

## سوالات میانترم

سیستمی که مانند انسان رفتار می کند را با ذکر مثال توضیح دهید؟

هنر ساخت ماشینهایی که کارهایی انجام میدهند که آن کارها فعالیت توسط انسان با فکر کردن انجام میشود. مطالعه برای ساخت کامپیوترهایی که کارهایی را انجام دهند که فعالانسان آنها را بهتر انجام میدهند. تست تورینگ مثالی مناسب برای این سیستم است. در این تست کامپیوتر توسط فردی محقق مورد آزمایش قرار میگیرد، به طوری که این فرد دور از کامپیوتر قرار دارد، کامپیوتر به پرسش های مطرح شده پاسخ میدهد. کامپیوتر وقتی از این تست عبور میکند که این شخص نتواند تشخیص دهد که پاسخ دهنده یک انسان است یا چیز دیگر. این تست باید قابلیت هایی نظیر (پردازش زبان طبیعی، بازنمایی دانش، استدلال خودکار، یادگیری ماشین، بینایی کامپیوتر، دانش رباتیک) داشته باشد.

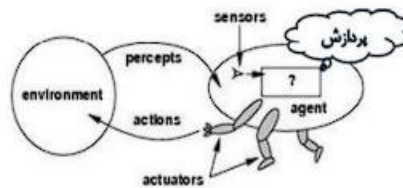
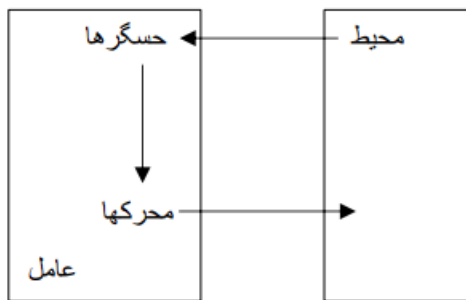
هدف از تفکر عاقلانه چیست و چه آورده ای در پی خواهد داشت؟

عاقلانه فکر کردن، به معنایی ساخت الگوهایی برای ساختارهای استدلالی است. در واقع عاقلانه فکر کردن یعنی مطالعه ی توانایی های ذهنی از طریق مدل های محاسباتی (منطق گرایی) عاقلانه فکر کردن مطالعه ی محاسباتی است که منجذب درک و استدلال شود. عاقلانه تفکر کردن رسم منطق گرایی در هوش مصنوعی برای ساخت سیستم های هوشمند است. در واقع برنامه هایی نوشته میشوند که میتوانند مسائل قابل حلی که در نماد گذاری منطقی توصیف میشوند را حل کنند. موانع اصلی این تفکر: 1) دریافت دانش غیر رسمی و تب دیل آن به دانش رسمی 2) تفاوت میان قادر به حل مسئله بودن در تئوری و در عمل (بن بست محاسباتی)

### جزای عامل و وظیفه عامل را بارسم شکل و تابع نویسی بررسی کنید؟

عامل هر چیزی است که قادر است محیط خود را از طریق حسگرها درک کند و از طریق محرک ها عمل کند. به عنوان مثال عامل رباتیک شامل دوربینهایی به عنوان سنسورها یا حسگر. موتورهای متعدد دی به عنوان محرک است. یا عامل انسان دارای چشم و گوش و اعضای دیگر برای حس کردن و دست و پا و دهان و اعضای دیگر به عنوان محرک است. عاملها از طریق حسگرها و محرکها با محیط در تعامل هستند. سنسور وظیفه دریافت مشخصه هایی از محیط را دارد و محرک وظیفه انجام اعمال بر روی محیط را دارد. عامل وظیفه دارد رشته دریافت های ورودی را به دنباله ای از اعمال نگاشت کند. بنابراین میتوان گفت عامل میتواند مانند تابع عمل کند.

$F : P^* \rightarrow A$  که  $A$  اعمال و  $P$  رشته دریافت ها است. عامل میتواند اعمال محیط خود را درک کند، اما تأثیر آنها بر روی محیط همیشه قابل پیش بینی نیست



## PEAS رابرای ربات فضاءنورد وفوتبالیست تشریح کنید؟

ربات فوتبالیست :

معیار کارایی : برد بازی - رعایت قوانین - سرعت عمل مناسب

محیط : زمین چمن - زمین خاکی - سالن ورزشی - زمین آسفالت - تیم خود - توپ - تیم حریف

عملگر : پاس دادن - گل زدن - حمله - دفاع

سنسور: سرعت سنج - فاصله یاب - بازوهای محرک - سنسور رو به عقب - سنسور رو به جلو - موقعیت یاب

ربات فضاءنورد :

معیار کارایی : دسته بندی صحیح تصاویر - کمترین هزینه - سرعت عمل مناسب - ایمنی

محیط : محل آزمایشگاه - فضا

عملگر : نمایش تصاویر طبقه بندی شده - تشخیص ها - نمایشگر

سنسور : آرایه ای از پیکسل های رنگی - دوربین - سونار(مکان یاب صوتی) - حسگر دما - حسگر فشار هوا -

حسگرهای شیمیایی

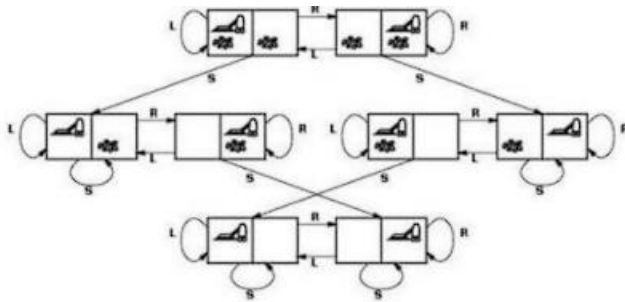
طبق شبه کد زیر چرا عامل مبتنی بر جدول به شکست مواجه می شود ؟ راهکار های پیشنهادی خود را نام برده و مختصری در خصوص هر کدام توضیح دهید ؟

```
function TABLE-DRIVEN-AGENT(percept) returns an action
  persistent: percepts, a sequence, initially empty
               table, a table of actions, indexed by percept sequences, initially fully specified

  append percept to the end of percepts
  action ← LOOKUP(percepts, table)
  return action
```

این برنامه یک برنامه عامل ساده است که دنباله ی ادراک را ردیابی کرده و از آن به عنوان شاخصی در جدول فعالیتها استفاده می کند تا تصمیم بگیرد چه کاری باید انجام دهد. محیط حسگرها محرکها عامل برای ساخت عامل خردمند ما باید جدولی بسازیم که برای هر دنباله ی ادراک ممکن، دارای فعالیتهای مناسبی باشد. رهیافت جدولی برای ساخت عامل باشکست مواجه میشود چون ما به ازای مجموعه ای ادراکات ممکن و تعداد کل ادراکاتی که عامل دریافت میکند برای جدول جستجو باید تعداد زیادی درایه داشته باشیم که امکان پذیر نیست. اندازه این جدول بیان میکنند که (1) هیچ عامل فیزیکی در این دنیا نمیتواند فضایی برای ذخیره این جدول داشته باشد: (2) طراح برای ایجاد جدول زمان مناسب ندارد: (3) هیچ عاملی نمیتواند تمام درایه های جدول را در تجربه خود به کارگیرد (4). حتی اگر محیط خیلی ساده باشد که اندازه جدول قابل تحمل باشد طراح نمیداند درایه های جدول را چگونه ذخیره کند. شبه کد بالا یک تابع عامل مطلوب را پیاده سازی می کند، ما باید بدانیم چگونه یک برنامه ای بنویسیم که با استفاده از یک کد کوچک به جای جدول بزرگ رفتار عقلایی را انجام دهد.

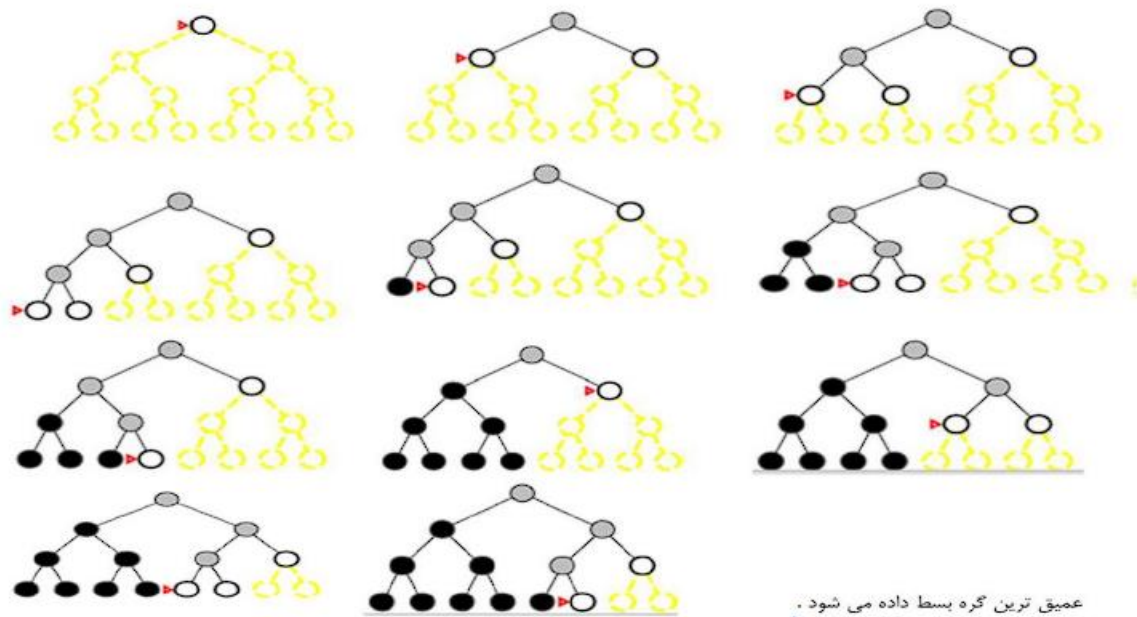
دنیای جاروبرقی را با توجه به فرموله سازی مسئله تشریح کنید؟



حالتها states: حالت به وسیله مکان عامل و مکانهای کثیف تعیین میشود. عامل در یکی از دو مکان است که هر کدام ممکن است کثیف باشند یا نباشند. پس  $2^2 \times 2^2 = 8$  حالت وجود دارد. یعنی  $(n^2 \cdot n)$  حالت. شروع: هر حالتی میتواند به عنوان حالت شروع باشد.

فعاليتها: (action) در این محیط جاروبرقی فقط سه فعالیت میتواند انجام دهد: (1) حرکت به سمت چپ (2) حرکت به سمت راست (3) عمل مکش آزمون هدف: بررسی میکند آیا تمام مکانها تمیز است یا خیر. هزینه ی مسیر: تعداد مراحل موجود در مسیر، هزینه ی مسیر است.

جست و جوی عمقی را با رسم مرحله به مرحله شرح دهید و در نهایت کارایی الگوریتم را با چهار معیار اندازه گیری بیان کنید ؟



جستجوی عمقی، عمیق ترین گره را بسط میدهد، جستجو از عمیق ترین سطح درخت جستجو ادامه می یابد، و وقتی وقتی گره ها بسط داده شدند از مرز حذف میشوند و جستجو به عمیق تری ن گره بعدی برمی گردد. جستجوی عمقی از صف LIFO استفاده میکند. در این صف جدیدترین گره تولید شده، برای بسط دادن انتخاب میشود، این گره باید عمیق ترین گره بسط نداده شده باشد .

### جستجوی عمقی :

کامل بودن : خیر ، مگر اینکه فضای حالت محدود باشد و حلقه تکرار وجود نداشته باشد .

بهینه بودن : خیر ، چون کامل نیست

پیچیدگی زمانی  $O(b^m)$ ، اگر  $m$  خیلی بزرگتر از  $d$  باشد به مراتب بدتر است / در بسیاری از مسائل سریعتر از جست وجوی BF است . پیچیدگی حافظه  $O(bm+1)$ ، در زمان عقبگرد حافظه آزاد می شود

ضمن بررسی الگوریتم جستجوی درختی شبه کد زیر را بررسی کنید که استراتژی در کدام از 4 توابع ، پیاده سازی شده است ، توابع را نام برده و عملکرد هر یک را بیان کنید ؟

```
function TREE-SEARCH(problem, fringe) return a solution or failure
  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
    if EMPTY?(fringe) then return failure
    node ← REMOVE-FIRST(fringe)
    if GOAL-TEST[node] then
      return SOLUTION(node)
    else
      fringe ← INSERT-ALL(EXPAND(node, problem), fringe)
```

در الگوریتم جستجوی درختی ، حالت شروع در ریشه درخت قرار می گیرد ، انشعابها ، فعالیتها و گره ها ، حالت های موجود هستند. ابتدا ریشه را بررسی میکنیم که آیا حالت هدف است یا خیر در صورتی که حالت هدف نبود آن را بسط میدهیم تا مجموعه ی جدیدی از حالتها به وجود آید ، بعد از آن حالتها را یکی یکی بررسی کرده تا زمانی که به آخرین گره برسیم که هیچ فرزندی ندارد. پس سراغ گره ها میرویم و یکی یکی بررسی میکنیم پس از آن گره هایی که ما را به هدف نمیرساند حذف میکنیم و این روش ادامه پیدا میکند تا به هدف برسیم . استراتژی های متفاوتی برای رسیدن به حالت هدف وجود دارد. استراتژی ما در اینجا این است که یک گره کاندید را بررسی کن اگر هدف نبود آن را بسط بده ، آنقدر این کار را تکرار کن تا به هدف برسی .

تابع: (first remove) اولین خانه را *fringe* میکند

تابع: (test goal) آیا به هدف رسیدیم؟ خیر. یک گره با توجه به استراتژی انتخاب کن

تابع: (expand) وقتی به هدف نرسیدیم گره ها را بسط بده .

تابع: (insert) گره های فرزند را در *fringe* بسط بده و نتایج را به جستجو اضافه کن . استراتژی در تابع insert پیاده سازی شده است.

شبه کد زیر مربوط به کدام جست و جوی ناآگاهانه می باشد ، از مزایای کدام جست و جوی های دیگر بهره برده است ، با ترسیم شکل توضیح دهید ؟

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
  for depth = 0 to  $\infty$  do
    result  $\leftarrow$  DEPTH-LIMITED-SEARCH(problem, depth)
    if result  $\neq$  cutoff then return result
```

این شبه کد مربوط به جستجوی عمقی تکرارشونده است ، که این الگوریتم از لحاظ زمانی از مرتبه جستجوی اول سطحی است و از لحاظ پیچیدگی حافظه از مرتبه جستجوی اول عمق بهره میبرد . جست و جوی عمقی تکراری ، یک استراتژی کلی است . این الگوریتم با شروع از مقدار صفر به عنوان عمق محدود ، مقدار آن را به تدریج اضافه می کند مانند یک و .. تا اینکه هدفی پیدا شود . هدف وقتی پیدا می شود که عمق محدود به  $d$  برسد ، که  $d$  عمق مربوط به عمیق ترین گره هدف است . این الگوریتم از مزایای جست و جوی عمقی و جست و جوی عرضی استفاده می کند فواید مربوط به این دو الگوریتم را با هم ترکیب می کند . این الگوریتم برای تعیین عمق محدود است که جست و جوی با عمق محدود را با حدود صعودی تکرار می کند و زمانی خاتمه می یابد که جوابی پیدا شود یا جست و جوی با عمق محدود مقدار *failure* را برگرداند که این عمل نشان می دهد جوابی وجود ندارد .

■ Limit=0

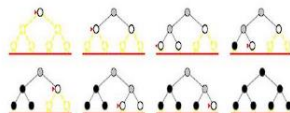
شکل :



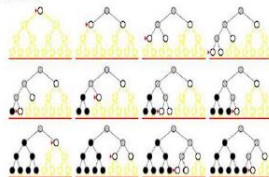
■ Limit=1



■ Limit=2



■ Limit=3





شش نوع جست و جو های ناآگاهانه جدول زیر را به تفکیک ، با چهار معیار مربوطه به اختصار شرح دهید ؟

Criterion	Breadth-First	Uniform-cost	Depth-First	Depth-limited	Iterative deepening	Bidirectional search
Complete?	YES*	YES*	NO	YES, if $l \geq d$	YES	YES*
Time	$b^{d+1}$	$b^{C*/\epsilon}$	$b^m$	$b^l$	$b^d$	$b^{d/2}$
Space	$b^{d+1}$	$b^{C*/\epsilon}$	$bm$	$bl$	$bd$	$b^{d/2}$
Optimal?	YES*	YES*	NO	NO	YES	YES

### 1) جست و جو ی سطحی

کامل بودن : بله / شرط : جواب بهینه در عمق  $d$  قابل دسترس باشد . فاکتور انشعاب  $b$  محدود باشد .

بهینه بودن : بله / شرط : مسیر ها فاقد هزینه باشند .

پیچیدگی زمانی : گره ریشه حداکثر دارای  $b$  فرزند است / هر فرزند نیز حداکثر دارای  $b$  فرزند است بنابراین در

سطح دوم  $b^2$  گره وجود دارد / با فرض اینکه جواب در عمق  $d$  باشد در بدترین حالت جواب باید در سمت

راست ترین گره باشد / تعداد نود های تولید شده از رابطه زیر محاسبه می شود

$$b^0 + b^1 + b^2 + b^3 + \dots + b^d + (b^{d+1} - b) = O(b^{d+1})$$

$$b^{d+1} - b = O(b^{d+1})$$

پیچیدگی حافظه : هم مرتبه پیچیدگی زمانی است .

2) جست و جو با هزینه یکنواخت

کامل بودن : بله / شرط : جواب در عمق قابل دسترس باشد . هزینه ها مقدار مثبت داشته باشند

بهینه بودن : بله / شرط : کامل باشد

پیچیدگی زمانی : فرض شود  $C^*$  هزینه مسیر بهینه است . فرض شود هزینه هر عمل حداقل  $e$  است . در بدترین حالت . است زمانی پیچیدگی  $O(b^C * e)$  پیچیدگی حافظه : هم مرتبه پیچیدگی زمانی است

3) جست و جوی عمق

کامل بودن : خیر / شرط : مگر اینکه فضای حالت محدود باشد و حلقه تکرار وجود نداشته باشد

بهینه بودن : خیر / زیرا کامل نیست

پیچیدگی زمانی :  $O(b^m)$  است ، اگر  $m$  خیلی بزرگتر از  $d$  باشد به مراتب بدتر است . در بسیاری از مسائل سریعتر از جست و جوی BF است

پیچیدگی حافظه :  $O(bm+1)$  در زمان عقبگرد حافظه آزاد می شود .

4) جست و جوی عمقی محدود

در حقیقت DF با عمق محدود  $L$  است

تعیین در همه مسائل امکان پذیر نمی باشد

اگر  $L < d$  آنگاه غیر کامل است

اگر  $L > d$  آنگاه کامل اما غیر بهینه است .

اگر  $d=L$  آنگاه کامل و بهینه است

پیچیدگی زمانی :  $O(b^L)$

پیچیدگی حافظه :  $O(bL)$

5) جست و جوی عمق ی تکراری

کامل بودن : بله / شرط : حلقه تکرار وجود نداشته باشد .

بهینه بودن : بله / اگر هزینه مسیر ها با هم برابر باشد.

پیچیدگی زمانی :  $O(b^d)$

پیچیدگی حافظه :  $O(bd)$

6) جست و جوی دو طرفه

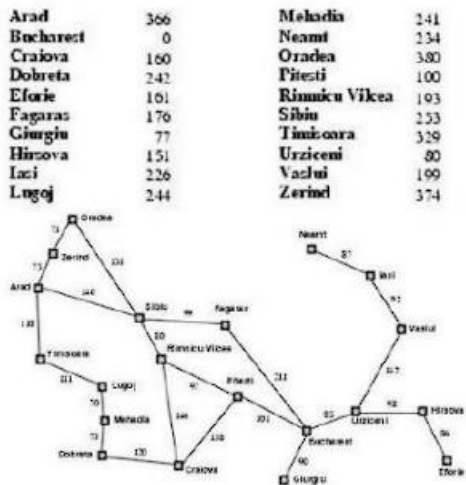
کامل بودن : بله / شرط : استفاده کردن از جست و جوی سطری

بهینه بودن : بله / شرط : استفاده کردن از جست و جوی سطری

پیچیدگی زمانی :  $O(b^{d/2})$

پیچیدگی حافظه :  $O(b^{d/2})$

جست وجوی  $A^*$  را با توجه به جدول SLD  $h$  با جست و جوی حریصانه Greedy search با رسم درختی به طور کامل توضیح داده و تفاوت ها را با دلیل ذکر کنید ؟



در این روش گره ها را با ترکیب  $g(n)$  یعنی هزینه رسیدن به گره و  $h(n)$  یعنی هزینه رسیدن از این گره به گره هدف ارزیابی می کند  $F(n) = g(n) + h(n)$ . یعنی  $f(n)$  هزینه برآورد شده ی ارزانترین جوار از طریق  $n$  است. پس باید به گره ای فکر کنیم که کمترین  $g(n)$  و  $h(n)$  را داشته باشد.

شناخته شده ترین جستجوی آگاهانه :

ایده: از بسط گرهبایی که به صرفه به نظر نمیرسند، اجتناب میکند

تابع ارزیابی :  $f(n) = g(n) + h(n)$

$g(n)$  : هزینه واقعی از گره شروع تا گره  $n$

$h(n)$  : هزینه تخمینی از گره  $n$  تا هدف

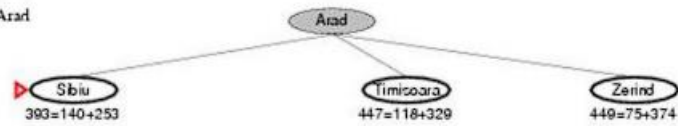
$f(n)$  : هزینه تخمینی از گره شروع تا هدف با عبور از گره  $n$

جستجوی  $A^*$  کامل وبهینه وبهینه موثر است. مرتبه زمانی و مکانی آن نمایی است.



$$f(\text{Arad}) = c(?, \text{Arad}) + h(\text{Arad}) = 0 + 366 = 366$$

After expanding Arad

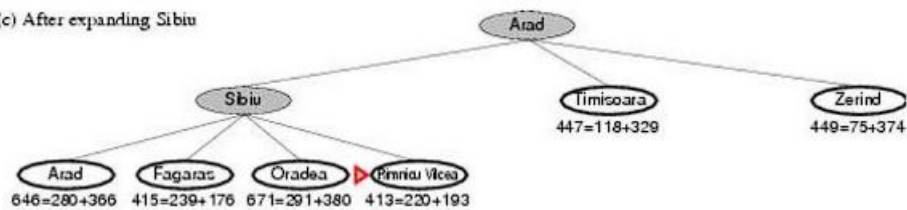


$$f(\text{Sibiu}) = c(\text{Arad}, \text{Sibiu}) + h(\text{Sibiu}) = 140 + 253 = 393$$

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

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

(c) After expanding Sibiu



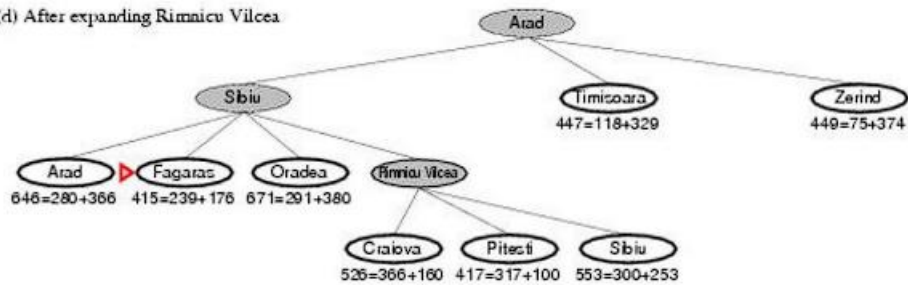
$$f(\text{Arad}) = c(\text{Sibiu}, \text{Arad}) + h(\text{Arad}) = 280 + 366 = 646$$

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

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

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

(d) After expanding Rimnicu Vilcea

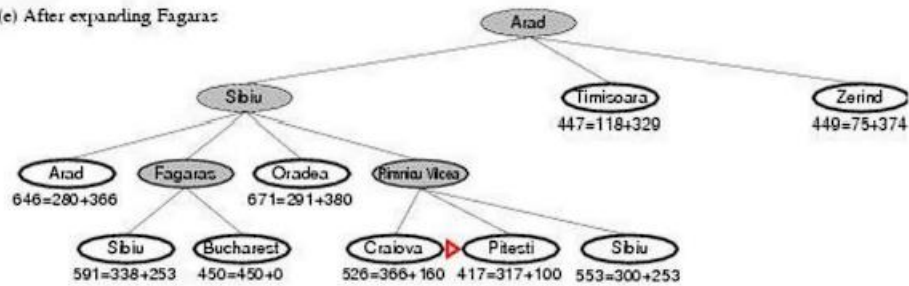


$$f(\text{Craiova}) = c(\text{Rimnicu Vilcea}, \text{Craiova}) + h(\text{Craiova}) = 366 + 160 = 526$$

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

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

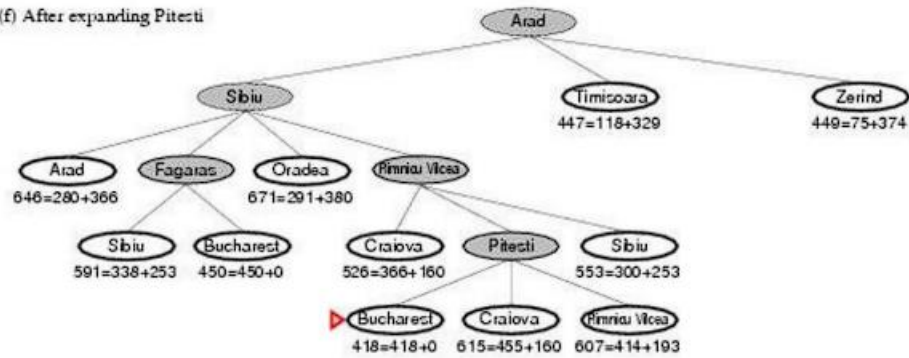
(e) After expanding Fagaras



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

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

(f) After expanding Pitesti



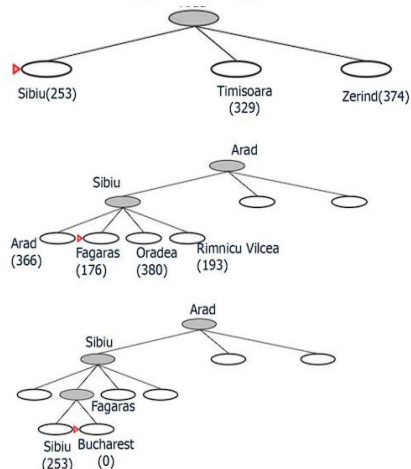
$$f(\text{Bucharest}) = c(\text{Pitesti, Bucharest}) + h(\text{Bucharest}) = 418 + 0 = 418$$

جستجوی حریصانه :  $f(n) = h(n)$  گره ای را بسط م دهد که به هدف نزدیکتر باشد. این جستجو کامل نیست چون حلقه تکرار دارد و بهینه هم نیست و مرتبه زمانی و مکانی آن  $O(b^m)$  است

Arad (366)



ریشه = حالت شروع



تفاوت الگوریتم حریصانه  $A^*$  در  $g(n)$  یعنی هزینه واقعی است  $A^*$ . جستجو را بهینه و کامل میکند. جستجوی حریصانه زود تصمیم می گیرد، اما در  $A^*$  مینیمم ترین گره انتخاب شده و به آن مینیمم هزینه واقعی اعتماد میکند.

الگوریتم زیر را شرح دهید و با توجه به جدول و شکل سوال 11 با رسم درخت جست و جو توضیح دهید؟

## Recursive best-first search

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

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  $\leftarrow$  EXPAND(node, problem)
  if successors is empty then return failure,  $\infty$ 
  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
```

این الگوریتم RBFS است که در آن :

1) بهترین گره برگ و بهترین جانشین برای آن انتخاب شود.

2) اگر مقدار بهترین گره برگ از جانشین آن بیشتر شد، آنگاه به مسیر جانشین عقبگرد شود.

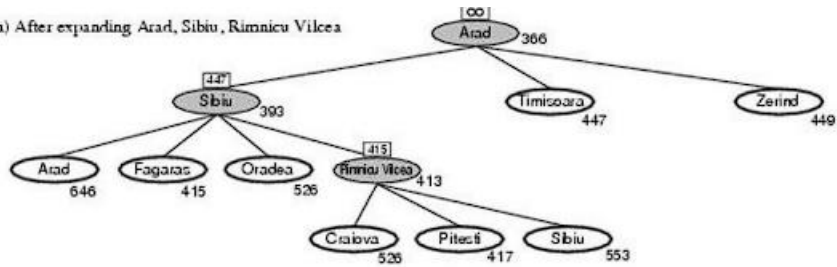
3) در حین عقبگرد، مقدار  $f(n)$  بروزرسانی شود

4) گره جانشین بسط داده شود

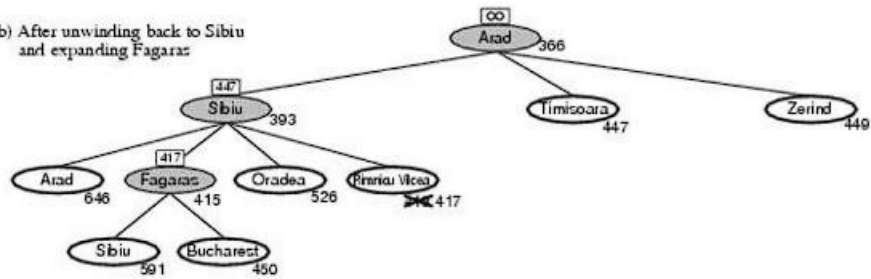
RBFS جستجوی به مراتب موثرتری از  $A^*$  ID است .

از تولید تعداد بسیار زیادی گره به دلیل تغییر عقیده رنج می برد . مانند  $A^*$  اگر  $h(n)$  قابل پذیرش باشد، بهینه است . پیچیدگی حافظه  $O(bd)$  است . پیچیدگی زمانی به کیفیت تابع هیوریستیک و میزان تغییر عقیده بستگی دارد

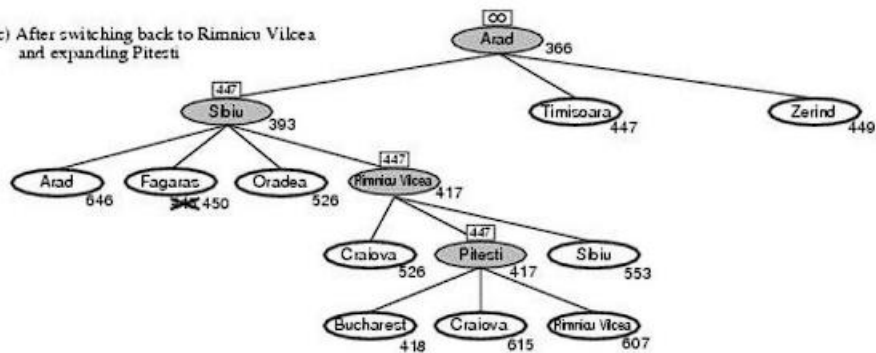
(a) After expanding Arad, Sibiu, Rimnicu Vilcea



(b) After unwinding back to Sibiu and expanding Fagaras



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





چند نوع تابع هیوریستیک را می توان برای پازل اعداد معرفی کرد ، با رسم شکل بررسی کنید ؟

$h_1$  تعداد کاشی هایی که سر جای خود نمی باشند.

$$h_1(s)=8$$

$h_2$  مجموع فاصله افقی - عمودی (منهتن) هر کاشی تا جای واقعی

$$h_2(s)=3+1+2+2+2+3+3+2=18$$

$h_3$  مجموع فاصله افقی - عمودی - قطری هر کاشی تا جای واقعی

$$h_3(s)=2+1+1+1+2+2+2+2=13$$

7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

Goal State

تابع هیوریستیک قابل پذیرش 1

از طریق نسخه ساده شده از مساله (version relax)

1h . هر کاشی می تواند به هر جایی منتقل شود

h2...هر کاشی می تواند به هر خانه همسایه منتقل شود .

ABSolver.. هزینه راه حل برای مکعب روبیک را تخمین میزند

## ابداع تابع هیوریستیک قابل پذیرش (۲)

از طریق نسخه کوچکتر از مساله (subproblem)

*	2	4
*		*
*	3	1

Start State

	1	2
3	4	*
*	*	*

Goal State

ابداع تابع هیوریستیک قابل پذیرش (3)

از طریق یادگیری از تجربه (experience from learning)

تجربه : حل تعداد بسیار زیادی از مساله

سه راه حل جهت ابداع تابع هیوریستیک نام برده و شرح دهید ؟

1) از طریق نسخه ساده شده از مساله

H 1 هر کاشی می تواند به هر جایی منتقل شود .

H 2 هر کاشی می تواند به هر خانه همسایه منتقل شود

ABSolver . هزینه راه حل برای مکعب روبیک را تخمین می زند .

2) از طریق نسخه کوچکتر از مساله

3) از طریق یادگیری از تجربه

تجربه : حل تعداد بسیار زیادی از مساله

نواع جست و جوی محلی را نام برده و ایده هر یک را بیان کنید ؟

جست و جوی تپه نوردی ، SA، پرتو محلی ، ژنتیک الگوریتم جست و جوی محلی تپه نوردی : این الگوریتم حلقه ای است که در جهت افزایش مقدار حرکت می کند (به طرف بالای تپه ) . وقتی به قله ای رسید که هیچ همسایه ای از آن بلند تر نیست خاتمه می یابد . الگوریتم جست و جوی محلی : SA این الگوریتم نسخه ای از تپه نوردی اتفاقی است و پایین آمدن از تپه مجاز است . حرکت به طرف پایین و به آسانی در اوایل زمانبندی annealing پذیرفته شده و با گذشت طمان کمتر اتفاق می افتد . الگوریتم جست و جوی پرتو محلی : نگهدار ی فقط یک گره در حافظه ، واکنش افراطی نسبت به مسئله محدودیت حافظه است . این الگوریتم به جای یک حالت ، k حالت را نگهداری می کند . این الگوریتم با k حالت که به طور تصادفی تولید شدند ، شروع می کند . در هر مرحله تمام پسین های همه حالت ها تولید می شوند . اگر یکی از آن ها هدف بود ، الگوریتم متوقف می شود ؛ وگرنه بهترین پسین را انتخاب و عمل را تکرار می کند . الگوریتم جست و جوی محلی ژنتیک : این الگوریتم شکلی از جست و جوی پرتو اتفاقی است که در آن ، حالت های پسین از طریق ترکیب دو حالت والد تولید می شوند . در مقایسه با انتخاب طبیعی ، مثل جست و جوی پرتو اتفاقی است ، با این تفاوت که اینجا با تولید مثل جنسی سروکار داریم نه غیر جنسی . این الگوریتم همانند جست و جوی پرتو محلی ، با مجموعه ای از k حالت که به طور تصادفی تولید شدند شروع می کند که به آن جمعیت گفته می شود .

الگوریتم زیر را شرح داده و انواع آن را نام برده و بررسی کنید ؟

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

**input:** *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*

الگوریتم بالا مربوط به الگوریتم جست و جوی محلی تپه نوردی می باشد . این الگوریتم حلقه ای است که در جهت افزایش مقدار حرکت می کند ( به طرف بالای تپه ) . وقتی به قله ای رسید که هیچ همسایه ای از آن بلند تر نیست خاتمه می یابد . در این الگوریتم درخت جست و جو را نگهداری نمی کند . لذا ساختمان داده گره فعلی فقط باید حالت و مقدار تابع هدف را نگهداری کند . تپه نوردی به همسایه های حالت فعلی نگاه می کند . مثل تالش برای یافتن قله کوه اورست در مه گرفتگی غلیظ ، در حالی که دچار فراموشی هستید . تپه نوردی گاهی جست و جوی محلی حریصانه نام دارد زیرا بدون اینکه قبال فکر کند به کجا برود ، حالت همسایه خوبی را انتخاب می کند . تپه نوردی معمولاً به سرعت به جواب پیش می رود ، زیرا به راحتی می تواند حالت بد را بهبود ببخشد .

انواع تپه نوردی :

تپه نوردی غیر قطعی : در بین حرکت های رو به بالا یکی به صورت تصادفی انتخاب می شود . البته احتمال انتخاب با شیب متناسب است

تپه نوردی با انتخاب اولین گزینه : گره ها تا حصول یک گره بهتر بسط داده می شوند .

تپه نوردی تصادفی : از حالت شروع مجدد تصادفی تا حصول جواب مجدداً شروع خواهد نمود.