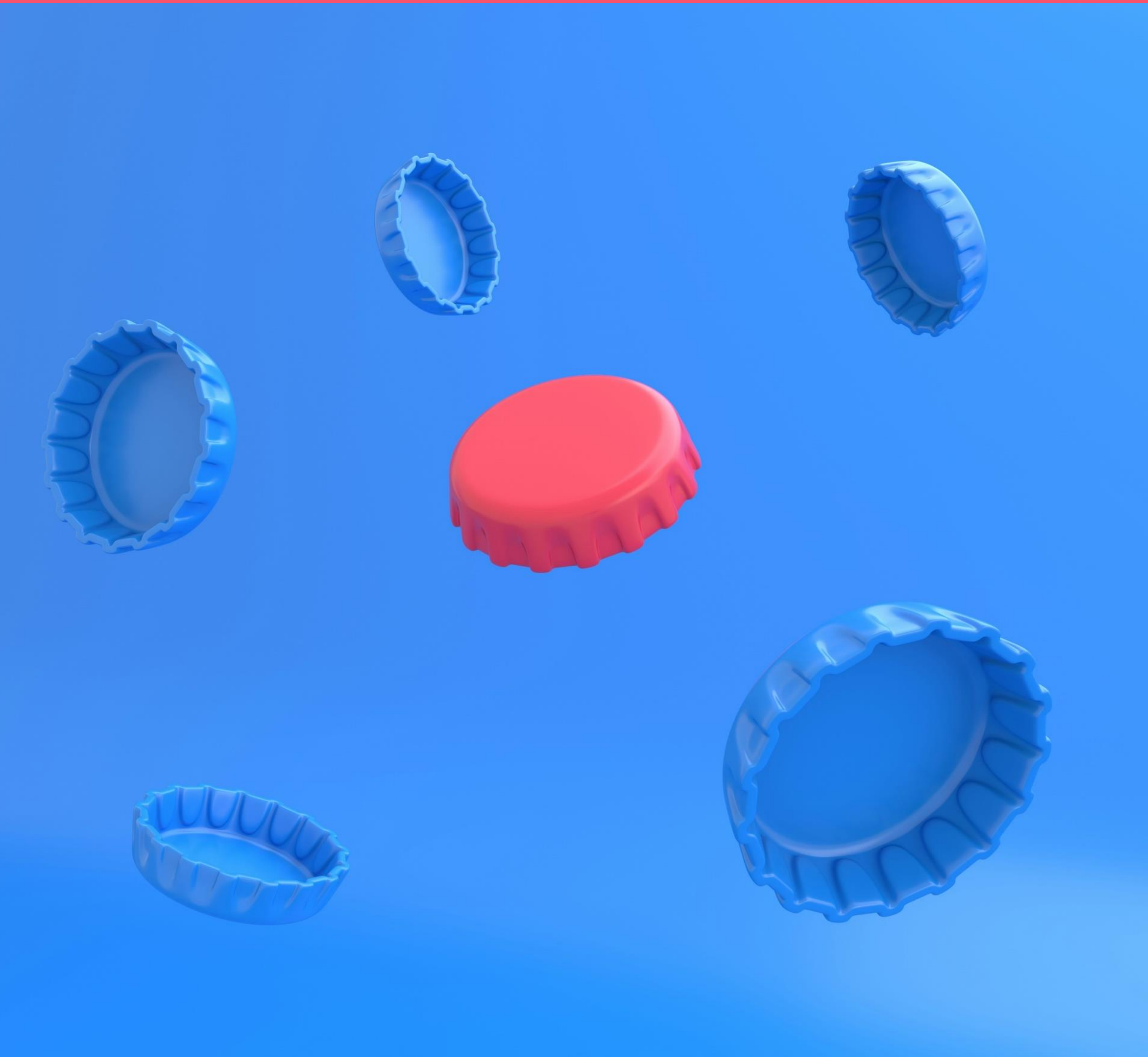
فصل سوم، حل مسئله با جستجو



- ## فصل سوم، حل مسئله با کمک جستجو

وقتی که دقیقاً معلوم نیست باید چه کاری انجام بدی، ممکنه لازم باشه از قبل برای کارات برنامه‌ریزی کنی؛ یعنی یه سری اقدام رو پشت سر هم بچینی تا به حالت یا هدفی که می‌خوای برسی. به این نوع عامل می‌گن «عامل حل مسئله» و فرایندی هم که طی می‌کنه «جست‌وجو» نام داره.

عامل‌های حل مسئله معمولاً وضعیت‌ها رو به صورت کامل و غیرقابل‌شکستن (اتمی) در نظر می‌گیرن، یعنی هیچ ساختار داخلی‌ای توی الگوریتم‌های حل مسئله براشون معلوم نیست. اما عامل‌هایی که وضعیت‌ها رو با جزئیات و ساختار (فکتورها) مدل می‌کنن، «عامل‌های برنامه‌ریز» نام دارن که در فصل‌های ۷ و ۱۱ مفصّل توضیح داده شده.

تو این فصل قراره چند تا الگوریتم جست‌وجو رو یاد بگیریم، اما فقط برای ساده‌ترین نوع محیط‌ها: محیط‌هایی که هر مرحله به صورت جدا (Episodic) انجام می‌شن، فقط یه عامل دارن، همه‌چیز جلوی چشم عامل هست (Fully Observable)، هیچ چیز شانسی یا تصادفی نیست (Deterministic)، محیط ثابت می‌مونه (Static)، وضعیت‌ها گسسته و مشخص هستن (Discrete)، و عامل از قبل با محیط آشناست (Known).

همچنین بین دو دسته الگوریتم فرق می‌ذاریم:
الگوریتم‌های آگاه (Informed) که عامل می‌تونه حدس بزنه چقدر هنوز تا هدف فاصله داره.
الگوریتم‌های ناآگاه (Uninformed) که چنین حدسی در کار نیست.

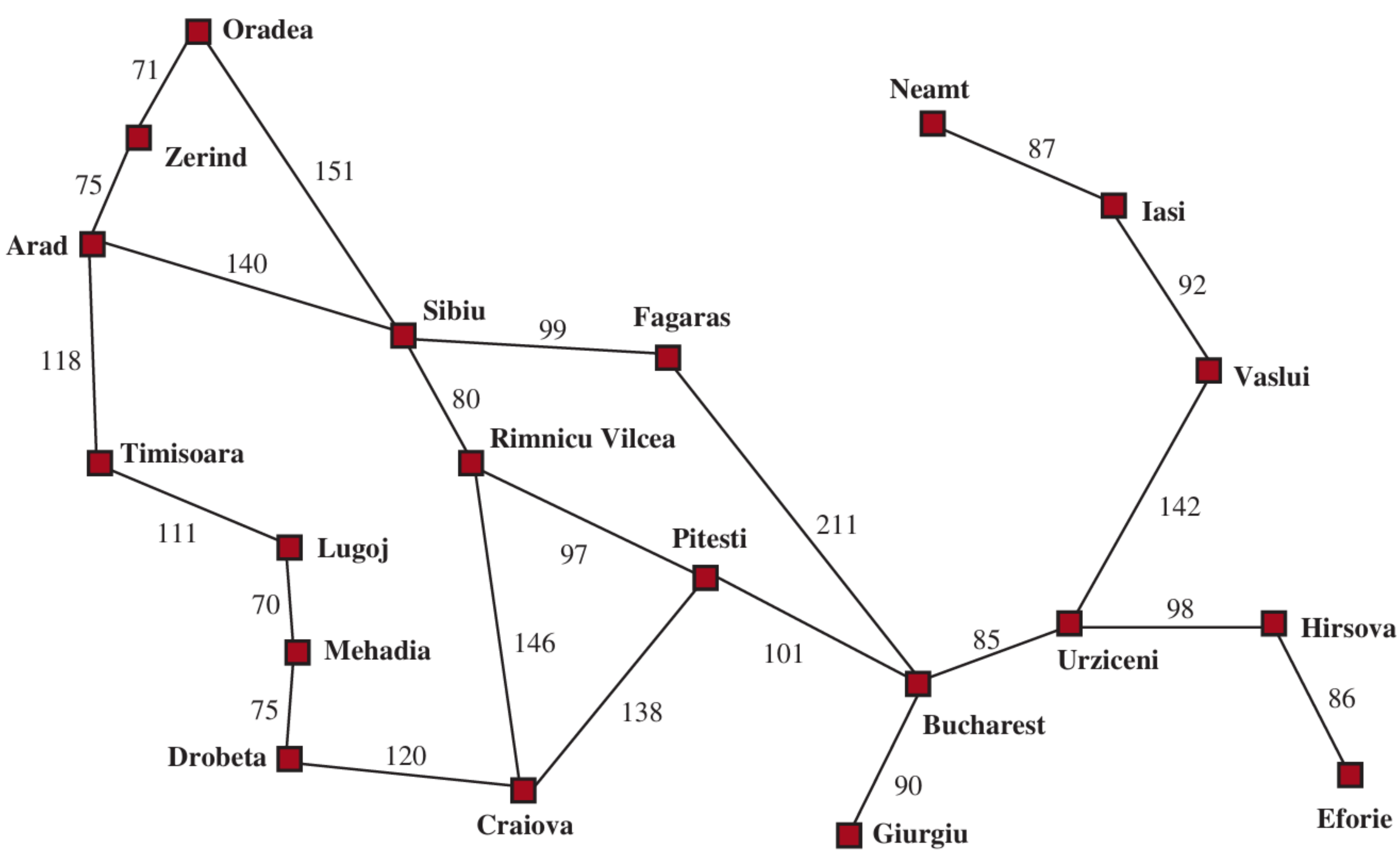
فصل ۴ میاد این محدودیت‌ها رو کمتر می‌کنه و می‌ره سراغ محیط‌های پیچیده‌تر، و فصل ۶ درباره محیط‌هایی بحث می‌کنه که چند عامل با هم دارن کار می‌کنن.

عامل حل مسئله

فرض کن یه عامل تو یه سفر تفریحی داریم که داره تو رومانی می‌چرخه. این عامل می‌خواد جاهای دیدنی رو ببینه، زبان رومانی‌ش رو بهتر کنه، از شب‌گردی لذت ببره، حالش خوب بمونه و این حرفا. انتخاب اینکه دقیقاً چه کار کنه یه مسألهٔ پیچیده‌ست.

حالا فرض کن عامل ما الان تو شهر آراد (Arad) هست و یه بلیت غیرقابل پس‌دادن (nonrefundable) برای پرواز فردا از بخارست (Bucharest) داره. طرف به تابلوهای خیابون نگاه می‌کنه و می‌بینه سه تا جاده از آراد خارج میشن: یکی می‌ره سیبیو (Sibiu)، یکی تیمیشوارا (Timișoara) و یکی زریند (Zerind). هیچ‌کدوم از این سه تا مقصد نهایی نیستن، پس تا وقتی نقشه یا اطلاعات دیگه‌ای نداشته باشه، نمی‌دونه کدوم جاده رو بره. اگه هیچی اضافه‌تر ندونه —یعنی محیط براش ناشناخته باشه— اون‌وقتا کاری بهتر از این نیست که یه جاده رو شانس و تصادف انتخاب کنه!

این وضعیت غم‌انگیز رو تو فصل ۴ بیشتر بررسی کردیم.



اما تو این فصل فرض می‌کنیم عامل ما همیشه یه سری اطلاعات درباره دنیا داره، مثل نقشه‌ای که تو صفحه قبل هست. با این اطلاعات، عامل می‌تونه این روند چهارمرحله‌ای حل مسئله رو دنبال کنه:

فرموله کردن هدف (Goal formulation):
اول هدفش رو مشخص می‌کنه: رسیدن به بخارست. داشتن هدف باعث میشه کار و رفتار عامل جهت‌دار بشه و فقط چیزهایی رو در نظر بگیره که به اون هدف می‌رسن.

فرموله کردن مسأله (Problem formulation):
بعد توضیح میده که چه وضعیت‌ها و چه کارهایی لازمه تا به هدف برسه—یه مدل انتزاعی از قسمت مربوط دنیا. برای این مثال، می‌تونیم وضعیتمون رو صرفاً با همین بررسی کنیم که «من الآن تو کدوم شهرم» و کارها هم «سفر از یک شهر به شهر مجاور». پس تنها دلیلی که وضعیت تغییر می‌کنه، عوض شدن شهر فعلیه.

جست‌وجو (Search):
قبل از اینکه تو دنیای واقعی قدم از قدم برداره، تو مدل ذهنی‌ش شبیه‌سازی می‌کنه: دنباله‌ای از حرکت‌ها رو بررسی می‌کنه تا ببینه کدوم مسیر واقعاً به بخارست می‌رسه. هر مسیری که به هدف برسونه «راه‌حل» نام داره. ممکنه اول چند تا مسیر رو امتحان کنه که به جایی نرسن، ولی بالاخره یا یه راه‌حل پیدا می‌کنه یا می‌فهمه هیچ راهی وجود نداره.

اجرا (Execution): حالا راه‌حلی که پیدا کرده رو مرحله به مرحله تو دنیای واقعی انجام میده.

نکته‌ی مهم اینه که تو یه محیط «کاملاً قابل‌مشاهده»، «قطعی» و «شناخته‌شده»، راه‌حل هر مسأله یک دنباله‌ی ثابت از اقداماته: اول برو سیبیو، بعد فاگارا، بعد بخارست. اگه مدل دقیقه، وقتی راه‌حل رو پیدا کردی، دیگه لازم نیست وسط اجرا به ورودی‌های جدید (percepts) توجه کنی—مثل این که چشمات رو ببندی—چون مطمئنی همین دنباله به هدفت می‌رسونه. کنترل‌کننده‌ها به این «سیستم حلقه‌باز» (open-loop) می‌گن؛ چون حلقه‌ی بازخورد بین عامل و محیط رو قطع می‌کنه.

ولی اگه احتمال بد باشه که مدل اشتباهه یا محیط غیرقطعی (nondeterministic) باشه، عامل ایمن‌تره اگه «سیستم حلقه‌بسته» (closed-loop) داشته باشه و حین مسیر هم ورودی‌ها رو چک کنه. تو محیط‌های نیمه‌قابل مشاهده یا غیرقطعی، راه‌حل دیگه یه دنباله‌ی قطعی نیست؛ بلکه یه «استراتژی شاخه‌ای»ئه که بر اساس هر پرسپت جدید، مسیر بعدی رو مشخص می‌کنه. مثلاً فرض کن برنامه‌ات اینه که از آراد بری سیبیو، ولی اگه به اشتباه رسیدی زریند یا تابلو «جاده بسته» دیدی، باید نقشه‌ی جایگزین داشته باشی.

یه مسئله‌ی جست‌وجو رو می‌شه این‌طوری تعریف کرد:

مجموعه‌ی حالت‌ها (State Space): همه‌ی حالت‌هایی که محیط ممکنه توش قرار داشته باشه.

حالت اولیه: همون جایی که عامل کارش رو ازش شروع می‌کنه. مثلاً «آراد».

حالت(های) هدف: جایی که می‌خوایم بهش برسیم. گاهی فقط یه حالته (مثل «بخارست»)، گاهی چندتا حالت آلترناتیو، یا حتی گاهی مشخص می‌کنیم هر حالتی که یه ویژگی خاص داره، هدفه (مثل تو دنیای جاروبرقی، «هیچ کدوم از خونه‌ها خاک نداشته باشن»). برای پشتیبانی از همه‌ی این حالت‌ها، یه تابع IS-GOAL داریم که بررسی می‌کنه آیا یک حالت هدفه یا نه.

اعمال ممکن (Actions): تو هر حالت s، تابع ACTIONS(s) یه لیست محدود از کارهایی رو که می‌شه انجام داد برمی‌گردونه. مثلاً: ACTIONS(Arad) = { ToSibiu, ToTimisoara, ToZerind }

مدل انتقال (Transition Model): تابع RESULT(s,a) نشون می‌ده که اگر تو حالت s عمل a رو انجام بدیم، به چه حالتی می‌رسیم. مثلاً: RESULT(Arad, ToZerind) = Zerind

هزینه‌ی هر عمل (Action Cost): تابع ACTION-COST(s,a,s') یا به‌اختصار c(s,a,s') نشون می‌ده انجام عمل a در حالت s تا رسیدن به s' چه هزینه‌ای داره. این هزینه باید بازتاب معیار عملکرد خودِ عامل باشه؛ مثلاً تو مسئله‌ی مسیریابی ممکنه مسافت جاده یا زمان طی‌شدن مسیر باشه.

یه مسیر (path) یعنی دنباله‌ای از عمل‌ها، و راه‌حل (solution) یعنی یه مسیر از حالت اولیه تا حالت هدف. چون هزینه‌ها رو جمع می‌کنیم (جمع‌شدن هزینه‌ی تک‌تک عمل‌ها)، یه راه‌حل بهینه اونیه که کمترین هزینه رو داشته باشه. برای ساده‌تر بودن، فرض می‌کنیم همه‌ی هزینه‌ها مثبت باشن.

این حالت‌ها و انتقال‌ها رو می‌شه مثل یه نمودار (graph) نشون داد که رئوسش حالات و یال‌هاش عمل‌ها هستن. مثلاً نقشه‌ی رومانی تو شکلی که دیدیم، همین‌طوره: هر جاده دو تا عمل داره، یکی برای هر جهت.

فرموله‌سازی مسئله

مدل ما برای مسئله‌ی «رسیدن به بخارست» یه توصیف انتزاعیه—یه نمایش ریاضیاتی که واقعیت رو کامل‌ش نمی‌کنه. اگه وضعیت ساده‌ی «آراد» رو با یه سفر واقعی از این سر کشور تا اون سرش مقایسه کنی، می‌بینی تو سفر واقعی کلی چیز باید در نظر گرفته بشه: همراها و دوستا، برنامه‌ی رادیو، مناظر کنار جاده، احتمال کنترل پلیس، فاصله تا ایستگاه استراحت بعدی، وضعیت جاده، آب‌وهوا، ترافیک و... ولی همه‌ی اینا رو از مدل حذف کردیم چون ربطی به پیدا کردن مسیر به بخارست نداره. به این کار حذف جزئیات یا «انتزاع» می‌گن. یه فرمول‌بندی خوب باید همون حد جزئیاتی رو داشته باشه که برای حل مسئله لازمه، نه بیشتر. اگه عمل‌ها روی سطح «قدم راست رو یه سانت جلو ببر» یا «فرمان رو یه درجه بچرخون» تعریف می‌شدن، عامل احتمالاً هیچ‌وقت از پارکینگ در نمی‌اومد چه برسه به بخارست!

حالا بیایم دقیق‌تر ببینیم سطح مناسب انتزاع چیه. فرض کن وضعیت‌ها و عمل‌های انتزاعی ما معادل دسته‌های بزرگی از وضعیت‌ها و توالی‌های عملیاتی دقیق‌تر باشن. مثلاً راه‌حل انتزاعی «آراد → سیبیو → رمینکو ویلچا → پیتشت → بخارست» در واقع معادل کلی مسیر دقیق‌تره؛ مثلاً ممکنه تو بخش سیبیو تا رمینکو ویلچا رادیو روشن باشه و بعدش خاموشش کنیم. انتزاع زمانی معتبره که هر راه‌حل انتزاعی بشه به شکلی تفصیلی اجراش کرد؛ به‌عبارت دیگه، برای هر حالتی که «آراد» باشه، باید یه مسیر دقیق باشه که به حالت «سیبیو» برسه، و همین‌طور ادامه پیدا کنه.

این انتزاع وقتی کاربردیه که انجام هر عمل توی راه‌حل، از حل کل مسئله ساده‌تر باشه؛ مثلاً رانندگی از آراد تا سیبیو یه راننده‌ی معمولی می‌تونه بدون جست‌وجوی اضافه یا برنامه‌ریزی جدید انجامش بده. پس انتخاب انتزاع خوب یعنی حذف حداکثر جزئیات تا وقتی که همچنان قابلیت تبدیل به راه‌حل دقیق واقعی حفظ بشه و عمل‌ها قابل اجرا باشن. اگه نمی‌تونستیم انتزاع‌های مفیدی بسازیم، عامل‌های هوشمند زیر حجم بی‌پایان جزئیات دنیای واقعی له می‌شدن.

مثال‌هایی از مسائل

رویکرد «حل مسئله» رو توی کلی محیط مختلف به کار بردن. اینجا چندتا از معروف‌ترینشون رو میاریم و بین «مسأله‌های استاندارد» و «مسأله‌های واقعی» فرق می‌ذاریم.

مسأله‌ی استاندارد: برای نشون دادن یا تمرین روش‌های حل مسئله‌ست. یه تعریف دقیق و کوتاه داره و برای محقق‌ها مثل یه معیار مقایسه‌است.

مسأله‌ی واقعی: مثل ناوبری ربات که مردم واقعاً استفاده می‌کنن. چون هر ربات حسگرهای خاص خودش رو داره، تعریفش استاندارد نیست و مختص خودش میشه.

مسأله‌های استاندارد

مسأله‌ی «جهان شبکه‌ای» (Grid World) یه صفحه‌ی شطرنجیه که هر خونه‌ش مربعئه و عامل می‌تونه بین خونه‌ها جابه‌جا بشه. معمولاً حرکت‌ها به خونه‌های مجاور بالا، پایین، چپ و راست محدودن (گاهی هم حرکت مورب مجازه). بعضی خونه‌ها می‌تونن اشیایی داشته باشن که عامل برداره، هل بده یا تعامل دیگه‌ای باهاشون داشته باشه؛ دیوار یا مانع هم ممکنه جلوی مسیر باشه و نتونی از خونه‌ی کنارش رد شی.

دنیای جاروبرقی (که اول فصل ۲ دیدیم) رو می‌شه این‌جوری مدل کرد:

حالت‌ها (States): تو هر حالت می‌گیم چه چیزی کجاست. اینجا «جاروبرقی» و «خاک‌ها» می‌تونن تو خونه‌ها باشن. تو نسخه‌ی ساده با دو خونه: جاروبرقی یا تو خونه‌ی A هست یا B، و هر خونه یا خاک داره یا نداره. پس می‌شه $2×2×2 = 8$ حالت. بصورت کلی با n خونه، می‌شه $n×2^n$ حالت.

حالت شروع (Initial State): هر کدوم از اون حالت‌ها می‌تونه نقطه‌ی شروع باشه.

عمل‌ها (Actions): تو نسخه‌ی دو خونه، سه تا عمل داشتیم: مکش (Suck)، برو چپ (Left)، برو راست (Right). تو شبکه‌ی چندخونه‌ای دوبعدی، معمولاً چهار جهت اصلی (بالا/پایین/چپ/راست) رو داریم، یا بعضی وقت‌ها از عمل‌های «جلوبرو»، «عقب‌رو»، «بپیچ راست» و «بپیچ چپ» استفاده می‌کنن که نسبت به جهت دید جاروبرقی تعریف می‌شن.

مدل انتقال (Transition Model):

مکش: هر خاکی رو از خونه‌ی فعلی برمی‌داره.

جلو/عقب: جاروبرقی رو یک خونه جلو یا عقب می‌بره، مگر اینکه دیوار باشه.

بپیچ راست/چپ: جهت نگاهش رو ۹۰ درجه عوض می‌کنه.

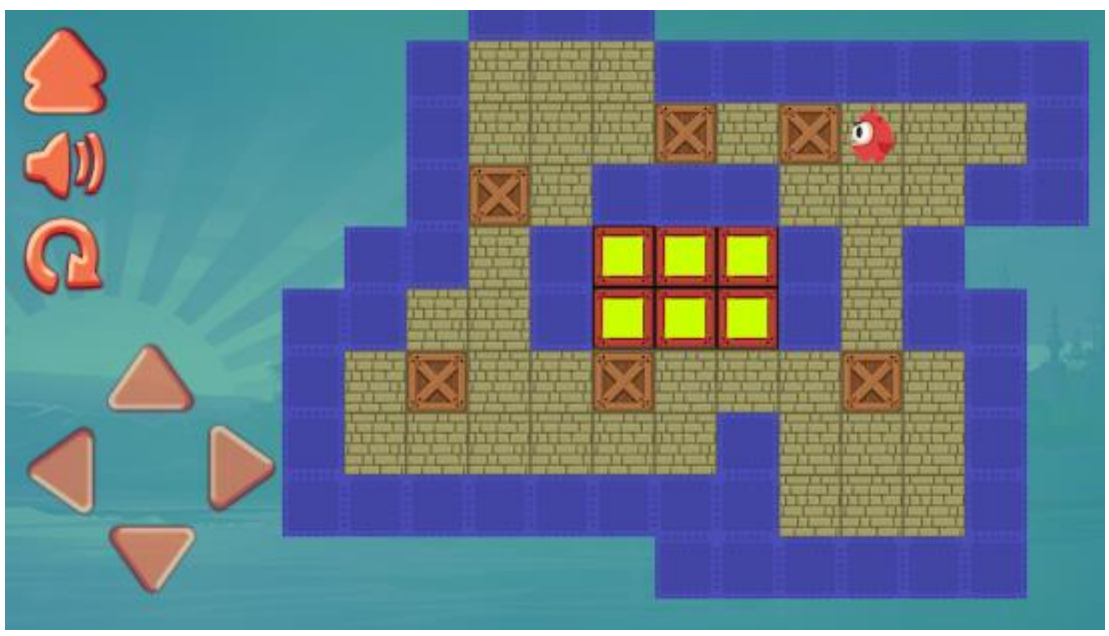
حالت‌های هدف (Goal States): حالتی که همه‌ی خونه‌ها تمیز باشن.

هزینه‌ی عمل (Action Cost): هر عمل هزینه‌ی ۱ داره.

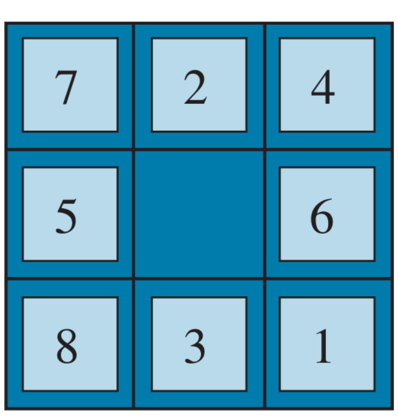
یه مسیر، یعنی دنباله‌ی چندتا عمل. راه‌حل هم همون مسیریه که از حالت شروع برسونه به حالت هدف. چون هزینه‌ها رو جمع می‌کنیم، راه‌حل بهینه اونیه که کمترین مجموع هزینه رو داشته باشه.

یه نوع دیگه از جهان شبکه‌ای، «ساکوبان» (Sokoban) هست که هدف عامل اینه چندتا جعبه‌ای رو که دور و بر شبکه پخش شدن، ببره و به جای مشخصی بذاره. تو هر خونه حداکثر یه جعبه می‌تونه باشه. وقتی عامل یه خونه جلو میره و اونجا جعبه باشه و پشتِ جعبه یه خونه خالی موجود باشه، هم جعبه و هم عامل یه قدم به جلو میرن.

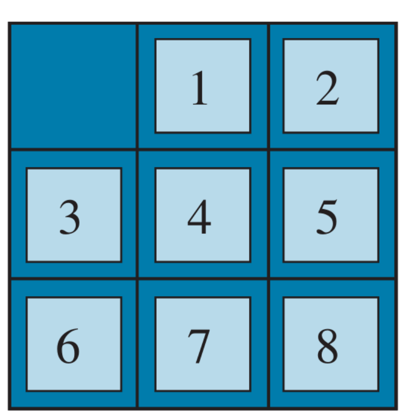
عامل نمی‌تونه جعبه رو تو خونه‌ی دیگه‌ای که پر از جعبه‌ست یا تو دیوار هل بده. تو دنیایی با n خونه‌ی بدون مانع و b تا جعبه، تعداد حالت‌ها میشه $n \times n!/(b!(n-b)!)$؛ مثلا تو یه صفحه‌ی ۸×۸ با دوازده جعبه، بیش از دویست تریلیون حالت داریم!



تو «پازل کاشی‌لغز» (Sliding-Tile Puzzle) هم تعدادی کاشی (یا بلوک) تو شبکه چیده شده و یه یا چند خونه خالی وجود داره تا بشه کاشی‌ها رو به سمت اون خونه خالی کشید. یه نمونه‌ی معروفش «راش آور» (Rush Hour) هست که ماشین و کامیون‌ها رو تو صفحه‌ی ۶×۶ حرکت می‌دیم تا یه ماشین خاص رو از ترافیک بیرون بیاریم. اما شاید شناخته‌شده‌ترینش، «پازل ۸» باشه: یه صفحه‌ی ۳×۳ با هشت کاشی شماره‌دار و یه خونه خالی، و بعدش «پازل ۱۵» روی صفحه‌ی ۴×۴. هدف اینه به یه حالت مشخص برسیم، مثلا همونی که تو شکل سمت راست می‌بینیم. فرمول استاندارد پازل ۸ رو تو صفحه بعدی توضیح میدیم.



Start State



Goal State

حالت‌ها: موقعیت هر کاشی رو مشخص می‌کنه.
حالت شروع: هر کدوم از حالت‌ها می‌تونه نقطه‌ی شروع باشه. توجه کن که به خاطر یه خاصیت parity، از نصف حالت‌ها میشه به هدف رسید و از نصف دیگه نه.
عمل‌ها: تو واقعیت، کاشی‌ها جابه‌جا می‌شن، ولی ساده‌ترین دید اینه که ما می‌گیم «خونه‌ی خالی» می‌ره چپ، راست، بالا یا پایین. اگه خونه‌ی خالی لبه یا گوشه باشه، بعضی جهت‌ها قابل اجرا نیستن.
مدل انتقال: نشون می‌ده بعد از هر عمل چه وضعیتی پیش میاد؛ مثلا اگه تو شکل شروع عمل «چپ» رو اجرا کنیم، عدد ۵ و خونه‌ی خالی با هم جابه‌جا می‌شن.
حالت هدف: معمولا همون حالتیه که اعداد پشت سر هم مرتب شدن، مثل شکل.
هزینه‌ی هر عمل: همیشه ۱ در نظر می‌گیریم.

یادت باشه که تو همه‌ی این مسئله‌ها داریم انتزاع انجام می‌دیم. مثلا تو پازل ۸ صرفا «قبل» و «بعد» جابه‌جایی کاشی‌ها مهمه و رد شدنشون از وسط کاشی‌ها رو در نظر نمی‌گیریم. کارهایی مثل تکون دادن قاب پازل وقتی کاشی‌ها گیر می‌کنن یا کشیدن کاشی با چاقو رو حذف کردیم. در واقع فقط قوانین اصلی رو نگه داشتیم و همه‌ی جزییات واقعیِ دست‌کاری فیزیکی رو کنار گذاشتیم.

یه سری مثال از «مسأله‌های دنیای واقعی» که با رویکرد حل مسئله انجام شده
مسیریابی (Route Finding): همون‌طور که دیدیم، مسأله‌ی مسیریابی با فهرستی از مکان‌ها و گذرگاه‌ها (یال‌ها) بینشون تعریف می‌شه. این الگوریتم‌ها خیلی جاها کاربرد دارن:
وب‌سایت‌ها و سیستم‌های خودرویی که مسیر رانندگی می‌دن—خیلی شبیه مثال رومانیِ ماست. فقط پیچیدگی‌هاش اینه که هزینه‌ی مسیر بسته به ترافیک و بسته شدن جاده‌ها تغییر می‌کنه.
مسیر دادن جریان ویدیو توی شبکه یا برنامه‌ریزی عملیات نظامی یا سیستم‌های برنامه‌ریزی سفر هوایی، که کلی پیچیدگی توشون دارند.

مسأله‌های گردش (Touring Problems): اینجا به جای یک مقصد، فهرستی از مقصدها داریم که باید یکی‌یکی دید.
مسألهٔ فروشنده‌ی دوره‌گرد (TSP): همه‌ی شهرها باید یک بار دیده بشن. هدف اینه یک گردش با هزینه کمتر از C پیدا کنیم یا کلاً کمترین هزینه رو داشته باشه.
حساب کنید چقدر روش‌های بهینه‌سازی روش کار شده—مثلاً تو بوستون با بهینه‌سازی مسیر اتوبوس‌های مدرسه‌ای، ۵ میلیون دلار صرفه‌جویی کردن، ترافیک و آلودگی رو کم کردن و وقت راننده‌ها و دانش‌آموزها رو نجات دادن!

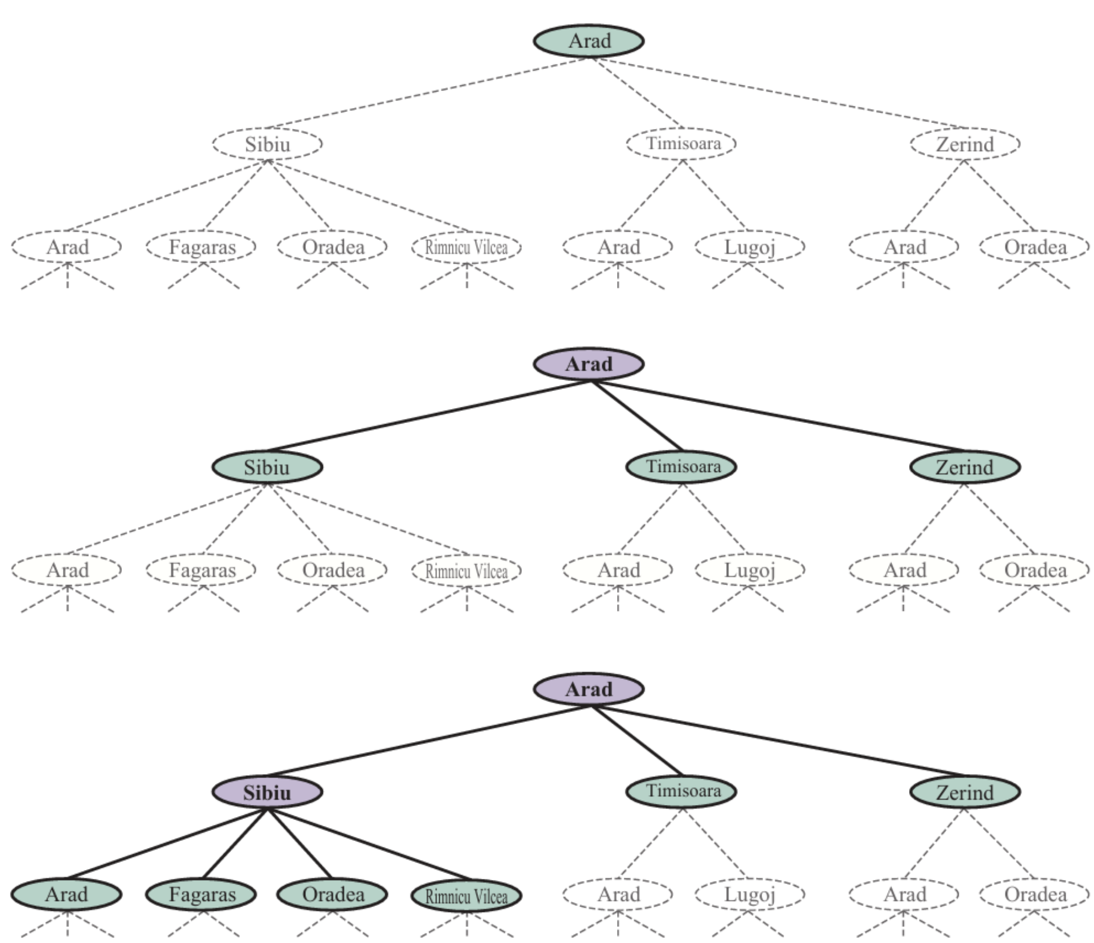
غیر از سفر، الگوریتم‌های جست‌وجو برای کارهایی مثل برنامه‌ریزی مسیر دریل‌های مدار چاپی یا چیدمان کالاها تو انبار هم استفاده می‌شن.

وقتی مسئله‌ی جست‌وجو رو داریم، یه الگوریتم جست‌وجو می‌گیره این مسئله رو و می‌گه یا راه‌حل پیدا شده یا نشد.
تو این فصل، الگوریتم‌هایی رو بررسی می‌کنیم که یک درخت جست‌وجو رو روی نمودار فضای حالت می‌ذارند، یعنی مسیرهای مختلف رو از حالت شروع تولید می‌کنند تا ببینند کدوم‌شون به هدف می‌رسه.
هر گره (node) توی درخت جست‌وجو نماینده‌ی یه حالت در فضای حالته و هر یال (edge) توی درخت نشون‌دهنده‌ی یه عمل (action) هست. ریشه‌ی درخت هم حالت شروع مسئله است.

خیلی مهمه که بدونیم فضای حالت با درخت جست‌وجو فرق داره:
فضای حالت خودش مجموعه (گاهی نامتناهی) همه‌ی حالت‌های ممکن و عمل‌هایی که حالت‌ها رو به هم وصل می‌کنه نشون می‌ده.
درخت جست‌وجو مسیرهایی رو نشون می‌ده که از حالت شروع به سمت هدف رشد می‌کنند.
درخت جست‌وجو ممکنه برای یک حالت، چند تا گره داشته باشه (چون ممکنه از راه‌های مختلف به یک حالت برسیم)، ولی هر گره توی درخت فقط یه مسیر یکتا تا ریشه داره (مثل همه‌ی درخت‌ها).

مثال آراد به بخارست (شکل پایین صفحه):
اول درخت با گره‌ی «آراد» شروع می‌شه.
اون گره باز (expand) می‌شه: عمل‌های ToSibiu، ToTimisoara، ToZerind رو بررسی می‌کنیم و با تابع RESULT می‌فهمیم هر کدوم به کدوم حالت می‌ره. برای هر حالت، یه گره فرزند می‌سازیم که پدرش گره‌ی «آراد» هست.
حالا باید انتخاب کنیم کدوم گره فرزند رو اول باز کنیم. این‌جوریه که جست‌وجو پیش میره—یه گزینه رو جلو می‌بریم و بقیه رو بعداً نگه می‌داریم. فرض کن اول «سیبیو» رو باز کنیم. حالا شش گره‌ی جدید داریم که هنوز باز نشده‌اند (مرز یا frontier درخت جست‌وجو).

هر حالتی که براش گره تولید شده باشه، می‌گیم «رسیده» (reached)، چه بازشده چه نه.



شکل ۳٫۵ نشون می‌ده چطور این درخت رو روی نمودار فضای حالت می‌ذارن. مرز جست‌وجو فضای حالت رو به دو بخش تقسیم می‌کنه:
منطقه‌ی داخلی که همه‌ی حالت‌هاش گره و باز شده‌اند، منطقه‌ی خارجی که هنوز بهشون نرسیدیم.

شکل ۳٫۶ هم این خواص رو شفاف‌تر می‌کنه.

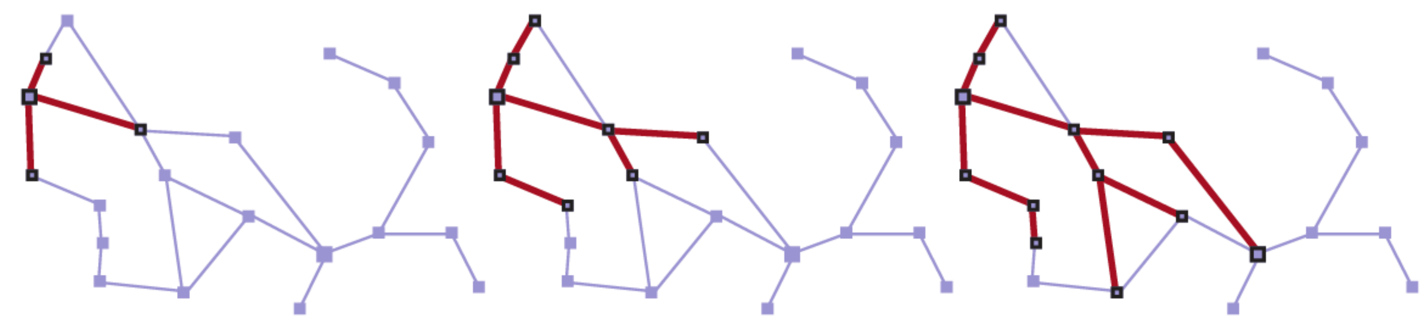


Figure 3.5 A sequence of search trees generated by a graph search on the Romania problem of Figure 3.1. At each stage, we have expanded every node on the frontier, extending every path with all applicable actions that don't result in a state that has already been reached. Notice that at the third stage, the topmost city (Oradea) has two successors, both of which have already been reached by other paths, so no paths are extended from Oradea.

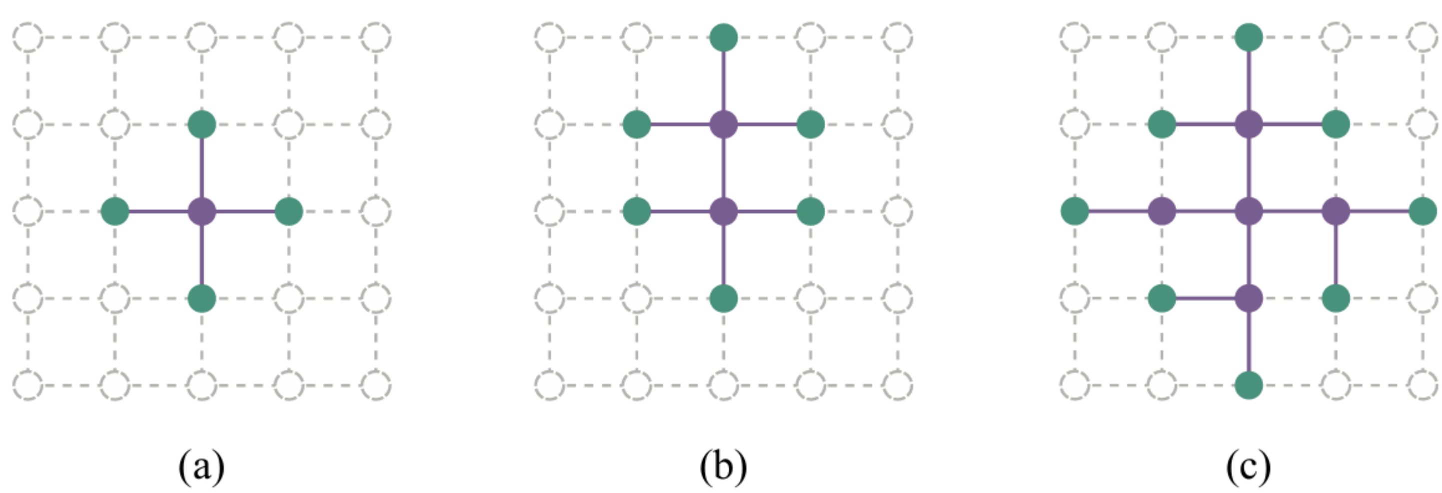


Figure 3.6 The separation property of graph search, illustrated on a rectangular-grid problem. The frontier (green) separates the interior (lavender) from the exterior (faint dashed). The frontier is the set of nodes (and corresponding states) that have been reached but not yet expanded; the interior is the set of nodes (and corresponding states) that have been expanded; and the exterior is the set of states that have not been reached. In (a), just the root has been expanded. In (b), the top frontier node is expanded. In (c), the remaining successors of the root are expanded in clockwise order.

جست‌وجوی اولین بهترین (Best First Search)

خب، از بین گره‌هایی که روی «مرز» (frontier) جمع شدن، چطوری انتخاب کنیم کدوم رو باز (expand) کنیم؟

یه روش کلی هست به اسم «بهترین-اول» (Best-First):

1. برای هر گره n یه تابع ارزیابی f(n) داریم.
2. هر بار می‌ریم سراغ گره‌ای که f(n) براش کمترین باشه.
3. اگه اون گره، وضعیتی باشه که هدفمون هست، کار تمومه و جواب رو برمی‌گردونیم.
4. وگرنه بچه‌های اون گره رو با EXPAND می‌سازیم و برای هر بچه:

اگه قبلاً بهش نرسیده بودیم، اضافه‌ش می‌کنیم به مرز.

اگه قبلاً رسیدیم ولی حالا با هزینه‌ی کمتر می‌رسیم، دوباره به مرز اضافه می‌کنیم.

5. همین کار رو ادامه می‌دیم تا یا راه‌حل پیدا بشه یا بفهمیم هیچ راهی نیست.

بسته به اینکه f(n) رو چطور تعریف کنیم (مثلاً تابع A* یا Greedy)، الگوریتم‌های متفاوتی به دست میاد.

ساختار داده‌ها برای جست‌وجو

برای مدیریت درخت جست‌وجو به یه ساختار داده نیاز داریم. هر گره تو درخت، این چهار تا فیلد رو داره:

STATE: خود وضعیت دنیای واقعی که گره نشون می‌ده.

PARENT: گره‌ای که باعث ساخت این گره شده (پدرش تو درخته).

ACTION: عملی که روی وضعیت پدر انجام شده تا این وضعیت جدید حاصل بشه.

PATH-COST یا g(node): جمع کل هزینه‌های مسیر از ریشه (شروع) تا این گره.

اگه از یه گره بیای عقب و مدام به PARENT نگاه کنی، می‌تونی کل مسیر و عمل‌های منتهی به اون گره رو پیدا کنی. از گره‌ی هدف که این کارو بکنی، راه‌حل به دست میاد.

برای «Frontier» هم یه صف (queue) لازمه که این عملیات رو داشته باشه:

IS-EMPTY: بگه خالیه یا نه.

POP: گره‌ی بالای صف رو برداره و بده بیرون.

TOP: گره‌ی بالای صف رو فقط نشون بده‌ش (ولی برنداره).

ADD: یه گره جدید اضافه کنه تو جای درستش.

سه جور صف معمولاً استفاده می‌شه:

صف اولویت‌دار (Priority Queue): اول گره با کمترین f(n)رو می‌ده بیرون—مناسب Best-First Search.

صف FIFO (First-In-First-Out): اول گره‌ای که زودتر اضافه شده رو می‌ده—مناسب جست‌وجوی عرضی (Breadth-First).

پشته یا LIFO: اول آخرین گره اضافه‌شده رو می‌ده—مناسب جست‌وجوی عمق‌اول (Depth-First).

حالت‌هایی که قبلاً رسیدیم رو معمولاً تو یه جدول (مثلاً هش‌تیبل) نگه می‌داریم که هر کلیدش یه وضعیت و مقدارش گره مربوطه. اینطوری می‌تونیم از تکرار مسیرهای زائد جلوگیری کنیم.

حلقه‌ها و مسیرهای زائد

اگه درخت جست‌وجو رو خوب نگاه کنی، می‌بینی ممکنه دوباره به «آراد» برگردیم—آراد به‌عنوان یه حالت تکراری تو درخت هست. این میشه یه حلقه (cycle) یا مسیر لوپی. با وجود اینکه فضای حالت فقط ۲۰ تا وضعیت داره، درخت ما چون حلقه‌ها رو نامحدود می‌تونه طی کنه، عملاً بینهایت گره داره!

به غیر از حلقه، مسیرهای زائد هم داریم—مثلاً رسیدن به سیبیو از مسیر ساده‌ی «آراد→سیبیو» (۱۴۰ مایل) یا مسیر طولانی «آراد→زِریند→اورادئا→سیبیو» (۲۹۷ مایل). مسیر دوم همون رسیدن رو تکرار کرده ولی کلی اضافات داره؛ پس بی‌خودیه.

مثلاً تو یه شبکه‌ی ۱۰×۱۰ بدون مانع، تعداد مسیرهای ۹ قدمی بیش از ۱۰۰ میلیون میشه! پس اگر بتونیم مسیرهای زائد رو حذف کنیم، می‌تونیم میلیون‌ها برابر سریع‌تر جست‌وجو کنیم. به قول معروف، الگوریتم‌هایی که گذشته رو فراموش می‌کنن محکوم به تکرار اشتباهات‌شون هستن.

سه راه حل اصلی داریم:

حفظ همه‌ی وضعیت‌های رسیده (Graph Search): مثل Best-First Search، جدول رسیدن داریم و بهترین مسیر به هر وضعیت رو نگه می‌داریم. مناسب وقتی فضای حالت پر از مسیر زائده و حافظه داریم.

بی‌خیال مسیرهای تکراری بشیم (Tree-Like Search): اصلاً جدول رسیدن نداریم، مناسب وقتی هیچ دو مسیری به یک وضعیت نمی‌رسن (مثلاً تو مونتاژ قطعات با ترتیب خاص). این کم حافظه‌تره ولی مسیر زائد زیاد بررسی می‌کنه.

فقط حلقه‌ها رو حذف کنیم (Cycle Checking): بدون جدول اضافه، با دنبال کردن پدرها تا ریشه می‌تونیم ببینیم این وضعیت قبلاً تو مسیر بوده یا نه. بعضی پیاده‌سازی‌ها کل زنجیره رو می‌گردن (حلقه‌های بلند و کوتاه رو حذف می‌کنن) و بعضی‌ها فقط چند تا پدر بالا رو نگاه می‌کنن تا فقط حلقه‌های کوتاه حذف بشن (سرعت بیشتر و حافظه ثابت).

هر روش بسته به مسئله و محدودیت حافظه، مزایا و معایب خودش رو داره.

چجوری عملکرد الگوریتم‌های جست‌وجو رو بسنجیم؟!

قبل از اینکه بریم سراغ طراحی الگوریتم‌های مختلف، باید بدونیم چجوری بینشون انتخاب کنیم. معمولاً چهار تا معیار اصلی داریم:

کامل بودن (Completeness): آیا وقتی راه‌حل وجود داره الگوریتم حتماً پیداش می‌کنه؟ و وقتی راهی نیست، درست می‌گه «راهی نیست»؟

بهینگی هزینه (Cost optimality): آیا الگوریتم کم‌هزینه‌ترین راه‌حل ممکن رو پیدا می‌کنه؟

پیچیدگی زمانی (Time complexity): چقدر طول می‌کشه جواب رو پیدا کنه؟ یا توی عمل چقدر وضعیت و عمل بررسی می‌کنه.

پیچیدگی فضایی-حافظه (Space complexity): برای جست‌وجو چقدر حافظه لازم داره؟

کامل بودن

فرض کن فقط یه هدف داریم که ممکنه هرجایی تو فضای حالت باشه؛ پس یه الگوریتم کامل باید بتونه هر حالتی که از شروع قابل دسترسه رو بالاخره بررسی کنه—مخصوصاً تو فضای حالت محدود، کافیه حلقه‌ها رو ببندیم (مثلاً آراد→سیبیو→آراد) و جلو بریم تا همه‌ی حالت‌های ممکن رو رد کنیم.

ولی تو فضای حالت نامتناهی، کار سخت‌تره. مثلاً اگه الگوریتم‌مون بدون قید، مداوم عدد قبلی رو فاکتوریل بگیره (۴ → !۴ → (!۴)! → ...)، یا تو یه شبکه‌ی بزرگ بی‌پایان فقط مستقیم بریم جلو و جلو، هیچ‌وقت برنمی‌گرده ولی عملاً بخش بزرگی از محیط رصد نشده می‌مونه.

پس تو حالت نامتناهی، برای کامل بودن باید یه روش سیستماتیک داشته باشیم—مثلاً روی شبکه‌ای که نامتناهی‌ئه، می‌تونیم اول همه‌ی خانه‌هایی که ۱ قدم از مبدأ فاصله دارن رو بررسی کنیم، بعد همه‌ی خانه‌هایی که ۲ قدم فاصله دارن، بعد ۳ قدم و ... (مثل حرکت مارپیچ). اما اگه راهی برای هدف وجود نداشته باشه، این روش هیچ‌وقت تموم نمی‌شه—چون هیچ‌وقت نمی‌تونیم مطمئن باشیم مرحله‌ی بعدی نره روی هدف.

پیچیدگی زمان و فضا

برای سنجش این دو، یه معیار معمول اینه که اندازه‌ی «گراف فضای حالت» رو در نظر بگیریم:

|V|: تعداد رأس‌ها (حالت‌ها).

|E|: تعداد یال‌ها (جفت‌های حالت/عمل).

اما تو خیلی از مسأله‌های هوش مصنوعی، خود گراف رو نگه نمی‌داریم و فقط با تابع انتقال و حالت شروع سر و کار داریم. اینجا معمولا از پارامترهای زیر استفاده می‌کنیم:

d یا depth: عمق (تعداد عمل‌های) راه‌حل بهینه (عمق بهینه‌ترین/کم‌عمق‌ترین جواب مسئله).

m: بیش‌ترین طول مسیر ممکن (عمق گراف).

b یا branch factor: ضریب انشعاب (تعداد بچه‌های هر گره).

این پارامترها کمک می‌کنن بفهمیم الگوریتم‌مون در بدترین حالت چقدر وضعیت می‌تونه باز کنه یا چقدر حافظه لازم داره.

وقتی که عامل هیچ سرنخی از این‌که چقدر به هدف نزدیکه نداره، می‌گیم داره با «جست‌وجوی بی‌اطلاع» (uninformed) کار می‌کنه. مثلاً اگه عامل ما تو آراد باشه و هدفش بخارسته، ولی هیچ اطلاعاتی از نقشه رومانی نداشته باشه، نمی‌دونه اول بره زِرِیند یا سیبیو بهتره. در مقابل، یه عامل «آگاه» (informed) می‌دونه سیبیو خیلی به بخارست نزدیک‌تره و احتمالاً اون مسیر رو امتحان می‌کنه.

جست‌وجوی عرض-اول (Breadth-First Search)

اگه همه‌ی عمل‌ها هزینه‌ی برابر داشته باشن، روش مناسب، جست‌وجوی عرض-اوله. اینجا اول ریشه (حالت شروع) باز می‌شه، بعد همه‌ی فرزندانش، بعد همه‌ی فرزندانِ فرزندان و الی آخر. چون این کار سیستماتیکه، حتی تو فضای حالت نامتناهی هم کامل عمل می‌کنه—یعنی اگه راه‌حلی باشه پیداش می‌کنه.

می‌تونیم Breadth-First رو با فراخوانی BEST-FIRST-SEARCH پیاده کنیم و بذاریم f(n) عمق نود (تعداد قدم‌ها تا نود) باشه. ولی می‌تونیم ساده‌تر و سریع‌ترش کنیم.

استفاده از صف FIFO: به جای صف اولویت‌دار، از یه صف اول-ورود-اول-خروج استفاده می‌کنیم که خودش ترتیب عمق رو حفظ می‌کنه. هر بچه‌ای که ساخته می‌شه عمقش بیشتره و می‌ره انتهای صف، بعد نوبت به گره‌های قدیمی‌تر می‌رسه.

ذخیره‌ی مجموعه‌ی حالات رسیده: چون هزینه‌ها برابرن، هیچ‌وقت یه مسیر بهتر پیدا نمی‌شه؛ پس کافیه فقط یه مجموعه از «حالاتی که تا حالا رسیدیم» نگه داریم، نه نگاشت به نودها. اینطوری می‌تونیم به‌محض ساخت گره جدید چک کنیم که آیا هدفه یا نه—یعنی «آزمون هدف زودهنگام یا early goal test»—و اگه هست، فوراً جواب رو تحویل بدیم، به جای این‌که صبر کنیم تا اون گره از صف بیرون بیاد.

شکل ۳٫۸ نشون می‌ده تو یه درخت دودویی چطور لایه‌به‌لایه گره‌ها گسترش پیدا می‌کنن و شکل ۳٫۹ هم الگوریتم بهینه‌شده‌ش رو با صف FIFO و آزمون هدف زودهنگام نمایش می‌ده.

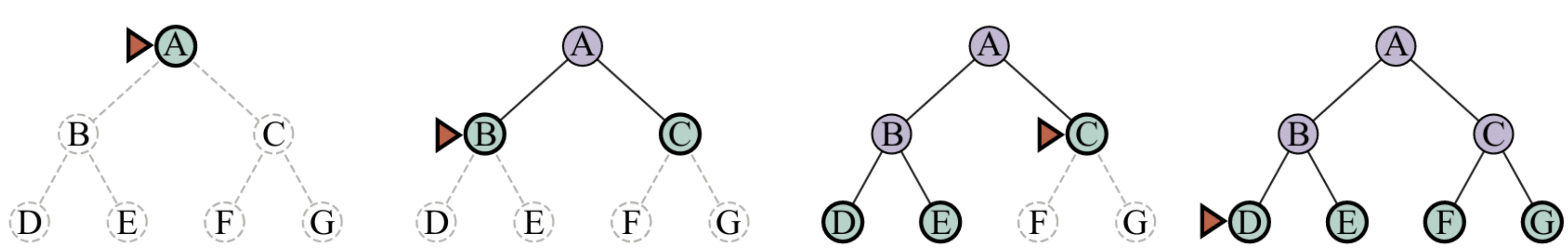


Figure 3.8 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by the triangular marker.

```
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
  node ← NODE(problem.INITIAL)
  if problem.IS-GOAL(node.STATE) then return node
  frontier ← a FIFO queue, with node as an element
  reached ← {problem.INITIAL}
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if problem.IS-GOAL(s) then return child
      if s is not in reached then
        add s to reached
        add child to frontier
  return failure

function UNIFORM-COST-SEARCH(problem) returns a solution node, or failure
  return BEST-FIRST-SEARCH(problem, PATH-COST)
```

Figure 3.9 Breadth-first search and uniform-cost search algorithms.

ویژگی‌ها

کامل: همیشه اگه راه‌حلی باشه پیداش می‌کنه.

بهینه از نظر تعداد قدم: چون وقتی داره عمق d رو می‌سازه، همه‌ی گره‌های عمق d > (کوچک‌تر از d) رو قبلاً ساخته، پس حتماً کوتاه‌ترین مسیر رو پیدا می‌کنه (وقتی همه‌ی هزینه‌ها برابرن).

پیچیدگی زمان و فضا: فرض کنیم ضریب انشعاب درخت b باشه (یعنی هر گره b تا فرزند). اگر راه‌حل در عمق D باشه، تعداد نودهای تولیدشده می‌شه: $1 + b + b^2 + \dots + b^d = O(b^d)$

و چون همه تو حافظه مونده، پیچیدگی زمان و فضا هر دو $O(b^d)$ هستن—خیلی ترسناکه!

برای مثال، اگه b=10 و سرعت ۱ میلیون نود در ثانیه و هر نود ۱ کیلوبایت، تا عمق d=10 حدود ۳ ساعت طول می‌کشه ولی ۱۰ ترابایت حافظه لازم داره. عمق ۱۴ حتی با حافظه‌ی نامحدود میشه ۳٫۵ سال!

پس در عمل، جست‌وجوی عرض-اول برای جزئیات خیلی کوچک جواب می‌ده، ولی برای نمونه‌های بزرگ معمولاً باید سراغ روش‌های آگاه‌تر یا الگوریتم‌های خاص‌تر بریم.

دیجکسترا یا جستجو با هزینه‌ی یک‌نواخت (Uniform-Cost Search)

وقتی هزینه‌ی کارها با هم فرق داره، می‌تونیم از بهترین-اول (Best First Search) استفاده کنیم ولی بذاریم تابع ارزیابی‌مون، مجموع هزینه‌ای باشه که از ریشه تا گره‌ی فعلی طی شده. تو دنیای علوم کامپیوتر تئوری بهش می‌گن «الگوریتم دیکسترا» و تو هوش مصنوعی می‌گن **«جست‌وجوی هزینه یکنواخت»**. ایده‌ش اینه که همان‌طور که جست‌وجوی عرض-اول در لایه‌های عمق جلو می‌ره (اول عمق ۱، بعد ۲ و الی آخر)، این یکی **بر اساس هزینه‌ی مسیر پیش می‌ره**؛ یعنی اول مسیرهای ارزون‌تر رو کامل می‌کنه، بعد کم‌کم مسیرهای گرون‌تر رو امتحان می‌کنه.
می‌تونیم همین کار رو با فراخوان BEST-FIRST-SEARCH انجام بدیم و بذاریم f(n) همون PATH-COST باشه (شکل ۳٫۹).

فرض کن از سیبیو می‌خوایم بریم بخارست (شکل ۳٫۱۰).
از سیبیو دو تا مسیر داریم: به رمنیکو ویلچا با هزینه‌ی ۸۰ و به فاگاراس با هزینه‌ی ۹۹.
رمنیکو ویلچا ارزون‌تره، پس اول به رمنیکو بازش می‌کنیم و بچه‌هاش رو اضافه می‌کنیم: پیتشتی با هزینه‌ی ۹۷+۸۰=۱۷۷.
حالا ارزون‌ترین گره تو frontier، فاگاراس با ۹۹ هست؛ بازش می‌کنیم و بچه‌هاش رو اضافه می‌کنیم: بخارست با هزینه‌ی ۲۱۱+۹۹=۳۱۰.
می‌دونیم که بخارست هدفه، ولی چون الگوریتم ما موقع تولید گره‌ها چک نمی‌کنه و فقط وقتی باز می‌کنه تست هدف می‌زنه، هنوز نمی‌دونه این مسیر به بخارست رسیده!
پس ادامه می‌ده و بعد پتیشتی (۱۷۷) رو باز می‌کنه و دوباره بخارست رو با هزینه‌ی ۱۰۱+۹۷+۸۰=۲۷۸ می‌سازه.
چون ۲۷۸ کمتر از ۳۱۰ ائه، مسیر قبلی رو با این جا به‌روز می‌کنه.
حالا ارزون‌ترین مسیر ۲۷۸ئه، بازش می‌کنه، می‌بینه هدفه و همونو برمی‌گردونه.

وقتی تست هدف رو زودهنگام (سرِ تولید) می‌ذاشتیم، زودتر بخارست ۳۱۰ رو پیدا می‌کردیم که گرون‌تر بود!

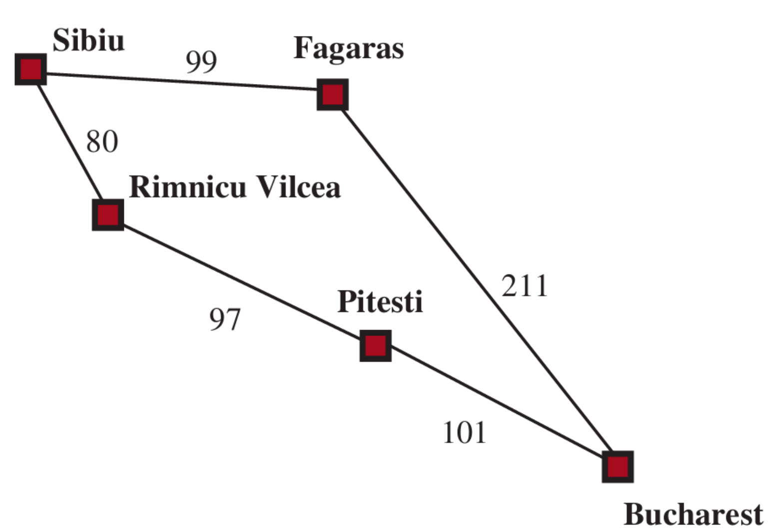


Figure 3.10 Part of the Romania state space, selected to illustrate uniform-cost search.

پیچیدگی الگوریتم

بیایم *C رو هزینه‌ی کمترین راه‌حل بدونیم و ε یه حد پایین‌تر مثبت برای هزینه‌ی هر عمل. در این صورت بدترین حالت زمان و فضا به صورت $O(b^{1+\lfloor \frac{C*}{\varepsilon} \rfloor})$ هست که ممکنه خیلی بزرگ‌تر از $O(b^d)$ باشه. چون اینجا الگوریتم اول همه‌ی مسیرهای با هزینه‌ی کم رو می‌گرده و بعد می‌ره سراغ مسیرهای گرون‌تر. اگر هم هزینه‌ها برابر باشن (همون حالتی که $O(b^{1+\lfloor \frac{C*}{\varepsilon} \rfloor})$ میشه $O(b^{d+1})$)، عملاً مثل عرض-اول رفتار می‌کنه.

این الگوریتم

کامله (اگر ε>0 باشه حتماً به جواب می‌رسه)

و بهینه از نظر هزینه (اولین بار که به هدف برسیم، کمترین هزینه‌ی ممکنه)

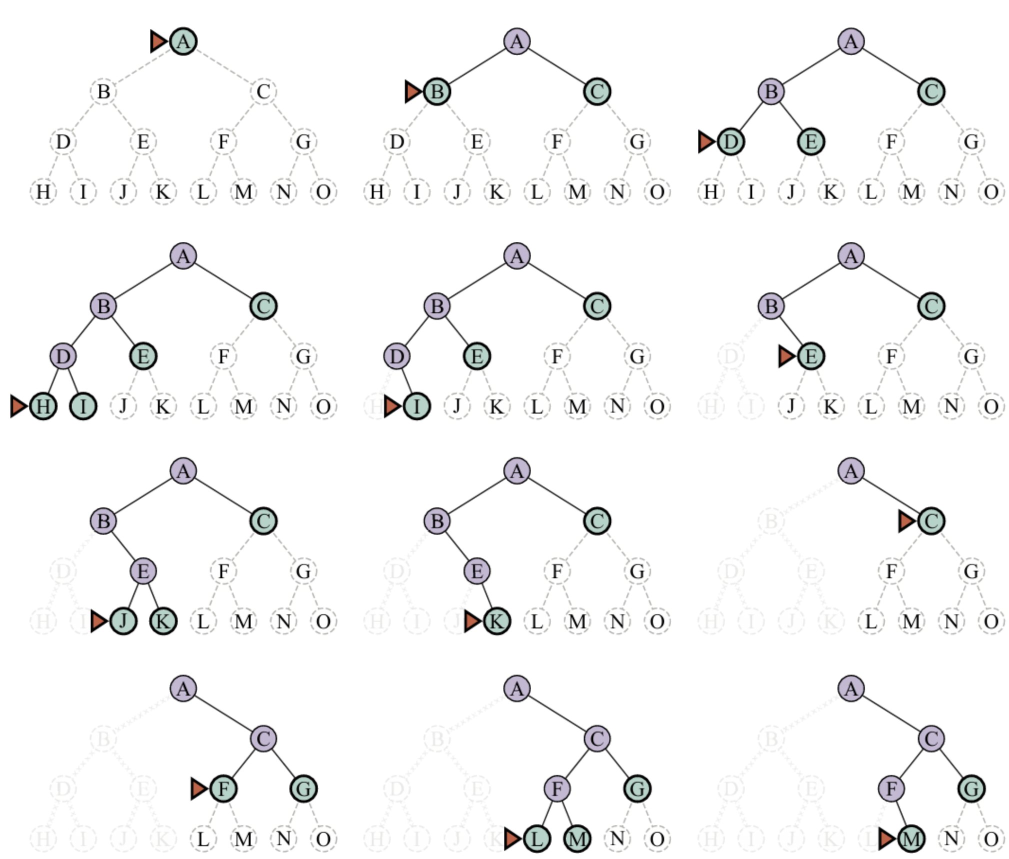
چون همیشه مسیرها رو به ترتیب هزینه‌شون بررسی می‌کنه و سرِ یه مسیر بی‌نهایت نمی‌مونه.

جستجوی عمق اول (Depth First Search)

جست‌وجوی عمق‌اول همیشه اول عمیق‌ترین نود روی frontier رو بسط (expand) می‌کنه. از نظر تئوری می‌شه با فراخون BEST-FIRST-SEARCH و گذاشتن f(n) = negative of the depth اجراش کرد، ولی معمولاً ساده‌ترش می‌کنن: جست‌وجو رو در قالب یک درخت می‌زنن و هیچی از «حالات رسیده» رو ذخیره نمی‌کنن. توی شکل پایین می‌بینی که اول میره تا ته درخت—تصور کن تا جایی که دیگه نودها بچه ندارن—بعد از اون یکی‌یکی برمی‌گرده بالا تا به اولین نودی برسه که هنوز بچه‌های بازنشده داره.

این روش اولین راه‌حل پیداشده رو برمی‌گردونه، حتی اگه گرون‌تر از بقیه باشه؛ پس بهینه نیست.
تو فضای حالت محدود و درختی، هم سریعه و هم کامل (چون درخته و حلقه و مسیر زائد هم نداریم).
اگه فضای حالت بدون حلقه باشه ولی نه الزاماً درخت، ممکنه یه حالت رو چند بار ببینه ولی بالاخره کل فضا رو می‌گرده.
اگه فضای حالت چرخه‌ای باشه، ممکنه تو یه حلقه‌ی بی‌نهایت گیر کنه، پس بعضی پیاده‌سازی‌ها اضافه می‌کنن که هر نود جدید رو با مسیرش به ریشه چک کنن و اگه قبلاً اون حالت تو مسیر بود، نرن تو حلقه.
تو فضای نامتناهی هم کامل نیست، چون ممکنه تا ابد تو یه مسیر عمیق پیش بره و هیچ‌وقت زوایه‌های جدید رو بررسی نکنه.

با این شرایط بد، چرا اصلاً کسی از Depth-First Search استفاده می‌کنه؟
جوابش ساده‌ست: حافظه‌ی خیلی کمی می‌خواد.
هیچ فهرستی از «حالات رسیده» نگه نمی‌داره.
مرز جست‌وجو خیلی کوچیکه—مثل یک شعاع در یک کره—در حالی که تو جست‌وجوی عرض-اول مرز مثل سطح کره بزرگ و بزرگ‌تر می‌شه.
پس وقتی حافظه‌ات کمه و ساختار مسأله جوریه که می‌تونی درختی روش جست‌وجو کنی، Depth-First Search انتخاب معقولیه.



توی یه فضای حالت درختیِ محدود مثل شکل صفحه قبل، جست‌وجوی عمق اول وقتی به‌صورت درخت‌گونه (بدون ذخیره‌ی حالت‌های رسیده) اجرا بشه، زمانش تقریباً متناسب با تعداد همه‌ی حالاتِ درخته، ولی حافظه‌ش فقط O(bm) هست؛ یعنی به ازای ضریب انشعاب b و بیش‌ترین عمق m، به همون اندازه حافظه لازم داره.
بعضی مسئله‌ها که با جست‌وجوی عرض اول به حافظه‌ای به اندازه‌ی اگزابایت نیاز دارن، با همین روش عمق اول فقط به چند کیلوبایت فضا نیاز پیدا می‌کنن.

جستجوی عمق محدود و تعمیق شونده‌ی تکراری

عمق محدود (Depth Limited Search):

برای اینکه Depth-First Search تو فضای نامتناهی تا ابد تو یه مسیر عمیق گم نشه، می‌تونیم یه حد عمق تعیین کنیم. یعنی وقتی عمق نود به L رسید، فرض می‌کنیم دیگه هیچ فرزندی نداره و بازش نمی‌کنیم.
پیچیدگی زمانیش میشه $O(b^L)$ و حافظه $O(bL)$. اما اگه L رو اشتباه انتخاب کنیم و از عمق راه‌حل کمتر باشه، الگوریتم هیچ‌وقت به جواب نمیرسه—یعنی دوباره ناقص میشه.

انتخاب حد عمق

گاهی میشه بر اساس دانش مسئله حد رو مشخص کرد. مثلاً تو نقشه‌ی رومانی فقط ۲۰ تا شهر داریم، پس L=19 کفایت می‌کنه. یا با کمی دقت می‌بینیم از هر شهر حداکثر با ۹ حرکت میشه به هر شهر دیگه رسید (قطر گراف).
این عدد، که بهش قطر گراف میگن، حد بهتریه و جست‌وجو رو کارآمدتر می‌کنه. ولی در عمل تا وقتی مسئله رو حل نکنیم، معمولاً عدد خوبش رو نمی‌دونیم.

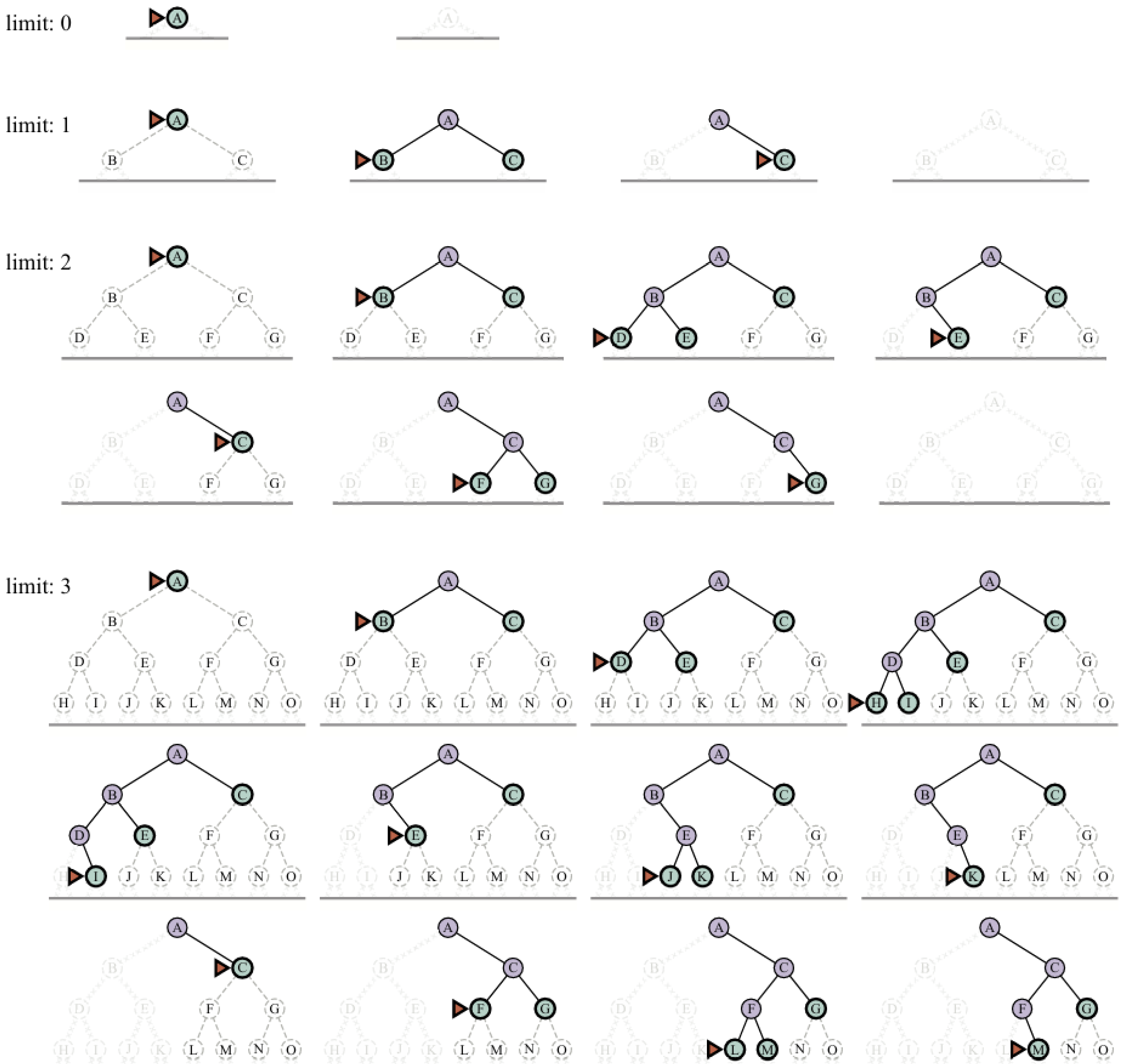
جست‌وجوی عمق‌افزایشی (Iterative Deepening Search):

برای حل مشکل تعیین حد عمق، Iterative Deepening همه‌ی عمق‌های ممکن رو از ۰ شروع تا … امتحان می‌کنه، یعنی:

1. عمقِ ۰
2. عمقِ ۱
3. عمقِ ۲
4. …

تا یا راه‌حل پیدا کنه یا Depth-Limited Search شکست بخوره (نه اینکه به حد برخورد کنه).

مثل Depth-First حافظه‌ی کمی نیاز داره: $O(b^d)$ وقتی جواب باشه، یا $O(b^m)$ تو فضای نامتناهی یا بی‌جواب.
مثل Breadth-First برای هزینه‌های برابر بهینه و کامل عمل می‌کنه (توی گراف‌های بی‌حلقه یا وقتی حلقه‌ها رو تشخیص بدیم).


limit: 0
limit: 1
limit: 2
limit: 3

جستجوی دو طرفه (Bidirectional Search)

یه روش دیگه هست به اسم **«جست‌وجوی دوطرفه» (Bidirectional Search)**. تا الان همش از حالت شروع جلو رفتیم تا برسیم به هدف، ولی اینجا هم‌زمان از دو طرف کار می‌کنیم: هم از حالت شروع جلو می‌ریم، هم از حالت(های) هدف عقب‌گرد می‌کنیم، و امیدواریم که این دو مسیر وسط راه به هم برسن. دلیلش اینه که به‌جای $O(b^d)$، دو تا $O(b^{\frac{d}{2}})$ می‌زنیم که جمع‌شون خیلی کمتر از $O(b^d)$ می‌شه. برای اینکار لازمه:

1. دو تا صف (frontier) نگه داریم—یکی برای جست‌وجوی جلو و یکی برای جست‌وجوی عقب.
2. دو تا جدول «حالات رسیده» داشته باشیم.
3. بتونیم عقب‌گرد هم بکنیم: یعنی اگه تو جست‌وجوی جلو دیدیم که S′ از S ساخته میشه، تو جست‌وجوی عقب هم بدونیم که S جانشین S′ هست.

وقتی مرزها به هم برخورد کنن، یعنی رسیدیم به یه حالت مشترک و راه‌حل پیدا شده.

مثل همیشه، می‌شه این ایده رو با روش‌های مختلف (مثل Best-First یا Uniform-Cost) ترکیب کرد.
مثلاً «بهترین-اول دوطرفه» داریم که هر بار از بین هر دو صف گره‌ای رو باز می‌کنه که کمترین مقدار تابع ارزیابی f رو داره. اگه f همون هزینه‌ی مسیر باشه، میشه «Uniform-Cost دوطرفه» که در این حالت هیچ گره‌ای با هزینه‌ی بالاتر از $O\left(\frac{C^*}{2}\right)$ باز نمی‌شه.

```
function BIBF-SEARCH(problem_F, f_F, problem_B, f_B) returns a solution node, or failure
  node_F ← NODE(problem_F.INITIAL)                    // Node for a start state
  node_B ← NODE(problem_B.INITIAL)                    // Node for a goal state
  frontier_F ← a priority queue ordered by f_F, with node_F as an element
  frontier_B ← a priority queue ordered by f_B, with node_B as an element
  reached_F ← a lookup table, with one key node_F.STATE and value node_F
  reached_B ← a lookup table, with one key node_B.STATE and value node_B
  solution ← failure
  while not TERMINATED(solution, frontier_F, frontier_B) do
    if f_F(TOP(frontier_F)) < f_B(TOP(frontier_B)) then
      solution ← PROCEED(F, problem_F, frontier_F, reached_F, reached_B, solution)
    else solution ← PROCEED(B, problem_B, frontier_B, reached_B, reached_F, solution)
  return solution

function PROCEED(dir, problem, frontier, reached, reached_2, solution) returns a solution
        // Expand node on frontier; check against the other frontier in reached_2.
        // The variable "dir" is the direction: either F for forward or B for backward.
  node ← POP(frontier)
  for each child in EXPAND(problem, node) do
    s ← child.STATE
    if s not in reached or PATH-COST(child) < PATH-COST(reached[s]) then
      reached[s] ← child
      add child to frontier
      if s is in reached_2 then
        solution_2 ← JOIN-NODES(dir, child, reached_2[s]))
        if PATH-COST(solution_2) < PATH-COST(solution) then
          solution ← solution_2
  return solution
```

مقایسه الگوریتم‌های جستجوی ناآگاهانه

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[1]	Yes[1,2]	No	No	Yes[1]	Yes[1,4]
Optimal cost?	Yes[3]	Yes	No	No	Yes[3]	Yes[3,4]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$

Figure 3.15 Evaluation of search algorithms. b is the branching factor; m is the maximum depth of the search tree; d is the depth of the shallowest solution, or is m when there is no solution; ℓ is the depth limit. Superscript caveats are as follows: [1] complete if b is finite, and the state space either has a solution or is finite. [2] complete if all action costs are $\geq \epsilon > 0$; [3] cost-optimal if action costs are all identical; [4] if both directions are breadth-first or uniform-cost.

جستجوهای آگاهانه (Heuristic)

توی این بخش می‌فهمیم چطور یه جست‌وجوی «آگاه»—یعنی جست‌وجویی که از سرنخ‌های مخصوص مسئله استفاده می‌کنه—می‌تونه خیلی سریع‌تر از جست‌وجوی ناآگاهانه جواب پیدا کنه. این سرنخ‌ها تو قالب یه تابع به اسم h(n) میان. h(n) = حدسِ (تخمین) هزینهٔ ارزان‌ترین مسیر از وضعیت گرهٔ n تا یکی از هدف‌ها.
مثلاً تو مسأله‌های مسیریابی، می‌تونیم فاصلهٔ مستقیم (پروازی) بین نقطهٔ فعلی و مقصد رو روی نقشه اندازه بگیریم و به‌عنوان h(n) در نظر بگیریم.

جستجوی اولین بهترین حریصانه (Greedy Best First Search)

جست‌وجوی بهترین-اول «حریصانه» (Greedy Best-First) در واقع همون جست‌وجوی بهترین-اوله که هر بار نودی رو باز می‌کنه که کمترین مقدار h(n) رو داره – یعنی همونی که از بقیه به نظر می‌رسه نزدیک‌تر به هدفه – چون فرض می‌کنیم این کار سریع‌تر ما رو به جواب می‌رسونه. پس تو این روش تابع ارزیابی میشه:

f(n) = h(n)

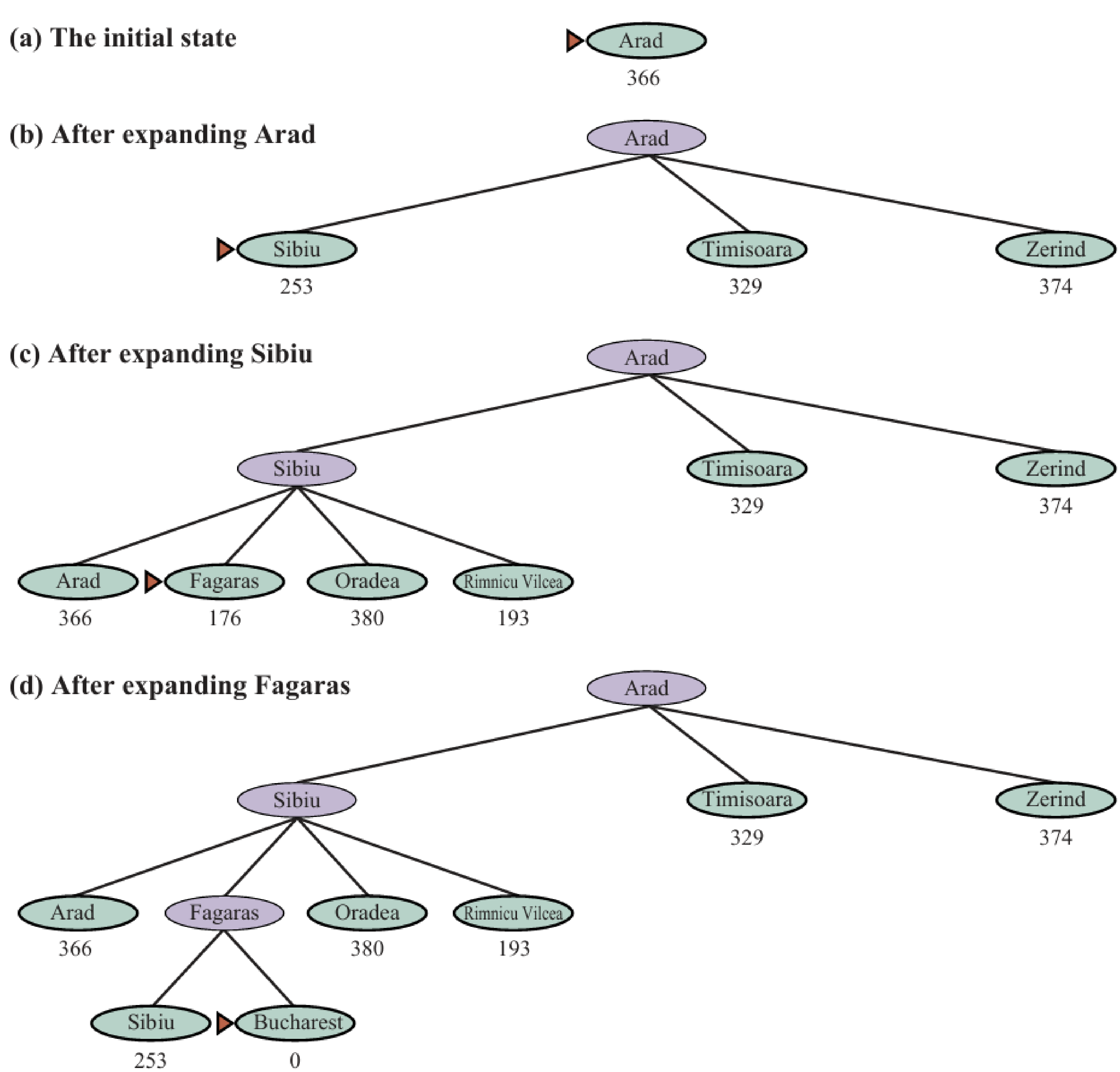
بیایید ببینیم این قضیه تو مسأله‌های مسیریابی رومانی چطور جواب می‌ده؛ ما از یه تابع راهنما استفاده می‌کنیم که فاصلهٔ خطراست بین هر شهر و بخارست رو حساب می‌کنه و بهش می‌گیم h_{SLD}. اگه هدفمون بخارسته، باید بدونیم فاصلهٔ خطراست هر شهر تا بخارست چقدره—این مقادیر تو شکل پایین اومده. برای مثال، $h_{SLD}(Arad) = 366$.
نکته‌ش اینه که این اعداد رو از خود تعریف مسأله (تابع‌های ACTIONS و RESULT) نمی‌تونیم دربیاریم؛ باید یه کم از دنیای واقعی سر دربیاریم تا بفهمیم فاصلهٔ خطراست با فاصلهٔ واقعی جاده‌ای رابطه داره و به همین خاطر این هیوریستیک می‌تونه مفید باشه.

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Figure 3.16 Values of h_{SLD}—straight-line distances to Bucharest.

شکل پایین نشون می‌ده که جست‌وجوی greedy best-first با استفاده از h_{SLD} چطور از آراد به سمت بخارست پیش می‌ره. اول از آراد، سیبیو باز می‌شه چون هوشگر می‌گه سیبیو از زِریند و تیمیشوارا به بخارست نزدیک‌تره. بعد نوبت فاگاراس می‌رسه چون حالا اون نزدیک‌ترین شهر به نظر میاد. فاگاراس هم که باز می‌شه، بخارست تولید می‌کنه و هدف به دست میاد. تو این مثال، جست‌وجوی greedy best-first با h_{SLD} بدون اینکه هیچ نودی رو باز کنه که تو مسیر راه‌حل نباشه، جواب رو پیدا می‌کنه. اما هزینه راه‌حلی که پیدا می‌کنه بهینه نیست: مسیر «آراد → سیبیو → فاگاراس → بخارست» حدود ۳۲ مایل طولانی‌تره نسبت به مسیر «آراد → رمنیکو ویلچا → پیتشتی → بخارست». به همین دلیله که بهش می‌گن «حریصانه» (greedy). هر مرحله سعی می‌کنه هر چی زودتر به هدف نزدیک بشه، اما این طمع‌ورزی گاهی می‌تونه نتیجه‌ی بدتری بده.

جست‌وجوی greedy best-first روی گراف تو فضای حالتِ محدود کامل عمل می‌کنه، ولی تو فضای بی‌نهایت کامل نیست. پیچیدگی زمان و فضا در بدترین حالت O(|V|)ئه. البته اگه هوشگر خوبی داشته باشیم، ممکنه پیچیدگی عملاً به $O(b^m)$ هم برسه و حسابی بهینه بشه.



جستجوی *A

جست‌وجوی *A (خوانده می‌شه «اِی‌ستار») پرکاربردترین الگوریتم جست‌وجوی آگاهه. این روش از سبک «بهترین-اول» استفاده می‌کنه و تابع ارزیابیش اینه: f(n) = g(n) + h(n).

g(n) همون هزینه‌ایه که از اول (حالت شروع) تا گره‌ی n طی شده.

h(n) یه حدس (تخمین) از هزینه‌ی کم‌هزینه‌ترین مسیر باقیمونده از n تا یکی از اهدافئه.

پس f(n) در حقیقت «هزینه‌ی تقریبی بهترین مسیریه که از n شروع بشه و به هدف برسه».

شکل ۳٫۱۸ (صفحه بعد) روند جست‌وجوی *A برای رسیدن به بخارست رو نشون می‌ده. مقدار g ها از هزینه‌های جاده‌ای شکل صفحه 40 حساب شده و h_{SLD} ها از شکل ۳٫۱۶ (صفحه 62). دقت کن که اولین بار بخارست تو مرحله‌ی (e) وارد مرز می‌شه، ولی بازش نمی‌کنیم—چون 450 = f داره و پایین‌ترین مقدار روی صف نیست؛ اون نود، پیتِستیه که 417 = f داره. یعنی ممکنه مسیری از پیتِستی به بخارست با هزینه‌ی ۴۱۷ وجود داشته باشه، پس فعلاً سراغ هزینه‌ی ۴۵۰ که گرون‌تره نمی‌ریم. تو مرحله‌ی (f)، مسیر جدیدی به بخارست با 418 = f کمینه، پس اون رو باز می‌کنیم و به عنوان جواب – یعنی کم‌هزینه‌ترین مسیر – برمی‌گردونیم.

*A کامله؛ یعنی حتماً اگه راه‌حلی باشه پیداش می‌کنه. اینکه بهینه از نظر هزینه هم باشه، بستگی به ویژگی «قابل قبول بودن» (Admissibility) داره. تابع هیوریستیک $h(n)$ قابل قبول هست اگر هیچ‌وقت هزینه‌ی مسیر باقیمانده تا هدف رو زیادتر از واقع برآورد نکنه—به عبارتی همیشه دست‌کم‌گیر باشه. با چنین هیوریستیک‌ئی می‌تونیم ثابت کنیم *A حتماً کم‌هزینه‌ترین مسیر رو برمی‌گردونه:

فرض کن بهترین (بهینه‌ترین) مسیر هزینه‌ش *C باشه، ولی الگوریتم مسیر گرون‌تری با هزینه‌ی C که C* < C برگردونده. پس باید یه نود n باشه که روی اون مسیر بهینه قرار داره ولی هنوز باز نشده (چون اگه باز شده بود، اون مسیر بهینه رو برمی‌گردوندیم). پس داریم:

اولی: C* < f(n) (چون n باز نشده)

دومی: f(n) = h(n) + g(n)

سومی: چون n روی مسیر بهینه‌ست، پس g(n) برابر هزینه‌ی مسیر بهینه تا n و h(n) دست‌کم هزینه‌ی بهینه از n تا هدفه، یعنی f(n) = h(n) + g*(n).

چهارمی: بخاطر قابل قبول بودن، h*(n) >= h(n)، پس f(n) <= h*(n) + g*(n).

و چون C* = h*(n) + g*(n)، داریم C* >= f(n).

این دو تا (اولی و آخری) با هم تناقض دارن (C* < f(n) و C* > f(n))، پس فرضمون اشتباهه و *A نمی‌تونه مسیری گرون‌تر از بهینه بده—حتما کم‌هزینه‌ترین مسیر رو برمی‌گردونه.

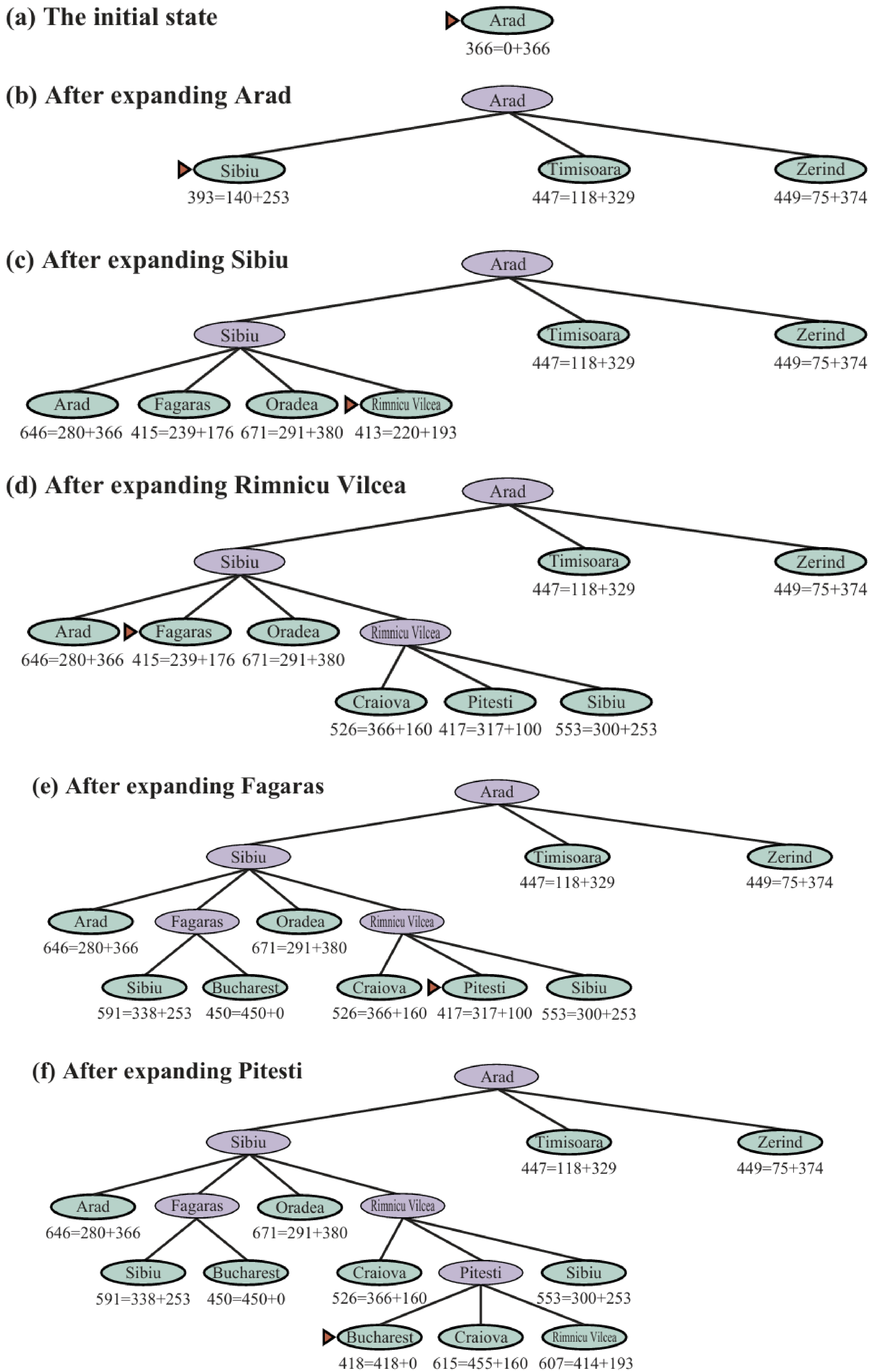


Figure 3.18 Stages in an A* search for Bucharest. Nodes are labeled with $f = g + h$. The h values are the straight-line distances to Bucharest taken from Figure 3.16.

یه خاصیت قوی‌تر هم هست به اسم سازگاری (Consistency) یا یکنوایی (Monotonic):
تابع h برای هر نود n و هر نودی که با عمل a ازش تولید میشه، یعنی 'n، باید این شرط برقرار باشه:

h(n) <= h(n') + c(n, a, n')

که (c(n, a, n' هزینه‌ی انجام همون عمله (مثلا هزینه رفتن از n به n‘). این یعنی برآورد هیوریستیک فاصله از n تا هدف، نه تنها نباید زیادتر از واقع باشه، بلکه باید از هزینه‌ی یک قدم جلو رفتن دل‌بخواه تا 'n به اضافه‌ی برآورد از 'n تا هدف هم تجاوز نکنه. این خاصیت تضمین می‌کنه که f مقدار غیرکاهشی داره و *A رو هم کاراتر می‌کنه.

این شرط در واقع همون «نابرابری مثلث» (triangle inequality) هست؛ یعنی هر ضلعِ مثلث نمی‌تونه از جمعِ دو ضلعِ دیگه بلندتر باشه (شکل ۳٫۱۹ رو ببین). یه مثال از هیوریستیک «سازگار» (consistent) همون فاصلهٔ خطراست h_{SLD} هست که توی مسیریابی رومانی استفاده کردیم.

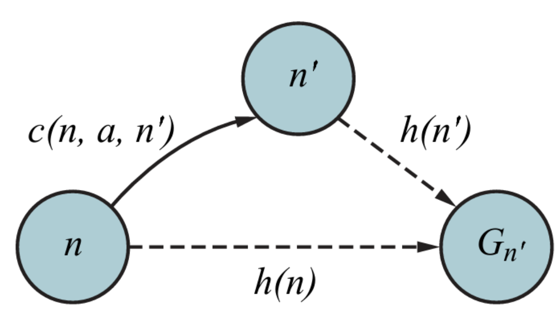


Figure 3.19 Triangle inequality: If the heuristic h is **consistent**, then the single number $h(n)$ will be less than the sum of the cost $c(n,a,a')$ of the action from n to n' plus the heuristic estimate $h(n')$.

هر هیوریستیک سازگار، خودش قبول‌پذیر (admissible) هم هست، ولی برعکسش لزوماً درست نیست. پس اگه هیوریستیک‌مون سازگار باشه، *A قطعاً کم‌هزینه‌ترین مسیر رو پیدا می‌کنه. ضمن اینکه با هیوریستیک سازگار وقتی برای اولین بار وارد یه حالت می‌شیم، حتماً از طریقِ یه مسیر بهینه بوده؛ بنابراین هیچ‌وقت لازم نیست یه حالت رو دوباره به صف جست‌وجو اضافه کنیم یا تو جدول «رسیده‌ها» بازنویسی‌اش کنیم.

اما اگه هیوریستیک‌مون سازگار نباشه، ممکنه چند تا مسیر مختلف به یه حالت برسن و هر بار مسیر جدید هزینه‌اش کمتر باشه؛ اون وقت مجبور می‌شیم هم تعداد زیادی نود تو صف نگه داریم و هم مرتب مسیرها رو آپدیت کنیم—که هم زمان می‌بره و هم حافظه زیادی مصرف می‌کنه. به همین دلیل بعضی پیاده‌سازی‌های A* طوری میشن که وقتی یه حالت رو یک بار به صف وارد کردیم، دیگه دوباره اون حالت رو وارد نکنن، بلکه اگه مسیرِ کم‌هزینه‌تری پیدا شد، همه‌ی فرزندان اون حالت رو با هزینه‌های به‌روز اصلاح کنن (برای همین لازم میشه که هم اشاره‌گر به پدر داشته باشیم و هم به فرزندان).

و در نهایت حتی اگه هیوریستیک‌مون قابل قبول نباشه، باز هم ممکنه A* بهینه باشه، در دو مورد:

۱. اگه حداقل یک مسیر بهینه وجود داشته باشه که روی تمام نودهای اون مسیر، هیوریستیک‌مون برآوردش اشتباه (بیش‌تر از واقع) نکرده باشه، الگوریتم همون مسیر رو پیدا می‌کنه—بی‌خیال بقیه‌ی حالت‌ها.

۲. اگه هزینه‌ی بهترین مسیر C* باشه و هزینه‌ی دومین مسیرِ بهینه C2، و هیوریستیک‌مون هیچ‌وقت بیشتر از (C2 - C*) اشتباه نکنه، آن‌وقت A* تضمین می‌کنه که حتماً کم‌هزینه‌ترین مسیر (همون با هزینه‌ی C*) رو برمی‌گردونه.

کانتورهای جستجو

یه راه مفید برای تصور جست‌وجو اینه که تو فضای حالت خطوط «هم‌ارزی» یا «کانتور» رسم کنیم، مثل نقشه‌های توپوگرافی. تو شکل ۳٫۲۰ این کار رو دیدیم: داخل خط 400 همه‌ی نودها f(n) = g(n) + h(n) <= 400 دارن، بین خط 400 و 500 همه 400 < f(n) <= 500 الی آخر. چون A* همیشه نود با کمترین مقدار f رو باز می‌کنه، می‌تونیم ببینیم جست‌وجو از شروع مثل یه سری حلقه‌ی متحدالمرکز با مقادیر f افزایشی عمل می‌کنه.

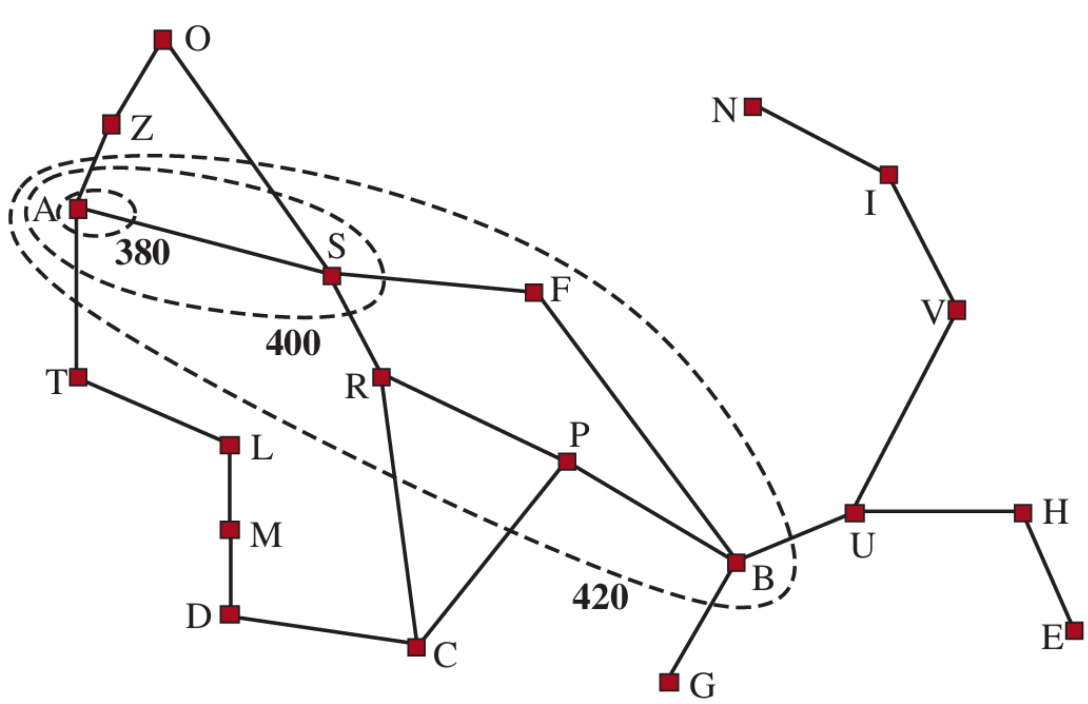


Figure 3.20 Map of Romania showing contours at $f = 380$, $f = 400$, and $f = 420$, with Arad as the start state. Nodes inside a given contour have $f = g + h$ costs less than or equal to the contour value.

اما مشخص نیست که همیشه مقدار f(n) = g(n) + h(n) روی مسیر پیوسته رشد کنه. وقتی از نود n یه قدم جلوتر می‌ریم به نود n′، هزینه از g(n) + h(n) میشه: g(n) + c(n, a, n′) + h(n′)، که با کسر کردن g(n) می‌بینیم واقعاً نرخ g + h فقط وقتی پیوسته افزایش پیدا می‌کنه که h(n) <= h(n′) + c(n, a, n′)، یعنی دقیقاً شرط «سازگاری» (consistency) روی هیوریستیک برقرار باشه. البته ممکنه تو مسیر گاهی f(n) ثابت بمونه—مثلاً تو شبکه‌ای که هر قدم هزینه‌ش یک واحده، وقتی همون‌قدر که g زیاد میشه، h کم میشه.

حالا اگه هزینه‌ی کمینه‌ی مسیر بهینه C* باشه، می‌تونیم این نکات رو بگیم:

1. حتماً همه‌ی نودهایی که روی هر مسیری باشن که برای هر نودش f(n) < C* هست باز می‌شن – این نودها نودهای حتماً باز شده محسوب می‌شن.
2. ممکنه قبل از رسیدن به خط کانتور f(n) = C* چندتا از همین نودها هم باز بشن – یعنی نودهایی با f(n) = C* رو هم تا وقتی گزینه بهتری نباشه باز می‌کنه.
3. هیچ نودی با f(n) > C* باز نمی‌شه.

وقتی هیوریستیک‌مون سازگار باشه، این خاصیت باعث می‌شه A* از نظر «توسعه نودها» بهینه باشه—هیچ الگوریتم دیگه‌ای که مسیرها رو از شروع گسترش بده و همون هیوریستیک رو داشته باشه، نمی‌تونه بدون اینکه حداقل همون نودهایی که A* باز می‌کنه، باز کنه به جواب برسه. (ممکنه تو بین نودهای f(n)=C* یکی زودتر به هدف برسه و یکی دیرتر؛ این بخت‌آزمایی به عنوان تفاوت در نظر گرفته نمی‌شه).

A* این کار رو با «هرس کردن» (pruning) نودهایی انجام می‌ده که برای یافتن مسیر بهینه ضروری نیستن. تو شکل ۳٫۱۸ می‌بینیم «تیمیشوارا» (f=447) و «زِریند» (f=449) هر دو بچه‌های ریشه هستن و Breadth-First یا Uniform-Cost اول بازشون می‌کردن، ولی A* هیچ‌وقت سراغشون نمی‌ره چون راه‌حل f=418 زودتر پیدا می‌شه. همین حذف به‌موقع باعث صرفه‌جویی عظیم در زمان و حافظه می‌شه.

این‌که A* کامل، بهینه و بهینه‌ترین بین همه‌ی الگوریتم‌های مشابهه خیلی خوبه، ولی چند تا نکته‌ی منفی هم داره: تو خیلی از مسائل تعداد نودهای بازشده همچنان می‌تونه نمایی باشه. مثلا فرض کن تو دنیای جاروبرقی یه جاروبرقی فوق‌العاده داریم که می‌تونه هر خانه‌ای رو بدون قدم‌زدن با هزینه‌ی 1 تمیز کنه؛ با N خانه‌ی کثیف در ابتدا، 2^N حالت داریم که هر زیرمجموعه‌ای از خانه‌ها تمیز شدن—و همه‌ی این حالت‌ها روی یک مسیر بهینه قرار می‌گیرن و چون f(n)<C* هستن، A* مجبور میشه همه‌شون رو باز کنه!

جست‌وجوی A* ویژگی‌های خیلی خوبی داره، اما تعداد زیادی نود رو باز می‌کنه. اگه حاضریم جواب‌هایی که پیدا می‌کنه صددرصد بهینه نباشن ولی «کافی و قابل قبول» باشن (یعنی Satisficing)، می‌تونیم نودهای کمتری بررسی کنیم و کلی تو زمان و حافظه صرفه‌جویی کنیم.

یکی از راه‌ها اینه که به A* اجازه بدیم از هوشگری استفاده کنه که قبول‌پذیر نیست—یعنی ممکنه هزینه‌ی باقیمانده تا هدف رو بیش‌تر از واقع بدن. اینطوری شانس پیدا کردن مسیر بهینه رو از دست می‌دیم، ولی هوشگر می‌تونه نزدیک‌تر به هزینه‌ی واقعی باشه و جست‌وجو رو متمرکزتر کنه.

مثلاً تو مهندسی راه‌سازی مفهوم شاخص انحراف (Detour Index) به کار میره: ضریبی که روی فاصلهٔ خطراست اعمال می‌کنن تا انحنای معمول جاده‌ها رو لحاظ کنن. شاخص انحراف ۱٫۳ یعنی اگه دو شهر ۱۰ مایل فاصله خطراست داشته باشن، یه برآورد خوب از فاصله‌ی واقعی مسیر بین‌شون میشه ۱۳ مایل. معمولاً این شاخص برای اکثر نواحی بین ۱٫۲ تا ۱٫۶ هست.

ما می‌تونیم این روش رو به هر مسأله‌ای به کار ببریم، نه فقط جاده و مسیر؛ این‌جا می‌رسیم به «A* وزنی» (Weighted A*) که توش به هیوریستیک وزن بیشتری می‌دیم. یعنی تابع ارزیابی میشه: $f(n) = g(n) + w \times h(n)$ که W ضریبی بزرگ‌تر از ۱ئه. تو شکل ۳٫۲۱ یه مثال روی شبکه‌شطرنجی داریم:
تو قسمت (a)، A* بهینه‌ترین جواب رو پیدا می‌کنه، ولی مجبور میشه کلی از فضای حالت رو بگرده.
اما تو قسمت (b)، A* وزنی مسیری پیدا می‌کنه که هزینه‌ش کمی بیشتره، ولی جست‌وجوش خیلی سریع‌تر تموم میشه. می‌بینیم که A* وزنی خط کانتور هزینه‌ها رو مستقیم می‌کشه سمت هدف، پس نودهای کمتری باز می‌شه.
البته اگه مسیر بهینه یه جایی بیفته بیرون از کانتور وزنیِ جست‌وجو، دیگه پیدا نمی‌شه.

به طور کلی، اگه هزینهٔ بهینه C^* باشه، A* وزنی چیزی بین C^* و $W \times C^*$ پیدا می‌کنه؛ ولی در عمل معمولا نزدیک‌تر به C^* می‌شه تا $W \times C^*$.

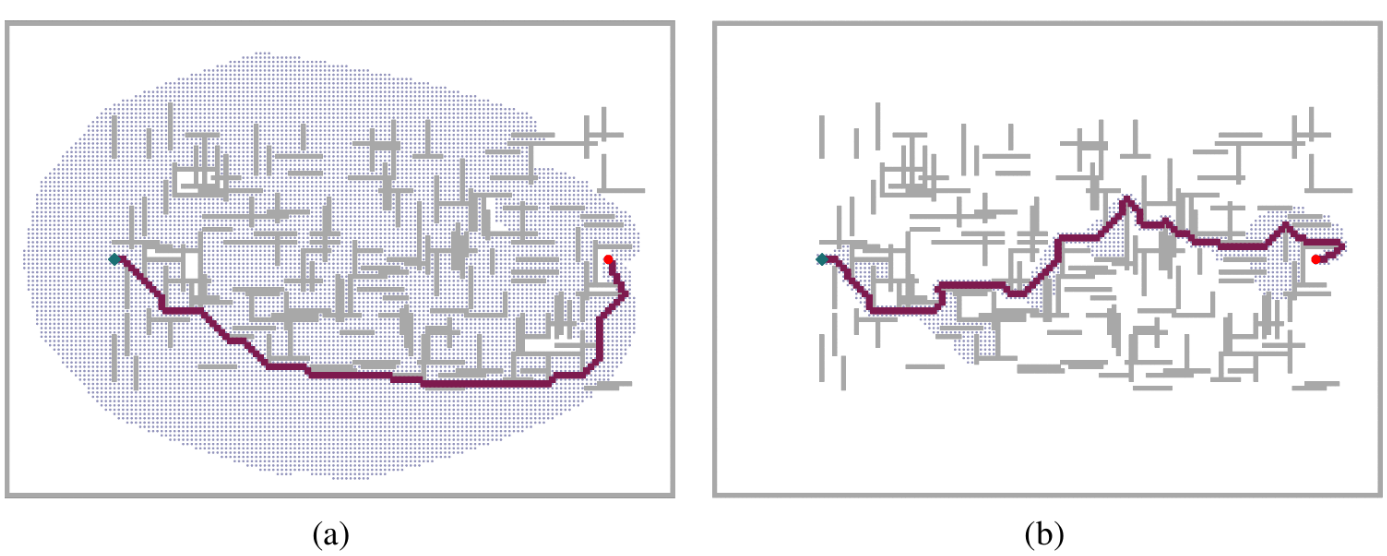


Figure 3.21 Two searches on the same grid: (a) an A* search and (b) a weighted A* search with weight $W = 2$. The gray bars are obstacles, the purple line is the path from the green start to red goal, and the small dots are states that were reached by each search. On this particular problem, weighted A* explores 7 times fewer states and finds a path that is 5% more costly.

جستجو با محدودیت حافظه (Memory Bounded Search)

یکی از مشکلات *A مصرف بالای حافظه‌ست. اینجا هم چند تا ترفند برای کم‌حافظه‌تر کردن پیاده‌سازی داریم، هم الگوریتم‌های جدیدی که از هر بایت حافظه‌ای به‌خوبی استفاده می‌کنن.

کپی وضعیت‌ها

معمولا وقتی می‌گیم یه وضعیت روی مرزه، هم تو صف مرز (برای بازشدن) ذخیره‌ش می‌کنیم و هم تو جدول «رسیده‌ها» (برای پرهیز از تکرار). بعضی وقتا اندازه‌ی صف خیلی کوچیکه و این کپی مشکلی ایجاد نمی‌کنه، ولی اگه پرحجم باشه، می‌تونیم تصمیم بگیریم که وضعیت رو فقط تو یکی از این دو جا نگه داریم. این کم‌سر و صدا نیست ولی مقداری حافظه نجات می‌ده.

پاک‌کردن وضعیت‌های بلااستفاده

اگه مطمئن باشیم بعد از یه مرحله هیچ راهی به یک وضعیت نیست—مثلاً با قانون منع U-turn یا با خاصیت «جداکنندگی» مرز—می‌تونیم اون وضعیت رو از جدول «رسیده‌ها» حذف کنیم. یا حتی می‌تونیم به ازای هر وضعیت «شمارنده ارجاع» نگه داریم و وقتی دیدیم دیگه هیچ مسیری تا اون وضعیت نمی‌رسه، پاکش کنیم.

بیم جست‌وجو (Beam Search)

اینجا به جای نگه‌داشتن همه‌ی نودهای مرز (frontier)، فقط k تا از بهترین‌ها (با کمترین f) رو نگه می‌داریم و بقیه رو دور می‌ریزیم. این‌طوری حافظه مصرفی و زمان اجرا کم می‌شه، ولی کامل نیست و بهترین مسیر رو هم پیدا نمی‌کنه—ولی معمولا نزدیک بهترین جوابه. می‌شه نسخه‌ای هم داشت که به جای k عدد ثابت، هر نودی که f-ش در حد معقول (مثلا حداکثر δ) از بهترین f هست رو نگه داره؛ اینطوری تو مواقعی که نودهای قوی کمیاب‌ن، تعداد بیشتری نگه می‌داره تا شانس پیدا کردن یه گزینه خوب بیشتر بشه.

IDA* (Iterative-Deepening A*)

همون ایده‌ی عمق‌افزایشی (Iterative Deepening) رو می‌آریم تو *A. این‌جا به جای عمق، «حد برش» رو روی $f=g+h$ می‌ذاریم. هر بار یک f-کانتور رو کامل می‌گردیم و کمترین fی که از اون کانتور بیشتر بوده رو برشِ بعدی می‌کنیم. اینطوری نیاز به جدول «رسیده» نداریم و حافظه به $O(d)$ کاهش پیدا می‌کنه (فقط عمق‌های قبلی رو تو استک نگه می‌داره) ولی بعضی نودها رو چند بار باز می‌کنه. برای مسأله‌هایی مثل ۸-پازل که fها عددیه، *IDA عالی عمل می‌کنه و نهایتا تا $C*$ مرحله پیش می‌ره.

RBFS (Recursive Best-First Search)

RBFS سعی می‌کنه مثل *A رفتار کنه ولی حافظه‌اش رو شبیه DFS محدود کنه. با بازگشتی جلو می‌ره، اما هر جا بفهمه هزینه‌ی مسیر فعلی از «بهترین جایگزین» تو یکی از اجدادش بیشتر شده، برمی‌گرده عقب و سراغ بهترین جایگزین می‌ره. ضمن بازگشت، مقدار f هر نودی که کنار می‌ذاره با «بهترین f بچه‌هاش» به‌روز می‌شه تا دفعه‌ی بعد بدونه دوباره اون شاخه رو امتحان کنه یا نه. RBFS نسبت به *IDA گاهی کاراترِ، ولی همچنان مشکل «بازبسط نودهای فراموش‌شده» رو داره—چون چندبار مسیرها رو از نو می‌سازه.

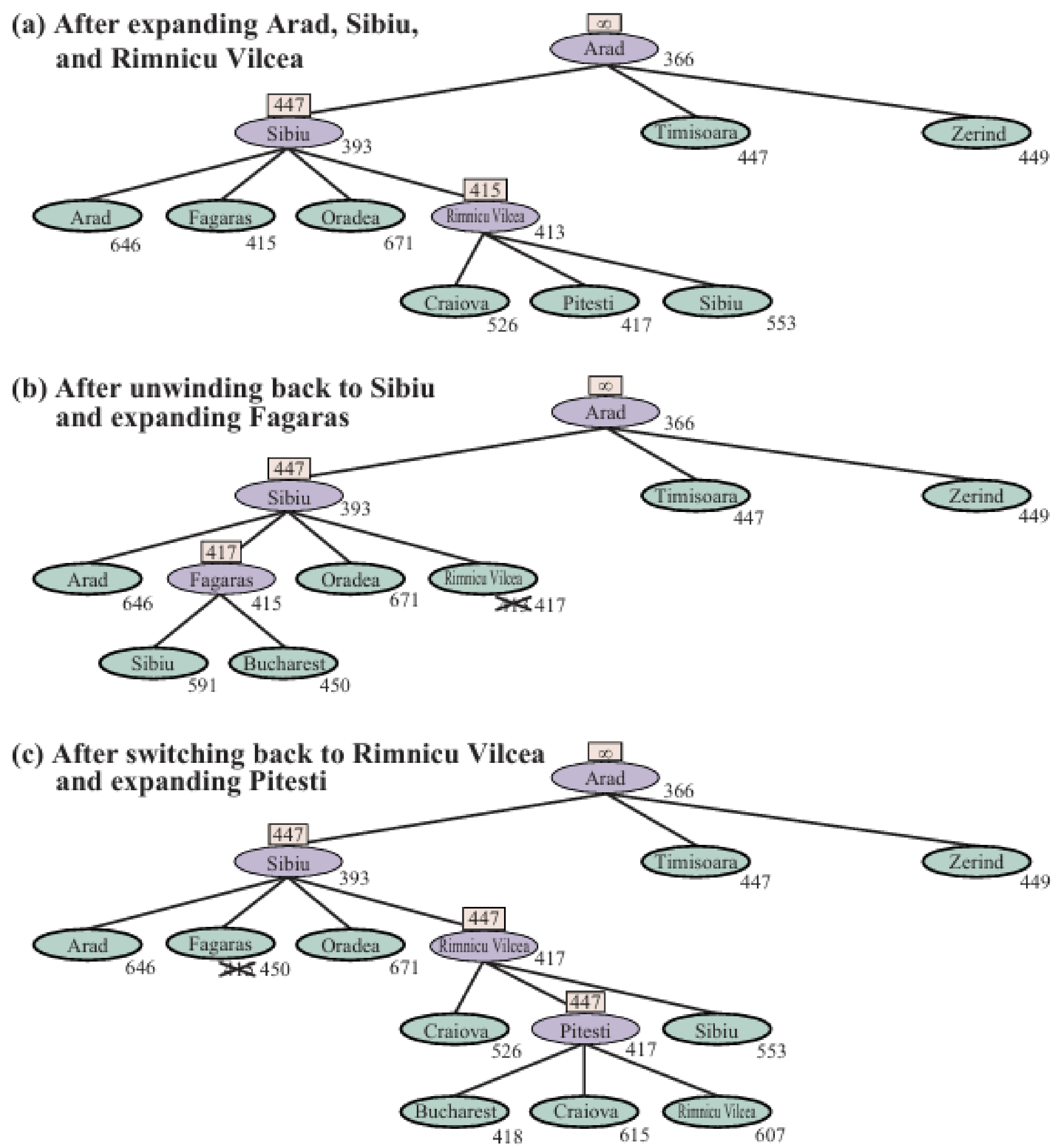


Figure 3.23 Stages in an RBFS search for the shortest route to Bucharest. The *f-limit* value for each recursive call is shown on top of each current node, and every node is labeled with its *f*-cost. (a) The path via Rimnicu Vilcea is followed until the current best leaf (Pitesti) has a value that is worse than the best alternative path (Fagaras). (b) The recursion unwinds and the best leaf value of the forgotten subtree (417) is backed up to Rimnicu Vilcea; then Fagaras is expanded, revealing a best leaf value of 450. (c) The recursion unwinds and the best leaf value of the forgotten subtree (450) is backed up to Fagaras; then Rimnicu Vilcea is expanded. This time, because the best alternative path (through Timisoara) costs at least 447, the expansion continues to Bucharest.

RBFS وقتی بهینه‌ست که هیوریستیک h(n) قابل قبول باشه. حافظه‌ای که RBFS لازم داره خطیه (O(bd)) و برابرِ عمق عمیق‌ترین جواب بهینه، ولی زمان اجراش خیلی بسته به دقت هیوریستیک و تعداد دفعاتیه که مسیر «بهترین» عوض می‌شه. RBFS نودها رو به ترتیب افزایش f باز می‌کنه، حتی اگه f تو بعضی مسیرها کم‌نظمی داشته باشه. هم IDA* و هم RBFS به این مشکل دچارند که حافظه‌شون خیلی کمیه و در میان‌مدت بیش از حد مسیرها رو دوباره زیر و رو می‌کنن.

بدیهیه که ایده‌مندانه باشه اول بفهمیم چقدر حافظه داریم و بعد روش رو بذاریم کلش رو استفاده کنه. دو تا الگوریتم برای این کار وجود داره: MA* (A* با محدودیت حافظه) و SMA* (نسخه‌ی ساده‌شده‌ش).
چون SMA* ساده‌تره، اینو توضیح می‌دم.

SMA* دقیقاً مثل A* کار می‌کنه: همیشه بهترین برگ (leaf) یعنی نودی که کمترین f رو داره باز می‌کنه تا جایی که حافظه پر بشه. وقتی حافظه لبریز شد و می‌خواد یه نود جدید اضافه کنه، باید یکی از نودهای قبلی رو حذف کنه. SMA* همیشه بدترین برگ (یعنی بالاترین f) رو می‌ریزه دور. بعد، مثل RBFS وقتی اون برگ فراموش‌شده رو حذف می‌کنه، مقدار f اون رو به پدرش (back up) می‌کنه؛ یعنی پدر مطلع می‌مونه که بهترین مسیر تو اون زیرشاخه چه ارزشی داشته. اینجوری وقتی همه‌ی گزینه‌های دیگه بدتر از اون زیرشاخه شدن، فقط همون زیرشاخه رو دوباره احیا می‌کنه. اگر همه‌ی نوادگانِ نودی پاک بشن، دیگه نمی‌دونیم از اون نود به کجا بریم، ولی می‌دونیم چقدر ارزش داره که یه مسیر ازش ادامه بدیم.

یه نکته‌ی ریز هم اینه که اگه همه‌ی برگ‌ها f یکسان داشته باشن، SMA* باید حواسش باشه وقتی برگ جدیدی رو باز می‌کنه، همون برگ رو پاک نکنه. پس اینجا «تازه‌ترین برگ با بهترین f» رو باز می‌کنه و «قدیمی‌ترین برگ با بدترین f» رو حذف می‌کنه. اگه فقط یه برگ باشه، یعنی درخت فعلی یه مسیر صاف ریشه تا برگه که کل حافظه رو پر کرده. اگه اون برگ هدف نباشه و حتی اگر بخشی از مسیر بهینه باشه، چون حافظه اجازه‌ی ادامه‌ی مسیر رو نمی‌ده، مثل اینه که اصلاً فرزندی نداره و می‌پره دورش.

ویژگی‌ها:
SMA* کامله اگه یه راه‌حل قابل‌دستیابی باشه (عمق کمترین هدف d کمتر از تعداد نودهایی‌ست که حافظه جا داره).
بهینه‌ست اگه هر مسیر بهینه‌ای، با حافظه‌ی موجود، قابل‌دسترس باشه؛ وگرنه بهترین راه‌حلی رو می‌ده که می‌تونسته پیدا کنه.
این الگوریتم برای مسائلی که حافظه‌ی محدودی دارن و فضای حالت پیچیده‌ست (مثلاً گراف، هزینه‌های متفاوت اعمال، ساخت نودها گرونه) خیلی مناسب و پُرکاربرده.