CSCI 104
Backtracking Search

Mark Redekopp
David Kempe
Sandra Batista
BACKTRACK SEARCH ALGORITHMS
Recursion and DFS

- Recursion forms a kind of Depth-First Search

```cpp
// user interface
void binCombos(int len)
{
    binCombos("", len);
}

// helper-function
void binCombos(string prefix, int len)
{
    if(prefix.length() == len )
        cout << prefix << endl;
    else {
        // recurse
        binCombos(prefix+"0", len);
        // recurse
        binCombos(prefix+"1", len);
    }
}
```
Generating All Combinations

- Recursion offers a simple way to generate all combinations of $N$ items from a set of options, $S$
  - Example: Generate all 2-digit decimal numbers ($N=2$, $S=\{0,1,...,9\}$)

void NDigDecCombos(string data, int n)
{
    if(data.size() == n)
      cout << data;
    else {
      for(int i=0; i < 10; i++){
        // recurse
        NDigDecCombos(data+(char)('0'+i),n);
      }
    }
}
Get the Code

• In-class exercises
  – nqueens-allcombos
  – nqueens

• On your VM
  – $ mkdir nqueens
  – $ cd nqueens
  – $ wget http://ee.usc.edu/~redekopp/cs104/nqueens.tar
  – $ tar xvf nqueens.tar

http://ee.usc.edu/~redekopp/cs104/nqueens.tar
Recursive Backtracking Search

• Recursion allows us to "easily" enumerate all solutions to some problem
• Backtracking algorithms...
  – Are often used to solve constraint satisfaction problem or optimization problems
    • Several items that can be set to 1 of N values under some constraints
  – Stop searching down a path at the first indication that constraints won't lead to a solution
• Some common and important problems can be solved with backtracking
• Knapsack problem
  – You have a set of objects 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 35 pounds
      • Object A: 7 pounds, $12 ea.  Object B: 10 pounds, $18 ea.
      • Object C: 4 pounds, $7 ea.  Object D: 2.4 pounds, $4 ea.
• Other examples:
  – Map Coloring
  – Traveling Salesman Problem
  – Sudoku
  – N-Queens
N-Queens Problem

- Problem: How to place N queens on an NxN chess board such that no queens may attack each other
- Fact: Queens can attack at any distance vertically, horizontally, or diagonally
- Observation: Different queen in each row and each column
- Backtrack search approach:
  - Place 1st queen in a viable option then, then try to place 2nd queen, etc.
  - If we reach a point where no queen can be placed in row i or we've exhausted all options in row i, then we return and change row i-1
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

• Now place others as viable
• After this configuration here, there are no locations in row 6 that is not under attack from the previous 5
• So go back to row 5 and switch assignment to next viable option and progress back to row 6
8x8 Example of N-Queens

• Now place others as viable
• After this configuration here, there are no locations in row 6 that is not under attack from the previous 5
• Now go back to row 5 and switch assignment to next viable option and progress back to row 6
• But still no location available so return back to row 5
8x8 Example of N-Queens

- Now place others as viable
- After this configuration here, there are no locations in row 6 that is not under attack from the previous 5
- Now go back to row 5 and switch assignment to next viable option and progress back to row 6
- But still no location available so return back to row 5
- But now no more options for row 5 so return back to row 4
- BACKTRACK!!!!
8x8 Example of N-Queens

• Now place others as viable
• After this configuration here, there are no locations in row 6 that is not under attack from the previous 5
• Now go back to row 5 and switch assignment to next viable option and progress back to row 6
• But still no location available so return back to row 5
• But now no more options for row 5 so return back to row 4
• Move to another place in row 4 and restart row 5 exploration
8x8 Example of N-Queens

- Now place others as viable
- After this configuration here, there are no locations in row 6 that is not under attack from the previous 5
- Now go back to row 5 and switch assignment to next viable option and progress back to row 6
- But still no location available so return back to row 5
- But now no more options for row 5 so return back to row 4
- Move to another place in row 4 and restart row 5 exploration
8x8 Example of N-Queens

- Now a viable option exists for row 6
- Keep going until you successfully place row 8 in which case you can return your solution
- What if no solution exists?
8x8 Example of N-Queens

• Now a viable option exists for row 6
• Keep going until you successfully place row 8 in which case you can return your solution
• 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
  - Means we found a solution
- Recursive case
  - Recursively try all column options for that queen
  - But haven't implemented check of viable configuration...

```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 particular column, let's keep a "threat" 2-D array indicating the threat level at each square on the board
  – Threat level of 0 means SAFE
  – When we place a queen we'll update squares that are now under threat
  – Let's name the array 't'

• Dynamically allocating 2D arrays in C/C++ doesn't really work
  – Instead conceive of 2D array as an "array of arrays" which boils down to a pointer to a pointer

```
int *q;  // pointer to array storing each queens location
int n;   // number of board / size
int **t; // thread 2D 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); // start search
    // deallocate arrays
    return 0;
}
```
N-Queens Solution Development

- After we place a queen in a location, let's check that it has no threats
- If it's safe then we update the threats (+1) due to this new queen placement
- Now recurse to next row
- If we return, it means the problem was either solved or more often, that no solution existed given our placement so we remove the threats (-1)
- Then we iterate to try the next location for this queen

```
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);
            }
        }
    }
}
```
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
  – Can use same code to add or remove a threat by passing in change

• Can't just use 2D array of booleans as a square might be under threat from two places and if we remove 1 piece we want to make sure we still maintain the threat

```java
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
00 int *q;  // queen location array
01 int n;   // number of board / size
02 int **t; // n x n threat array
03
04 int main()
05 {
06     q = new int[n];
07     t = new int*[n];
08     for(int i=0; i < n; i++){
09         t[i] = new int[n];
10         for(int j = 0; j < n; j++){
11             t[i][j] = 0;
12         }
13     }
14     // do search
15     if( !search(0) )
16         cout << "No sol!" << endl;
17     // deallocate arrays
18     return 0;
19 }
20 void addToThreats(int row, int col, int change)
21 {
22     for(int j = row+1; j < n; j++){
23         // go down column
24         t[j][col] += change;
25         // go down right diagonal
26         if( col+(j-row) < n )
27             t[j][col+(j-row)] += change;
28         // go down left diagonal
29         if( col-(j-row) >= 0)
30             t[j][col-(j-row)] += change;
31     }
32 }
33
34 bool search(int row)
35 {
36     if(row == n){
37         printSolution(); // solved!
38         return true;
39     }
40     else {
41         for(q[row]=0; q[row]<n; q[row]++)
42             // check that col: q[row] is safe
43             if(t[row][q[row]] == 0){
44                 // if safe place and continue
45                 addToThreats(row, q[row], 1);
46                 bool status = search(row+1);
47                 if(status) return true;
48                 // if return, remove placement
49                 addToThreats(row, q[row], -1);
50             }
51         return false;
52     }
53 }
54 int *q;  // queen location array
55 int n;   // number of board / size
56 int **t; // n x n threat array
57
58 int main()
59 {
60     q = new int[n];
61     t = new int*[n];
62     for(int i=0; i < n; i++){
63         t[i] = new int[n];
64         for(int j = 0; j < n; j++){
65             t[i][j] = 0;
66         }
67     }
68     // do search
69     if( !search(0) )
70         cout << "No sol!" << endl;
71     // deallocate arrays
72     return 0;
73 }
74 void addToThreats(int row, int col, int change)
75 {
76     for(int j = row+1; j < n; j++){
77         // go down column
78         t[j][col] += change;
79         // go down right diagonal
80         if( col+(j-row) < n )
81             t[j][col+(j-row)] += change;
82         // go down left diagonal
83         if( col-(j-row) >= 0)
84             t[j][col-(j-row)] += change;
85     }
86 }
87
88 bool search(int row)
89 {
90     if(row == n){
91         printSolution(); // solved!
92         return true;
93     }
94     else {
95         for(q[row]=0; q[row]<n; q[row]++)
96             // check that col: q[row] is safe
97             if(t[row][q[row]] == 0){
98                 // if safe place and continue
99                 addToThreats(row, q[row], 1);
100                bool status = search(row+1);
101                if(status) return true;
102                // if return, remove placement
103                 addToThreats(row, q[row], -1);
104             }
105         return false;
106     }
107 }
108 int *q;  // queen location array
109 int n;   // number of board / size
110 int **t; // n x n threat array
111 int main()
112 {
113     q = new int[n];
114     t = new int*[n];
115     for(int i=0; i < n; i++){
116         t[i] = new int[n];
117         for(int j = 0; j < n; j++){
118             t[i][j] = 0;
119         }
120     }
121     // do search
122     if( !search(0) )
123         cout << "No sol!" << endl;
124     // deallocate arrays
125     return 0;
126 } 
```
General Backtrack Search Approach

- Select an item and set it to one of its options such that it meets current constraints
- Recursively try to set next item
- If you reach a point where all items are assigned and meet constraints, done...return through recursion stack with solution
- If no viable value for an item exists, backtrack to previous item and repeat from the top
- If viable options for the 1st item are exhausted, no solution exists
- Phrase:
  - Assign, recurse, unassign

General Outline of Backtracking Sudoku Solver

```cpp
bool sudoku(int **grid, int r, int c)
{
    if( allSquaresComplete(grid) )
        return true;
    
    // iterate through all options
    for(int i=1; i <= 9; i++){
        grid[r][c] = i;
        if( isValid(grid) ){
            bool status = sudoku(...);
            if(status) return true;
        }
    }
    return false;
}
```

Assume r,c is current square to set and grid is the 2D array of values