Search

CSCI 2400 "Part 3a"
Instructor: David Byrd

Disclaimer: Lecture notes can't and won't cover everything I say in class. You should attend class each day and use these for review or reinforcement.

4 Search Agents / Planning Problem

In the field of AI, when we refer to Search without any modifier, we usually mean Global Search or "Classical" Search algorithms. This is the subfield that includes the algorithms most commonly encountered in lower-level Computer Science courses: breadth-first search and depth-first search. Another way to think of Search is as a class of algorithms or agents that solve the Planning problem.

4.1 Requirements

Planning in AI refers to the task in which an agent will examine all available information to create a plan of action to reach some goal and then execute the entire plan consecutively (without further "thought"). Because the agent does all of its thinking up front, before beginning to act, it cannot respond to changes or unexpected events while executing the plan. Thus for an agent to enjoy the guaranteed success of its plan, there are a few requirements:

- Fully observable state
- Static environment
- Deterministic actions and environment
- Discrete, finite states and actions

That is: the agent must be able to know everything about the problem before it begins acting; the world must not change while the agent is thinking; the world must not change except as a result of the agent's actions *and* those changes must be consistent and predictable; the world must contain a finite number of states; and the agent must have access to a finite set of actions.

If any of these requirements are not met, Search is probably not the right approach to the problem. We will relax many of these constraints as the class progresses.

(*Aside:* It may have occured to you, "But I could just run search again after each action, and then it would be okay if the world changed in an unexpected manner!" If so, you were

right, and this is called *replanning*. Consider this, however: the proposed solution is to run an *exponential time* algorithm *at every step* of the process. Ouch.)

4.2 Definition

There is a customary, formal definition for a problem which contains all required information to allow a search agent to create a successful plan:

- S : set of possible world states
- S_0 : single initial state of the world
- S_F : set of goal states, $|S_F| > 0$
- A : set of permissible actions
- T : matrix of transitions $T(s, a) \rightarrow s'$
- C : matrix of transition costs or "edge weights" (optional)

If T is fully specified, it is often not necessary to define A. It is necessary to define A if a successor function will be used in place of T (see below).

A *state* is a unique configuration of the world. The *state space* is the entire graph defined by the above inputs, with states as nodes, actions/transitions as edges, and optionally costs as edge weights.

The state space is the *only* thing a search agent is ever exploring. Do not think of search in terms of physical locations; you are not searching a "map". Rather you are finding a path (list of edges) that will get you from one state (node) to another state (node). It makes no difference what those nodes or edges "mean" in the real world, if there even is a real world!

4.3 Goal

The goal of a search agent is to return a sequence of *actions* that will transform the world from the initial state S_0 into *any* goal state S_F.

4.4 Example State Space

The classic *Tower of Hanoi* puzzle makes one good example of a state space that does not involve walking around on a map. In it, there are three vertical pegs that can hold discs with holes in the middle. The initial state of the puzzle is that there are three discs on the leftmost peg arranged with the smallest disc on top and the largest disc on the bottom. The goal is to have the discs in the same arrangement on a different peg. There are two rules:

1. You may only move one disc at a time.
2. You may never place a larger disc on top of a smaller disc.

We can easily formulate this as a planning problem for a search agent: each possible arrangement of three discs on three pegs will be a "state", and moving one disc to a new peg will be an "action". There is one initial state described above. There are two goal states in which the discs are in order on either the middle or the right peg.

Here is a partial illustration of the state space:

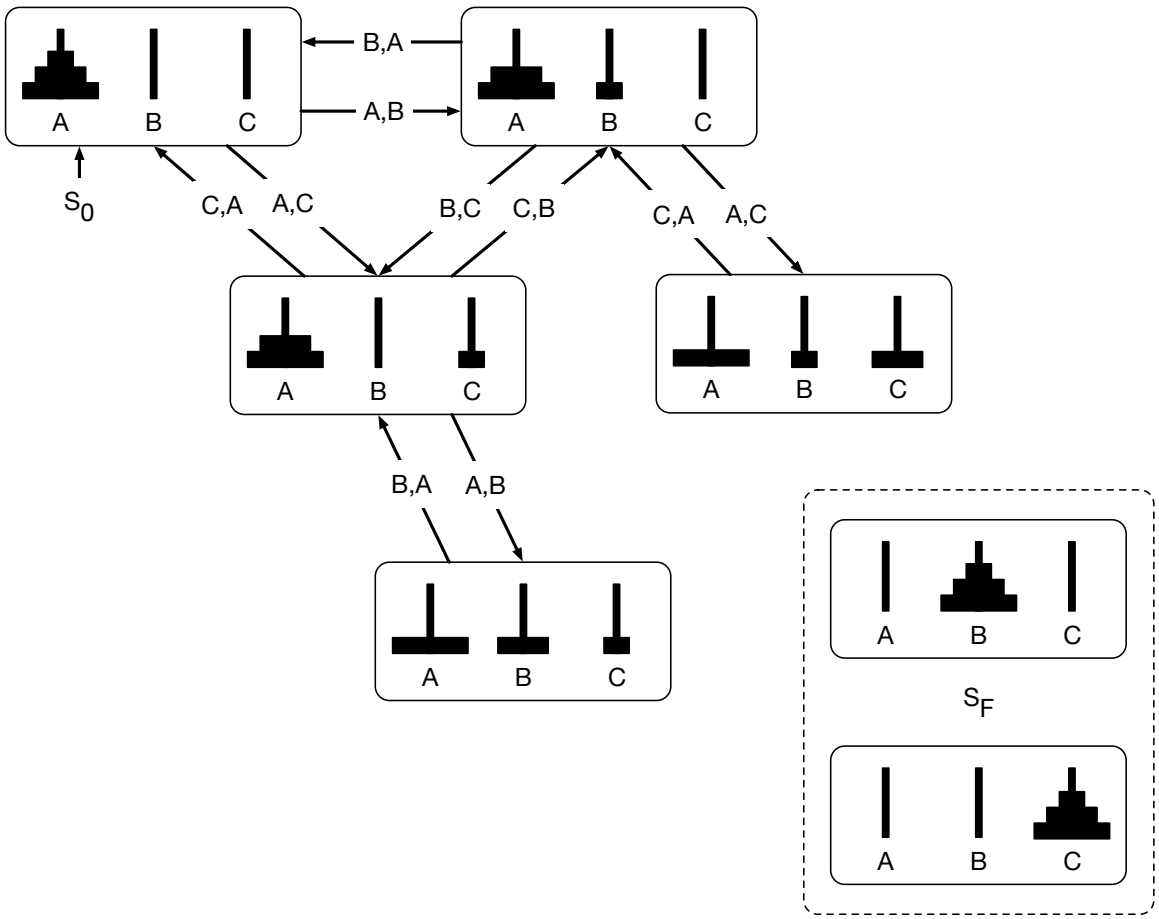

Figure 1: Partial state space for the Tower of Hanoi problem.

The edges in the state space represent *transitions* which result from *actions*. (For some problems, you can simply consider transitions to *be* the actions.) Here the actions are labeled with the source peg and destination peg for moving a single disc. Thus A, B indicates moving the top disc from peg A to peg B. In this problem, notice that all moves are reversible. This may not always be the case.

Remember the "fully observable" requirement for Search and Planning: the entire state space, with all states (nodes) and actions/transitions (arrows), is already known. What is *not* known is a path from the start to the goal. The task for your search algorithm is to return a *sequence of actions* that will transform the world from S_0 in the top left to either of the two S_F states in the bottom right.

One solution to the Tower of Hanoi puzzle for three discs and three pegs is:

$$(A, B), (A, C), (B, C), (A, B), (C, A), (C, B), (A, B) \tag{1}$$

which first stacks the two smaller discs on C, then moves the largest disc to B, then re-stacks the two smaller discs on to B as well.

4.5 Evaluating on the Fly

For some search problems, like the Tower of Hanoi or the Annoying Cat World presented in class, it is reasonable to simply enumerate all possible transitions $T : (s, a) \rightarrow s'$, where taking action a from state s results in new state s'. It is also reasonable to enumerate all possible goal states S_F. Annoying Cat World has 32 transitions, including the *nap* action, and two goal states. Tower of Hanoi with three pegs and three discs has 78 transitions and two goal states.

For other problems, it is clearly *un*reasonable to enumerate T or S_F. In the game of (western) Chess there are on the order of 10^{45} possible states (board configurations) of which 10^{43} are checkmate (goal) states. Instead of attempting to list these, it is common to create a *successors(state)* function and a *goal(state)* function.

A *successors(state)* function receives as input a specific state of the search problem and returns a list of all possible subsequent states that could immediately follow. The *goal(state)* function requires the same input and returns a single boolean value answering the question: "Is this state a goal state?"

Using these two functions permits a rule-based approach, creating pieces of T or S_F on-the-fly and only as needed. In the case of chess, *successors()* will return one successor state for each legal move for each piece currently on the board and belonging to the current player. Again in the case of chess, *goal()* will answer the question: "Is the current player's king in the capture path of an opponent's piece? If so, is there any single legal action the current player can take to change this condition?" (If the answers are yes and no respectively, the game is over, because this is the definition of a checkmate.)

4.6 Design Considerations

It is important to consider your goal when devising or applying a search algorithm.

- *Minimum requirements:* Are you trying to find just *any* solution? Do you require the solution with the least actions taken? The solution with the lowest cost? Are there any useful "hints" (i.e. domain knowledge) you can give the agent?

- *Evaluation:* What kind of time and space complexity is okay? Is memory at a premium? Do you require *completeness* (guaranteed to find a solution if one exists)? What about *optimality* (guaranteed to find the best solution for some definition of best)?

- *Computation time vs Action cost:* There is a trade-off between spending time devising a better plan (computing) vs the cost of executing a potentially sub-optimal plan (edge weights). If it takes an extra week of computation to find a path that is shorter by

two days of travel time, it would have been better to leave a week ago and take the sub-optimal path – you'd be there already!

- *Thinking vs doing:* When designing your algorithm, remember that your agent is *not* acting or moving whlie performing the search. It is *planning*, that is *imagining* actions and their possible consequences. Only after it has a complete plan (i.e. a solution to the search problem) does it begin acting. If you ever find yourself thinking "but what happens if my agent walks into a dead end?" then you are probably not thinking about search correctly.

5 Uninformed Search

The first set of methods to consider are collectively called brute-force search or *uninformed search*. By uninformed, we mean the algorithm has available to it only that information in the standard Search problem definition: S, S_0, S_F, A, T, C. The approach is simple: try every path until one works.

There are benefits to uninformed search:

- Broadly applicable to many problems
- Many types of uninformed search from which to select
- Agent may find unexpected solution due to *lack* of domain knowledge
- Agent need not know anything other than what actions it can perform
- Possible to search from/to different start/goal states without changing any other part of the problem or agent

Of course there are also drawbacks:

- Can be extremely slow and memory intensive even for modest problems
- Performs very badly in non-deterministic environments, which is most of the real world

Why would you ever use it then? You can think of uninformed search as the "default" when you can't find a better approach to your particular problem.

5.1 Random Search

The most basic uninformed search algorithm would simply select an action at random at every time step. The pseudocode for this is wonderfully simple (see Algorithm 1), and even random search has a few good points. Its memory requirements are basically none and it is very straightforward to code! However, it is easy to see that infinite loops are possible, meaning that the algorithm is neither complete nor optimal, and the worst-case run-time ("Big-O") is actually infinity. This is not great.

How could we improve on random search to address these problems? We could remember where we have been and avoid revisiting the same state – that would solve our completeness problem and the infinite loops. We could also choose where to explore next in some systematic manner. If we do that cleverly, we can solve the optimality problem also!

Algorithm 1 The simplest search: choose random actions.

function RANDOMSEARCH(S, A, T, S_0, S_F)
  $s \leftarrow S_0$
  while $s \notin S_F$ **do**
    $a \leftarrow$ random_choice(A)
    $s' \leftarrow T[s, a]$
    $s = s'$
  end while
end function

5.2 Breadth First Search

To improve upon random search, we have said we should cleverly select the order of states to visit and to avoid revisiting states. We do this by introducing two data structures:

- **Open list** Holds states to which we have discovered a path, but that we have not yet explored (or "expanded" or "visited"). Sometimes called the *visit* or *to_visit* or *frontier* list.
- **Closed list** Holds states we have already explored. Sometimes called the *visited* or *done* list.

Note that these structures need not be actual lists. Commonly, *open* will be a stack, queue, or priority queue, and *closed* will be a set. [*Aside:* In addition to the final path discovered, we will often consider the length of the closed list at the end of the algorithm as a measure of efficiency. That is: "How many nodes did the algorithm have to explore before it found a solution?" Finding the same correct path with less exploration (calculation or time spent thinking) would obviously be preferable!]

By selecting a FIFO Queue (first in, first out) as the open list, we obtain the *breadth-first search* algorithm or *BFS*. Please see the "part 3b" lecture notes entitled "Generic Search Algorithm" for pseudocode and a walkthrough of running it on an example problem.

What behavior do we create by using an open queue? When discovering new states, the algorithm will place them at the end of the line for visitation. In other words, it will not visit the newly discovered states until it has explored *all* previously discovered states. This engenders a very important property: *breadth-first search explores states in order of increasing "hops" from the start state.* It will explore all states one action from the start, then explore all states two actions from the start, and so on.

Breadth-first search is *complete* because it can potentially search all states eventually, provided the state space is finite. It is also *optimal* for an unweighted graph, because it explores all states at depth n before any state at depth $n + 1$. Its space complexity is not ideal, as it must keep *all* discovered states in memory due to the way it "round robins" back to different branches of the search tree. Its time complexity is exponential like all search algorithms. In this case, $O(b^d)$ where b is the branching factor (average number of successor states) and d the depth of the solution.

5.3 Depth First Search

By instead selecting a LIFO Stack (last in, first out) as the *open list*, we obtain the *depth-first search* algorithm or *DFS*. Again, see the "part 3b" lecture notes for pseudocode and a walkthrough.

Using an open stack instead of a queue gives the algorithm different behavior. Now when discovering new states, the algorithm will place them at the *front* of the line for visitation. That is, it will explore all newly discovered states before returning to any previously discovered states.

This gives DFS an interesting property: *depth-first search tries to stay the maximum possible distance away from the start state at all times*. The intuition here is that the goal is probably far from the start, and so getting as far from the start as possible, as fast as possible, is the quickest way to find the goal. Thus DFS will only backtrack to other paths when its current path terminates with a dead-end (or loops to already-visited states).

Depth-first search is *complete* because, like breadth-first, it can potentially search all states, provided the state space is finite. It is *not optimal*. It simply tries to find *any* path as quickly as possible. Its space complexity is better than BFS because it can prune/delete portions of the state space after reaching a dead end. Its time complexity is worse than BFS in the *worst* case, because it can *miss the goal*: $O(b^m)$ where b is the branching factor and m the maximum depth expanded. However its *average* time complexity is better than BFS because it can find a goal at level N before exploring all states at levels $< N$, which BFS cannot do.

5.4 Uniform Cost Search

If we change the *open list* to use a Priority Queue which accepts $(cost, object)$ tuples and always dequeues the lowest cost object, and we use "total path cost so far" as that cost, we obtain *uniform cost search* or *UCS*. It is easy to modify the generic search algorithm to also handle the UCS case.

Per the above description, UCS searches all lower-cost paths before searching any higher-cost path. The only drawback to UCS is this: it is not necessarily searching *closer to the goal*, just searching *less far from the start*. There is still a lot of room for improvement in terms of finding the least cost path *more quickly*.

Uniform cost search is complete because again, it can potentially search all states. It is *also optimal* because it searches paths strictly in order of increasing total path cost. When a goal state is dequeued for the first time, it *must* represent the cheapest path, because any lower-cost path would have been dequeued previously. Its time and space complexity are exponential in terms of the branching factor, the cost of the best path, and the minimum edge weight. (See the book for more.)

There are multiple ways to recover the solution path for UCS and the previous algorithms. You can build a search tree as you go by placing the initial state at the root and adding child nodes for each successor state discovered, then walking the tree backwards from the goal (a leaf node) to the start (root node) using parent pointers. Alternatively, even for UCS, you can use a method similar my BFS/DFS pseudocode by realizing the priority queue allows *any* Python object as its data payload. Thus you could, for example, enqueue states

like so: `(cost so far, (state, [path taken to reach state]))`. Then when a state is dequeued, you always know the path you took to get there *and* the cost for that exact path.

Next up: Generic Search Algorithm.

Acknowledgements and thanks to Professors Mark Riedl and Jim Rehg of the Georgia Tech School of Interactive Computing.