Blind Search (Draft)

The first topic of this course is **search algorithms**. Unsurprisingly, these algorithms are designed to solve **search problems**. Examples of search problems include:

- *path planning*: given a graph where vertices are locations and edges are distances between them, find a path from a start/source node to a goal/destination node (i.e., think Google maps)
- *theorem proving*: given a set of axioms, some rules of inference, and a select formula, find a proof of the formula, if one exists
- *game playing*: given the rules of game (e.g., chess), find an optimal policy, meaning an optimal mapping from states to moves
- *class scheduling*: given degree requirements, course offerings and times, and student preferences, find a(n optimal) schedule

Examples of search algorithms include:

1. blind search: depth-first, breadth-first, and iterative deepening.
2. heuristic search, in which the search is guided by an heuristic evaluation function: greedy search, beam search, A^* and IDA^* search
3. adversarial search for game playing: minimax algorithm, $\alpha\beta$ pruning

We will cover these algorithms in turn during the next few lectures.

1 Shakey the Robot

Shakey the robot was one of the earliest robots able to perceive, reason, and act in its environment. Shakey was built to autonomously navigate in its surroundings. Specifically, it was initialized in some start state, and then asked to find its way to a goal state, circumventing any obstacles encountered along the way. Fast forward 75 years. Today's self-driving cars are intended to do much of the same.[1]

Like Shakey and self-driving cars, navigation is a problem that you all experience daily, when going from one class to another, or from your dorm room to the dining hall. One of the central themes of this course is **problem formulation**. More often than not, the challenge of solving a problem lies in formulating it. If you can formalize a problem as an instance of a problem for which solutions are known (e.g., search), then solving it becomes as simple as applying those algorithms (e.g., BFS or DFS).

[1]One big difference between Shakey and today's self-driving cars is that the obstacles in Shakey's environment were static, whereas self-driving cars must circumvent moving obstacles with "minds of their own."

2 Basic Search Problem

The first class of problems we encounter in this course are basic search problems. A *basic search problem* is a 4-tuple $\langle X, S, G, \mathcal{T} \rangle$, where

- X is a set of states
- $S \subseteq X$ is a nonempty set of *start* states
- $G \subseteq X$ is a nonempty set of *goal* states
- $\mathcal{T} : X \rightrightarrows X$ is a state transition model, mapping a state $x \in X$ to a set $\mathcal{T}(x)$ of successor states

Navigation as Search Given this general problem definition, we can formulate the problem of navigating around Brown's campus as follows:

- X: the set of all intersections on a campus map
- S: the origin of your journey
- G: your desired destination
- $\mathcal{T}$: a mapping from a state to all states "reachable" from that state (e.g., intersections one block away, accounting for dead ends).

The **solution** to a basic search problem is a *path* $P = \{x_1, \ldots, x_k\}$, meaning a sequence of states, beginning at some start state $x_1 \in S$ and ending at some goal state $x_k \in G$, with $x_{i+1} \in T(x_i)$ for all $i \in \{2, \ldots, k-1\}$. (N.B., sometimes we are only interested in the goal state, and not the path to the goal state.)

Note that there may be many solutions to a basic search problem: i.e., many paths from a start state to a goal. For example, there are multiple routes from your dorm room to this classroom. Often, we are interested in optimal solutions. An **optimal** solution to a basic search problem is a shortest path, meaning one that traverses the fewest number of states en route from the start state to its goal.

Knuth's Conjecture In 1964, Donald Knuth conjectured that, starting from the number 4, you can reach any positive integer by applying a sequence of factorial, square root, and floor operators [1].

Here's one way to reach 5:

$$\lfloor(\sqrt{\sqrt{\sqrt{\sqrt{\sqrt{\sqrt{(4!)!}}}}}})\rfloor = 5$$

How would you formulate Knuth's Conjecture as search?

- X: the set of positive numbers ($\mathbb{R}^+$)
- S: 4
- G: $\{5, 6, 7, \ldots\}$
- $\mathcal{T}$: $\mathcal{T}(x) \rightarrow \{x!, \sqrt{x}, \lfloor x \rfloor\}$

N.B. Without proof to the contrary, it seems safe to assume the state space in this search problem is infinite.

Assumptions In these notes, we restrict our attention to search problems on **locally finite** graphs. In locally finite graphs, the degree of every vertex is finite (i.e., no vertex has an infinite number of neighbors).

That said, X itself need *not* be finite. X may be infinite, and there may be search paths of infinite length.

Since X can, in general, be very large—consider, for example, searching for an optimal policy in the game of chess—we usually cannot store all of X in memory. On the contrary, we conduct search by visiting states one-by-one, and keeping track of the set of states we are currently searching (the **open** set; also called the **fringe**), and, when memory constraints allow, which states we have already searched (the **closed** set).

3 Generic Search Algorithm

The blind search algorithms we discuss in this lecture are all instantiations of the following generic search algorithm. This algorithm maintains an open set of states, initialized to the set of start states, which it considers in turn as potential goal states. For each such state, it either deems it a goal and returns, or it discards it and inserts its successors into the open set to also be considered in turn as potential goal states.

When a state is deleted from the open set and tested to see if it is a goal or not, we say that state has been **visited**. When a state's successors are added to the open set, we say that state has been **expanded**.

Search	
Inputs	search problem $\langle X, S, G, \mathcal{T} \rangle$
Output	(path to) goal node
Initialize	$O = S$ is the list of open nodes

`while` (O is not empty) `do`

1. delete some node $n \in O$
2. if $n \in G$, return (path to) n
3. insert $\mathcal{T}(n)$ to O

`fail`

Table 1: Generic Search Algorithm.

This algorithm is designed for trees, not graphs. It does not maintain a closed set, because there is exactly one path to every node in a tree.

Exercise Modify this algorithm so that it searches graphs, not just trees, by keeping track of a closed set as well as an open one, and being sure not to revisit already visited states. You can assume there is sufficient memory in which to store all visited states, and sufficient time to test membership in this set.

4 Evaluation Criteria

The following criteria are used to evaluate search algorithms:

- time complexity: how much time is required to find solution?

- space complexity: how much space is required to find solution?
- completeness: is the algorithm guaranteed to find solution, if one exists?
- optimality: does it find an optimal solution?

The analyses and algorithms presented in this lecture depend on the assumption that the search space is a tree of (possibly infinite) depth d with finite branching factor b. The depth of a tree is defined as the maximum number of hops from the root node to any other node in the tree. The branching factor of a tree is defined as the maximum number of children of any node in the tree.

5 Breadth-First Search

The main idea of breadth-first search (BFS) is to visit all the states at depth i before visiting those at depth $i+1$. BFS is implemented by storing the open set as a *queue*, and accessing its entries in a first-in-first-out (FIFO) fashion.

BFS	
Inputs	search problem $\langle X, S, G, \mathcal{T} \rangle$
Output	(path to) goal node
Initialize	$O = S$ is the list of open nodes

`while` (O is not empty) `do`

1. delete *first* node $n \in O$
2. if $n \in G$, return (path to) n
3. append $\mathcal{T}(n)$ to *back* of O

`fail`

Table 2: Breadth-First Search.

BFS is complete: it is guaranteed to find a solution, if one exists. Moreover, BFS is optimal: it always finds a goal state of minimal distance from the start state.

In the worst case, BFS expands every node, which takes time as follows:

$$1 + b + b^2 + \ldots + b^d = \sum_{i=0}^{d} b^i = \frac{b^{d+1} - 1}{b - 1} = O(b^d)$$

In terms of space, BFS maintains a queue of potentially all the states at each depth: at depth d the length of this queue is b^d. The space complexity of BFS is thus exponential in d.

6 Depth-First Search

The main idea of depth-first search (DFS) is to always visit a state in the open set of maximal depth, among all open states. DFS is implemented by storing the open set as a *stack*, and accessing its entries in a last-in-first-out fashion (LIFO).

DFS($X, S, G, \mathcal{T}$)	
Inputs	search problem
Output	(path to) goal node
Initialize	$O = S$ is the list of open nodes

`while` (O is not empty) `do`

1. delete *first* node $n \in O$
2. if $n \in G$, return (path to) n
3. prepend $\mathcal{T}(n)$ to *front* of O

`fail`

Table 3: Depth-First Search.

Like BFS, the time complexity of DFS is $O(b^d)$. It is exponential in d because DFS searches the entire tree in the worst case. The space complexity of DFS, however, is linear in d: at most b nodes are stored at each of the d depths: i.e., DFS is $O(bd)$.

DFS is neither complete nor optimal. What if we were to try using DFS to solve Knuth's conjecture, and it applied the factorial operation repeatedly? The state would only ever increase and we'd never reach our goal! (Incomplete)

What about a robot who is planning a path to move 1 meter to the east? If DFS started searching west first, the would robot travel all the way around the world before reaching its goal! (Suboptimal)

7 Iterative Deepening

Iterative deepening (ID) is an optimal search algorithm with the space requirements of DFS—it requires memory linear in d—and the performance properties of BFS—it is complete and optimal. The main idea of iterative deepening is to repeatedly search in depth-first fashion, over subgraphs of depth 0, depth 1, depth 2, and so on, until a goal is found.

ID is optimal and complete: it is guaranteed to find a goal if one exists, and an optimal goal when there are multiple solutions.

Like DFS and BFS, the time complexity of ID is $O(b^d)$: in the worst case it is exponential in d. ID expands nodes at depth 0 $d+1$ times, at depth 1 d times, ..., and at depth d 1 time. Thus, the total time required is given by:

$$1 + \underbrace{1+b}_{d=1} + \underbrace{1+b+b^2}_{\text{depth } 2} + \ldots + \underbrace{1+b+b^2+\ldots+b^d}_{\text{depth } d} = \sum_{c=0}^{d} \sum_{i=0}^{c} b^i = \sum_{c=0}^{d} \frac{b^{c+1}-1}{b-1} = O(b^d)$$

Finally, ID iteratively performs depth-first searches; thus, its space complexity is that of DFS, namely $O(bd)$.

ID($X, S, G, \mathcal{T}$)

Inputs	search problem
Output	(path to) goal node
Initialize	$c = 0$ is the cutoff depth
	$O = S$ is the list of open nodes

`while` *goal not found* `do`

1. `while` (O is not empty) `do`
   (a) delete *first* node $n \in O$
   (b) if $n \in G$, return (path to) goal n
   (c) if depth(n) $\leq c$
       i. prepend $\mathcal{T}(n)$ to *front* of O
2. increment c, $O = S$

Table 4: Iterative Deepening.

8 Summary

Criteria	DFS	BFS	ID
Time	$O(b^d)$	$O(b^d)$	$O(b^d)$
Space	$O(bd)$	$O(b^d)$	$O(bd)$
Completeness	NO	YES	YES
Optimality	NO	YES	YES

1. BFS is preferred if the branching factor is small
2. ID is preferred if the depth is large: the search space is deep (possibly infinite), particularly if goals are known to be shallow
3. DFS is preferred if the maximum depth of the goal nodes is known: if this depth is n, modify DFS to search only to depth n

References

[1] Donald E. Knuth. Representing numbers using only one 4. *Mathematics Magazine*, 37(5):308–310, 1964.