Here is some code to print nQueens solution boards (rotated by 90° to simplify things):

```haskell
showQueens :: Int -> String
showQueens size [] = 
  showQueens size (q:qs) = (concat $ replicate (q-1) ".") ++ "Q" ++ 
  (concat $ replicate (size-q) ".") ++ "Q" 
  showQueens size qs

nQueensBoard :: Int -> String
nQueensBoard n = showQueens n $ nQueens n
```

The Haskell Platform allows you to even solve the 22-queens problem within about 15 seconds:

```haskell
main = putStrLn $ nQueensBoard 22
```

---

A General Graph-Searching Algorithm

We just saw how backtracking can lead to very efficient search when we continuously check whether we are stuck in a dead end. We would not be able to solve the 22-Queens problem with brute force, i.e., breadth-first search. That would create a tree with at least 1 + 22 + 22^2 + ... + 22^{21} + 1 = (22^{22} - 1)/21 + 1 = 1.6 \times 10^{28} nodes before we could possibly find a solution. Even if we could create a billion nodes per second and store them somehow, it would still take us over 500 billion years to build the complete tree.

---

A General Backtracking Algorithm

Here is some code to print nQueens solution boards (rotated by 90° to simplify things):

```haskell
fQueens :: Int -> [Int] -> Bool
fQueens row plan 1 = dist = True
  isOK (p:ps) (row:rowdist, rowdist) = False
    if elem p (row, row-dist) then False
    else isOK ps (dist + 1)
```

We can write our general backtracking function:

```haskell
backtrack :: (a->[a] -> Bool) -> [[a]] -> [a]
backtrack fCheck levels = bt [] levels
  where bt plan [] = null
    bt plan (b:bs) = if not (fCheck $ b) || null result
      then bt plan (bs:ls)
      else result
        where result = bt (b:plan) ls
```

Use it to solve n-queens problem:

```haskell
nQueens n = backtrack fQueens n (repeat [1..n])
```

---

A General Backtracking Algorithm

In some cases, we may even have an idea which decisions during search are most likely to get us to the goal quickly. Then we should take advantage of this information. Unfortunately, neither depth-first nor breadth-first search are flexible enough to allow this. Instead, we need a general graph-searching algorithm. It can be tuned towards depth-first or breadth-first search but also allows intermediate variants that can be guided by task-specific search heuristics.

Let’s see how this magic works...
A General Graph-Searching Algorithm

1. Create search tree $T_r$ consisting only of the start node (root) $n_0$. Put $n_0$ on an ordered list called OPEN.
2. Create empty list called CLOSED.
3. If OPEN is empty, exit with failure.
4. Move the first node on OPEN, called $n$, to CLOSED.
5. If $n$ is a goal node, exit with success; the solution is the path from $n$ to $n_0$ along the edges in $T_r$.
6. Expand node $n$ by generating a set $M$ of successors and connect them to $n$. Also put the members of $M$ on OPEN.
7. Reorder the list OPEN according to a specific rule.
8. Go to step 3.

Rules for the Graph-Searching Algorithm

What happens if we simply append new nodes to the end of OPEN?
- The algorithm turns into breadth-first search.

What happens if we add new nodes at the beginning of OPEN?
- The algorithm turns into depth-first search.

What happens if we sort the nodes according to a heuristic evaluation function $f'$?
- The algorithm turns into best-first search.

Best-First (Heuristic) Search

- We can often improve search efficiency by implementing a heuristic evaluation function $f'$ that estimates which node is the best one to expand in a given situation.
- Small values of $f'$ indicate the best nodes.
- In the general graph-searching algorithm, sort the nodes on the OPEN list in ascending order with regard to their $f'$-values.
- This way, the supposedly “best” nodes are expanded first.

Example: The Eight-puzzle

The task is to transform the initial puzzle into the goal state by shifting numbers up, down, left or right.

<table>
<thead>
<tr>
<th>2 8 3</th>
<th>1 6 4</th>
<th>7 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1 2 3</th>
<th>8 4 7 6 5</th>
</tr>
</thead>
</table>

Best-First (Heuristic) Search

How can we find an appropriate evaluation function?
- Heuristic evaluation functions have to be “custom-made” for individual problems.
- Often, it is easy to see what a reasonable (but usually not optimal) function could look like.
- For the Eight-puzzle, it seems like a good idea to expand those nodes first that lead to configurations similar to the goal state.
- In our first attempt, we will simply define $f'$ as the number of digits in the puzzle that are not in the correct (goal) position yet.

Best-First (Heuristic) Search

To further unnecessary expansions
Best-First (Heuristic) Search

- You see that with this simple function $f$ the algorithm is still likely to perform unnecessarily many steps.
- We should increase the “cost” of searches in greater depth to make the algorithm avoid them.
- To achieve this, we simply add the current depth of a node to its $f$-value.
- Now $f(n) = \text{depth of } n \text{ in the tree} + \text{number of digits in the puzzle in incorrect position}$. 
- Let us try again!

Best-First (Heuristic) Search

Using the improved function $f'$, the best-first algorithm discovers the solution much more efficiently than it did before.

To formalize things, we now have a function $f'(n) = g'(n) + h'(n)$, where $g'(n)$ is an estimate of the depth (or cost) of $n$, and $h'(n)$ is an estimate of the minimal distance (or cost) from $n$ to a goal node.

While $f(n)$, $g'(n)$, and $h'(n)$ are estimates, $f(n) = g(n) + h(n)$ is the actual minimal cost of any path going from $n_0$ through $n$ to a goal node.

Best-First (Heuristic) Search

And what happens if the state-space graph for our problem is not a tree?

For example, is the Eight-puzzle state-space graph a tree?

No. In this puzzle, actions are reversible. Any successor of a node $n$ has $n$ as a successor.

In order to overcome this problem, we have to modify Step 6 in our general graph-searching algorithm.

Improved Graph-Searching Algorithm

1. Create search tree $T_r$ consisting only of the start node (root) $n_0$. Put $n_0$ on an ordered list called OPEN.
2. Create empty list called CLOSED.
3. If OPEN is empty, exit with failure.
4. Move the first node on OPEN, called $n$, to CLOSED.
5. If $n$ is a goal node, exit with success; the solution is the path from $n_0$ to $n$ along the edges in $T_r$.
6. Expand node $n$ by generating a set $M$ of successors that are not already ancestors of $n$ in $T_r$. Connect each node in $M$ to node $n$. Also put the members of $M$ on OPEN.
7. Reorder the list OPEN according to a specific rule.
8. Go to step 3.
Algorithm A*

The algorithm shown on the previous slide, using a heuristic estimation function $f(n) = g'(n) + h'(n)$, is also called Algorithm A*.

Under the following conditions, A* is guaranteed to find the minimal cost path:

- Each node in the graph has a finite number of successors (or no successors).
- All edges in the graph have costs greater than some positive amount $\varepsilon$.
- For all nodes $n$ in the search graph, $h'(n) \leq h(n)$, that is, $h'(n)$ never overestimates the actual value $h(n)$ – an optimistic estimator.