

بِسْمِ اللَّهِ الرَّحْمَنِ الرَّحِيمِ



هوش مصنوعی

فصل ۳

حل مسئله با جستجو (۱)

Solving Problems by Searching (1)

کاظم فولادی قلعه
دانشکده مهندسی، دانشکدگان فارابی
دانشگاه تهران

<http://courses.fouladi.ir/ai>

هوش مصنوعی

حل مسئله با جستجو

۱

عوامل‌های
حل مسئله

عامل حل مسئله به عنوان نوعی از عامل مبتنی بر هدف

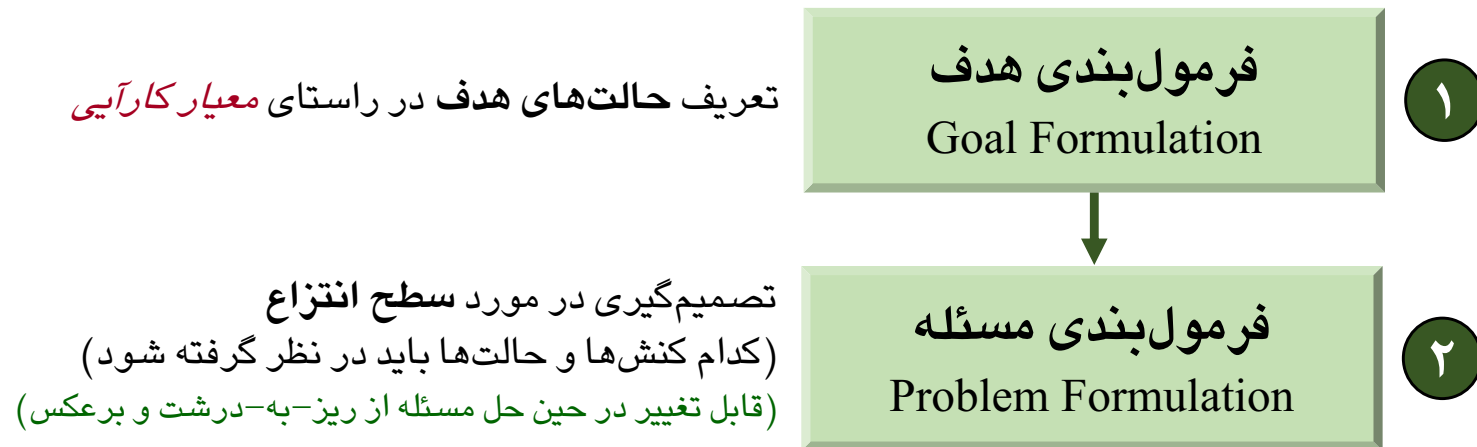


برنامه‌ی عامل حل مسئله، در فضای کنش‌های عامل‌ها به جستجو می‌پردازد تا متناسب‌ترین دنباله‌ی کنش‌ها که عامل را با کمترین هزینه به هدفش می‌رساند، پیدا کند.

الگوریتم‌های جستجو

الگوریتم‌های جستجوی همه‌منظوره	
آگاهانه <i>Informed</i>	ناآگاهانه <i>Uninformed</i>
عامل در مورد اینکه کجا به دنبال راه‌حل مسئله بگردد، کم و بیش راهنماهایی دارد.	عامل بجز تعریف مسئله، هیچ اطلاعاتی در مورد مسئله‌ی تحت بررسی خود ندارد.

فرمول‌بندی مسئله، تابعی از فرمول‌بندی هدف



فرآیند حل مسئله



عامل‌های حل مسئله

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action  
persistent: seq, an action sequence, initially empty  
             state, some description of the current world state  
             goal, a goal, initially null  
             problem, a problem formulation  
  
state ← UPDATE-STATE(state, percept)  
if seq is empty then  
    goal ← FORMULATE-GOAL(state)  
    problem ← FORMULATE-PROBLEM(state, goal)  
    seq ← SEARCH(problem)  
    if seq = failure then return a null action  
action ← FIRST(seq)  
seq ← REST(seq)  
return action
```

خصوصیات محیط وظیفه برای عامل حل مسئله

مشاهده‌پذیر کامل
Fully Observable

مشاهده‌پذیر جزئی
Partially Observable

تک عاملی
Single-agent

چند عاملی
Multiagent

قطعی
Deterministic

استراتژیک
Strategic

اتفاقی
Stochastic

مقطعی
Episodic

دنباله‌ای
Sequential

ایستا
Static

نیمه پویا
Semidynamic

پویا
Dynamic

گسسته
Discrete

پیوسته
Continuous

شناخته شده
Known

ناشناخته
Unknown

⇐ راه حل هر مسئله، دنباله‌ای ثابتی از کنش‌هاست.

راه حل بدون توجه به ادراک‌های ورودی در هر گام اجرا می‌شود (اجرای چشم بسته):

سیستم حلقه باز (open-loop)

مثال: مسئله‌ی رومانی

در یک روز تعطیل در کشور رومانی، اکنون در Arad هستیم و باید قبل از فردا به Bucharest برسیم.

بودن در Bucharest

فرمول‌بندی هدف
Goal Formulation

۱

حالت‌ها: شهرهای مختلف
کنش‌ها: رانندگی بین شهرها

فرمول‌بندی مسئله
Problem Formulation

۲

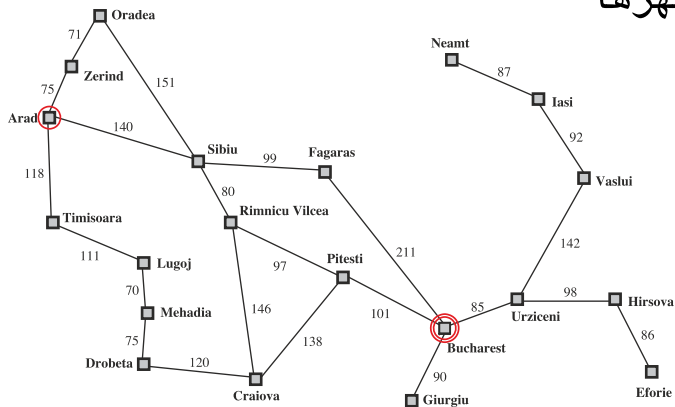
؟

جستجو
Search

۳

راه‌حل
Solution

۴

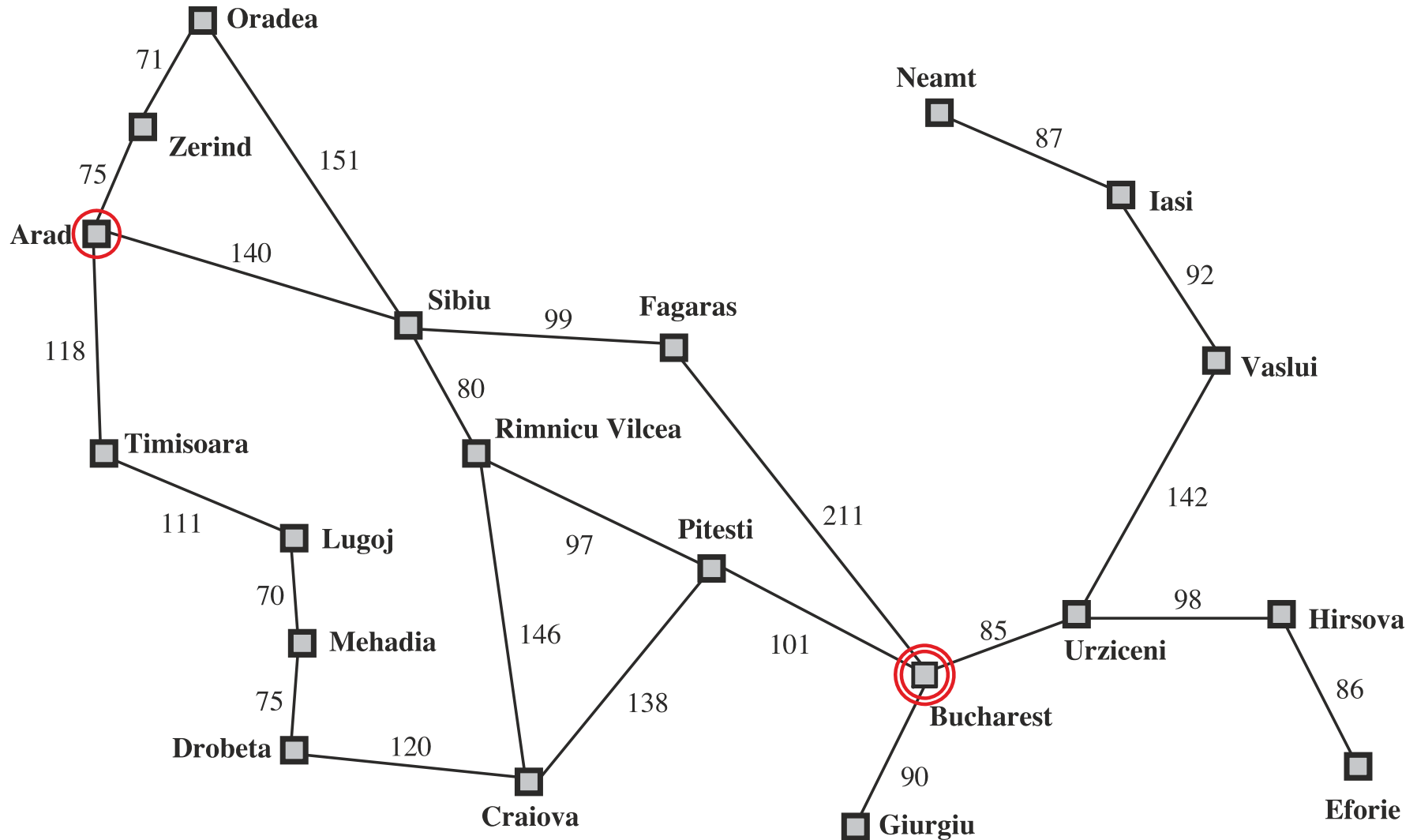


دنباله‌ای از شهرها

مثلاً: Arad → Sibiu → Fagaras → Bucharest

مثال: مسئله رومانی

نقشه



فرمول بندی مسئله

مؤلفه های پنج گانه ی تعریف مسئله

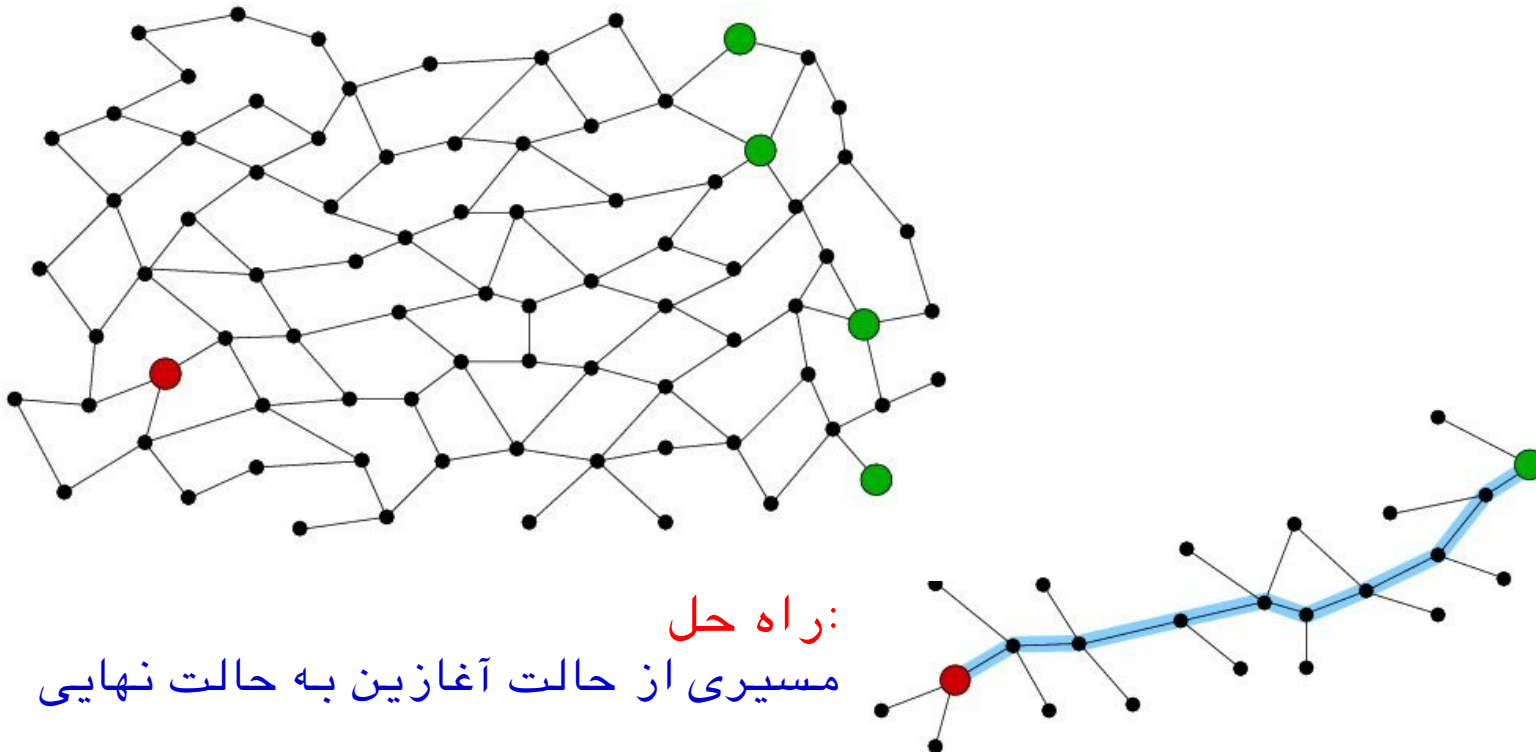
فضای حالت مسئله problem state-space	حالت آغازین Initial State	حالتی که عامل از آن شروع می کند.
	کنش های ممکن Available Actions	کنش های ممکن عامل در هر حالت s : $ACTIONS(s)$
	مدل گذار Transition Model	در هر حالت s ، هر کنش a چه می کند؟: $RESULT(s, a)$
	آزمون هدف Goal Test	تعیین کننده ی اینکه حالت جاری هدف است یا خیر: صریح (explicit): بیان مجموعه حالات هدف ضمنی (implicit): بیان ویژگی هدف
	تابع هزینه ی کنش Action Cost Function	هزینه ی گذار از یک حالت به حالت دیگر با یک کنش $c(s, a, s')$
تابع هزینه ی مسیر (path cost function): تابعی که به هر مسیر یک هزینه ی عددی نسبت می دهد: مجموع هزینه ی کنش های آن مسیر		
مجموعه ی حالت های دسترس پذیر از حالت آغازین با حداقل یک دنباله از کنش ها		

مؤلفه های پنج گانه ی تعریف مسئله

فضای حالت

STATE SPACE

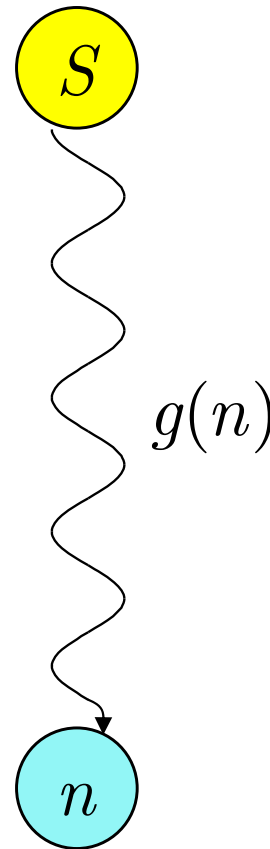
فضای حالت: مجموعه‌ی همه‌ی حالت‌های ممکن در مسئله و رابطه‌های آنها
 در قالب یک گراف نمایش داده می‌شود.
 مجموعه‌ی حالت‌های دسترس‌پذیر از حالت آغازین با حداقل یک دنباله از کنش‌ها

گراف فضای حالت:

تابع هزینه‌ی مسیر

PATH COST

هزینه‌ی مسیر هر گره: میزان هزینه‌ی پرداخت شده از گره‌ی آغازین تا آن گره



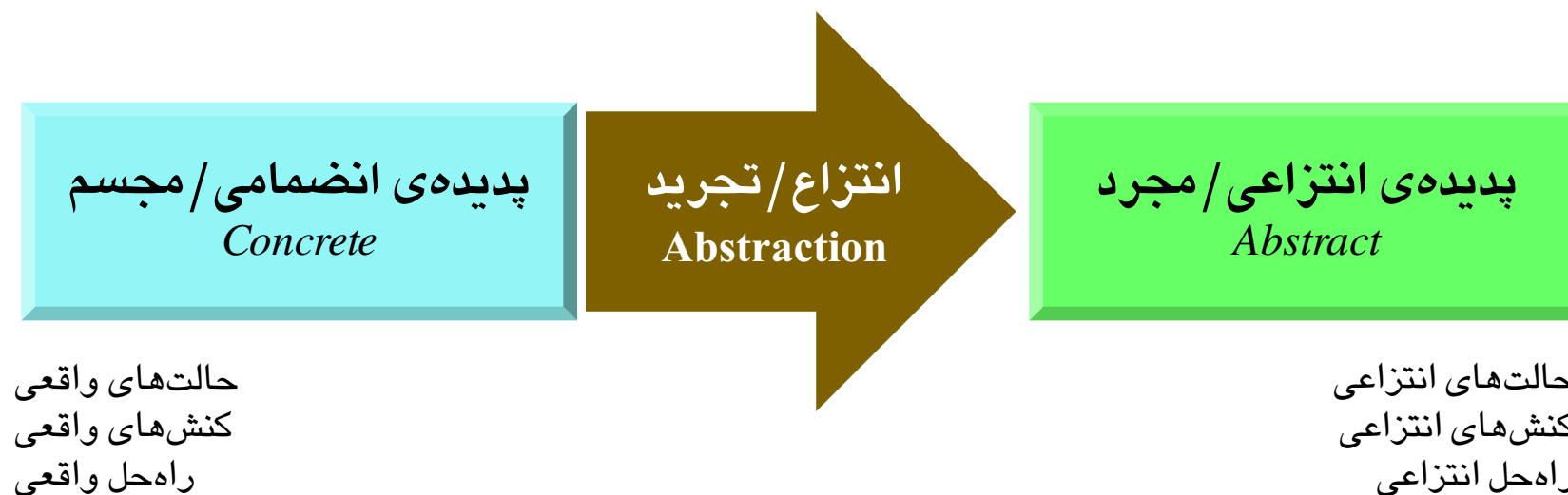
انتزاع

برای تعریف مسئله و تبیین فضای حالت مسئله

ABSTRACTION

انتزاع (تجريد): فرآیند حذف جزئیات از یک بازنمایی

فرمول‌بندی مسئله منجر به یک مدل می‌شود: توصیف ریاضی انتزاعی



* پدیده‌ی انتزاعی باید نسبت به پدیده‌ی انضمامی ساده‌تر باشد.

هنر حل مسئله: (تعیین سطح انتزاع)

اینکه تشخیص دهیم در تعریف حالت‌ها و کنش‌های مسئله،

چه چیزهایی باید در نظر گرفته شود و چه چیزهایی باید نادیده گرفته شود.

حل مسئله با جستجو

۲

مسئله‌های
نمونه

مسئله‌های نمونه

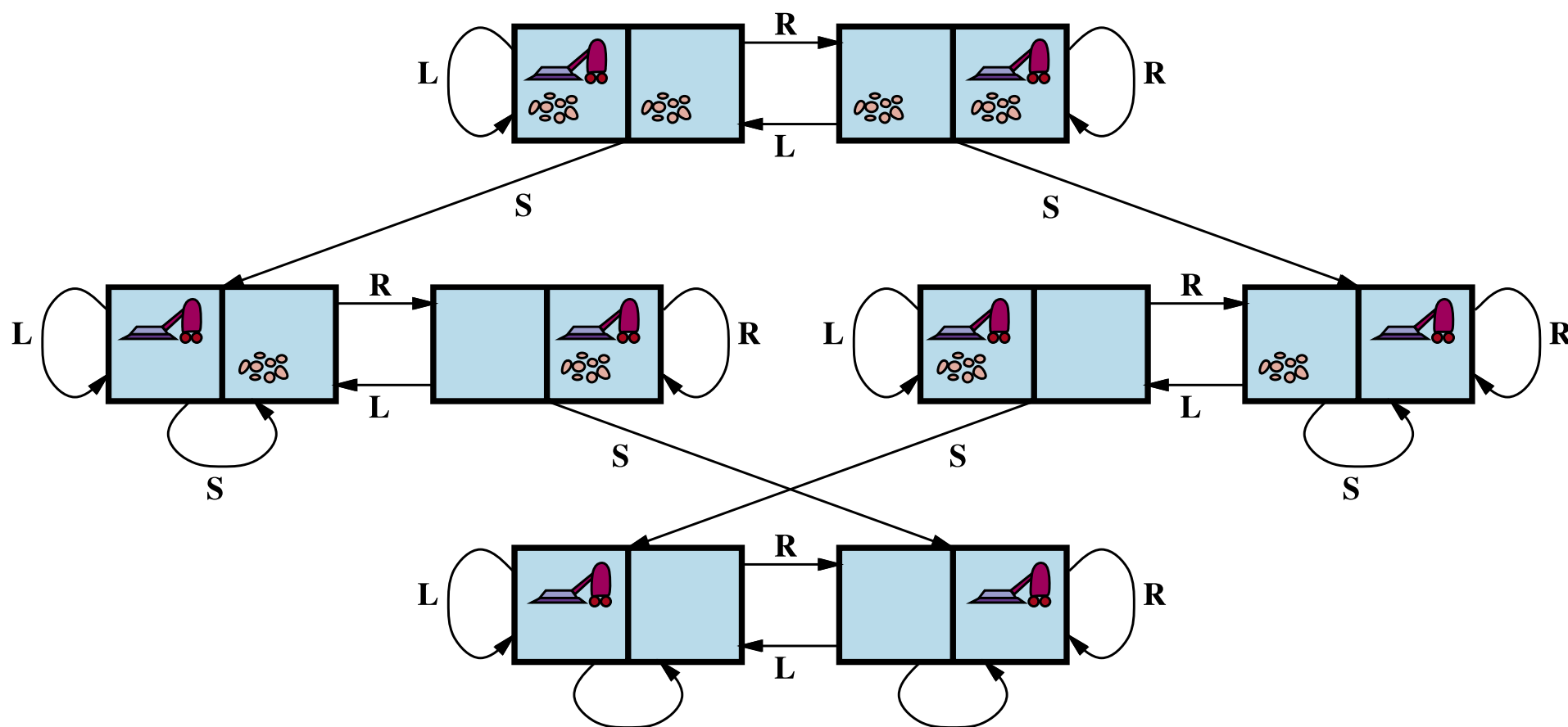
مسئله‌های نمونه	
مسئله‌های دنیای واقعی <i>Real-world Problems</i>	مسئله‌های استانداردسازی شده <i>Standardized Problems</i>
مسائل با تعریف مفصل‌تر و منعطف‌تر، حل آنها برای مردم اهمیت دارد.	مسائل با تعریف خلاصه و دقیق برای نمایش، تمرین و مقایسه‌ی کارآیی روش‌های حل مسئله

- Route-finding problem
- Touring Problems
e.g. traveling salesperson problem (TSP)
- VLSI Layout
Cell layout
Channel routing
- Robot navigation
- Automatic assembly sequencing
- Protein design
- Internet searching
- ...

- Vacuum world
- 8-puzzle (Sliding-block puzzles)
- Generating arbitrary integers
by starting from 4 using a sequence of operations,
including factorial, square root, and floor
- 8-queens problem
- Cryptarithmethic
- Missionaries and cannibals problem
(River crossing puzzles)
- ...

مثال: دنیای جاروبرقی

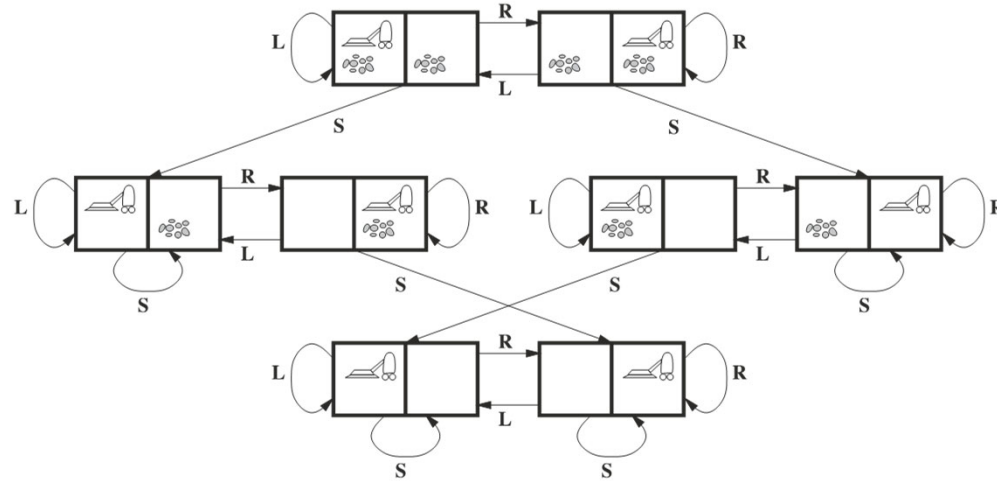
گراف فضای حالت



مثال: دنیای جاوربرقی

تعریف مسئله

VACCU WORLD



نمونه‌ای از مسائل

grid world problems

مسائل دنیای مشبک:

آرایه‌ای دوبعدی از سلول‌های مربعی که در آن عامل می‌تواند از سلولی به سلول دیگر حرکت کند.

مکان عامل و مکان‌های آلوده (نادیده گرفتن مقدار آلودگی)

حالت‌ها

States

هر یک از حالت‌ها ممکن است.

حالت آغازین

Initial State

Left, Right, Suck, NoOp

کنش‌های ممکن

Available Actions

مطابق گراف فضای حالت

مدل گذار

Transition Model

ضمنی (implicit): حالتی که در آن تمام خانه‌ها تمیز باشند.

آزمون هدف

Goal Test

هزینه‌ی گام (step cost): هر کنش ۱ واحد (صفر برای *NoOp*) / تابع هزینه مسیر جمعی

تابع هزینه‌ی کنش

Action Cost Function

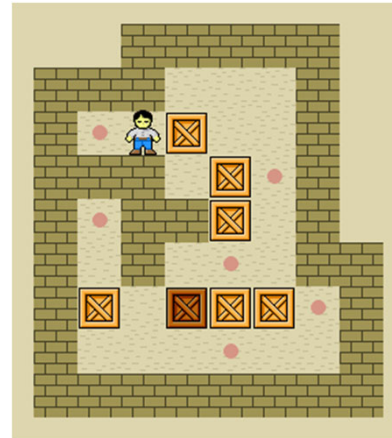
مؤلفه‌های پنج‌گانه‌ی تعریف مسئله

مثال: معمای سوکوبان

تعریف مسئله

SOKOBAN PUZZLE

- حداکثر یک جعبه در هر سلول می‌تواند قرار بگیرد.
- وقتی یک عامل به سمت یک سلول حاوی یک جعبه حرکت کند و یک سلول خالی در سمت دیگر جعبه وجود داشته باشد، آن‌گاه هم جعبه و هم عامل به سمت جلو حرکت می‌کنند.
- عامل نمی‌تواند یک جعبه را به داخل جعبه‌ی دیگر یا دیوار هل بدهد.



نمونه‌ای دیگر از مسائل

grid world problems

مسائل دنیای مشبک:

هدف عامل،

هل دادن تعدادی جعبه پراکنده در یک مشبک به سمت مکان‌های ذخیره‌سازی مشخص شده است.

مکان عامل و مکان‌های جعبه‌ها، موانع و مکان‌های ذخیره‌سازی در مشبک		حالت‌ها States
هر یک از حالت‌ها ممکن است.	n تعداد سلول‌های غیرمانع b تعداد جعبه‌ها $n \frac{n!}{b!(n-b)!} =$ تعداد حالت‌ها	حالت آغازین Initial State
$PushLeft, PushRight, PushUp, PushDown, NoOp$		کنش‌های ممکن Available Actions
مطابق گراف فضای حالت		مدل گذار Transition Model
ضمنی (implicit): حالتی که در آن جعبه‌ها در مکان‌های ذخیره‌سازی قرار بگیرند.		آزمون هدف Goal Test
هزینه‌ی گام (step cost): هر کنش ۱ واحد (صفر برای $NoOp$) / تابع هزینه مسیر جمعی		تابع هزینه‌ی کنش Action Cost Function

مؤلفه‌های پنج‌گانه‌ی تعریف مسئله

مثال: معمای ۸-

تعریف مسئله

8-PUZZLE

7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

Goal State

نمونه‌ای از مسائل

sliding-block puzzles

 n -puzzle

در حالت کلی NP-hard

موقعیت کاشی‌ها در قالب برداری از اعداد صحیح مثبت (نادیده گرفتن موقعیت‌های میانی)

حالت‌ها

States

هر یک از حالت‌ها ممکن است.

حالت آغازین

Initial State

حرکت دادن خانه‌ی خالی به چپ، راست، بالا، پایین

کنش‌های ممکن

Available Actions

مطابق تعریف

مدل گذار

Transition Model

صریح (explicit): حالت هدف (داده شده)

آزمون هدف

Goal Test

هزینه‌ی گام (step cost): هر حرکت ۱ واحد / تابع هزینه مسیر جمعی

تابع هزینه‌ی کنش

Action Cost Function

مؤلفه‌های پنج‌گانه‌ی تعریف مسئله

مثال: تولید عدد صحیح دلخواه از ۴ با دنباله‌ای از عملیات فاکتوریل، جذر و کف (مسئله‌ی کنوت)

تعریف مسئله

KNUTH PROBLEM

$$\lfloor \sqrt{\sqrt{\sqrt{\sqrt{\sqrt{(4!)!}}}}} \rfloor = 5$$

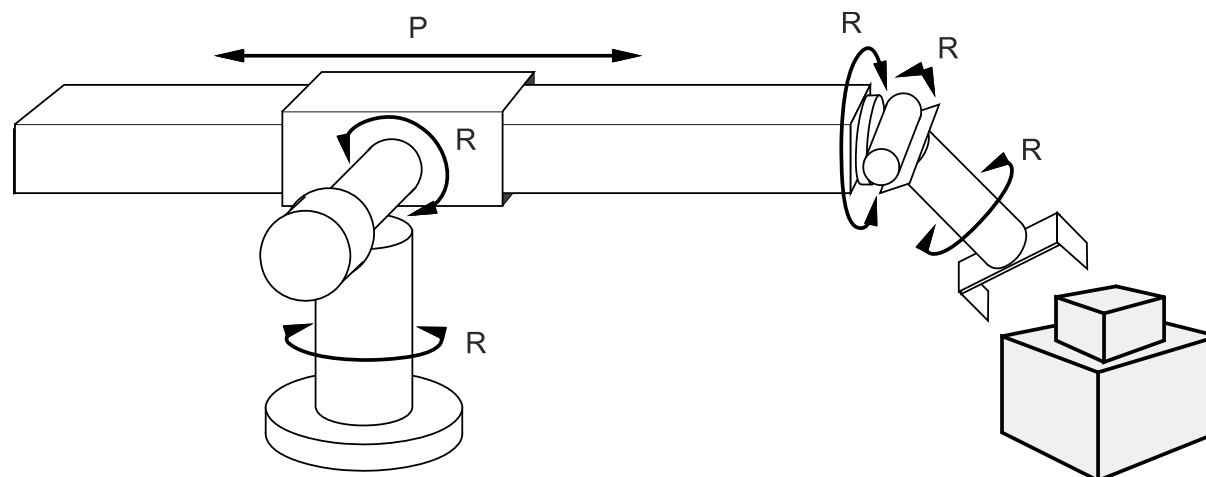
با شروع از عدد ۴، آیا دنباله‌ای از عملیات فاکتوریل، جذر و کف می‌توان به هر عدد صحیح دلخواهی رسید؟

Donald Knuth (1964)

اعداد مثبت	حالت‌ها States	مؤلفه‌های پنج‌گانه‌ی تعریف مسئله
۴	حالت آغازین Initial State	
اعمال فاکتوریل (فقط روی اعداد صحیح)، جذر، کف	کنش‌های ممکن Available Actions	
بر اساس تعریف ریاضی عملگرها	مدل گذار Transition Model	
صریح (explicit): عدد صحیح دلخواه	آزمون هدف Goal Test	
هزینه‌ی گام (step cost): هر عملیات ۱ واحد / تابع هزینه مسیر جمعی	تابع هزینه‌ی کنش Action Cost Function	

مثال: اسمبل کردن با ربات

تعریف مسئله



ROBOTIC ASSEMBLY

حالت‌ها States	مؤلفه‌های پنج‌گانه‌ی تعریف مسئله
مقدار حقیقی مختصات: زوایای مفصل‌های ربات، اجزای شیئی که باید اسمبل شود.	حالت آغازین Initial State
حالت اولیه‌ی ربات و اجزای شیئی	کنش‌های ممکن Available Actions
حرکت پیوسته‌ی مفصل‌های ربات	مدل گذار Transition Model
مطابق تعریف	آزمون هدف Goal Test
کل زمان اجرا	تابع هزینه‌ی کنش Action Cost Function
ضمنی (implicit): کامل شدن شیئی اسمبل شده و جدا شده از ربات	

مؤلفه‌های پنج‌گانه‌ی تعریف مسئله

حل مسئله با جستجو

۳

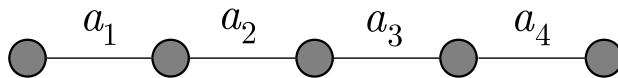
جستجو
به دنبال
راه حل

درخت جستجو

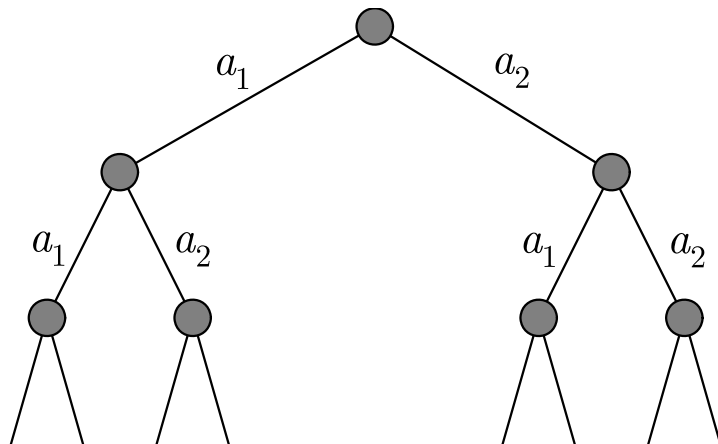
SEARCH TREE

راه حل
Solution

یک دنباله از کنش‌ها



الگوریتم‌های جستجو، دنباله‌های مختلف کنش‌های ممکن را بررسی می‌کنند:
دنباله‌های مختلف کنش‌های ممکن با شروع از حالت آغازین، یک درخت جستجو را شکل می‌دهند.



درخت جستجو
Search Tree

گره‌ها
Nodes

شاخه‌ها
Branches

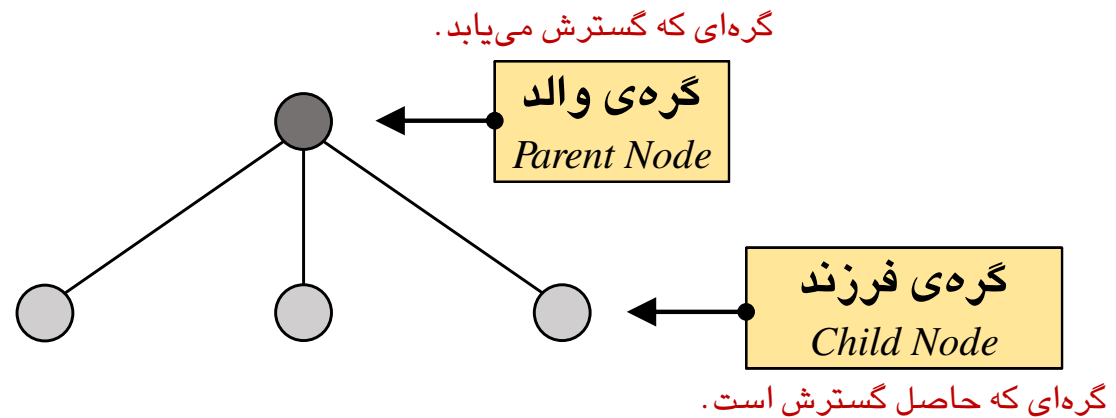
متناظر با
حالت‌ها (در فضای حالت مسئله)

متناظر با
کنش‌های عامل

گسترش حالت در درخت جستجو

SEARCH TREEگسترش
Expanding

اعمال همه‌ی کنش‌های مجاز به حالت جاری و تولید مجموعه حالت جدید



کناره

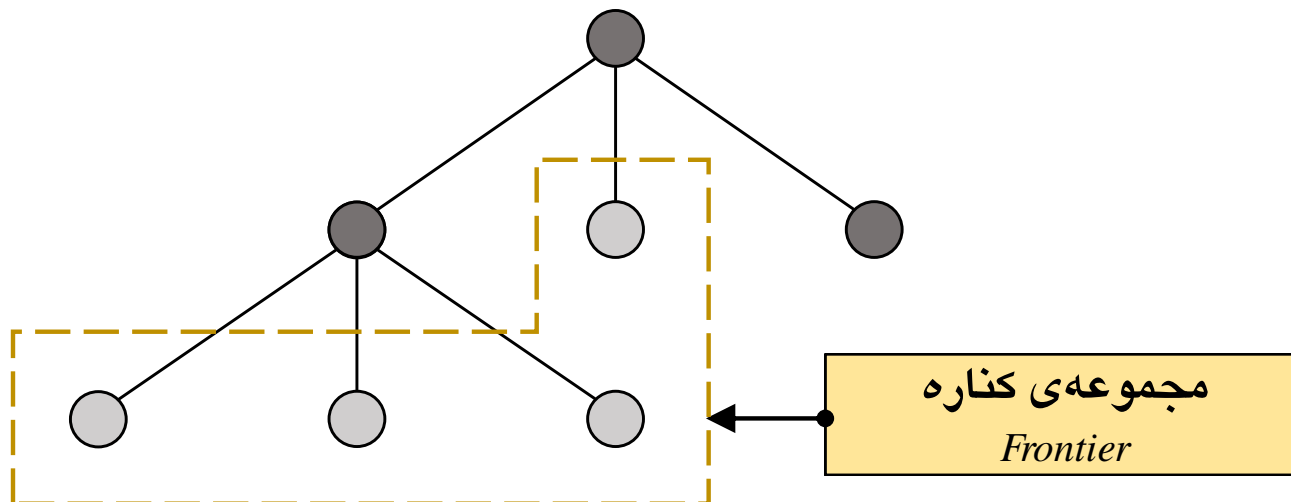
لیست باز

FRONTIER

مجموعه‌ی همه‌ی گره‌های برگ آماده برای گسترش در هر نقطه

لیست باز
*Open list*کناره
Frontier

گره‌ی بدون فرزند در درخت

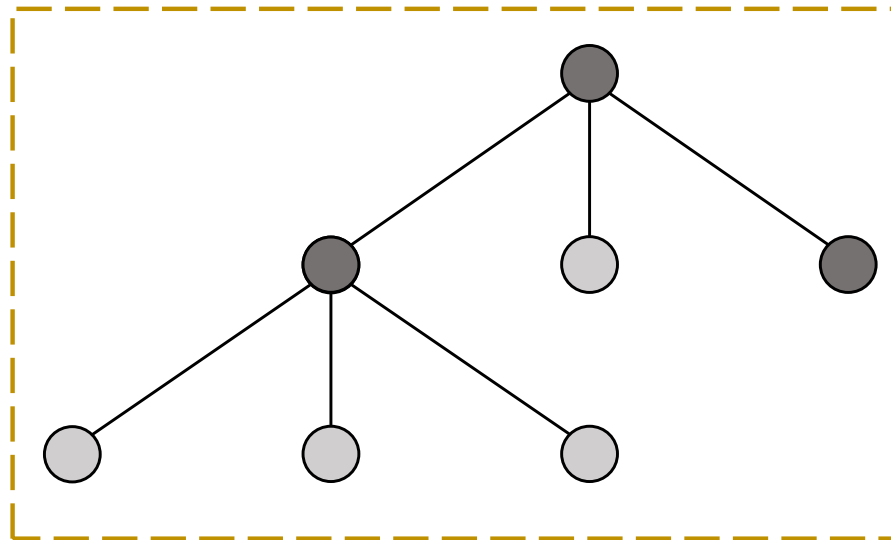
مجموعه‌ی کناره
Frontier

دسترسی شده

REACHED

مجموعه‌ی همه‌ی حالت‌هایی که گره‌ای برای آن تولید شده باشد و به آن رسیده باشیم.
(چه آن گره گسترش یافته باشد چه گسترش نیافته باشد.)

دسترسی شده
Reached



مجموعه‌ی دسترسی شده
Reached

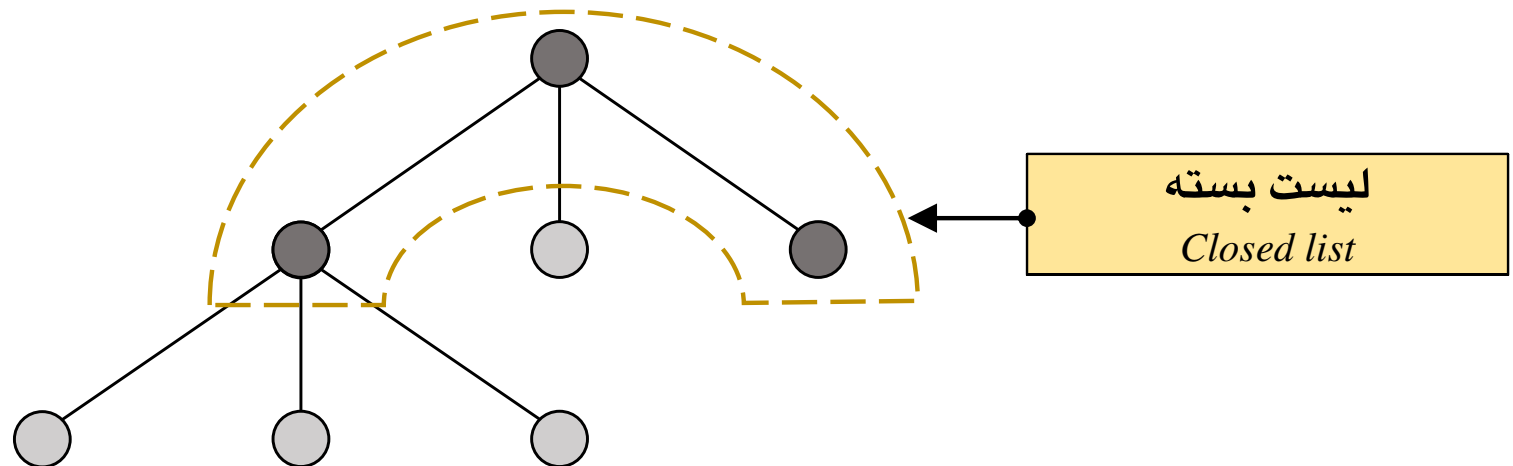
لیست بسته

CLOSED LIST

مجموعه‌ی همه‌ی گره‌هایی که تاکنون گسترش پیدا کرده‌اند.

لیست بسته
Closed list

$$\text{Closed list} = \text{Reached} - \text{Frontier}$$



استراتژی جستجو

استراتژی جستجو Search Strategy

چگونگی انتخاب گره‌ای که در مرحله‌ی بعدی باید گسترش پیدا کند.

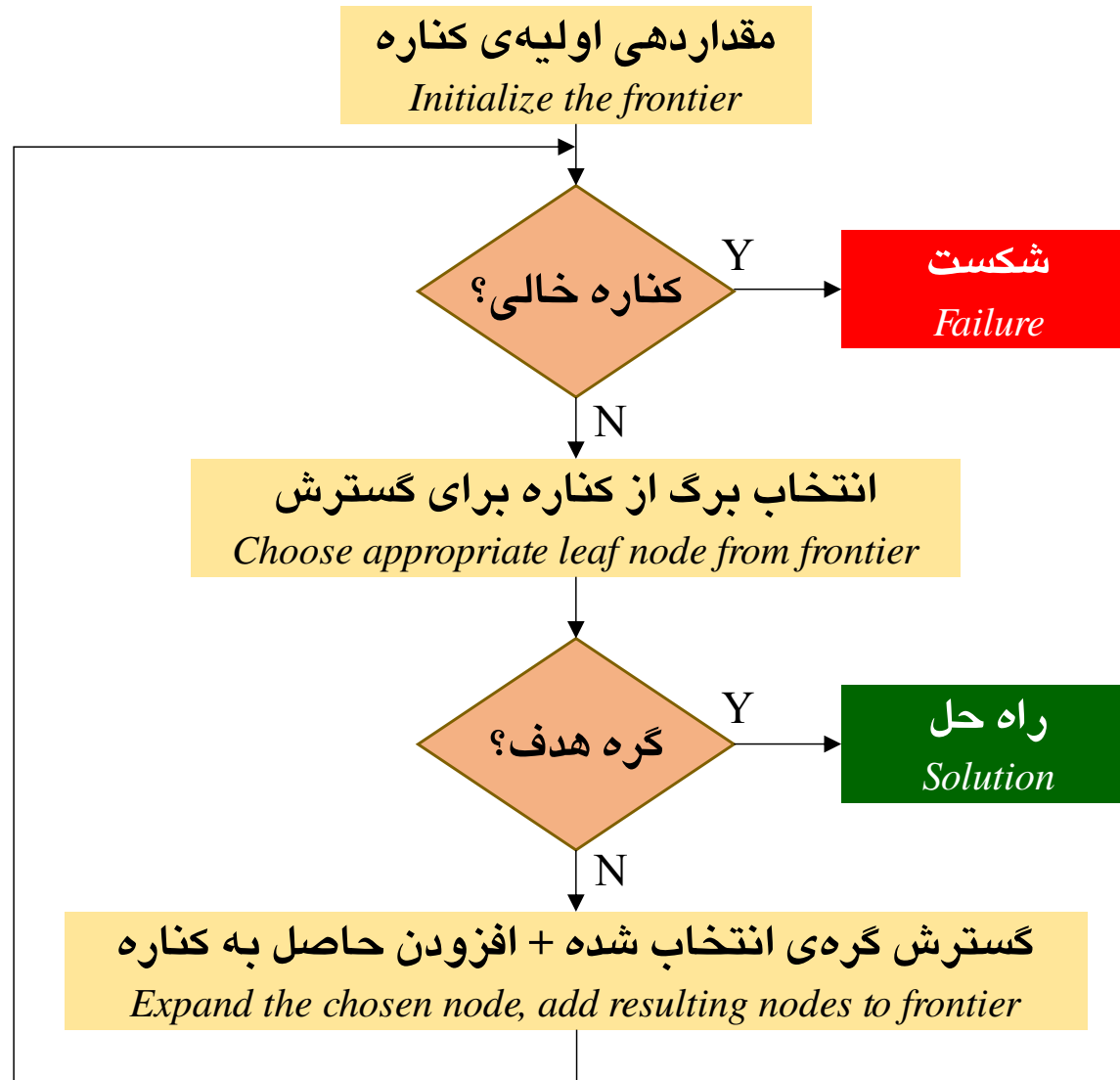
همه‌ی الگوریتم‌های جستجو در ساختار کلی مشترک هستند
تفاوت اصلی آنها در **استراتژی جستجو** است.

الگوریتم جستجوی درختی

```
function TREE-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier
```

الگوریتم جستجوی درختی

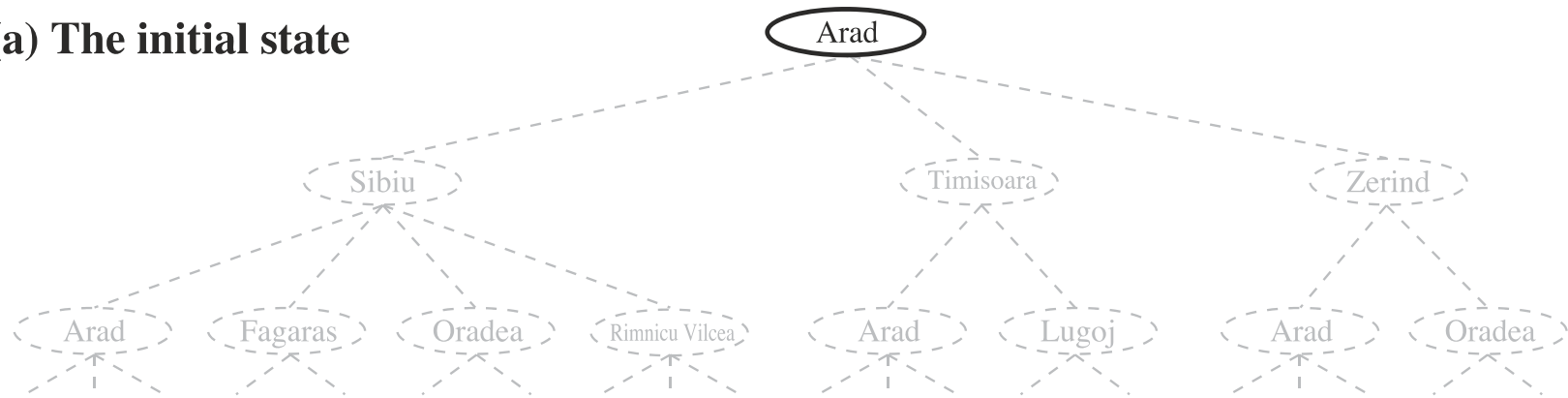
فلوچارت



الگوریتم جستجوی درختی

مثال: ۱ از ۳

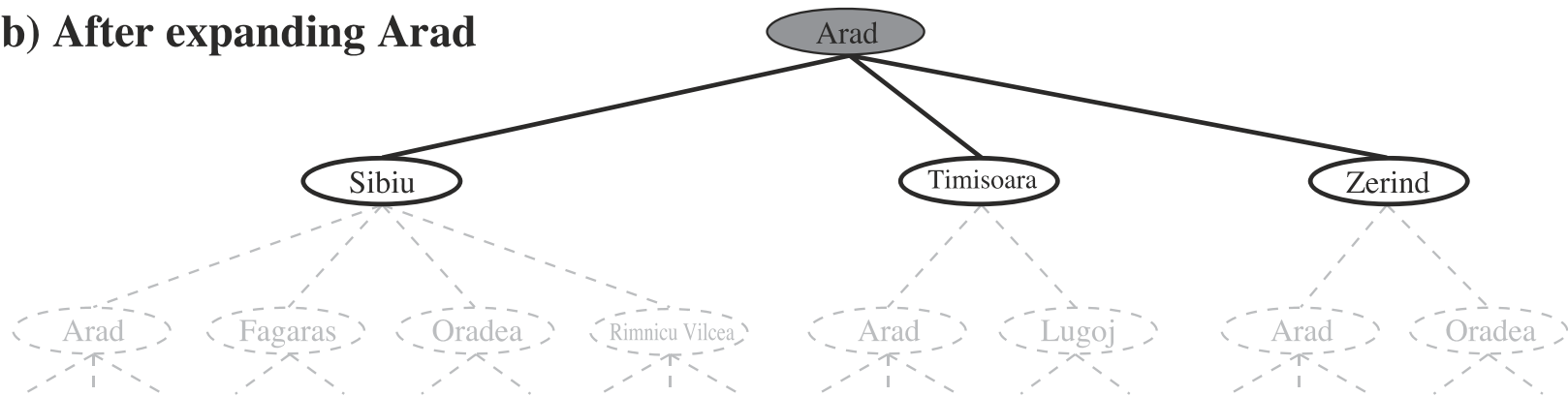
(a) The initial state



الگوریتم جستجوی درختی

مثال: ۲ از ۳

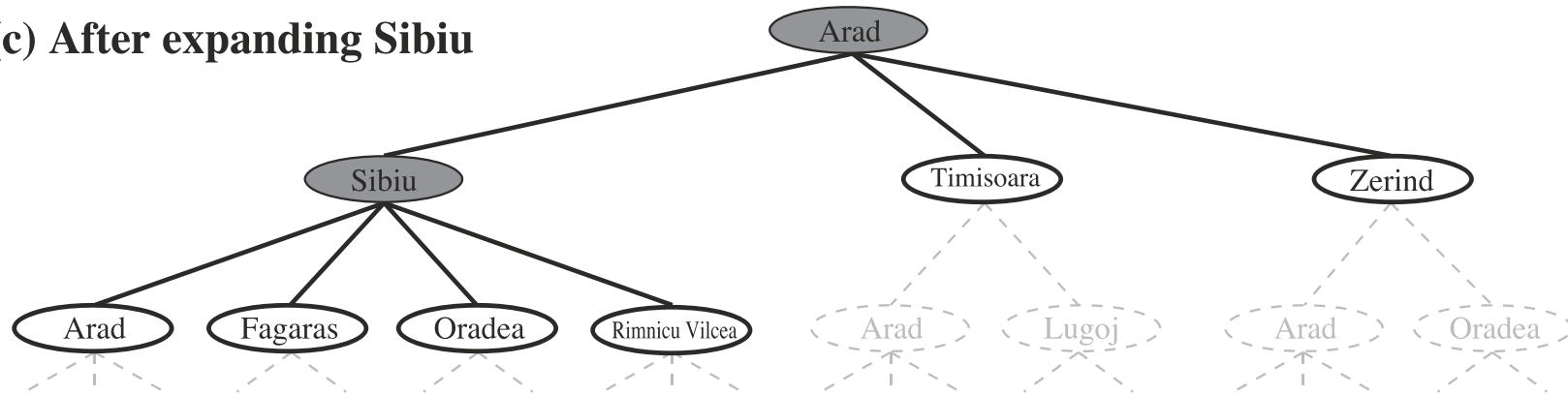
(b) After expanding Arad



الگوریتم جستجوی درختی

مثال: ۳ از ۳

(c) After expanding Sibiu



الگوریتم جستجوی بهترین-اول

BEST-FIRST SEARCH

```

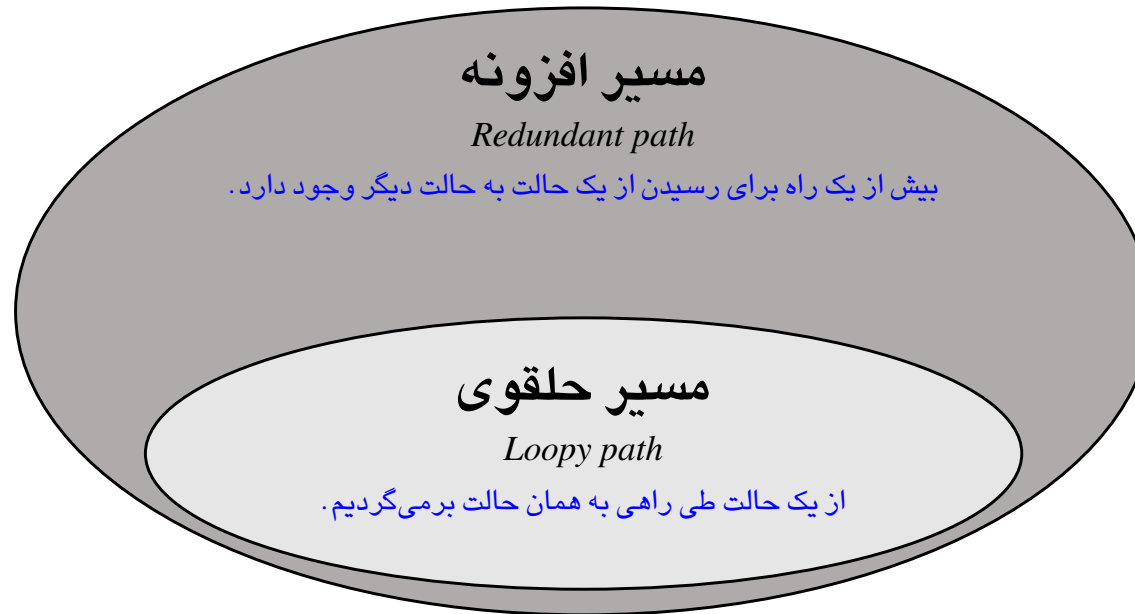
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node  $\leftarrow$  NODE(STATE=problem.INITIAL)
  frontier  $\leftarrow$  a priority queue ordered by f, with node as an element
  reached  $\leftarrow$  a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node  $\leftarrow$  POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s  $\leftarrow$  child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s]  $\leftarrow$  child
        add child to frontier
  return failure

function EXPAND(problem, node) yields nodes
  s  $\leftarrow$  node.STATE
  for each action in problem.ACTIONS(s) do
    s'  $\leftarrow$  problem.RESULT(s, action)
    cost  $\leftarrow$  node.PATH-COST + problem.ACTION-COST(s, action, s')
    yield NODE(STATE=s', PARENT=node, ACTION=action, PATH-COST=cost)

```

اداره کردن حالت‌های تکراری در درخت جستجو

REPEATED STATES



- در برخی موارد می‌توان خود مسئله را به گونه‌ای تعریف کرد که مسیرهای افزونه را حذف کند.
 (مثلاً در مسئله n -وزیر هر بار وزیر جدید در سمت چپ‌ترین ستون خالی اضافه شود).
- در برخی موارد دیگر، مسیرهای افزونه **اجتناب‌ناپذیر** هستند:
 شامل همه‌ی مسائلی که کنش‌های آنها وارون‌پذیر (reversible) است.
 (مثل مسئله‌ی مسیریابی و معمای بلوک‌های لغزان).

اجتناب از مسیرهای افزونه و حالت‌های تکراری

با جستجوی گرافی

الگوریتم‌هایی که تاریخچه‌ی خود را فراموش می‌کنند، محکوم به تکرار آن هستند.

Algorithms that forget their history are doomed to repeat it.

راه اجتناب از مسیرهای افزونه، به خاطر آوردن حالت‌های گسترش‌یافته‌ی قبلی است.



استفاده از الگوریتم جستجوی گرافی

(جستجوی درختی + مجموعه‌ی کشف‌شده‌ها برای به‌یادآوری همه‌ی گره‌های گسترش‌یافته)

گره‌های تازه تولید شده با گره‌های تولید شده از قبل مقایسه می‌شوند



در مجموعه‌ی *explored* یا *frontier*

در صورت مطابقت، به جای اضافه شدن به frontier نادیده گرفته می‌شوند.

مجموعه‌ی کشف‌شده

لیست بسته

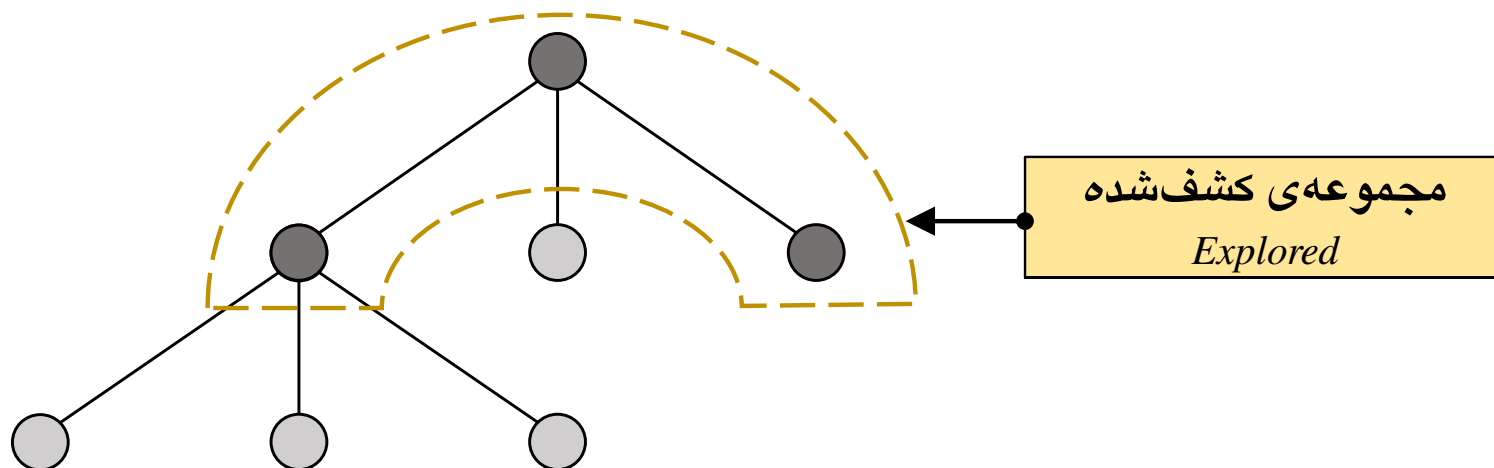
EXPLORED SET

مجموعه‌ی همه‌ی گره‌های بررسی شده و گسترش‌یافته

لیست بسته

Closed list

مجموعه‌ی کشف‌شده

Explored Set

الگوریتم جستجوی گرافی

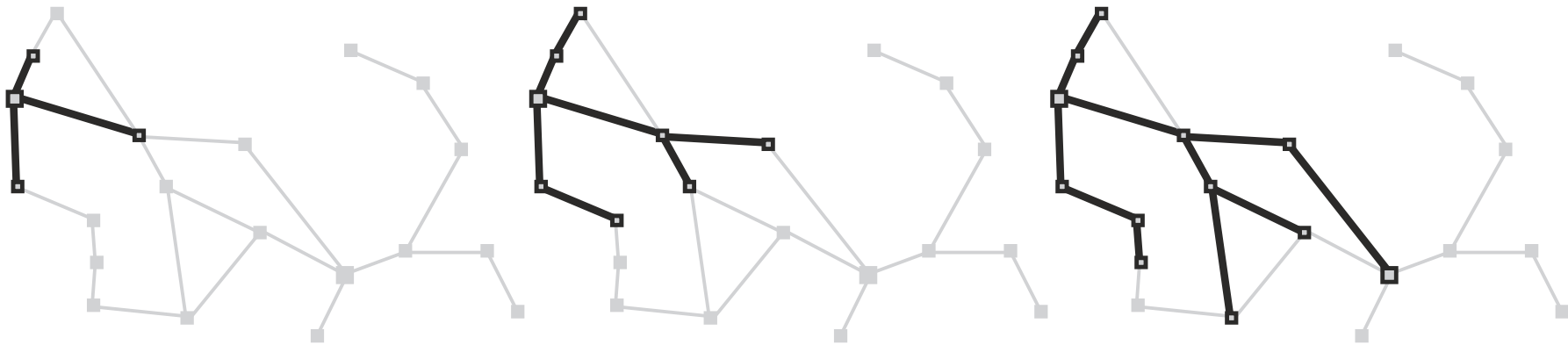
جستجوی گرافی: الگوریتمی که مسیرهای افزونه را بررسی می‌کند.

```
function GRAPH-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  initialize the explored set to be empty
  loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
    only if not in the frontier or explored set
```

* بخش‌های سیاه: لازم برای اداره کردن حالت‌های تکراری

جستجوی گرافی

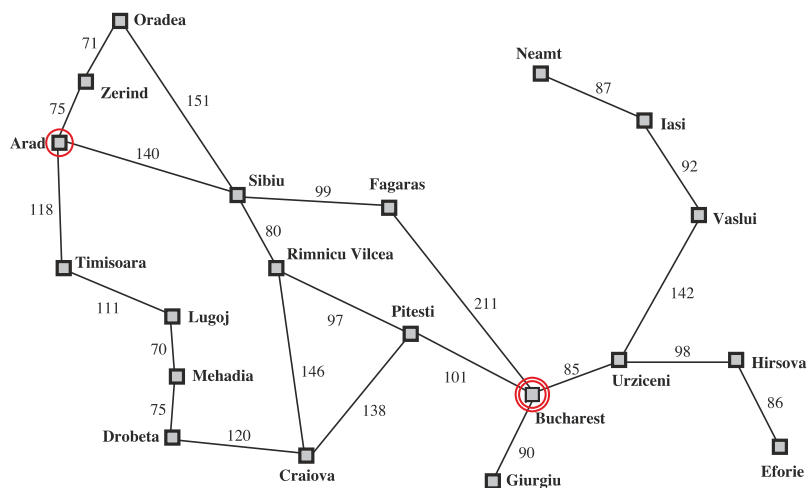
مثال



دنباله‌ی درخت‌های جستجو تولید شده توسط جستجوی گرافی برای مسئله‌ی رومانی

در هر مرحله، هر مسیر به اندازه‌ی یک گام گسترش می‌یابد.

* در مرحله‌ی سوم، شمالی‌ترین شهر (Oradea) بن‌بست است:
هر دوی مابعدهای آن هم‌اکنون توسط مسیرهای دیگر کشف شده‌اند.



الگوریتم جستجوی گرافی

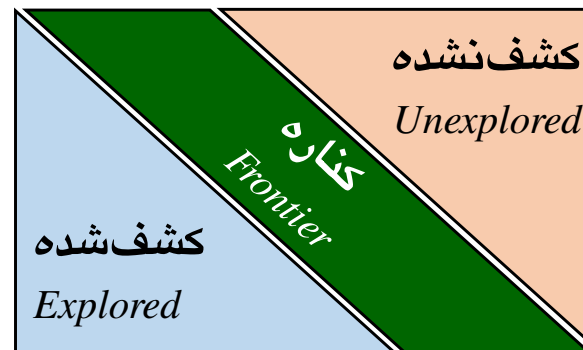
خصوصیات

درخت جستجوی ساخته شده توسط الگوریتم جستجوی گرافی حداکثر شامل یک کپی از هر حالت است.

می‌توان به آن به عنوان رشد یک درخت مستقیماً بر روی گراف فضای حالت فکر کرد.



کناره (frontier) گراف فضای حالت را به دو ناحیه‌ی کشف‌شده و کشف‌نشده تقسیم می‌کند.

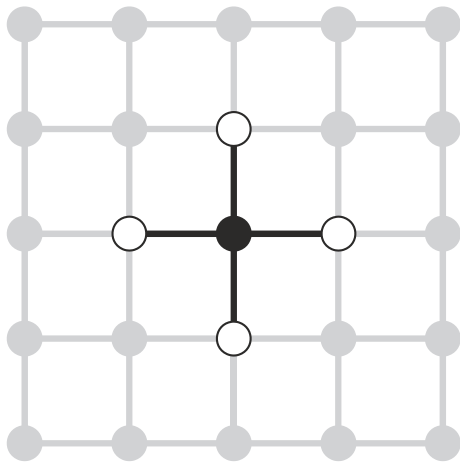


گراف فضای حالت

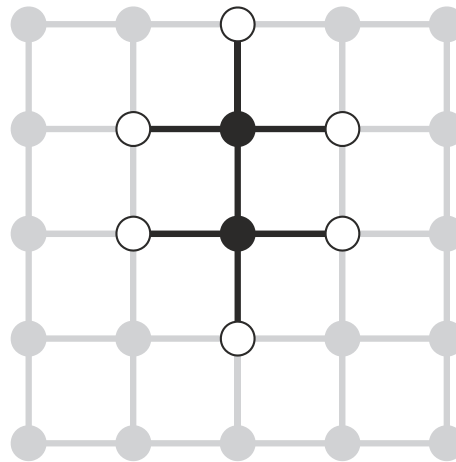
⇐ هر مسیر از حالت آغازین به یک حالت کشف‌نشده باید از یک حالت در کناره عبور کند.

جستجوی گرافی

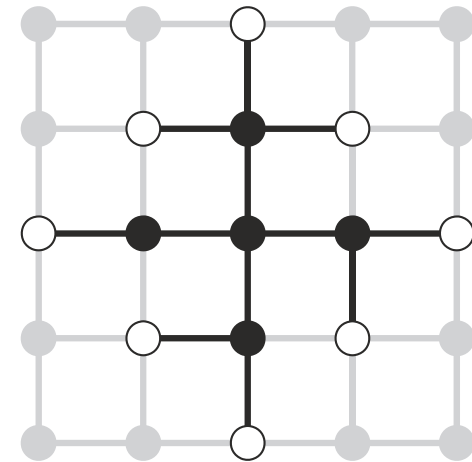
خاصیت جداکنندگی

GRAPH-SEARCH: SEPARATION PROPERTY

(a)



(b)



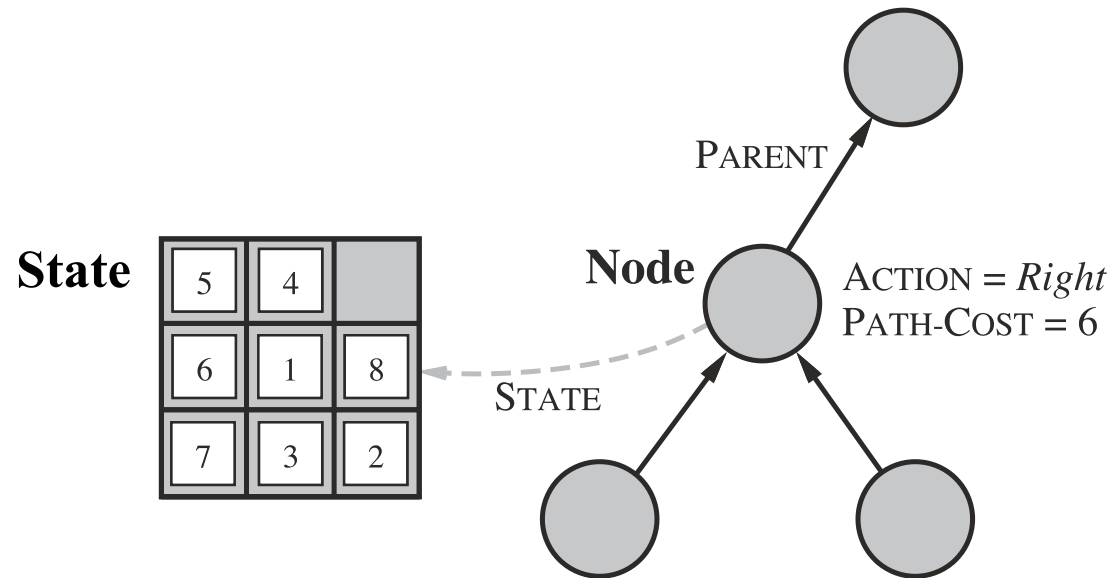
(c)

نمایش خاصیت جداکنندگی جستجوی گرافی ترسیم شده بر روی مسئله‌ی توری مستطیلی (rectangular grid)

کناره	frontier	○
کشف شده	explored	●
کشف نشده	unexplored	●

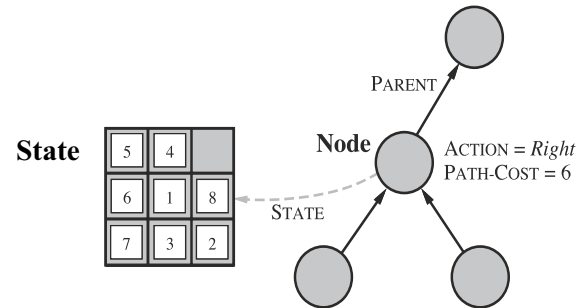
کناره، همیشه ناحیه‌ی کشف شده را از ناحیه‌ی کشف نشده‌ی فضای حالت جدا می‌کند.

ساختمان داده‌ی «گره» در درخت جستجو



n			
$n.STATE$	$n.PARENT$	$n.ACTION$	$n.PATH-COST$
حالت متناظر با گره از فضای حالت	گره‌ی مولد این گره در درخت جستجو	کنش اعمال شده به والد برای تولید این گره	هزینه‌ی مسیر از گره‌ی آغازین تا این گره $g(n)$

تمایز «حالت» و «گره»



گره <i>Node</i>	حالت <i>State</i>
در درخت جستجو	در فضای حالت مسئله
یک ساختمان داده: جزء تشکیل دهنده درخت جستجو	(بازنمایی) یک پیکربندی فیزیکی
دارای والد (parent)	دارای ماقبلها (predecessors)
دارای فرزندان (children)	دارای مابعدها (successors)
دارای عمق (depth)	—
دارای هزینه‌ی مسیر	—

تولید گرهی فرزند

function CHILD-NODE(*problem, parent, action*) **returns** a node

return a node with

STATE = *problem.RESULT(parent.STATE, action)*,

PARENT = *parent*, ACTION = *action*,

PATH-COST = *parent.PATH-COST + problem.STEP-COST(parent.STATE, action)*

پیاده‌سازی مجموعه‌ی «کناره»

مجموعه‌ی کناره در قالب یک صف (*queue*) پیاده‌سازی می‌شود.

<i>queue</i>			
EMPTY?(<i>queue</i>)	POP(<i>queue</i>)	TOP(<i>queue</i>)	ADD(<i>element</i> , <i>queue</i>)
True برمی‌گرداند فقط زمانی که صف خالی باشد.	اولین عنصر بالای صف را حذف می‌کند و آن را برمی‌گرداند.	اولین عنصر بالای صف را برمی‌گرداند بدون آنکه آن را حذف کند.	یک عنصر را در صف درج می‌کند و صف جدید را برمی‌گرداند.

نوع صف با ترتیب (order) درج گره‌ها در صف مشخص می‌شود.

FIFO queue	first-in, first-out	قدیمی‌ترین عنصر صف را پاپ می‌کند.
LIFO queue (Stack)	last-in, first-out	جدیدترین عنصر صف را پاپ می‌کند.
Priority queue		با اولویت‌ترین عنصر صف را پاپ می‌کند (بر اساس یک تابع ترتیب‌دهی)

پیاده‌سازی مجموعه‌ی «کشف‌شده»

مجموعه‌ی کشف‌شده در قالب یک جدول درهم‌سازی (*hash table*) پیاده‌سازی می‌شود.
(با هدف بررسی کارآمد حالت‌های تکراری)

زمان درج و مراجعه (*insert and lookup*) در جدول درهم‌سازی تقریباً ثابت است
(مستقل از تعداد حالت‌های ذخیره شده در آن).

* در پیاده‌سازی جدول درهم‌سازی توجه به مفهوم درست تساوی حالت‌ها بسیار مهم است (فرم‌های قانونمند).

معیارهای ارزیابی کارایی الگوریتم/ استراتژی جستجو

الگوریتم‌ها/ استراتژی‌های جستجو در امتداد ابعاد زیر ارزیابی می‌شوند:			
۱	۲	۳	۴
تمامیت <i>Completeness</i>	بهینگی هزینه <i>Cost Optimality</i>	پیچیدگی زمانی <i>Time Complexity</i>	پیچیدگی فضایی <i>Space Complexity</i>

آیا تضمینی وجود دارد
این الگوریتم راه‌حلی را
در صورت وجود بیابد؟
(الگوریتم باید سیستماتیک
باشد: شیوهی کاوش فضای
حالت نامتناهی به گونه‌ای باشد
که سرانجام بتواند به هر حالت
متصل به حالت آغازین برسد.)

آیا این استراتژی
لزوماً راه‌حل بهینه را
پیدا می‌کند؟

چه قدر زمان برای پیدا
کردن راه‌حل لازم است؟
(معمولاً بر حسب تعداد گره‌های
تولیدشده/ گسترش‌یافته)

چه قدر حافظه برای انجام
جستجو لازم است؟
(معمولاً بر حسب حداکثر تعداد
گره‌های موجود در حافظه)

حداکثر تعداد انشعاب یک گره در درخت جستجو	<i>Branching factor</i>	فاکتور انشعاب حداکثر	b	اندازه‌گیری پیچیدگی‌های زمانی و فضایی با پارامترهای
عمق کم عمق‌ترین هدف	<i>depth of l.c. solution</i>	عمق راه‌حل با کم‌ترین هزینه	d	
حداکثر طول یک مسیر در گراف فضای حالت (شاید ∞)	<i>Max. depth of s.s.</i>	حداکثر عمق فضای حالت	m	

بِسْمِ اللَّهِ الرَّحْمَنِ الرَّحِيمِ



هوش مصنوعی

فصل ۳

حل مسئله با جستجو (۲)

Solving Problems by Searching (2)

کاظم فولادی قلعه
دانشکده مهندسی، دانشکدگان فارابی
دانشگاه تهران

<http://courses.fouladi.ir/ai>

۴

استراتژی‌های
جستجوی
ناآگاهانه
(کورکورانه)

استراتژی‌های جستجوی ناآگاهانه

الگوریتم‌های جستجوی همه‌منظوره	
آگاهانه <i>Informed</i>	ناآگاهانه <i>Uninformed</i>
عامل در مورد اینکه کجا به دنبال راه حل مسئله بگردد، کم و بیش راهنماهایی دارد.	عامل بجز تعریف مسئله، هیچ اطلاعاتی در مورد مسئله‌ی تحت بررسی خود ندارد.

- جستجوی عرض-اول (Breadth-first: BFS)
- جستجوی هزینه-یکنواخت (Uniform-Cost: UCS)
- جستجوی عمق-اول (Depth-first: DFS)
- جستجوی عمقی محدودشده (Depth-limited: DLS)
- جستجوی عمیق‌کننده‌ی تکراری (Iterative Deepening: IDS)
- جستجوی دوطرفه (Bidirectional)

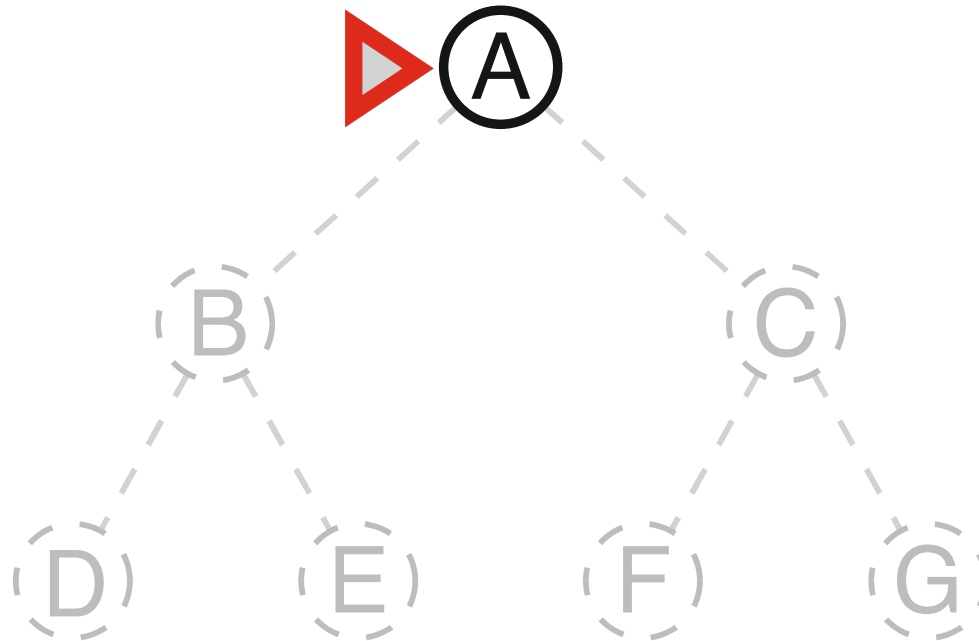
جستجوی عرض-اول

BREADTH-FIRST SEARCH

کم عمق ترین گره در کناره را گسترش دهید.
Expand shallowest node in the frontier.

قاعده
BFS

پیاده سازی: کناره، یک صف FIFO است (فرزندان جدید به انتهای صف اضافه می شوند)



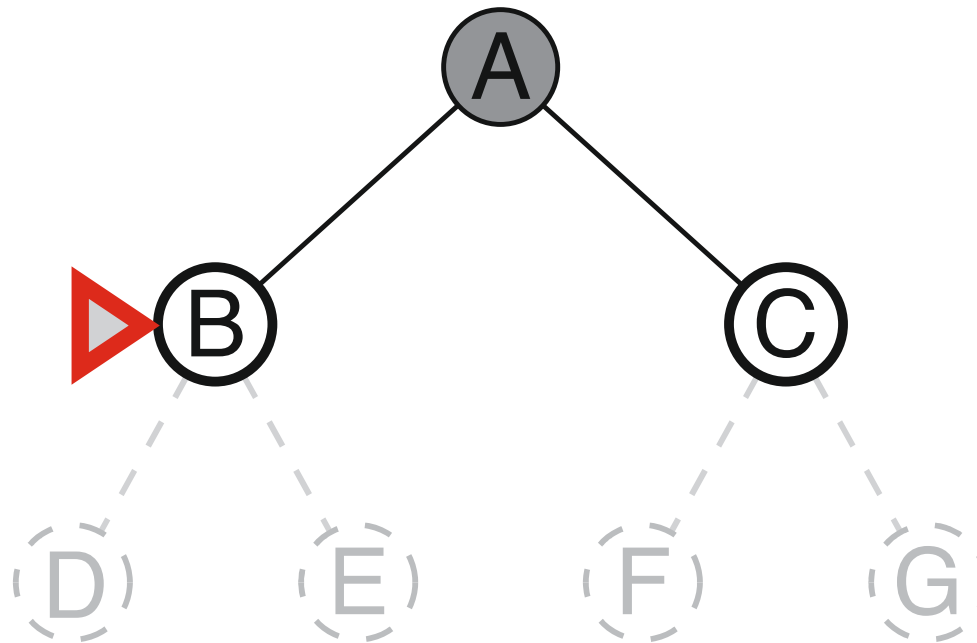
جستجوی عرض-اول

BREADTH-FIRST SEARCH

کم عمق ترین گره در کناره را گسترش دهید.
Expand shallowest node in the frontier.

قاعده
BFS

پیاده سازی: کناره، یک صف FIFO است (فرزندان جدید به انتهای صف اضافه می شوند)



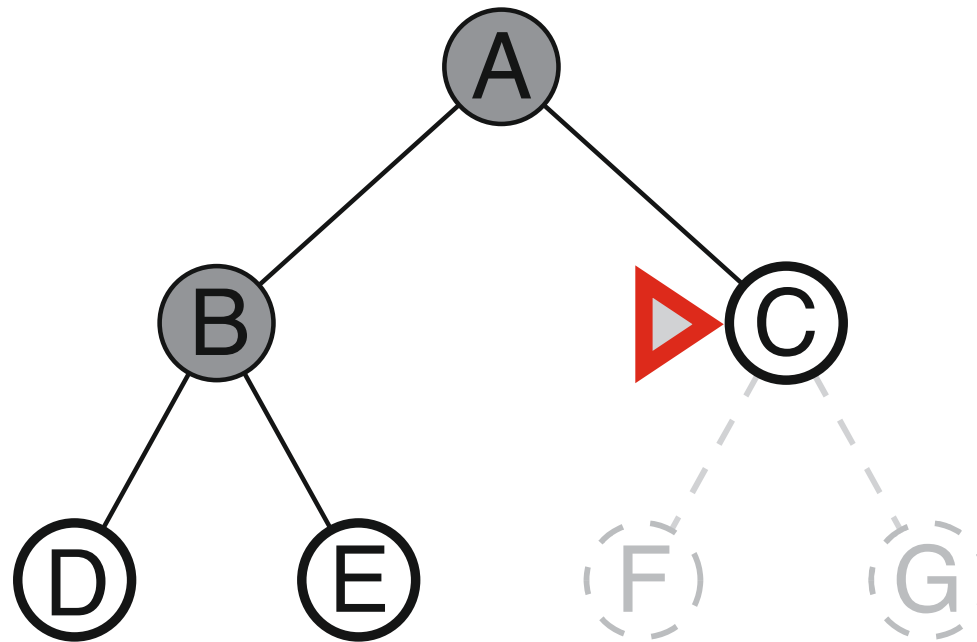
جستجوی عرض-اول

BREADTH-FIRST SEARCH

کم عمق ترین گره در کناره را گسترش دهید.
Expand shallowest node in the frontier.

قاعده
BFS

پیاده سازی: کناره، یک صف FIFO است (فرزندان جدید به انتهای صف اضافه می شوند)



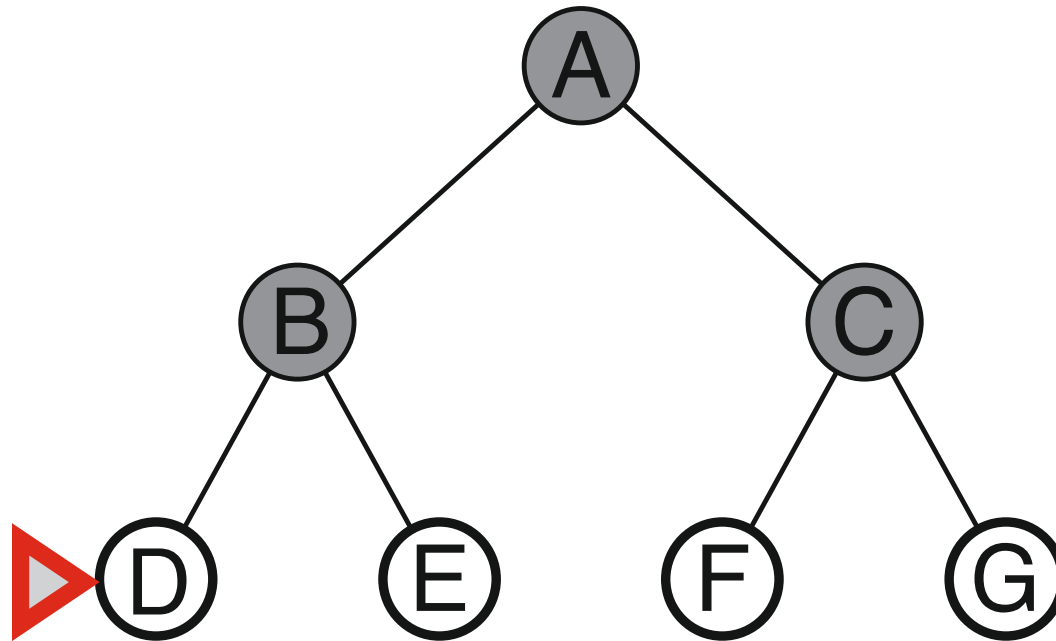
جستجوی عرض-اول

BREADTH-FIRST SEARCH

کم عمق ترین گره در کناره را گسترش دهید.
Expand shallowest node in the frontier.

قاعده
BFS

پیاده سازی: کناره، یک صف FIFO است (فرزندان جدید به انتهای صف اضافه می شوند)



جستجوی عرض-اول

شبه کد پیاده‌سازی با الگوریتم جستجوی بهترین-اول

BEST-FIRST SEARCH

```

function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node  $\leftarrow$  NODE(STATE=problem.INITIAL)
  frontier  $\leftarrow$  a priority queue ordered by f, with node as an element
  reached  $\leftarrow$  a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node  $\leftarrow$  POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s  $\leftarrow$  child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s]  $\leftarrow$  child
        add child to frontier
  return failure

```

$f(n)$ = عمق گره n = تعداد کنش‌های انجام گرفته برای رسیدن به گره n

```

function EXPAND(problem, node) yields nodes
  s  $\leftarrow$  node.STATE
  for each action in problem.ACTIONS(s) do
    s'  $\leftarrow$  problem.RESULT(s, action)
    cost  $\leftarrow$  node.PATH-COST + problem.ACTION-COST(s, action, s')
    yield NODE(STATE=s', PARENT=node, ACTION=action, PATH-COST=cost)

```

جستجوی عرض-اول

شبیه‌کد کارآمدتر

```

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

```

- ویژگی‌ها:
- استفاده از صف FIFO به جای صف اولویت
- مجموعه‌ی *reached* به صورت مجموعه‌ای از حالت‌ها به جای نگاشتی از حالت‌ها به گره‌ها (آزمون هدف زودهنگام)

جستجوی عرض-اول

آزمون هدف زودهنگام / آزمون هدف دیرهنگام

BREADTH-FIRST SEARCH

آزمون هدف

آزمون هدف دیرهنگام

Late Goal Test

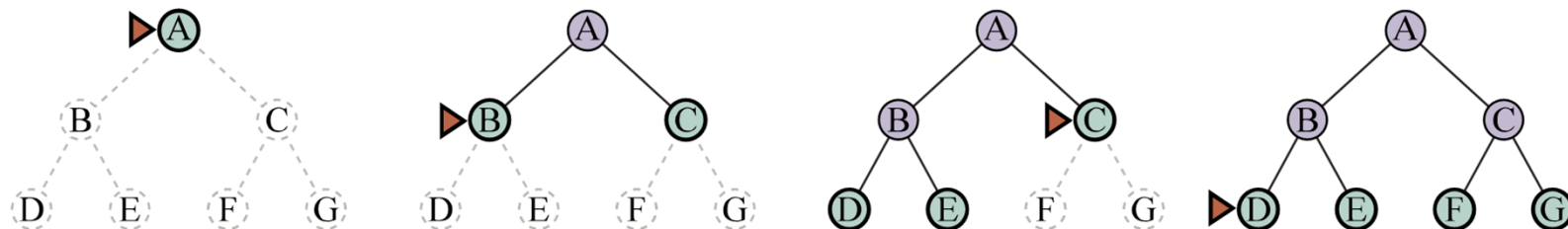
بررسی هدف بودن یک گره
پس از انتخاب آن برای گسترش
(پاپ شدن آن از صف)

پیاده‌سازی کارآمدتر و در
عین حال کامل برای BFS

آزمون هدف زودهنگام

Early Goal Test

بررسی هدف بودن یک گره
به محض تولید آن



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

جستجوی عرض-اول

ویژگی‌ها

ارزیابی الگوریتم جستجوی عرض-اول			
۱	۲	۳	۴
تمامیت <i>Completeness</i>	بهینگی هزینه <i>Cost Optimality</i>	پیچیدگی زمانی <i>Time Complexity</i>	پیچیدگی فضایی <i>Space Complexity</i>

بله

اگر b متناهی باشد

بله

اگر هزینه‌ی مسیر، تابع
غیرنزولی از عمق گره
باشد:
مثلاً
هزینه‌ی گام ۱ واحد

نمایی

نمایی

همه‌ی گره‌ها را در حافظه
نگه می‌دارد.

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

* مشکل فضای مصرفی بسیار بزرگ (و جدی‌تر از زمان) است: تولید 10 MB/sec گره، یعنی 2hrs = 860 GB

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Figure 3.13 Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 1 million nodes/second; 1000 bytes/node.

جستجوی هزینه-یکنواخت

جستجوی کمترین هزینه (الگوریتم دایکسترا)

UNIFORM-COST SEARCH (LEAST-COST SEARCH / DIJKSTRA'S ALGORITHM)

کم هزینه‌ترین گره در کناره را گسترش دهید.

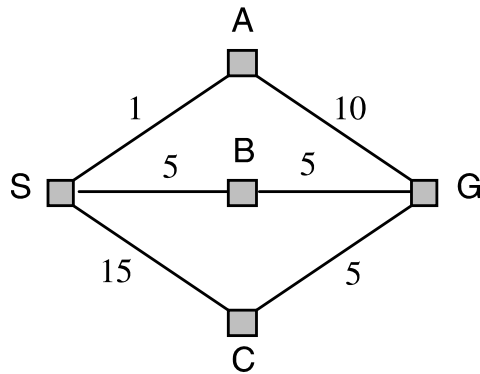
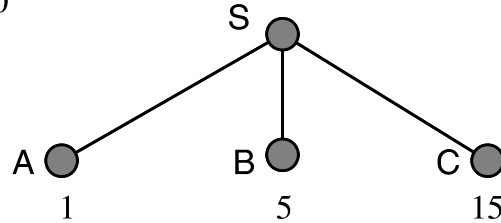
Expand least-cost node in the frontier.

قاعده

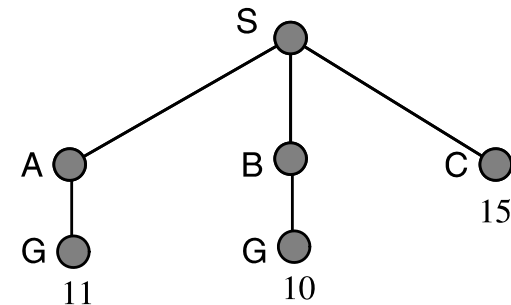
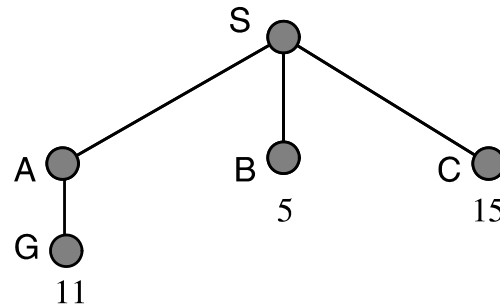
UCS

پیاده‌سازی: کناره، یک صف اولویت است (ترتیب گره‌ها بر اساس هزینه‌ی مسیر آنها)

S
0



(a)

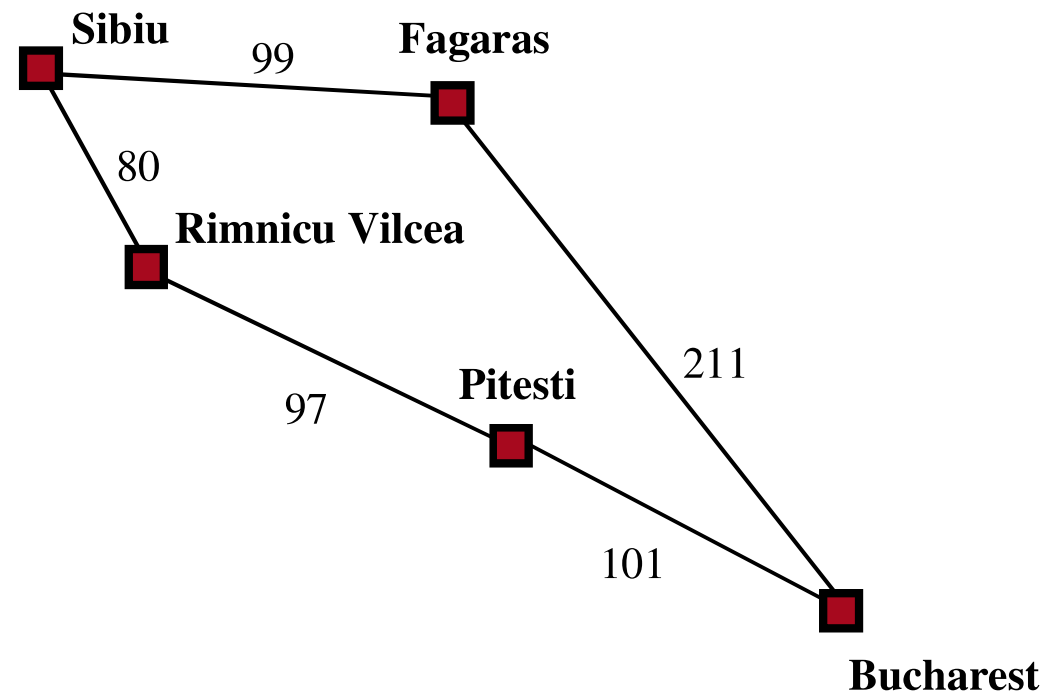


(b)

جستجوی هزینه-یکنواخت

شبکه

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



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

جستجوی هزینه-یکنواخت

ویژگی‌ها

ارزیابی الگوریتم جستجوی هزینه-یکنواخت			
۱	۲	۳	۴
تمامیت <i>Completeness</i>	بهینگی هزینه <i>Cost Optimality</i>	پیچیدگی زمانی <i>Time Complexity</i>	پیچیدگی فضایی <i>Space Complexity</i>
بله	بله	نمایی	نمایی
به شرط مثبت بودن هزینه‌ی گام‌ها $c(s, a, s') \geq \varepsilon > 0$	(گره‌ها بر حسب هزینه‌ی مسیر به طور صعودی بررسی می‌شوند.)	متناسب با تعداد گره‌هایی که هزینه‌ی مسیر آنها حداکثر برابر با هزینه‌ی راه‌حل بهینه C^* باشد. $O(b^{\lfloor C^* / \varepsilon \rfloor + 1})$	متناسب با تعداد گره‌هایی که هزینه‌ی مسیر آنها حداکثر برابر با هزینه‌ی راه‌حل بهینه C^* باشد. $O(b^{\lfloor C^* / \varepsilon \rfloor + 1})$

اگر هزینه‌ی هر گره برابر با عمق گره باشد، یا هزینه‌ی همه‌ی گام‌ها مساوی باشد، آن‌گاه UCS به BFS تبدیل می‌شود.

جستجوی عمق-اول

DEPTH-FIRST SEARCH

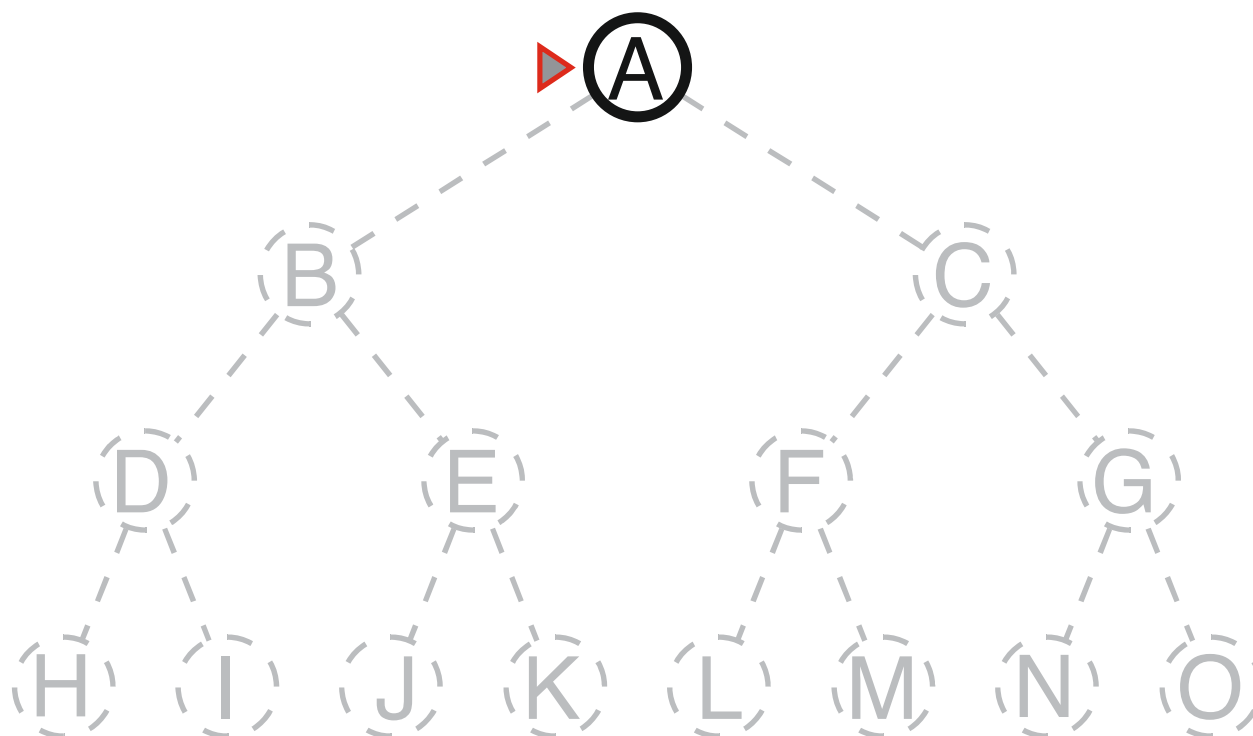
پرعمق‌ترین گره در کناره را گسترش دهید.

Expand deepest node in the frontier.

قاعده

DFS

پیاده‌سازی: کناره، یک صف LIFO (پشته) است (فرزندان جدید به ابتدای صف اضافه می‌شوند)



جستجوی عمق-اول

DEPTH-FIRST SEARCH

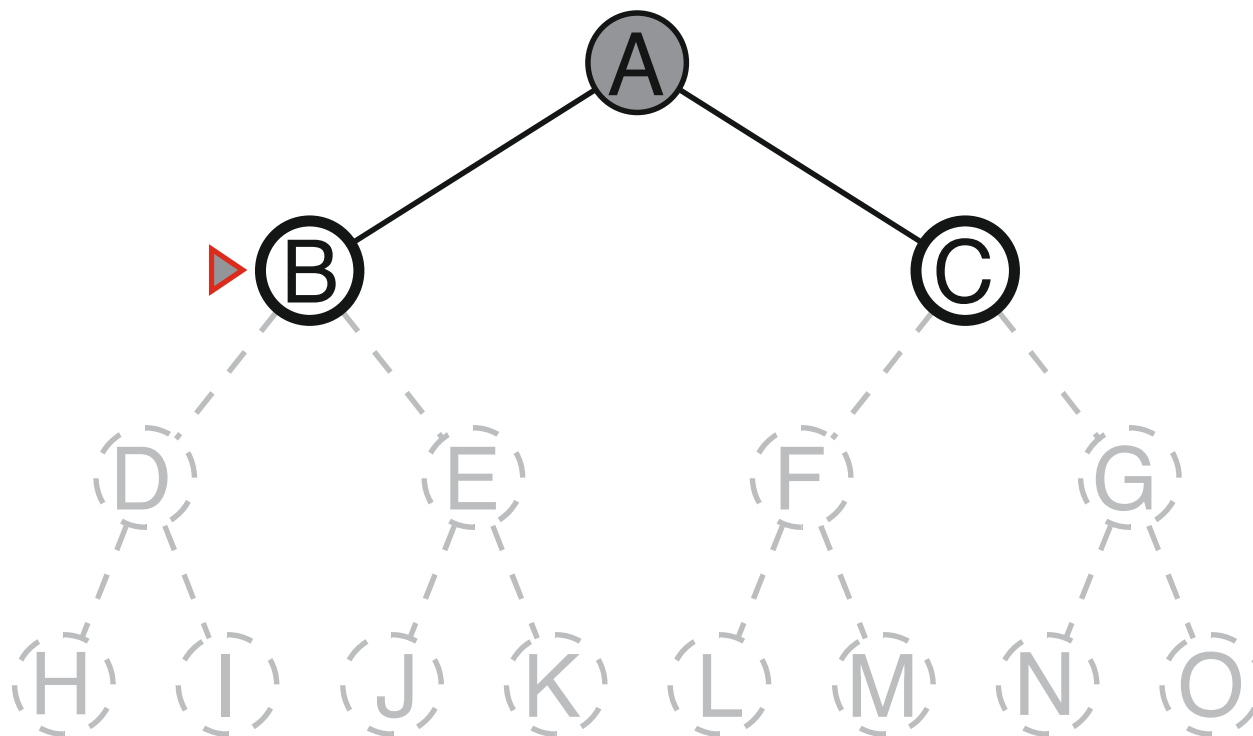
پرعمق‌ترین گره در کناره را گسترش دهید.

Expand deepest node in the frontier.

قاعده

DFS

پیاده‌سازی: کناره، یک صف LIFO (پشته) است (فرزندان جدید به ابتدای صف اضافه می‌شوند)



جستجوی عمق-اول

DEPTH-FIRST SEARCH

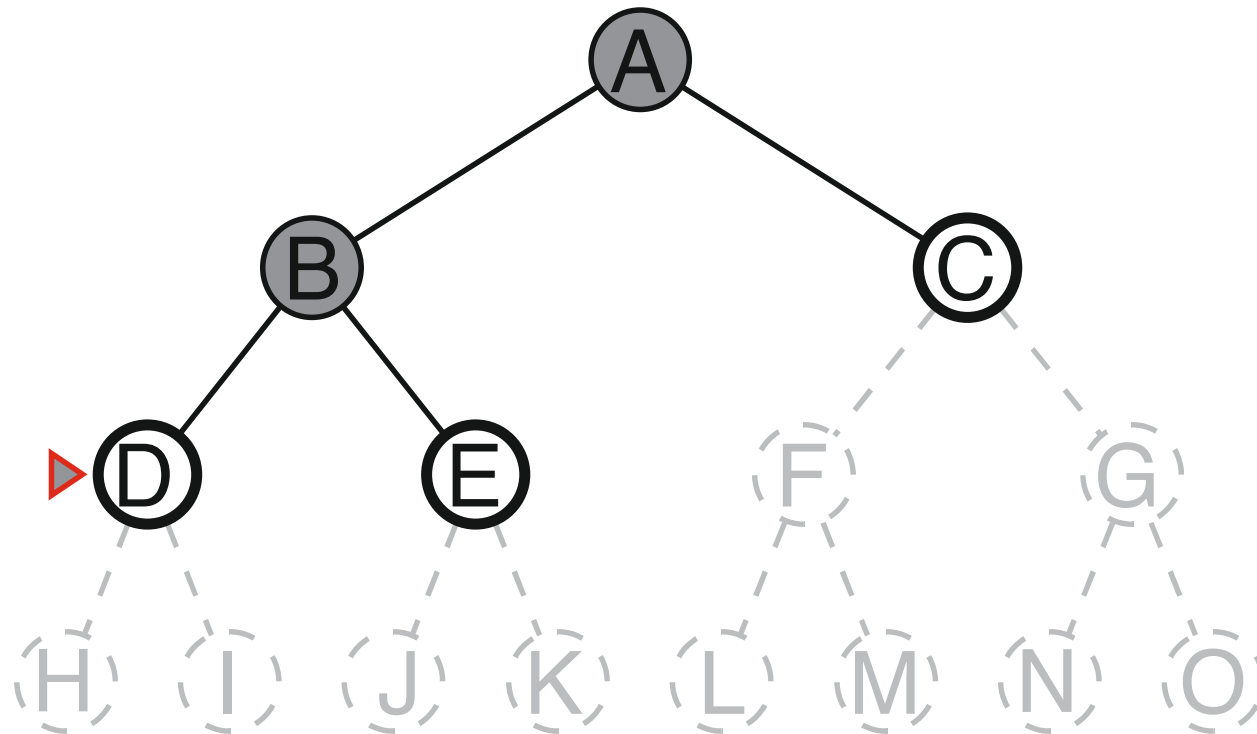
پرعمق‌ترین گره در کناره را گسترش دهید.

Expand deepest node in the frontier.

قاعده

DFS

پیاده‌سازی: کناره، یک صف LIFO (پشته) است (فرزندان جدید به ابتدای صف اضافه می‌شوند)



جستجوی عمق-اول

DEPTH-FIRST SEARCH

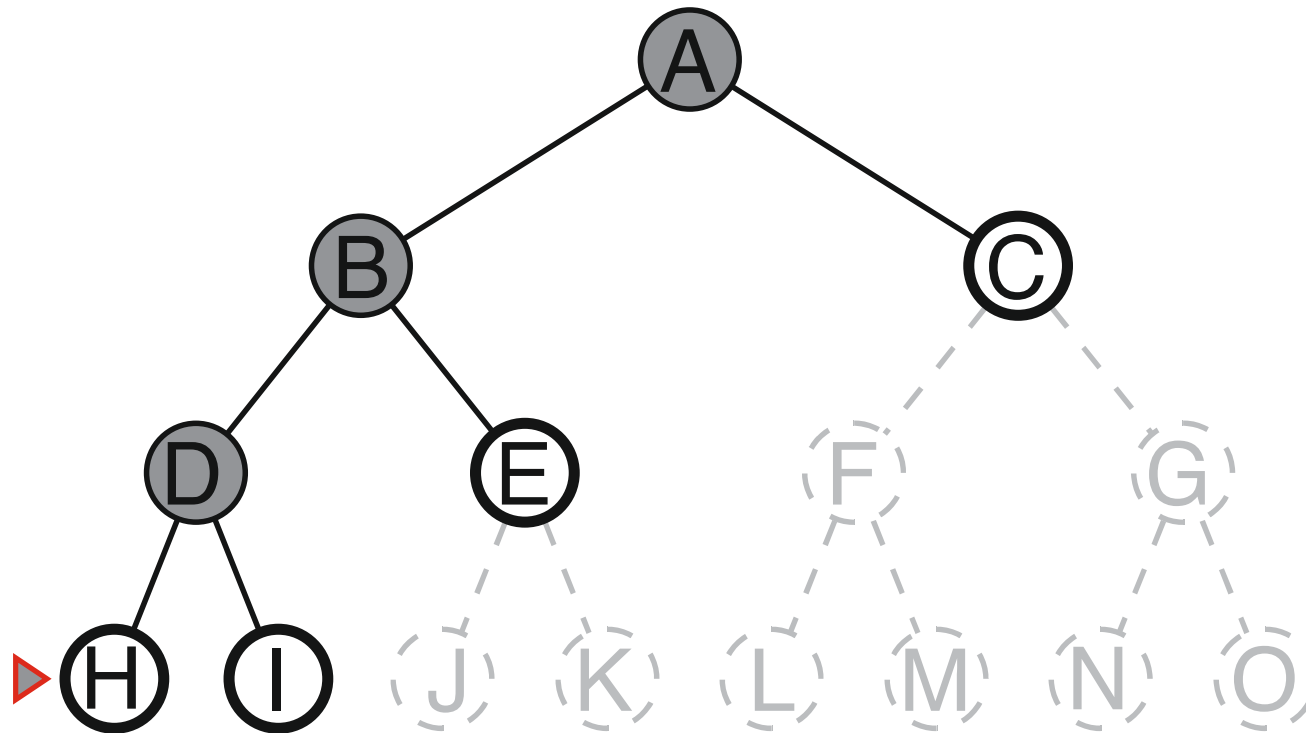
پر عمق ترین گره در کناره را گسترش دهید.

Expand deepest node in the frontier.

قاعده

DFS

پیاده سازی: کناره، یک صف LIFO (پشته) است (فرزندان جدید به ابتدای صف اضافه می شوند)



جستجوی عمق-اول

DEPTH-FIRST SEARCH

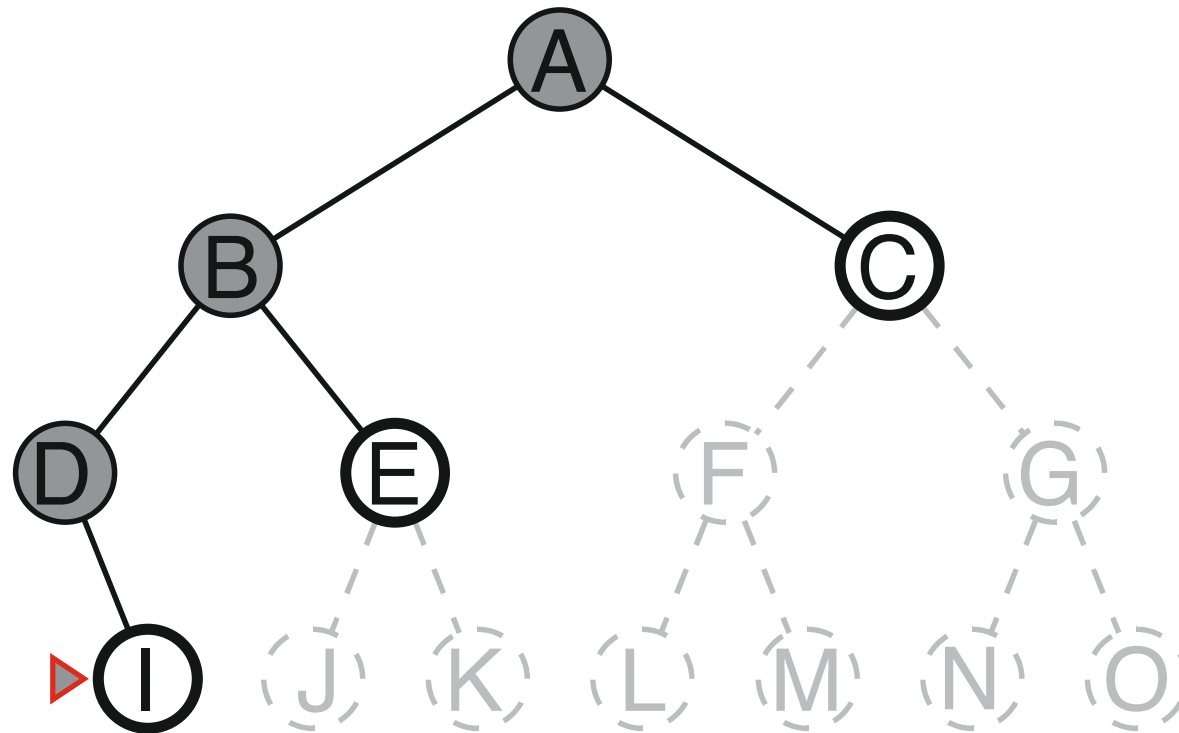
پر عمق ترین گره در کناره را گسترش دهید.

Expand deepest node in the frontier.

قاعده

DFS

پیاده سازی: کناره، یک صف LIFO (پشته) است (فرزندان جدید به ابتدای صف اضافه می شوند)



جستجوی عمق-اول

DEPTH-FIRST SEARCH

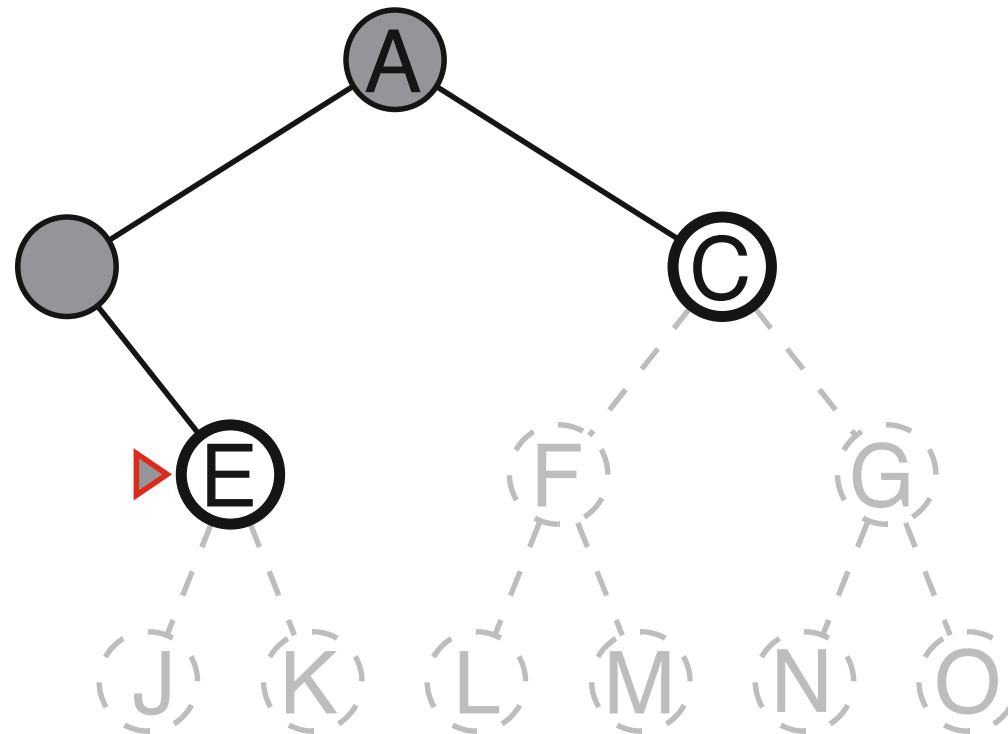
پر عمق ترین گره در کناره را گسترش دهید.

Expand deepest node in the frontier.

قاعده

DFS

پیاده سازی: کناره، یک صف LIFO (پشته) است (فرزندان جدید به ابتدای صف اضافه می شوند)



جستجوی عمق-اول

DEPTH-FIRST SEARCH

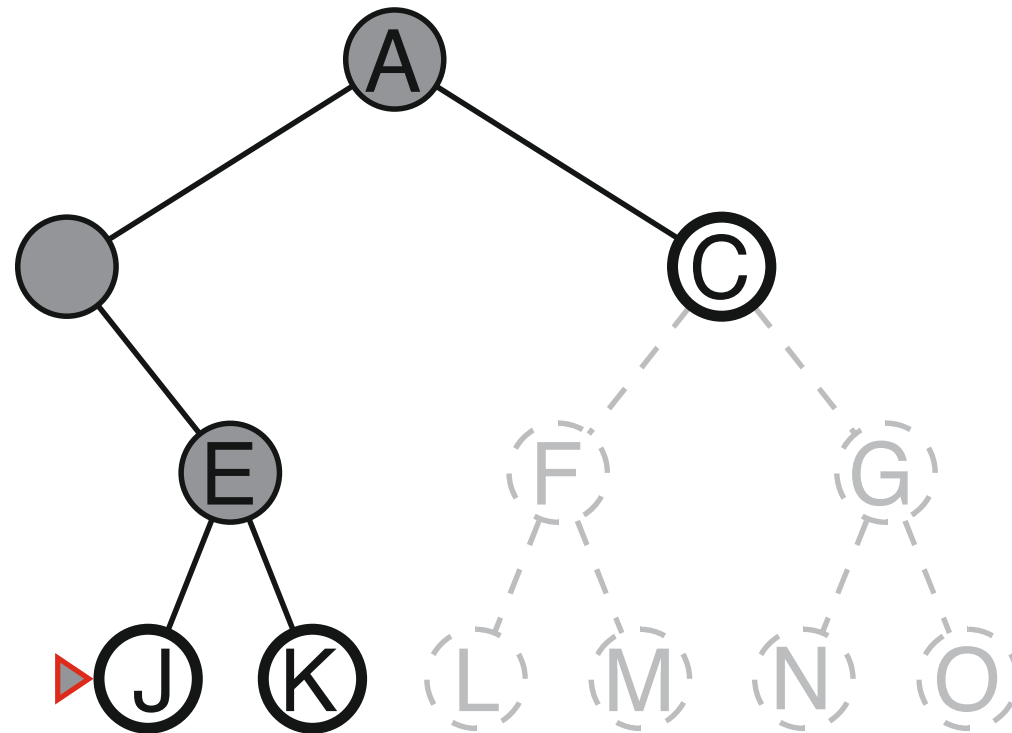
پرعمق‌ترین گره در کناره را گسترش دهید.

Expand deepest node in the frontier.

قاعده

DFS

پیاده‌سازی: کناره، یک صف LIFO (پشته) است (فرزندان جدید به ابتدای صف اضافه می‌شوند)



جستجوی عمق-اول

DEPTH-FIRST SEARCH

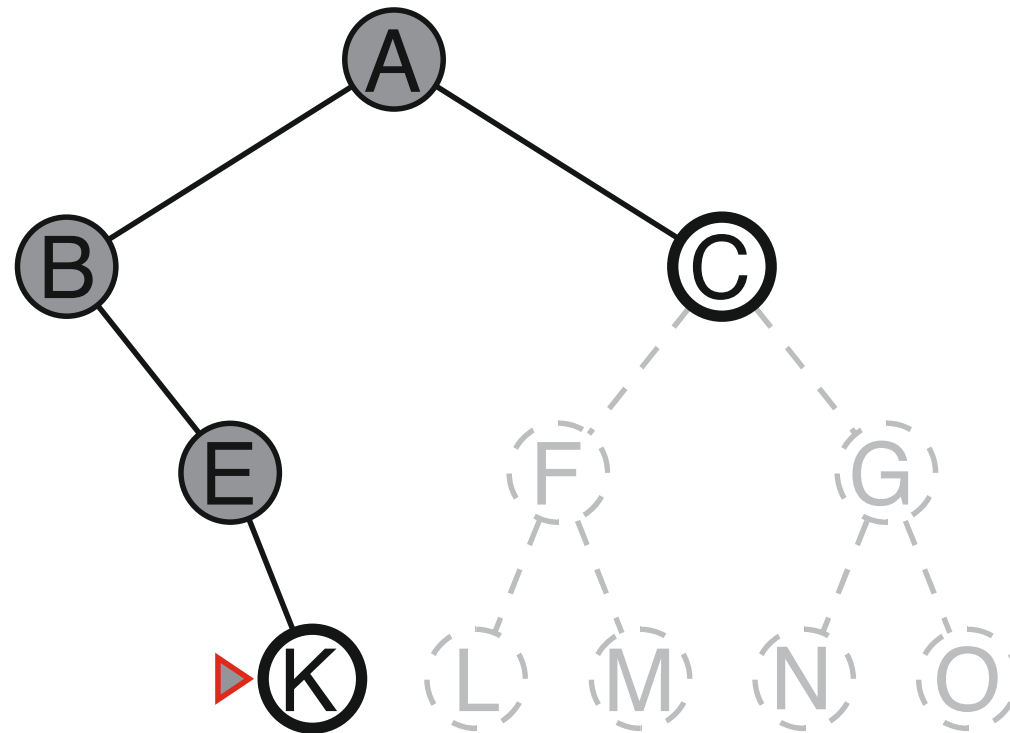
پرعمق‌ترین گره در کناره را گسترش دهید.

Expand deepest node in the frontier.

قاعده

DFS

پیاده‌سازی: کناره، یک صف LIFO (پشته) است (فرزندان جدید به ابتدای صف اضافه می‌شوند)



جستجوی عمق-اول

DEPTH-FIRST SEARCH

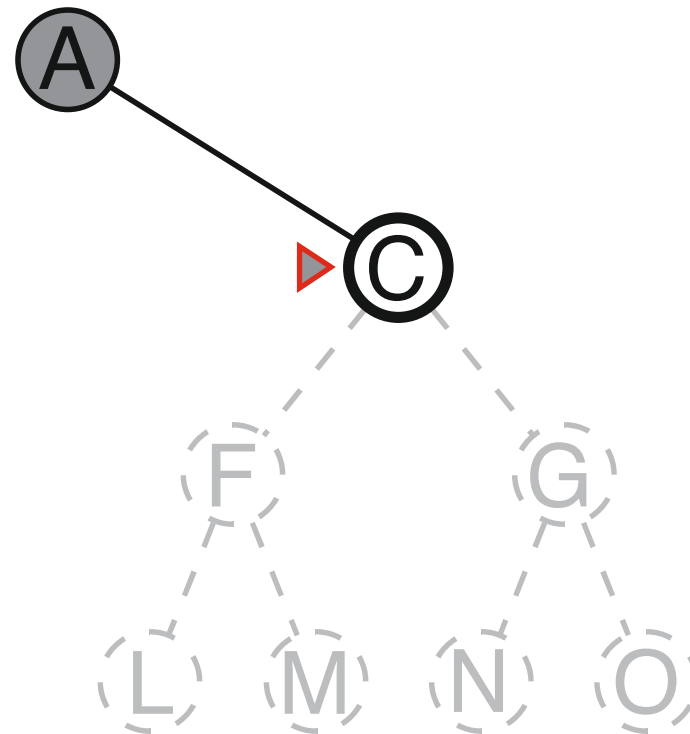
پر عمق ترین گره در کناره را گسترش دهید.

Expand deepest node in the frontier.

قاعده

DFS

پیاده سازی: کناره، یک صف LIFO (پشته) است (فرزندان جدید به ابتدای صف اضافه می شوند)



جستجوی عمق-اول

DEPTH-FIRST SEARCH

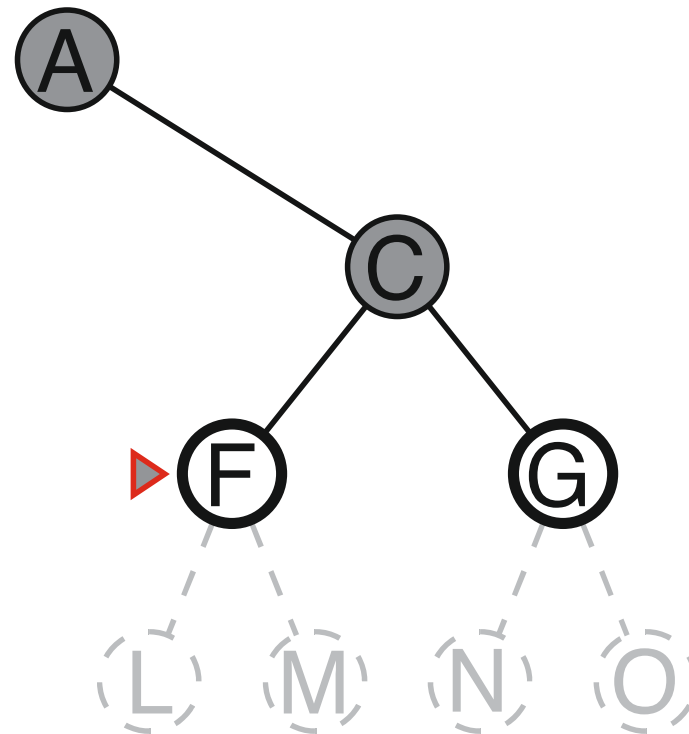
پرعمق‌ترین گره در کناره را گسترش دهید.

Expand deepest node in the frontier.

قاعده

DFS

پیاده‌سازی: کناره، یک صف LIFO (پشته) است (فرزندان جدید به ابتدای صف اضافه می‌شوند)



جستجوی عمق-اول

DEPTH-FIRST SEARCH

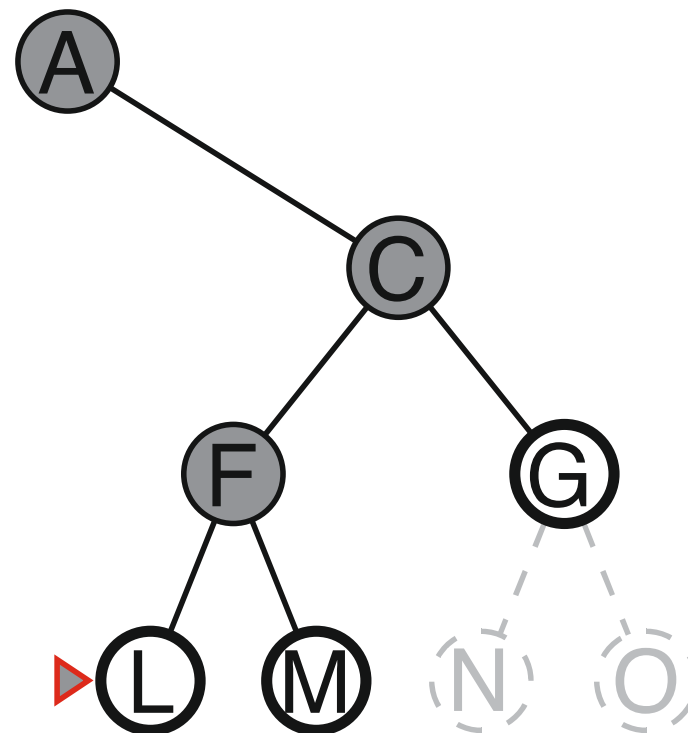
پرعمق‌ترین گره در کناره را گسترش دهید.

Expand deepest node in the frontier.

قاعده

DFS

پیاده‌سازی: کناره، یک صف LIFO (پشته) است (فرزندان جدید به ابتدای صف اضافه می‌شوند)



جستجوی عمق-اول

DEPTH-FIRST SEARCH

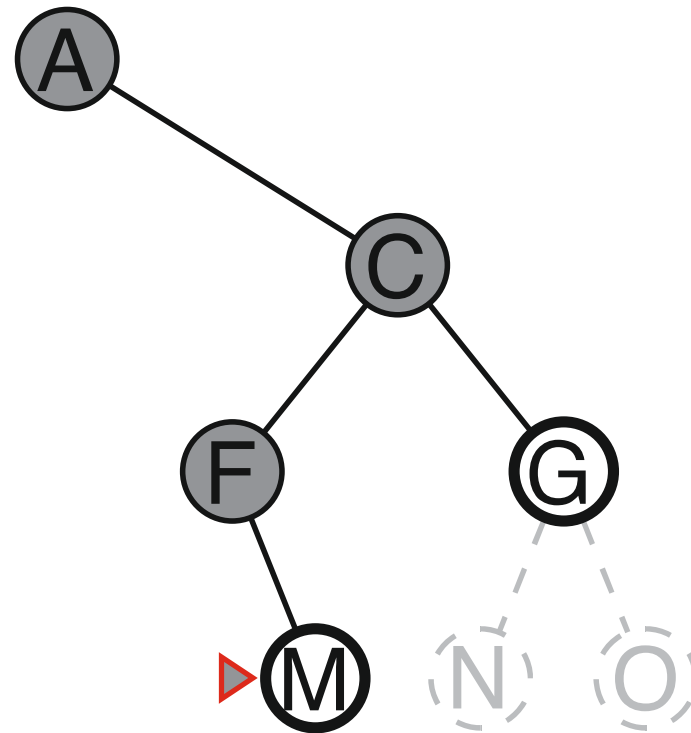
پر عمق ترین گره در کناره را گسترش دهید.

Expand deepest node in the frontier.

قاعده

DFS

پیاده سازی: کناره، یک صف LIFO (پشته) است (فرزندان جدید به ابتدای صف اضافه می شوند)



جستجوی عمق-اول

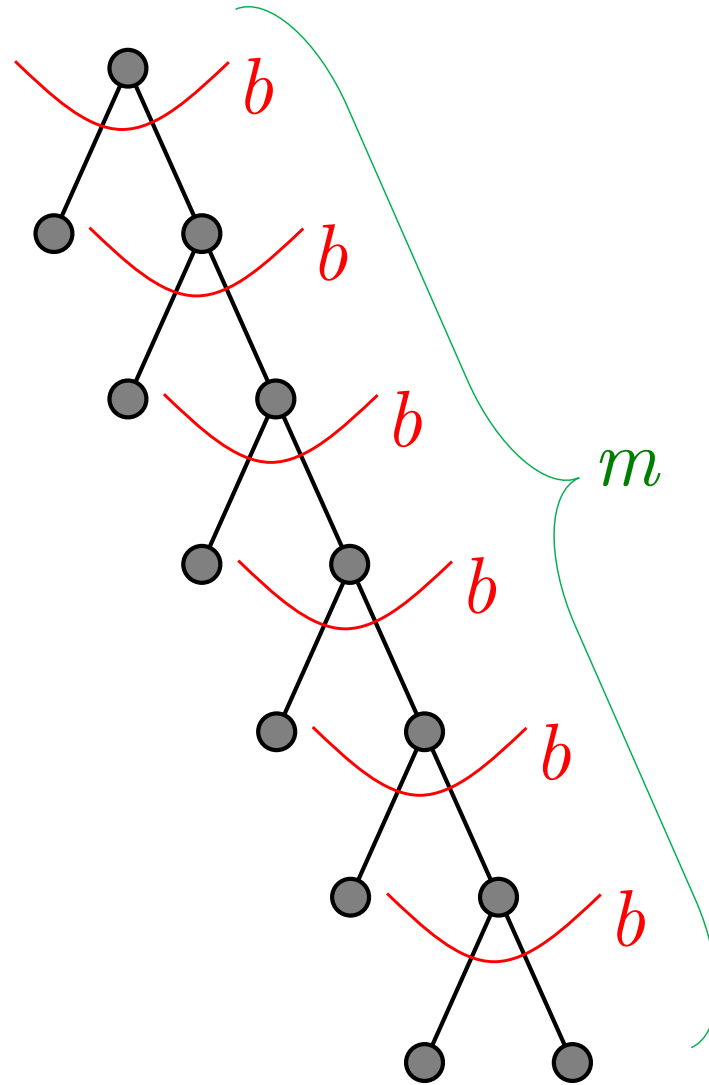
ویژگی‌ها

ارزیابی الگوریتم جستجوی عمق-اول			
۱	۲	۳	۴
تمامیت <i>Completeness</i>	بهینگی هزینه <i>Cost Optimality</i>	پیچیدگی زمانی <i>Time Complexity</i>	پیچیدگی فضایی <i>Space Complexity</i>
خیر	خیر	نمایی	خطی
در فضاهای حالت با عمق نامتناهی یا فضاهای دارای حلقه، تضمینی ندارد که به جواب برسد. در فضاهای حالت با عمق متناهی به شرط بررسی حالت‌های تکراری، کامل است.	گره‌ی هدف با هزینه‌ی مسیر بدتر، ممکن است زودتر ملاقات شود.	بدتر از عرض-اول در بدترین حالت $O(b^m)$ $m \gg d$	فقط گره‌هایی که در مسیر ریشه به گره‌ی جاری قرار دارند، به همراه همزادهایشان نگهداری می‌شوند. $O(bm)$

اگر تعداد راه‌حل‌های مسئله زیاد باشد، DFS احتمالاً سریع‌تر از روش BFS به جواب می‌رسد.

جستجوی عمق-اول

فضای مصرفی



$$O(bm)$$

جستجوی عقب‌گرد

حالت خاصی از جستجوی عمق-اول

BACKTRACKING SEARCH

مشابه روش عمق-اول

با این تفاوت که در هر مرحله، فقط یکی از فرزندان هر گره تولید می‌شود (به جای همه‌ی فرزندان).

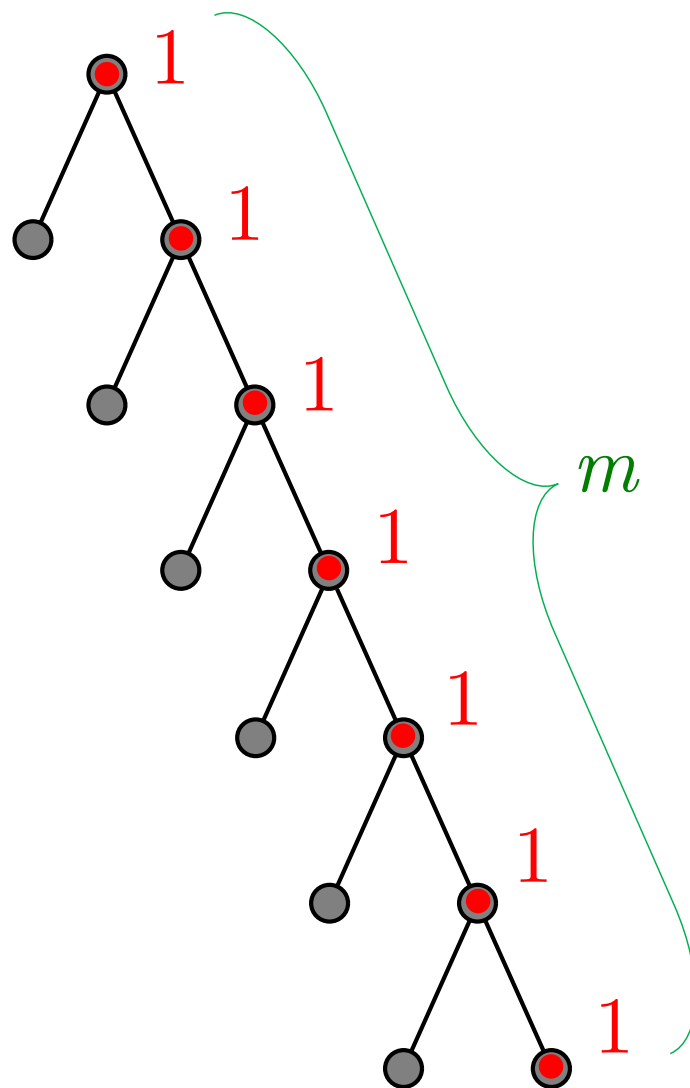
مزیت: میزان فضای مصرفی (و نیز زمان مصرفی) کمتر می‌شود.

$$O(m)$$

- هر گره‌ی والد گسترش داده شده به طور جزئی، به خاطر می‌آورد که کدام فرزند باید در مرحله‌ی بعد تولید شود.
- ترفند: هر فرزند با ایجاد تغییر مستقیم در توصیف حالت جاری (به جای تولید ابتدایی آن) تولید می‌شود.

جستجوی عقب‌گرد

فضای مصرفی



$$O(m)$$

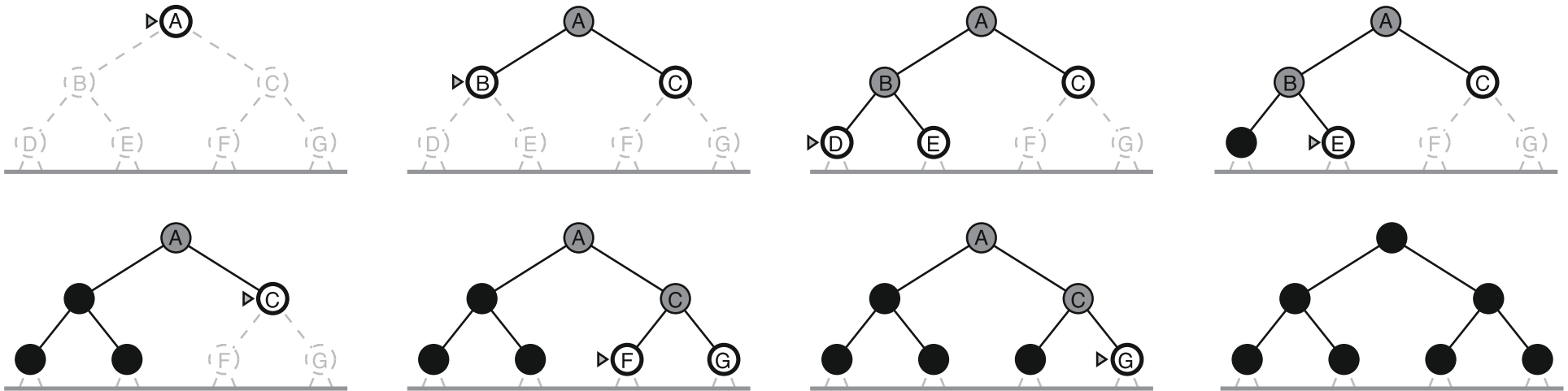
جستجوی عمقی محدودشده

DEPTH-LIMITED SEARCH

جستجوی عمق-اول با حد عمق l
Execute depth-first search with depth limit 'l'.

قاعده
DLS

پیاده‌سازی: گره‌های عمق l مابعد (فرزند) ندارند.



$$l = 2$$

جستجوی عمقی محدود شده

شبکه

```
function DEPTH-LIMITED-SEARCH(problem,  $\ell$ ) returns a node or failure or cutoff  
  frontier  $\leftarrow$  a LIFO queue (stack) with  $\text{NODE}(\text{problem}.\text{INITIAL})$  as an element  
  result  $\leftarrow$  failure  
  while not IS-EMPTY(frontier) do  
    node  $\leftarrow$  POP(frontier)  
    if problem.IS-GOAL(node.STATE) then return node  
    if DEPTH(node) >  $\ell$  then  
      result  $\leftarrow$  cutoff  
    else if not IS-CYCLE(node) do  
      for each child in EXPAND(problem, node) do  
        add child to frontier  
  return result
```

جستجوی عمقی محدودشده

ویژگی‌ها

ارزیابی الگوریتم جستجوی عمقی محدودشده			
۱	۲	۳	۴
تمامیت <i>Completeness</i>	بهینگی هزینه <i>Cost Optimality</i>	پیچیدگی زمانی <i>Time Complexity</i>	پیچیدگی فضایی <i>Space Complexity</i>
خیر	خیر	نمایی	خطی
مگر اینکه $l \geq d$	گره‌ی هدف با هزینه‌ی مسیر بدتر، ممکن است زودتر ملاقات شود.	مشابه روش عمق-اول $O(b^l)$	فقط گره‌هایی که در مسیر ریشه به گره‌ی جاری قرار دارند، به همراه همزادهایشان نگهداری می‌شوند. $O(bl)$

قطر (diameter) فضای حالت، گزینه‌ی مناسبی برای حد l است.

DFS حالت خاص DLS با $l = \infty$ است.

جستجوی عمیق‌کننده‌ی تکراری

جستجوی عمق-اول عمیق‌کننده‌ی تکراری

ITERATIVE DEEPENING SEARCH (ITERATIVE DEEPENING DEPTH-FIRST SEARCH)

تکرار جستجوی عمقی محدودشده با افزایش حد عمق در هر مرحله

Execute DLS by increasing depth limit for each phase

قاعده

IDS

پیاده‌سازی: اجرای DLS در یک حلقه برای $l = 0$ تا $l = \infty$

Limit = 0



جستجوی عمیق‌کننده تکراری

جستجوی عمق-اول عمیق‌کننده تکراری

ITERATIVE DEEPENING SEARCH (ITERATIVE DEEPENING DEPTH-FIRST SEARCH)

تکرار جستجوی عمقی محدودشده با افزایش حد عمق در هر مرحله

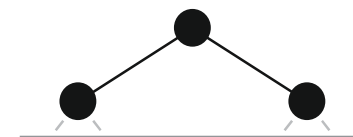
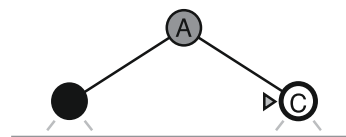
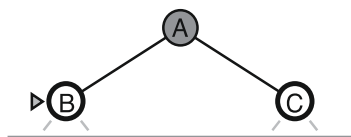
Execute DLS by increasing depth limit for each phase

قاعده

IDS

پیاده‌سازی: اجرای DLS در یک حلقه برای $l = 0$ تا $l = \infty$

Limit = 1



جستجوی عمیق‌کننده تکراری

جستجوی عمق-اول عمیق‌کننده تکراری

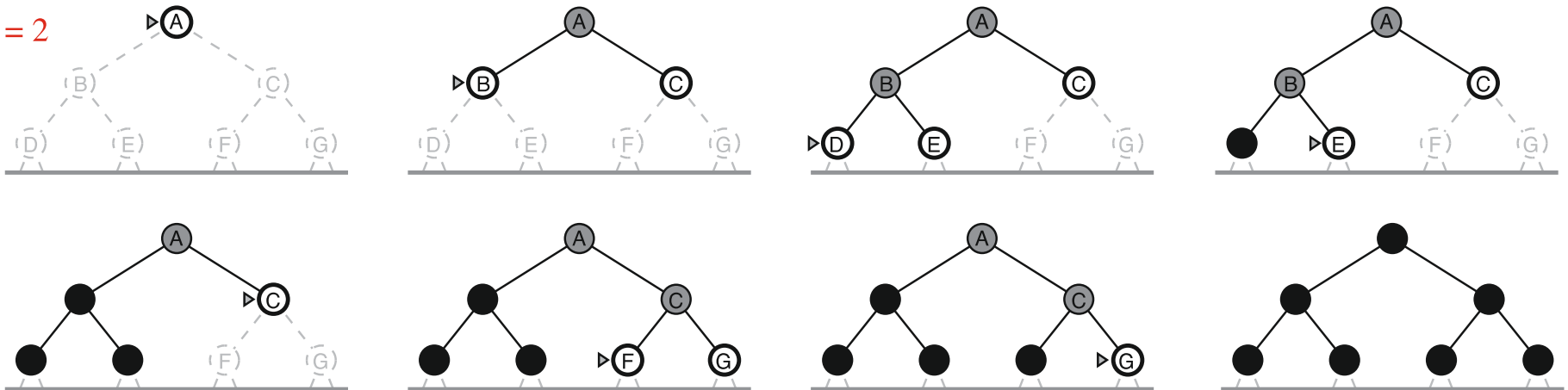
ITERATIVE DEEPENING SEARCH (ITERATIVE DEEPENING DEPTH-FIRST SEARCH)

تکرار جستجوی عمقی محدودشده با افزایش حد عمق در هر مرحله
Execute DLS by increasing depth limit for each phase

قاعده
IDS

پیاده‌سازی: اجرای DLS در یک حلقه برای $l = 0$ تا $l = \infty$

Limit = 2



جستجوی عمیق‌کننده تکراری

جستجوی عمق-اول عمیق‌کننده تکراری

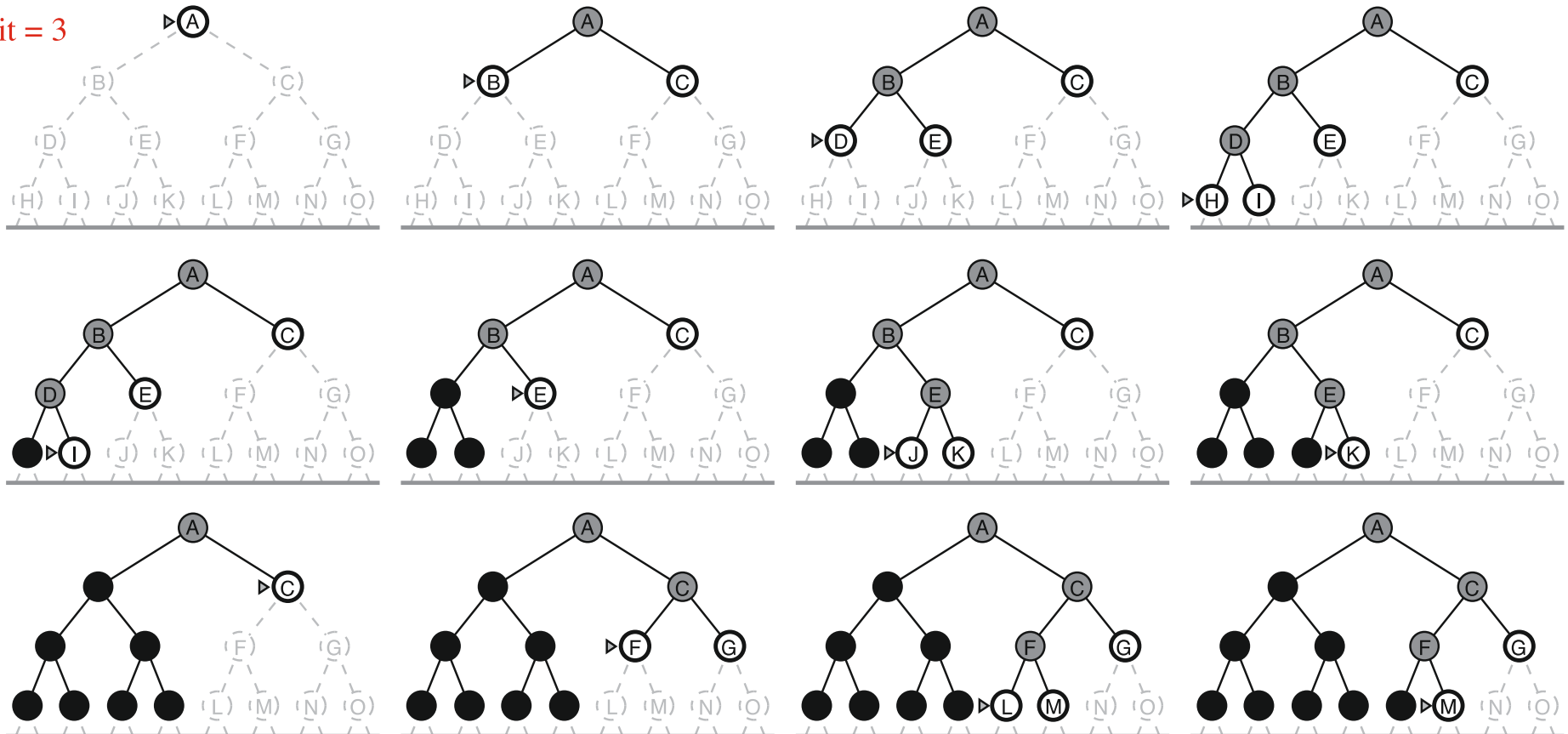
ITERATIVE DEEPENING SEARCH (ITERATIVE DEEPENING DEPTH-FIRST SEARCH)

تکرار جستجوی عمقی محدودشده با افزایش حد عمق در هر مرحله
Execute DLS by increasing depth limit for each phase

قاعده
IDS

پیاده‌سازی: اجرای DLS در یک حلقه برای $l = 0$ تا $l = \infty$

Limit = 3



جستجوی عمیق‌کننده‌ی تکراری

شبکه‌کد

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

جستجوی عمیق‌کننده‌ی تکراری

ویژگی‌ها

ارزیابی الگوریتم جستجوی عمیق‌کننده‌ی تکراری			
۱	۲	۳	۴
تمامیت <i>Completeness</i>	بهینگی هزینه <i>Cost Optimality</i>	پیچیدگی زمانی <i>Time Complexity</i>	پیچیدگی فضایی <i>Space Complexity</i>

بله

بله

نمایی

خطی

به شرط اینکه هزینه‌ی
مسیر هر گره تابعی
غیرنزولی از عمق گره
باشد (مثلاً هزینه‌ی گام
۱ باشد).

$$O(b^d)$$

مشابه روش عمق-اول

$$O(bd)$$

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

IDS مزایای روش‌های عرض-اول و عمق-اول را همزمان دارد:

بهترین روش جستجوی ناآگاهانه: زمانی که فضای جستجو بزرگ و عمق راه‌حل نامشخص است.

مقایسه: BFS با IDS

$$b = 10, \quad d = 5$$

$$N(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$$

$$N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 = 111,110$$

$$N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100$$

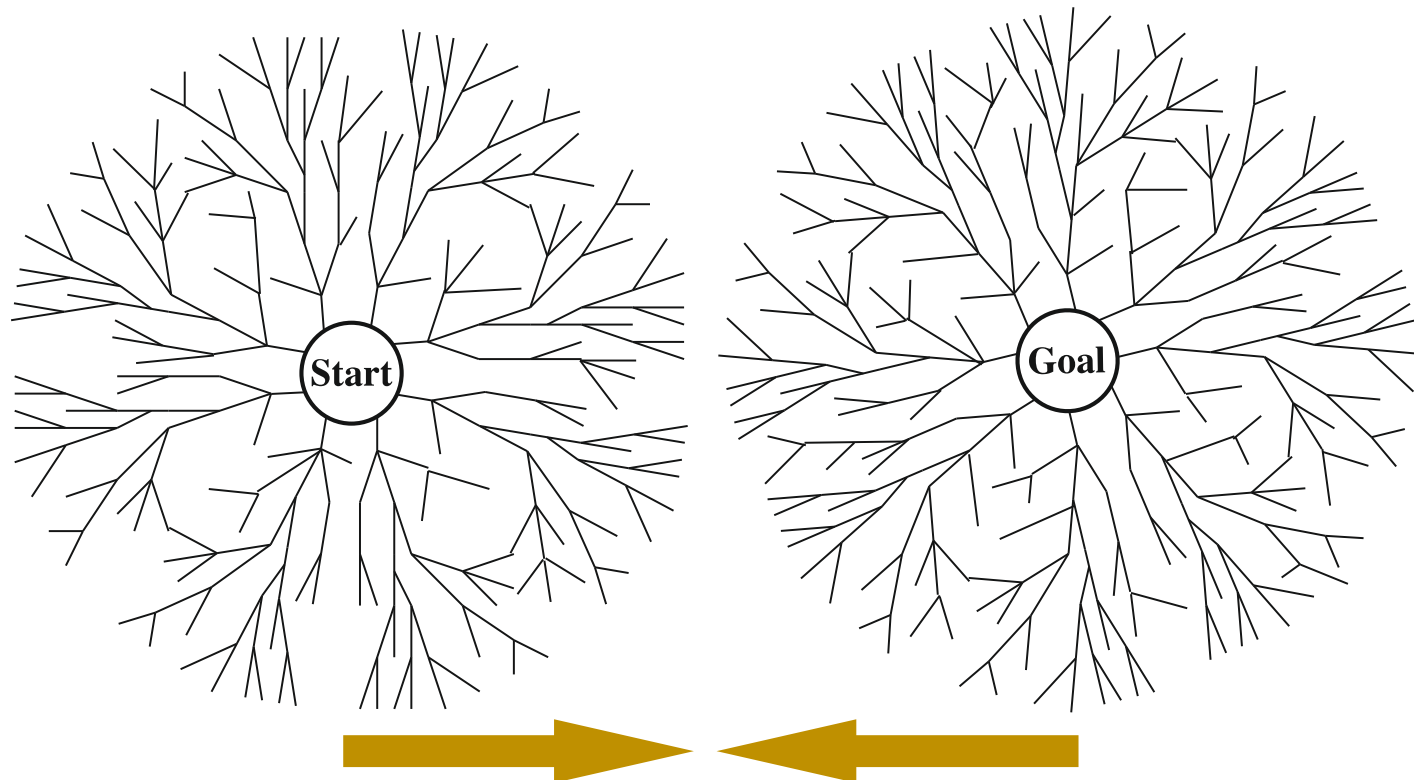
پیچیدگی زمانی به لحاظ مجانبی مشابه روش عرض-اول است.

مشکل: مقداری هزینه اضافی برای تولید چندباره‌ی سطوح بالایی درخت وجود دارد، اما زیاد بزرگ نیست؛ و با بزرگ‌تر شدن b این نسبت کوچکتر می‌شود.

جستجوی دو طرفه

BIDIRECTIONAL SEARCH

دو جستجو به طور همزمان انجام می شود: از حالت اولیه به جلو (پیشرو) + از حالت هدف به عقب (پسرو)	قاعده <i>BDS</i>
پیاده سازی: مطابق قاعده: زمانی که دو جستجو به هم می رسند، متوقف می شویم.	



جستجوی دوطرفه

شبه کد

BIDIRECTIONAL SEARCH

```

function BIBF-SEARCH( $problem_F, f_F, problem_B, f_B$ ) returns a solution node, or failure
   $node_F \leftarrow \text{NODE}(problem\_F.INITIAL)$  // Node for a start state
   $node_B \leftarrow \text{NODE}(problem\_B.INITIAL)$  // Node for a goal state
   $frontier_F \leftarrow$  a priority queue ordered by  $f_F$ , with  $node_F$  as an element
   $frontier_B \leftarrow$  a priority queue ordered by  $f_B$ , with  $node_B$  as an element
   $reached_F \leftarrow$  a lookup table, with one key  $node_F.STATE$  and value  $node_F$ 
   $reached_B \leftarrow$  a lookup table, with one key  $node_B.STATE$  and value  $node_B$ 
   $solution \leftarrow failure$ 
  while not TERMINATED( $solution, frontier_F, frontier_B$ ) do
    if  $f_F(\text{TOP}(frontier_F)) < f_B(\text{TOP}(frontier_B))$  then
       $solution \leftarrow \text{PROCEED}(F, problem_F, frontier_F, reached_F, reached_B, solution)$ 
    else  $solution \leftarrow \text{PROCEED}(B, problem_B, frontier_B, reached_B, reached_F, solution)$ 
  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.
   $node \leftarrow \text{POP}(frontier)$ 
  for each  $child$  in EXPAND( $problem, node$ ) do
     $s \leftarrow child.STATE$ 
    if  $s$  not in  $reached$  or  $\text{PATH-COST}(child) < \text{PATH-COST}(reached[s])$  then
       $reached[s] \leftarrow child$ 
      add  $child$  to  $frontier$ 
    if  $s$  is in  $reached_2$  then
       $solution_2 \leftarrow \text{JOIN-NODES}(dir, child, reached_2[s])$ 
      if  $\text{PATH-COST}(solution_2) < \text{PATH-COST}(solution)$  then
         $solution \leftarrow solution_2$ 
  return  $solution$ 

```

جستجوی دوطرفه

ویژگی‌ها

ارزیابی الگوریتم جستجوی دوطرفه			
۱	۲	۳	۴
تمامیت <i>Completeness</i>	بهینگی هزینه <i>Cost Optimality</i>	پیچیدگی زمانی <i>Time Complexity</i>	پیچیدگی فضایی <i>Space Complexity</i>
بله	بله	نمایی	نمایی

به شرط اینکه در دو
جهت از BFS استفاده
شود و b متناهی باشد.

به شرط اینکه در دو
جهت از BFS استفاده
شود و هزینه‌ی مسیر
هر گره تابعی غیرنزولی
از عمق گره باشد.

مشابه روش عرض-اول، اما
فاکتور انشعاب جذر می‌شود.

$$O(2b^{d/2}) = O(b^{d/2}) = O((\sqrt{b})^d)$$

مشکل اصلی: پیدا کردن ماقبل یک گره (predecessor) است.
کنش‌ها باید وارون‌پذیر باشند.

– در بعضی مسائل **حالت‌های هدف چندگانه** هستند و تعیین نقطه شروع جستجوی پس‌رو ساده نیست.

– نیازمند روشی کارآمد برای تعیین اشتراک دو مجموعه (تعیین رسیدن دو جستجو به هم): جدول درهم‌سازی $O(1)$

روش‌های جستجوی ناآگاهانه

جدول مقایسه

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	$O(b^d)$	$O(b^{1+\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$

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.

مقایسه‌های فوق برای جستجوهای درختی است که وقوع حالت‌های تکراری را بررسی نمی‌کنند.

برای جستجوی گرافی که وقوع حالت‌های تکراری را بررسی می‌کنند:

جستجوی عمق اول برای فضاهای حالت متناهی تمامیت دارد و پیچیدگی‌های زمانی و فضایی به اندازه‌ی فضای حالت محدود می‌شود (تعداد رأس‌ها و یال‌ها)

بِسْمِ اللَّهِ الرَّحْمَنِ الرَّحِيمِ



هوش مصنوعی

فصل ۳

حل مسئله با جستجو (۳)

Solving Problems by Searching (3)

کاظم فولادی قلعه
دانشکده مهندسی، دانشکدگان فارابی
دانشگاه تهران

<http://courses.fouladi.ir/ai>

۵

استراتژی‌های
جستجوی
آگاهانه
(هیوریستیک)

استراتژی‌های جستجوی آگاهانه

جستجوی هیوریستیک

الگوریتم‌های جستجوی همه‌منظوره	
آگاهانه <i>Informed</i>	ناآگاهانه <i>Uninformed</i>
عامل در مورد اینکه کجا به دنبال راه حل مسئله بگردد، کم و بیش راهنماهایی دارد.	عامل بجز تعریف مسئله، هیچ اطلاعاتی در مورد مسئله‌ی تحت بررسی خود ندارد.

- جستجوی بهترین-اول (Best-first)
- جستجوی حریصانه (Greedy)
- جستجوی A^*

جستجوی بهترین-اول

BEST-FIRST SEARCH

بهترین گره در کناره را گسترش دهید.

Expand best node in the frontier.

قاعده

Best-FS

پیاده‌سازی: کناره، یک صف اولویت است (ترتیب گره‌ها بر اساس تابع ارزیابی $f(n)$)

جستجوی بهترین-اول

Best-First Search

جستجوی حریصانه

Greedy Search

جستجوی A^*

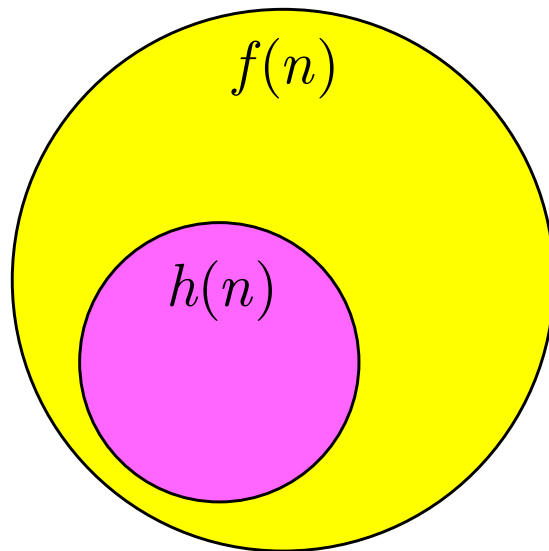
A Search*

از هر دوی جستجوی درختی و جستجوی گراف‌ی می‌توان استفاده کرد.

تابع ارزیابی

EVALUATION FUNCTION

میزان مطلوب / خوب بودن یک گره n با تابع ارزیابی $f(n)$ مشخص می شود.



$$f(n)$$

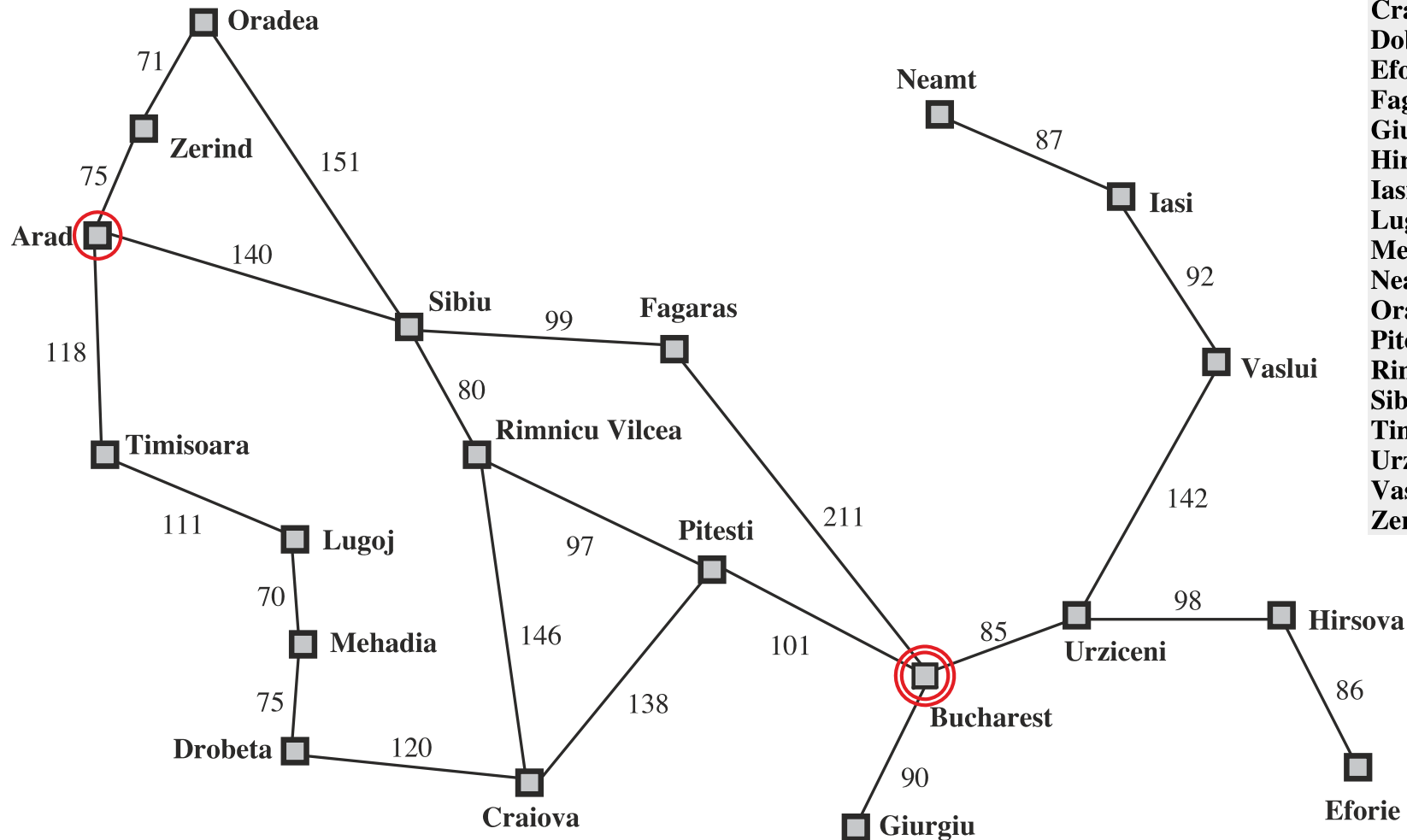
(مهم ترین مؤلفه ی تابع ارزیابی)
تخمین مطلوب بودن یک گره برای رسیدن به یک هدف

تابع هیوریستیک
Heuristic Function

$$h(n)$$

مثال: مسئله‌ی رومانی

نقشه: فاصله‌ی گام‌ها به کیلومتر + فاصله‌ی خط مستقیم هر شهر تا بخارست



Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

جستجوی حریصانه

جستجوی بهترین-اول حریصانه

GREEDY SEARCH (GREEDY BEST-FIRST SEARCH)

گره‌ای را گسترش بدهید که **به نظر می‌رسد** به هدف نزدیک‌تر است.

*Expand the node that **appears** to be closest to goal.*

قاعده

GBS

پیاده‌سازی: کناره، یک صف اولویت است (ترتیب گره‌ها بر اساس تخمین هزینه‌ی آنها تا هدف)

$$f(n) = h(n)$$

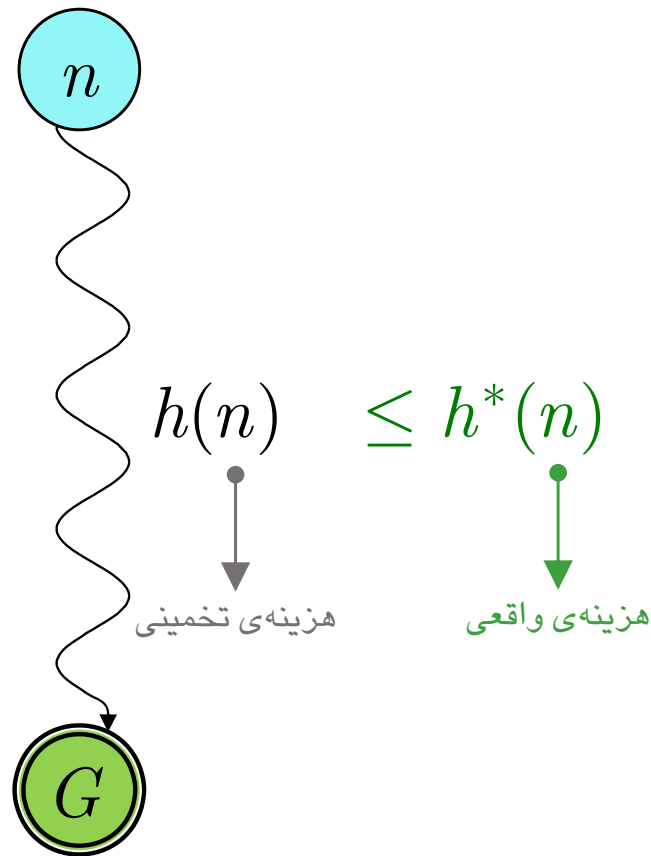


میزان هزینه تا هدف

تابع هیوریستیک

HEURISTIC FUNCTION

هزینه‌ی تخمینی بر اساس ارزان‌ترین مسیر از حالت گرهی n تا یک حالت هدف



$$h(G) = 0 \text{ برای هر گرهی هدف}$$

جستجوی حریصانه

جستجوی بهترین-اول حریصانه

GREEDY SEARCH (GREEDY BEST-FIRST SEARCH)

گره‌ای را گسترش بدهید که به نظر می‌رسد به هدف نزدیک‌تر است.

Expand the node that appears to be closest to goal.

قاعده

GBS

پیاده‌سازی: کناره، یک صف اولویت است (ترتیب گره‌ها بر اساس تخمین هزینه‌ی آنها تا هدف)

$h(n) = h_{\text{SLD}}(n) = \text{straight-line distance from } n \text{ to Bucharest}$

(a) The initial state



جستجوی حریصانه

جستجوی بهترین-اول حریصانه

GREEDY SEARCH (GREEDY BEST-FIRST SEARCH)

گره‌ای را گسترش بدهید که به نظر می‌رسد به هدف نزدیک‌تر است.

Expand the node that appears to be closest to goal.

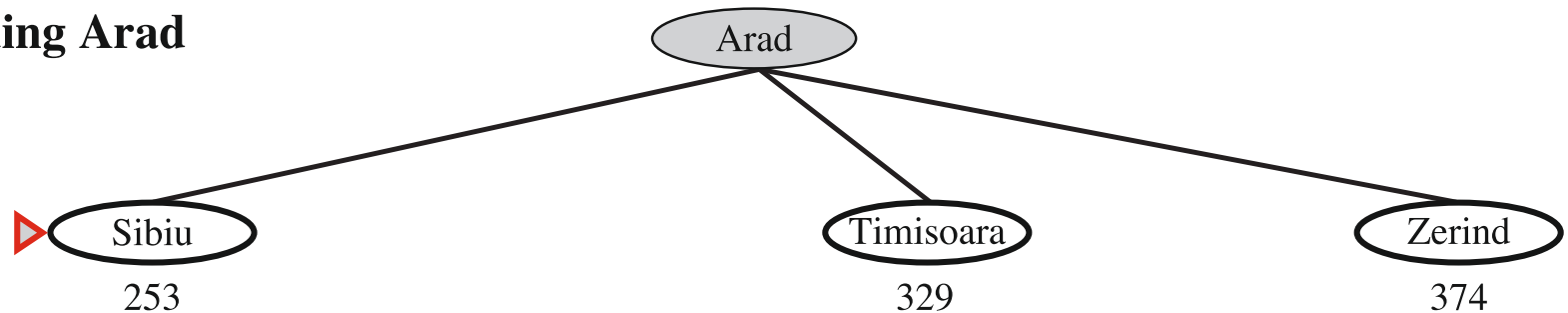
قاعده

GBS

پیاده‌سازی: کناره، یک صف اولویت است (ترتیب گره‌ها بر اساس تخمین هزینه‌ی آنها تا هدف)

$h(n) = h_{\text{SLD}}(n) = \text{straight-line distance from } n \text{ to Bucharest}$

(b) After expanding Arad



جستجوی حریصانه

جستجوی بهترین-اول حریصانه

GREEDY SEARCH (GREEDY BEST-FIRST SEARCH)

گره‌ای را گسترش بدهید که **به نظر می‌رسد** به هدف نزدیک‌تر است.

*Expand the node that **appears** to be closest to goal.*

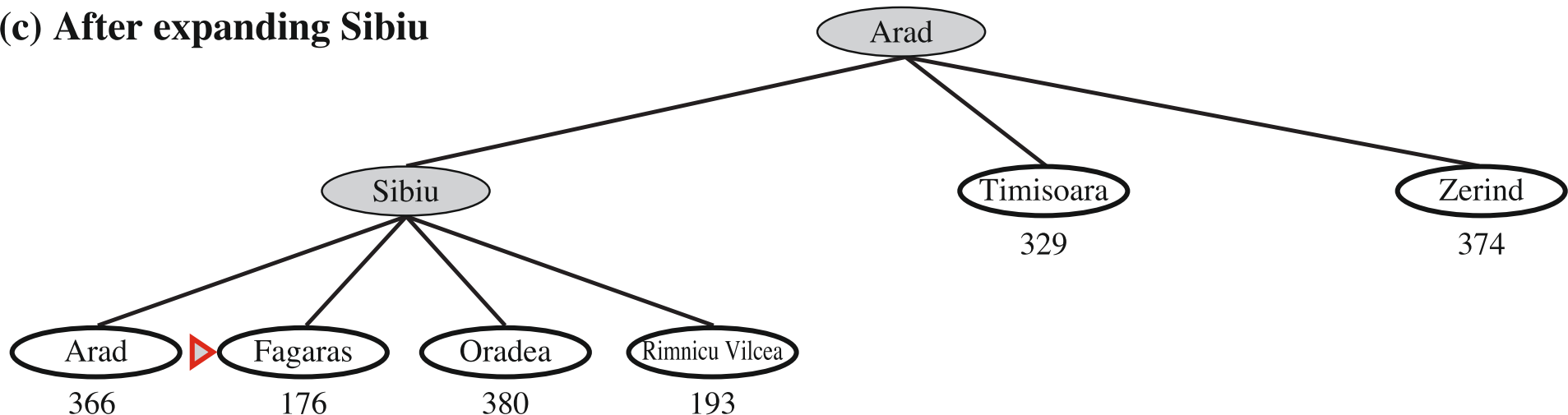
قاعده

GBS

پیاده‌سازی: کناره، یک صف اولویت است (ترتیب گره‌ها بر اساس تخمین هزینه‌ی آنها تا هدف)

$h(n) = h_{\text{SLD}}(n) = \text{straight-line distance from } n \text{ to Bucharest}$

(c) After expanding Sibiu



جستجوی حریصانه

جستجوی بهترین-اول حریصانه

GREEDY SEARCH (GREEDY BEST-FIRST SEARCH)

گره‌ای را گسترش بدهید که به نظر می‌رسد به هدف نزدیک‌تر است.

Expand the node that appears to be closest to goal.

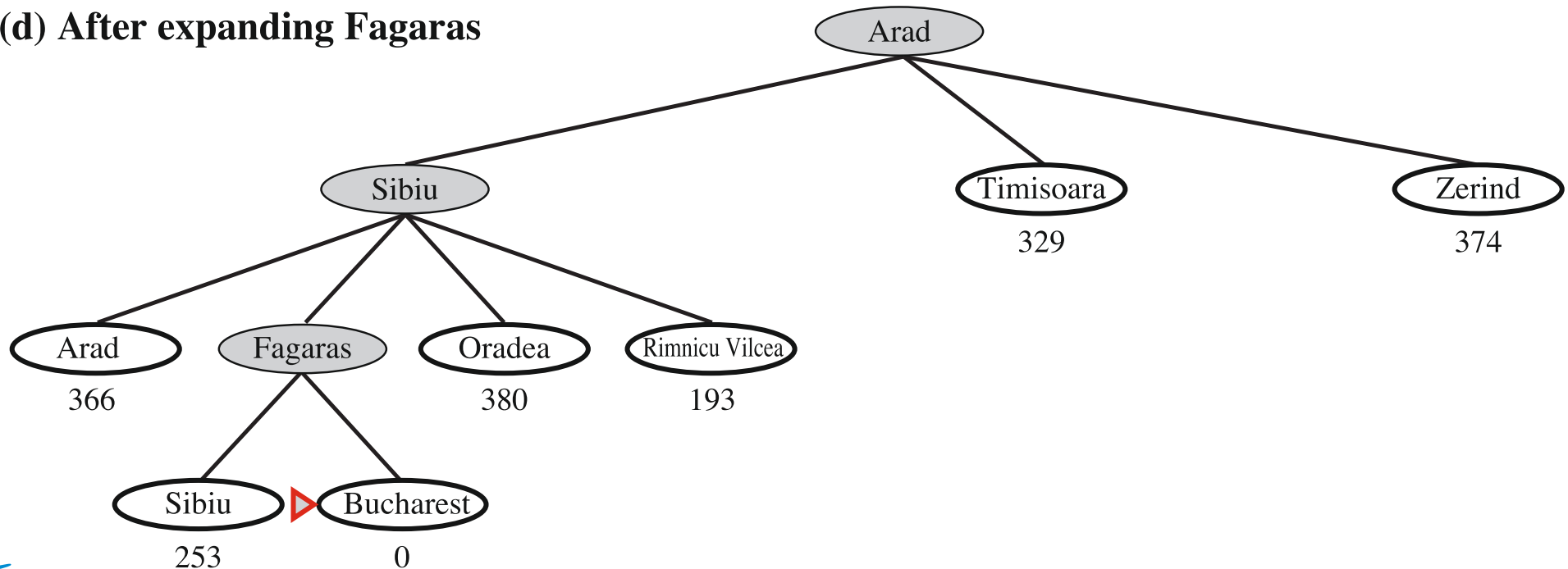
قاعده

GBS

پیاده‌سازی: کناره، یک صف اولویت است (ترتیب گره‌ها بر اساس تخمین هزینه‌ی آنها تا هدف)

$h(n) = h_{\text{SLD}}(n) = \text{straight-line distance from } n \text{ to Bucharest}$

(d) After expanding Fagaras



جستجوی حریصانه

ویژگی‌ها

ارزیابی الگوریتم جستجوی حریصانه			
۱	۲	۳	۴
تمامیت <i>Completeness</i>	بهینگی هزینه <i>Cost Optimality</i>	پیچیدگی زمانی <i>Time Complexity</i>	پیچیدگی فضایی <i>Space Complexity</i>
خیر	خیر	نمایی	نمایی
ممکن است در حلقه‌ی نامتناهی گیر بیفتد، مگر در فضای حالت متناهی با بررسی حالت‌های تکراری	گره‌ی هدف با هزینه‌ی مسیر بدتر، ممکن است زودتر ملاقات شود.	$O(b^m)$ البته یک تابع هیوریستیک خوب می‌تواند بهبود قابل‌توجهی ایجاد کند.	$O(b^m)$ همه‌ی گره‌ها را در حافظه نگه می‌دارد.

فضای جستجو با انتخاب مناسب‌تر $h(n)$ محدودتر می‌شود.

جستجوی A*

A* SEARCH

گره‌ای را گسترش دهید که ارزیابی بهتری دارد.
Expand the best node according to the evaluation function.

قاعده

A*

پیاده‌سازی: کناره، یک صف اولویت است (ترتیب گره‌ها بر اساس مقدار تابع ارزیابی آنها)

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

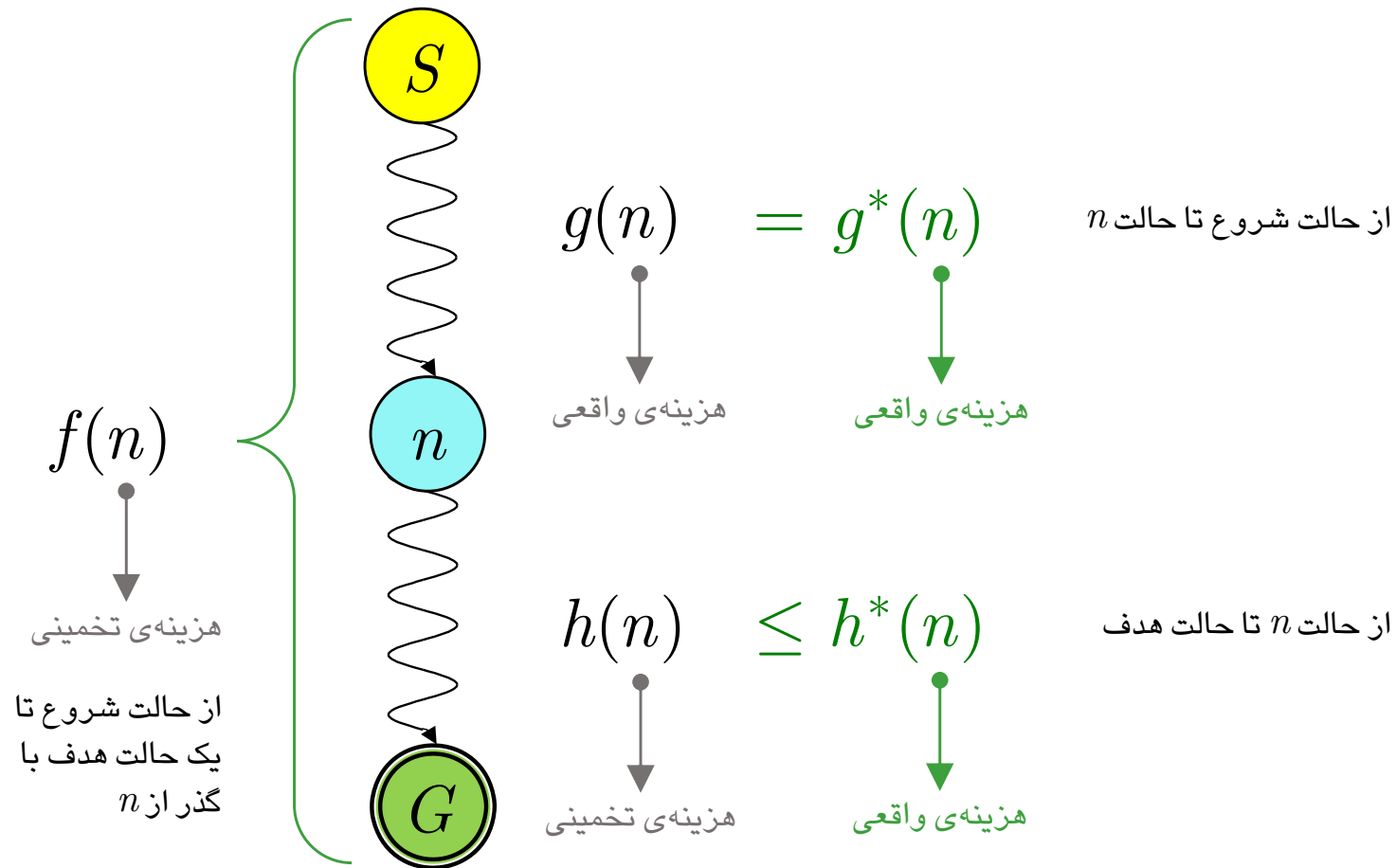
اساس روش: اجتناب از گسترش مسیرهایی که هم‌اکنون گران هستند.

تابع ارزیابی در روش جستجوی A^*

مجموع تابع هزینه‌ی مسیر + تابع هیوریستیک

A^* EVALUATION FUNCTION

هزینه‌ی تخمینی بر اساس ارزان‌ترین مسیر از حالت شروع تا یک حالت هدف با گذر از حالت گره‌ی n



جستجوی A*

A* SEARCH

گره‌ای را گسترش بدهید که ارزیابی بهتری دارد.
Expand the best node according to the evaluation function.

قاعده

A*

پیاده‌سازی: کناره، یک صف اولویت است (ترتیب گره‌ها بر اساس مقدار تابع ارزیابی آنها)

(a) The initial state



$$366 = 0 + 366$$

$$f = g + h$$

جستجوی A*

A* SEARCH

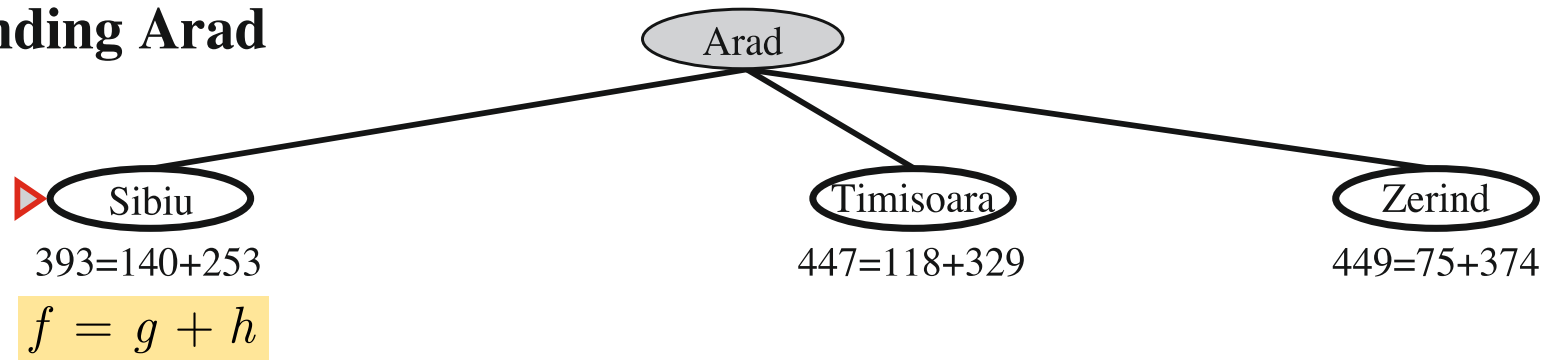
گره‌ای را گسترش بدهید که ارزیابی بهتری دارد.
Expand the best node according to the evaluation function.

قاعده

A*

پیاده‌سازی: کناره، یک صف اولویت است (ترتیب گره‌ها بر اساس مقدار تابع ارزیابی آنها)

(b) After expanding Arad



جستجوی A*

A* SEARCH

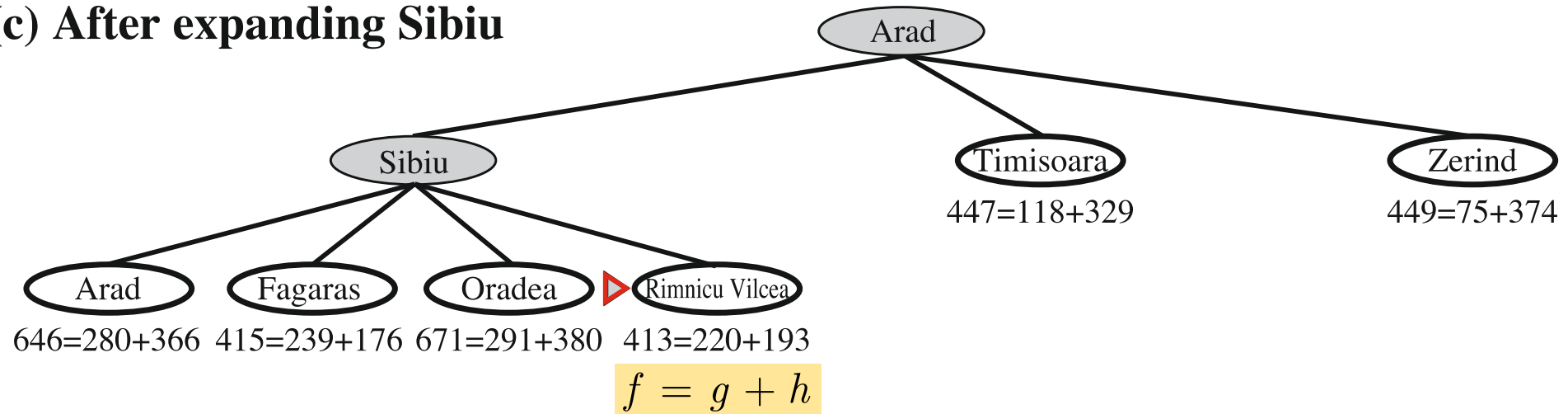
گره‌ای را گسترش بدهید که ارزیابی بهتری دارد.
Expand the best node according to the evaluation function.

قاعده

A*

پیاده‌سازی: کناره، یک صف اولویت است (ترتیب گره‌ها بر اساس مقدار تابع ارزیابی آنها)

(c) After expanding Sibiu



جستجوی A*

A* SEARCH

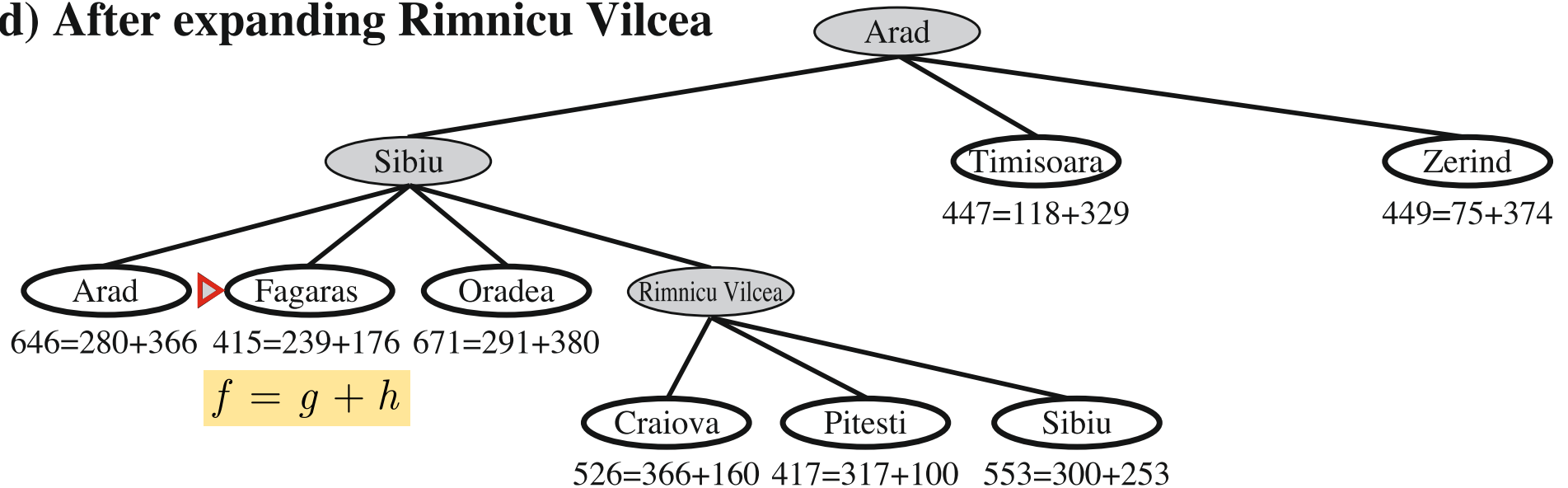
گره‌ای را گسترش بدهید که ارزیابی بهتری دارد.
Expand the best node according to the evaluation function.

قاعده

A*

پیاده‌سازی: کناره، یک صف اولویت است (ترتیب گره‌ها بر اساس مقدار تابع ارزیابی آنها)

(d) After expanding Rimnicu Vilcea



جستجوی A*

A* SEARCH

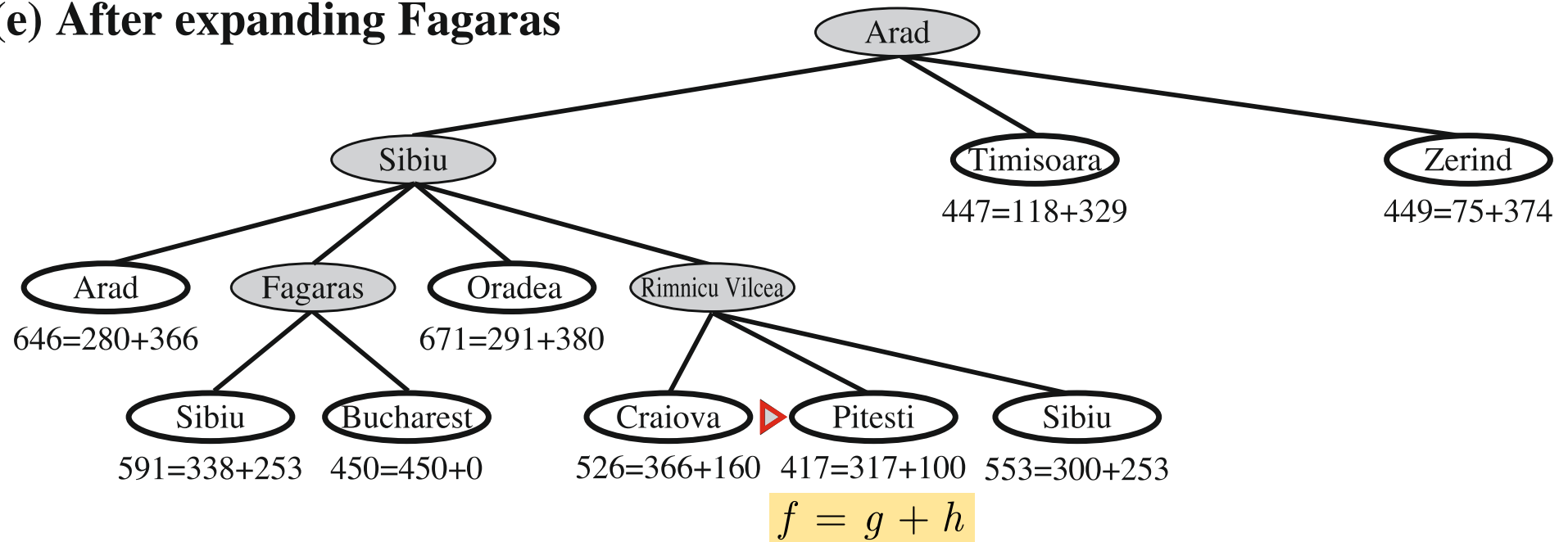
گره‌ای را گسترش بدهید که ارزیابی بهتری دارد.
Expand the best node according to the evaluation function.

قاعده

A*

پیاده‌سازی: کناره، یک صف اولویت است (ترتیب گره‌ها بر اساس مقدار تابع ارزیابی آنها)

(e) After expanding Fagaras



جستجوی A*

A* SEARCH

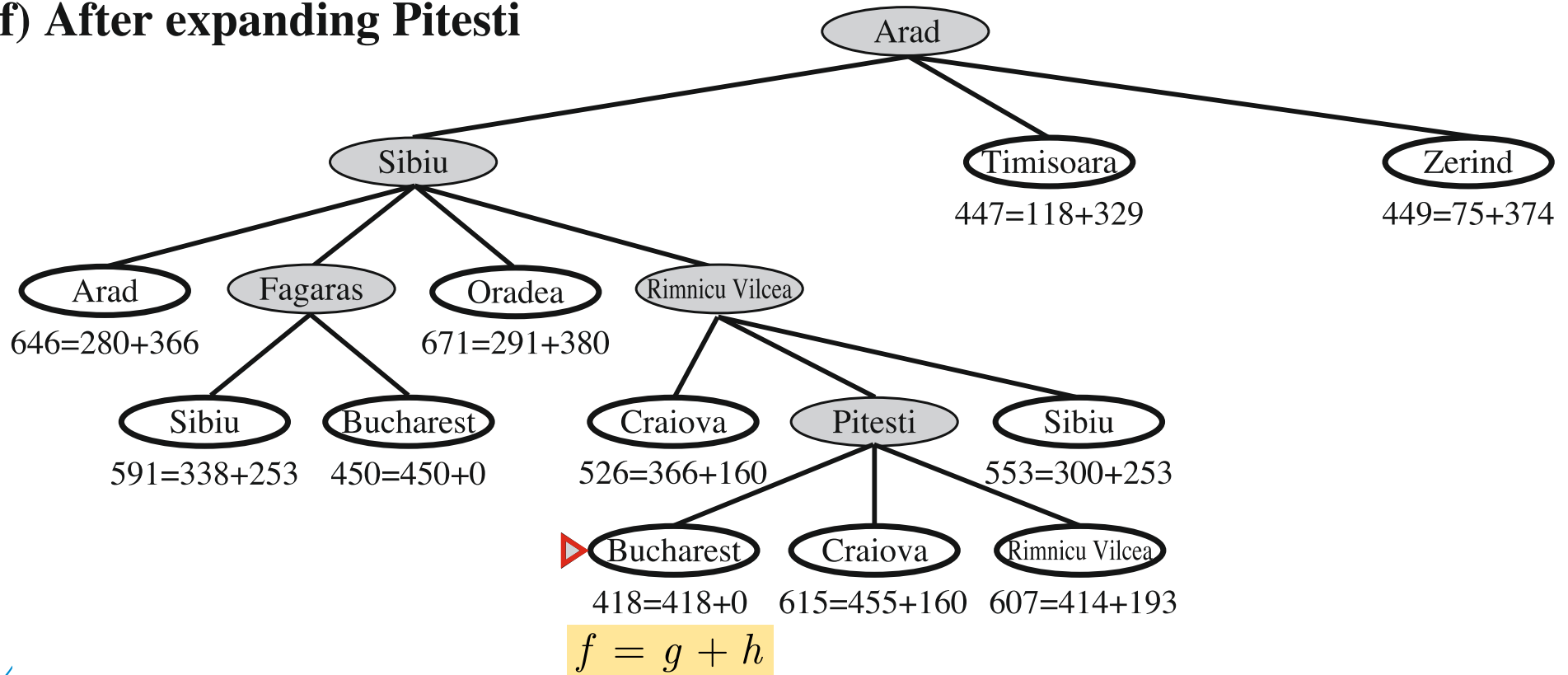
گره‌ای را گسترش بدهید که ارزیابی بهتری دارد.
Expand the best node according to the evaluation function.

قاعده

A*

پیاده‌سازی: کناره، یک صف اولویت است (ترتیب گره‌ها بر اساس مقدار تابع ارزیابی آنها)

(f) After expanding Pitesti



جستجوی A*

ویژگی‌ها

ارزیابی الگوریتم جستجوی A*			
۱	۲	۳	۴
تمامیت <i>Completeness</i>	بهینگی هزینه <i>Cost Optimality</i>	پیچیدگی زمانی <i>Time Complexity</i>	پیچیدگی فضایی <i>Space Complexity</i>
بله	بله	نمایی	نمایی
مگر اینکه بی‌نهایت گره وجود داشته باشد که $f(n) \leq f(G)$ - گره‌ای با $b = \infty$ داریم. - مسیری با هزینه‌ی متناهی اما شامل بی‌نهایت گره داریم.	گره‌های دارای هزینه‌ی بیشتر گسترش داده نمی‌شوند، مگر اینکه گره‌های دارای هزینه‌ی کمتر گسترش داده شده باشند.	$O(b^{\epsilon d})$ $\epsilon = (h^* - h) / h^*$ نمایی بر حسب خطای نسبی در h ضرب در عمق راه‌حل	$O(b^{\epsilon d})$ همه‌ی گره‌ها را در حافظه نگاه می‌دارد.

فضای جستجو با انتخاب مناسب‌تر $h(n)$ محدودتر می‌شود.

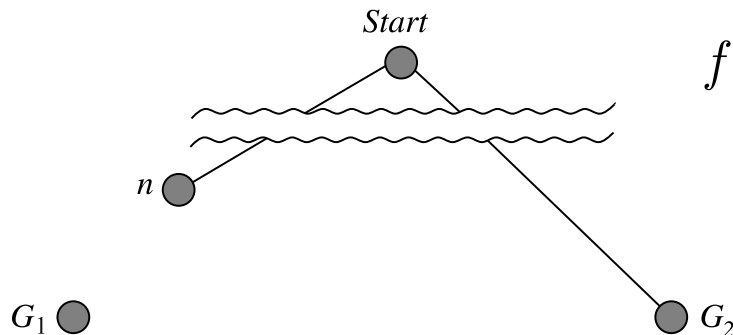
قضایای A^* : بهینه بودن A^*

اثبات قضیه

روش جستجوی A^* بهینه است.

- فرض می‌کنیم یک هدف زیربهینه G_2 تولید شده باشد و در صف کناره واقع باشد.
- فرض کنید n یک گرهی گسترش نیافته بر روی کوتاهترین مسیر به سوی یک هدف بهینه G_1 باشد

در این صورت داریم:



$$\begin{aligned}
 f(G_2) &= g(G_2) && \text{since } h(G_2) = 0 \\
 &> g(G_1) && \text{since } G_2 \text{ is suboptimal} \\
 &\geq g(n) + h(n) && \text{since } h \text{ is admissible} \\
 &= f(n)
 \end{aligned}$$

در نتیجه، A^* هرگز G_2 را برای گسترش انتخاب نمی‌کند، زیرا $f(G_2) > f(n)$.

قضایای A*: بهینه بودن A*

اثبات قضیه

روش جستجوی A* بهینه است.

$$f(n) > C^* \quad (\text{otherwise } n \text{ would have been expanded})$$

$$f(n) = g(n) + h(n) \quad (\text{by definition})$$

$$f(n) = g^*(n) + h(n) \quad (\text{because } n \text{ is on an optimal path})$$

$$f(n) \leq g^*(n) + h^*(n) \quad (\text{because of admissibility, } h(n) \leq h^*(n))$$

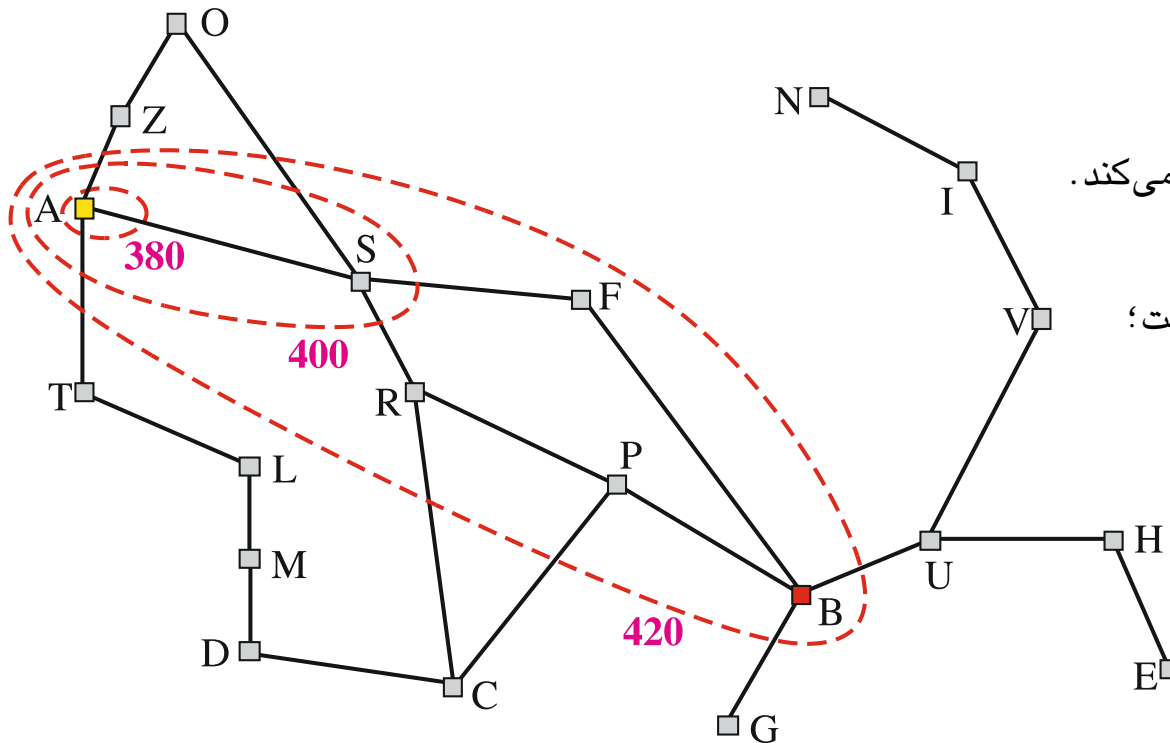
$$f(n) \leq C^* \quad (\text{by definition, } C^* = g^*(n) + h^*(n))$$

The first and last lines form a contradiction, so the supposition that the algorithm could return a suboptimal path must be wrong—it must be that A* returns only cost-optimal paths.

بهینه بودن A^*

کانتورهای f

روش جستجوی A^* گره‌ها را بر حسب افزایش مقدار f گسترش می‌دهد.



به طور تدریجی f -contour ها را اضافه می‌کند.
(در مقایسه با BFS که لایه اضافه می‌کرد.)

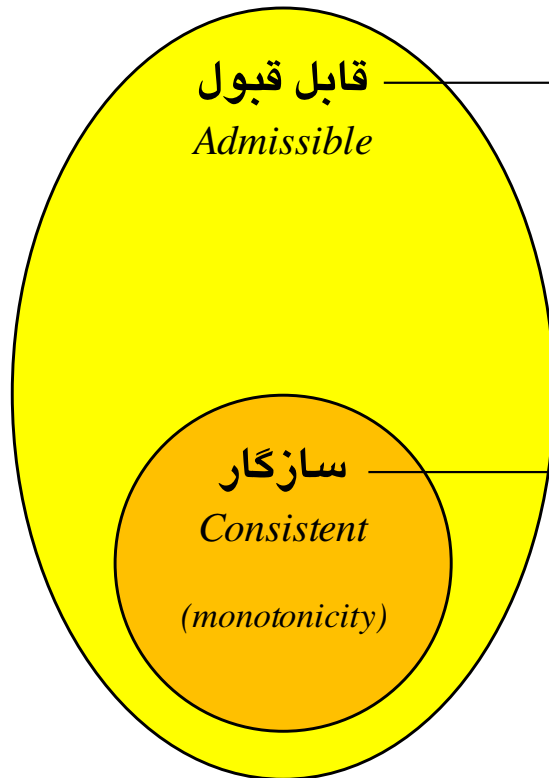
کانتور i شامل همه‌ی گره‌ها با $f = f_i$ است؛
که در آن $f_i \leq f_{i+1}$

- A^* همه‌ی گره‌ها با $f(n) < C^*$ را گسترش می‌دهد. (داخل کانتور جواب بهینه)
- A^* برخی گره‌ها با $f(n) = C^*$ را گسترش می‌دهد. (روی کانتور جواب بهینه)
- A^* هیچ گره‌ای با $f(n) > C^*$ را گسترش نمی‌دهد. (خارج کانتور جواب بهینه)

C^* هزینه‌ی راه‌حل بهینه است.

خواص تابع هیوریستیک

تابع هیوریستیک باید «قابل قبول» باشد و بهتر است که «سازگار» هم باشد.



هیچ گاه «بیش برآورد» overestimate نمی کند.

$$\forall n \quad 0 \leq h(n) \leq h^*(n)$$

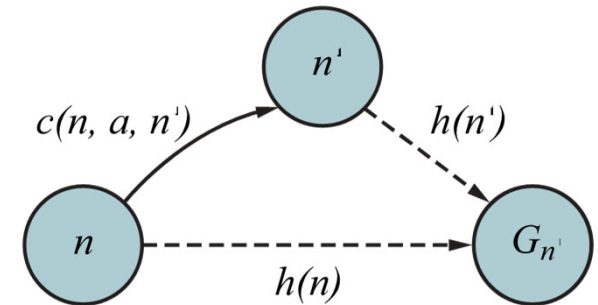
$$\Rightarrow h(G) = 0 \quad (\text{for any goal state } G)$$

از نامساوی مثلثی تبعیت می کند.

$$h(n) \leq c(n, a, n') + h(n')$$

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

$\Rightarrow$ یعنی $f(n)$ در امتداد هر مسیری غیرنزولی است



قضایای A^* : بهینه بودن A^* با توجه به تابع هیوریستیک و الگوریتم عمومی جستجو

قابل قبول
Admissible

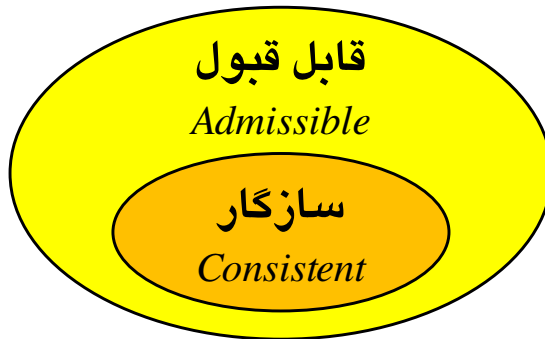
اگر $h(n)$

باشد، الگوریتم جستجوی A^* با استفاده از TREE-SEARCH بهینه است.

سازگار
Consistent

اگر $h(n)$

باشد، الگوریتم جستجوی A^* با استفاده از GRAPH-SEARCH بهینه است.



قابل قبول
Admissible

باشد، آن گاه

سازگار
Consistent

اگر $h(n)$

خواهد بود.

سازگار
Consistent

اگر $h(n)$

باشد، مقدار $f(n)$ در امتداد هر مسیری **غیرنزولی** است (نامساوی مثلثی).

جستجوی ارضاکننده

هیوریستیک‌های غیرقابل قبول و A^* وزن دار

SATISFICING SEARCH: INADMISSIBLE HEURISTICS AND WEIGHTED A^*

روش جستجوی A^* ویژگی‌های خوبی دارد اما گره‌های زیادی را گسترش می‌دهد.

راه حل

می‌توانیم گره‌های کمتری را مورد اکتشاف قرار دهیم (زمان و فضای کمتر)
اگر بخواهیم راه‌های زیربینه اما «به اندازه‌ی کافی خوب» را بپذیریم: راه‌حل‌های ارضاکننده

اگر به جستجوی A^* اجازه بدهیم از یک هیوریستیک غیرقابل قبول استفاده کند (که ممکن است بیش‌برآورد کند)، آن‌گاه ریسک ازدست دادن راه‌حل بهینه را خواهیم داشت اما این هیوریستیک می‌تواند به‌طور بالقوه دقیق‌تر باشد، از طریق کاهش تعداد گره‌های گسترش‌یافته.

For example, road engineers know the concept of a **detour index**, which is a multiplier applied to the straight-line distance to account for the typical curvature of roads. A detour index of 1.3 means that if two cities are 10 miles apart in straight-line distance, a good estimate of the best path between them is 13 miles. For most localities, the detour index ranges between 1.2 and 1.6.

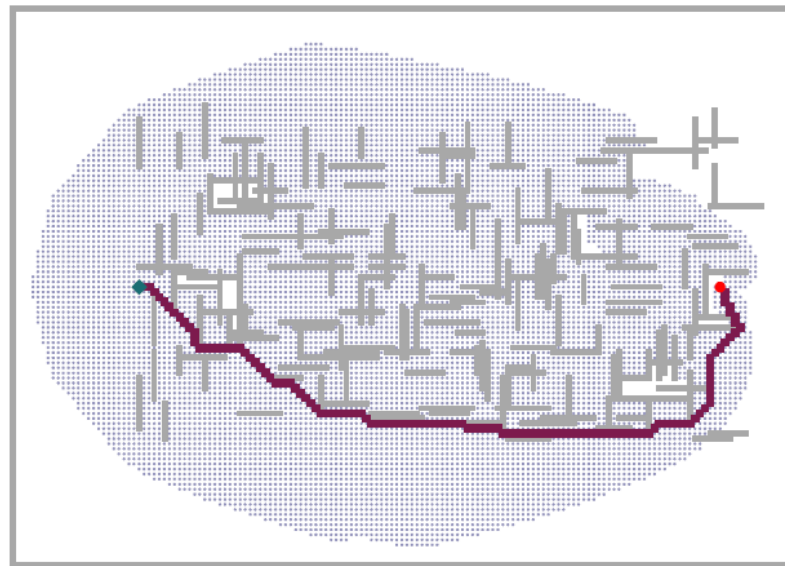
$$f(n) = g(n) + W \times h(n), \text{ for some } W > 1.$$

We can apply this idea to any problem, not just ones involving roads, with an approach called **weighted A^* search** where we weight the heuristic value more heavily, giving us the above evaluation function.

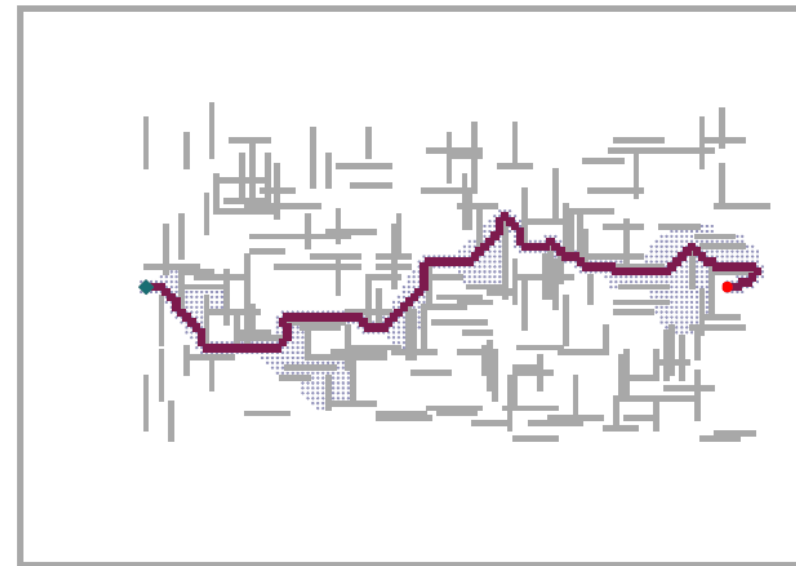
هیوریستیک غیر قابل قبول
Inadmissible Heuristic

جستجوی A^* وزن دار
Weighted A^ Search*

جستجوی ارضاکننده

هیوریستیک‌های غیرقابل قبول و A^* وزن‌دار: مثالSATISFICING SEARCH: INADMISSIBLE HEURISTICS AND WEIGHTED A^* 

(a)



(b)

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.

جستجوی ارضاکننده

جستجوی A^* وزن دار

SATISFICING SEARCH: WEIGHTED A^* SEARCH

We have considered searches that evaluate states by combining g and h in various ways; weighted A^* can be seen as a generalization of the others:

$$A^* \text{ search: } g(n) + h(n) \quad (W = 1)$$

$$\text{Uniform-cost search: } g(n) \quad (W = 0)$$

$$\text{Greedy best-first search: } h(n) \quad (W = \infty)$$

$$\text{Weighted } A^* \text{ search: } g(n) + W \times h(n) \quad (1 < W < \infty)$$

You could call weighted A^* “somewhat-greedy search”: like greedy best-first search, it focuses the search towards a goal; on the other hand, it won’t ignore the path cost completely, and will suspend a path that is making little progress at great cost.

الگوریتم‌های جستجوی زیربینه

SUBOPTIMAL SEARCH ALGORITHMS

الگوریتم‌های جستجوی زیربینه

Suboptimal Search Algorithms

با معیارهایی برای آنچه «به اندازه‌ی کافی خوب» شمرده می‌شود، مشخص می‌شود.

جستجوی زیربینه‌ی کران‌دار

Bounded Suboptimal Search

راه‌حلهایی با تضمین هزینه‌ی بینه با ضریب ثابت W (مانند A^* وزن‌دار)

جستجوی هزینه-بی‌کران

Unbounded-Cost Search

راه‌حلی با هر هزینه‌ای پذیرفته می‌شود، مادامی‌که بتوانیم آن را سریعاً بیابیم.

جستجوی هزینه-کران‌دار

Bounded-Cost Search

راه‌حلی را می‌یابیم که هزینه‌ی آن کمتر از یک ثابت C باشد.

جستجوی سریع

Speedy Search

نسخه‌ای از جستجوی بهترین-اول حریصانه که از تعداد تخمینی کنش‌های لازم برای رسیدن به هدف به‌عنوان هیوریستیک استفاده می‌کند، بدون توجه به هزینه‌ی آن کنش‌ها.

جستجوی هیوریستیک حافظه-محدود

MEMORY-BOUNDED HEURISTIC SEARCH

راه حل

استفاده از روش‌های جستجوی هیوریستیک حافظه-محدود

نقطه ضعف اصلی روش جستجوی A^*

مشکل زمان محاسبه‌ی نمایی نیست،

بلکه مشکل فضای محاسبه‌ی نمایی است:

چون همه‌ی گره‌های تولید شده را در حافظه نگه می‌دارد،
معمولاً پیش از کم آوردن زمان، فضا کم می‌آورد.

جستجوی A^* حافظه-محدود ساده‌شده
Simplified Memory-bounded A^* (SMA*)

جستجوی A^* حافظه-محدود
Memory-bounded A^* (MA*)

جستجوی بهترین-اول بازگشتی
Recursive Best First Search (RBFS)

جستجوی A^* عمیق‌کننده‌ی تکراری
Iterative deepening A^* (IDA*)

جستجوی پرتوی
Beam Search

جستجوی هیوریستیک حافظه-محدود

Memory-bounded Heuristic Search

جستجوی پرتوی

BEAM SEARCH

Beam search limits the size of the frontier.

The easiest approach is

to keep only the k nodes with the best f -scores, discarding any other expanded nodes.

This of course makes the search *incomplete* and *suboptimal*, but we can choose to make *good use of available memory*, and the algorithm *executes fast* because it expands fewer nodes.

For many problems it can find good near-optimal solutions.

You can think of uniform-cost or A* search as spreading out everywhere in concentric contours, and think of beam search as exploring only a focused portion of those contours, the portion that contains the best candidates.

An alternative version of beam search doesn't keep a strict limit on the size of the frontier but instead keeps every node whose f -score is within δ of the best f -score. That way, when there are a few strong-scoring nodes only a few will be kept, but if there are no strong nodes then more will be kept until a strong one emerges.

جستجوی بهترین-اول بازگشتی

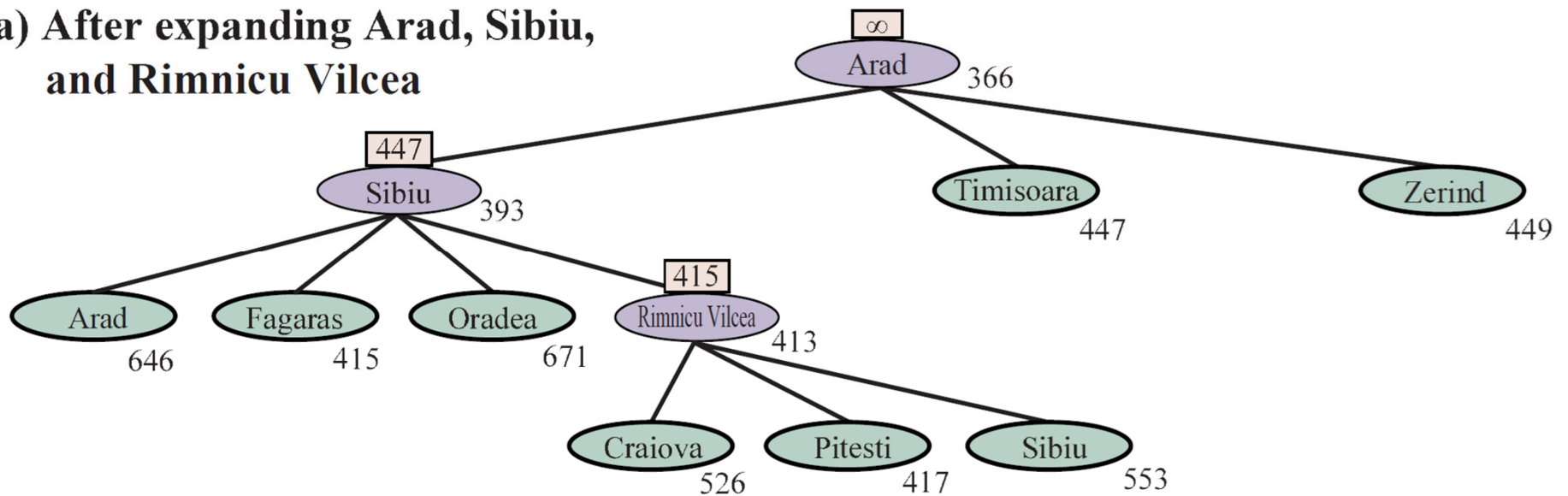
RECURSIVE BEST-FIRST SEARCH

function RECURSIVE-BEST-FIRST-SEARCH(*problem*) **returns** a solution or *failure*
 solution, fvalue $\leftarrow$ RBFS(*problem*, NODE(*problem*.INITIAL), ∞)
return *solution*

function RBFS(*problem, node, f_limit*) **returns** a solution or *failure*, and a new *f*-cost limit
 if *problem*.IS-GOAL(*node*.STATE) **then return** *node*
 successors $\leftarrow$ LIST(EXPAND(*node*))
 if *successors* is empty **then return** *failure, ∞*
 for each *s* **in** *successors* **do** // update *f* with value from previous search
 s.f $\leftarrow$ max(*s*.PATH-COST + *h*(*s*), *node.f*)
 while true do
 best $\leftarrow$ the node in *successors* with lowest *f*-value
 if *best.f* > *f limit* **then return** *failure, best.f*
 alternative $\leftarrow$ the second-lowest *f*-value among *successors*
 result, best.f $\leftarrow$ RBFS(*problem, best, min(f_limit, alternative)*)
 if *result* $\neq$ *failure* **then return** *result, best.f*

جستجوی بهترین-اول بازگشتی

مثال (۱ از ۳)

RECURSIVE BEST-FIRST SEARCH**(a) After expanding Arad, Sibiu, and Rimnicu Vilcea**

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*.

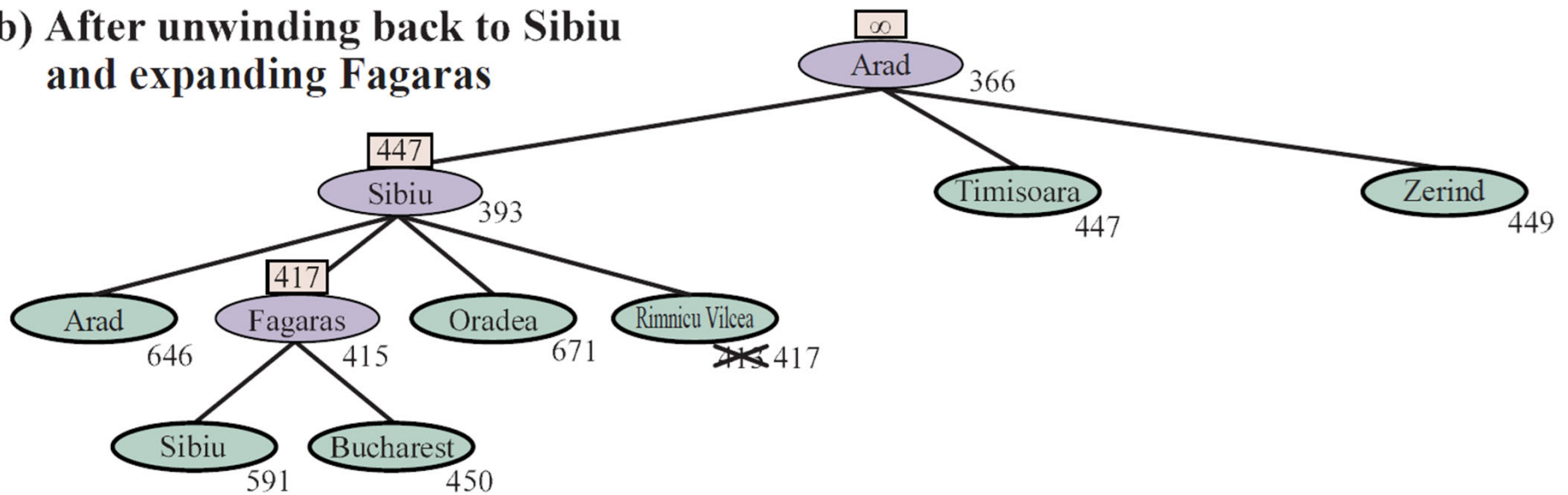
(a) The path via Rimnicu Vilcea is followed until the current best leaf (Pitesti) has a value that is worse than the best alternative path (Fagaras).

جستجوی بهترین-اول بازگشتی

مثال (۲ از ۳)

RECURSIVE BEST-FIRST SEARCH

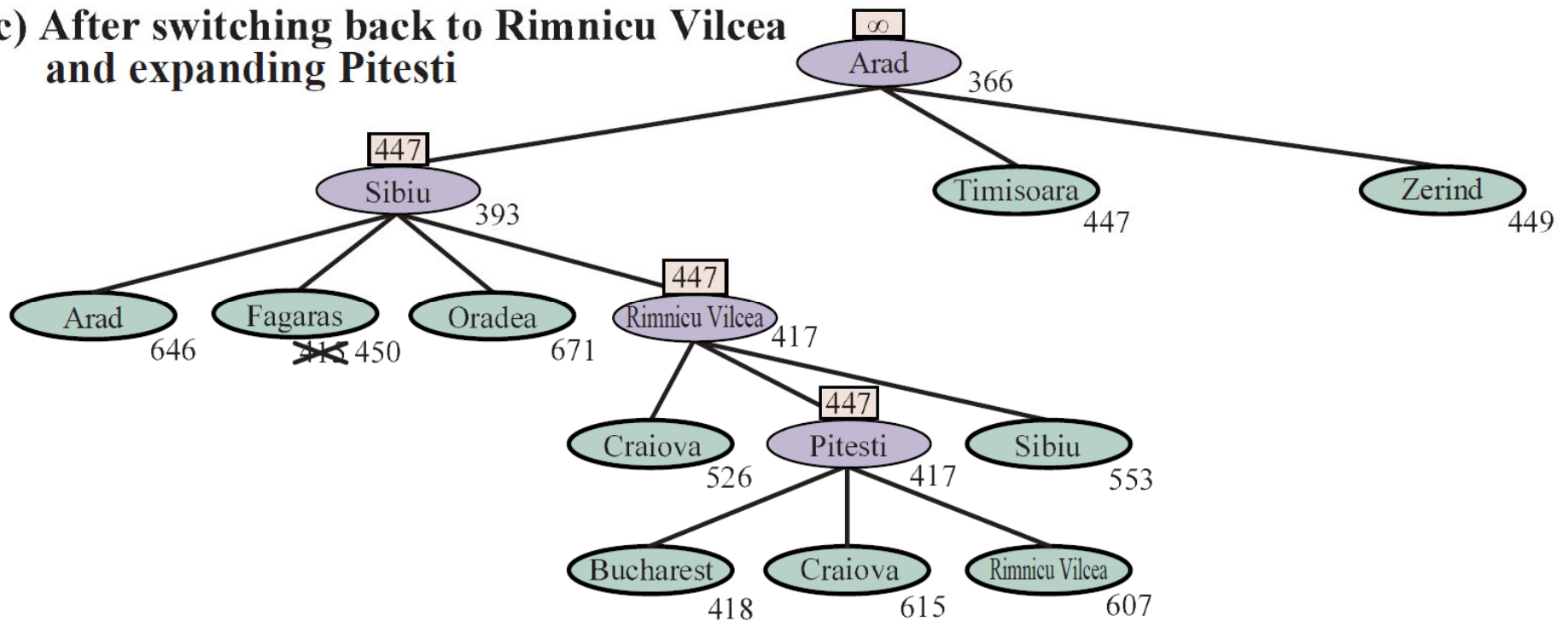
(b) After unwinding back to Sibiu
and expanding Fagaras



(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.

جستجوی بهترین-اول بازگشتی

مثال (۳ از ۳)

RECURSIVE BEST-FIRST SEARCH**(c) After switching back to Rimnicu Vilcea and expanding Pitesti**

(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.

جستجوی هیوریستیک دو طرفه

BIDIRECTIONAL HEURISTIC SEARCH

$$f_F(n) = g_F(n) + h_F(n)$$

برای گره‌های در جهت پیشرو (forward)
حالت آغازین به عنوان ریشه

$$f_B(n) = g_B(n) + h_B(n)$$

برای گره‌های در جهت پسرو (backward)
حالت هدف به عنوان ریشه

$$lb(m, n) = \max(g_F(m) + g_B(n), f_F(m), f_B(n))$$

$$f_2(n) = \max(2g(n), g(n) + h(n))$$

جستجوی هیوریستیک دو طرفه

مثال

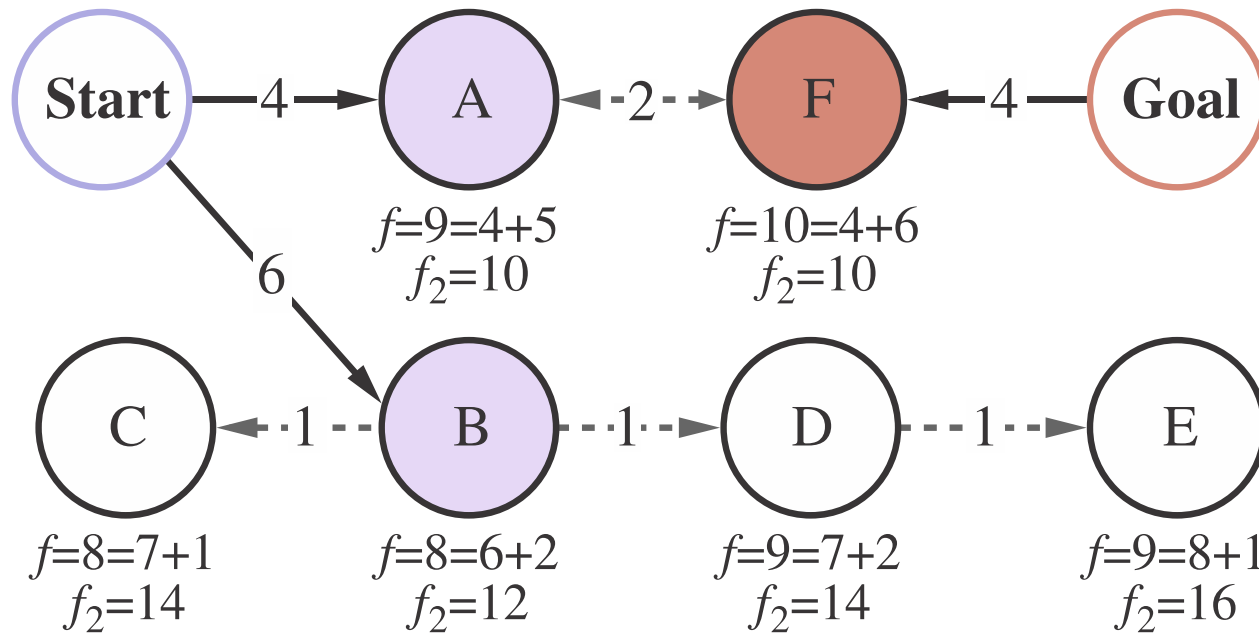
BIDIRECTIONAL HEURISTIC SEARCH

Figure 3.24 Bidirectional search maintains two frontiers: on the left, nodes A and B are successors of Start; on the right, node F is an inverse successor of Goal. Each node is labeled with $f = g + h$ values and the $f_2 = \max(2g, g + h)$ value. (The g values are the sum of the action costs as shown on each arrow; the h values are arbitrary and cannot be derived from anything in the figure.) The optimal solution, Start-A-F-Goal, has cost $C^* = 4 + 2 + 4 = 10$, so that means that a meet-in-the-middle bidirectional algorithm should not expand any node with $g > \frac{C^*}{2} = 5$; and indeed the next node to be expanded would be A or F (each with $g = 4$), leading us to an optimal solution. If we expanded the node with lowest f cost first, then B and C would come next, and D and E would be tied with A, but they all have $g > \frac{C^*}{2}$ and thus are never expanded when f_2 is the evaluation function.

جستجوی هیوریستیک دو طرفه

ویژگی‌ها

BIDIRECTIONAL HEURISTIC SEARCH

Bidirectional search is sometimes more efficient than unidirectional search, sometimes not.

In general, if we have a very good heuristic, then A* search produces search contours that are focused on the goal, and adding bidirectional search does not help much.

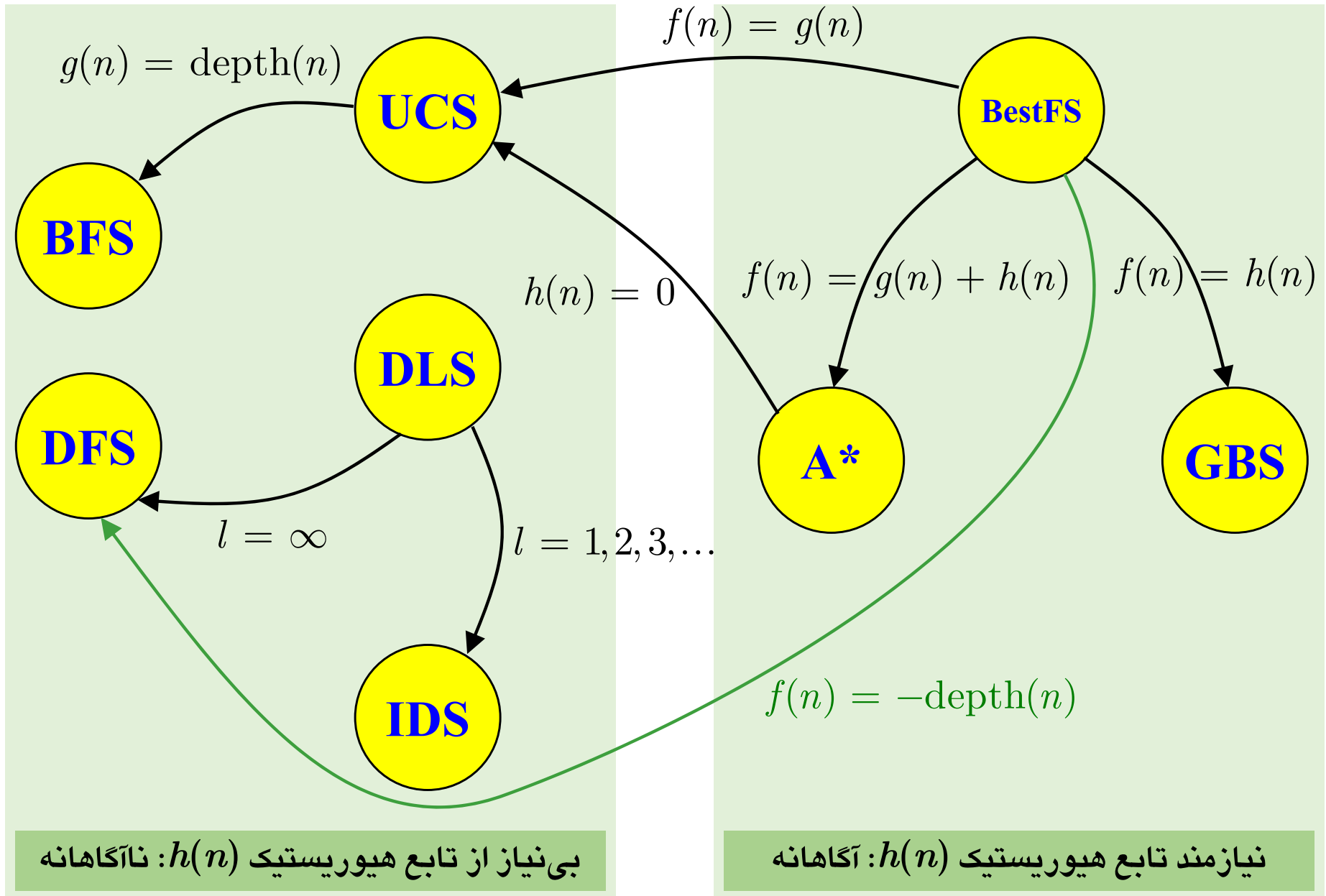
With an average heuristic, bidirectional search that meets in the middle tends to expand fewer nodes and is preferred.

In the worst case of a poor heuristic, the search is no longer focused on the goal, and bidirectional search has the same asymptotic complexity as A*.

Bidirectional search with the f_2 evaluation function and an admissible heuristic h is complete and optimal.

روش‌های جستجوی کلاسیک

روابط میان روش‌ها



۶

توابع هیوریستیک

توابع هیوریستیک

HEURISTIC FUNCTIONS

$$h(n)$$

پرسش‌های کلیدی در مورد تابع هیوریستیک برای یک مسئله خاص

(۱) کدام $h(n)$ بهتر است؟

(۲) چگونه یک $h(n)$ قابل قبول تولید کنیم؟

قواعد انتخاب تابع هیوریستیک: هیوریستیک غالب

(۱) استفاده از مسئله‌های راحت‌شده

(۲) استفاده از یادگیری از تجربه

(۳) استفاده از پایگاه‌داده‌ی الگو

هیوریستیک قابل قبول

مثال: هیوریستیک برای مسئله ۸ معمای

 $h_1(n)$ = تعداد کاشی‌های قرارگرفته در جای نادرست

برای مسئله ۸ معمای

 $h_2(n)$ = مجموع فاصله‌ی شهری (Manhattan) هر کاشی تا جای درست خود

5	4	
6	1	8
7	3	2

Start State

1	2	3
8		4
7	6	5

Goal State

$$h_1(S) = 6$$

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

فاکتور انشعاب مؤثر

روشی برای تشخیص کیفیت یک هیوریستیک

EFFECTIVE BRANCHING FACTOR

فاکتور انشعاب مؤثر

$$b^*$$

اگر تعداد گره‌های تولید شده توسط A^* برای یک مسئله‌ی خاص N باشد و عمق راه‌حل برابر با d باشد،
 b^* برابر است با فاکتور انشعاب یک درخت یکنواخت به عمق d با $N + 1$ گره

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

مقدار ایده‌آل b^* برابر با 1 است.

$$\begin{bmatrix} d = 5 \\ N = 52 \\ b^* = 1.92 \end{bmatrix}$$

مثال

b^* برای نمونه‌های مختلف مسئله می‌تواند متغیر باشد،
 اما معمولاً برای نمونه‌های به‌اندازه‌ی کافی دشوار مسائل نسبتاً ثابت است.

فاکتور انشعاب مؤثر

مثال: مسئله‌ی معمایی ۸

d	Search Cost (nodes generated)			Effective Branching Factor		
	IDS	$A^*(h_1)$	$A^*(h_2)$	IDS	$A^*(h_1)$	$A^*(h_2)$
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	—	539	113	—	1.44	1.23
16	—	1301	211	—	1.45	1.25
18	—	3056	363	—	1.46	1.26
20	—	7276	676	—	1.47	1.27
22	—	18094	1219	—	1.48	1.28
24	—	39135	1641	—	1.48	1.26

$$h_1(n) = \text{تعداد کاشی‌های قرارگرفته در جای نادرست}$$

$$h_2(n) = \text{مجموع فاصله‌ی شهری (Manhattan) هر کاشی تا جای درست خود}$$

5	4	
6	1	8
7	3	2

Start State

1	2	3
8		4
7	6	5

Goal State

غلبه

غلبه‌ی یک هیوریستیک بر یک هیوریستیک دیگر

DOMINANCE

اگر برای دو هیوریستیک قابل قبول h_1 و h_2 داشته باشیم

$$\forall n \ h_1(n) \geq h_2(n)$$

می‌گوییم h_1 بر h_2 غلبه می‌کند.

هیوریستیک **غالب** (بزرگ‌تر) برای جستجو بهتر است.

میزان اطلاعات بیشتری دارد.

Given any admissible heuristics h_a, h_b ,

$$h(n) = \max(h_a(n), h_b(n))$$

is also admissible and dominates h_a, h_b

تولید هیوریستیک قابل قبول

چند قاعده برای انتخاب یک هیوریستیک مناسب

قواعد انتخاب هیوریستیک قابل قبول $h(n)$

$h(n)$ نباید بیش برآورد (overestimate) کند.

$h(n)$ باید تا جای ممکن بزرگ باشد.

$h(n)$ باید از نظر محاسباتی ارزان باشد.

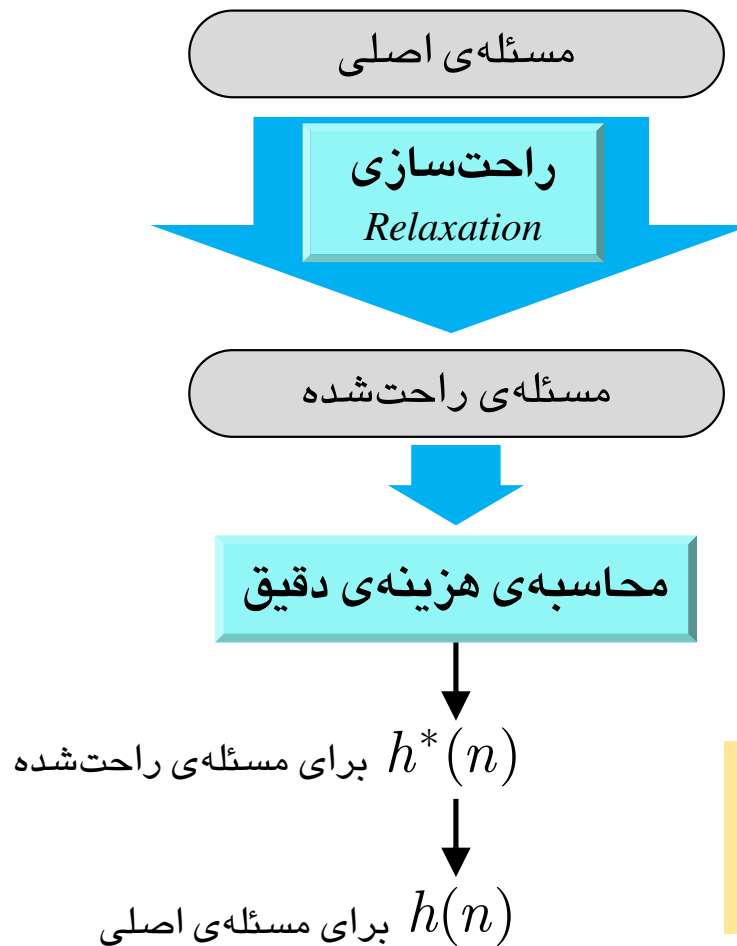
$$0 \leq h(n) \leq h^*(n)$$

تولید هیوریستیک قابل قبول

مسئله‌های راحت شده

RELAXED PROBLEMS

استخراج هیوریستیک قابل قبول با
استفاده از هزینه‌ی دقیق راه حل یک نسخه‌ی راحت‌تر شده (relaxed) از مسئله‌ی اصلی



نکته‌ی کلیدی:

هزینه‌ی راه حل بهینه‌ی یک مسئله‌ی راحت شده
نباید از هزینه‌ی راه حل بهینه‌ی مسئله‌ی اصلی بیشتر باشد.

تولید هیوریستیک قابل قبول

مسئله‌های راحت شده: مثال (مسئله‌ی معمای ۸): (۱)

5	4	
6	1	8
7	3	2

Start State

1	2	3
8		4
7	6	5

Goal State

یک کاشی می‌تواند به خانه‌ی خالی مجاور جابجا شود.

مسئله‌ی اصلی

راحت‌سازی
Relaxation

یک کاشی می‌تواند به هر جایی جابجا شود.

مسئله‌ی راحت شده

محاسبه‌ی هزینه‌ی دقیق

هزینه‌ی دقیق هر حالت تا هدف:
 = تعداد کاشی‌های قرارگرفته در جای نادرست
 (تک تک کاشی‌های نادرست باید به جای درست خود منتقل شوند)

 $h^*(n)$ برای مسئله‌ی راحت شده

$h(n)$ برای مسئله‌ی اصلی
 $h_1(n)$ = تعداد کاشی‌های قرارگرفته در جای نادرست

تولید هیوریستیک قابل قبول

مسئله‌های راحت شده: مثال (مسئله‌ی معمای ۸): (۲)

5	4	
6	1	8
7	3	2

Start State

1	2	3
8		4
7	6	5

Goal State

یک کاشی می‌تواند به خانه‌ی خالی مجاور جابجا شود.

مسئله‌ی اصلی

راحت‌سازی
Relaxation

یک کاشی می‌تواند به هر خانه‌ی مجاور جابجا شود.

مسئله‌ی راحت شده

محاسبه‌ی هزینه‌ی دقیق

هزینه‌ی دقیق هر حالت تا هدف:

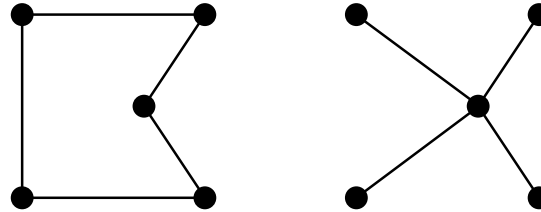
= مجموع فاصله‌ی شهری (Manhattan) هر کاشی تا جای درست خود
(هر کاشی نادرست به اندازه‌ی فاصله‌ی خود تا جای درست نیاز به جابجایی دارد.)

 $h^*(n)$ برای مسئله‌ی راحت شده

$h_2(n)$ = مجموع فاصله‌ی شهری (Manhattan) هر کاشی تا جای درست خود
 $h(n)$ برای مسئله‌ی اصلی

تولید هیوریستیک قابل قبول

مسئله‌های راحت شده: مثال (مسئله‌ی فروشنده‌ی دوره‌گرد: TSP)



کوتاه‌ترین دور گذرنده از همه‌ی شهرها، هر کدام فقط یک بار

مسئله‌ی اصلی

راحت‌سازی
Relaxation

کوتاه‌ترین مسیر گذرنده از همه‌ی شهرها، هر کدام به هر تعداد

مسئله‌ی راحت شده

محاسبه‌ی هزینه‌ی دقیق

هزینه‌ی دقیق هر حالت تا هدف:
 $O(n^2)$
درخت پوشای کمینه (minimal spanning tree: MST)
= مجموع وزن یال‌های درخت پوشای کمینه
(MST کم‌ترین هزینه‌ی اتصال همه‌ی رأس‌های گراف (شهرها) را برمی‌گرداند.)

$h^*(n)$ برای مسئله‌ی راحت شده

$h(n)$ برای مسئله‌ی اصلی = مجموع وزن یال‌های درخت پوشای کمینه

تولید هیوریستیک قابل قبول

یادگیری هیوریستیک‌ها از تجربه

LEARNING HEURISTICS FROM EXPERIENCE

یادگیری هیوریستیک $h(n)$ از روی تجربه
(تجربه: به معنی حل یک مسئله به دفعات است)

استفاده از یادگیری استقرائی
برای ساخت تابع $h(n)$
می‌تواند هزینه‌های راه‌حل را برای سایر حالت‌هایی که در حین جستجو ظاهر می‌شوند، پیش‌بینی کند.
با استفاده از تکنیک‌هایی چون:
شبکه‌های عصبی، درخت‌های تصمیم، یادگیری تقویتی، ...

با داشتن ویژگی‌های (feature) مناسب:

$$x_1, x_2, \dots, x_k$$

تابع هیوریستیک به صورت عمومی زیر در نظر گرفته می‌شود:

$$h(n) = c_1 x_1(n) + c_2 x_2(n) + \dots + c_k x_k(n)$$

روش یادگیری، چگونگی تعیین ضرایب ویژگی‌ها را مشخص می‌کند.

تولید هیوریستیک قابل قبول

یادگیری هیوریستیک‌ها از تجربه: مثال (مسئله‌ی معمایی ۸)

5	4	
6	1	8
7	3	2

Start State

1	2	3
8		4
7	6	5

Goal State

ویژگی‌های (feature) زیر در نظر گرفته می‌شوند:

x_1 = تعداد کاشی‌های قرارگرفته در جای نادرست

x_2 = تعداد جفت کاشی‌های مجاور که در حالت هدف نیز مجاور هستند.

تابع هیوریستیک به صورت عمومی زیر در نظر گرفته می‌شود:

$$h(n) = c_1 x_1(n) + c_2 x_2(n)$$

روش یادگیری، چگونگی تعیین ضرایب ویژگی‌ها را مشخص می‌کند.

تولید هیوریستیک قابل قبول

استفاده از پایگاه داده‌ی الگو

PATTERN DATABASE

استخراج هیوریستیک $h(n)$ از روی هزینه‌ی راه‌حل یک زیرمسئله از مسئله‌ی داده‌شده

ایده:

این هزینه‌های راه‌حل بهینه برای هر نمونه زیرمسئله‌ی ممکن ذخیره می‌شود.

چون هزینه‌ی راه‌حل بهینه‌ی زیرمسئله، کران پایینی برای هزینه‌ی مسئله‌ی کامل است.

مقدار تابع هیوریستیک برای هر حالت کاملی که در روند جستجو بدان برخورد می‌کنیم، با مراجعه به پیکربندی زیرمسئله‌ی متناظر در پایگاه داده محاسبه می‌شود.

ساخت پایگاه داده‌ی الگو

با جستجوی پس‌رو از حالت هدف به سمت عقب و ثبت هزینه‌ی هر الگوی جدیدی که با آن مواجه می‌شویم

تولید هیوریستیک قابل قبول

استفاده از پایگاه داده‌ی الگو (مثال)

تقریباً ۱۰۰۰ برابر
افزایش کارایی در حل
مسئله‌ی معمای ۱۵-
با استفاده از این روش

7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

Goal State

هزینه‌ی راه‌حل
بهینه‌ی زیرمسئله،
کران پایینی برای
هزینه‌ی مسئله‌ی
کامل است.

*	2	4
*		*
*	3	1

Start State

	1	2
3	4	*
*	*	*

Goal State

Figure 3.30 A subproblem of the 8-puzzle instance given in Figure 3.28. The task is to get tiles 1, 2, 3, and 4 into their correct positions, without worrying about what happens to the other tiles.

تولید هیوریستیک با لندمارک‌ها

GENERATING HEURISTICS WITH LANDMARKS

$$h_L(n) = \min_{L \in Landmarks} C^*(n, L) + C^*(L, goal)$$

تولید هیوریستیک با لندمارک‌ها

GENERATING HEURISTICS WITH LANDMARKS

7 h 17 min (578 km)

via A1 and E81

Fastest route, the usual traffic

Arad

Romania

> Get on A1 from Strada Andrei Șaguna, Strada Dorobanți and Calea Bodrogului/DJ682F
8 min (4.3 km)

> Get on DN68A/E673 in Județul Timiș from A1
53 min (101 km)

> Get on A1 in Șolmuş from DN68A/E673 and DN7/E68
1 h 13 min (75.7 km)

> Follow A1, DN1/DN7/E68/E81 and A1 to DN1/DN7/E68/E81 in Județul Sibiu. Take the DN7/DN1 exit from A1
1 h 38 min (131 km)

> Continue to Râmnicu Vâlcea
1 h 21 min (86.4 km)

↑ Continue onto DN7/E81
Continue to follow E81
2 h 15 min (177 km)

> Take Splaiul Independenței to Bulevardul Unirii/E81
9 min (3.5 km)

Bucharest

Romania

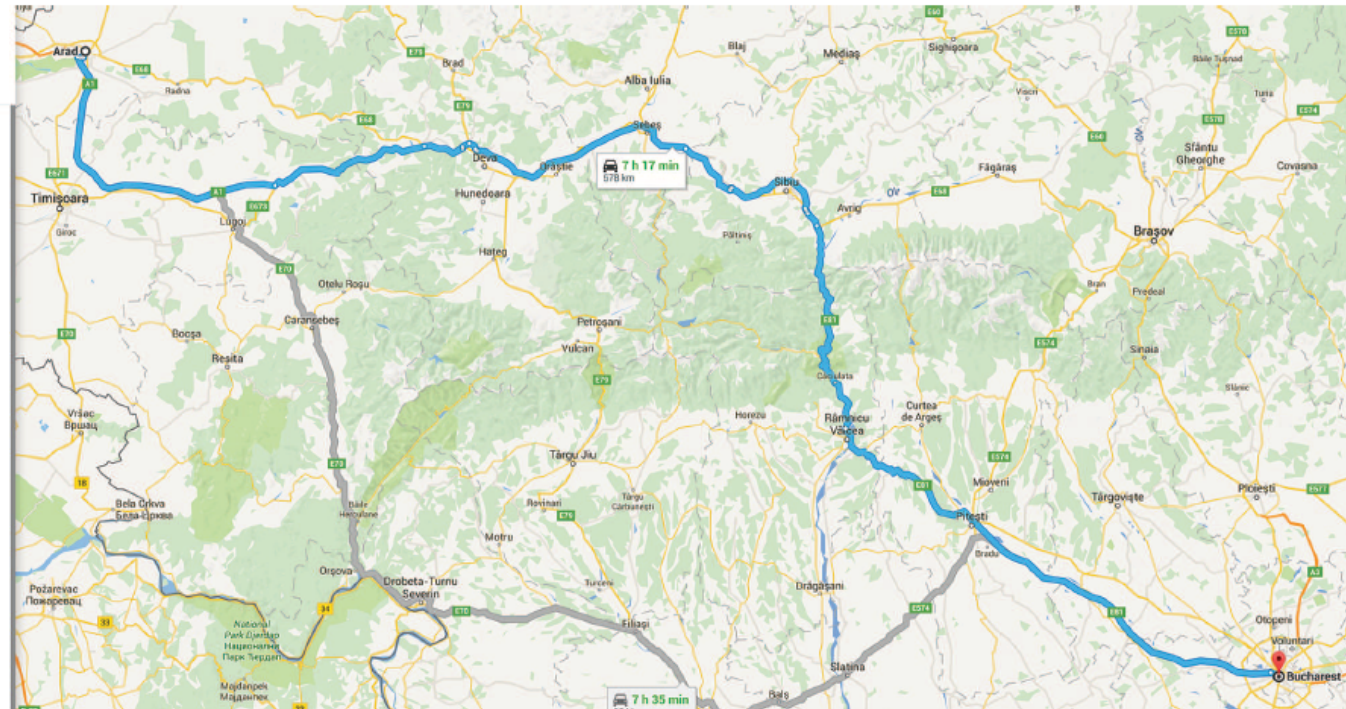


Figure 3.28 A Web service providing driving directions, computed by a search algorithm.

تولید هیوریستیک با لندمارک‌ها

GENERATING HEURISTICS WITH LANDMARKS

$$h_{DH}(n) = \max_{L \in Landmarks} |C^*(n, L) - C^*(goal, L)|$$

یادگیری برای جستجوی بهتر

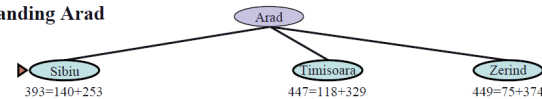
استفاده از فضای حالت فراسطحی

LEARNING TO SEARCH BETTER

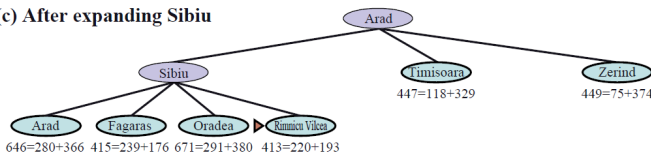
(a) The initial state



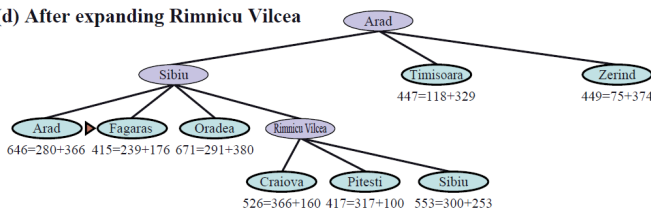
(b) After expanding Arad



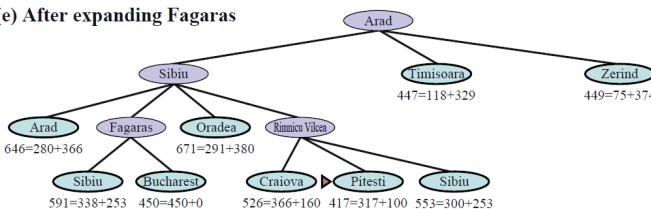
(c) After expanding Sibiu



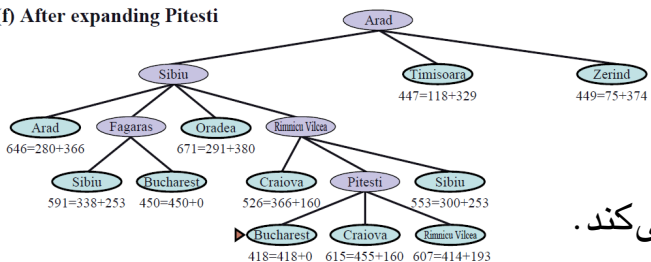
(d) After expanding Rimnicu Vilcea



(e) After expanding Fagaras



(f) After expanding Pitesti



فضای حالت در سطح شیء
Object-level State Space

حالت، وضعیت فعلی در درخت جستجو است.
کنش، عملیاتی مجاز بر روی حالت فعلی است.

فضای حالت فراسطحی
Metalevel State Space

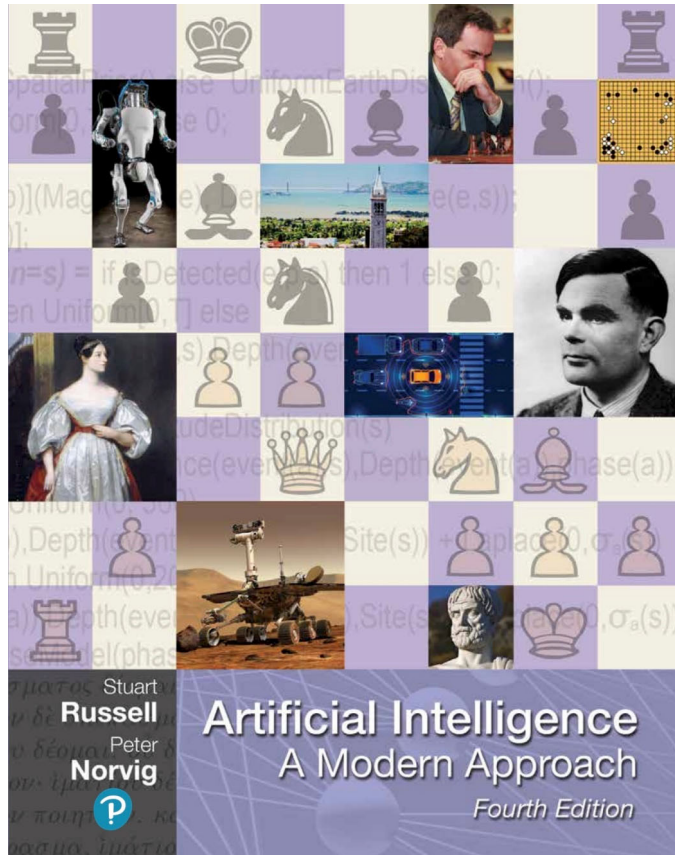
حالت، کل درخت جستجوی فعلی است.
کنش، عملیاتی است که در درخت جستجو تغییری ایجاد می‌کند.

حل مسئله با جستجو

۷

منابع،
مطالعه،
تکلیف

منبع اصلی



Stuart Russell and Peter Norvig,
Artificial Intelligence: A Modern Approach,
 4th Edition, Prentice Hall, 2020.

Chapter 3

CHAPTER 3

SOLVING PROBLEMS BY SEARCHING

In which we see how an agent can look ahead to find a sequence of actions that will eventually achieve its goal.

When the correct action to take is not immediately obvious, an agent may need to *plan ahead*: to consider a *sequence* of actions that form a path to a goal state. Such an agent is called a **problem-solving agent**, and the computational process it undertakes is called **search**.

Problem-solving agent
Search

Problem-solving agents use **atomic** representations, as described in Section 2.4.7—that is, states of the world are considered as wholes, with no internal structure visible to the problem-solving algorithms. Agents that use **factored** or **structured** representations of states are called **planning agents** and are discussed in Chapters 7 and 11.

We will cover several search algorithms. In this chapter, we consider only the simplest environments: episodic, single agent, fully observable, deterministic, static, discrete, and known. We distinguish between **informed** algorithms, in which the agent can estimate how far it is from the goal, and **uninformed** algorithms, where no such estimate is available. Chapter 4 relaxes the constraints on environments, and Chapter 5 considers multiple agents.

This chapter uses the concepts of asymptotic complexity (that is, $O(n)$ notation). Readers unfamiliar with these concepts should consult Appendix A.

3.1 Problem-Solving Agents

Imagine an agent enjoying a touring vacation in Romania. The agent wants to take in the sights, improve its Romanian, enjoy the nightlife, avoid hangovers, and so on. The decision problem is a complex one. Now, suppose the agent is currently in the city of Arad and has a nonrefundable ticket to fly out of Bucharest the following day. The agent observes street signs and sees that there are three roads leading out of Arad: one toward Sibiu, one to Timisoara, and one to Zerind. None of these are the goal, so unless the agent is familiar with the geography of Romania, it will not know which road to follow.¹

If the agent has no additional information—that is, if the environment is **unknown**—then the agent can do no better than to execute one of the actions at random. This sad situation is discussed in Chapter 4. In this chapter, we will assume our agents always have access to information about the world, such as the map in Figure 3.1. With that information, the agent can follow this four-phase problem-solving process:

- **Goal formulation:** The agent adopts the **goal** of reaching Bucharest. Goals organize behavior by limiting the objectives and hence the actions to be considered.

Goal formulation

¹ We are assuming that most readers are in the same position and can easily imagine themselves to be as clueless as our agent. We apologize to Romanian readers who are unable to take advantage of this pedagogical device.