مترجم: سهراب جلوه گر
ویرایش دوّم ، بهار ۱۳۸۸

خلاصه ی ریوس مطالب

- **جستجوی اوّل - بهترین** [۱]
- **جستجوی A^***
- **ابتکارات** [۲] **(اکتشافات)**

آگاهانه کردن جستجو

الگوریتم های قبلی ، می توانستند راه حل ها را پیدا کنند ، امّا ناکارامد بودند ؛ گره ها به صورت نمایی به وجود می آمدند . با استفاده از آگاهی هایی که در مورد ساختار مسأله داریم می توانیم کارایی را بهتر کنیم . البتّه باید توجّه کنیم که این دانش را باید از جایی دیگر بگیریم و این دانش باید درست باشد .

چرا از دانش در یک مسأله استفاده می نماییم ؟

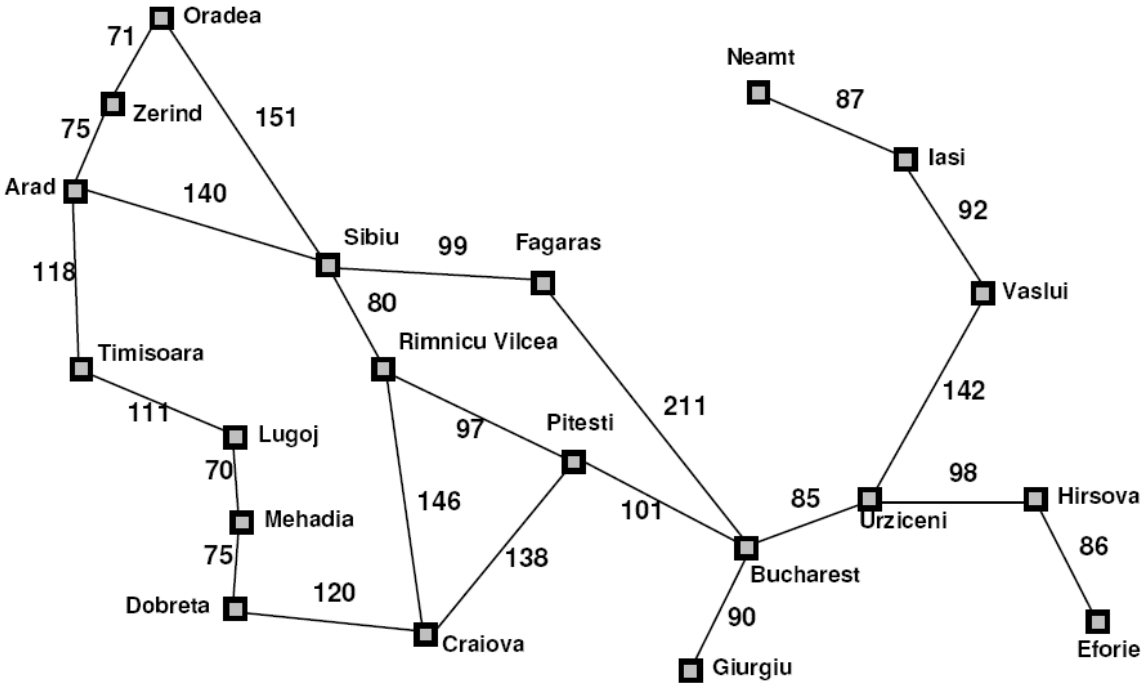
---

[۱] Best-first

[۲] Heuristics

پس ایده ی جستجوی اوّل – بهترین استفاده از یک تابع ارزیابی می باشد .

مسأله ی کشور رومانی با ارزیابی مراحل به صورت کیلومتر



آراد	۳۶۶	فاگاراس	۱۷۸	مهادیا [۱]	۲۴۱	سیبیو	۲۵۳

[۱] Mehadia

۳۲۹	تیمیسوارا [۱]	۲۳۴	نمت [۲]	۷۷	گیورگیو [۳]	۰	بخارست
۸۰	ارزیسنی [۴]	۳۸۰	ارادیا [۵]	۱۵۱	هیرسوا [۶]	۱۶۰	کرایوا [۷]
۱۹۹	واسلوی [۸]	۹۸	پیتستی [۹]	۲۲۶	یاسی	۲۴۲	دوبرتا [۱۰]
۳۷۴	زریند	۱۹۳	ریمنیکو ویلسیا	۲۴۴	لوگوج [۱۱]	۱۶۱	افری [۱۲]

توجّه کنید که اکتشاف (که به صورت جدولی در بالا داده شده است) جزیی از تعریف اوّلیّه ی مسأله نمی باشد و در این مورد به صورت یک جدول خارجی داده شده است .

---

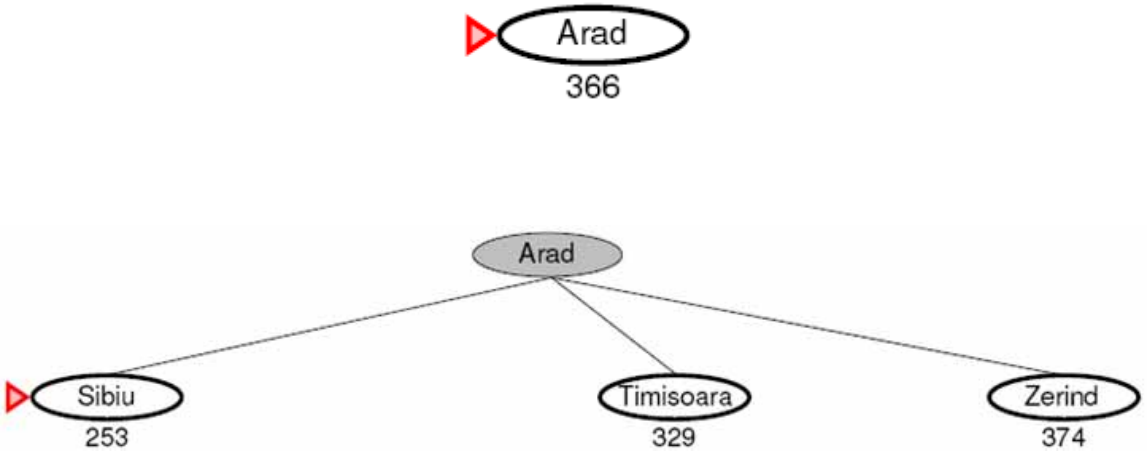
[۱] Timisoara
[۲] Neamt
[۳] Giurgiu
[۴] Urziceni
[۵] Oradea
[۶] Hirsova
[۷] Craiova
[۸] Vaslui
[۹] Pitesti
[۱۰] Dobreta
[۱۱] Lugoj
[۱۲] Eforie

مسافت خطی مستقیم به شهر بخارست

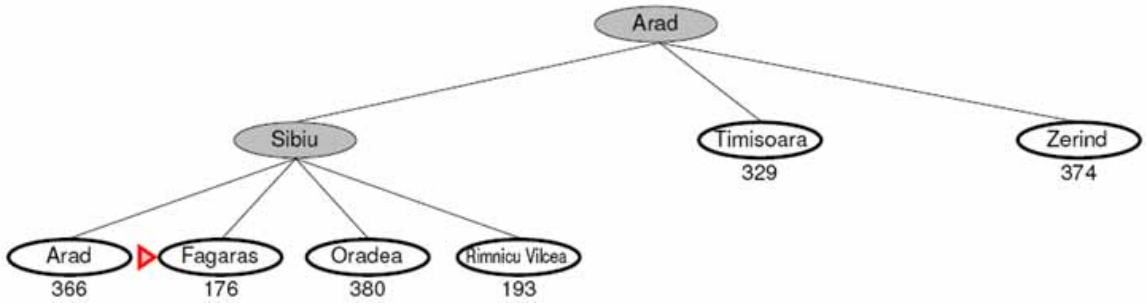
جستجوی حریصانه [۱]

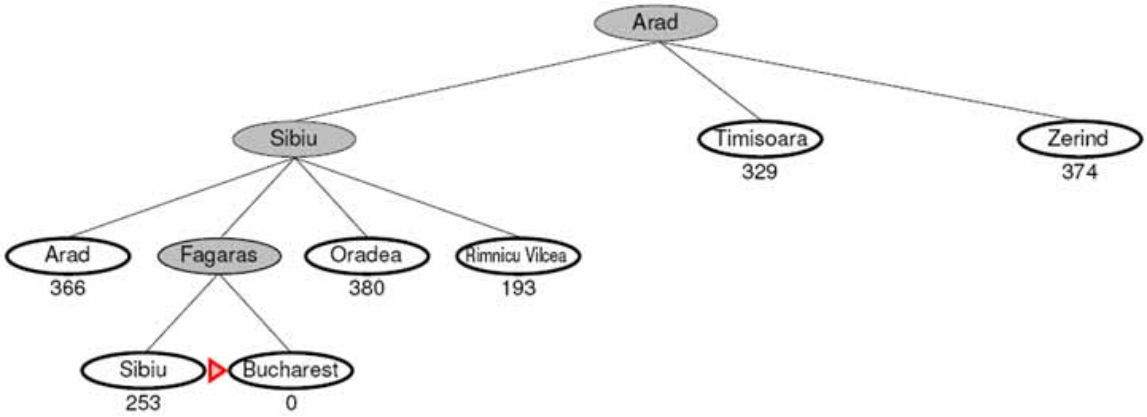
در این روش ، در ارزیابی ، فقط از مکاشفه استفاده می شود f(n)=h(n) و الگوریتم ، گرهی را که به نظر می رسد به هدف نزدیک تر است را توسعه می دهد . و تابع ارزیابی h(n) (ابتکاری) برابر است با تخمین ارزش از n تا نزدیک ترین هدف و $h_{SLD}(n)$ برابر است با فاصله ی خطی مستقیم از n تا شهر بخارست .

مثالی از جستجوی Greedy



[۱] Greedy search





خصوصیّات جستجوی حریصانه

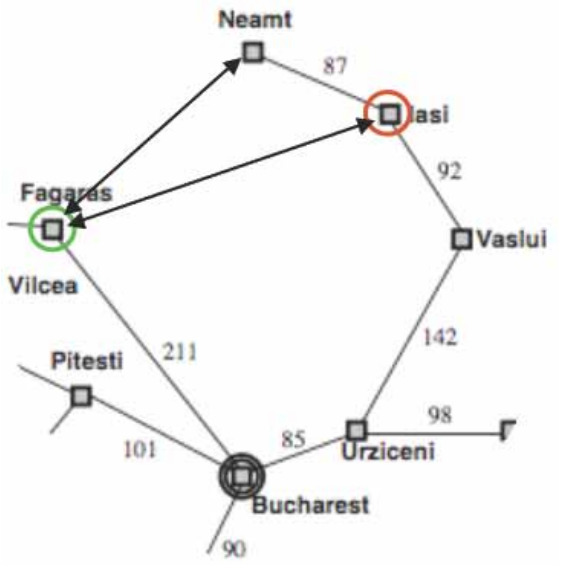
کامل بودن ؟؟ نه ممکن است در حلقه ها گیر کند ، مثلا ، اگر مقصد ما شهر Oradea باشد ، داریم :

lasi → Neamt → lasi → Neamt →

به عنوان مثالی دیگر ، اگر بخواهیم از یاسی [۱] به فاگاراس برویم ؛ میان یاسی و نمت [۱] حلقه به وجود می آید که برای رفع این مشکل باید از وضعیّت های ملاقات شده خودداری نماییم .

---

[۱] Iasi



این روش در فضاهای جستجوی محدود با تست وضعیّت تکرار شده ، **کامل** است .

زمان ؟؟ : $O(b^m)$، امّا با یک ابتکار صحیح می توان بهبودی قابل توجّهی به دست آورد .

فضا ؟؟ : $O(b^m)$ ، همه ی گره ها را در حافظه نگـه مـی دارد . کـه b ، فـاکتور انشـعاب و m عمـق درخت جستجو می باشد .

بهینگی ؟؟ نه ، یک مسیر را به سوی هدف دنبال می نماید ؛ شبیه جستجوی اوّل – عمق می باشد .

جستجوی A^*

مسأله ای که در مورد روش های جسـتجوی اوّل – بهتـرین و حریصـانه وجـود دارد ، ایـن اسـت کـه هزینه ای که مسیر تا کنون داشته است مورد توجّه قرار نگرفته است . روش جستجوی A^* از به وجـود آوردن (توسعه دادن) مسیرهایی که پرهزینه هستند اجتناب می کند .

---

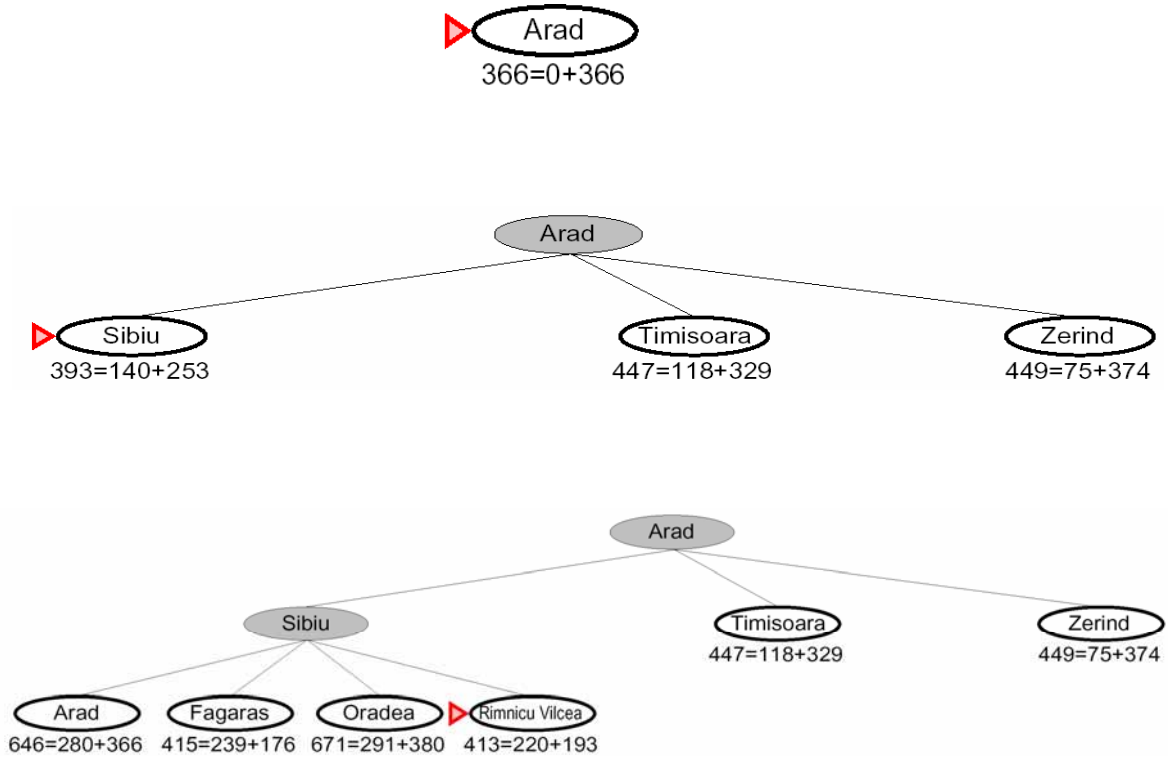
[1] Neamt

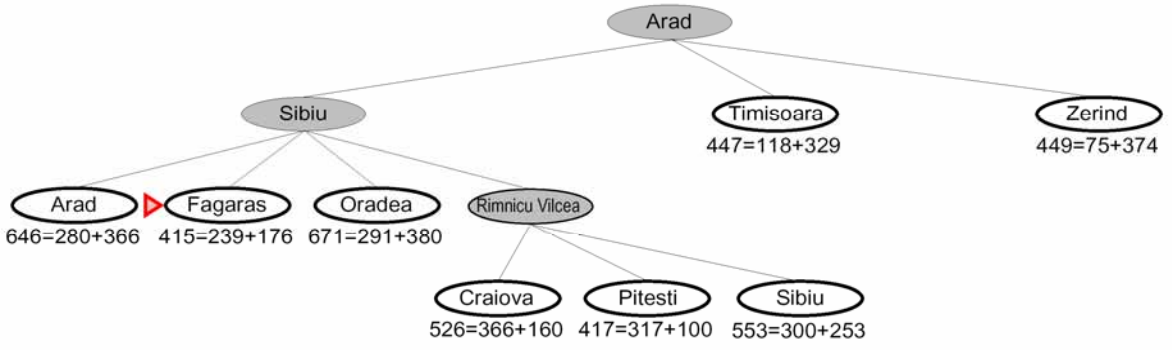
در این روش ، تابع ارزیابی $f(n)=g(n)+h(n)$ می باشد . که $g(n)$ = هزینه ای که تا کنون برای رسیدن به n صرف شده است . $h(n)$ = هزینه ی تخمین زده شده برای n و $f(n)$ = کل هزینه ی تخمین زده شده ی مسیر از n تا هدف .

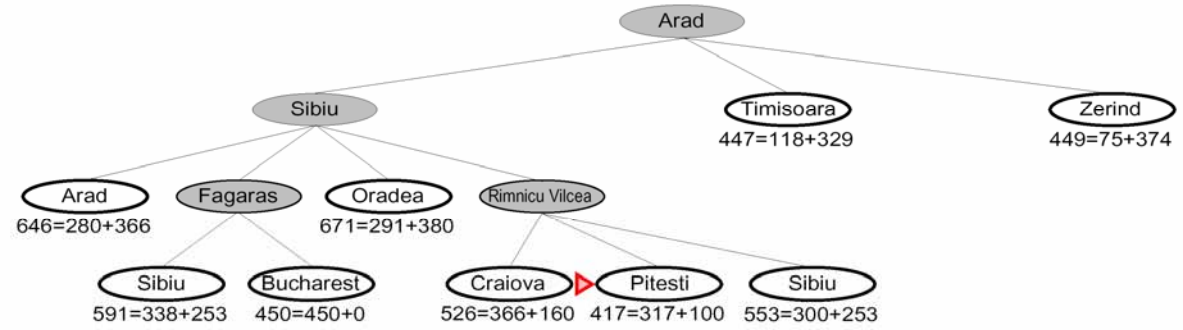
روش جستجوی A^* از یک روش ابتکاری مجاز برای جستجو استفاده می کند . در واقع ، جستجوی A^* ترکیب جستجوی با هزینه ی یکسان و جستجوی حریصانه می باشد .

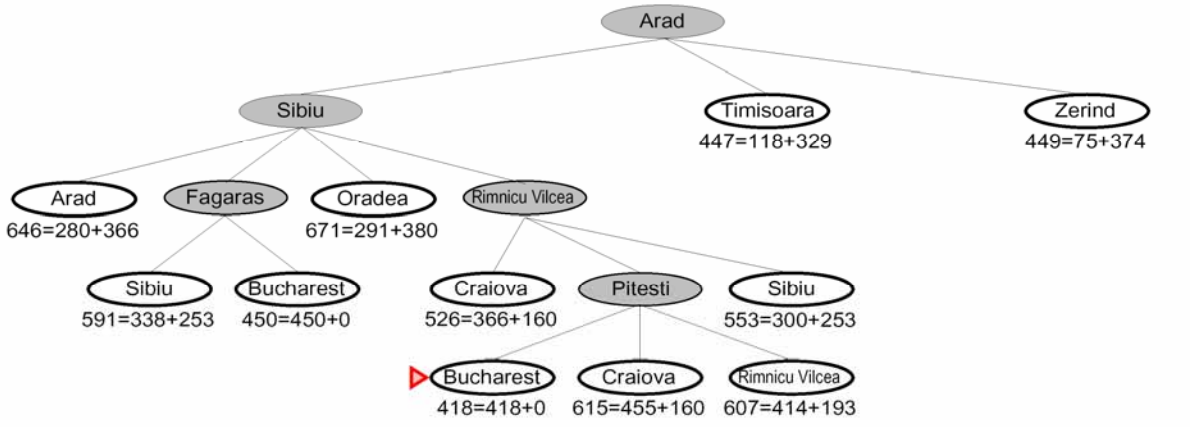
**** قضیه : جستجوی A^*، جستجویی بهینه یا مطلوب است .***

مثالی برای جستجوی A^*




Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
Craiova
526=366+160
Pitesti
417=317+100
Sibiu
553=300+253


Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
Oradea
671=291+380
Rimnicu Vilcea
Sibiu
591=338+253
Bucharest
450=450+0
Craiova
526=366+160
Pitesti
417=317+100
Sibiu
553=300+253



بهینگی جستجوی A^*

برای تضمین بهینگی مکاشفه نباید هزینه ی رسیدن به هدف را زیاد برآورد نماییم در مکاشفه ی قابل قبول داریم ؛ $h(n) \le h^*(n)$ که $h^*(n)$ هزینه ی واقعی رسیدن به هدف می باشد . در مسیریابی ، فاصله ی خطی مستقیم قابل قبول می باشد . این روش قابلیت قبول بهینگی را فقط برای الگوریتم جستجوی کلّی تضمین می کند .

الگوریتم

تابع General-Search(problem,Queuing-Fn) یک راه حل یا عدم موفّقیّت را برمی گرداند

fringe ← make-queue(make-node(initial-state[problem]))

در حلقه کارهای زیر را انجام بده

در صورتی که fringe خالی می باشد عدم موفّقیّت را برگردان

node ← Remove-Front(fringe)

در صورتی که Goal-Test[problem] به کار گرفته شده برای وضعیّت (گره) موفق شد ، گره را برگردان

fringe ← Queuing-Fn(fringe,Expand(node,Operators[problem]))

تابع Graph-Search(problem,Queuing-Fn) یک راه حل یا شکست را برمی گرداند

closed ← Φ

fringe ← make-queue(make-node(initial-state[problem]))

در حلقه کارهای زیر را انجام بده

در صورتی که fringe خالی است عدم موفّقیّت را برگردان

node ← Remove-Front(fringe)

در صورتی که Goal-Test[problem] به کار رفته برای State(node) موفّقیّت آمیز بود node را برگردان

در صورتی که $node \notin closed$ آن گاه

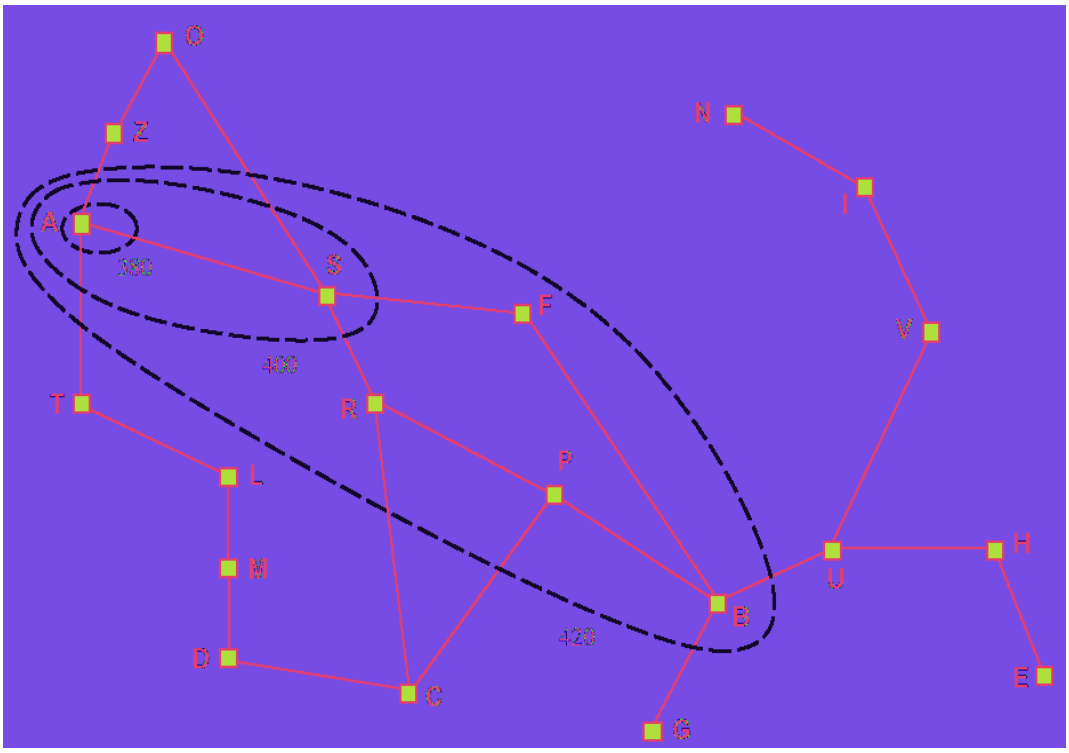
node را به closed اضافه نما

← Queuing-Fn(fringe,Expand(node,Operators[problem]))

fringe

پایان الگوریتم

سازگاری (یکنوایی) اکتشافات



خصوصیّات A^*

کامل بودن ؟؟ بله ، مگر این که تعداد نامحدودی گره با $f(n)<=f(G)$ وجود داشته باشد .

زمان ؟؟به صورت نمایی می باشد .

فضا ؟؟ تمام گره ها را در حافظه نگه می دارد و به صورت نمایی می باشد .

بهینه ؟؟ بله – تا زمانی که f_i تمام نشده است نمی تواند f_{i+1} را توسعه بدهد.

اگر C^*هزینه ی مسیر راه حل بهینه باشد ؛

- A^* تمام گره های با $f(n) < C^*$ را توسعه می دهد .

- A^* بعضی از گره هایی که $f(n) = C^*$ است را توسعه می دهد .
- A^* گره هایی که $f(n) > C^*$ است را توسعه نمی دهد .

روش A^* به طور موثری برای هر تابع ارزیابی بهینه مـی باشـد . یـک الگـوریتم ممکـن است که راه حل بهینه را در صورتی که همه ی گره های با $f(n) < C^*$ را توسعه ندهد ، گم کند .

بهترکردن A^*

A^* یکی از الگوریتم های کلیدی در هوش مصنوعی می باشد ، امّا دارای اشکال در حافظـه ی مـورد نیاز می باشد چون که تمام گره های ملاقات شده را در حافظه نگه می دارد و برای مسایل با سـاختار بـزرگ ، عملی نمی باشد ؛ برای رفع این مشکل از روش عمیق شونده ی تکراری استفاده می کنیم .

روش A^* عمیق شونده ی تکراری

در این روش ، به جای محدود کردن عمق ، به صورت تدریجی محدودیّت تابع ارزیابی افزایش داده می شود و جستجوی اوّل عمق با هزینه ی محدود شده را برای تابع ارزیابی انجام می دهد . مشکل این روش ، مقدار افزایش تدریجی محدوده ی ارزیابی است . برای رفع ایـن مشـکل یـک راه ایـن اسـت کـه مقـدار را در محدوده ی مقدار مرحله ی قبلی در نظر بگیریم و روش دیگر این است که محدوده را به وسیله ی یک مقـدار ثابت به نام ε افزایش دهیم .[1]

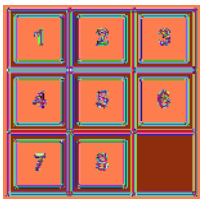
اکتشافات (ابتکارات)

پیدا کردن یک تابع مکاشفه ای خوب برای جستجو کاری بسیار مهم است . تا کنـون مـا از فاصـله ی اقلیدسی استفاده می کردیم که برای مسایل مسیریابی ، خوب بود . امّا مکاشفات را برای مسایل جدید چگونـه

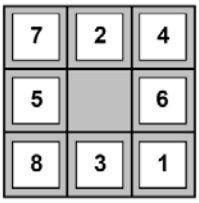
[1] برگرفته از مطالب Milos Hauskrecht استاد دانشگاه ایالت پیتسبورگ کشور ایالات متحده ی آمریکا

پیدا کنیم ؟ جواب این است که راه استانداردی وجود ندارد ، در بیش تر موارد باید مکاشفات به صورت دستی پیدا شوند .

ابتکارات قابل قبول [۱] – برای پازل ۸- تایی : $h_1(n)$ = تعداد کاشی های در جای نادرست گذاشته شده . $h_2(n)$ = مجموع فاصله تا جزیره ی مانهاتان [۲] (به تعداد کاشی (خانه) های در جای درست قرار داده شده توجّه نمایید)



حالت نهایی



حالت اوّلیّه

$h_1(S) = ??6$

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

خلاصه – توابع مکاشفه ای ، هزینه ی کوتاه ترین مسیر را تخمین می زنند. ابتکارات خوب می توانند به طور چشمگیری هزینه ی جستجو را کاهش دهند. جستجوی اوّل بهترین و حریصانه ی اوّل - بهترین پایین ترین h را توسعه می دهد و کامل نیست و همیشه هم بهینه نیست . جستجوی A^* پایین ترین g+h را

---

[۱] admissible heuristics

[۲] Manhattan

توسعه می دهد و کامل و بهینه می باشد و همچنین بهینگی موثری دارد (تا حد شکستن گره ها و برای جستجوی مستقیم) و ابتکارات قابل قبول می توانند از راه حل مستقیم مسایل مجاز مشتق شوند.

مترجم: سهراب جلوه گر
ویرایش دوّم ، بهار ۱۳۸۸

فصل پنجم

الگوریتم های جستجوی محلّی[1]

[1] local search algorithms

مترجم: سهراب جلوه گر
ویرایش دوّم ، بهار ۱۳۸۸

رئوس مطالب

- تپّه نوردی (بالا رونده از تپّه) [۱]
- گرم و سرد کردن شبیه سازی شده [۲]
- الگوریتم های ژنتیکی ، پیدایشی یا تکوینی [۳] (به طور خلاصه [۴])
- جستجوی محلّی در فضاهای پیوسته (خیلی به طور خلاصه)

مسایل بهینه سازی

---

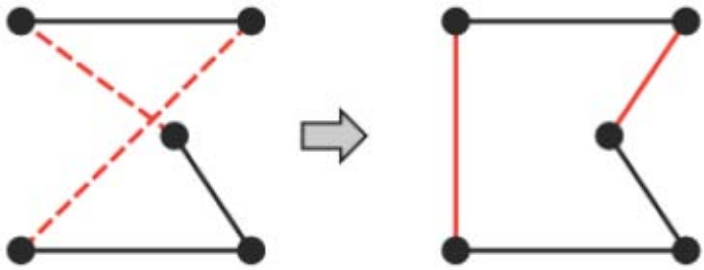
[۱] Hill-climbing
[۲] Simulated annealing
[۳] Genetic algorithms(GA)
[۴] briefly

مترجم: سهراب جلوه گر
ویرایش دوّم ، بهار ۱۳۸۸

در برخی از ساختارهای ترکیبی که ما در تلاش برای بهینه سازی آن ها هستیم ، محدودیّت ها باید در نظر گرفته شوند و تابع هزینه باید بهینه شود . اغلب پیدا کردن راه حل آسان است ، امّا پیداکردن بهترین راه حل سخت می باشد و پیداکردن همه ی راه حل های ممکن هم غیر ممکن می باشد و ما دست کم ، یک راه حل خوب را می خواهیم .

الگوریتم های بهبود یافته ی تکراری [۱] - در تعدادی از مسایل بهینه سازی ، مسیر ، نامربوط است ؛ وضعیّت (حالت) هدف ، خودش راه حل است . بنابراین ، برای فضای حالت ، مجموعه ای از پیکربندی های کامل ؛ پیکربندی بهینه ، یا ارضای محدودیّت ها ، پیکربندی و برنامه ی زمانی را پیدا کنید. در موارد مشابه ، شما می توانید از الگوریتم های بهبود یافته ی تکراری استفاده نمایید ؛ یک حالت جاری تک را نگهداری نمایید و برای بهبود آن تلاش نمایید. در فضای ثابت ، جستجوی برخط به اندازه ی جستجوی برون خط مناسب است .

مثال : مسأله ی فروشنده ی دوره گرد - با هر مسیر کامل شروع کنید و معاوضه های دو به دو را انجام دهید.



مثال : وزیر های شطرنج n – تایی [۲] ، n وزیر را در یک صفحه ی شطرنج n×n بدون این که هیچ یک از دو وزیر در سطر [۱] ، ستون [۲] همانند قرار گیرند یا به طور مورب [۳] ، همانند قرار گیرند

---

[۱] iterative improvement algorithms

[۲] n-queens

، قرار دهید . برای کاهش تعداد ناسازگاری ها ، یک وزیر را حرکت دهید . تقریبا همیشه مسایل وزیرهای چند گانه تقریبا به سرعت برای هر n بزرگ حل می شوند ، مثلا برای n = یک میلیون

جستجو در مسأله ی بهینه

دو نوع روش وجود دارد : یکی روش **سازنده** [۴] که از یک وضعیّت ابتدایی شروع کنید و به طرف یک راه حل حرکت نمایید ؛ به عنوان مثال ، در مورد مسأله ی مسافرت شخص دوره گرد ، از یک شهر شروع نمایید و یک مسیر را با ملاقات همه ی شهرها به دست آورید . و دیگری روش **تکراری** که در این روش ، با یک راه حل شروع نمایید که می تواند غلط یا غیربهینه باشد و آن را به طرف یک راه حل بهینه ببرید ؛ به عنوان مثال ، در مورد مسافرت شخص دوره گرد ، با یک مسیر شروع کنید و هزینه ی آن را با تعویض شهرها بهبود دهید .

از روش سازنده استفاده نمی کنیم ، چون که هزینه فقط به راه حل بستگی دارد ، نه به چگونگی آن ؛ روش جستجوی آگاهانه (مثل A^*) وقتی که ما یک راه حل برای همه ی مسیرها داریم خوب کار می کند . از جستجوی ناآگاهانه هم استفاده نمی کنیم (چون فضای حالت در آن خیلی بزرگ می باشد) .

جستجوی محلّی

الگوریتم هایی که تا حالا ما برّرسی کرده ایم ، تمام مسیر را از وضعیّت اوّلیه به وضعیّت نهایی نگهداری می کنند و این باعث مصرف حافظه ی زیاد یا فضای زیاد در مورد مسایل بزرگ می شود . برای

---

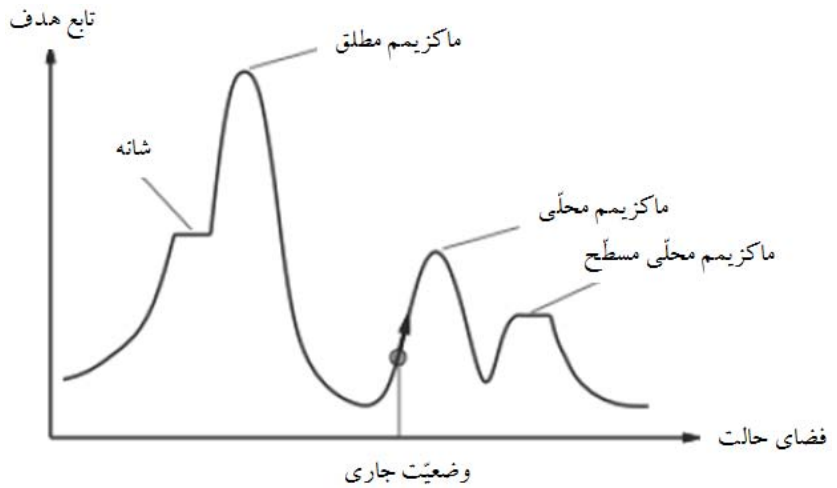
۱ row
۲ column
۳ diagonal
۴ constructive

مترجم: سهراب جلوه گر
ویرایش دوّم ، بهار ۱۳۸۸

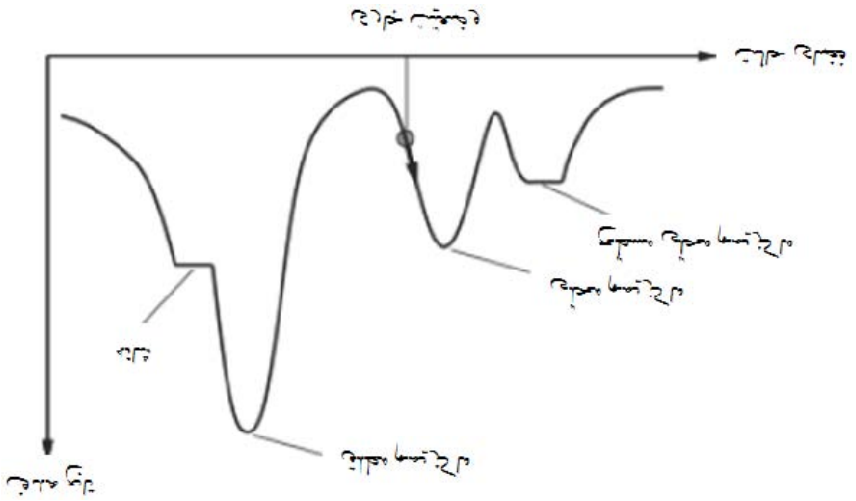
برخی از مسایل ، اطّلاعات مسیر ، ضروری (حیاتی) می باشد ؛ مثل ، مسیر یابی و پازل ۸ – تایی . در این مسایل ما هدف را می دانیم ، امّا این که چگونه به آن برسیم را نمی دانیم . برای دسته ای دیگر از مسایل ، ممکن است ما نگران چگونگی رشته ی عملکردها نباشیم و پیدا کردن راه حل بهینه ، چیزی است که مهم می باشد مثل : زمانبندی ، طرح VLSI و پنهان شناسی (رمزنویسی) [۱] . در این موارد ، ما می توانیم با اطمینان ، برخی از اطّلاعات مسیر را حذف نماییم . ***یک الگوریتم جستجو که فقط از وضعیّت جاری استفاده می کند ، یک الگوریتم جستجوی محلّی نام دارد .*** مزایای این روش عبارتند از : به حافظه ی ثابت ، نیازمندیم و می توانیم راه حل هایی باورنکردنی را در مورد فضاهای بزرگ پیدا نماییم . و معایب این روش عبارتند از : معمولاً ، بهینه نمی باشد – فقط بهینه های محلّی ، پیدا خواهند شد و می تواند به علت کمبود حافظه ، مردد باشد . در این روش ، فضای حالت شبیه روی زمین دارای تپّه ها و دره ها می باشد و ارزیابی توسط تابع هزینه (ارزیابی) داده شده است .

[۱] cryptography

فضای حالت یک بعدی [۱] برای مثال ها ، آسان تر نمایش داده می شود .



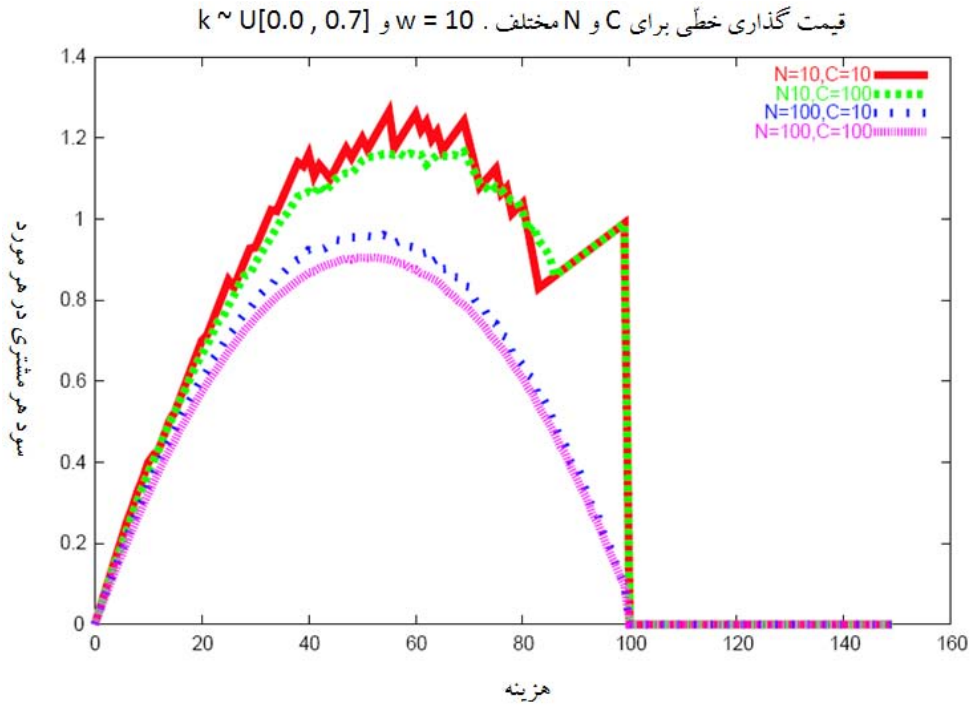
در این مورد ، ماکزیمم و مینیمم قابل تعویض اند ، برای این کار فقط مقدار را با استفاده از منفی کردن معکوس نمایید .



[۱] one-dimensional

دورنما (چشم انداز [۱]) جستجو

جستجوی محلّی ، معمولاً برای مسایل بهینه سازی ، مفید می باشد ؛ "پارامترهایی را پیدا نمایید که $o(x)$ بیشینه / کمینه باشد " ، این روش زمانی که فضای حالت ، ترکیبی از پارامترها می باشد ، مشکل است . در صورتی که n پارامتر وجود داشته باشد ، ما می توانیم یک فضای $n+1$ بعدی را تصوّر نماییم ، که n بعد اوّل ، پارامترهای تابع می باشند و $n+1$ امین بعد ، *تابع هدف* [۲] می باشد که ما این فضا را یک چشم انداز جستجو می نامیم . توجّه کنید نقاط بهینه روی تپّه ها هستند و گودی ها ، راه حل های ضعیف هستند . یک چشم انداز یک بعدی را در شکل زیر مشاهده می نمایید :

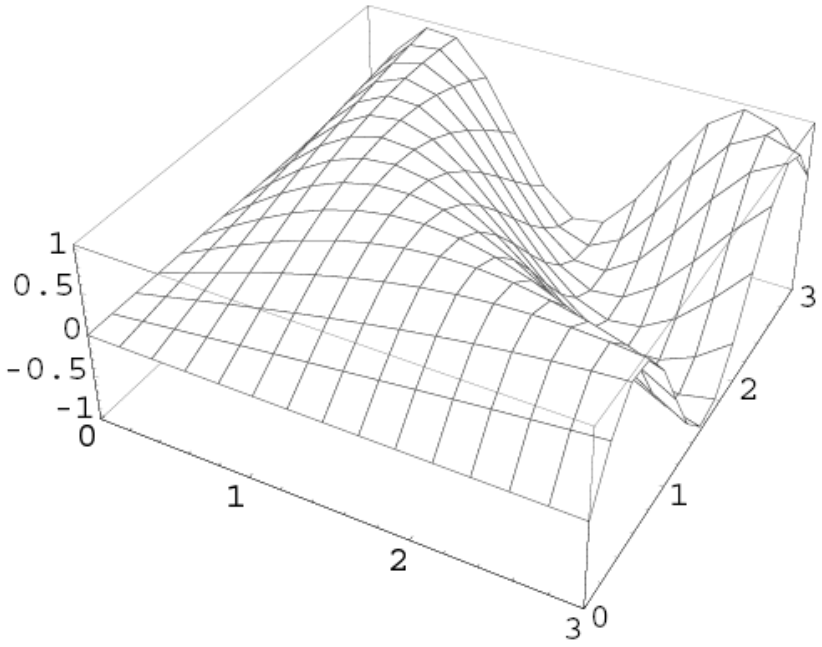


---

[۱] landscape

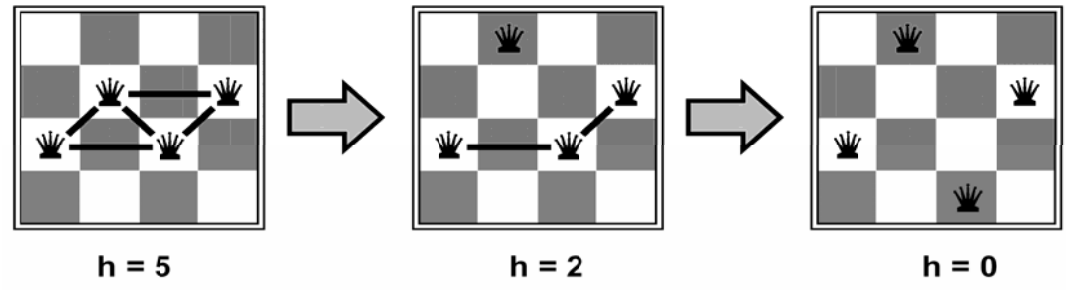
[۲] objective function

در شکل زیر یک چشم انداز دوبعدی نشان داده شده است :



دورنماها ، یک روش نمایش خوب برای الگوریتم های جستجوی محلّی هستند . بالا رفتن از یک تپّه را در نظر مجسم کنید ؛ این مورد یک راه را برای تشخیص مسایل آسان از مسایل سخت ، ارایه می کند . حالت های آسان ؛ تعداد قلّه های کم ، سطوح صاف ، بدون برآمدگی (خط الرّاس) / مسطّح (فلات) هستند و حالت های سخت : تعداد قلّه های زیاد ، سطوح دندانه دار یا ناپیوسته (گسسته) هستند . در شکل زیر که شکل مسأله ی ۴ – وزیر است وقتی که $h = 0$ می باشد ما هیچ حالت نادرستی را نداریم که در این مورد کاملاً راحت هستیم ولی وقتی که $h = 2$ است ما دارای دو حالت نادرست هستیم (چون که دو وزیر در یک ردیف قرار گرفته اند و دو وزیر دیگر به صورت مورّب قرار گرفته اند) و کار ما نسبت به حالتی که $h = 0$ بود سخت تر است و در حالت $h = 5$ چون که پنج مورد نادرست داریم کار ما نسبت به دو مورد قبلی که h

0 = و 2 = h بود سخت تر است . در حالتی که 0 = h است می توانیم تصوّر کنیم که در جایی مسطح قرار گرفته ایم و در حالتی که 5 = h است می توان تصوّر کرد که در شیب یک تپّه قرار گرفته ایم .[۱]



الگوریتم تپّه نوردی

ساده ترین شکل جستجوی محلّی ، جستجوی تپّه نوردی می باشد و دارای روشی ساده است به این ترتیب که در هر نقطه ، به جانشینان (همسایه ها) نگاه کنید و در جهت بهتر ، حرکت نمایید . این روش خیلی شبیه به جستجوی حریصانه می باشد و به حافظه ی خیلی کمی نیاز دارد . فلات می تواند سبب این شود که الگوریتم ، بی هدف ، سرگردان باشد .

جستجوی محلّی در حساب دیفرانسیل و انتگرال[۲]

ابتدا ریشه های یک معادله f(x)=0 را به دست آورید ، f ، تشخیص پذیر می باشد . با فرض یک x_1 ، $f(x_1)$ و $f'(x_1)$ را به دست آورید . از خط تانژانت برای $f(x_1)$ استفاده نمایید تا x_2 را انتخاب نمایید (شیب= $f'(x_1)$) . و برای سایر نقطه ها از فرمول $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_1)}$ استفاده نمایید ؛ این کاری که انجام دادید یک جستجوی تپّه نوردی می باشد و برای موارد مسطح ، خیلی خوب می باشد .

---

[۱] مترجم

[۲] calculus

الگوریتم

تابع Hill-Climbing(problem) یک وضعیّت را برمی گرداند

current ← Initial-State(problem)

در حلقه کارهای زیر را انجام بده

بالاترین جانشین مقداردهی شده ی current را در neighbour قرار بده

در صورتی که Value(neighbour)≤Value(current) بود current را برگردان

current ← neighbour

الگوریتم تپّه نوردی (به بیان دیگر)

تابع Hill-Climbing(problem) یک حالت که یک ماکزیمم محلّی است را برمی گرداند

ورودی ها ، problem که یک مسأله است می باشد

متغیّرهای محلّی current و neighbor که هر کدام یک گره هستند ، می باشند.

current ← Make-Node(Initial-State[problem])

کارهای زیر را انجام بده

بالاترین مقداردهی successor از current ← neighbor (مقدار بالا ترین جانشین current را در neighbor قرار می دهد)

اگر Value[neighbor] ≤ Value[current] می باشد State[current] را برگردان

current ← neighbor

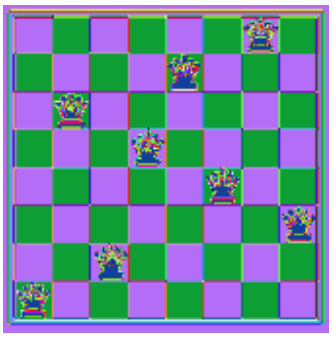
پایان حلقه

پایان الگوریتم

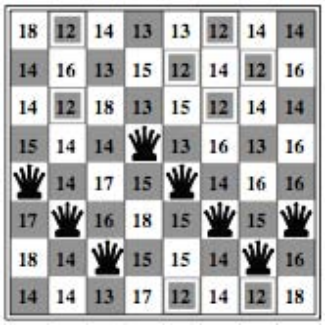
شروع مجدّد تصادفی الگوریتم بالا رونده از تپّه بر ماکزیمم محلّی غلبه می کند .

مثال جستجوی تپّه نوردی

فرمول بندی کامل برای ۸ – وزیر ، تابع جانشین : یک وزیر را به مربّع دیگر در همان ستون ببرید (۸*۷=۵۶ جانشین) که هزینه برابر است با تعداد جفت هایی که در جای نامناسب قرار دارند .



هزینه ی کمینه ی محلّی = ۱



هزینه ی فعلی =۱۷

خصوصیّات تپّه نوردی

مترجم: سهراب جلوه گر
ویرایش دوّم ، بهار ۱۳۸۸

پیاده سازی آن آسان می باشد ؛ مقدار کمی از حافظه را استفاده می نماید ولی تپّه نوردی ، زیاد تحت تأثیر شکل فضای حالت است در ماکزیمم محلّی کم خوب می باشد ولی برای فضاهای جوجه تیغی (دندانه دار) مانند بد است . در شانه ها ، ماکزیمم محلّی و خط الراس [۱] ها دچار مشکل می شود ؛ در ماکزیمم محلّی نمی تواند قله ی بالاتر را ببیند . در شانه ، راهی به طرف بیرون را نمی تواند پیدا نماید و در خط الرّاس ها برای حرکت ، شما باید به طرف پایین حرکت نمایید و حرکت های بد را مجاز می داند .

تنوعات تپّه نوردی

حرکت های غیرمستقیم [۲] : روی کف [۳] (جای مسطح) حرکت می نماید **و** در حلقه نمی افتد (یک تعداد بیشینه از حرکات را ثبت می کند)

تپّه نوردی اتّفاقی : در میان حرکت های به طرف سربالایی یکی را به تصادف انتخاب می نماید .

اوّلین انتخاب تپّه نوردی : اوّلین حرکت سربالایی را انتخاب می کند .

تپّه نوردی با شروع مجدد تصادفی : چند تپّه نوردی از وضعیّت های اوّلیه ی انتخاب شده به صورت تصادفی است و تولید وضعیّت های تصادفی می تواند غیر جزیی باشد .

بهبود روش تپّه نوردی

---

[۱] ridge

[۲] sideways move

[۳] plateau

مترجم: سهراب جلوه گر
ویرایش دوّم ، بهار ۱۳۸۸

تپّه نوردی ، می تواند خوشایند باشد ، برنامه نویسی آن آسان می باشد ، به فضای کمی نیازمند می باشد و در ضمن ، ممکن است ما ، روشی بهتر را نداشته باشیم . حال این سؤال مطرح می شود که چگونه این روش ، بهبود می یابد ؟ با استفاده از موارد زیر این روش بهبود می یابد :

- تپّه نوردی تصادفی – به صورت تصادفی حرکت های بالای تپّه را انتخاب نمایید
- تپّه نوردی با شروع مجدّد تصادفی
- تا زمانی که به یک بهینه برسید ، ادامه دهید
- یک وضعیّت اوّلیه را به صورت تصادفی ، انتخاب نمایید
- مجدّداً ، اجرا نمایید .
- بعد از n تکرار ، بهترین راه حل را نگهداری نمایید .

روش شبیه سازی گرم و سرد کردن

ضعف روش تپّه نوردی ، این می باشد که هرگز به طرف پایین تپه حرکت نمی کند و شبیه به جستجوی حریصانه ، نمی تواند "پشتیبان [1]" داشته باشد ؛ گرم و سرد کردن شبیه سازی شده ، تلاشی برای برطرف نمودن این موارد می باشد . ***ایده ی این روش ، گریختن از ماکزیمم محلّی با اجازه دادن به بعضی از حرکت های بد یا نادرست می باشد . امّا به تدریج تعداد و فراوانی [2] حرکت های بد کاهش می یابد .*** در واقع این روش تپه نوردی را با حرکت تصادفی ترکیب می نماید .

---

[1] back up

[2] frequency

گرم و سرد کردن : فلزات را با حرارت دادن آن ها و رساندن دمای آن ها به دمایی بالا و سردکردن تدریجی آن ها سخت می کنند .

اگر یک توپ پینگ پنگ را به عمیق ترین جای یک سطح دارای گودی (کمینه یا مینیمم مطلق) بیندازید ، خارج کردن آن ***سخت تر*** *از وقتی است که آن را در یک کمینه ی محلّی می اندازید .*

شبیه سازی گرم و سرد کردن

وقتی که یک فلز مثل آهن را خیلی گرم می کنید دارای ارتعاش های زیادی است ولی وقتی که به تدریج دما کم می شود ارتعاش های آن هم به تدریج کم تر می شود در تپه نوردی این کار مثل این است که از کمینه ی محلّی با اجازه دادن به برخی از حرکت های بد بگریزید ، امّا به تدریج تعداد و فراوانی آن ها را کاهش دهید .

الگوریتم شبیه سازی گرم و سرد کردن :

تابع Simulated-Annealing(problem,schedule) یک راه حل را برمی گرداند

ورودی ها ، یک مسأله و schedule که یک نگاشت از زمان به دما می باشد ، هستند .

متغیّرهای محلّی عبارتند از : current ، که یک گره است و next ، که یک گره است و T ، که یک پروب کنترل کننده ی دما از مراحل پایین تر است .

current ← Make-Node(Initial-State[problem])

برای مقادیر t = 1 تا ∞ کارهای زیر را انجام بده :

T ← schedule[t]

در صورتی که T=0 بود گره ی current را برگردان

یک جانشین به تصادف انتخاب شده از گره ی current را به گره ی next نسبت بده

$\Delta E \leftarrow$Value[next] – Value[current]

در صورتی که $\Delta E > 0$ بود محتوای گره ی next را در گره ی current بریز

در غیر این صورت ، محتوای گره ی next را فقط با احتمال $e^{\Delta(E/T)}$ در گره ی current کپی کن

پایان الگوریتم

روش شبیه سازی گرم و سرد کردن ، همیشه حرکت های خوب را می پذیرد ولی در آن احتمال پذیرش یک حرکت بد وجود دارد .

خصوصیّات شبیه سازی گرم و سرد کردن – در دمای ثابت T ، حالت کاری ممکن است به توزیع بولتزمان [۱] برسد .

$$p(x) = \alpha e^{\frac{E(x)}{kT}}$$

T با سرعت کمی کاهش داده می شود ، در نتیجه ، همیشه به بهترین حالت x^* می رسد . زیرا برای T های کوچک داریم :

$$e^{\frac{E(x^*)}{kT}} / e^{\frac{E(x)}{kT}} = e^{\frac{E(x^*)-E(x)}{kT}} >> 1$$

آیا این الزاما یک ضمانت قابل قبول است ؟؟

---

[۱] Boltzman

مترجم: سهراب جلوه گر
ویرایش دوّم ، بهار ۱۳۸۸

از روش شبیه سازی گرم و سرد کردن در بسیاری از موارد مثل قالب بندی VLSI [۱] ، زمانبندی خطوط هوایی و غیره استفاده می شود . شبیه سازی گرم و سرد کردن در صورتی که T به اندازه ی کافی با سرعت کم تر ، کاسته شود کامل و بهینه می باشد . به سادگی به الگوریتم تپه نوردی اضافه می شود . *ضعف این روش در انتخاب یک زمانبندی خنک کننده ی خوب می باشد .*

جستجوی پرتومحلّی [۲]

ایده : به جای یک حالت k ، حالت های k را نگه می دارد و بالاترین مقدار k را در بین همه ی آن ها (جانشینان یا نامزدها) انتخاب می کند . همانند جستجوهای k که در حالت موازی اجرا می شوند ، نمی باشد . جستجو می کند تا حالت های خوب دیگر جستجو ها را پیدا کند و آن ها را به هم پیوند دهد . این روش اغلب کامل است ، ولی تمام حالت های k در نهایت در بالای تپه ی محلّی قرار می گیرند ؛ در این مورد راه حل این است که جانشینان k را به طور تصادفی انتخاب نماییم . این روش ، یک روش برمبنای مسیر است که به چند مسیر " اطراف " مسیر جاری نگاه می کند و k وضعیّت را به جای یک وضعیّت در حافظه نگهداری می نماید و با k گره ی به طور تصادفی انتخاب شده شروع می کند ؛ تمام جانشینان تولید می شوند ، در صورتی که هدف پیدا شود متوقّف می شود و در غیر این صورت تعداد k تا از بهترین جانشینان انتخاب می شوند . در این روش اطّلاعات میان وضعیّت ها می توانند مشترک شوند و به وعده داده شده ترین ناحیّه ها حرکت می نماید . روش جستجوی پرتو محلّی اتّفاقی تعداد k جانشین را به صورت تصادفی انتخاب می نماید که این کار به وسیله ی احتمالی که توسط تابع ارزیابی تشخیص داده می شود انجام می شود .

الگوریتم های ژنتیکی

---

[۱] مخفف Very Large Scale Integration می باشد ؛ تعداد زیادی ترانزیستور که روی یک صفحه یا تراشه (chip) ی کوچک قرار گرفته باشند (مترجم) .

[۲] local beam search

مترجم: سهراب جلوه گر
ویرایش دوّم ، بهار ۱۳۸۸

الگوریتم های ژنتیکی می توانند به صورت یک شکل از جستجوی موازی تپه نوردی ، تصوّر شوند . ایده ی اساسی این است که برخی از راه حل ها را به صورت تصادفی ، انتخاب نمایید و برای به دست آوردن راه حل های جدید ، بهترین تکه های راه حل ها را با هم ترکیب نمایید و این کار را تکرار نمایید . در این روش جانشین (کاندید یا نامزد) ها ، تابعی از دو وضعیّت می باشند .

الگوریتم های ژنتیکی *= جستجوی پرتوی محلّی اتّفاقی + تولید جایگزین هایی از جفت حالت ها*

جستجوی پرتو محلّی به شکل " انتخاب طبیعی " : جانشینان (موجود تولید شده توسط والدین [۱]) یک وضعیّت (زیستمند [۲]) نسل بعدی را با توجّه به مقدارشان (سازگاری [۳] شان) تولید می کنند .

الگوریتم های ژنتیکی (GA ها) : الگوریتم های نمونه بعد از تکامل زیستی هستند .

در روش جستجوی پرتو اتّفاقی ، وضعیّت های جانشین به صورت تنوعاتی از دو وضعیّت والد تولید می شوند ، نه فقط یکی .

اصطلاحات الگوریتم ژنتیکی

جمعیّت [۴] ، به صورت اوّلیه مجموعه ای از k وضعیّت تولید شده به صورت تصادفی است و مجموعه ای از وضعیّت های مشتق شده است .

---

۱ offspring
۲ organism
۳ fitness
۴ population

نسل [۱] ، جمعیّت در یک نقطه از زمان است و معمولاً ، انتشار [۲] برای تمام جمعیّت همزمان [۳] می شود .

فرد [٤] ، یک عنصر از جمعیّت است و به صورت یک رشته با الفبای محدود توصیف می شود .

تابع مناسب [٥] ، هر چقدر که تابع مناسب تری انتخاب کنیم منجر به شانس های بهتر برای تولید مجدد می شود و واژه ی مناسب بودن بیان کننده ی خوبی یک راه حل می باشد .

ژنوم [٦] ، یک راه حل یا وضعیّت است .

ویژگی / ژن [۷] - یک پارامتر یا متغیّر وضعیّت است .

یک الگوریتم ژنتیکی پایه

یک جمعیّت تصادفی را تولید کن و آن را در pop قرار بده

مادامی که انجام نشده است ، کارهای زیر را انجام بده

برای هر p در pop

p را مورد ارزیابی قرار بده

---

[۱] generation
[۲] propagation
[۳] synchronize
[٤] individual
[٥] fitness function
[٦] genome ؛ تمامی ترکیب ژنتیکی یک فرد یا جمعیتی که بوسیله کروموزومها به ارث می رسد .
[۷] trait/gene

مثال crossover

s1: (100 101 110) (000 101 001 010 110) = 4 5 6 0 5 1 2 6

s2: (001 000 101) (110 111 010 110 111) = 1 0 5 6 7 2 6 7

مکان هندسی را برابر ۹ قرار دهید

s3 = (100 101 110) (110 111 010 110 111) 4 5 6 6 7 2 6 7

s4 = (001 000 101) (000 101 001 010 110) 1 0 5 0 5 1 2 6

سپس ، جهش را اعمال نمایید . با احتمال m (برای m کوچک) به صورت تصادفی ، یک بیت در راه حل را انتخاب نمایید . بعد از تولید یک جمعیّت جدید از همان اندازه به صورت جمعیّت قدیمی ، جمعیّت قبلی را حذف نمایید و دوباره شروع نمایید . crossover ، قطعات راه حل های موفق به صورت جزیی را با هم ترکیب می کند . ژن های نزدیک تر به هم ، برای ماندن با هم در رشته ی نسل ها ، بهتر می باشند . این کار ، رمز کردن را مهم می کند . جهش ، راه حل های جدید را به جمعیّت اعمال می کند . در صورتی که یک ژن از جمعیّت اوّلیه گم شده بود ، crossover نمی تواند آن را تولید کند .

اصول الگوریتم ژنتیکی

تولید مجدد توسط crossover : دو قسمت کردن [۱] والدین در یک نقطه ی crossover داده شده و ترکیب شده ؛ این کار دو وضعیّت جدید را تولید می نماید .

نقطه ی crossover : برای جلو بردن کار استفاده می شود و اغلب به صورت تصادفی انتخاب می شود ؛ برای هر دو والد باید یکسان باشد و باید نسل های قابل قبول را تولید نماید .

---

[۱] split