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.

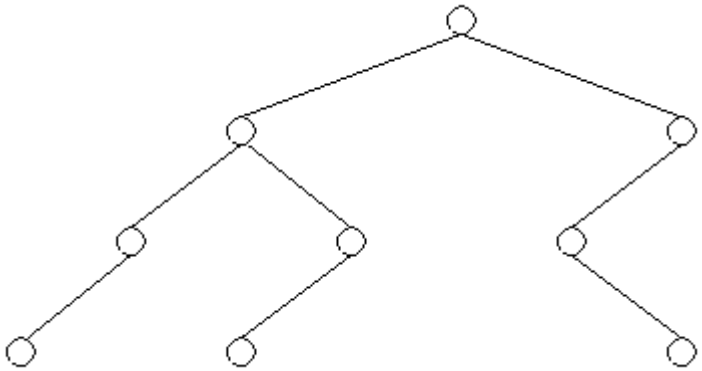
```
1  search(col)
2          if  filled  all  columns
3                  print  solution  and  exit

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)
```

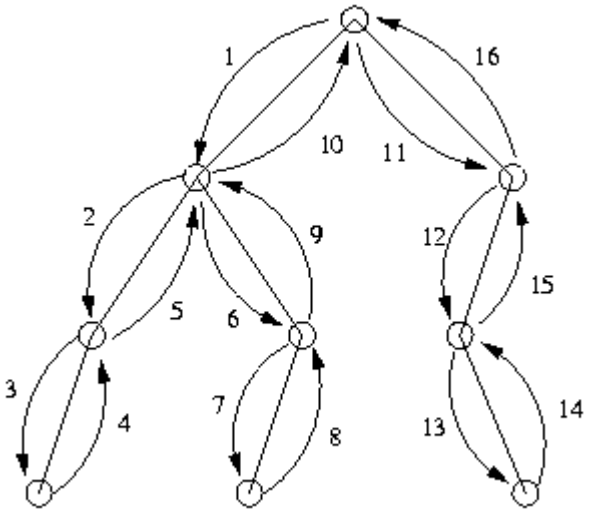
Calling `search(0)` begins the search. This runs quickly, since there are relatively few choices at each step: once a few queens are on the board, the number of non-attacked squares goes down dramatically.

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.

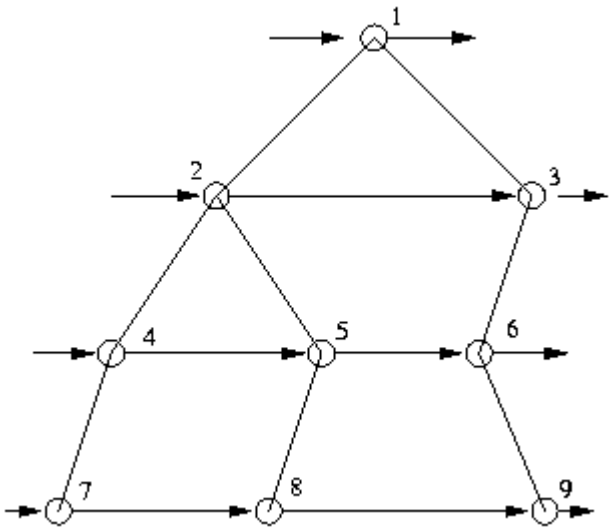
Breadth First Search (BFS)

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 *n*th 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.

译 by Lucky Crazy

样例：n 皇后问题 [经典问题]

将 n 个皇后摆放在一个 $n \times n$ 的棋盘上，使得每一个皇后都无法攻击到其他皇后。

深度优先搜索 (DFS)

显而易见，最直接的方法就是把皇后一个一个地摆放在棋盘上的合法位置上，枚举所有可能寻找可行解。可以发现在棋盘上的每一行（或列）都存在且仅存在一个皇后，所以，在递归的每一步中，只需要在当前行(或列）中寻找合法格，选择其中一个格摆放一个皇后。

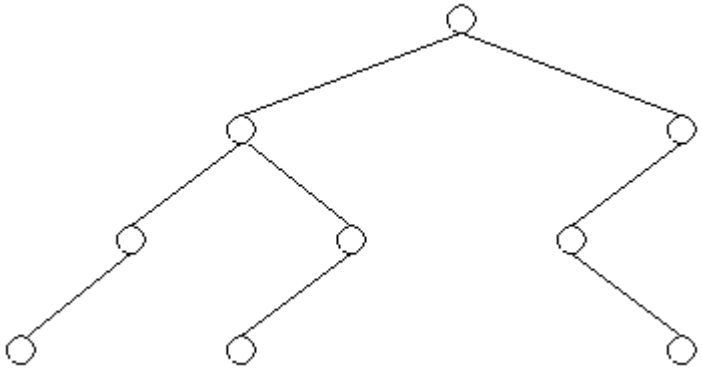
```
1  search(col)
2          if  filled  all  columns
3                  print  solution  and  exit

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)
```

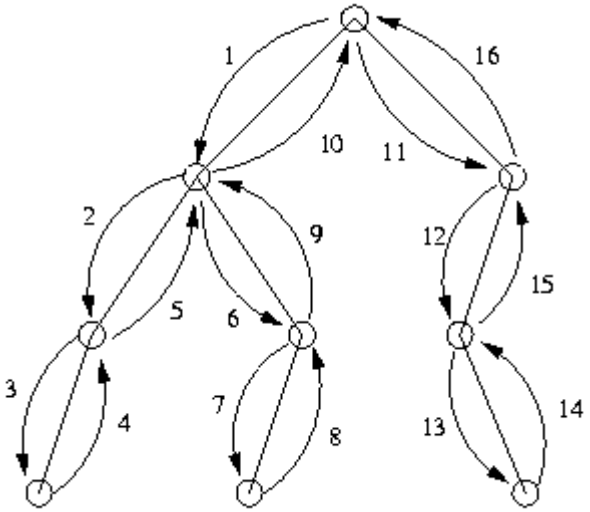
从 search(0)开始搜索，由于在每一步中可选择的节点较少，该方法可以较快地求解：当一定数量的皇后被摆放在棋盘上后那些不会被攻击到的节点的数量将迅速减少。

这是**深度优先搜索**的一个经典例题，该算法总是能尽可能快地抵达搜索树的底层：当 k 个皇后被摆放到棋盘上时，可以马上确定如何在棋盘上摆放下一个皇后，而不需要去考虑其他的顺序摆放皇后可能造成的影响（如当前情况是否为最优情况），该方法有时可以在找到可行解之前避免复杂的计算，这是十分值得的。

深度优先搜索具有一些特性，考虑下图的搜索树：



该算法用逐步加层的方法搜索并且适当时回溯，在每一个已被访问过的节点上标号，以便下次回溯时不会再次被搜索。绘画般地，搜索树将以如下顺序被遍历：



复杂度：

假设搜索树有 d 层（在样例中 $d=n$ ，即棋盘的列数）。再假设每一个节点都有 c 个子节点（在样例中，同样 $c=n$ ，即棋盘的行数，但最后一层没有子节点，除外）。那么整个搜索花去的时间将与 c^d 成正比，是指数级的。但是其需要的空间较小，除了搜索树以外，仅需要用一个栈存储当前路径所经过的节点，其空间复杂度为 $O(d)$ 。

样例：骑士覆盖问题[经典问题]

在一个 $n \times n$ 的棋盘中摆放尽量少的骑士，使得棋盘的每一格都会被至少一个骑士攻击到。但骑士无法攻击到它自己站的位置.

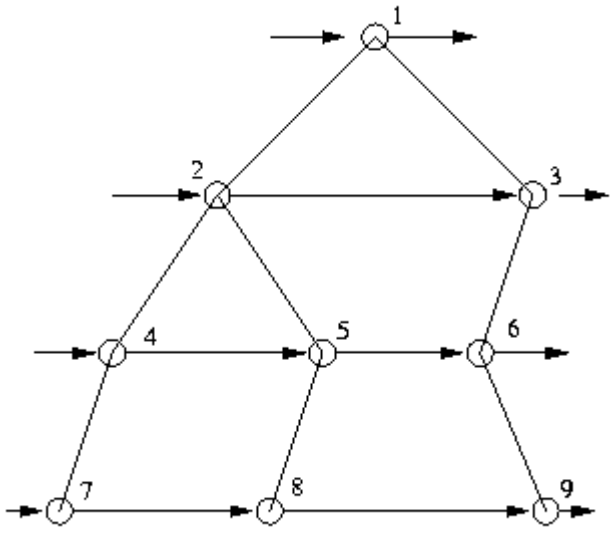
广度优先搜索 (BFS)

在这里，最好的方法莫过于先确定 k 个骑士能否实现后再尝试 $k+1$ 个骑士，这就叫**广度优先搜索**。通常，广度优先搜索需用队列来帮助实现。

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

search()
        enqueue initial state
        while !empty(queue)
                state = get state from queue
                process(state)
```

广度优先搜索得名于它的实现方式：每次都先将搜索树某一层的所有节点全部访问完毕后再访问下一层，再利用先前的那颗搜索树，广度优先搜索以如下顺序遍历：



首先访问根节点，而后是搜索树第一层的所有节点，之后第二层、第三层……以此类推。

复杂度：

广度优先搜索所需的空间与深度优先搜索所需的不同（ n 皇后问题的空间复杂度为 $O(n)$ ），广度优先搜索的空间复杂取决于每层的节点数。如果搜索树有 k 层，每个节点有 c 个子节点，那么最后将可能有 c^k 个数据被存入队列，这个复杂度无疑是巨大的。所以在使用广度优先搜索时，应小心处理空间问题。

迭代加深搜索（ID）

广度优先搜索可以用迭代加深搜索代替。迭代加深搜索实质是限定下界的深度优先搜索，即首先允许深度优先搜索搜索 k 层搜索树，若没有发现可行解，再将 $k+1$ 后再进行一次以上步骤，直到搜索到可行解。这个“模仿广度优先搜索”搜索法比起广搜是牺牲了时间，但节约了空间。

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

        if  depth  ==  0
                return

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

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

复杂度：

ID 时间复杂度与 DFS 的时间复杂度（$O(n)$）不同，另一方面，它要更复杂，某次 DFS 若限界 k 层，则耗时 c^k。若搜索树共有 d 层，则一个完整的 DFS-ID 将耗时 $c^0 + c^1 + c^2 + \ldots + c^d$。如果 c = 2，那么式子的和是 $c^{d+1} - 1$，大约是同效 BFS 的两倍。当 c > 2 时（子节点的数目大于 2），差距将变小：ID 的时间消耗不可能大于同效 BFS 的两倍。 所以，但数据较大时，ID-DFS 并不比 BFS 慢，但是空间复杂度却与 DFS 相同，比 BFS 小得多。

算法选择：

当你已经知道某题是一道搜索题，那么选择正确的搜索方式是十分重要的。下面给你一些选择的依据。

简表：

搜索方式	时间	空间	使用情况
DFS	$O(c^k)$	$O(k)$	必须遍历整棵树，要求出解的深度或经的过节点，或者你并不需要解的深度最小。

BFS	$O(c^d)$	$O(c^d)$	了解到解十分靠近根节点，或者你需要解的深度最小。
DFS+ID	$O(c^d)$	$O(d)$	需要做 BFS，但没有足够的空间，时间却很充裕。

d ：解的深度
k ：搜索树的深度
$d <= k$

记住每一种搜索法的优势。如果要求求出最接近根节点的解，则使用 BFD 或 ID。而如果是其他的情况，DFS 是一种很好的搜索方式。如果没有足够的时间搜出所有解。那么使用的方法应最容易搜出可行解，如果答案可能离根节点较近，那么就应该用 BFS 或 ID，相反，如果答案离根节点较远，那么使用 DFS 较好。还有，请仔细小心空间的限制。如果空间不足以让你使用 BFS，那么请使用迭代加深吧！

类似问题：

超级质数肋骨[USACO 1994 决赛]

一个数，如果它从右到左的一位、两位直到 N 位（N 是）所构成的数都是质数，那么它就称为超级质数。例如：233、23、2 都是质数，所以 233 是超级质数。要求：读入一个数 N（N <= 9），编程输出所有有 N 位的超级质数。

这题应使用 DFS，因为每一个答案都有 N 层（最底层），所以 DFS 是最好的.

Betsy 的旅行 [USACO 1995 资格赛]

一个正方形的小镇被分成 NxN ($2 <= N <= 6$) 个小方格，Besty 要从左上角的方格到达左下角的方格，并且经过每一次方格都恰好经过一次。编程对于给定的 N 计算出 Besty 能采用的所有的旅行路线的数目。

这题要求求出解的数量，所以整颗搜索树都必须被遍历，这就与可行解的位置与出解速度没有关系了。所以这题可以使用 BFS 或 DFS，又因为 DFS 需要的空间较少，所以 DFS 是较好的.

奶牛运输 [USACO 1995 决赛]

奶牛运输公司拥有一辆运输卡车与牧场 A ，运输公司的任务是在 A，B，C，D，E，F 和 G 七个农场之间运输奶牛。每两个农场之间的路程（可以用 floyed 改变）已给出。每天早晨，运输公司都必须确定一条运输路线，使得运输的总距离最短。但必须遵守以下规则：

- 农场 A 是公司的基地。每天的运输都必须从这开始并且在这结束。
- 卡车任何时刻都只能最多承载一头奶牛。
- 给出的数据是奶牛的原先位置与运输结束后奶牛所在的位置。

而你的任务是在上述规则内寻找最短的运输路线。

在发现最优解时必须比较所有可行解，所以必须遍历整棵搜索树。所以，可以用 DFS 解题.

横越沙漠［1992 IOI］

一群沙漠探险者正尝试着让他们中的一部分人横渡沙漠。每一个探险者可以携带一定数量的水，同时他们每天也要喝掉一定量的水。已知每个探险者可携带的水量与每天需要的水量都不同。给出每个探险者能携带的水量与需要的水量与横渡沙漠所需的天数，请编程求出最多能有几个人能横渡沙漠。所有探险者都必须存活，所以有些探险者在中途必须返回，返回时也必须携带足够的水。当然，如果一个探险者返回时有剩余的水（除去返回所需的水以外），他可以把剩余的水送给他的一个同伴，如果它的同伴可以携带的话。

这题可以分成两个小问题，一个是如何为探险者分组，另一个是某些探险者应在何处返回。所以使用 ID-DFS 是可行的。首先尝试每一个探险者能否独自横渡，然后是两个人配合，三个人配合。直到结束。