#### **FINAL REPORT : Algorithms Final Lab**

**NAME** : RAHUL THAPAR

ID : 1410110321

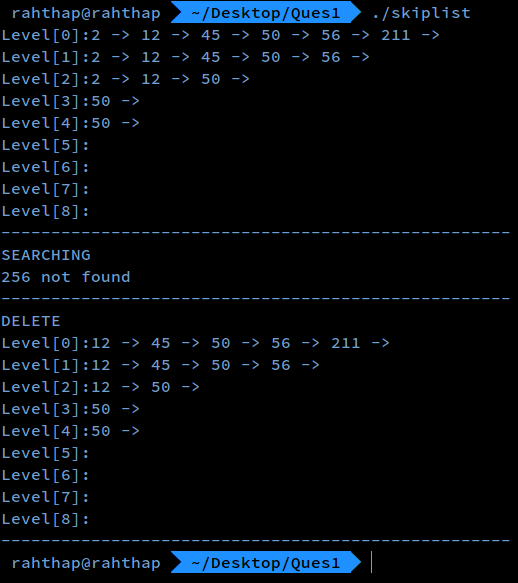
DATE : 20-April 2017

**LIST**

|  |  |
| --- | --- |
| [Ques 1](#Ques1) | Completed |
| [Ques 2](#Ques2) | Completed |
| [Ques 3](#Ques3) | Partially Completed |
| [Ques 4](#Ques4) | Completed |
| [Ques 5](#Ques5) | Completed |

**QUES 1**

Data Structure with log(n) complexity : SKIP LIST



CODE

#include "skiplist.h"

static node \*create\_node(int level, int key, object \*obj){

int i;

node \*nd = (node \*)malloc(sizeof(node) + level \* sizeof(node \*));

nd->obj = obj;

nd->key = key;

for (i = 0; i < level; i++) {

nd->forward[i] = NULL;

}

return nd;

}

skiplist \*create\_skiplist(void){

skiplist \*sl = (skiplist \*)malloc(sizeof(skiplist));

sl->head = create\_node(MAX\_LEVEL, 0, NULL);

sl->level = 1;

return sl;

}

static void free\_node(node \*nd){

free(nd);

}

void free\_skiplist(skiplist \*sl){

node \*nd, \*next;

nd = sl->head->forward[0];

free\_node(sl->head);

while (nd) {

next = nd->forward[0];

free\_node(nd);

nd = next;

}

free(sl);

}

static int random\_level(){

int level = 1;

while ((rand() & 0xFFFF) < (0.5 \* 0xFFFF)) {

level += 1;

}

return (level < MAX\_LEVEL) ? level : MAX\_LEVEL;

}

void insert(skiplist \*sl, int key, object \*obj){

node \*update[MAX\_LEVEL];

node \*nd;

int i, level;

nd = sl->head;

for (i = sl->level - 1; i >= 0; i--) {

while (nd->forward[i] != NULL && nd->forward[i]->key < key)

nd = nd->forward[i];

update[i] = nd;

}

level = random\_level();

if (level > sl->level) {

for (i = sl->level; i < level; i++) {

update[i] = sl->head;

}

sl->level = level;

}

nd = create\_node(level, key, obj);

for (i = 0; i < level; i++) {

nd->forward[i] = update[i]->forward[i];

update[i]->forward[i] = nd;

}

}

static void delete\_node(skiplist \*sl, node \*nd, node \*\*update){

int i;

for (i = 0; i < sl->level; i++) {

if (update[i]->forward[i] == nd) {

update[i]->forward[i] = nd->forward[i];

}

}

for (i = i - 1; i >= 0; i--) {

if (sl->head->forward[i] == NULL) {

sl->level--;

}

}

}

void delete(skiplist \*sl, int key){

node \*update[MAX\_LEVEL], \*nd;

int i;

nd = sl->head;

for (i = sl->level - 1; i >= 0; i--) {

while (nd->forward[i] && nd->forward[i]->key < key) {

nd = nd->forward[i];

}

update[i] = nd;

}

nd = nd->forward[0];

if (nd && nd->key == key) {

delete\_node(sl, nd, update);

free\_node(nd);

}

}

node \*find(skiplist \*sl, int key){

node \*nd;

int i;

nd = sl->head;

for (i = sl->level - 1; i >= 0; i--) {

while (nd->forward[i] != NULL) {

if (nd->forward[i]->key < key)

nd = nd->forward[i];

else if (nd->forward[i]->key == key)

return nd->forward[i];

else

break;

}

}

printf(" %d not found\n", key);

return NULL;

}

void print(skiplist \*sl){

node \*nd;

int i;

for (i = 0; i <= MAX\_LEVEL; i++) {

nd = sl->head->forward[i];

printf("Level[%d]:", i);

while (nd) {

printf("%d -> ", nd->key);

nd = nd->forward[i];

}

printf("\n");

}

}

#### 

#### **Ques 2**

* Generate a random graph with n > 10 nodes.
* Generate random n^2/2 directly edges.
* Assign a weight in the range (1, 10).
* Apply Dijkstra’s algorithm.
* Find out the single source shortest path.

##### **Generate Random Graph**

Graph::Graph(**int** n) : number\_of\_vertices(n)

{

connectivity\_matrix = **new** **float** \*[number\_of\_vertices];

**int** i;

**for** (i = 0; i < number\_of\_vertices; i++)

{

connectivity\_matrix[i] = **new** **float**[number\_of\_vertices];

}

}

Graph::~Graph()

{

**int** i;

open\_set.clear();

closed\_set.clear();

**for** (i = 0; i < number\_of\_vertices; i++)

{

**delete** [] connectivity\_matrix[i];

}

**delete** [] connectivity\_matrix;

connectivity\_matrix = NULL;

}

**##### Initializing Graph**

**void** Graph::initiate\_graph(**float** d, **float** min, **float** max)

{

**if** (NULL == connectivity\_matrix){

cout << "Memory Allocate Failed!" <<endl;

**return** ;

}

**if** ((min <= 0.0) || (max <= 0.0) || (min >= max)){

cout << "Invalid Edge Cost Range!" <<endl;

**return** ;

}

graph\_density = d;

**int** i, j;

**float** random;

srand((**unsigned** **int**)time(NULL));

**for** (i = 0; i < number\_of\_vertices; i++)

{

connectivity\_matrix[i][i] = 0.0; *//the cost from i to i is 0*

**for** (j = i + 1; j < number\_of\_vertices; j++)

{

random = random\_generator(0.0, 1.0); *//get a decimal between 0 and 1*

**if** (random >= graph\_density) *//there is no path if random is less than density*

{

connectivity\_matrix[i][j] = 0.0;

connectivity\_matrix[j][i] = connectivity\_matrix[i][j]; *//undirected graph*

}

**else** *//else, there is a path*

{

connectivity\_matrix[i][j] = random\_generator(min, max); *//get a value between min to max, default is 1 to 10*

connectivity\_matrix[j][i] = connectivity\_matrix[i][j];

}

}

}

}

##### **Generate random cost edges**

**float** random\_generator(**float** lower, **float** upper)

{

*//Get random Cost*

**int** range = (**int**)upper \* 10 - (**int**)lower \* 10;

**int** temp = rand() % range + (**int**)lower \* 10;

**return** (**float**)temp / 10;

}

##### **Apply Dijkstra’s Algorithm**

void Graph::dijkstra\_algorithm(int s, int t){

**if** (closed\_set.empty() == true) {

Vertex V;

V.vertex\_no = s;

V.cost\_from\_start = 0;

V.path\_from\_start.push\_back(s);

closed\_set.push\_back(V);

}

int current = s;

**if** ((false == update\_open\_set(closed\_set.back())) && (open\_set.empty() == true)) {

**return**; //stop **when** open set **is** **not** updated **and** it **is** empty

}**else**{

current = update\_closed\_set();

}

**if** (current == t){

**return**; //stop **when** destination **is** included **in** closed set

} **else**{

dijkstra\_algorithm(current, t);

}

}

##### **Find Shortest Path**

**float** Graph::get\_shortest\_path(**int** s, **int** t){

**if** ((s < 1) || (s > number\_of\_vertices)) {

cout << "No Such Start Vertex " << s <<endl;

**return** 0.0;

}

**if** ((t < 1) || (t > number\_of\_vertices)) {

cout << "No Such End Vertex " << t <<endl;

**return** 0.0;

}

**if** (s == t) {

**return** 0.0;

}

dijkstra\_algorithm(s - 1, t - 1); *//perform dijkstra's algorithm*

Vertex T = closed\_set.back(); *//the last member of closed set*

**if** (T.vertex\_no != t - 1) *//the destination is not in*

{

cout << "No Path From " << s << " To " << t << endl;

open\_set.clear();

closed\_set.clear();

**return** -1;

}

cout <<" "<<s <<"\t" <<" "<<t << "\t";

cout <<T.cost\_from\_start <<"\t\t";

list<**int**>::iterator iter;

**for** (iter = T.path\_from\_start.begin(); iter != T.path\_from\_start.end(); iter++){

**if** (\*iter != T.path\_from\_start.back()){

cout << (\*iter) + 1 << "->";

}**else**{

cout << (\*iter) + 1 <<endl;

}

}

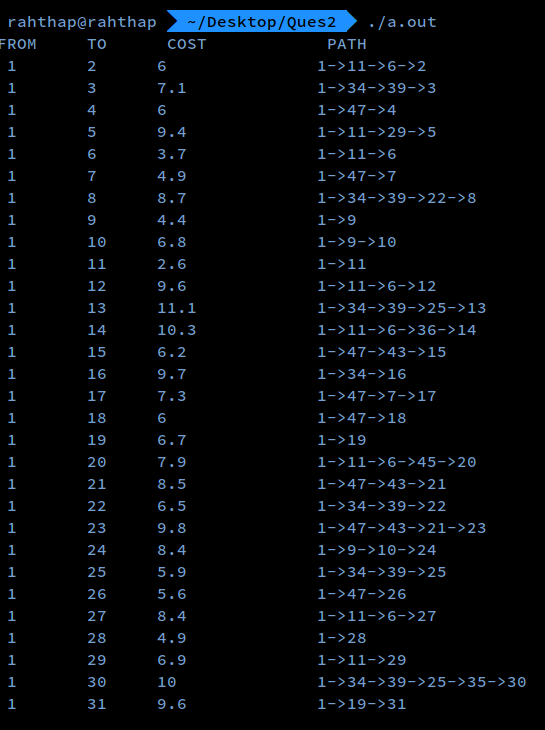
open\_set.clear();

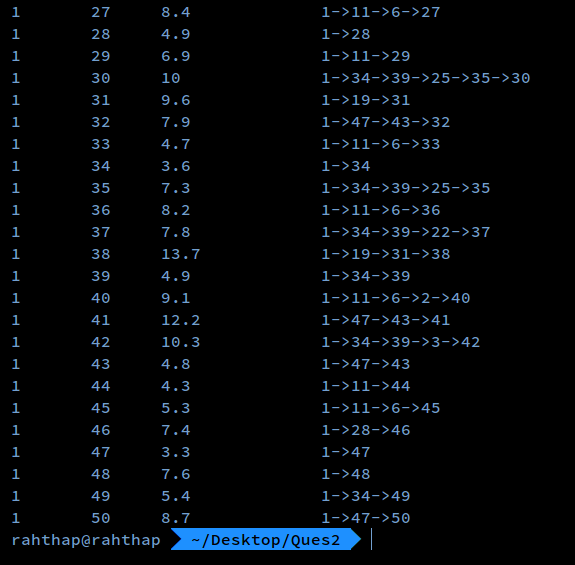
closed\_set.clear();

**return** T.cost\_from\_start;

}

**SCREENSHOTS**





#### **Ques 3**

Max Flow Problem :

// COULD NOT IMPLEMENT INTO CODE

**Approach taken :**

For a given graph (map of the town) every edge/ block connecting the other block has max capacity 1. Once the block is visited it becomes

0/1 -----> 1/1

So, the other boy cannot take this path. At the corner however they can take, this means:

A ------ 1/1 --------> B (Already traversed by boy 1)

If (Boy 2 moves from B -------> A ) //There is no arrow from B to A

Then : the flow capacity becomes 1/1 --------> 0/1 i.e. At the corner boy2 crosses boy 1

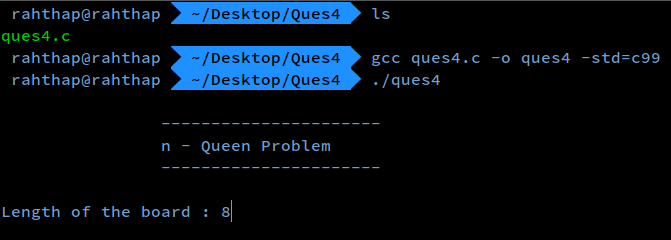
When max flow = 2 : Both the boys can go to the same school.

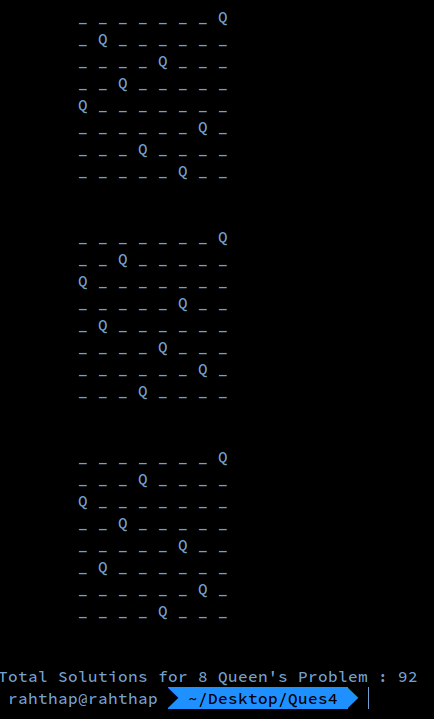
**STUCK AT?**

Not able to make 1/1 ------> 0/1 after boy 1 has already taken that block at the corner.

#### **Ques 4**

N-Queen Problem





**CODE**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <math.h>**

**int** count=0;

**int** position\_judge (**int** row,**int** column,**int** \*a){

**int** judge=1;

**int** line\_count;

**for** (line\_count=0; line\_count<row; line\_count++) {

**if** (column==a[line\_count])

judge=0;

**else** **if** (abs(column-a[line\_count])==abs(row-line\_count))

judge=0;

}

**return** judge;

}

**void** position\_print (**int** n,**int** \*a){

**int** row;

**int** column;

printf("\t");

**for** ( row=0; row<n; row++) {

**for** (column=0; column<a[row]; column++)

printf("\_ ");

printf("%c",'Q');

printf(" ");

**for** (column=a[row]+1; column<n; column++)

printf("\_ ");

printf("\n");

printf("\t");

}

printf("\n\n");

}

**void** find\_next (**int** n,**int** row,**int** \*a){

row++;

**int** column;

**for** (column=0; column<n; column++)

**if** (position\_judge(row,column,a)==1)

{

a[row]=column;

**if** (row<n-1)

find\_next(n,row,a);

**else** **if** (position\_judge(row,column,a)==1)

{

position\_print(n,a);

count ++;

}

}

}

**int** main(){

printf("\n\t\t----------------------\n");

printf("\t\tn - Queen Problem \n");

printf("\t\t----------------------\n\n");

**int** n=8,row,column;

printf("Length of the board : ");

scanf ("%d",&n);

**int** a[n];

**for** (**int** z=0; z<n; z++)

a[z]=0;

row=0;

**for** (column=0; column<n; column++) {

a[row]=column;

find\_next(n,row,a);

}

printf("Total Solutions for %d Queen's Problem : %d\n",n,count);

**return** 0;

}

#### **Ques 5**

Input Array Size : N

Number of Processors : P

Constraint : N >> P

#### **Normal Bubble Sort Algorithm**

Bubble-sort (A)

for i = 1 to N **do**

**for** j = N **to** i +1 **do**

**If** A[j] < A[j-1] **then**

**E**xchange A[j] ↔ A[j-1]

##### **Explanation :**

##### The above algorithm simply means that

##### Start from the first element of the array.

##### Compare 2 consecutive elements.

##### If the present element is greater than the element to right : SWAP them.

##### When no swapping is required : elements are sorted.

#### **Parallel Bubble Sort Algorithm**

Bubble Sort has various parallel variants such as :

1. Odd-Even Transposition
2. Cocktail sort

Bubble Sort (A)

**begin**

**for** i = 1 **to** N **do**

**begin**

**if** i **is** odd **then**

**for** j = 0 **to** n/2-1 **do**

**If** A[2i+1] > A[2i+2] **then**

Interchange A[2i+1] ↔ A[2i+2]

**else**

**if** i **is** even **then**

**for** j =1 **to** n/2 -1 **do**

**If** A[2i] > A[2i+1] **then**

Interchange A[2i] ↔ A[2i+1]

**END** **for**

**END**

##### **Explanation:**

##### The idea is processors are grouped int odd/even and even/odd pairs.

##### Odd/even Phase : The odd processes P compare and exchange their elements with the even processors P+1.

##### Even/Odd Phase : The even processes compare and exchange their elements with the odd processors P+1.

##### **Analysis of this Parallel Algorithm**

Both the phases of the algorithm requires O(N) comparisons.

Taking the worst case where all the elements are sorted **in** Descending order **and** we have to sort them **in** ascending order :

8 7 6 5 4 3 2 1

Pass 1a : 7 8 5 6 3 4 1 2

Pass 1b : 7 5 8 3 6 1 4 2

Pass 2a : 5 7 3 8 1 6 2 4

Pass 2b : 5 3 7 1 8 2 6 4

Pass 3a : 3 5 1 7 2 8 4 6

Pass 3b : 3 1 5 2 7 4 8 6

Pass 4a : 1 3 2 5 4 7 6 8

Pass 4b : 1 2 3 4 5 6 7 8 <- SORTED

This means that **is** we have 8 elements **then** we have to **do** 4 complete passes i.e. 4\*2 = 8 passes to sort the elements. So we have to **do** n passes **in** an array **in** the worst case where n **is** the number **of** elements **in** the array.

#### **How is this different from Normal Bubble Sort :**

Bubble sort **is** inherently sequential because every step **of** computation **is** dependent on the result **of** the previous step. By "offsetting" the computation slightly however, we get rid **of** **this** dependency, allowing the sorting **of** adjacent pairs to be carried out **in** parallel - That's the difference between odd-even and bubble sort.﻿