8 Algorithm Design. Greedy Algorithms

8.1 Objectives

The learning objectives of this laboratory session are:

- To understand the challenges of implementing *greedy* algorithm development method.
- To get hands-on experience with implementing greedy algorithms.
- To develop critical thinking concerning implementation decisions algorithms which use a greedy approach.

The outcomes for this session are:

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

8.2 Brief Theory Reminder

Greedy Algorithm Development

A greedy algorithm tries to make a locally optimal choice at each stage with the hope of finding a global optimum. This method applies to problems where out of a set, say A, a subset B must be selected, such a way that it satisfies certain requirements. Once an element was selected, it is included in the final solution. Once excluded, it will never be examined again. This is the reason for which it has been called "greedy". The choice made by a greedy algorithm may depend on choices made so far, but not on future choices or all the solutions to the subproblem. It iteratively makes one greedy choice after another, reducing each given problem into a smaller one. In other words, a greedy algorithm never reconsiders its choices. This is what is called the *greedy choice property*. The method finds a single solution to the given problem.
In general, greedy algorithms have five components:

1. A *candidate set*, from which a solution is created
2. A *selection function*, which chooses the best candidate to be added to the solution
3. A *feasibility function*, that is used to determine if a candidate can be used to contribute to a solution
4. An *objective function*, which assigns a value to a solution, or a partial solution, and
5. A *solution function*, which will indicate when we have discovered a complete solution

Two variations of the method are generally used. The steps of one variant are:

1. Start with an empty B set.
2. Select an unselected element from set A. If there are no more unselected elements then stop.
3. If adding it to set B leads to a solution then add it to B. Else continue with step 2.

This could be sketched as shown in Listing 8.1. There, the function $select(A, B, i, x)$ implements the criterion used in obtaining the final solution.

Listing 8.1: A generic implementation of a greedy algorithm

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

/* A = set of nA candidate elements
   B = set of nB solution elements */
void greedy1(int A[MAXN], int nA, int B[MAXN], int *nB)
{
    int x, v, i;
    *nB = 0; /* empty solution set */
    for (i = 0; i < nA; i++)
    {
        select(A, B, i, x);
        /* select x, the first of A[i], A[i + 1], ..., A[n - 1],
           and swap it with element at position i */
        v = checkIfSolution(B, x);
        /* v = 1 if by adding x we get a solution and v = 0 otherwise */
        if (v == 1)
            add2Solution(B, x, *nB);
        /* add x to B, and increase the number of elements in B */
    }
}
```

The steps of another variant are:

1. Set the order used to consider the elements of set A.
2. Select one current element from set A. If there are no more elements to consider, then stop.
3. If including the current element into set B can lead to a possible solution then add it to set B. Otherwise continue with step 2.

A sketch of an implementation is shown in Listing 8.2.

Listing 8.2: Another generic implementation of a greedy algorithm

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

/* A = set of nA candidate elements
   B = set of nB solution elements */
void greedy2(int A[MAXN], int nA, int B[MAXN], int *nB)
{
    int x, v, i;
    *nB = 0; /* empty solution set */

    process(A, nA);  /* rearrange A */
    for (i = 0; i < nA; i++)
    {
        x = A[i];
        checkIfSolution(B, x, v);
        /* v = 1 if by adding x we get a solution and v = 0 otherwise */
        if (v == 1)
            add2Solution(B, x, *nB);
            /* add x to B, and increase the number of elements in B */
    }
}
```

An Example of a Greedy Algorithm: Prim's Algorithm

The minimum spanning tree problem is stated as follows:

Let $G = (V, E)$ be an undirected connected graph. Each edge $(i, j) \in E$ has a non-negative cost attached to it. We have to find a partially connected graph, say $A = (V, T)$, such that the sum of the costs of the edges in T to be a minimum. Note that this partial graph is the *minimum spanning tree*.

One solution is Prim's algorithm. The steps of this algorithm are:

1. Initially, include only one node in the tree T. (It does not matter which, so we can safely start with node 1.) The set of arcs in the tree is empty.
2. Select a minimum cost arc, having one end in the tree and the other in the rest of vertices. Repeat this step $n - 1$

times. To avoid passing through all arcs at each step, we can use an n-component vector v defined as:

$$U_i = \begin{cases} 0 & \text{if vertex } i \in T \\ k & \text{if vertex } i \notin T; \\ & k \in T \text{ is a node such that} \\ & (i,k) \text{ is a minimum cost edge} \end{cases}$$

Initially, $v[1] = 0$, and $v[2] = v[3] = \ldots = v[n] = 1$, i.e initially the tree is $A = (\{1\}, \emptyset)$.

Listing 8.3 shows an implementation.

Listing 8.3: An implementation of Prim's algorithm.

```
#include <stdio.h>
#include <limits.h>
#define MAXN ?  // ? = a suitable value for maximum number of nodes
#define INFTY INT_MAX

void Prim2(int n, int c[MAXN][MAXN], int e[MAXN][2], int *cost)
/* n = number of vertices;
   c = cost matrix;
   e = edges of the minimum spanning tree;
   cost = cost of the minimum spanning tree */
{
    int v[MAXN];
    /* v[i] = 0 if i is in the MST;
       v[i] = j if i is not in the MST;
       j is a node of the tree such that (i, j) is a minimum cost edge */
    int j, min;

    *cost = 0;
    v[1]=0;
    for (int i = 2; i <= n; i++)
        v[i] = 1; /* tree is ({1},{}) */
    /* find the rest of edges */
    for (int i = 1; i <= n - 1; i++)
    { /* find an edge to add to the tree */
        min = INFTY;
        for (int k  = 1; k <= n; k++)
            if (v[k] != 0)
                if (c[k][v[k]] < min)
                {
                    j = k;
                    min = c[k][v[k]];
                }
        e[i][0] = v[j];
        e[i][1] = j;
        *cost += c[j][v[j]];
        /* update vector v */
        v[j]= 0;
        for (int k = 1; k <= n; k++)
            if (v[k] != 0 &&
                    c[k][v[k]] > c[k][j])
                v[k] = j;
    }
}
int main(int argc, char *argv[])
{
    int n; /* number of nodes */
    int c[MAXN][MAXN    ]; /* costs */
    int e[MAXN][2]; /* tree edges */
    int i, j, k, cost;

    printf("\nNumber of nodes in graph G: ");
    scanf("%d", &n);
    while ('\n' != getchar());
    for (i = 1; i <= n; i++)
        for (j = 1; j <= n; j++)
            c[i][j] = INFTY;
    /* read cost matrix (integers) */
    for (i = 1; i < n; i++)
    {
        do
        {
```

```
            printf(
            "\nNode adjacent to %2d [0=finish]:",
                            i);
            scanf("%d", &j);
            while ('\n' != getchar());
            if (j > 0)
            {
                printf("\nCost c[%d][%d]:", i, j);
                scanf("%d", &c[i][j]);
        while ('\n' != getchar());
               c[j][i] = c[i][j];
                /* c is symmetric */
            }
        }
        while (j > 0);
    }
    Prim2(n, c, e, &cost);
    printf("\nThe cost of MST is %d", cost);
    printf("\nTree edges\tEdge cost\n");
    for (i = 1; i <= n - 1; i++)
        printf("%2.2d - %2.2d\t%10d\n",
            e[i][0], e[i][1],
            c[e[i][0]][e[i][1]]);
  return 0;
}
```

Another Example of a Greedy Algorithm: A Job Sequencing Problem

In this problem, an array of jobs is given, where every job has a deadline and associated profit if the job is finished before the deadline. Every job takes at least a single unit of time, so the minimum possible deadline for any job is 1. How to maximize total profit if only one job can be scheduled at a time?

Here are some examples:

```
Input: Four Jobs with following deadlines and profits
  JobID    Deadline      Profit
    a         4            20
    b         1            10
    c         1            40
    d         1            30
Output: Following is maximum profit sequence of jobs
        c, a

Input:  Five Jobs with following deadlines and profits
   JobID     Deadline     Profit
     a         2           100
     b         1           19
     c         2           27
     d         1           25
     e         3           15
Output: Following is maximum profit sequence of jobs
        c, a, e
```

A greedy algorithm to solve this problem is:

1. Sort all n jobs in decreasing order of their profit.
2. Initialize the result sequence as the first job in sorted jobs.
3. Repeat for the remaining $n - 1$ jobs
   If the current job can fit in the current result sequence without missing the deadline,
   add current job to the result.
   Else
   ignore the current job.

An implementation of the greedy job scheduling algorithm is shown in Listing 8.4

Listing 8.4: An implementation of Greedy Job scheduling Algorithm.

```
#include <stdio.h>
#include <stdlib.h>

typedef enum {false, true} BooleanT;
// A structure to represent a job
typedef struct
{
    char id;      // Job Id
    int deadline;     // Deadline of job
    int profit;  // Profit if job is over before or on deadline
} JobT;

// This function is used for sorting all jobs according to profit
int compareJobs(const void *a, const void *b)
{
    return ((JobT *)a)->profit < ((JobT *)b)->profit;
}

int min(int a, int b)
{
    return a > b? b: a;
}
// Returns minimum number of platforms reqquired
void printJobScheduling(JobT arr[], int n)
{
    // Sort all jobs according to decreasing order of prfit

    qsort (arr, n, sizeof (JobT), compareJobs);
    int result[n]; // To store result (Sequence of jobs)
    BooleanT slot[n];  // To keep track of free time slots

    // Initialize all slots to be free
    for (int i = 0; i < n; i++)
        slot[i] = false;

    // Iterate through all given jobs
    for (int i = 0; i < n; i++)
    {
        // Find a free slot for this job (Note that we start
        // from the last possible slot)
        for (int j = min(n, arr[i].deadline)-1; j >= 0; j--)
        {
            // Free slot found
            if (slot[j] == false)
            {
                result[j] = i;  // Add this job to result
                slot[j] = true; // Make this slot occupied
                break;
            }
        }
    }

    // Print the result
    for (int i=0; i<n; i++)
        if (slot[i])
            printf("%c ", arr[result[i]].id);;
}

int main()
{
    JobT arr[5] = { {'a', 2, 100}, {'b', 1, 19}, {'c', 2, 27},
        {'d', 1, 25}, {'e', 3, 15}
    };
    int n = sizeof(arr)/sizeof(arr[0]);
    puts("The maximum profit sequence of jobs:");
    printJobScheduling(arr, n);
    return 0;
}
```

8.3 Laboratory Assignments

Mandatory Assignments

Solve the following problem using the greedy approach:

8.3.1. The knapsack problem. The maximum weight which can be carried in a knapsack is W. There are n different items, each of weight w_i and value v_i. Find what amount of each item can be packed into the knapsack such a way that the maximum value is carried. Two cases should be considered:

a) An item may be taken only as a whole (i.e. a can).
b) A supply can be partly included (e.g. loafs of bread).

I/O description. Input: First line: number of items, n, and the knapsack capacity, W, i.e nW, separated by one space. Next n lines: pairs of $w_i v_i$ (weight of item i, a space, value of item i).
Output: two lines with total weight, a space and total value, for the 0-1, and for the fractional problem, respectively.

8.3.2. The activity problem. A set $S = 1, 2, ..., n$ of activities using the same resource is given (e.g. using the same room). The resource may be used by a single activity at any one moment. An activity, say i, has a start time t_{s_i} and an finish time t_{f_i}, where $t_{s_i} < t_{f_i}$. An activity i, if chosen, lasts for $[t_{s_i}, t_{f_i})$. Activities i and j are compatible if their time spans do not intersect. Select a maximal set o mutually compatible activities.
I/O description. Input. First line: number of activities, n. Next n lines: pairs of s_i f_i, one pair per line, with the two values separated by one space.
Output: the list of scheduled activities, on one line, separated by one space. (Remember that activities are identified by strictly positive natural numbers.)

Extra Credit Assignments

8.3.3. You want to repaint your house entirely for an upcoming occasion. The total area of your house is D units. There are a total of N workers. The i^{th} worker has his available time T_i, hiring cost X_i and "speed" Y_i. This means that he is available for hiring from time T_i and remains available ever since. Once available, you can hire him with cost X_i, after which he will start painting the house immediately, covering exactly Y_i units of house with paint per time unit. You may or may not hire a worker and can also hire or fire him at any later point of time. However, no more than 1 worker can be painting the house at a given time. Since you want the work to be done as fast as possible, figure out a way to hire the workers, such that your house gets painted at the earliest possible time, with minimum cost to spend for hiring workers. Note: You can hire a previously hired worker without paying him again. I/O description. Input. The first line of input contains two integers with values for N and D, separated by one space. The i^{th} line of the next N contains data for the i^{th} worker: three integers, representing $T_i X_i Y_i$.
Output one integer, the *minimum cost* that you can spend in order to get your house painted at the earliest. Sample input:

```
3 3
1 1 1
2 2 2
3 1 5
```

Output for sample input:

```
3
```

Explanation. We can hire the first worker at cost of 1 and second at cost of 2, to finish painting at the minimum time of exactly 2 time units.

Optional Assignments

8.3.4. Consider m vectors, $V_1, V_2, \ldots, V_m$, of $n_1, n_2, \ldots, n_m$ elements respectively, with elements sorted ascending, based on a key. These vectors are then merged, obtaining another vector of length $n_1 + n_2 + \ldots + n_m$, also sorted ascending. It is known that merging two vectors of n_1 and n_2 elements respectively requires time proportional to their lengths. Determine the optimal order for merging all given vectors.
I/O description. Input: first line contains the number of vectors, m. Next m lines: first element on each line is the number of elements in the current vector, n_i; then n_i elements of vector i follow on the same line;the separator is one space.
Output: A list of vector ids' pairs (numbers identifying vectors).

8.3.5. You are a product engineer and would like to improve the quality of duct tapes that your company manufactures. An entire tape can be represented as a single row of N cells. Each cell has its *stickiness* factor, which stands for its ability to stick to an object. We say that a tape is a good quality product, if and only if the total sum of *stickiness* factors of grid cells in any of its subarray of size K is at least D.
To make a quality product, consider an augment operation in which you can choose any subarray of size at most K and a real number (say R), and multiply *stickiness* factor of all its cells by R. For each augment operation, the subarray and the value of R can be chosen arbitrarily. However, the size of chosen subarray for augmentation can be at most K, as mentioned before. Your task is to calculate the minimum number of augment operations needed to be performed in order to transform the given tape to a good quality product.
I/O description. Input. The first line contains three space separated integers N, K and D, as described above. The next line contains N space separated integers, where the i^{th} integer (denoted by A_i) is the stickiness factor of the i^{th} cell.
Output. An integer, which denotes the minimum number of augment operations needed to be performed in order to transform the tape to a good quality product. In case, if it is impossible to achieve, print -1 instead. Sample input.

```
3 2 4
1 1 1
```

Output for sample input.

```
1
```

Explanation. We can multiply the second element by 3. Now the tape becomes: `1 3 1`. Any subarray of size 2 has now sum = 4, which satisfies the required condition. We used 1 augment operation, hence the answer is 1.