Lecture 3: Overview of Search

CS1571 (9/6/2016)

Summary

In this lecture, we talked about formalizing the process of searching through a state space. We talk about ways to evaluate search strategies.

Review

Scenario

- Three missionaries and three cannibals want to cross a river.
- There is a boat that can hold two people.
- Constraints:
    - The missionaries (if present) cannot be outnumbered by the cannibals on either bank.
    - The boat cannot cross the river by itself.

Formulate this problem so that we may perform goal-oriented search.

- State: # of missionaries and cannibals on each side of the river, and the location of the boat
    - Though in practice, we just need to count # of people on one side of the river (and whether the boat is there)
    - Init state: (3C, 3M, 1B)
    - Goal state: (0C, 0M, 0B)
- Actions:
    - A1: Put two missionaries on the boat (0C, 2M, 1B)
    - A2: Put one missionary on the boat (0C, 1M, 1B)
    - A3: Put two cannibals on the boat (2C, 0M, 1B)
    - A4: Put one cannibal on the boat (1C, 0M, 1B)
    - A5: Put one of each on the boat (1C, 1M, 1B)
- Transition Model: If current state has 1B, subtract action vector from current state to get new state vector. If current state has 0B, add action vector from current state vector to get new vector.
- Goal test: Current state == goal state?
- path cost: # of actions

Searching the State Space

We want to find a solution path in the state space. However, it’s important to note that the state space graph is **implicit** in the sense that we don’t store the whole thing in memory. (It exists in its entirety only as a mathematical idea implied by the start state and the transition model.)

Key point of search: Build a search tree (or graph) to keep track of what parts of the state space has already been explored or should be explored next.

- Important: state space does not equal search tree. The search tree records our exploration over the state space. For example, a search tree might be much larger than the state space due to duplicate states. (Note: we also don't keep the entire search tree in memory.)
    - In other words: a node of a search tree may point to a particular state, and that node's children may point to the corresponding successor states of the parent state, but *that node != that state*.

```
Function tree-search (problem): returns a solution, or fail
    Init frontier list with root node (which contains a pointer to the init state)
    loop do:
        if the frontier list is empty :
            then return fail
        Choose & remove a node from frontier
        if the state that the node points to is a goal state :
            then return solution (i.e., path from current node back to root)
        Expand that node, add children nodes (each points to a valid successor state)
        to frontier
```

Dealing with duplicate states: if we also maintain a list that keeps track of which states we've already seen, our search will be more efficient. In the pseudocode below, the difference from tree-search is highlighted in green.

```
Function graph-search (problem): returns a solution, or fail
    Init frontier list with root node (has init state of problem), and init the explored list
    (note: element type is state, not node)  to empty.
    loop do:
        If frontier is empty :
            then return fail
        Choose & remove a node from frontier
        if the node contains goal state :
            then return solution (i.e., path from current node back to root)
        Add the state in that node to explored
        Expand that node, add those children nodes points to successor states not
        already in explored, or already pointed to by another node in frontier to
        frontier
```

Search Strategies

We will see quite a few strategies. They basically vary in how they handle the two red lines in the pseudocode above. Before we go into the details of different strategies, let's first ask: How can we compare different strategies (how do we know we have a good one?)

- Completeness
- Optimality
- Time complexity
- Space complexity

Mini Big-O Digression

Suppose we have a function whose runtime depends on some input n. We want to know, roughly, how long it would take to run the program in terms of n.

- This is useful for comparing the efficiencies of different algorithms
- We typically care more about big growth differences than absolute differences
    - If program A takes n+constant seconds to run while program B takes just n seconds to run, they are both still O(n) -- when n is big, the constant overhead on A is overshadowed by n.
    - If program A takes n seconds and program B takes 2n seconds, we arguably still don't care; they are both still O(n) -- their time complexity both grow linearly. We prefer O(n) over program C that might take n^2 seconds.
        - Question: if program A takes n^m seconds and program B takes $n^{m/2}$ seconds, are they in the same big O class or not?
    - Some programs don't have a deterministic run time -- say if you are sequentially searching down an array of size n for something; if you're lucky, you might find your target at the first try. For the purpose of time analysis, we care about an upper-bound: this program will take no more than some amount of time expressed in terms of a mathematical function of n.
- Review of the survey question

```
Function print_x(int n):
     while (n >= 1):
          for (i=1; i<=n; i++):
               Print ("x");
          n = n/2;
```

This is a somewhat tricky question because n is halved in each outer loop, but the inner loop depends on (the updated value of) n as well.

Let’s first consider the easier problem of nested loops:

```
Function print_x_new(int n):
     j=n
     while (j >=1 ):
          for (i=1; i<=n; i++):
               Print ("x");
          j--;
```

This prints out x -- n^2 times. Here, we are not changing n, we just have two counters nested within each other.

Now suppose we modify the function slightly:

```
Function print_x_new2(int n):
     j=n
     while (j >=1 ):
          for (i=1; i<=n; i++):
               Print ("x");
          j = j/2;
```

This prints out x -- n lg n times. Here, the outer loop is decreasing at a much faster rate than the inner loop (j=n, n/2, n/4, etc.) This gives us O(n lg n).

Now let’s consider what happens if there is a dependency between inner and outer counters. Below is a slightly nicer version of the original survey question.

```
Function print_x_new2(int n):
     j=n
     while (j >=1 ):
          for (i=1; i<=j; i++):
               Print ("x");
          j = j/2;
```

This time, the inner loop only goes up to j each time. So the first time, it’ll print n x’s, but the next time it’ll only print n/2 x’s, etc. So the total time is:

$$n + n/2 + n/4 + \ldots 1$$

This will sum to a bit less than 2n. So the time complexity is O(n).

NOTE: someone asked me after class whether they can say lg(n) * lg (n) = $(\lg(n))^2$ and whether that simplifies to n. I nodded too quickly while erasing the board, but on my walk to the office, noticed the problem. The answer is actually no. Note that $2^{\lg n} = n$, but $(\lg n)^2$ doesn’t simplify further. (Also, while we’re on the subject of log arithmetics, it’s useful to remember that $\lg (n^2) = 2 \lg n$). Here, you cannot simply multiply the number of times the inner and outer loops were executed because the inner loop is dependent on the outer loop.

Corresponding Readings

- Chapter 3.4 of Russell, S. and Norvig, P. (2010). Artificial Intelligence: A Modern Approach. 3rd ed. Prentice Hall.