CSE 373 Analysis of Algorithms
Fall 2016
Instructor: Prof. Sael Lee

LEC14: BACKTRACKING (PP. 230-247)
(CH7 COMBINATORIAL SEARCH AND HEURISTIC METHODS)

Lecture slide courtesy of Prof. Steven Skiena

SUDOKU

		1 2
	3 5	
6		7
7		3
	4	8
1		
	1 2	
8		4
5		6

6	7	3	8	9	4	5	1	2
9	1	2	7	3	5	4	8	6
8	4	5	6	1	2	9	7	3
7	9	8	2	6	1	3	5	4
5	2	6	4	7	3	8	9	1
1	3	4	5	8	9	2	6	7
4	6	9	1	2	8	7	3	5
2	8	7	3	5	6	1	4	9
3	5	1	9	4	7	6	2	8

SOLVING SUDOKU

- Solving Sudoku puzzles involves a form of exhaustive search of possible configurations.
- However, exploiting constraints to rule out certain possibilities for certain positions enables us to ***prune*** the search to the point people can solve Sudoku by hand.
- **Backtracking** is the key to implementing exhaustive search programs correctly and efficiently.

BACKTRACKING

- **Backtracking** is a systematic method to iterate through all the possible configurations of a search space.
- It is a general algorithm/technique which must be customized for each individual application.
- In the general case, we will **model our solution** as a vector $a = (a_1, a_2,\ldots,a_n)$, where each element a_i is selected from a finite ordered set S_i.
  - Such a vector might represent an arrangement where a_i contains the *i*th element of the permutation.
  - Or the vector might represent a given subset S, where a_i is true if and only if the *i*th element of the universe is in S.

THE IDEA OF BACKTRACKING

- At each step in the backtracking algorithm, we start from a given partial solution, say, a = (a_1, a_2,...,a_k), and try to extend it by adding another element at the end.
- After extending it, we must **test**
  - whether what we have so far is a solution.
  - If not, we must then check whether the partial solution is still potentially extendible to some complete solution.
  - If so, recur and continue.
  - If not, we delete the last element from a and try another possibility for that position, if one exists.

RECURSIVE BACKTRACKING

```
Backtrack-DFS(A, k)
    if A = (a1, a2, ..., ak) is a solution, report it.
    else
        k = k + 1
        compute Sk          /* possible next states */
        while Sk = Ø do
            ak = an element in Sk
            Sk = Sk − ak
            Backtrack-DFS(A, k)
```

BACKTRACKING AND DFS

- Backtracking is really just **depth-first search** on an implicit graph of configurations, ie partition solutions.
- Backtracking can easily be used to iterate through all subsets or permutations of a set.
- Backtracking ensures correctness by enumerating all possibilities.
- For backtracking to be efficient, we must prune the search space.

IMPLEMENTATION

```
bool finished = FALSE;                  /* found all solutions yet? */

backtrack(int a[], int k, data input)
{
    int c[MAXCANDIDATES];  /* candidates for next position */
    int ncandidates;       /* next position candidate count */
    int i;                 /* counter */
    if (is_a_solution(a,k,input))
        process_solution(a,k,input);
    else {
        k = k+1;
        construct_candidates(a,k,input,c,&ncandidates);
        for (i=0; i<ncandidates; i++) {
            a[k] = c[i];
            make_move(a,k,input);
            backtrack(a,k,input);
            unmake_move(a,k,input);
            if (finished) return;       /* terminate early */
        }
    }
}
```

is_a_solution(a,k,input)

- This Boolean function tests whether the first k elements of vector a are a complete solution for the given problem.
- The last argument, *input*, allows us to pass general information into the routine.

construct_candidates(a,k,input,c,ncandidates)

- This routine fills an array *c* with the complete set of possible candidates for the *k*th position of a, given the contents of the first *k-1* positions.
- The number of candidates returned in this array is denoted by *ncandidates*.

process_solution(a,k)

- This routine prints, counts, or somehow processes a complete solution once it is constructed.
- Backtracking ensures correctness by enumerating all possibilities.
- It ensures efficiency by never visiting a state more than once.
- Because a new candidates array c is allocated with each recursive procedure call, the subsets of not-yet-considered extension candidates at each position will not interfere with each other.

make_move(a,k,input)

unmake_move(a,k,input)

- These routines enable us to modify a data structure in response to the latest move, as well as clean up this data structure if we decide to take back the move.
- Such a data structure could be rebuilt from scratch from the solution vector *a* as needed, but this is inefficient when each move involves incremental changes that can easily be undone.

CONSTRUCTING ALL SUBSETS

- How many subsets are there of an n-element set?
- To construct all 2^n subsets, set up an array/vector of n cells, where the value of a_i is either true or false, signifying whether the ith item is or is not in the subset.
- To use the notation of the general backtrack algorithm, S_k = *(true, false)*, and v is a solution whenever k >= n.
- What order will this generate the subsets of {1,2,3}?

$$(1) \rightarrow (1,2) \rightarrow (1,2,3)* \rightarrow$$

$$(1,2,-)* \rightarrow (1,-) \rightarrow (1,-,3)* \rightarrow$$

$$(1,-,-)* \rightarrow (1,-) \rightarrow (1) \rightarrow$$

$$(-) \rightarrow (-,2) \rightarrow (-,2,3)* \rightarrow$$

$$(-,2,-)* \rightarrow (-,-) \rightarrow (-,-,3)* \rightarrow$$

$$(-,-,-)* \rightarrow (-,-) \rightarrow (-) \rightarrow ()$$

CONSTRUCTING ALL SUBSETS

- We can construct the 2^n subsets of n items by iterating through all possible 2^n length-n vectors of *true* or *false*, letting the ith element denote whether item i is or is not in the subset.
- Using the notation of the general backtrack algorithm, S_k = (*true; false*), and a is a solution whenever k >= n.

```
is_a_solution(int a[], int k, int n)
{
        return (k == n);        /* is k == n? */
}
```

CONSTRUCTING ALL SUBSETS

```
construct_candidates(int a[], int k, int n, int c[], int *ncandidates)
{
    c[0] = TRUE;
    c[1] = FALSE;
    *ncandidates = 2;
}

process_solution(int a[], int k)
{
    int i; /* counter */
    printf("{");
    for (i=1; i<=k; i++)
        if (a[i] == TRUE) printf(" %d",i);
    printf(" }\n");
}
```

MAIN ROUTINE: SUBSETS

- Finally, we must instantiate the call to backtrack with the right arguments.

```
generate_subsets(int n)
{
    int a[NMAX];                    /* solution vector */
    backtrack(a,0,n);
}
```

CONSTRUCTING ALL PERMUTATIONS

- How many permutations are there of an n-element set?
- To construct all n! permutations, set up an array/vector of n cells, where the value of a_i is an integer from 1 to n which has not appeared thus far in the vector, corresponding to the ith element of the permutation.
- To use the notation of the general backtrack algorithm, S_k =*(1, ..., n)* - *v*, and v is a solution whenever k >= n.

$$
\begin{aligned}
(1) &\rightarrow (1,2) \rightarrow (1,2,3)* \rightarrow (1,2) \rightarrow (1) \rightarrow (1,3) \rightarrow \\
(1,3,2)* &\rightarrow (1,3) \rightarrow (1) \rightarrow () \rightarrow (2) \rightarrow (2,1) \rightarrow \\
(2,1,3)* &\rightarrow (2,1) \rightarrow (2) \rightarrow (2,3) \rightarrow (2,3,1)* \rightarrow (2,3) \rightarrow () \\
(2) &\rightarrow () \rightarrow (3) \rightarrow (3,1)(3,1,2)* \rightarrow (3,1) \rightarrow (3) \rightarrow \\
(3,2) &\rightarrow (3,2,1)* \rightarrow (3,2) \rightarrow (3) \rightarrow ()
\end{aligned}
$$

CONSTRUCTING ALL PERMUTATIONS

- To avoid repeating permutation elements,
- S_k =*(1, …, n)* - *a*, and a is a solution whenever k = n

```
construct_candidates(int a[], int k, int n, int c[], int *ncandidates)
{
        int i;                          /* counter */
        bool in_perm[NMAX];             /* who is in the permutation? */
        for (i=1; i<NMAX; i++) in_perm[i] = FALSE;
        for (i=0; i<k; i++) in_perm[ a[i] ] = TRUE;

        *ncandidates = 0;
        for (i=1; i<=n; i++)
                if (in_perm[i] == FALSE) {
                        c[ *ncandidates] = i;
                        *ncandidates = *ncandidates + 1;
                }
}
```

AUXILIARY ROUTINES

- Completing the job of generating permutations requires specifying process solution and is a solution, as well as setting the appropriate arguments to backtrack.
- All are essentially the same as for subsets:

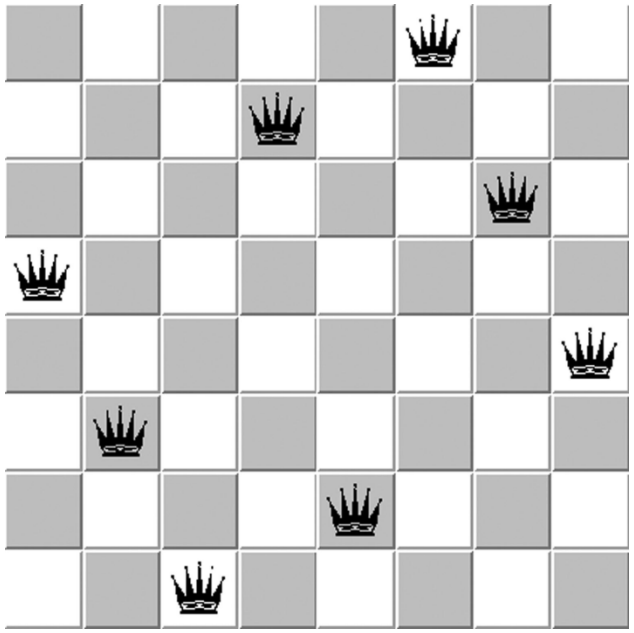
```
process_solution(int a[], int k)
{
     int i; /* counter */
          for (i=1; i<=k; i++)
                    printf(" %d",a[i]);
          printf("\n");
}
```

```
is_a_solution(int a[], int k, int n)
{
          return (k == n);
}
```

MAIN PROGRAM: PERMUTATIONS

```
generate_permutations(int n)
{
    int a[NMAX];                    /* solution vector */
    backtrack(a,0,n);
}
```

THE EIGHT-QUEENS PROBLEM



The eight queens problem is a classical puzzle of positioning eight queens on an 8x8 chessboard such that no two queens threaten each other

EIGHT QUEENS: REPRESENTATION

- What is concise, efficient representation for an n-queens solution, and how big must it be?
- Since no two queens can occupy the same column, we know that the n columns of a complete solution must form a permutation of n. By avoiding repetitive elements, we reduce our search space to just 8! = 40,320 – clearly short work for any reasonably fast machine.
- The critical routine is the candidate constructor. We repeatedly check whether the kth square on the given row is threatened by any previously positioned queen. If so, we move on, but if not we include it as a possible candidate:

CANDIDATE CONSTRUCTOR: EIGHT QUEENS

```
construct_candidates(int a[], int k, int n, int c[], int *ncandidates)
{
        int i,j; (* counters *)
        bool legal_move; (* might the move be legal? *)

        *ncandidates = 0;
        for (i=1; i<=n; i++) {
                legal_move = TRUE;
                for (j=1; j<k; j++) {
                        if (abs((k)-j) == abs(i-a[j])) (* diagonal threat *)
                                legal_move = FALSE;
                        if (i == a[j]) (* column threat *)
                                legal_move = FALSE;
                }
                if (legal_move == TRUE) {
                        c[*ncandidates] = i;
                        *ncandidates = *ncandidates + 1;
                }
        }
}
```

- The remaining routines are simple, particularly since we are only interested in counting the solutions, not displaying them:

```
process_solution(int a[], int k)
{
        int i; (* counter *)

        solution_count ++;
}

is_a_solution(int a[], int k, int n)
{
        return (k == n);
}
```

FINDING THE QUEENS: MAIN PROGRAM

```
nqueens(int n)
{
        int a[NMAX]; (* solution vector *)

        solution_count = 0;
        backtrack(a,0,n);
        printf("n=}
```

- This program can find the 365,596 solutions for n = 14 in minutes.

CAN EIGHT PIECES COVER A CHESS BOARD?

- Consider the 8 main pieces in chess (king, queen, two rooks, two bishops, two knights). Can they be positioned on a chessboard so every square is threatened?

			N	B			R
			N				
R							
	B						
		Q			K		

COMBINATORIAL SEARCH

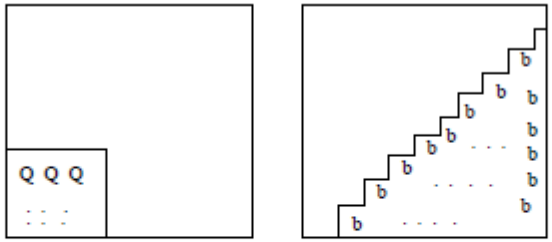
- Only 63 square are threatened in this configuration. Since 1849, no one had been able to find an arrangement with bishops on different colors to cover all squares.
- We can resolve this question by searching through all possible board configurations *if* we spend enough time.
- We will use it as an example of how to attack a combinatorial search problem.
- With clever use of backtracking and pruning techniques, surprisingly large problems can be solved by exhaustive search.

HOW MANY CHESS CONFIGURATIONS MUST BE TESTED?

- Picking a square for each piece gives us the bound:
- $64!=(64 - 8)! = 178, 462, 987, 637, 760 \approx 10^{15}$
- Anything much larger than 10^8 is unreasonable to search on a modest computer in a modest amount of time.

EXPLOITING SYMMETRY

- However, we can exploit symmetry to save work. With reflections along horizontal, vertical, and diagonal axis, the queen can go in only 10 non-equivalent positions.
- Even better, we can restrict the white bishop to 16 spots and the queen to 16, while being certain that we get all distinct configurations.



$$16 \times 16 \times 32 \times 64 \times 2080 \times 2080 = 2,268,279,603,200 \approx 10^{12}$$

COVERING THE CHESS BOARD

- In covering the chess board, we prune whenever we find there is a square which we *cannot* cover given the initial configuration!
- Specifically, each piece can threaten a certain maximum number of squares (queen 27, king 8, rook 14, etc.) We *prune* whenever the number of unthreated squares exceeds the sum of the maximum remaining coverage.
- This backtrack search eliminates 95% of the search space, when the pieces are ordered by decreasing mobility.
- With precomputing the list of possible moves, this program could search 1,000 positions per second.

END GAME

- But this is still too slow!

$$10^{12}=10^3 = 10^9 \text{ seconds} > 1000 \text{ days}$$

- Although we might further speed the program by an order of magnitude, we need to prune more nodes!
- By using a more clever algorithm, we eventually were able to prove no solution existed, in less than one day's worth of computing.
- You too can fight the combinatorial explosion!