Lecture 15: Backtracking

Steven Skiena

Department of Computer Science
State University of New York
Stony Brook, NY 11794–4400

http://www.cs.stonybrook.edu/~skiena

Topic: Problem of the Day

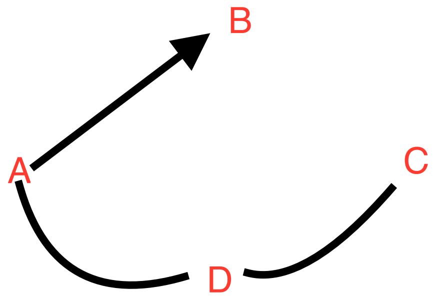
Problemi di conteggio combinatoriali sono risoliti con la tecnica del backtracking poichè la forza bruta potrebbe non essere efficiente.

Problem of the Day

grafo

Let $G = (V, E)$ be a directed, weighted graph such taht all weights are positive. Let v and w be two vertices in G, and $k \leq |V|$ be an integer. Design an algorithm to find the shortest path from v to w that contains exactly k edges. Note that the path need not be simple.

Questions?



Topic: Backtracking

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.

SFRUTTARE I VINCOLI DEL PROBLEMA PER RIDURRE LO SPAZIO DI RICERCA

Backtracking

Backtracking is a systematic method to iterate through all possible configurations of a search space. It is a general algorithm which must be customized for each application.

We model our solution as a vector $a = (a_1, a_2, ..., a_n)$, 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 ith element of the permutation. Or the vector might represent a given subset S, where a_i is true if and only if the ith 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 test whether what we have so far is a complete solution.

If not, the critical issue is whether the current partial solution a is potentially extendible to a solution.

- If so, recur and continue.
- If not, delete the last element from a and try another possibility for that position if one exists.

Questions?

Topic: Backtracking Implementation

Recursive Backtracking

Backtrack(a, k)
if a is a solution, print(a)
else {
$k = k + 1$
compute S_k
while $S_k \neq \emptyset$ do
a_k = an element in S_k
S_k = $S_k - a_k$
Backtrack(a, k)
}

Tolgo dalle insieme delle mie alternative quelle che ho già trovatp

Backtracking and DFS

Backtracking is really just depth-first search on an implicit graph of configurations.

- 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 dead or redundent branches of the search space whenever possible.

Backtracking Implementation

```
void backtrack(int a[], int k, data input) {
    int c[MAXCANDIDATES];       /* candidates for next position */
    int nc;                     /* 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, &nc);
        for (i = 0; i < nc; 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 to evaluate whether a is a solution.

`construct_candidates(a,k,input,c,nc)`

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

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.

Questions?

Topic: Constructing Subsets by Backtracking

Constructing all Subsets

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 $k \geq n$.

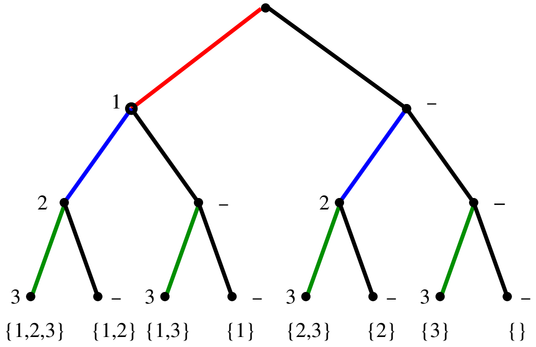
Subset Generation Tree / Order

What order will this generate the subsets of $\{1, 2, 3\}$?

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

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

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



Using Backtrack to Construct Subsets

We can construct all subsets of n items by iterating through all 2^n length-n vectors of *true* or *false*, letting the ith element denote whether item i is (or is not) in the subset.

Thus the candidate set $S_k = (true, false)$ for all positions, and a is a solution when $k \geq n$.

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

void construct_candidates(int a[], int k, int n, int c[], int *nc) {
    c[0] = true;
    c[1] = false;
    *nc = 2;
}
```

Process the Subsets

Here we print the elements in each subset, but you can do whatever you want – like test whether it is a vertex cover solution. . .

```
void process_solution(int a[], int k, int input) {
    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.

```
void generate_subsets(int n) {
    int a[NMAX];                           /* solution vector */

    backtrack(a, 0, n);
}
```

Questions?

Topic: Constructng Permutations by Backtracking

Constructing all Permutations

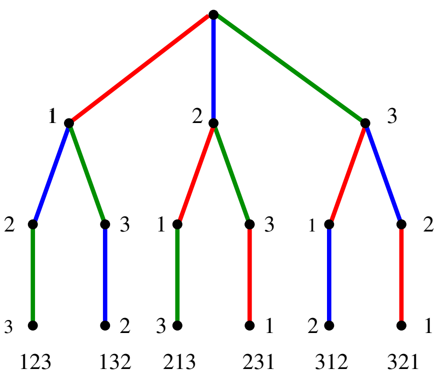
How many permutations are there of an n-element set?

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, $S_k = (1, \ldots, n) - v$, and v is a solution whenever $k \geq n$.

Permutation Generation Tree / Order

$$
\begin{array}{rcl}
(1) & \rightarrow & (1,2) \rightarrow {\color{red}(1,2,3)} \rightarrow (1,2) \rightarrow (1) \rightarrow (1,3) \rightarrow \\
{\color{red}(1,3,2)} & \rightarrow & (1,3) \rightarrow (1) \rightarrow () \rightarrow (2) \rightarrow (2,1) \rightarrow \\
{\color{red}(2,1,3)} & \rightarrow & (2,1) \rightarrow (2) \rightarrow (2,3) \rightarrow {\color{red}(2,3,1)} \rightarrow (2,3) \rightarrow () \\
(2) & \rightarrow & () \rightarrow (3) \rightarrow (3,1){\color{red}(3,1,2)} \rightarrow (3,1) \rightarrow (3) \rightarrow \\
(3,2) & \rightarrow & {\color{red}(3,2,1)} \rightarrow (3,2) \rightarrow (3) \rightarrow ()
\end{array}
$$



Constructing All Permutations

To avoid repeating permutation elements, $S_k = \{1, \ldots, n\} - a$, and a is a solution whenever $k = n$:

```
void construct_candidates(int a[], int k, int n, int c[], int *nc) {
    int i;                  /* counter */
    bool in_perm[NMAX];     /* what is now in the permutation? */

    for (i = 1; i < NMAX; i++) {
        in_perm[i] = false;
    }

    for (i = 1; i < k; i++) {
        in_perm[a[i]] = true;
    }

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

Auxilliary 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:

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

    for (i = 1; i <= k; i++) {
        printf(" %d", a[i]);
    }
    printf("\n");
}


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

Main Program: Permutations

```
void generate_permutations(int n) {
    int a[NMAX];                        /* solution vector */

    backtrack(a, 0, n);
}
```