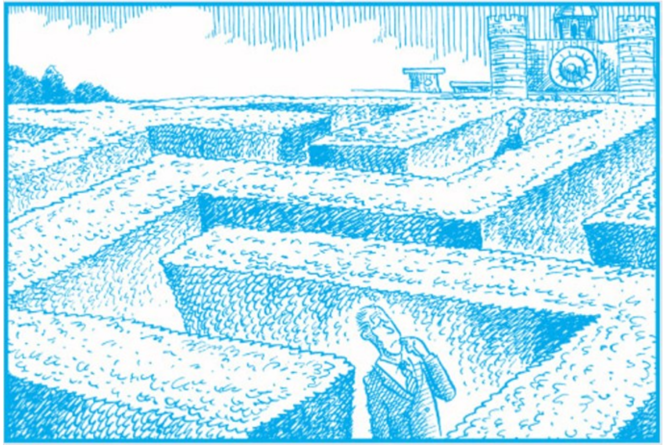
Lecture 16: Chapter 5 Part 1

Backtracking

CS3310

Backtracking

- Suppose we are finding our way through a hedge maze.
- When we reach a fork, we pick a random direction and follow that path until we reach a dead end, at which point we turn around and return to the fork.
- We then pick a different direction from that fork and walk until we reach another dead end.



- It would be so much quicker to solve such a maze if at each fork, there were a sign telling us which directions lead to only dead ends!

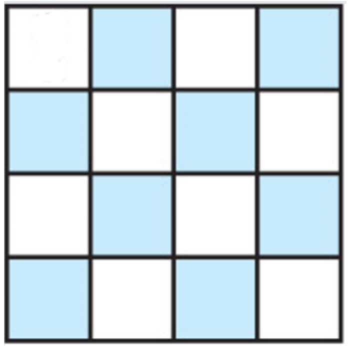
Backtracking

- **Backtracking** algorithms solve problems in which a **sequence** of objects is to be chosen from a specified **set** of objects.
  - This sequence must satisfy some **criterion**.
- We can write a backtracking algorithm to find *every* sequence that satisfies the criterion or one to simply find a *single* such sequence.
- A classic example: the n-Queens problem.
  - How can we position n queens on an $n \times n$ chessboard so no queens threaten each other?

Criterion: No two queens can threaten each other.

Set: ??

Sequence: ??



Backtracking

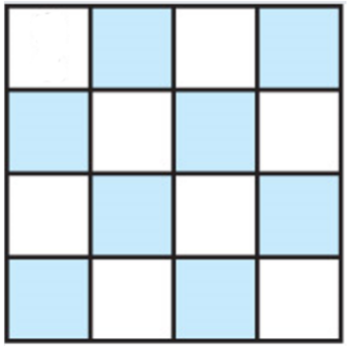
- **Backtracking** algorithms solve problems in which a **sequence** of objects is to be chosen from a specified **set** of objects.
    - This sequence must satisfy some **criterion**.
- We can write a backtracking algorithm to find *every* sequence that satisfies the criterion or one to simply find a *single* such sequence.
- A classic example: the n-Queens problem.
    - How can we position n queens on an $n \times n$ chessboard so no queens threaten each other?

Criterion: No two queens can threaten each other.

Set: The n^2 positions on the board in which a queen can be placed:

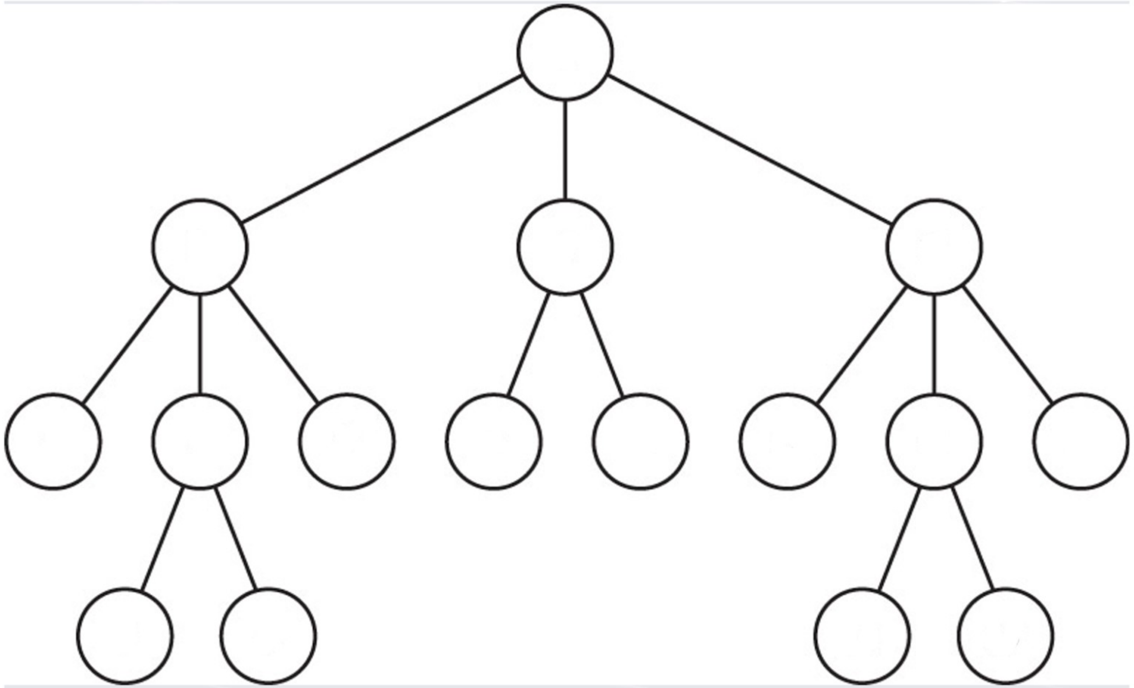
If $n = 4$: {<1, 1>, <1, 2>, <1, 3>, <1, 4>, <2, 1>, <2, 2>, …<4, 4>}

Sequence: n positions from the above set. If queens are placed in each of these n positions, no two should threaten each other.



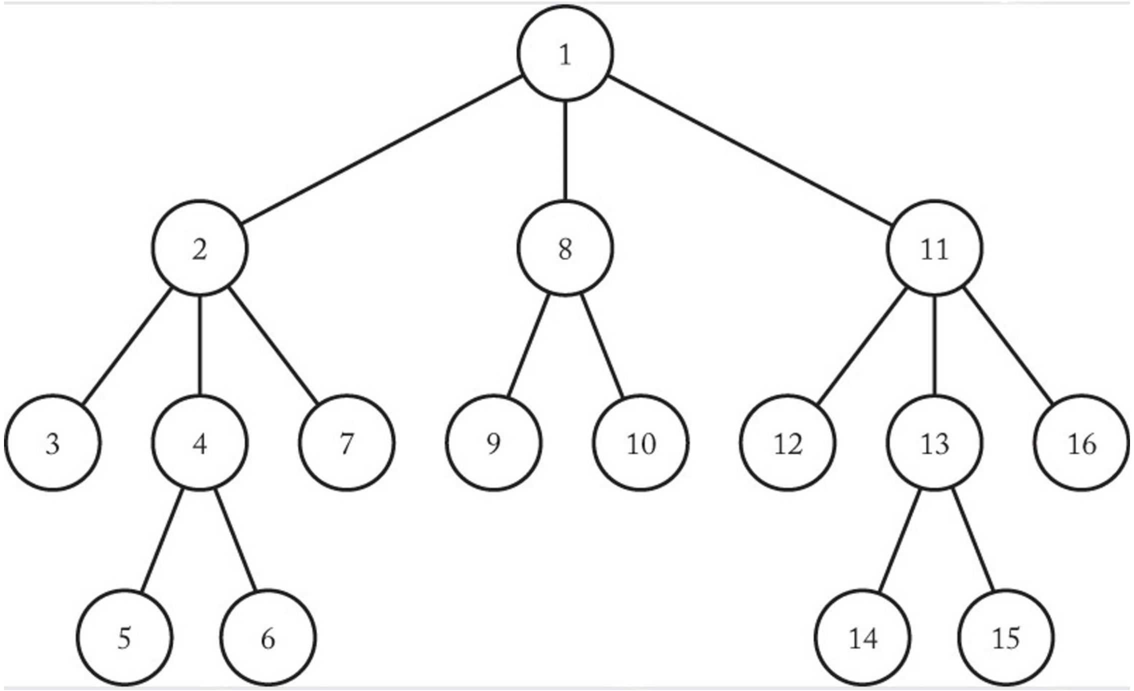
Depth-First Search Review

- Before discussing backtracking further, let's review a **depth-first search** of a tree
  - i.e. a **preorder traversal**.
- In what order do we visit the following nodes in such a search?



Depth-First Search Review

- This tree's nodes are visited in the following order in a **preorder traversal**.



Depth-First Search

- In a **preorder** traversal:
  - The tree's root node is visited first.
  - Each descendant of that node is then visited.
    - We will arbitrarily visit them from left to right
  - When we visit a node, we immediately visit its children from left to right
    - i.e. A path is followed as deeply as possible until a dead end is reached (we reach a leaf).
- At a dead end, we back up until we reach a node with an unvisited child.
- We visit that unvisited child and proceed to go as deep as possible again.

Depth-First Search

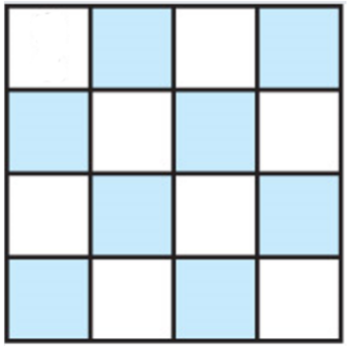
```
void depth_first_tree_search(node v)
{
          node u;
          visit v;
          for (each child u of v)
                    depth_first_tree_search(u);
}
```

- This pseudocode does not state that the children must be visited in any order, but we will arbitrarily visit them from left to right.

n-Queens Problem

We can illustrate backtracking with an instance of the ***n*-Queens** problem when $n = 4$

- **Goal**: place four queens on a 4×4 chessboard so no two queens threaten each other.
- We build a tree, each node containing an ordered pair $<i, j>$.
- A path from the root to a leaf is a **candidate solution** for the problem.
  - The pair $<i, j>$ existing in a candidate solution indicates that a queen is placed in the ith row and jth column.
- What info about how each queen can be placed can we use to reduce how many nodes we need to create?



n-Queens Problem

What info about how each queen can be placed can we use to reduce how many nodes we need to create?

- We can't have two queens in the same row so we can create candidate solutions by constructing a tree in which the possible columns for the first queen (the queen in row 1) are stored in level-1 nodes. The possible columns for the second queen (the queen in row 2) are stored in level-2 nodes, etc.
- This type of tree is called a **state space tree**.
- How deep would such a tree be for the 4-Queens problem?

n-Queens Problem

What info about how each queen can be placed can we use to reduce how many nodes we need to create?

- We can't have two queens in the same row so we can create candidate solutions by constructing a tree in which the possible columns for the first queen (the queen in row 1) are stored in level-1 nodes. The possible columns for the second queen (the queen in row 2) are stored in level-2 nodes, etc.
- This type of tree is called a **state space tree**.
- How deep would such a tree be for the 4-Queens problem?
  - 4 levels (one for each row in a 4×4 chessboard)

n-Queens Problem

Suppose we have the following candidate solution:

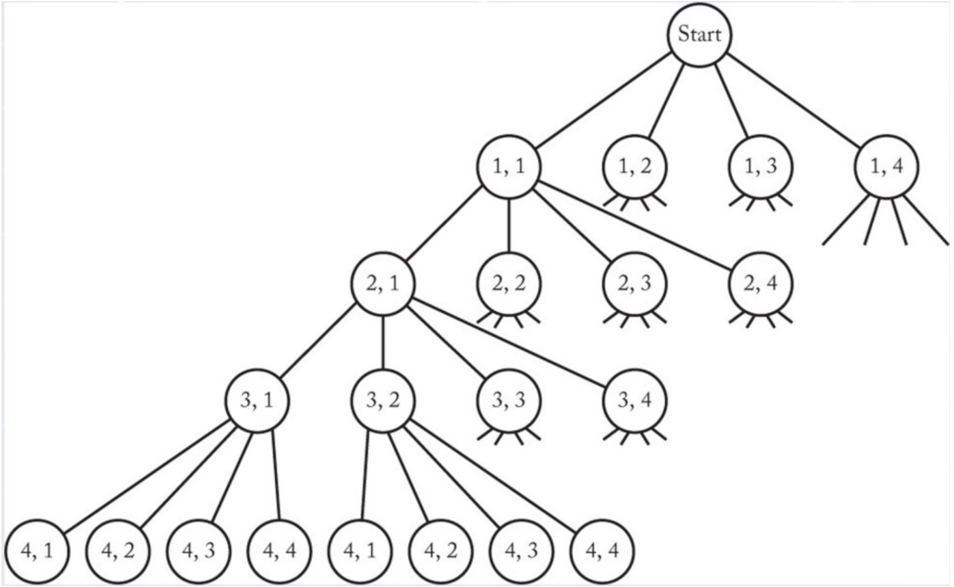
- <1, 1>, <2, 3>, <3, 2>, <4, 1>

Where would the four queens be placed?

n-Queens Problem

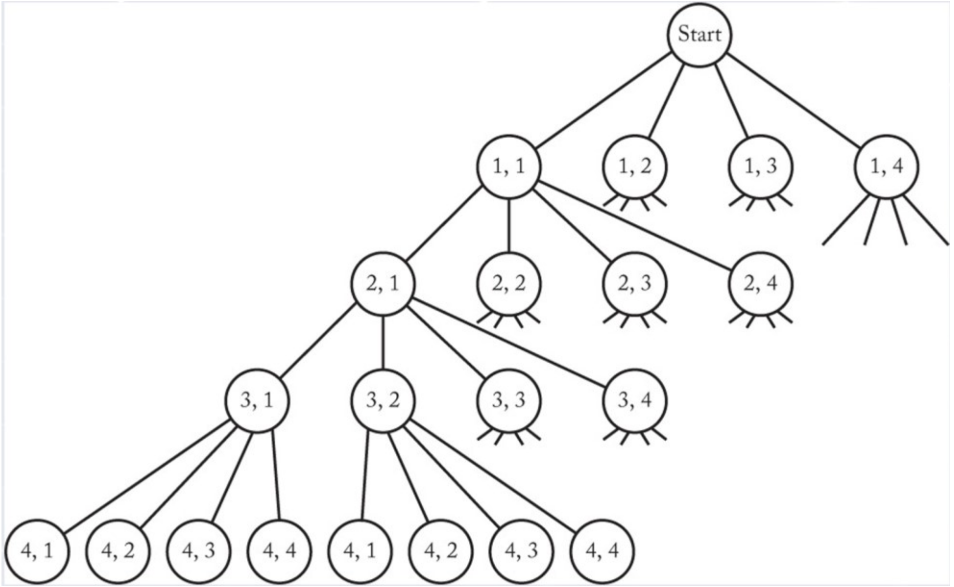
This image shows *part* of the state space tree for the instance of the n-Queens problem where $n = 4$.

- Note that the root of the tree is a **dummy node**.
- When visiting the root, no queens have been placed yet.
- It exists in the tree as a starting off point.



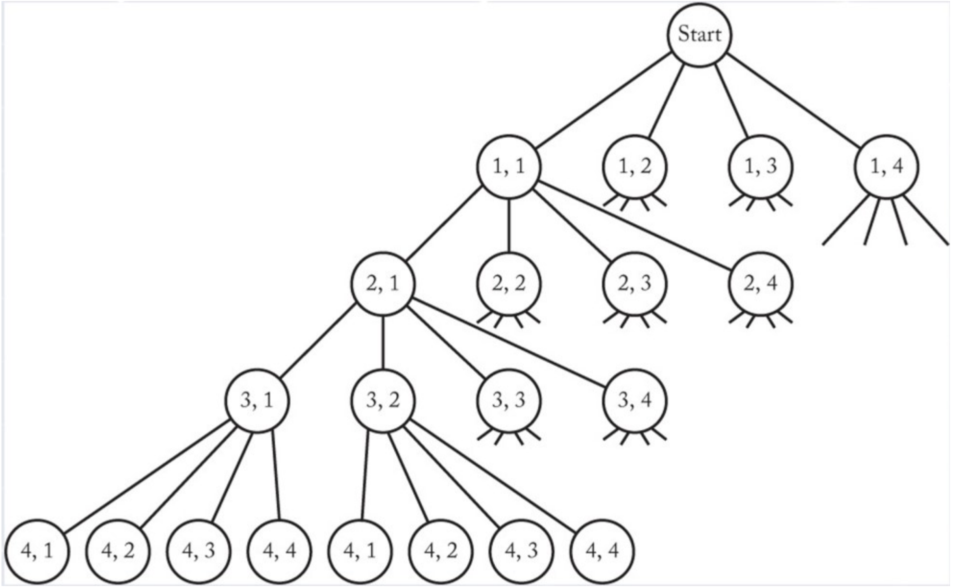
n-Queens Problem

- From the root, we first visit Node (1, 1)
  - We are attempting to place the first queen at location <1, 1>.
- From that Node, we try placing the second row queen in each column.
- From each of these possibilities, we try placing the queen in the third row in each column, etc.



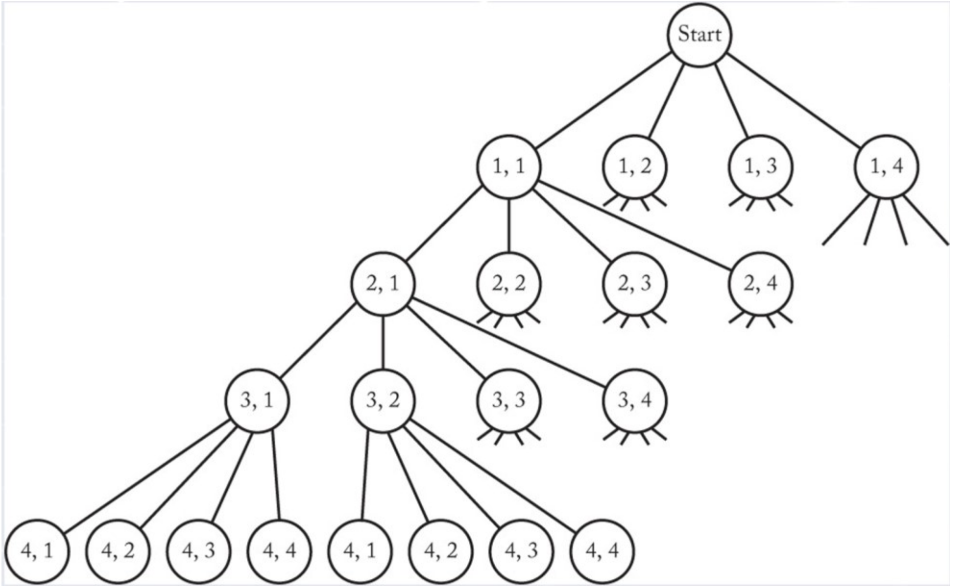
n-Queens Problem

- When traversing a state space tree, how do we know if we have reached a candidate solution?



n-Queens Problem

- When traversing a state space tree, how do we know if we have reached a candidate solution?
- When we reach a leaf! At that point, we have placed 1 queen in each row.
- Each path from root to leaf represents a candidate solution.
- In a depth-first search, which candidate solutions do we consider first and second?



n-Queens Problem

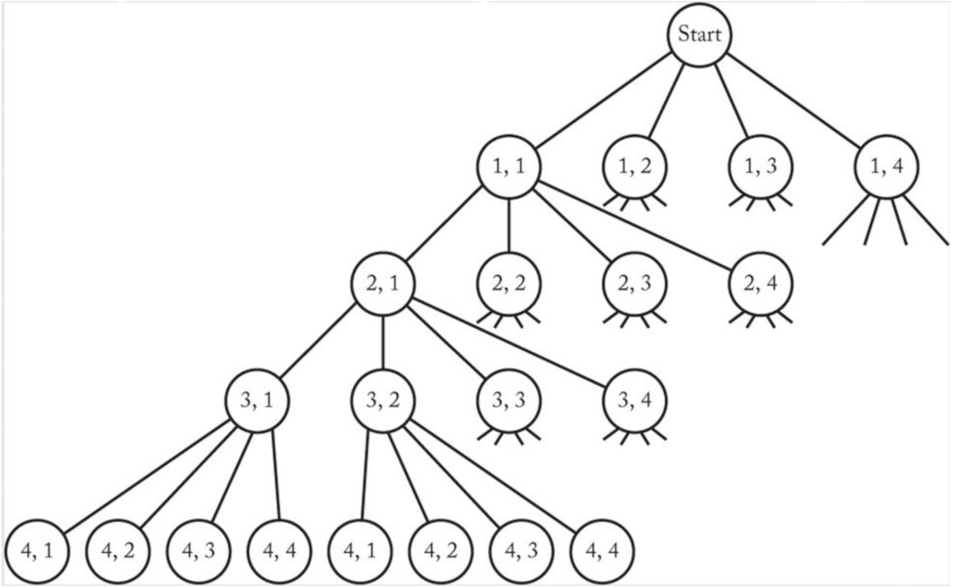
- In a depth-first search, which candidate solutions do we consider first and second?

First

<1, 1>, <2, 1>, <3, 1>, <4, 1>

Second

<1, 1>, <2, 1>, <3, 1>, <4, 2>



n-Queens Problem

The first five candidate solutions considered (none turn out to be *actual* solutions):

[<1, 1>, <2, 1>, <3, 1>, <4, 1>]

[<1, 1>, <2, 1>, <3, 1>, <4, 2>]

[<1, 1>, <2, 1>, <3, 1>, <4, 3>]

[<1, 1>, <2, 1>, <3, 1>, <4, 4>]

[<1, 1>, <2, 1>, <3, 2>, <4, 1>]

As with the hedge maze, a depth-first search follows many paths further than needed.

- Can we confidently turn around early in the middle of any of the above paths?

n-Queens Problem

The first five candidate solutions considered (none turn out to be *actual* solutions):

[<1, 1>, **<2, 1>, <3, 1>, <4, 1>]**

[<1, 1>, **<2, 1>, <3, 1>, <4, 2>]**

[<1, 1>, **<2, 1>, <3, 1>, <4, 3>]**

[<1, 1>, **<2, 1>, <3, 1>, <4, 4>]**

[<1, 1>, **<2, 1>, <3, 2>, <4, 1>]**

- In *each* of these paths we can turn around when we reach <2, 1>!
  - No two queens can be in the same column.
  - If a queen is in <1, 1> and we try to place another in <2, 1>, we can stop following that path since any candidate solution containing <1, 1> and <2, 1> has two queens in the same column.

n-Queens Problem

Can we stop following this path?

[<1, 1>, <2, 2>...]

n-Queens Problem

Can we stop following this path?
[<1, 1>, <2, 2>...]

- **Yes**! No two queens can be on the same diagonal. There is no point in checking any deeper in the tree than <2, 2>.

- **Backtracking** is the procedure by which, after we determine a node only leads to dead ends, we go back (or “backtrack”) to that node’s parent and proceed with the search.

- When we reach <2, 2> in the above path, we call that node **nonpromising** and immediately return to <1, 1>, visiting node <2, 3> next.

A node is **promising** if we can’t determine if it only leads to dead ends *when we visit it*.

Backtracking

In Summary:

- Backtracking consists of performing a depth-first search of a state space tree.
- Whenever we visit a node, we check to see if it is promising.
  - If we determine that visiting any of its children only leads to dead ends, it is nonpromising and we backtrack to the its parent.
    - This is called **pruning** the state space tree.
  - If we cannot determine that visiting any of its children only leads to dead ends, it is promising and visit its children in order.
- The subtree consisting of the visited nodes is called the **pruned state space tree**.

A General Backtracking Algorithm

- Every backtracking algorithm takes a similar general form:

```
void checknode (node v)
        node u;
        if (promising(v))
                if (there is a solution at v)
                        write the solution;
                else
                        for (each child u of v)
                                checknode(u);
```

- The root of a state space tree (a dummy node) is passed to `checknode` at the top level.
- When we reach a solution, we print it out (our *n*-Queens algorithm will print out every solution)
- When a node we visit is *nonpromising*, we return, thus **pruning** that subtree.
- **Note**: the *promising* function is different for each backtracking algorithm.

A General Backtracking Algorithm

- Let’s write out the steps taken to find the first solution to the 4-Queens problem
- Recall that the root of our tree is a dummy node, indicating a start point from which no queens have been placed on the board.
- We visit the root and find that it is promising.

Which node do we visit next, and is it promising or not?

A General Backtracking Algorithm

Root is promising

- <1, 1> is promising
    - <2, 1> is nonpromising (queen 1 is in column 1)
    - <2, 2> is nonpromising (queen 1 is on the left diagonal)
    - <2, 3> is promising
        - <3, 1> is nonpromising (queen 1 is in column 1)
        - <3, 2> is nonpromising (queen 2 is on the right diagonal)
        - <3, 3> is nonpromising (queen 2 is in column 3)
        - <3, 4> is nonpromising (queen 2 is on left diagonal)
        - All of <2, 3>'s children are dead ends.
    - Backtrack to <1, 1> which still has an unvisited child, <2, 4>
    - <2, 4> is promising

A General Backtracking Algorithm continued

- <1, 1> is promising
  - <2, 4> is promising
    - <3, 1> is nonpromising (queen 1 is in column 1)
    - <3, 2> is promising
      - <4, 1> is nonpromising (queen 1 is in column 1)
      - <4, 2> is nonpromising (queen 3 is in column 2)
      - <4, 3> is nonpromising (queen 3 is on left diagonal)
      - <4, 4> is nonpromising (queen 2 is in column 4)

      All of <3, 2>'s children are nonpromising
    - Backtrack to <2, 4> which still has unvisited children
    - <3, 3> is nonpromising (queen 2 is on right diagonal)
    - <3, 4> is nonpromising (queen 2 is in column 4)

    All of <2, 4>'s children are nonpromising. All of <1, 1>'s children are nonpromising. Backtrack to root!

A General Backtracking Algorithm continued

Root is promising

- <1, 2> is promising
  - <2, 1> is nonpromising (queen 1 is on right diagonal)
  - <2, 2> is nonpromising (queen 1 is in column 2)
  - <2, 3> is nonpromising (queen 1 is on left diagonal)
  - <2, 4> is promising
    - <3, 1> is promising
      - <4, 1> is nonpromising (queen 3 is in column 1)
      - <4, 2> is nonpromising (queen 1 is in column 2)
      - <4, 3> is promising. **First solution found!**

First solution: <1, 2>, <2, 4>, <3, 1>, <4, 3>

What do we do from here?

A General Backtracking Algorithm continued

Root is promising

- <1, 2> is promising
  - <2, 1> is nonpromising (queen 1 is on right diagonal)
  - <2, 2> is nonpromising (queen 1 is in column 2)
  - <2, 3> is nonpromising (queen 1 is on left diagonal)
  - <2, 4> is promising
    - <3, 1> is promising
      - <4, 1> is nonpromising (queen 3 is in column 1)
      - <4, 2> is nonpromising (queen 1 is in column 2)
      - <4, 3> is promising. **First solution found!**

First solution: <1, 2>, <2, 4>, <3, 1>, <4, 3>

What do we do from here? Keep checking nodes for the next solution!

A General Backtracking Algorithm continued

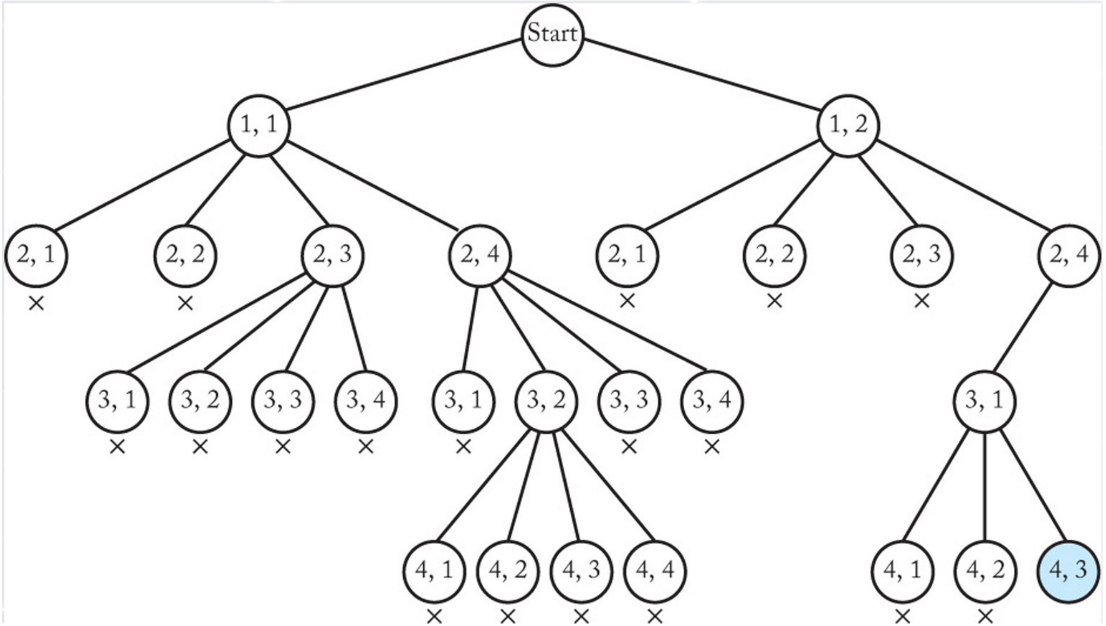
Root is promising

- <1, 2> is promising
    - <2, 1> is nonpromising (queen 1 is on right diagonal)
    - <2, 2> is nonpromising (queen 1 is in column 2)
    - <2, 3> is nonpromising (queen 1 is on left diagonal)
    - <2, 4> is promising
        - <3, 1> is promising
            - <4, 1> is nonpromising (queen 3 is in column 1)
            - <4, 2> is nonpromising (queen 1 is in column 2)
            - <4, 3> is promising. **First solution found!**
            - <4, 4> is nonpromising (queen 2 is in column 4)
            - All of <3, 1>'s children have been visited.
        - Backtrack to <2, 4> which still has unvisited children
        - <3, 2> is nonpromising (queen 1 is in column 2)
        - <3, 3> is nonpromising (queen 2 is on left diagonal)

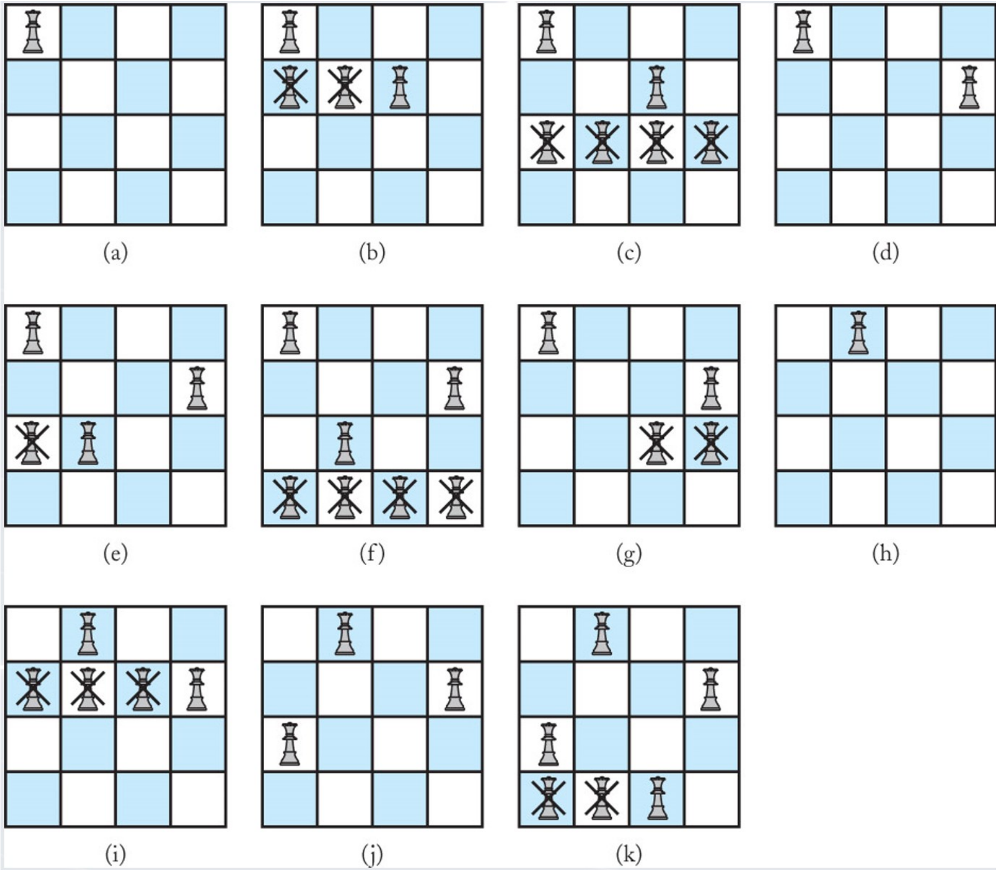
... etc

A General Backtracking Algorithm

This portion of the pruned state space tree shows the nodes visited when calculating the first solution of the n-Queens problem with n = 4.



A General Backtracking Algorithm



A General Backtracking Algorithm

Note: a backtracking algorithm doesn't create an actual tree.

- Rather, we only keep track of the nodes in the current path being investigated.
    - The state space tree exists **implicitly** since it is not actually constructed.
- For the *n*-Queens problem, a one-dimensional array `col` holds *n* integers such that `col[i]` = the column where the queen in the *i*th row is located.
    - i.e. `col[1] = 2` indicates the current path has a queen in location <1, 2>

A General Backtracking Algorithm

Suppose `col` has the following values:

{ 1, 3, 4, ? }

- This indicates we are attempting to place a queen in <3, 4>
- How do we determine if this is promising?

A General Backtracking Algorithm

Suppose `col` has the following values:

{ 1, 3, 4, ? }

- This indicates we are attempting to place a queen in <3, 4>
- How do we determine if this is promising?

Check Columns

- See if any previously entered queens are in the same column (column 4)
  - If `col[k] == col[i]` for any $1 < k < i$, we can backtrack

Check Diagonals

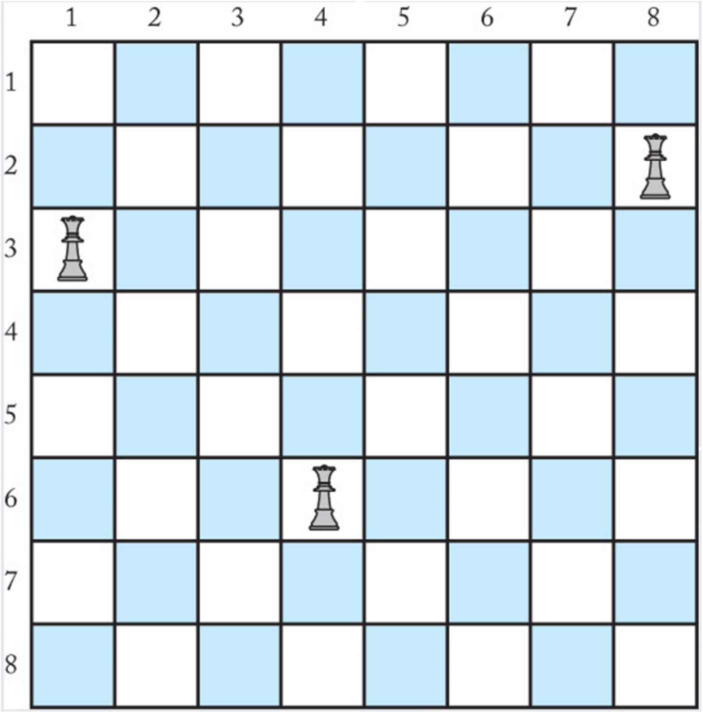
- How do we check diagonals?

A General Backtracking Algorithm

How do we check diagonals?

Let $n = 8$ and queens be at <3, 1>, <2, 8>, <6, 4>

- The queen at <6, 4> is being threatened by both the queen at <3, 1> and <2, 8>
- How can our program determine this?



A General Backtracking Algorithm

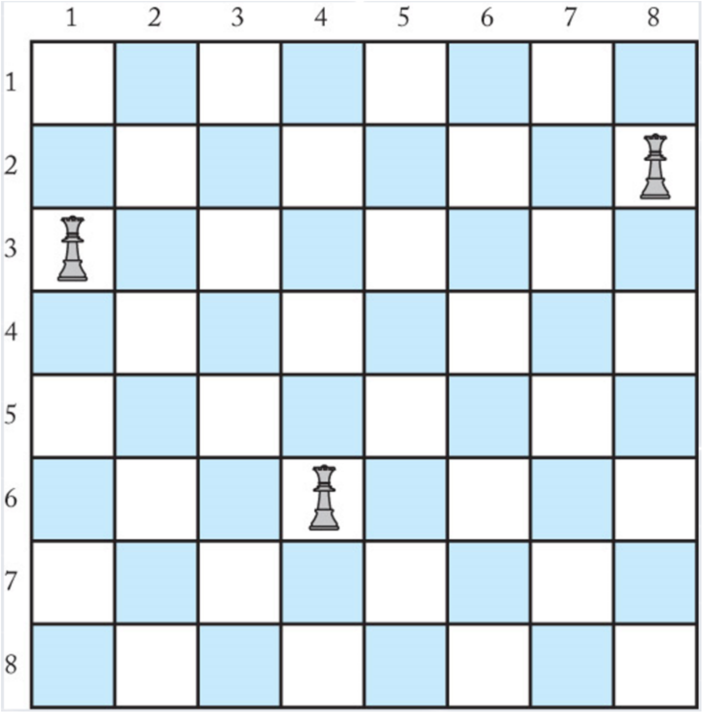
How do we check diagonals?

Let $n = 8$ and queens be at <3, 1>, <2, 8>, <6, 4>

- The queen at <6, 4> is being threatened by both the queen at <3, 1> and <2, 8>
- How can our program determine this?

`col[6] - col[3]` = 4 - 1 = 3.

- The queen in row 6 is 3 columns from the queen in row 3.
- 6 - 3 is also 3. Both queens are 3 rows away from each other. Therefore, they are the same # of columns and rows away.

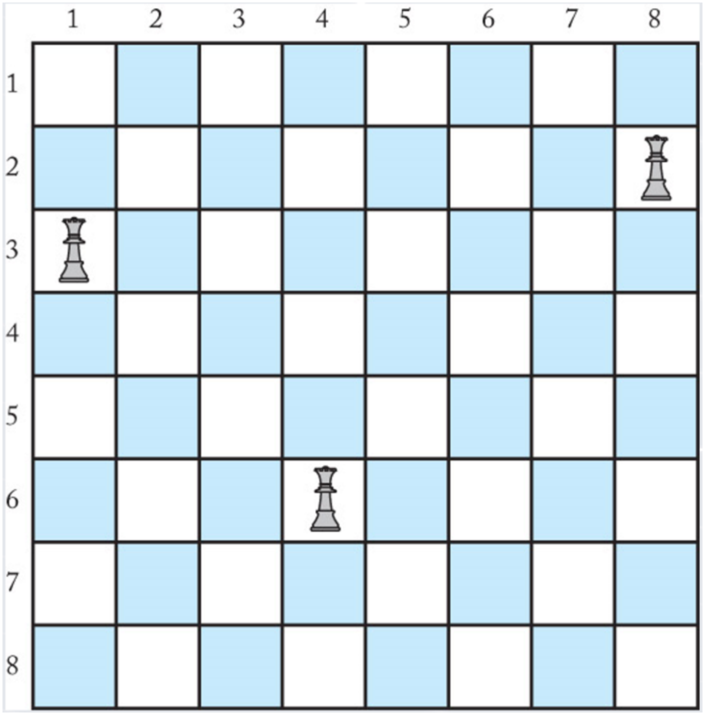


A General Backtracking Algorithm

More generally:

```
if col[i] - col[k] = i - k
or col[i] - col[k] = k - i
then queens in rows i and k are on same diagonal.
```

Can we simplify this further?



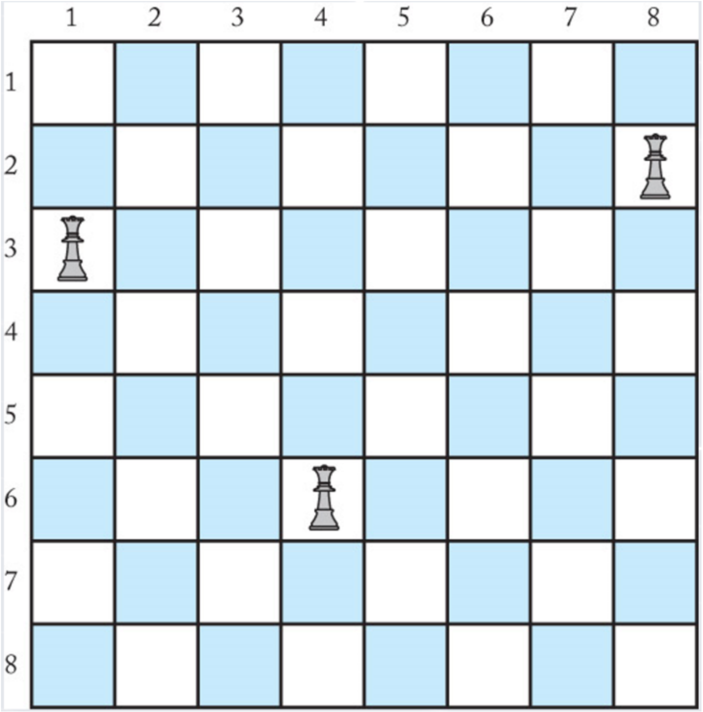
A General Backtracking Algorithm

More generally:

```
if col[i] - col[k] = i - k
or col[i] - col[k] = k - i
then queens in rows i and k are on same diagonal.
```

Can we simplify this further?

```
if abs(col[i] - col[k]) == i - k
then queens in rows i and k are on same diagonal.
```



n-Queens Backtracking Algorithm

Problem: Position n queens on a chessboard so that no two are in the same row, column, or diagonal.

Inputs: Positive integer n

Outputs: All possible ways n queens can be placed on an $n \times n$ chessboard so that no two queens threaten each other.

Each output consists prints n values from the array of integers `col` which is indexed from 1 to n, where `col[i]` is the column where the queen in the ith row is placed.

n-Queens Backtracking Algorithm

Note: At the top level, we call `queens(0)`. 0 represents the dummy node

```
void queens (index i)
{
            index j;
            if (promising(i))
                        if (i == n) // we reached a promising leaf. We are at a solution
                                    cout << col[1] through col[n];
                        else
                                    for (j = 1; j <= n; j++)   // see if queen in (i + 1)st
                                                col[i + 1] = j;  // row can be positioned in
                                                queens(i + 1);   // each of the n columns
}
```

n-Queens Backtracking Algorithm

```
bool promising (index i)
{
          index k = 1;
          bool promising = true;

     // check all previously selected queens to ensure none threaten queen in row i.
          while (k < i && promising)
          {
                    if (col[i] == col[k] || abs(col[i] - col[k]) == i - k)
                              promising = false;
                    k ++;
          }
          return promising;
}
```

- Returns true when the dummy node is passed: 0 < *k*'s initial value.
- Returns false if any of the previously placed queens are in the same column as the queen in row *i* or if they are on the same diagonal; true otherwise

In-Class Exercise

1. Show the first two solutions to the *n*-Queens problem for *n* = 5. Show nodes visited.

i.e.

<1, 1>

    <2, 1>

    <2, 2>

    <2, 3>

        <3, 1>

etc...