**Assignment 2**

**Aim :**

Implement Best first search (Best-Solution but not always optimal) and A\* algorithm (Always gives optimal solution)

**Theory :**

In **best first search**, we expand the node closest to the goal node. The “closeness” is estimated by a heuristic h(x) .

Heuristic: A heuristic h is defined as-

h(x) = Estimate of distance of node x from the goal node.

Lower the value of h(x), closer is the node from the goal.

Strategy: Expand the node closest to the goal state, i.e. expand the node with lower h value.

**A\* Tree Search**, or simply known as A\* Search, combines the strengths of uniform-cost search and greedy search. In this search, the heuristic is the summation of the cost in UCS, denoted by g(x), and the cost in greedy search, denoted by h(x). The summed cost is denoted by f(x).

Heuristic: The following points should be noted wrt heuristics in A\* search.

f(x) = g(x) + h(x)

Here, h(x) is called the forward cost, and is an estimate of the distance of the current node from the goal node.

And, g(x) is called the backward cost, and is the cumulative cost of a node from the root node.

A\* search is optimal only when for all nodes, the forward cost for a node h(x) underestimates the actual cost h\*(x) to reach the goal. This property of A\* heuristic is called admissibility

.

Admissibility:   0 <= h(x) <= h^\*(x)

A\* tree search works well, except that it takes time re-exploring the branches it has already explored. In other words, if the same node has expanded twice in different branches of the search tree, A\* search might explore both of those branches, thus wasting time

A\* Graph Search, or simply Graph Search, removes this limitation by adding this rule: do not expand the same node more than once.

Heuristic. Graph search is optimal only when the forward cost between two successive nodes A and B, given by h(A) - h (B) , is less than or equal to the backward cost between those two nodes g(A -> B). This property of graph search heuristic is called consistency.

Consistency:   h(A) - h(B) \leqslant g(A \to B)

**Code :**

**Best first search :**

**#include<iostream>**

**#include<vector>**

**#include<queue>**

**using namespace std;**

**typedef vector<vector<pair<int,int> > > vvpii;**

**struct node {**

**string n\_name;**

**int g\_val;**

**};**

**struct compare\_struct {**

**bool operator ()(node const& n1, node const& n2) {**

**return n1.g\_val>n2.g\_val;**

**}**

**};**

**inline int char\_to\_int(char c) { return (int)c-48; }**

**void display\_graph(vector<vector<pair<int,int> > > graph) {**

**cout<<endl;**

**for(int i = 0 ; i<graph.size(); i++) {**

**cout<<i<<" -> ";**

**for(int j = 0; j< graph[i].size(); j++) {**

**cout<<"("<<graph[i][j].first<<" ,"<<graph[i][j].second<<") ";**

**}**

**cout<<endl;**

**}**

**cout<<endl;**

**}**

**void print\_open\_closed\_list(priority\_queue<node, vector<node>, compare\_struct> open, vector<node> closed) {**

**cout<<"Open List : "<<endl;**

**for(auto temp = open; !temp.empty(); temp.pop()) cout<<"( "<<temp.top().n\_name<<", "<<temp.top().g\_val<<" )";**

**cout<<endl;**

**cout<<"Closed List : "<<endl;**

**for(auto itr : closed) cout<<"( "<<itr.n\_name<<", "<<itr.g\_val<<" )";**

**cout<<endl<<endl;**

**}**

**void bfs(vvpii graph, int h[], int start, int end) {**

**priority\_queue<node, vector<node>, compare\_struct> open;**

**vector<node> closed;**

**node s\_n = {to\_string(start), h[0]};**

**closed.push\_back(s\_n);**

**while(1) {**

**node c\_n = closed.back();**

**int c\_n\_name = char\_to\_int(c\_n.n\_name.back());**

**if(c\_n\_name == end) break;**

**for(auto itr : graph[c\_n\_name]) {**

**node nn = {c\_n.n\_name+to\_string(itr.first), h[itr.first]};**

**open.push(nn);**

**}**

**print\_open\_closed\_list(open, closed);**

**closed.push\_back(open.top());**

**open.pop();**

**}**

**print\_open\_closed\_list(open, closed);**

**}**

**int main() {**

**vvpii graph;**

**cout<<"Enter number of nodes in graph : ";**

**int n; cin>>n;**

**graph.resize(n);**

**int heuristic[n];**

**for(int i = 0; i<n; i++) {**

**cout<<"Enter heuristic value of node "<<i<<" : ";**

**cin>>heuristic[i];**

**}**

**cout<<endl;**

**for(int i =0 ; i<n; i++) {**

**cout<<"Number of childs for node "<<i<<" : ";**

**int n\_childs; cin>>n\_childs;**

**for(int j = 0; j<n\_childs; j++) {**

**cout<<"Enter node name and its distance : ";**

**int c\_n, c\_d; cin>>c\_n>>c\_d;**

**graph[i].push\_back(make\_pair(c\_n,c\_d));**

**}**

**}**

**display\_graph(graph);**

**bfs(graph, heuristic, 0, 4);**

**return 0;**

**}**

**// 5 ->number of nodes**

**// 8 6 2 1 0 ->heuristic values**

**// 2 1 1 2 4 ->no. Of adjecent nodes, (node, dist) ...**

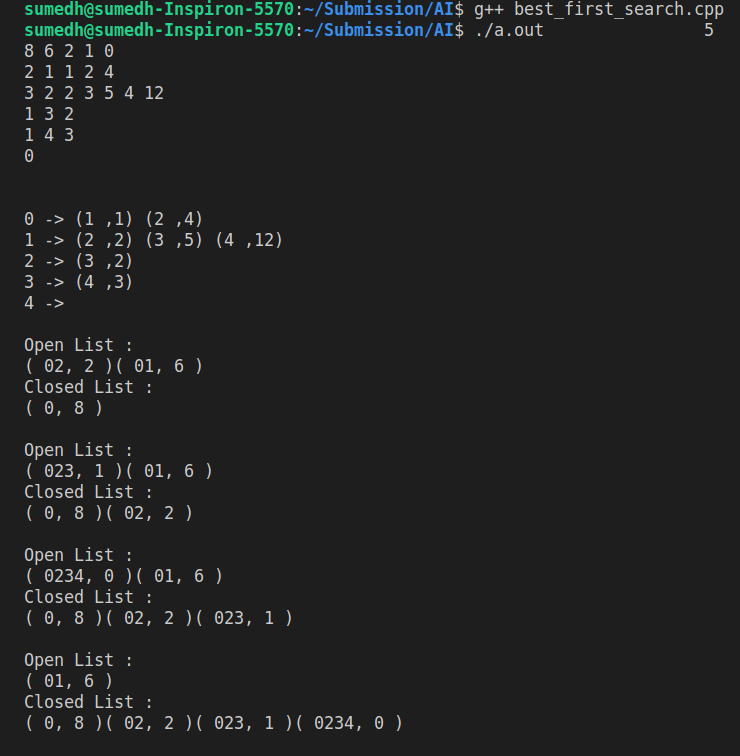
**// 3 2 2 3 5 4 12**

**// 1 3 2**

**// 1 4 3**

**// 0**

**Output :**



**A\* Algorithm :**

#include <iostream>

#include <vector>

#include <queue>

#include <algorithm>

using namespace std;

inline int char\_to\_int(char c) {return (int)c-48;}

struct node {

string n\_name;

int g\_val, f\_val;

};

inline void print\_node(node n){

cout<<"( "<<n.n\_name<<", "<<n.g\_val<<", "<<n.f\_val<<" ) "<<endl;

}

void display\_graph(vector<vector<pair<int,int> > > graph) {

cout<<endl;

for(int i = 0 ; i<graph.size(); i++) {

cout<<i<<" -> ";

for(int j = 0; j< graph[i].size(); j++) {

cout<<"("<<graph[i][j].first<<" ,"<<graph[i][j].second<<") ";

}

cout<<endl;

}

cout<<endl;

}

struct compare\_struct{

bool operator()(node const& n1, node const& n2) {

if(n1.f\_val == n2.f\_val) return n1.g\_val>n2.g\_val;

return n1.f\_val>n2.f\_val;

}

};

inline void print\_open\_closed\_list(priority\_queue<node, vector<node>, compare\_struct> open, vector<node> closed) {

cout<<endl<<"open list \t";

for(auto temp = open; !temp.empty(); temp.pop()) cout<<"( "<<temp.top().n\_name<<", "<<temp.top().g\_val<<", "<<temp.top().f\_val<<" ) ";

cout<<endl<<"closed list \t";

for(auto itr: closed) cout<<"( "<<itr.n\_name<<", "<<itr.g\_val<<", "<<itr.f\_val<<" ) ";

cout<<endl;

}

void a\_star(vector<vector<pair<int,int> > > graph, int h[], int start , int goal) {

priority\_queue<node, vector<node>, compare\_struct> open;

vector<node> closed;

node s\_n = {to\_string(start), 0, h[start]};

open.push(s\_n);

closed.push\_back(s\_n);

open.pop();

while(1) {

node c\_n = closed.back();

int c\_n\_name = char\_to\_int(c\_n.n\_name.back());

if(c\_n\_name == goal) break;

for(auto itr : graph[c\_n\_name]) {

node n\_n = { c\_n.n\_name+to\_string(itr.first), c\_n.g\_val+itr.second , c\_n.g\_val+itr.second+h[itr.first]};

open.push(n\_n);

}

print\_open\_closed\_list(open, closed);

closed.push\_back(open.top());

open.pop();

}

print\_open\_closed\_list(open, closed);

}

int main() {

int n\_nodes;

cout<<"Total number of Nodes in a graph : ";

cin>>n\_nodes;

vector<vector<pair<int,int> > > graph(n\_nodes);

int heuristic\_val[n\_nodes] = {0};

for(int i = 0 ; i < n\_nodes; i++){

cout<<"Heuristic value of node "<<i<<" : ";

cin>>heuristic\_val[i];

int n\_childs;

cout<<"Number of childs for node "<<i<<" : ";

cin>>n\_childs;

for(int j = 0; j < n\_childs; j++) {

int c\_node, e\_dist;

cin>>c\_node>>e\_dist;

graph[i].push\_back(make\_pair(c\_node, e\_dist));

}

}

display\_graph(graph);

a\_star(graph, heuristic\_val, 0, 5);

return 0;

}

// input

// 5

// 8 2 1 1 2 4

// 6 3 2 2 3 5 4 12

// 2 1 3 2

// 1 1 4 3

// 0 0

Output :

