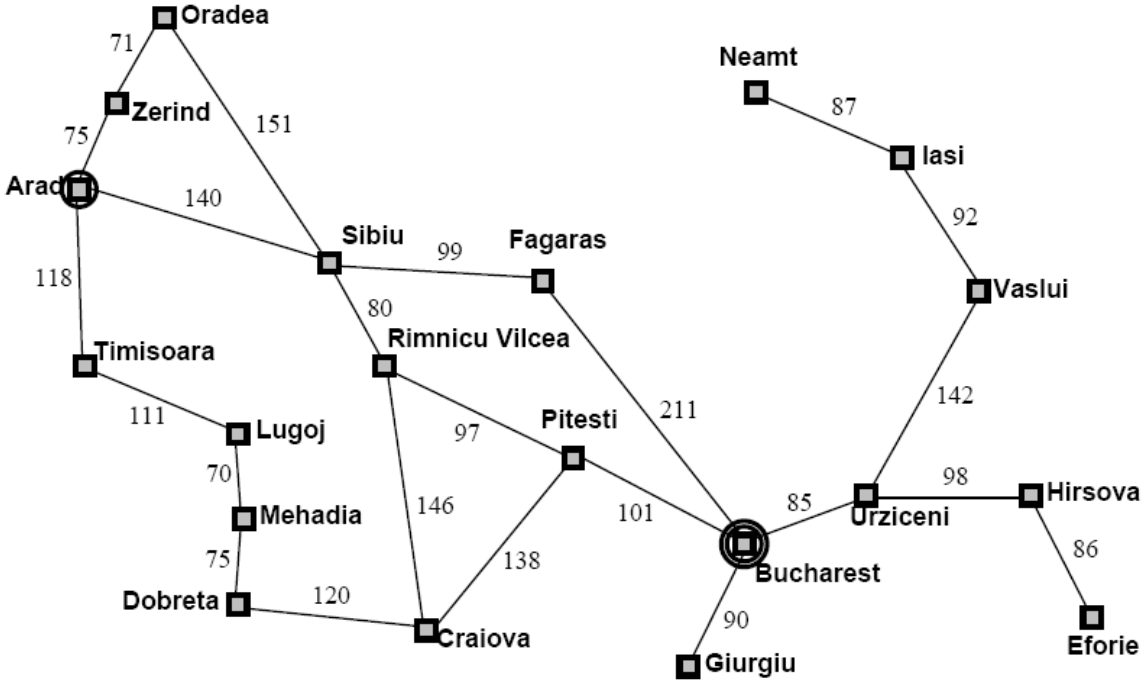
فرمول بندی مسأله [۲] : حالت ها ، شهرهای مختلف هستند و عملکردها ، حرکت بین دو شهر است .

پیداکردن راه حل : ترتیب شهرها به صورت Bucharest , Fagaras , Sibiu و Arad می باشد .

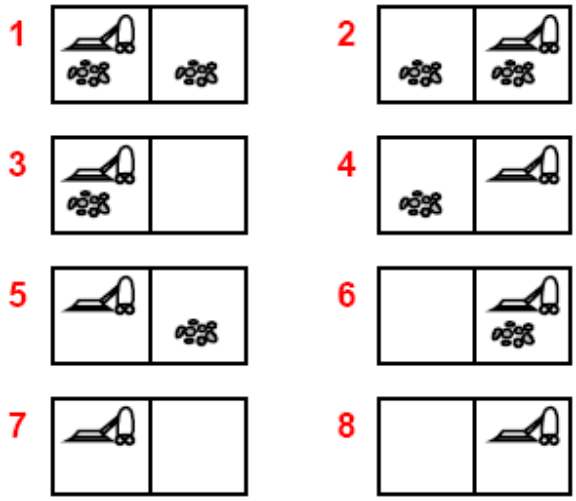


مثال : دنیای جاروبرقی – وضعیّت های عامل جاروبرقی ، تمام ترکیب های دریافت ها (درک ها) هستند و محیط کاملاً قابل مشاهده می باشد . وضعیّت های ۷ و ۸ وضعیّت های هدف می باشند چون کثیف نمی باشند . فرض می کنیم برای هر عمل ، هزینه برابر ۱ باشد .

---

[۱] Bucharest

[۲] Formulate problem



اگر در شماره ی پنج باشیم ، راه حل [Right,Suck] است . اگر در شماره های ۱ ، ۲ ، ۳ ، ۴ ، ۵ ، ۶ ، ۷ یا ۸ باشیم ، راه حل [Right,Suck,Left,Suck] می باشد . در این جا باید به این نکته توجّه کنیم که طبق قانون مرفی [۱] ، مکیدن (عمل مکش جاروبرقی) می تواند یک فرش تمیز را کثیف کند و حس کننده ی محلّی [۲] کار تشخیص کثیف بودن و فقط تعیین محل را انجام می دهد . پس راه حل این است که [اگر کثیف بود مکش کن ، راست] یا [اگر کثیف بود مکش کن ، چپ] .

فرمول بندی مسأله ی تک حالته

یک مسأله توسط چهار عنصر زیر تعریف می شود :

حالت اوّلیه : در شهر Arad هستیم .

تابع جانشین : S(x) = مجموعه ای از جفت حالات عملکرد ، به عنوان مثال ؛
S(Arad)={<Arad → Zerind,Zerind>,…}

---

[۱] Murphy's Law

[۲] local sensing

آزمایش هدف : می تواند ، صریح [۱] باشد مثلا ، " در بخارست " =x یا غیرصریح [۲] باشد ، مثلا ، NoDirect(x) باشد .

هزینه ی مسیر : مجموعه ای از فاصله ها ، تعدادی از عملیّات اجرا شده و ..... . c(x,a,y) یک هزینه ی مرحله [۳] است ، طبق پیش فرض باید بزرگ تر یا مساوی صفر باشد یا c(x,a,y)>=0 . یک راه حل ، ترکیبی از عملیّات است . سیر آن ، از حالت اوّلیه به یک حالت هدف است.

انتخاب یک فضای حالت (وضعیّت)

دنیای واقعی خیلی پیچیده است ، در نتیجه فضای حالت باید از راه حل مسأله جدا شود . حالت (مجرد) = مجموعه ای از حالت های واقعی است . عملکرد (مجزا) = مخلوطی پیچیده از عملکردهای واقعی ، مثلا ، "Arad → Zerind" و مجموعه ای پیچیده از مسیر های ممکن ، انحراف ها ، توقف ها و ..... را ارایه می کند . برای تحقق پذیری ، هر حالت واقعی در شهر آراد باید برای بعضی حالت های واقعی در شهر زریند [۴] ضمانت شود . راه حل (مجزا) = مجموعه ای از مسیرهای واقعی که راه حل هایی در دنیای واقعی هستند . هر عمل مجزا باید " آسان تر " از مسأله ی اصلی باشد.

جستجو در فضای حالت

بیش تر مسایل جستجو خیلی بزرگ تر از آن هستند که در حافظه جا گیرند . ما یک درخت جستجو را با شروع از وضعیّت اوّلیه و به کارگیری مکرر تابع جانشین (مولد) ، تولید می نماییم . ایده ی اصلی این است که از یک وضعیّت ، توجّه کنید که چه چیزهایی می تواند انجام شود . سپس توجّه کنید که از هریک

---

[۱] explicit
[۲] implicit
[۳] step cost
[۴] zerind

مترجم: سهراب جلوه گر
ویرایش دوّم ، بهار ۱۳۸۸

از وضعیّت های آن ها چه چیزهایی می توانند انجام بگیرند . در اینجا سؤالاتی به وجود می آید که برخی از آن ها عبارتند از ، آیا پیداکردن راه حل بهینه ، ضمانت شده است ؟ تا کی ، جستجو انجام خواهد شد ؟ به چه مقدار از فضا نیاز خواهیم داشت ؟

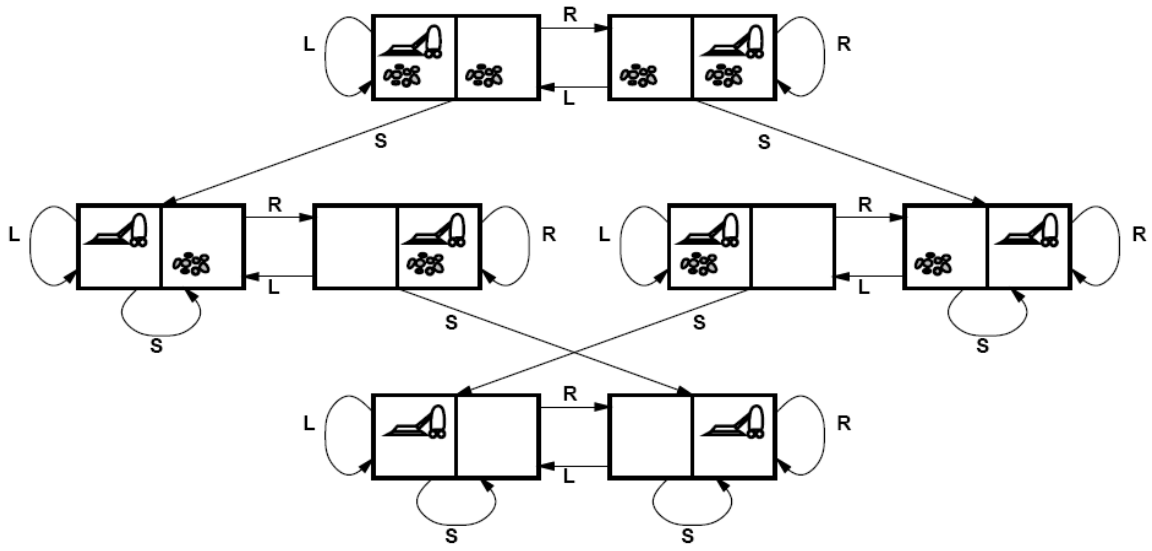
مثال : گراف فضای حالت جاروبرقی

وضعیّت ها ؟؟ کثیفی واقعی و جای روبوت (بدون توجّه به میزان کثیفی و .....)

عملکردها ؟؟ چپ (Left) ، راست (Right) ، مکش (Suck) ، بدون عملکرد (NoOp)

آزمون هدف ؟؟ بدون کثیفی بودن

هزینه ی مسیر ؟؟ در هر عملکرد یک بار (صفر برای بدون عملکرد (NoOp))



مثال : پازل هشت تایی

مترجم: سهراب جلوه گر
ویرایش دوّم ، بهار ۱۳۸۸

7	2	4
5		6
8	3	1

وضعیّت شروع

1	2	3
4	5	6
7	8	

وضعیّت هدف

حالت ها ؟؟ کاشی ها (بدون توجّه به وضعیّت های میانی)

عملکردها ؟؟ حرکت کردن در کاشی های خالی چپ ، راست ، بالا ، پایین

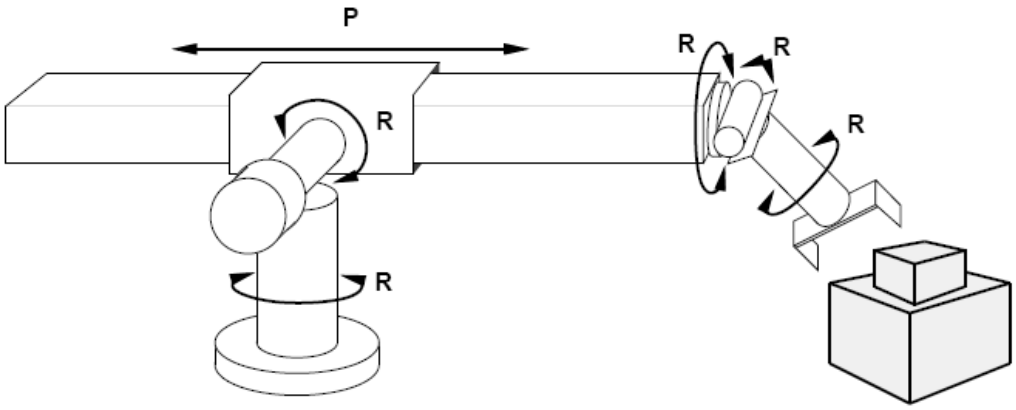
آزمایش هدف ؟؟ = وضعیّت هدف (داده شده)

هزینه ی مسیر ؟؟ در هر حرکت یک بار

نکته : راه حل معمولی پازل n – تایی ، NP-hard می باشد. مسایل NP-hard مسایلی هستند که در یک زمان چند جمله ای نمی توانند حل شوند . [1]

مثال : سرهم بندی روبوتیک

[1] lorrie.cranor.org/pubs/diss/node1.html



حالت ها ؟؟ مختصات [۱] مقدار دهی صحیح شده ی [۲] تکه های متصل شده ی روبوت برای سرهم بندی

عملکردها ؟؟ حرکات [۳] پیوسته ی [۴] اتصالات [۵] روبوت

آزمایش هدف ؟؟ کامل کردن سرهم بندی **بدون این که ربوت را شامل شود!**

ارزیابی مسیر ؟؟ زمان اجرا کردن

تعریف : **لیستی از گره ها که تولید شده اند امّا هنوز توسعه داده نشده اند را fringe می نامیم .**

الگوریتم های جستجوی درختی

---

[۱] coordinates
[۲] real-valued
[۳] motions
[۴] continuous
[۵] joints

الگوریتم :

تابع TREE-SEARCH(problem , sterategy) یک راه حل یا عدم موفّقیّت را برمی گرداند .

در ابتدا درخت جستجو از حالت اوّلیه ی مسأله استفاده می کند .

در حلقه

اگر کاندید یا داوطلبی برای توسعه نبود عدم موفّقیّت را برگردان

یک گره [۱] برگ [۲] را با توجّه به استراتژی یا روش انتخاب کن

اگر محتوای گره یک حالت هدف بود ، یک راه حل متناظر را برگردان

در غیراین صورت گره را توسعه بده و گره های منتج را به درخت جستجو اضافه کن

پایان حلقه

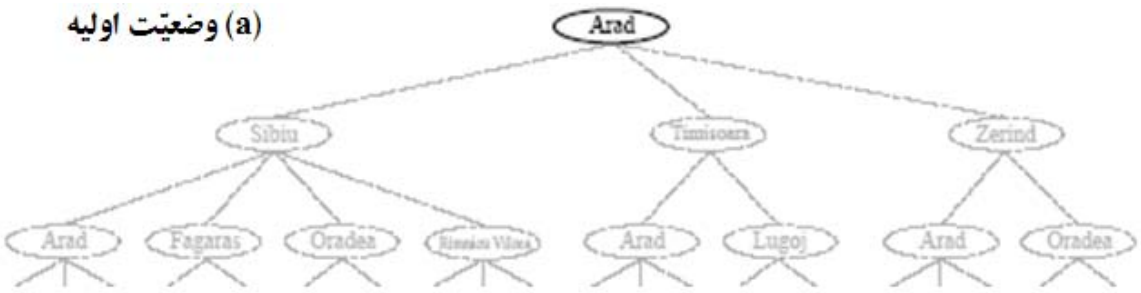
مثال جستجوی درختی

درخت جستجو ؛ در گراف جستجو ، یک وضعیّت می تواند از چند مسیر قابل دسترسی باشد و ریشه یک گره جستجو متناظر با وضعیّت اوّلیه (آراد) می باشد

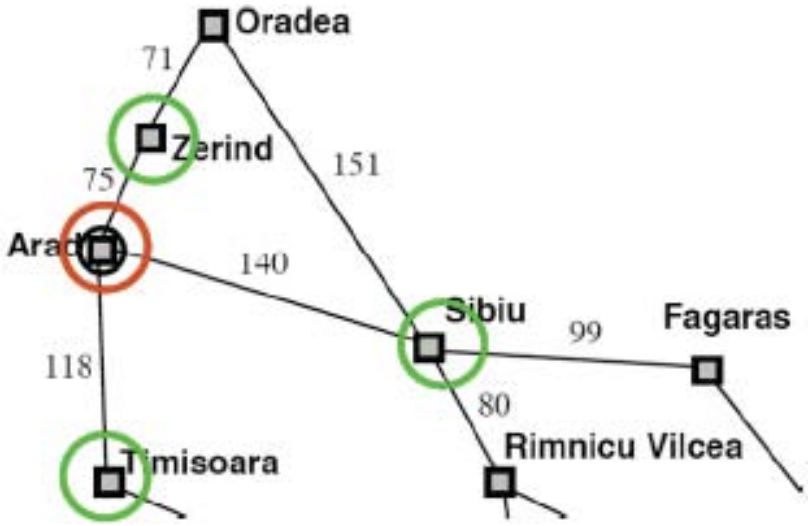
---

[۱] node

[۲] leaf

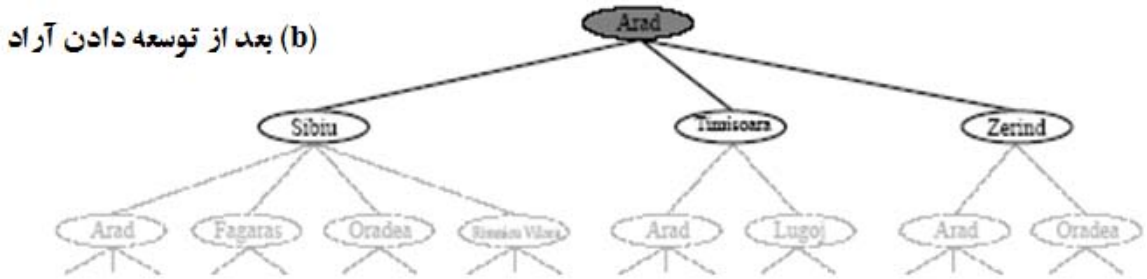


شهرهای متصل شده به وضعیّت اوّلیه عبارتند از : سیبیو [۱] ، تیمیسوارا [۲] و زرینـد و کاندیدهای حرکـت بعدی در شکل زیر نشان داده شده اند :

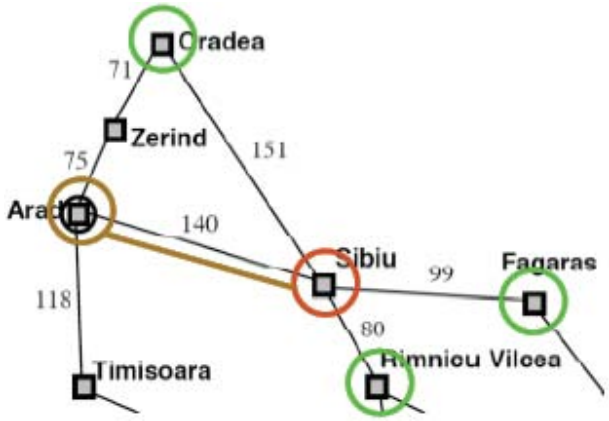


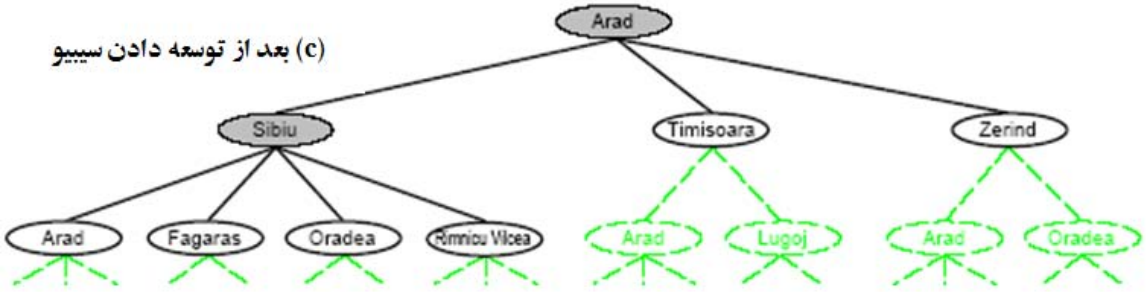
[۱] Sibiu

[۲] Timisoara



کاندیداهای حرکت بعدی عبارتند از : آراد ، فاگاراس [۱] ، اُرادیا [۲] و ریمنیکو ویلک [۳]





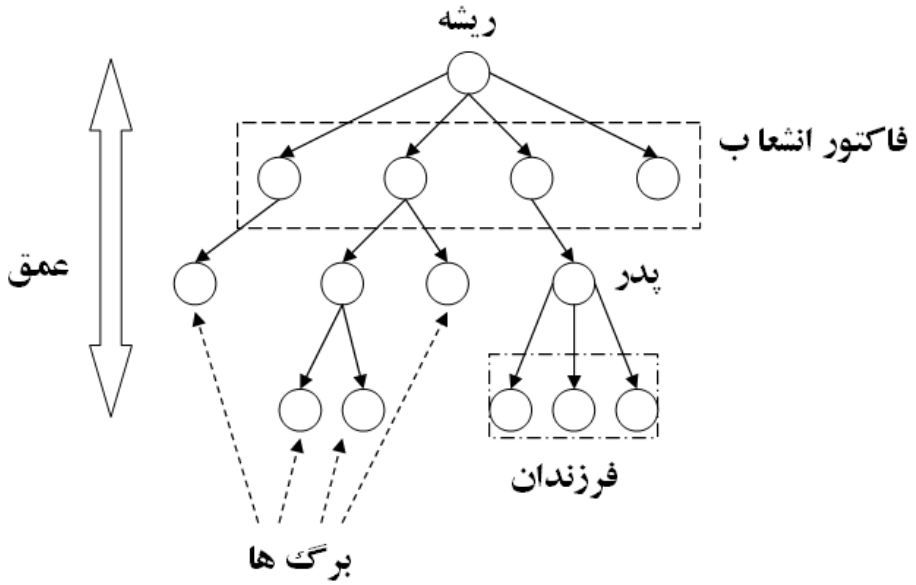
[۱] Fagaras

[۲] Oradea

[۳] Rimnicu Vilc

درخت ها

یک درخت ، یک گراف [۱] بدون حلقه ی [۲] مستقیم [۳] می باشد . گراف مستقیم ، مجموعه ای از گره های متصل شده توسط لبه ها (یال ها) [۴] می باشد . مسیر ، رشته ای از لبه ها (که امکان دارد تکرار هم شده باشند) است . در درخت ، حداکثر یک مسیر متصل کننده ی هر جفت از گره ها وجود دارد (بدون حلقه است) . درخت ها به دلیل این که رفتار الگوریتمی خوبی دارند به صورت گسترده ای در علم کامپیوتر استفاده می شوند و دارای قدرت زیاد برای تعریف الگوریتم های بازگشتی می باشند . در زیر تصویری از یک درخت را مشاهده می نمایید :



[۱] graph
[۲] acyclic
[۳] direct
[۴] edges

توجّه کنید که گره ها حداکثر دارای یک پدر هستند و ریشه ، گره ای متمایز است که دارای پـدری نمی باشد در ضمن ، عمق هم که برای خاتمه ی الگوریتم ، مهم است می تواند نامحدود باشد .

الگوریتم جستجوی درختی

الگوریتم

تابع General-Search(problem,strategy) یک راه حل یا عدم موفّقیّت را به وجود می آورد

درخت جستجو را با استفاده از حالت اوّلیه ی مسأله مقداردهی اوّلیه می نماید

در حلقه کارهای زیر را انجام بده

در صورتی که داوطلبی برای توسعه وجود ندارد ، عدم موفّقیّت را برگردان

یک گره برگ را برای توسعه با توجّه به روش انتخاب نما

در صورتی که گره شامل یک وضعیّت هدف می باشد راه حل متناظر را برگردان

در غیر این صورت گره را توسعه بده و گره های نتیجه شده را به درخت جستجو اضافه نما

پایان الگوریتم

روش : روش جستجو براساس ترتیبی که براساس آن گره هـا توسـعه داده مـی شـوند مشـخص مـی شود.

وضعیّت ها و گره های درخت

یک حالت ، یک پیکربندی فیزیکی (نمایشی از یک پیکربندی فیزیکی) است . یک گره ، ساختمان داده ای است که جزء اصلی یک درخت جستجو ، شامل والد [۱] ، فرزندان [۲] ، عمق [۳] و ارزیابی مسیر g(x) است . **حالت ها یا وضعیّت ها** ، پدر (والد) ، فرزند ، عمق یا ارزیابی مسیر ندارند . از یک تابع مثل تابع EXPAND برای ساختن گره های جدید می توانیم استفاده می کنیم و از SuccessorFn می توان برای ساختن حالت های مورد نظر در مسأله استفاده کرد .

الگوریتم جستجوی درخت عمومی

الگوریتم

تابع Tree-Search(problem,fringe) یک راه حل و یا اشکال را برمی گرداند .

fringe ← Insert(Make-Node(Initial-State[problem]),fringe)

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,State(node)) ، گره را برگردان

fringe ← InsertAll(Expand(node,problem),fringe)

---

[۱] parent

[۲] children

[۳] depth

تابع Expand(node,problem) مجموعه ای از گره ها را برمی گرداند

یک مجموعه ی خالی ← successors

برای هر عملکرد و نتیجه در Successor-Fn(problem,State[node]) کارهای زیر را انجام بده

یک گره ی (Node) جدید ← s

Parent-Node[s] ← node ; Action[s] ← action ; State[s] ← result

Path-Cost[s] ← Path-Cost[node]+Step-Cost(node,action,s)

Depth[s] ← Depth[node]+1

S را به successors اضافه کن

Successors را برگردان

پیاده سازی الگوریتم های جستجو

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

اگر fringe خالی می باشد عدم موفّقیّت را برگردان

node ← Remove-Front(fringe)

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

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

Queuing-Fn(queue,elements) یک تابع صف بندی است که مجموعه ای از عناصر را به صف وارد می نماید و ترتیب توسعه ی عناصر را تشخیص می دهد . تنوعات تابع صف بندی ، تنوعات الگوریتم جستجو را به وجود می آورد .

روش های جستجو

یک روش با انتخاب ترتیبی از توسعه ی گره تعریف می شود .

ارزیابی روش ها :

تمامیّت : اگر راه حلی وجود داشته باشد همیشه روش ، آن را پیدا می نماید ؟

پیچیدگی زمانی : تعداد گره های تولید شده / توسعه داده شده

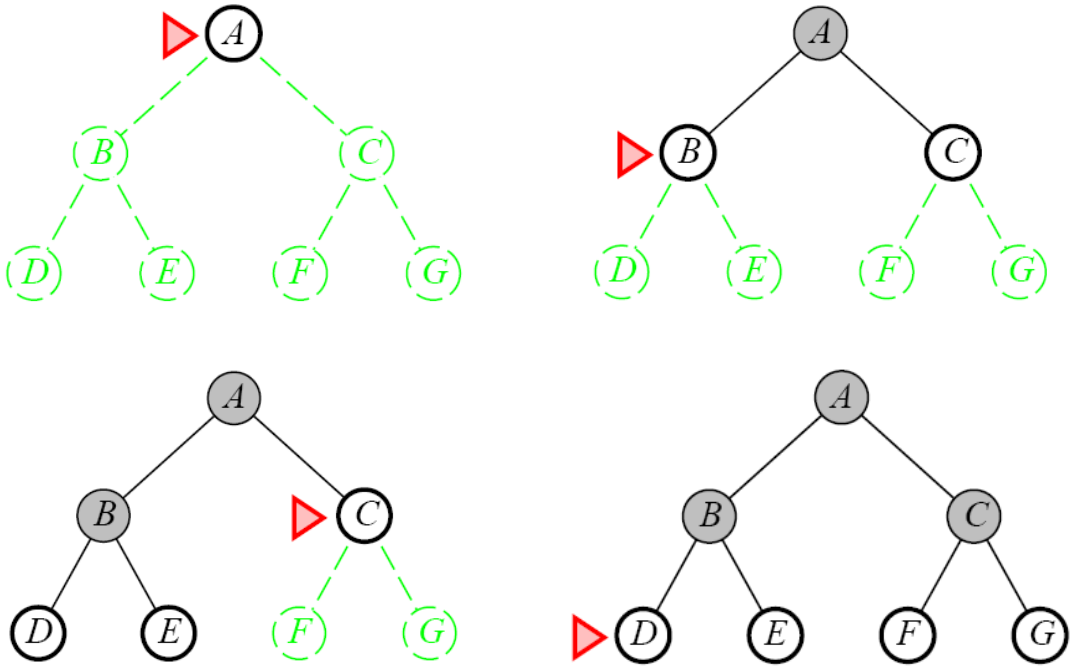
پیچیدگی فضایی : بیشینه ی تعداد گره های موجود در حافظه

بهینگی : آیا همیشه روش ، کم هزینه ترین راه حل را پیدا می کند ؟

پیچیدگی زمانی و فضایی با این موارد ارزیابی می شوند : بیشینه ی فاکتور انشعاب درخت جستجو ، عمق کم هزینه ترین راه حل ، بیشینه ی عمق درخت جستجو که شاید بی نهایت باشد .

جستجوی نا آگاهانه

ساده ترین الگوریتم های جستجو ، آن هایی هستند که هیچ اطّلاعات اضافی که در شرح مسأله وجود دارد را استفاده نمی نمایند ، ما این روش ها را روش های جستجوی نا آگاهانه می نامیم . برخی اوقات



مثال دیگر : آراد به بخارست

- آراد را از صف بردارید
- سیبیو ، تیمیسوارا و زرینند را به صف ، اضافه نمایید
- سیبیو را از صف بردارید و تست نمایید
- ارادیا ، فاگاراس ، ریمنیکو ویلسیا را به صف ، اضافه نمایید
- تیمیسوارا را از صف بردارید و تست نمایید
- لوگاج را به صف ، اضافه نمایید

- ...

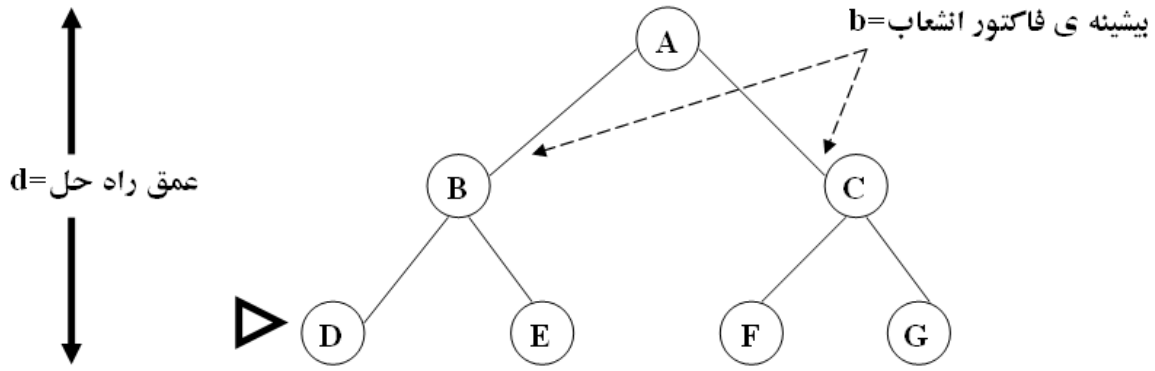
برخی نکات ریز

چگونه ما از ملاقات مجدد آراد جلوگیری نماییم ؟ با استفاده از لیست closed ، یک لیست از وضعیّت های توسعه داده شده را نگهداری نماییم . توجّه کنید که راه حل ما یک مسیر می باشد ، نه یک شهر.

چگونه ما از وارد کردن دوباره ی ارادیا ، اجتناب نماییم ؟ open-list : یک لیست از وضعیّت هایی که تولید شده اند امّا ، توسعه داده نشده اند .

چرا ما آزمون هدف را در زمانی که فرزندان را تولید کردیم ، به کار نبردیم ؟ در واقع ، هیچ تفاوتی ندارد . گره ها ملاقات می شوند و به همان روش یا راه ، تست می شوند .

خصوصیّات جستجوی اوّل سطح



کامل ؟؟ آیا جستجوی اوّل سطح ، راه حل را پیدا می کند ؟ بله (در صورتی که b محدود باشد)

توضیح : تصوّر نمایید که راه حل در عمق n می باشد . از آنجایی که همه ی گره ها یا در عمق n یا در بالای n ، قبل از هر چیزی در عمق n+1 ملاقات می شوند ، یک راه حل ، پیدا خواهد شد .

زمان ؟؟ جستجوی اوّل سطح ، به زمان اجرای $O(b^{d+1})$ نیاز دارد . که در آن b ، فاکتور انشعاب ، میانگین تعداد فرزندان می باشد . جستجوی اوّل سطح $1+b+b^2+b^3+...+b^d+b(b^d-1)=O(b^{d+1})$ گره را ملاقات خواهد کرد (به d و چگونگی آن در فرمول توجّه نمایید)

فضا ؟؟ $O(b^{d+1})$ (هر گره را در حافظه نگهداری می کند)

توضیح : جستجوی اوّل سطح ، باید همه ی درخت جستجو را در حافظه نگهداری نماید (زیرا ما می خواهیم رشته ی عملکردها را برای رسیدن به هدف ، بدانیم) .

بهینگی ؟؟ اگر چند راه حل وجود داشته باشد ، آیا جستجوی اوّل سطح ، بهترین راه حل را پیدا می کند ؟ بله

توضیح : جستجوی اوّل سطح ، کم عمق ترین راه حل در درخت جستجو را پیدا می نماید . در صورتی که هزینه ی مراحل ، یکسان باشند ، این روش بهینه خواهد بود ، در غیر این صورت ، لزوماً بهینه نخواهد بود .

- آراد ← سیبیو ← فاگاراس ← بخارست در ابتدا پیدا خواهند شد (فاصله = ۴۵۰)
- آراد ← سیبیو ← ریمنیکو ویلسیا ← پیتستی ← بخارست ، کوتاه تر می باشد . (فاصله = ۴۱۸)

فضا ، مسأله ای بزرگ است ، چون که باید گره ها در حافظه نگهداری شوند ؛ فضای زیادی اشغال می کند . می تواند گره ها را با سرعت ۱۰۰ مگابایت در ثانیه [۱] تولید کند بنابراین می تواند در بیست و چهار (۲۴) ساعت هشت هزار و ششصد و چهل گیگا بایت (۸۶۴۰) تولید نماید. در کل ، فضای مورد نیاز جستجوی اوّل سطح ، مسأله ای بزرگ تر از زمان مورد نیاز می باشد .

---

[۱] 100MB/sec

مترجم: سهراب جلوه گر
ویرایش دوّم ، بهار ۱۳۸۸

جستجوی با هزینه ی یکسان

به یاد بیاورید که جستجوی اوّل سطح در زمانی که هزینه ی مراحل ، غیر یکسان باشد ، غیربهینه می باشد . ما می توانیم این اشکال را رفع نماییم برای این کار اوّل کوتاه ترین مسیرها را توسعه می دهیم و یک هزینه ی مسیر را به گره های توسعه داده شده ، اضافه می کنیم . از یک صف اولویّت هم برای منظم کردن آن ها به صورت افزایشی بر اساس هزینه ی مسیر ، استفاده می نماییم . این روش ، کوتاه ترین مسیر را پیدا خواهد کرد . در صورتی که هزینه ها ، یکسان باشند ، این روش با جستجوی اوّل سطح ، برابر می باشد . پس ، جستجوی با هزینه ی یکسان ، کم هزینه ترین گره توسعه داده نشده را توسعه می دهد . در پیاده سازی ، ریشه ، صفی با ارزیابی مسیر است که کم ترین در اوّل صف قرار دارد .

خصوصیات جستجوی با هزینه ی یکسان

کامل ؟؟ بله ، اگر $\text{cost} \geq \epsilon$.

زمان ؟؟ تعداد گره ها در صورتی که g کوچک تر یا مساوی هزینه ی راه حل مطلوب باشد و $O(b^{\lceil C^*/\epsilon \rceil})$ در جایی که C^* راه حل مطلوب باشد خوب است .

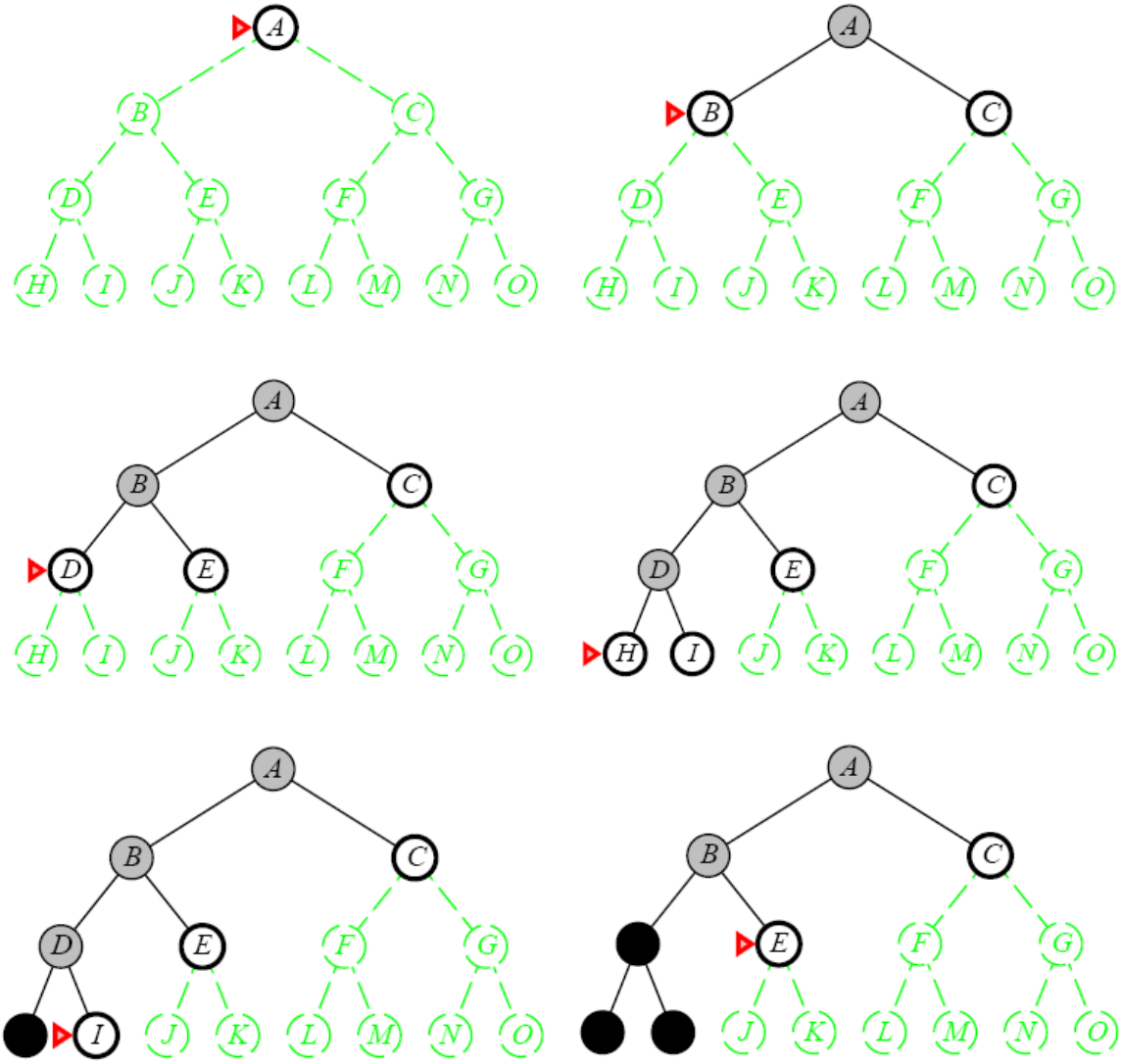
فضا ؟؟ تعداد گره ها در صورتی که مقدار g کوچک تر یا مساوی هزینه (بها) ی راه حل مطلوب باشد و $O(b^{\lceil C^*/\epsilon \rceil})$

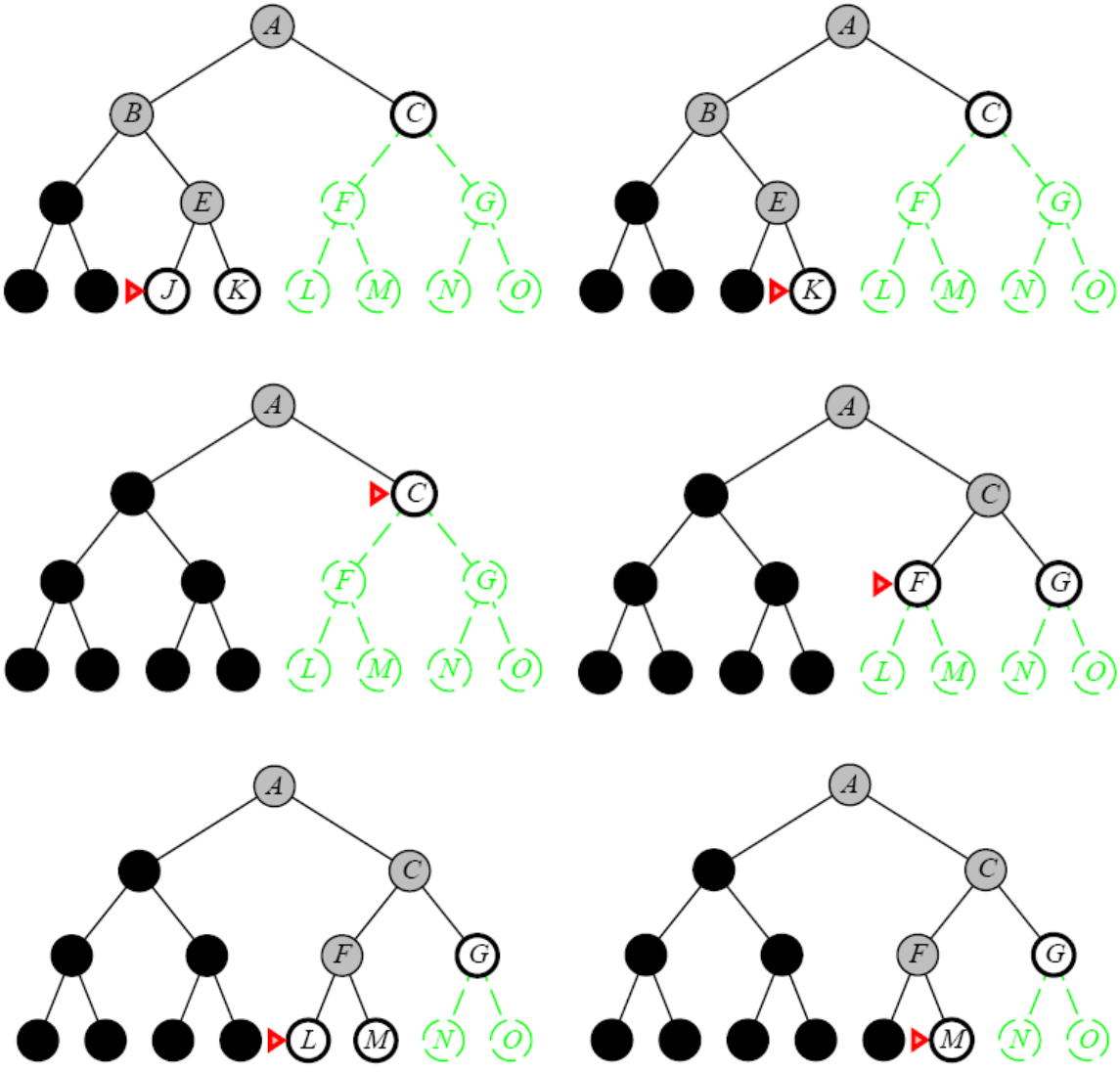
بهینگی ؟؟ بله – گره ها را با افزایش مقدار $g(n)$ توسعه می دهد. عمیق ترین گره ی توسعه داده نشده را توسعه می دهد.

برای پیاده سازی ، ریشه ، صفی LIFO است و جانشین ها را [به ترتیب] در جلو قرار می دهد.

جستجوی اوّل عمق

مترجم: سهراب جلوه گر
ویرایش دوّم ، بهار ۱۳۸۸





مثال دیگر : آراد به بخارست (به صورت پشته)

- آراد را از پشته بردارید (pop)
- سیبیو ، تیمیسوارا و زرند را به پشته اضافه نمایید (push)

- سیبیو را از پشته بردارید و تست نمایید
- ارادیا ، فاگاراس و ریمنیکو ویلسیا را به پشته اضافه نمایید
- ارادیا را ازپشته بردارید و تست نمایید
- فاگاراس را از پشته بردارید و تست نمایید
- بخارست را به پشته اضافه نمایید
- ...

خصوصیّات جستجوی اوّل عمق

کامل بودن ؟؟ نه : در مواردی که عمق نامحدود است و در مورد حلقه ها درست کار نمی کند . بازنگری ، برای جلوگیری کردن از حالت های تکرار شده در طول مسیر منجر به کامل شدن در حالت های محدود می شود . به عبارت دیگر ، ما می توانیم در یک مسیر طولانی ، به صورت نامحدود ، سرگردان باشیم بدون این که به یک راه حل برسیم .

زمان ؟؟ $O(b^m)$ در صورتی که m به مراتب بزرگ تر از d باشد ، بسیار بد است . ولی اگر راه حل ها پیچیده باشند ، این روش به مراتب سریع تر از جستجوی اوّل سطح می باشد. توجّه کنید که m ، بیشینه ی عمق درخت می باشد . همچنین در برخی از موارد ، m ممکن است نامحدود باشد .

فضا ؟؟ O(bm) ، که فضایی خطی می باشد . ما فقط نیاز به نگهداشتن شاخه ای که به تازگی جستجو شده است داریم و این حسن بزرگ جستجوی اوّل عمق می باشد .

بهینگی ؟؟ نه . ما ممکن است یک راه حل را در عمق n ، تحت یک فرزند بدون دیدن یک راه حل کوتاه تر ، تحت فرزند دیگر پیدا نماییم . (در مثال قبل که با استفاده از پشته آن را حل کردیم ، اگر اوّل ریمنیکو ویلسیا را از پشته برمی داشتیم چه اتّفاقی می افتاد ؟)

اوّل ، یک سؤال

چرا ما به الگوریتم هایی که یک جستجوی جامع را انجام می دهند توجّه می کنیم ؟ آیا چیزی سریع تر وجود ندارد ؟ بسیاری از مسایلی که ما به آن ها توجّه می کنیم ، NP – کامل می باشند و در مورد آن ها هیچ الگوریتم با زمان چندجمله ای شناخته شده ای وجود ندارد و بدتر از آن ، تعدادی هم غیرقابل تخمین می باشند .

جستجوی اوّل سطح

اگر جستجوی اوّل سطح را با استفاده از یک صف پیاده سازی کنیم ، تمام گره های در سطح n ، قبل از هر گره در سطح n+1 ارزیابی می شوند و این دارای این حسن است که ، کامل می باشد و در زمانی که هزینه ی مرحله به صورت یکسان می باشد ، راه حل بهینه را پیدا خواهد نمود و دارای این ضعف است که به فضایی به صورت نمایی نیازمندیم .

جستجوی اوّل عمق

اگر جستجوی اوّل سطح را با استفاده از پشته پیاده سازی نماییم ؛ ابتدا یک گره توسعه داده می شود ، سپس فرزندانش توسعه داده می شوند و ... و در هر لحظه ، فقط یک شاخه از درخت جستجو در حافظه نگهداری می شود . حسن این روش این است که به فضایی به صورت خطی نیازمندیم و عیب آن این است که نا کامل و غیربهینه می باشد .

جلوگیری از جستجوی نامحدود

چند روش برای جلوگیری از جستجوی اوّل عمق نامحدود وجود دارد ، یکی استفاده از closed-list است که ممکن است همیشه به ما کمک نکند . در این صورت ما باید تعداد زیادی گره را به صورت نمایی در حافظه نگهداری نماییم . دو روش دیگر جستجوی با عمق محدود شده و جستجوی عمیق کننده ی تکراری جستجوی اوّل عمق می باشند .

جستجوی با عمق محدود شده

با محدود نکردن عمق l با جستجوی اوّل عمق برابر است و گره های در عمق l هیچ جانشینی ندارند. **در واقع ،** جستجوی با عمق محدود شده ، همان جستجوی اوّل عمق است که در آن عمق جستجو محدود شده است (تا یک عمقی در درخت پایین می رویم و بیش تر از آن پایین نمی رویم) و عملیّات جستجو در این عمق ، متوقف می شود . در این جا مشکل دیگری به وجود می آید و آن این است که اگر راه حل در عمقی بیش تر از l باشد ، چطور ؟ چگونه ما یک l خوب را به دست آوریم ؟ . در مسأله ی رومانی ، ما می دانیم که بیست (۲۰) شهر وجود دارد ، بنابراین l=19 ، یک انتخاب خوب می باشد . در مورد پازل ۸ – تایی چه طور ؟

شبه کد پیاده سازی روش جستجوی با عمق محدود شده با استفاده از پشته :

```
push(initialState)
do
node = pop( )
if goalTest(node)
        return node
else
        if depth(node) < limit
```

children = successor-fn(node)

for child in children

push(child)

پیاده سازی بازگشتی :

تابع function Depth-Limited-Search(problem,limit) مقادیر soln ، fail و یا cutoff را برمی گرداند

Recursive-DLS(Make-Node(Initial-State[problem]),problem,limit)

تابع function Recursive-DLS(node,problem,limit) مقادیر soln ، fail و یا cutoff را برمی گرداند

cutoff-occurred? ← false

در صورتی که Goal-Test(problem,State[node]) صحیح بود گره (node) را برگردان . (در صورتی که مفدار تابع درست بود node را برگردان ;)

در غیر این صورت ، اگر Depth[node]=limit بود ، مقدار cutoff را برگردان .

در غیر این صورت ، برای هر successor (جانشین) در Expand(node,problem) کارهای زیر را انجام بده :

result ← Recursive-DLS(successor,problem,limit)

اگر result=cutoff بود سپس مقدار true را در cutoff-occurred بریز .

در غیر این صورت ، اگر result < > failure بود result را برگردان (< > علامت نامساوی می باشد) .

در صورتی که cutoff-occurred صحیح بود cutoff را برگردان و در غیر این صورت failure را برگردان.

پایان الگوریتم

جستجوی عمیق شونده ی تکراری

بر مبنای ایده ی جستجوی با عمق محدود شده ، توسعه می یابد . جستجوی با عمق محدود شده را ابتدا با عمق l=1 ، سپس با عمق l=2 و ... انجام دهید . سرانجام ، l=d می شود ، این به این معنی می باشد که جستجوی عمیق کننده ی تکراری ، کامل می باشد . اشکال این روش این است که برخی از گره ها تولید شده اند و چند بار ، توسعه داده شده اند . در سطح یک ؛ تعداد b گره ، تعداد d مرتبه تولید می شوند . در سطح دو ؛ تعداد b^2 گره ، d-1 بار تولید می شوند و ... در نتیجه در سطح d ، تعداد b^d بار تولید می شوند . در این روش ، مجموع زمان اجرا برابر $O(b^d)$ می باشد . که اندکی کم تر از تعداد گره های تولید شده در جستجوی اوّل سطح می باشد و هنوز به حافظه ای به صورت خطی نیازداریم . **پس** ، عمق محدودشده را با افزایش محدودیّت ها به کار می گیرد . الگوریتم این روش به صورت زیر است :

تابع function Iterative-Deepening-Search(problem) یک راه حل را برمی گرداند .

ورودی ها problem که یک مسأله است ، می باشد

برای depth مساوی با صفر تا بی نهایت (∞) کارهای زیر را انجام بده

result ← Depth-Limited-Search(problem,depth)

در صورتی که result< >cutoff بود result را برگردان ;

پایان حلقه

پایان الگوریتم

شبه کدی به صورت دیگر برای جستجوی عمیق شونده ی تکراری :

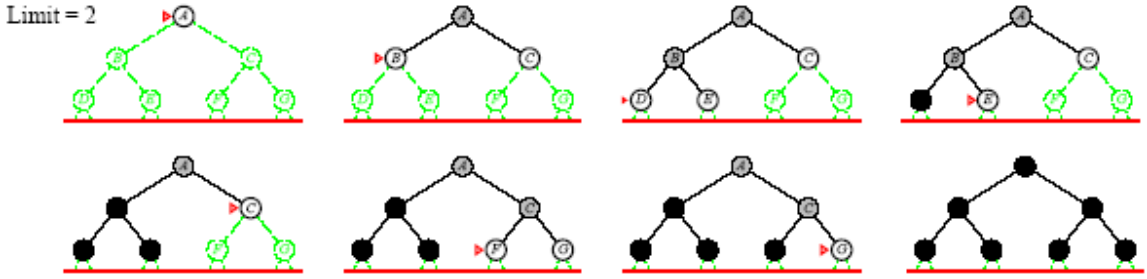
```
d = 0
while (1)
result = depth-limited-search(d)
if result == goal
return result
else
d = d + 1
```

مثال :

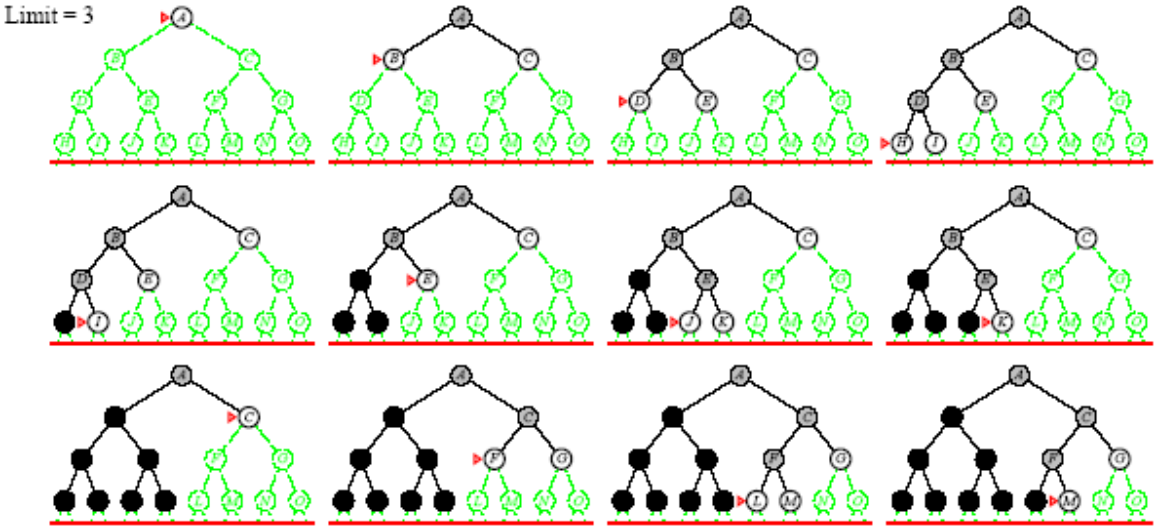
جستجوی عمیق شونده ی تکراری با l=0

جستجوی عمیق شونده ی تکراری با l=1

جستجوی عمیق شونده ی تکراری با l=2



جستجوی عمیق شونده ی تکراری با l=3



جستجوی عمیق شونده ی تکراری ، جستجوی اوّل سطح و اوّل عمق را با هم ترکیب می کند و لایه ها را شبیه جستجوی اوّل سطح به وجود می آورد ، امّا در هر اجرا یک اوّل عمق را انجام می دهد که باعث صرفه جویی در فضا می شود . در این روش ، وضعیّت ها چند بار توسعه داده می شوند .

در واقع ، جستجوی عمیق شونده ی تکراری ، شبیه جستجوی اوّل سطح می باشد که در آن همه ی گره های در عمق n قبل از هر گره در عمق n+1 برّرسی می شوند . نظیر جستجوی اوّل سطح ، ما می توانیم بهینگی را در جهان های با هزینه ی غیریکسان با توسعه دادن با توجّه به هزینه ی مسیر و ترجیحا با استفاده از

مترجم: سهراب جلوه گر
ویرایش دوّم ، بهار ۱۳۸۸

عمق ، به دست آوریم . این روش ، *جستجوی دراز کننده ی طول* [۱] نـام دارد . همـه ی مسـیرها را در هزینـه ی کم تر از p جستجو می کند . p را با δ افزایش می دهد . در جهان های پیوسته ، δ چه باید باشد ؟

جریان معکوس [۲]

در زمانی که جستجوی اوّل عمق و همتایانش به وضعیّت عدم موفّقیّت می رسند ، چه اتّفاقی می افتـد ؟ به بالای گره ی پدر (والد) می روند و برادر بعدی را امتحان می کنند . و فرض را بر این قرار می دهند کـه جدیدترین عملکرد انتخاب شده ، عملکردی است که باعـث عـدم موفّقیّـت شـده اسـت . ایـن روش ، *جریـان معکوس به ترتیب وقوع* [۳] نام دارد و جدیدترین کاری را کـه انجـام داده ایـد ، بـی اثـر نماییـد . مسـأله ای کـه وجود دارد این است که عدم موفّقیّت ، شاید یک نتیجه از یک تصمیم گیری قبلی باشـد ؛ مثـال ، ۴ – وزیـر . محدودیّت ها می توانند به شما برای محدود کردن اندازه ی فضای جستجو کمـک نماینـد . *جریـان معکـوس هوشمند* [۴] برای تحلیل دلیلی برای عدم موفّقیّت و غیرفعّال کردن جستجو برای آن نقطه تلاش می نماید و مـی تواند جدیدترین متغیّر ناسازگار را غیرفعال نماید (جریان معکوس) ، همچنـین مـی توانـد برّرسـی مسـتقیم [۵] را انجام دهد ؛ آیا یک انتساب ممکن مقدارها برای متغیّرهایی در این نقطه وجود دارد ؟

خصوصیات جستجوی عمیق شونده ی تکراری

کامل بودن ؟؟ بله

زمان ؟؟ $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

---

[۱] iterative lengthening search
[۲] backtracking
[۳] chronological backtracking
[۴] intelligent backtracking
[۵] forward checking

فضا ؟؟ O(bd)

بهینگی؟ بله اگر گام cost برابر یک باشد . می تواند برای به وجود آوردن درخت uniform-cost بازنگری شود . به عنوان مثال ، مقایسه ی عددی برای b=10 و d=5 ، برای دورترین برگ سمت راست :

$$N(IDS) = 50 + 400 + 3000 + 20000 + 100000 = 123450$$

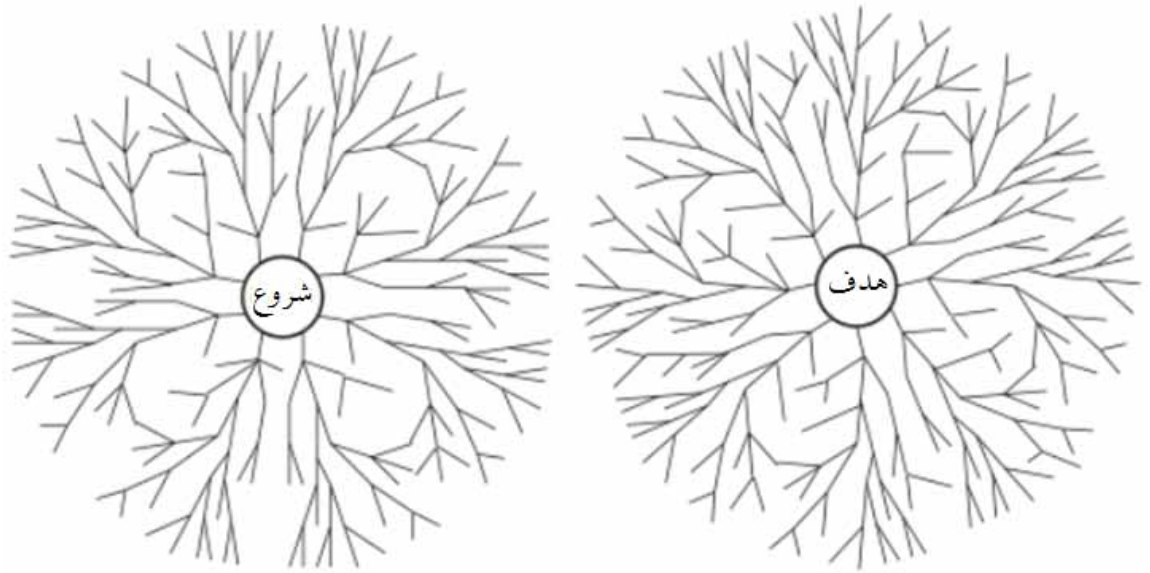
$$N(BFS) = 10 + 100 + 1000 + 10000 + 100000 + 999990 = 1111100$$

IDS (جستجوی عمیق شونده ی تکراری) بهتر عمل می کند ، چون که گره های دیگر درون عمق d توسعه داده نمی شوند . BFS (جستجوی اوّل سطح) می تواند برای مدت زمانی که یک گره تولید می شود بازنگری شود.

جستجوی دوطرفه [1]

دو جستجوی شبیه را اجرا می نماید ، یکی در جهت وضعیّت اوّلیه و دیگری در خلاف جهت و از طرف وضعیّت هدف و در زمانی که دو جستجو به هم می رسند جستجو خاتمه می یابد .

[1] bidirectional search



خصوصیّات جستجوی دو طرفه

- در صورتی که هر دو جستجو اوّل سطح باشند کامل است .
- زمان : $O(b^{d/2})$
- فضا : $O(b^{d/2})$
- در صورتی که هر دو روش اوّل سطح باشند بهینه می باشد

اشکالات [۱] این روش : تولید قبلی ها (معکوس کردن عملگرها) و هماهنگی میان دو جستجو و این که جستجو باید گره ها را در حافظه نگهداری نماید (برای برّرسی این که آیا گره ها قبلا ملاقات شده اند یا نه)

[۱] drawbacks

مترجم: سهراب جلوه گر
ویرایش دوّم ، بهار ۱۳۸۸

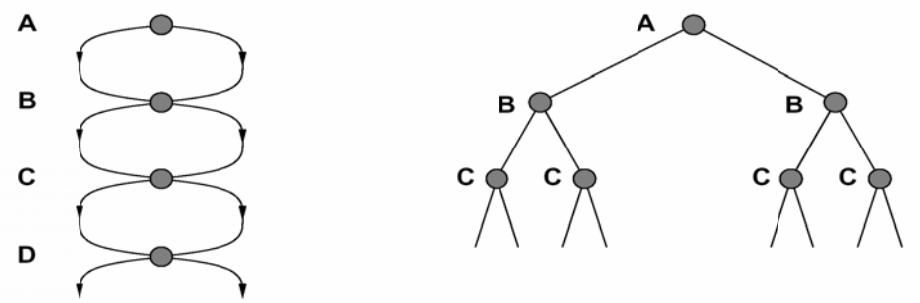
خلاصه ی الگوریتم ها

مقیاس [۱]	اوّل سطح	با هزینه ی یکسان	اوّل عمق	عمق محدود شده	عمیق شونده ی تکراری
کامل بودن ؟؟	بله *	بله *	نه	بله ، در صورتی که $l>=d$ باشد	بله
زمان ؟؟	$b^{d}+1$	$b^{\lceil C^{*}/\epsilon \rceil}$	b^{m}	b^{l}	b^{d}
فضا ؟؟	$b^{d}+1$	$b^{\lceil C^{*}/\epsilon \rceil}$	bm	bl	bd
بهینگی ؟؟	بله *	بله	نه	نه	بله *

در جدول بالا ، b : فاکتور انشعاب ، d : عمق کم عمق ترین راه حل ، m : بیشینه ی عمق درخت جستجو و l : محدودیّت عمق می باشند .

حالت (وضعیّت) های تکرار شده – *عدم موفّقیّت در پیداکردن حالت های تکرار شده می تواند یک مسأله ی خطی را تبدیل به یک مسأله ی نمایی کند .*

[۱] Criterion



راه حل ایـن مشـکل ، نگهـداری تاریخچـه اسـت ؛ گـره هـای ملاقـات شـده (توسـعه داده شـده) در closed list نگهداری می شوند و گره جـاری اگـر بـه تـازگی در closed list بـوده ، حـذف مـی شـود . مشکل این روش این است که بهینگی ممکن است از بین برود چون همه ی گره هـا در حافظـه نگهـداری مـی شوند .

الگوریتم جستجوی گراف

تابع function Graoh-Search(problem,fringe) یک راه حل یا عدم موفّقیّت را برمی گرداند .

یک مجموعه ی خالی را به closed تخصیص بده .

fringe ← Insert(Make-Node(Initial-State[problem]),fringe)

در حلقه کارهای زیر را انجام بده

در صورتی که fringe خالی باشد عدم موفّقیّت را بر گردان

node ← Remove-Front(fribge)

در صورتی که Goal-Test(problem,State[node]) موفّقیّت آمیز بود ، گره (node) را برگردان

در صورتی که State[node] در closed نبود

State[node] را به closed اضافه نما

InsertAll(Expand(node,problem),fringe) را در fringe بریز

پایان الگوریتم

خلاصه – فرمول بندی مسأله ، معمولاً به در کنار هم قرار دادن جزییات دنیای واقعی برای تعریف یک فضای حالت که بتواند به طور عملی به وجود آید نیاز دارد . تنوع روش ها یا استراتژی های جستجوی ناآگاهانه : جستجوهای عمیق شونده ی تکراری [۱] فقط به صورت فضای خطی استفاده می شوند و زمان به مراتب بیش تری نسبت به سایر الگوریتم های ناآگاهانه ندارند . جستجوی گرافی می تواند به مراتب موثرتر از جستجوی درختی باشد .فرمول بندی یک مسأله ی جستجو با استفاده از وضعیّت اوّلیه ، آزمون هدف ، عملیّاتی که باید انجام شوند ، تابع جانشین (مولد) و هزینه ی مسیر ، منجر به جستجو در یک فضای حالت با استفاده از یک درخت جستجو می شود . الگوریتم ها عبارتند از : جستجوی اوّل سطح ، جستجوی اوّل عمق ، جستجوی با هزینه ی یکسان ، جستجوی با عمق محدود شده و جستجوی عمیق کننده ی تکراری

[۱] iterative deeping search

مترجم: سهراب جلوه گر

ویرایش دوّم ، بهار ۱۳۸۸

فصل چهارم

جستجوی آگاهانه

(مکاشفه ای)