CS2303

Approach to HW3

September 12, 2019

Outline I

- Planning an approach to homework
- Command line input – brief review
- Open a file for read – brief review
- Adjacency matrix representation of graph
- Graphs and their Minimum Spanning Trees
- Tree search
- Adding up treasure and comparing with limit

We can plan an approach.

- This approach is not mandatory,
- you can create your own approach.
- This is merely an example.

Begin by processing command line arguments.

- The file to be read,
- the maximum number of rooms to be searched, and
- the maximum amount of treasure that can be collected
- are command line arguments.

We can interpret "command line" arguments. I

```
#include "production.h"
#include <string.h> //for strcpy

bool production(int argc, char* argv[])
{
bool answer = false;
if(argc <=1) //no interesting information
{
    puts("Didn't find any arguments.");
    fflush(stdout);
}
else //there is interesting information
{
    printf("Found %d arguments.\n", argc);
    fflush(stdout);
    long rows_L;
```

We can interpret "command line" arguments. II

```
    int rows =0;
    long cols_L;
    int cols = 0;
    long gens_L;
    int gens = 0;
    bool print = false;
    bool pause = false;
    char filename[FILENAMELENGTHALLOWANCE];
    for(int i = 1; i<argc; i++) //don't want to read argv[0
    {//argv[i] is a string
       //in this program our arguments are NR, NC, gens, //
       //because pause is optional, argc could be 6 or 7
       //because print is optional (if print is not present
char* ePtr =(char*) malloc(sizeof(char*));

switch(i)
{
```

We can interpret "command line" arguments. III

```
case 1:
    //this is NR
    rows_L = strtol(argv[i], &ePtr, 10);
    rows = (int) rows_L;
    break;
case 2:
    //this is NC
    cols_L = strtol(argv[i], &ePtr, 10);
        cols = (int) cols_L;
    break;
case 3:
    //this is gens
    gens_L = strtol(argv[i], &ePtr, 10);
    gens = (int) gens_L;
    break;
case 4:
    //this is filename
```

We can interpret "command line" arguments. IV

```
        if(strlen(argv[i]>FILENAMELENGTHALLOWANCE))
        {
            puts("Filename is too long.");
        }
        else
        {
            strcpy(filename, argv[i]);
        }
            break;
    case 5:
        //this is the optional print
            print= true;
            break;
    case 6:
        //this is the optional pause
        pause = true;
        break;
```

We can interpret “command line” arguments. V

```
default:
    puts("Unexpected argument count.");
break;
    }
  }
}
return answer;
}
```

Binary files can be read using C.

```
FILE* fp = fopen("sampleBinaryOutput.txt", "r"); //<- read
int a = -2;
double pi = 8.1;
fread(&a, sizeof(int),1, fp);
fread(&pi, sizeof(double),1, fp);
fclose(fp);
printf("Retrieved %d and %f from file.\n", a, pi);
```

Obtain data from file.

- It is mandatory to read the data from the file.
- The method reading the file "knows" the structure of the text file.
- The number of rooms/nodes is the first value, it is an integer.
- The following lines, one for each room, are of length, the number of rooms.
- In each such line, the values are 1 for a connection and 0 for no connection.
- Note that you should put the file into your directory structure, and use the name of the place where you put it, in your code.

See how the file was written. I

```
int Production::readFile()
{
    int ans = -1;
    int nrooms = -1;
    ifstream inFile("C:\\Users\\Therese\\Documents\\2019Fal
    if(!inFile)
    {
        cerr << "File could not be opened."<<endl;
        exit(1);
    }
    inFile >> nrooms;
    int* arrayP = (int*) malloc (nrooms*nrooms*sizeof(int))

    std::cout<<"read nrooms from file, obtained "<<nrooms <
```

See how the file was written. II

```
for(int roomr = 0; roomr<nrooms; roomr++)
{
    *(arrayP + (roomr * nrooms)+ roomr) = 1;//main diag

    for(int roomc = 0; roomc < roomr; roomc++)
    {
        int bit_I = -1;

        inFile>>bit_I;

        *(arrayP + (roomr * nrooms)+ roomc) = bit_I;
        *(arrayP + (roomc * nrooms)+ roomr) = bit_I;//s

    }
}
//lower left is filled with ints, also upper right
```

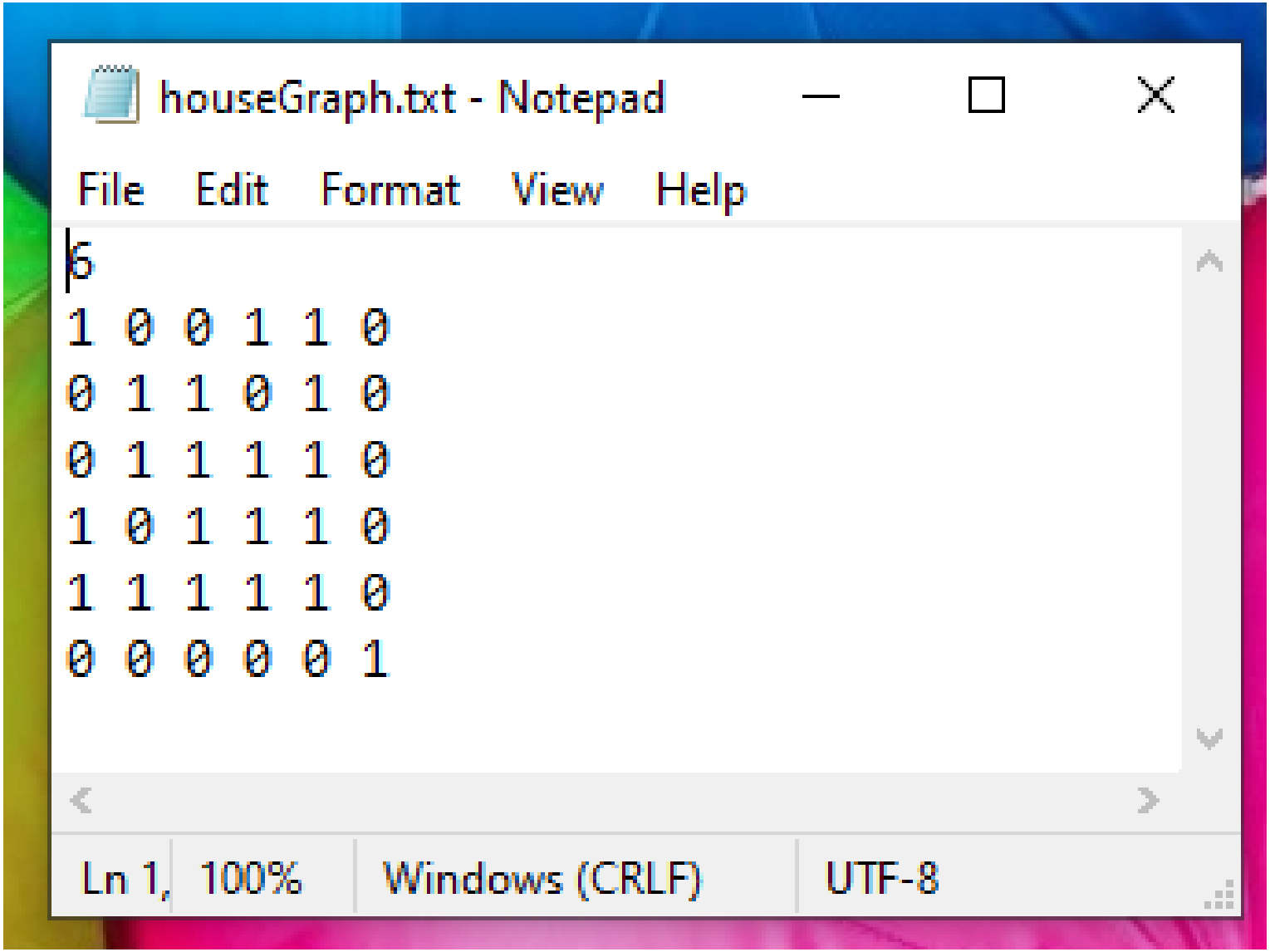
See how the file was written. III

```
//now print the matrix onto console
for(int roomr = 0; roomr<nrooms; roomr++)
{
    for(int roomc = 0; roomc<nrooms; roomc++)
    {
        std::cout << *(arrayP + (roomr * nrooms)+ roomc
        if(roomc<(nrooms-1))
        {
            std::cout <<  " " ;
        }
        else
        {
            std::cout << std::endl;
        }
    }
}
inFile.close();
```

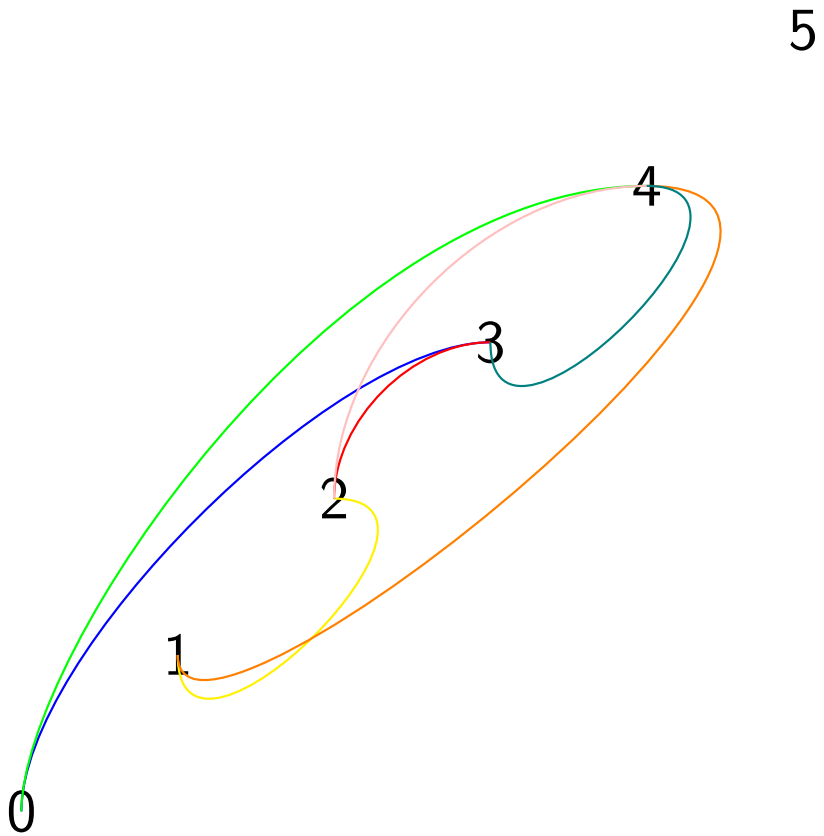
See how the file was written. IV

```
    return ans;
}
```

We can read this text file with a text editor.



We can diagram this graph.



There are functions/methods appropriate for an adjacency matrix.

- "Is node a immediately adjacent to node b?" is a reasonable question.
- "Add an edge from a to b." is a reasonable command, as is
- "Remove the edge from a to b."

We can write tests for these functions. I

```
/*
* Tests.cpp
*
*/

#include "Tests.h"
#include "Production.h"
#include "AdjMatrix.h"
#include <iostream>
using namespace std;

Tests::Tests()
{
// TODO Auto-generated constructor stub

}
```

We can write tests for these functions. II

```
Tests::~Tests() {
// TODO Auto-generated destructor stub
}

bool Tests::tests(){
bool answer = true;

bool ok1 = testReadFile();
bool ok2 = testConstructAdjacencyMatrix();
bool ok3 = testIsEdge();
bool ok4 = testAddEdge();
bool ok5 = testRemoveEdge();
answer = ok1 && ok2 && ok3 && ok4 && ok5;
return answer;
}
```

We can write tests for these functions. III

```
bool Tests::testReadFile()
{
    bool ans = false;
    cout << "Testing read file"<<endl;
    Production* prod = new Production();
    int howManyRooms = prod->readFile();
    if(howManyRooms == 6)
    {
        ans = true;
    }
    return ans;
}

bool Tests::testConstructAdjacencyMatrix()
{
    bool answer = false;
    AdjMatrix* amP = new AdjMatrix(12);
```

We can write tests for these functions. IV

```
    std::cout << "Constructed" <<endl;

    if(amP->isEdge(4,3))
    {
        cout << "Edge too soon." << endl;
    }
    amP->addEdge(3, 4);
    if(amP->isEdge(4,3))
    {
        answer = true;
    }
    return answer;
}

bool Tests::testIsEdge()
{
    bool answer = true;
```

We can write tests for these functions. V

```
    return answer;
}
bool  Tests::testAddEdge()
{
    bool answer = true;
    return answer;
}
bool  Tests::testRemoveEdge()
{
    bool answer = false;
    AdjMatrix* amP = new AdjMatrix(12);

    if(amP->isEdge(4,3))
    {
        cout << "Edge too soon." << endl;
```

We can write tests for these functions. VI

```
        }
        amP->addEdge(3, 4);
        if(amP->isEdge(4,3))
        {
            cout << "Edge added." << endl;
        }
        amP->removeEdge(4,3);
        if(amP->isEdge(4,3))
        {
            cout << "Edge not removed." << endl;
            answer = false;
        }

    return answer;
    }
```

Code for adjacency matrix. I

```
#ifndef ADJMATRIX_H_
#define ADJMATRIX_H_


#include <iostream>
using namespace std;

class AdjMatrix {
    private:
        bool** adjMatrixArray = (bool**) 0;
        int numVertices = 10;
    public:
        AdjMatrix();
        AdjMatrix(int numVertices);
        void addEdge(int i, int j);
        void removeEdge(int i, int j);
```

Code for adjacency matrix. II

```
        bool isEdge(int i, int j);
        bool formsCycle(int i, int j);
        int howManyVertices();
        virtual ~AdjMatrix();
};
#endif /* ADJMATRIX_H_ */

#include "AdjMatrix.h"

AdjMatrix::AdjMatrix() {
    AdjMatrix(10); //default size is 10
}
AdjMatrix::AdjMatrix(int nvertices) {
    this->numVertices = nvertices;
    adjMatrixArray = new bool*[numVertices];
    for (int i = 0; i < numVertices; i++) {
        adjMatrixArray[i] = new bool[numVertices];
```

Code for adjacency matrix. III

```
            for (int j = 0; j < numVertices; j++)
                adjMatrixArray[i][j] = false;
        }
    }

    void AdjMatrix::addEdge(int i, int j) {
        adjMatrixArray[i][j] = true;
        adjMatrixArray[j][i] = true;
    }

    void AdjMatrix::removeEdge(int i, int j) {
        adjMatrixArray[i][j] = false;
        adjMatrixArray[j][i] = false;
    }

    bool AdjMatrix::isEdge(int i, int j) {
        return adjMatrixArray[i][j];
```

Code for adjacency matrix. IV

```
}
int AdjMatrix::howManyVertices()
{
    return numVertices;
}
bool AdjMatrix::formsCycle(int i, int j){
    return false;
}

AdjMatrix::~AdjMatrix() {
// TODO Auto-generated destructor stub
}
```

With confidence that these functions work, we can use them in production. I

```
//use the file content to construct the AdjMatrix instance
AdjMatrix* amP = new AdjMatrix(nrooms);
for(int roomr = 0; roomr<nrooms; roomr++)
{
    for(int roomc = 0; roomc < roomr; roomc++)
    {
        if(*(arrayP + (roomr * nrooms)+ roomc))
        {
            amP->addEdge(roomr, roomc);
        }
    }
}
```

With a graph, we can determine a minimum spanning tree. I

```
MinimumSpanningTree::MinimumSpanningTree(AdjMatrix* am_P) {
    // TODO get the list of edges
    //get the number of vertices, nVs
    //start with the first in the list
    //starting a new AdjMatrix
    //for subsequent edges
    //check if adding it makes a cycle
    //if no, incorporate that edge into the new AdjMatrix
    //keep going until nVs edges are included
    treeMatrix = new AdjMatrix();

    int nVs = am_P->howManyVertices();
    cout << "Found " << nVs << " vertices." << endl;
    int vsSoFar = 0;
```

With a graph, we can determine a minimum spanning tree. II

```
bool done = false;
for(int row=0; (row<nVs) && !done; row++)
{
    for(int col = 0; (col<nVs)&&!done; col++)
    {
        if ((vsSoFar == 0) && am_P->isEdge(row, col))
        {
            treeMatrix->addEdge(row, col);
            //first found edge is added
            vsSoFar = 1;
        }
        else if( (vsSoFar < (nVs-1)) && !(treeMatrix->i
        {
            treeMatrix->addEdge(row, col);
```

With a graph, we can determine a minimum spanning tree. III

```
                    vsSoFar++;
                    cout << "added " << vsSoFar << " edges." <<
                 }
                if(vsSoFar >= (nVs-1))
                {
                    done = true;

                }
            }
        }
    }
```

We can search a tree in C++. I

- Recall we can search a tree,
- either breadth first or depth first
- by starting with a node (the root)
- and then putting each of its immediate children into a Stack or (FIFO) Queue.
- We then dequeue a node, and repeat,
- until there are no more nodes.

By turning our layout into a tree, we can search the house.

- There are algorithms for turning a graph into a tree,
- one is Kruskal's.
- We have an easy case of Kruskal's,
- because our edges are equally weighted.
- As a challenge problem, we could weight the edges by the treasure in the room of the second node of the edge.
- comments were given above for Kruskal's algorithm.