# Tema de Casa

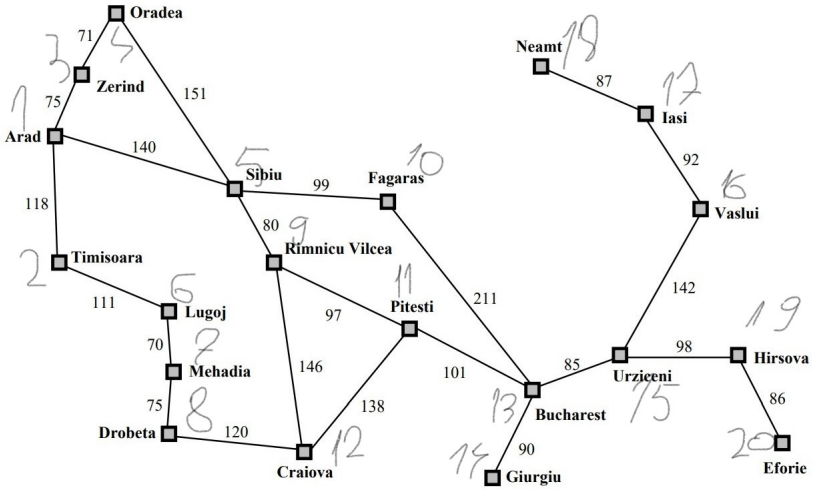
Filip Alina-Andreea
Calculatoare Romana
Anul II
Grupa 2

May 22, 2020

# 1 Problem statement

Suppose two friends live in different cities on a map, such as the Romania map shown in Figure 2. On every turn, we can simultaneously move each friend to a neighboring city on the map. The amount of time needed to move from city i to neighbor j is equal to the road distance d(i, j) between the cities, but on each turn the friend that arrives first must wait until the other one arrives (and calls the first on his/her cell phone) before the next turn can begin. We want the two friends to meet as quickly as possible. a. Write a detailed formulation for this search problem. b. Identify a search algorithm for this task and explain your choice.

a.In problema este vorba de 2 prieteni care vor sa se intalneasca cat mai repede posibil intr-un oras de pe harta Romaniei.




Orasele apar numerotate deoarece i-am conferit fiecaruia un id prin care este reprezentat in cod.

Prieteni se pot misca dintr-un oras in altul in acelasi timp, deci ruta fiecaruia ar trebui sa fie egala cu ruta celuilalt prieten.Problema apare atunci cand traseul cel mai scurt pentru ei este alcatuit dintr-un numar par de orase, adica fiecare se afla intr-un oras a, respectiv b si nu este niciun oras c intre orasele a si b. Atunci unul din ei se misca spre orasul celuilalt, permitand ca ruta fiecaruia sa fie alcatuita din n , respectiv n+1 orase. Acestea sunt structurile rutelor fiecarui prieten.

Timpul necesar pentru intalnire este egal cu distanta drumului. Deci prieteni ar trebui sa o ia pe drumul cel mai scurt dintre ei.

Daca ar fi sa alegem rute astfe incat rutele fiecarui prieten sa fie egale, ar trebui poate sa alegem rute mai lungi.

b.Eu am ales sa folosesc un Bidirectional Bfs in cautarea rutelor posibile pentru prieteni deoarece parea foarte logic ca fiecare sa il caute pe celalalt iar cand se intalnesc sa se formeze o ruta posibila pentru intalnirea lor.

## 2 Pseudocode of algorithms

Pentru realizarea acestui program am utilizat mai multi algoritmi impartiti in mai multe fisiere sursa.

```
1: function MAIN                                          ▷ In RandomNames.java
2:     cities[] ← namesofcities
3:     n ← 10
4:     for i ← 1 to n do
5:         nr1 ← Rand()
6:         nr2 ← Rand()
7:         City1 ← cities[nr1]
8:         City2 ← cities[nr2]
9:         choosenCities[] ← City1, City2
       WRITEINFILE(choosenCities, i)
10:    end for
11: end function
```

```
1: function MAIN                                          ▷ In Main.java
2:     cities[] ← cities
3:     City1 ← "X"
4:     City2 ← "X"
5:     n ← 10
6:     for i ← 1 to n do
7:         choosenCities[] ← ReadFromFile(i)
8:         City1 ← choosenCities[0]
9:         City2 ← choosenCities[1]
       ASTARSEARCH(cities[], City1, City2)
       WRITEINFILE(paths)
10:    end for
11: end function
```

```
1: function ASTARSEARCH(cities[], City1, City2)           ▷ In AStar.java
       FINDPATHBIBFS(cities[], City1, City2)
2: end function
```

```
1: function CalculateH(List Cities)
2:     h ← 0
3:     children[] ← citychildren
4:     while i< size(List) do
5:         children ← List(i).getChildren()
6:         i ← i+1
7:         for j ← 1 to size(children) do
8:             if List(i).name == children[j].name then
9:                 h ← h+children[i].value
10:            end if
11:        end for
12:    end while
13:    return h
14: end function
```

```
1: function findPathBiBFS(cities[], Source, Destination)   ▷ In BiBFS.java
2:     BFS srcData ← cities[Source]
3:     BFS destData ← cities[Destination
4:     collision ← null
5:     while !isFinished(srcData) and !isFinished(destData) do
6:         collision ← searchLevel(cities[], Source, Destination)
7:         if collision! =null then
       creatPaths(Source, Destination, collision)
8:         end if
9:         collision ← searchLevel(cities[], Destination, Source)
10:        if collision! =null then
       creatPaths(Destination, Source, collision)
11:        end if
12:    end while
13: end function
```

```
1: function searchLevel(cities[], BFS primary, BFS secundary)
2:     count ← primary.size(toVisit)
3:     for j ← 1 to count) do
4:         pathNode ← primary.toVisit.poll()
5:         cityid ← pathNode.getCity()
6:         if secondary.visited.containsKey(cityid) then
7:             return pathNode.getCity();
8:         end if
9:         city ← pathNode.getCity()
10:        children ← city.getChildren()
11:        for City i1 : children.keySet() do
12:            if !primary.visited.containsKey(i1.getID()) then
13:                child ← cities.get(i1.getID())
       primary.visited.put(i1.getID(), next)
       primary.toVisit.add(next)
14:            end if
15:        end for
16:    end for
17:    return null
18: end function
```

```
1: function getCity                                   ▷ In PathNode.java
2:     return City
3: end function
```

```
1: function collapse
2:     node ← this
3:     while node! = null do
        path.addFirst(node.city)
4:         node ← previousNode
5:     end while
6:     return path
7: end function
```

```
1: function createPaths(BFS bfsl, BFS bfs2,connection)
2:     end1 ← bfs1.connection
3:     end2 ← bfs2.connection
4:     pathOne ← endl.collapse()
5:     pathTwo ← end2.collapse()
6:     H ← CalculateH(pathOne)+CalculateH(pathTwo)
        addPaths(pathOne, pathTwo, H)
7: end function
```

```
1: function addPaths(route1,route2,h)                 ▷ In PriorityQueue.java
2:     aux[][] ← route1, route2
        aux.put(route1, route2) route.put(aux, h)
3: end function
```

```
1: function minH
2:     min ← 32000
3:     for i1: route.KeySet() do
4:         if route(i1)<min then
5:             minroute ← i1
6:             min ← route(i1)
7:         end if
8:     end for
9: end function
```

```
1: function printPaths
        minH
2:     route1, route2 ← " "
3:     for i2 : minroute.keySet()) do
4:         for i ← 1 to size(i2) do
5:             route1 ← route1+i2[i]
6:         end for
7:     end for
8:     for i3 : minroute.values()) do
9:         for j ← 1 to size(i3) do
10:            route2 ← route1+i2[j]
11:        end for
12:    end for
13: end function
```

```
1: function GETINFO                                          ▷ In City.java
2:     return info
3: end function
```

```
1: function GETCHILDREN
2:     return children
3: end function
```

```
1: function GETID
2:     return cityIDn
3: end function
```

```
1: function ADDCHILD(child, value)
       CHILDREN.ADD(child, value)
2: end function
```

```
1: function ADDCHILD(child, value)                           ▷ In BFS.java
2:     return toVisit.isEmpty()
3: end function
```

# 3 Application outline

In continuare voi vorbi despre componentele proiectului.

## 3.1 Architectural overview

In fisierul Code din arhiva se va gasi un folder numit TwoFriends care va contine 2 Packets: Application (contine clasele pentu rezolvarea problemei) si Random (contine clasa care genereaza numele random ale oraselor de pe harte).

Packet Application contine 7 clase: Main, City, BFS, BiBFS, PathNode, PriorityQueue, AStar; fiecare continand diferite functii si variabile.

In Random, se gaseste clasa RandomNames care contine o functie ce genereaza orasele initiale pentru cei doi prieteni si le introduce in 10 fisiere input (fiecare fisier contine 2 orase).

## 3.2 Input format

Fisierele input contin un vector de tip string care contine 2 elemente: orasul primului prieten , orasul celui de al 2-lea prieten.

Vectorul este impartit dupa extragerea din fisier si introdus in apelul functiei care incepe procesul de rezolvare a problemei (gasire a drumului cel mai scurt pentru fiecare prieten).

### 3.3 Output format

In fisierele output se gaseste rezolvarea pentru problema, adica cele 2 cai pentru prieteni. Aceste cai incep fiecare cu orasul de inceput al fiecarui copil si se termina cu un oras comun, adica locul lor de inatlnire. Caile ar trebui sa fie fie egale , fie cu o diferenta de cel mult un oras intre ele.

Tot in fisier este specificat inainte de drumurile fiecarui prieten si orasele introduse in input si timpul efectiv de executie al problemei in nanosecunde (pentru o precizie mai mare).

### 3.4 Modules

1. AStar: Aceasta clasa contine 2 functii, o functie care apeleaza functia BiBFS pentru identificarea coliziunilor in graph si o functie care calculeaza distanta unui rute introduse pe harta.

2. BFS: Aceasta clasa contine 2 liste ca parametru: o lista care contine nodurile ce trebuie vizitate si o lista care contine nodurile/orasele ce au fost deja vizitate de algoritmul de cautare. Ca functii se regaseste functia constructor care introduce orasul de plecare (primul oars ) in liste si o functie care verifica cand nu mai exista noduri care trebuie vizitate.

3. BiBFS: Aceasta clasa contine 2 functii: findPathBiBFS care identifica coliziunile iar daca acestea apar se va apela functia care creeaza caile pentru cei doi prieteni si searchLevel care conduce cautarea de la BFS-ul primar spre cel secundar, in functie de orasele introduse din findPath-BiBFS. Tot in find PathBiBFS este apelata functia searchLevel pentru sursa-¿destinarie si destinatie-¿sursa.

4. City: Aceasta clasa reprezinta fiecare nod (fiecare nod din graf este de tip City), ea contine o lista cu copii/vecinii nodului/orasului, un id prin care este identificat fiecare oras si un string cu informatii (aici se introduce numele orasului). Ca si functii se identifica functii simple: constructorul, getInfo, getChildren si addChildren.

5. Main: Aici are loc procesul de citire din fisier, masurare a timpuli de executie, apelare a functiei care incepe rezolvarea problemei, apoi reintroducere in fisiere de tip output. Deoarece in fisierele output se regasesc si datele din input se poate deschide doar fisierul output pentru verificarea functionatitatii programului. De asemenea aici se creeaza PriorityQueue pentru fiecare input introdus, unde se stocheaza solutiile.

6. PathNode: Este o clasa care alcatuieste legaturile dintre fiecare oras identificat ca parte din solutie. Ea contine o variabila de tip City (un nod contine un oras/ este un oras) si variabila de tip PathNode care contine nodul anterior. Ca functii avem getCity, collapse, createPaths.

7. PriorityQueue: Aici se stocheaza toate solutiile posibile pentru problema intr-un tabel route (caile posibile pt fiecare prieten si distanta totala a ambelor cai). Mai exista o variabila care contine solutia cu distanta minima de parcurs dintre cele gasite. Ca si functii se regasesc: functia constructor care creaza cele 2 liste, Functia de adaugare a unei solutii impreuna cu distanta ei addPaths, functia de identificare a solutiei optime minH, si functia de printare a solutiei optime Printpaths.

## 3.5 Functions

1. AStar
   (a) AStarSearch: Functia are rolul de a apela functia findPathBiBFS. Parametri functie sunt: o lista de orase(ArrayList¡City¿ cities), numele orasul primului prieten(String City1) , numele orasul celui de al 2-lea prieten(String City2), un PriorityQueue creat pentru input-ul introdus. Functia nu returneaza nimic catre functia apelanta(void).
   (b) CalculateH: Calculeaza distanta pentru o lista de orase. Parametri: LinkedList¡City¿ a reprezinta lista pentru care se doreste calculul distantei dintre toate orasele componente. Functia returneaza un intreg care reprezinta distanta parcursa in lista.
2. BFS
   (a) BFS(constructorul): Crearea unei cautari, introducerea orasului de plecare in lista nodurilor vizitate si crearea caii care acesta. Parametri: City root orasul de plecare al cautarii. Constructorul nu returneaza nimic.
   (b) ifFinished: Varifica daca mai sunt noduri de vizitat in lista( daca lista cu noduri de vizitat este goala). Returneaza false daca mai sunt noduri in lista si True daca lista este goala.
3. BiBFS
   (a) findPathBiBFS: Cauta coliziuni intre cele doua cautari create in functie si apeleaza functia de creare a cailor fiecarui prieten cand o coliziune apare, de asemenea identifica orasele dupa numele lor si le foloseste mai departe in calcule. Parametri: ArrayList¡City¿ cities este lista cu orasele de pe harta, String source este orasul sursa(al primului prieten), String destination este orasul destinatie (al celui de al 2-lea prieten) ,PriorityQueue p1 este clasa in care se salveaza solutiile posibile gasite creata pentru input-ul respectiv. Functia este de tip void (nu returneaza nimic)
   (b) searchLevel: Executa o cautare pe un anumit nivel (de la sursa la destinatie sau invers). Parametri introdusi sunt: ArrayList¡City¿ cities lista cu orasele de pe harta, BFS primary (bfs-ul care incepe de la sursa,respectiv destinatie in functie de cel introdus la apel), BFS

secondary(bfs-ul care incepe de la destinatie,respectiv sursa in functie de cel introdus la apel). Functia returneaza orasul de coliziune dintre cele doua cautari, iar daca nu se gaseste niciunul returneaza null.

4. City
   (a) City(constructor): Creeaza un oras cu numele instrodus in info si id-ul introdus. Parametri: String name reprezinta numele orasului, int id care reprezinta numarul prin care se indentifica orasul. Constructorul nu returneaza nimic.
   (b) getInfo: Returneaza catre functia apelanta informatiile despre oras(numele sau). Functia nu are parametri. Valoarea returnata este de tip String.
   (c) getChildren: Returneaza o lista cu copii orasului, fiecare copil are o valoare pentru orasul respectiv. Functia nu are parametri. Tipul datei returnate este un HashMap¡City, Integer¿.
   (d) addChild: Adauga un oras copil la lista cu copii a orasului. Parametri: City child reprezinta orasul ce urmeaza a fi introdus in lista, int value reprezinta valoarea pe care copilul o are pentru oras. Functia nu returneaza nimic.
5. Main
   (a) main: In aceasta functie se executa extragerea input-ului din cele 10 fisiere, crearea PriorityQueue pentru fiecare input, descrierea graph-ului, apelarea functiei care rezolva problema pt fiecare input si scrierea in fiseiere output a datelor de intrare, solutiei si timpului de executie. Functia nu are parametri si nu returneaza nici o valoare.
6. PathNode
   (a) getCity: Returneaza orasul din nod. Functia nu are parametri. Valoarea returnata este de tip City.
   (b) collapse: Creeaza o lista cu orasele prin care trece un prieten pana la orasul de intalnire, se incepe alcatuirea listei de la orasul de intalnire spre orasul de plecare. Se returneaza o lista de orase LinkedList¡City¿.
   (c) createPaths: Creeaza cele doua rute, ale primului prieten, respectiv al celui de al 2-lea, calculeaza distanta totala a cailor si apeleaza functia care le introduce in PriorityQueue. Parametri: BFS bfsl care incepe de la orasul sursa/destinatie , BFS bfs2 care incepe de la destinatie/sursa, int connection id-ul orasului unde se intalnesc cele doua cautari, PriorityQueue p1 unde se salveaza cele 2 cai impreuna cu distanta lor totala. Functia nu returneaza nimic.
7. PriorityQueue
   (a) PriorityQueue(constructor): Creeaza HashMap-ul care contine toate solutiile gasite pentru problema impreuna cu distantele aferente si

HashMap-ul care contine solutia cu distanta minima. Nu are parametri, nu returneaza nimic.

(b) addPaths: Adauga o solutie in HashMap-ul clasei impreuna cu distanta sa. Parametri: LinkedList¡City¿ route1 lista cu orasele prim care trece primul prieten, LinkedList¡City¿ route2 lista cu orasele prin care trece al 2-lea prieten, int h distanta totala a celor doua rute. Functia nu returneaza nimic.

(c) minH: Identifica solutia cu distanta minima si o salveaza in variabila minroute din clasa PriorityQueue. Functia nu are parametri si nu returneaza nimic.

(d) Printpaths: Functia alcatuieste un String care contine numele oraselor prin care trece fiecare prieten, acest string va fi introdus in fisierul output. Am dorit sa scriu un String in fisier in loc de lista deoarece voiam o reprezentare mai frumoasa a solutiei in fisier. Functia nu are parametri. Tipul variabilei returneate este String.

8. RandomNames

(a) main: Creeaza 10 fisiere input care contin nume random dintr0o lista cu numele fiecarui oras de pe harta. Se genereaza 2 numere random apoi se extrag numele oraselor de la pozitia respectiva in lista si se scriu in fisier. Functia nu are parametri si nu returneaza nimic.

# 4 Concluzii

Aceasta problema a fost destul de dificil de implementat. Cel mai dificil a fost identificarea unui algoritm de cautare corespunzator. Ideea cu bidirectional search a aparut in timp ce cautam probleme asemanatoare.

Nu am reusit sa implementez o metoda prin care fiecare prieten sa astepte pana celalalt ajunge intr-un anumit oras, mai exact nu am stiut unde sa fac asta astfel incat sa nu se dea peste cap intreg programul. Am incercat o metoda cu intreruperi dar nu am dus-o pana la capat caci se strica algoritmul.

Reprezentarea graph-ului nu este ideala si nu este aplicabila pentru introducerea unui graph de dimensiuni mai mari.

A fost dificila si reprezentarea input-ului in fisier si cea a output-ului.

Algoritmul afiseaza cateodata o cale mai lunga cand exista o cale mai scurta pentru intalnirea prietenilor in unele tipuri de input.

# 5 References

https://www.geeksforgeeks.org/bidirectional-search/

https://stackoverflow.com/questions/6558458/how-to-decide-whether-two-persons-are-connected

https://cs.stackexchange.com/questions/1113/how-to-construct-the-found-path-in-bidirectional-search

https://www.coursera.org/lecture/algorithms-on-graphs/bidirectional-search-t6k1V