Search Techniques

Sample Problem: *n* Queens [Traditional]

Place *n* queens on an *n x n* chess board so that no queen is attacked by another queen.

Depth First Search (DFS)

The most obvious solution to code is to add queens recursively to the board one by one, trying all possible queen placements. It is easy to exploit the fact that there must be exactly one queen in each column: at each step in the recursion, just choose where in the current column to put the queen.

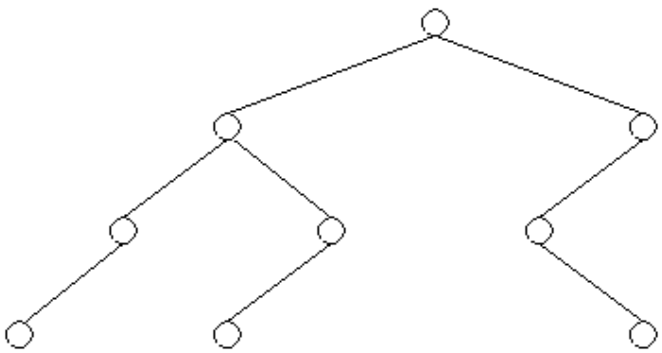
```
1 search(col)
2     if filled all columns
3         print solution and exit

4   for each row
5       if board(row, col) is not attacked
6           place queen at (row, col)
7           search(col+1)
8           remove queen at (row, col)
```

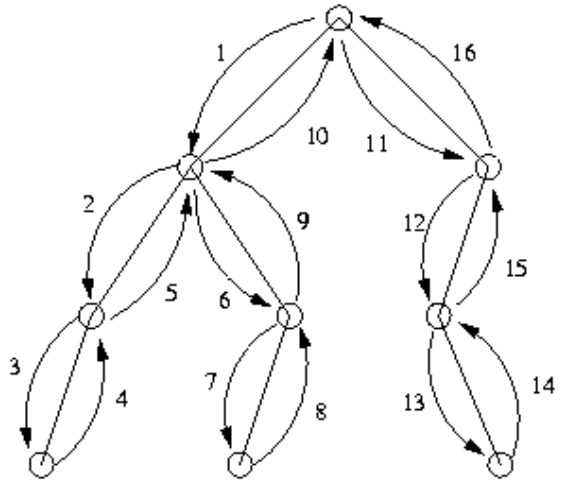
Calling `search(0)` begins the search. This runs quickly, since there are relatively few choices at each step: once a few queens are on the board, the number of non-attacked squares goes down dramatically.

This is an example of *depth first search*, because the algorithm iterates down to the bottom of the search tree as quickly as possible: once *k* queens are placed on the board, boards with even more queens are examined before examining other possible boards with only *k* queens. This is okay but sometimes it is desirable to find the simplest solutions before trying more complex ones.

Depth first search checks each node in a search tree for some property. The search tree might look like this:



The algorithm searches the tree by going down as far as possible and then backtracking when necessary, making a sort of outline of the tree as the nodes are visited. Pictorially, the tree is traversed in the following manner:



Complexity

Suppose there are *d* decisions that must be made. (In this case *d=n*, the number of columns we must fill.) Suppose further that there are C choices for each decision. (In this case *c=n* also, since any of the rows could potentially be chosen.) Then the entire search will take time proportional to c^d, i.e., an exponential amount of time. This scheme requires little space, though: since it only keeps track of as many decisions as there are to make, it requires only O(*d*) space.

Sample Problem: Knight Cover [Traditional]

Place as few knights as possible on an *n x n* chess board so that every square is attacked. A knight is not considered to attack the square on which it sits.

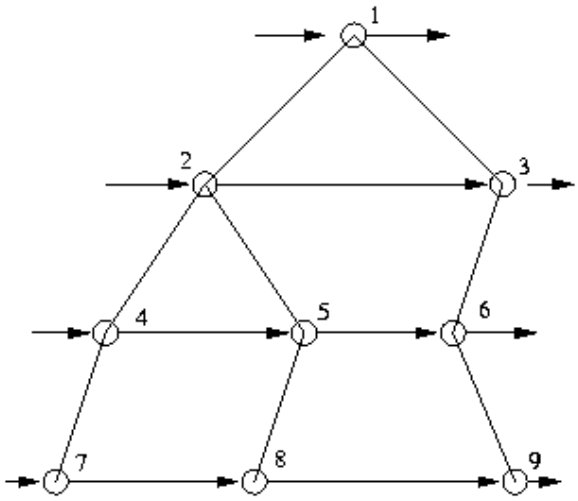
Breadth First Search (BFS)

In this case, it is desirable to try all the solutions with only *k* knights before moving on to those with *k+1* knights. This is called **breadth first search**. The usual way to implement breadth first search is to use a queue of states:

```
1 process(state)
2     for each possible next state from this one
3         enqueue next state

4 search()
5     enqueue initial state
6     while !empty(queue)
7         state = get state from queue
8         process(state)
```

This is called breadth first search because it searches an entire row (the breadth) of the search tree before moving on to the next row. For the search tree used previously, breadth first search visits the nodes in this order:



It first visits the top node, then all the nodes at level 1, then all at level 2, and so on.

Complexity

Whereas depth first search required space proportional to the number of decisions (there were *n* columns to fill in the *n* queens problem, so it took O(*n*) space), breadth first search requires space exponential in the number of choices.

If there are *c* choices at each decision and *k* decisions have been made, then there are c^k possible boards that will be in the queue for the next round. This difference is quite significant given the space restrictions of some programming environments.

[Some details on why c^k*: Consider the nodes in the recursion tree. The zeroeth level has 1 nodes. The first level has c nodes. The second level has* c^2 *nodes, etc. Thus, the total number of nodes on the k-th level is* c^k.]

Depth First with Iterative Deepening (ID)

An alternative to breadth first search is *iterative deepening*. Instead of a single breadth first search, run D depth first searches in succession, each search allowed to go one row deeper than the previous one. That is, the first search is allowed only to explore to row 1, the second to row 2, and so on. This ``simulates'' a breadth first search at a cost in time but a savings in space.

```
 1 truncated_dfsearch(hnextpos, depth)
 2     if board is covered
 3         print solution and exit

 4     if depth == 0
 5         return

 6     for i from nextpos to n*n
 7         put knight at i
 8         truncated_dfsearch(i+1, depth-1)
 9         remove knight at i

10 dfid_search
11     for depth = 0 to max_depth
12        truncated_dfsearch(0, depth)
```

Complexity

The space complexity of iterative deepening is just the space complexity of depth first search: O(*n*). The time complexity, on the other hand, is more complex. Each truncated depth first search stopped at depth *k* takes c^k time. Then if *d* is the maximum number of decisions, depth first iterative deepening takes $c^0 + c^1 + c^2 + ... + c^d$ time.

If $c = 2$, then this sum is $c^{d+1} - 1$, about twice the time that breadth first search would have taken.

When c is more than two (i.e., when there are many choices for each decision), the sum is even less: iterative deepening cannot take more than twice the time that breadth first search would have taken, assuming there are always at least two choices for each decision.

Which to Use?

Once you've identified a problem as a search problem, it's important to choose the right type of search. Here are some things to think about.

In a Nutshell

Search	Time	Space	When to use
DFS	$O(c^k)$	$O(k)$	Must search tree anyway, know the level the answers are on, or you aren't looking for the shallowest number.
BFS	$O(c^d)$	$O(c^d)$	Know answers are very near top of tree, or want shallowest answer.
DFS+ID	$O(c^d)$	$O(d)$	Want to do BFS, don't have enough space, and can spare the time.

d is the depth of the answer
k is the depth searched
$d <= k$

Remember the ordering properties of each search. If the program needs to produce a list sorted shortest solution first (in terms of distance from the root node), use breadth first search or iterative deepening. For other orders, depth first search is the right strategy.

If there isn't enough time to search the entire tree, use the algorithm that is more likely to find the answer. If the answer is expected to be in one of the rows of nodes closest to the root, use breadth first search or iterative deepening. Conversely, if the answer is expected to be in the leaves, use the simpler depth first search.

Be sure to keep space constraints in mind. If memory is insufficient to maintain the queue for breadth first search but time is available, use iterative deepening.

Sample Problems

Superprime Rib [USACO 1994 Final Round, adapted]

A number is called superprime if it is prime and every number obtained by chopping some number of digits from the right side of the decimal expansion is prime. For example, 233 is a superprime, because 233, 23, and 2 are all prime. Print a list of all the superprime numbers of length n, for $n <= 9$. The number 1 is not a prime.

For this problem, use depth first search, since all the answers are going to be at the nth level (the bottom level) of the search.

Betsy's Tour [USACO 1995 Qualifying Round]

A square township has been partitioned into n^2 square plots. The Farm is located in the upper left plot and the Market is located in the lower left plot. Betsy takes a tour of the township going from Farm to Market by walking through every plot exactly once. Write a program that will count how many unique tours Betsy can take in going from Farm to Market for any value of $n <= 6$.

Since the number of solutions is required, the entire tree must be searched, even if one solution is found quickly. So it doesn't matter from a time perspective whether DFS or BFS is used. Since DFS takes less space, it is the search of choice for this problem.

Udder Travel [USACO 1995 Final Round; Piele]

The Udder Travel cow transport company is based at farm A and owns one cow truck which it uses to pick up and deliver cows between seven farms A, B, C, D, E, F, and G. The (commutative) distances between farms are given by an array. Every morning, Udder Travel has to decide, given a set of cow moving orders, the order in which to pick up and deliver cows to minimize the total distance traveled. Here are the rules:

- The truck always starts from the headquarters at farm A and must return there when the day's deliveries are done.
- The truck can only carry one cow at a time.
- The orders are given as pairs of letters denoting where a cow is to be picked up followed by where the cow is to be delivered.

Your job is to write a program that, given any set of orders, determines the shortest route that takes care of all the deliveries, while starting and ending at farm A.

Since all possibilities must be tried in order to ensure the best one is found, the entire tree must be searched, which takes the same amount of time whether using DFS or BFS. Since DFS uses much less space and is conceptually easier to implement, use that.

Desert Crossing [1992 IOI, adapted]

A group of desert nomads is working together to try to get one of their group across the desert. Each nomad can carry a certain number of quarts of water, and each nomad drinks a certain amount of water per day, but the nomads can carry differing amounts of water, and require different amounts of water. Given the carrying capacity and drinking requirements of each nomad, find the minimum number of nomads required to get at least one nomad across the desert.

All the nomads must survive, so every nomad that starts out must either turn back at some point, carrying enough water to get back to the start or must reach the other side of the desert. However, if a nomad has surplus water when it is time to turn back, the water can be given to their friends, if their friends can carry it.

Analysis: This problem actually is two recursive problems: one recursing on the set of nomads to use, the other on when the nomads turn back. Depth-first search with iterative deepening works well here to determine the nomads required, trying first if any one can make it across by themselves, then seeing if two work together to get across, etc.

Addition Chains

An addition chain is a sequence of integers such that the first number is 1, and every subsequent number is the sum of some two (not necessarily unique) numbers that appear in the list before it. For example, 1 2 3 5 is such a chain, as 2 is 1+1, 3 is 2+1, and 5 is 2+3. Find the minimum length chain that ends with a given number.

Analysis: Depth-first search with iterative deepening works well here, as DFS has a tendency to first try 1 2 3 4 5 ... n, which is really bad and the queue grows too large very quickly for BFS.