Ayush Goyal

190905522 CSE D 62

DAA Lab 4 (Week 4) BRUTE FORCE TECHNIQUE – II

Q1) Write a program for assignment problem by brute-force technique and analyse its time efficiency. Obtain the experimental result of order of growth and plot the result.

CODE:

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

void swap(int *arr, int i, int j)
{
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}
int findCost(int **mat, int num, int *arr, int *cnt)
{
    int result = 0;
    for (int i = 0; i < num; i++)
    {
        (*cnt)++;
        result += mat[i][arr[i]];
    }
    return result;
}

int fact(int num)
{
    int result = 1;
    for (int i = 1; i <= num; i++)
    {
        result *= i;
    }
    return result;
}

int getCeil(int *current, int first, int l, int h)
{
    int ceilindex = l;
    for (int i = l + 1; i <= h; i++)
        if (current[i] > first && current[i] < current[ceilindex])
            ceilindex = i;
```

```c
    return ceilindex;
}

void reverse(int *current, int start, int end)
{
    while (start < end)
    {
        swap(current, start, end);
        start++;
        end--;
    }
}

void lexicoNext(int *current, int num)
{
    int i;
    for (i = num - 2; i >= 0; --i)
        if (current[i] < current[i + 1])
            break;
    int ceilindex = getCeil(current, current[i], i + 1, num - 1);
    swap(current, i, ceilindex);
    reverse(current, i + 1, num - 1);
}

int solve(int **mat, int num)
{
    int *current = (int *)calloc(num, sizeof(int));
    int *best = (int *)calloc(num, sizeof(int));
    for (int i = 0; i < num; i++)
    {
        current[i] = i;
    }
    int iterate = fact(num);
    int min = __INT_MAX__;
    int temp, cnt = 0;
    while (iterate--)
    {
        temp = findCost(mat, num, current,&cnt);
        if (temp < min)
        {
            min = temp;
            for (int i = 0; i < num; i++)
            {
                best[i] = current[i];
            }
        }
        lexicoNext(current, num);
```

```c
    }
    printf("Minimum cost is : %d\nThe jobs assigned from person 1 to %d : ", min, num);
    for (int i = 0; i < num; i++)
    {
        printf("%d ", best[i] + 1);
    }
    printf("\n");
    return cnt;
}

int main()
{
    int num;
    printf("Enter the number of jobs and people : ");
    scanf("%d", &num);
    printf("Enter the adjacency matrix : \n");
    int **mat = (int **)calloc(num, sizeof(int *));
    for (int i = 0; i < num; i++)
    {
        mat[i] = (int *)calloc(num, sizeof(int));
        for (int j = 0; j < num; j++)
        {
            scanf("%d", &mat[i][j]);
        }
    }
    int count = solve(mat, num);
    printf("The number of operations is : %d\n", count);
    return 0;
}
```

Input/Output:

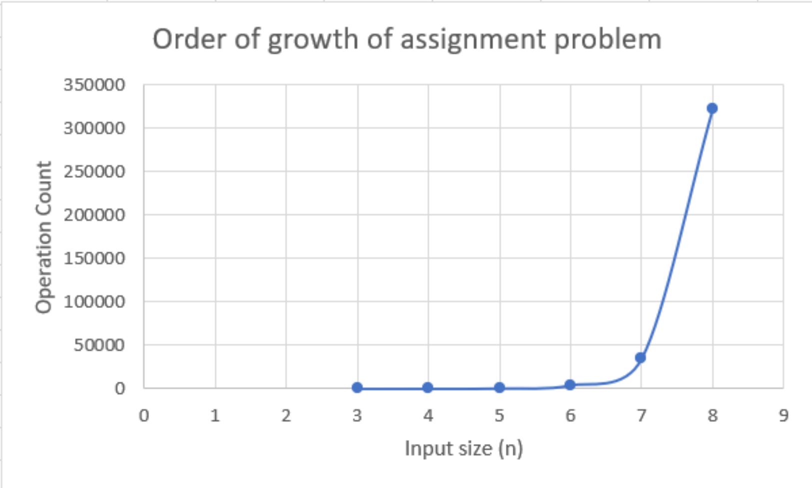
```
D:\CSE\DAA Lab\Week 4>gcc assignmentproblem.c -o assignmentproblem

D:\CSE\DAA Lab\Week 4>assignmentproblem
Enter the number of jobs and people : 4
Enter the adjacency matrix :
10 3 8 9
7 5 4 8
6 9 2 9
8 7 10 5
Minimum cost is : 17
The jobs assigned from person 1 to 4 : 2 1 3 4
The number of operations is : 96

D:\CSE\DAA Lab\Week 4>
```

Time Efficiency Analysis:

The order of growth of the assignment algorithm will be **O(n*n!)**. It can be seen from the graph where we plot the operation count vs the input size. Basic Operation is taken as addition in the loop which calculates cost for all n! cases possible.

Graph and Table:

Input Siz	Operation Coun
3	18
4	96
5	600
6	4320
7	35280
8	322560



Q2) Write a program for depth-first search of a graph. Identify the push and pop order of vertices.

CODE:

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

int isEmpty(int top)
{
    if (top == -1)
    {
        return 1;
    }
    return 0;
}

int isFull(int top, int size)
{
```

```c
    if (top == size - 1)
    {
        return 1;
    }
    return 0;
}

void push(int **Stack, int *top, int *size, int key)
{
    if (isFull(*top, *size))
    {
        *Stack = (int *)realloc(*Stack, sizeof(int) * (*size) * 2);
        *size *= 2;
    }
    (*top)++;
    (*Stack)[*top] = key;
}

int pop(int **Stack, int *top)
{
    int temp = (*Stack)[*top];
    (*top)--;
    return temp;
}

void display(int *Stack, int top)
{
    if (isEmpty(top))
    {
    }
    else
    {
        printf("stack : ");
        int i;
        for (i = 0; i <= top; i++)
        {
            printf("%d ", *(Stack + i));
        }
        printf("\n");
    }
}

void insertEdgeM(int **matrix, int first, int second)
{
    matrix[first][second] = 1;
    matrix[second][first] = 1;
}
```

```c
void dispM(int **matrix, int n)
{
    for (int i = -1; i < n; i++)
    {
        if (i != -1)
            printf("%d -> ", i);
        for (int j = 0; j < n; j++)
        {
            if (i == -1)
            {
                if (j == 0)
                {
                    printf("\t");
                }
                printf("%d\t", j);
                continue;
            }
            if (j == 0)
            {
                printf("\t");
            }
            printf("%d\t", matrix[i][j]);
        }
        printf("\n");
    }
}

void DFS(int **matrix, int num, int **Stack, int *top, int *size)
{
    int *visited = (int *)calloc(num, sizeof(int));
    for (int i = 0; i < num; i++)
    {
        visited[i] = 0;
    }
    char result[100];
    int resindex = 0;
    char popped[100];
    int popindex = 0;
    push(Stack, top, size, 0);
    printf("pushed : %d\n", 0);
    char p = (char)('0' + 0);
    result[resindex++] = p;
    display(*Stack, *top);
    visited[0] = 1;
    int cur = *Stack[*top];
    int flag, ele;
    while (1)
    {
```

```
        if(!(isEmpty(*top))){
            flag = 0;
            for (int i = 0; i < num; i++)
            {
                if (visited[i] == 0 && matrix[cur][i] == 1)
                {
                    visited[i] = 1;
                    printf("pushed : %d\n", i);
                    p = (char)('0' + i);
                    result[resindex++] = p;
                    push(Stack, top, size, i);
                    display(*Stack, *top);
                    flag = 1;
                    break;
                }
            }
            if (flag == 0)
            {
                ele = pop(Stack, top);
                p = (char)('0' + ele);
                popped[popindex++] = p;
                printf("popped : %d\n", ele);
                display(*Stack, *top);
            }
            cur = (*Stack)[*top];
        }
        else{
            flag = 1;
            for (int i = 0; i < num && flag; i++){
                if(visited[i]==0){
                    visited[i] = 1;
                    printf("pushed : %d\n", i);
                    p = (char)('0' + i);
                    result[resindex++] = p;
                    push(Stack, top, size, i);
                    display(*Stack, *top);
                    flag = 0;
                    cur = (*Stack)[*top];
                    break;
                }
            }
            if(flag == 1){
                break;
            }
        }

    }
    while (!(isEmpty(*top)))
```

```c
    {
        int rem = pop(Stack, top);
        p = (char)('0' + rem);
        popped[popindex++] = p;
        printf("popped : %d\n", rem);
        display(*Stack, *top);
    }
    printf("The DFS is : ");
    for (int i = 0; i < resindex; i++)
    {
        printf("%c ", result[i]);
    }
    printf("\nThe pop order is : ");
    for (int i = 0; i < popindex; i++)
    {
        printf("%c ", popped[i]);
    }
    printf("\n");
}

int main()
{
    int num = 9;
    int **matrix = (int **)calloc(num, sizeof(int *));
    for (int i = 0; i < num; i++)
    {
        matrix[i] = (int *)calloc(num, sizeof(int));
        for (int j = 0; j < num; j++)
        {
            matrix[i][j] = 0;
        }
    }
    int m,n;
    do{
        printf("\nEnter edges to be joined : ");
        scanf("%d %d",&m,&n);
        if(m!=-1 && n!=-1)
            insertEdgeM(matrix, m, n);
    }while(m!=-1);
    dispM(matrix, num);
    int top = -1, size = 2;
    int *Stack = (int *)calloc(size, sizeof(int));
    DFS(matrix, num, &Stack, &top, &size);
    return 0;
}
```

Input/Output:

```
D:\CSE\DAA Lab\Week 4>gcc dfs.c -o dfs

D:\CSE\DAA Lab\Week 4>dfs

Enter edges to be joined : 0 1

Enter edges to be joined : 1 7

Enter edges to be joined : 7 2

Enter edges to be joined : 2 3

Enter edges to be joined : 0 3

Enter edges to be joined : 0 8

Enter edges to be joined : 8 4

Enter edges to be joined : 4 3

Enter edges to be joined : 2 5

Enter edges to be joined : 5 6

Enter edges to be joined : -1 -1
        0       1       2       3       4       5       6       7       8
0 ->    0       1       0       1       0       0       0       0       1
1 ->    1       0       0       0       0       0       0       1       0
2 ->    0       0       0       1       0       1       0       1       0
3 ->    1       0       1       0       1       0       0       0       0
4 ->    0       0       0       1       0       0       0       0       1
5 ->    0       0       1       0       0       0       1       0       0
6 ->    0       0       0       0       0       1       0       0       0
7 ->    0       1       1       0       0       0       0       0       0
8 ->    1       0       0       0       1       0       0       0       0
pushed : 0
stack : 0
pushed : 1
stack : 0 1
pushed : 7
stack : 0 1 7
pushed : 2
stack : 0 1 7 2
pushed : 3
stack : 0 1 7 2 3
pushed : 4
stack : 0 1 7 2 3 4
pushed : 8
stack : 0 1 7 2 3 4 8
```

```
popped : 8
stack : 0 1 7 2 3 4
popped : 4
stack : 0 1 7 2 3
popped : 3
stack : 0 1 7 2
pushed : 5
stack : 0 1 7 2 5
pushed : 6
stack : 0 1 7 2 5 6
popped : 6
stack : 0 1 7 2 5
popped : 5
stack : 0 1 7 2
popped : 2
stack : 0 1 7
popped : 7
stack : 0 1
popped : 1
stack : 0
popped : 0
The DFS is : 0 1 7 2 3 4 8 5 6
The pop order is : 8 4 3 6 5 2 7 1 0

D:\CSE\DAA Lab\Week 4>
```

Time Efficiency Analysis:

We have used the Depth First Search algorithm using the adjacency matrix. We take a graph with “v” vertices. Time spent in inserting and deleting items from the stack is O(1) Number of vertices is v therefore, O(v).
Since we iterate over v vertices again for every vertex v to check if it is adjacent or not, we can conclude that the order of growth is **$O(|v|^2)$**

Q3) Write a program for breadth-first search of a graph.

CODE:

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

int isEmpty(int front, int rear)
{
    if (front == rear)
    {
        return 1;
    }
    return 0;
}

int isFull(int front, int rear, int maxsize)
{
    if ((rear + 1) % maxsize == front)
    {
        return 1;
    }
    return 0;
}

void display(int *Queue, int front, int rear, int maxsize)
{
    if (isEmpty(front, rear))
    {
    }
    else
    {
        printf("queue : ");
        for (int i = (front + 1) % maxsize; i != (rear + 1) % maxsize; i = (i
+ 1) % maxsize)
        {
            printf("%d ", Queue[i]);
        }
        printf("\n");
    }
}

void push(int **Queue, int *front, int *rear, int *maxsize, int key)
{
    if (isFull(*front, *rear, *maxsize))
    {
        int *newQueue = (int *)calloc((*maxsize) * 2, sizeof(int));
```

```
        int i, j;
        for (i = 1, j = (*front + 1) % (*maxsize); j != (*rear + 1) % (*maxsiz
e); j = (j + 1) % (*maxsize), i++)
        {
            newQueue[i] = (*Queue)[j];
        }
        int *temp = *Queue;
        *Queue = newQueue;
        *maxsize = *maxsize * 2;
        *front = 0;
        *rear = --i;
        free(temp);
    }
    *rear = (*rear + 1) % (*maxsize);
    (*Queue)[*rear] = key;
}

int pop(int **Queue, int *front, int *rear, int *maxsize)
{
    int temp = (*Queue)[(*front + 1) % (*maxsize)];
    *front = (*front + 1) % (*maxsize);
    return temp;
}
void insertEdge(int **matrix, int first, int second)
{
    matrix[first][second] = 1;
    matrix[second][first] = 1;
}

void dispM(int **matrix, int n)
{
    for (int i = -1; i < n; i++)
    {
        if (i != -1)
            printf("%d -> ", i);
        for (int j = 0; j < n; j++)
        {
            if (i == -1)
            {
                if (j == 0)
                {
                    printf("\t");
                }
                printf("%d\t", j);
                continue;
            }
            if (j == 0)
            {
```

```c
                printf("\t");
            }
            printf("%d\t", matrix[i][j]);
        }
        printf("\n");
    }
}

void BFS(int **matrix, int num, int **Queue, int *front, int *rear, int *maxsi
ze)
{
    int *visited = (int *)calloc(num, sizeof(int));
    for (int i = 0; i < num; i++)
    {
        visited[i] = 0;
    }
    char result[100];
    int resultIndex = 0;
    char popped[100];
    int poppedIndex = 0;
    push(Queue, front, rear, maxsize, 0);
    printf("pushed : %d\n", 0);
    char p = (char)('0' + 0);
    result[resultIndex++] = p;
    display(*Queue, *front, *rear, *maxsize);
    visited[0] = 1;
    int cur = *Queue[*front];
    int flag, ele;
    while (1)
    {
        if (!(isEmpty(*front, *rear)))
        {
            for (int i = 0; i < num; i++)
            {
                if (visited[i] == 0 && matrix[cur][i] == 1)
                {
                    visited[i] = 1;
                    char p = (char)('0' + i);
                    result[resultIndex++] = p;
                    push(Queue, front, rear, maxsize, i);
                    printf("Pushed : %d\n", i);
                    display(*Queue, *front, *rear, *maxsize);
                }
            }
            ele = pop(Queue, front, rear, maxsize);
            p = (char)('0' + ele);
            popped[poppedIndex++] = p;
            printf("Popped : %d\n", ele);
```

```c
            display(*Queue, *front, *rear, *maxsize);
            cur = (*Queue)[(*front + 1) % (*maxsize)];
        }
        else
        {
            flag = 1;
            for (int i = 0; i < num && flag; i++)
            {
                if (visited[i] == 0)
                {
                    visited[i] = 1;
                    printf("Pushed : %d\n", i);
                    p = (char)('0' + i);
                    result[resultIndex++] = p;
                    push(Queue, front, rear, maxsize, i);
                    display(*Queue, *front, *rear, *maxsize);
                    cur = (*Queue)[(*front + 1) % (*maxsize)];
                    flag = 0;
                    break;
                }
            }
            if (flag == 1)
            {
                break;
            }
        }
    }
    while (!(isEmpty(*front, *rear)))
    {
        ele = pop(Queue, front, rear, maxsize);
        p = (char)('0' + ele);
        popped[poppedIndex++] = p;
        printf("Popped : %d\n", ele);
        display(*Queue, *front, *rear, *maxsize);
    }
    printf("The BFS is : ");
    for (int i = 0; i < resultIndex; i++)
    {
        printf("%c ", result[i]);
    }
    printf("\nThe pop order is : ");
    for (int i = 0; i < poppedIndex; i++)
    {
        printf("%c ", popped[i]);
    }
    printf("\n");
}
```

```c
int main()
{
    int num = 9;
    int **matrix = (int **)calloc(num, sizeof(int *));
    for (int i = 0; i < num; i++)
    {
        matrix[i] = (int *)calloc(num, sizeof(int));
        for (int j = 0; j < num; j++)
        {
            matrix[i][j] = 0;
        }
    }
    int m,n;
    do{
        printf("\nEnter edges to be joined : ");
        scanf("%d %d",&m,&n);
        if(m!=-1 && n!=-1)
            insertEdge(matrix, m, n);
    }while(m!=-1);
    dispM(matrix, num);
    int front = 0, rear = 0, maxsize = 2;
    int *Queue = (int *)calloc(maxsize, sizeof(int));
    BFS(matrix, num, &Queue, &front, &rear, &maxsize);
    return 0;
}
```

Input/Output:

```
D:\CSE\DAA Lab\Week 4>gcc bfs.c -o bfs

D:\CSE\DAA Lab\Week 4>bfs

Enter edges to be joined : 0 1

Enter edges to be joined : 1 7

Enter edges to be joined : 2 7

Enter edges to be joined : 2 3

Enter edges to be joined : 0 3

Enter edges to be joined : 0 8

Enter edges to be joined : 4 8

Enter edges to be joined : 4 3

Enter edges to be joined : 2 5

Enter edges to be joined : 5 6

Enter edges to be joined : -1 -1
        0       1       2       3       4       5       6       7       8
0 ->    0       1       0       1       0       0       0       0       1
1 ->    1       0       0       0       0       0       0       1       0
2 ->    0       0       0       1       0       1       0       1       0
3 ->    1       0       1       0       1       0       0       0       0
4 ->    0       0       0       1       0       0       0       0       1
5 ->    0       0       1       0       0       0       1       0       0
6 ->    0       0       0       0       0       1       0       0       0
7 ->    0       1       1       0       0       0       0       0       0
8 ->    1       0       0       0       1       0       0       0       0
pushed : 0
queue : 0
Pushed : 1
queue : 0 1
Pushed : 3
queue : 0 1 3
Pushed : 8
queue : 0 1 3 8
Popped : 0
queue : 1 3 8
Pushed : 7
queue : 1 3 8 7
Popped : 1
queue : 3 8 7
```

```
Pushed : 2
queue : 3 8 7 2
Pushed : 4
queue : 3 8 7 2 4
Popped : 3
queue : 8 7 2 4
Popped : 8
queue : 7 2 4
Popped : 7
queue : 2 4
Pushed : 5
queue : 2 4 5
Popped : 2
queue : 4 5
Popped : 4
queue : 5
Pushed : 6
queue : 5 6
Popped : 5
queue : 6
Popped : 6
The BFS is : 0 1 3 8 7 2 4 5 6
The pop order is : 0 1 3 8 7 2 4 5 6

D:\CSE\DAA Lab\Week 4>
```

Time Efficiency Analysis:

We have used the Breadth First Search algorithm using the adjacency matrix. We take a graph with “v” vertices. Time spent in inserting and deleting items from the queue is O(1).
Number of vertices is v therefore, O(v).
Since we iterate over v vertices again for every vertex v to check if it is adjacent or not, we can conclude that the order of growth is **$O(|v|^2)$.**

THE END