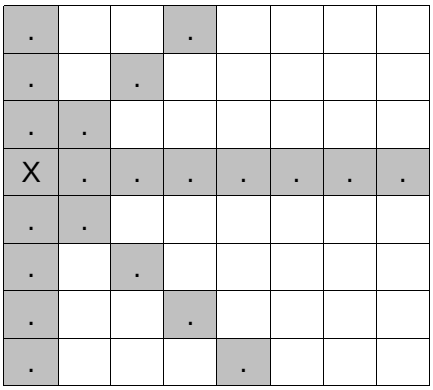
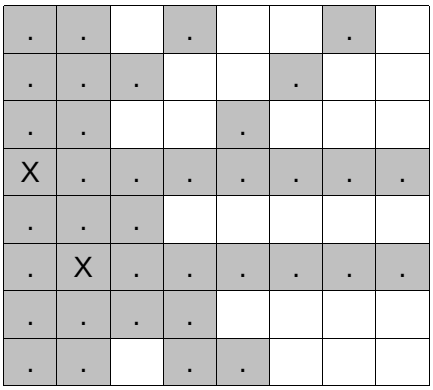
Branch and Bound for 8-Queens

Branch and Bound is a technique that is widely used for speeding up a backtracking algorithm. Some people consider it to be a part of backtracking, so you may hear the term "backtracking" used in the sense of "backtracking with branch-and-bound".

The idea is simple. We have a recursive algorithm that tries to build a solution part by part, and when it gets into a dead end, then it has either built a solution or it needs to go back (backtrack) and try picking different values for some of the parts. We check whether the solution we have built is a valid solution only at the deepest level of recursion – when we have all parts picked out. Branch-and-bound says that sometimes, we can notice that after building only a partial solution there is no need to go any deeper because we are heading into a dead end.

This is best illustrated by an example. Consider the 8-queens algorithm from last time. The idea was: pick a position for the 0^{th} queen (8 possibilities). Then pick a a position for the 1^{st} queen (7 possibilities). But wait, we don't really have 7 possibilities. If we, for example, have placed queen 0 on row 3, then queen 1 can not be placed on rows 2, 3 or 4 – those cells are under attack from queen 0. As far as we can see just from the first 2 queens, queen 1 has only 5 valid positions.





After picking a position for queen 1, queen 2 has even fewer options (3 in the example above) because most of the cells in its column are under attack from the first 2 queens. If we ensure that we never place a queen on an attacked square, then we can dramatically reduce the branching factor of the recursive function. Instead of doing 8*7*6*...*1 recursive calls, we can do a lot fewer, and as a nice bonus, we no longer need to verify whether we have a valid configuration at the end. It is guaranteed to be valid because from the very beginning we make sure we never place a queen on an attacked square.

Now we need to figure out an efficient way of keeping track of which cells are under attack. We could simply keep an 8-by-8 boolean matrix just like in the pictures above and update it each time we placed a queen, but that requires linear time to update, and it is unclear how it can be updated when we remove a queen (in order to backtrack). There is a better way. We have to ensure 4 things:

1. No two queens share a column.
2. No two queens share a row.
3. No two queens share a / diagonal.
4. No two queens share a \ diagonal.

Number 1 is automatic because of the way we store the solution. Number 2 is done with the used[] array. For 3 and 4, we can do the same thing – keep two arrays that will tell us which diagonals are occupied.

Consider the following trick. If we take a cell with coordinates (r,c), what are the values of (r+c) and (r-c+7)? They are plotted in the following two tables (rows are numbered from top to bottom; columns go from left to right; both are 0-indexed).

0	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8
2	3	4	5	6	7	8	9
3	4	5	6	7	8	9	10
4	5	6	7	8	9	10	11
5	6	7	8	9	10	11	12
6	7	8	9	10	11	12	13
7	8	9	10	11	12	13	14

r+c

7	6	5	4	3	2	1	0
8	7	6	5	4	3	2	1
9	8	7	6	5	4	3	2
10	9	8	7	6	5	4	3
11	10	9	8	7	6	5	4
12	11	10	9	8	7	6	5
13	12	11	10	9	8	7	6
14	13	12	11	10	9	8	7

r - c+7

Let's call (r+c) the "slash code" of the cell (r,c) and (r-c+7) – the "backslash code". Then two queens share a /-diagonal if they have the same slash code, and they share a \-diagonal if they have the same backslash code. These codes are integers between 0 and 14, so we can deal with them in exactly the same way as we have dealt with used rows, and we can perform updates in O(1) time.

Now before we place queen i on row j, we first check whether row j is used (look in the used[] array). Then we check whether slash code (j+i) or backslash code (j-i+7) are used (look in the 2 new arrays). If yes, then we have to try a different location for queen i. If not, then we mark the row and the two diagonals as used and recurse on queen i+1. After the recursive call returns and before we try another position for queen i, we need to reset the row, slash code and backslash code as unused again, like in the code from the previous notes.

Once we have placed all 8 queens, we increment the global counter because we have just found a new valid configuration – no additional check is required.

The "+7" in the backslash code is there to ensure that the codes are never negative because we are using the codes as indices in an array. If the size of the board were n instead of 8, then we would need to add (n-1) instead of 7.

This implementation will be able to solve a 13-by-13 board in under a second. The key idea behind branch-and-bound is to spend some extra work during each recursive call in order to make as few recursive calls as possible. The deeper the recursion goes, the slower the program, so if we can ensure that it only goes as deep as is absolutely necessary, then we get 2 benefits out of it. First, the number of recursive calls can be reduced. Second, we can avoid the validity check at the end.

Printing All Paths from s to t

Here is another application of branch-and-bound. Suppose we are given an unweighted graph and two vertices, s and t. We are interested in printing out all the possible paths from s to t. There could be a lot of them. In a complete graph (when we have all the possible $\frac{n(n-1)}{2}$ edges), there are more than (n-2)! different paths from s to t. So if we are given a complete graph with 20 vertices, then there is no way to solve this problem, no matter which algorithm we use – there is simply too much output

to produce. But if the amount of output (the number of possible paths) is not too big, then we can solve this with backtracking. We are going to generate all the possible paths recursively, and we will use branch-and-bound to ensure that we never explore a path that leads to a dead end. This way, the amount of work we have to do will be proportional to the size of the output. This is a very useful property for an algorithm to have. It ensures that if the output size is not too big, then the algorithm will work. And if the output size is too big, then no algorithm will work anyway.

First of all, we need a way of dividing a solution into parts. Each solution is a path from s to t. A logical way of partitioning it is to consider the individual vertices on the path. We know that the first vertex must be s, and if our last vertex so far is u, then the next vertex could be any vertex v, as long as there is an edge from u to v and we have not gone through v yet. We can represent a path by an array of vertices, and we will build it recursively. Here is an implementation of backtracking (without branch-and-bound).

Example 1:

```
int path[128], seen[128], graph[128][128], n, s, t;

// path[i] is the vertex we are at — decide where to go next
void boogie( int i ) {
    int u = path[i];

    if( u == t ) {
        // reached t: print the path
        printf( "%d", s );
        for( int j = 1; j <= i; j++ ) printf( " %d", path[j] );
        printf( "\n" );
        return;
    }

    // find a neighbour to visit next
    for( int v = 0; v < n; v++ ) {
        if( !graph[u][v] ) continue;           // need to have edge (u,v)
        if( seen[v] ) continue;                // can not repeate vertices

        path[i + 1] = v;

        seen[v] = true;                        // mark v as visited
        boogie( i + 1 );                       // compute the rest of the path
        seen[v] = false;                       // backtrack (un-visit v)
    }
}

int main() {
    // ...initialize graph[][], n, s and t

    memset( seen, 0, sizeof( seen ) );
    path[0] = s;
    seen[s] = true;
    boogie( 0 );

    return 0;
}
```

path[] stores our path so far (it starts at s and ends at u). If the last vertex we visit is t, stop and print the path. Otherwise, pick the next vertex to visit. Technically, the seen[] array is unnecessary because seen[x] is true if and only if x is on the path and appears in the first i+1 entries of path[]. It does, however, speed up the algorithm because without seen[], we would have to scan path[] every time to see which vertices we have already visited.

Note that if we remove the last line of boogie() ("seen[v] = false;"), then this would be DFS. This is why recursive backtracking of this form is sometimes referred to as "forgetting DFS" – because whenever it goes up the DFS tree, it forgets having visited the children and will probably end up visiting them again. That single line changes the running time from linear to exponential!

So that's the brute force solution. It works fine for small graphs and graphs that have few paths, but it fails miserably in the following case. Consider a graph with n=20 vertices, s=0, t=1. There is an edge from vertex 0 to vertex 1. There is another edge from 0 to 2, and then there is an edge between each pair of vertices (u,v) that are both bigger than or equal to 2. Clearly, there is only one path from s to t – you have to take the (0,1) edge, and that's it. The algorithm will find that path very quickly because 1 is the first vertex we visit when starting from 0. But then the code will follow the edge (0,2) and spend forever exploring that huge part of the graph with no hope of ever getting to the destination. That's unfortunate – if only the program could be smart enough to know that once it has reached vertex 2, there is no chance of ever getting to t.

What we need in boogie() is a way of checking whether it is possible to get from our current position, u, to vertex t without using any of the seen vertices. This is a simple find-a-path-from-x-to-y problem that we can solve with either BFS or DFS. x is our current vertex and y is the destination, t. This is what we will do – during each call to boogie(), the first thing we will do is run, say, BFS from u and see if we can reach t. If not, return. Otherwise, proceed as before. We have to make sure that the BFS is only allowed to use vertices that do not appear in the seen[] array (we can't re-visit vertices).

This will cost us $O(n^2)$ time per iteration of boogie() to run BFS. It seems like a lot because it is multiplied by the number of recursive calls, but it speeds up the algorithm dramatically. For the example above, there will only be 3 calls to boogie() made: one on vertex 0, one on vertex 1 and one on vertex 2. In general, the new version will take O((size of output)*n*(n*n)) time because for each path that we print, there will be at most n recursive calls made, and each one will execute one BFS. If there are too many paths, then this algorithm is slow, but so is any other algorithm that solves this problem. The best you can possibly hope for when solving any problem is O(size of output) because we have to output the answer.