Artificial Intelligence
Chapter-4
Informed Search Methods

By : J.Razmara

Azarbaijan University

مقدمه

- دانش رهیافتی – اکتشافی (Heuristic) :

نوعی استراتژی جستجو که به صورت رهیافتی و استدلال سرانگشتی، در اغلب موارد ما را به جواب نزدیکتر می کنند.

- دانش رهیافتی بر گرفته از ساختار مسئله بوده و هدف از بکارگیری آنها تسریع در حل مسئله است.

انواع روشهای جستجوی آگاهانه

- جستجوی اول بهترین Best First Search-BFS
  - جستجوی حریصانه Greedy Search
  - جستجوی A* A*-Search
- جستجوی عمقی تکراری Iterative Deepening A*-IDA*
- جستجوی حافظه محدود ساده شده Simplified Memory Bounded A*
- جستجوی تپه نوردی Hill Climbing

جستجوی اول بهترین

- در الگوریتم General-Search تنها در تابع صف مورد استفاده می توان دانش هیوریستیک را اعمال کرد.
- تابع ارزیابی(Evaluation function) با استفاده از دانش هیوریستیک، میزان مطلوب بودن هر گره برای بسط دادن را تخمین می زند.
- هر بار مطلوب ترین گره انتخاب شده بسط داده می شود.

جستجوی اول بهترین

function BEST-FIRST-SEARCH (*problem*, EVAL-FN) **returns** a solution sequence

input: *problem*, a problem

EVAL-FN, an evaluation function

QUEUEING-FN ← a function that orders nodes by EVAL-FN

return GENERAL-SEARCH (*problem,* QUEUEING-FN)

جستجوی اول بهترین

- دو نوع خاص از جستجوی اول بهترین:
  - جستجوی حریصانه Greedy Search
  - جستجوی A* A*-Search

جستجوی حریصانه

- استراتژی حداقل هزینه تخمین زده شده برای رسیدن به هدف را دنبال می کند.
- یکی از ساده ترین استراتژیهای جستجوی اول بهترین است.

- گرهی که به نظر به هدف نزدیکتر است ابتدا بسط داده می شود.
- با استفاده از یک تابع هیوریستیک چنین گرهی شناسائی می شود:

ارزانترین هزینه تخمین زده شده از گره n *به هدف* = $h(n)$

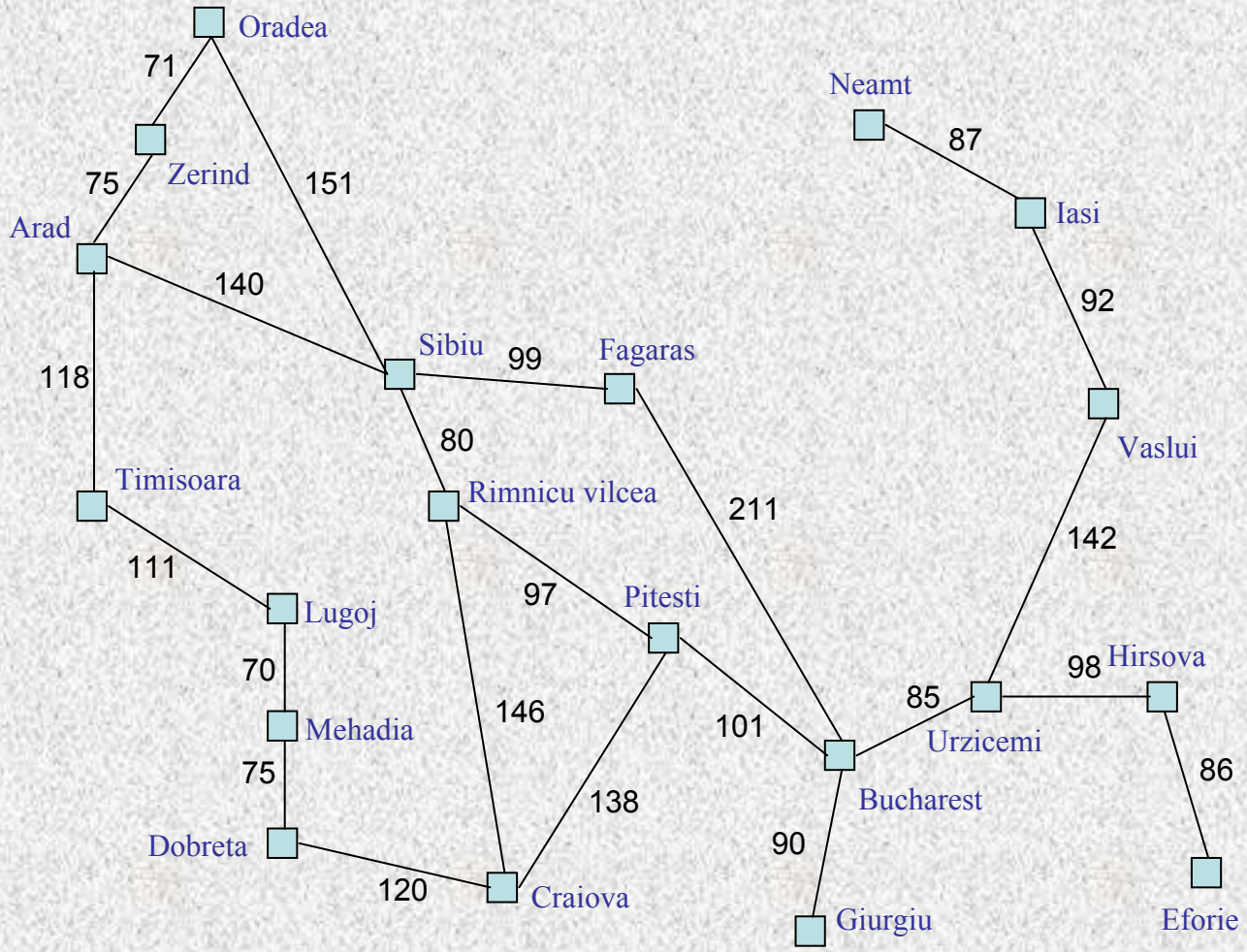
جستجوی حریصانه

function GREEDY-SEARCH (*problem*) **returns** a solution or failure

return BEST-FIRST-SEARCH (*problem*, h)

جستجوی حریصانه

- مثال : فاصله بین شهرهای رومانی بر حسب کیلومتر



Single-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urzicemi	80
Vaslui	199
Zerind	374

جستجوی حریصانه

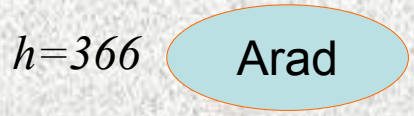
- مثال : حرکت از شهر Arad به شهر Bucharest

فاصله مستقیم از شهر n تا Bucharest = $h(n)$

- جستجوی حریصانه گرهی را دنبال می کند که به نظر نزدیکترین شهر به گره هدف یعنی شهر Bucharest است.

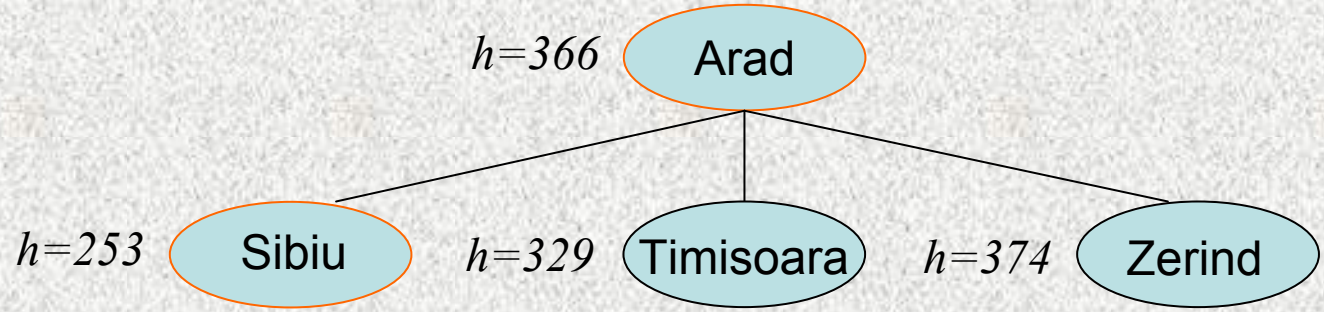
جستجوی حریصانه

- مثال : حرکت از شهر Arad به شهر Bucharest



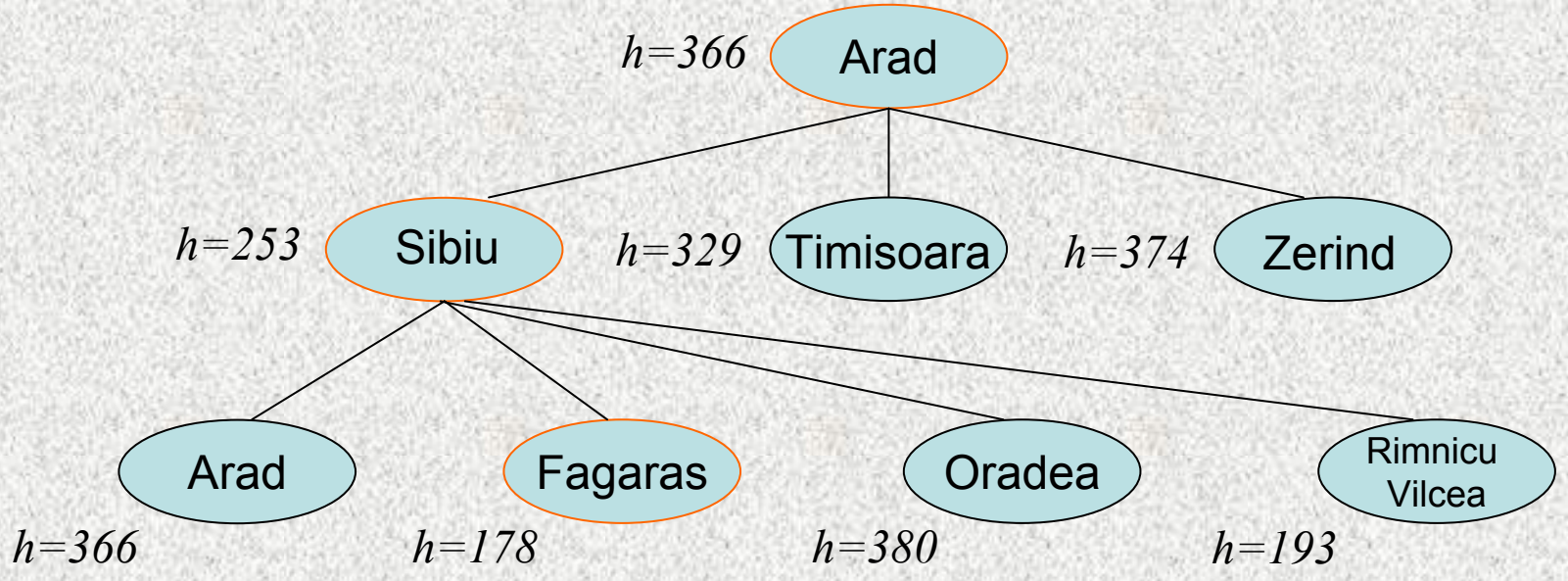
جستجوی حریصانه

- مثال : حرکت از شهر Arad به شهر Bucharest



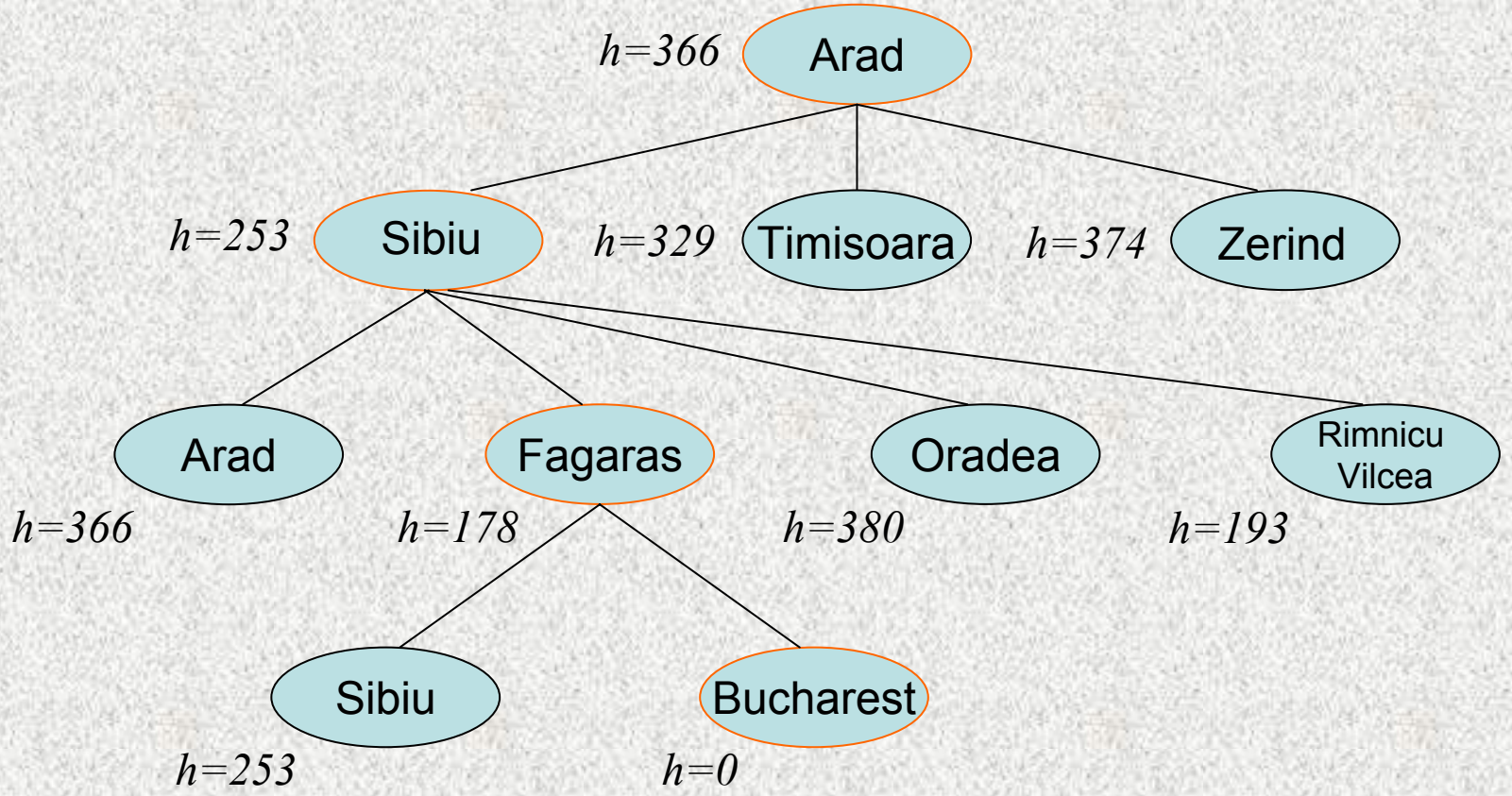
جستجوی حریصانه

- مثال : حرکت از شهر Arad به شهر Bucharest



جستجوی حریصانه

- مثال : حرکت از شهر Arad به شهر Bucharest



جستجوی حریصانه

- الگوریتم ترجیح می دهد سریعترین مسیر را انتخاب نماید بدون اینکه نگران طولانی بودن آن باشد، به همین دلیل نام جستجوی حریصانه برای آن مناسب است.

جستجوی حریصانه

- ارزیابی روش:
  - کامل بودن: کامل نیست، چرا که ممکن است در حلقه گیر کند.

    بطور مثال از شهر Isai به شهر Fagaras :

    Isai → Neamt → Isai → Neamt → …

    در صورتیکه وجود گره های تکراری در مسیر کنترل شود مسیر کامل خواهد بود.

  - بهینه بودن: بهینه نیست. (رجوع به مثال قبل)
  - پیچیدگی زمانی: $O(b^m)$ ، m: عمق جستجو
  - پیچیدگی حافظه: $O(b^m)$ ، به دلیل نگهداری تمام گرهها در حافظه

جستجوی A*

- جستجوی A* دو روش جستجوی حریصانه و هزینه یکسان را با هم ترکیب می کند:
  - جستجوی حریصانه هزینه تخمین زده شده به سمت هدف $h(n)$ را به حداقل می رساند ولی نه کامل و نه بهینه است.
  - جستجوی با هزینه یکسان هزینه مسیر $g(n)$ را به حداقل می رساند ولی مشکل حافظه و زمان دارد.
- تابع ارزیابی: $f(n)=g(n)+h(n)$
  - $g(n)$: هزینه مسیر طی شده از گره ریشه تا گره n
  - $h(n)$: هزینه تخمین زده شده از گره n تا گره هدف
  - $f(n)$: هزینه تخمین زده شده راه حل از طریق گره n

جستجوی A*

- در جستجوی A* همواره $h(n) \leq h^*(n)$ می باشد که در آن $h^*(n)$ هزینه واقعی از n تا هدف است. به عبارت دیگر مقدار h هرگز هزینه ای بیش از تخمین برای رسیدن به هدف نخواهد داشت.

- مثال: در مسئله مسیریابی بین دو شهر با استفاده از تابع $h(n)$، هزینه تخمین زده شده هیچگاه بیشتر از هزینه واقعی نخواهد بود.

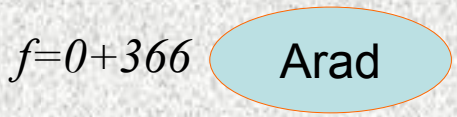
جستجوی A*

function A*-SEARCH (*problem*) **returns** a solution or failure

return BEST-FIRST-SEARCH (*problem*, g+h)

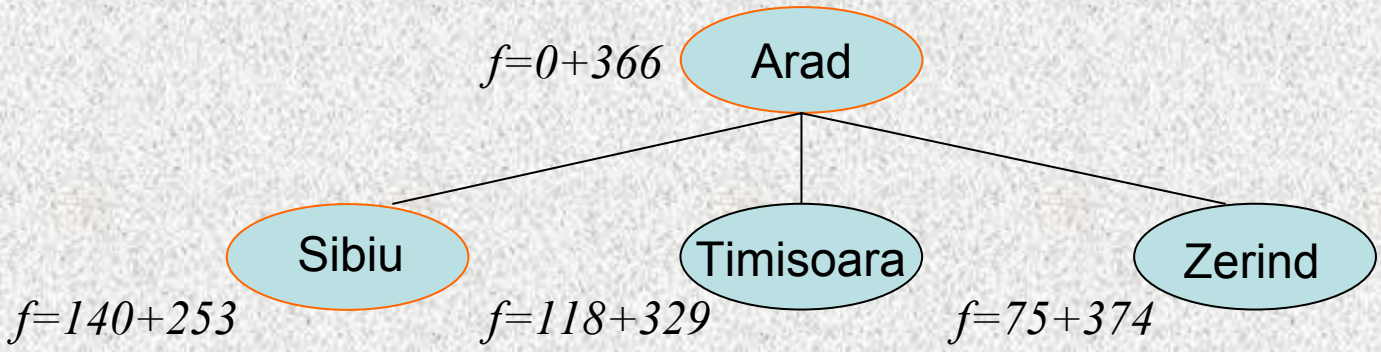
جستجوی A*

- مثال : حرکت از شهر Arad به شهر Bucharest



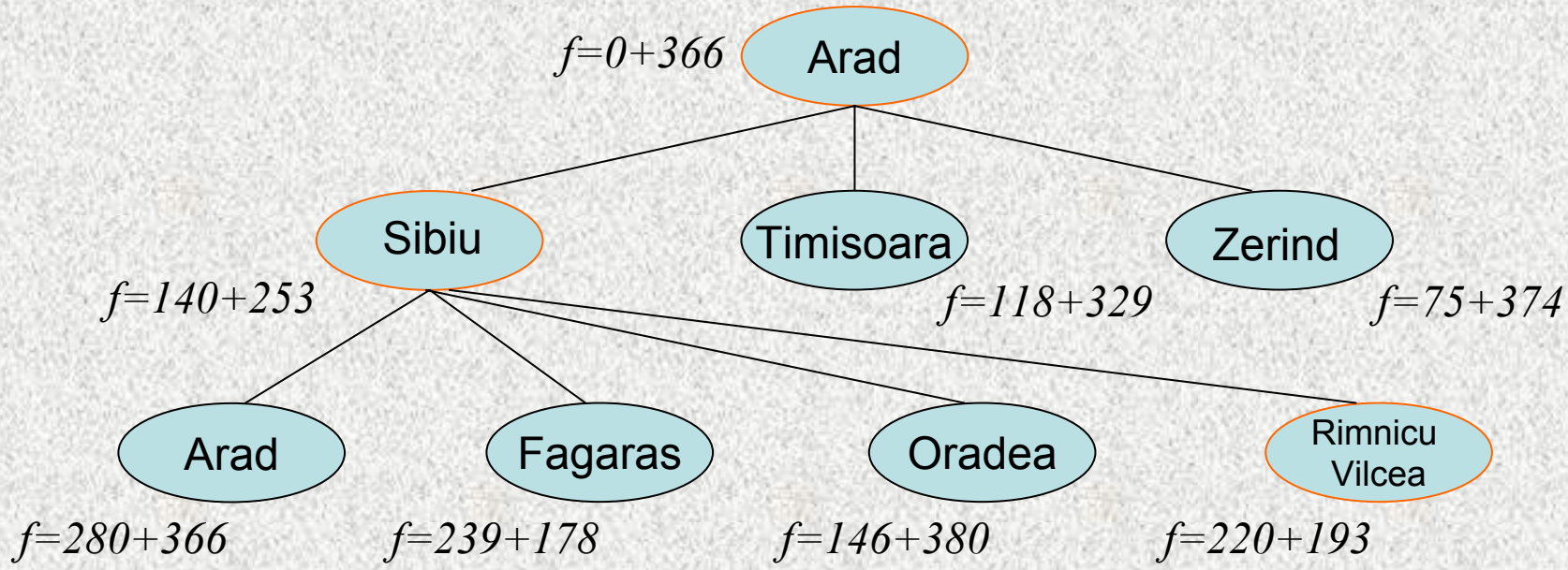
جستجوی A*

- مثال : حرکت از شهر Arad به شهر Bucharest



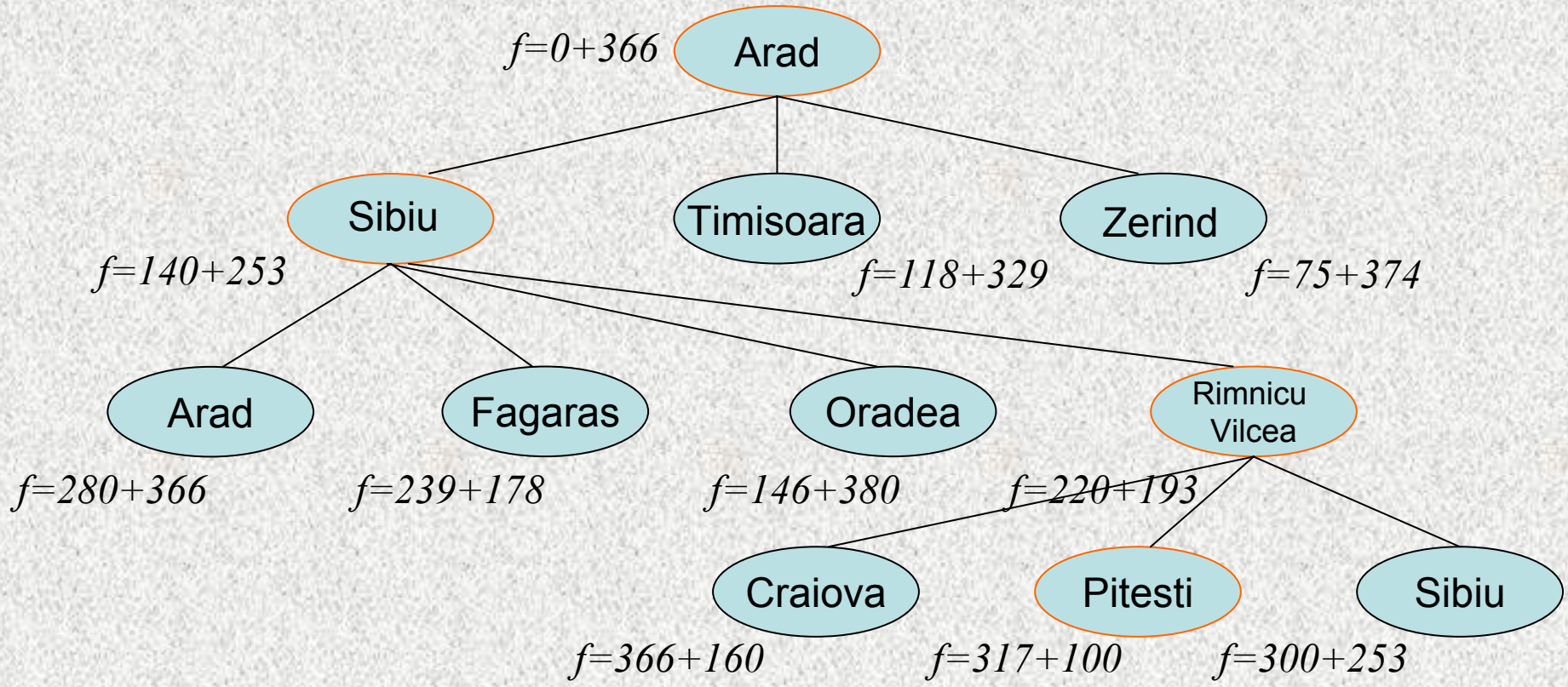
جستجوی A*

- مثال : حرکت از شهر Arad به شهر Bucharest



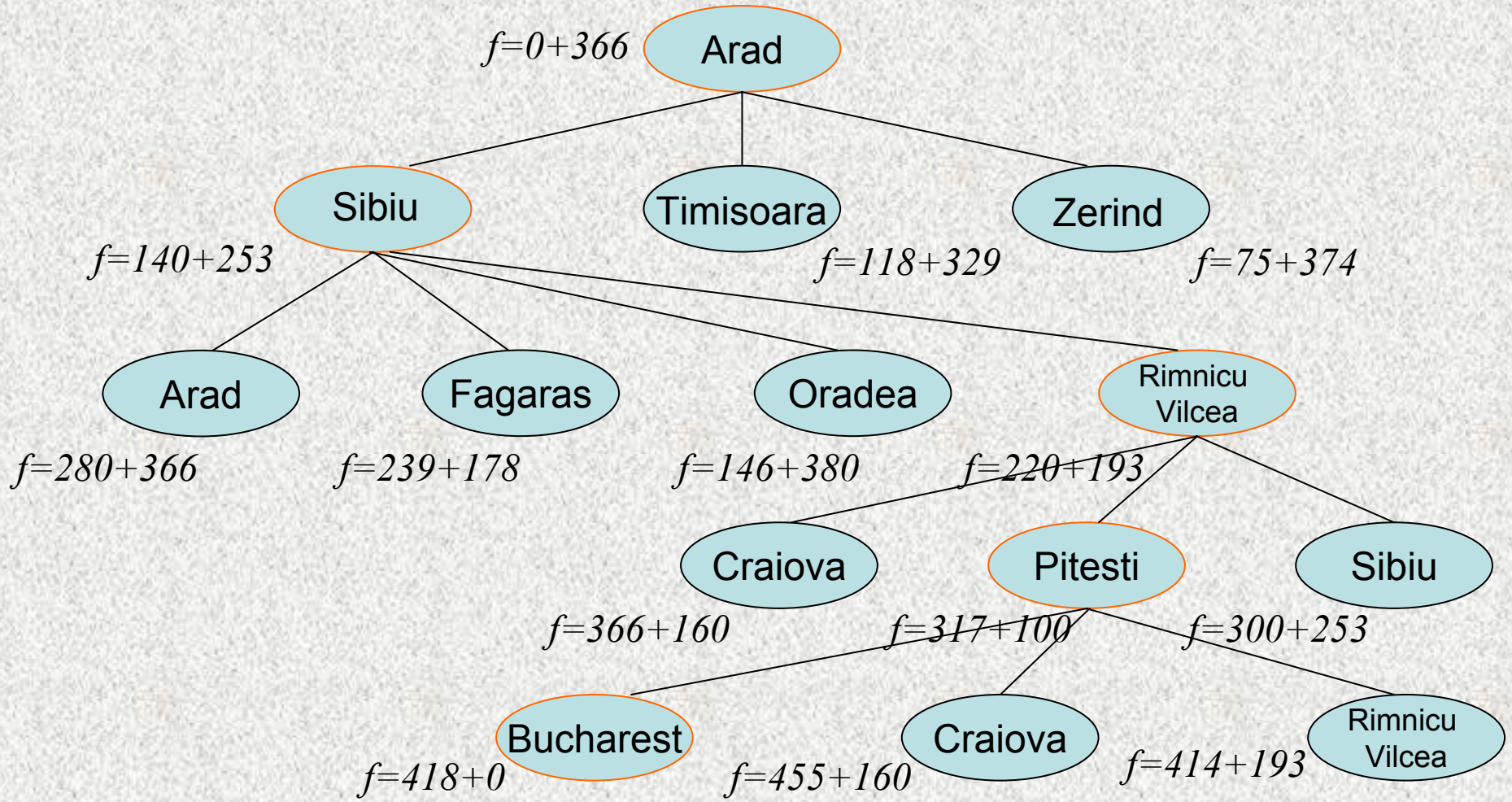
جستجوی A*

- مثال : حرکت از شهر Arad به شهر Bucharest



جستجوی A*

- مثال : حرکت از شهر Arad به شهر Bucharest



جستجوی A*

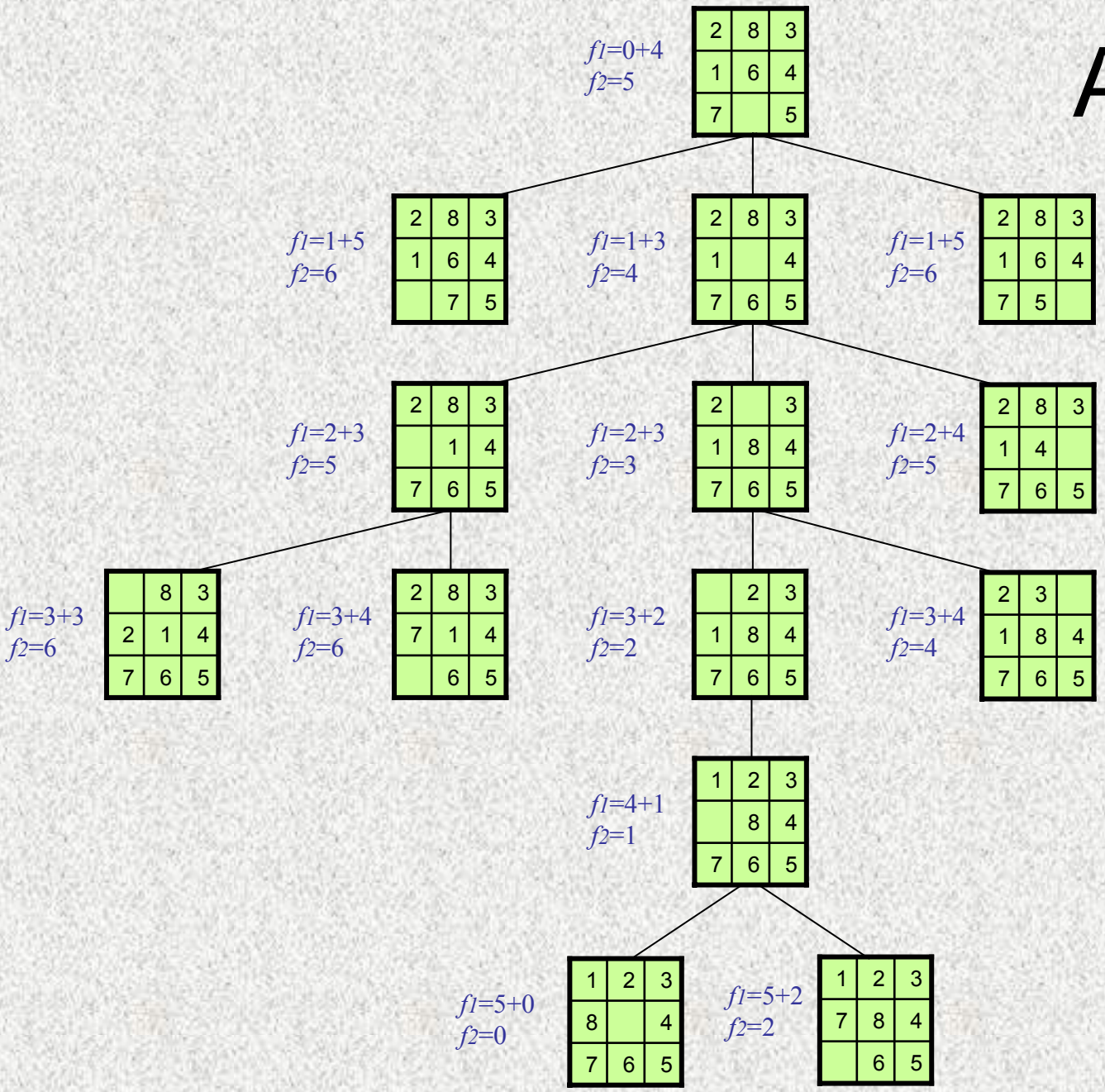
- ارزیابی روش:
  - کامل بودن: کامل است.
  - بهینه بودن: بهینه است.
  - پیچیدگی زمانی: نمائی
  - پیچیدگی حافظه: تمام گرهها در حافظه نگهداری می شود.

جستجوی A*

- ضرورت طراحی و انتخاب تابع هیوریستیک مناسب
- بررسی یک مثال – مسئله معمای هشت
  - $f1$: تعداد خانه هائی که در محل خود نیستند + هزینه مسیر طی شده
  - $f2$: مجموع فاصله خانه ها از محلهای هدف آنها

جستجوی A*

- بررسی یک مثال
  - مسئله معمای هشت



فاکتور انشعاب موثر

- تعریف- فاکتور انشعاب موثر

(Effective Branching Factor – EBF)

اگر مجموع تعداد گره های بسط داده شده توسط الگوریتم جستجو N باشد و عمق راه حل d تعریف شود b* فاکتور انشعاب برای یک درخت یکنواخت با عمق d خواهد بود تا گره های N را نگهدارد. لذا :

$$N = 1 + b^* + b^{*2} + b^{*3} + \ldots + b^{*d}$$

مثال: اگر الگوریتم A* راه حلی را در عمق ۵ با استفاده از ۲۵ گره پیدا کند در اینصورت:

$$25 = 1 + b^* + b^{*2} + b^{*3} + \ldots + b^{*5} \quad \rightarrow b^*=1.91$$

فاکتور انشعاب موثر

- فاکتور انشعاب موثر مقدار ثابتی دارد و لذا اندازه گیری آن برای مسائل با وضعیتهای محدود راهنمای خوبی برای بررسی کیفیت تابع هیوریستیک میباشد.

- هر چه فاکتور انشعاب برای یک تابع هیوریستیک به یک نزدیکتر باشد، تابع بهتر بوده و دارای کیفیت بالاتری است.

ارزیابی و مقایسه

d	Search cost			Effective Branching Factor		
	IDS	A*(f_1)	A*(f_2)	IDS	A*(f_1)	A*(f_2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	364404	227	73	2.78	1.42	1.24
14	3473941	539	113	2.83	1.44	1.23
16	-	1301	211	-	1.45	1.25
18	-	3056	363	-	1.46	1.26
20	-	7276	679	-	1.47	1.27

ارزیابی و مقایسه

- الگوریتم A* با تابع $f2$ همواره گره های کمتری را نسبت به A* با تابع $f1$ بسط می دهد.

بهبود جستجوی A*

- در حل بعضی مسائل پیچیده، جستجوی A* با کمبود حافظه مواجه می شود. برای حل اینگونه مسائل دو الگوریتم برای بهبود A* در مصرف حافظه وجود دارد:
  - جستجوی عمیق کننده تکراری A*

    Iterative Deepening A* - IDA*

    ایده اصلی بهره گیری از روش جستجوی عمقی تکراری در الگوریتم A* است.
  - جستجوی ساده شده با حافظه محدود A*

    Simplified Memory Bounded A* - SMA*

    ایده اصلی کاهش طول صف برای مقابله با محدودیت حافظه می باشد.

جستجوی IDA*

- در این الگوریتم به منظور کاهش میزان مصرف حافظه، جستجوی عمقی تکراری با جستجوی A* ترکیب می شود.
- در این الگوریتم به جای محدوده عمقی مورد استفاده در جستجوی عمقی تکراری، از محدوده $f\text{-}cost$ استفاده می شود.
- در هر بار تکرار الگوریتم، گره ها با $f(n) \leq f\text{-}cost$ بسط داده می شود و در صورت عدم موفقیت در رسیدن به هدف، محدوده جستجو یعنی $f\text{-}cost$ افزایش یافته و دوباره الگوریتم تکرار می شود.

جستجوی IDA*

```
function IDA*-SEARCH (problem)  returns a solution sequence
    input: problem, a problem
    local variables: f-limit, the current f-cost limit
                     root, a node
    root ← MAKE-NODE (INITIAL_STATE [problem])
    f-limit ← f-Cost (root)
    loop do
        solution, f-limit ← DFS-CONTOUR (root, f-limit)
        if solution is not-null then return solution
        if f-limit = ∞ then return failure
    end
```

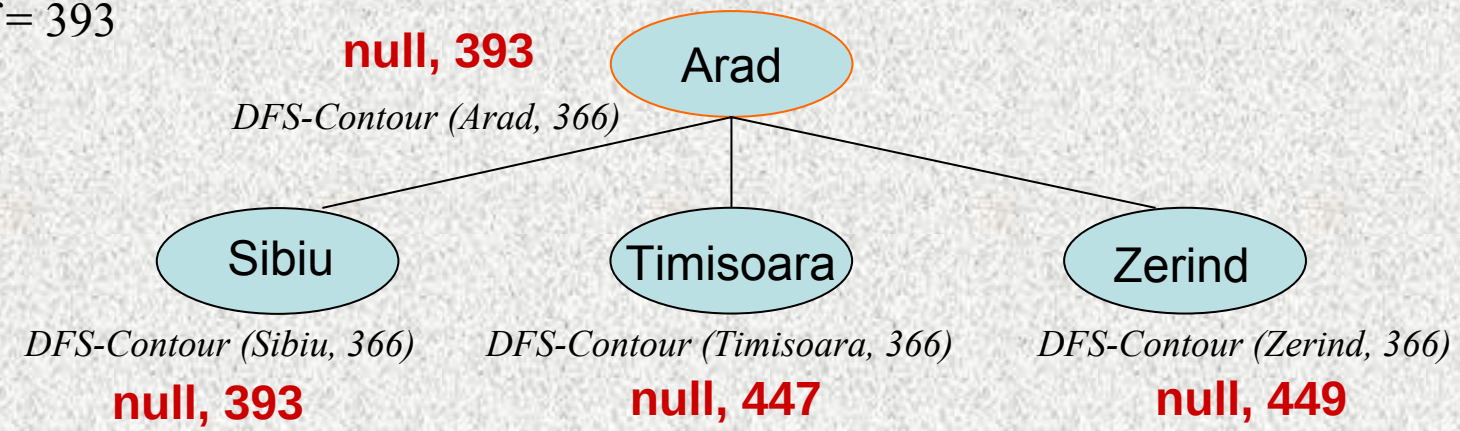
جستجوی IDA*

```
function DFS-CONTOUR (node, f-limit) returns a solution sequence and a new f-Cost
    inputs: node, a node
            f-limit, the current f-cost limit
    local variables: next-f, the f-cost limit for the next contour, initially ∞
    if f-Cost [node] > f-limit then return null, f-Cost [node]
    if GOAL-TEST [problem] (STATE [node]) then return node, f-limit
    for each node s in SUCCESSOR (node) do
        solution, new-f ← DFS-CONTOUR (s, f-limit)
        if solution is not-null then return solution, f-limit
        next-f ← MIN (next-f, new-f)
    end
    return null, next-f
```

جستجوی IDA*

- مثال : حرکت از شهر Arad به شهر Bucharest

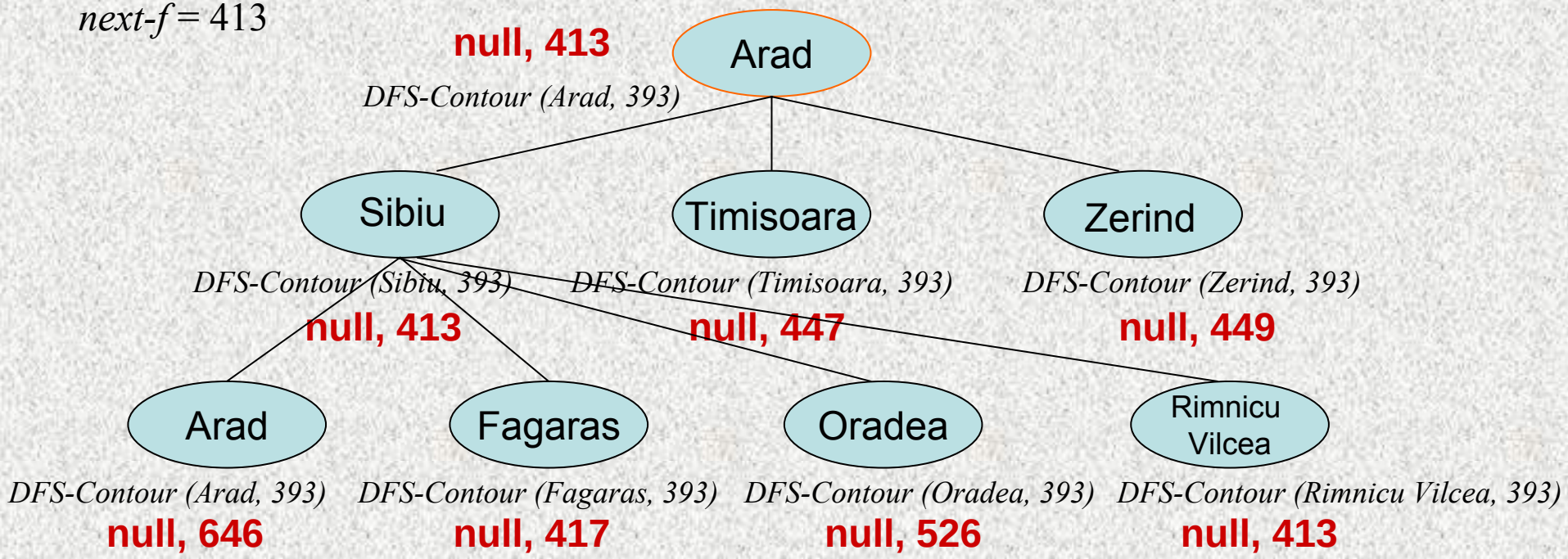
1st iteration
f-limit = 366
next-f = 393



جستجوی IDA*

- مثال : حرکت از شهر Arad به شهر Bucharest

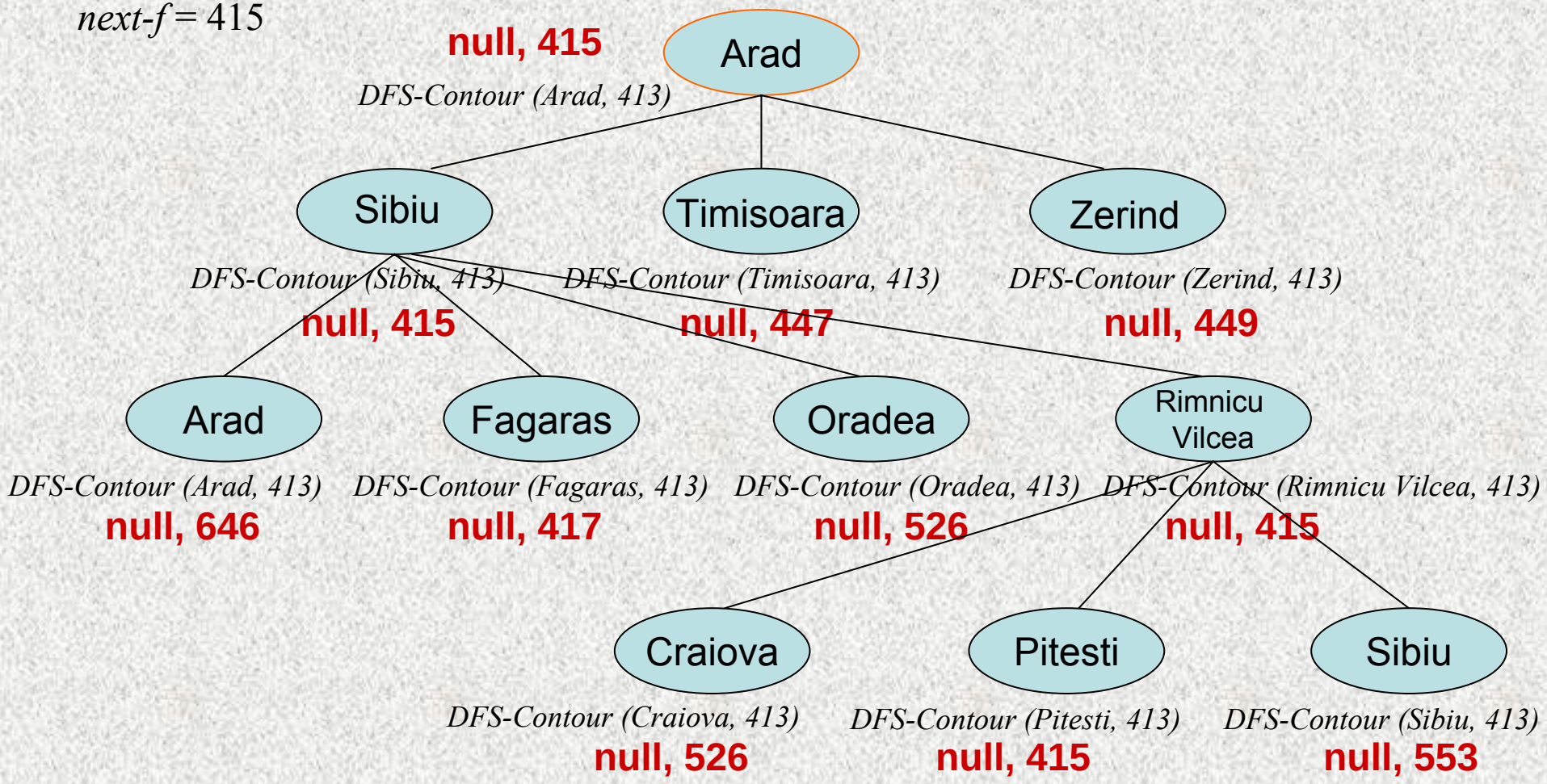
2st iteration
f-limit = 393
next-f = 413



جستجوی IDA*

- مثال : حرکت از شهر Arad به شهر Bucharest

3rd iteration
f-limit = 413
next-f = 415



جستجوی IDA*

- ارزیابی روش:
  - کامل بودن: کامل است.
  - بهینه بودن: بهینه است.
  - پیچیدگی زمانی: شدیدا" به تعداد مقادیر تولید شده توسط تابع هیوریستیک بستگی دارد.

بطور مثال در مسئله معمای هشت با تابع $f2$ (فاصله خانه ها از محل آنها) مقدار تابع فقط دو یا سه بار در طول مسیر افزایش می یابد، بنابراین جستجو فقط دو یا سه تکرار انجام می دهد.

  - پیچیدگی حافظه: جستجو به صورت عمقی است، لذا فضای حافظه را متناسب با طولانی ترین مسیر پیموده شده مصرف می کند. معمولا bd تخمین خوبی برای مصرف حافظه در IDA* است.

جستجوی SMA*

- جستجوی A* حداکثر تعداد bd گره در حافظه نگه می دارد اما همواره محاسبات را تکرار می کند.
- در صورتیکه حافظه بیشتری موجود باشد در جستجوی SMA* تعداد گره های بیشتری را می توان در حافظه ذخیره نمود تا از تکرار محاسبات کم کرد.
  - در صورت پر شدن حافظه نیاز به حذف بعضی از گره ها وجود دارد.
  - سطحی ترین گره هائی که مقدار *f-Cost* بیشتری دارند ابتدا حذف می شوند.
  - برای به خاطر آوردن گره های حذف شده، اطلاعات مربوط به بهترین مسیر پیموده شده در گره پدر نگهداری می شود.
  - یک زیر درخت تنها زمانی دوباره ایجاد می شود که ثابت شود تمام مسیرهای موجود بدتر از مسیر حذف شده باشد.

جستجوی SMA*

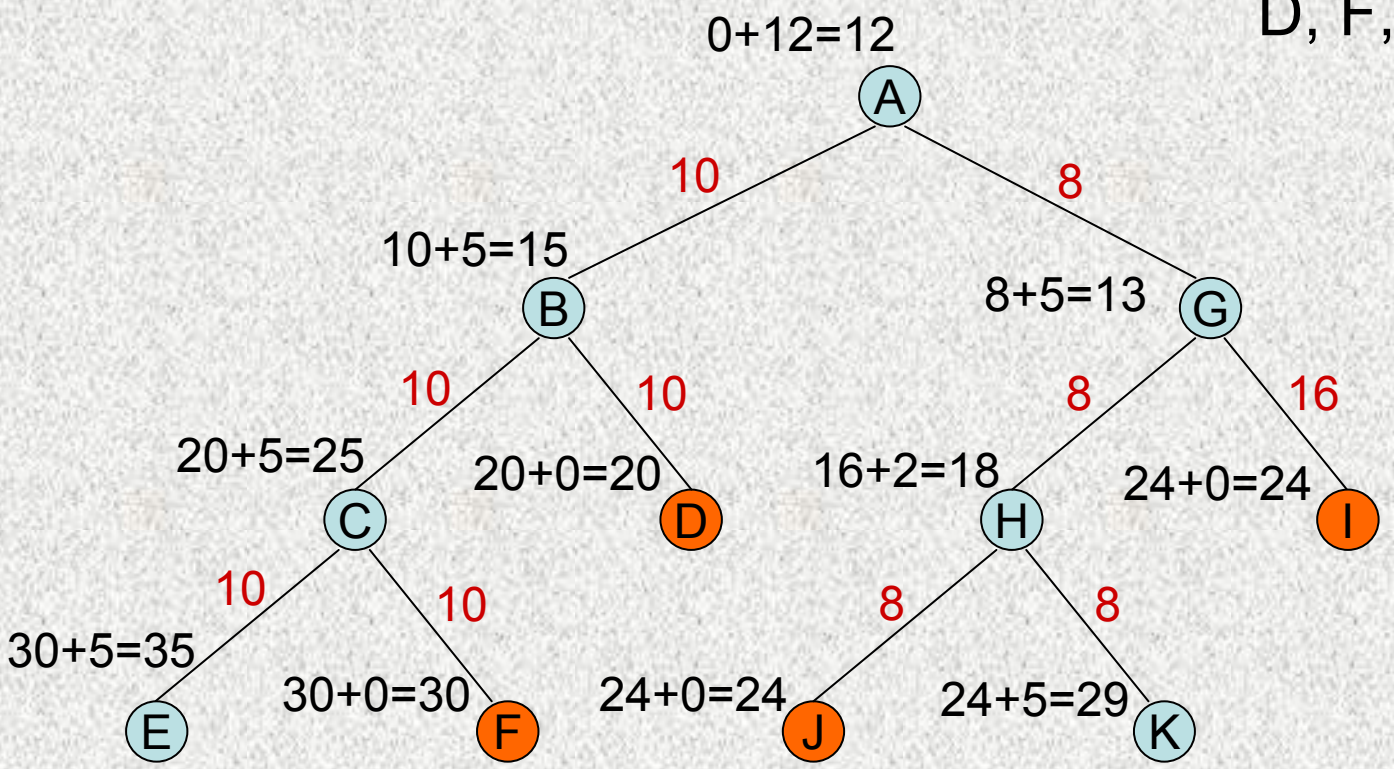
```
function SMA*-SEARCH (problem)  returns a solution sequence
    input: problem, a problem
    local variables: Queue, a queue of nodes ordered by f-cost
    Queue ← MAKE-QUEUE (MAKE-NODE (INITIAL_STATE [problem]))
    loop do
        if Queue is empty then return failure
        n ← deepest least-f-cost node in Queue
        if GOAL-TEST (n) then return success
        s ← NEXT-SUCCESSOR (n)
        if s is not a goal and is at maximum depth then f(s) ← ∞
        else f(s) ← MAX ( f(n), g(s) +h(s) )
        if all of n's successors have been generated then
            update n's f-cost and those of its ancestors if necessary
        if SUCCESSORS (n) all in memory then remove n from Queue
        if memory is full then
            delete shallowest, highest f-cost node in Queue
            remove it from its parents successor list
            insert its parent on Queue if necessary
        insert s on Queue
    end
```

جستجوی SMA*

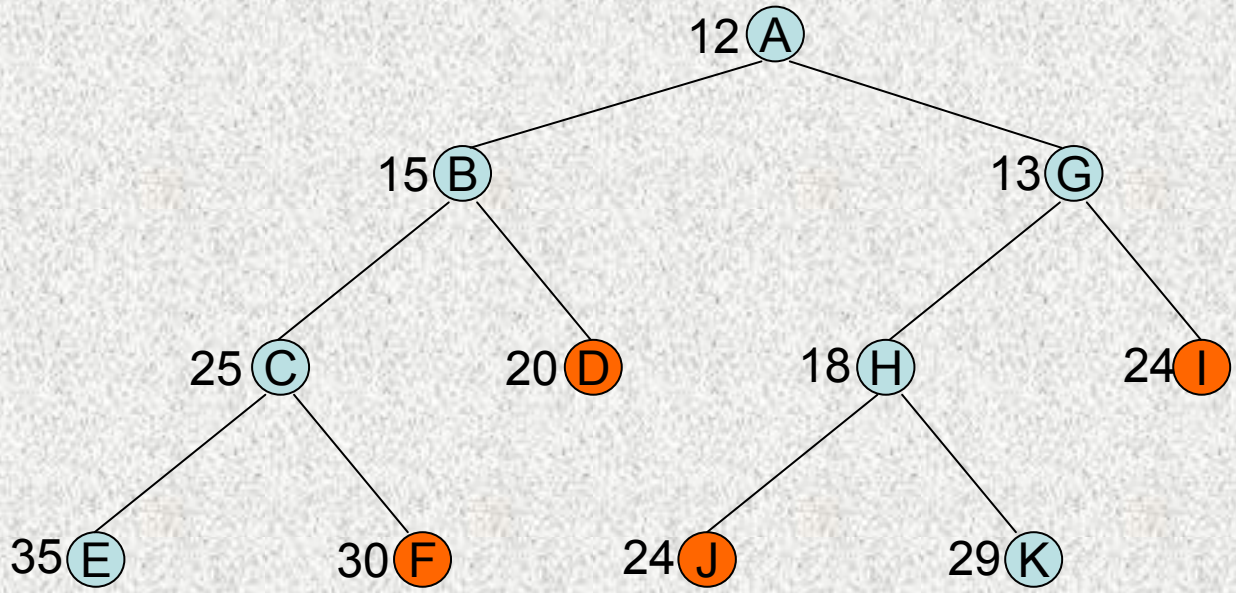
- مثال-

گره های هدف: D, F, I, J

ظرفیت حافظه: ۳ گره



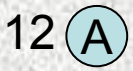
جستجوی SMA*



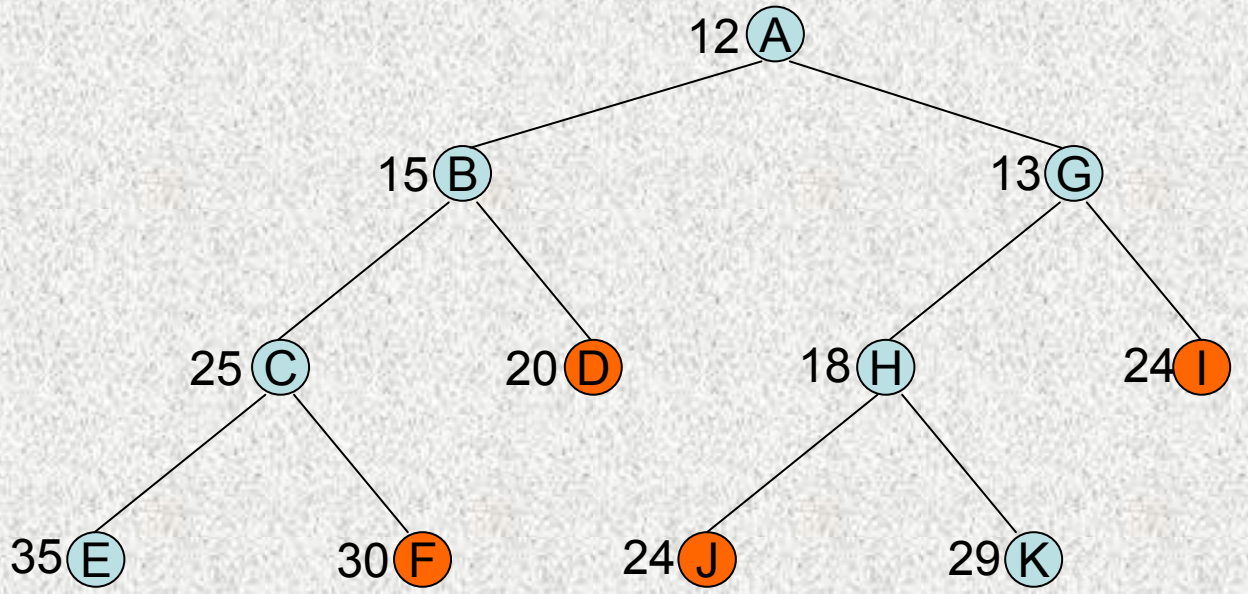
- مثال-

گره های هدف: D, F, I, J

ظرفیت حافظه: ۳ گره



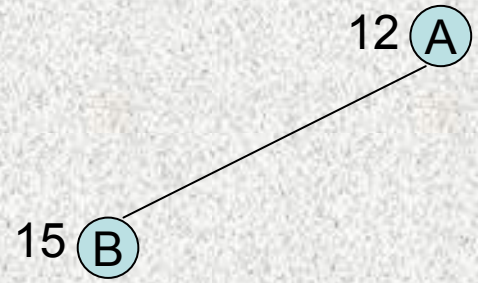
جستجوی SMA*



- مثال-

گره های هدف: D, F, I, J

ظرفیت حافظه: ۳ گره

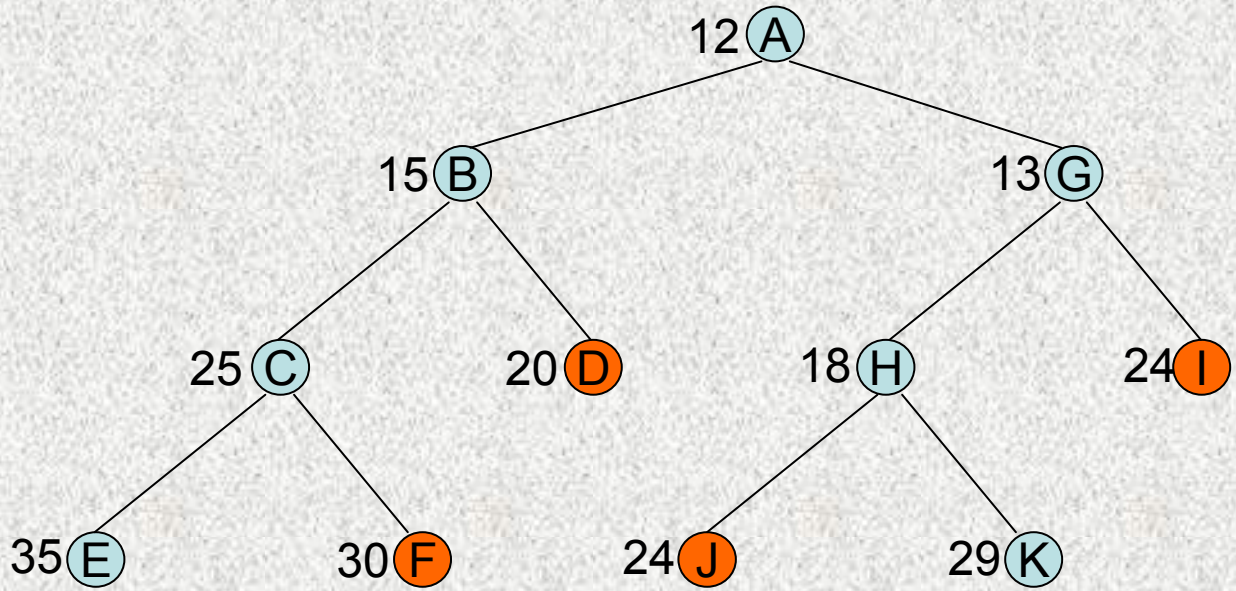


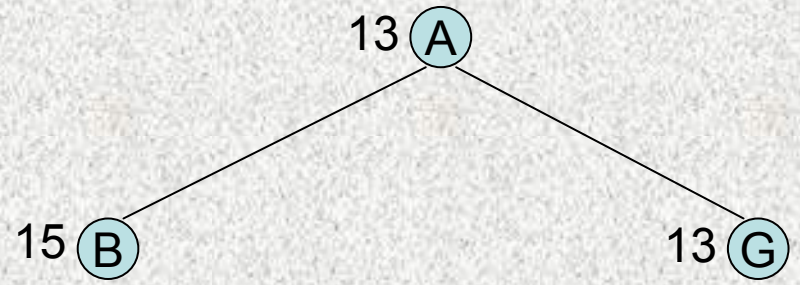
جستجوی SMA*

- مثال-

گره های هدف: D, F, I, J

ظرفیت حافظه: ۳ گره



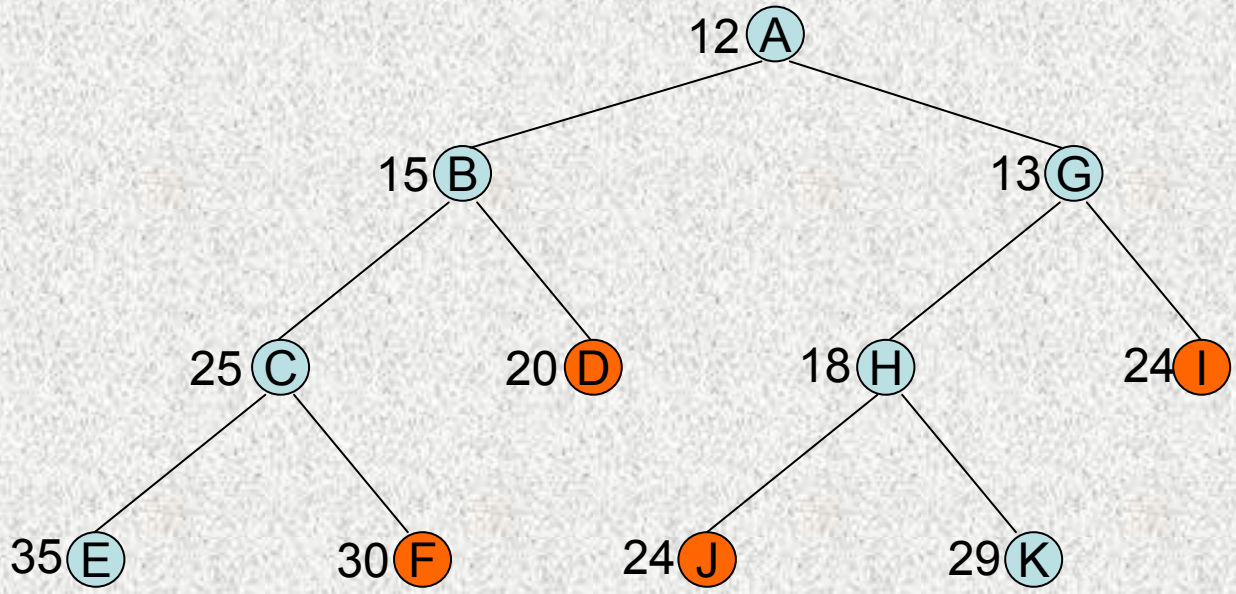


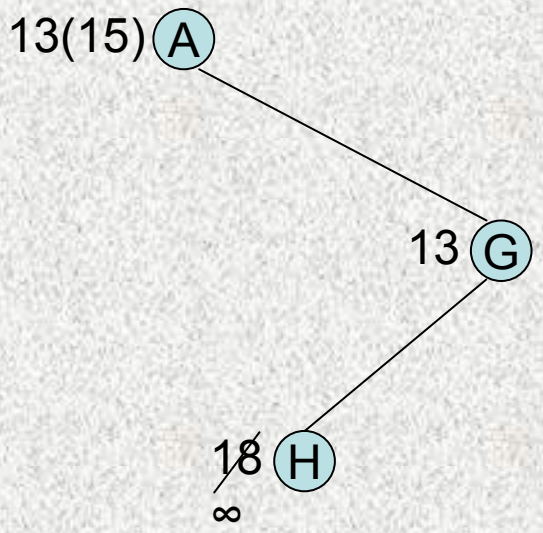
جستجوی SMA*

- مثال-

گره های هدف: D, F, I, J

ظرفیت حافظه: ۳ گره



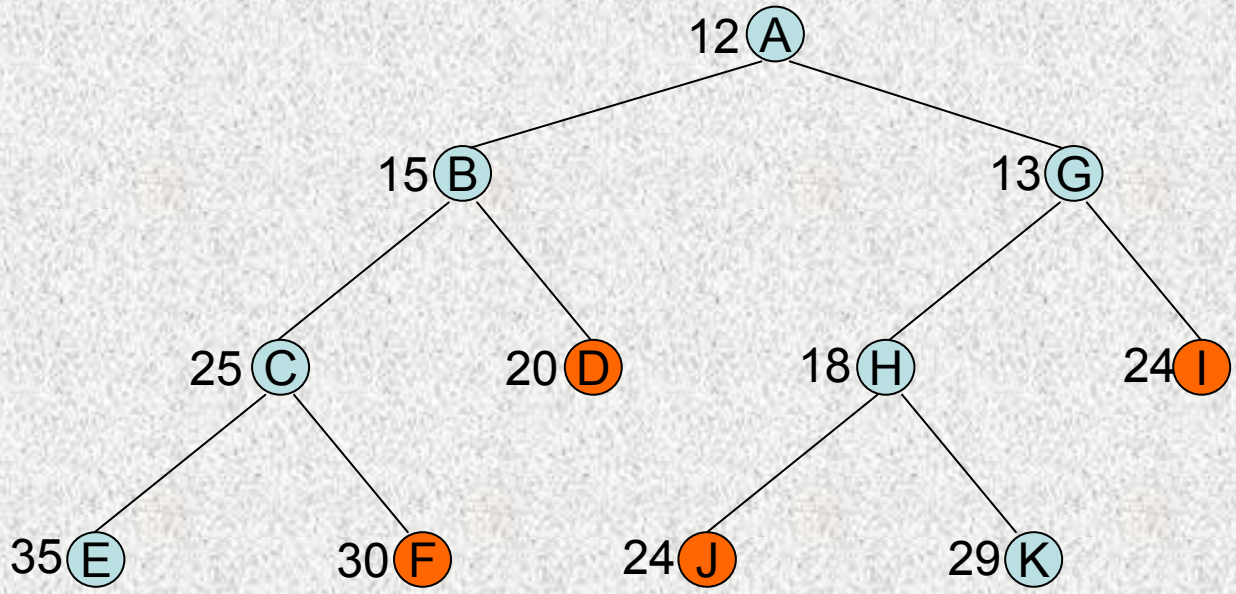


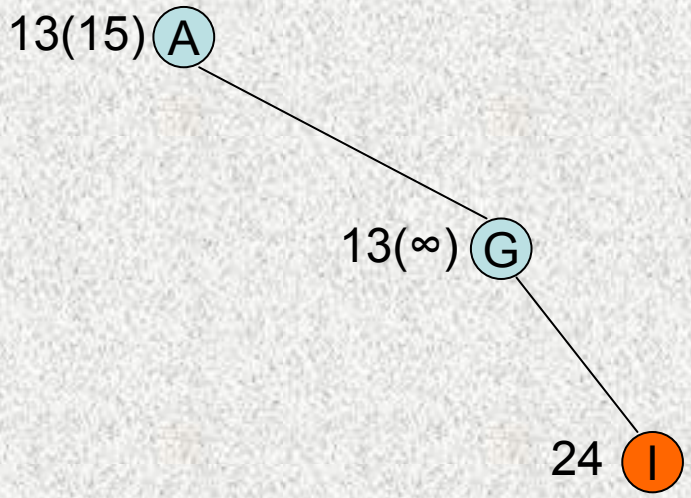
جستجوی SMA*

- مثال-

گره های هدف: D, F, I, J

ظرفیت حافظه: ۳ گره



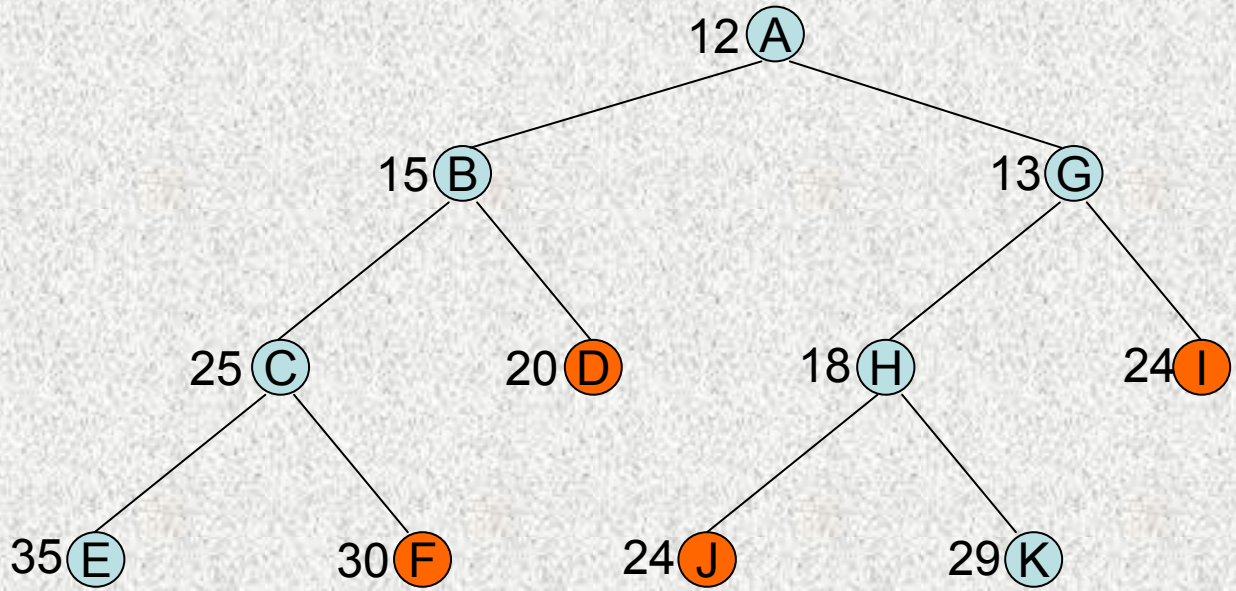


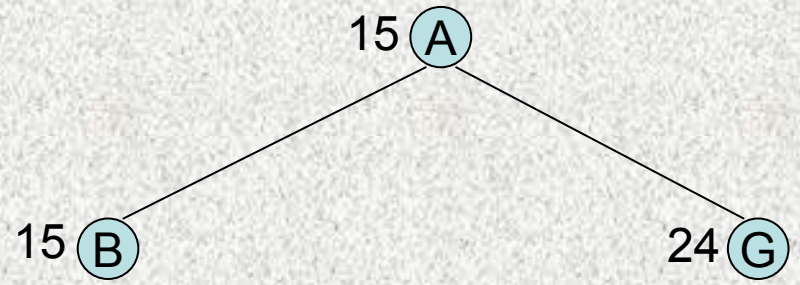
جستجوی SMA*

- مثال-

گره های هدف: D, F, I, J

ظرفیت حافظه: ۳ گره



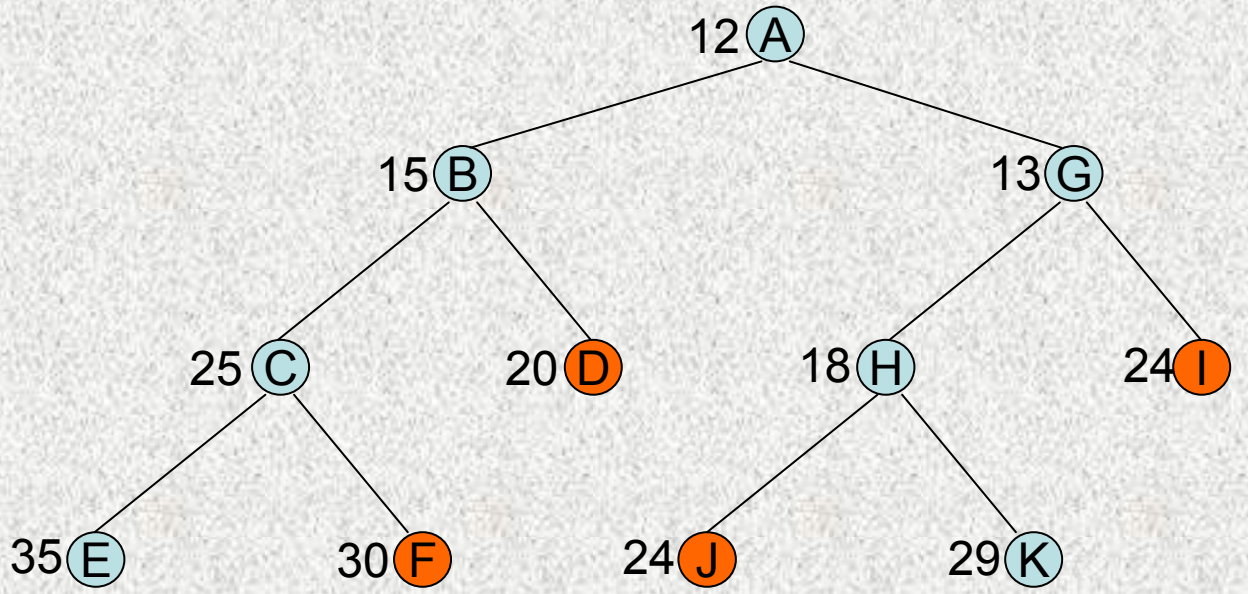


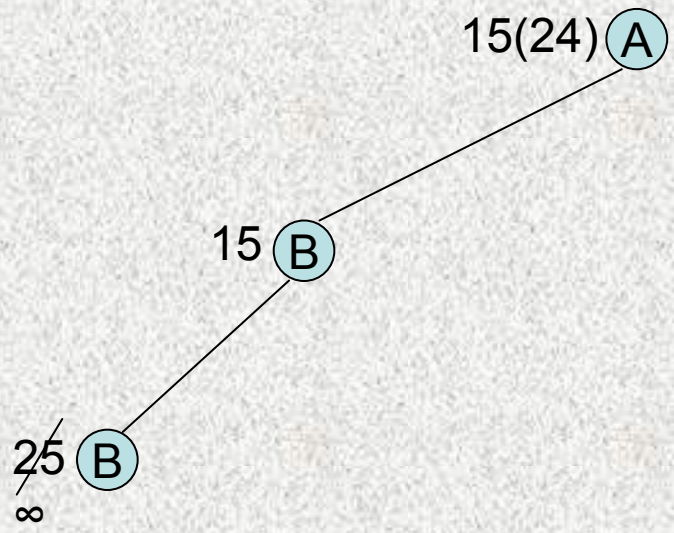
جستجوی SMA*

- مثال-

گره های هدف: D, F, I, J

ظرفیت حافظه: ۳ گره



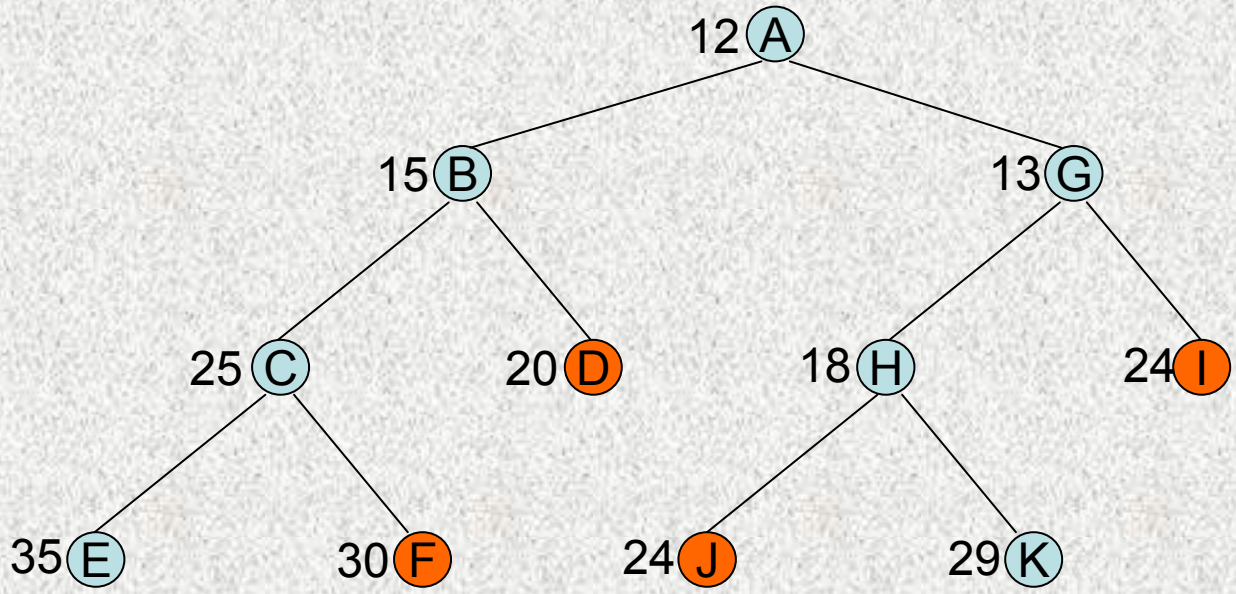


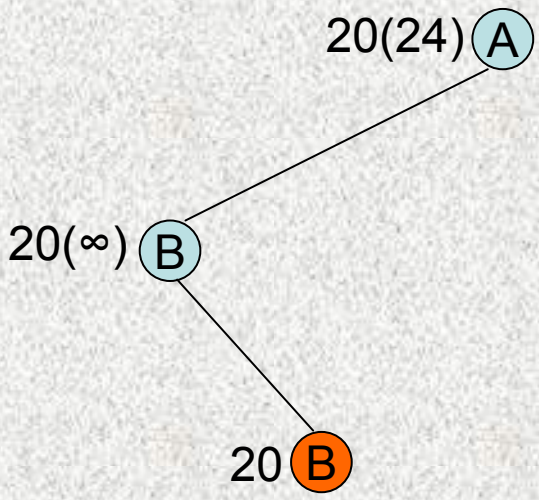
جستجوی SMA*

- مثال-

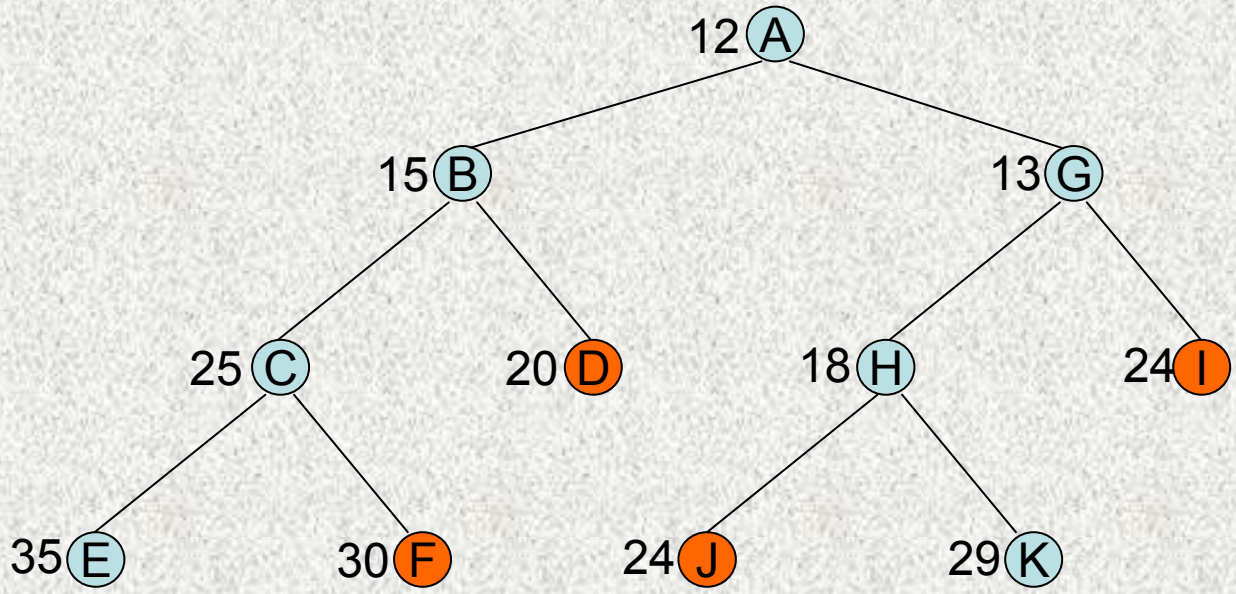
گره های هدف: D, F, I, J

ظرفیت حافظه: ۳ گره





جستجوی SMA*



- مثال–

گره های هدف: D, F, I, J

ظرفیت حافظه: ۳ گره

- در مثال فوق، حافظه کافی برای کم عمق ترین مسیر بهینه وجود دارد.
- اگر J هزینه ۱۹ به جای ۲۴ داشت الگوریتم قادر به یافتن آن نبود.
- با داشتن فضای حافظه منطقی، الگوریتم قادر به حل مسائل مشکل تری نسبت به A* می باشد بدون آنکه مشکل سرریزی گره های اضافی مطرح باشد.

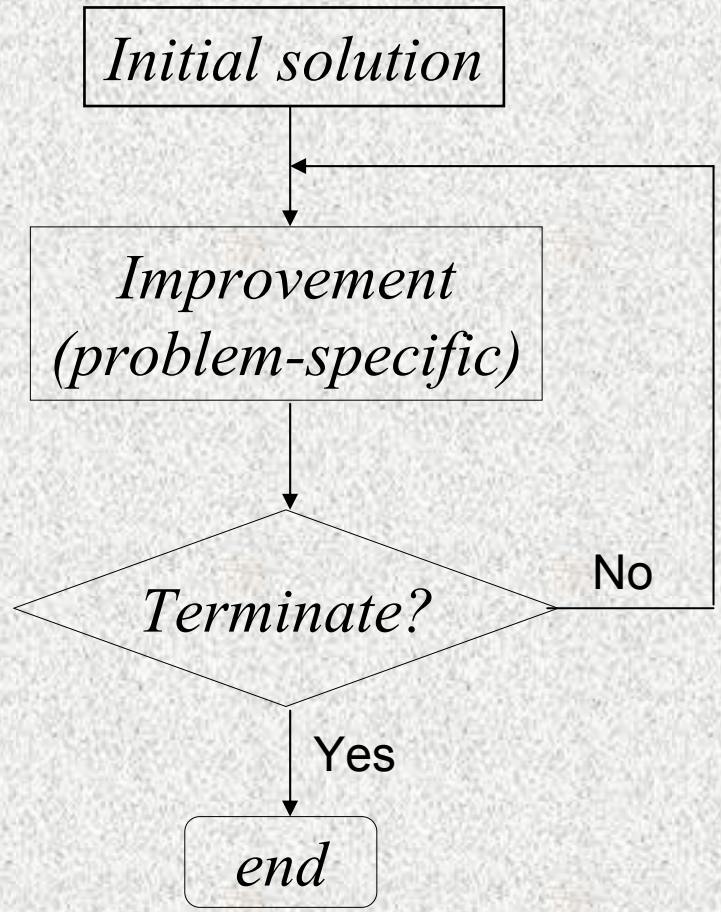
جستجوی SMA*

- ارزیابی روش:
  - کامل بودن: در صورت وجود حافظه کافی برای ذخیره کم عمق ترین مسیر، کامل است.
  - بهینه بودن: در صورت وجود حافظه کافی برای ذخیره بهترین مسیر، بهینه است، در غیر اینصورت بهترین مسیر در محدوده حافظه موجود را بر می گرداند.
  - از تکرار محاسبات تا جائیکه حافظه اجازه دهد جلوگیری می کند.
  - می تواند از تمام حافظه مجاز استفاده کند.
  - بنابراین الگوریتم کارا-بهینه است در صورتیکه حافظه کافی برای ذخیره درخت جستجوی کامل در دسترس باشد

الگوریتم اصلاح تکراری

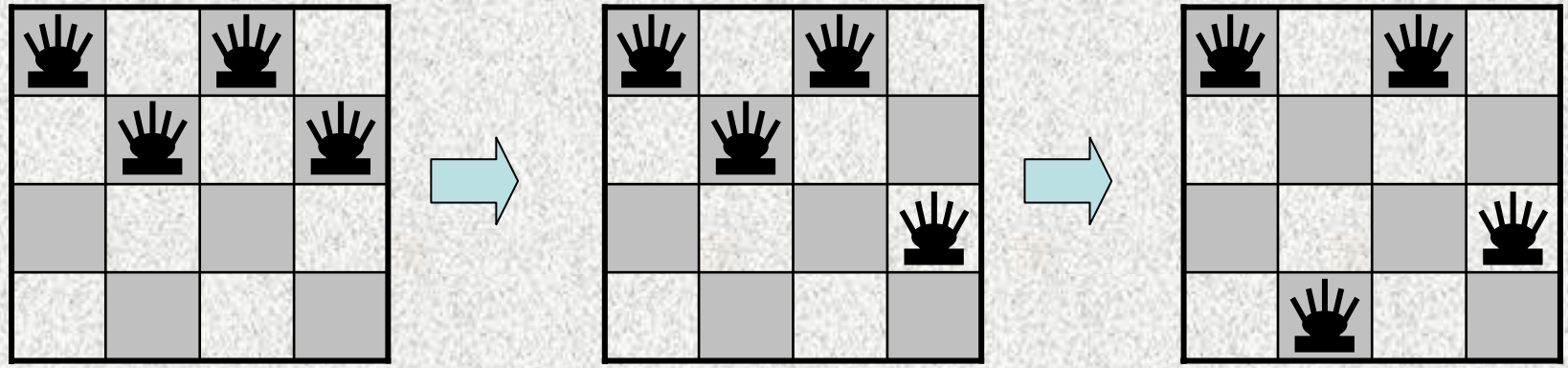
- در بسیاری از مسائل، مسیر طی شده برای رسیدن به هدف مهم نبوده و تنها رسیدن به جواب اهمیت دارد.
  - تا جائی که ممکن است پول بیشتری کسب کن.
- در چنین مسائلی الگوریتمهای اصلاح تکراری بهتر کار می کند.
- الگوریتمهای اصلاح تکراری یک مسیر راه حل را در نظر گرفته و سعی می کنند آنرا بهبود بخشند.

الگوریتم اصلاح تکراری



الگوریتم اصلاح تکراری

- مثال – مسئله n- وزیر : می خواهیم وزیر ها را طوری جا به جا کنیم که تعداد برخوردها کاهش یابد.



جستجوی تپه نوردی

- جستجوی تپه نوردی Hill climbing:

شامل حلقه ساده ای است که در آن، الگوریتم همواره در جهت رسیدن به نوک قله تلاش می کند.

- برای بررسی حرکت بعدی از یک تابع ارزیابی استفاده می کند که Value نامیده می شود.

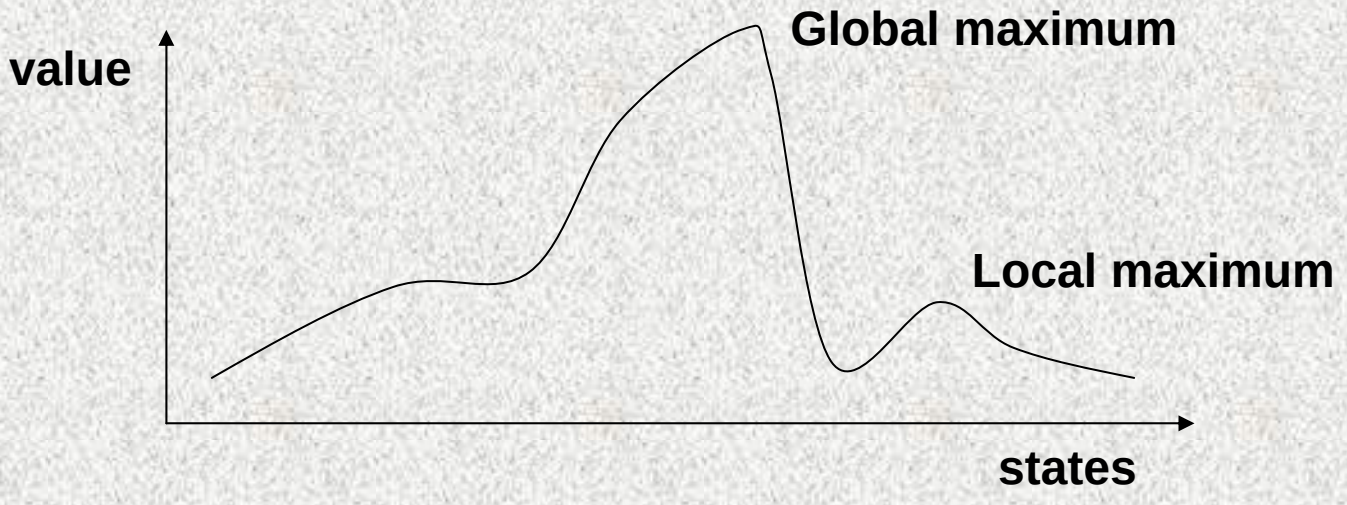
جستجوی تپه نوردی

```
function HILL-CLIMBING (problem)  returns a solution sequence
    input: problem, a problem
    local variables: current, a node
                     next, a node
    current ← MAKE-NODE (INITIAL_STATE [problem])
    loop do
        next ← a highest-valued successor of current
        if value [next] < value [current] then return current
        current ←  next
    end
```

جستجوی تپه نوردی

- در صورت بروز یکی از سه حالت زیر الگوریتم دچار مشکل می شود:
  - قرار گرفتن در یک ماکزیمم محلی که قله ای پائین تر از بلند ترین قله است.
  - قرار گرفتن در یک فلات که در آن تابع ارزیاب دارای مقادیر یکنواخت است.
  - حرکت در نزدیکی نوک قله که شیب ناگهان کمتر شده و پیشرفت به کندی انجام می گیرد.

در هر یک از موارد فوق، الگوریتم به نقطه ای میرسد که پیشرفت کند صورت می گیرد.



جستجوی تپه نوردی

- تنها راه حل مشکل فوق، شروع تصادفی از یک نقطه دیگر می باشد.

(Random Restart Hill Climbing)

به این ترتیب که یکسری جستجوهای تپه نوردی را با شروع تصادفی از یک نقطه دیگر، تا توقف آنها و یا رسیدن به نوک قله انجام می دهد و در نهایت بهترین آنها را انتخاب می کند.