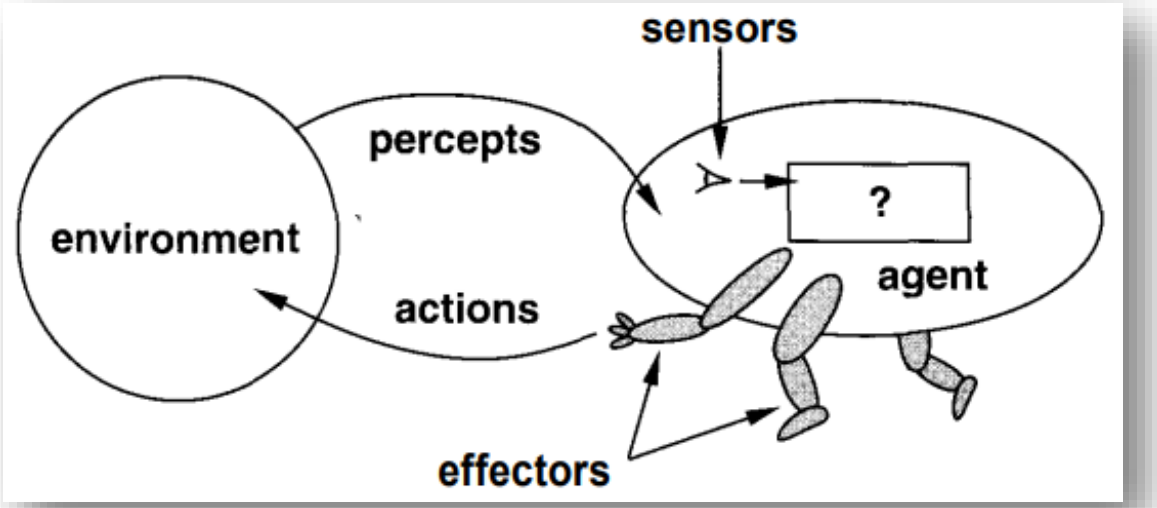
هوش مصنوعی

(جستجو-بخش اول)

صادق اسکندری - دانشکده علوم ریاضی، گروه علوم کامپیوتر

eskandari@guilan.ac.ir

یادآوری ...



هر چیزی که در یک محیط قرار گرفته و با استفاده از سنسورهای خود محیط را درک کرده و با استفاده از افکتورهای خود بر روی محیط عمل انجام می دهد.

مثال:

عامل	محیط	سنسورها	ادراکات	افکتورها	اعمال
راننده تاکسی	ترافیک	چشم – گوش – ..... سرعت سنج – آمپر بنزین	سرعت – مسافرین – وضعیت ماشین – ترافیک	دنده – پدال گاز – بوق – زبان – دست – ...	بوق زدن – سوار کردن مسافر – گاز دادن – ترمز کردن – ....
شطرنج باز	بازی شطرنج	چشم و گوش	حرکات حریف – موقعیت مهره ها – ساعت –	دست – زبان	حرکت دادن مهره ها – کیش دادن – ...

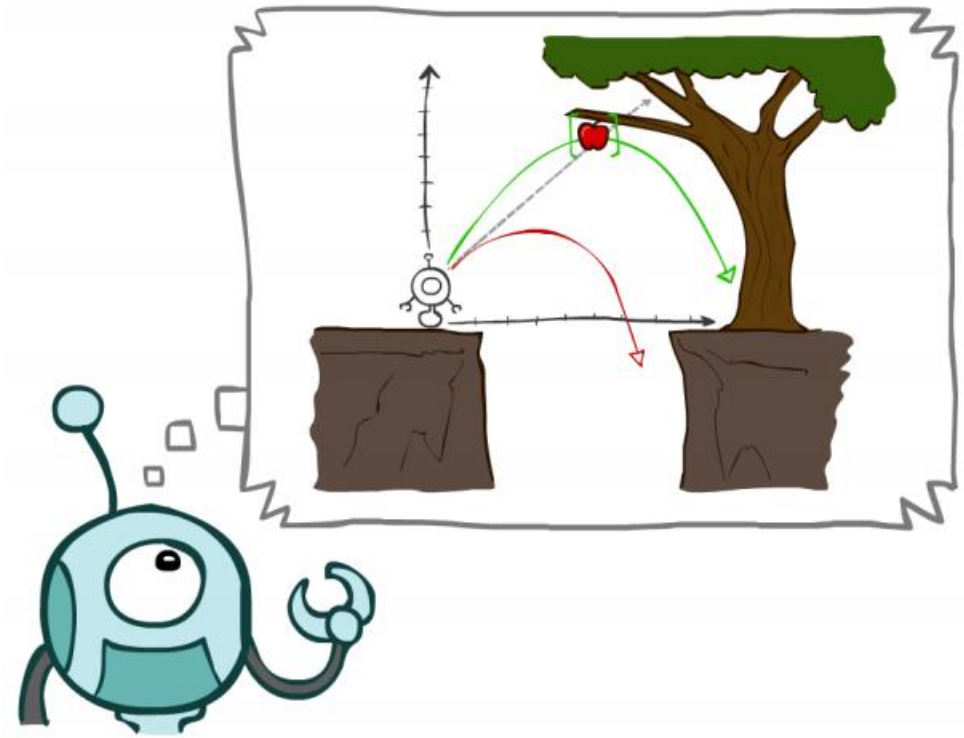
یادآوری ...

ساختار کلی یک عامل:

$$\text{Agent} = \text{Architecture} + \text{Program}$$

در این درس فرض بر این است که یک معماری در اختیار ما قرار داده شده است (سنسورها و افکتورها مشخص هستند) و ما باید برنامه آن را مشخص کنیم.

مباحث امروز



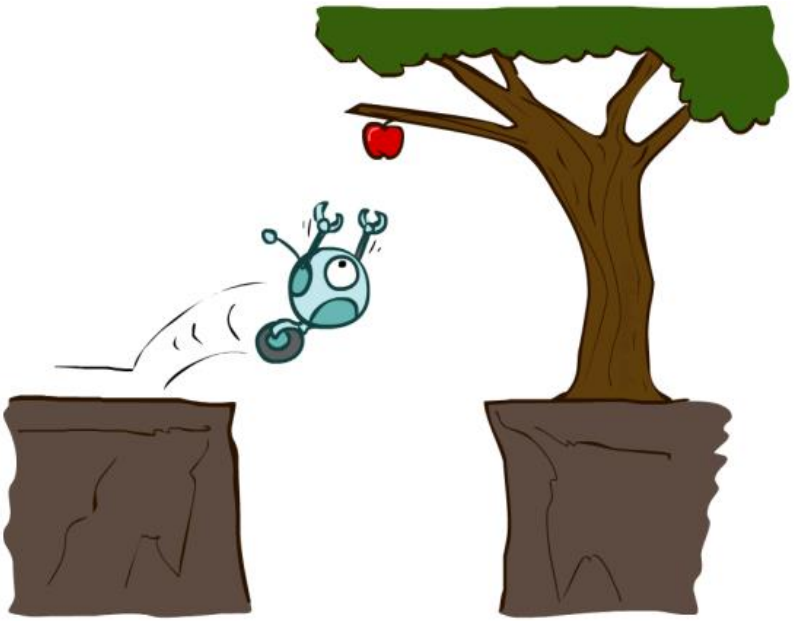
عامل های برنامه ریز

مسائل جستجو

روش های جستجوی کورکورانه

- جستجوی عمق اول (Depth First Search)
- جستجوی سطح اول (Breadth First Search)
- جستجوی هزینه یکنواخت (Uniform Cost Search)
- جستجوی عمیق سازی تکراری (Iterative Deeping Search)

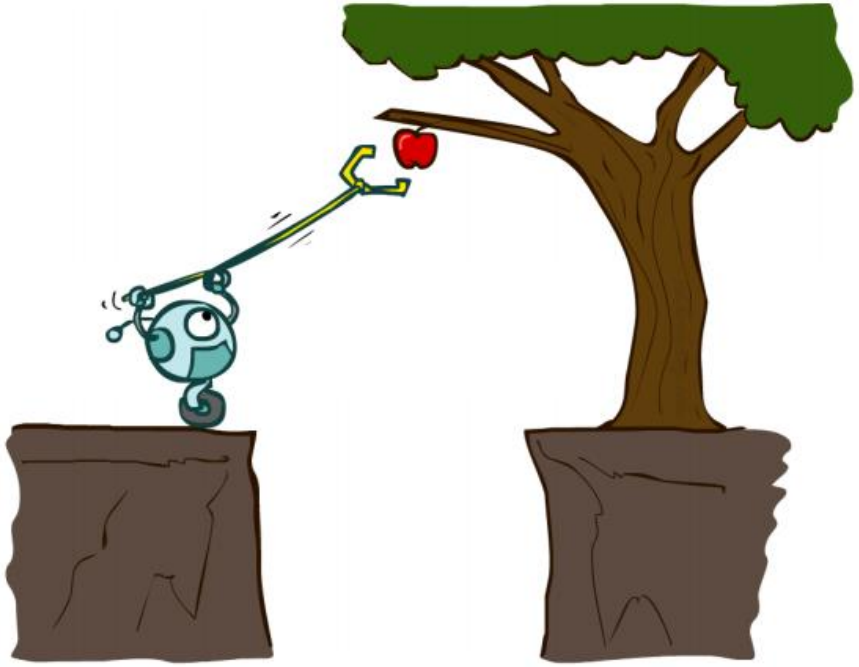
عامل های رفلکس



تصمیمات در هر لحظه را فقط بر اساس ادراکات همان لحظه می گیرند.

نتایج و پیامدهای اعمال خود را قبل از انجام عمل نمی سنجند.

عامل های برنامه ریز



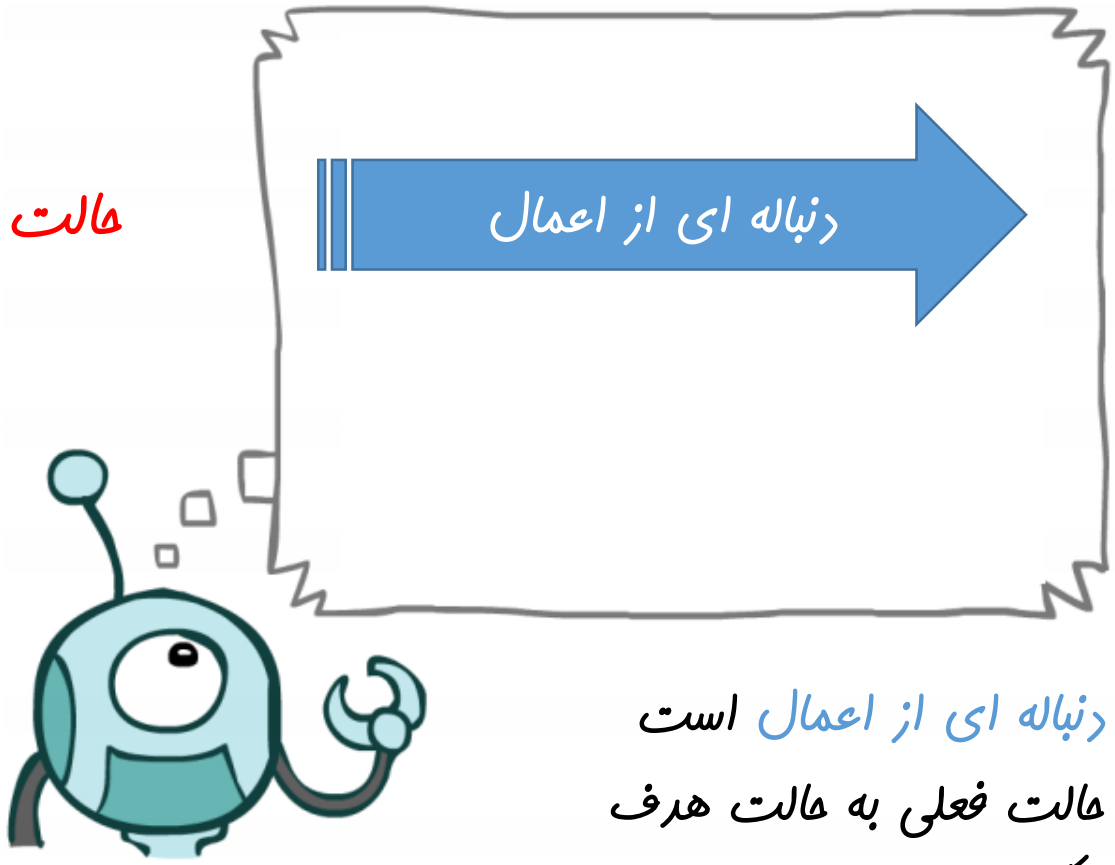
همواره قبل از انجام عمل، نتیجه آن را می سنجند.

همواره یک مدل از نحوه تغییر جهان در پاسخ با اعمال خود دارد.

هر عملی را برای رسیدن به یک هدف انجام می دهد.

عامل های برنامه ریز

در مسائل برنامه ریزی، هدف تغییر وضعیت جهان از حالت فعلی به یک حالت هدف است.



وظیفه یک عامل برنامه ریز، یافتن دنباله ای از اعمال است که با انجام آنها، وضعیت جهان از حالت فعلی به حالت هدف تبدیل می شود. به این عمل جستجو گفته می شود.

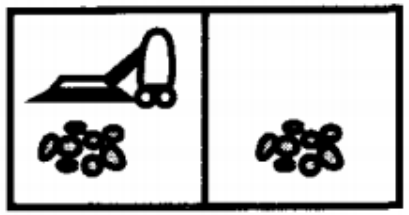
عامل های برنامه ریز

در مسائل برنامه ریزی، هدف تغییر وضعیت جهان از حالت فعلی به یک حالت هدف است.

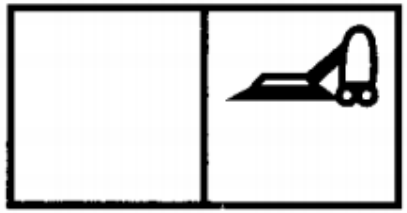
حالت فعلی جهان

دنباله ای از اعمال

حالت هدف



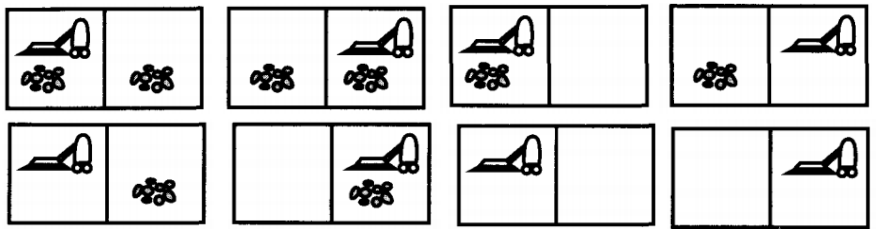
$S \rightarrow R \rightarrow S$



مسائل جستجو

یک مسئله جستجو شامل موارد زیر است:

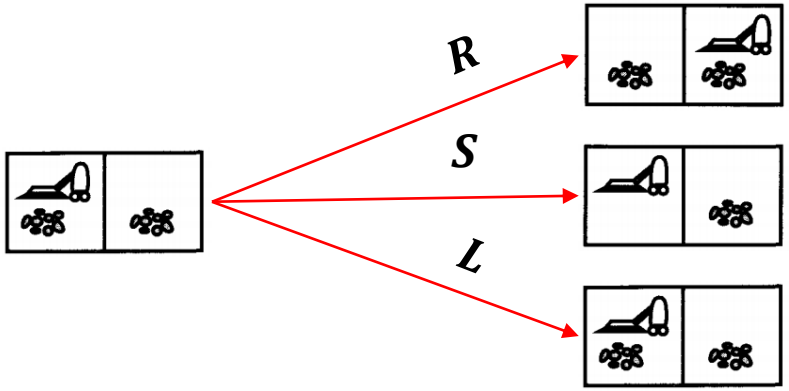
۱- یک فضای حالت (State Space)



۲- مجموعه اعمال (Actions)

$$\{S, L, R\}$$

۳- یک تابع بعدی (Successor Function): تابعی که با دریافت یک حالت و یک عمل، نتیجه انجام آن عمل در آن حالت را مشخص می کند.



مسائل جستجو

یک مسئله جستجو شامل موارد زیر است:

۴- یک تابع هزینه مسیر: تابعی که یک مسیر (دنباله ای از اعمال) را به عنوان ورودی دریافت کرده و هزینه آن را برمی گرداند. معمولا هزینه مسیر برابر با مجموع هزینه های گامهای مسیر می باشد.

۵- یک حالت اولیه (Initial State)

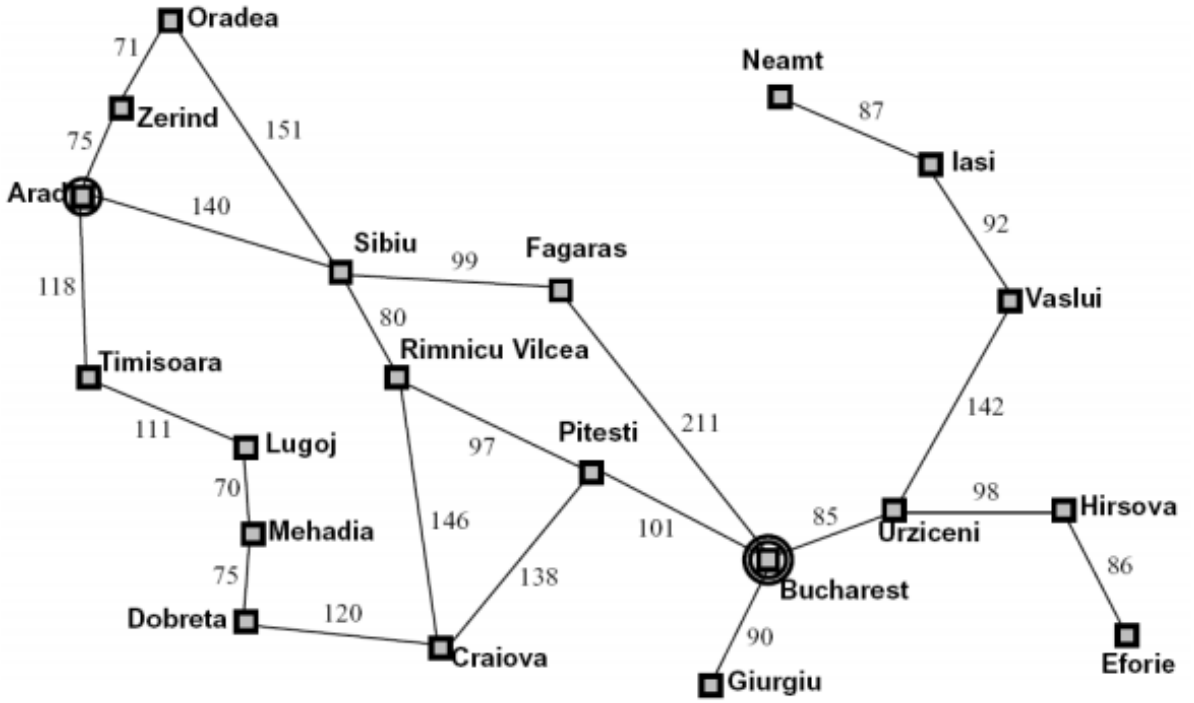
۶- یک تابع تست هدف (Goal Test Function)

$$G: State\ Space \rightarrow \{True, False\}$$

یک جواب (Solution) برای مسئله جستجو دنباله ای از اعمال (مسیر) است که حالت اولیه را به حالت هدف تبدیل می کند.

مسائل جستجو

مثال: مسافرت در رومانی



فضای حالت: بودن در هر یک از شهرها

تابع بعدی و اعمال: رفتن به شهرهای همسایه

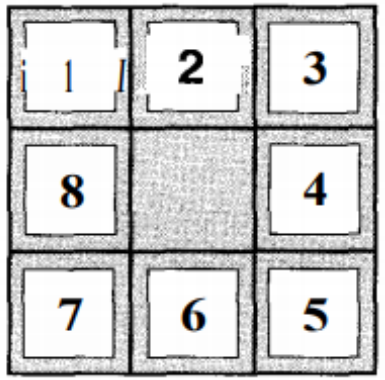
تابع هزینه: مجموع مسافت میان شهرها در مسیر

حالت اولیه: Arad

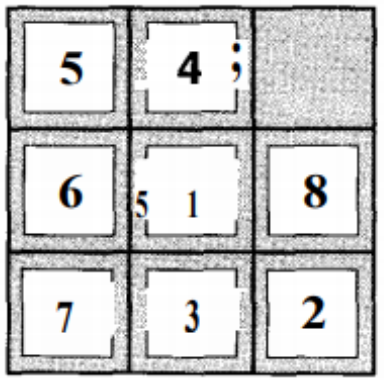
تابع تست هدف: آیا شهر (حالت) فعلی Bucharest است؟

مسائل جستجو

مثال: مسأله ۸-پازل (8-Puzzle)



حالت هدف



حالت شروع

فضای حالت: هر ترتیبی از خانه ها

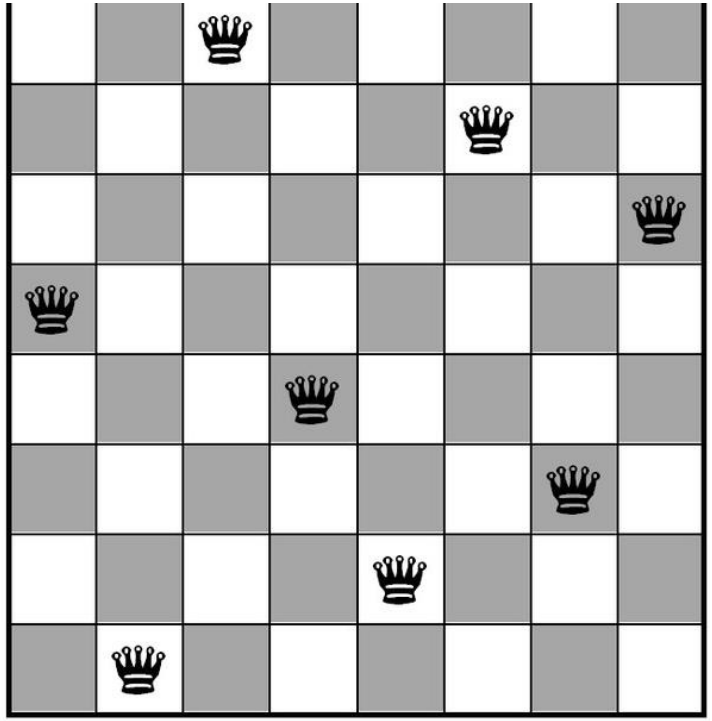
تابع بعدی و اعمال: حرکت خانه خالی به سمت
{بالا، پایین، چپ، راست}

تابع هزینه: هزینه هر گام برابر یک واحد و هزینه مسیر برابر با تعداد گام ها

حالت اولیه: هر حالتی از فضای حالت

تابع تست هدف: آیا خانه ها مرتب هستند؟

مسائل جستجو



مثال: مسأله ۸ وزیر

فضای حالت: تمامی چینش های ۸ وزیر در صفحه شطرنج

تابع بعدی و اعمال: جابجا کردن هر یک از وزیرها در خانه های شطرنج

تابع هزینه: هزینه هر گام (جابجا کردن) برابر یک واحد و
هزینه مسیر برابر با تعداد گام ها

حالت اولیه: یک چینش تصادفی

تابع تست هدف: آیا تمامی وزیرها در موقعیت امن هستند؟

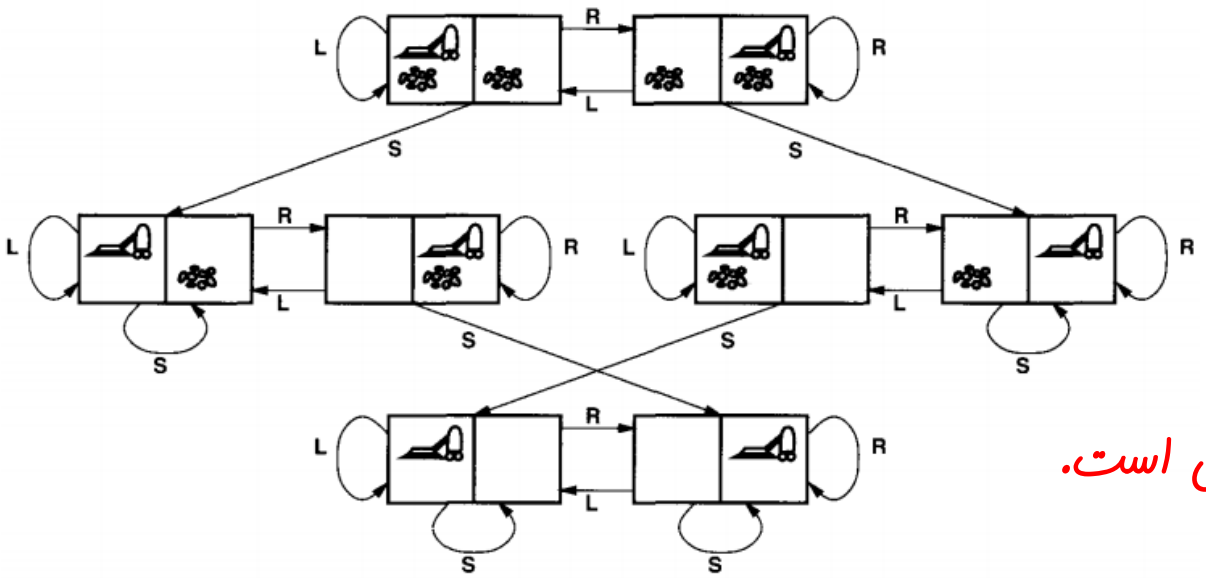
گراف فضای حالت

گراف فضای حالت (State Space Graph) یا گراف جستجو یک نمایش ریاضی برای مسئله جستجو است.

گره ها نمایش دهنده حالت های جهان هستند.

یالها نشان دهنده تابع بعدی (نتیجه اعمال) هستند.

در این گراف هر حالت فقط یکبار ظاهر می شود.



برای اغلب مسائل، تشکیل این گراف امکان پذیر نیست زیرا اندازه آن بسیار بزرگ خواهد بود. ولی برای تحلیل، ایده مناسبی است.

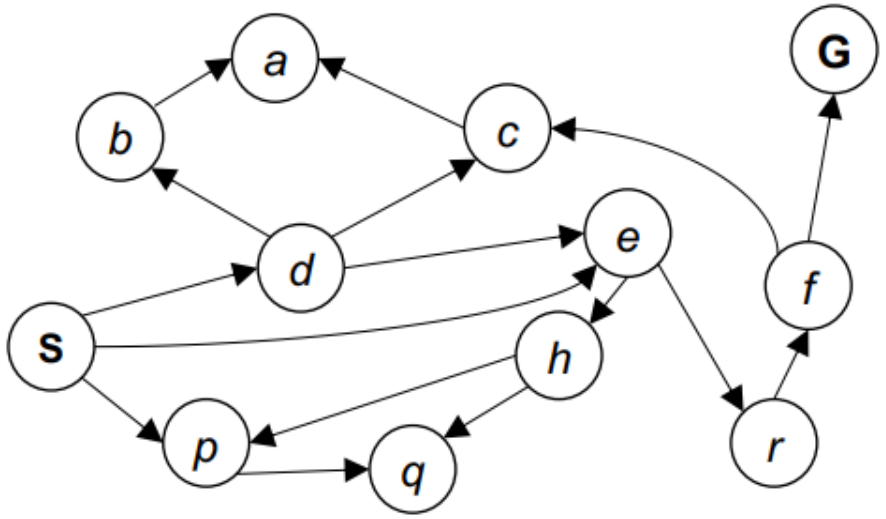
گراف فضای حالت

گراف فضای حالت (State Space Graph) یا گراف جستجو یک نمایش ریاضی برای مسئله جستجو است.

گره ها نمایش دهنده حالت های جهان هستند.

یالها نشان دهنده تابع بعدی (نتیجه اعمال) هستند.

در این گراف هر حالت فقط یکبار ظاهر می شود.



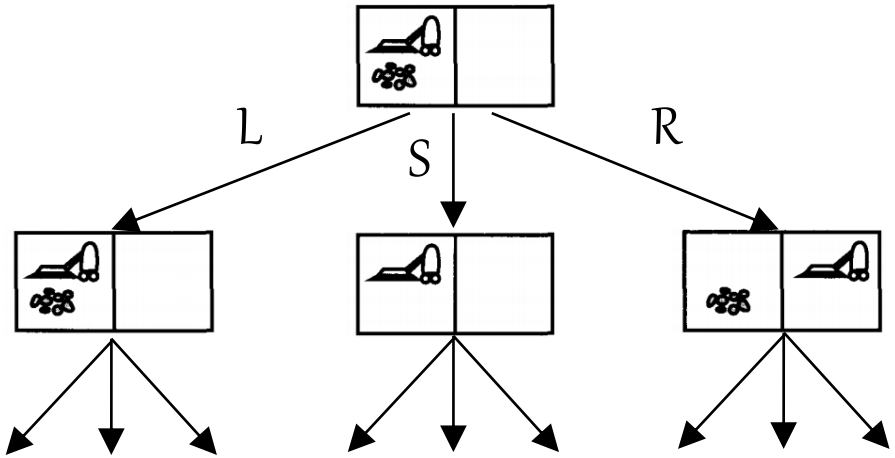
برای اغلب مسائل، تشکیل این گراف امکان پذیر نیست زیرا اندازه آن بسیار بزرگ خواهد بود. ولی برای تحلیل، ایده مناسبی است.

درخت جستجو

یک درخت جستجو یک درخت اگر-آنگاه برای برنامه ها و نتایج آنها است.

حالت اولیه ریشه درخت است

فرزندان معادل با حالت های بعدی هستند

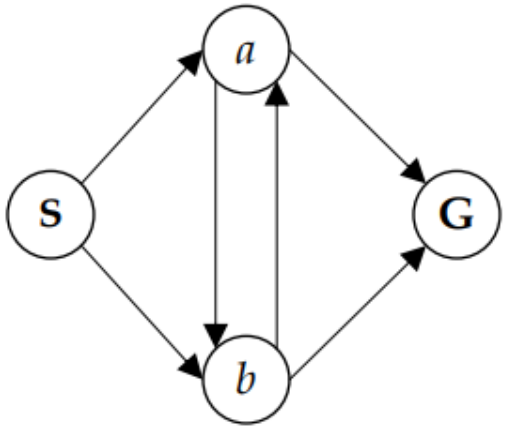


حالت فعلی

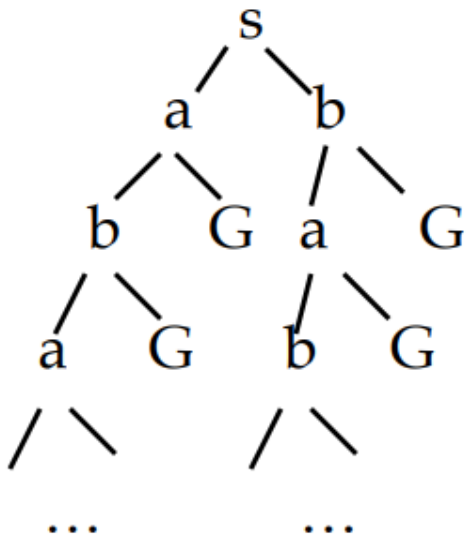
آینده های ممکن

درخت جستجو در مقابل گراف فضای حالت

یک گراف فضای حالت
با چهار حالت



درخت جستجو



مهم: در یک درخت جستجو، ممکن است ساختارهای تکراری فراوانی دیده شود. بنابراین، اندازه درخت جستجو می تواند بی نهایت باشد.

پیاده سازی درخت جستجو

نمایش گرههای درخت: برای نمایش گرهها از ساختمان داده ای با پنج عنصر زیر استفاده می کنیم:

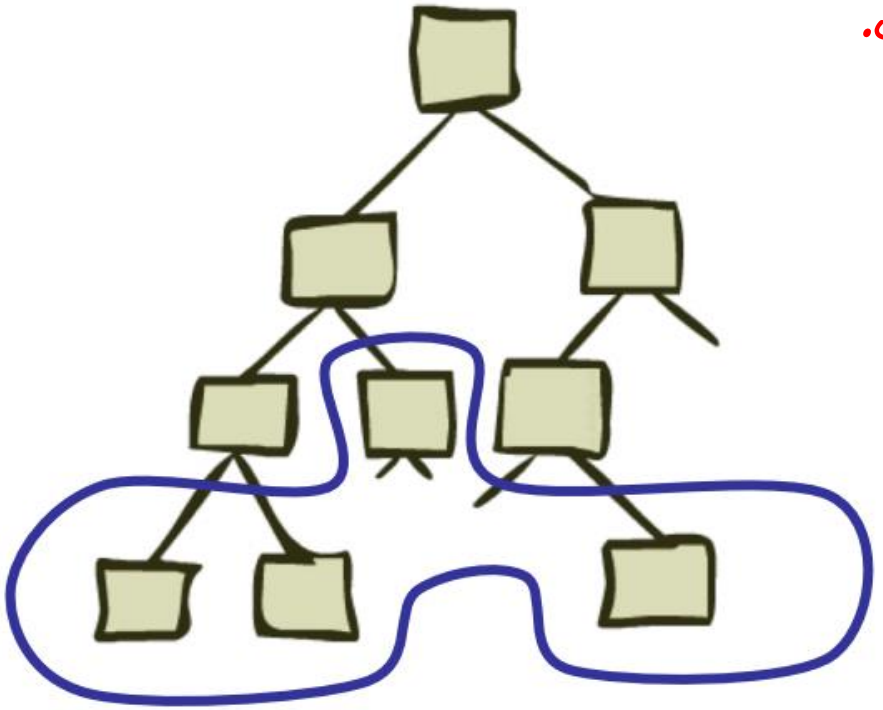
State: حالتی که این گره نمایش دهنده آن است

Parent Node: گرهی در درخت جستجو که این گره از آن ایجاد شده است.

Operator: عملی که باعث به وجود آمدن این گره از والد شده است.

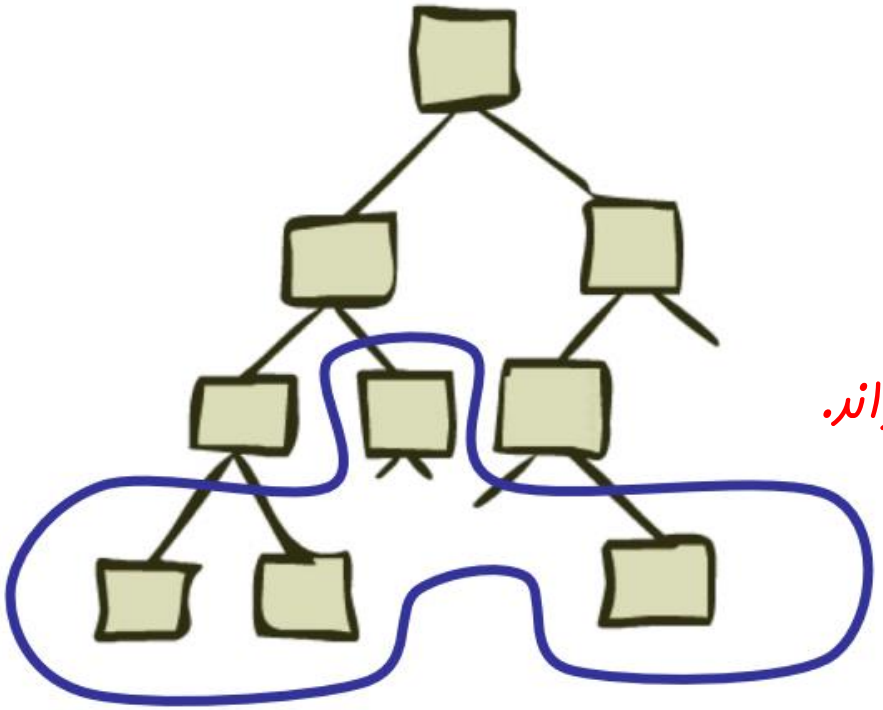
Depth: تعداد گرههای مسیر از ریشه تا این گره

Path Cost: هزینه مسیر از ریشه تا این گره



پیاده سازی درخت جستجو

علاوه براین، نیاز به محلی برای نگهداری گرههایی داریم که منتظر بسط داده شدن هستند.
بدین منظور از یک صف (Queue) استفاده خواهیم کرد.



اعمال زیر بر روی این صف قابل انجام هستند:

MAKE-QUIUE(Elements): یک صف با عناصر داده شده ایجاد می کند.

EMPTY?(Queue): آیا صف خالی است؟

REMOVE-FRONT(Queue): عنصر ابتدای صف را حذف کرده و برمی گرداند.

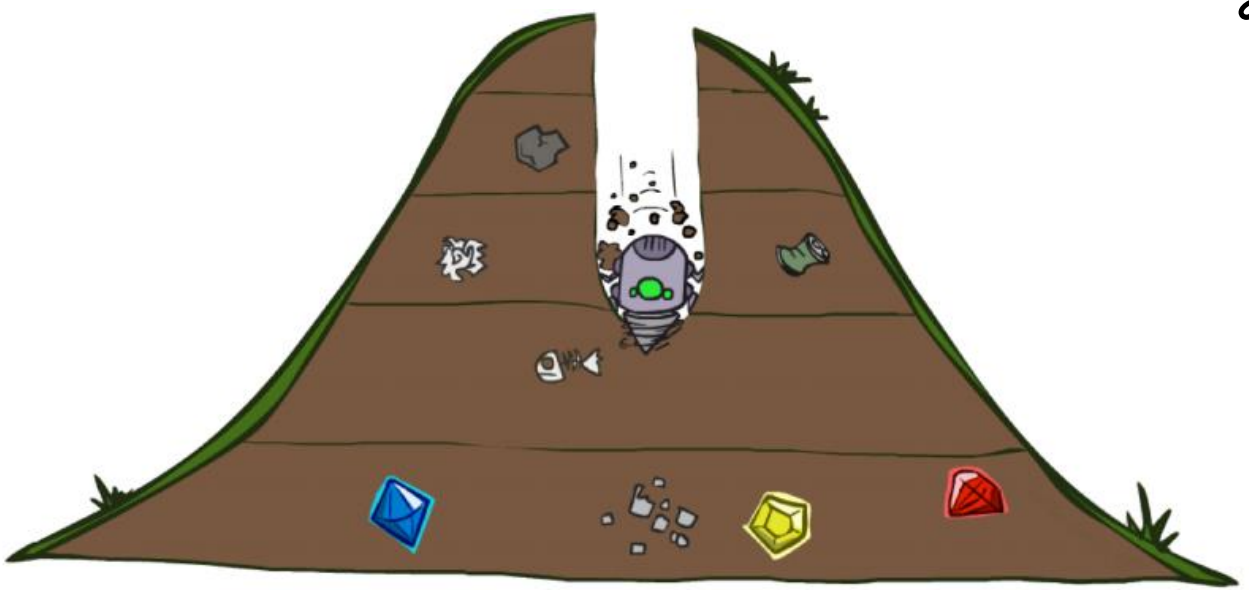
QUEUING-FN(Elements,Queue): مجموعه ای از عناصر را در صف قرار می دهد. اینکه عناصر جدید به کجای صف اضافه شوند، استراتژی های مختلف جستجو را تعریف میکند.

```
function GENERAL-SEARCH( problem, QUEUING-FN) returns a solution, or failure

    nodes ← MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem]))
    loop do
        if nodes is empty then return failure
        node ← REMOVE-FRONT(nodes)
        if GOAL-TEST[problem] applied to STATE(node) succeeds then return node
        nodes ← QUEUING-FN(nodes, EXPAND(node, OPERATORS[problem]))
    end
```

در زمان ایجاد درخت جستجو، تعدادی گره جدید تولید شده و منتظر بسط داده شدن هستند. اینکه کدامیک باید زودتر بررسی و بسط داده شوند، استراتژی جستجو نامیده می شود.

استراتژی های جستجو: عمق اول

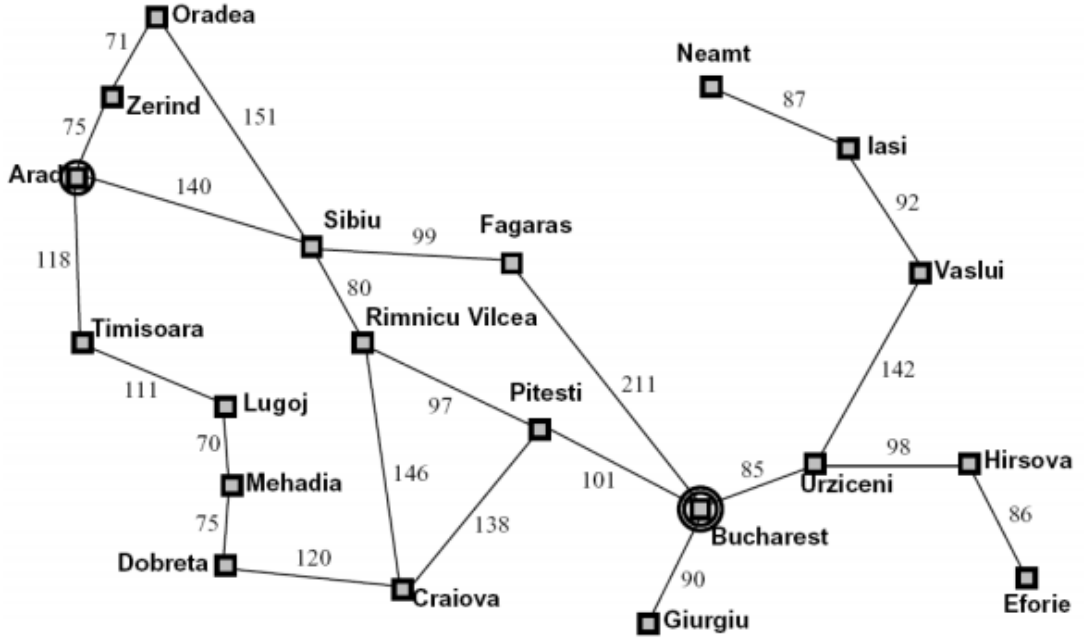


در این استراتژی، همواره عمیق ترین گره زودتر از بقیه بسط داده می شود.

در این استراتژی، فرزندان تولید شده از بسط گرهها، به ابتدای صف اضافه می شوند. به عبارت دیگر، صف همانند یک پشته عمل می کند.

استراتژی های جستجو: عمق اول

مثال: مسافرت در رومانی

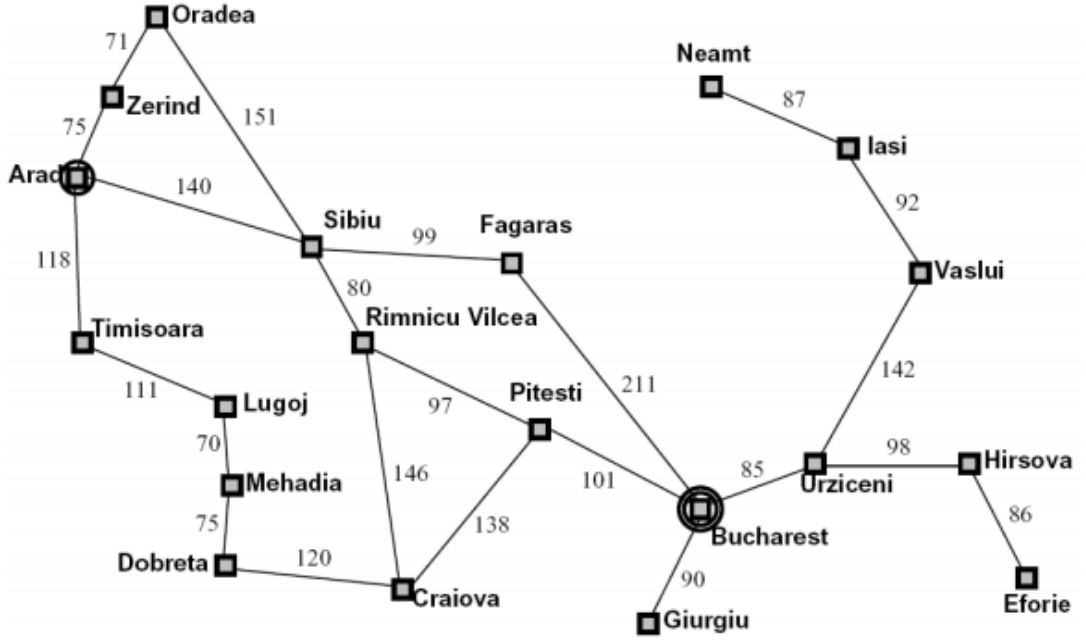


1
STATE: Arad
PARENT: null
OPERATOR: null
DEPTH: 0
PATH-COST: 0

nodes

1

nodes ← MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[*problem*]))

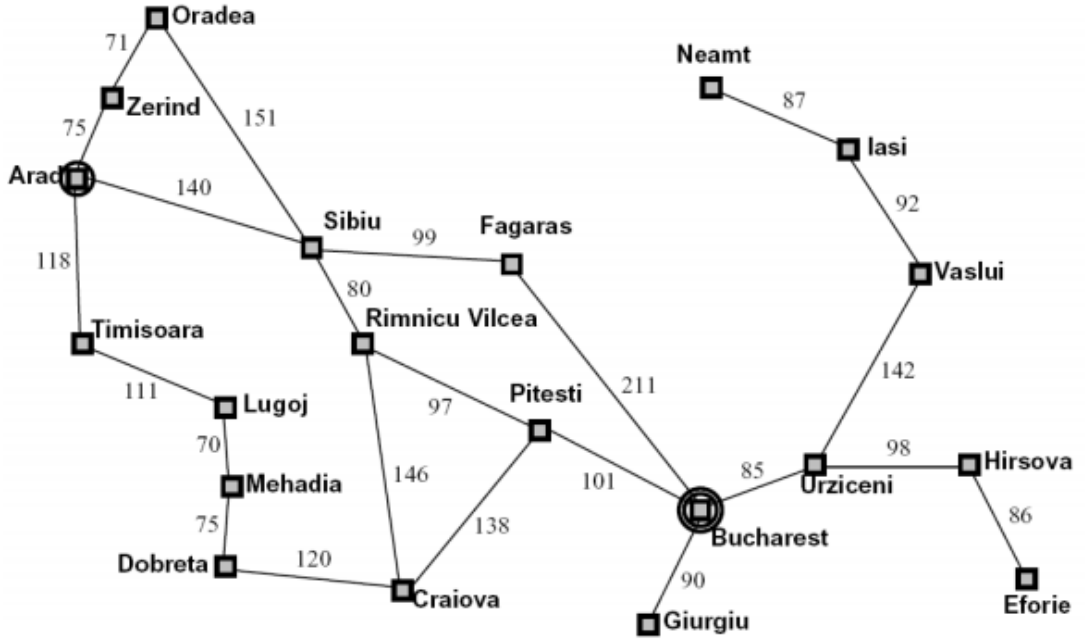


1
STATE: Arad **PARENT**: null **OPERATOR**: null **DEPTH**: 0 **PATH-COST**: 0

nodes

1

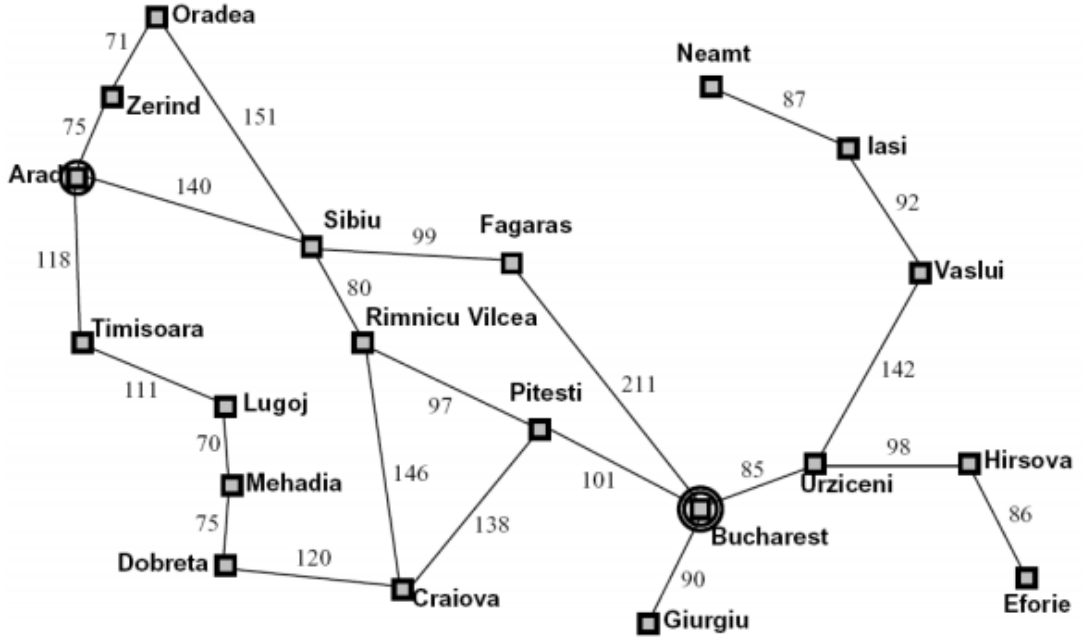
loop do
if *nodes* is empty **then return** failure
node ← REMOVE-FRONT(*nodes*)
if GOAL-TEST[*problem*] applied to STATE(*node*) succeeds **then return** *node*
nodes ← QUEUING-FN(*nodes*, EXPAND(*node*, OPERATORS[*problem*]))



nodes

1
STATE: Arad **PARENT**: null **OPERATOR**: null **DEPTH**: 0 **PATH-COST**: 0

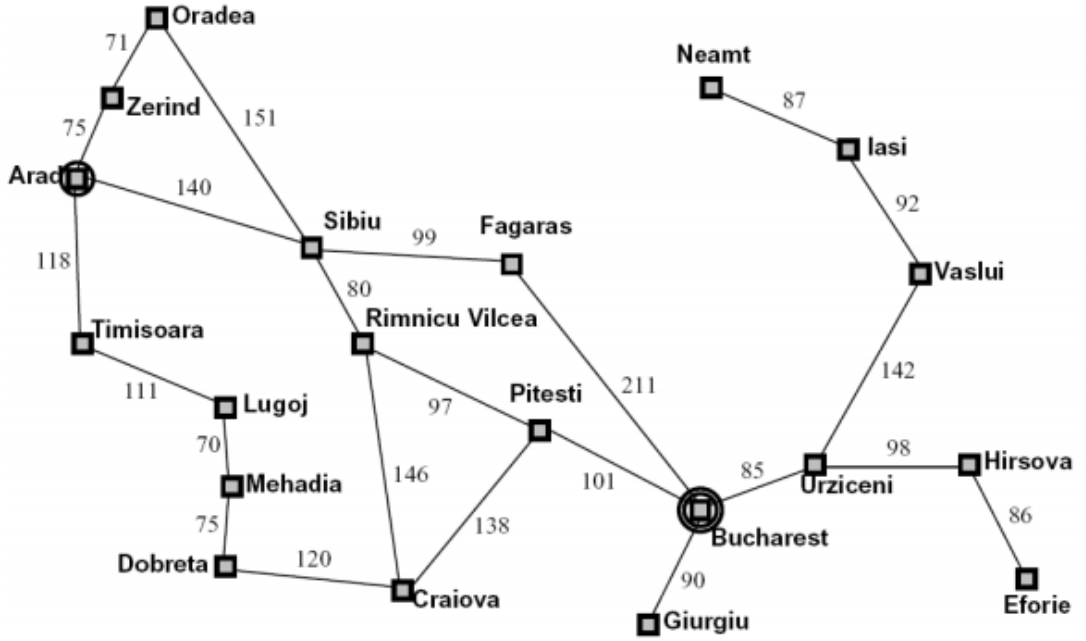
loop do
  if *nodes* is empty **then return** failure
  node ← REMOVE-FRONT(*nodes*)
  if GOAL-TEST[*problem*] applied to STATE(*node*) succeeds **then return** *node*
  nodes ← QUEUING-FN(*nodes*, EXPAND(*node*, OPERATORS[*problem*]))



1
STATE: Arad **PARENT**: null **OPERATOR**: null **DEPTH**: 0 **PATH-COST**: 0

nodes

loop do
  if *nodes* is empty **then return** failure
  node ← REMOVE-FRONT(*nodes*)
  if GOAL-TEST[*problem*] applied to STATE(*node*) succeeds **then return** *node*
  nodes ← QUEUING-FN(*nodes*, EXPAND(*node*, OPERATORS[*problem*]))



nodes

1

STATE: Arad
PARENT: null
OPERATOR: null
DEPTH: 0
PATH-COST: 0

2

STATE: Zerind
PARENT: 1
OPERATOR: go Zerind
DEPTH: 1
PATH-COST: 75

3

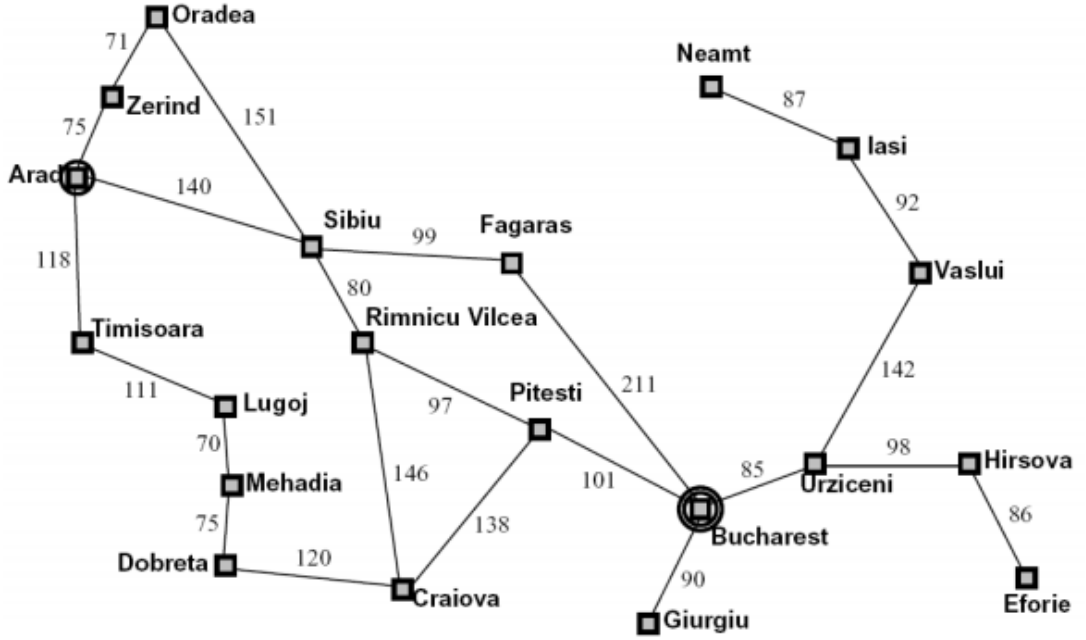
STATE: Sibiu
PARENT: 1
OPERATOR: go Sibiu
DEPTH: 1
PATH-COST: 140

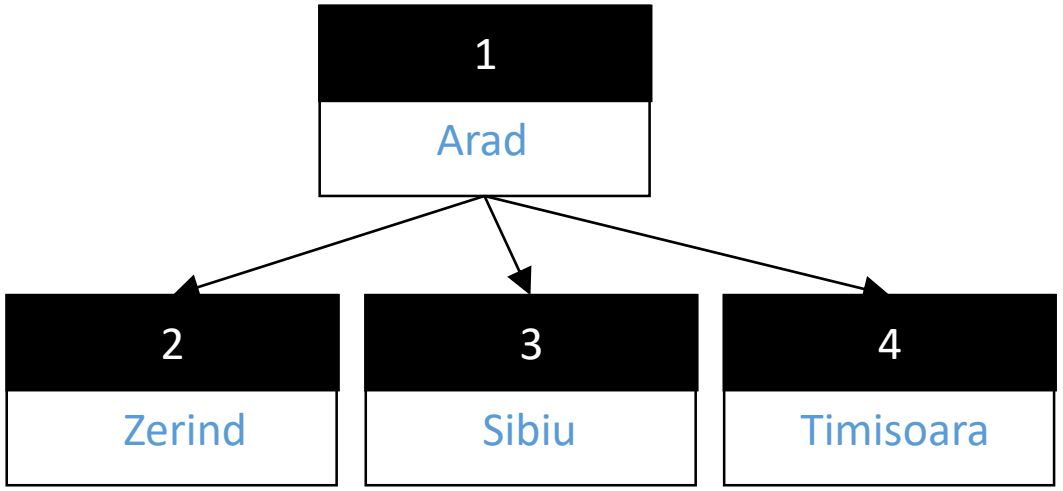
4

STATE: Timisoara
PARENT: 1
OPERATOR: go Tim…
DEPTH: 1
PATH-COST: 118

2
3
4

loop do
  if *nodes* is empty **then return** failure
  node ← REMOVE-FRONT(*nodes*)
  if GOAL-TEST[*problem*] applied to STATE(*node*) succeeds **then return** *node*
  nodes ← QUEUING-FN(*nodes*, EXPAND(*node*, OPERATORS[*problem*]))





nodes

2

3

4

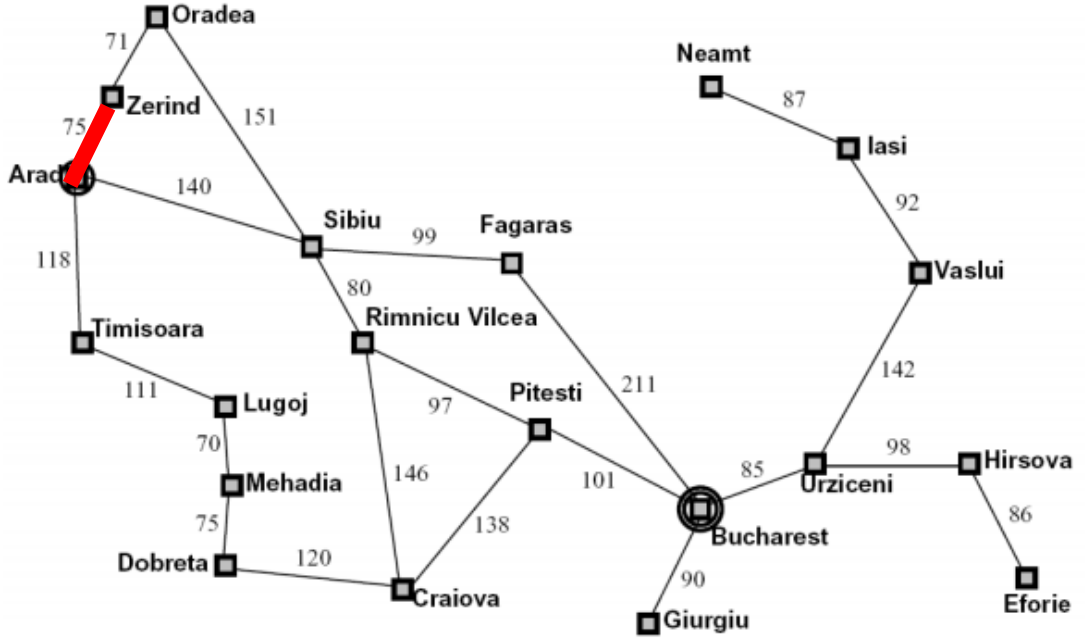
loop do

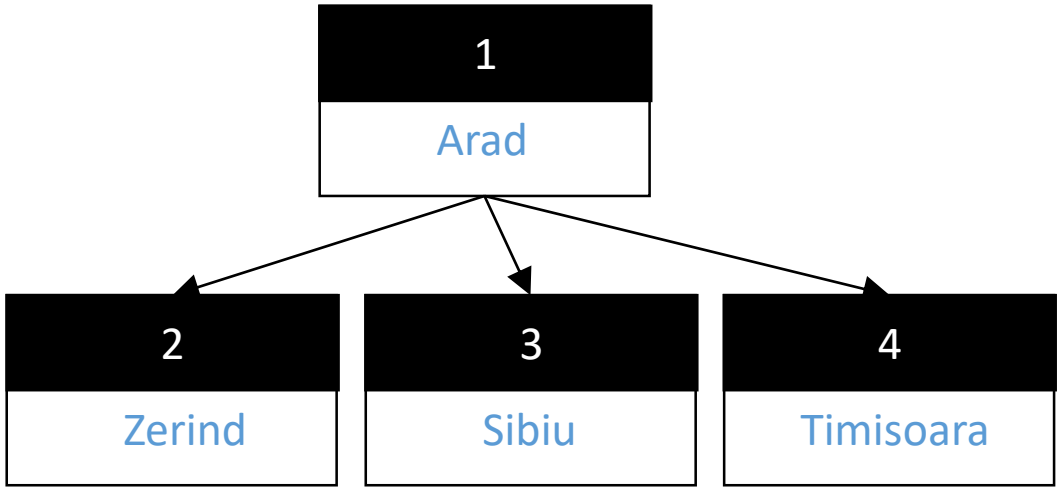
if *nodes* is empty **then return** failure

node ← REMOVE-FRONT(*nodes*)

if GOAL-TEST[*problem*] applied to STATE(*node*) succeeds **then return** *node*

nodes ← QUEUING-FN(*nodes*, EXPAND(*node*, OPERATORS[*problem*]))



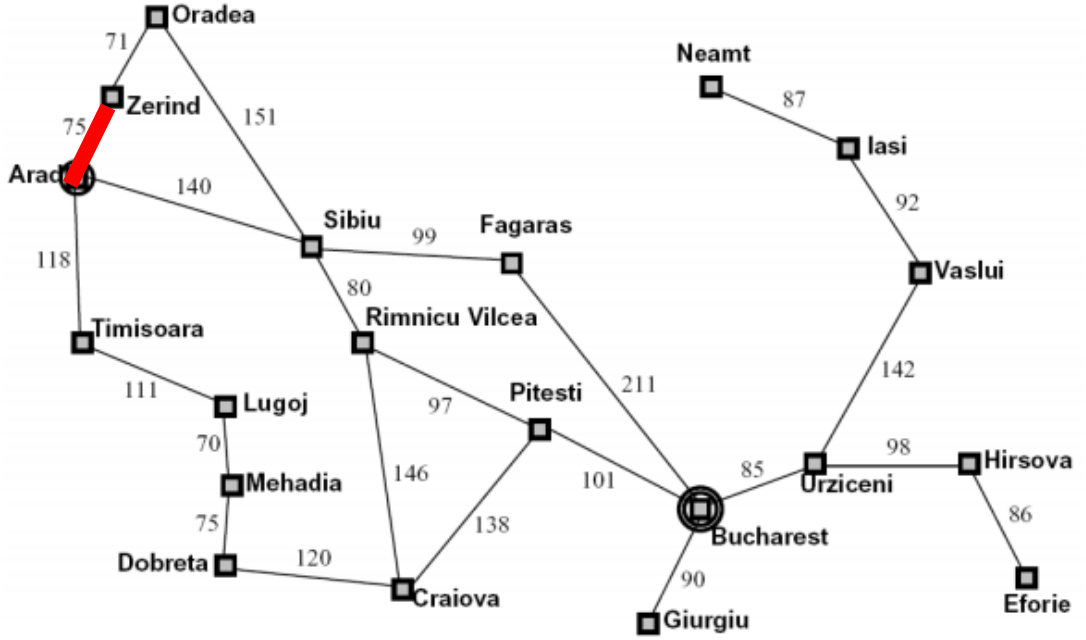


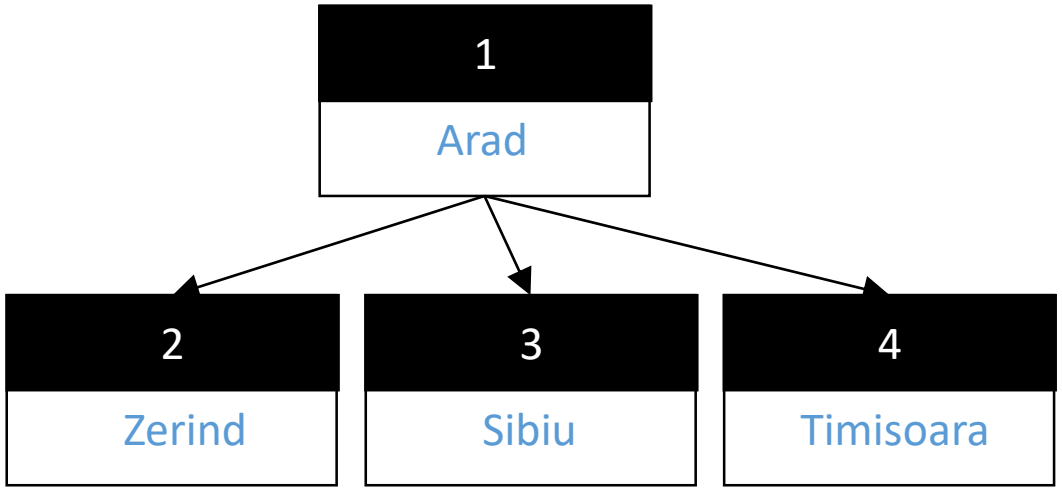
nodes

3

4

```
loop do
    if nodes is empty then return failure
    node ← REMOVE-FRONT(nodes)
    if GOAL-TEST[problem] applied to STATE(node) succeeds then return node
    nodes ← QUEUING-FN(nodes, EXPAND(node, OPERATORS[problem]))
```



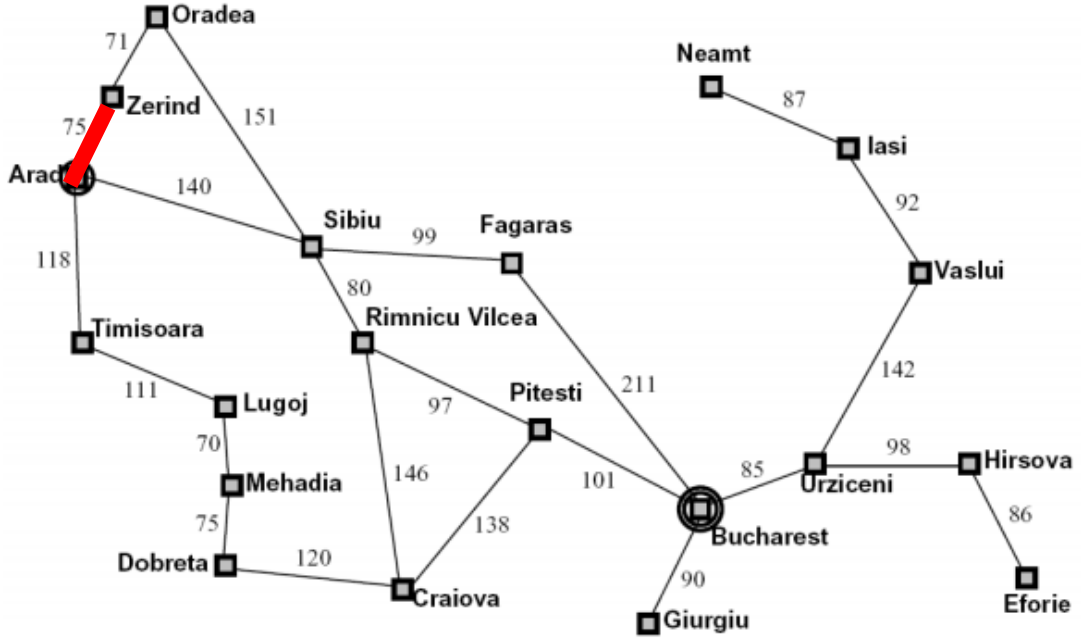


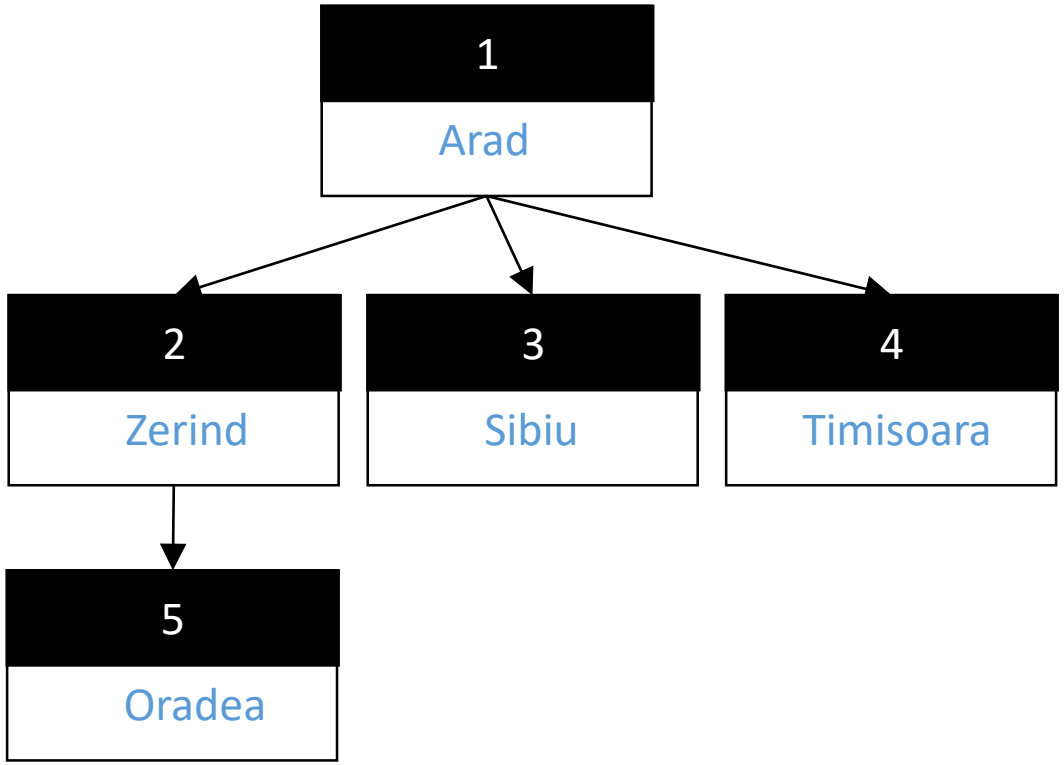
nodes

3

4

loop do
  if *nodes* is empty **then return** failure
  node ← REMOVE-FRONT(*nodes*)
  if GOAL-TEST[*problem*] applied to STATE(*node*) succeeds **then return** *node*
  nodes ← QUEUING-FN(*nodes*, EXPAND(*node*, OPERATORS[*problem*]))



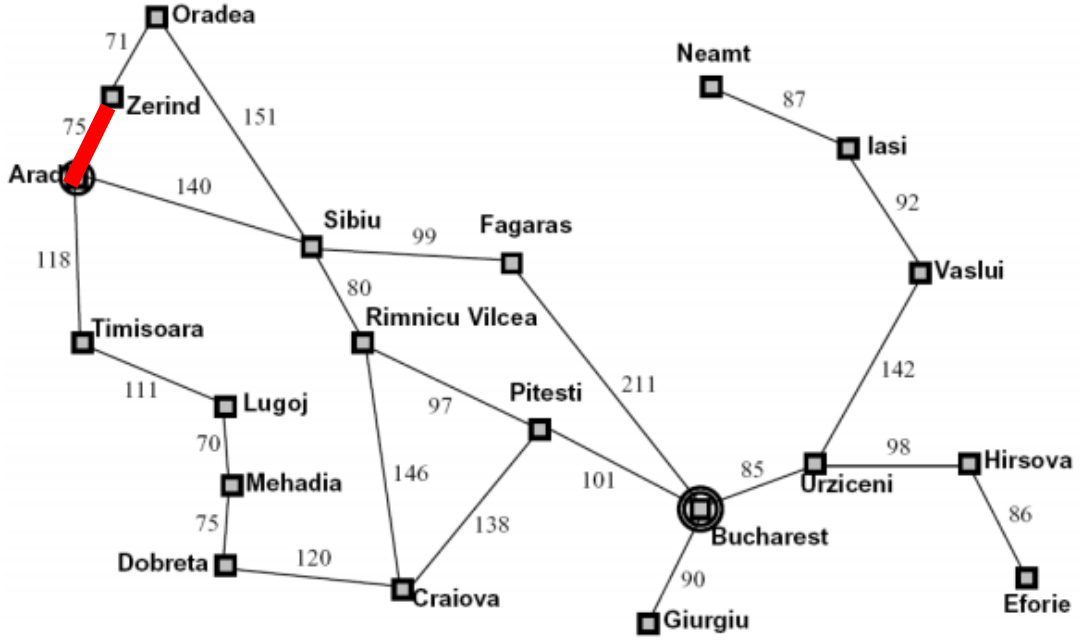


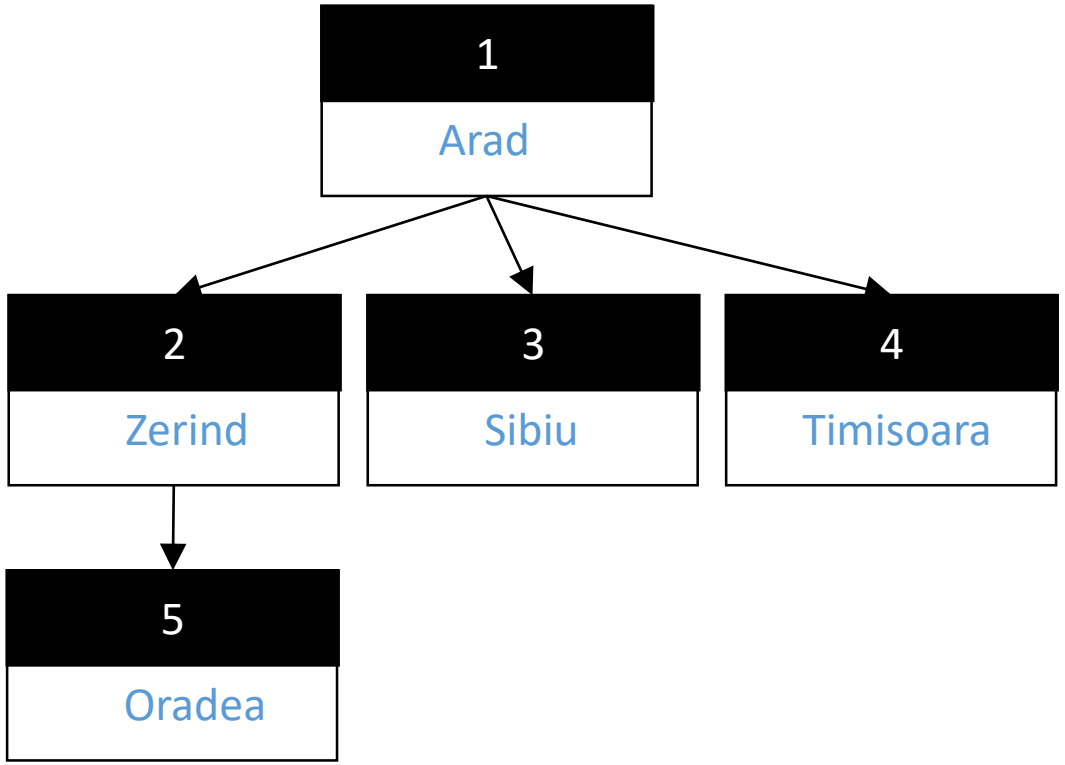
nodes

5
3
4

loop do
  if *nodes* is empty **then return** failure
  node ← REMOVE-FRONT(*nodes*)
  if GOAL-TEST[*problem*] applied to STATE(*node*) succeeds **then return** *node*
  nodes ← QUEUING-FN(*nodes*, EXPAND(*node*, OPERATORS[*problem*]))

با فرض اینکه مسیرها یک طرفه باشد





nodes

5
3
4

ادامه به عنوان تمرین

```
loop do
    if nodes is empty then return failure
    node ← REMOVE-FRONT(nodes)
    if GOAL-TEST[problem] applied to STATE(node) succeeds then return node
    nodes ← QUEUING-FN(nodes, EXPAND(node, OPERATORS[problem]))
```

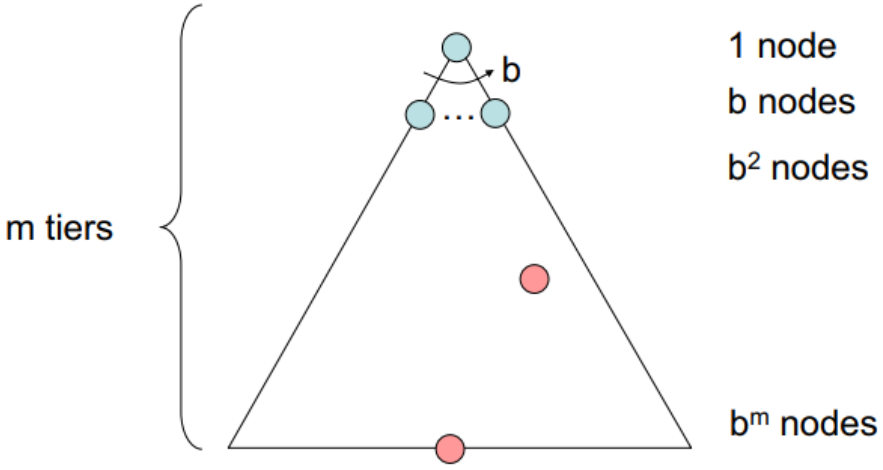
ارزیابی استراتژی های جستجو

معیارهای ارزیابی استراتژی های جستجو:

- کامل بودن (Completeness): در صورت وجود پاسخ، آیا استراتژی جستجو تضمین می کند که آن را خواهد یافت؟
- پیچیدگی زمانی (Time Complexity): چه مدت زمان نیاز است تا الگوریتم پاسخ را بیابد.
- پیچیدگی فضایی (Space Complexity): چه مقدار حافظه نیاز است تا الگوریتم کار خود را انجام دهد.
- بهینگی (Optimality): چه مقدار حافظه نیاز است تا الگوریتم کار خود را انجام دهد.

شکل کامل درخت جستجو

- **B**: فاکتور انشعاب
- **m**: حداکثر عمق
- جواب ها می توانند در عمق های مختلف باشند.



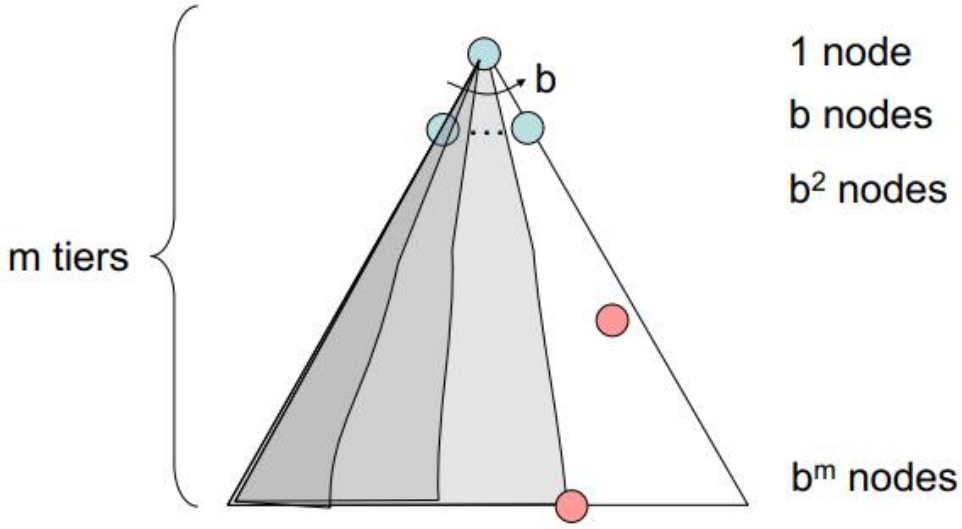
تعداد کل گره های درخت $1 + b + b^2 + \cdots + b^m = O(b^m)$

ارزیابی استراتژی عمق اول

DFS چه گرههایی را بسط می دهد؟

این الگوریتم همواره یک پیشوند چپ از کل درخت جستجو را بسط می دهد

در صورتی که حداکثر عمق محدود باشد، می تواند کل درخت جستجو را پردازش کند.



پیچیدگی زمانی

در صورتی که حداکثر عمق محدود باشد، دارای پیچیدگی زمانی $O(b^m)$ است.

پیچیدگی فضایی

تنها همموالدهای گرههای تا ریشه را نگهداری می کند. بنابراین پیچیدگی فضایی $O(bm)$ است.

آیا DFS کامل است؟

خیر، m می تواند بینهایت باشد. تنها در صورتی که m محدود باشد یا از گرههای تکراری جلوگیری کنیم، این استراتژی کامل خواهد بود.

آیا DFS بهینه است؟

خیر، این الگوریتم همواره سمت چپ ترین گره را بدون در نظر گرفتن عمق یا هزینه می یابد.