روش های جستجوی آگاهانه

فصل چهارم
سید ناصر رضوی
Email:razavi@Comp.iust.ac.ir
۱۳۸۶

مقدمه

- جستجوی اول-بهترین (Best-First Search)
  - جستجوی حریصانه (Greedy search)
  - جستجوی A* و خصوصیات آن
- جستجوی عمیق کننده تکراری A* و SMA*
- جستجوی اول بهترین بازگشتی (RBFA*)
- هیوریستیک ها (Heuristics)
- الگوریتم های جستجوی محلی
  - جستجوی تپه نوردی (Hill Climbing)
  - جستجوی Simulated Annealing
  - الگوریتم های ژنتیک

مشکلات روش های جستجوی ناآگاهانه

- معیار انتخاب گره بعدی برای گسترش دادن تنها به شماره سطح آن بستگی دارد.
- از ساختار مسأله بهره نمی برند.
- درخت جستجو را به یک روش از پیش تعریف شده گسترش می دهند. یعنی قابلیت تطبیق پذیری با آنچه که تا کنون در مسیر جستجو دریافته اند و نیز حرکتی که می تواند خوب باشد ندارند.

مرور: جستجوی درخت

function GENERAL-SEARCH(*problem*, *strategy*) **returns** a solution , or failure

initialize the search tree using the initial state of *problem*

loop do

if there are no candidates for expansion **then return** failure

choose a leaf node for expansion according to *strategy*

if the node contains the goal state **then return** the corresponding solution

else expand the node and add the resulting node to the search tree

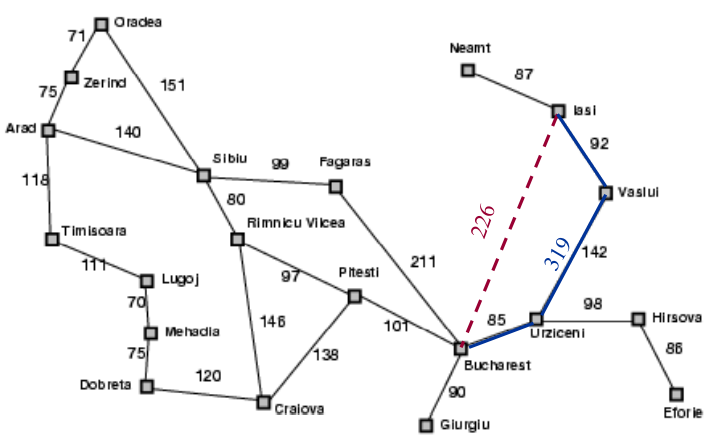
end

- استراتژی توسط ترتیب گسترش یافتن گره ها تعریف می شود.

جستجوی اول- بهترین

- ایده: برای هر گره از یک تابع ارزیابی (evaluation function) استفاده کن.
  - تخمین "میزان مطلوب بودن" گره
  - ←هربار مطلوب ترین گره گسترش نیافته را بسط می دهد.
- پیاده سازی:
  fringe یک صف می باشد که بر اساس میزان مطلوب بودن گره ها مرتب می باشد.
- موارد خاص:
  - جستجوی حریصانه (Greedy search)
  - جستجوی A*

نقشه رومانی به همراه هزینه مراحل برحسب *km*

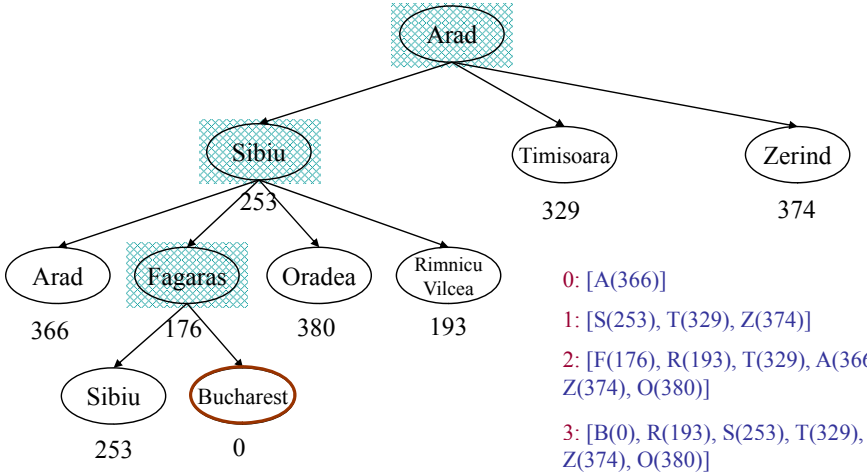


Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

جستجوی اول-بهترین حریصانه

- تابع ارزیابی $f(n) = h(n)$ (هیوریستیک)
- = هزینه تخمینی از گره n تا نزدیکترین هدف
- مثال: $h_{SLD}(n)$ = فاصله مستقیم از n تا بخارست.
- جستجوی حریصانه گره ای را گسترش می دهد که به نظر می رسد نزدیکترین گره به هدف (بخارست) باشد.

مثال جستجوی حریصانه



0: [A(366)]

1: [S(253), T(329), Z(374)]

2: [F(176), R(193), T(329), A(366), Z(374), O(380)]

3: [B(0), R(193), S(253), T(329), A(366), Z(374), O(380)]

4: [R(193), S(253), T(329), A(366), Z(374), O(380)]

خواص جستجوی حریصانه

- **کامل؟** خیر، ممکن است در حلقه گیر کند.
  - مثال: حالت اولیه Iasi، حالت هدف Fagaras

Iasi → Neamt → Iasi → Neamt → …

- **پیچیدگی زمانی؟** $O(b^m)$، اما با یک هیوریستیک خوب می تواند بسیار بهبود یابد.
- **پیچیدگی حافظه؟** $O(b^m)$، تمام گره ها را در حافظه نگه می دارد.
- **بهینه؟** خیر، (با توجه به مثال قبل)

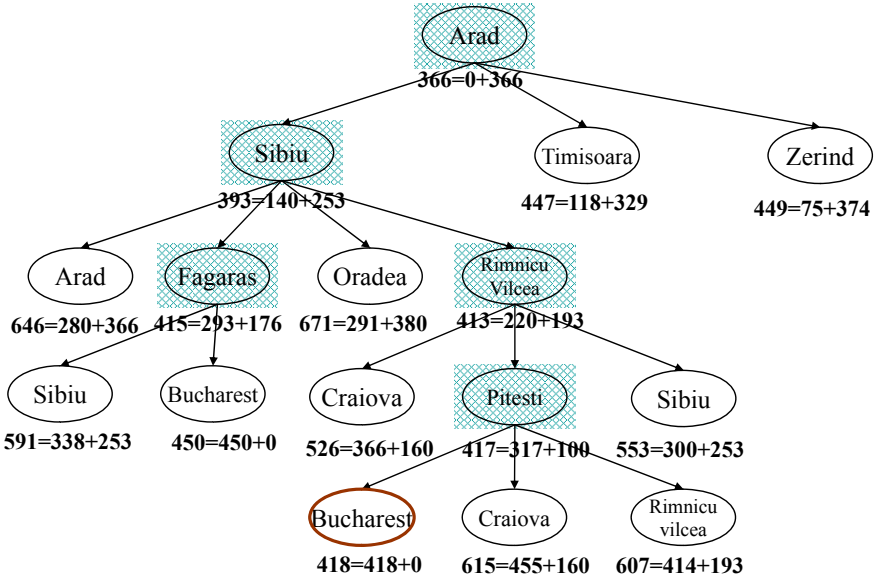
جستجوی A*

- ایده: از گسترش مسیرهایی که تاکنون مشخص شده پرهزینه می باشند، اجتناب کن.
- A*: ترکیب مزایای UCS و جستجوی حریصانه:
  - جستجوی حریصانه $h(n)$ را حداقل می کند، نه کامل و نه بهینه
  - جستجوی UCS، هزینه مسیر را حداقل می کند؛ کامل و بهینه است؛ می تواند بسیار زمانبر باشد.
- تابع ارزیابی

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

- $g(n)$: هزینه مسیر پیموده شده تا n.
- $h(n)$: هزینه تخمینی ارزانترین مسیر راه حل از n تا **هدف**.
- $f(n)$: هزینه تخمینی ارزانترین راه حل که از n می گذرد.

مثال جستجوی A*



جستجوی A*

A* از یک هیوریستیک قابل قبول (admissible) استفاده می کند،
یعنی، همواره $h(n) \le h^*(n)$ که در آن $h^*(n)$ هزینه واقعی n تا هدف می باشد.

هم چنین داریم
$h(n) \ge 0$ و
$h(G) = 0$ (G یک هدف می باشد).

به طور کلی :

$$0 \le h(n) \le h^*(n)$$

مثال: در هیوریستیک $h_{SLD}(n)$ هیچگاه هزینه تخمینی بیشتر از هزینه واقعی نخواهد بود. (کوتاهترین فاصله بین دو نقطه، فاصله مستقیم آن دو نقطه می باشد).

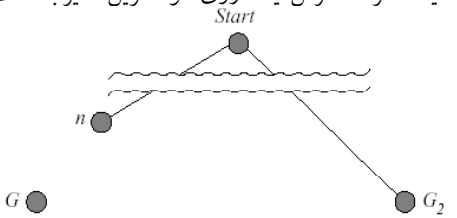
هیوریستیک قابل قبول

- یک هیوریستیک مانند $h(n)$ قابل قبول است اگر برای هر گره n داشته باشیم: $h(n) \le h^*(n)$ که $h^*(n)$ هزینه واقعی برای رسیدن به هدف از گره n می باشد.
- یک هیوریستیک قابل قبول هرگز هزینه رسیدن به هدف را بیش از حد تخمین نمی زند، یعنی خوش بینانه است.
- مثال: هیوریستیک $h_{SLD}(n)$ (هیچگاه فاصله واقعی را بیش از حد تخمین نمی زند).
- قضیه: اگر $h(n)$ قابل قبول باشد، A* با استفاده از TREE-SEARCH بهینه است.

بهینگی A* (اثبات)

- فرض کنید یک جواب زیر بهینه مانند G_2 تولید شده و در صف قرار دارد.

هم چنین فرض کنید n یک گره گسترش نیافته روی کوتاهترین مسیر به هدف بهینه G باشد(؟؟)



$f(G_2) = g(G_2)$ چون، $h(G_2) = 0$

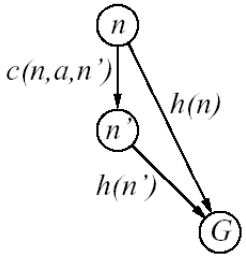
$> g(G)$ چون، G_2 زیر بهینه است

$\ge f(n)$ چون، h قابل قبول است

چون $f(G_2) \ge f(n)$، الگوریتم A^* هیچ وقت G_2 را برای گسترش یافتن انتخاب نمی کند.

اثبات لم: سازگاری (Consistency)

- یک هیوریستیک سازگار است اگر : $h(n) \le c(n, a, n') + h(n')$
- اگر h سازگار باشد، داریم:

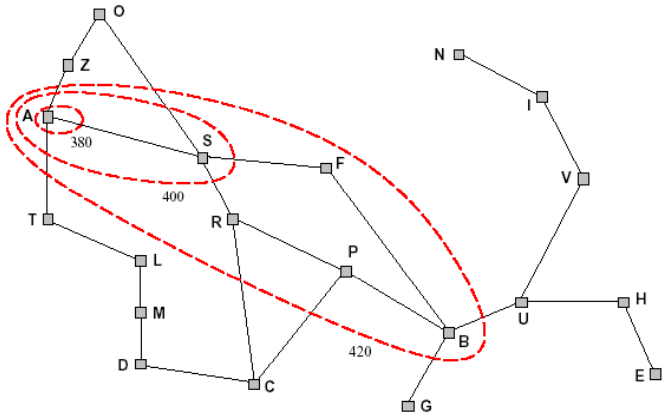


$$\begin{aligned} f(n') &= g(n') + h(n') \\ &= g(n) + c(n, a, n') + h(n') \\ &\ge g(n) + h(n) \\ &\ge f(n) \end{aligned}$$

- یعنی، $f(n)$ در طول هر مسیری غیر کاهشی می باشد. (یکنوایی، monotonicity)
- قضیه: اگر $h(n)$ سازگار باشد، A* با استفاده از GRAPH-SEARCH بهینه است.

بهینگی A*

- **لم:** A* گره ها را براساس ترتیب صعودی مقادیر f گسترش می دهد.
- A* به تدریج f-contour ها را اضافه می کند.
- کانتور i شامل تمام گره ها با $f = f_i$ می باشد، که $f_i < f_{i+1}$



خصوصیات A*

- کامل؟ بله، مگر اینکه تعدادی نامحدود گره با $f \leq f(G)$ وجود داشته باشد. A* در گراف های متناهی محلی (با فاکتور انشعاب محدود) کامل است به شرط آنکه هزینه تمام عملگرها مثبت باشد.
  - گره ای با فاکتور انشعاب نامحدود وجود داشته باشد.
  - مسیری با هزینه محدود اما با تعداد گره های نامحدود وجود داشته باشد.

- پیچیدگی زمانی؟ نمایی بر حسب [خطای نسبی h * طول راه حل]
مگر اینکه خطا در تابع کشف کننده رشدی سریعتر از لگاریتم هزینه مسیر واقعی نداشته باشد، به زبان ریاضی:

$$| h(n) - h^*(n)| \leq O (\log h^*(n))$$

خصوصیات A*

- پیچیدگی فضا؟؟ تمام گره ها را در حافظه نگه می دارد.

- بهینه؟؟ بله – نمی تواند f_{i+1} را گسترش دهد مگر f_i تمام شده باشد.
  - A* تمام گره ها با $f(n) < f^*$ را گسترش می دهد.
  - A* برخی از گره ها با $f(n) = f^*$ را گسترش می دهد.
  - A* گره ای با $f(n) > f^*$ را هرگز گسترش نمی دهد.

- A* دارای کارآیی بهینه (optimally efficient) می باشد!

جستجوی هیوریستیک با حافظه محدود

- چند راه حل برای مسأله حافظه در A* (با حفظ خصوصیات کامل بودن و بهینگی):
  - جستجوی عمیق کننده تکرای A* (IDA*)
    - مانند IDS ولی به جای محدوده عمقی از محدوده f-cost ($g + h$) استفاده می شود
  - جستجوی اول-بهترین بازگشتی (RBFS)
    - یک الگوریتم بازگشتی با فضای خطی که سعی می کند از جستجوی اول-بهترین استاندارد تقلید کند
  - جستجوی (ساده شده) A* با حافظه محدود ((S)MBA*)
    - حذف بدترین گره وقتی که حافظه پر می باشد

جستوی اول-بهترین بازگشتی(RBFS)

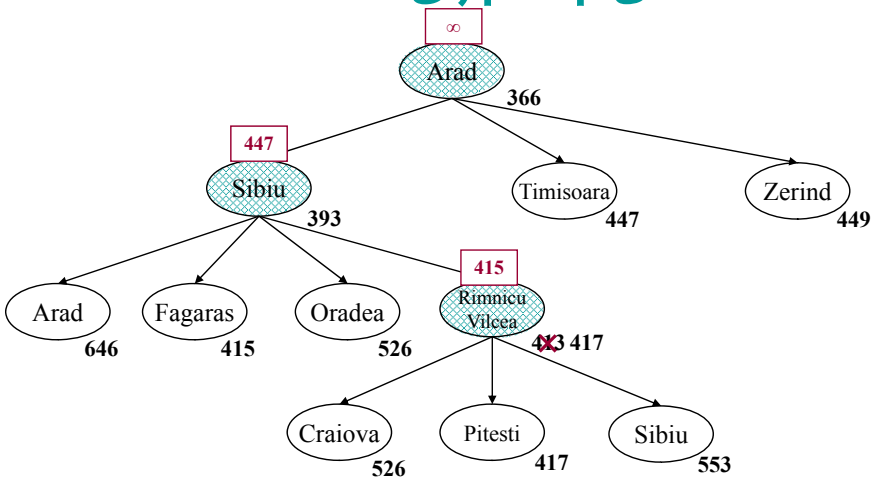
```
function RECURSIVE-BEST-FIRST-SEARCH(problem) return a solution or failure
    return RFBS(problem,MAKE-NODE(INITIAL-STATE[problem]),∞)

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

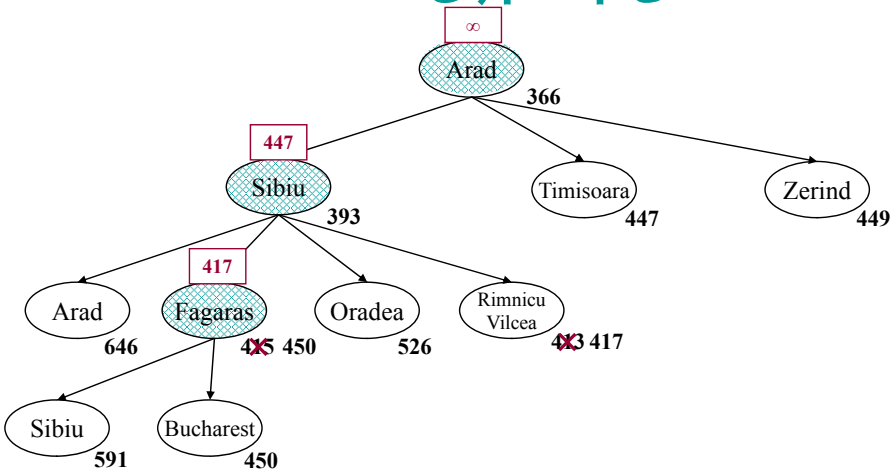
جستجوی RBFS

- مقدار f-value مربوط به بهترین مسیر جایگزین موجود را نگهداری می کند.
  - اگر f-value فعلی از f-value مسیر جایگزین تجاوز کند، آنگاه به این مسیر جایگزین بازگشت می کند.
  - در بازگشت به عقب مقدار f-value مربوط به هر گره موجود در مسیر را با بهترین مقدار f-value فرزندانش جایگزین می کند.
  - بنابراین گسترش مجدد نتیجه فعلی هنوز ممکن می باشد.

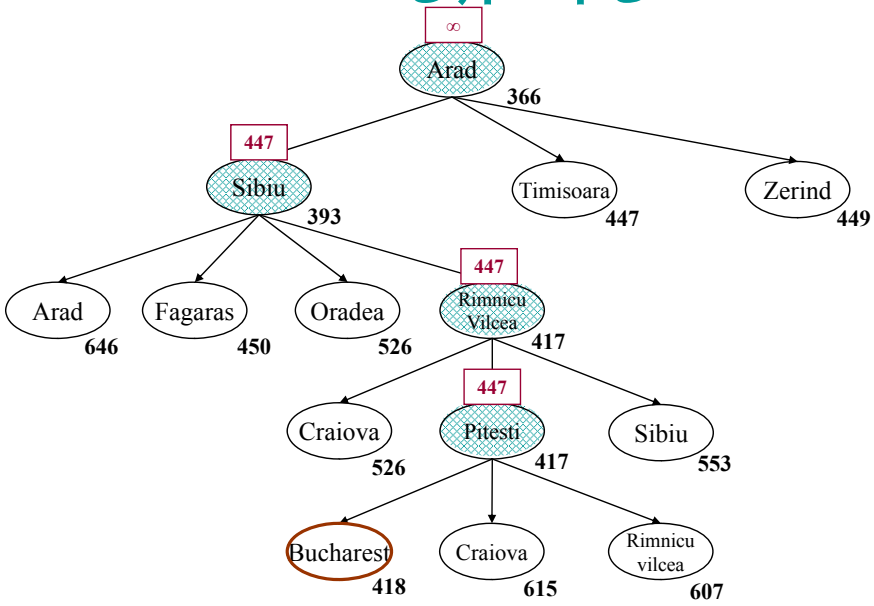
مثال جستجوی RBFS



مثال جستجوی RBFS



مثال جستجوی RBFS



ارزیابی RBFS

- RBFS کمی کارآتر از IDA* می باشد
  - هنوز گسترش اضافی گره ها وجود دارد(تغییر عقیده)
- RBFS هم مانند A*، اگر $h(n)$ قابل قبول باشد بهینه است
- پیچیدگی حافظه $O(bd)$
  - IDA* فقط یک عدد را نگهداری می کند (حد فعلی *f-cost*)
- تعیین پیچدگی زمانی مشکل است
  - به دقت تابع هیوریستیک و میزان تغییر بهترین مسیر در اثر بسط گره ها بستگی دارد.
  - مانند IDA* در معرض افزایش نمایی پیچیدگی زمانی قرار دارد
- RBFS و IDA* هر دو از میزان بسیار کم حافظه رنج می برند.
  - با اینکه حافظه زیادی وجود دارد، نمی توانند از ان استفاده کنند.

(Simplified) Memory Bounded A*

- ایده: استفاده از تمامی حافظه موجود
  - یعنی، گسترش بهترین گره های برگی تا زمانی که حافظه موجود پر شود
  - در صورت پر شدن حافظه، SMA* بدترین گره برگ (با بیشترین مقدار *f-value*) را از حافظه حذف می کند
  - مانند RBFS اطلاعات گره های فراموش شده را در پدرشان ذخیره می کند

- اگر تمام برگ ها دارای f-value برابر باشند چه می شود؟
  - ممکن است یک گره را هم برای گسترش و هم برای حذف انتخاب کند
  - راه حل SMA*:
    - گسترش: بهترین گره برگ که از همه جدید تر است
    - حذف: بدترین گره که از همه قدیمی تر است

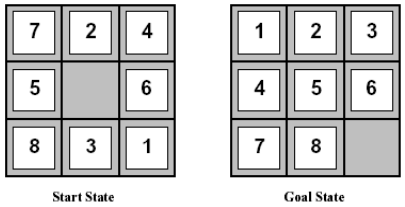
  - SMA* کامل است اگر راه حل قابل دستیابی وجود داشته باشد و بهینه است اگر راه حل بهینه قابل دستیابی باشد

هیوریستیک های قابل قبول

مثال برای معمای هشت:

$h_1(n)$ = number of misplaced tiles

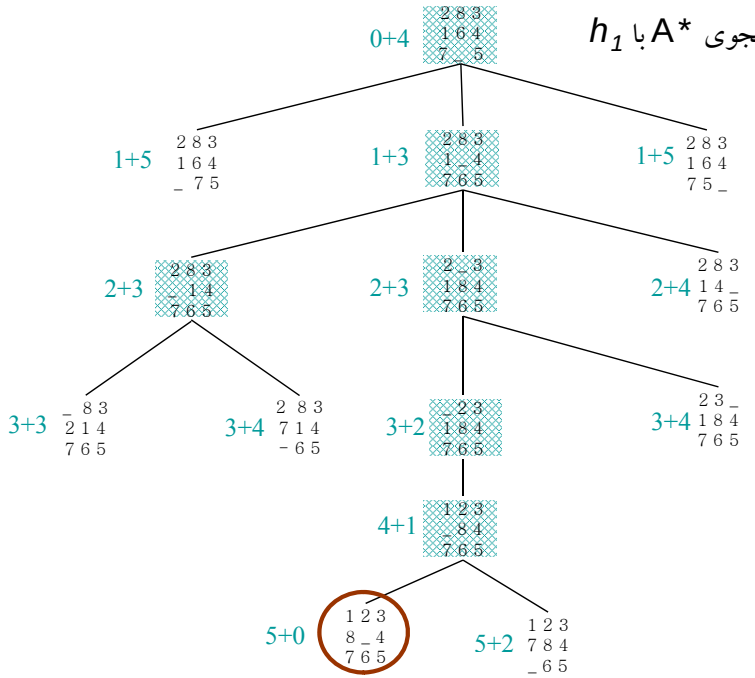
$h_2(n)$ = total Manhattan distance



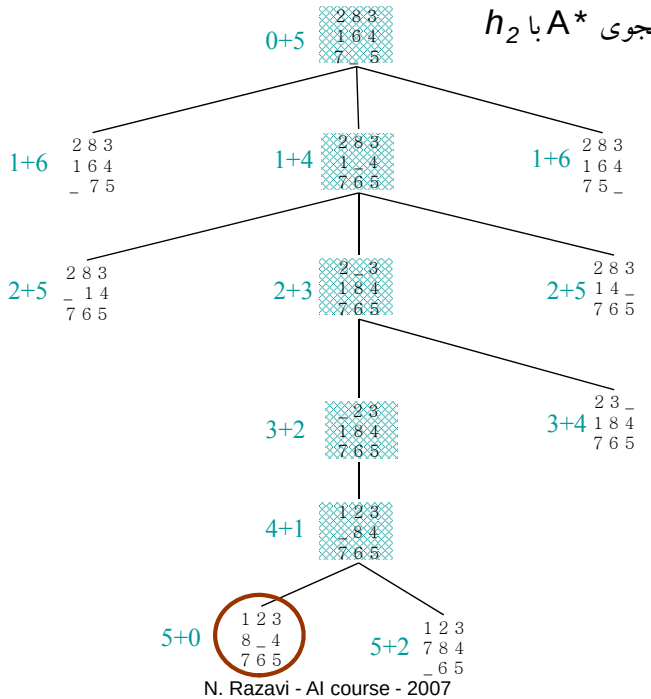
$h_1(S) = 6$

$h_2(S) = 4 + 0 + 3 + 3 + 1 + 0 + 2 + 1 = 14$

- مثال: جستجوی A* با h_1



- مثال: جستجوی A* با h_2



تسلط (Dominance)

- اگر بازاء هر n داشته باشیم، $h_2(n) \geq h_1(n)$ (و هر دو هیوریستیک قابل قبول باشند)
- آنگاه h_2 بر h_1 تسلط دارد و برای جستجو بهتر می باشد.
- مثال هزینه جستجو: تعداد گره های تولید شده در عمق های مختلف:

$d = 14 \rightarrow$ IDS = 3,473,941
A*(h_1) = 539
A*(h_2) = 113

$d = 24 \rightarrow$ IDS = 54,000,000,000
A*(h_1) = 39,135
A*(h_2) = 1641

فاکتور انشعاب موثر (b*)

- فاکتور انشعاب موثر (EBF):
اگر تعداد گره های گسترش یافته توسط روال جستجو N باشد و عمق راه حل d باشد، b^* به صورت زیر محاسبه می شود:

$$N + 1 = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d$$

- مثال: اگر A* راه حلی را در عمق ۵ با استفاده از ۵۲ گره پیدا کند، فاکتور انشعاب موثر را محاسبه کنید.

پاسخ:

$$53 = 1 + b^* + (b^*)^2 + \ldots (b^*)^5 \rightarrow b^* = 1.92$$

- در یک هیوریستیک هر چه فاکتور انشعاب به یک نزدیکتر باشد، بهتر است و آن هیوریستیک کیفیت کشف کنندگی بیشتری دارد.

مقایسه بین هزینه جستجو و فاکتور انشعاب موثر

	Search Cost			Effective Branching Factor		
d	**IDS**	**A*(h_1)**	**A*(h_2)**	**IDS**	**A*(h_1)**	**A*(h_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	6,384	39	25	2.80	1.33	1.24
10	47,127	93	39	2.79	1.38	1.22
12	364,404	227	73	2.78	1.42	1.24
14	3,473,941	539	113	2.83	1.44	1.23
16	-	1,301	211	-	1.45	1.25
18	-	3,056	363	-	1.46	1.26
20	-	7,276	679	-	1.47	1.27
22	-	18,094	1,219	-	1.48	1.28
24	-	39,135	1,641	-	1.48	1.26

ابداع توابع هیوریستیک

- می توان هیوریستیکهای قابل قبول را برای یک مسأله، از *هزینه دقیق* راه حل یک نسخه راحت (relaxed) از مسأله بدست آورد.
- مثال: قانون معمای هشت. یک کاشی می تواند از خانه A به خانه B برود، اگر A مجاور B باشد و B خالی باشد.
- اگر قوانین معمای هشت به گونه ای راحت شوند که یک کاشی بتواند **به هر خانه ای** حرکت کند، هیوریستیک $h_1(n)$ کوتاهترین راه حل را می دهد.
- اگر قوانین به گونه ای راحت شوند که یک کاشی بتواند **به هر خانه مجاور** حرکت کند، هیوریستیک $h_2(n)$ کوتاهترین راه حل را می دهد.
- **نکته کلیدی:** هزینه راه حل بهینه یک مسأله راحت، بیشتر از هزینه راه حل بهینه در مسأله واقعی نیست.

الگوریتم های جستجوی محلی و مسائل بهینه سازی

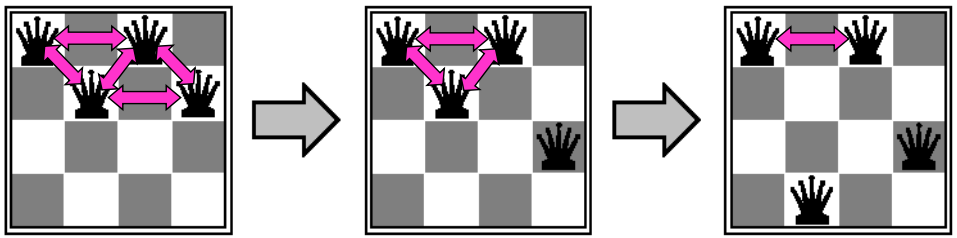
- در بسیاری از مسائل بهینه سازی، مسیر راه حل اهمیت ندارد؛ خود حالت هدف پاسخ مسأله می باشد.
- فضای حالت = مجموعه پیکره بندی های«کامل»؛
- یافتن پیکره بندی بهینه، مانند TSP
- یافتن یک پیکره بندی که محدودیت های مسأله را ارضاء کند، مانند مساله n-وزیر
- در چنین مواردی می توان از **الگوریتم های جستجوی محلی** بهره گرفت.
- **یک حالت«فعلی» را به تنهایی در نظر بگیر؛ سعی کن آن را بهبود بخشی.**

الگوریتم های جستجوی محلی و مسائل بهینه سازی

- **جستجوی محلی** = استفاده از یک حالت فعلی و حرکت به حالت های همسایه
- مزایا:
  - استفاده از حافظه بسیار کم
  - یافتن راه حل های معقول در اغلب موارد در فضاهای حالت بزرگ و یا نامحدود
- مفید برای مسائل بهینه سازی محض
  - یافتن بهترین حالت بر طبق تابع هدف (objective function)

مثال: n-وزیر

- مسأله : n وزیر را در یک صفحه شطرنج $n * n$ به گونه ای قرار بده که هیچ دو وزیری در یک سطر، ستون و یا قطر قرار نگیرند.
- یکی از وزیرها را در ستون خودش به گونه ای جابه جا کن که تعداد برخورد ها کاهش یابد.

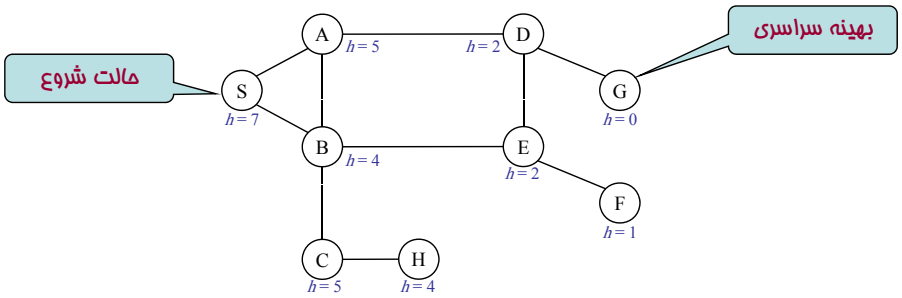


تپه نوردی(یا گرادیان صعودی/نزولی)

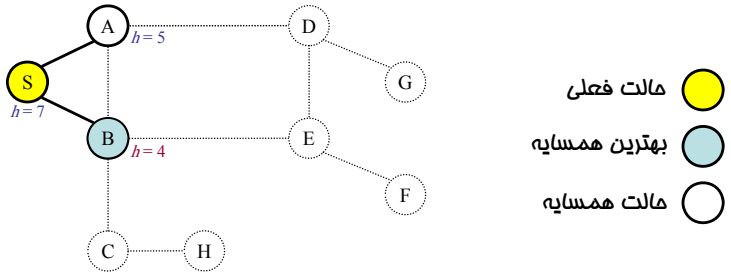
«مانند بالا رفتن از کوه اورست در مه غلیظ با ضعف حافظه»

```
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 ← MAKE-NODE(INITIAL-STATE[problem])
   loop do
      neighbor ← a highest-valued successor of current
      if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
      current ← neighbor
```

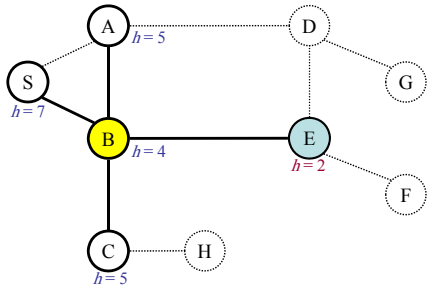

مثال: جستجوی تپه نوردی

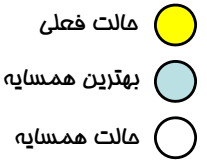


مثال: جستجوی تپه نوردی

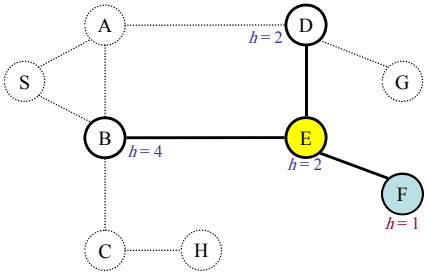


مثال: جستجوی تپه نوردی





مثال: جستجوی تپه نوردی

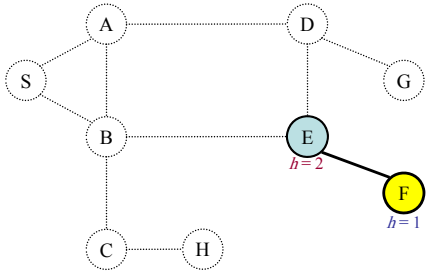


حالت فعلی

بهترین همسایه

حالت همسایه

مثال: جستجوی تپه نوردی

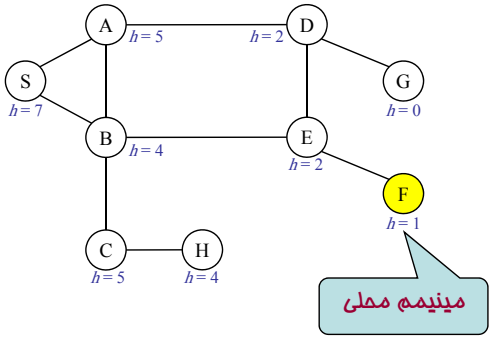


حالت فعلی

بهترین همسایه

حالت همسایه

مثال: جستجوی تپه نوردی

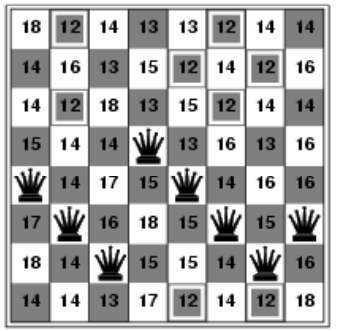


حالت فعلی

بهترین همسایه

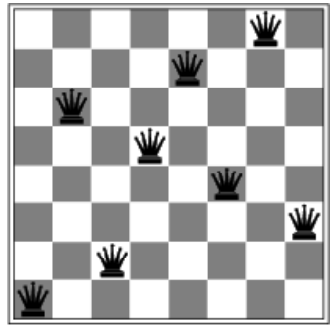
حالت همسایه

جستوی تپه نوردی : مثال ۸ وزیر



$h = 17$

۵ حرکت

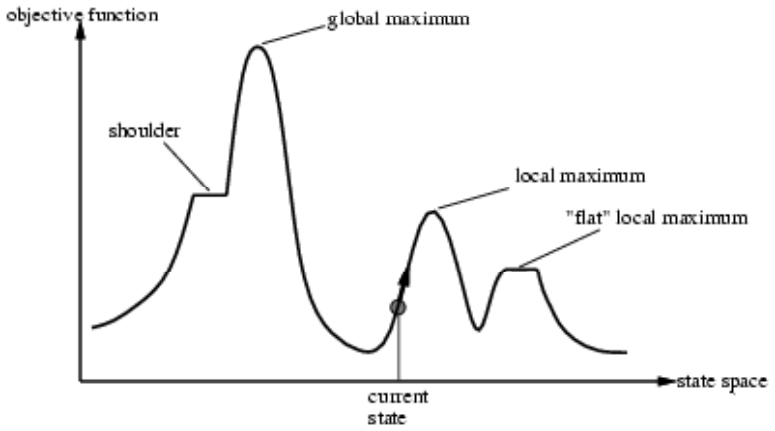


$h = 1$

h = تعداد جفت وزیرهایی که بطور مستقیم و یا بطور غیر مستقیم یکریگر را تهدید می کنند.

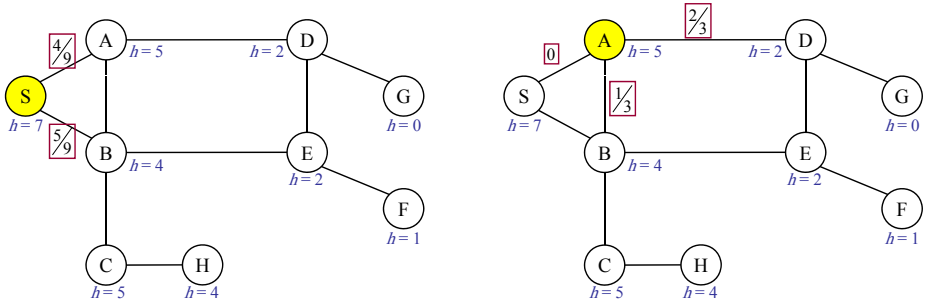
تپه نوردی (ادامه)

- **مشکل**: بسته به حالت اولیه مسأله ممکن است در ماکزیمم محلی گیر کند.



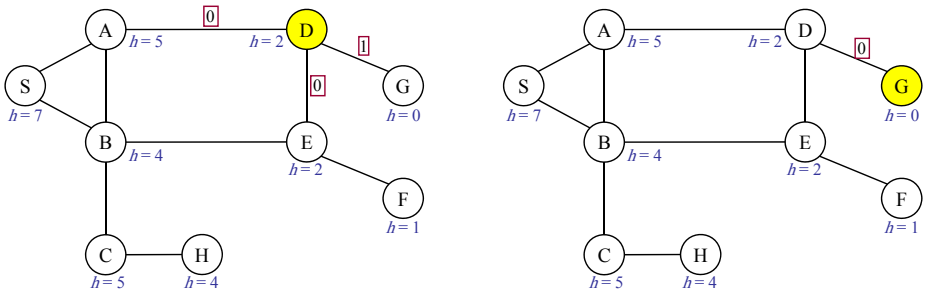
انواع دیگر تپه نوردی (تپه نوردی اتفاقی)

- تپه نوردی اتفاقی
  - انتخاب تصادفی در میان حرکت های رو به بالا
  - احتمال انتخاب می تواند متناسب با شیب حرکت تغییر کند



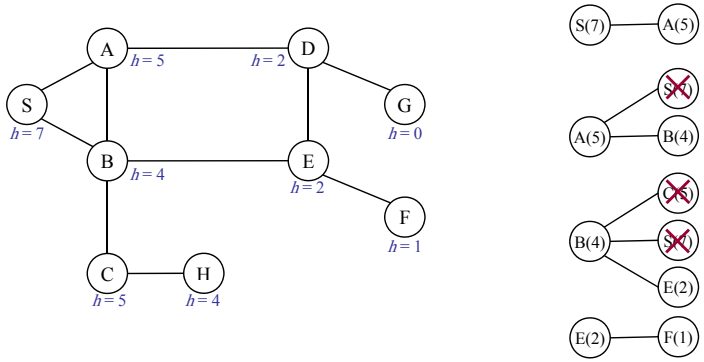
انواع دیگر تپه نوردی (تپه نوردی اتفاقی)

- تپه نوردی اتفاقی
  - انتخاب تصادفی در میان حرکت های رو به بالا
  - احتمال انتخاب می تواند متناسب با شیب حرکت تغییر کند



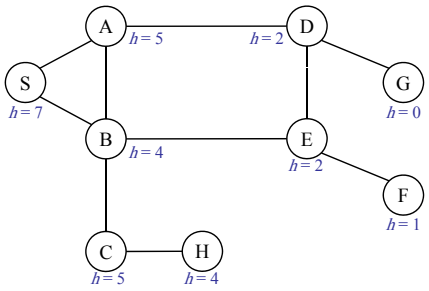
انواع دیگر تپه نوردی (تپه نوردی اولین انتخاب)

- تپه نوردی اولین انتخاب
  - همان تپه نوردی اتفاقی که حالت های بعدی را به طور تصادفی تولید می کند تا یکی از آنها بهتر از حالت فعلی باشد



انواع دیگر تپه نوردی (تپه نوردی با شروع مجدد تصادفی)

- تپه نوردی با شروع مجدد تصادفی (ایده: اگر شکست خوردی دوباره سعی کن)
  - سعی می کند که از گیر افتادن در ماکزیمم محلی اجتناب کند



S → B → E → F ✗

C → B → E → F ✗

H ✗

C → H ✗

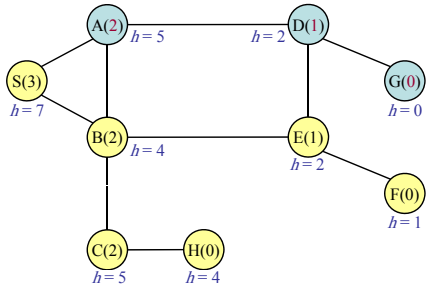
A → D → G

مشکل: معمولاً در یک مسأله در دنیای واقعی، قبل از حل مسأله جواب بهینه را نمی دانیم !

راه حل: تکرار تپه نوردی تا زمانی که وقت داریم.

انواع دیگر تپه نوردی (تپه نوردی با شروع مجدد تصادفی)

- محاسبه تعداد متوسط مراحل در تپه نوردی با شروع تصادفی مجدد
  - تعداد تکرارها = عکس احتمال موفقیت
  - تعداد متوسط مراحل = تعداد شکست ها * تعداد متوسط مراحل شکست + ۱ * تعداد متوسط مراحل موفقیت



احتمال موفقیت = $\frac{1}{3}$

تعداد تکرارها = 3

تعداد متوسط مراحل موفقیت = 1

تعداد متوسط مراحل شکست = $\frac{8}{6}$

$(3-1)\times \frac{8}{6} + 1 \times 1 \cong 4$

Simulated Annealing

- **ایده**: از ماکزیمم های محلی با انجام حرکت های« بد» فرار کن

اما به تدریج اندازه و تعداد حرکات بد (به سمت پایین) را کم کن

function SIMULATED-ANNEALING(*problem, schedule*) **returns** a solution state
  inputs: *problem*, a problem
    schedule, a mapping from time to "temperature"
  local variables: *current*, a node
    next, a node
    T, a "temperature" controlling prob. of downward steps
  current ← MAKE-NODE(INITIAL-STATE[*problem*])
  for *t* ← 1 **to** ∞ **do**
    T ← *schedule*[*t*]
    if *T* = 0 **then return** *current*
    next ← a randomly selected successor of *current*
    Δ*E* ← VALUE[*next*] – VALUE[*current*]
    if Δ*E* > 0 **then** *current* ← *next*
    else *current* ← *next* only with probability $e^{\Delta E/T}$

آنلینگ شبیه سازی شده

- پیشروی مانند تپه نوردی می باشد، اما در هر مرحله حالت بعدی به طور تصادفی انتخاب می شود.
- اگر حالت بعدی انتخاب شده بهتر باشد، همواره به آن حالت بعدی خواهیم رفت.
- در غیر این صورت، تنها با یک احتمال به آن حالت خواهیم رفت و این احتمال به صورت نمایی کاهش می یابد.
- تابع احتمال:

$e^{\Delta E/T}$

T: تعداد مراحل بهبود راه حل

ΔE: میزان کاهش در هر مرحله

خصوصیات آنلینگ شبیه سازی شده

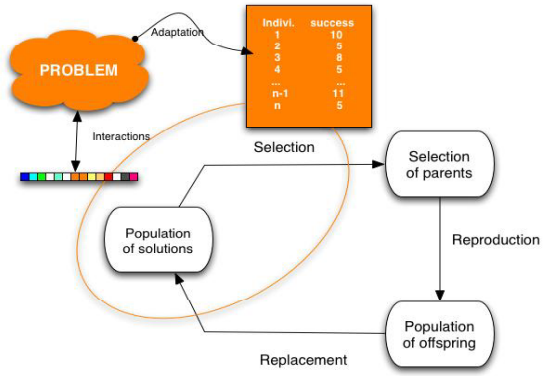
- در مقادیر بالاتر T، احتمال انجام حرکات بد (رفتن به سمت پایین)بیشتر است. (مانند جستجوی تصادفی رفتار می کند)
- با کاهش T این احتمال کاهش یافته و در $T = 0$ این احتمال به صفر می رسد. (مانند تپه نوردی رفتار می کند)
- می توان ثابت کرد که اگر T به به اندازه کافی آرام کاهش بیابد، جستجوی SA یک پاسخ بهینه (global optimum) با احتمالی که به سمت یک میل می کند خواهد یافت.
- کاربردها:
  - حل مسائل VLSI
  - برنامه ریزی
  - اعمال بهینه سازی
  - زمانبندی خطوط هوایی

Local beam search
(جستجوی محلی دسته ای)

- شروع با k حالت که به طور تصادفی ایجاد شده اند.
- در هر تکرار، تمام فرزندان برای هر k حالت تولید می شوند.
- اگر یکی از آنها حالت هدف بود جستجو متوقف می شود و در غیر این صورت از میان لیست کامل فرزندان k تا از بهترین ها انتخاب می شوند و مرحله بالا تکرار می شود.

الگوریتم ژنتیک

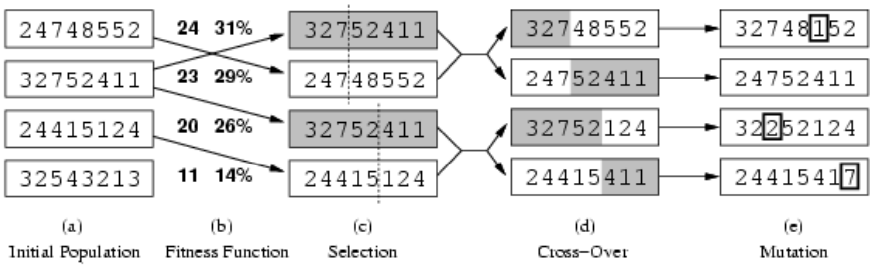
- نوعی از local beam search به همراه ترکیب جنسی



الگوریتم های ژنتیک

- یک حالت بعدی با ترکیب دو حالت پدر ایجاد می شود.
- شروع با k حالت تصادفی (جمعیت)
- یک حالت با یک رشته بر روی یک مجموعه محدود بازنمایی می شود (اغلب رشته ای از صفر و یک) – (کروموزوم)
- تابع ارزیابی (تابع برازندگی – Fitness function) : مقادیر بالاتری را برای حالات بهتر ایجاد می کند.
- نسل بعدی جمعیت با انجام اعمال زیر روی جمعیت فعلی تولید می شود:
  - انتخاب (Selection)
  - آمیزش (Crossover)
  - جهش (Mutation)

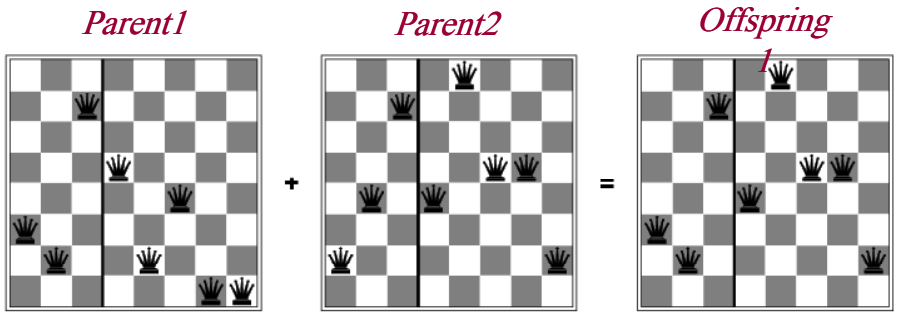
الگوریتم های ژنتیک



- تابع برازندگی : تعداد جفت وزیر هایی که یکدیگر را تهدید نمی کنند
  (مینیمم: صفر، ماکزیمم: 8 * 7/2 = 28)
- 24 / (24+23+20+11) = 31%
- 23 / (24+23+20+11) = 29%

الگوریتم های ژنتیک

- یک مثال برای عمل Crossover



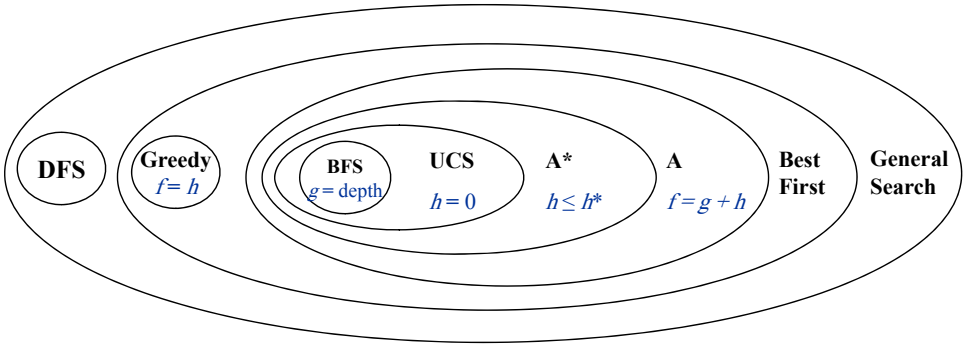
(3, 2, 7, 5, 2, 4, 1, 1) + (2, 4, 7, 4, 8, 5, 5, 2) = (3, 2, 7, 4, 8, 5, 5, 2)

الگوریتم ژنتیک

function GENETIC-ALGORITHM(*population,* FITNESS-FN) **returns** an individual
  input: *population*, a set of individuals
    FITNESS-FN, a function that measures the fitness of an individual

  repeat
    new_population ← empty set
    loop for *i* **from** 1 **to** SIZE(*population*) **do**
      x ← RANDOM_SELECTION(*population*, FITNESS_FN)
      y ← RANDOM_SELECTION(*population*, FITNESS_FN)
      child ← REPRODUCE(*x*,*y*)
      if (small random probability) **then** *child* ← MUTATE(*child*)
      add *child* to *new_population*
    population ← *new_population*
  until some individual is fit enough or enough time has elapsed
  return the best individual in population, according to FITNESS-FN

دسته بندی استراتژی های جستجو

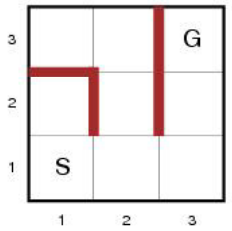


مسائل اکتشافی

- تا کنون تمام الگوریتم ها offline بودند
  - Offline = راه حل قبل از اجرای آن معین است
  - Online = محاسبه و عمل بصورت یک در میان
  - جستجوی online برای محیط های پویا و نیمه پویا ضروری می باشد
    - در نظر گرفتن تمامی امکان های مختلف غیر ممکن است
- استفاده شده در مسائل اکتشافی
  - حالت ها و اعمال ناشناخته
  - مثال: یک روبات در یک محیط جدید، یک نوزاد و ...

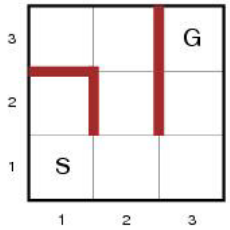
مسائل جستجوی online

- دانش عامل:
  - عمل (ها): لیست اعمال مجاز در حالت s
  - $c(s, a, s')$: تابع هزینه گام (پس از تعیین s')
  - GOAL-TEST(s)
- عامل می تواند حالت قبلی را تشخیص دهد
- اعمال قطعی هستند
- دستیابی به هیوریستیک قابل قبول $h(s)$
- مانند فاصله مانهاتانی

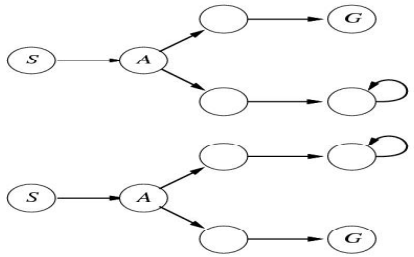


مسائل جستجوی online

- هدف: رسیدن به حالت هدف با حداقل هزینه
  - هزینه = کل هزینه مسیر پیموده شده
  - نسبت رقابتی = مقایسه هزینه با هزینه مسیر راه حل در حالتی که فضای جستجو شناخته شده باشد
  - می تواند نا محدود باشد
  - در مواردی که عامل به طور تصادفی به یک بن بست می رسد



استدلال دشمنانه



- یک دشمن را تصور کنید که قادر است در حین کاوش عامل فضای حالت را بسازد
- حالت های ملاقات شده S و A. حالت بعدی؟
  - در یکی از فضاهای حالت شکست می خورد
  - هیچ الگوریتمی نمی تواند از بن بست ها در تمامی فضاهای حالت اجتناب کند

عامل های جستجوی Online

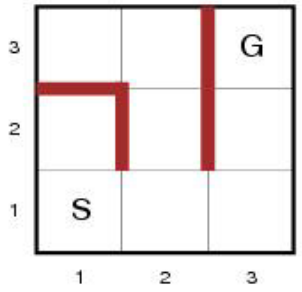
- عامل یک نقشه از محیط نگهداری می کند
  - نقشه بر اساس ورودی ادراکی بهنگام می شود
  - از این نقشه برای انتخاب عمل بعدی استفاده می شود
- به تفاوت مثلا با A* دقت کنید
  - یک نسخه online تنها می تواند گرهی را گسترش دهد که به طور فیزیکی در آن قرار داشته باشد

Online DFS-Agent

```
function ONLINE-DFS-AGENT(s') returns an action
    input: s', a percept identifying current state
    static: result, a table indexed by action and state, initially empty
            unexplored, a table that lists for each visited state, the action not yet tried
            unbacktracked, a table that lists for each visited state, the backtrack not yet tried
            s,a, the previous state and action, initially null

    if GOAL-TEST(s') then return stop
    if s' is a new state then unexplored[s'] ← ACTIONS(s')
    if s is not null then do
            result[a,s] ← s'
            add s to the front of unbackedtracked[s']
    if unexplored[s'] is empty then
            if unbacktracked[s'] is empty then return stop
            else a ← an action b such that result[b, s']=POP(unbacktracked[s'])
    else a ← POP(unexplored[s'])
    s ← s'
    return a
```

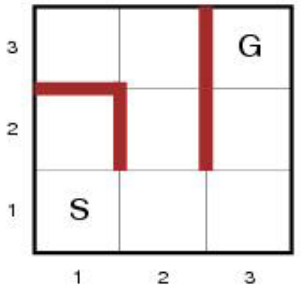

جستجوی عمقی online ، مثال



- مساله مسیر پر پیچ و خم بر روی یک صفحه ۳*۳
- حالت اولیه: $s'=(1, 1)$
- Result و Unexplored(UX) و Unbacktracked(UB) و...

  تهی هستند
- S و a نیز تهی هستند

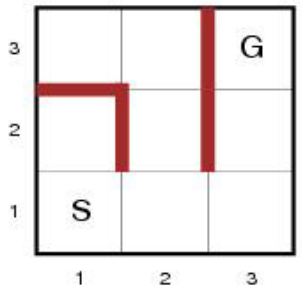
جستجوی عمقی online، مثال



- GOAL-TEST((1, 1))?
  - S not = G thus false
- (1,1) a new state?
  - True
  - ACTION((1,1)) -> UX[(1,1)]
    - {RIGHT,UP}
- s is null?
  - True (initially)
- UX[(1,1)] empty?
  - False
- POP(UX[(1,1)]) -> a
  - a = UP
- s = (1,1)
- Return a

جستجوی عمقی online، مثال

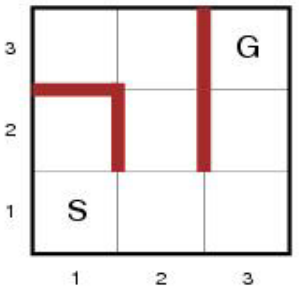
S'=(1,1)



- GOAL-TEST((1,1))?
  - S not = G thus false
- (1,1) a new state?
  - false
- *s* is null?
  - false (*s* = (2, 1))
  - result [DOWN, (2, 1)] <- (1,1)
  - UB[(1,1)] = {(2,1)}
- UX[(1,1)] empty?
  - False
- *a* = RIGHT, *s* = (1,1)
- return *a*

جستجوی عمقی online، مثال

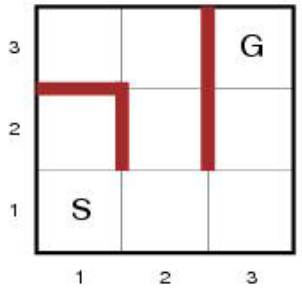
S'=(1,2)



- GOAL-TEST((1,2))?
  - S not = G thus false
- (1,2) a new state?
  - True, UX[(1,2)]={RIGHT,UP,LEFT}
- *s* is null?
  - false (s = (1,1))
  - result [RIGHT, (1,1)] <- (1, 2)
  - UB[(1,2)] = {(1,1)}
- UX[(1, 2)] empty?
  - False
- *a* = LEFT, *s* =(1, 2)
- return *a*

جستجوی عمقی online، مثال

S'=(1,1)



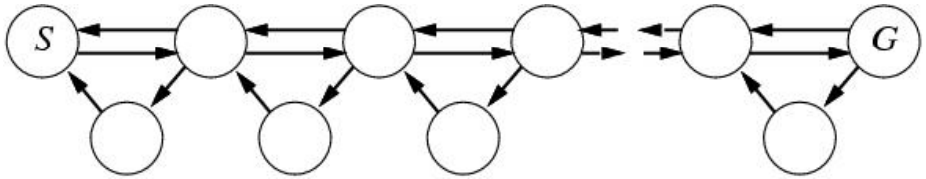
- GOAL-TEST((1, 1))?
  - S not = G thus false
- (1, 1) a new state?
  - false
- *s* is null?
  - false (*s* = (1, 2))
  - result [LEFT, (1, 2)] <- (1, 1)
  - UB[(1, 1)]={(1, 2), (2, 1)}
- UX[(1, 1)] empty?
  - True
  - UB[(1, 1)] empty? False
- *a* = *b* for *b* in result [*b*, (1,1)] = (1, 2)
  - *b* = RIGHT
- *a* = RIGHT, s = (1, 1) …

جستجوی عمقی online، مثال

- بدترین حالت: هر گره دو بار ملاقات شده است
- یک عامل ممکن است در حالی که به پاسخ نزدیک است، یک راه طولانی را بپیماید
- یک روش عمیق کننده تکرای online می تواند این مشکل را حل کند
- جستجوی عمقی online فقط وقتی کار می کند که اعمال قابل برگشت باشند

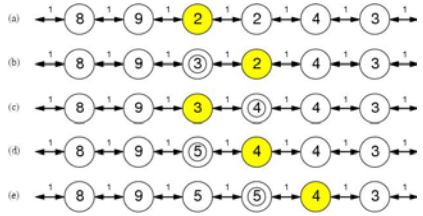
جستجوی محلی online

- تپه نوردی online می باشد
  - یک حالت ذخیره شده است
- عملکرد بد به دلیل وجود ماکزیمم های محلی
  - شروع مجدد تصادفی غیر ممکن است
- راه حل: حرکت تصادفی باعث کاوش می شود (می تواند حالات بسیاری را به صورت نمایی تولید کند)



جستجوی محلی online

- راه حل ۲: اضافه نمودن حافظه به تپه نورد
- ذخیره بهترین تخمین فعلی $H(s)$ از هزینه رسیدن به هدف
- $H(s)$ در ابتدا تخمین هیوریستیک $h(s)$ می باشد
- پس از آن بر اساس تجربه بهنگام می شود (شکل پایین)



Learning real-time A*

```
function LRTA*-COST (s, a, s',H) return an cost estimate
    if s' is undefined the return h(s)
    else return c(s, a, s') + H[s']

function LRTA*-AGENT (s') return an action
    input: s', a percept identifying current state
    static: result, a table indexed by action and state, initially empty
            H, a table of cost estimates indexed by state, initially empty
            s,a, the previous state and action, initially null

    if GOAL-TEST (s') then return stop
    if s' is a new state (not in H) then H[s'] ← h(s')
    unless s is null
            result[a,s] ← s'
            H[s] ← MIN  LRTA*-COST (s, b, result [b, s], H)
                    b ∈ ACTIONS (s)
     a ← an action b in ACTIONS (s') that minimizes LRTA*-COST (s', b, result [b, s'], H)
    s ← s'
    return a
```