

جست و آگاهانه و اکتشاف

- روش های هوشمند برای این طراحی شده اند تا به جای جستجوی تمامی گره ها، با هدف و جهت معین بخشی از گره ها را جستجو کرده و پاسخ را بین آنها بیابید. برای حصول این هدف دو کار باید صورت گیرد:

– تابعی معین کند که گره جاری، گره مناسبی برای رسیدن به جواب هست یا خیر (تابع کشف کننده)

– استراتژی جستجویی که قادر به استفاده از این تابع کشف کننده باشد.

جست و جوی آگاهانه و اکتشاف

متمدهای جستجوی آگاهانه

جستجوی محلی و بهینه سازی

تپه نوردی

شبیه سازی حرارت

پرتو محلی

الگوریتمهای ژنتیک

بهترین جستجو

مريضانه

A*

IDA*

RBFS

MA* و SMA*

جست و جوی آگاهانه و اکتشاف

تعاریف

↪ تابع هزینه مسیر، $g(n)$: هزینه مسیر از گره اولیه تا گره n

↪ تابع اکتشافی، $h(n)$: هزینه تخمینی ارزان ترین مسیر از گره n به گره هدف

↪ تابع ارزیابی، $f(n)$: هزینه تخمینی ارزان ترین مسیر از طریق n

$$f(n) = g(n) + h(n)$$

اگر n گره هدف باشد در نتیجه $h(n) = 0$ خواهد بود

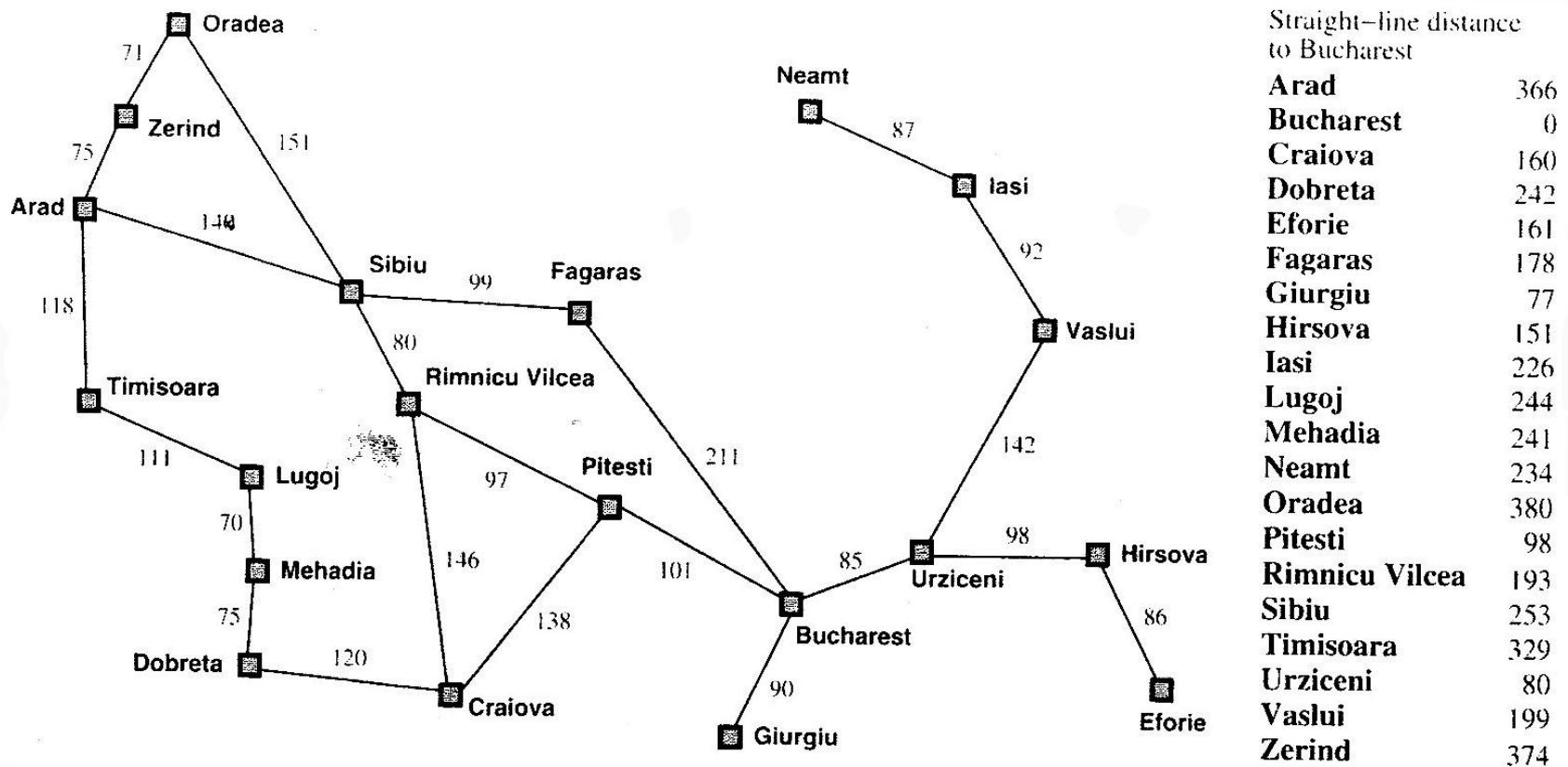
جستجوی اول- بهترین حریصانه

- جستجوی اول- بهترین حریصانه تلاش می کند گرهی را که نزدیک ترین به هدف است توسعه دهد، یعنی مشابه آنکه به سرعت به دنبال جواب برویم.
- بنابراین گره ها را تنها بر مبنای تابع اکتشافی ارزیابی می کند:

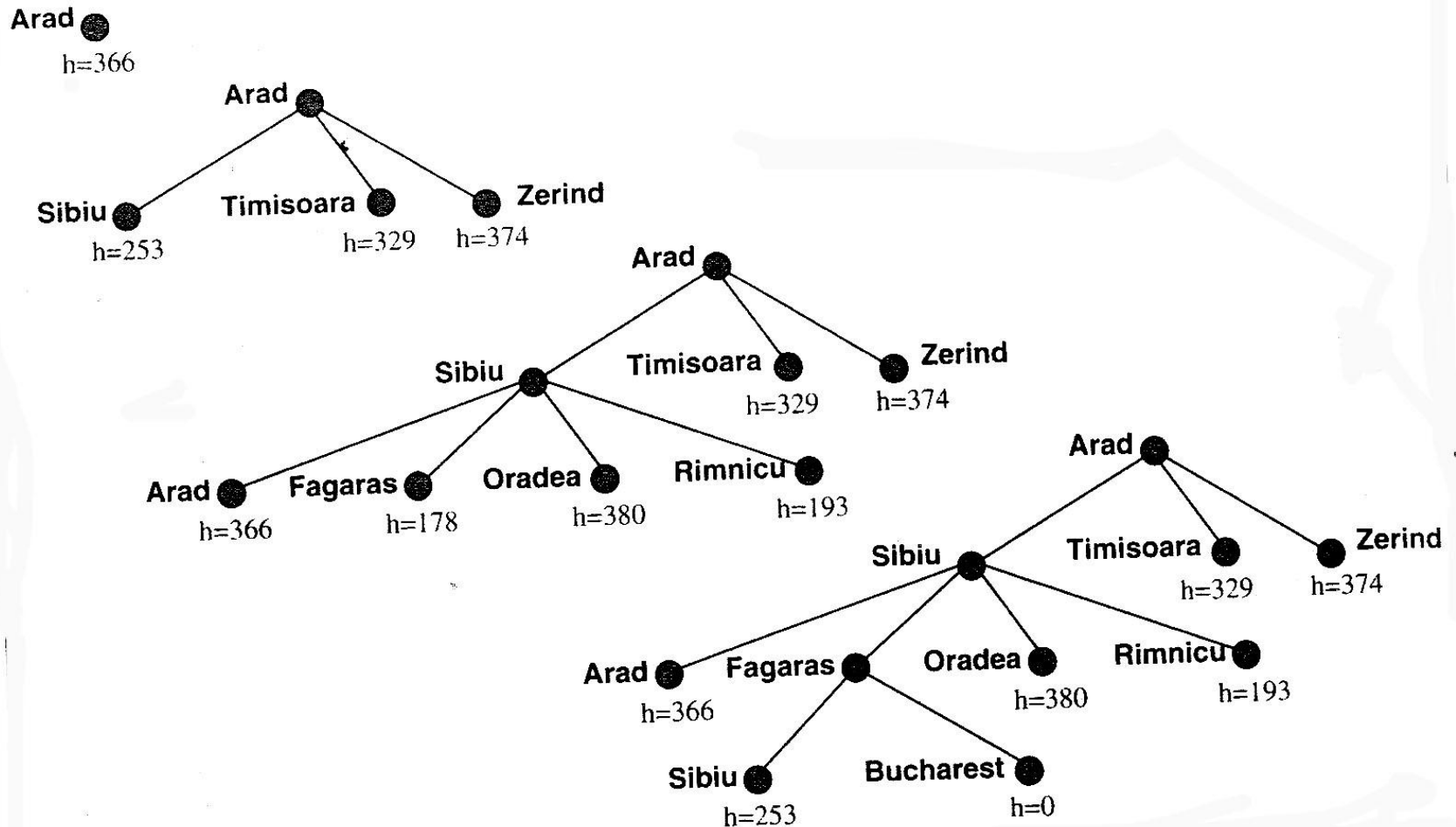
$$f(n) = h(n)$$

جستجوی اول- بهترین حریصانه

- برای یافتن مسیر در نقشهٔ رومانی از تابع اکتشافی خط مستقیم استفاده کرده ایم.



جستجوی اول- بهترین هزینه



جست و جوی آگاهانه و اکتشاف

جستجوی حریصانه

کامل بودن: خیر بهینگی: خیر

زیرا ممکن است در مسیر بی نهایتی بیفتد و هرگز دیگر گزینه ها را انتخاب نکند

پیچیدگی زمانی: $O(b^m)$

پیچیدگی فضا: $O(b^m)$

جستجوی A^* حداقل سازی هزینه پاسخ تخمین زده شده

- A^* گره ها را با ترکیب:

- $g(n)$ هزینه رسیدن به گره n

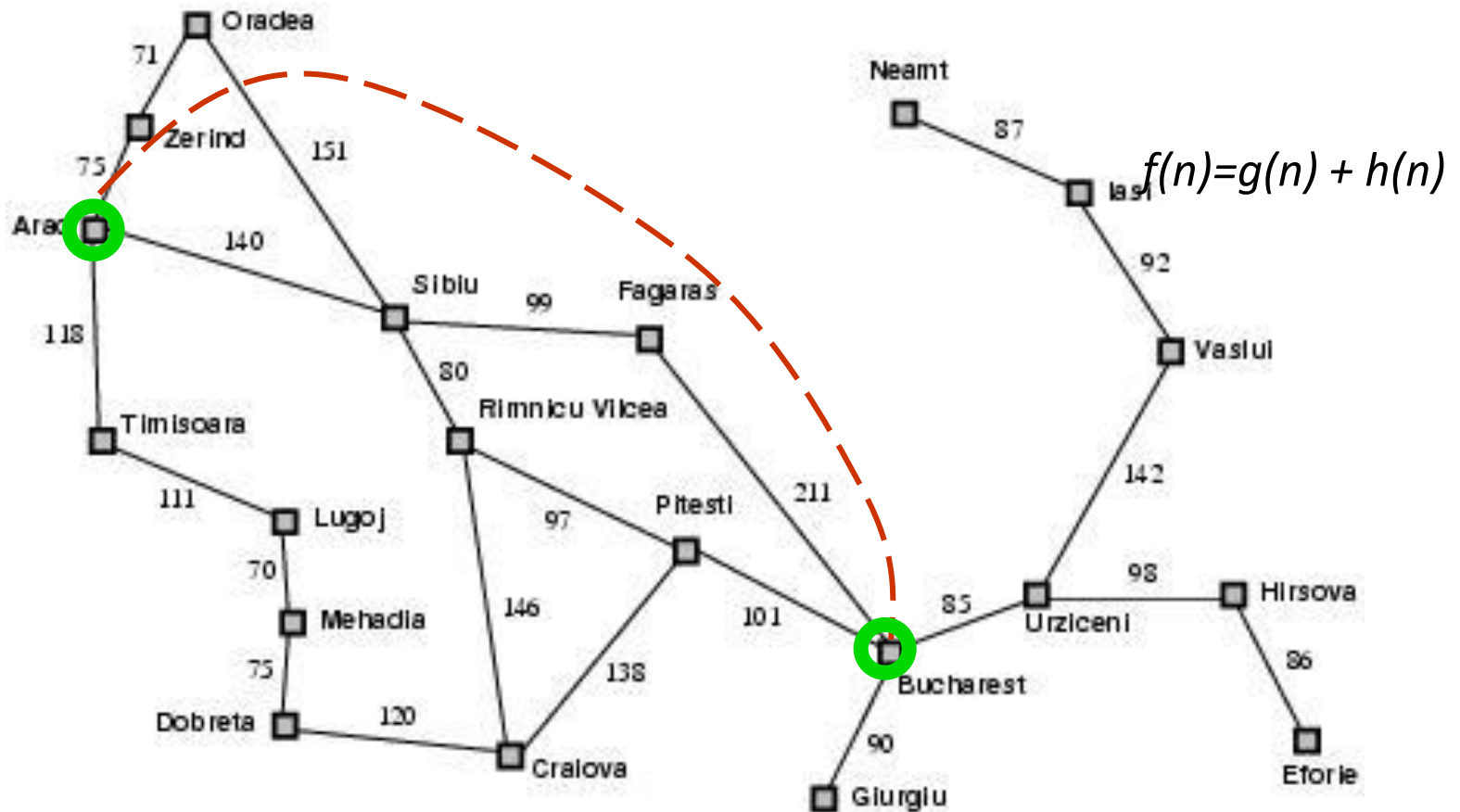
- $h(n)$ هزینه رفتن به هدف ارزیابی می کند:

$$f(n) = g(n) + h(n)$$

- $f(n)$ تخمین هزینه کوتاه ترین مسیر پاسخ با فرض عبور از n

- تابع اکتشافی قابل قبول (Admissible heuristic function):
توابعی هستند که هزینه حل مسئله را از مقدار واقعی آن کمتر در نظر می گیرند.

مثال از جستجوی A^*



جستجوی A^* در نقشه رومانی

(a) The initial state

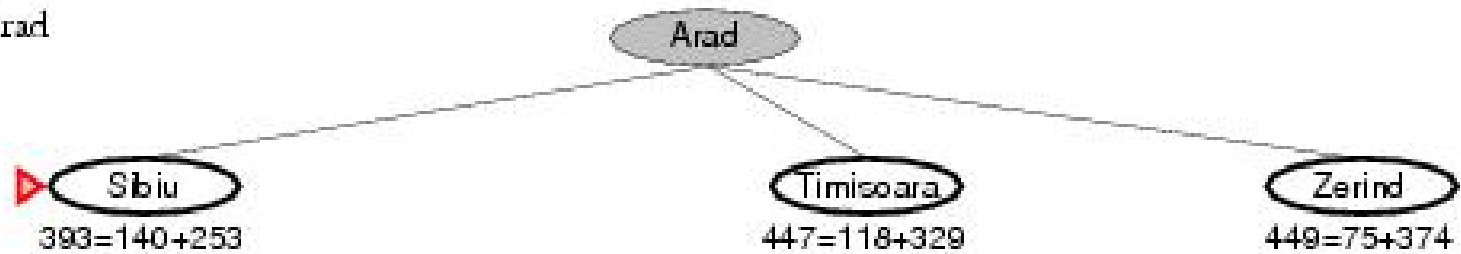


جستجوی Bucharest با شروع از Arad

$$f(\text{Arad}) = g(\text{Arad}) + h(\text{Arad}) = 0 + 366 = 366$$

جستجوی A^* در نقشه رومانی

After expanding Arad



Arad را باز کرده و $f(n)$ را برای هر یک از زیربرگها محاسبه میکنیم:

$$f(\text{Sibiu}) = c(\text{Arad}, \text{Sibiu}) + h(\text{Sibiu}) = 140 + 253 = 393$$

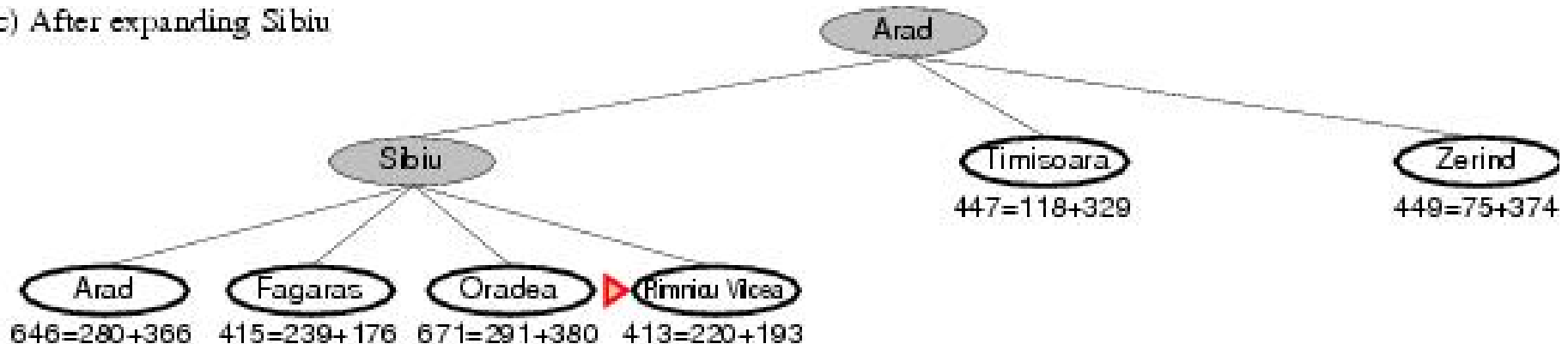
$$f(\text{Timisoara}) = c(\text{Arad}, \text{Timisoara}) + h(\text{Timisoara}) = 118 + 329 = 447$$

$$f(\text{Zerind}) = c(\text{Arad}, \text{Zerind}) + h(\text{Zerind}) = 75 + 374 = 449$$

بهترین انتخاب شهر Sibiu است

جستجوی A^* در نقشه رومانی

(c) After expanding Sibiu



Sibiu را باز کرده و $f(n)$ را برای هر یک از زیربرگها محاسبه میکنیم:

$$f(\text{Arad}) = c(\text{Sibiu}, \text{Arad}) + h(\text{Arad}) = 280 + 366 = 646$$

$$f(\text{Fagaras}) = c(\text{Sibiu}, \text{Fagaras}) + h(\text{Fagaras}) = 239 + 179 = 415$$

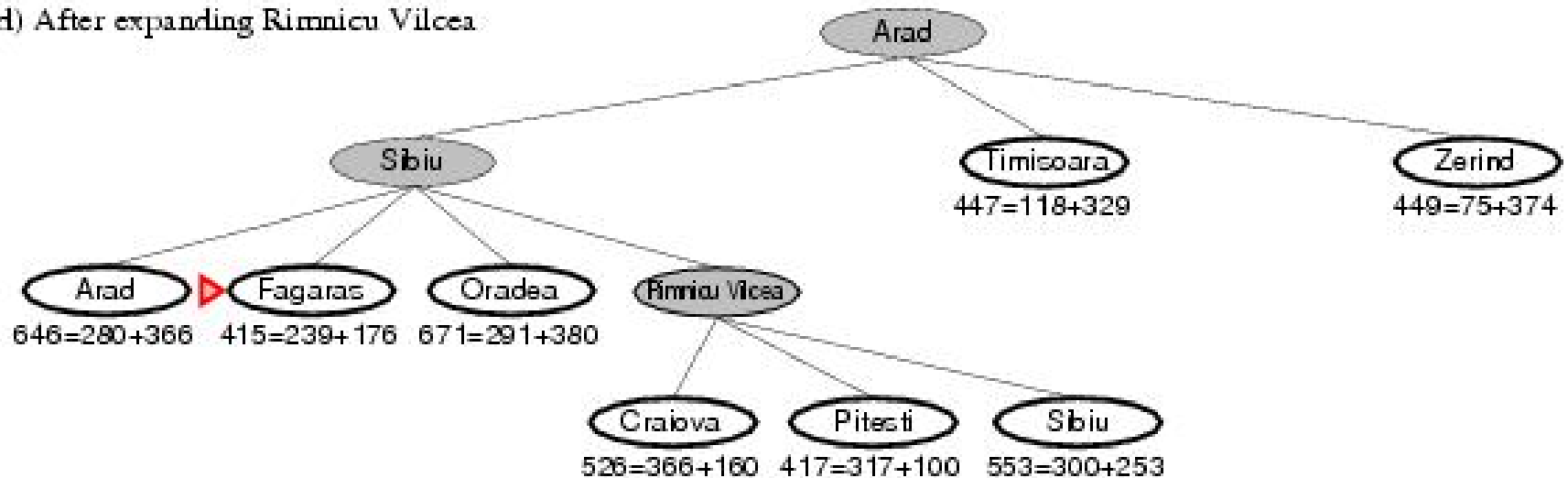
$$f(\text{Oradea}) = c(\text{Sibiu}, \text{Oradea}) + h(\text{Oradea}) = 291 + 380 = 671$$

$$f(\text{Rimnicu Vilcea}) = c(\text{Sibiu}, \text{Rimnicu Vilcea}) + h(\text{Rimnicu Vilcea}) = 220 + 192 = 413$$

بهترین انتخاب شهر Rimnicu Vilcea است

جستجوی A^* در نقشه رومانی

(d) After expanding Rimnicu Vilcea



Rimnicu Vilcea را باز کرده و $f(n)$ را برای هر یک از زیربرگها محاسبه میکنیم:

$$f(\text{Craiova}) = c(\text{Rimnicu Vilcea}, \text{Craiova}) + h(\text{Craiova}) = 360 + 160 = 526$$

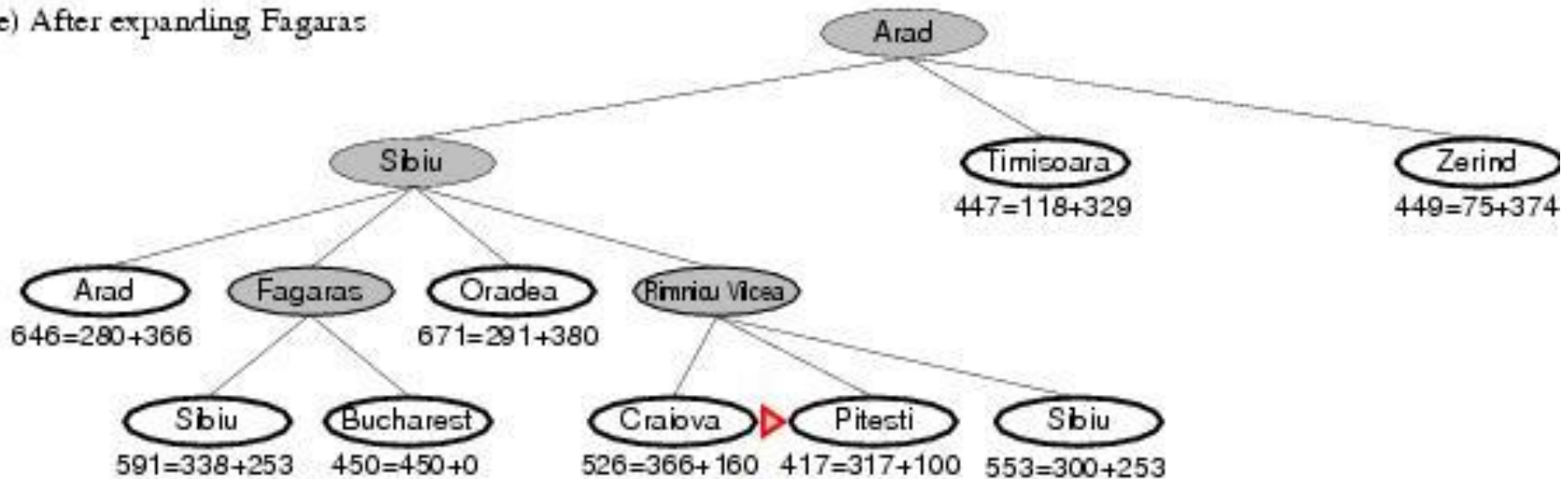
$$f(\text{Pitesti}) = c(\text{Rimnicu Vilcea}, \text{Pitesti}) + h(\text{Pitesti}) = 317 + 100 = 417$$

$$f(\text{Sibiu}) = c(\text{Rimnicu Vilcea}, \text{Sibiu}) + h(\text{Sibiu}) = 300 + 253 = 553$$

بهترین انتخاب شهر Fagaras است

جستجوی A^* در نقشه رومانی

(e) After expanding Fagaras



Fagaras را باز کرده و $f(n)$ را برای هر یک از زیربرگها محاسبه میکنیم:

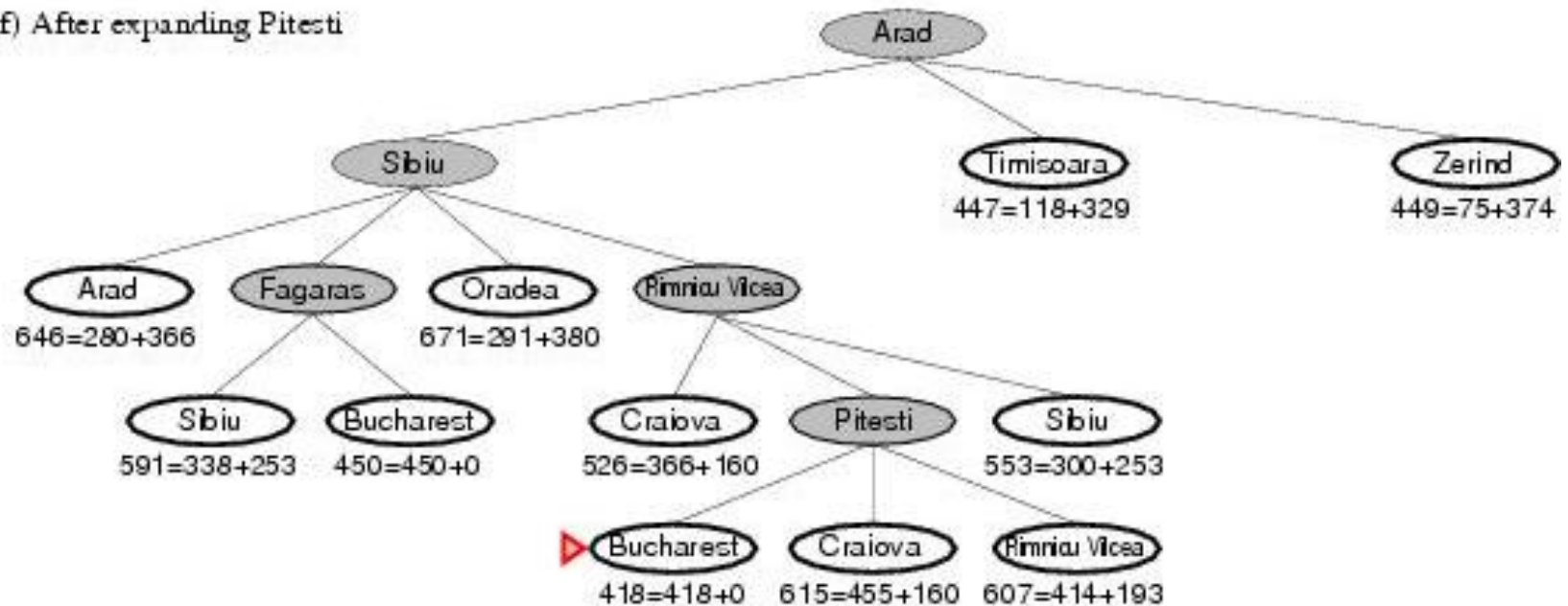
$$f(\text{Sibiu}) = c(\text{Fagaras}, \text{Sibiu}) + h(\text{Sibiu}) = 338 + 253 = 591$$

$$f(\text{Bucharest}) = c(\text{Fagaras}, \text{Bucharest}) + h(\text{Bucharest}) = 450 + 0 = 450$$

بهترین انتخاب شهر !!! Pitesti است

جستجوی A^* در نقشه رومانی

(f) After expanding Pitesti



Pitesti را باز کرده و $f(n)$ را برای هر یک از زیربرگها محاسبه میکنیم:

$$f(\text{Bucharest}) = c(\text{Pitesti}, \text{Bucharest}) + h(\text{Bucharest}) = 418 + 0 = 418$$

بهترین انتخاب شهر !!! Bucharest است

جستجوی A^*

کامل بودن: بله

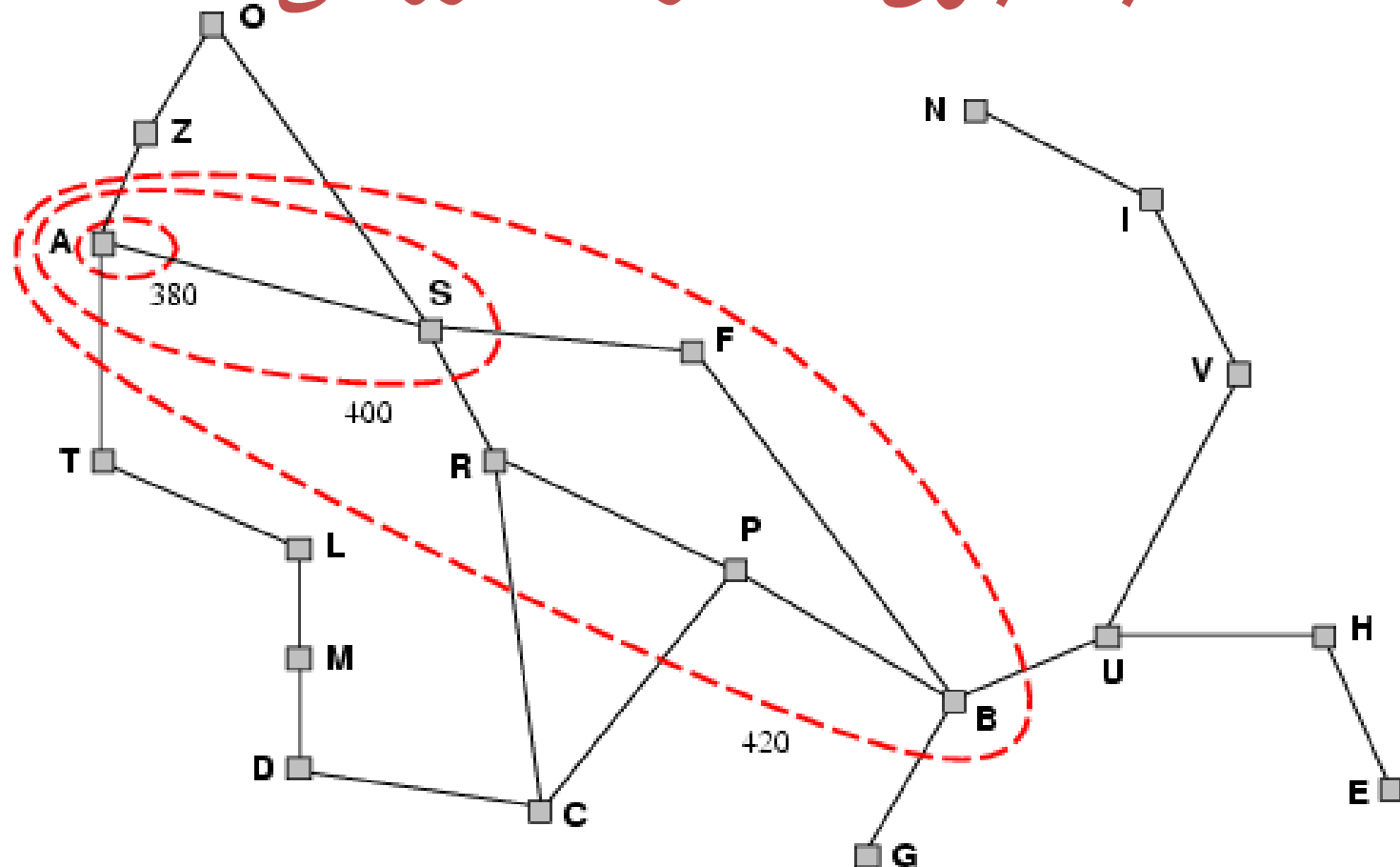
بهینگی: بله

پیچیدگی زمانی: $O(b^m)$

پیچیدگی فضا: $O(b^m)$

جست و جوی آگاهانه و اکتشاف

جستجوی A^* در نقشه رومانی



جستجوی اکتشافی با حافظه محدود IDA^*

ساده ترین راه برای کاهش حافظه مورد نیاز A^* استفاده از عمیق کننده تکرار در زمینه جست و جوی اکتشافی است.

الگوریتم عمیق کننده تکرار A^* $\longrightarrow IDA^*$

در جستجوی IDA^* مقدار برش مورد استفاده، عمق نیست بلکه هزینه $f\text{-cost}(g+h)$ است.

IDA^* برای اغلب مسئله های با هزینه های مرحله ای، مناسب است و از سر بار ناشی از نگهداری صف مرتبی از گره ها اجتناب میکند

بهترین جستجوی بازگشتی RBFS

ساختار آن شبیه جست و جوی عمقی بازگشتی است، اما به جای اینکه دائماً به طرف پایین مسیر حرکت کند، مقدار f مربوط به بهترین مسیر از هر جد گره فعلی را نگهداری میکند، اگر گره فعلی از این حد تجاوز کند، بازگشتی به عقب برمیگردد تا مسیر دیگری را انتخاب کند.

این جستجو اگر تابع اکتشافی قابل قبولی داشته باشد، بهینه است.

پیچیدگی فضایی آن $O(bd)$ است

تعیین پیچیدگی زمانی آن به دقت تابع اکتشافی و میزان تخییر بهترین مسیر در اثر بسط گره ها بستگی دارد.

بهترین جستجوی بازگشتی RBFS

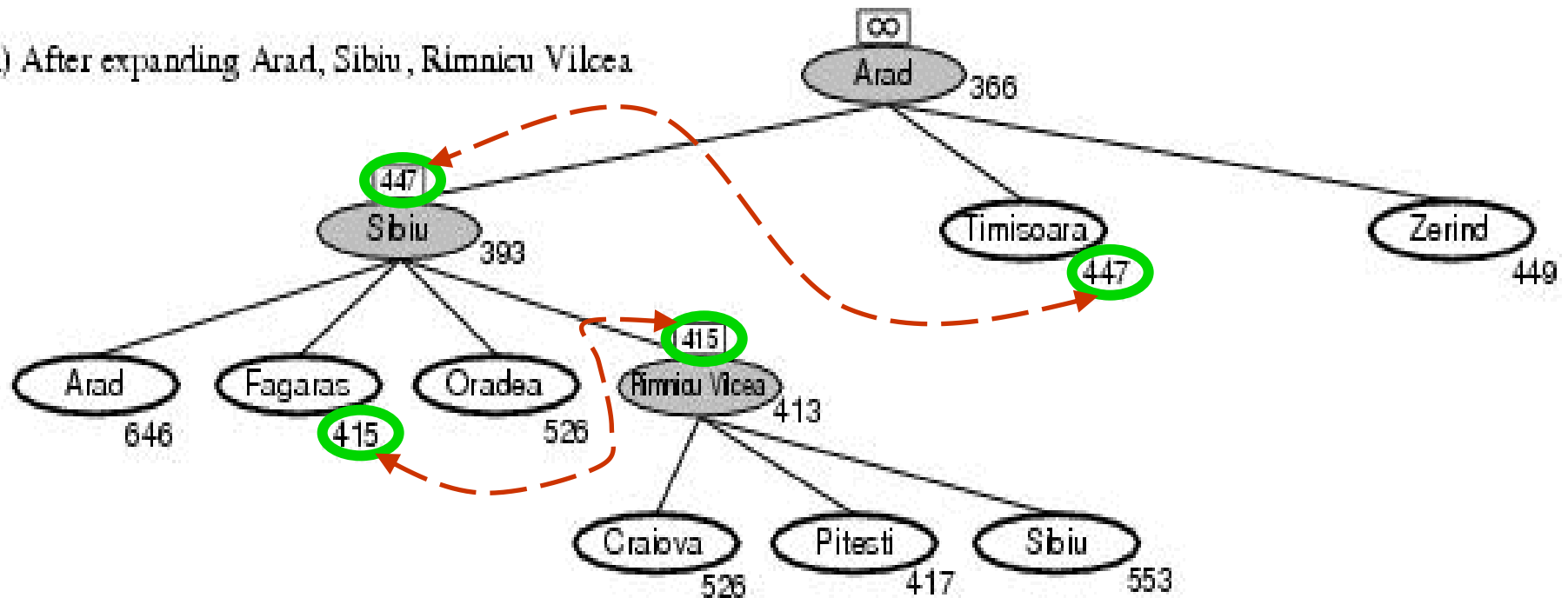
function RECURSIVE-BEST-FIRST-SEARCH(*problem*) **returns** a solution, or failure
RBFS(*problem*, MAKE-NODE(INITIAL-STATE[*problem*]), ∞)

function RBFS(*problem*, *node*, *f-limit*) **returns** a solution, or failure and a new *f*-cost limit
if GOAL-TEST[*problem*](STATE[*node*]) **then return** *node*
successors $\leftarrow$ EXPAND(*node*, *problem*)
if *successors* is empty **then return** failure, ∞
for each *s* **in** *successors* **do**
 $f[s] \leftarrow \max(g(s) + h(s), f[node])$
repeat
 best $\leftarrow$ the lowest *f*-value node in *successors*
 if $f[best] > f_limit$ **then return** failure, $f[best]$
 alternative $\leftarrow$ the second-lowest *f*-value among *successors*
 result, $f[best] \leftarrow$ RBFS(*problem*, *best*, $\min(f_limit, alternative)$)
 if *result* $\neq$ failure **then return** *result*

بهترین جستجوی بازگشتی RBFS

در نقشه رومانی

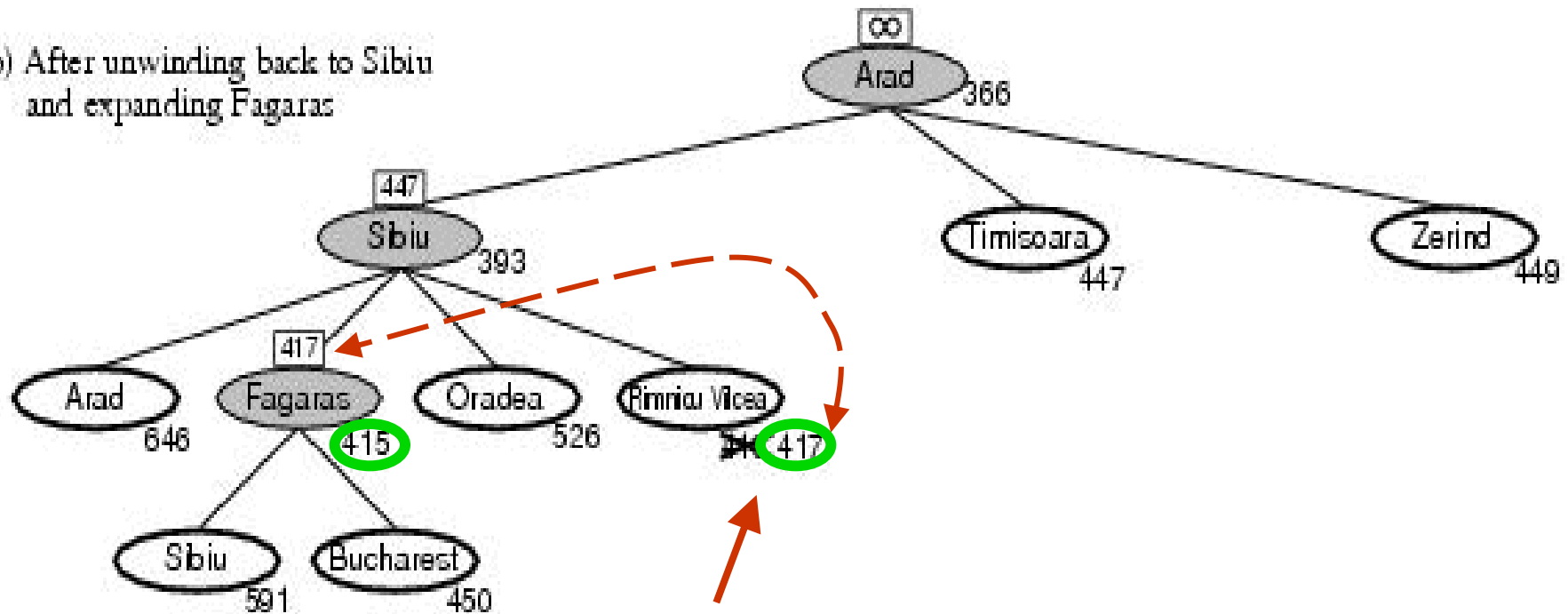
(a) After expanding Arad, Sibiu, Rimnicu Vilcea



بهترین جستجوی بازگشتی RBFS

در نقشه رومانی

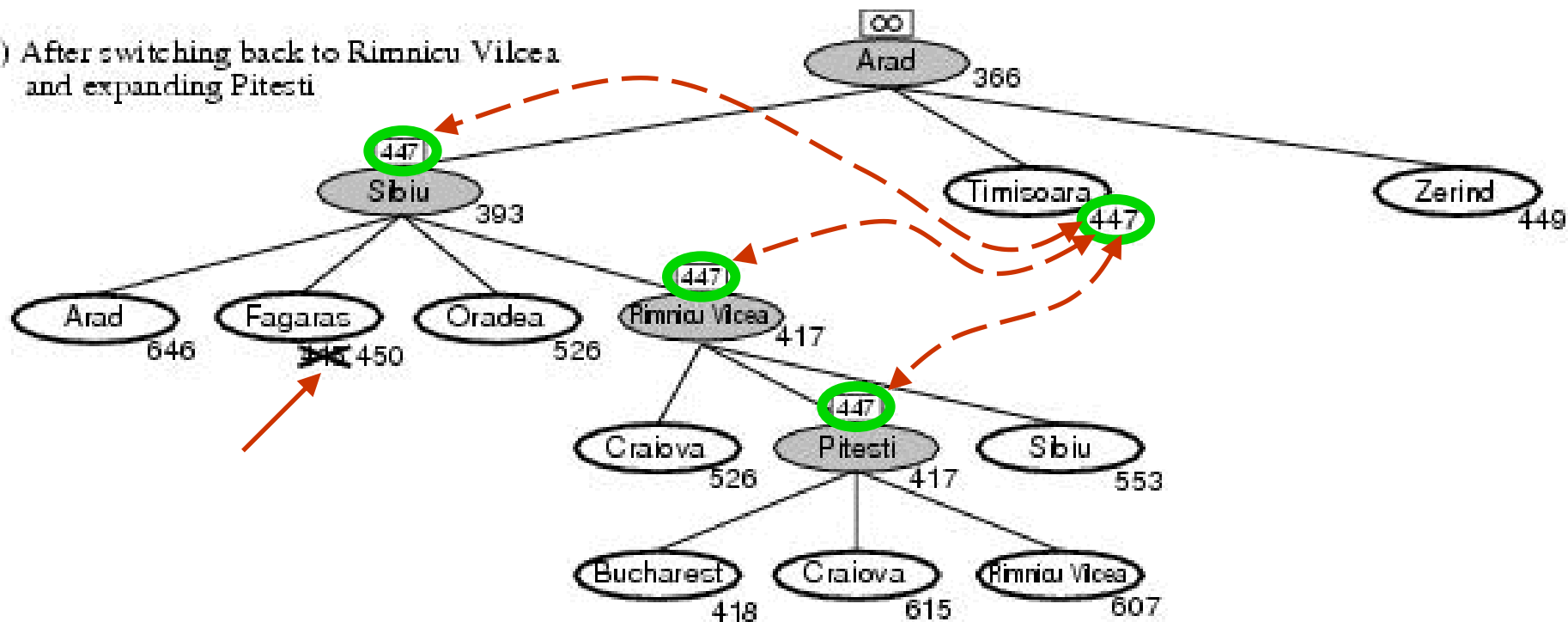
(b) After unwinding back to Sibiu and expanding Fagaras



بهترین جستجوی بازگشتی RBFS

در نقشه رومانی

(c) After switching back to Rimnicu Vilcea and expanding Pitesti



بهترین جستجوی بازگشتی RBFS

RBFS تا مدی از IDA^* کارآمدتر است، اما گره های زیادی تولید میکند.

IDA^* و RBFS در معرض افزایش توانی پیچیدگی قرار دارند که در جست و جوی گرافها مرسوم است، زیرا نمیتوانند حالت های تکراری را در غیر از مسیر فعلی بررسی کنند. لذا، ممکن است یک حالت را چندین بار بررسی کنند.

IDA^* و RBFS از فضای اندکی استفاده میکنند که به آنها آسیب میرساند. IDA^* بین هر تکرار فقط یک عدد را نگهداری میکند که مقدار فعلی هزینه f است. RBFS اطلاعات بیشتری در حافظه نگهداری میکند

جستجوی حافظه محدود ساده SMA^*

➡ SMA^* بهترین برگ را بسط میدهد تا حافظه پر شود. در این نقطه بدون از بین بردن گره های قبلی نمیتواند گره جدیدی اضافه کند

➡ SMA^* همیشه بدترین گره برگ را حذف میکند و سپس از طریق گره فراموش شده به والد آن برگ میگردد. پس جد زیر درخت فراموش شده، کیفیت بهترین مسیر را در آن زیر درخت میداند

➡ اگر عمق سطحی ترین گره هدف کمتر از حافظه باشد، کامل است.

➡ SMA^* بهترین الگوریتم همه منظوره برای یافتن حلهای بهینه میباشد

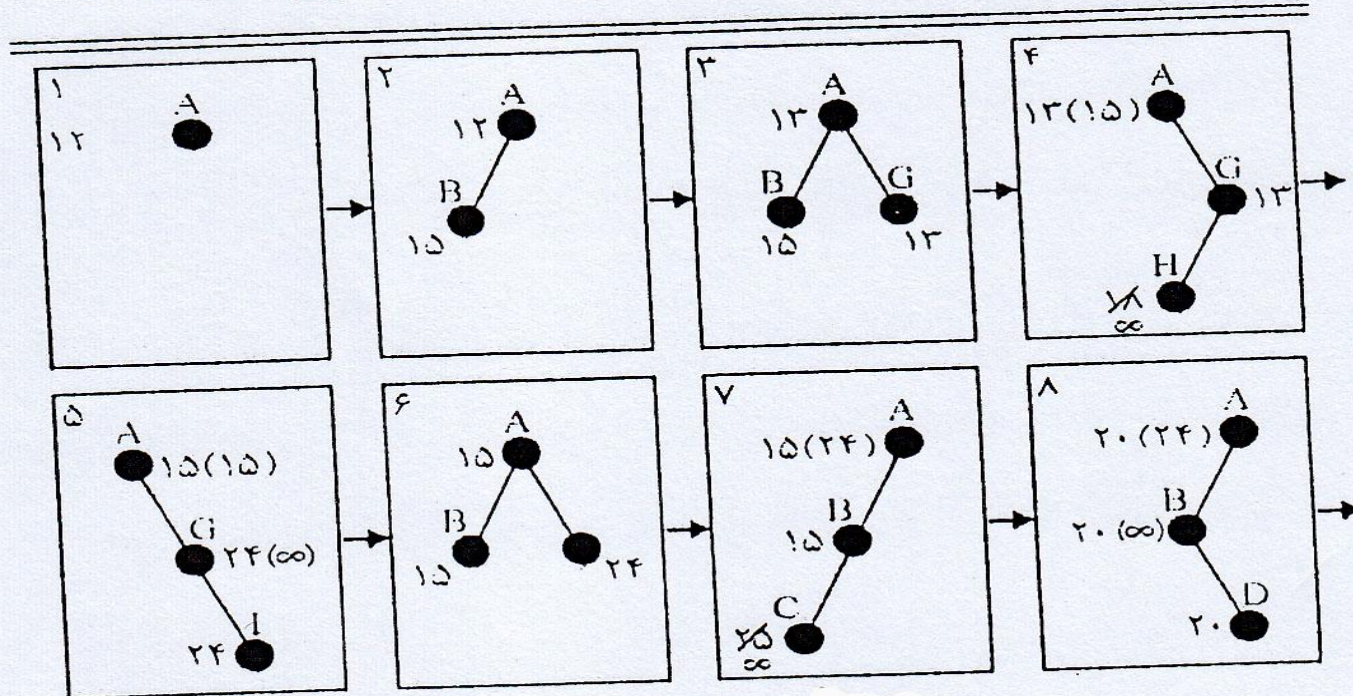
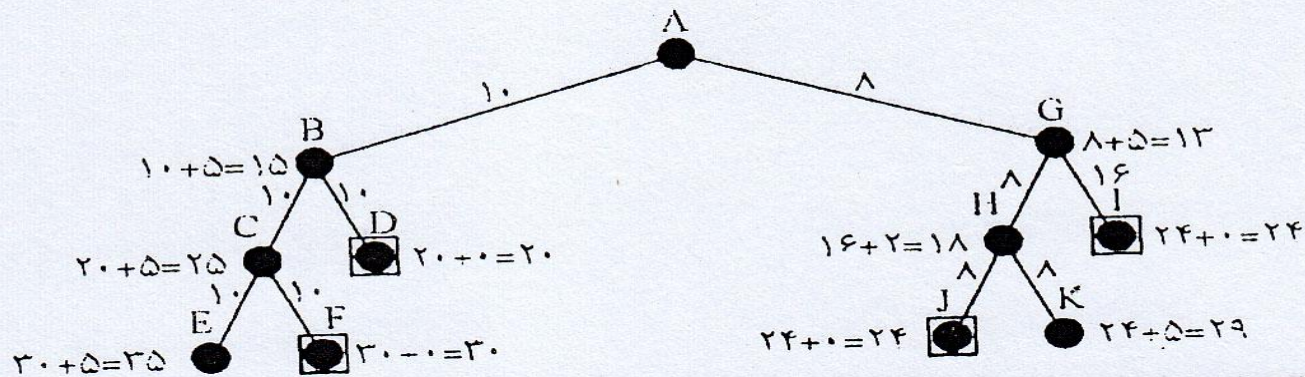
*SMA جستجوی حافظه محدود ساده

↗ اگر مقدار f تمام برگها یکسان باشد و الگوریتم یک گره را هم برای بسط و هم برای حذف انتخاب کند، SMA^* این مسئله را با بسط بهترین برگ جدید و حذف بهترین برگ قدیمی حل میکند

↗ ممکن است SMA^* مجبور شود دائماً بین مجموعه ای از مسیرهای حل کاندید تخییر موضع دهد، در حالی که بخش کوچکی از هر کدام در حافظه جا شود

↗ محدودیتهای حافظه ممکن است مسئله ها را از نظر زمان محاسباتی، غیر قابل حل کند.

جستجوی حافظه محدود ساده *SMA



توابع اکتشافی

7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

Goal State

مثال برای معمای ۸

میانگین هزینه حل تقریباً ۲۲ مرحله و فاکتور انشعاب در حدود ۳ است.

جست و جوی جامع تا عمق ۲۲: $3^{22} \approx 3.1 \times 10^{10}$

با انتخاب یک تابع اکتشافی مناسب میتوان مراحل جستجو را کاهش داد

دو روش اکتشافی متداول برای معمای ۸

7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

Goal State

تعداد کاشیها در مکانهای نادرست
در حالت شروع
 $h_1 = 8$

h_1 اکتشاف قابل قبولی است، زیرا هر
کاشی که در جای نامناسبی قرار دارد،
حداقل یکبار باید جابجا شود

دو روش اکتشافی متداول برای معمای ۸

7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

Goal State

مجموعه فواصل کاشیها از موقعیتهای هدف آنها h_2
در حالت شروع

$$h_2 = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18$$

چون کاشیها نمیتوانند در امتداد قطر جا به جا شوند، فاصله ای که محاسبه میکنیم مجموع فواصل افقی و عمودی است. این فاصله را فاصله بلوک شهر یا فاصله مانهاتان مینامند.

دو روش اکتشافی متداول برای معمای ۸

7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

Goal State

مجموعه فواصل کاشیها از موقعیتهای هدف آنها $h_2 =$
 h_2 قابل قبول است، زیرا هر جابجایی که میتواند
انجام گیرد، به اندازه یک مرحله به هدف نزدیک
میشود.

هیچ کدام از این برآوردها، هزینه واقعی راه حل نیست
هزینه واقعی ۳۶ است

جست و جوی آگاهانه و اکتشاف

اثر دقت اکتشاف بر کارایی

اگر برای هر گره n داشته باشیم: $h_2(n) \geq h_1(n)$

➤ h_2 بر h_1 غالب است

➤ غالب بودن مستقیماً به کارایی ترجمه میشود

➤ تعداد گره هایی که با بکارگیری h_2 بسط داده میشود، هرگز بیش از بکارگیری h_1 نیست

همیشه بهتر است از تابع اکتشافی با مقادیر بزرگ استفاده کرد، به شرطی که زمان محاسبه اکتشاف، خیلی بزرگ نباشد

مسئله راحت (relax)

- مسئله راحت مسئله ای است که تعدادی از محدودیت های عملگرهای آن حذف شده باشند.
- جواب دقیق یک مسئله راحت معمولاً یک تخمین برای مسئله اصلی است.
- مثال محمای ۸ : یک چهار خانه می تواند از خانه A به خانه B حرکت کند، اگر A همسایه B باشد و B یک خانه خالی باشد.
 - یک خانه مربع می تواند از خانه A به خانه B برود اگر B خالی باشد.
 - یک خانه مربع می تواند از خانه A به خانه B برود.
 - یک خانه مربع می تواند از خانه A به خانه B برود اگر مجاور همدیگر باشند.

الگوریتم‌های جست و جوی محلی و بهینه سازی

الگوریتم‌های قبلی، فضای جست و جو را به طور سیستماتیک بررسی میکنند

➤ تا رسیدن به هدف یک یا چند مسیر نگهداری میشوند

➤ مسیر رسیدن به هدف، راه حل مسئله را تشکیل میدهد

الگوریتم‌های محلی مسیر رسیدن به هدف مهم نیست

➤ مثال: مسئله ۸ وزیر

دو امتیاز عمده جست و جوی محلی

➤ استفاده از حافظه کمی

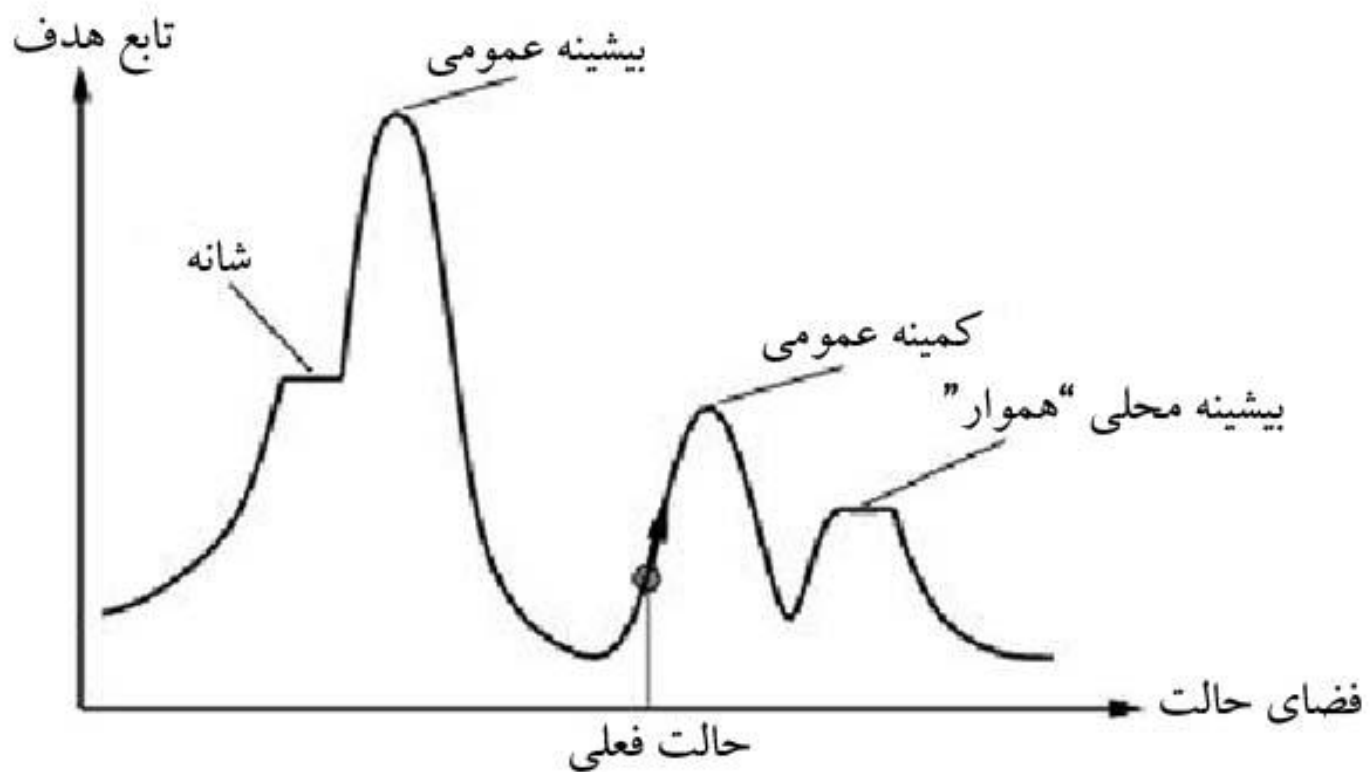
➤ ارائه راه‌های منطقی در فضاهای بزرگ و نامتناهی

این الگوریتمها برای حل مسائل بهینه سازی نیز مفیدند

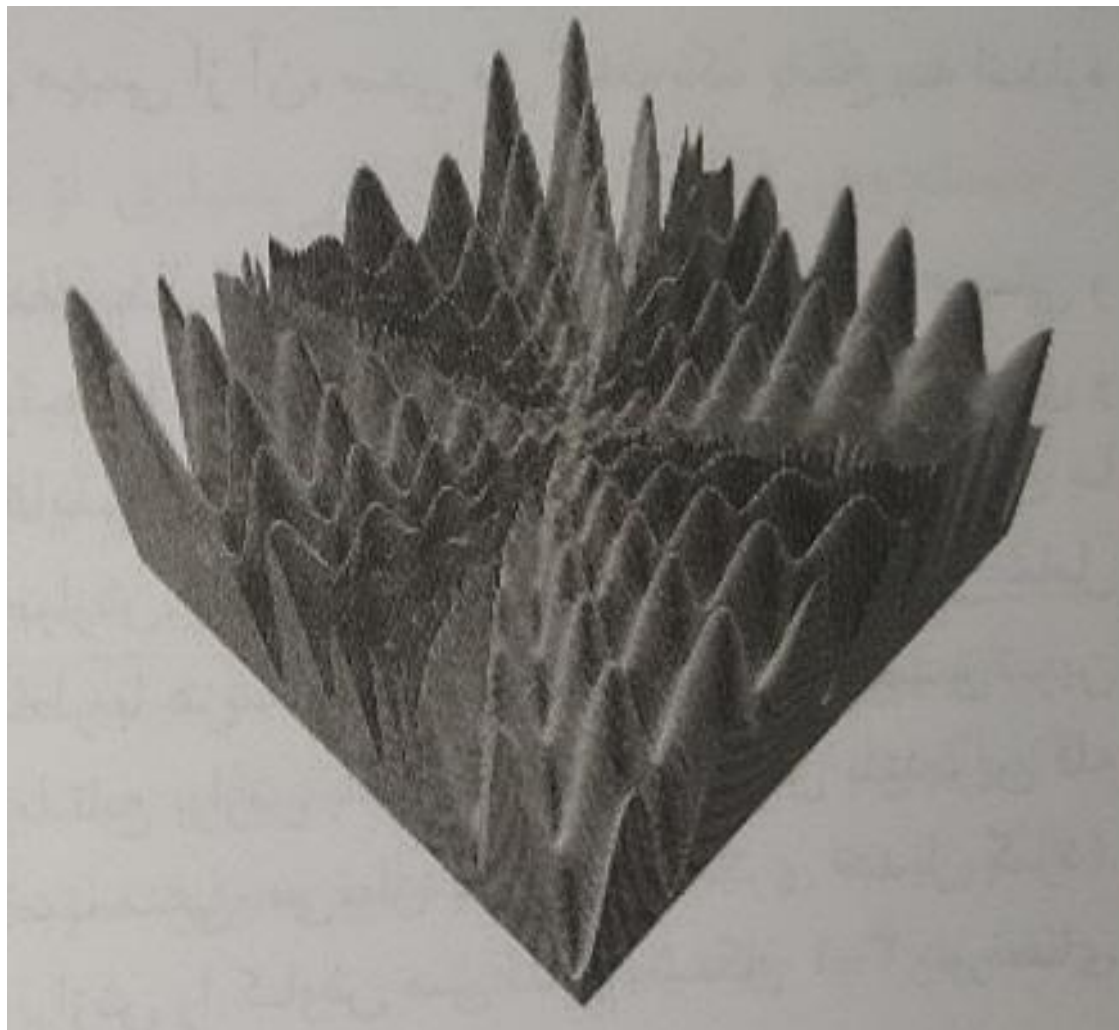
➤ یافتن بهترین حالت بر اساس تابع هدف

جست و جوی آگاهانه و اکتشاف

الگوریتم های جست و جوی محلی و بهینه سازی



فضای جستجوی دوبعدی



جستجوی تپه نوردی (Hill-Climbing)

- در جهت افزایش مقدار پیوسته حرکت می کند
- زمانی متوقف می شود که به نقطه اوجی برسد که همسایه بالاتری نداشته باشد.
- الگوریتم درخت جستجو نمی سازد،
 - بنابراین ساختار داده گره جاری تنها نیاز به ثبت حالت و مقدار تابع هدف دارد.
- تپه نوردی ماوراء همسایگان بلافاصل گره جاری را نگاه نمی کند.
 - این همانند آن است که بخواهیم بالای قله اورست را در حالی پیدا کنیم که قله در مه غلیظی فرو رفته و ما دچار فراموشی هستیم.

جستجوی تپه نوردی (Hill-Climbing)

function HILL-CLIMBING(*problem*) **returns** a state that is a local maximum

inputs: *problem*, a problem

local variables: *current*, a node
neighbor, a node

current $\leftarrow$ MAKE-NODE(INITIAL-STATE[*problem*])

loop do

neighbor $\leftarrow$ a highest-valued successor of *current*

if VALUE[*neighbor*] $\leq$ VALUE[*current*] **then return** STATE[*current*]

current $\leftarrow$ *neighbor*

جستجوی تپه نوردی (Hill-Climbing)

مثال: مسئله ۸ وزیر

↗ مسئله ۸ وزیر با استفاده از فرمولبندی حالت کامل

↗ در هر حالت ۸ وزیر در صفحه قرار دارند

↗ تابع جانشین: انتقال یک وزیر به مربع دیگر در همان ستون

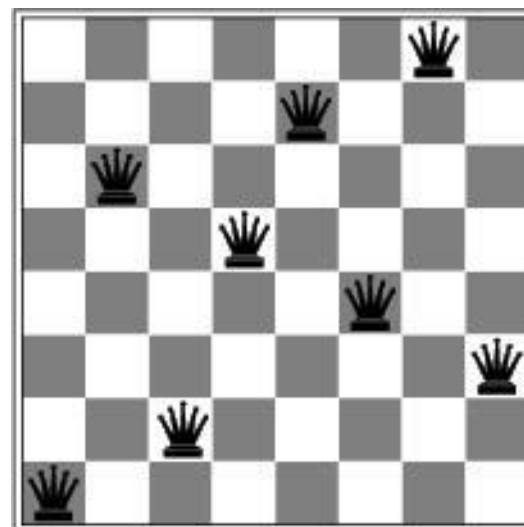
↗ تابع اکتشاف: جفت وزیرهایی که نسبت به هم گارد دارند

جستجوی تپه نوردی (Hill-Climbing)

الف

18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14	♔	13	16	13	16
♔	14	17	15	♔	14	16	16
17	♔	16	18	15	♔	15	♔
18	14	♔	15	15	14	♔	16
14	14	13	17	12	14	12	18

ب



الف- حالت با هزینه $h=17$ که مقدار h را برای هر جانشین نشان میدهد

ب- کمینه محلی در فضای حالت ۸ وزیر؛ $h=1$

جستجوی تپه نوردی (Hill-Climbing)

ملقه ای که در جهت افزایش مقدار حرکت میکند (بطرف بالای تپه)
رسیدن به بلندترین قله در همسایگی حالت فعلی، شرط خاتمه است.

ساختمان داده گره فعلی، فقط حالت و مقدار تابع هدف را نگه میدارد

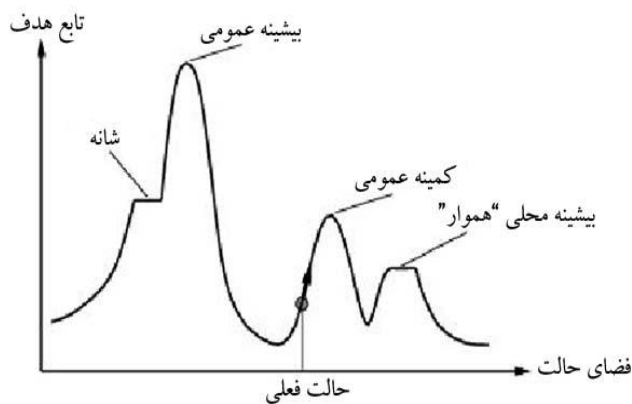
جست و جوی محلی مریضانه نیز نام دارد
بدون فکر قبلی حالت همسایه فوبی را انتخاب میکند

تپه نوردی به دلایل زیر میتواند متوقف شود:

➤ بیشینه محلی

➤ شانه

➤ فلات



Iterative Improvement 3

Simple Iterative Improvement or Hill Climbing:

- Candidate is always and only accepted if cost is lower (or fitness is higher) than current configuration
- Stop when no neighbor with lower cost (higher fitness) can be found

Disadvantages:

- Local optimum as best result
- Local optimum depends on initial configuration
- Generally, no upper bound can be established on the number of iterations

How to cope with disadvantages

1. **Repeat** algorithm many times with **different initial configurations**
2. Use information **gathered in previous runs**
3. Use a more **complex Generation Function** to jump out of local optimum
4. Use a more complex **Evaluation Criterion** that accepts sometimes (randomly) also solutions away from the (local) optimum

جستجوی بازپخت شبیه سازی شده (Simulated Annealing)

تپه نوردی مرکب با حرکت تصادفی

شبیه سازی حرارت: حرارت با درجه بالا و به تدریج سرد کردن

مقایسه با حرکت توپ

توپ در فرود از تپه به عمیق ترین شکاف میرود

با تکان دادن سطح توپ از بیشینه محلی خارج میشود

با تکان شدید شروع (دمای زیاد)

بتدریج تکان کاهش (به دمای پایین تر)

با کاهش زمانبندی دما به تدریج، الگوریتم یک بهینه عمومی را می یابد

جستجوی بازپخت شبیه سازی شده (Simulated Annealing)

- به جای انتخاب بهترین حرکت، حرکتی تصادفی انتخاب خواهد شد.
- اگر حرکت، وضعیت را بهبود دهد همواره پذیرفته خواهد شد.
- در غیر این صورت، الگوریتم حرکت را با احتمالی کمتر از ۱ قبول خواهد کرد.
- این احتمال به صورت نمایی با "بد بودن حرکت" کاهش می یابد
- مقدار ΔE که با آن ارزیابی بدتر شده است.

$$\Delta E$$

$$\Delta E = Value[next] - Value[current]$$

$$P = \frac{1}{\frac{-\Delta E}{T}}$$

Simulation of cooling (Metropolis 1953)

- At a fixed temperature T :
- Perturb (randomly) the current state to a new state
- ΔE is the difference in energy between current and new state
- If $\Delta E < 0$ (new state is lower), accept new state as current state
- If $\Delta E \geq 0$, accept new state with probability
$$Pr(\text{accepted}) = \exp(-\Delta E / k_B.T)$$
- Eventually the systems evolves into thermal equilibrium at temperature T ; then the formula mentioned before holds
- When equilibrium is reached, temperature T can be lowered and the process can be repeated

Control Parameters

1. How to define equilibrium?
2. How to calculate new temperature for next step?

1. Definition of equilibrium

1. Definition is reached when we cannot yield any significant improvement after certain number of loops
2. A constant number of loops is assumed to reach the equilibrium

2. Annealing schedule (i.e. How to reduce the temperature)

1. A constant value is subtracted to get new temperature, $T' = T - T_d$
2. A constant scale factor is used to get new temperature, $T' = T * R_d$
 - A scale factor usually can achieve better performance

Example of Simulated Annealing

- **Traveling Salesman Problem (TSP)**
 - Given 6 cities and the traveling cost between any two cities
 - A salesman need to start from city 1 and travel all other cities then back to city 1
 - Minimize the total traveling cost

Example: SA for traveling salesman

- Solution representation
 - An integer list, i.e., (1,4,2,3,6,5)
- Search mechanism
 - Swap any two integers (except for the first one)
 - (1,4,2,3,6,5) $\rightarrow$ (1,4,3,2,6,5)
- Cost function

salesman

- Temperature

1. Initial temperature determination

1. Initial temperature is set at such value that there is around 80% acceptance rate for “bad move”
2. Determine acceptable value for $(C_{\text{new}} - C_{\text{old}})$

2. Final temperature determination

- Stop criteria
- Solution space coverage rate

- Annealing **schedule** (i.e. How to reduce the temperature)

- A constant value is subtracted to get new temperature, $T' = T - T_d$
- For instance new value is 90% of previous value.
- Depending on solution space coverage rate

Homogeneous Algorithm of Simulated Annealing

initialize;

REPEAT

REPEAT

perturb (config.i $\rightarrow$ config.j, ΔC_{ij});

IF $\Delta C_{ij} < 0$ THEN accept

ELSE IF $\exp(-\Delta C_{ij}/c) > \text{random}[0,1)$ THEN accept;

IF accept THEN update(config.j);

UNTIL *equilibrium is approached sufficient closely;*

c := next_lower(c);

UNTIL *system is frozen or stop criterion is reached*

In homogeneous algorithm the value of c is kept constant in the inner loop and is only decreased in the outer loop

جست و جوی پرتو محلی

➤ به جای یک حالت، k حالت را نگهداری میکند

➤ حالت اولیه: k حالت تصادفی

➤ گام بعد: جانشین همه k حالت تولید میشود

➤ اگر یکی از جانشین ها هدف بود، تمام میشود

➤ وگرنه بهترین جانشین را انتخاب کرده، تکرار میکند

➤ تفاوت عمده با جستجوی شروع مجدد تصادفی

➤ در جست و جوی شروع مجدد تصادفی، هر فرایند مستقل از بقیه اجرا میشود

➤ در جست و جوی پرتو محلی، اطلاعات مفیدی بین k فرایندهای موازی مبادله میشود

➤ جست و جوی پرتو غیرقطعی

➤ به جای انتخاب بهترین k از جانشینها، k جانشین تصادفی را بطوریکه احتمال انتخاب یکی

تابعی صعودی از مقدارش باشد، انتخاب میکند