11 Algorithm Design. Backtracking and Branch and Bound

11.1 Objectives

The learning objectives of this laboratory session are:

- To understand the challenges of implementing *backtracking* and *branch-and-bound* algorithm development methods.
- To get hands-on experience with implementing algorithms developed using the two mentioned methods.
- To develop critical thinking concerning implementation decisions algorithms which use any of the two approaches.

The outcomes for this session are:

- Improved skills in C programming.
- A clear understanding of backtracking and branch-and-bound algorithms.
- The ability to make implementation decisions concerning backtracking and branch-and-bound algorithms.
- The ability to decide when backtracking and branch-and-bound methods are suitable for solving a problem.

Backtracking

Backtracking is used in developing algorithms for problems where a number of sets, say n, are given. Let those sets be $S_1, S_2, \ldots, S_n$, where $n_i = |S_i|$ (the number of components of set S_i). We are required to find the elements of a vector $X = (x_1, x_2, \ldots, x_n) \in S = S_1 \times S_2 \times \ldots \times S_n$ such that a relation $\varphi(x_1, x_2, \ldots, x_n)$ holds for the elements of vector X. The relation φ is called an *internal* relation, the set $S = S_1 \times S_2 \times \ldots \times S_n$ is called the *space of possible solutions*, and the vector $X = (x_1, x_2, \ldots, x_n)$ is called a *result*.
Backtracking finds all the possible results of the problem. We can then pick up one result that satisfies an additional condition.
Backtracking eliminates the need to generate all the $\prod_{i=1}^{n} n_S$ possible solutions. To achieve that, when generating vector X we have to obey the following conditions:

1. x_k is assigned a value only if $x_1, x_2, \ldots, x_{k-1}$ have already been assigned values.
2. After x_k was assigned a value, check the *continuation* relation $\varphi(x_1, x_2, \ldots, x_k)$ to see if it makes sense to evaluate x_{k+1}. If condition $\varphi(x_1, x_2, \ldots, x_k)$ is not satisfied, then we pick up a new value for $x_k \in S_k$ and check φ again. If the set of choices for x_k becomes empty, we restart by selecting another value for x_{k-1} and so forth. This reduction in the value of k is the origin of the name of the method. It suggests that when no advance is possible, we back-track the sequence of the current solution.
   There is a strong relationship between the internal and the continuation condition. An optimal setting of the condition needed to continue greatly reduces the number of computations.

A non-recursive backtracking algorithm may be sketched as shown in Listing 11.1, and a recursive one as shown in Listing 11.2

Listing 11.1: A generic pseudocode of non-recursive backtracking.

```
#define MAXN ? /* ? = a suitable value */

void nonRecBackTrack(int n)
/* sets S_i and the corresponding number of elements in each set,
  n_S, are assumed global */
{
    int x[MAXN];
    int k, v;

    k=1;
    while (k > 0)
    {
        v = 0;
        while (∃ untested α ∈ S_k && v == 0)
        {
            x[k] = α;
            if (φ(x[1], x[2],..., x[k]))
                v = 1;
```

```
        }
        if (v == 0)
            k--;
        else
        if (k == n)
            listOrProcess(x, n)
            /* list or process solution */
        else
            k++;
    }
}
```

Listing 11.2: A generic pseudocode of non-recursive backtracking.

```
#define MAXN ? /* ? = a suitable value */
int x[MAXN];

/* n, the number of sets S_k, the sets S_k and the corresponding number of elements
   in each set, n(S_k), are assumed to be global */
void recBackTrack(int k)
{
    int j;

    for (j = 1; j <= nS[k]; j++)
    {
        x[k] = S_k[j];
        /* the j^th element of set S_k */
        if (φ(x[1], x[2],..., x[k]))
            if (k < n)
                recBackTrack(k + 1);
            else
              listOrProcess(x, n); /* list or process solution */
    }
}
```

To process the whole space invoke $recBackTrack(1)$.

An Example of a Backtracking Algorithm. The n Queens Placement Problem

The problem is stated as follows:

> Find all the arrangements of n queens on an $n \times n$ chessboard such that no queen would threaten another, i.e. no two queens are on the same line, column or diagonal.

As on each line there should be only one queen, the solution may be presented as a vector $X = (x_1, x_2, \ldots, x_n)$, where x_i is the column where a queen is placed on line i.
The continuation conditions are:

1. No two queens may be on the same column, i.e. $X[i] \neq X[j] \quad \forall i \neq j$
2. No two queens may be on the same diagonal, i.e. $|k - i| \neq |X[k] - X[i]| \quad$ for $i = 1, 2, \ldots, k-1$.

A non-recursive solution for the n Queens placement problem is shown in Listing 11.3. A recursive implementation is given in Listing 11.4.

Listing 11.3: A non-recursive implementation of the n Queens problem.

```
#include <stdio.h>
#include <stdlib.h>
#define MAXN 10

void nonRecQueens(int n)
/* find all possible arrangements of n queens on a chessboard such that no queen
   threatens another */
{
    int x[MAXN];
    int v;
    int i, j, k, solNb;

    solNb = 0;
    k = 1;
    x[k] = 0;
```

```
    while(k > 0)
    { /* find a valid arrangement on line k */
        v=0;
        while(v==0 && x[k] <= n - 1)
        {
            x[k]++;
            v = 1;
            i = 1;
            while (i <= k - 1 && v == 1)
                if (x[k] == x[i] || abs(k - i) == abs(x[k] - x[i]))
                    v=0;
                else
                    i++;
        }
        if (v == 0)
            k = k - 1;
        else
        {
            if (k== n)
            { /* display chessboard */
                solNb++;
                printf("\nSolution %d\n", solNb);
                for (i = 1; i <= n; i++)
                {
                    for (j = 1; j <= n; j++)
                        if (x[i] == j)
                            printf("1");
                        else
                          printf("0");
                        printf("\n");
                }
                while ('\n' != getchar());
            }
            else
            {
                k++;
                x[k]=0;
            }
        }
    }
}
int main(void)
{
  int n;

  printf("\nNumber of queens=");
  scanf("%d", &n);
    while ('\n' != getchar());
  nonRecQueens(n);
  printf("\nEND\n");
  return 0;
}
```

Listing 11.4: A recursive implementation of the n Queens problem.

```
#include <stdio.h>
#include <stdlib.h>
#define MAXN 10
int x[MAXN];
int n;          /* chessboard size */
int solNb;  /* solution number */
enum { FALSE=0, TRUE=1 };
/* function which checks the continuation conditions */
int phi(int k)
{
  for (int p = 1; p <= k - 1; p++)
    if (x[k] == x[p] || abs(k - p) == abs(x[k] - x[p]))
      return FALSE;
  return TRUE;
}
/* find all possible arrangements of n queens on a chessboard such that no queen
   threatens another */
void recQueens(int k)
{
```

```
  for (int j = 1; j <= n; j++)
  {
    x[k] = j;
    if (phi(k) == TRUE)
      if (k < n)
        recQueens(k + 1);
      else
      { /* list solution */
        solNb++;
        printf("\nSolution %d\n", solNb);
        for (int i = 1; i <= n; i++)
        {
          for (int p = 1; p <= n; p++)
            if (x[i] == p)
              printf("1");
            else
              printf("0");
          printf("\n");
        }
        while ('\n' != getchar());
      }
  }
}
int main(void)
{
  printf("\nNumber of queens=");
  scanf("%d", &n);
  while ('\n' != getchar());
  solNb = 0;
  recQueens(1);
  printf("\nEND\n");
  return 0;
}
```

Branch and Bound

The branch and bound approach is related to backtracking. What makes them somewhat different is the order of state space traversal and also the way they prune the subtrees which cannot lead to a solution.

Branch and bound applies to problems where the initial state, say s_0, and the final state, say s_f, are both known. In order to reach s_f from s_0, a series of decisions are made. We are interested in minimizing the number of intermediate states.
Let us assume that these states are nodes in a graph, and an edge indicates that a decision changed state s_i into state s_{i+1}. We have to impose a restriction on the nodes of the graph, that is "no two nodes may hold the same state", in order to prevent the graph to become infinite. Thus, the graph reduces to a tree. We then generate the tree up to the first occurrence of the node holding the final state.
The graph may also be traversed depth-first or breadth-first, but then the time needed to get a solution is longer. We can get a superior strategy by selecting the closest-to-final node out of the descendants of the current node (i.e. current state). We use a cost function c, defined on the tree nodes, to evaluate the distance to the final state. Based on the values supplied by this function, we can select a minimum cost node from the set of descendants of the current node. Such a traversal is called a *best-cost* traversal.
An ideal function used to measure the distance from one node to the final one is:

$$c(x) = \begin{cases} level(x) - 1 & \text{if } x = \text{final node} \\ +\infty & \text{if } x = \text{terminal node and } x \neq \text{final node} \\ \min c(y) & \text{if } x \neq \text{final node} \end{cases}$$

where y is a terminal node located in the subtree rooted at node x. But, the function c defined in this way is not applicable, because its computation needs traversing all the nodes in the tree − and this is what we would like to avoid. Still, we have to notice that if we choose to compute c, then descending the tree to the final node involves traversing an already known path, passing through the vertices x which have $c(x) = c(root)$. Because using c is unpractical, we define an approximation for it, in one of the following two ways:

1. $\hat{c}(x)$=level of vertex x + distance($current\ state$, $final\ state$), or
2. $\hat{c}(x)$=cost($parent(x)$) + distance($current\ state$, $final\ state$),

where by "level of vertex x" we mean the number of decisions made to reach the current configuration.
The form of the "distance" function is problem specific. E.g., for the game known as PERSPICO (see section 11.2), the distance is the number of tiles which are displaced.
Each solution is assumed to be expressible as an array x (as was seen in Backtracking). A predictor, called an approximate cost function $\hat{c}(x)$, is assumed to have been defined. A *live* node is a node that has not been expanded. A *dead* node is a node that has been expanded. The expanded node (or $eNode$ for short) is the live node with the best $\hat{c}$ value.

Listing 11.5: Pseudocode for generic branch-and-bound.

```
boolean branchAndBound()
{
    eNode = new(node);  // this is the root node which is the dummy start node
    H = createHeap(); // A heap for all the live nodes
    // H is a min-heap for minimization problems,
    // and a max-heap for maximization problems.
        while (true)
        {
            if (eNode is a final leaf)
            {  // eNode is an optimal solution
               print out the path from eNode to the root;
               return true;
            }
            expand(eNode);
            if (H is empty)
            { // report that there is no solution;
                        return false;
            }
            eNode = deleteTop(H);
   }
}
void expand(e)
{
  Generate all the children of e;
  Compute the approximate cost value ĉ of each child;
  Insert each child into the heap H;
}
```

11.2 Laboratory Assignments

Mandatory Assignments

Implement modular C programs having their input and output stored in files, with names given as command line arguments, which solve the following problems using branch and bound:

11.1. The game called PERSPICO. There are 15 tiles, numbered from 1 to 15, enclosed in a rectangular frame of size 4×4, and thus there is one empty position. Any tile adjacent to the empty position can be moved into this empty position. The game starts with an arbitrary initial distribution of the 15 tiles and the empty position. Figure 11.1 gives an example of initial and final configuration. I/O description. Input: 4 lines with 4 tile numbers each,

1		3	4
5	2	7	8
9	6	10	11
13	14	15	12

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Figure 11.1: Example positions for the game called PERSPICO.

representing a start position starting with the top row, followed by 4 lines of 4 numbers for the corresponding final position. Below is an example:

```
1␣␣␣0␣␣3␣␣4
5␣␣␣2␣␣7␣␣8
9␣␣␣6␣10␣11
13␣14␣15␣12

1␣␣␣2␣␣3␣␣4
5␣␣␣6␣␣7␣␣8
```

```
9␣␣10␣11␣12
13␣14␣15
```

Output: consecutive board positions in the same format as the input, separated by one line of hyphens ('−' character).

Extra Credit Problems

11.2. There are $2n$ natives on a river bank. Half of them (n), are man eaters. They wish to cross the river using a boat which can take at most k persons in one trip. If on either the bank or in the boat there are more cannibals than normal people, then the cannibals will eat the others. Find a way to take them all on the opposite bank without losing people to the cannibals and without changing numbers (i.e. use other people than the initial ones). Note that the boat needs at least one person to cross the river. Initially, all people are on the left bank.
I/O description. Input: n k, n = number of cannibals, k = boat capacity, e.g. for $n = 12$, and $k = 5$:

```
12␣5
```

Output (initial situation for the same values):

```
L:␣12c␣12n
B:␣0c␣0n
R:␣0c␣0n
```

Here `L` means left bank, `R` means right bank, and `B` means boat. Other letters mean `c`=cannibal, `n`=normal.

Optional Assignments

11.3. There are n cars[1] on a rail track, numbered with distinct values from the set $\{1, 2, \ldots, n\}$. A crane available at that location may pick at most k cars from the rail track and place them at the end of the sequence of cars (imagine this as being on the right hand side). Then, these cars are pushed to form a continuous line of cars.
Given the initial order of the cars, you are required to find (if possible) the minimum number of operations which the crane must execute to get the cars in sorted order, i.e $1, 2, \ldots, n$.
I/O description. Input: sequence of car numbers, separated by spaces all on one line. Output: number of operations.

11.4. There are n goats lined on a bridge, going all in the same direction. From the opposite direction, other n goats are coming towards them. The goats cannot go around each other, but each goat can skip over one single goat of the opposite group and can advance if there is an empty space ahead. Using these two possibilities for a move, make the goat groups cross the bridge.
I/O description. Input: number of goats in one line. Output: sequence of configurations, e.g. initially:

```
[0]␣Op:␣none
a5␣a4␣a3␣a2␣a1␣___␣b1␣b2␣b3␣b4␣b5
[1]␣Op:␣advance␣a1
a5␣a4␣a3␣a2␣___␣a1␣b1␣b2␣b3␣b4␣b5
[2]␣Op:␣jump␣b1
a5␣a4␣a3␣a2␣b1␣a1␣___␣b2␣b3␣b4␣b5
```

Here `[0]` is the step number, `a5 a4 a3 a2 a1` is the line of goats going from left to right ___ denotes an empty position, and `Op:` indicates the operation one of `none, advance, jump`.

[1] not automobiles!