REPORT: Maximum Linear Arrangement Project

Dante Aviñó, Miguel Borge

1. Choices and difficulties & Test cases

- **Program 1**
  In this first program we had to implement three functions one to read the graph from the input; another to read a vector with the linear arrangement (corresponding to π) and finally, on that takes two parameters, i.e. the graph and that vector, producing D. In order to given a single graph as input, and the position of its vertices, write to the standard output the value of D.

```
int main(){
    // Call Read graph function
    graph g;
    g = ReadGraph();

    // Call GraphPos function
    vector<int> pi;
    pi = GraphPos(g.vertices());

    // Call Compute D function taking two a graph and vector pi as parameters
    int D;
    D = ComputeD(g, pi);

    // Print D to standard output
    cout << D << endl;

    return 0;
}
```

- **Program 2**
  In the second program we had to implement an algorithm given various graphs as input and produces would produce as output the value of D for each of them when vertex labels are interpreted as
  vertex positions.

```
int main(){
    // Read from stdin num of graphs to read.
    int AllG;
    cin >> AllG;

    for(int i = 0; i < AllG; i++){
        // Create graph object filled by ReadGraph function.
        graph g = ReadGraph();
```

```
        // Vector pi; size (vertices num); values (0 to n-1).
        vector <int> pi = Pi(g.vertices());

        // Append to AllD the max Distance of graph with positions pi.
        cout << ComputeD(g, pi) << endl;
    }

    return 0;
}
```

- **Program 3**
  In the third we had designed a program that takes various graphs as input and produces as output the value of Dmax for each of them. Calculated the results using an exhaustive enumeration of the n! linear arrangements. Use the method described next_permutation method from algorithm c++ library.

```
int main(){
    // Read from stdin num of graphs to read.
    int AllG;
    cin >> AllG;

    for(int i = 0; i < AllG; i++){
        // Create graph object filled by ReadGraph function.
        graph g = ReadGraph();
        // Vector pi; size (vertices num); values (0 to n-1).
        vector <int> pi = Pi(g.vertices());

        int Dmax = 0;
        // Permutate all possible positions
        do {
            // Call compute D taking two params (vertices) method
            int D = ComputeD(g, pi);
            // If new D is greater than current Dmax, update Dmax
            if (D > Dmax){Dmax = D;}
        } while ( next_permutation(pi.begin(), pi.end()) );

        // Print Dmax to stdout
        cout << Dmax << endl;
    }

    return 0;
}
```

- **Program 4**
  In the fourth we had to design a similar program to program3 but using a recursive algorithm for the exhaustive exploration.

```
void RecExplorerAux(const graph& g, vector<int>& pi, int& Dmax, int i){
    {
        // If index is 0
        if (i==0) {
```

```
            // Compute D of current permutation
            int D = ComputeD(g, pi);
            // If current D is grater than previous, update Dmax
            if (D > Dmax){Dmax = D;}
        } else {
            // For recursive call, iterate while j<i
            for (int j=0; j<i; j++) {
                // Swap element in pi index j <--> for element in index i-1
                swap(pi[j], pi[i-1]);
                // Call recursive function with index - 1
                RecExplorerAux(g, pi, Dmax, i-1);
                // Reverse swap
                swap(pi[j], pi[i-1]);
            }
        }
    }
}

int RecExplorer(const graph& g, vector<int>& pi){
    // Initialize Dmax to 0
    int Dmax = 0;
    // Call recursive exploration function
    RecExplorerAux(g, pi, Dmax, g.vertices());
    // Return Dmax
    return Dmax;
}

int main(){
    // Storing number of graphs we will have to read.
    int AllG;
    cin >> AllG;

    for(int i = 0; i < AllG; i++){
        // Create graph object filled by ReadGraph function.
        graph g = ReadGraph();
        // Vector pi; size (vertices num); values (0 to n-1).
        vector <int> pi = Pi(g.vertices());

        // Call recursive explorer functioons
        int Dmax = RecExplorer(g, pi);
        // Store Dmax into AllD vector
        cout << Dmax << endl;
    }
    return 0;
}
```

- **Program 5**
  In the last program we had to produce a variant of the previous program where the results are obtained applying a branch & bound technique over the implicit tree.

```
void BBExplorerAux(const graph& g, vector<int>& pi, int& Dmax, const int Dlimit,
int i){
    {
        // If index is 0
```

```
        if (i==0) {
            // Compute D of current permutation
            int D = ComputeD(g, pi);
            // If current D is grater than previous, update Dmax
            if (D > Dmax){Dmax = D;}
        } else {
            // For recursive call, iterate while j<i
            for (int j=0; j<i; j++) {
                // Swap element in pi index j <--> for element in index i-1
                swap(pi[j], pi[i-1]);
                // Call recursive function with index - 1
                BBExplorerAux(g, pi, Dmax, Dlimit, i-1);
                // Reverse swap
                swap(pi[j], pi[i-1]);
            }
        }
    }
}

int BBExplorer(const graph& g, vector<int>& pi, const int Dlimit){
    // Initialize Dmax to 0
    int Dmax = ComputeD(g, pi);
    // Call recursive exploration function
    BBExplorerAux(g, pi, Dmax, Dlimit, g.vertices());
    // Return Dmax
    return Dmax;
}

int main(){
    int AllG;
    cin >> AllG;
    for (int i = 0; i < AllG; i++){
        graph g = ReadGraph();
        vector<int> pi = Pi(g.vertices());
        int Dmax = BBExplorer(g, pi, ComputeD(g, pi));
        cout << Dmax << endl;
    }
}
```

- **Testing**:
  Test cases used in the different programs we have created different types of inputs with extreme cases such as:
  - With 0 number of Graphs, to check the response of our program; Duplicating the same edges, to s

  Two bash files have been created in order to test some of features of out programs:
  - test.sh:
    - Used to execute all the five programs given input0.txt and redirect the gotten output to file myout0.txt to the corresponding test directory of each program, then comparing this output with the expected one.
  - time.sh

- Using the time function check the execution time of all programs taking godzilla6.txt as input and redirecting the resulting value to a time.txt file in the corresponding test directory of each program.

2. Ideas or tricks you have applied to improve efficiency of the algorithms.

- Create a vector containing the computed maximum distances of each graph instead of directly printing the is more time efficient, but we should also consider that this method will use more memory therefore decreasing the memory efficiency.

program 2 (runs 188.37% times faster)

Printing directly to stdout:
real 0m51,655s
user 0m26,297s
sys 0m3,628s
Store distances in vector then printing vector to stdout (program4b):
real 0m27,422s
user 0m19,055s
sys 0m2,931s

program 3 (runs 109.86% times faster)

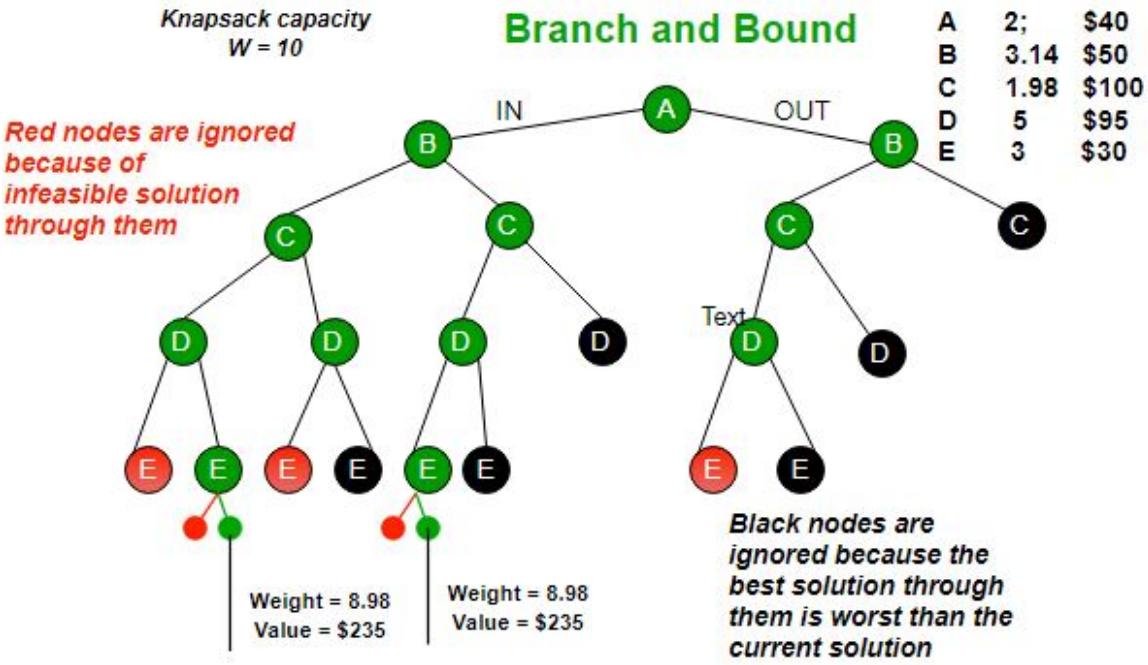
Printing directly to stdout:
real 0m35,567s
user 0m35,437s
sys 0m0,128s
Store distances in vector then printing vector to stdout (program3b):
real 0m32,375s
user 0m32,327s
sys 0m0,048s

program 4 (runs 115.27% times faster)

Printing directly to stdout:
real 0m13,989s
user 0m13,903s
sys 0m0,084s
Store distances in vector then printing vector to stdout (program4b):
real 0m12,136s
user 0m12,096s
sys 0m0,024s

3. Branch & Bound technique

- In the last program we were asked to use Branch & Bound technique, an algorithm design paradigm that is generally used to solve combinatorial optimization problems. These problems are usually exponential in terms of time complexity and may require exploration of all possible permutations in the worst case. The Branch and Bound algorithm technique solves these problems relatively quickly.

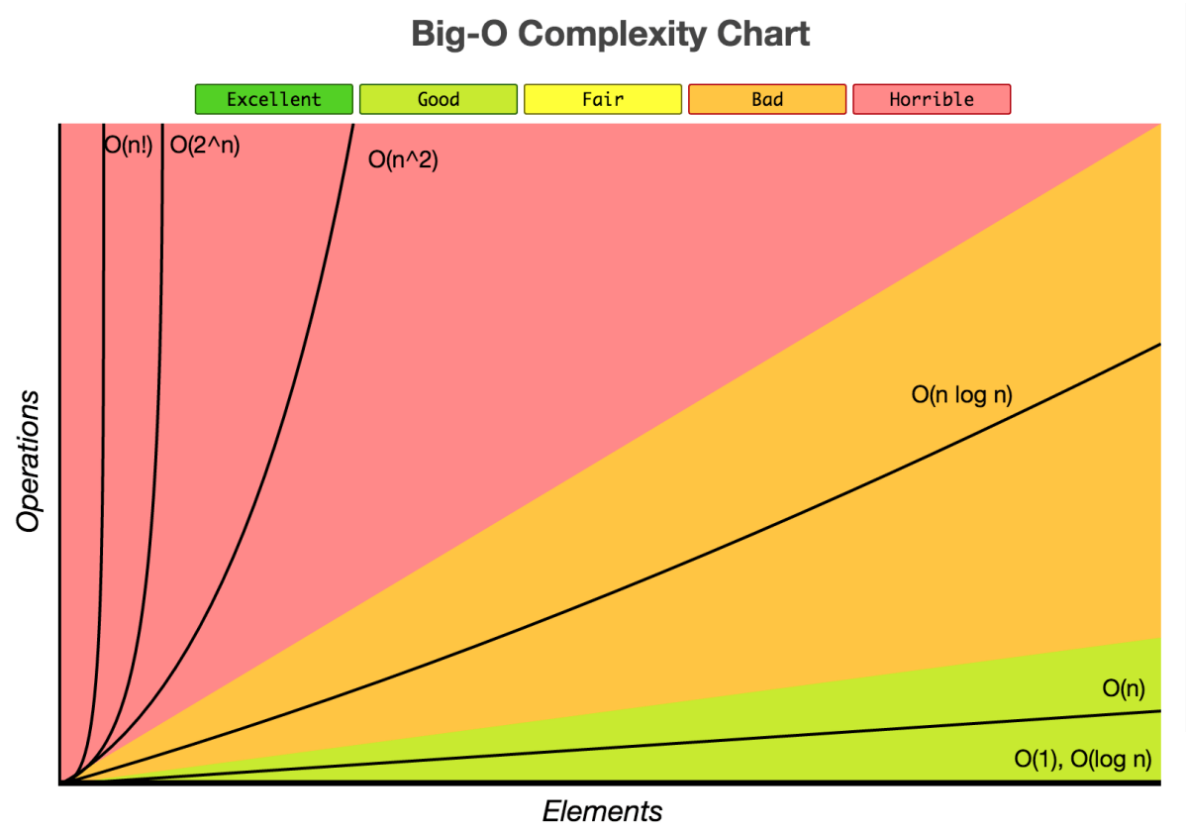


4. Running time & PC specifications

Program	Compilation	Test suit	t_i	r_i
1	YES	YES	X	X
2	YES	YES	0,244s	X
3	YES	YES	31,764s	1
4	YES	YES	11,811s	0,3718
5	YES	YES	10.340s	0.3255

- PC specifications:
  - CPU: Intel Core i7-5600U
    - 2 Cores (number of independent central processing units in a single computing component
    - With processor base frequency of 2.60 GHz
  - M.2 Solid-state drive (SSD) to store the files.

5. Discussion of the results on the execution time in Table 1



- Note that the fastest program in the table is program2 as it's not computing all permutations of a given graph, just assuming a particular configuration where positions are defined by vertex label.
- Program3 and Program4 are similar with given size in godzilla6.txt as they're exploring the same space and time complexity *$O(n! * n) = O(n!)$*, in the other hand Program5 as it's using branch and bound method and won't have to explore the whole tree of permutations will reduce this time complexity.