Search Algorithms

Search is an approach to finding desired data in the presence of a collection of other data. The question is which subgroups of data to examine first. The purpose of sorting and other data ordering techniques such as array indexing, sorted lists, binary search trees, hash tables, and priority queues, is to make search more efficient.

Data Structures

dictionary:	a collection of searchable records
record:	information associated with a search key
key:	data field of search

Types of Search

sequential: look through the data array until the search object is found
average cost = N/2 comparisons
can use any of the sorting algorithms to improve this search algorithm

binary: divide and conquer, analogous to quick-sort, assumes ordered keys
maximum cost = lnN+1 comparisons
variant is **interpolation** search, guess location of target as entry rather than middle of dictionary.

binary search tree: build a balanced tree so that comparisons are always lnN in efficiency
maintenance of the balanced tree structure is expensive.
This technique trades search efficiency for data sorting efficiency.
worst case is when files are in order (or reverse order)

Techniques for balancing tree search

2-3-4 trees: have multiple keys associated with each node which separate children into organized groups
worst case = lnN+1 nodes

red-black trees: use an extra bit to encode 2-3-4 nodes into binary nodes
extra key is “red” or “black”, black meaning “smaller than” or “greater than” the node key and red meaning “split 3 nodes into 2”.

Generic Search

When searching an organized data structure such as a list or a tree, there are a variety of strategic approaches, falling into three broad categories:
brute-force, *iterative*, and *heuristic*.

Brute force techniques	depth-first breadth-first
Iterative techniques	iterative deepening iterative broadening
Heuristic techniques	hill-climbing beam search best-first adversarial search

For *depth-first* search, the tree is descended until a leaf node is reached, then the search backs up to the nearest node with has not been explored. The trouble with depth-first search is that some descents may go on for a long time, while most of the rest of tree remains unexplored. One way to manage the exploration is to collect a list of children nodes to be explored, putting the newest ones on the top of a priority queue.

For *breadth-first* search, the tree is descended one level at a time. All nodes at each level are explored before going on to the next deeper layer. The trouble with breadth-first search is that the desired leaf nodes may be the last to be explored. One way to manage breadth-first search is to collect a list of children nodes to be explored, putting the newest ones (for the next level) on the bottom of a priority queue.

Iterative techniques are designed to avoid the problems with pure depth and breadth-first searches. The idea is to elect to go to a particular depth, and then change from depth-first to breadth-first. Similarly, a search could elect to completely explore only those nodes with a limited fanout, or breadth, before converting to depth-first. One way to manage iterative approaches is to collect a list of children nodes to be explored, putting the desired nodes on the top of the priority queue, and the postponed nodes on the bottom of the queue.

Heuristic approaches depend upon an evaluation function computed for each node. Nodes to be explored are placed on a priority queue in order of their value. For *hill-climbing*, the evaluation function estimates the remaining distance to the bottom of the tree, in an attempt to improve depth-first search. Analogously, *beam search* selects the best nodes at the same level, attempting to limit breadth-first search. *Best-first* selects the node with the best evaluation, regardless of depth or breadth.

Finally, *adversarial search* applies to situations in which search turns are taken by two competing opponents (like in checkers and chess). The evaluation function estimates the best strategy, taking into account opposing choices as well as positive moves. One way to think of adversarial search is that a tree is being searched, but every other move is directed by an anti-search, which attempts to make the search fail.

Graph Search

Techniques for dealing with graph data structures generalize those for tree data structures, since trees are a special case of graphs.