Search Me Quickly

Prof. Robert C. Berwick

Agenda
1. Administrivia
2. Intelligence = “Knowledge Based System” (KBS) = Knowledge + Search
3. How to search optimally: B&B, A*

1. Administrivia

2. What is intelligence? Ans: Good, optimal searching

Search is composed of 5 main features:
• DATA STRUCTURE (aka a Problem Representation) – mapping from problem into a graph, from graph into search tree
  1. The START state.
  2. The GOAL state (or states)
  3. Given an arbitrary state, the SUCCESSORS of that state (or a successor function that computes this
  4. A queue to keep track of how we are searching through the graph
• CONTROL STRUCTURE
  5. Search STRATEGY that determines ORDER in which we search the queue.

Search arises in many AI contexts, both as finding a goal, and in the more obvious one of finding a path (through a problem space or a real space). Let’s look at some examples of the goal-finding sort first. These are perhaps the most ‘natural’ to visualize (as in the online demo), but actually the least frequently used in AI.

To talk about moving through a space, it is natural to introduce the notions of graphs and trees. A directed graph is like a set of one-way streets – a finite set of vertices (nodes) and links (edges) connecting the nodes. An undirected graph - two-way streets. A cycle (loop) in a graph is a sequence of edges that starts and ends at the same node. A tree is a directed graph without cycles. We can turn graph search problems into tree search problems by (1) replacing each undirected links by 2 directed links (going in opposite directions) and (2) avoiding cycles on any path.

3. Representing problems as search problems

The key lesson for this recitation: searching is not just about maps! It applies whenever we can abstract a problem as a choice amongst alternatives, a set of states of the problem (the problem or state space), a special start state, a goal state, and a way to get from one state to another (the next/valid or legal moves) Let’s do a few examples so you can see how this conversion works.

1. Farmer, goose, grain (Startup firms and Oligopolies).
A farmer wants to move a fox, a goose, grain, and the farmer across a river. The boat is so tiny that it can hold only one of the possessions across on any trip. Also, an unattended fox will eat a goose, and an unattended goose will eat the grain. What should the farmer do?
  • How do we represent the States? (one state) – tells us how to represent the state space
  • How do we represent the start state?
  • How do we represent the goal state?
  • How do we represent legal moves (transitions) from state to state?
Next we want to turn this *graph* into a *tree*. Then we want to try out our ‘blind’ search methods on this tree. Example: Oligopolies and Startups

### 4. Search me – the framework

A **partial path** $N$ is a path from the start node to some node $X$, e.g., $(S A B X)$. The **head** of a partial path is the most recent node of the path, e.g. $X$.

- Let $Q$ be a list of partial paths, e.g. $(S A B X) (S A B C) \ldots$.
- Let $S$ be the start node and $G$ the Goal node.

**Search framework pseudocode**

1. Initialize $Q$ with partial path $(S)$ as only entry; set Visited = $S$  
   *(Note change from slide pseudocode)*
2. If $Q$ is empty, fail. Else, pick some partial path $N$ from $Q$.
3. If head($N$) = $G$ (goal), return $N$ (we’ve reached the goal); $N$ is the successful path from $S$ to $G$.
4. Else Remove $N$ from $Q$.
5. Find all the descendants of head($N$) not in Visited and create all the one-step extensions of $N$ to each descendant.
6. Add to $Q$ all the extended paths; add descendants of head($N$) to Visited.
7. Go to step 2.

#### Note 1:
There are two choices remaining:
- Where to pick elements $N$ from $Q$ in step 2.
- Where to add the new path extensions to $Q$ in step 6.

#### Note 2:
The Winston book does not use a Visited list

**Note 3:** We could stop at step 6 if the extended paths at that point reach the goal, but this won’t work for optimal searches, so we use the more general test in Step 3.

**Implementing Depth-First Search**

Our control choices: (1) Pick $N$ from the *first* element of the $Q$; (2) Add new path extensions to the *front* of $Q$.

Let’s try this out on our example. Here are the first 3 iterations:

1. Initial step: partial path $N = (1)$, Visited set= 1
2. $Q$ is not empty, so pick FIRST of partial paths; this is 1
3. Not at goal, so
4. Remove 1 from $Q$.
5. Find all descendants of head($N$), = 1 not in Visited = 2; & create all 1-step extensions, (2, 1);
6. Add this path to $Q$; add head of this path to Visited. So Visited= 2, 1 and $Q=$(2, 1)
7. Go to Step 2.

2. $Q$ is not empty, so pick FIRST of the partials, i.e., (2, 1).
3. Not at goal, so
4. Remove (2, 1) from $Q$.
5. Find all descendants of (2,1) not in visited list =3 & create all 1-step extensions, (3, 2, 1)
6. Add this path to $Q$; add head of this path to Visited. So Visited= 3,2, 1 and $Q=$(3, 2, 1)
7. Go to step 2.

2. $Q$ is not empty, so pick FIRST of the partials, i.e., (3, 2, 1).
3. Not at goal, so
4. Remove (3, 2, 1) from $Q$.
5. Find all descendants of (3, 2, 1) not in visited list =4, 5, 6 & create all 1-step extensions, (4, 3, 2, 1); (5, 3, 2, 1) and (8, 3, 2, 1)
6. Add these paths to $Q$; add heads of these paths to Visited. So Visited= 8,5,4,3,2, 1 and $Q=$(4 3, 2, 1), (5, 3, 2,1) and (8, 3, 2, 1)
7. Go to step 2
Implementing Breadth-first search

Our control choices: (1) Pick $N$ from the first element of the $Q$; (2) Add new path extensions to the end of $Q$.

Now, you try this one on your own- follow the slides