Lecture 21
Backtracking Algorithms with Search Space Pruning

EECS 281: Data Structures & Algorithms

Outline

- Review
  - Backtracking vs. Branch and Bound
- Backtracking General Form
- N-Queens

Types of Algorithm Problems

- Constraint Satisfaction problems
  - Can we satisfy all given constraints?
  - If yes, how do we satisfy them?
    - Need a specific solution
  - May have more than one solution
  - Examples: sorting, puzzles, GRE/analytical
- Optimization problems
  - Must satisfy all constraints (can we?) and
  - Must minimize an objective function subject to those constraints

Types of Algorithm Problems

- Constraint Satisfaction problems
  - Go over all possible solutions
  - Does a given input combination satisfy all constraints?
  - Can stop when a satisfying solution is found
- Optimization problems
  - Similar, except we also need to compute the objective function every time
  - Stopping early = non-optimal solution

Review

Backtracking is to *Constraint Satisfaction*

AS

Branch and Bound is to *Optimization*

General Form: Backtracking

```
type checknode(node v)
    if (promising(v))
        if (solution(v))
            write solution
        else
            for each child node u of v
                checknode(u)
```

General Form: Backtracking

`solution(v)`

- Check ‘depth’ of solution (constraint satisfaction)

`promising(v)`

- Different for each application

`checknode(v)`

- Called only if promising and not solution

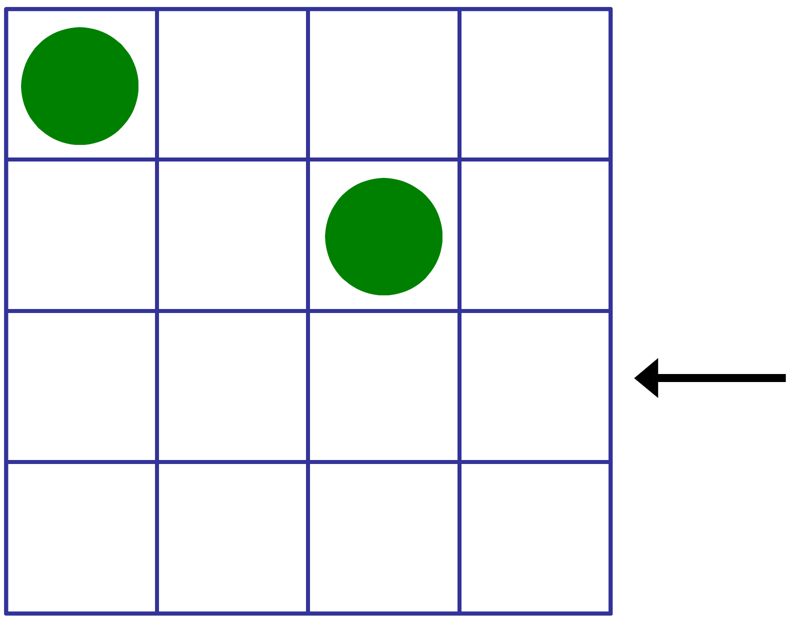
Specific Example: N-Queens

- N = 1: Can 1 queen be placed on a 1x1 board so that it doesn’t threaten another?
- N = 2
- N = 3
- N = 4
- N = 5
- …

8 Queens: Search Space

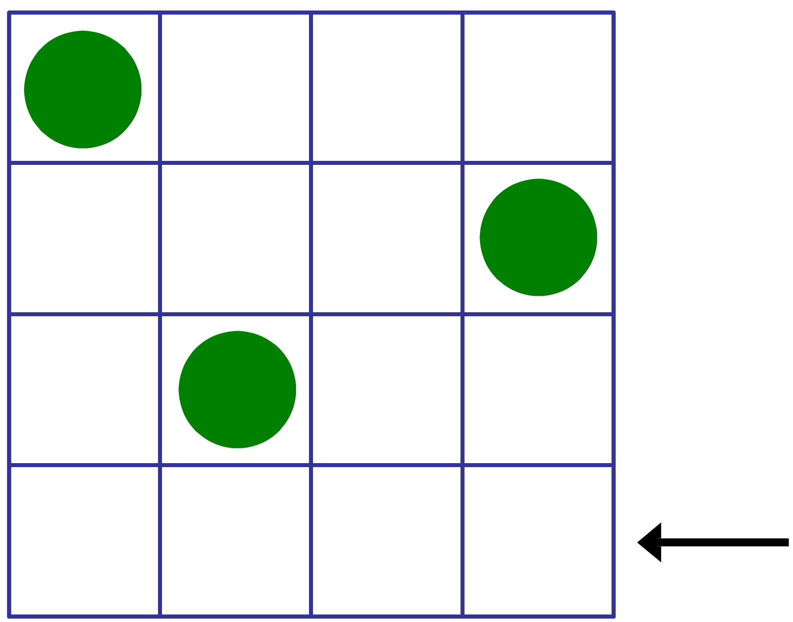
- Note that the search space is fairly large:
  - 16,772,216 possibilities
  - 92 solutions
- Reduce the search space

Backtracking



Where can a queen be placed in this row?

Backtracking



Where can a queen be placed in this row?

Constraint Satisfaction: N-Queens

- Could require one solution
- Could require several solutions
- Could require all solutions

Specific Form: N-Queens

`solution(v)`

- Check 'depth' of solution (constraint satisfaction)
- Placed queen on each row
- That is, depth = N

`checknode(v)`

- Called only if promising and not solution
- Recursive call to all positions (columns) of queen within row

Specific Form: N-Queens

`promising(v)`

- Called for each node
- Assume functions that return column and/or row location of any queen, i, where i < v:
  - col(i) // returns column location of queen #i
  - row(i) // returns row location of queen #i
- <u>NOT</u> promising if:
  - In same column as any preceding queen
    col(i) == col(v)
  - On same diagonal as any preceding queen
    abs(col(i) – col(v)) == abs(row(i) – row(v))

Summary: N-Queens

For 4-Queens

- Entire tree has 256 leaves
- Backtracking enables searching of 19 nodes before finding first solution
- Promising:
  - May lead to solution
- Not promising:
  - Will never lead to solution
  - Therefore should be pruned

Summary: Backtracking

- Backtracking allows pruning of branches that are not promising
- All backtracking algorithms have a similar form
- Often, most difficult part is determining nature of **`promising()`**

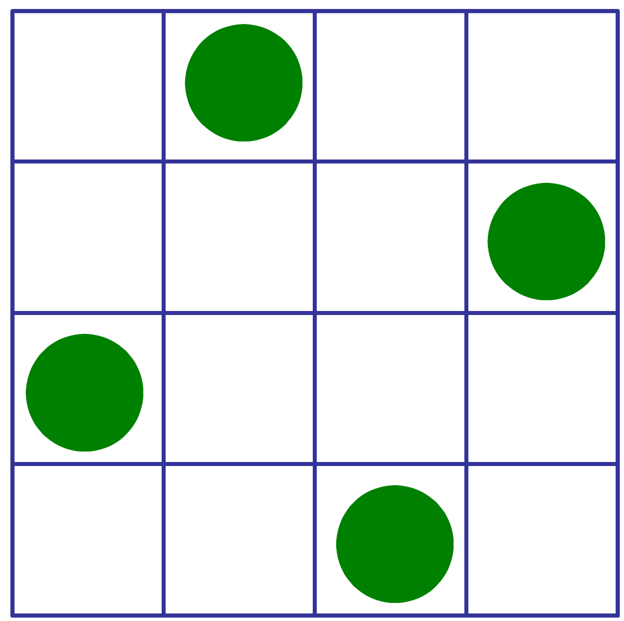
N-Queens Implementation

- We know that for queens:
  - Each row will have exactly one queen
  - Each column will have exactly one queen
  - Each diagonal will have at most one queen
- Don’t model the chessboard as 2D array!
  - Instead, use 1D arrays of rows, columns and diagonals
- To simplify the presentation, we will study for smaller chessboard, 4x4

Implementing the Chessboard

First: we need to define an array to store the location of queens placed so far

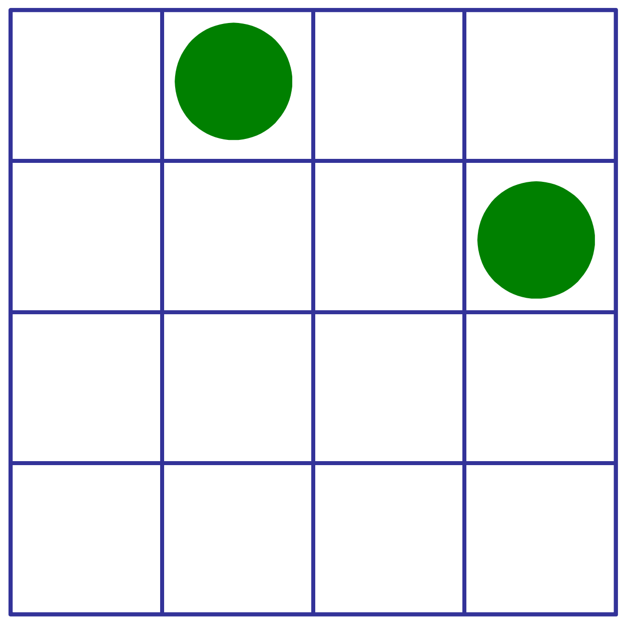
positionInRow
1
3
0
2



Implementing the Chessboard (cont.)

We need an array to keep track of the availability status of the column when we assign queens

Suppose that we have placed two queens

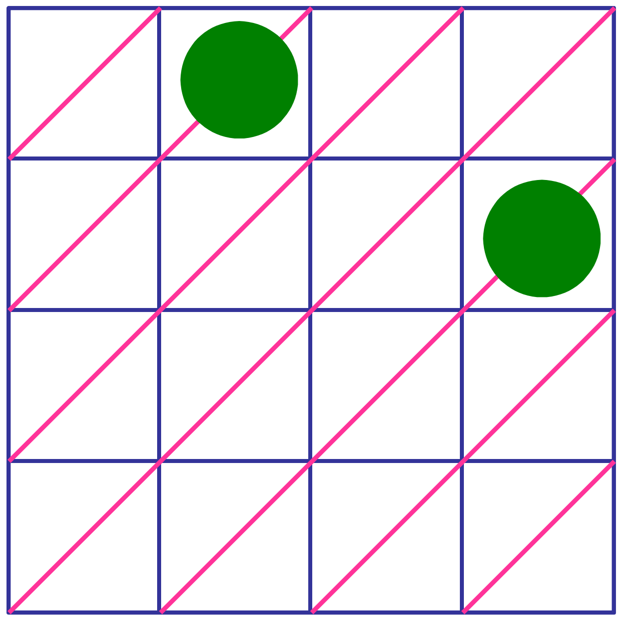


T	**F**	**T**	**F**

Implementing the Chessboard (cont.)

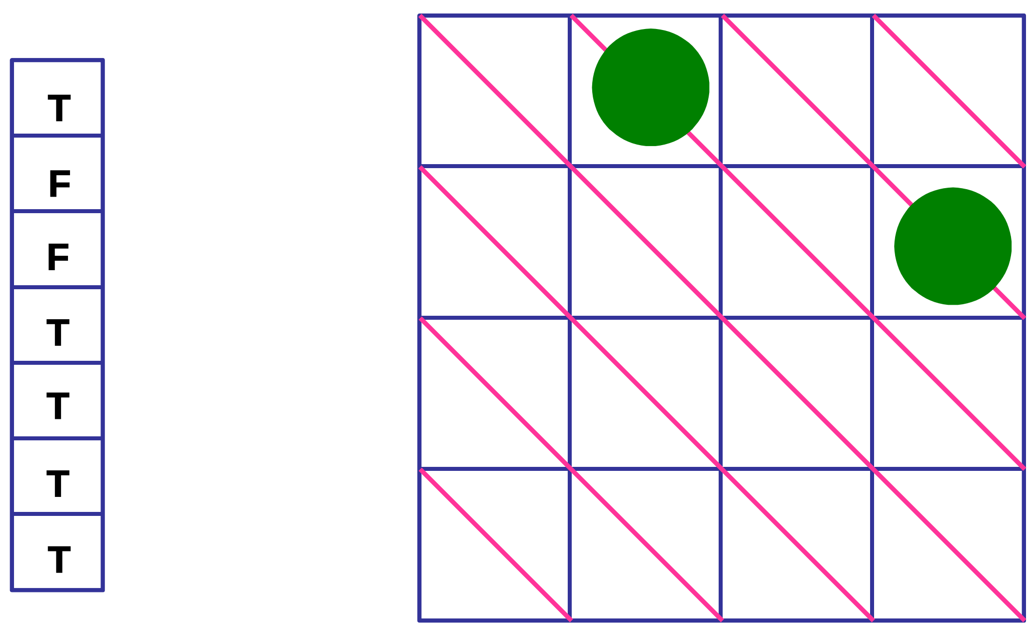
We have 7 left diagonals (2 * N - 1); we want to keep track of available diagonals after queens are placed (start at upper left)

T
F
T
T
F
T
T



Implementing the Chessboard Cont'd

We have 7 right diagonals; we want to keep track of available diagonals after queens are placed (start at upper right)



The putQueen() Recursive Method

```
putQueen(int row) {
  for (int col = 0; col < squares; col++)
    if (   column[col] == available
        && leftDiagonal[row + col] == available
        && rightDiagonal[row – col + (squares - 1)] == available) {
      positionInRow[row] = col;
      column[col] = !available;
      leftDiagonal[row + col] = !available;
      rightDiagonal[row – col + (squares - 1)] = !available;
      if (row < squares - 1)
        putQueen(row + 1);
      else
        print("solution found");

      // Undo this move and thus backtrack
      column[col] = available;
      leftDiagonal[row + col] = available;
      rightDiagonal[row – col + (squares - 1)] = available;
    } // if
} // putQueen()
```