Lecture 8
Backtracking

Euiseong Seo
(euiseong@skku.edu)
Backtracking

• Systematic method to iterate through all the possible configurations of a search space
• General algorithm/technique which must be customized for each individual application
General Backtracking

- Solution is a vector $a = (a_1, a_2, \ldots, a_n)$
- $a_i$ is selected from a finite ordered set $S_i$
  1. At each step, start from a given partial solution $a = (a_1, a_2, \ldots, a_k)$
  2. Try to extend it by adding another element at the end
  3. After extending it, test whether what we have so far is a solution
     1. If not, check whether the partial solution is still potentially extendible to some complete solution
        1. If so, recur and continue
        2. If not, we delete the last element from $a$ and try another possibility for that position, if one exists
     2. If so, finishes
Implementation

```c
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];
            backtrack(a,k,input);
            if (finished) return;   /* terminate early */
        }
    }
}
```
Application-Specific Routines

- `is_a_solution(a, k, input)`
- `construct_candidates(a, k, input, c, ncandidate)`
- `process_solution(a, k)`
Constructing All Subsets

- Using the general backtracking algorithm
- $S_k = (\text{true, false})$
- $a$ is a solution whenever $k \geq n$
is_a_solution(int a[], int k, int n)
{
    return (k == n);  /* is k == n? */
}

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

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

- $S_k = \{1,2,\ldots n\} - a$
  - To avoid repeating permutation elements
- $a$ is a solution whenever $k = n$

```c
construct_candidates(int a[], int k, int n, int c[], int *ncandidates)
{
    int i;
    bool in_perm[NMAX];
    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;
        }
}
```
Constructing All Permutations

```c
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);
}

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

4-Queens Problem
N-Queens Problem

- Applying backtracking when n = 8
  - $S_k = \text{(queen, no queen)}$
  - $2^{64}$ search space

- Think $a_i$ is $i$-th queen’s position
  - $64^8$
  - Exploit the fact that all queens are identical
N-Queens Problem

- Only a single queen can exist in a row
- A row must have a queen
- We can save search space by utilizing these facts
  - Pruning
  - $8!$ of complexity
N-Queens Problem

```c
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;
        }
    }
}
```
15 Puzzle Problem
Tug of War

A tug of war is being arranged for the office picnic. The picnickers must be fairly divided into two teams. Every person must be on one team or the other, the number of people on the two teams must not differ by more than one, and the total weight of the people on each team should be as nearly equal as possible.

Sample Input
1
3
100
90
200

Sample Output
190 200
Tomy has many paper squares. The side length (size) of them ranges from 1 to \(N - 1\), and he has an unlimited number of squares of each kind. But he really wants to have a bigger one – a square of size \(N\).

He can make such a square by building it up from the squares he already has. For example, a square of size 7 can be built from nine smaller squares as shown below:

There should be no empty space in the square, no extra paper outside the square, and the small squares should not overlap. Further, Tomy wants to make his square using the minimal number of possible squares. Can you help?

Input
The first line of the input contains a single integer \(T\) indicating the number of test cases. Each test case consists of a single integer \(N\), where \(2 \leq N \leq 50\).

Output
For each test case, print a line containing a single integer \(K\) indicating the minimal number of squares needed to build the target square. On the following \(K\) lines, print three integers \(x, y, l\) indicating the coordinates of top-left corner (1 \(\leq x, y \leq N\)) and the side length of the corresponding square.

You have unlimited squares, of which side length ranges from 1 to \(N-1\). How can you build a square of side length \(N\)?