BACKTRACK SEARCH ALGORITHMS
Recursive Backtracking Search

• Recursion allows us to enumerate all solutions to some problem
• Backtracking algorithms are often used to solve constraint satisfaction problems or optimization problems
  – Find optimum solution that meet some constraints

• Key property of backtracking search:
  – Stop searching down a path at the first indication that constraints won't lead to a solution

• Knapsack problem
  – You have a set of products with a given weight and value. Suppose you have a knapsack that can hold N pounds, which subset of objects can you pack that maximizes the value.
  – Example:
    • Knapsack can hold 11 pounds
    • Product A: 7 pounds, $12 ea.  Product B: 10 pounds, $18 ea.
    • Product C: 4 pounds, $7 ea.  Product D: 2.4 pounds, $4 ea.

• Other examples:
  – Map Coloring, Satisfiability, Sudoku, N-Queens
N-Queens Problem

• Problem: How to place N queens on an N x N chess board such that no queens may attack each other.

• Queens can attack at any distance vertically, horizontally, or diagonally.

• Observation: Different queen in each row and each column.

• Backtrack search approach:
  – Place 1\textsuperscript{st} queen in a viable option then, then try to place 2\textsuperscript{nd} queen, etc.

  – If no queen can be placed in row or there are no more options in row to try to place queen, backtrack to redo work in previous row.
4x4 Example of N-Queens
8x8 Example of N-Queens

• Now place 2\textsuperscript{nd} queen
8x8 Example of N-Queens

- Now place others as viable
- After this configuration here, there are no locations in row 6 that are not under attack from the previous 5
- BACKTRACK!!!
8x8 Example of N-Queens

• Backtrack: go back to row 5 and switch assignment to next viable option and progress back to row 6
8x8 Example of N-Queens

• Still no location available in row 6, so BACKTRACK!
• Backtrack to row 5 to check for another location
8x8 Example of N-Queens

• Backtrack: Check the next free location in row 5
• There are no more locations for queen in row 5 so BACKTRACK!
8x8 Example of N-Queens

- To backtrack, return back to row 4
- Move Queen in row 4 to another place in row 4 and restart row 5 exploration
8x8 Example of N-Queens

- Move to another place in row 4 and restart row 5 exploration
8x8 Example of N-Queens

• Now a viable location for a Queen exists for row 6
• Keep placing Queens until all rows including row 8 has Queen
• Return this configuration of Queens as solution
• What if no solution exists?
8x8 Example of N-Queens

• What if no solution exists?
  – Row 1 queen would have exhausted all her options and still not find a solution
Backtracking Search

• Recursion can be used to generate all options
  – 'brute force' / test all options approach
  – Test for constraint satisfaction only at the bottom of the 'tree'

• But backtrack search attempts to 'prune' the search space
  – Rule out options at the partial assignment level

Brute force enumeration might test only when a complete assignment is made (i.e. all 4 queens on the board)
N-Queens Solution Development

• Let's develop the code
• 1 queen per row
  – Use an array where index represents the queen (and the row) and value is the column
• Start at row 0 and initiate the search [i.e. search(0) ]
• Base case:
  – Rows range from 0 to n-1 so STOP when row == n
  – Solution found!
• Recursive case
  – Recursively try all column options for that queen

Index = Queen i in row i
q[i] = column of queen i

```c
int *q; // pointer to array storing each queens location
int n; // number of board / size

void search(int row)
{
  if(row == n)
    printSolution(); // solved!
  else {
    for(q[row]=0; q[row]<n; q[row]++){
      search(row+1);
    }
  }
}
```
N-Queens Solution Development

- To check whether it is safe to place a queen in a column, keep a threat 2-D array indicating the threat level at each square on the board
  - Threat level of 0 means SAFE
  - Update squares under threat of queen placement

```c
int *q; // pointer to array storing each queens location
int n; // number of board / size
int **t; // thread 2D array

main()
{
    q = new int[n];
    t = new int*[n];
    for(int i=0; i < n; i++){
        t[i] = new int[n];
        for(int j = 0; j < n; j++){
            t[i][j] = 0;
        }
    }
    search(0); // start search
deallocate arrays
    return 0;
}
```

Index = Queen i in row i
q[i] = column of queen i

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
N-Queens Solution Development

- Check if queen placed is safe
- Update the threats (+1) due to this new queen placement
- Recurse to next row
- If return, no solution existed for given placement, so Backtrack!
- To Backtrack: i) remove threats and ii) iterate to try the next location for this queen

```c
int *q; // pointer to array storing each queens location
int n; // number of board / size
int **t; // n x n threat array
void search(int row)
{
    if(row == n)
        printSolution(); // solved!
    else {
        for(q[row]=0; q[row]<n; q[row]++){
            // check that col: q[row] is safe
            if(t[row][q[row]] == 0){
                // if safe place and continue
                addToThreats(row, q[row], 1);
                search(row+1);
                // if return, remove placement
                addToThreats(row, q[row], -1);
            }
        }
    }
}
```

Index = Queen $i$ in row $i$
$q[i] =$ column of queen $i$

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Safe to place queen in upper left

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Now add threats

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Upon return, remove threat and iterate to next option

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
addToThreats Code

• Observations
  – Already a queen in every higher row so addToThreats only needs to deal with positions lower on the board
    • Iterate row+1 to n-1
  – Enumerate all locations further down in the same column, left diagonal and right diagonal
  – Add or remove a threat by passing in change
  – Change is +1 to add threats and -1 to remove threats

```c
void addToThreats(int row, int col, int change)
{
    for(int j = row+1; j < n; j++){
        // go down column
        t[j][col] += change;
        // go down right diagonal
        if( col+(j-row) < n )
            t[j][col+(j-row)] += change;
        // go down left diagonal
        if( col-(j-row) >= 0 )
            t[j][col-(j-row)] += change;
    }
}
```
### N-Queens Solution

```c
int *q; // queen location array
int n; // number of board / size
int **t; // n x n threat array

int main()
{
    q = new int[n];
    t = new int*[n];
    for(int i=0; i < n; i++) {
        t[i] = new int[n];
        for(int j = 0; j < n; j++) {
            t[i][j] = 0;
        }
    }

    search(0);
}

void addToThreats(int row, int col, int change)
{
    for(int j = row+1; j < n; j++) {
        // go down column
        t[j][col] += change;
        // go down right diagonal
        if (col+(j-row) < n) t[j][col+(j-row)] += change;
        // go down left diagonal
        if (col-(j-row) >= 0) t[j][col-(j-row)] += change;
    }
}

void search(int row)
{
    if(row == n) {
        printSolution(); // solved!
        return true;
    }
    else {
        for(q[row]=0; q[row]<n; q[row]++) {
            // check that col: q[row] is safe
            if(t[row][q[row]] == 0) {
                // if safe place and continue
                addToThreats(row, q[row], 1);
                search(row+1);

                // if return, remove placement
                addToThreats(row, q[row], -1);
            }
        }
    }
}
```