

UNIVERSITETI I PRISHTINES FAKULTETI I SHKENCAVE MATEMATIKO-NATYRORE



Tema: Depth limited Search

Lënda: Inteligjencë Artificiale

Punuar nga:

Ardit Maliqi
Atlant Klaiqi
Andi Alidema

Ligjëruesit e lëndës:

Prof. Eliot Bytyqi
Asst.Besnik Duriqi

Historiku Depth Limited Search (DLS)

Depth-limited search (DLS) është një algoritëm kërkimi i përdorur në intelejencën artificiale dhe shkencën kompjuterike për të eksploruar dhe kërkuar një graf ose një pemë. Është një variant i Depth-first search (DFS), ku kërkimi është i kufizuar në një thellësi maksimale, përtej së cilës algoritmi do të ndalojë së eksploruari.

Historia e depth-limited search mund të gjurmohet në ditët e para të shkencës kompjuterike dhe inteligencës artificiale. Në vitet 1950 dhe 1960, studiuesit po zhvillonin sisteme të hershme të intelejencës artificiale dhe një nga sfidat kryesore ishte se si të kërkoni me efikasitet nëpër hapësira të mëdha kërkimi. Depth-first search ishte një nga algoritmet më të hershme dhe më të thjeshta të kërkimit të zhvilluara, dhe shpesh përdorej në këto sisteme të hershme të AI.

Megjithatë, depth-first search ka një pengesë të madhe, që është se mund të ngecë në unaza të pa fundme kur kërkoni nëpër një hapësirë kërkimi të pafund ose ciklike. Për të kapërcyer këtë kufizim, studiuesit zhvilluan depth-limited search, i cili kufizon thellësinë e kërkimit, duke siguruar që ai të mos ngecë në një unazë të pafundme.

Një nga referencat e para për depth-limited search në literaturë duket të jetë në një punim të vitit Donald Michie dhe James Scott, ku ata përshkruajnë një sistem të hershëm të AI të quajtur "Teoricen Logic" Teoricieni i Logjikës përdori kërkimin e kufizuar në thellësi për të kërkuar prova në problemet logjike.

Që atëherë, kërkimi i kufizuar në thelli është përdorur gjérësisht në AI dhe shkencën kompjuterike duke aplikuar për një sërë problemesh, duke përfshirë gjetjen e shtigjeve, luajtjen e lojës, përpunimin e gjuhës natyrore dhe më shumë. Në disa raste, depth-limited search është zgjeruar me heuristikën shtesë ose teknika krasitjeje për të përmirësuar performancën dhe efikasitetin e tij.

Depth-limited search është një strategji kërkimi e përdorur në AI dhe shkencën kompjuterike. Ai përfshin eksplorimin e një peme kërkimi deri në një thellësi të caktuar, përtej së cilës kërkimi përfundon. Disa nga përparësitë e depth-limited search:

Përparësitë e depth-limited search:

- Kursen kohë kujtesë: Depth-limited search mund të kursejë shumë kohë dhe memorie duke mos eksploruar të gjithë hapësirën e kërkimit. Kjo veçanërisht e dobishme kur hapësira e kërkimit është shumë e madhe ose e pafundme.
- Garanton plotësinë: Depth-limited search është i plotë nëse kufiri i thellësisë është më i madh ose i barabartë me thellësinë e zgjidhjes më të cekët në pemën e kërkimit. Kjo

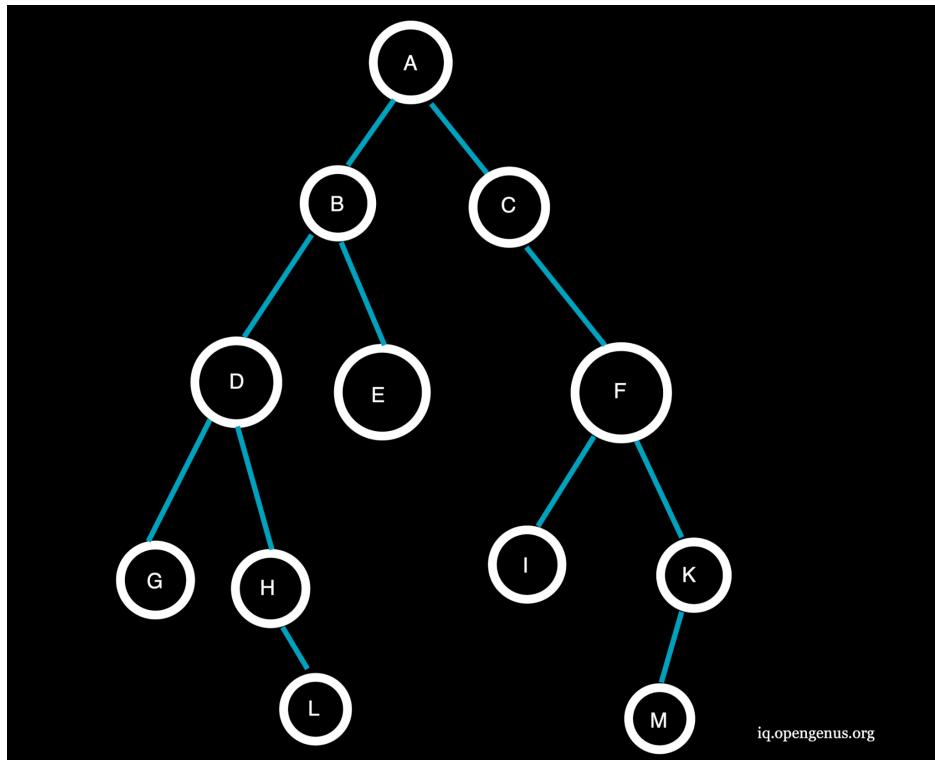
do të thotë se është e garantuar gjetja e një zgjidhjeje nëse ekziston brenda kufirit të thellësisë së specifikuar.

- Mund të trajtojë hapësira të pafundme kërkimi: Depth-limited search mund të trajtojë hapësira të pafundme kërkimi duke eksploruar pemën e kërkimit deri në një thellësi të caktuar dhe më pas duke kthyer një rezultat. Kjo e bën të dobishme për problemet ku hapësira e kërkimit është e pafundme ose shumë e madhe për t'u eksploruar plotësisht.

Pseudokodi për Depth-limited Search(DLS)

```
function DEPTH-LIMITED-SEARCH(problem,  $\ell$ ) returns a node or failure or cutoff
  frontier  $\leftarrow$  a LIFO queue (stack) with NODE(problem.INITIAL) as an element
  result  $\leftarrow$  failure
  while not IS-EMPTY(frontier) do
    node  $\leftarrow$  POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    if DEPTH(node) >  $\ell$  then
      result  $\leftarrow$  cutoff
    else if not IS-CYCLE(node) do
      for each child in EXPAND(problem, node) do
        add child to frontier
  return result
```

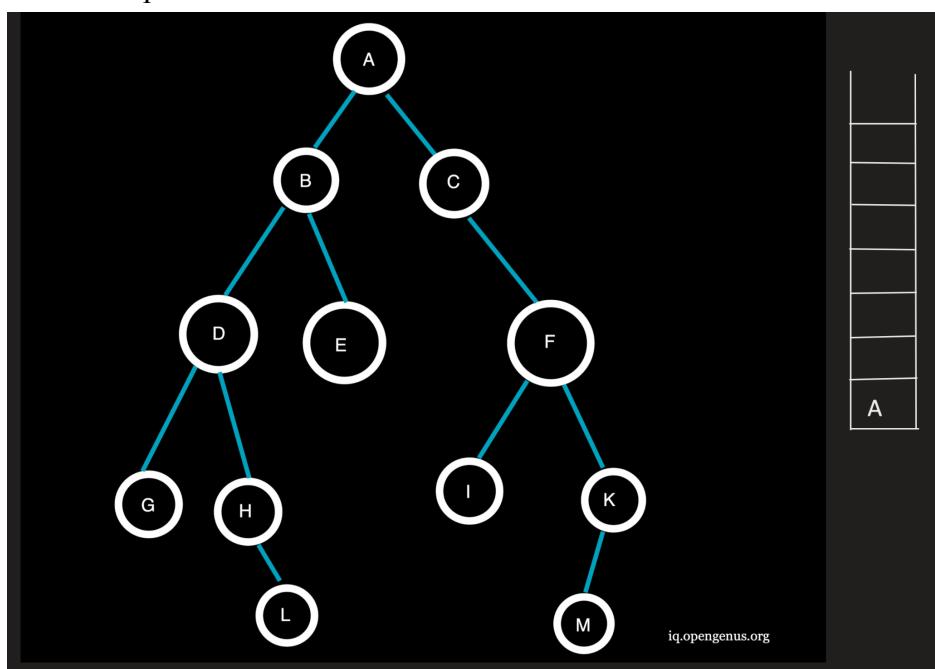
Shembull i përdorimit të Depth-limite Search(DLS)



Konsiderojmë shembullin e dhënë më lart me depth-limit = 2, goal-node = H dhe source-node = A.

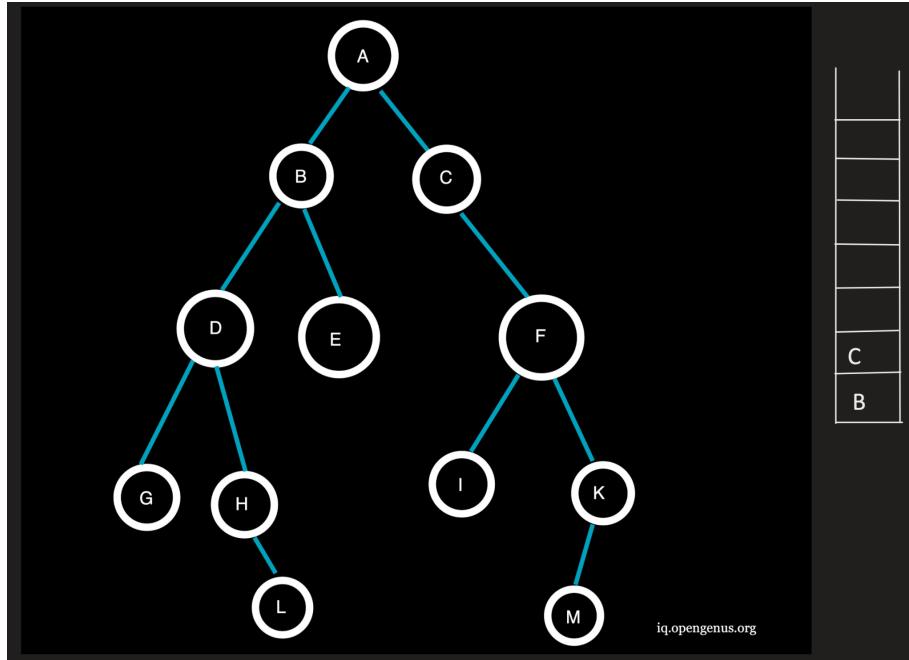
Hapi 1:

Elementi i parë i sorce-node futet në stack.



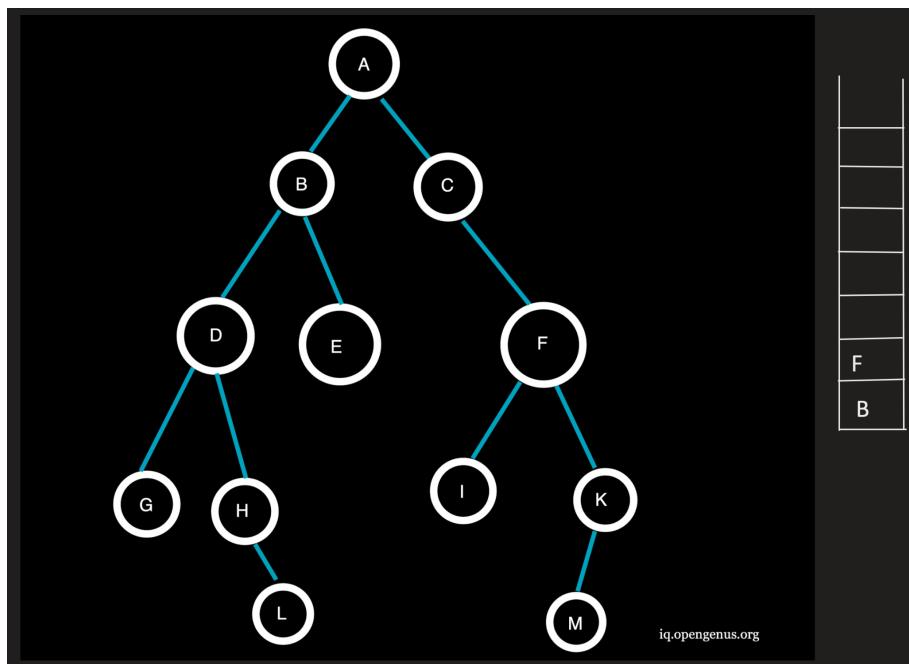
Hapi 2:

Pasi A është elementi më i lartë në stack, atëherë ai largohet nga stack dhe nodes fqinjë B dhe C futen në stack(depth është 1, pra më e vogël se limiti që i kemi caktuar 2)



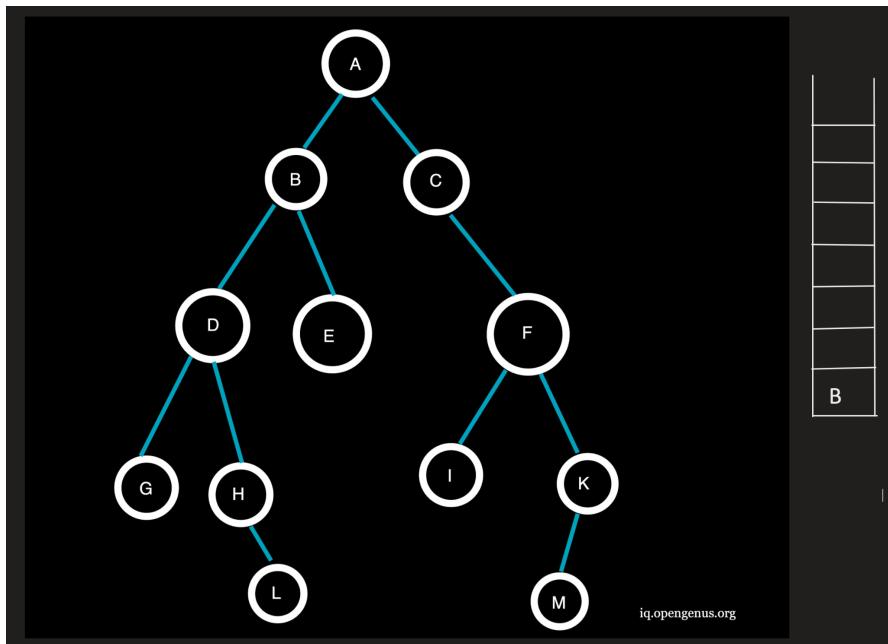
Hapi 3:

Tani pasi C është elementi më i lartë në stack, ai largohet duke u zgjeruar me node F në thellësi 2 pra sa është edhe limiti. Deri tani janë vizituar nodes:AC



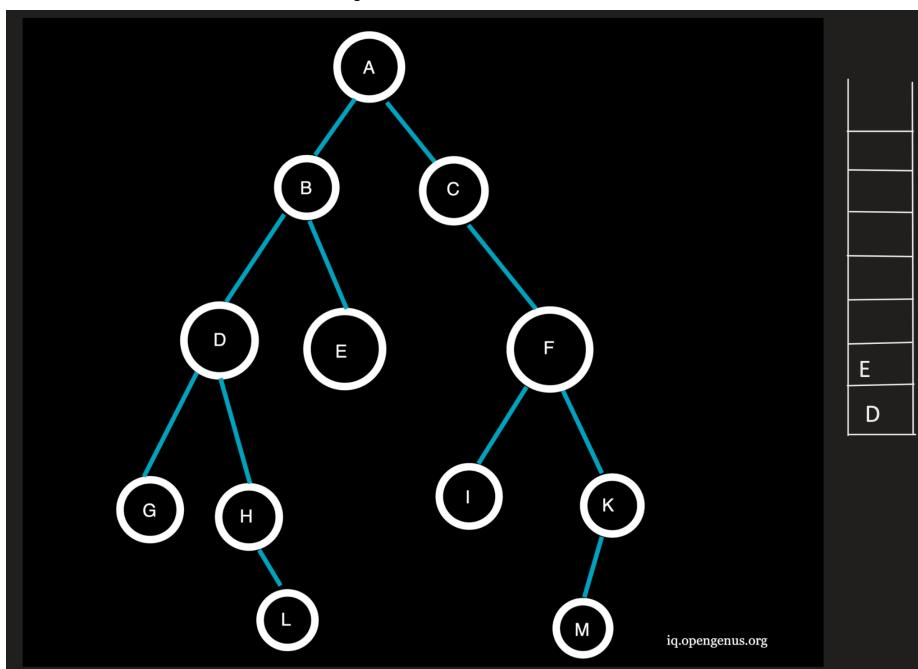
Hapi 4:

Pasi F është elementi më i lartë në stack, ai largohet nga stack dhe nodes fqinje nuk futen në stack, për arsyje se thellësia e tyre kalon limitin e thellësisë së paracaktuar nga ne. Elementet e vizituara deri tani janë: ACF



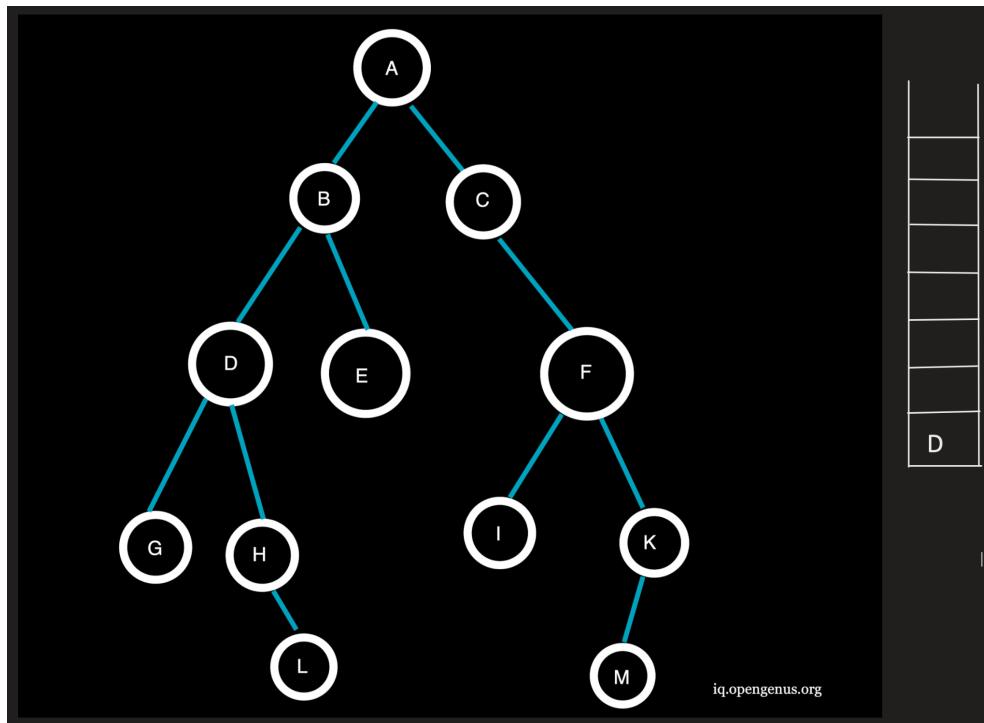
Hapi 5:

Tani B largohet nga stack dhe zgjerohet në nodes fqinje D dhe E, që kanë depth=2 <= 1. Nodes të vizituara deri tani janë ACBF



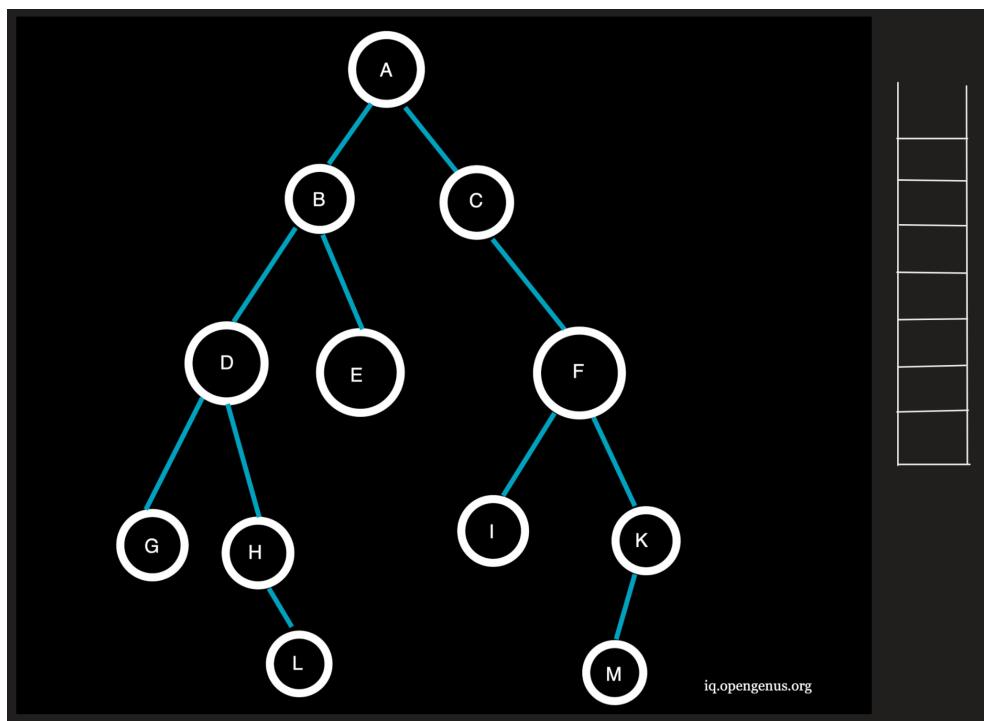
Hapi 6:

E pasi ësthë elementi më i lartë në stack largohet dhe pasi nuk ka asnjë element fqinj, asgjë nuk futet në stack. Nodes të vizituara deri tani janë ACFBE



Hapi 7:

Pasi D është elementi më i lartë në stack, ai largohet nga stack por nuk zgjerohet pasi kjo e tejkalon limitin e paracaktuar nga ne, andaj nodes G dhe H nuk futen në stack. Nodes të vizituara janë këto ACFBED.



Pasi stack tani ësthë bosh, të gjitha nodes që përfshihen brenda limitit të thellësisë janë vizituar, po target-i jonë pra node H nuk është arritur për shkak të zgjedhjes së keqe të depth-limit.

Matja e performancës së Depth-Limited Search

Matja e performancës së algoritmeve bëhet në bazë të katër kriterive:

Completeness, Time Complexity, Space Complexity dhe Cost-Optimality.

Completeness: Depth-Limitet Search është komplet nëse goal-node gjendet mbi limitin e paracaktuar.

Time Complexity: $O(b^l)$ ku b është faktori i branching (numri i fëmijëve në secilin node) ndërsa l është limiti i paracaktuar.

Space Complexity: $O(bl)$

Cost-Optimality: Edhe në rastet kur jep zgjidhje DLS nuk është optimal.

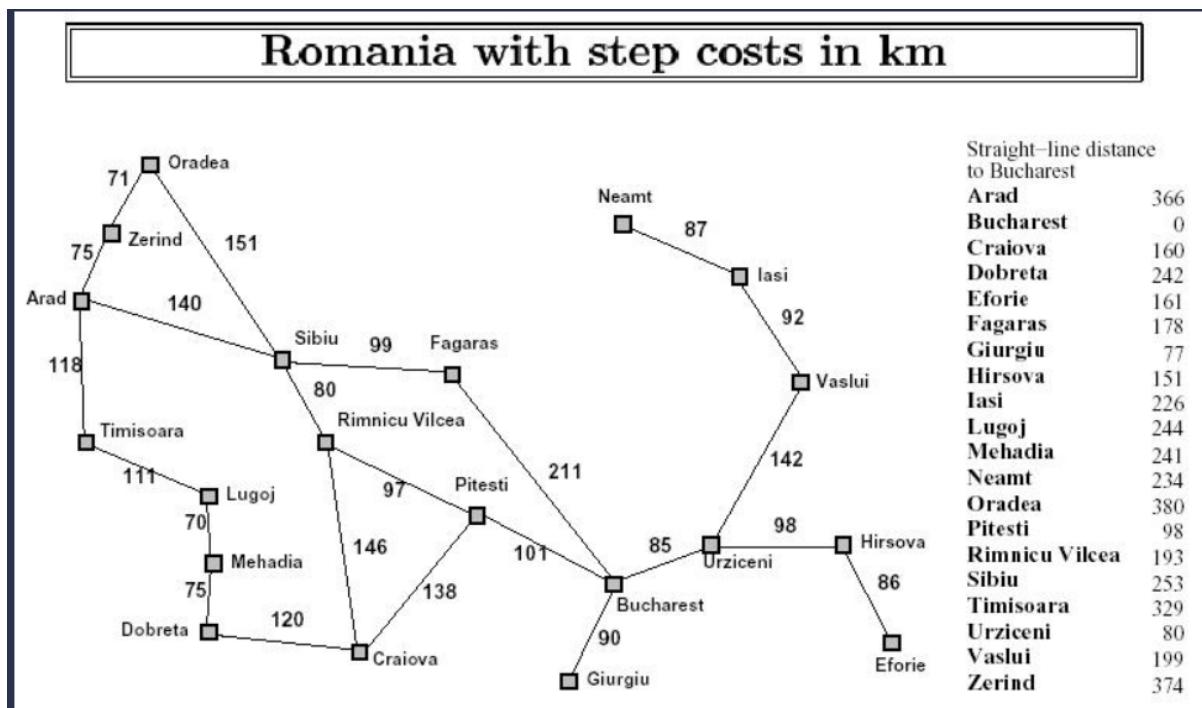
Implementing

Implementimi në Java është bërë përmes 7 klasave: NavigationProblem, NavState, Search, SearchDLS, State, Problem, Run.

Për të përdorur implementimin tonë për të zgjidhur një problem, duhet definuar problemi ne këtë mënyrë:

Use MyProblem extends Problem and MyProblemState extends State

Problemi të cilin ne e kemi zgjidhur është , problemi i navigimit ne Romani nga qyteti Arad në Bucharest. Ku harta e qyteteve është:



Gjatë shfaqjes së rezultateve kemi përdorur këto ID për një paraqitje më të lehtë:

1. Arad
 2. Zerind
 3. Oradea
 4. Sibia
 5. Timisoara
 6. Lugoj
 7. Mehadia
 8. Dobreta
 9. Craiova
 10. Rimnicu Vilcea
 11. Pitesti
 12. Fagaras
 13. Bucharest
 14. Giurgiu
 15. Urziceni

16. Hirsova
17. Eforie
18. Vaslui
19. Iasi
20. Neamt

Klasa NavigationProblem:

Paraqet problemin e navigimit nëpër Romani.

```
import java.util.ArrayList;
import java.util.List;

public class NavigationProblem extends Problem {
    private int[][] map;
    private int[] straightLineDistance;

    public NavigationProblem() {
        initializeProblem();
    }

    public boolean goalTest(State state) {
        return goalState.equals(state);
    }

    @Override
    public ArrayList<Integer> actions(State state) {
        ArrayList<Integer> actions = new ArrayList<>();
        for (int i = 0; i < map[((NavState) state).getId()].length;
i++) {
            if (map[((NavState) state).getId()][i] != 0) {
                if (state.parent != null) {
                    if (i != ((NavState) state.parent).getId())
                        actions.add(i);
                } else
                    actions.add(i);
            }
        }
        return actions;
    }

    @Override
```

```

public State nextState(State state, int action) {
    if (actions(state).contains(action)) {
        NavState nextState = new NavState(action);
        nextState.parent = state;
        nextState.act = action;
        nextState.pathCost = state.pathCost + stepCost(state,
action, nextState);
        return nextState;
    } else {
        System.out.println("wrong action");
        return null;
    }
}

@Override
public int stepCost(State firstState, int action, State
secondState) {
    return stepCost(firstState, secondState);
}

public int stepCost(State firstState, State secondState) {
    if (firstState instanceof NavState && secondState instanceof
NavState) {
        if (map[((NavState) firstState).getId()][((NavState)
secondState).getId()] != 0)
            return map[((NavState) firstState).getId()][((NavState)
secondState).getId()];
        else {
            System.out.println("invalid step cost");
            return -1;
        }
    } else {
        System.out.println("invalid step cost");
        return -1;
    }
}

public int stepCost(int firstState, int secondState) {
    return stepCost(new NavState(firstState), new
NavState(secondState));
}

@Override

```

```

public int pathCost(List<Integer> path) {
    int cost = 0;
    for (int i = 0; i < path.size() - 2; i++) {
        cost += stepCost(path.get(i), path.get(i + 1));
        if (i == path.size() - 3) {
            cost += stepCost(path.get(i + 1), 0);
        }
    }
    return cost;
}

@Override
public int h(State state) {
    return straightLineDistance[((NavState) state).getId()];
}

private void initializeProblem() {
    initialState = new NavState(0);
    goalState = new NavState(12);
    map = new int[][]{
        {0, 75, 0, 140, 118, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  

        {0, 0, 0, 0},  

        {75, 0, 71, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  

        {0, 0, 0},  

        {0, 71, 0, 151, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  

        {0, 0, 0},  

        {140, 0, 151, 0, 0, 0, 0, 0, 0, 0, 80, 0, 99, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  

        {0, 0, 0, 0},  

        {118, 0, 0, 0, 0, 111, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  

        {0, 0, 0},  

        {0, 0, 0, 0, 0, 111, 0, 70, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  

        {0, 0, 0},  

        {0, 0, 0, 0, 0, 0, 70, 0, 75, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  

        {0, 0, 0},  

        {0, 0, 0, 0, 0, 0, 0, 120, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  

        {0, 0, 0},  

        {0, 0, 0, 0, 0, 0, 0, 120, 0, 146, 138, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  

        {0, 0, 0},  

        {0, 0, 0, 0, 0, 0, 0, 146, 0, 97, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  

        {0, 0, 0},  

        {0, 0, 0, 0, 0, 0, 0, 138, 97, 0, 0, 0, 101, 0, 0, 0, 0, 0, 0, 0, 0, 0},  

        {0, 0, 0, 0}
    };
}

```

```

        {0, 0, 0, 99, 0, 0, 0, 0, 0, 0, 0, 0, 0, 211, 0, 0, 0, 0,
0, 0, 0 },
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 101, 211, 0, 90, 85, 0,
0, 0, 0 },
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 90, 0, 0, 0, 0, 0,
0, 0 },
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 85, 0, 0, 0, 98, 0,
0, 0, 0 },
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 98, 0, 86,
0, 0, 0 },
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 86, 0, 0,
0, 0 },
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 142, 0, 0,
0, 92, 0 },
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 87 },
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 87, 0 }
    };
    straightLineDistance = new int[]{366, 374, 380, 253, 329, 244,
241, 242,
160, 193, 98, 178, 0, 77, 80, 151, 161, 199, 226, 234};
}
}

```

Klasa NavState:

Kjo klasë përdoret për të modeluar gjendjen(state) të secilit qytet në Navigation Problem:

```

public class NavState extends State {
    private int id;
    public int getId() {
        return id;
    }

    public NavState(int id) {
        super();
        this.id = id;
    }
}

```

```

@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (!(obj instanceof NavState))
        return false;
    NavState other = (NavState) obj;
    return this.id == other.id;
}

@Override
public String toString() {
    return "" + id;
}
}

```

Klasa Problem:

Kjo klasë i përmbanë klasat abstrakte të zgjidhjes së një problemi si
goalTest,actions,netState,pathCost...

```

import java.util.ArrayList;
import java.util.List;

public abstract class Problem {
    protected State initialState;
    protected State goalState;

    public State getInitialState() {
        return initialState;
    }

    abstract public boolean goalTest(State state);

    abstract public ArrayList<Integer> actions(State state);

    abstract public State nextState(State state, int action);

    abstract public int stepCost(State firstState, int action, State
secondState);

    abstract public int pathCost(List<Integer> path);
}

```

```

/**
 * heuristic function, the cost to get from the node to the goal
 * you should override it based on your problem to use Astar search
 */
public int h(State state) {
    return 0;
}
}

```

Klasa Run:

Klasa Run i bashkon të gjitha klasat tjera dhe përmbanë metodën main.

```

import java.util.Scanner;

public class Run {

    public static void main(String[] args) {
        Problem problem = new NavigationProblem();
        Search search;
        Scanner scanner = new Scanner(System.in);
        System.out.print("Do you want to run in graph mode(0) or tree
mode(1): ");
        int graphChoice = scanner.nextInt();
        boolean isGraph = true;
        if (graphChoice == 1)
            isGraph = false;
        System.out.print("please enter the depth: ");
        int depth = scanner.nextInt();
        search = new SearchDLS(isGraph, depth);
        search.setProblem(problem);
        search.execute();
        showResultOfSearch(search);
    }

    public static void showResultOfSearch(Search search) {
        System.out.println("Result of the " +
search.getClass().getSimpleName());
        System.out.print("path: 0 ");
        for (int i = search.getPath().size() - 2; i >= 0; i--) {
            System.out.print(search.getPath().get(i) + " ");
        }
        System.out.println();
    }
}

```

```
        System.out.println("path cost: " + search.anser.pathCost);

        System.out.println("Depth of the result: " +
(search.getPath().size() - 1));

        System.out.println("Number of node that has been seen: " +
search.getNodeSeen());

        System.out.println("Number of node that has been expanded: " +
search.getNodeExpand());

        System.out.println("Maximum number of nodes kept in memory: " +
search.getMaxNodeKeptInMemory());
    }

}
```

Klasa Search:

```
import java.util.LinkedList;

public abstract class Search {
    protected int nodeSeen;
    protected int nodeExpand;
    protected LinkedList<Integer> path; //action sequence
    protected int pathCost;
    protected int maxNodeKeptInMemory; //maximum node kept in memory
at a time
    protected State answer;

    protected Problem problem;
    protected boolean isGraph;
    protected LinkedList<State> f; //frontier list
    protected LinkedList<State> e; //explored list

    public Search(boolean isGraph) {
        this.isGraph = isGraph;
        nodeSeen = 0;
        nodeExpand = 0;
        path = new LinkedList<>();
        pathCost = 0;
        maxNodeKeptInMemory = 0;
        f = new LinkedList<>();
        e = new LinkedList<>();
    }

    public int getNodeSeen() {
        return nodeSeen;
    }

    public void setNodeSeen(int value) {
        nodeSeen = value;
    }

    public int getNodeExpand() {
        return nodeExpand;
    }

    public void setNodeExpand(int value) {
        nodeExpand = value;
    }

    public LinkedList<Integer> getPath() {
        return path;
    }

    public void setPath(LinkedList<Integer> value) {
        path = value;
    }

    public int getPathCost() {
        return pathCost;
    }

    public void setPathCost(int value) {
        pathCost = value;
    }

    public int getMaxNodeKeptInMemory() {
        return maxNodeKeptInMemory;
    }

    public void setMaxNodeKeptInMemory(int value) {
        maxNodeKeptInMemory = value;
    }

    public Problem getProblem() {
        return problem;
    }

    public void setProblem(Problem value) {
        problem = value;
    }

    public boolean getIsGraph() {
        return isGraph;
    }

    public void setIsGraph(boolean value) {
        isGraph = value;
    }

    public void addFrontier(State state) {
        f.add(state);
    }

    public void removeFrontier() {
        f.remove();
    }

    public void addExplored(State state) {
        e.add(state);
    }

    public void removeExplored() {
        e.remove();
    }

    public void clearFrontier() {
        f.clear();
    }

    public void clearExplored() {
        e.clear();
    }

    public void printFrontier() {
        System.out.println("Frontier: " + f);
    }

    public void printExplored() {
        System.out.println("Explored: " + e);
    }
}
```

```

        return nodeSeen;
    }

    public int getNodeExpand() {
        return nodeExpand;
    }

    public int getMaxNodeKeptInMemory() {
        return maxNodeKeptInMemory;
    }

    public LinkedList<Integer> getPath() {
        return path;
    }

    public void setProblem(Problem problem) {
        this.problem = problem;
    }

    public Problem getProblem() {
        return problem;
    }

    abstract public void execute();

    public void search() {

    }

    protected void createSolutionPath(State state) {
        State temp = state;
        while (temp != null) {
            path.add(temp.act);
            temp = temp.parent;
        }
    }

    protected void showLists() {
        System.out.print("frontier list: ");
        for (State state : f) {
            System.out.print(state + ", ");
        }
        System.out.print("\texplored list: ");
        for (State state : e) {

```

```

        System.out.print(state + ", ");
    }
    System.out.println();
}
}

```

Klasa State:

```

public abstract class State {
    protected State parent;
    protected int act; //action that caused to reach this state
    protected int pathCost;

    public State() {
        parent = null;
        act = -1;
        pathCost = 0;
    }

    public int getAct() {
        return act;
    }
}

```

Klasa SearchDLS:

Klasa kryesore sa i përketë implementimit të DLS:

```

public class SearchDLS extends Search {
    private int limit;

    public SearchDLS(boolean isGraph, int limit) {
        super(isGraph);
        if (limit < 0)
            System.out.println("invalid limit");
        else
            this.limit = limit;
    }
}

```

```

@Override
public void execute() {
    search();
}

@Override
public void search() {
    search(problem.getInitialState(), limit);
}

public int search(State node, int limit) {
    if (problem.goalTest(node)) {
        anséer = node;
        createSolutionPath(node);
        return 1;
    } else if (limit == 0) {
        return 0;
    } else {
        nodeExpand++;
        boolean cutoffOccurred = false;
        for (Integer action : problem.actions(node)) {
            State child = problem.nextState(node, action);
            nodeSeen++;
            if (isGraph) {
                if (!e.contains(child)) {
                    e.add(node);
                    int result = search(child, limit - 1);
                    if (result == 0) {
                        cutoffOccurred = true;
                    } else if (result != -1) {
                        return result;
                    }
                }
            } else {
                int result = search(child, limit - 1);
                if (result == 0)
                    cutoffOccurred = true;
                else if (result != -1)
                    return result;
            }
        }
    }
}

```

```
    maxNodeKeptInMemory = Integer.max(maxNodeKeptInMemory,
e.size());
```

}

//memory optimization because it's not part of the solution

```
if (isGraph)
    e.remove(node);
```

if (cutoffOccurred)
 return 0;
else
 return -1;

}

```
}
```