimal.
   vi. Time: $O(b^m)$ where b = number of branches, m = maximum depth
   vii. Space = $O(b^m)$
   viii. Even though we don't explore the whole tree, we have to consider it in the analysis. This makes no sense, but is apparently true anyway.

d. A* Search
   i. The idea: avoid expanding paths that are already expensive.
   ii. h(n) = cost (estimated) from n to the cloest goal (same as before)
   iii. g(n) = cost so far to reach n
   iv. f(n) = estimated total cost through n to the goal
   v. A* uses an *admissible* heuristic, where $h(n) \leq h^*(n)$ where $h^*(n)$ is the *true* cost from n.
   vi. So the heuristic is never worse than the true cost, meaning our gready approach will always work.
   vii. Do allow backtracking now, in that all possible cities are kept on the fringe.
   viii. Complete? Yes, unless there are infinitely many nodes.
   ix. Time: Exponential.
   x. Space: All nodes in memory
   xi. Applied to the 8-puzzle:
      1. $h_1(n)$ = number of misplaced squares
      2. $h_2(n)$ = Manhattan distance (how *far* is each tile from where it needs to be?)
      3. Both are either better than or equivalent to how many moves it will really take, so this is an admissible heuristic
   xii. Relaxed Problem
      1. Admissible heuristics are derived from the *exact* cost of a *relaxed* problem.
      2. In the Budapest example, we solve the problem that allows direct flights directly between cities and calculate the exact cost of that (as opposed to estimating the cost of point-to-point driving)

e. Hill-Climbing
   i. The Analogy: You're climbing a mountain in a heavy fog with amnesia
   ii. Feel around with a stick to find the steepest grade, and go in that direction.
   iii. This finds the local maximum

f. Simulated Annealing
   i. We want to avoid getting stuck on local maxima (what a fun word!) instead of getting to the top of the mountain
   ii. Make some "bad" moves periodically (deliberately go in what you think is the wrong direction).
   iii. Decrease the size and frequency of these over time.

VI. Game Playing

a. An opponent will be making moves you cannot predict
b. Thus this isn't quite like a search problem

c. The solution is a *strategy* specifying a move for each possible opponent move.
d. Types of Games
    i. Deterministic / not (no dice, cards, or elements of chance)
    ii. Perfect / imperfect information (does the opponent hold any information secret?)
    iii. So we have a 4 x 4 grid combining these two factors
        1. Deterministic + Perfect: Chess, checkers, Go, Othello
        2. Chance + Perfect: Monopoly, backgammon
        3. Chance + Imperfect, Poker
e. Perfect Play
    i. We want an agent that will always either win or draw
    ii. In Tic-Tac-Toe, it's possible to build a complete tree of all possible moves, run min-max, and pick the optional move.
    iii. We may have resource limitations that infringe on our ability to build a perfect tree for a more complicated game.
f. Minimax
    i. Choose a move to maximize my utility given an opponent out to minimize my utility
    ii. Complete? Yes, if the tree is finite
    iii. Optimal? Yes, but if the opponent isn't optimal and you haven't stored the entire tree (i.e. you've stored only the part of the tree that assumes the opponent will make an optimal choice) then you'll have to make a random (sub-optimal) choice
    iv. Resource Limits: Don't search all the way to the bottom of the tree. Pick a depth and then estimate the desirability of that state
    v. Evaluation Functions
        1. How exactly do we evaluate desirability without further exploring the tree?
        2. Chess Example:
            a. Give each piece a weight and value based on whether or not it's on the board.
            b. It's worth having the opponent's pieces gone. It's not good to have yours gone.
        3. NB: The exact values don't matter at all. Any monotonic function will work, since all we consider is which are greater or less than others.
g. α-β Pruning
    i. Pruning does not change the final result
    ii. Changing the order in which moves are sorted can change the effectiveness of pruning (can trim more or fewer nodes)
    iii. Called α-β because you're keeping α as your best value, β is opponent's best

VII. Planning
a. Search vs. Planning
    i. Think about having three goals: get milk, get bananas, get an electric drill
    ii. A search algorithm fails miserably since the range of possible activities is great
    iii. We want an efficient (but not necessarily completely optimal) way of solving all goals.
    iv. An after-the-fact heuristic goal test would be inadequate
    v. Planning Systems
        1. Open up action and goal representation to allow selection
        2. Divide and Conquer by splitting the goal into subgoals
        3. Relax the requirement for sequential construction of solutions
b. STRIPS
    i. Stanford Research Institute Problem Solver
    ii. A tool to provide tidily arranged action descriptions
    iii. Abstracts away some important details.
    iv. Aids in creating efficient algorithms
    v. Action: Buy(x). Precondition: At(p) Sells(p, x) Have(x)

vi. Drawn As:

c. Partially Ordered Planning
   i. "POP" Algorithm
   ii. Components:
      1. Start step (initial state)
      2. Finish step (has a goal description as precondition)
      3. Causal links from the outcome (of one step) to the precondition (of another step)
      4. Temporal ordering between step pairs
   iii. The main task is to create causal links
   iv. Definition: Open condition: Preconditions aren't causally linked yet
   v. A plan is complete iff every precondition is achieved. A precondition is achieved iff it is the effect of an earlier step and no possibly intervening step undoes it.
   vi. Process
      1. How are we supposed to generate causal links?
      2. Operators
         a. Add a link from an action to an open condition
         b. Add a step to fulfill some open condition
         c. Order two steps to remove possible conflicts
      3. Backtrack if there's an unachievable open condition or an irresolvable conflict
   vii. Algorithm Sketch

```
Function POP(initial, goal, operators) returns a Plan
      Plan <- MAKE_MINIMAL_PLAN(initial, goal)
      loop do
            If IsSolution(plan) return plan
            S_need, C <- Select Subgoal(plan)
            ChooseOperator(plan, operators, S_need, C)
            ResolveThreats(plan)
      end
```

   viii. Clobbering
      1. Destroys the condition achieved by a causal link. Go(Home) clobbers At(Supermarket)
      2. Demote: put before Go(supermarket)
      3. Promote: put after it
   ix. Properties
      1. Nondeterministic (when picking a subgoal: have no defined way to choose one)
      2. Sound
      3. Complete
      4. Systematic (doesn't repeat anything you've done)