BLG372E

Analysis of Algorithms

2013 Spring

Report of Project 1

Date of Submission : 24.03 2013

Student Name: Volkan İlbeyli

Student Number : 040100118

Instructor : Hazım Ekenel

Task 1

We are given a hospital – resident matching prolem and are expected to stable-match them according to the provided quota of the hospitals. There are two classes used in the design of the program, which are:

- Hospital
- Resident

Data Structures

Hospital

The hospital data structure is given below:

```
class Hospital {
public:
    //constructors & destructors
    Hospital();
    Hospital(const int, const int, const int);
    Hospital(const Hospital& orig);
    virtual ~Hospital();

    //getters & setters
    int getNumber() const;
    int getQuota() const;
    int** getPreferenceList();

    //member functions
    void addPreference(const int, const int);
    void addResident(Resident* resident);
    bool isAdmissable(Resident* resident);
    Resident* findWeakest();
    void removeResident(Resident* resident);
    void printList();
    bool isResidentIn(int);
private:
    int number; //hospital number
    int* preferenceList[2]; //1st row: preference order; 2nd row: admission status
    int quota;
    list<Resident*> admittedResidents;
};
```

A hospital contains a `number`(ID), a preference list which is taken inverse during the preference addition, a quota for admitted residents, and a list of residents who are admitted to that hospital. The methods are named in a self-explanatory manner. For further information, once can read the comments in the definitions of the methods. `findWeakest()` method among the member functions simply returns the admitted resident with the lowest preference order.

```
void Hospital::addPreference(const int residentNumber, const int order){
    this->preferenceList[0][residentNumber] = order;
}
```

Besides that the hospital's `addPreference()` function simply reverts the preference list while adding preference to the preference list.

Resident

The resident data structure is given below:

```
class Resident {
public:
    //constructors & destructors
    Resident();
    Resident(const int, const int);
    Resident(const Resident& orig);
    virtual ~Resident();

    //getters, setters
    int getNumber() const;
    int* getPreferenceList() const;
    bool getAdmissionStatus() const;
    int getProposalCount() const;
    int* getCurrentPreference() const;
    void setAdmissionStatus(const bool);
    Hospital* getAdmittedHospital() const;
    void setAdmittedHospital(Hospital*);

    //member functions
    void addPreference(const int, const int);
    void incrementProposalCount();
    void incrementCurrentPreference();
    bool isAdmittedTo(Hospital*);
private:
    int number;     //resident number
    int proposalCount;
    int* preferenceList;    //not inverse, unlike Hospitals pref list
    bool admissionStatus;
    int* currentPreference;
    Hospital* admittedHospital;
};
```

The resident class also have a number(ID) variable in the member field. The `int proposalCount;` keeps track of the proposal count made by the resident, which is used in the stable match algorithm to determine whether the resident is matched or not. The `int* currentPreference;` pointer keeps track of the preference and is incremented in the stable match algorithm. The `Hospital* admittedHospital;` variable keeps track of the hospital which the resident is admitted to. For further information about the functions, one can refer to the functions definitions in the .cpp files.

The Algorithm

The algorithm first initializes a stack called `stack<Resident*> freeResidentStack;` to keep track of the residents who are not admitted to a hospital, and the algorithm terminates when the stack is emptied. The stack is populated so that the resident at the top has the number 0 (initial resident). The algorithm iterates through the free residents preference list and checks if the hospitals quota is full or not. If its not full, the resident is admitted to that hospital. If the quota is full, then the algorithm checks if the resident has a higher preference order in the hospital's preference list than any resident's preference order who is previously admitted to that hospital (Note that higher order actually means a preference number with a lower value). If the resident is not admissable to that hospital, the `currentPreference` pointer is incremented and if the pointer had been pointing to the last preference in the preference list of the resident, then the resident is popped from the stack and she becomes unmatched due to quota limitation and preference order. If the resident is admissable to that hospital, than the resident in the hospital with the lowest preference order (higher preference value) is kicked out, and pushed to the `freeResidentStack`.

On the next page is the flowchart.

n: resident number, **m**: hospital number, **q**: quota.
I found using stack more convenient than a `freeResidentList` to avoid $O(m^2)$ running time (when a resident is kicked out, it should not start from the 1^{st} preference since she definitely won't be able to get in), and achieved O(m) running time for resident preference iterations, unlike the given sample running of stable matching algorithm.

Considering the whole algorithm, the running time can be calculated as:

Initialize hospitals and residents: **O(nm)**
Initialize `freeResidentStack`: **O(n)**
Algorithm:

n residents, say, have to propose to all the hospitals: **O(nm + m/q)**
Quota has effect on the running time, since it determines the comparison length when a resident is admissible to a hospital.

Below is the running times, tested for 300.000 times and the mean value is taken.

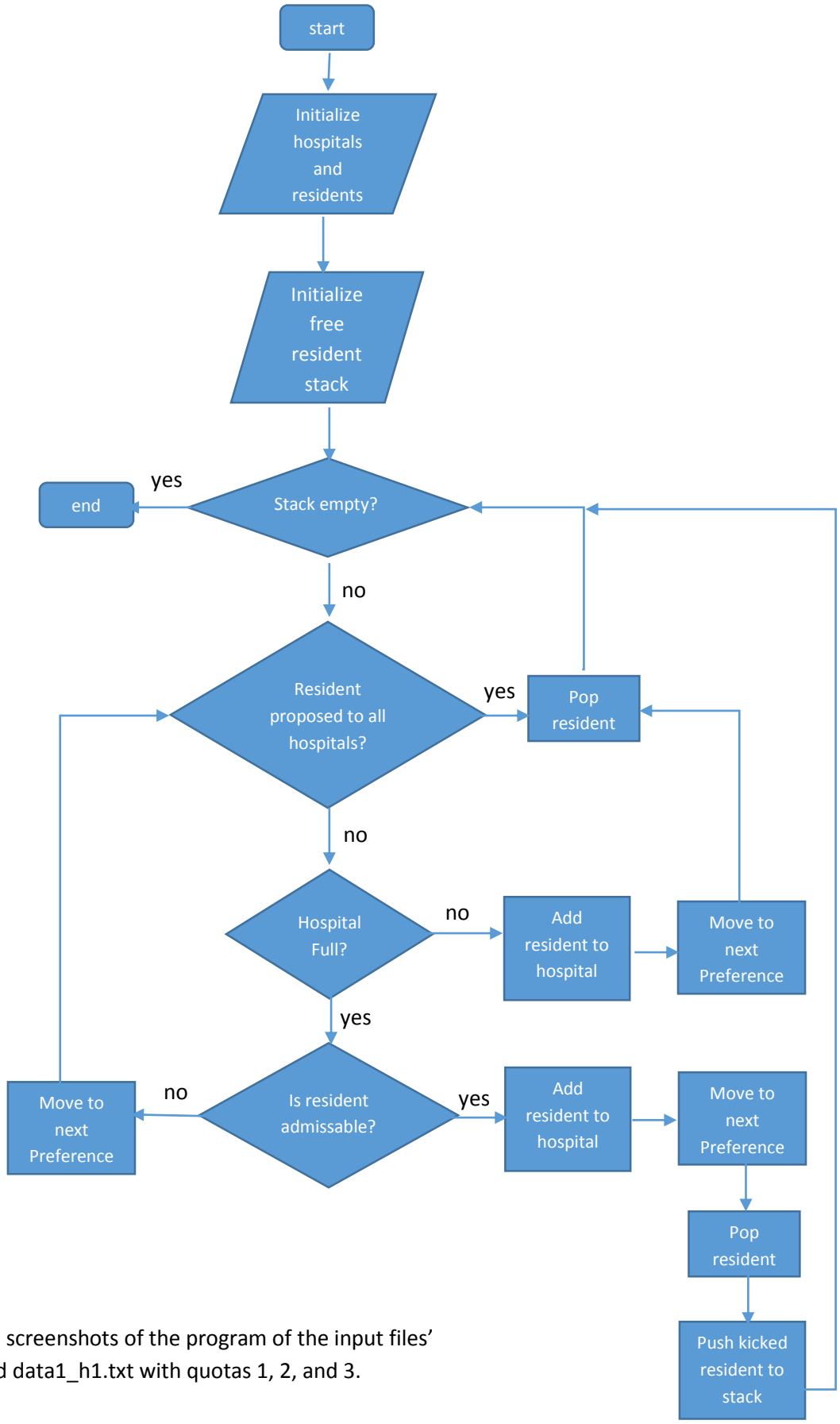
File/quota	3	5	10
data1_{h,r}1	14.2 ms	9.5 ms	11.5 ms
data1_{h,r}2	99.7 ms	99 ms	34 ms
data1_{h,r}3	23 ms	16.3 ms	15.3 ms
data1_{h,r}4	174 ms	57 ms	28 ms

Compilation

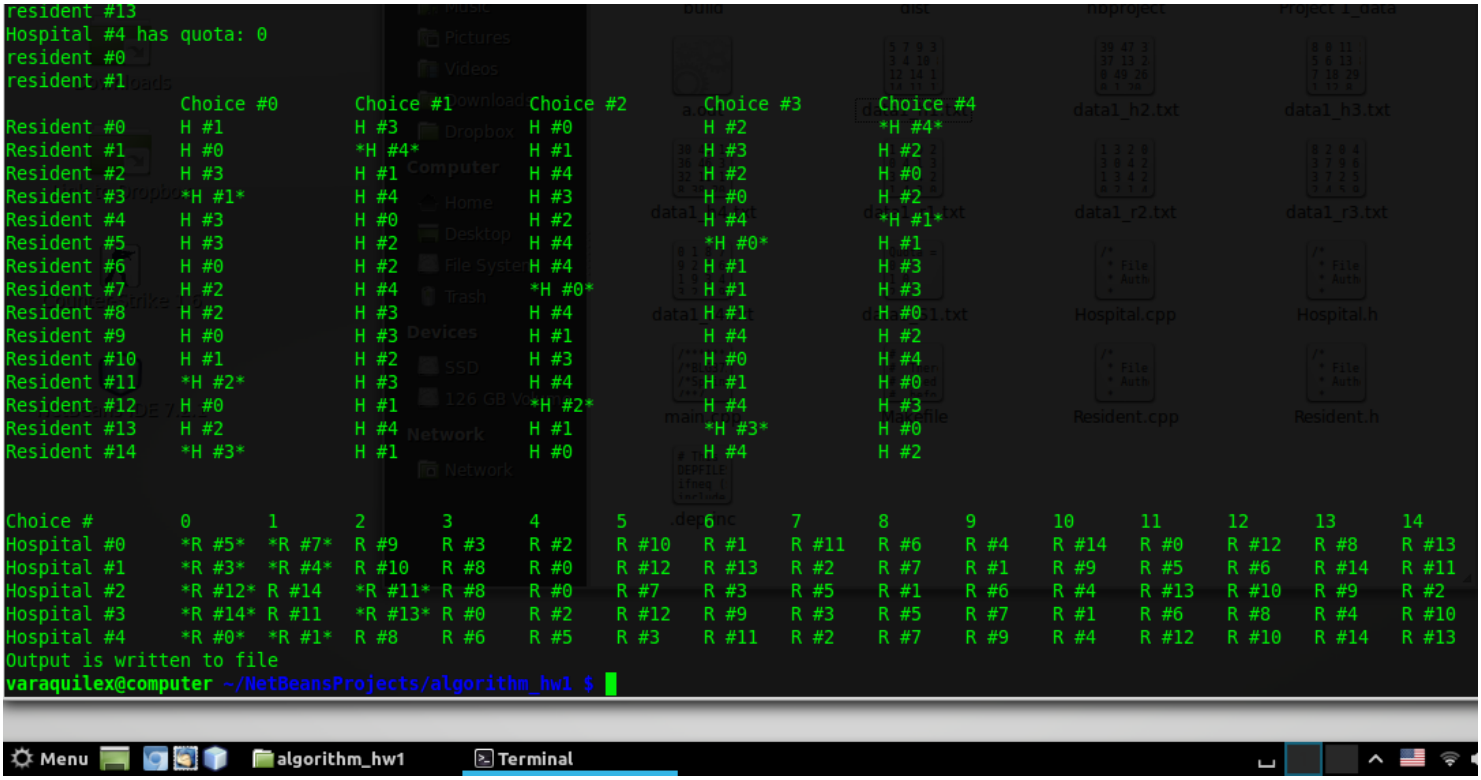
For compiling and running on Linux, type the following commands:

```
g++ main.cpp Hospital.cpp Resident.cpp –o task1
./task1 data1_h1.txt data1_r1.txt 3
```

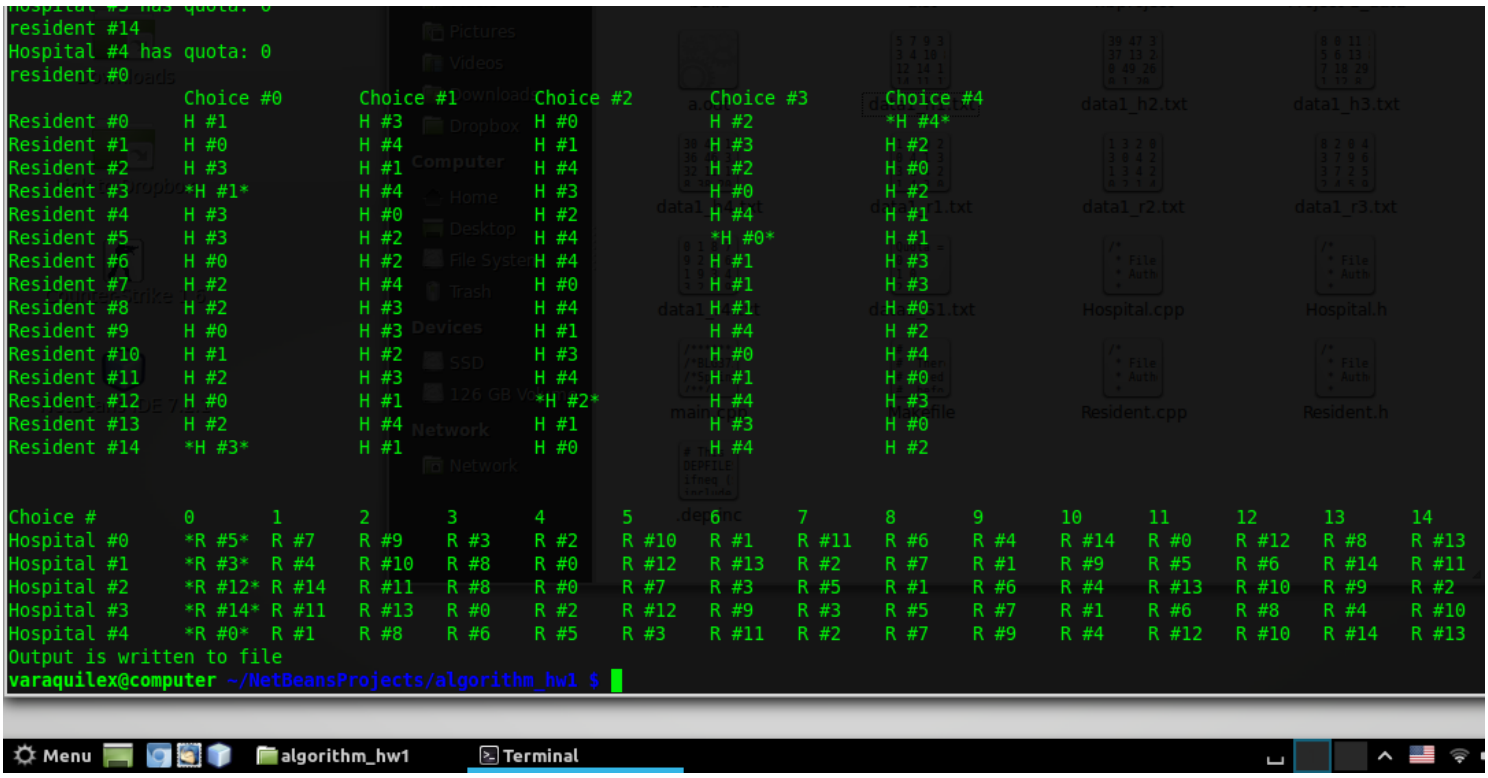
If an error occurs, you will see it on the screen and the main function will return. See source codes for error checking mechanisms.



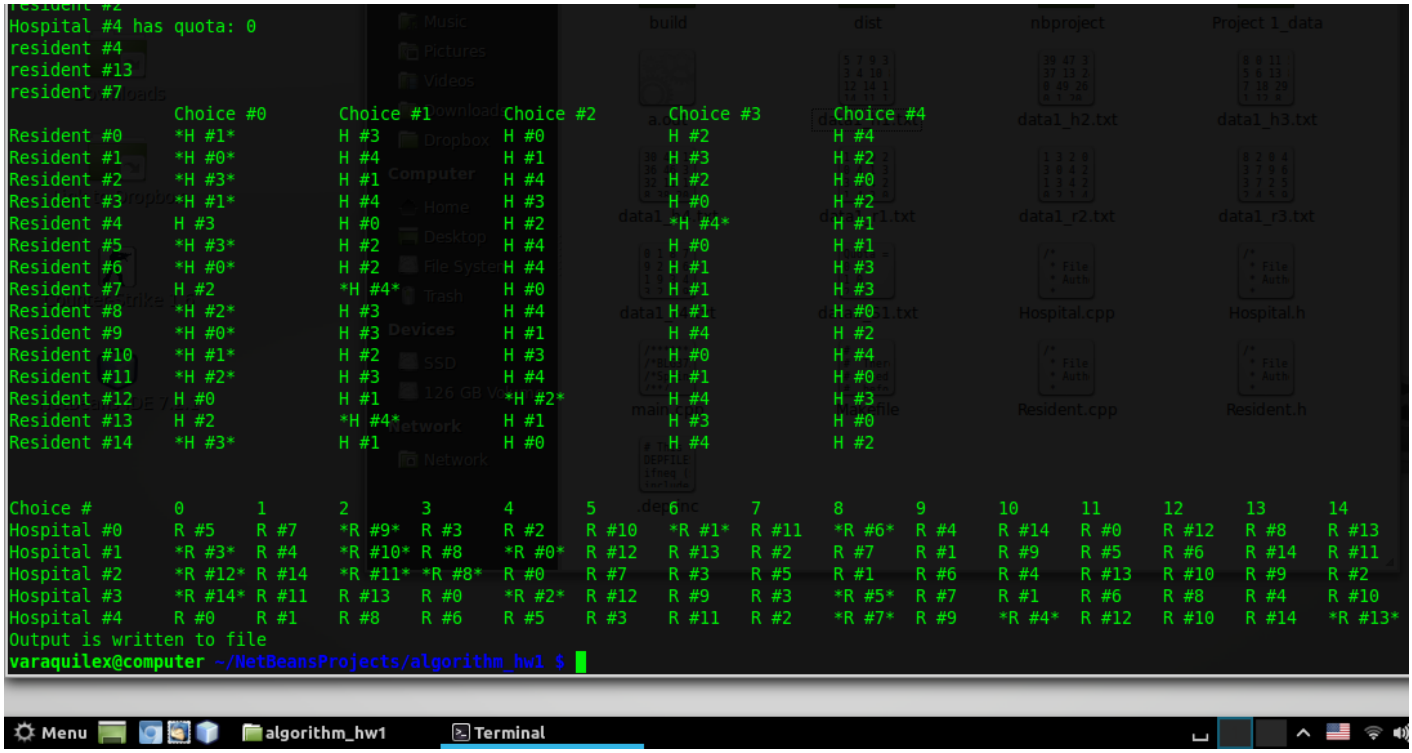
Below are some screenshots of the program of the input files' data1_r1.txt and data1_h1.txt with quotas 1, 2, and 3.



quota = 2



quota =1



quota = 3

Also note that one can easily track the algorithm by simple scrolling up through the command line.

Task 2

In task 2, we are given 5 graphs, not-quite-a-trees, and asked to print out the number of connected pairs that are in the same distance from the root node.

Two classes were used in this task for modeling the problem:

- Graph
- Node

Data Structures

Node

```
class Node {
public:
    //constructors & destructors
    Node();
    Node(const Node& orig);
    Node(const int);
    virtual ~Node();

    //getters & setters
    int getDistance() const;
    void setDistance(const int);
    int getId() const;
    list<Node*>* getAdjList();
    void setVisited(bool);
    bool getVisited() const;
    void setPrevious(Node*);
    Node* getPrevious() const;

    //member functions
    void addAdjacent(Node*);
    void printAdjList() const;
    bool isInAdjList(Node*) const;
private:
    int id;
    int distance;   //distance from the root
    list<Node*> adjacentList;
    bool visited;   //for dijkstra's algorithm
    Node* previous; //the node which has this node as its adjacent
};
```

A node have an id and distance which keeps the track of the distance between the node and the root node. `bool visited` variable is required for the dijkstra's algorithm when determining the minimum distances of nodes from the root node as well as the `Node* previous` variable. For further details, refer to the comments in the bodies of the functions.

Graph

```
class Graph {
public:
    //constructors & destructors
    Graph();
    Graph(const Graph& orig);
    virtual ~Graph();

    //getters & setters
    list<Node*> getNodeList();
    list<Node*>* getNodeListAddr();
    void setMaxDistance(const int);
    int getMaxDistance() const;

    //member functions
    void addNode(Node*, Node*);
    void printNodeList() const;
    bool isInList(Node*) const;
    void dijkstra();    //sets the distances of nodes in the graph
private:
    list<Node*> nodeList;
    int maxDistance;    // = max level
};
```

The graph class have 2 members: a node list and an integer representing the maximum distance from the root.

Since the the `void addNode(Node*, Node*);` function has a long definition in code length, It'll be described verbally, right here. The two parameters represent a node and it's adjacent. The function first checks if the first parameter is in the list of the graph by the `bool Graph::isInList(Node* n) const` function, and then again iterates through the list to search for the adjacent node. There are 4 possible situations when adding a node to a graph:

1. **The node and its adjacent are not in the graph's node list.**
   In this situation the node and it's adjacent is added to the list
2. **The node is not in the list, but it's adjacent is in the list.**
   The node is added to the list and the adjacent node is added to the `adjacentList` of the node provided as the first parameter.
3. **The node is in the list and its adjacent is not in the list**
   Adjacent node is added to the both list and the `adjacentList` of the node provided as the first parameter.
4. **The node and it's adjacent are both in the list**
   The adjacent node is added to the `adjacentList` of the node provided as the first parameter.

By this way, the nodes are connected correctly and a change made to a node affects the nodes which are not in the graph's list but are in a nodes adjacency list. The first design had flaws, such as, there were two different objects with the same id, so a change made to one of them did not affect the other node with the same id. For further details about the functions, refer to comments in the function bodies.

Algorithm

The main program first reads the input file and constructs the graphs. Here is the first function:

```
bool readFromFile(const char* fileName, Graph graph[5]){
    //reads from file and initializes the graphs by
    //setting the distances of each node from the root
    FILE* fp;

    fp = fopen(fileName, "r");
    if(fp == NULL){ //error check
        cout << "Error reading file: " << fileName << endl;
        return false;
    }

    //read content
    int id, adjId;
    for(int j=0 ; j<5 ; j++){   //graph counter
        int setLength;  // # of connections given in the file
        fscanf(fp, "%d\n", &setLength);
        for(int i=0 ; i<setLength ; i++){
            fscanf(fp, "%d %d\n", &id, &adjId);
            graph[j].addNode(new Node(id), new Node(adjId));
        }
        graph[j].dijkstra();    //sets the nodes' distances to root
    }
    fclose(fp);
    return true;
}
```

In this function, two member functions are called, `addNode()` and `dijkstra()`. `addNode()` is explained in the previous page. Dijkstra's algorithm traverses through the nodes and sets the distances of the nodes from the root. The algorithm of dijkstra's is almost the same as the original dijkstra's algorithm only with a few modifications. When a circle is encountered such as the 4th input in the file (1->2 , 2->4 , 4->3 , 3->1) the dijkstra's non-modified algorithm sets the distance of the 3rd node as 3 instead of one, so a few lines added to the dijkstra's algorithm (pseudo code: http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm) .

The terminating condition is modified as follows:

```
for(list<Node*>::iterator it = n->getAdjList()->begin() ; it != n->getAdjList()->end()
; it++){
    distance = n->getDistance()+1;
    if((*it)->getDistance()==0){    //if the adjacent node is the root
            n->setDistance(1);  //set distance to 1
    }   //if its not the root
    else if(distance < (*it)->getDistance()){   //and distance is less than its
current value
        (*it)->setDistance(distance);   //decrease key (distance)
        (*it)->setPrevious(n);  //make the adjacent node's previous node n
        if(distance > maxDistance)  //set max distance
            maxDistance = distance;
    }

}
```

The

```
if((*it)->getDistance()==0){    //if the adjacent node is the root
            n->setDistance(1);  //set distance to 1
```

block is added as if the node is adjacent of the root, so it's distance is set to 1. The Dijkstra's algorithm runs in $O(n^2)$ time. n x n: n for the not-yet-visited nodes (the node list), and n for the adjacent node list of the nodes.

After the distances are set, at the same time the levels have been determined. Since the distance and level means the same thing in this situation, I decided to stick with the distance. The problem required us to find out the number of connected pairs that are in the same distance from the root node. After the distances are set, It's time to check if the nodes with the same distance are connected to each other. This is done in the main program, in the `int checkConnection(list<Node*>);` function.

```
//count pairs
int pairCount[5] = {0};
list<Node*> nodeSet;    //nodes in a graph with same distance

for(int j=0 ; j<5 ; j++){   //do for all 5 graphs
    for(int i=1 ; i<=graph[j].getMaxDistance() ; i++){  //start from 2nd level (not
root), iterate to the max level
        //add the nodes that are in the same level to the nodeSet
        for(list<Node*>::iterator it = graph[j].getNodeListAddr()->begin() ; it !=
graph[j].getNodeListAddr()->end() ; it++){
            if((*it)->getDistance() == i)   //if in the same level
                nodeSet.push_back(*it); //add to nodeSet
        }

        if(nodeSet.size() <= 1){    //if there are less than 2 nodes in the set
            nodeSet.clear();    //clear the list
            break;          //and move to next level
        }

        //if there are more than 1 nodes in the same level
        //check if they have a connection with each other
        pairCount[j] += checkConnection(nodeSet);   //and count meanwhile
        nodeSet.clear();    //clear the set after checking the connections
    }
    cout << "*** graph " << j << ": "  << pairCount[j] << " ***" << endl;
}
```

The `maxDistance` variable comes handy in this situation: It defines the iteration count as in first checking the 2nd level, then 3rd and it goes on like this until the `maxDistance` (highest level) is reached. The function first iterates through the graph's node list and adding the nodes with the same distance into a local list, `list<Node*> nodeSet`. If there are less than or equal to 1 node in the `nodeSet`, the search iterates to the upper level. When it finds a connection between pairs, it adds to the pair count variable. Checking connection of the `nodeSet` is done with the `checkConnection()` function.

```
int checkConnection(list<Node*> nodeSet){
    //this function counts the connected pairs in the same level
    //(nodeSet is given such that the nodes it contain are in the same level)
    int count = 0;
    for(list<Node*>::iterator it = nodeSet.begin() ; it != nodeSet.end() ; it++){
        if(++it != nodeSet.end()){  //compare an element with others by starting from
the next element
            list<Node*>::iterator ite = it;     //2nd loops loop variable is set to
the next element
            it--;    //iterator should be set back to its original position
            for( ite ; ite != nodeSet.end() ; ite++){   //checking adj list of node
                if((*it)->isInAdjList(*ite))
                    count++;
            }
        }
        else
            break;
    }
    return count;
}
```

Comments explain the whole function. The function runs in O(n x (n-1)) time due to comparisons which is $O(n^2)$.

The whole program runs in $O(n^2)$ time since it's the highest order of growth the functions in the whole program have.

Compilation

For compiling and running on Linux, type the following commands:

```
g++ main.cpp Graph.cpp Node.cpp –o task2
./task2 data2.txt
```

If an error occurs, you will see it on the screen and the main function will return. See source codes for error checking mechanisms.