هوش مصنوعی

فصل ۳

حل مسئله با جستجو (۱)

Solving Problems by Searching (1)

کاظم فولادی قلعه

دانشکده مهندسی، پردیس فارابی

دانشگاه تهران

http://courses.fouladi.ir/ai

هوش مصنوعی

حل مسئله با جستجو

۱

عامل‌های حل مسئله

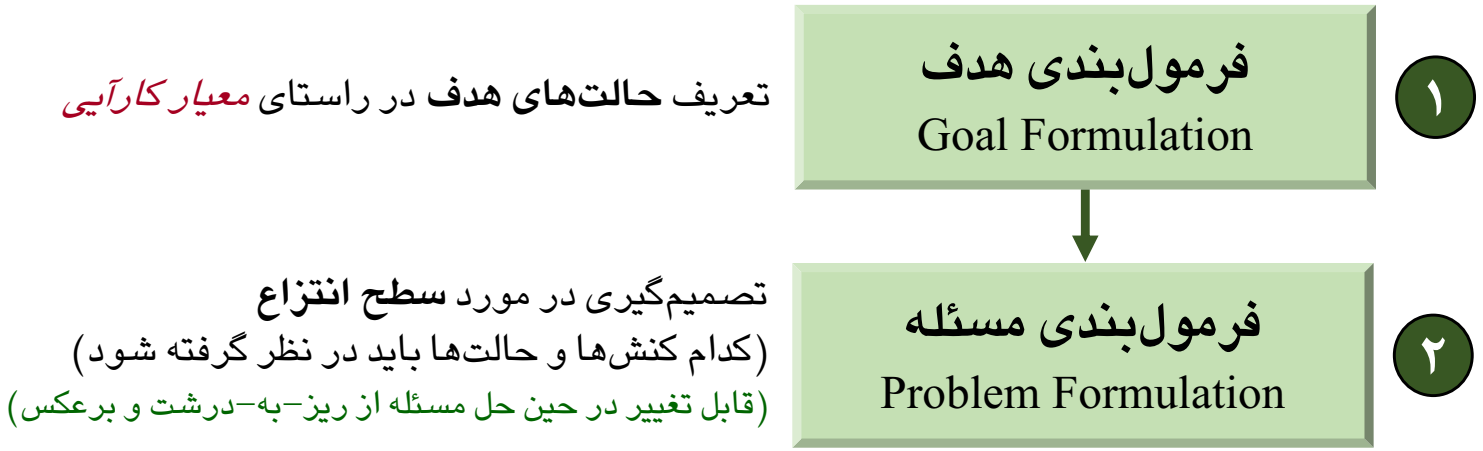
عامل حل مسئله به عنوان نوعی از عامل مبتنی بر هدف

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	
ناآگاهانه *Uninformed*	**آگاهانه** *Informed*
عامل بجز تعریف مسئله، هیچ اطلاعاتی در مورد مسئله‌ی تحت بررسی خود ندارد.	عامل در مورد اینکه کجا به دنبال راه‌حل مسئله بگردد، کم و بیش راهنماهایی دارد.

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*	نیمه‌پویا *Semidynamic*	پویا *Dynamic*
گسسته *Discrete*		پیوسته *Continuous*
شناخته‌شده *Known*		ناشناخته *Unknown*

⇐ راه‌حل هر مسئله، **دنباله‌ی ثابتی از کنش‌ها**ست.

راه‌حل بدون توجه به ادراک‌های ورودی در هر گام اجرا می‌شود (اجرای چشم‌بسته):

سیستم حلقه‌باز (open-loop)

مثال: مسئله‌ی رومانی

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

۱ فرمول‌بندی هدف
Goal Formulation

بودن در Bucharest

۲ فرمول‌بندی مسئله
Problem Formulation

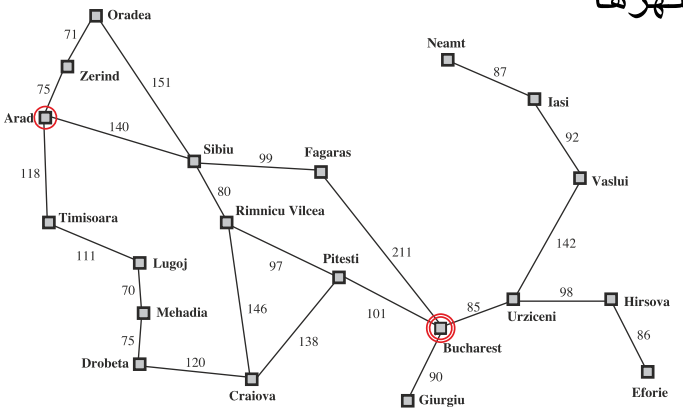
حالت‌ها: شهرهای مختلف
کنش‌ها: رانندگی بین شهرها

۳ جستجو
Search

؟

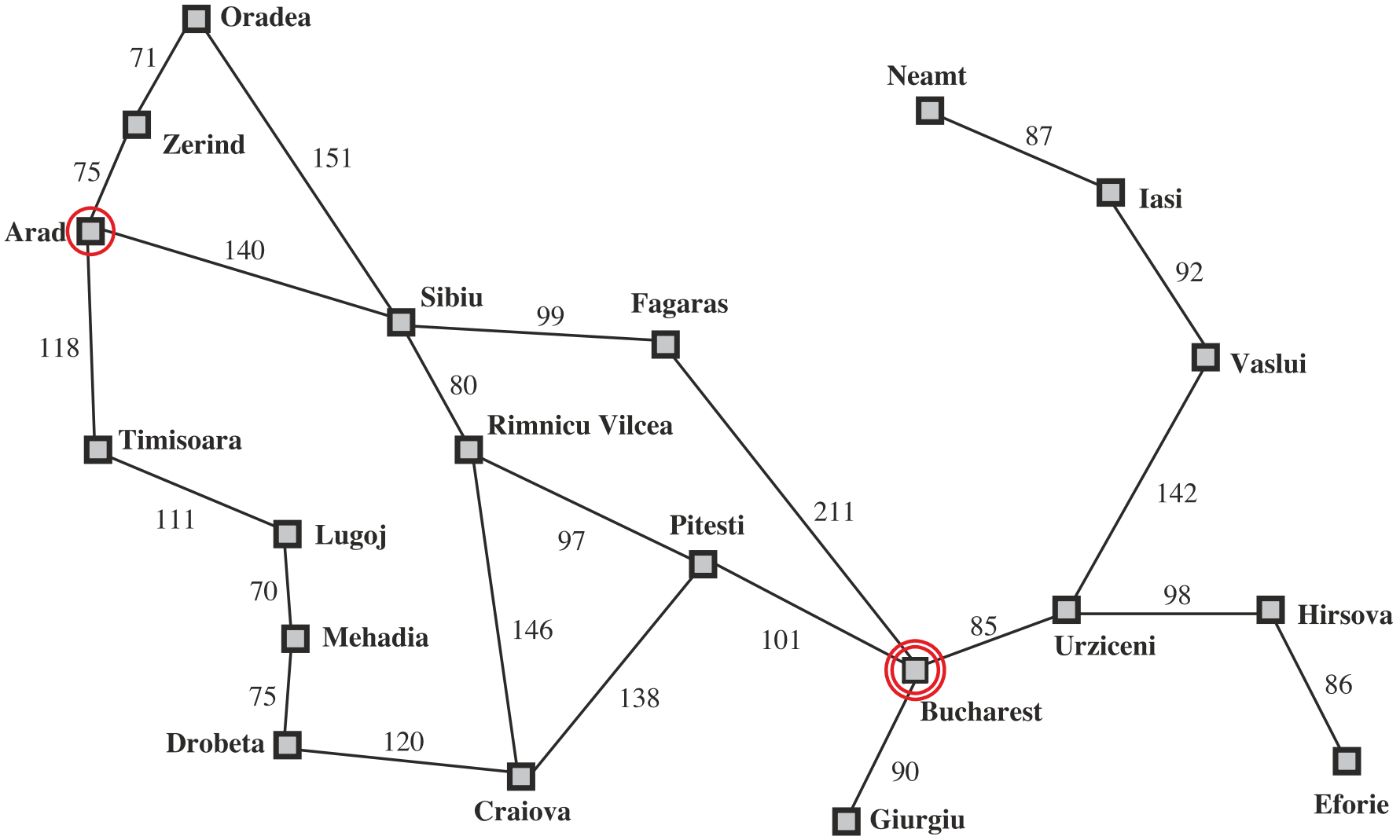
۴ راه‌حل
Solution

دنباله‌ای از شهرها
مثلاً: Arad → Sibiu → Fagaras → Bucharest



مثال: مسئله‌ی رومانی

نقشه



فرمول‌بندی مسئله

مؤلفه‌های پنج‌گانه‌ی تعریف مسئله

مؤلفه‌های پنج‌گانه‌ی تعریف مسئله

حالت آغازین Initial State	حالتی که عامل از آن شروع می‌کند.
کنش‌های ممکن Available Actions	کنش‌های ممکن عامل در هر حالت s : $\text{ACTIONS}(s)$
مدل گذار Transition Model	در هر حالت s ، هر کنش a چه می‌کند؟ : $\text{RESULT}(s,a)$
آزمون هدف Goal Test	تعیین کننده‌ی اینکه حالت جاری هدف است یا خیر: **صریح** (explicit): بیان مجموعه حالات هدف **ضمنی** (implicit): بیان ویژگی هدف
تابع هزینه‌ی کنش Action Cost Function	هزینه‌ی گذار از یک حالت به حالت دیگر با یک کنش $c(s,\ a,\ s')$

تابع هزینه‌ی مسیر (path cost function): تابعی که به هر مسیر یک هزینه‌ی عددی نسبت می‌دهد: مجموع هزینه‌ی کنش‌های آن مسیر

فضای حالت مسئله *problem state-space*

مجموعه‌ی حالت‌های دسترس‌پذیر از حالت آغازین با حداقل یک دنباله از کنش‌ها

فضای حالت

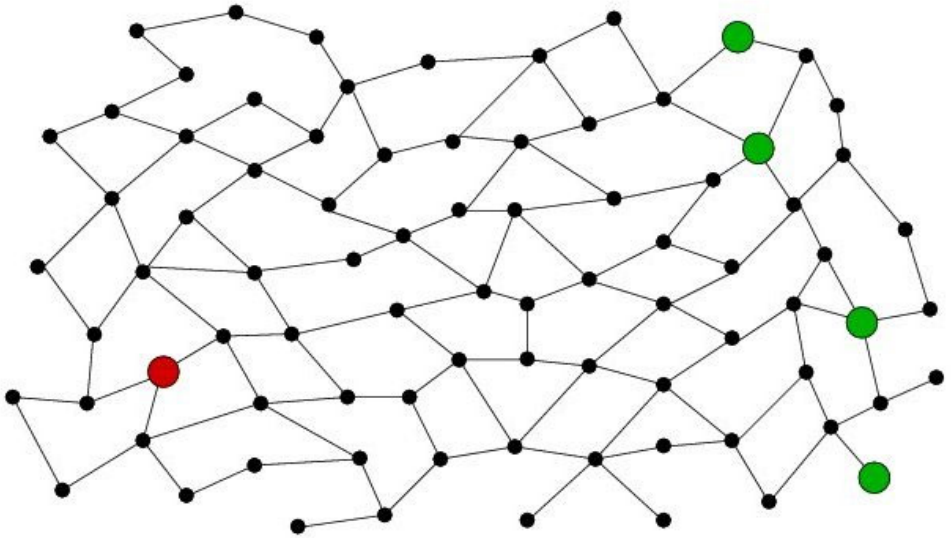
STATE SPACE

فضای حالت: مجموعه‌ی همه‌ی حالت‌های ممکن در مسئله و رابطه‌های آنها

در قالب یک گراف نمایش داده می‌شود.

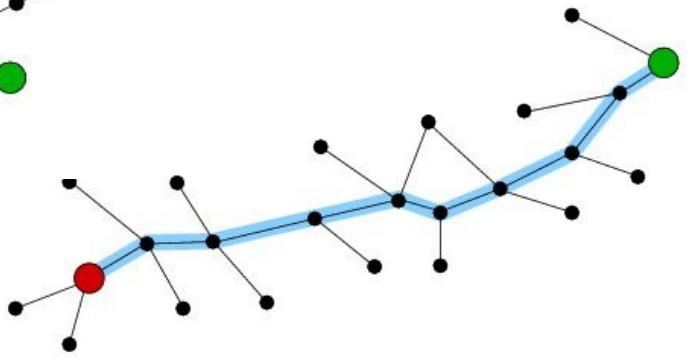
مجموعه‌ی حالت‌های دسترس‌پذیر از حالت آغازین با حداقل یک دنباله از کنش‌ها

گراف فضای حالت:



راه حل:

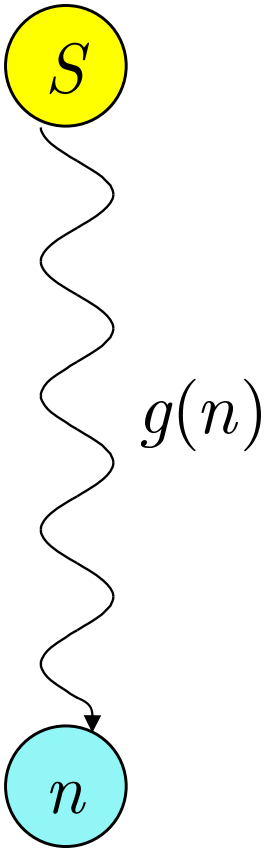
مسیری از حالت آغازین به حالت نهایی



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 (تجرید): 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.

هوش مصنوعی

حل مسئله با جستجو

۲

مسئله‌های نمونه

مسئله‌های نمونه

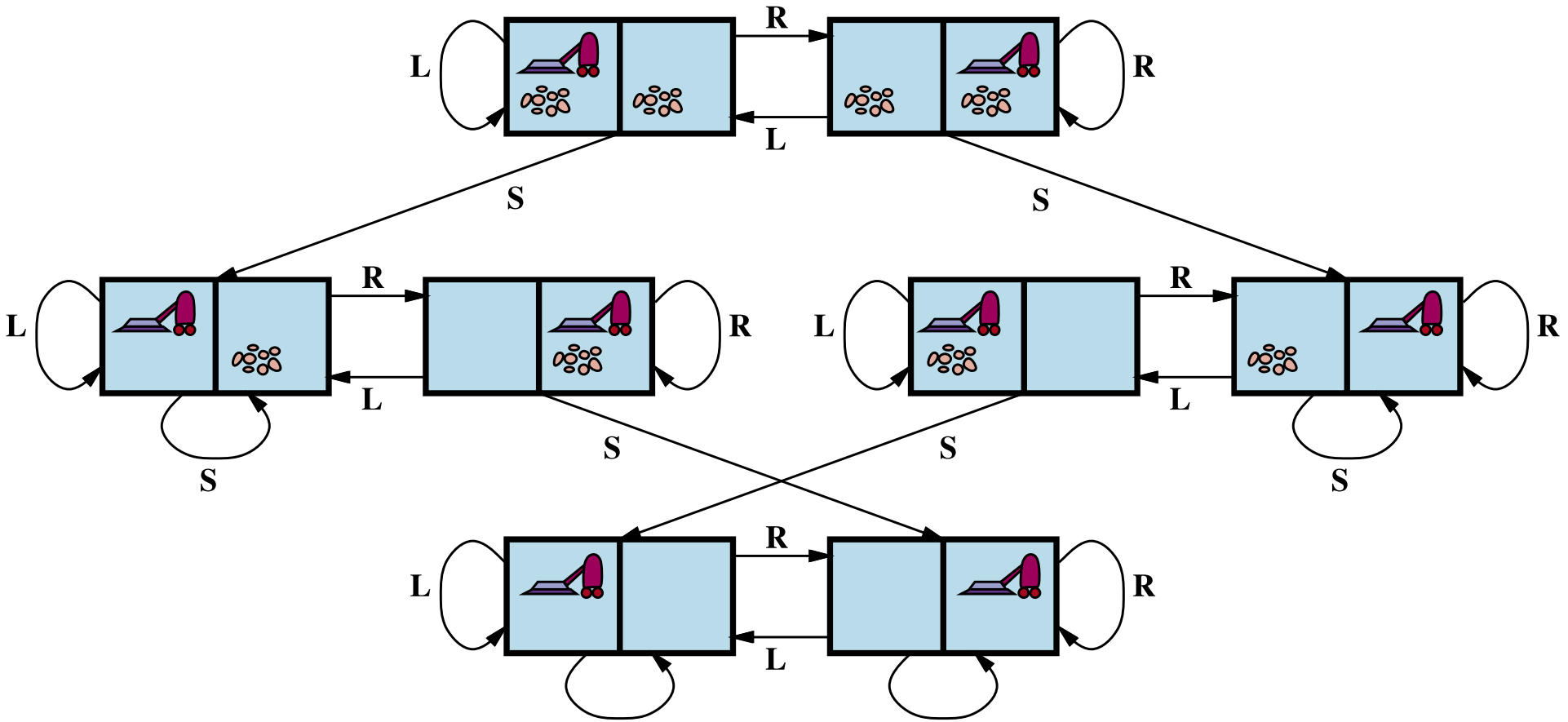
مسئله‌های نمونه	
مسئله‌های استانداردسازی شده *Standardized Problems*	**مسئله‌های دنیای واقعی** *Real-world Problems*
مسائل با تعریف خلاصه و دقیق برای نمایش، تمرین و مقایسه‌ی کارآیی روش‌های حل مسئله	مسائل با تعریف مفصل‌تر و منعطف‌تر، حل آنها برای مردم اهمیت دارد.

- 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)
- …

- 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
- …

مثال: دنیای جاروبرقی

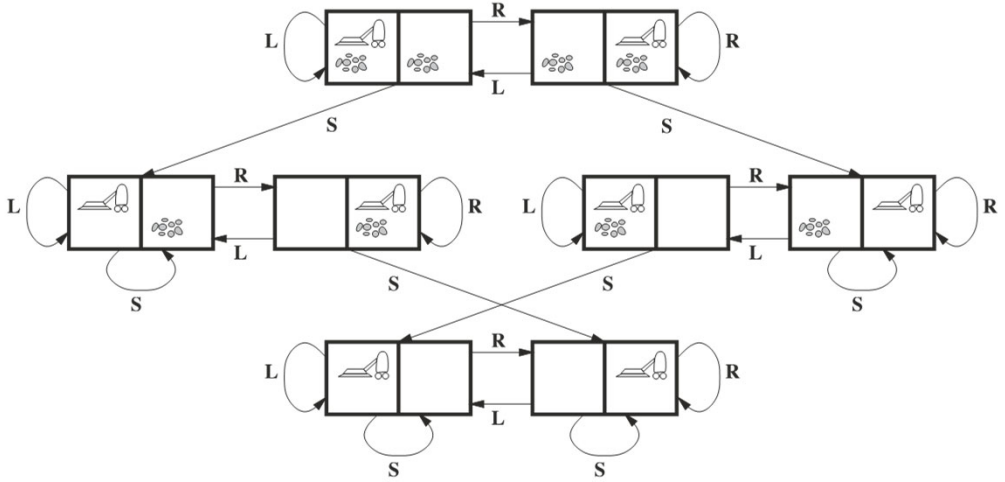
گراف فضای حالت



مثال: دنیای جاوربرقی

تعریف مسئله

VACCUM 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.

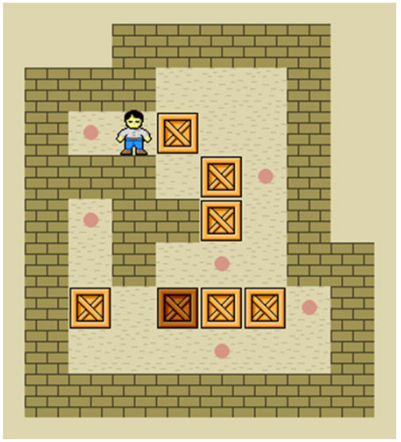
مؤلفه‌های پنج‌گانه‌ی تعریف مسئله	
حالت‌ها States	مکان عامل و مکان‌های آلوده (نادیده گرفتن مقدار آلودگی)
حالت آغازین Initial State	هر یک از حالت‌ها ممکن است.
کنش‌های ممکن Available Actions	*Left, Right, Suck, NoOp*
مدل گذار Transition Model	مطابق گراف فضای حالت
آزمون هدف Goal Test	ضمنی (implicit): حالتی که در آن تمام خانه‌ها تمیز باشند.
تابع هزینه‌ی کنش Action Cost Function	هزینه‌ی گام (step cost): هر کنش ۱ واحد (صفر برای *NoOp*)/ تابع هزینه مسیر جمعی

مثال: معمای سوکوبان

تعریف مسئله

نمونه‌ای دیگر از مسائل
grid world problems
مسائل دنیای مشبک:

هدف عامل،
هل دادن تعدادی جعبه پراکنده در یک مشبکه
به سمت مکان‌های ذخیره‌سازی مشخص شده است.



SOKOBAN PUZZLE

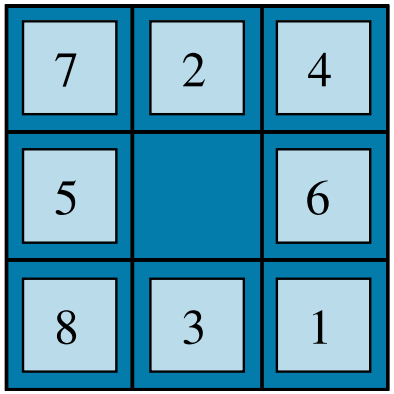
ـ حداکثر یک جعبه در هر سلول می‌تواند قرار بگیرد.
ـ وقتی یک عامل به سمت یک سلول حاوی یک جعبه حرکت کند و یک سلول خالی در سمت دیگر جعبه وجود داشته باشد، آن‌گاه هم جعبه و هم عامل به سمت جلو حرکت می‌کنند.
ـ عامل نمی‌تواند یک جعبه را به داخل جعبه‌ی دیگر یا دیوار هل بدهد.

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

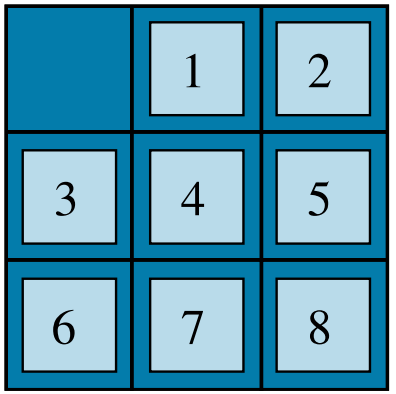
مثال: معمای–۸

تعریف مسئله

8-PUZZLE



Start State



Goal State

نمونه‌ای از مسائل
sliding-block puzzles
n-puzzle
در حالت کلی NP-hard

مؤلفه‌های پنج‌گانه‌ی تعریف مسئله	
حالت‌ها States	موقعیت کاشی‌ها در قالب برداری از اعداد صحیح مثبت (نادیده گرفتن موقعیت‌های میانی)
حالت آغازین Initial State	هر یک از حالت‌ها ممکن است.
کنش‌های ممکن Available Actions	حرکت دادن خانه‌ی خالی به چپ، راست، بالا، پایین
مدل گذار Transition Model	مطابق تعریف
آزمون هدف Goal Test	صریح (explicit): حالت هدف (داده شده)
تابع هزینه‌ی کنش Action Cost Function	هزینه‌ی گام (step cost): هر حرکت ۱ واحد / تابع هزینه مسیر جمعی

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

تعریف مسئله

KNUTH PROBLEM

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

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

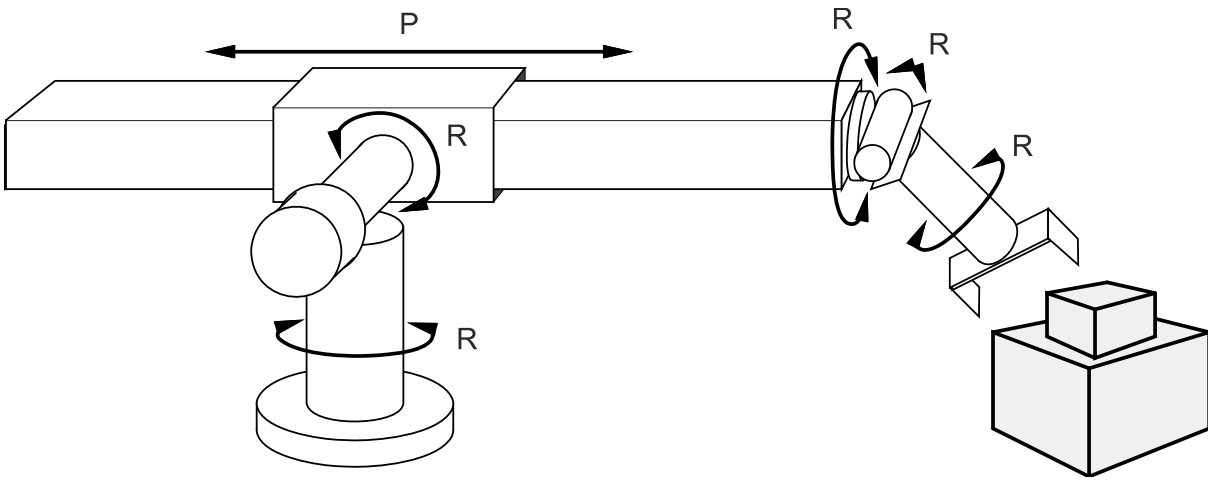
Donald Knuth (1964)

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

مثال: اسمبل کردن با ربات

تعریف مسئله

ROBOTIC ASSEMBLY



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

مؤلفه‌های پنج‌گانه‌ی تعریف مسئله

هوش مصنوعی

حل مسئله با جستجو

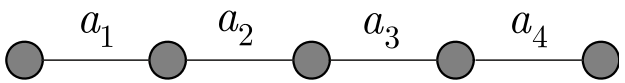
۳

جستجو به‌دنبال راه‌حل

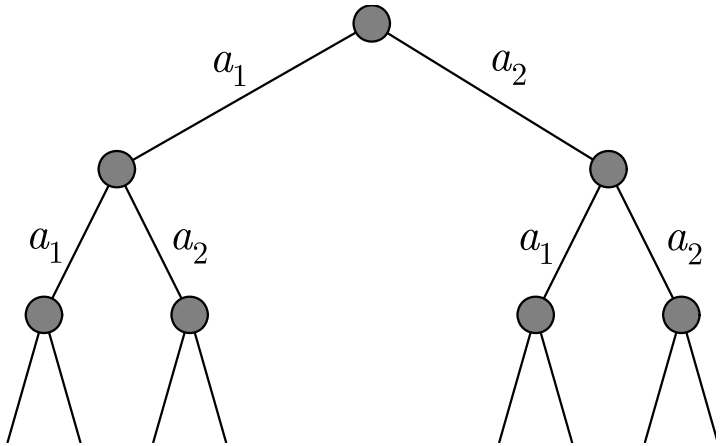
درخت جستجو

SEARCH TREE

راه‌حل *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.

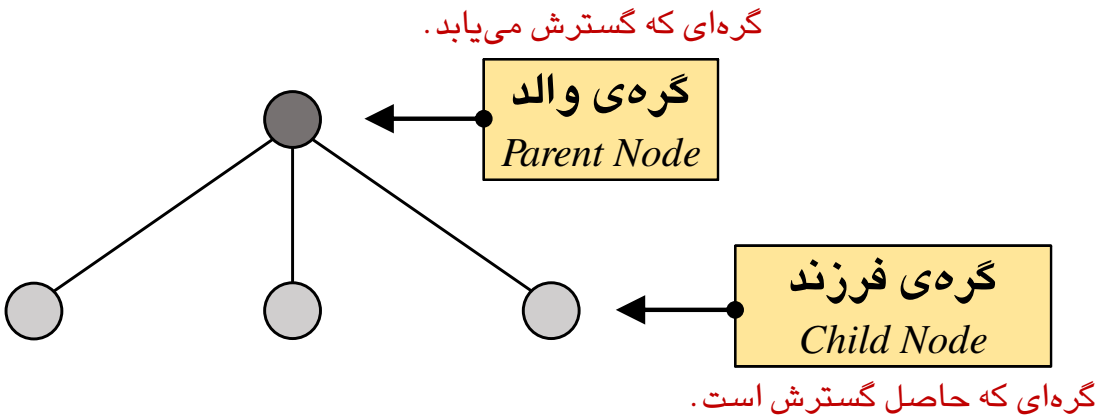


درخت جستجو *Search Tree*	
شاخه‌ها *Branches*	گره‌ها *Nodes*
متناظر با **کنش‌های عامل**	متناظر با **حالت‌ها** (در فضای حالت مسئله)

گسترش حالت در درخت جستجو

SEARCH TREE

گسترش *Expanding*	اعمال همه‌ی کنش‌های مجاز به حالت جاری و تولید مجموعه حالت جدید



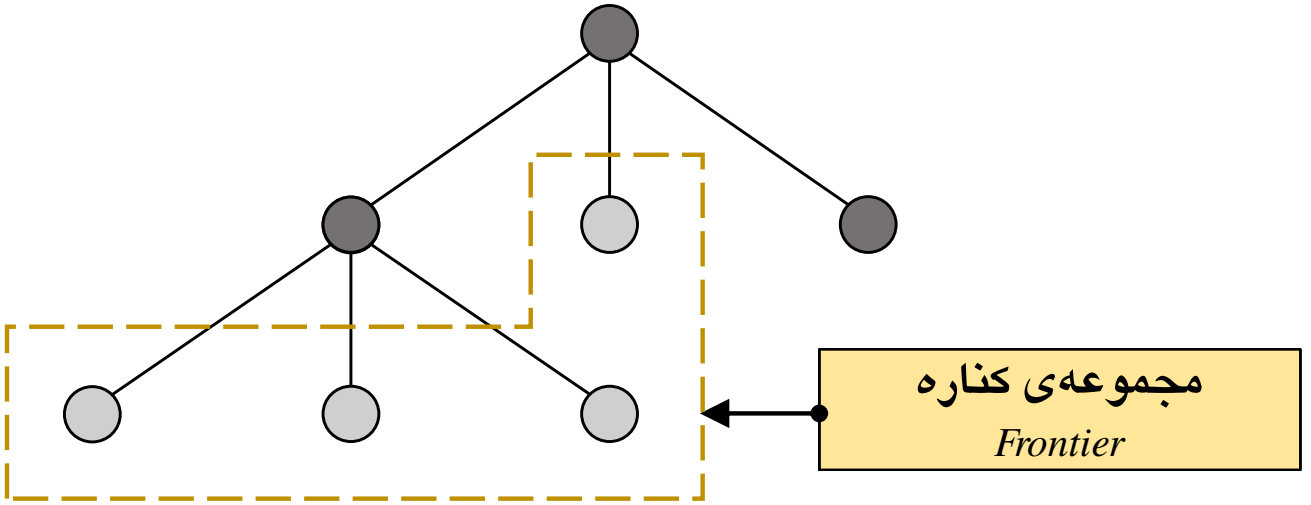
کناره

لیست باز

FRONTIER

کناره *Frontier*	لیست باز *Open list*	مجموعه‌ی همه‌ی گره‌های برگ آماده برای گسترش در هر نقطه

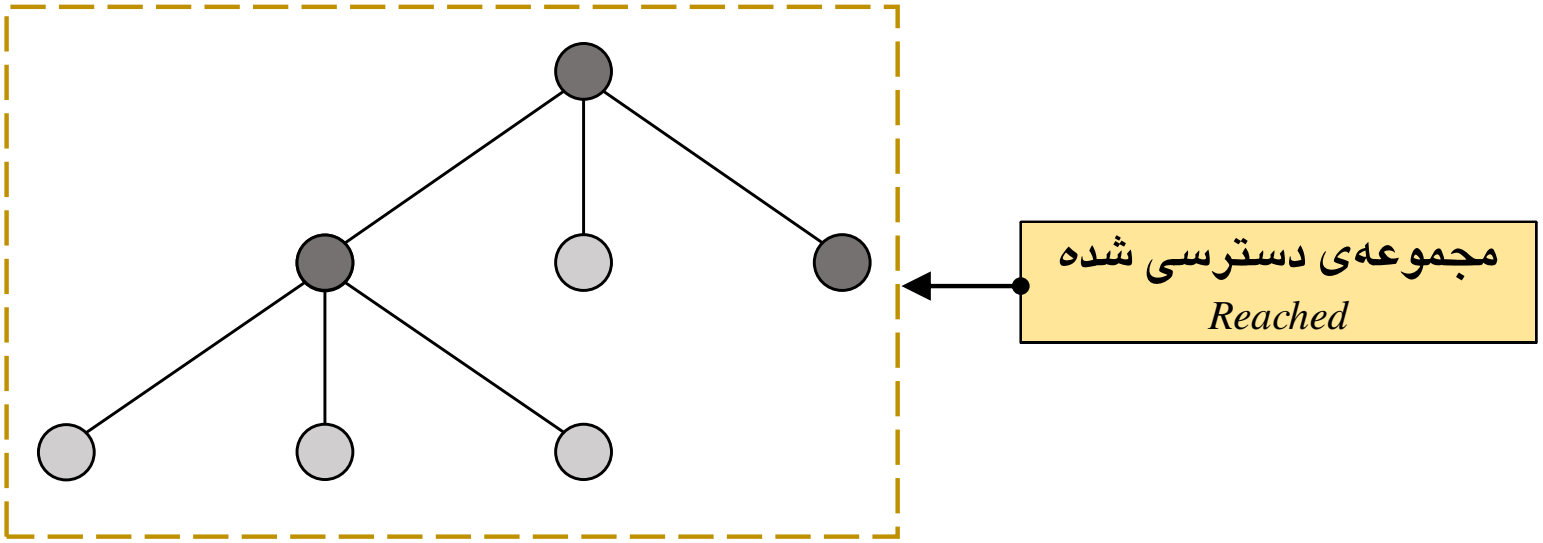
گره‌ی بدون فرزند در درخت



دسترسی شده

REACHED

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.)

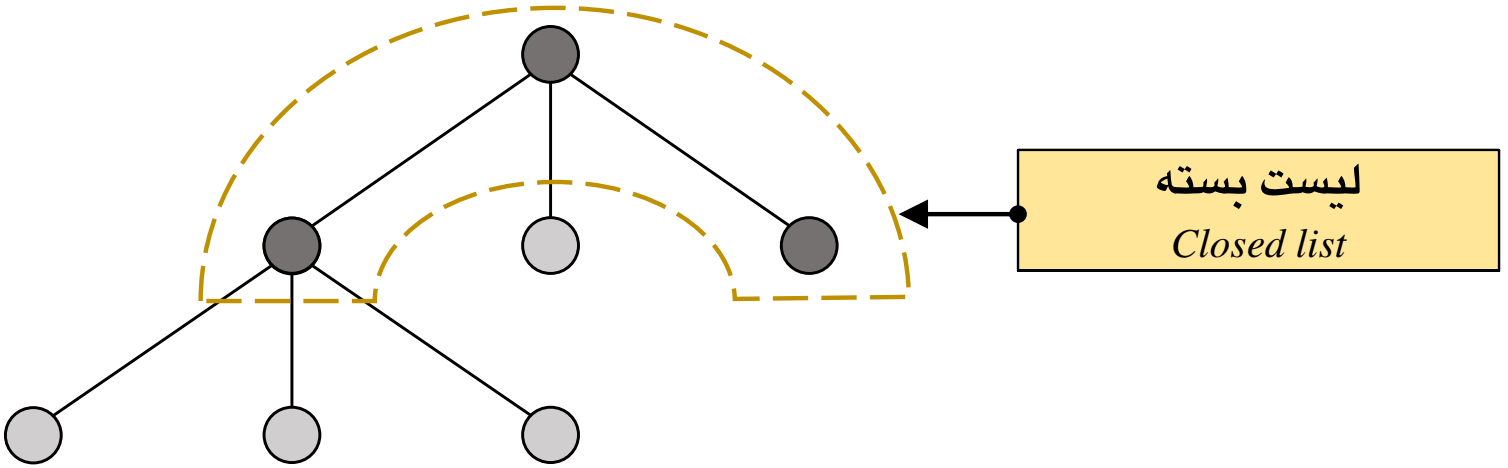


لیست بسته

CLOSED LIST

لیست بسته *Closed list*	مجموعه‌ی همه‌ی گره‌هایی که تاکنون گسترش پیدا کرده‌اند.

$$Closed\ list = Reached - Frontier$$



استراتژی جستجو

استراتژی جستجو
Search Strategy

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

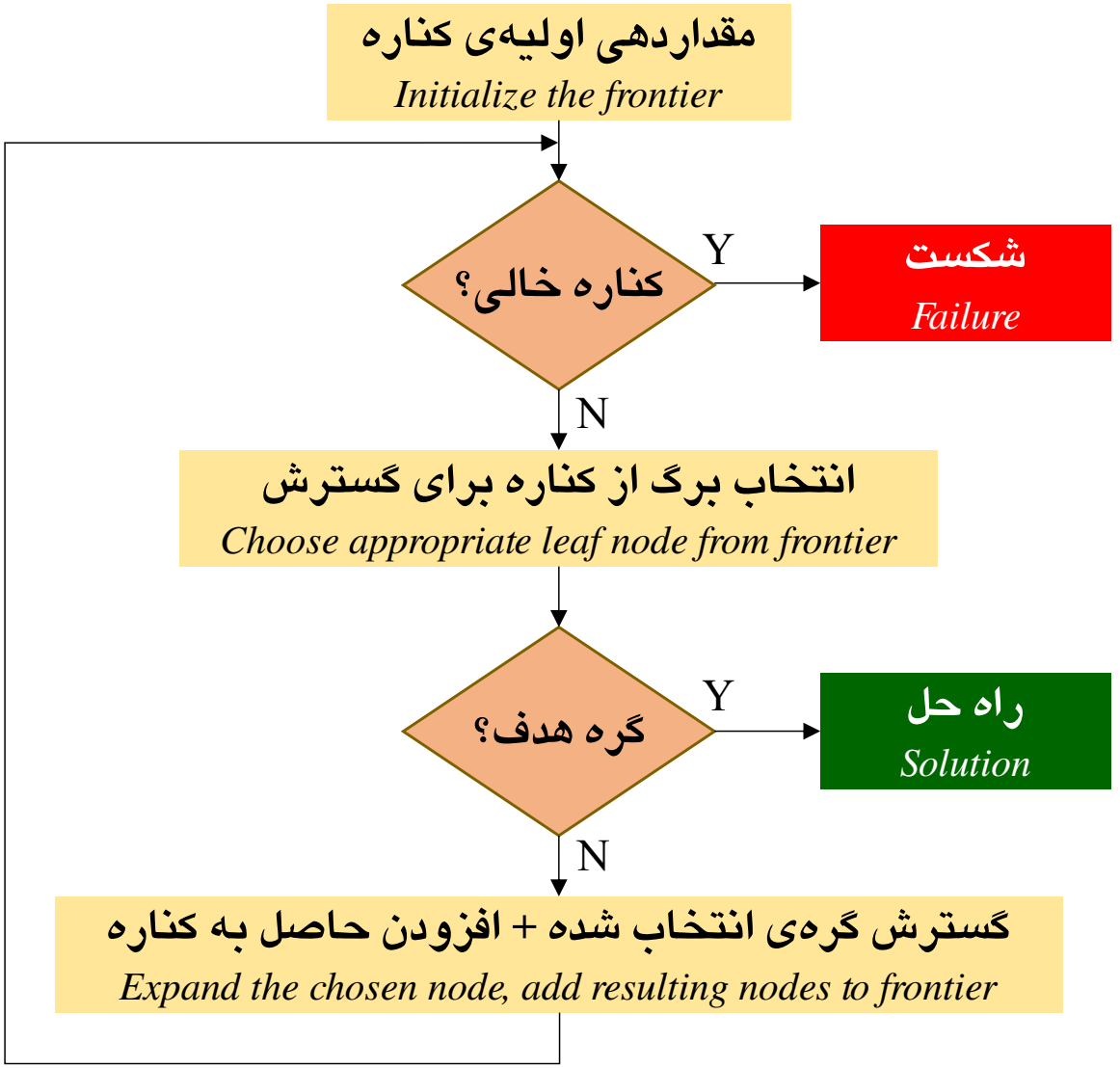
همه‌ی الگوریتم‌های جستجو در ساختار کلی مشترک هستند
تفاوت اصلی آنها در **استراتژی جستجو** است.

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

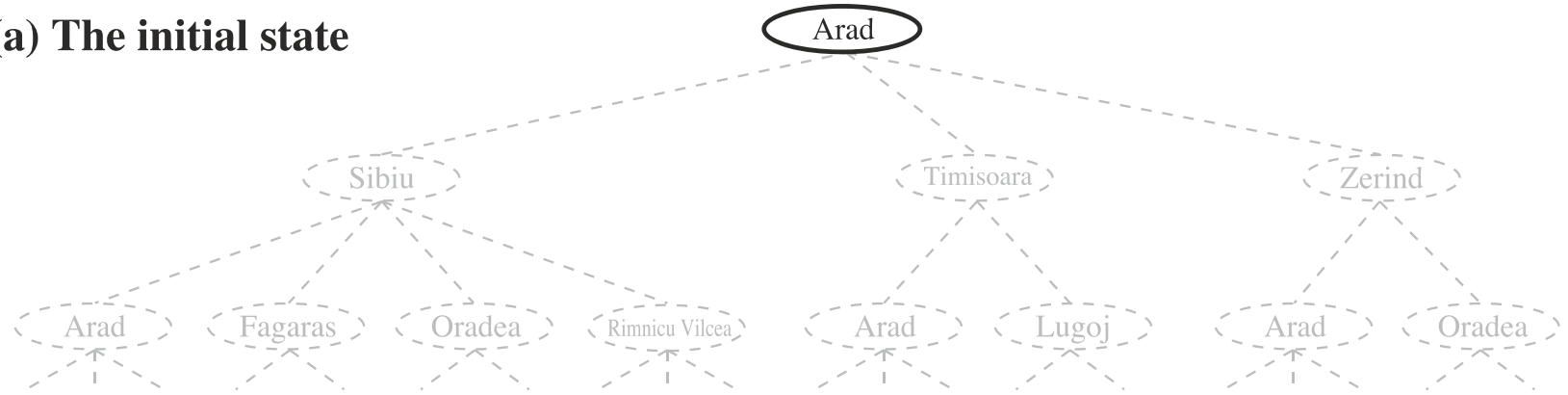
فلوچارت



الگوریتم جستجوی درختی

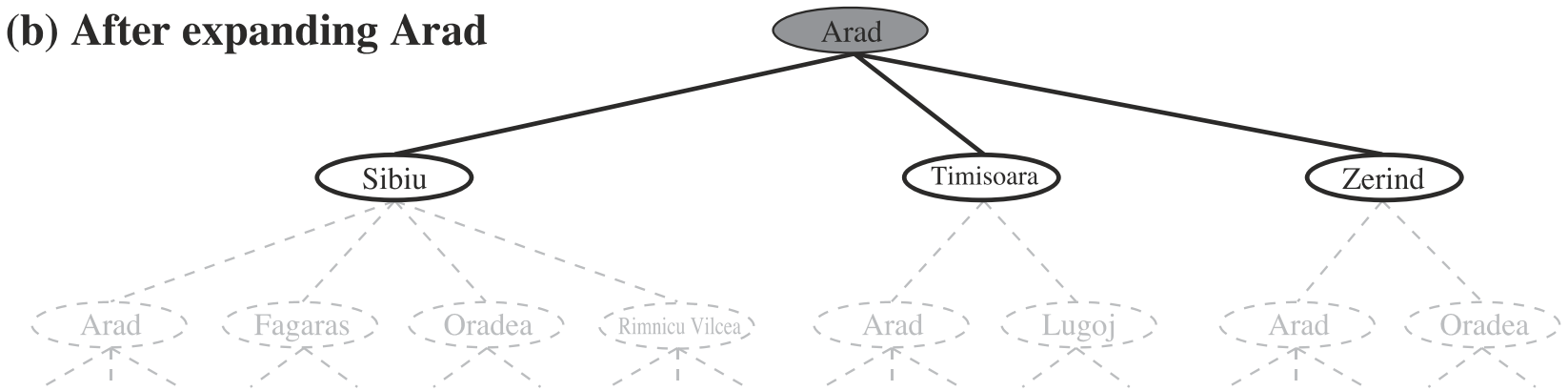
مثال: ۱ از ۳

(a) The initial state



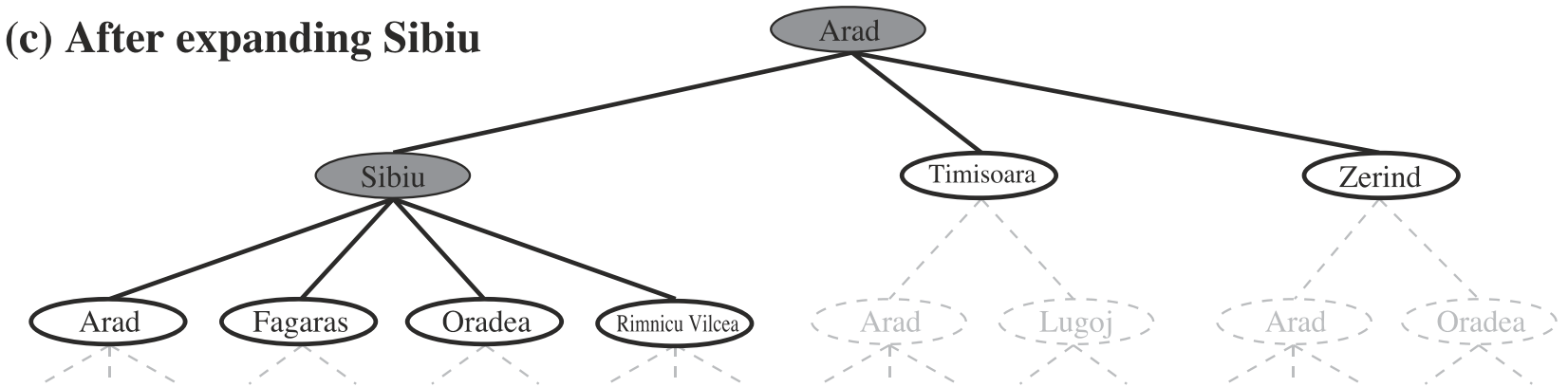
الگوریتم جستجوی درختی

مثال: ۲ از ۳



الگوریتم جستجوی درختی

مثال: ۳ از ۳



الگوریتم جستجوی بهترین–اول

BEST-FIRST SEARCH

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

function EXPAND(problem, node) yields nodes
    s ← node.STATE
    for each action in problem.ACTIONS(s) do
        s′ ← problem.RESULT(s, action)
        cost ← 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.
(مثلا در مسئله‌ی n–وزیر هر بار وزیر جدید در سمت‌چپ‌ترین 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، به‌خاطر 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*

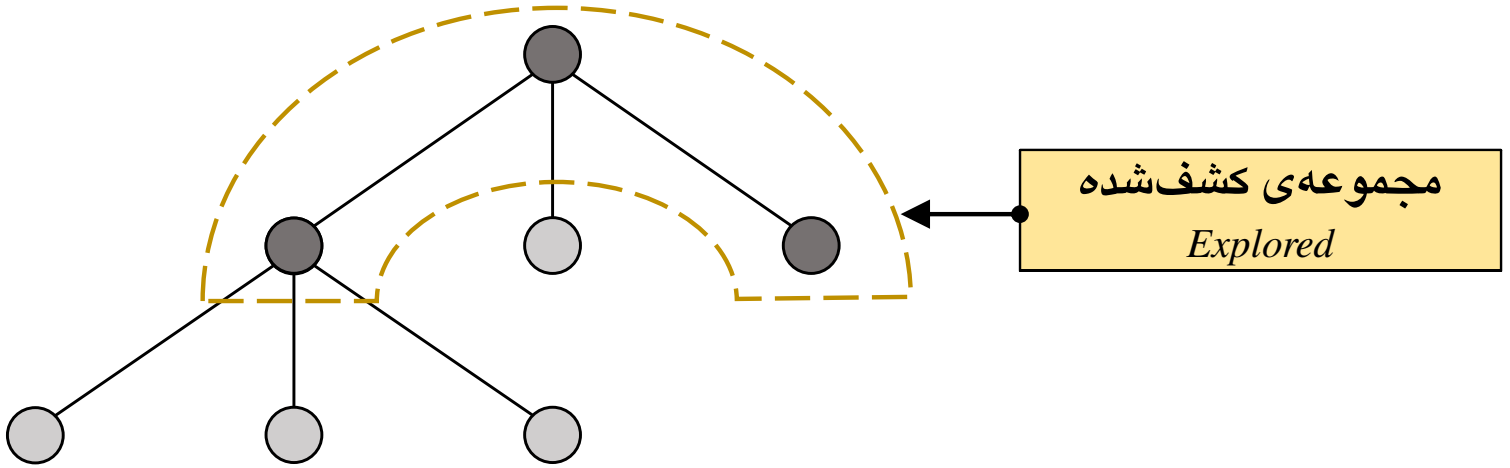
در صورت مطابقت، به جای اضافه شدن به frontier نادیده گرفته می‌شوند.

مجموعه‌ی کشف‌شده

لیست بسته

EXPLORED SET

مجموعه‌ی کشف‌شده *Explored Set*	لیست بسته *Closed list*	مجموعه‌ی همه‌ی گره‌های بررسی شده و گسترش‌یافته



الگوریتم جستجوی گرافی

جستجوی گرافی: الگوریتمی که مسیرهای افزونه را بررسی می‌کند.

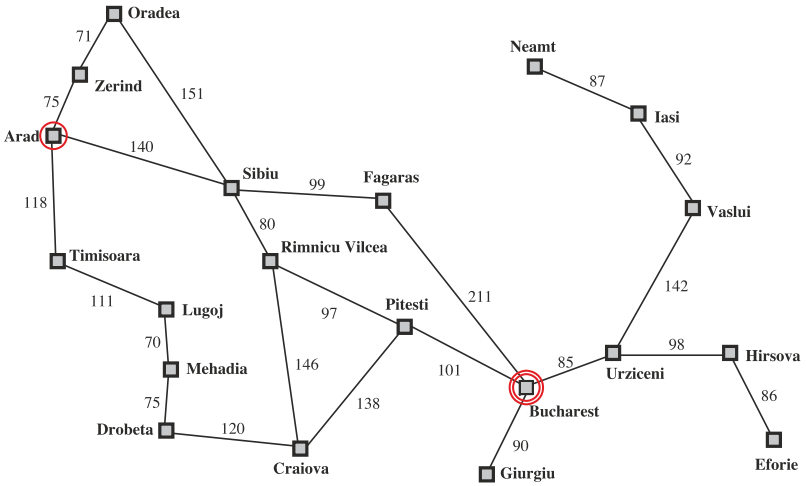
function GRAPH-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  initialize the explored set to be empty
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the 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.

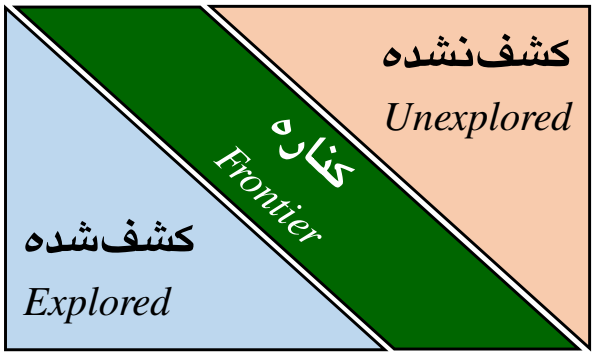
الگوریتم جستجوی گرافی

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.



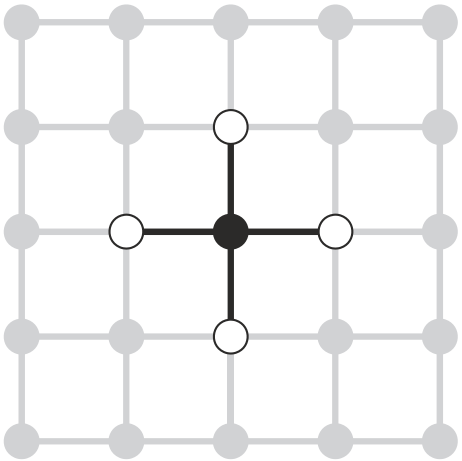
گراف فضای حالت

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

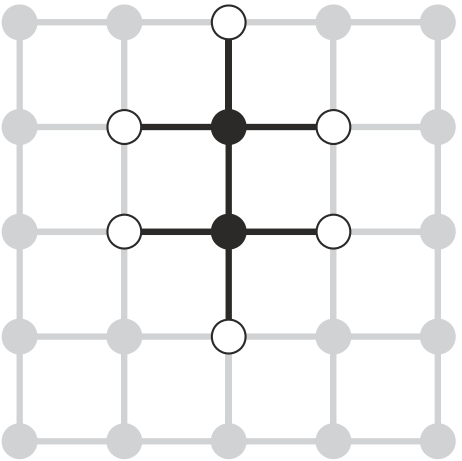
جستجوی گرافی

خاصیت جداکنندگی

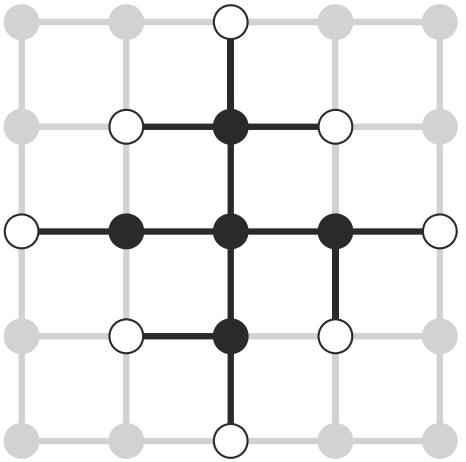
GRAPH-SEARCH: SEPARATION PROPERTY



(a)



(b)



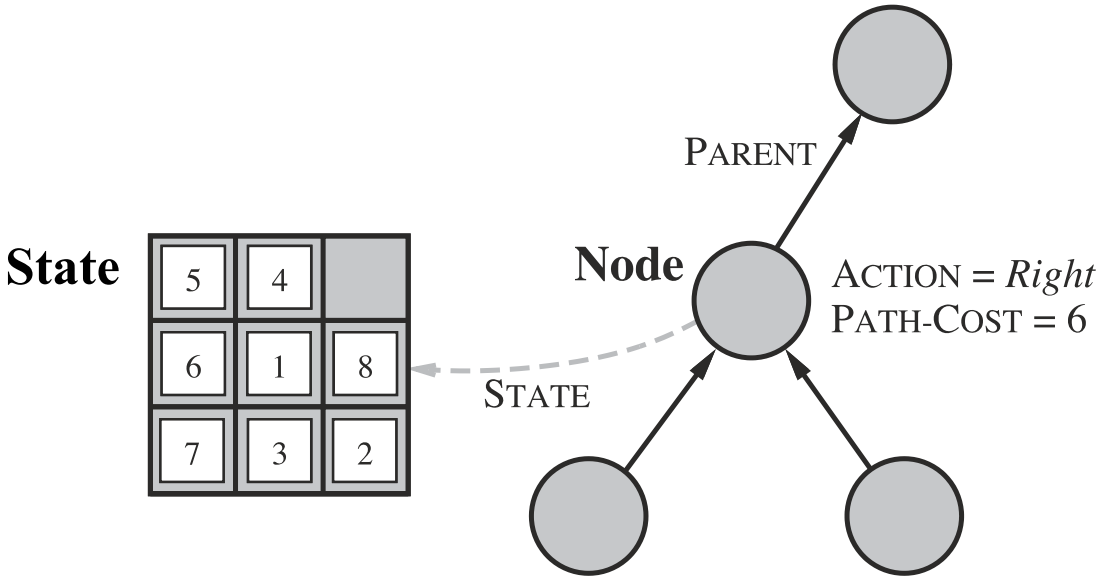
(c)

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

کناره	frontier	○
کشف‌شده	explored	●
کشف‌نشده	unexplored	●

کناره، همیشه ناحیه‌ی کشف‌شده را از ناحیه‌ی کشف‌نشده‌ی فضای حالت جدا می‌کند.

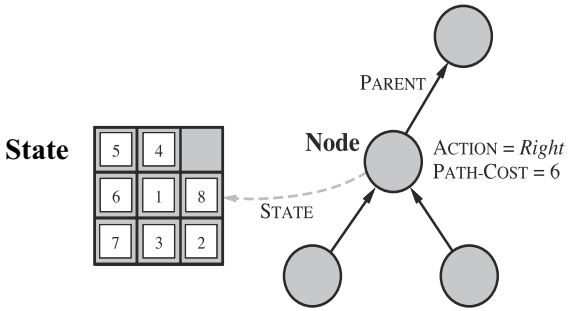
ساختمان‌داده‌ی «گره» در درخت جستجو



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»



حالت *State*	گره *Node*
در فضای حالت مسئله	در درخت جستجو
(بازنمایی) یک پیکربندی فیزیکی	یک ساختمان‌داده: جزء تشکیل‌دهنده‌ی درخت جستجو
دارای ماقبل‌ها (predecessors)	دارای والد (parent)
دارای مابعدها (successors)	دارای فرزندان (children)
–	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.

$queue$

EMPTY?($queue$)	POP($queue$)	TOP($queue$)	ADD($element$, $queue$)
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 کارآیی الگوریتم/استراتژی جستجو

الگورتیم‌ها/استراتژی‌های جستجو در امتداد ابعاد زیر ارزیابی می‌شوند:

۴	۳	۲	۱
پیچیدگی فضایی *Space Complexity*	**پیچیدگی زمانی** *Time Complexity*	**بهینگی هزینه** *Cost Optimality*	**تمامیت** *Completeness*
چه‌قدر حافظه برای انجام جستجو لازم است؟ (معمولاً بر حسب حداکثر تعداد گره‌های موجود در حافظه)	چه‌قدر زمان برای پیدا کردن راه‌حل لازم است؟ (معمولاً بر حسب تعداد گره‌های تولیدشده/گسترش‌یافته)	آیا این استراتژی لزوماً راه‌حل بهینه را پیدا می‌کند؟	آیا تضمینی وجود دارد این الگوریتم راه‌حلی را در صورت وجود بیابد؟ (الگوریتم باید **سیستماتیک** باشد: شیوه‌ی کاوش فضای حالت نامتناهی به‌گونه‌ای باشد که سرانجام بتواند به هر حالت متصل به حالت آغازین برسد.)

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

بسم الله الرحمن الرحیم

هوش مصنوعی

فصل ۳

حل مسئله با جستجو (۲)

Solving Problems by Searching (2)

کاظم فولادی قلعه

دانشکده مهندسی، پردیس فارابی

دانشگاه تهران

http://courses.fouladi.ir/ai

هوش مصنوعی

حل مسئله با جستجو

۴

استراتژی‌های جستجوی ناآگاهانه (کورکورانه)

استراتژی‌های جستجوی ناآگاهانه

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

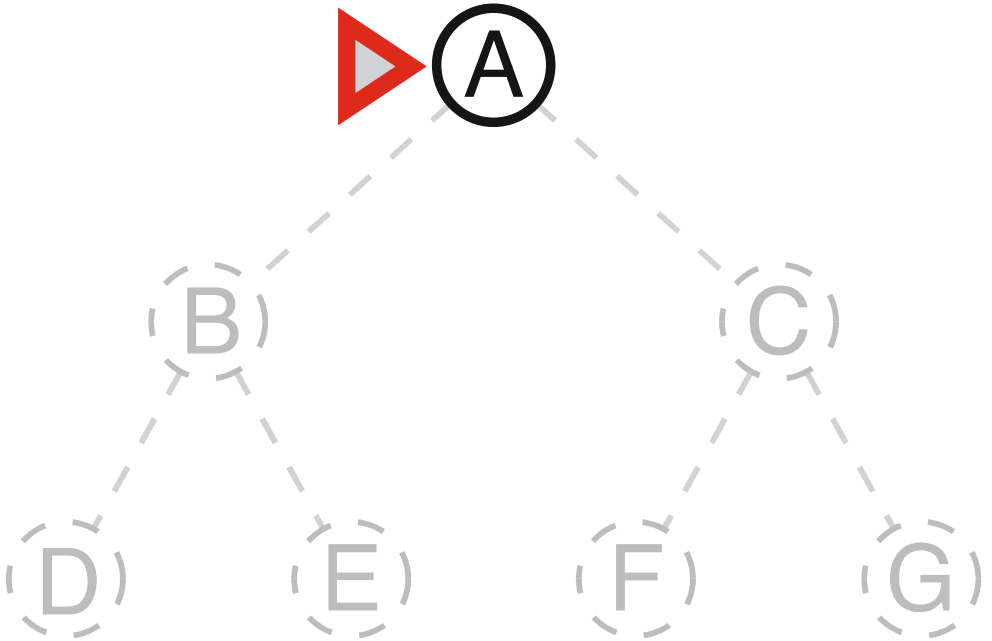
- جستجوی عرض–اول (Breadth-first: BFS)
- جستجوی هزینه–یکنواخت (Uniform-Cost: UCS)
- جستجوی عمق–اول (Depth-first: DFS)
- جستجوی عمقی محدودشده (Depth-limited: DLS)
- جستجوی عمیق‌کننده‌ی تکراری (Iterative Deepening: IDS)
- جستجوی دوطرفه (Bidirectional)

جستجوی عرض–اول

BREADTH-FIRST SEARCH

قاعده *BFS*	کم‌عمق‌ترین گره در کناره را گسترش دهید. *Expand shallowest node in the frontier.*

پیاده‌سازی: کناره، یک صف FIFO است (فرزندان جدید به انتهای صف اضافه می‌شوند)

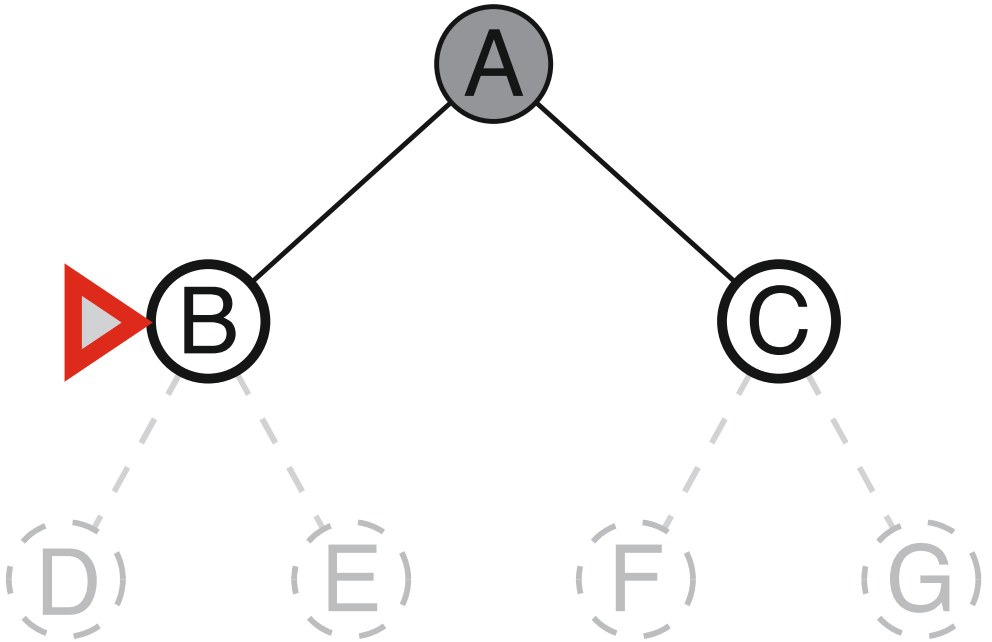


جستجوی عرض–اول

BREADTH-FIRST SEARCH

قاعده	کم‌عمق‌ترین گره در کناره را گسترش دهید. *Expand shallowest node in the frontier.*
BFS	

پیاده‌سازی: کناره، یک صف FIFO است (فرزندان جدید به انتهای صف اضافه می‌شوند)

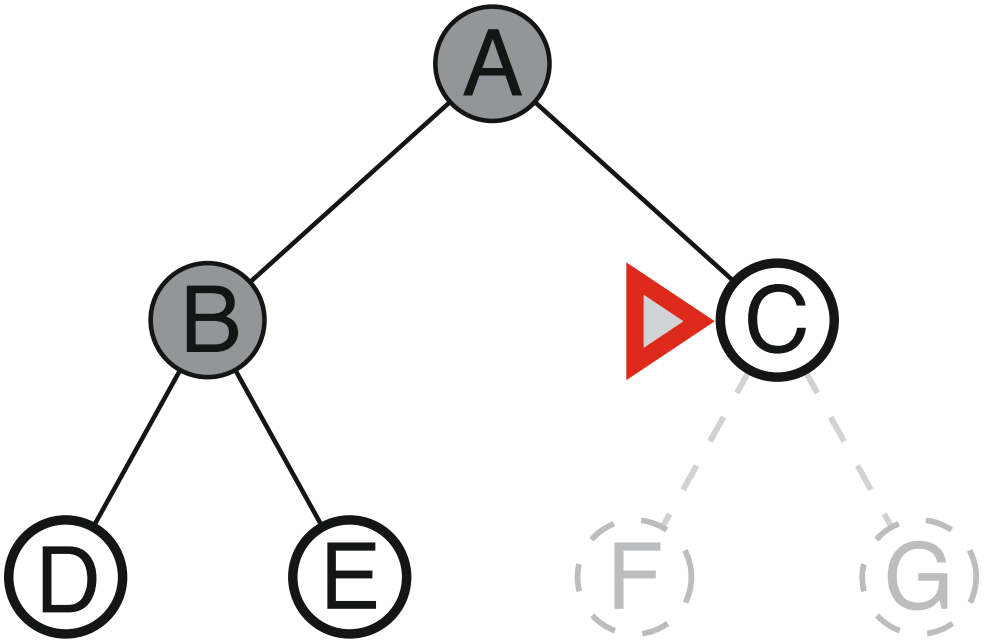


جستجوی عرض–اول

BREADTH-FIRST SEARCH

قاعده *BFS*	کم‌عمق‌ترین گره در کناره را گسترش دهید. *Expand shallowest node in the frontier.*

پیاده‌سازی: کناره، یک صف FIFO است (فرزندان جدید به انتهای صف اضافه می‌شوند)

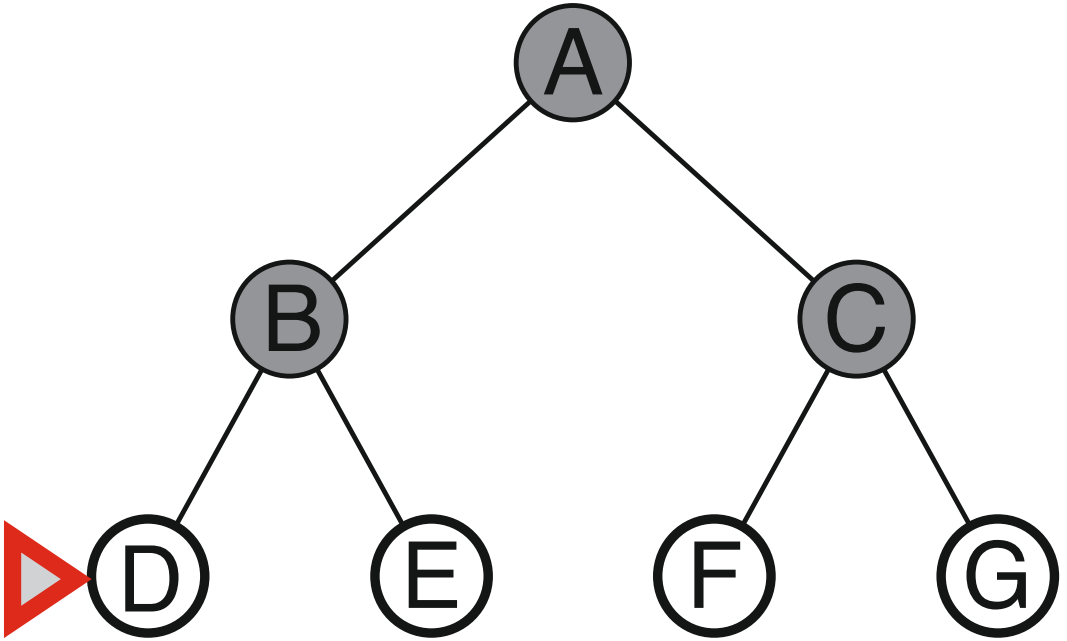


جستجوی عرض–اول

BREADTH-FIRST SEARCH

قاعده *BFS*	کم‌عمق‌ترین گره در کناره را گسترش دهید. *Expand shallowest node in the frontier.*

پیاده‌سازی: کناره، یک صف FIFO است (فرزندان جدید به انتهای صف اضافه می‌شوند)



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

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

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

جستجوی عرض-اول

شبه‌کد کارآمدتر

```
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
```

ویژگی‌ها:

- استفاده از صف FIFO به‌جای صف اولویت
- مجموعه‌ی *reached* به‌صورت مجموعه‌ای از حالت‌ها به‌جای نگاشتی از حالت‌ها به گره‌ها (آزمون هدف زودهنگام)

جستجوی عرض–اول

آزمون هدف زودهنگام / آزمون هدف دیرهنگام

BREADTH-FIRST SEARCH

آزمون هدف

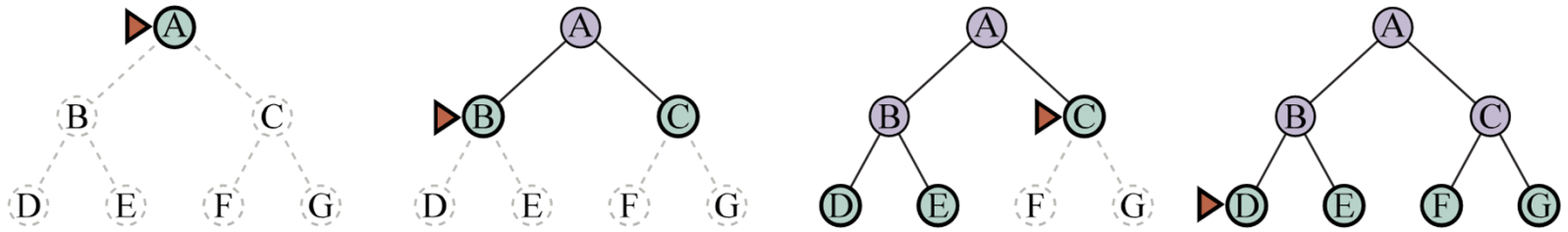
آزمون هدف زودهنگام
Early Goal Test

بررسی هدف بودن یک گره
به‌محض تولید آن

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



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			
۴	۳	۲	۱
پیچیدگی فضایی *Space Complexity*	**پیچیدگی زمانی** *Time Complexity*	**بهینگی هزینه** *Cost Optimality*	**تمامیت** *Completeness*
نمایی	نمایی	بله	بله
همه‌ی گره‌ها را در حافظه نگه می‌دارد.		اگر هزینه‌ی مسیر، تابع غیرنزولی از عمق گره باشد: مثلاً هزینه‌ی گام ۱ واحد	اگر b متناهی باشد

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

* 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-rst search. The numbers shown assume branching factor $b = 10$; 1 million nodes/second; 1000 bytes/node.

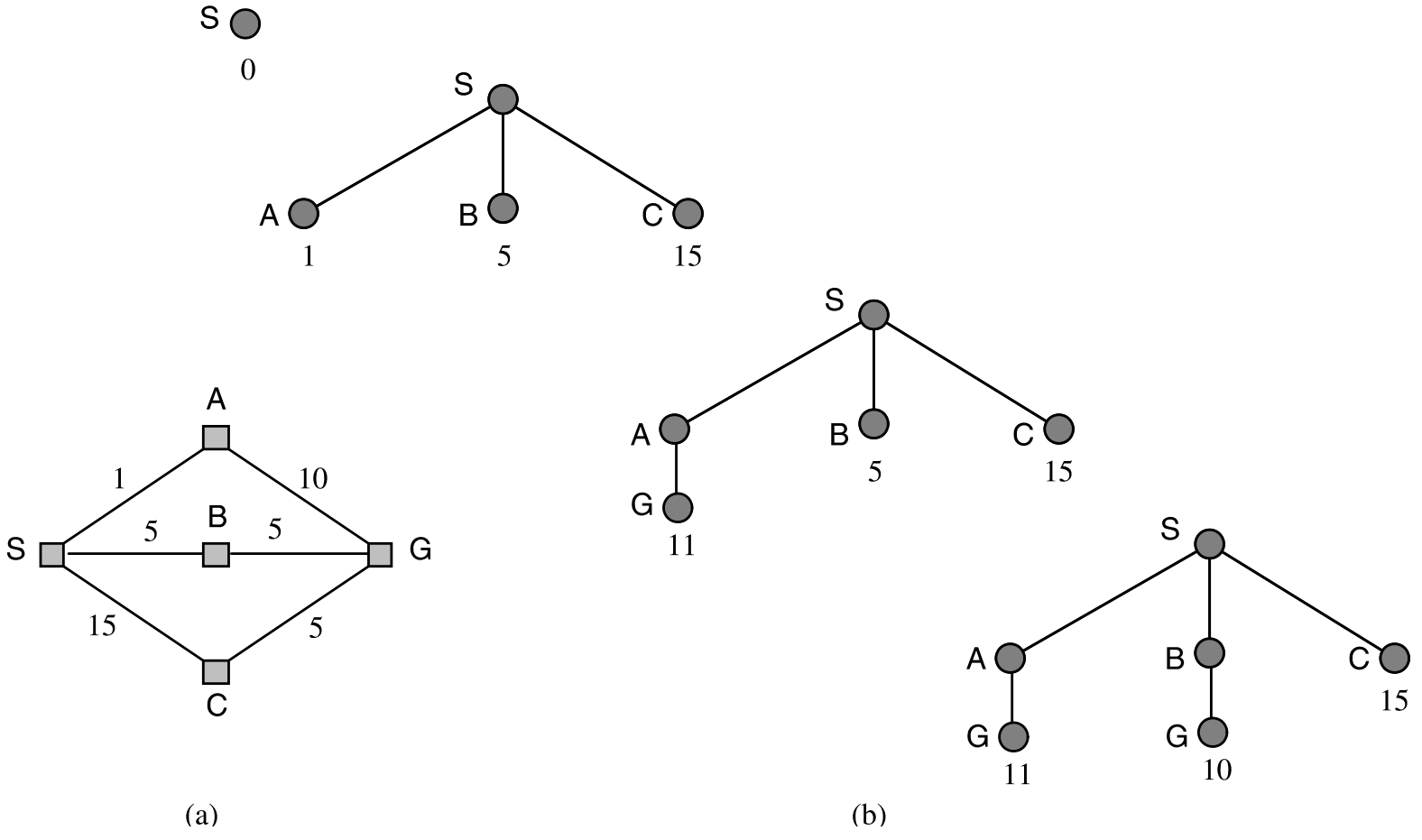
جستجوی هزینه–یکنواخت

جستجوی کم‌ترین هزینه (الگوریتم دایکسترا)

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

قاعده *UCS*	کم‌هزینه‌ترین گره در کناره را گسترش دهید. *Expand least-cost node in the frontier.*

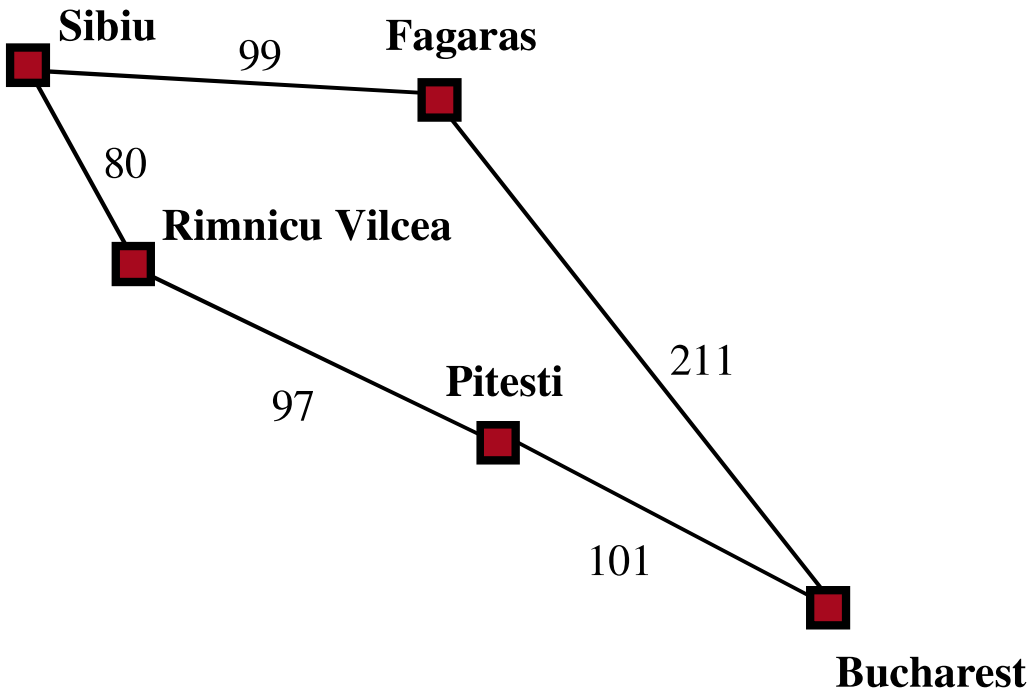
پیاده‌سازی: کناره، یک صف اولویت است (ترتیب گره‌ها بر اساس هزینه‌ی مسیر آنها)



جستجوی هزینه–یکنواخت

شبه‌کد

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 *Completeness*	**بهینگی هزینه** *Cost Optimality*	**پیچیدگی زمانی** *Time Complexity*	**پیچیدگی فضایی** *Space Complexity*
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. $O(b^{\lfloor C^*/\varepsilon \rfloor + 1})$

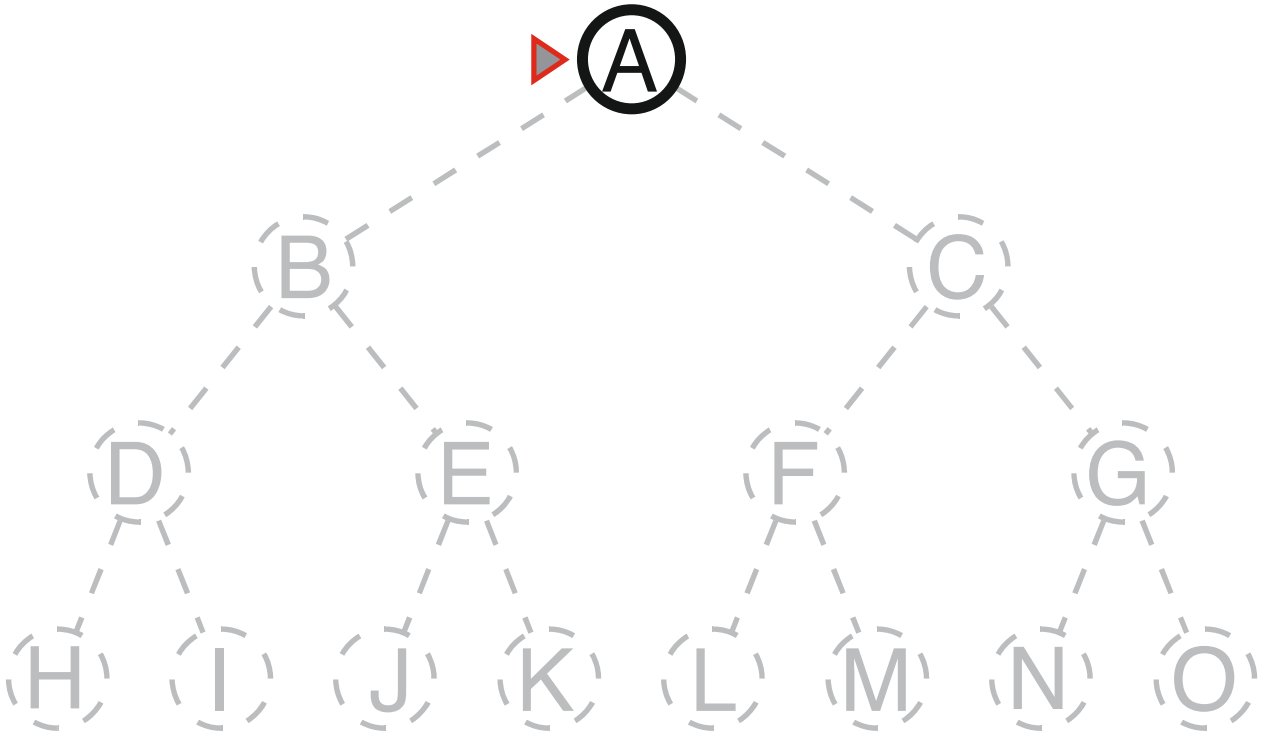
اگر هزینه‌ی هر گره برابر با عمق گره باشد، یا هزینه‌ی همه‌ی گام‌ها مساوی باشد، آن‌گاه UCS به BFS تبدیل می‌شود.

جستجوی عمق-اول

DEPTH-FIRST SEARCH

قاعده *DFS*	پرعمق‌ترین گره در کناره را گسترش دهید. *Expand deepest node in the frontier.*

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

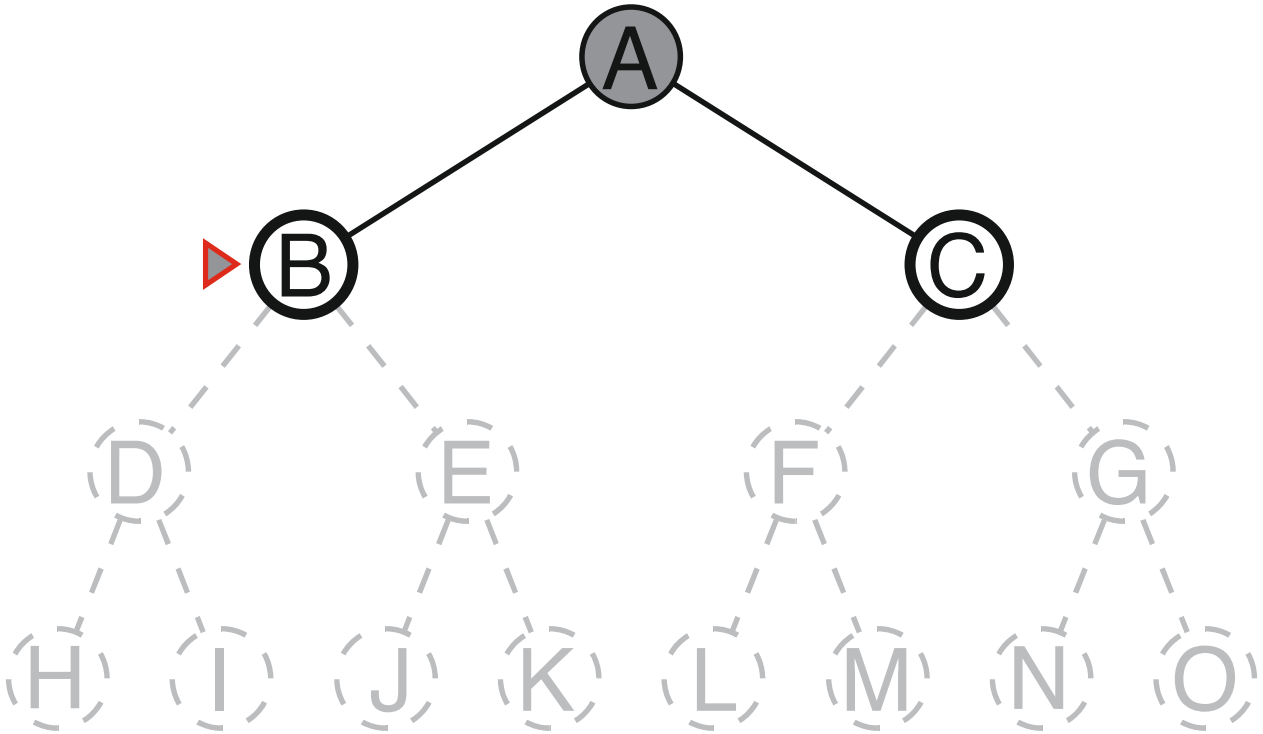


جستجوی عمق–اول

DEPTH-FIRST SEARCH

قاعده *DFS*	پرعمق‌ترین گره در کناره را گسترش دهید. *Expand deepest node in the frontier.*

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

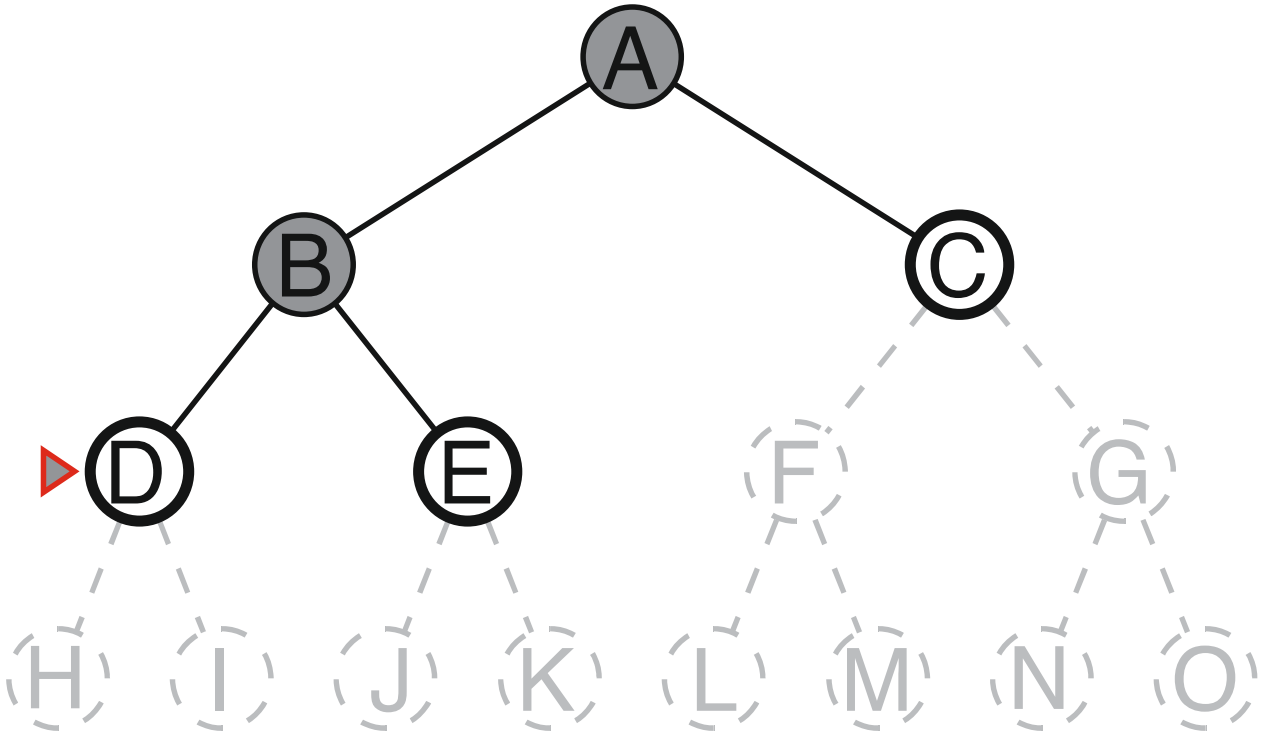


جستجوی عمق-اول

DEPTH-FIRST SEARCH

قاعده *DFS*	پرعمق‌ترین گره در کناره را گسترش دهید. *Expand deepest node in the frontier.*

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

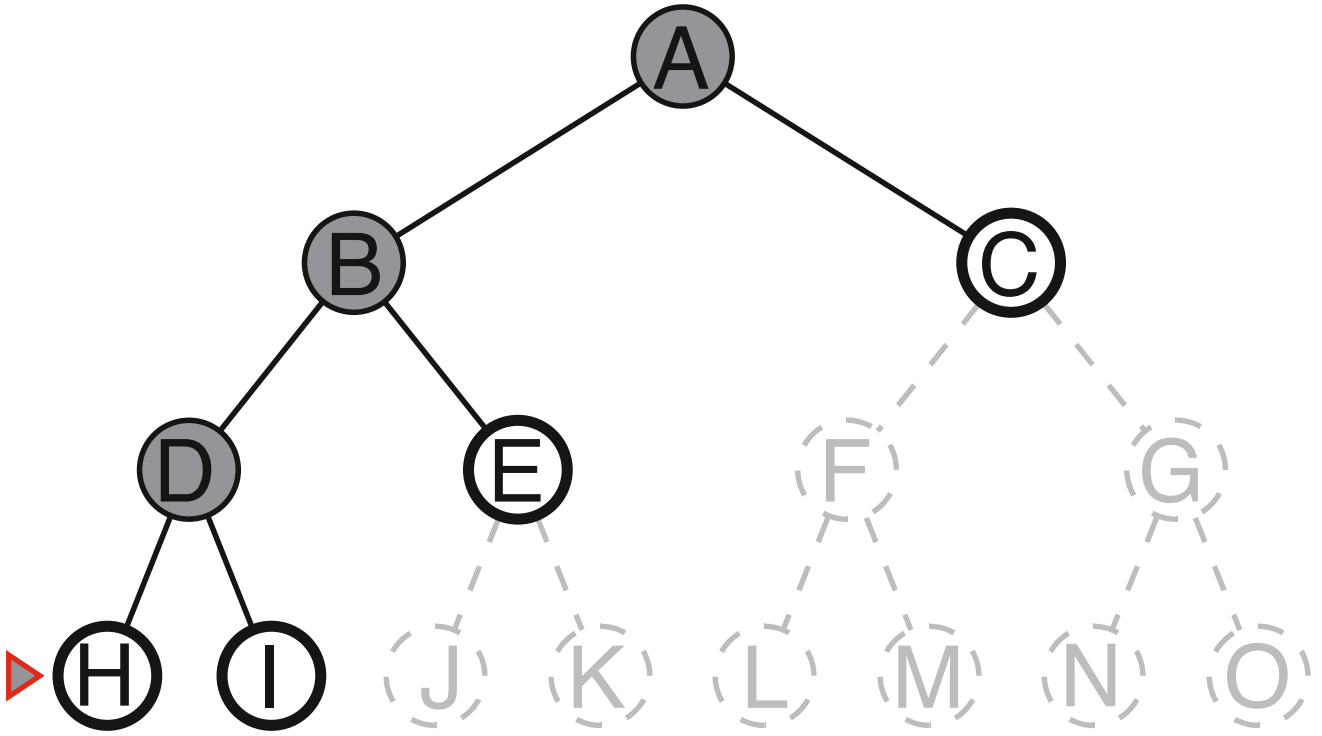


جستجوی عمق-اول

DEPTH-FIRST SEARCH

قاعده *DFS*	پرعمق‌ترین گره در کناره را گسترش دهید. *Expand deepest node in the frontier.*

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

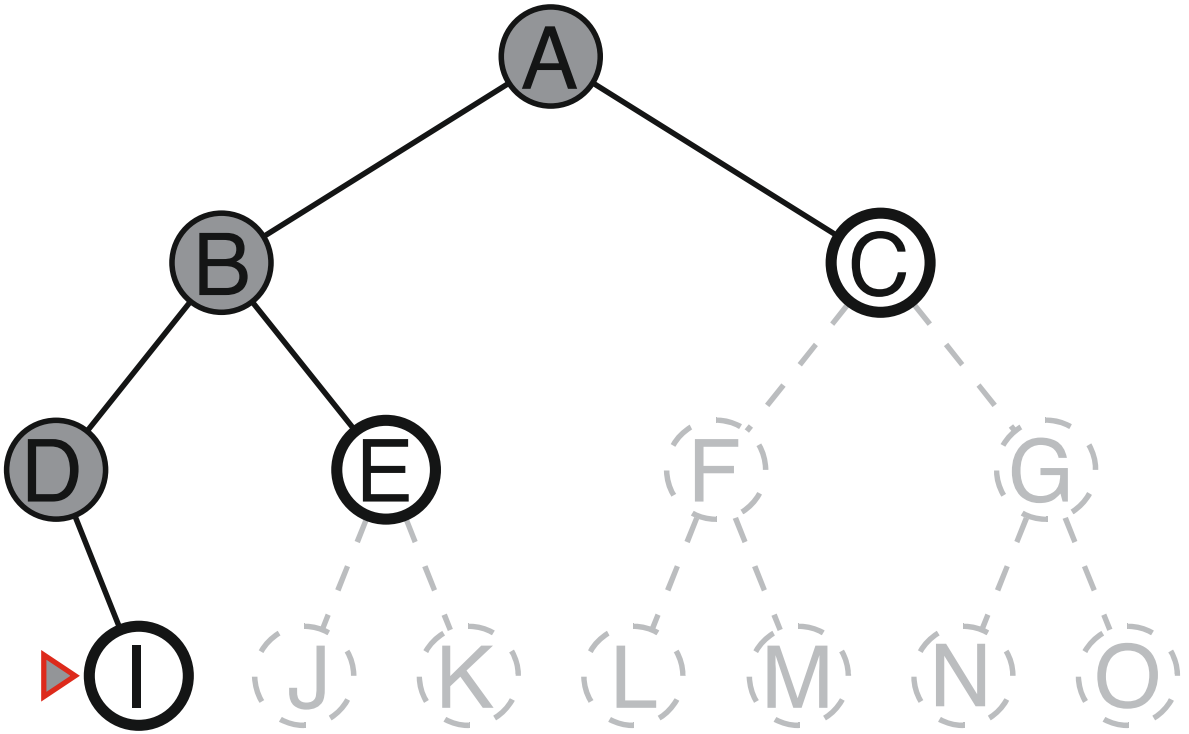


جستجوی عمق-اول

DEPTH-FIRST SEARCH

قاعده *DFS*	پرعمق‌ترین گره در کناره را گسترش دهید. *Expand deepest node in the frontier.*

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

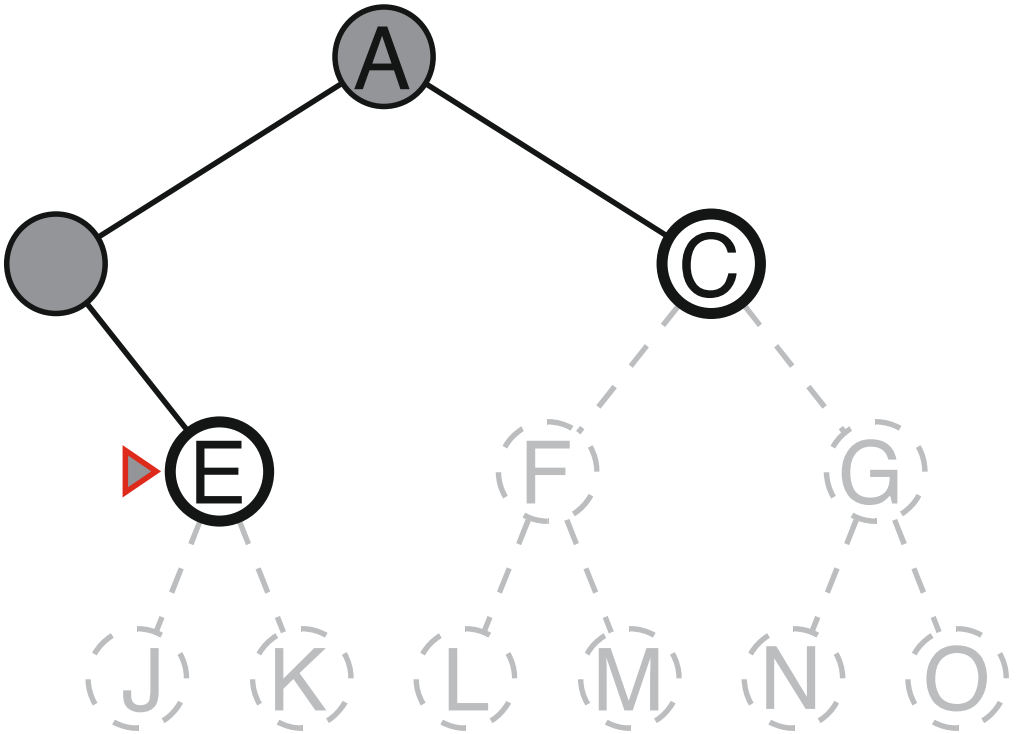


جستجوی عمق-اول

DEPTH-FIRST SEARCH

قاعده *DFS*	پرعمق‌ترین گره در کناره را گسترش دهید. *Expand deepest node in the frontier.*

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

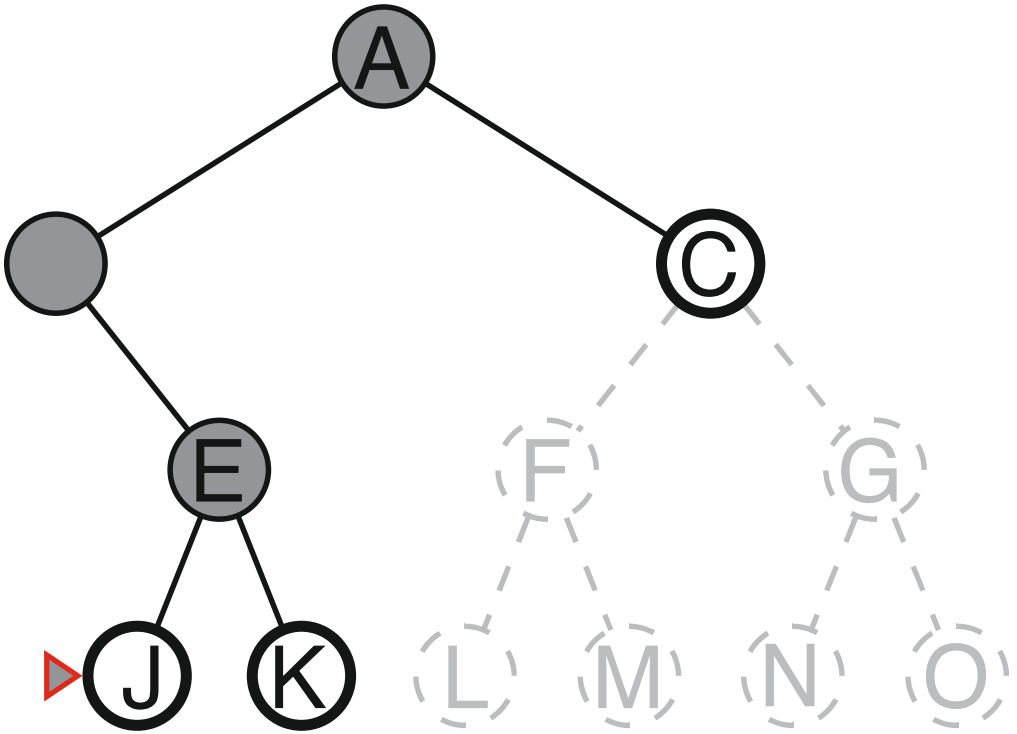


جستجوی عمق–اول

DEPTH-FIRST SEARCH

قاعده *DFS*	پرعمق‌ترین گره در کناره را گسترش دهید. *Expand deepest node in the frontier.*

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

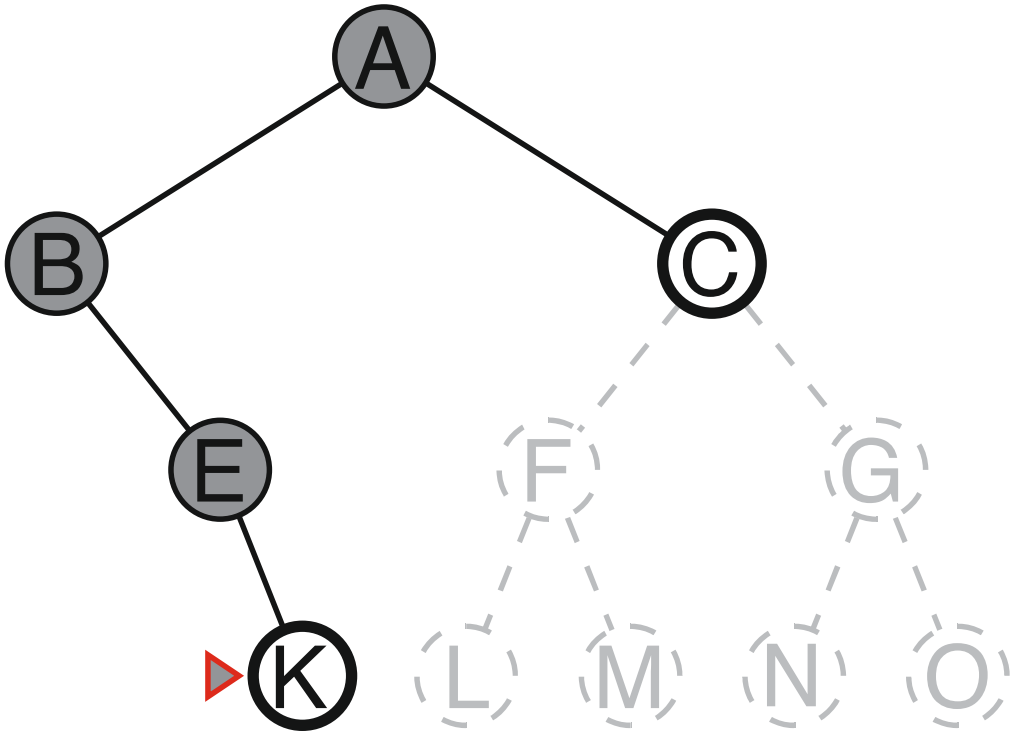


جستجوی عمق–اول

DEPTH-FIRST SEARCH

قاعده *DFS*	پرعمق‌ترین گره در کناره را گسترش دهید. *Expand deepest node in the frontier.*

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

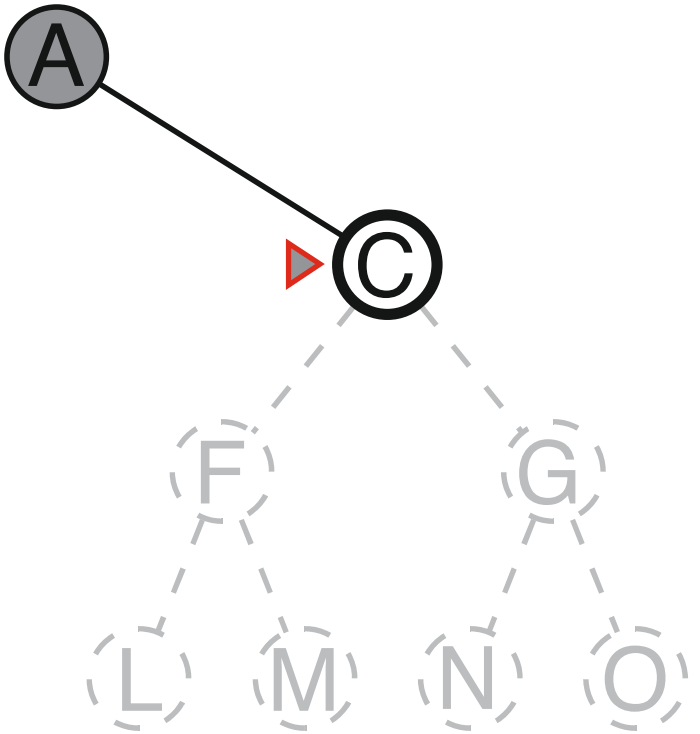


جستجوی عمق-اول

DEPTH-FIRST SEARCH

قاعده *DFS*	پرعمق‌ترین گره در کناره را گسترش دهید. *Expand deepest node in the frontier.*

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

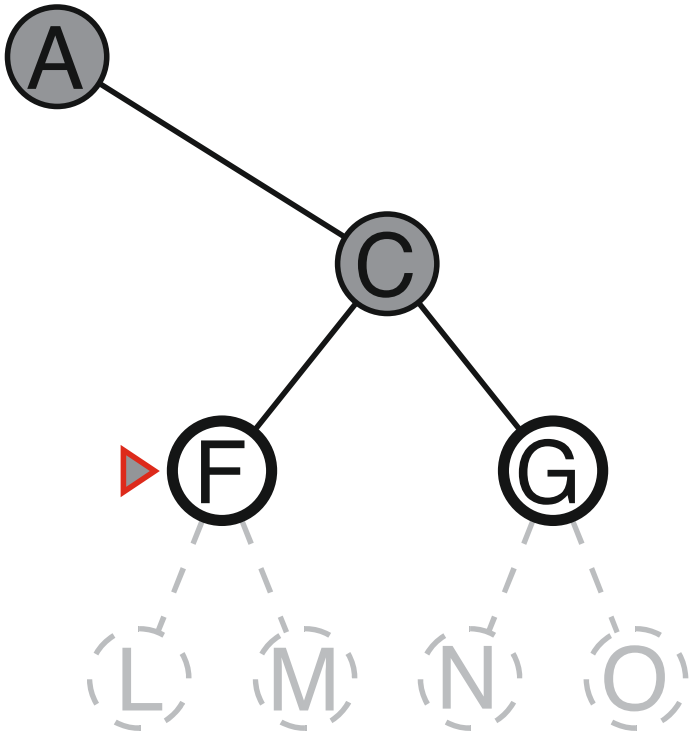


جستجوی عمق-اول

DEPTH-FIRST SEARCH

قاعده *DFS*	پرعمق‌ترین گره در کناره را گسترش دهید. *Expand deepest node in the frontier.*

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

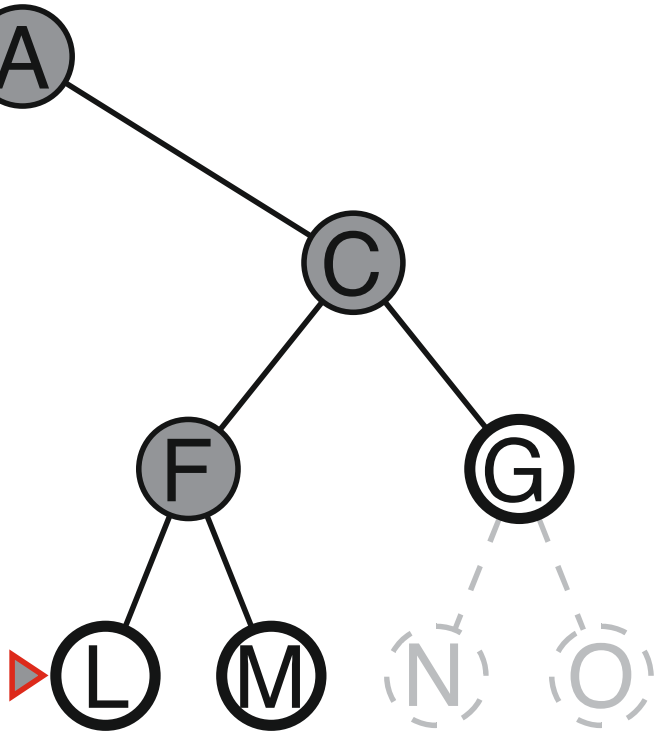


جستجوی عمق-اول

DEPTH-FIRST SEARCH

قاعده *DFS*	پرعمق‌ترین گره در کناره را گسترش دهید. *Expand deepest node in the frontier.*

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

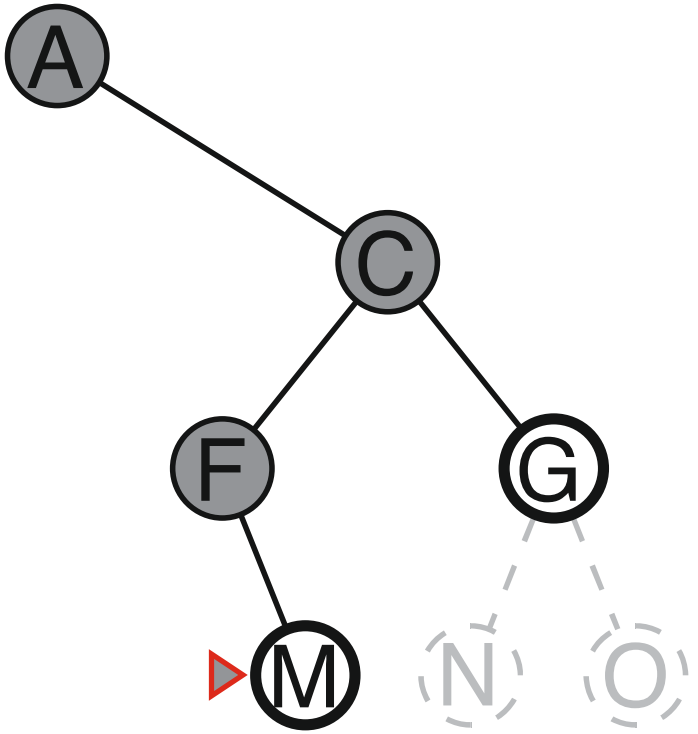


جستجوی عمق-اول

DEPTH-FIRST SEARCH

قاعده *DFS*	پرعمق‌ترین گره در کناره را گسترش دهید. *Expand deepest node in the frontier.*

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



جستجوی عمق-اول

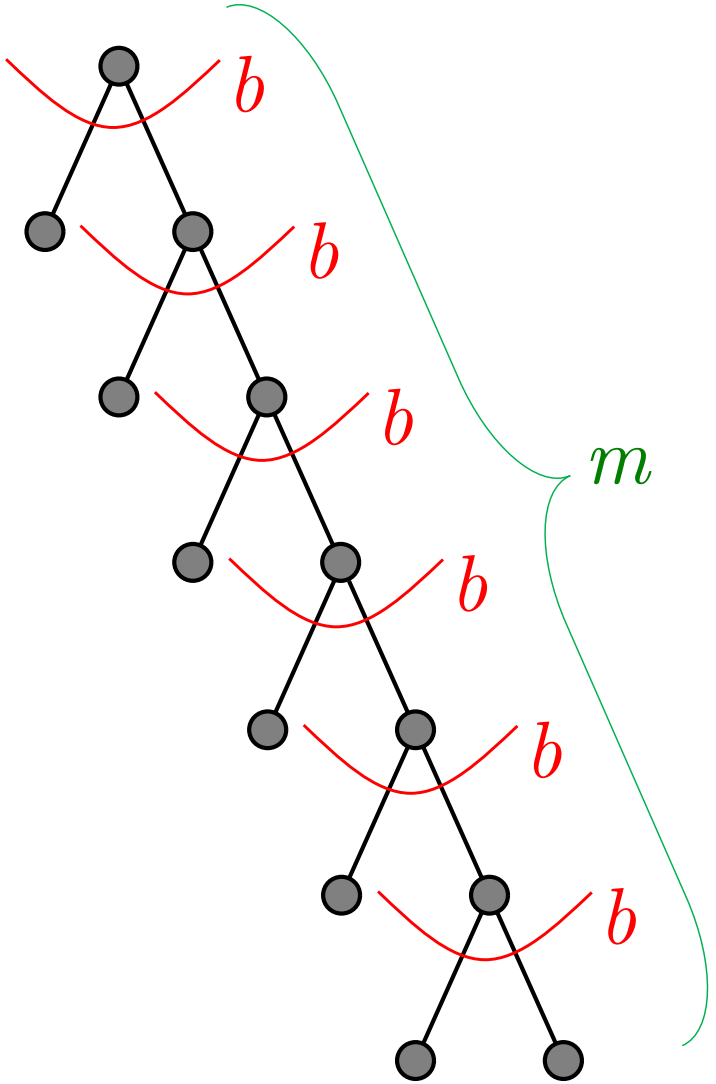
ویژگی‌ها

ارزیابی الگوریتم جستجوی عمق-اول			
۴	۳	۲	۱
پیچیدگی فضایی *Space Complexity*	**پیچیدگی زمانی** *Time Complexity*	**بهینگی هزینه** *Cost Optimality*	**تمامیت** *Completeness*
خطی	نمایی	خیر	خیر
فقط گره‌هایی که در مسیر ریشه به گره‌ی جاری قرار دارند، به همراه همزادهایشان نگهداری می‌شوند.	بدتر از عرض-اول در بدترین حالت	گره‌ی هدف با هزینه‌ی مسیر بدتر، ممکن است زودتر ملاقات شود.	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.**
$O(bm)$	$O(b^m)$ $m \gg d$		

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 (به‌جای همه‌ی فرزندان).

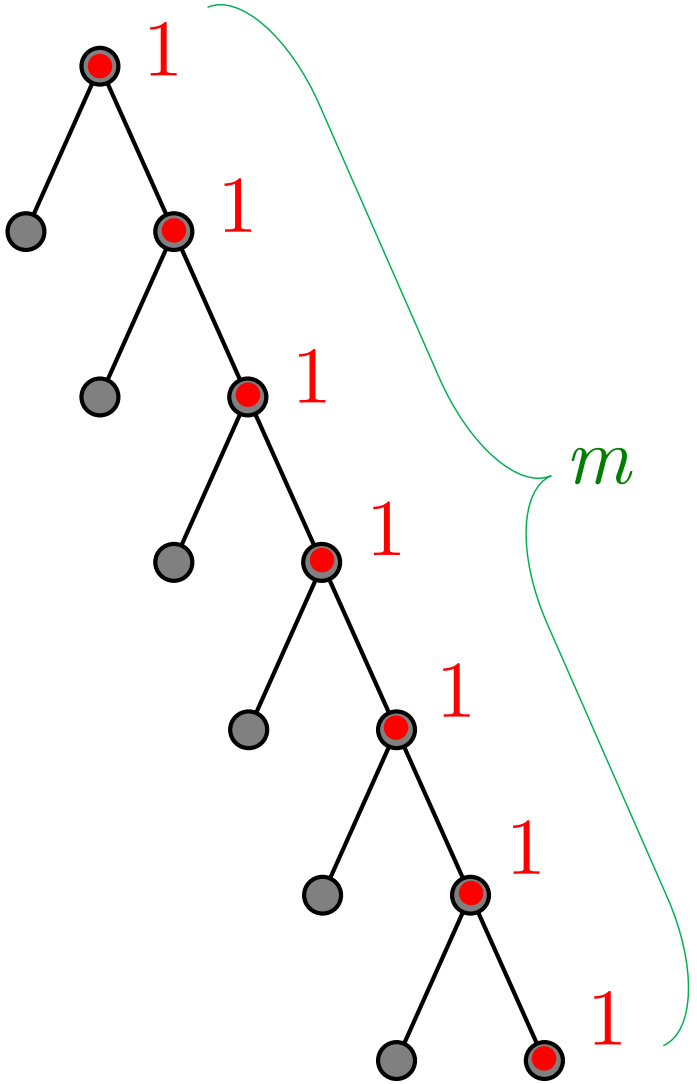
مزیت: میزان فضای مصرفی (و نیز زمان مصرفی) کمتر می‌شود.

$$O(m)$$

- هر گره‌ی والد گسترش‌داده 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 (به‌جای تولید ابتدایی آن) تولید می‌شود.

جستجوی عقب‌گرد

فضای مصرفی



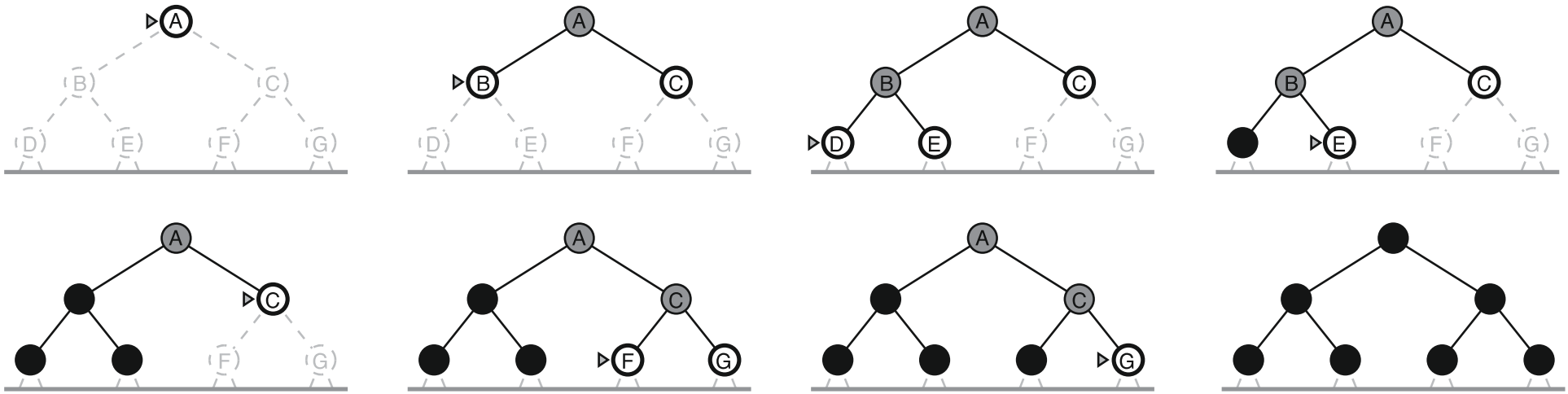
$$O(m)$$

جستجوی عمقی محدودشده

DEPTH-LIMITED SEARCH

قاعده	جستجوی عمق–اول با حد عمق l
DLS	*Execute depth-first search with depth limit 'l'.*

پیاده‌سازی: گره‌های عمق l مابعد (فرزند) ندارند.



$$l = 2$$

جستجوی عمقی محدودشده

شبه‌کد

```
function DEPTH-LIMITED-SEARCH(problem, ℓ) returns a node or failure or cutoff
  frontier ← a LIFO queue (stack) with NODE(problem.INITIAL) as an element
  result ← failure
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    if DEPTH(node) > ℓ then
      result ← 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			
۴	۳	۲	۱
پیچیدگی فضایی *Space Complexity*	**پیچیدگی زمانی** *Time Complexity*	**بهینگی هزینه** *Cost Optimality*	**تمامیت** *Completeness*
خطی	نمایی	خیر	خیر
فقط گره‌هایی که در مسیر ریشه به گره‌ی جاری قرار دارند، به همراه همزادهایشان نگهداری می‌شوند. $O(bl)$	مشابه روش عمق–اول $O(b^l)$	گره‌ی هدف با هزینه‌ی مسیر بدتر، ممکن است زودتر ملاقات شود.	مگر اینکه $l \geq d$

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)

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

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

Limit = 0

جستجوی عمیق‌کننده‌ی تکراری

جستجوی عمق–اول عمیق‌کننده‌ی تکراری

ITERATIVE DEEPENING SEARCH (ITERATIVE DEEPENING DEPTH-FIRST SEARCH)

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

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

جستجوی عمیق‌کننده‌ی تکراری

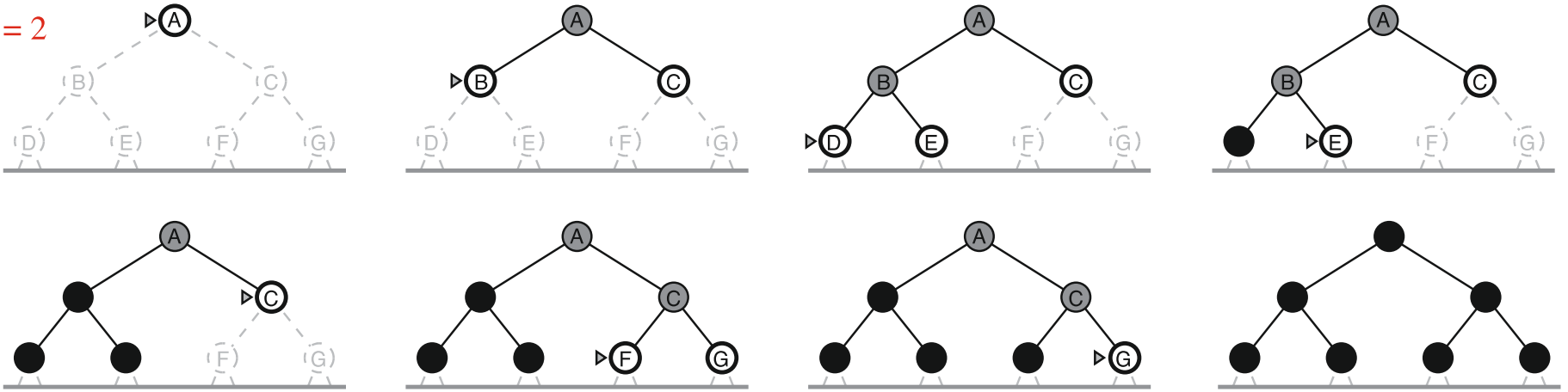
جستجوی عمق–اول عمیق‌کننده‌ی تکراری

ITERATIVE DEEPENING SEARCH (ITERATIVE DEEPENING DEPTH-FIRST SEARCH)

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

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

Limit = 2



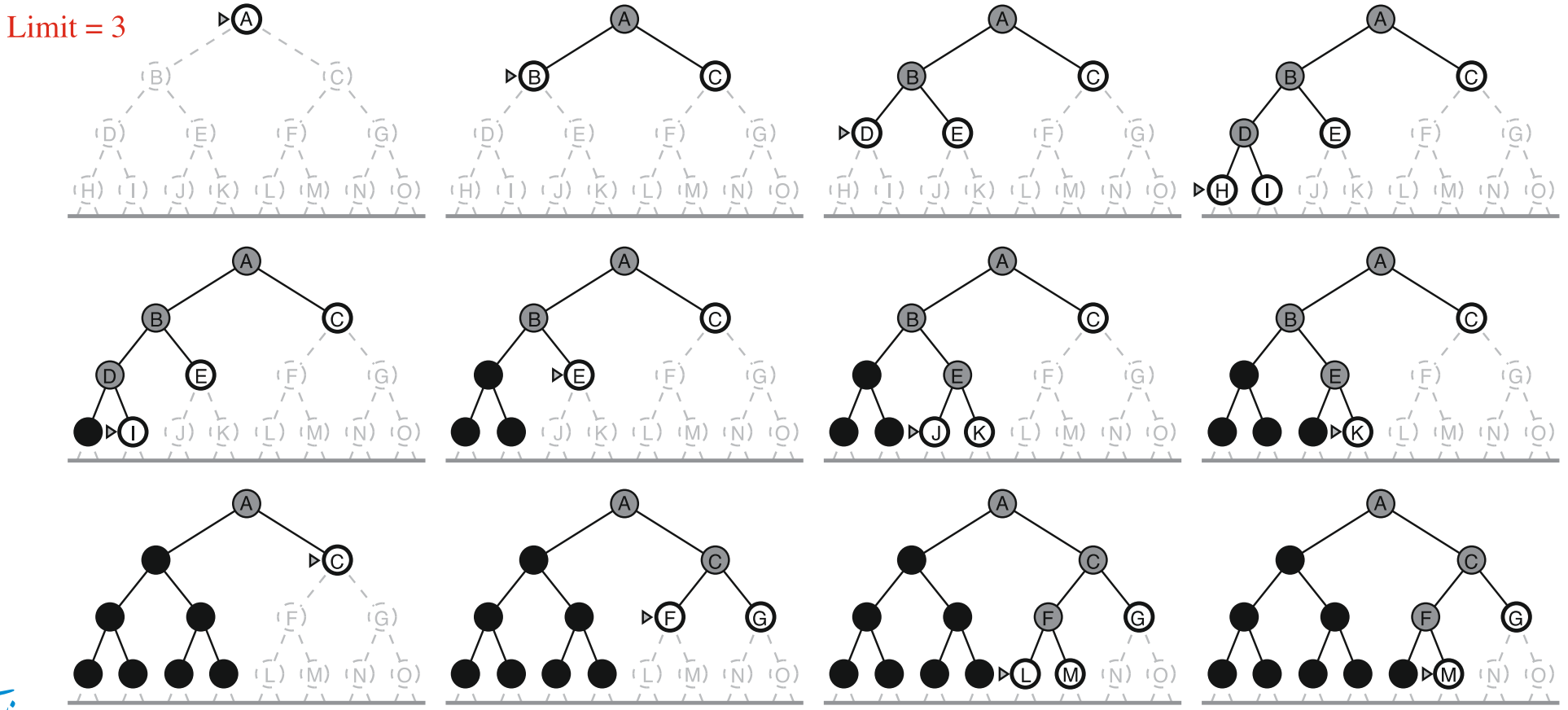
جستجوی عمیق‌کننده‌ی تکراری

جستجوی عمق–اول عمیق‌کننده‌ی تکراری

ITERATIVE DEEPENING SEARCH (ITERATIVE DEEPENING DEPTH-FIRST SEARCH)

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

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



جستجوی عمیق‌کننده‌ی تکراری

شبه‌کد

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

جستجوی عمیق‌کننده‌ی تکراری

ویژگی‌ها

ارزیابی الگوریتم جستجوی عمیق‌کننده‌ی تکراری			
۴	۳	۲	۱
پیچیدگی فضایی *Space Complexity*	**پیچیدگی زمانی** *Time Complexity*	**بهینگی هزینه** *Cost Optimality*	**تمامیت** *Completeness*
خطی	نمایی	بله	بله
مشابه روش عمق–اول $O(bd)$	$O(b^d)$	به شرط اینکه هزینه‌ی مسیر هر گره تابعی غیرنزولی از عمق گره باشد (مثلاً هزینه‌ی گام ۱ باشد).	

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

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

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

مقایسه: BFS با IDS

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

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

بررسی هدف به محض تولید گره $N(\text{BFS}) = 10 + 100 + 1,\ 000 + 10{,}000 + 100{,}000 = 111{,}110$

بررسی هدف پس از انتخاب از کناره $N(\text{BFS}) = 10 + 100 + 1{,}000 + 10{,}000 + 100{,}000 + 999{,}990 = 1{,}111{,}100$

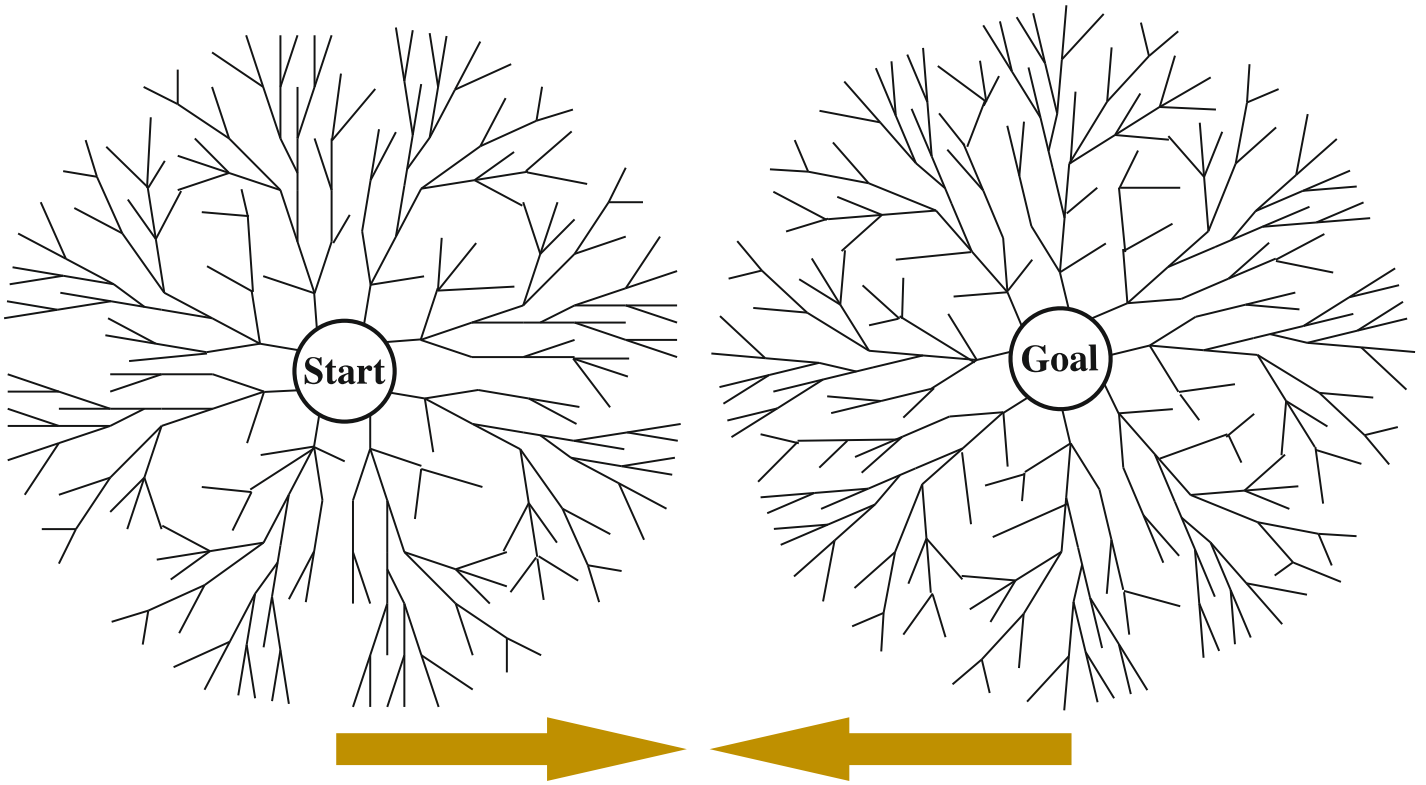
پیچیدگی زمانی به‌لحاظ مجانبی مشابه روش عرض–اول است.
مشکل: مقداری هزینه‌ی اضافی برای تولید چندباره‌ی سطوح بالایی درخت وجود دارد، **اما** زیاد بزرگ نیست؛
و با بزرگ‌تر شدن b این نسبت کوچکتر می‌شود.

جستجوی دوطرفه

BIDIRECTIONAL SEARCH

قاعده *BDS*	دو جستجو به طور همزمان انجام می‌شود: از حالت اولیه به جلو (پیشرو) + از حالت هدف به عقب (پسرو)

پیاده‌سازی: مطابق قاعده: زمانی که دو جستجو به هم می‌رسند، متوقف می‌شویم.



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 ← 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
```

جستجوی دوطرفه

ویژگی‌ها

ارزیابی الگوریتم جستجوی دوطرفه			
۴	۳	۲	۱
پیچیدگی فضایی *Space Complexity*	**پیچیدگی زمانی** *Time Complexity*	**بهینگی هزینه** *Cost Optimality*	**تمامیت** *Completeness*
نمایی	نمایی	بله	بله
مشابه روش عرض–اول، اما فاکتور انشعاب جذر می‌شود.	مشابه روش عرض–اول، اما فاکتور انشعاب جذر می‌شود. $O(2b^{d/2}) = O(b^{d/2}) = O((\sqrt{b})^d)$	به شرط اینکه در دو جهت از BFS استفاده شود و هزینه‌ی مسیر هر گره تابعی غیرنزولی از عمق گره باشد.	به شرط اینکه در دو جهت از BFS استفاده شود و b متناهی باشد.

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	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
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.

مقایسه‌های فوق برای جستجوهای درختی است که وقوع حالت‌های تکراری را بررسی نمی‌کنند.
برای جستجوی گرافی که وقوع حالت‌های تکراری را بررسی می‌کنند:
جست‌وجوی عمق اول برای فضاهای حالت متناهی تمامیت دارد و پیچیدگی‌های زمانی و فضایی به اندازه‌ی فضای حالت محدود می‌شود (تعداد رأس‌ها و یال‌ها)

بسم الله الرحمن الرحیم

هوش مصنوعی

فصل ۳

حل مسئله با جستجو (۳)

Solving Problems by Searching (3)

کاظم فولادی قلعه
دانشکده مهندسی، پردیس فارابی
دانشگاه تهران

http://courses.fouladi.ir/ai

هوش مصنوعی

حل مسئله با جستجو

۵

استراتژی‌های جستجوی آگاهانه (هیوریستیک)

استراتژی‌های جستجوی آگاهانه

جستجوی هیوریستیک

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

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

جستجوی بهترین–اول

BEST-FIRST SEARCH

قاعده *Best-FS*	بهترین گره در کناره را گسترش دهید. *Expand best node in the frontier.*

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

جستجوی بهترین–اول
Best-First Search

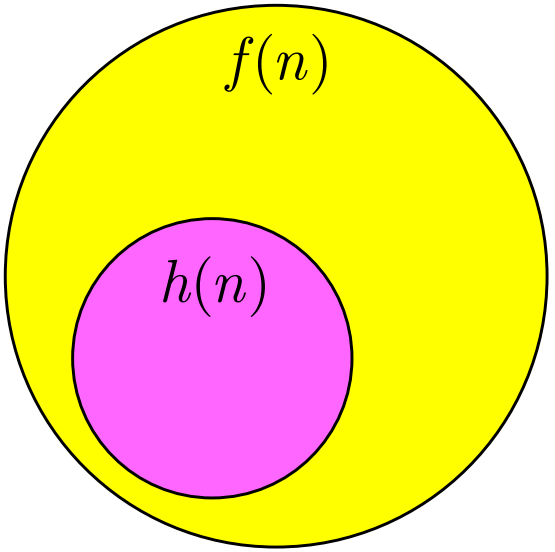
- **جستجوی A***
  A Search*
- **جستجوی حریصانه**
  Greedy Search

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 می‌شود.



$$f(n)$$

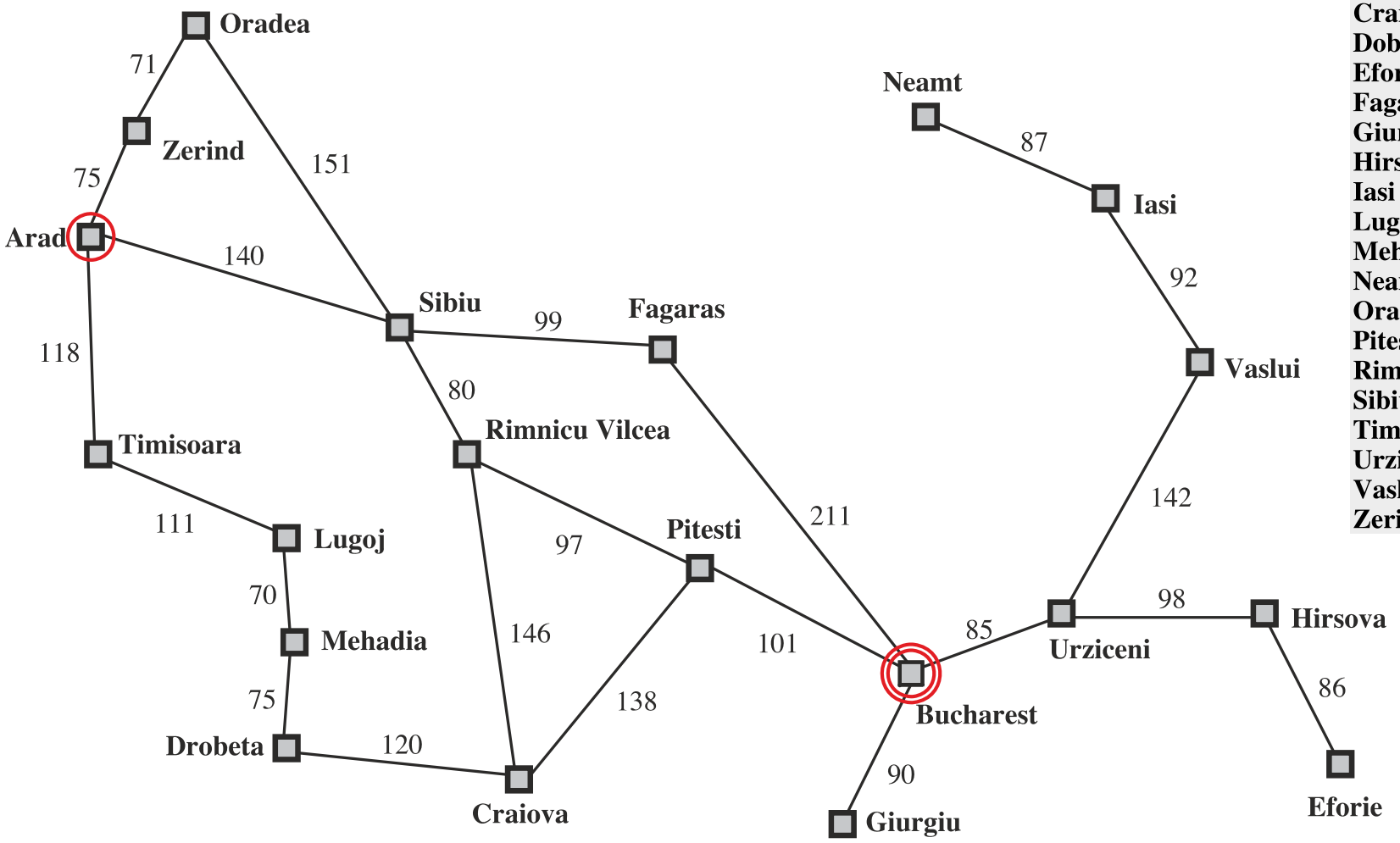
تابع هیوریستیک
Heuristic Function

$$h(n)$$

(مهم‌ترین مؤلفه‌ی تابع ارزیابی)
تخمین مطلوب بودن یک گره برای رسیدن به یک هدف

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)

قاعده *GBS*	گره‌ای را گسترش بدهید که به نظر می‌رسد به هدف نزدیک‌تر است. *Expand the node that appears to be closest to goal.*

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

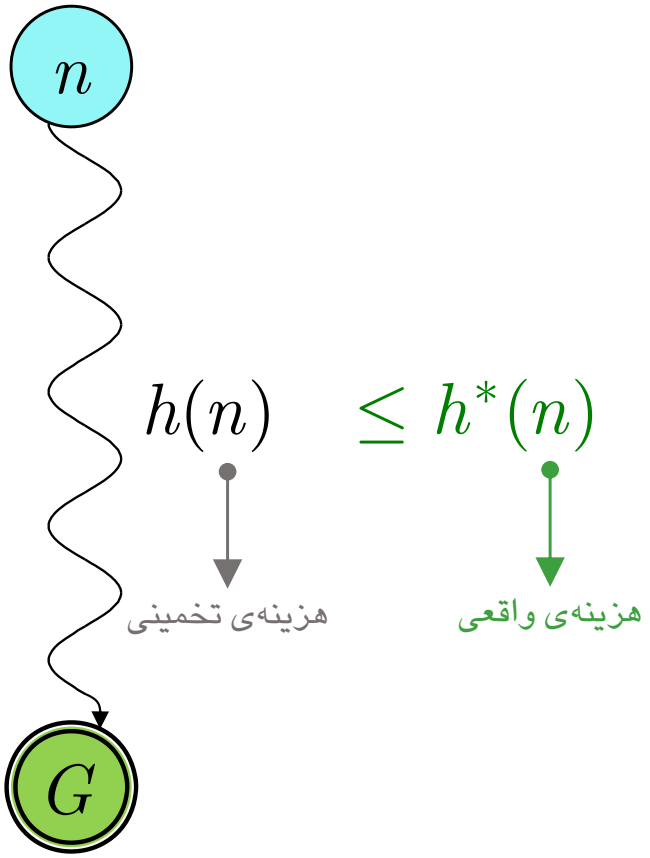
$$f(n) = h(n)$$

میزان هزینه تا هدف

تابع هیوریستیک

HEURISTIC FUNCTION

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



برای هر گره‌ی هدف $h(G) = 0$

جستجوی حریصانه

جستجوی بهترین–اول حریصانه

GREEDY SEARCH (GREEDY BEST-FIRST SEARCH)

قاعده *GBS*	گره‌ای را گسترش بدهید که به نظر می‌رسد به هدف نزدیک‌تر است. *Expand the node that appears to be closest to goal.*

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

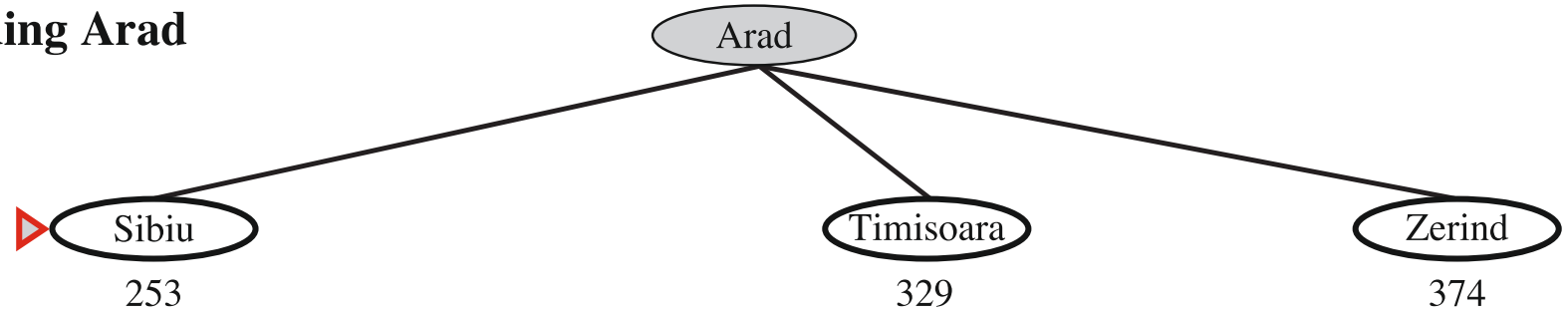
GREEDY SEARCH (GREEDY BEST-FIRST SEARCH)

قاعده GBS	گره‌ای را گسترش بدهید که به نظر می‌رسد به هدف نزدیک‌تر است. *Expand the node that appears to be closest to goal.*

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

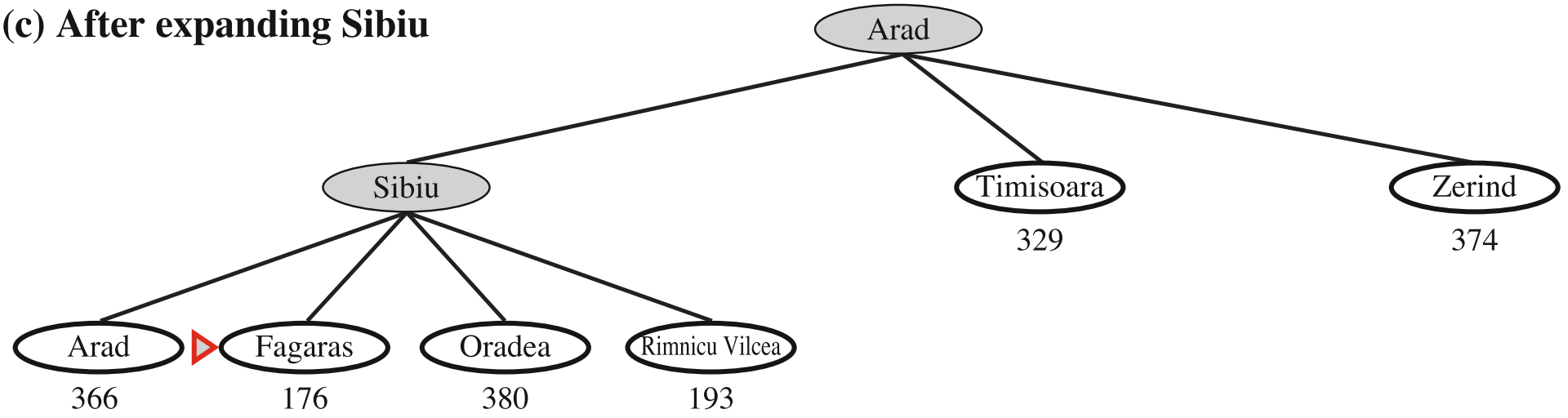
GREEDY SEARCH (GREEDY BEST-FIRST SEARCH)

قاعده *GBS*	گره‌ای را گسترش بدهید که به نظر می‌رسد به هدف نزدیک‌تر است. *Expand the node that appears to be closest to goal.*

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

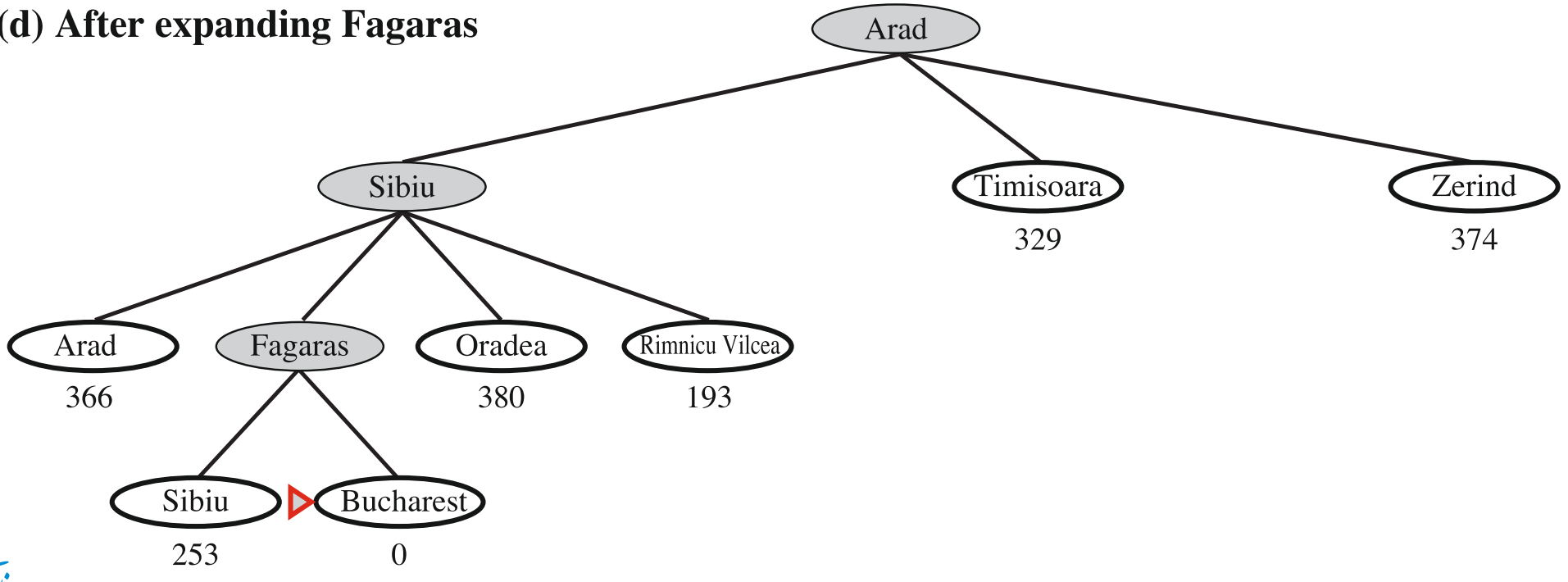
GREEDY SEARCH (GREEDY BEST-FIRST SEARCH)

قاعده GBS	گره‌ای را گسترش بدهید که به نظر می‌رسد به هدف نزدیک‌تر است. *Expand the node that appears to be closest to goal.*

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

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

جستجوی A*

A* SEARCH

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

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

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

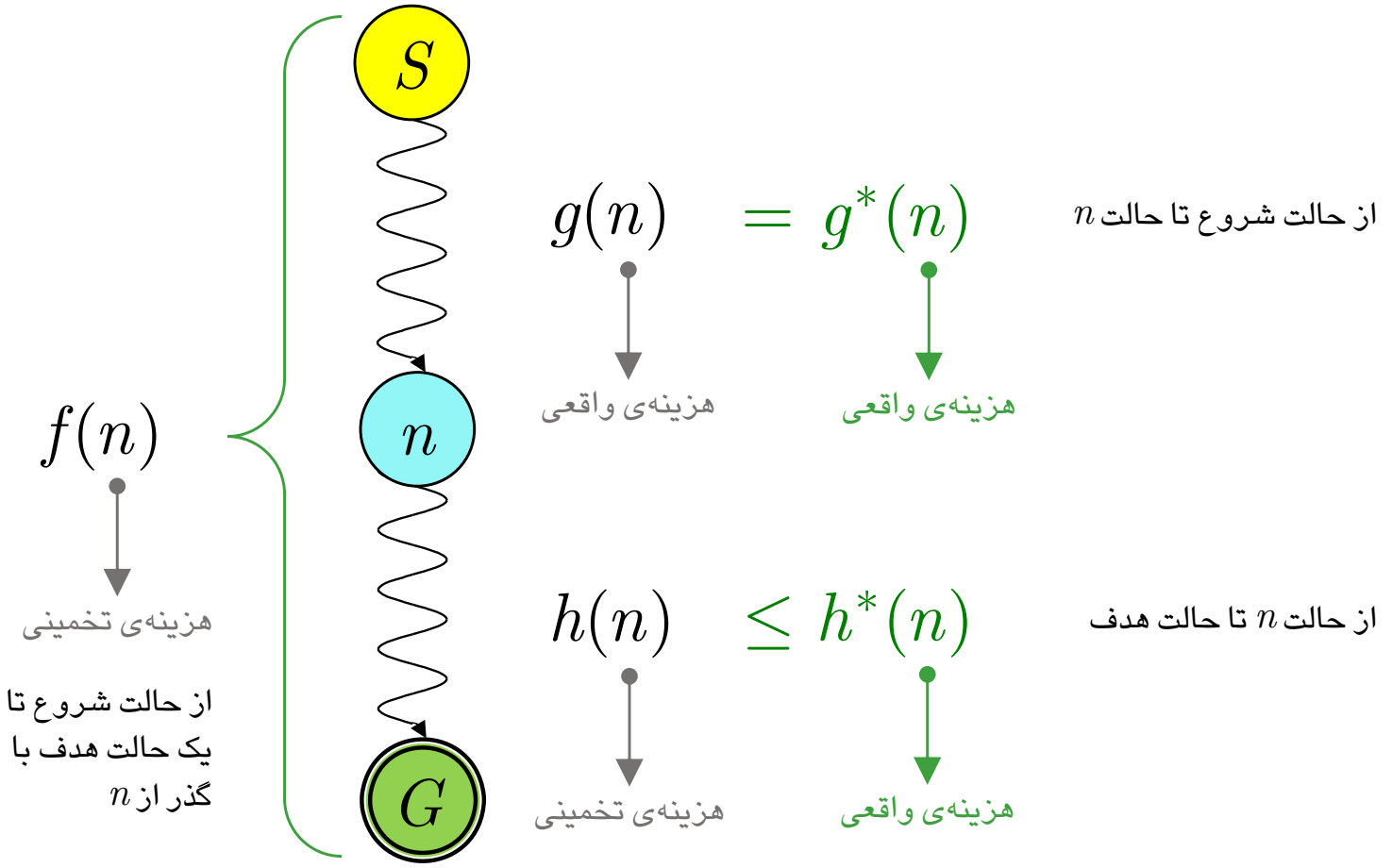
اساس روش: اجتناب از گسترش مسیرهایی که هم‌اکنون گران هستند.

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

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

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

(a) The initial state

366=0+366

$$f = g + h$$

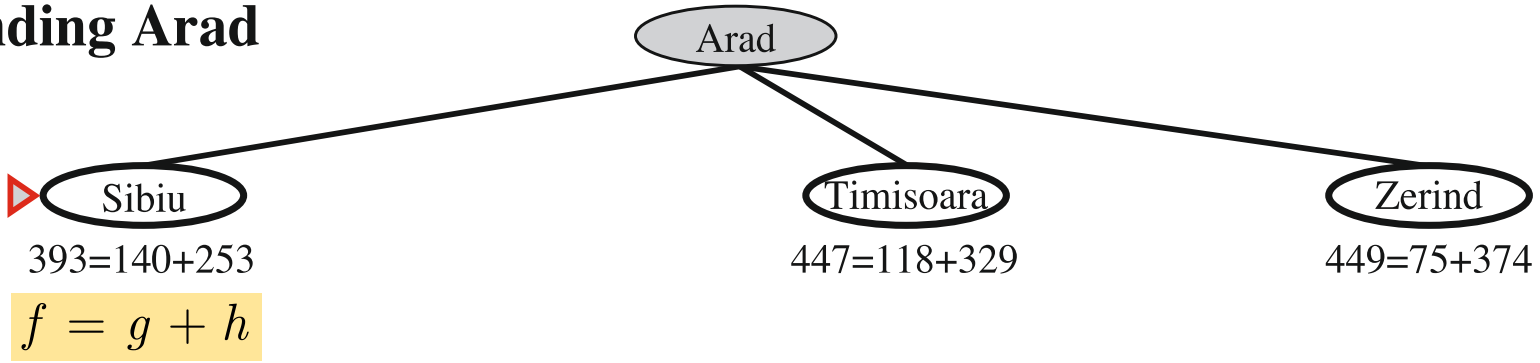
جستجوی A*

A* SEARCH

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

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

(b) After expanding Arad



$f = g + h$

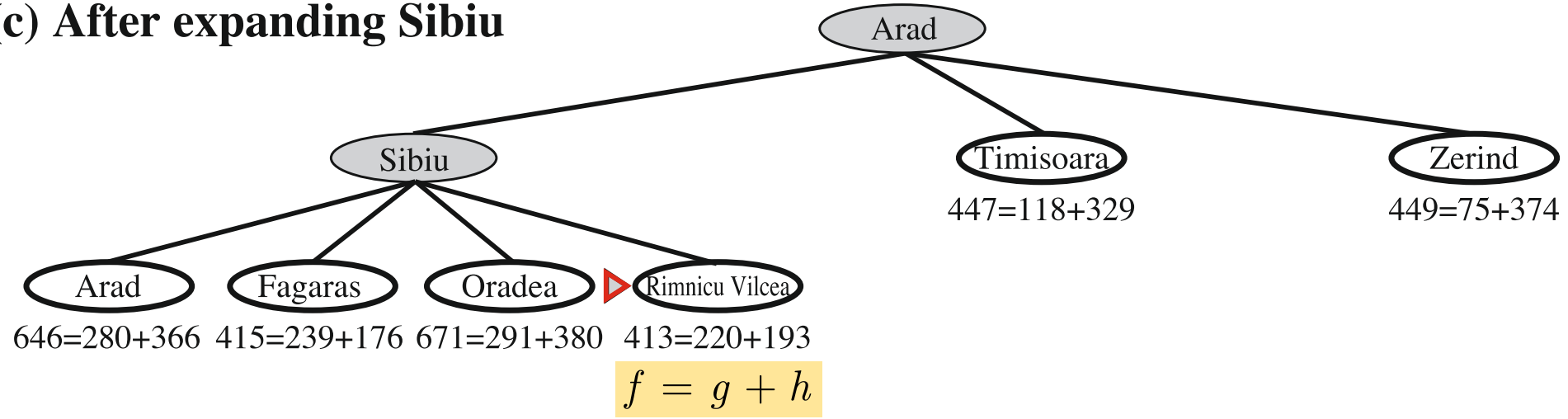
جستجوی A*

A* SEARCH

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

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

(c) After expanding Sibiu



$$f = g + h$$

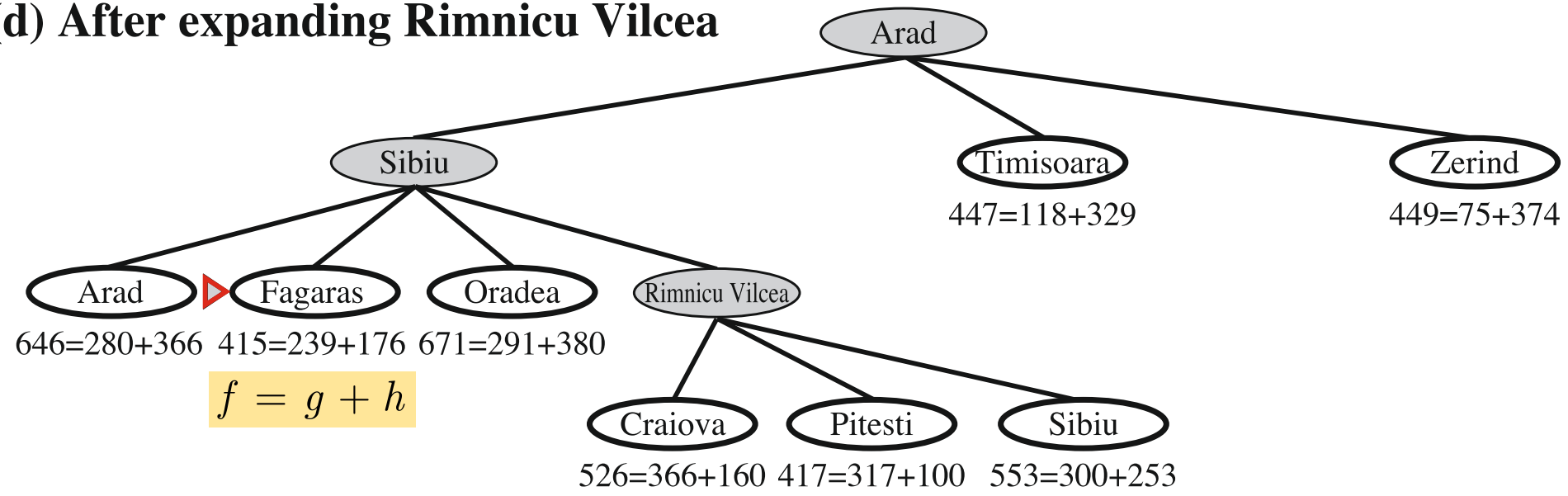
جستجوی A*

A* SEARCH

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

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

(d) After expanding Rimnicu Vilcea



$f = g + h$

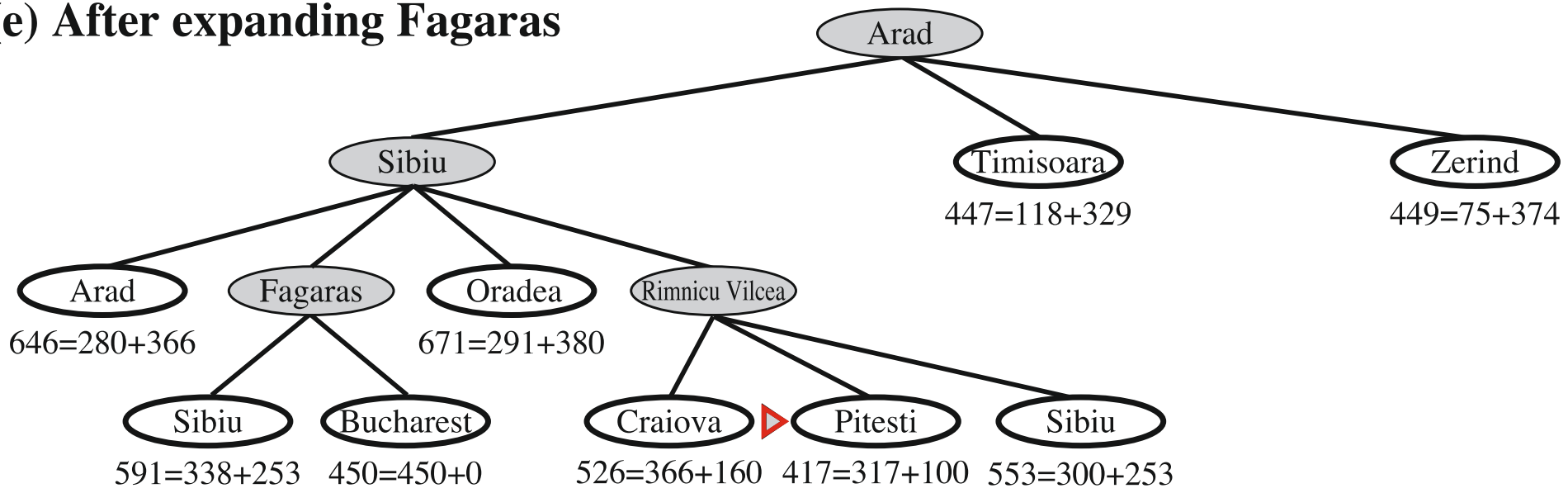
جستجوی A*

A* SEARCH

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

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

(e) After expanding Fagaras



$f = g + h$

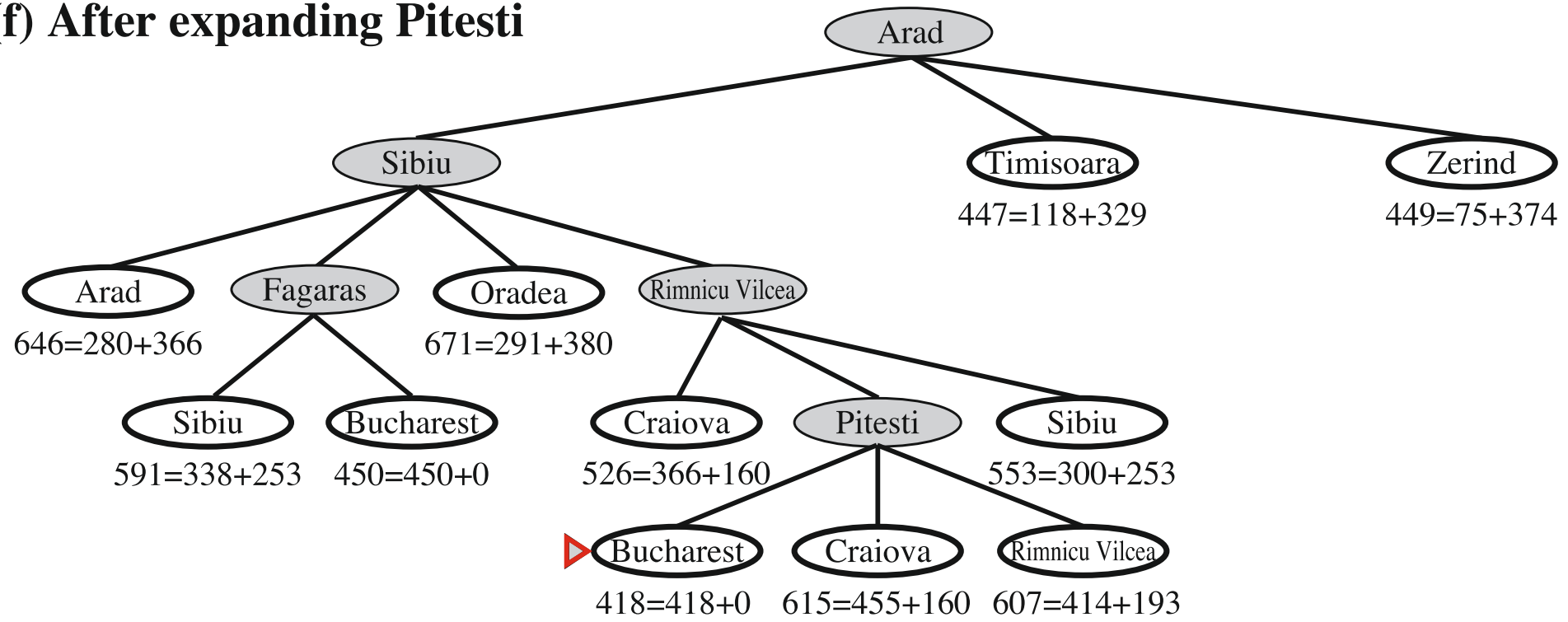
جستجوی A*

A* SEARCH

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

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

(f) After expanding Pitesti



$f = g + h$

جستجوی A*

ویژگی‌ها

ارزیابی الگوریتم جستجوی A*			
۴	۳	۲	۱
پیچیدگی فضایی *Space Complexity*	**پیچیدگی زمانی** *Time Complexity*	**بهینگی هزینه** *Cost Optimality*	**تمامیت** *Completeness*
نمایی	نمایی	بله	بله
$O(b^{\epsilon d})$	$O(b^{\epsilon d})$ $\epsilon = (h^* - h) / h^*$	گره‌های دارای هزینه‌ی بیشتر گسترش داده نمی‌شوند، مگر اینکه گره‌های دارای هزینه‌ی کمتر گسترش داده شده باشند.	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.
همه‌ی گره‌ها را در حافظه نگه می‌دارد.	نمایی بر حسب خطای نسبی در h ضرب در عمق راه‌حل		

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

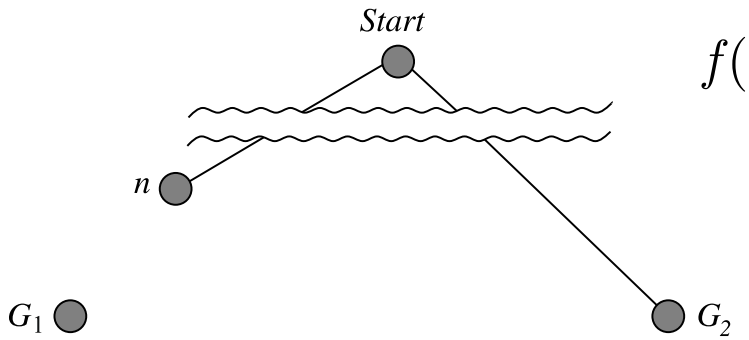
قضایای A*: بهینه بودن A*

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 گسترش‌نیافته بر روی کوتاهترین مسیر به سوی یک هدف بهینه‌ی 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.

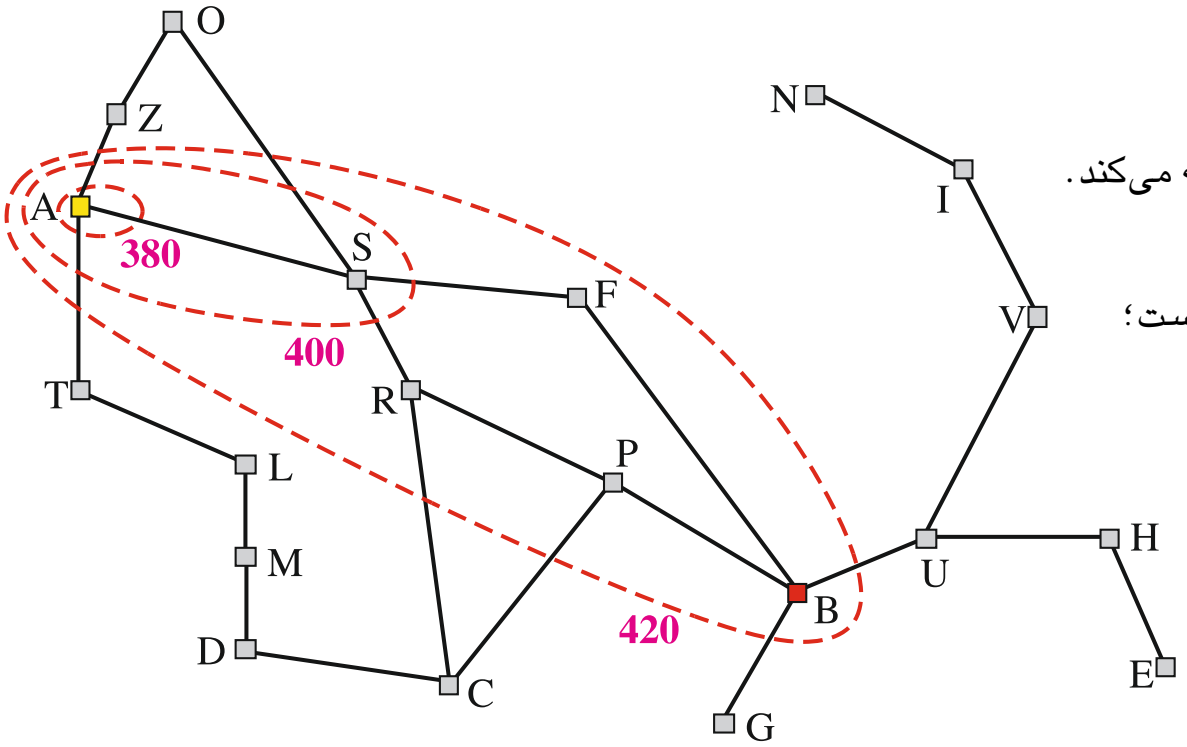
$$
\begin{aligned}
f(n) &> C^* \quad (\text{otherwise } n \text{ would have been expanded}) \\
f(n) &= g(n) + h(n) \quad (\text{by definition}) \\
f(n) &= g^*(n) + h(n) \quad (\text{because } n \text{ is on an optimal path}) \\
f(n) &\leq g^*(n) + h^*(n) \quad (\text{because of admissibility}, h(n) \leq h^*(n)) \\
f(n) &\leq C^* \quad (\text{by definition}, C^* = g^*(n) + h^*(n))
\end{aligned}
$$

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.)

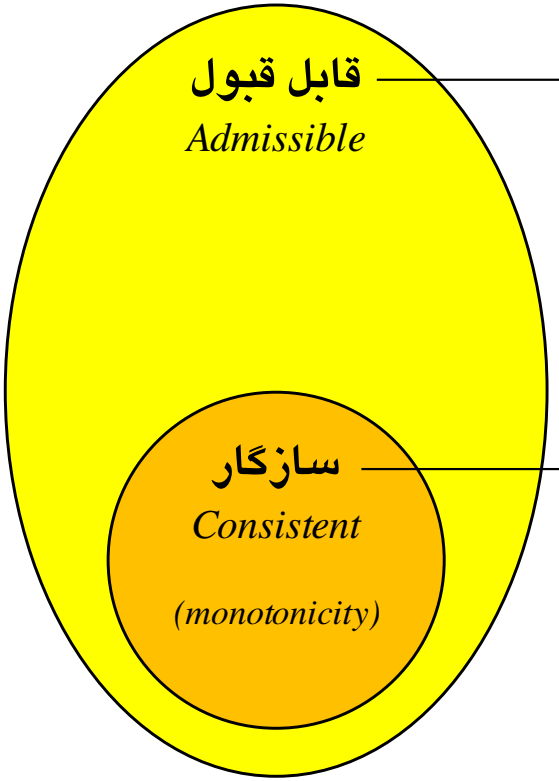
کانتور i شامل همه‌ی گره‌ها با $f = f_i$ است؛
که در آن $f_i \leq f_{i+1}$

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

C^* هزینه‌ی راه‌حل بهینه است.

خواص تابع هیوریستیک

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



هیچ‌گاه «بیش‌برآورد» overestimate نمی‌کند.

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

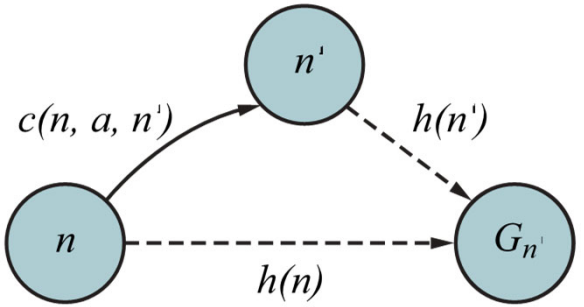
$$\Rightarrow h(G) = 0 \quad \text{(for any goal state } G\text{)}$$

از نامساوی مثلثی تبعیت می‌کند.

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

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

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



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

اگر $h(n)$	**قابل قبول** *Admissible*	باشد، الگوریتم جستجوی A* با استفاده از TREE-SEARCH بهینه است.
اگر $h(n)$	**سازگار** *Consistent*	باشد، الگوریتم جستجوی A* با استفاده از GRAPH-SEARCH بهینه است.

اگر $h(n)$	**سازگار** *Consistent*	باشد، آن‌گاه	**قابل قبول** *Admissible*	خواهد بود.
اگر $h(n)$	**سازگار** *Consistent*	باشد، مقدار $f(n)$ در امتداد هر مسیری غیرنزولی است (نامساوی مثلثی).		

جستجوی ارضاکننده

هیوریستیک‌های غیرقابل قبول و A* وزن‌دار

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 راه‌حل‌های زیربهینه اما «به‌اندازه‌ی 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.

هیوریستیک غیر قابل قبول
Inadmissible Heuristic

↓

جستجوی A* وزن‌دار
Weighted A Search*

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.

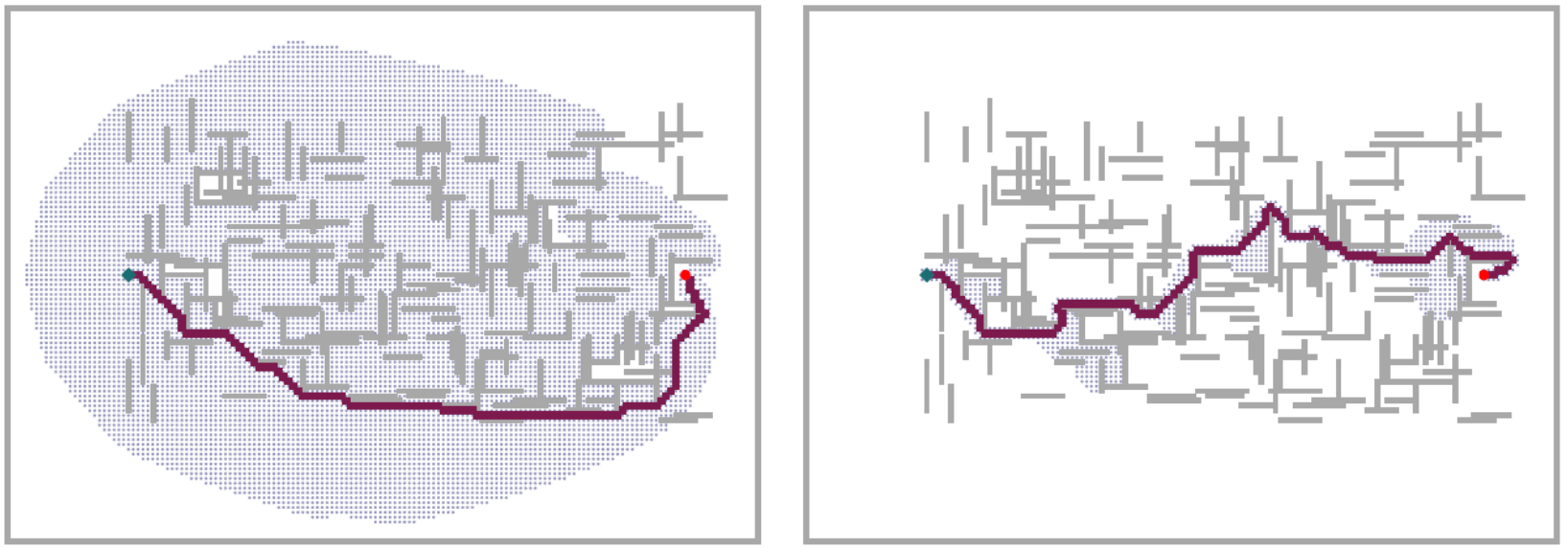
$$f(n) = g(n) + W \times h(n), \text{ for some } W > 1.$$

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.

جستجوی ارضاکننده

هیوریستیک‌های غیرقابل قبول و A* وزن‌دار: مثال

SATISFICING 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* وزن‌دار

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:

$$\begin{aligned}
\text{A* search:} &\quad g(n) + h(n) &\quad (W = 1)\\
\text{Uniform-cost search:} &\quad g(n) &\quad (W = 0)\\
\text{Greedy best-first search:} &\quad h(n) &\quad (W = \infty)\\
\text{Weighted A* search:} &\quad g(n) + W \times h(n) &\quad (1 < W < \infty)
\end{aligned}$$

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 زیربهینه

SUBOPTIMAL SEARCH ALGORITHMS

الگوریتم‌های جستجوی زیربهینه
Suboptimal Search Algorithms

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

جستجوی زیربهینه‌ی کران‌دار
Bounded Suboptimal Search

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

جستجوی هزینه–کران دار
Bounded-Cost Search

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

جستجوی هزینه–بی‌کران
Unbounded-Cost Search

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

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

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.**

راه حل

استفاده از روش‌های جستجوی هیوریستیک حافظه–محدود

جستجوی هیوریستیک حافظه–محدود
Memory-bounded Heuristic Search

- **جستجوی پرتوی** *Beam Search*
- **جستجوی A* عمیق‌کننده‌ی تکراری** *Iterative deepening A* (IDA*)*
- **جستجوی بهترین–اول بازگشتی** *Recursive Best First Search (RBFS)*
- **جستجوی A* حافظه–محدود** *Memory-bounded A* (MA*)*
- **جستجوی A* حافظه–محدود ساده‌شده** *Simplified Memory-bounded A* (SMA*)*

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

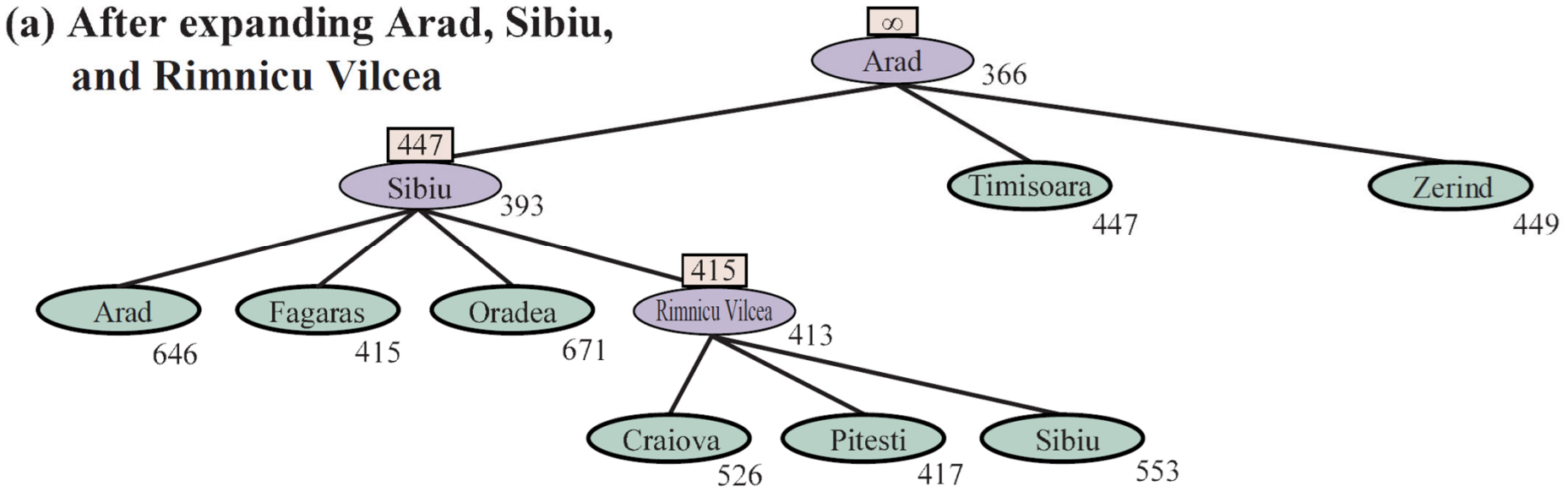
```
function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution or failure
    solution, fvalue ← RBFS(problem, NODE(problem.INITIAL), ∞)
  return solution

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 ← 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 ← max(s.PATH-COST + h(s), node.f))
  while true do
      best ← the node in successors with lowest f-value
      if best.f > f limit then return failure, best.f
      alternative ← the second-lowest f-value among successors
      result, best.f ← RBFS(problem, best, min(f_limit, alternative))
      if result ≠ failure then return result, best.f
```

جستجوی بهترین–اول بازگشتی

مثال (۱ از ۳)

RECURSIVE BEST-FIRST SEARCH

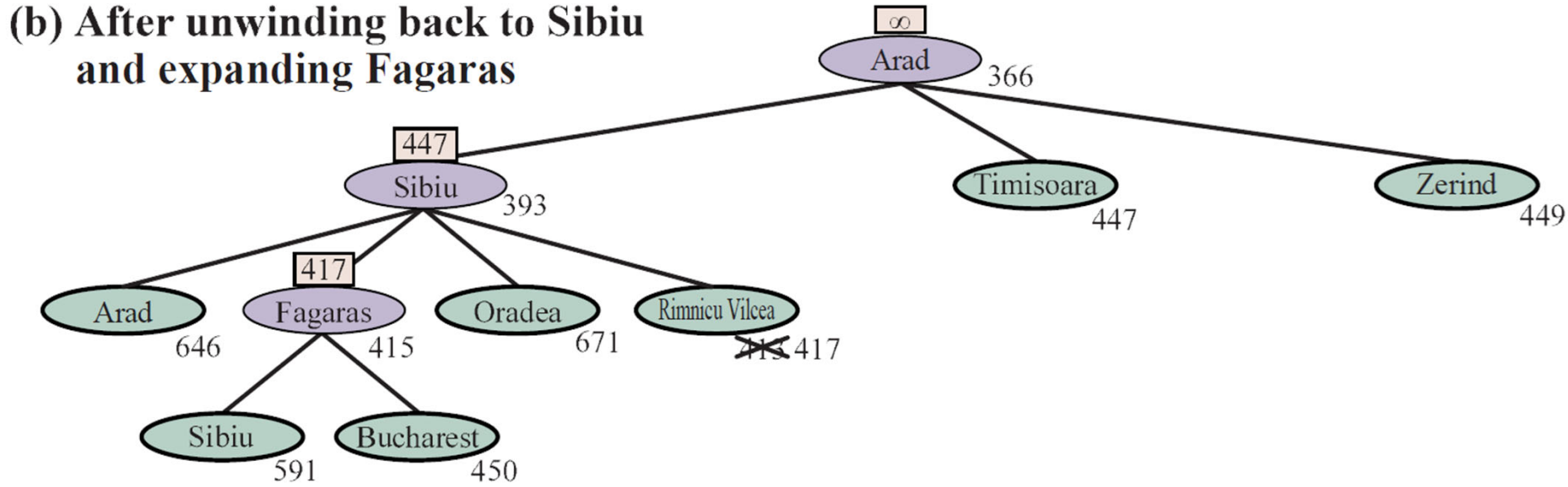


Stages in an RBFS search for the shortest route to Bucharest.
The f-*limit* value for each recursive call is shown on top of each current node, and every node is labeled with its f-cost.
(a) The path via Rimnicu Vilcea is followed until the current best leaf (Pitesti) has a value that is worse than the best alternative path (Fagaras).

جستجوی بهترین–اول بازگشتی

مثال (۲ از ۳)

RECURSIVE BEST-FIRST SEARCH

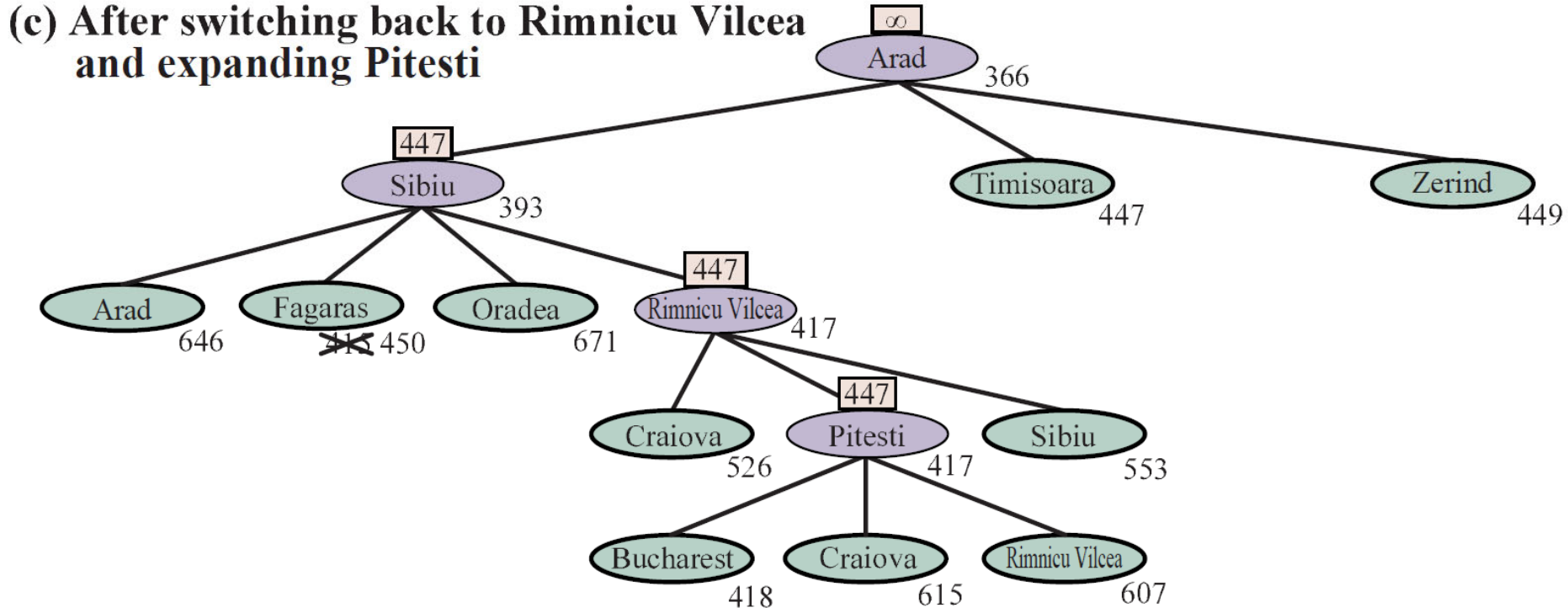


(b) The recursion unwinds and the best leaf value of the forgotten subtree (417) is backed up to Rimnicu Vilcea; then Fagaras is expanded, revealing a best leaf value of 450.

جستجوی بهترین–اول بازگشتی

مثال (۳ از ۳)

RECURSIVE BEST-FIRST SEARCH



(c) 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 دوطرفه

BIDIRECTIONAL HEURISTIC SEARCH

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

برای گره‌های در جهت پیشرو (forward)
حالت آغازین به‌عنوان ریشه

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

برای گره‌های در جهت پسرو (backward)
حالت هدف به‌عنوان 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 دوطرفه

مثال

BIDIRECTIONAL HEURISTIC SEARCH

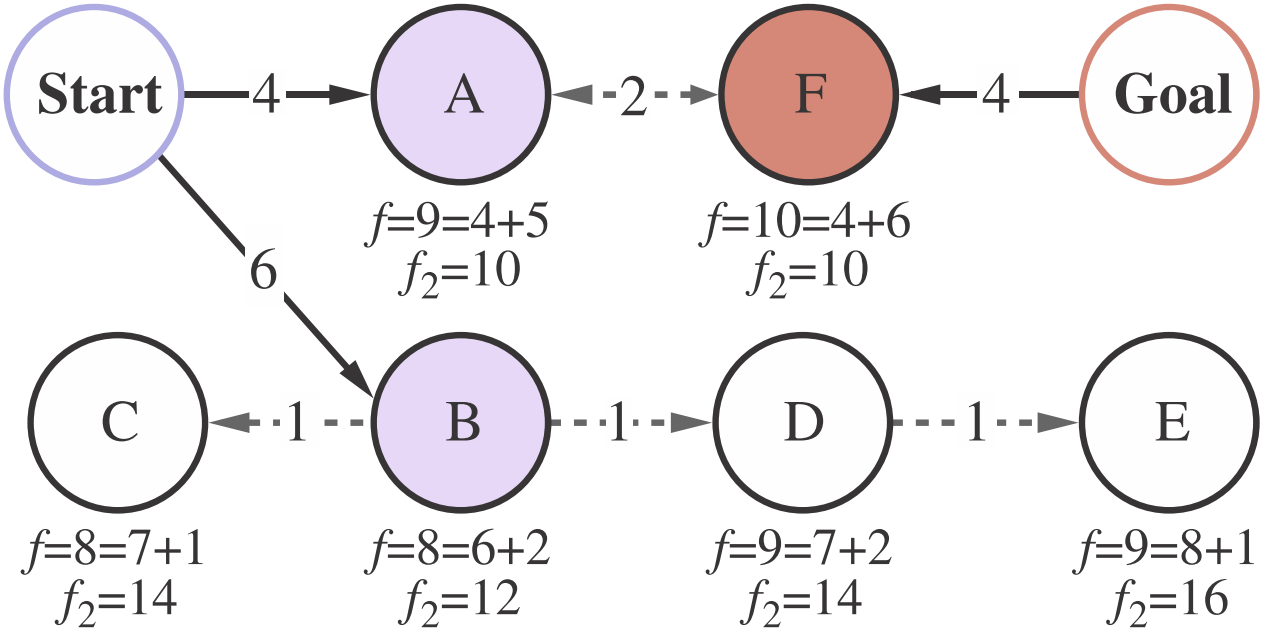


Figure 3.24 Bidirectional search maintains two frontiers: on the left, nodes A and B are successors of Start; on the right, node F is an inverse successor of Goal. Each node is labeled with $f = g + h$ values and the $f_2 = \max(2g, g + h)$ value. (The g values are the sum of the action costs as shown 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 دوطرفه

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.

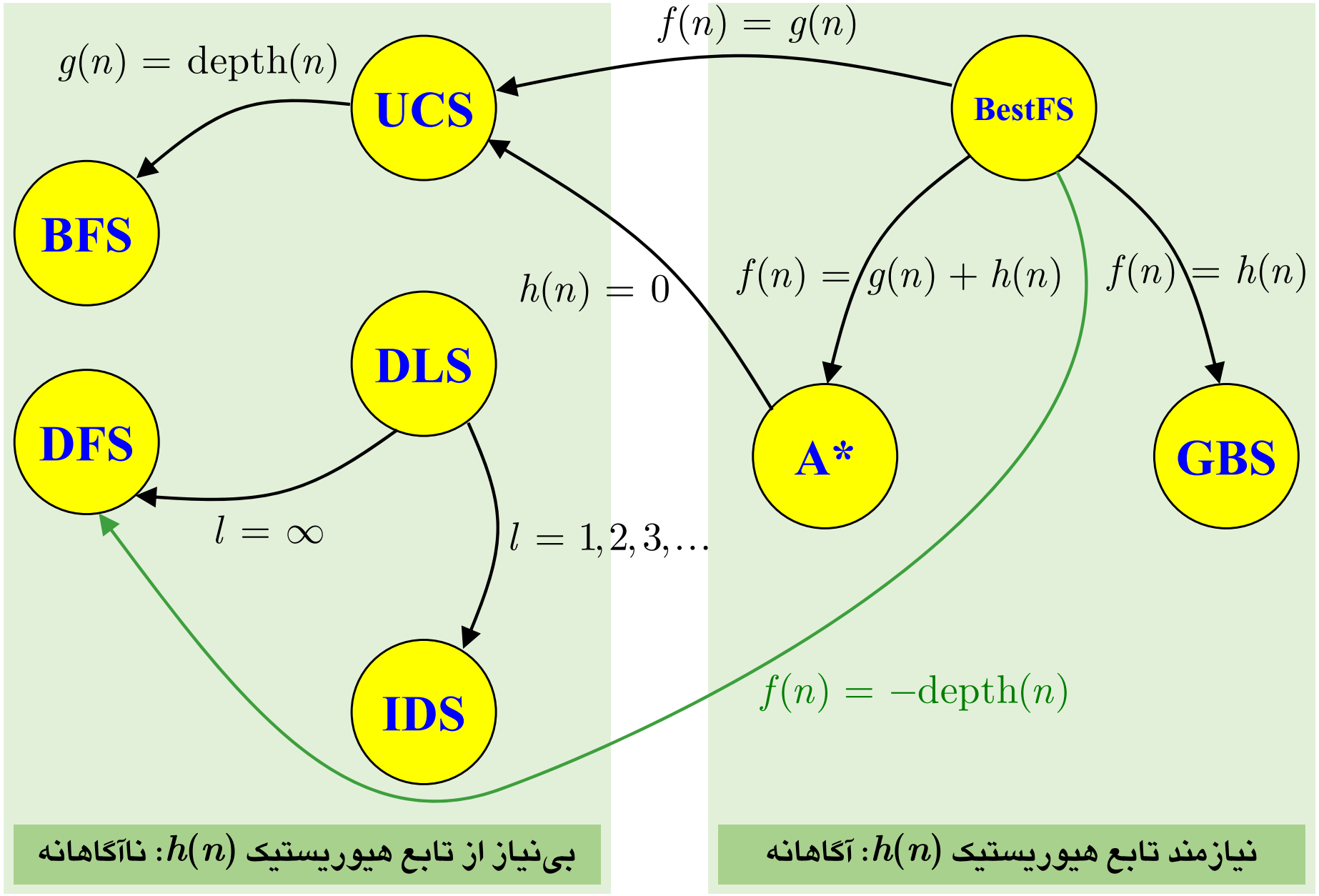
With an average heuristic, bidirectional search that meets in the middle tends to expand fewer nodes and is preferred.

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



هوش مصنوعی

حل مسئله با جستجو

۶

توابع هیوریستیک

توابع هیوریستیک

HEURISTIC FUNCTIONS

$$h(n)$$

پرسش‌های کلیدی در مورد **تابع هیوریستیک** برای یک مسئله‌ی 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**

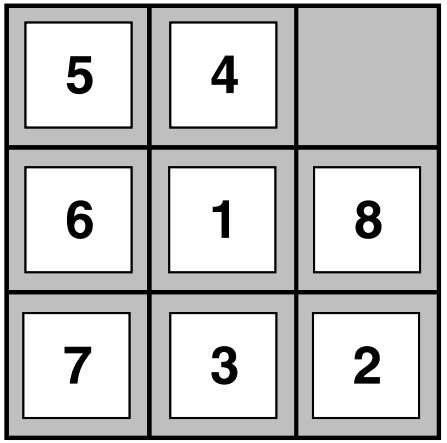
هیوریستیک قابل‌قبول

مثال: هیوریستیک برای مسئله‌ی معمای ۸

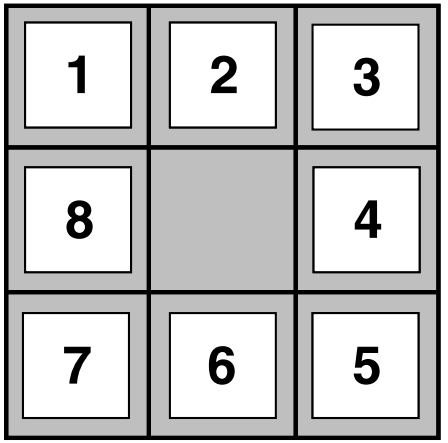
برای مسئله‌ی معمای ۸

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

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



Start State



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

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

مثال

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

مقدار ایده‌آل b^* برابر با 1 است.

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

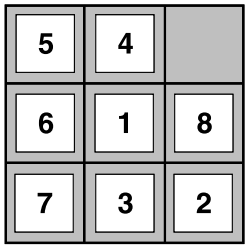
فاکتور انشعاب مؤثر

مثال : مسئله‌ی معمای ۸

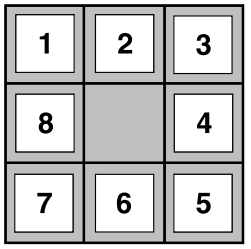
	Search Cost (nodes generated)			Effective Branching Factor		
d	IDS	A*(h_1)	A*(h_2)	IDS	A*(h_1)	A*(h_2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	–	539	113	–	1.44	1.23
16	–	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

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

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



Start State



Goal State

غلبه

غلبه‌ی یک هیوریستیک بر یک هیوریستیک دیگر

DOMINANCE

اگر برای دو هیوریستیک قابل‌قبول 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 بیش‌برآورد (overestimate) کند.
- $h(n)$ باید تا جای ممکن بزرگ باشد.
- $h(n)$ باید از نظر محاسباتی ارزان باشد.

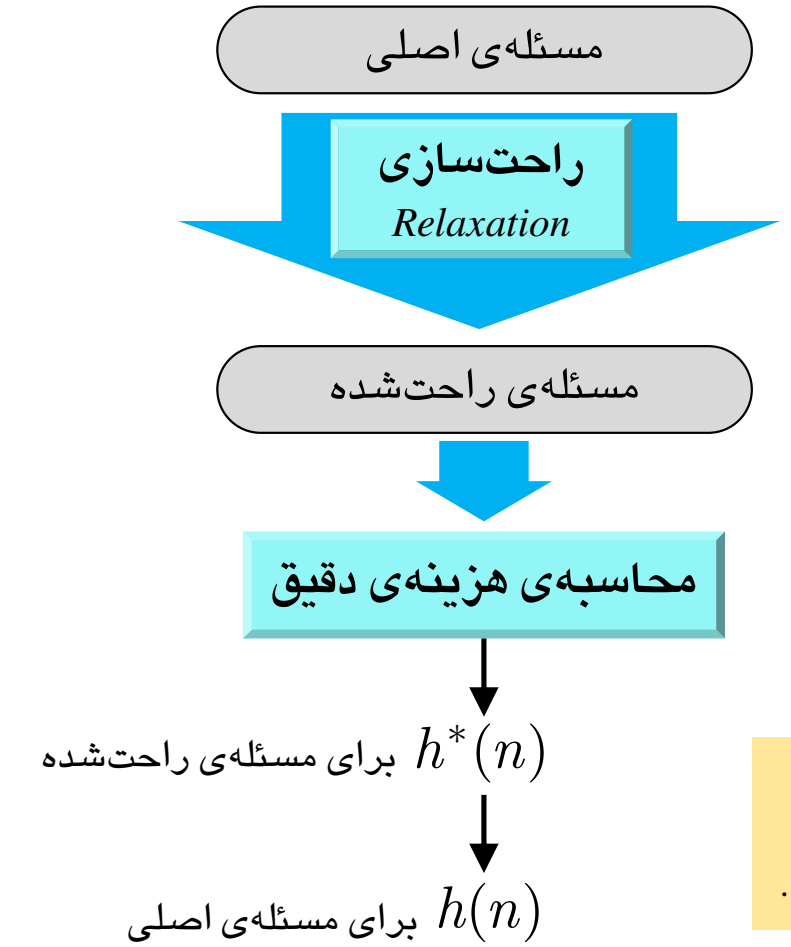
$$0 \leq h(n) \leq h^*(n)$$

تولید هیوریستیک قابل قبول

مسئله‌های راحت‌شده

RELAXED PROBLEMS

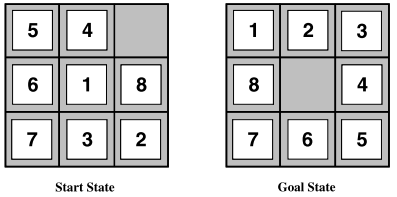
استخراج **هیوریستیک قابل قبول** با
استفاده از هزینه‌ی دقیق راه‌حل یک نسخه‌ی راحت‌تر شده (relaxed) از مسئله‌ی اصلی



نکته‌ی کلیدی:
هزینه‌ی راه‌حل بهینه‌ی یک مسئله‌ی راحت‌شده
نباید از هزینه‌ی راه‌حل بهینه‌ی مسئله‌ی اصلی بیشتر باشد.

تولید هیوریستیک قابل قبول

مسئله‌های راحت‌شده: مثال (مسئله‌ی معمای ۸): (۱)



مسئله‌ی اصلی

یک کاشی می‌تواند **به خانه‌ی خالی مجاور** جابجا شود.

راحت‌سازی
Relaxation

مسئله‌ی راحت‌شده

یک کاشی می‌تواند **به هر جایی** جابجا شود.

محاسبه‌ی هزینه‌ی دقیق

$h^*(n)$ برای مسئله‌ی راحت‌شده

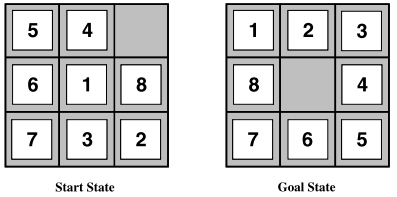
هزینه‌ی دقیق هر حالت تا هدف:
= تعداد کاشی‌های قرارگرفته در جای نادرست
(تک تک کاشی‌های نادرست باید به جای درست خود منتقل شوند)

$h(n)$ برای مسئله‌ی اصلی

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

تولید هیوریستیک قابل قبول

مسئله‌های راحت‌شده: مثال (مسئله‌ی معمای ۸): (۲)



مسئله‌ی اصلی

یک کاشی می‌تواند **به خانه‌ی خالی مجاور** جابجا شود.

راحت‌سازی
Relaxation

مسئله‌ی راحت‌شده

یک کاشی می‌تواند **به هر خانه‌ی مجاور** جابجا شود.

محاسبه‌ی هزینه‌ی دقیق

برای مسئله‌ی راحت‌شده $h^*(n)$

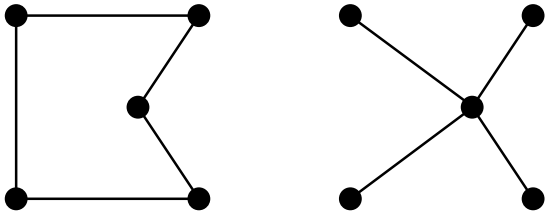
هزینه‌ی دقیق هر حالت تا هدف:
= مجموع فاصله‌ی شهری (Manhattan) هر کاشی تا جای درست خود
(هر کاشی نادرست به اندازه‌ی فاصله‌ی خود تا جای درست نیاز به جابجایی دارد.)

برای مسئله‌ی اصلی $h(n)$

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

تولید هیوریستیک قابل قبول

مسئله‌های راحت‌شده: مثال (مسئله‌ی فروشنده‌ی دوره‌گرد: TSP)



مسئله‌ی اصلی

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

راحت‌سازی
Relaxation

مسئله‌ی راحت‌شده

کوتاه‌ترین مسیر گذرنده از همه‌ی شهرها، **هرکدام به هر تعداد**

محاسبه‌ی هزینه‌ی دقیق

هزینه‌ی دقیق هر حالت تا هدف: $O(n^2)$

درخت‌پوشای کمینه (minimal spanning tree: MST)

= مجموع وزن یال‌های درخت پوشای کمینه

(MST کم‌ترین هزینه‌ی اتصال همه‌ی رأس‌های گراف (شهرها) را برمی‌گرداند.)

برای مسئله‌ی راحت‌شده $h^*(n)$

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

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:

$$x_1, x_2, \ldots, x_k$$

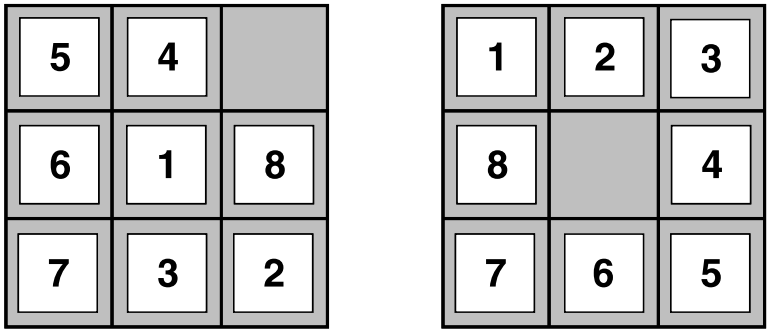
تابع هیوریستیک به صورت عمومی زیر در نظر گرفته می‌شود:

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

روش یادگیری، چگونگی تعیین ضرایب ویژگی‌ها را مشخص می‌کند.

تولید هیوریستیک قابل قبول

یادگیری هیوریستیک‌ها از تجربه: مثال (مسئله‌ی معمای۸)



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.

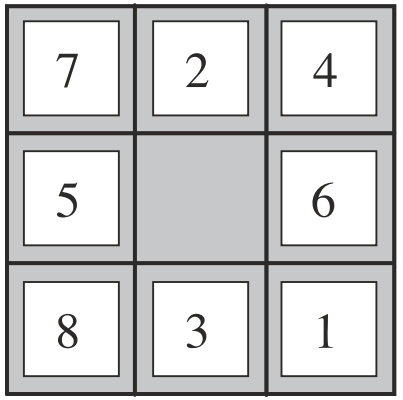
ساخت پایگاه داده‌ی الگو

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

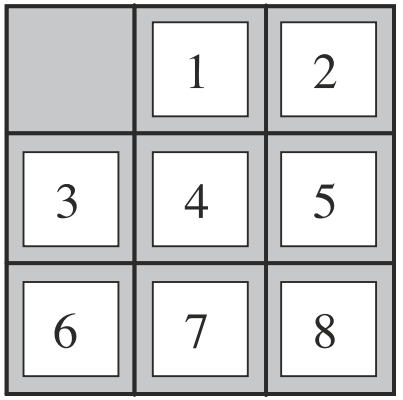
تولید هیوریستیک قابل قبول

استفاده از پایگاه داده‌ی الگو (مثال)

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

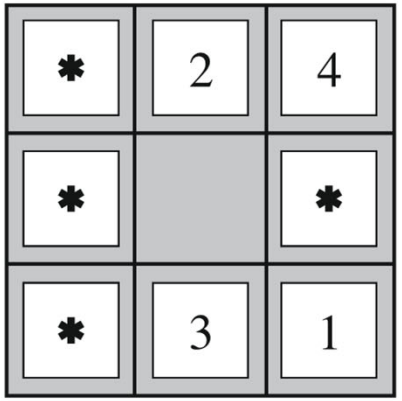


Start State

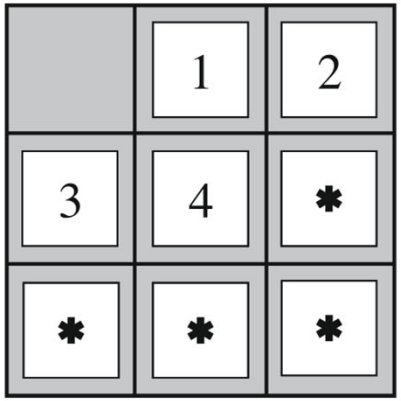


Goal State

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



Start State



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

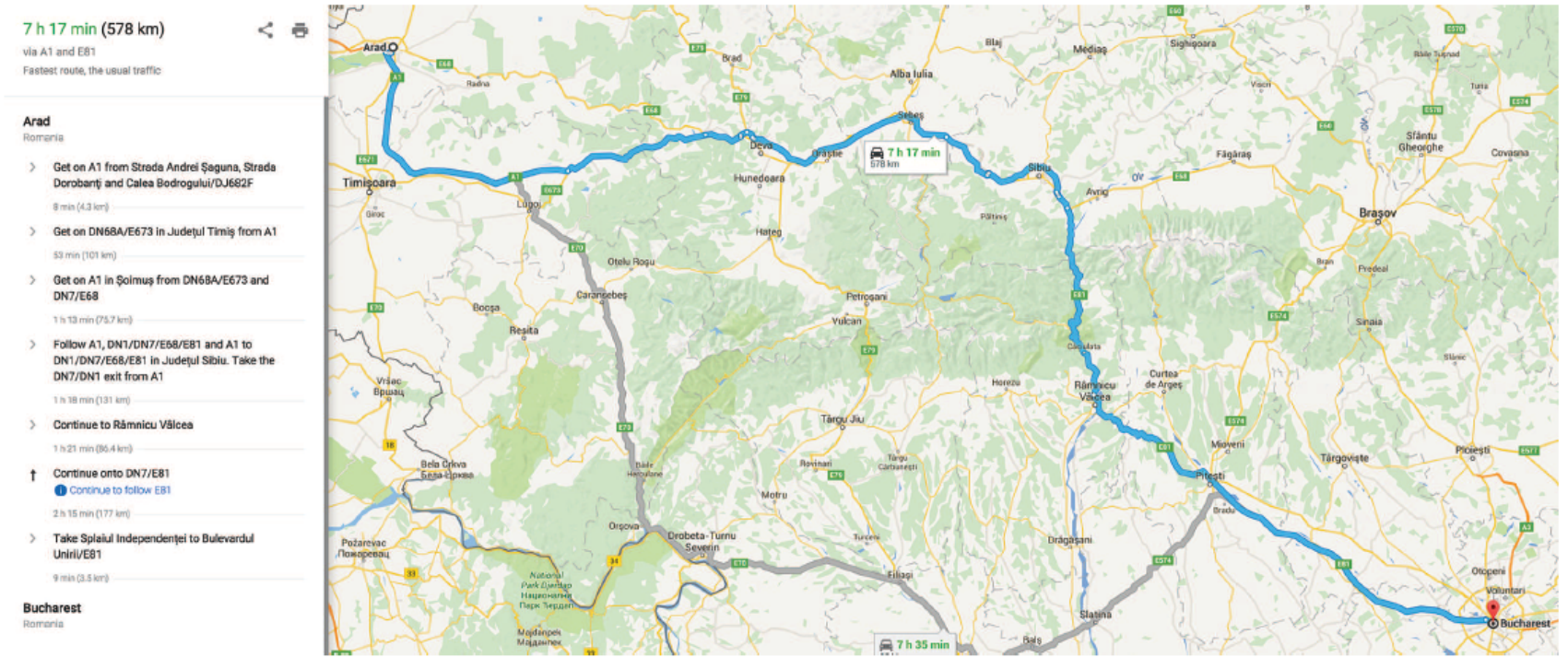


Figure 3.28 A Web service providing driving directions, computed by a search algorithm.

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

LEARNING TO SEARCH BETTER

یادگیری هیوریستیک‌ها از روی تجربه

LEARNING HEURISTICS FROM EXPERIENCE

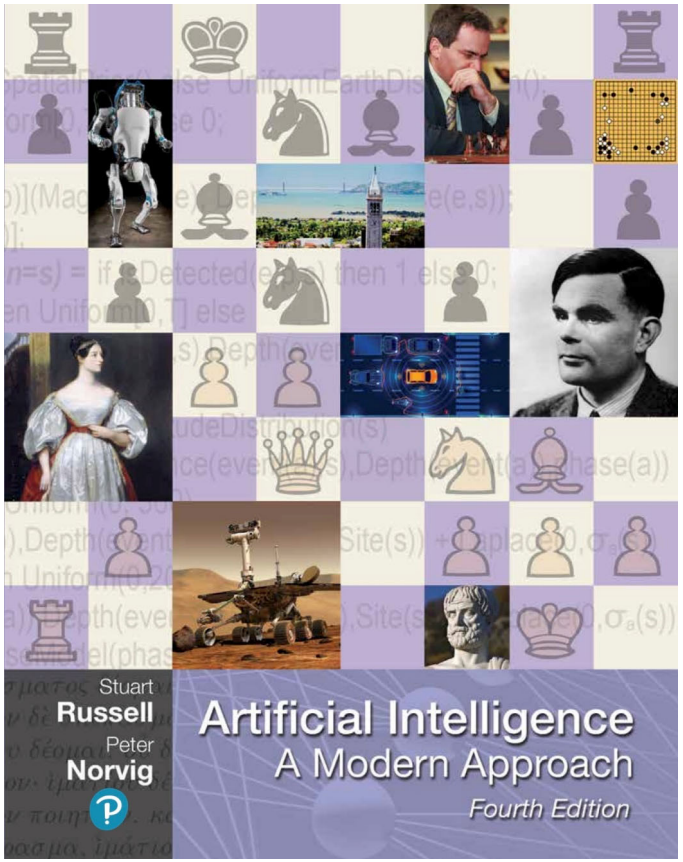
هوش مصنوعی

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