Изкуствен интелект / интелигентни системи

гл. асистент д-р Венета Табакова-Комсалова
ФМИ, ПУ „П. Хилендарски“, Пловдив

Методи за търсене

Съществуват различни възможности за реализиране на методите за търсене в Пролог. Съществено е избраната структура на възлите. Желателно е реализацията на методите да бъде по възможност универсална, т.е. да може да се прилага за различни приложения. Това не е напълно възможно, поради избраната структура на възлите – методите трябва да отчитат избраната форма на възлите.

Практическа задача: Мисионери и канибали

„Мисионери и канибали“ е добър пример на задача за търсене. Проблемът се отнася за трима мисионери и трима канибали, всичките шест първоначално са от едната страна на реката. Има една лодка, която ще се използва, за да пренесе мисионерите и канибалите от другата страна на реката. Лодката е за двама пътници като няма начин за изпращане на лодката през реката, без най-малко един пътник да е в лодката. Заплахата е, че, ако канибалите са повече от мисионерите при всякакви обстоятелства канибалите ще сготвят и изядат мисионерите. Използвайте метод за търсене, за да се напише програма на Пролог, която търси начини да се превозват всички шест лица от другата страна на реката. Предложение: да се използва представяне на състоянията [M,C,B], където M е броят на мисионери и C е броят на канибали на брега B. Началното състояние е [3, 3, ляв], а целта е [3, 3, десен]. Да се напишат предикатите за "start", "goal", "next_state" и "safe_state", да се прибавят към предиката за търсене, за да се получи пълна програма за решаване на тази задача.

```
   A < 2, C is A + 2.
move([A,B,w],[C,D,e],'Edin misioner i edin kanib
   A < 3, B < 3, C is A + 1, D is B + 1.
move([A,B,w],[A,D,e],'Edin kanibal otplava obrat
   B < 3, D is B + 1.
move([A,B,w],[A,D,e],'Dvama kanibali otplavat ob
   B < 2, D is B + 2.

safe([A,B,_]) :-
   (A =< B ; B = 0),
   C is 3-A, D is 3-B,
   (C =< D; D = 0).
myProgEntryPoint :- solve.
```

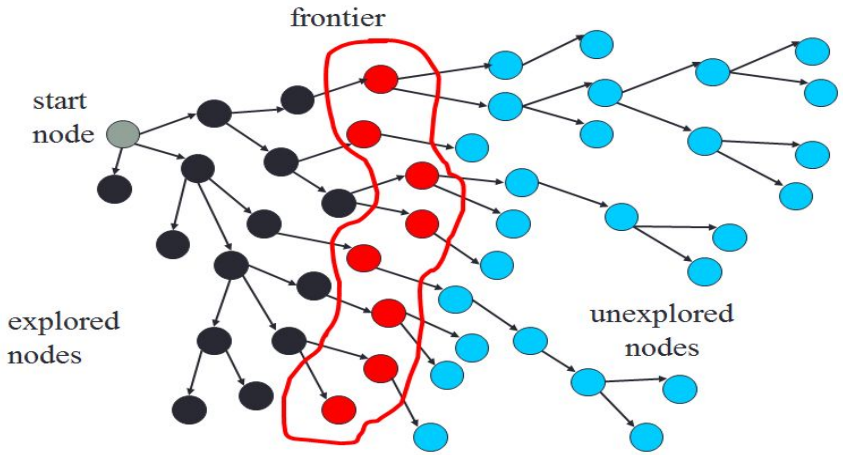
```
Opitvame slednoto:[0, 0, w] Dvama misioneri otplavat natatuk.


[3, 3, e] --> [1, 3, w]Dvijenie:Dvama misioneri otplavat natatuk.
[1, 3, w] --> [2, 3, e]Dvijenie:Edin misioner otplava obratno.
[2, 3, e] --> [0, 3, w]Dvijenie:Dvama misioneri otplavat natatuk.
[0, 3, w] --> [1, 3, e]Dvijenie:Edin misioner otplava obratno.
[1, 3, e] --> [1, 1, w]Dvijenie:Dvama kanibali otplavat natatuk.
[1, 1, w] --> [2, 2, e]Dvijenie:Edin misioner i edin kanibal otplavat
obratno.
```

```
?- solve.
```

Examples▲ History▲ Solutions▲ table results Run!

Общ алгоритъм за търсене:

- даден е граф
- начални състояния
- целеви състояния
- последователно да се изследват пътищата от началните състояния
- Поддържа се фронт/граница (frontier) от пътищата, които са били изследвани
- По време на процеса на търсене фронта се разширява в посока към неизследваните възли, докато се достигне до целеви възел



Предполагаме, че след като алгоритъма за търсене намери един път, може да му бъде зададено да търси още решения и тогава процесът трябва да продължи.
Начинът, по който фронта се разширява и това точно коя стойност от фронта се избира дефинира стратегията на търсене (search strategy).

Основни алгоритми

- Неинформирано (сляпо) търсене
- Информирано (евристично) търсене

Основни алгоритми

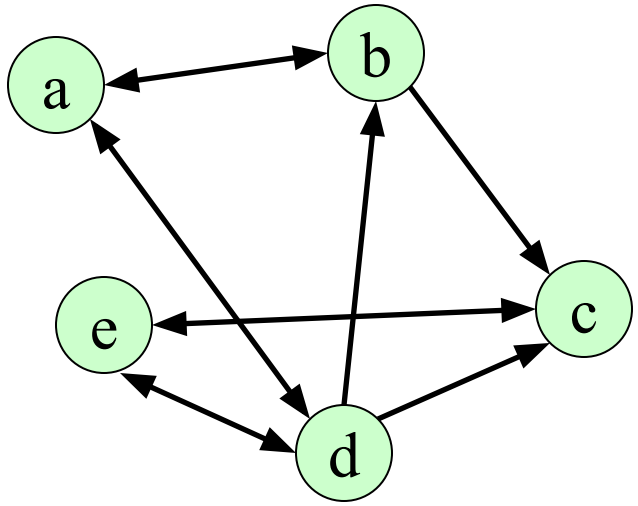
- Неинформирано (сляпо) търсене
  - търсене първо в дълбочина **(Depth-first search)**
  - търсене първо в ширина **(Breadth-first search)**
  - търсене в ограничена дълбочина **(Depth-bound search)**
  - итеративно търсене по нива **(Iterative-deepening search)**

Търсене в дълбочина

- При търсенето в дълбочина фронтът се обработва като стек
- Ако фронта е **[p1,p2, . . .]**
  - избира се **p1**
  - пътищата p_1', p_1'',…, $p_1^{(k)}$, които разширяват **p1** се добавят в началото на стека (преди **p2**), т.е. **[**p_1', p_1'',…, $p_1^{(k)}$**, p2, . . .]**
  - **p2** се обработва едва след като всички пътища, които са продължение на **p1** са били изследвани

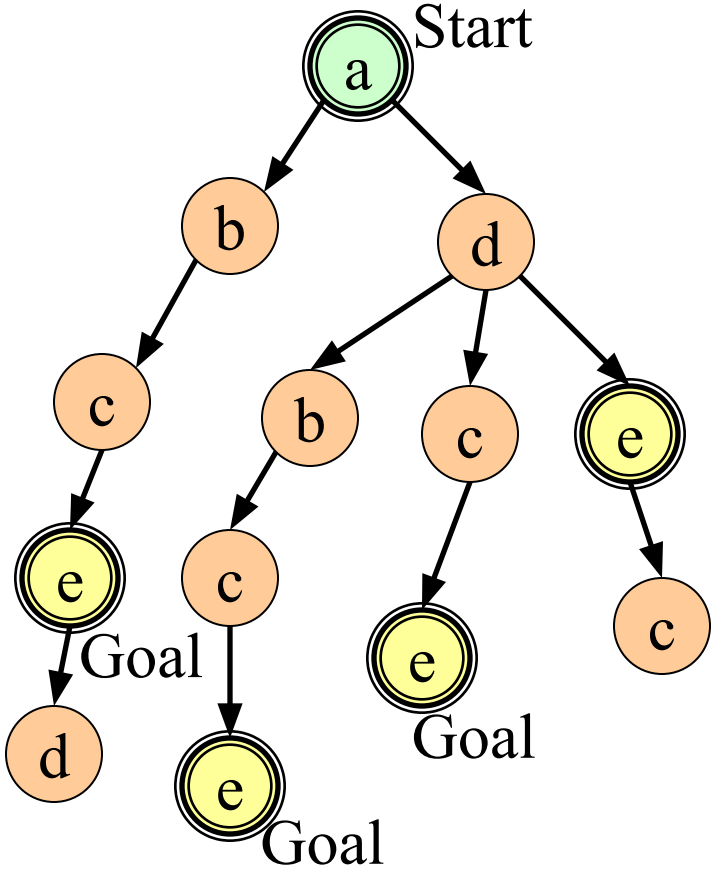
Представяне на графа

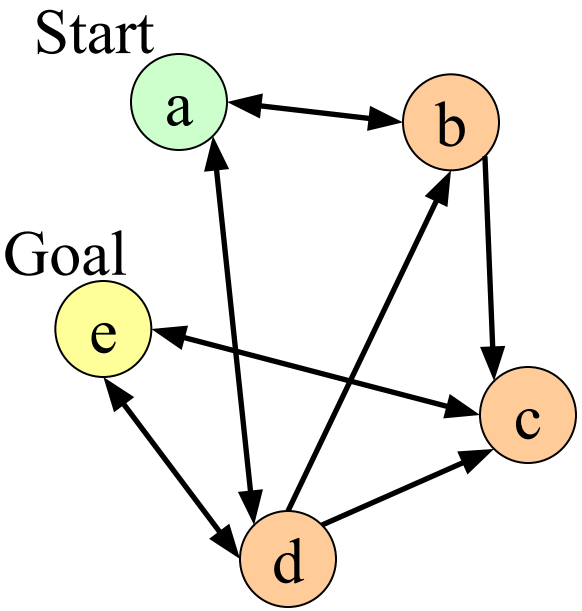
```
arc(Node1,Node2).
arc(Node1,Node2,Cost).

arc(a,b).       arc(d,e).
arc(a,d).       arc(e,c).
arc(b,a).       arc(e,d).
arc(b,c).
arc(c,e).
arc(d,a).
arc(d,b).
arc(d,c).
```



Построяване на дървото

```
arc(a,b).
arc(a,d).
arc(b,a).
arc(b,c).
arc(c,e).
arc(d,a).
arc(d,b).
arc(d,c).
arc(d,e).
arc(e,c).
arc(e,d).
```





Дърво на търсене

След като е формулиран един проблем, той трябва да бъде решен. Понеже решенията са последователност от действия, алгоритмите за търсене изследват последователности от възможни оператори (ВО). Обикновено търсенето се извършва итеративно, на отделни стъпки. Процесът на търсене може да се представи като дърво, което наричаме „дърво на търсене“.

В процеса на търсене „дървото на търсене“ се изгражда последователно на следните стъпки:

Стъпка 1: Начално състояние - търсенето започва от началното състояние, като първо се тества дали това не е целевото състояние. Началното състояние става корен на „дървото на търсене“ и актуално състояние.

Стъпка 2: Разширяване на актуалното състояние – при разширяване на актуално достигнатото състояние се:
Прилага всеки възможен оператор (ВО) върху актуалното състояние.
Генерира се ново множество от състояния-наследници.
Въвеждат се нови ребра в дървото на търсене.

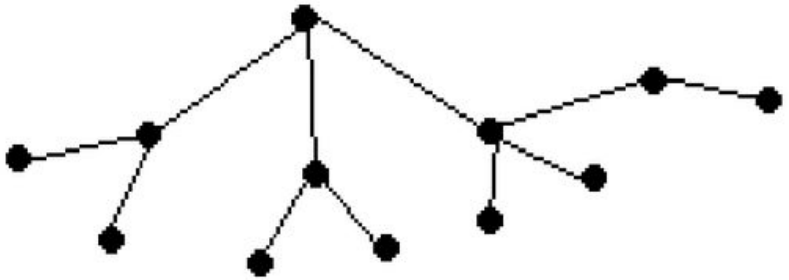
Стъпка 3: Избор на актуално състояние – от състоянията-наследници се избира ново актуално състояние.

Стъпка 4: Проверка за целево състояние. Ако актуално достигнатото състояние е целево състояние, тогава търсенето завършва. В противен случай се връщаме в Стъпка 2.

При тази схема в рамките на един цикъл определено (актуалното) състояние се проследява по-нататък, докато другите остават за следващо проследяване, ако с предишните не сме намерили решение.

Дървото е свързан, ориентиран граф, който не съдържа затворени вериги. Обикновено се бележи с Т.

- Дървото има един специален начален възел, който наричаме **корен (root).**
- Дърво с корен бележим - (T,r).
- Възли, от които не излиза ребро се наричат **листа**.
- Останалите възли се наричат **вътрешни възли**.
- Дължината на пътя от корена до най-далечното листо се нарича **височина на дървото.**



Това дърво има:

- Един корен
- Четири вътрешни възела
- Седем листа

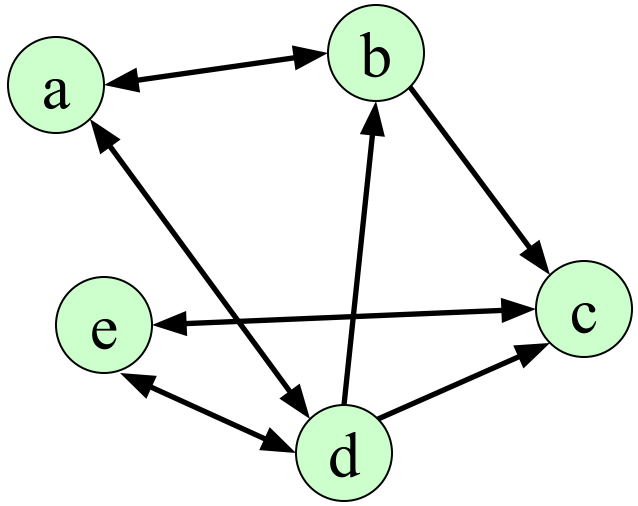
Основно свойство: Дървото има само един път от всяко листо до корена. Всеки граф с такова свойство е дърво.

Нека обобщим: Дърво на търсене е резултат от реализирането на търсене в едно ПС чрез постъпково определяне на актуално състояние и разширяване на ПС с нови състояния до откриването на целта. Решението е последователността от действия (ВО) чрез които се преминава от началното до целевото състояние.

Дърво на търсене – структура, която се получава като резултат от търсене в едно ПС чрез постъпково определяне на актуално състояние и разширяване на ПС с нови състояния до откриването на целта.
Дървото - свързан, ориентиран граф, който не съдържа затворени вериги.
Корен на дърво – начален възел
Вътрешни възли- възли в дървото, от които изличат ребра
Листа на дървото – възли, от които не излизат ребра.

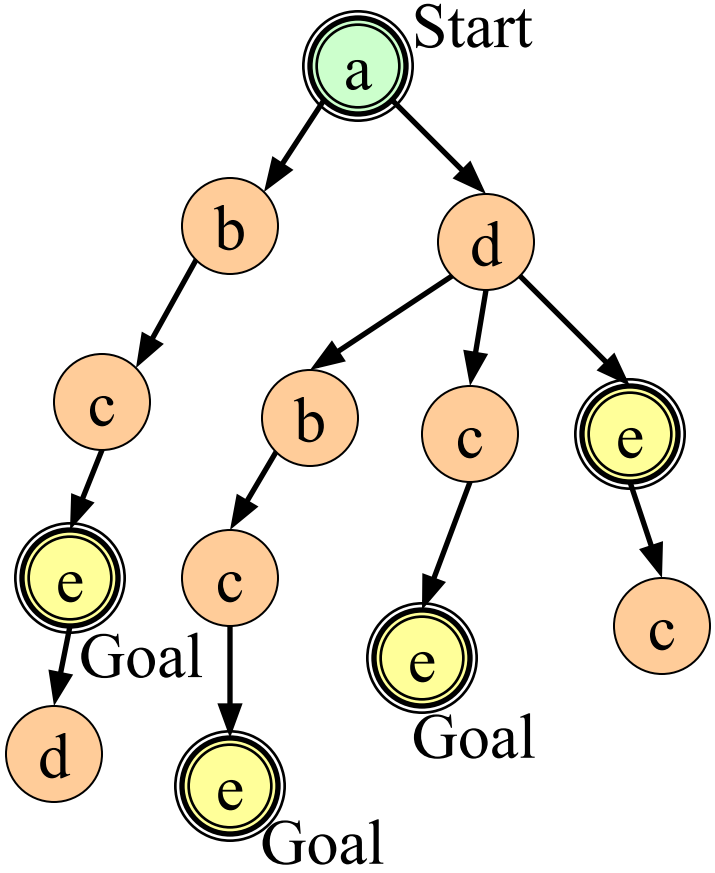
Представяне на графа

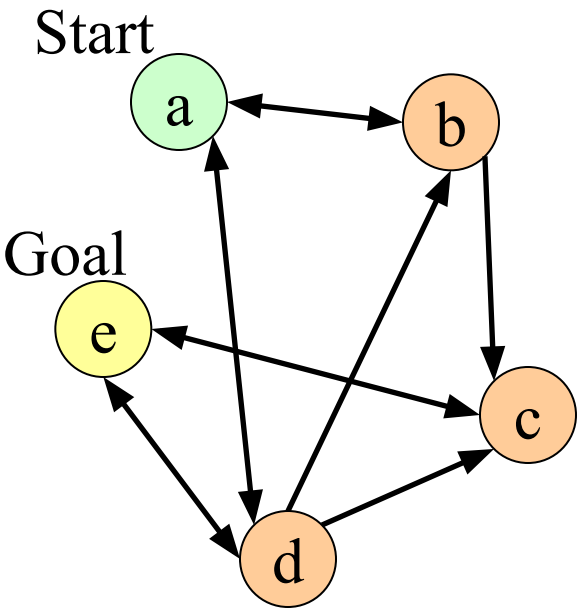
```
arc(Node1,Node2).
arc(Node1,Node2,Cost).

arc(a,b).       arc(d,e).
arc(a,d).       arc(e,c).
arc(b,a).       arc(e,d).
arc(b,c).
arc(c,e).
arc(d,a).
arc(d,b).
arc(d,c).
```



Построяване на дървото

```
arc(a,b).
arc(a,d).
arc(b,a).
arc(b,c).
arc(c,e).
arc(d,a).
arc(d,b).
arc(d,c).
arc(d,e).
arc(e,c).
arc(e,d).
```





Основни алгоритми

Пълно (сляпо) търсене

методите използват само необходимата информация за представяне на проблема. Всичкото, което могат да правят е генериране на наследници и различаване на целеви състояния. Отделните стратегии се различават по последователността, в която се разширяват възлите.

Информирано (евристично) търсене

Освен информацията, необходима за формулиране на „проблема“, разполагаме с чувствителна допълнителна информация, която може да се добие от реалната задача или от разглежданата приложна област. Тази допълнителна информация се използва за определяне на възела, който да бъде разширен най-напред. Обикновено това знание ни позволява да зададем някаква оценъчна функция. Търсенето, използващо такава функция се нарича „първо най-добро търсене“. Съществуват различни методи, които се различават по използваната оценъчна функция.

Сляпо търсене

При сляпото търсене всички възли-наследници са равноправни и всички те се приемат в границата. Това е така, понеже не разполагаме с информация, която да ни позволи да категоризираме възлите по отношение на „благонадеждност“ за намиране на (добро) решение.

Едно от основните предимства на сляпото търсене е, че то ни дава възможност за систематично „претърсване“ на цялото ПС. В резултат намираме всички решения на проблема. Ако е зададена функция на разходите, този вид търсене позволява да се открие „най-доброто“ решение.

Недостатъците на „сляпото търсене“ са свързани с така наречения „комбинаторен взрив“, който ще бъде представен по-късно.

Съществуват три базови методи за сляпо търсене:

Търсене в ширина;

Търсене в дълбочина;

Търсене с еднакви разходи.

Евристично търсене

За разлика от сляпото търсене при евристичното търсене разполагаме с допълнителна информация, която ни позволява да категоризираме възлите по отношение на „благонадеждност“ за намиране на (добро) решение. Така може да селектираме възлите-наследници и само някои от тях да се приемат в границата, т.е. само те могат да бъдат разширявани в бъдеще. Евристичното търсене възниква в резултат на усилията за справяне с неблагоприятните ефектите на „комбинаторния взрив“.

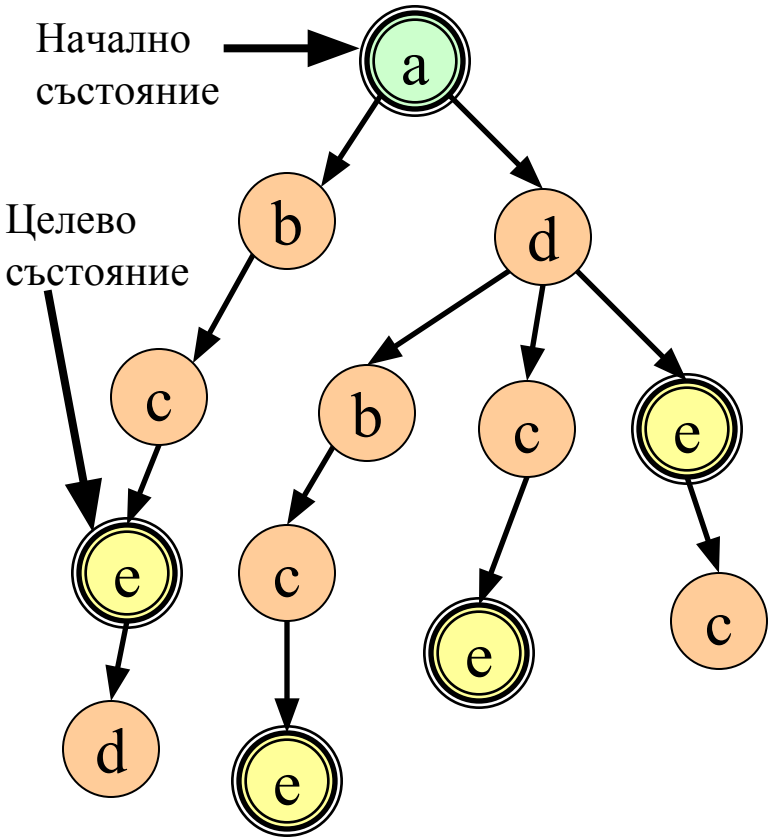
Съществуват два базови методи за евристично търсене:

Лакомо търсене;

A* търсене.

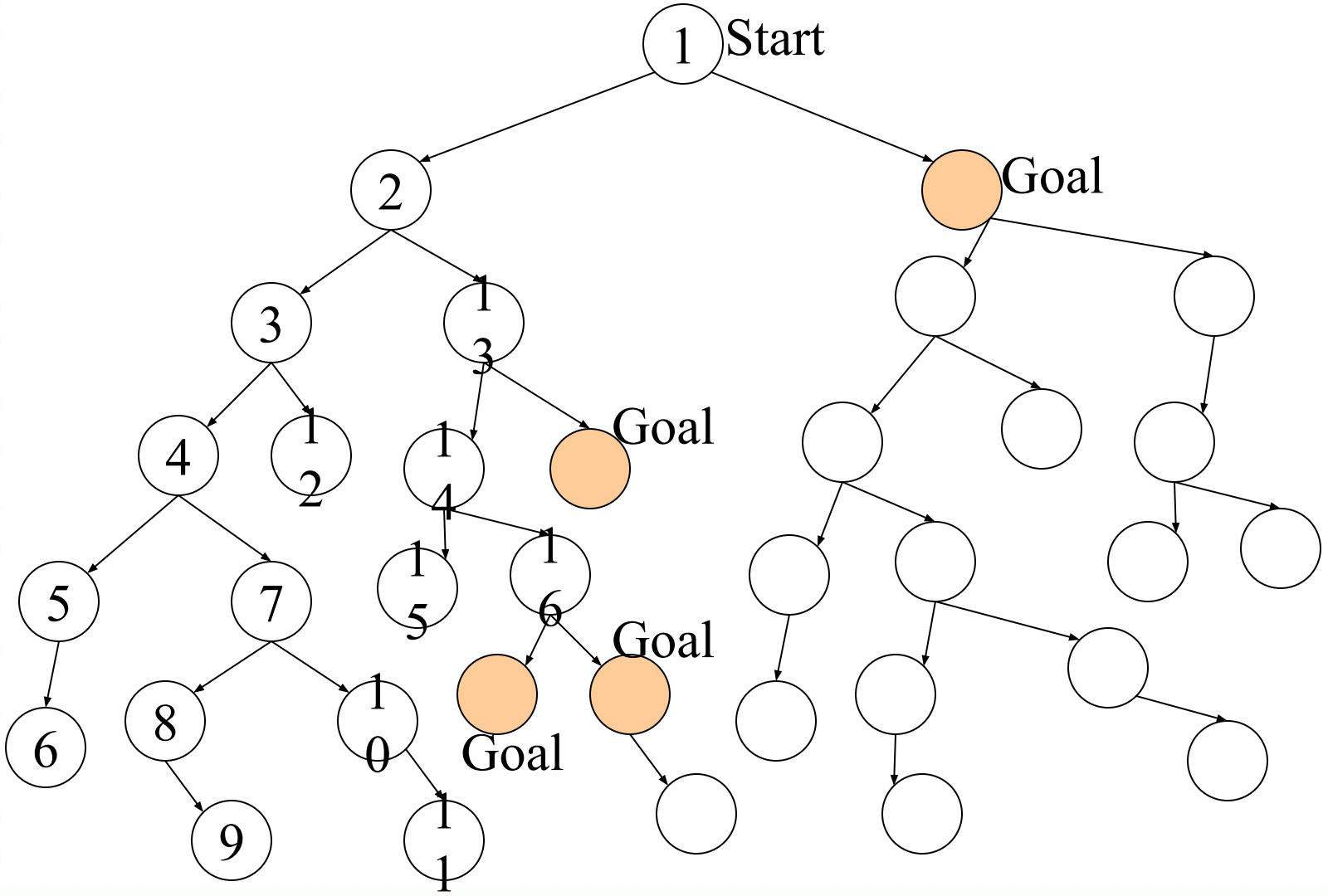
За резултатно използване на методите за търсене от изключително значение е оценка на тяхната „комплексност“.

Представяне на фронта при Търсене в дълбочина




```
[[a]]
[[b a] [d a]]
[[c b a] [d a]]
[[e c b a] [d a]]
I-во решение:
[e c b a] => [a b c e]
Преудовлетворяване:
[[d a]]
[[b d a] [c d a] [e d a]]
[[c b d a] [c d a] [e d a]]
[[e c b d a] [c d a] [e d a]]
II-ро решение:
[e c b d a] => [a d b c e]
Преудовлетворяване:
[[c d a] [e d a]]
[[e c d a] [e d a]]
III-то решение:
[e c d a] => [a d c e]
Преудовлетворяване:
[[e d a]]
IV-то решение:
[e d a] => [a d e]
```

Търсене в дълбочина



Търсене в дълбочина (Depth-first)

```
depth_first([Path|Stack],Goal,FinalPath) :-
    extend(Path,NewPaths),
    append(NewPaths, Stack, NewStack),
    depth_first(NewStack,Goal,FinalPath).
```

Търсене в дълбочина (Depth-first)

```
depth_first([[Goal|Path]|_],Goal,FinalPath):-
    reverse([Goal|Path],FinalPath).
```

Разширяване на фронта

```
extend([Node|Path],NewPaths) :-
    findall([NewNode,Node|Path],
            (arc(Node,NewNode),
            \+ member(NewNode,[Node|Path])),
            NewPaths).
```

Търсене в дълбочина (Depth-first)

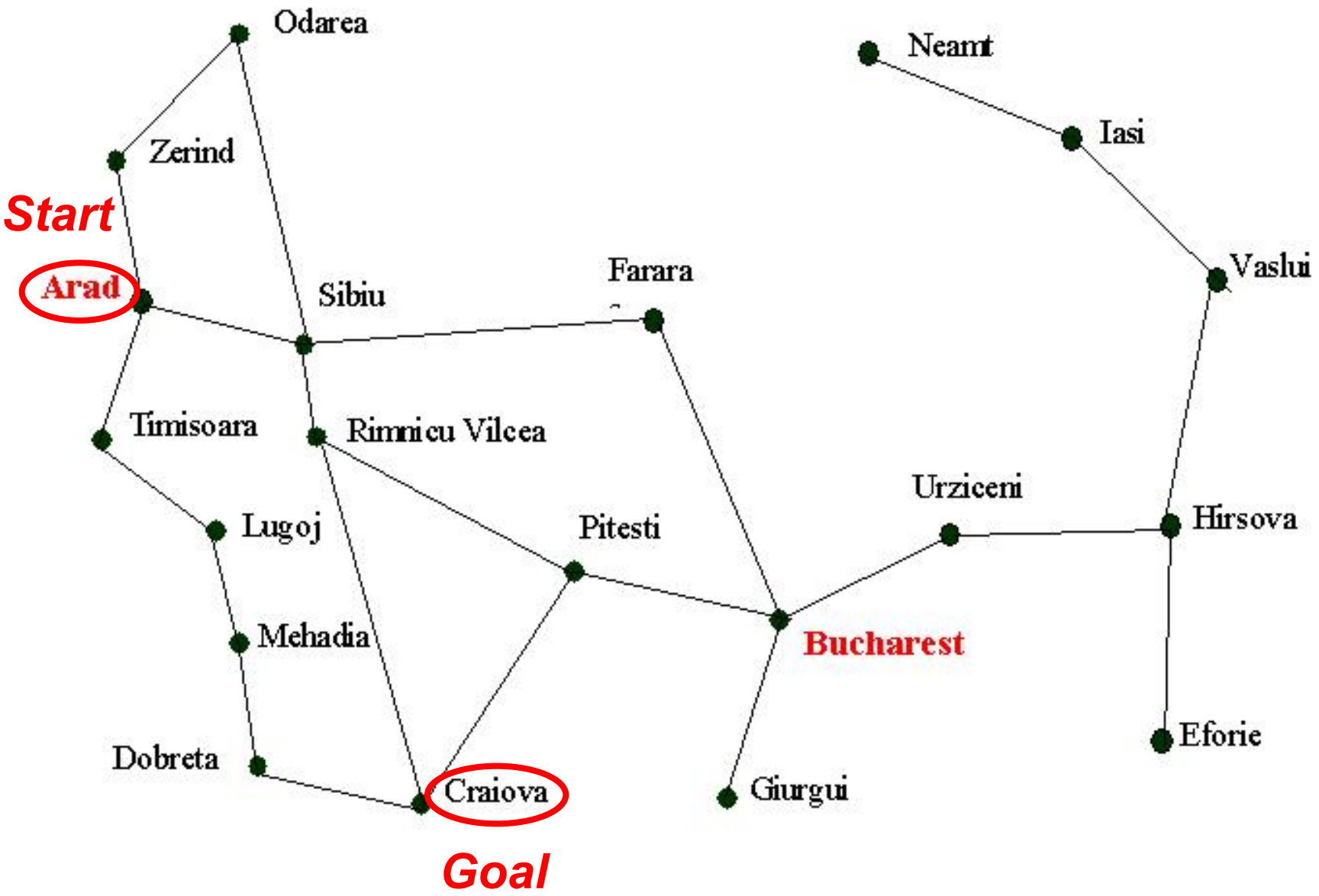
```
depth_first([[Goal|Path]|_],Goal,FinalPath):-
      reverse([Goal|Path],FinalPath).

depth_first([Path|Stack],Goal,FinalPath)  :-
     extend(Path,NewPaths),
     append(NewPaths,Stack,NewStack),
     depth_first(NewStack,Goal,FinalPath).

extend([Node|Path],NewPaths)  :-
     findall([NewNode,Node|Path],
               (arc(Node,NewNode),
               \+ member(NewNode,[Node|Path])),
               NewPaths).
```

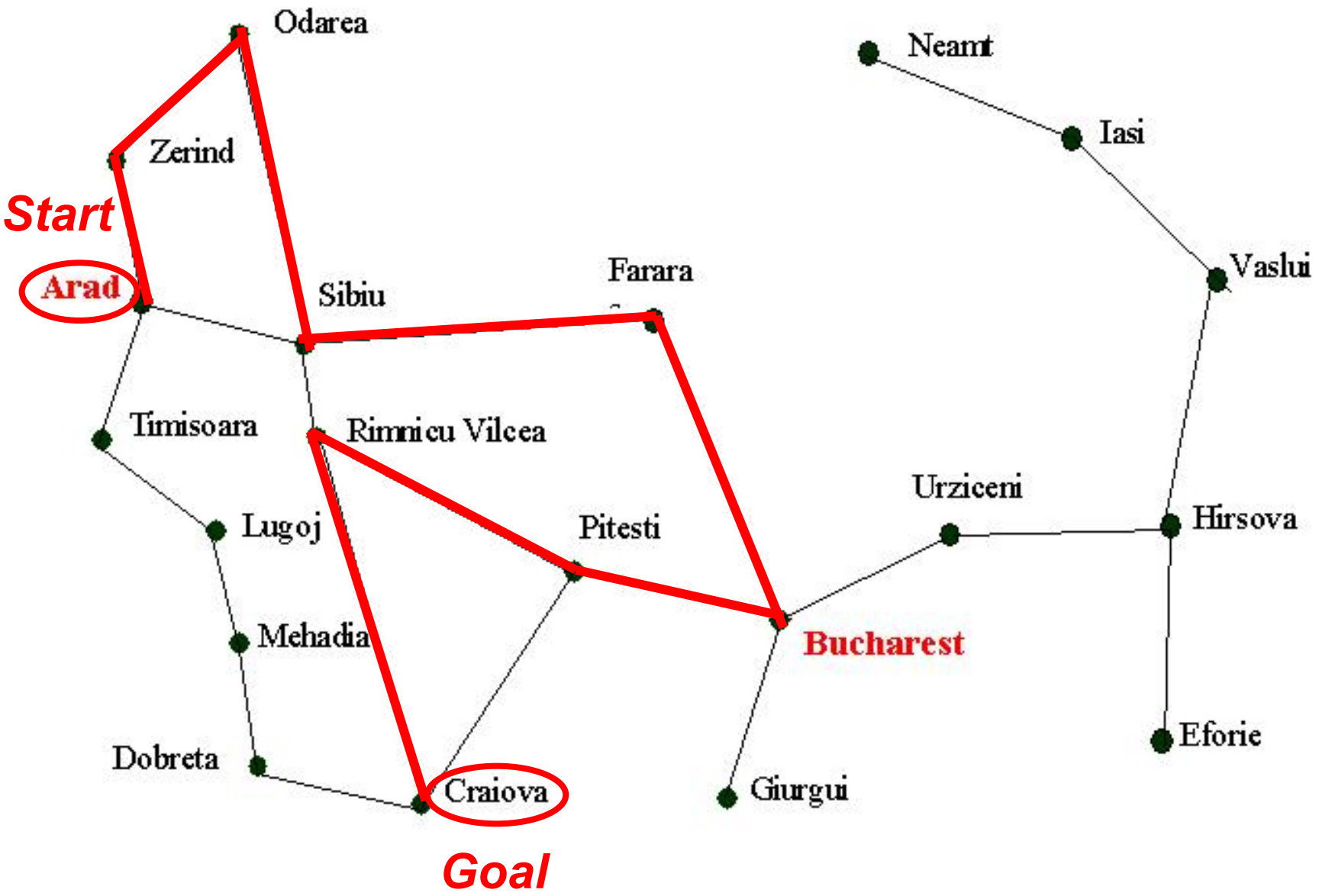
Пример

```
arc('Arad','Zerind',75).
arc('Arad','Sibiu',140).
arc('Arad','Timisoara',118).
arc('Bucharest','Fagaras',211).
arc('Bucharest','Pitesti',101).
arc('Bucharest','Giurgiu',90).
arc('Bucharest','Urziceni',85).
arc('Craiova','Dobreta',120).
arc('Craiova','Rimnicu',146).
arc('Craiova','Pitesti',138).
arc('Dobreta','Mehadia',75).
arc('Dobreta','Craiova',120).
arc('Eforie','Hirsova',86).
arc('Fagaras','Sibiu',99).
arc('Fagaras','Bucharest',211).
arc('Giurgiu','Bucharest',90).
arc('Hirsova','Urziceni',98).
arc('Hirsova','Eforie',86).
arc('Iasi','Neamt',87).
arc('Iasi','Vaslui',92).
arc('Lugoj','Timisoara',111).
arc('Lugoj','Mehadia',70).
arc('Mehadia','Lugoj',70).
arc('Mehadia','Dobreta',75).
arc('Neamt','Iasi',87).
arc('Oradea','Zerind',71).
arc('Oradea','Sibiu',151).
arc('Pitesti','Rimnicu',97).
arc('Pitesti','Craiova',138).
arc('Pitesti','Bucharest',101).
arc('Rimnicu','Sibiu',80).
arc('Rimnicu','Pitesti',97).
arc('Rimnicu','Craiova',146).
arc('Sibiu','Arad',140).
arc('Sibiu','Oradea',151).
arc('Sibiu','Fagaras',99).
arc('Sibiu','Rimnicu',80).
arc('Timisoara','Arad',118).
arc('Timisoara','Lugoj',111).
arc('Urziceni','Bucharest',85).
arc('Urziceni','Hirsova',98).
arc('Urziceni','Vaslui',142).
arc('Vaslui','Iasi',92).
arc('Vaslui','Urziceni',142).
arc('Zerind','Arad',75).
arc('Zerind','Oradea',71).
```

Пример

```
| ?-
  depth_first([['Arad']],'Craiova',Path).

Path = ['Arad','Zerind','Oradea','Sibiu',
        'Fagaras','Bucharest','Pitesti',
        'Rimnicu','Craiova']?;
Path= ['Arad','Zerind','Oradea','Sibiu',
       'Fagaras','Bucharest',
       'Pitesti','Craiova']
```

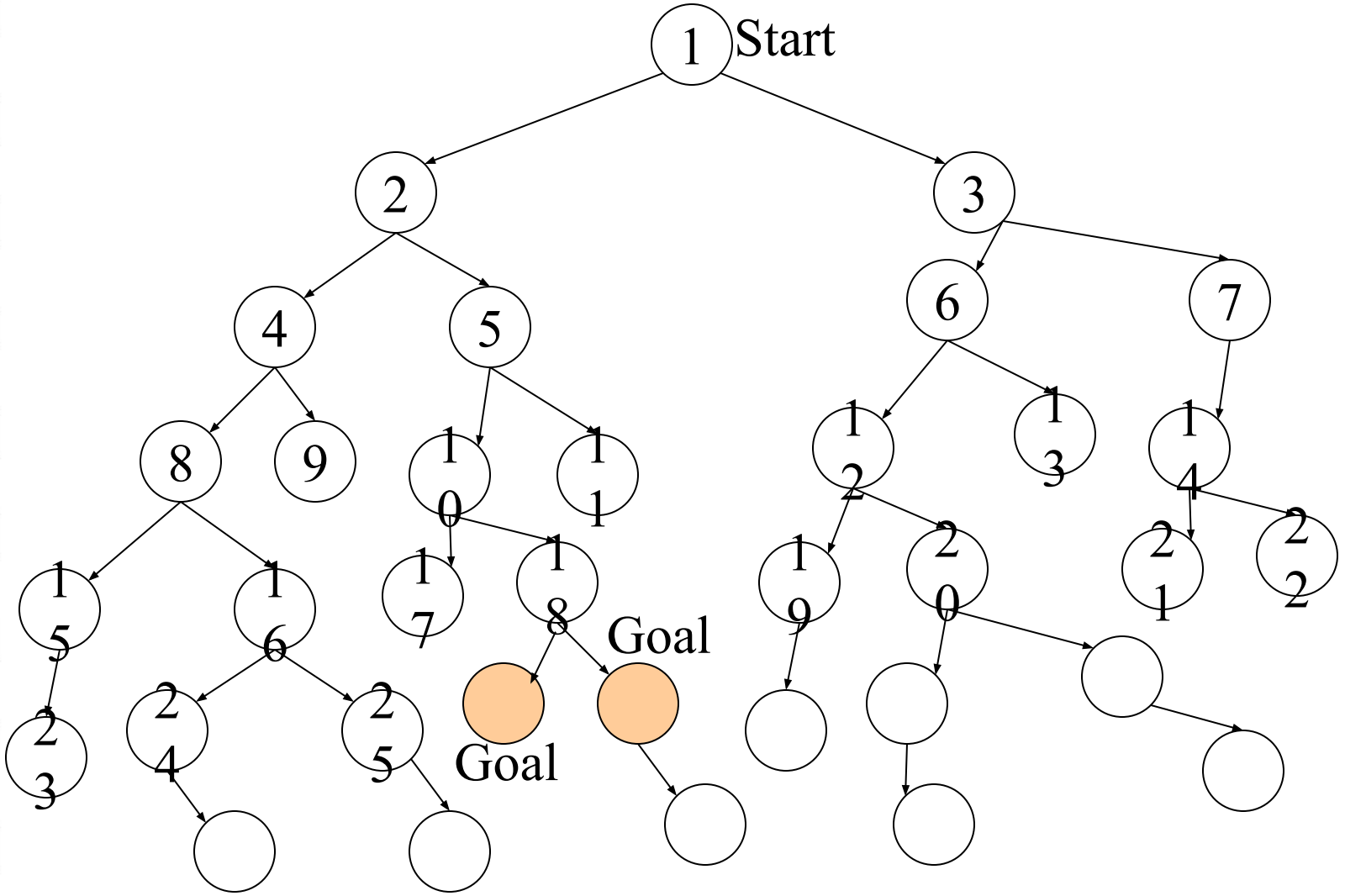

Odarea
Neamt
Zerind
Iasi
Start
Arad
Sibiu
Farara
Vaslui
Timisoara
Rimnicu Vilcea
Urziceni
Lugoj
Pitesti
Hirsova
Mehadia
Bucharest
Eforie
Dobreta
Craiova
Giurgui
Goal


Odarea
Neamt
Zerind
Iasi
Start
Arad
Sibiu
Farara
Vaslui
Timisoara
Rimnicu Vilcea
Urziceni
Lugoj
Pitesti
Hirsova
Mehadia
Bucharest
Eforie
Dobreta
Craiova
Giurgui
Goal

Търсене в ширина

- При търсенето в ширина фронтът се обработва като опашка
- Ако фронта е **[p1,p2, . . .pn]**
  - избира се **p1**
  - пътищата p_1', p_1'',…, $p_1^{(k)}$, които разширяват **p1** се добавят в края на опашката (след **pn**), т.е. **[p2, …,pn,** p_1', p_1'',…,$p_1^{(k)}$**]** и се обработват едва след като всички пътища **p2, …, pn**, се изследват
- Намира най-краткия път

Търсене в ширина



Търсене в дълбочина (Depth-first)

```
depth_first([Path|Stack],Goal,FinalPath) :-
    extend(Path,NewPaths),
    append(NewPaths, Stack, NewStack),
    depth_first(NewStack,Goal,FinalPath).
```

Търсене в ширина (Breadth-first)

```
breadth_first([Path|Stack],Goal,FinalPath)
:-
    extend(Path,NewPaths),
    append(NewPaths, Stack, NewStack),
    breadth_first(NewStack,Goal,FinalPath).
```

Търсене в ширина (Breadth-first)

```
breadth_first([Path|Queue],Goal,FinalPath):-
    extend(Path,NewPaths),
    append(NewPaths, Queue, NewQueue),
    breadth_first(NewQueue,Goal,FinalPath).
```

Търсене в ширина (Breadth-first)

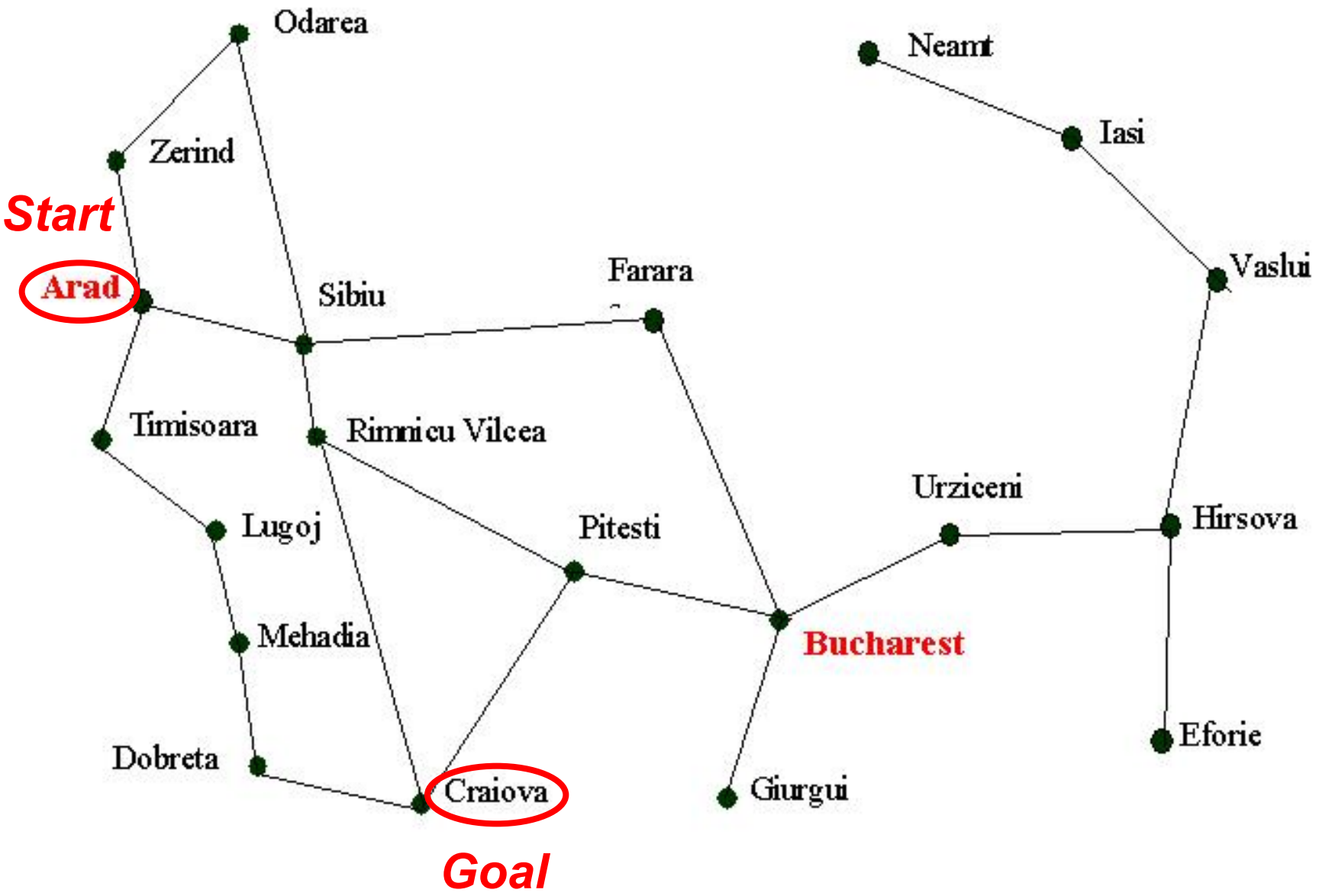
```
breadth_first([Path|Queue],Goal,FinalPath):-
    extend(Path,NewPaths),
    append(Queue, NewPaths, NewQueue),
    breadth_first(NewQueue,Goal,FinalPath).
```

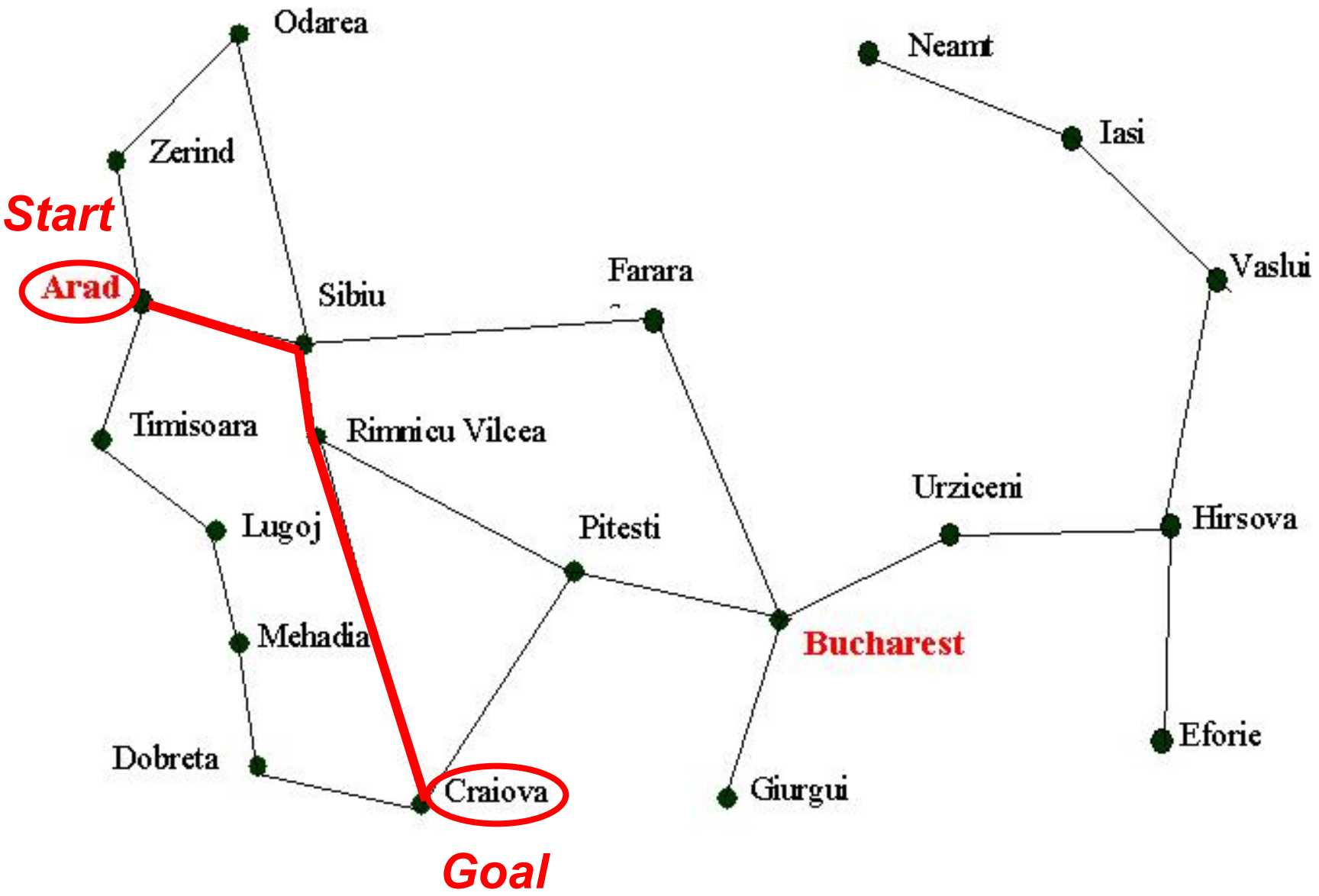
Търсене в ширина (Breadth-first)

```
breadth_first([[Goal|Path]|_],Goal, FinalPath):-
      reverse([Goal|Path],FinalPath).

breadth_first([Path|Queue],Goal,FinalPath) :-
     extend(Path,NewPaths),
     append(Queue,NewPaths,NewQueue),
     breadth_first(NewQueue,Goal,FinalPath).

extend([Node|Path],NewPaths) :-
     findall([NewNode,Node|Path],
               (arc(Node,NewNode,_),
               \+ member(NewNode,[Node|Path])),
               NewPaths).
```

Пример

```
?- breadth_first([['Arad']],'Craiova',Path).

Path =['Arad','Sibiu','Rimnicu','Craiova']
```


Odarea
Neamt
Zerind
Iasi
Start
Arad
Sibiu
Farara
Vaslui
Timisoara
Rimnicu Vilcea
Urziceni
Lugoj
Pitesti
Hirsova
Mehadia
Bucharest
Eforie
Dobreta
Craiova
Giurgui
Goal

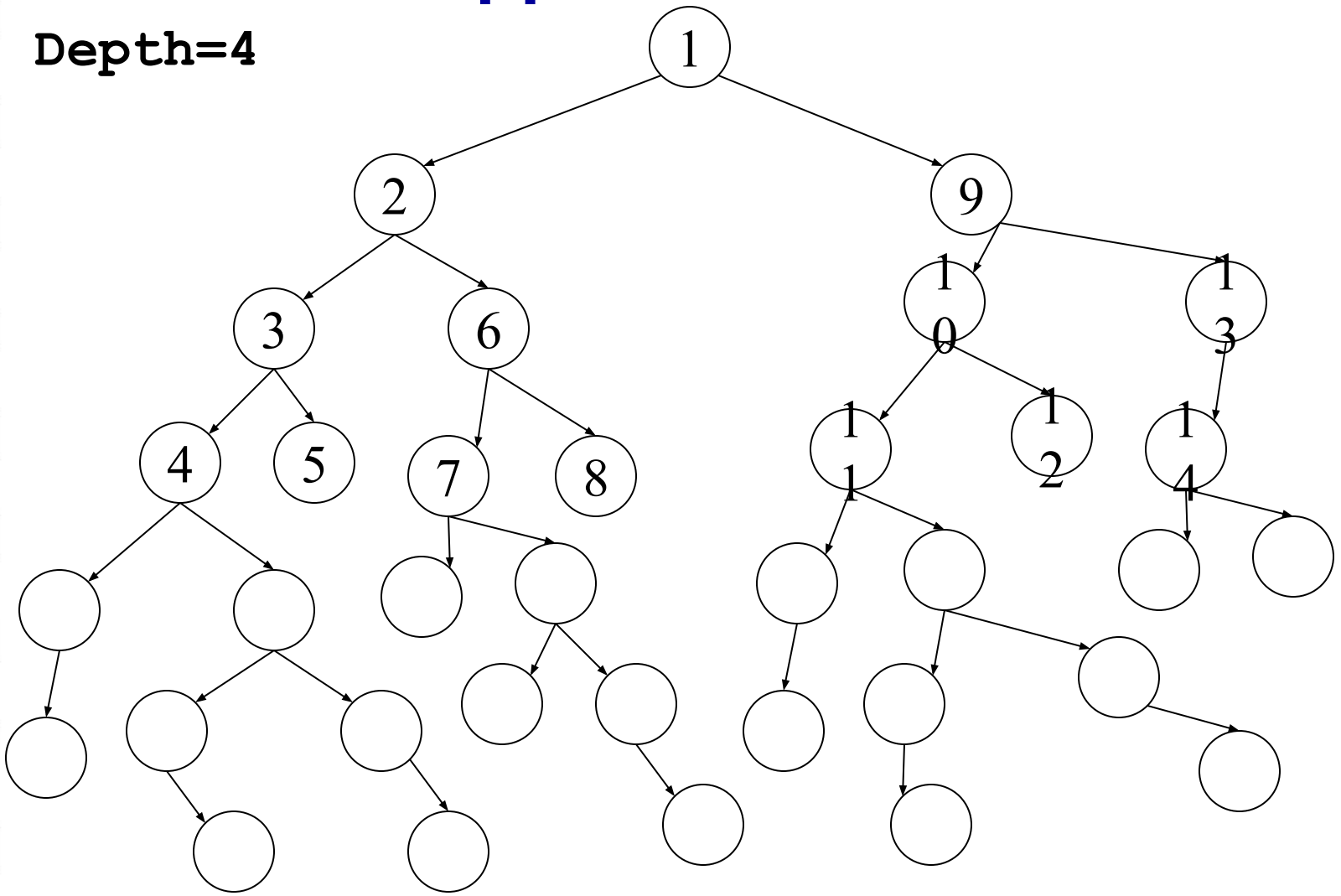

Odarea
Neamt
Zerind
Iasi
Start
Arad
Sibiu
Farara
Vaslui
Timisoara
Rimnicu Vilcea
Urziceni
Lugoj
Pitesti
Hirsova
Mehadia
Bucharest
Eforie
Dobreta
Craiova
Giurgui
Goal

Търсене в ограничена дълбочина

- Задава се максимална дълбочина **`Depth`**, на която ще търсим решение
- Търсим път, който минава през не повече от **`Depth`** на брой върха на графа

Търсене в ограничена дълбочина

Depth=4



Търсене в дълбочина (Depth-first)

```
depth_first([Path|Stack],Goal,FinalPath) :-
    extend(Path,NewPaths),
    append(NewPaths, Stack, NewStack),
    depth_first(NewStack,Goal,FinalPath).
```

Търсене в ограничена дълбочина (Depth-bound)

```
depth_bound(Depth,[Path|Stack],Goal,FinalPath):-
    extend(Path,NewPaths),
    append(NewPaths, Stack, NewStack),
    depth_bound(Depth,NewStack,Goal,FinalPath).
```

Търсене в ограничена дълбочина (Depth-bound)

```
depth_bound(Depth,[Path|Stack],Goal,FinalPath):-
    extend_bound(Depth,Path,NewPaths),
    append(NewPaths, Stack, NewStack),

  depth_bound(Depth,NewStack,Goal,FinalPath)
  .
```

Търсене в ограничена дълбочина (Depth-bound)

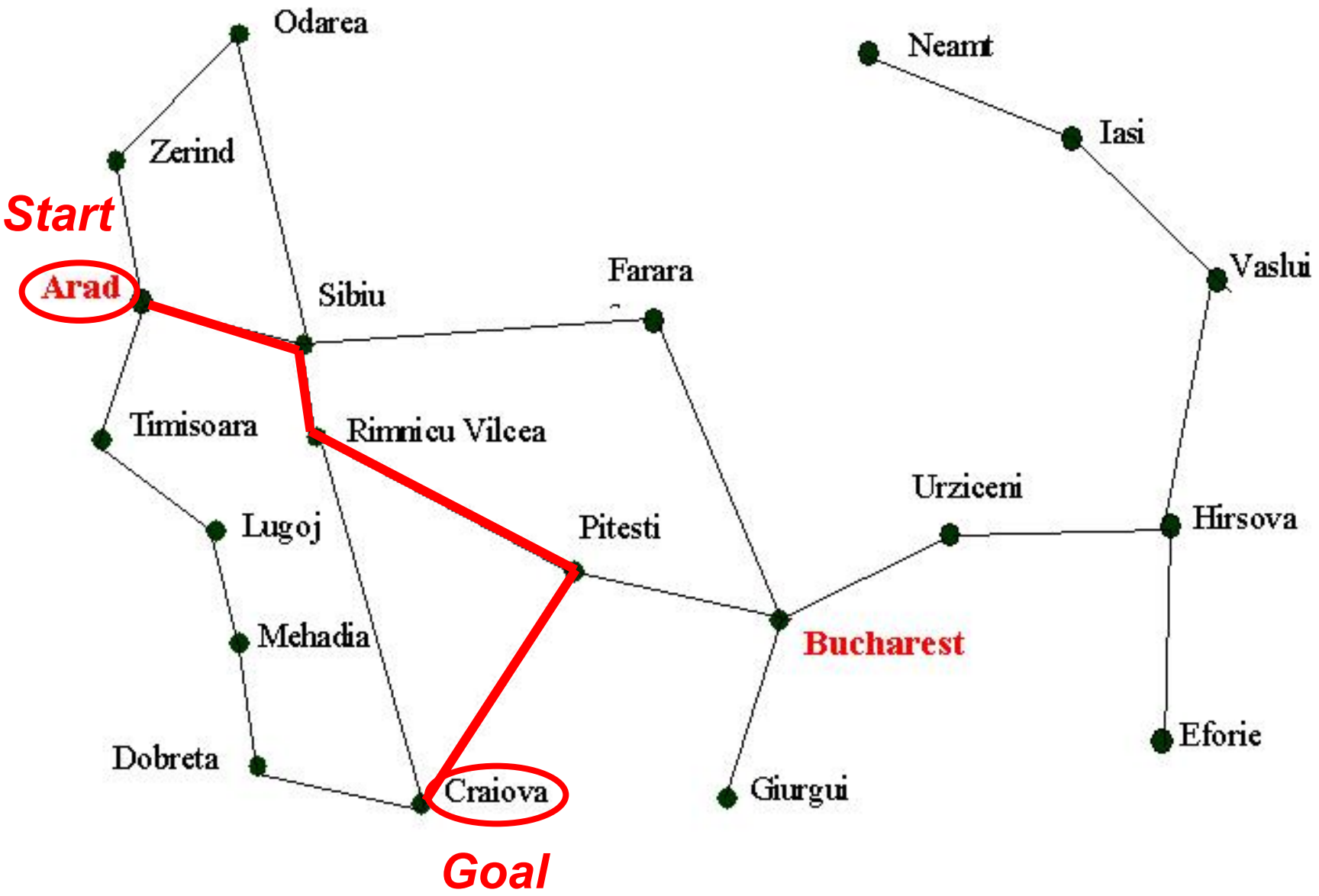
```
extend_bound(Depth,Path,NewPaths) :-
    length(Path,Len),
    Len < Depth, !,
    extend(Path,NewPaths).

extend_bound(_,_,[]).

extend([Node|Path],NewPaths) :-
    findall([NewNode,Node|Path],
            (arc(Node,NewNode,_),
            \+ member(NewNode, [Node|Path])),
            NewPaths).
```

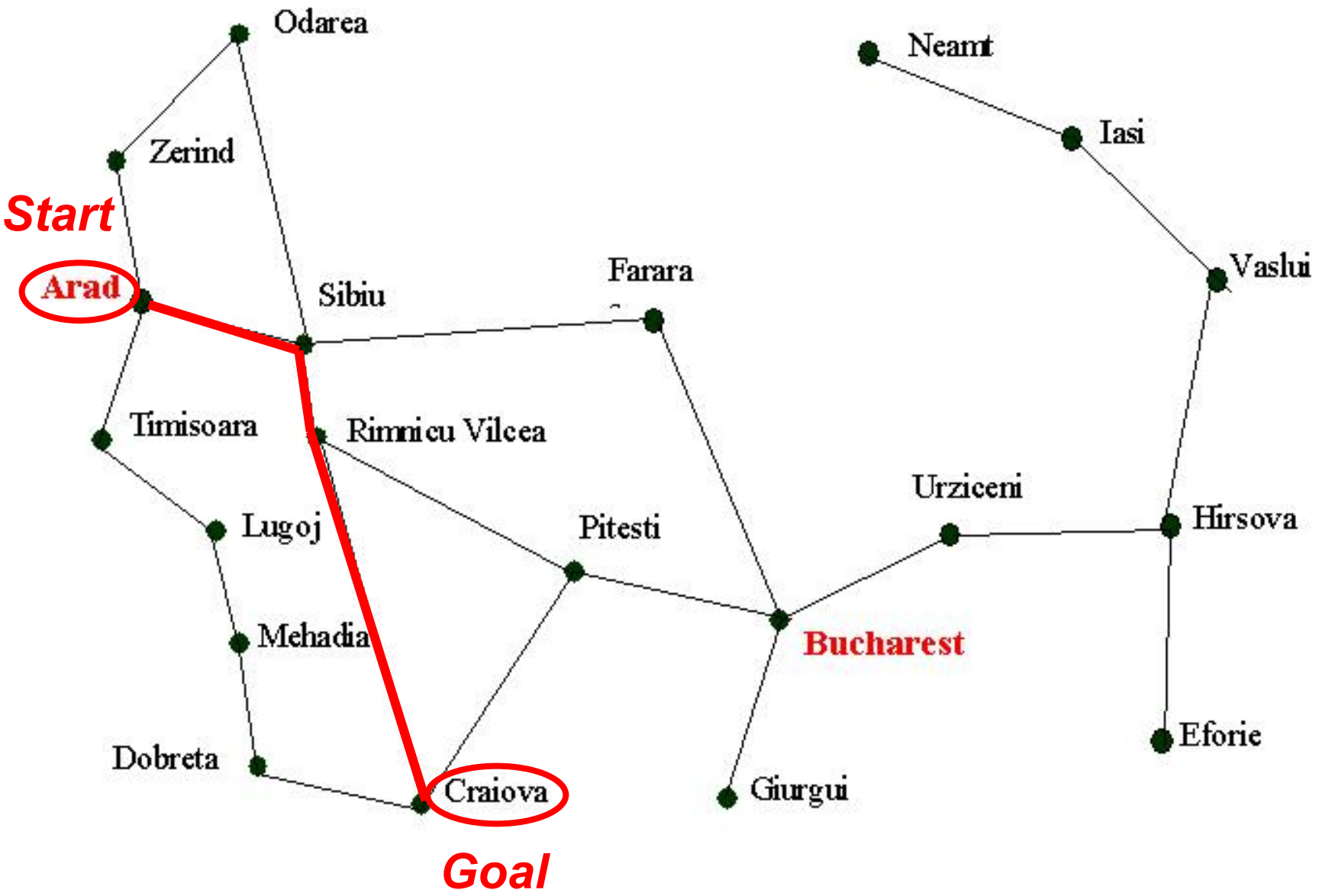
Търсене в ограничена дълбочина (Depth-bound)

```
depth_bound(Depth,[[Goal|Path]|_],Goal,FinalPath):-
      reverse([Goal|Path],FinalPath).

depth_bound(Depth,[Path|Stack],Goal,FinalPath):-
     extend_bound(Depth,Path,NewPaths),
     append(NewPaths,Stack,NewStack),
     depth_bound(Depth,NewStack,Goal,FinalPath).

extend_bound(Depth,Path,NewPaths) :-
     length(Path,Len),
     Len < Depth, !,
     extend(Path,NewPaths).

extend_bound(_,_,[]).
```

Пример

```
|?-
  depth_bound(5,[['Arad']],'Craiova',Path).

Path=['Arad','Sibiu','Rimnicu',
     'Pitesti','Craiova'] ?;

Path=['Arad','Sibiu','Rimnicu','Craiova'] ?;

no
```


Odarea
Neamt
Zerind
Iasi
Start
Arad
Sibiu
Farara
Vaslui
Timisoara
Rimnicu Vilcea
Urziceni
Lugoj
Pitesti
Hirsova
Mehadia
Bucharest
Eforie
Dobreta
Craiova
Giurgui
Goal


Odarea
Neamt
Zerind
Iasi
Start
Arad
Sibiu
Farara
Vaslui
Timisoara
Rimnicu Vilcea
Urziceni
Lugoj
Pitesti
Hirsova
Mehadia
Bucharest
Eforie
Dobreta
Craiova
Giurgui
Goal

Итеративно търсене по нива

- Често пространството от състояния е много голямо и се изискват много ресурси (време и памет), за да бъде поддържан големия фронт, което прави процеса на търсене много бавен и труден.
- Основната идея е да се преизчисляват елементите от фронта, вместо да се пазят
- Прави се търсене в ограничена дълбочина при **`Depth=1, 2, …`**
- Ако целевият път не се намери на дълбочина **`d`** се търси на дълбочина **`d+1`**.
- Намира най-краткия път

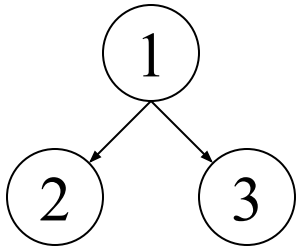
Итеративно търсене по нива

- Ако не съществува път от началното до целевото състояние в дадения граф и вече е построено и е изследвано цялото пространство от състоянията, тогава всяко нарастване на дълбочината **Depth** ще води до построяване на едно и също пространство и многократното му претърсване.
- За да се гарантира завършване на процеса, той се ограничава, като нарастването на дълбочината **Depth** се ограничава до предварително зададена максимална стойност **Max**, т.е. **Depth ≥ Max.**

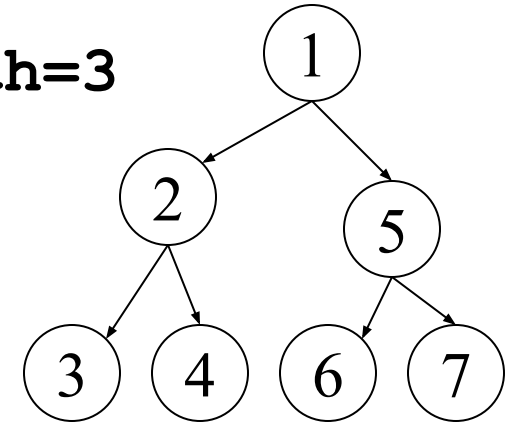
Итеративно търсене по нива

Depth=1

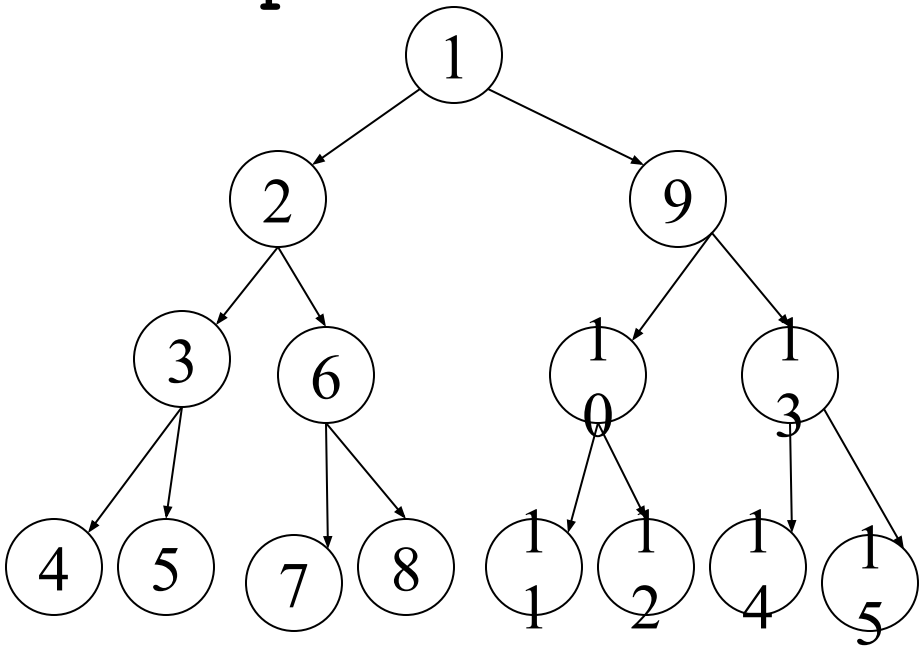
Depth=2



Depth=3



Depth=4



Итеративно търсене по нива (Iterative deepening)

- Тук като груба оценка за максималната дълбочина е използван броя на дъгите в графа.
- Търсенето започва при **Depth=1**

```
iterative_deepening(Stack,Goal,FinalPath) :-
   findall(arc(X,Y,Z),arc(X,Y,Z),Graph),
   length(Graph,Max),
   iterative_deepening1(1,Stack,Goal,FinalPath,Max).
```

Итеративно търсене по нива (Iterative deepening)

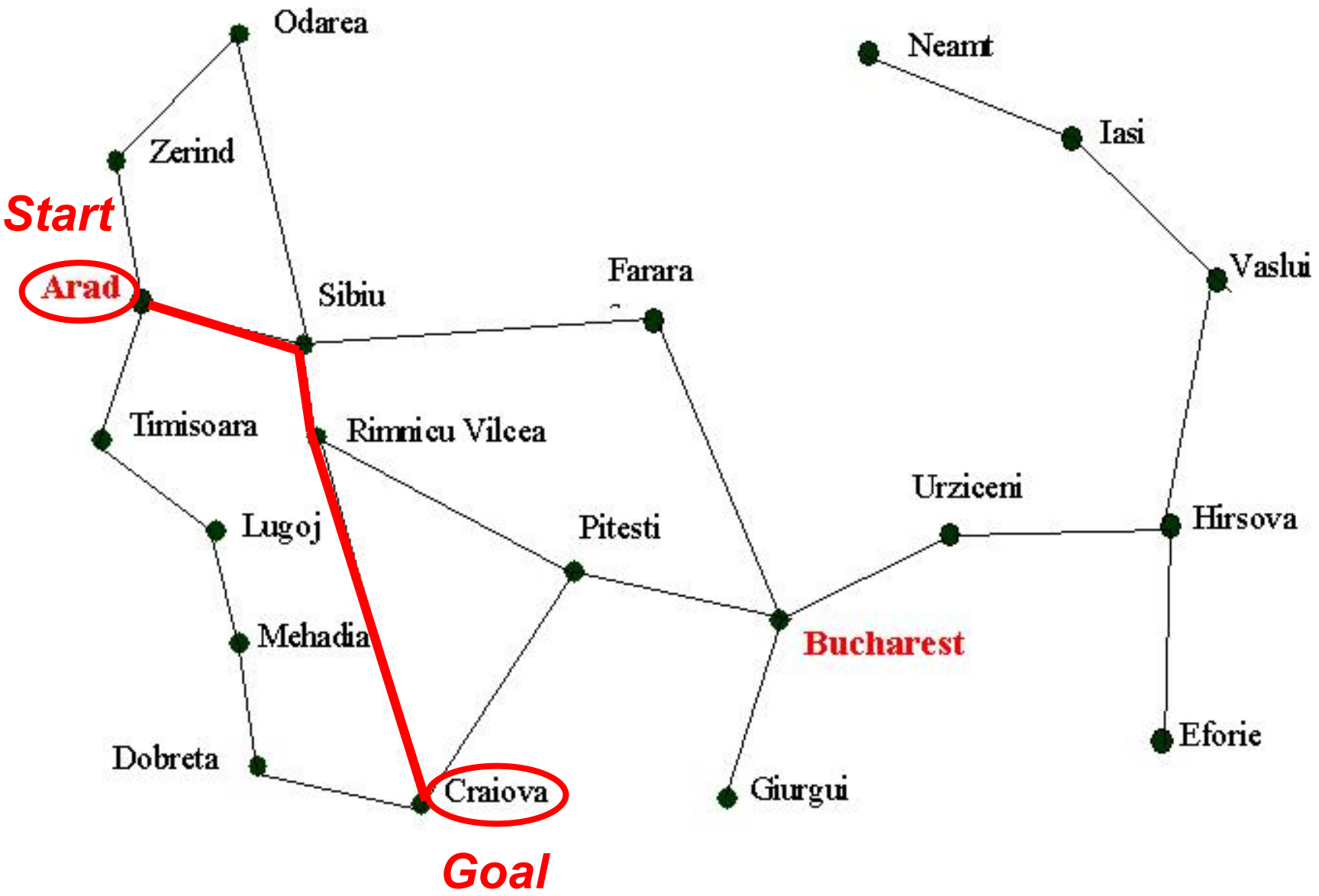
```
iterative_deepening1(Depth,Stack,Goal,Path,Max):-
      nl,write(depth=Depth),
      depth_bound(Depth,Stack,Goal,Path).

iterative_deepening1(Depth,Stack,Goal,Path,Max):-
    Depth1 is Depth+1,
    Depth1 =< Max,!,
    iterative_deepening1(Depth1,Stack,Goal,Path,Max).
```

Итеративно търсене по нива (Iterative deepening)

```
iterative_deepening(Stack,Goal,Path) :-
    findall(arc(X,Y,Z),arc(X,Y,Z),Graph),
    length(Graph,N),
    iterative_deepening1(1,Stack,Goal,Path,N).

iterative_deepening1(Depth,Stack,Goal,Path,Max) :-
    nl,write(depth=Depth),
    depth_bound(Depth, Stack,Goal,Path).

iterative_deepening1(Depth,Stack,Goal,Path,Max) :-
    Depth1 is Depth+1, Depth1 =< Max,!,
  iterative_deepening1(Depth1,Stack,Goal,Path,Max)
  .
```

Пример

```
|?- iterative_deepening([['Arad']],'Craiova',Path).

depth=1
depth=2
depth=3
depth=4
Path = ['Arad','Sibiu','Rimnicu','Craiova'] ?
```


Odarea
Neamt
Zerind
Iasi
Start
Arad
Sibiu
Farara
Vaslui
Timisoara
Rimnicu Vilcea
Urziceni
Lugoj
Pitesti
Hirsova
Mehadia
Bucharest
Eforie
Dobreta
Craiova
Giurgui
Goal

Основни алгоритми

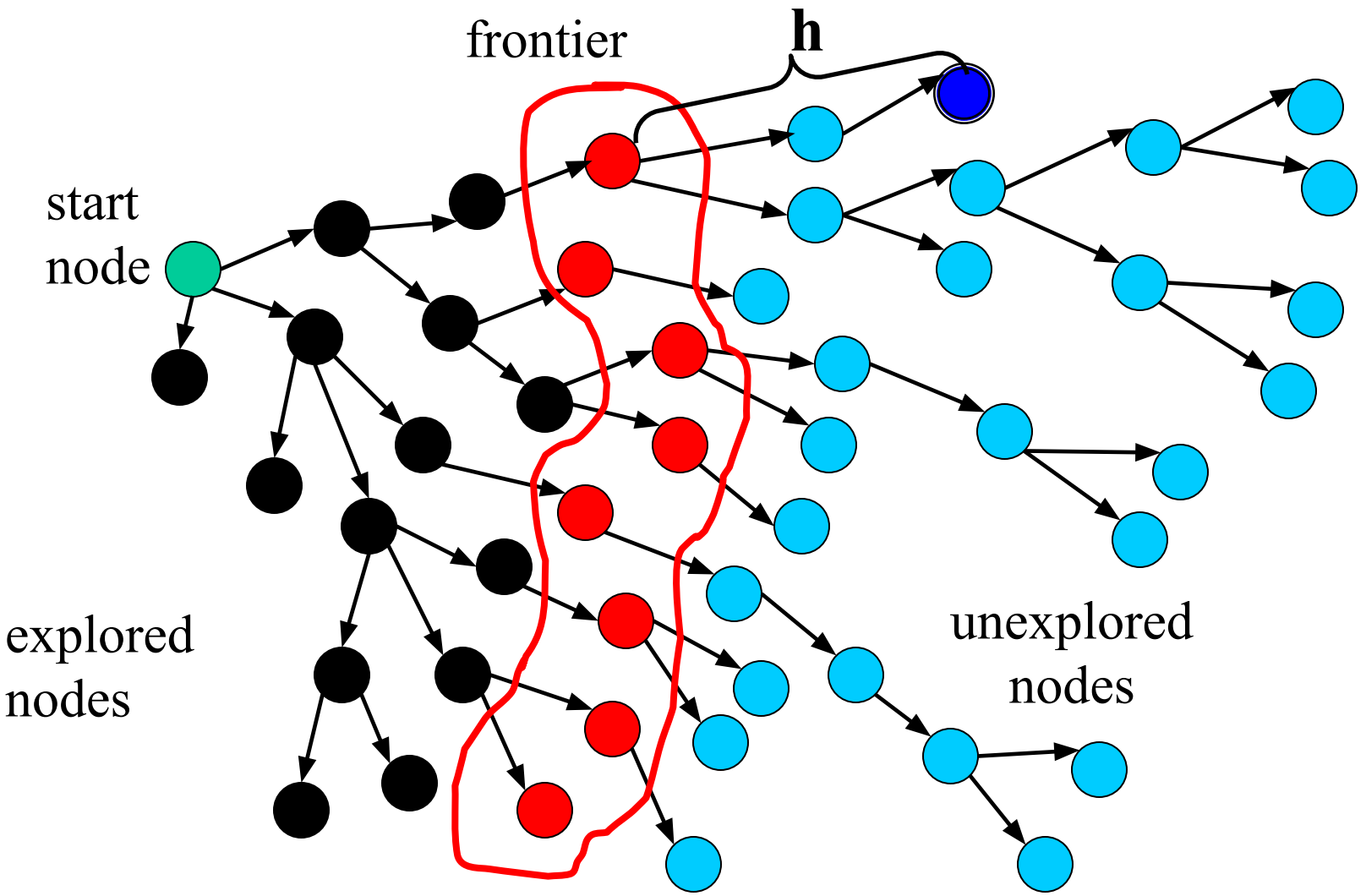
- Неинформирано (сляпо) търсене
- Информирано (евристично) търсене

Общ алгоритъм

- Поддържа се фронт/граница (frontier) от пътищата, които са били изследвани
- По време на процеса на търсене фронта се разширява в посока към неизследваните възли, докато се достигне до целеви възел
- При разширяването на фронта се изчисляват евристичните оценки на пътищата
- Фронтът се сортира в нарастващ ред на евристичните оценки на пътищата

Основни алгоритми

- Информирано (евристично) търсене
  - метод на най-доброто спускане **(Best-first search)**
  - търсене в лъч **(Beam search)**
  - метод на изкачването **(Hill Climbing)**
  - **A***


frontier
h
start
node
explored
nodes
unexplored
nodes

Евристична оценка

- Евристична оценка:
  - оценява броя ходове от дадено състояние до целта или дава стойността на пътя до целта
  - Трябва да е 0 в целевото състояние
  - Трябва да се изчислява лесно, т.е. да не изисква прекалено много ресурси (време и памет)
  - **h(Path,Goal,H)**

```
h([Node|Path],Goal,H) :-
   stright_line_distance(Node,Goal,H).
```

Метод на най-доброто спускане
(Best-first search)

- Фронтът се обработва като сортиран списък в нарастващ ред на текущите евристични оценки на пътищата
- Ако фронтът е $[p_1, p_2, \ldots, p_n]$
  - избира се p_1
  - пътищата p_1', p_1'',…, $p_1^{(k)}$, които разширяват **p1** се добавят към фронта и той се сортира в нарастващ ред на оценките на пътищата $[q_1, q_2, \ldots, q_{n+k}]$
  - на следващата стъпка се обработва първо пътя с най-добра текуща оценка, т.е. q_1

Метод на най-доброто спускане
(Best-first search)

```
best_first([[Goal|Path]|_],Goal,FinalPath):-
     reverse([Goal|Path],FinalPath).

best_first([Path|List],Goal,FinalPath):-
    extend(Path,NewPaths),
    append(List,NewPaths,NewList),
    sort(NewList,Goal,SortedList),
    best_first(SortedList,Goal,FinalPath).
```

Метод на най-доброто спускане
(Best-first search)

```
extend([Node|Path],NewPaths) :-
  findall([NewNode,Node|Path],
          (arc(Node,NewNode,_),
          not member(NewNode,[Node|Path])),
          NewPaths).

reverse([],[]).
reverse([X|Rest],List):-
     reverse(Rest,NewRest),
     append(NewRest,[X],List).
```

Метод на най-доброто спускане
(Best-first search)

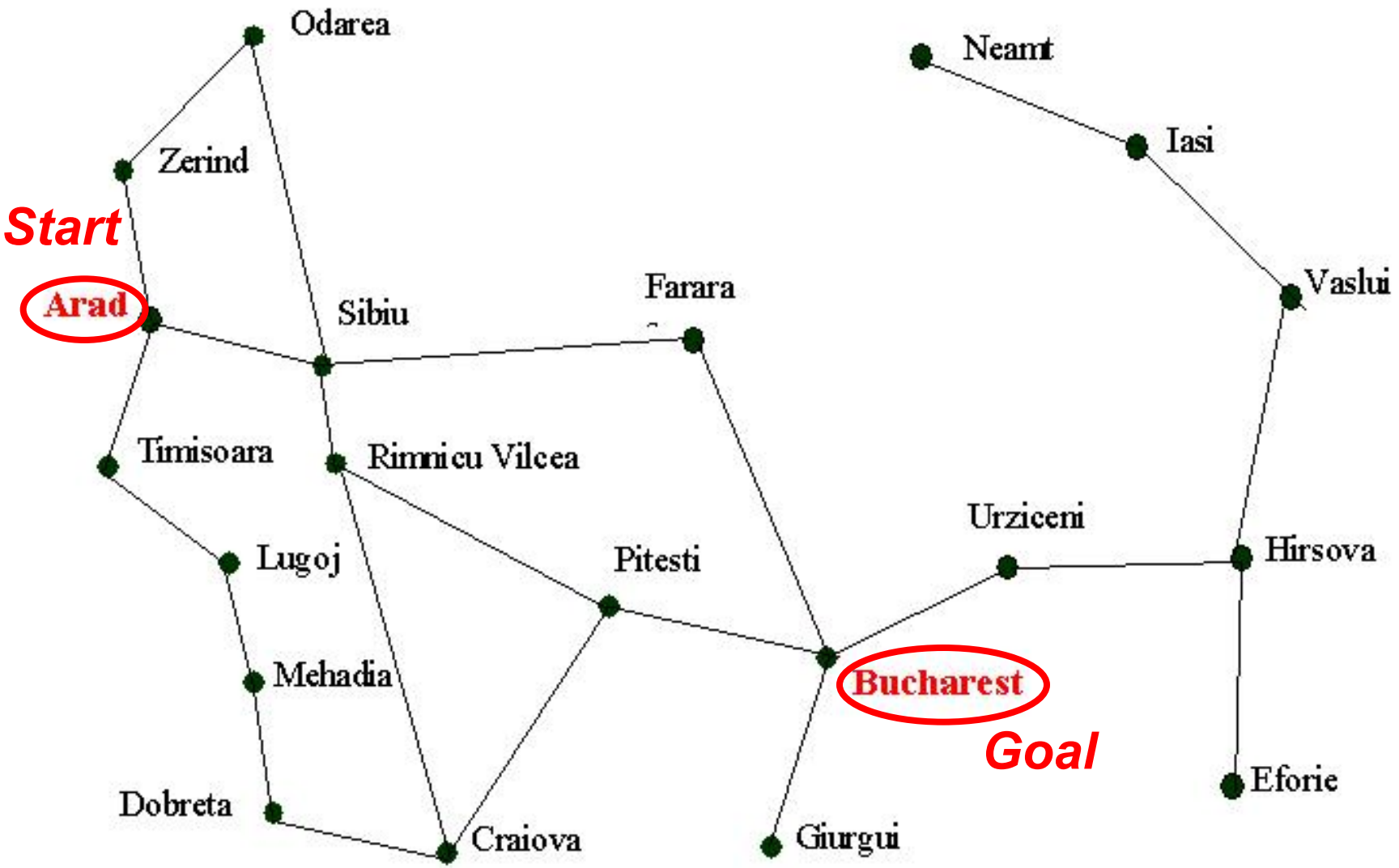
```
sort([],Goal,[]).
sort(List,Goal,[MinPath|SortedList]):-
      min(List,Goal,MinPath,NewList),
      sort(NewList,Goal, SortedList).

min([Path1|List],Goal,Path2,[Path1|NewList]):-
      min(List,Goal,Path2,NewList),
      h(Path1,Goal,H1),
      h(Path2,Goal,H2),
      H2<H1,!.
min([Path|List],Goal,Path,List).
```

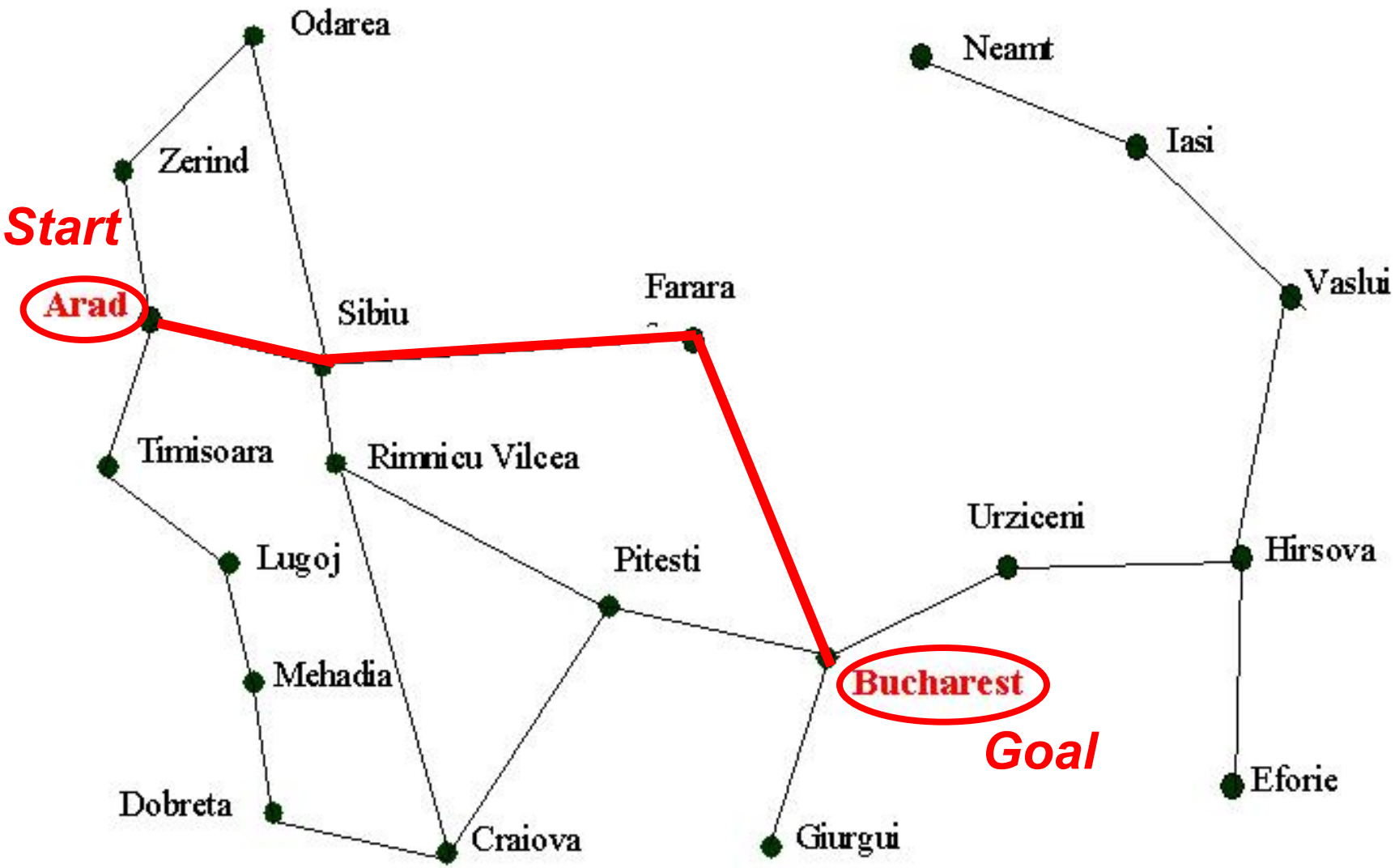
Пример

```
arc('Arad','Zerind',75).
arc('Arad','Sibiu',140).
arc('Arad','Timisoara',118).
arc('Bucharest','Fagaras',211).
arc('Bucharest','Pitesti',101).
arc('Bucharest','Giurgiu',90).
arc('Bucharest','Urziceni',85).
arc('Craiova','Dobreta',120).
arc('Craiova','Rimnicu',146).
arc('Craiova','Pitesti',138).
arc('Dobreta','Mehadia',75).
arc('Dobreta','Craiova',120).
arc('Eforie','Hirsova',86).
arc('Fagaras','Sibiu',99).
arc('Fagaras','Bucharest',211).
arc('Giurgiu','Bucharest',90).
arc('Hirsova','Urziceni',98).
arc('Hirsova','Eforie',86).
arc('Iasi','Neamt',87).
arc('Iasi','Vaslui',92).
arc('Lugoj','Timisoara',111).
arc('Lugoj','Mehadia',70).
arc('Mehadia','Lugoj',70).
arc('Mehadia','Dobreta',75).
arc('Neamt','Iasi',87).
arc('Oradea','Zerind',71).
arc('Oradea','Sibiu',151).
arc('Pitesti','Rimnicu',97).
arc('Pitesti','Craiova',138).
arc('Pitesti','Bucharest',101).
arc('Rimnicu','Sibiu',80).
arc('Rimnicu','Pitesti',97).
arc('Rimnicu','Craiova',146).
arc('Sibiu','Arad',140).
arc('Sibiu','Oradea',151).
arc('Sibiu','Fagaras',99).
arc('Sibiu','Rimnicu',80).
arc('Timisoara','Arad',118).
arc('Timisoara','Lugoj',111).
arc('Urziceni','Bucharest',85).
arc('Urziceni','Hirsova',98).
arc('Urziceni','Vaslui',142).
arc('Vaslui','Iasi',92).
arc('Vaslui','Urziceni',142).
arc('Zerind','Arad',75).
arc('Zerind','Oradea',71).
```

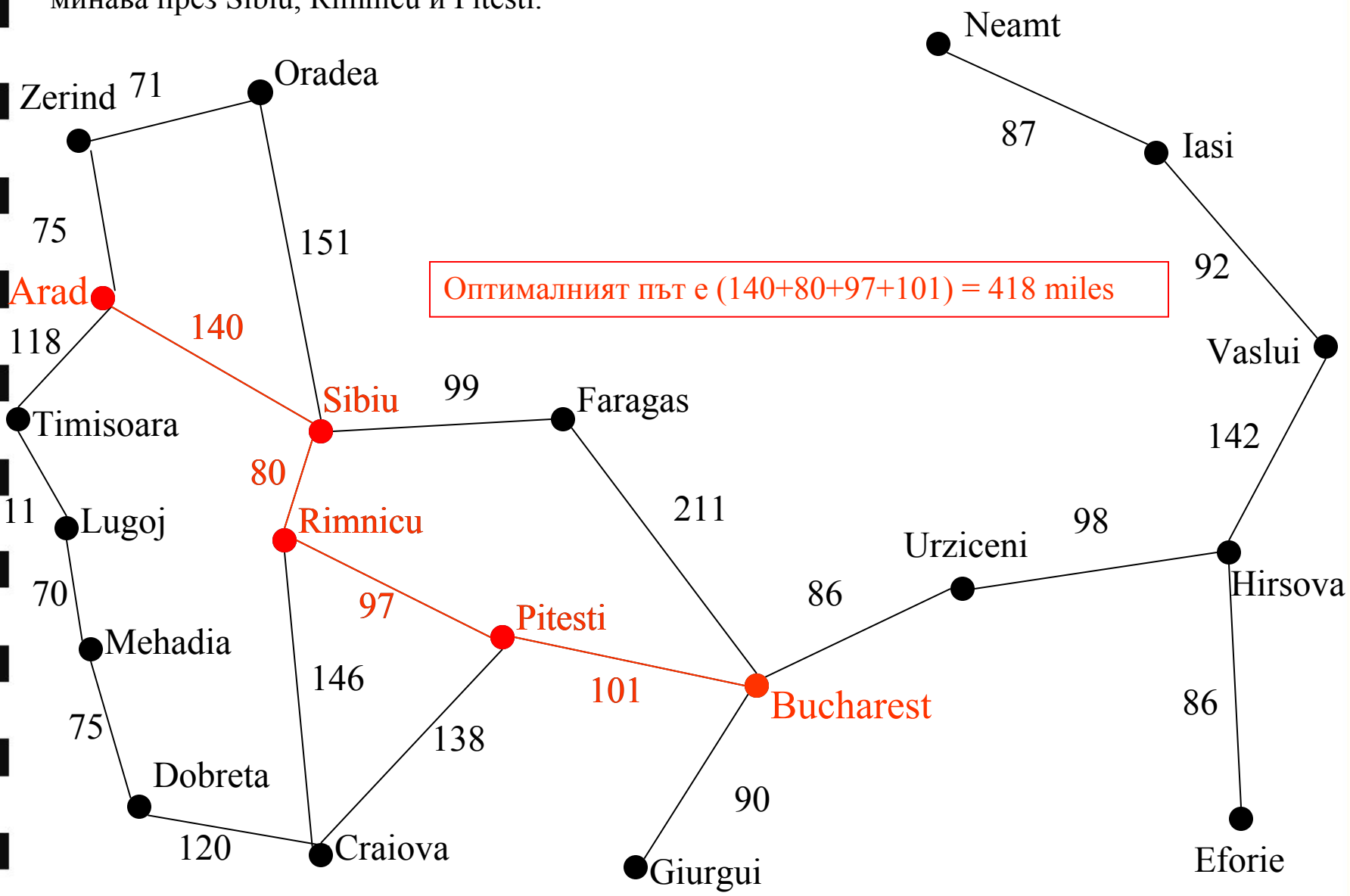
Пример

```
?- best_first([['Arad']],'Bucharest',Path).

Path =
  ['Arad','Sibiu','Fagaras','Bucharest']
```


Odarea
Neamt
Zerind
Iasi
Start
Arad
Sibiu
Farara
Vaslui
Timisoara
Rimnicu Vilcea
Urziceni
Lugoj
Pitesti
Hirsova
Mehadia
Bucharest
Goal
Eforie
Dobreta
Craiova
Giurgui


Odarea
Neamt
Zerind
Iasi
Start
Arad
Sibiu
Farara
Vaslui
Timisoara
Rimnicu Vilcea
Urziceni
Lugoj
Pitesti
Hirsova
Mehadia
Bucharest
Goal
Eforie
Dobreta
Craiova
Giurgui

Оптималният (най-краткият) път между двата града (Arad и Bucharest) минава през Sibiu, Rimnicu и Pitesti.



Метод на изкачването
(Hill Climbing)

- Съответства на търсене в лъч с размер **B=1**
- Фронтът съдържа само най-добрия текущ път
- Ако фронтът е $[p_1]$
  - пътищата p_1', p_1'',…, $p_1^{(k)}$, които разширяват **p1** се добавят към фронта и той се сортира в нарастващ ред на оценките на пътищата $[q_1, q_2, \ldots, q_k]$
  - Във фронта се оставя само най-добрия път $[q_1]$
  - на следващата стъпка се обработва пътя с най-добра текуща оценка, т.е. q_1

Метод на изкачването
(Hill Climbing) - 1

```
hill_climbing(List,Goal,FinalPath):-
    beam_search(1,List,Goal,FinalPath).
```

Метод на изкачването
(Hill Climbing) - 2

```
hill_climbing([Goal|Path],Goal,FinalPath):-
     reverse([Goal|Path],FinalPath).

hill_climbing(Path,Goal,FinalPath):-
    extend(Path,NewPaths),
   best_path(NewPaths,Goal,BestPath),
    hill_climbing(BestPath,Goal,FinalPath).
```

Метод на изкачването
(Hill Climbing) - 2

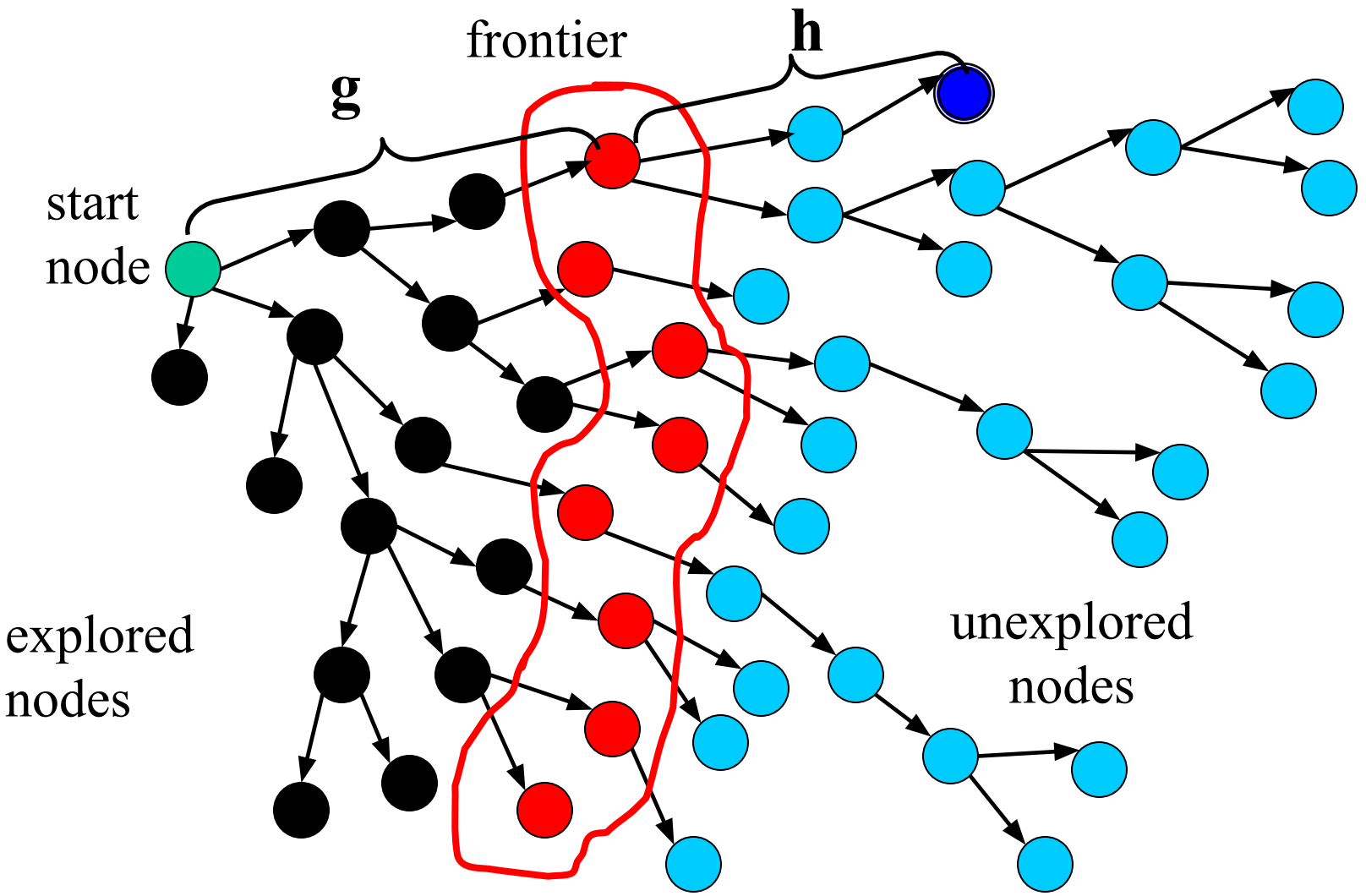
```
extend([Node|Path],NewPaths) :-
  findall([NewNode,Node|Path],
           (arc(Node,NewNode,_),
           not member(NewNode,[Node|Path])),
           NewPaths).

reverse([],[]).
reverse([X|Rest],List):-
      reverse(Rest,NewRest),
      append(NewRest,[X],List).
```

Метод на изкачването
(Hill Climbing) - 2

```
best_path([Path1|List],Goal,Path2):-
      best_path(List,Goal,Path2),
     h(Path1,Goal,H1),
     h(Path2,Goal,H2),
     H2<H1,!.
best_path([Path|List],Goal,Path).
```

A*

- Фронтът се обработва като сортиран списък в нарастващ ред на текущите евристични оценки на пътищата
- Ако фронтът е $[p_1, p_2, \ldots, p_n]$
  - избира се p_1
  - пътищата p_1', p_1'',…, $p_1^{(k)}$, които разширяват **p1** се добавят към фронта и той се сортира в нарастващ ред на оценките на пътищата $[q_1, q_2, \ldots, q_{n+k}]$
  - на следващата стъпка се обработва първо пътя с най-добра текуща оценка, т.е. q_1



f = g+h

A*

```
a_star([[Goal|Path]|_],Goal,FinalPath):-
   reverse([Goal|Path], FinalPath).

a_star([Path|List],Goal,FinalPath) :-
    extend(Path,NewPaths),
    append(List,NewPaths,NewList),
    sort(NewList,Goal,SortedList),
    a_star(SortedList,Goal,FinalPath).
```

A*

```
sort([],Goal,[]).
sort(List,Goal,[MinPath|SortedList]):-
      min(List,Goal,MinPath,NewList),
      sort(NewList,Goal, SortedList).

min([Path1|List],Goal,Path2,[Path1|NewList]):-
      min(List,Goal,Path2,NewList),
      f(Path1,Goal,F1),
      f(Path2,Goal,F2),
      F2<F1,!.
min([Path|List],Goal,Path,List).
```

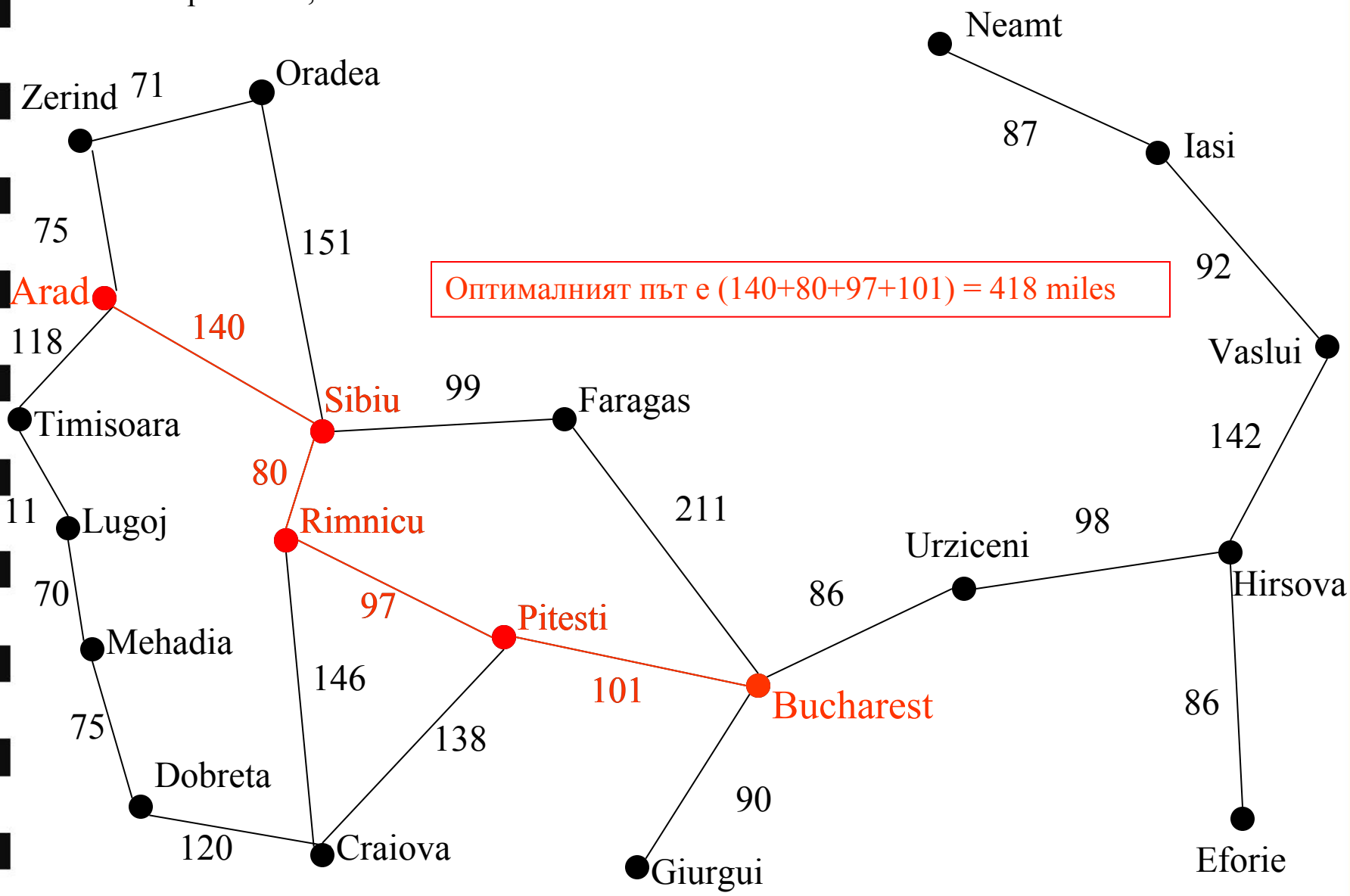
A*

```
h([Node|Path],Goal, H) :-
    stright_line_distance(Node,Goal,H).

f(Path,Goal,F) :-
    reverse_path_cost(Path,G),
    h(Path,Goal,H),
    F is G+H.

reverse_path_cost([A,B],Cost) :-
    arc(B,A,Cost).

reverse_path_cost([A,B|Path],Cost) :-
    arc(B,A,Cost1),
    reverse_path_cost([B|Path],Cost2),
    Cost is Cost1+Cost2.
```

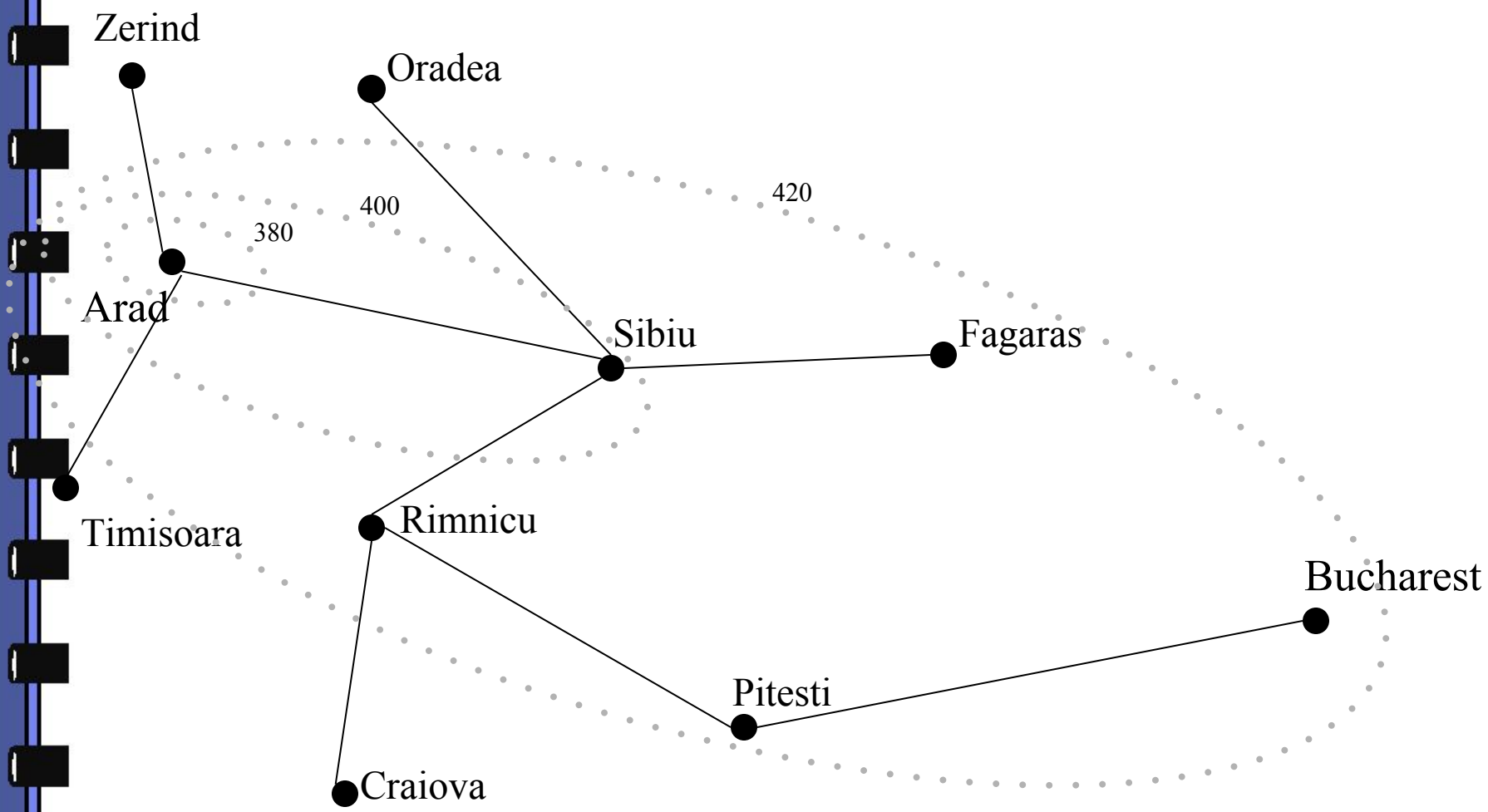
A*

```
extend([Node|Path],NewPaths)  :-
   findall([NewNode,Node|Path],
             (arc(Node,NewNode,_),
             not member(NewNode,[Node|Path])),
             NewPaths).

reverse([],[]).
reverse([X|Rest],List):-
     reverse(Rest,NewRest),
     append(NewRest,[X],List).
```

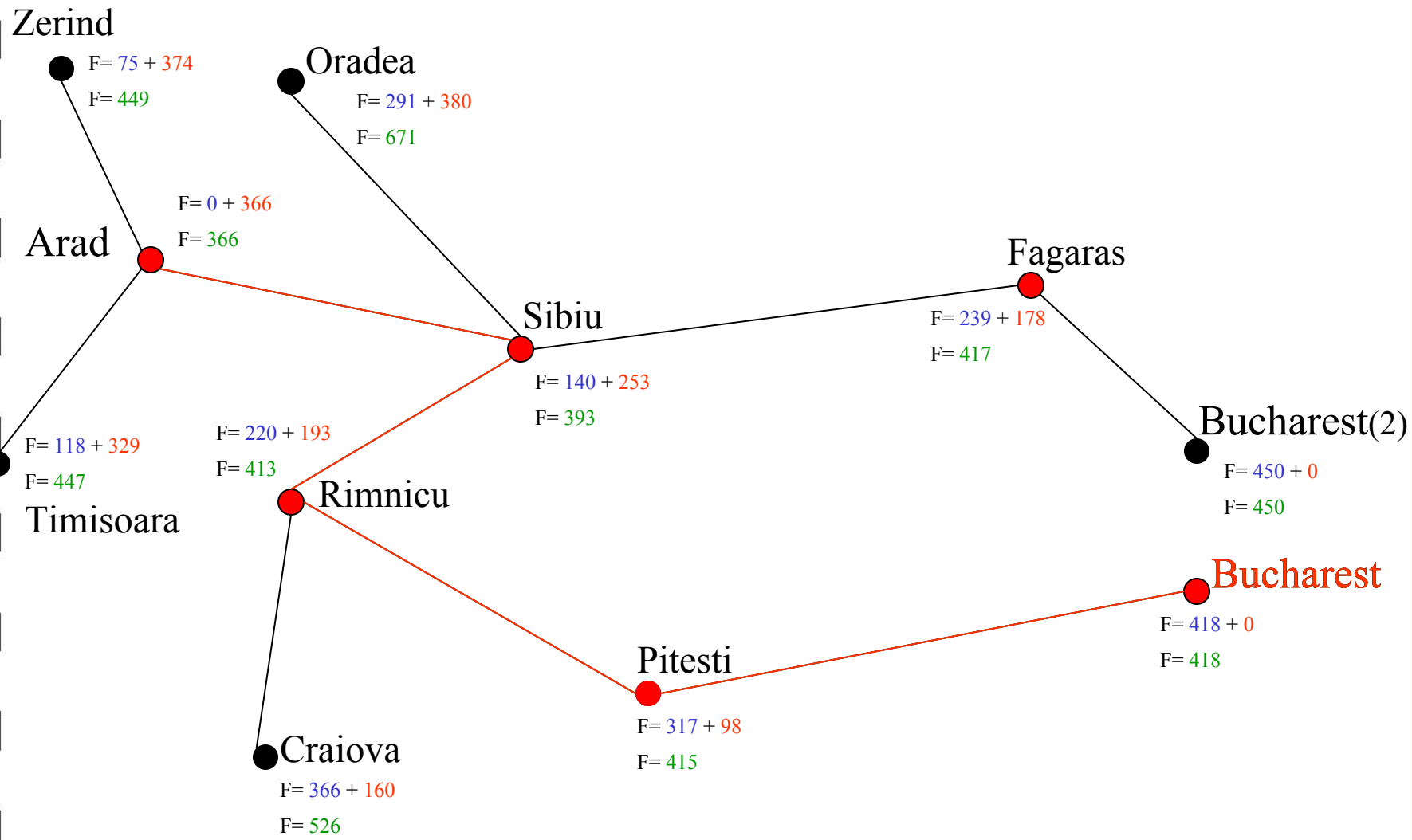
Оптималният (най-краткият) път между двата града (Arad и Bucharest) минава през Sibiu, Rimnicu и Pitesti.



Карта на Румъния, показваща областите на f = 380, f = 400 и f = 420 при начално състояние Arad. **Забележка:** Състоянията вътре в дадена област имат оценка f по-малка отколкото на стойността на тази област.

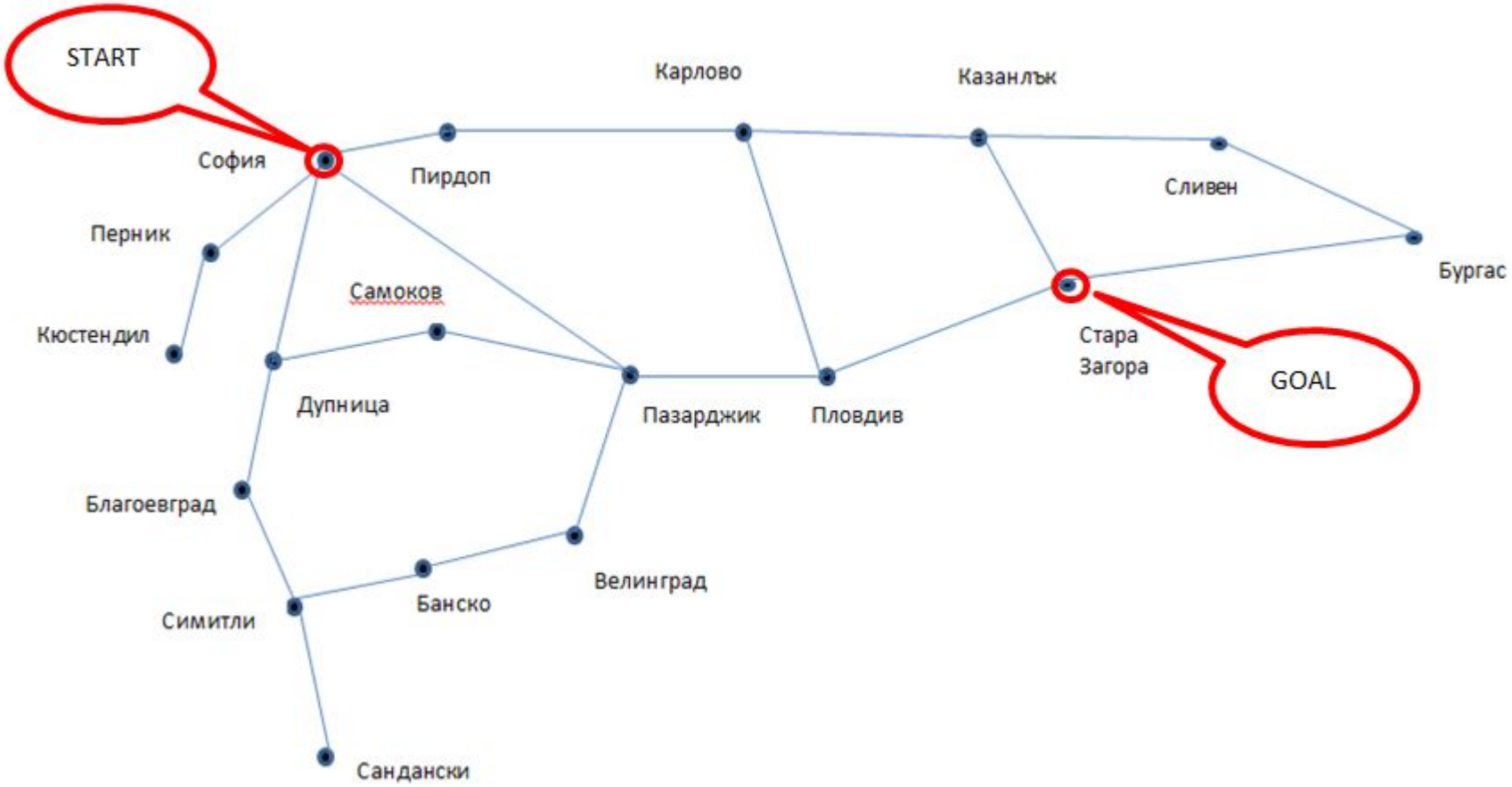


Началното състояние е Arad. Цената за достигане до Arad *от* Arad (стойността на ***g***) е 0 miles. Разстоянието по права линия от Arad до Bucharest (стойността на ***h***) е 366 miles. Общата оценка е (***f*** = ***g*** + ***h***) 366 miles.



ПРИМЕР

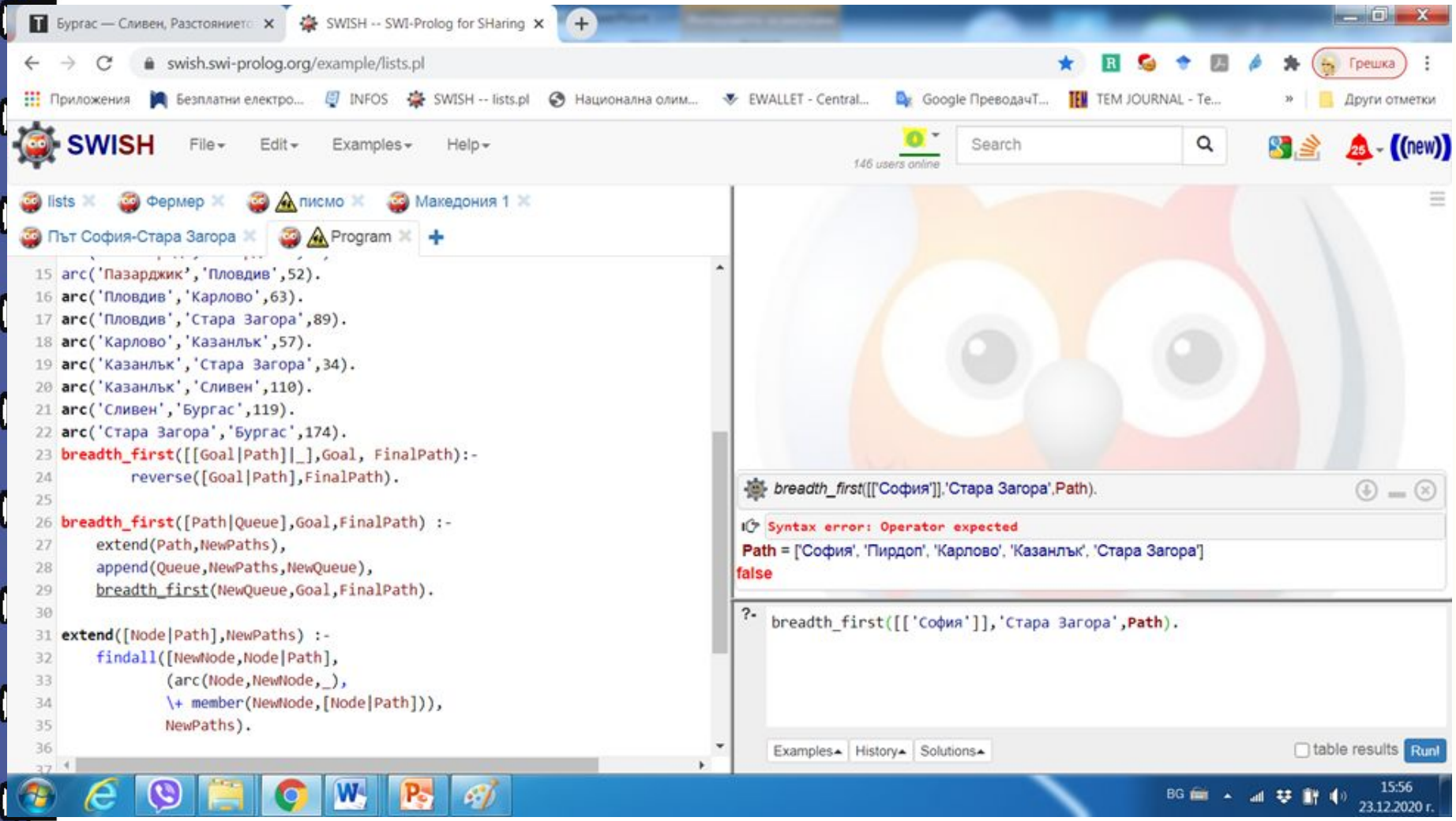
Пример:



```
arc('София','Перник',26).
arc('София','Пазарджик',126).
arc('София','Дупница',71).
arc('София','Пирдоп',89).
arc('Пирдоп','Карлово',58).
arc('Перник','Кюстендил',61).
arc('Дупница','Благоевград',67).
arc('Дупница','Самоков',40).
arc('Самоков','Пазарджик',89).
arc('Благоевград','Симитли',45).
arc('Симитли','Сандански',53).
arc('Симитли','Банско',58).
arc('Банско','Велинград',89).
arc('Велинград','Пазарджик',49).
arc('Пазарджик','Пловдив',52).
arc('Пловдив','Карлово',63).
arc('Пловдив','Стара Загора',89).
arc('Карлово','Казанлък',57).
arc('Казанлък','Стара Загора',34).
arc('Казанлък','Сливен',110).
arc('Сливен','Бургас',119).
```

?- depth_first([['София']],'Стара Загора',Path).

Path = ['София', 'Пирдоп', 'Карлово', 'Казанлък', 'Стара Загора'] ;


Бургас — Сливен, Разстоянието
SWISH -- SWI-Prolog for SHaring
swish.swi-prolog.org/example/lists.pl
SWISH
File
Edit
Examples
Help
Search
146 users online
lists
Фермер
писмо
Македония 1
Път София-Стара Загора
Program
arc('Пазарджик','Пловдив',52).
arc('Пловдив','Карлово',63).
arc('Пловдив','Стара Загора',89).
arc('Карлово','Казанлък',57).
arc('Казанлък','Стара Загора',34).
arc('Казанлък','Сливен',110).
arc('Сливен','Бургас',119).
arc('Стара Загора','Бургас',174).
breadth_first([[Goal|Path]|_],Goal, FinalPath):-
reverse([Goal|Path],FinalPath).
breadth_first([Path|Queue],Goal,FinalPath) :-
extend(Path,NewPaths),
append(Queue,NewPaths,NewQueue),
breadth_first(NewQueue,Goal,FinalPath).
extend([Node|Path],NewPaths) :-
findall([NewNode,Node|Path],
(arc(Node,NewNode,_),
\+ member(NewNode,[Node|Path])),
NewPaths).
breadth_first(['София']],'Стара Загора',Path).
Syntax error: Operator expected
Path = ['София', 'Пирдоп', 'Карлово', 'Казанлък', 'Стара Загора']
false
?-
breadth_first([['София']],'Стара Загора',Path).
Examples
History
Solutions
table results
Run!
BG
15:56
23.12.2020 г.