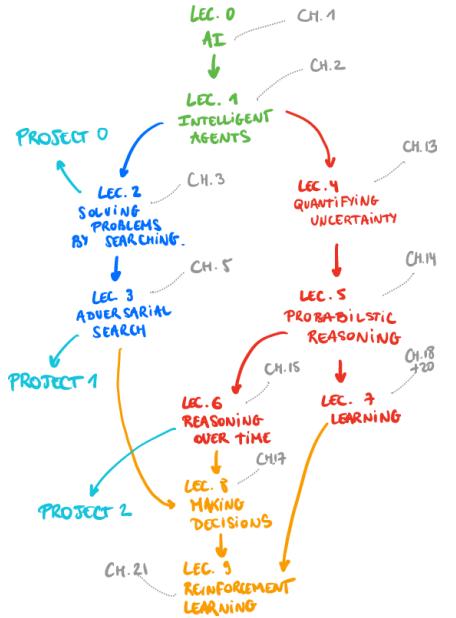


บทนำสู่ปัญญาประดิษฐ์

บทบรรยายที่ 2: การแก้ปัญหาด้วยการค้นหา

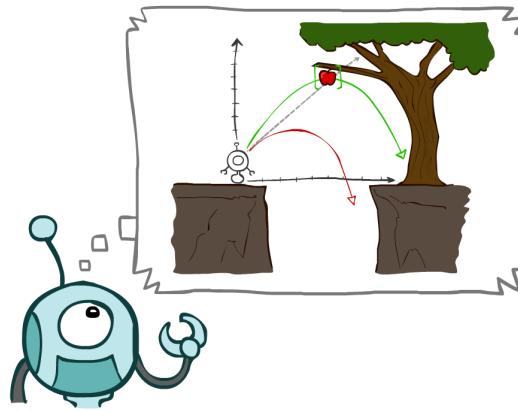
อิทธิพล พองแก้ว

[ittipon@g.sut.ac.th]



วันนี้เราจะเรียน

- Planning agents
- Search problems
- วิธีการค้นหาแบบไม่มีข้อมูล
 - Depth-first search
 - Breadth-first search
 - Uniform-cost search
- วิธีการค้นหาแบบมีข้อมูล
 - A*
 - Heuristics

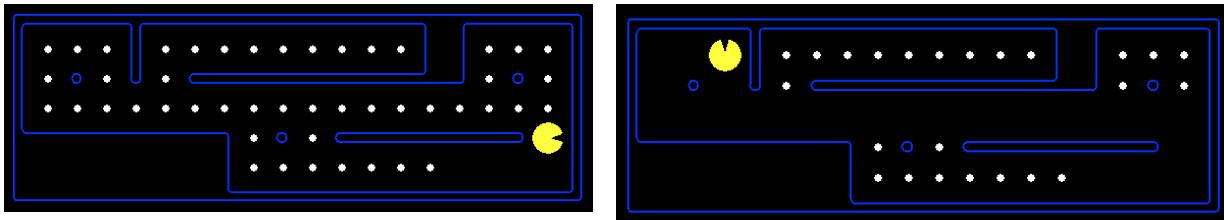


Planning agents

Reflex agents

Reflex agents

- เลือกการกระทำบนพื้นฐานของการรับรู้ในขณะนั้น
- อาจมี model ของสถานะปัจจุบันของโลก
- ไม่พิจารณาผลลัพธ์ในอนาคตของการกระทำ
- พิจารณาเพียง โลกในขณะนี้เป็นอย่างไร



ตัวอย่างเช่น reflex agent ธรรมชาติที่ใช้กฎเบื้องต้นไป-การกระทำ สามารถเคลื่อนที่ไปยังจุดถ้ามีจุดอยู่ในบริเวณใกล้เคียง ไม่มีการวางแผนในการตัดสินใจนี้

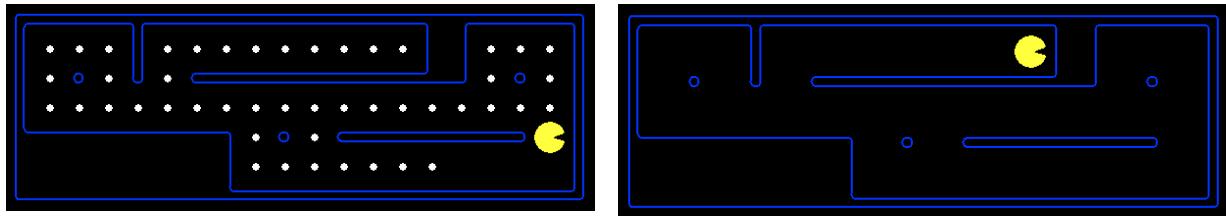
Problem-solving agents

สมมติฐาน:

- สภาพแวดล้อมแบบ single-agent, observable, deterministic และ known

Problem-solving agents

- ตัดสินใจบนพื้นฐานของผลลัพธ์ (สมมติ) ของการกระทำ โดยพิจารณา โลกจะเป็นอย่างไร
- ต้องมี model ของวิธีที่โลกพัฒนาไปตามการกระทำ
- กำหนดเป้าหมายอย่างชัดเจน



Planning agent มองหาลำดับการกระทำเพื่อกินจุดทึ่งหมาย

```

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  $\leftarrow$  UPDATE-STATE(state, percept)
  if seq is empty then
    goal  $\leftarrow$  FORMULATE-GOAL(state)
    problem  $\leftarrow$  FORMULATE-PROBLEM(state, goal)
    seq  $\leftarrow$  SEARCH(problem)
    if seq = failure then return a null action
  action  $\leftarrow$  FIRST(seq)
  seq  $\leftarrow$  REST(seq)
  return action

```

Figure 3.1 A simple problem-solving agent. It first formulates a goal and a problem, searches for a sequence of actions that would solve the problem, and then executes the actions one at a time. When this is complete, it formulates another goal and starts over.

Offline vs. Online solving

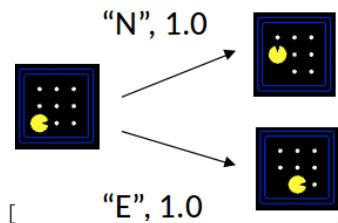
- Problem-solving agents เป็นแบบ **offline** การแก้ปัญหาจะดำเนินการ "หลับตา" โดยไม่สนใจการรับรู้
- การแก้ปัญหาแบบ **Online** เกี่ยวข้องกับการกระทำโดยไม่มีความรู้ที่สมบูรณ์ ในการนี้น้ำลำดับการกระทำการอาจถูกคำนวณใหม่ในแต่ละขั้นตอน

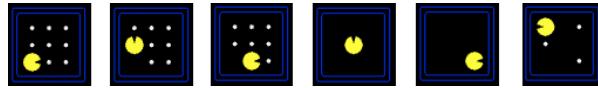
Search problems

Search problems

Search problem ประกอบด้วยส่วนประกอบต่อไปนี้:

- การแทนค่าของ **states** ของ agent และสภาพแวดล้อม
- Initial state** ของ agent
- คำอธิบายของ **actions** ที่ใช้ได้กับ agent ในสถานะ s แสดงด้วย **actions (s)**
- Transition model** ที่คืนค่าสถานะ $s' = \text{result}(s, a)$ ที่เกิดจากการทำ action a ในสถานะ s
- Goal test a** ในสถานะ s
 - เราบอกว่า s' เป็น **successor** ของ s ถ้ามี action ที่ยอมรับได้จาก s ไป s'





- รวมกัน initial state, actions และ transition model จะกำหนด **state space** ของปัญหา คือ เซตของสถานะทั้งหมดที่สามารถเข้าถึงได้จาก initial state ด้วยลำดับ action ใดๆ
 - State space สร้าง directed graph:
 - nodes = states
 - links = actions
 - Path คือลำดับของ states ที่เชื่อมต่อด้วย actions
- **Goal test** ที่กำหนดว่าการแก้ปัญหารรุ่ป้าหมายในสถานะ s หรือไม่
- **Path cost** ที่กำหนดค่าตัวเลขให้กับแต่ละ path
 - ในคอร์สนี้ เราจะ假定ว่า path cost สอดคล้องกับผลรวมของ **step costs** เชิงบวก $c(s,a,s')$ ที่เกี่ยวข้องกับ action a ใน s ที่นำไปสู่ s'

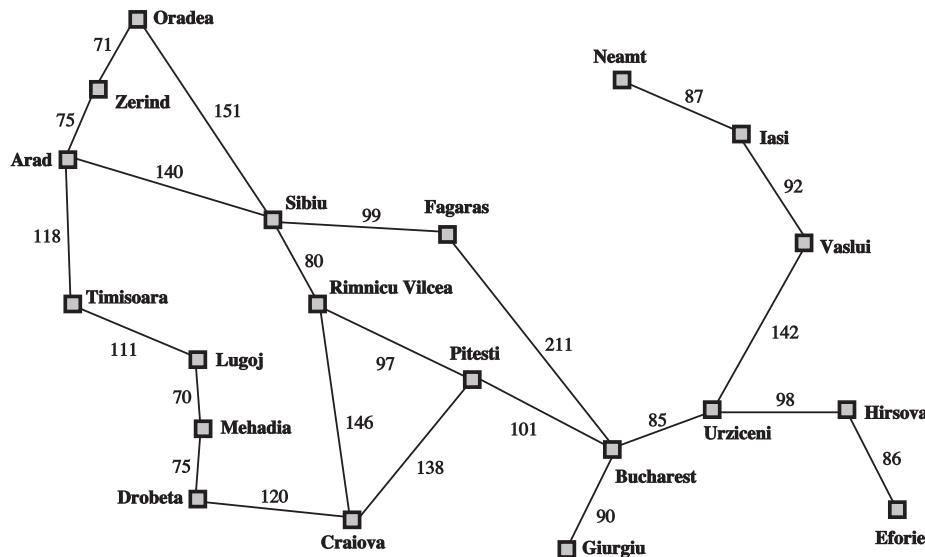
Solution ของปัญหาคือลำดับ action ที่นำจาก initial state ไปสู่ goal state

- คุณภาพของ solution วัดด้วย path cost function
- Optimal solution มี path cost ต่ำสุดในบรรดา solutions ทั้งหมด

แบบฝึกหัด

จะเป็นอย่างไรถ้าสภาพแวดล้อมเป็นแบบ partially observable? non-deterministic?

ตัวอย่าง: การเดินทางในโรมาเนีย



จะไปจาก Arad ถึง Bucharest ได้อย่างไร?

- การแทนค่า states: เมืองที่เรารอยู่
 - $s \in \{in(Arad), in(Bucharest), \dots\}$
- Initial state = เมืองที่เราเริ่มต้น
 - $s_0 = in(Arad)$
- Actions = การไปจากเมืองปัจจุบันไปยังเมืองที่เขื่อมต่อโดยตรง
 - $actions(s_0) = \{go(Sibiu), go(Timisoara), go(Zerind)\}$
- Transition model = เมืองที่เราไปถึงหลังจากขับรถไป
 - $result(in(Arad), go(Zerind)) = in(Zerind)$
- Goal test: ว่าเรารอยู่ใน Bucharest หรือไม่
 - $s \in \{in(Bucharest)\}$
- Step cost: ระยะทางระหว่างเมือง

การเลือก state space

โลกจริงมีความซับซ้อนอย่างไรขีดจำกัด

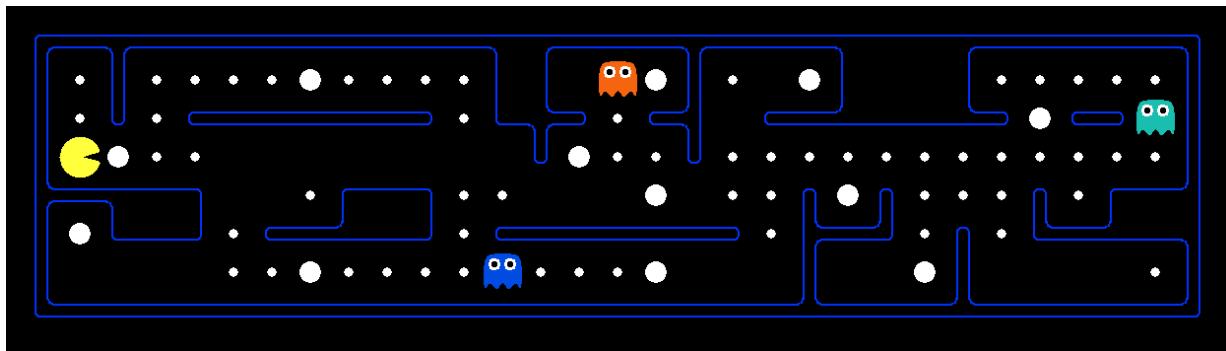
- **World state** รวมรายละเอียดทุกอย่างของสภาพแวดล้อม
- **Search state** เก็บเฉพาะรายละเอียดที่จำเป็นสำหรับการวางแผน



Search problems เป็น **models**

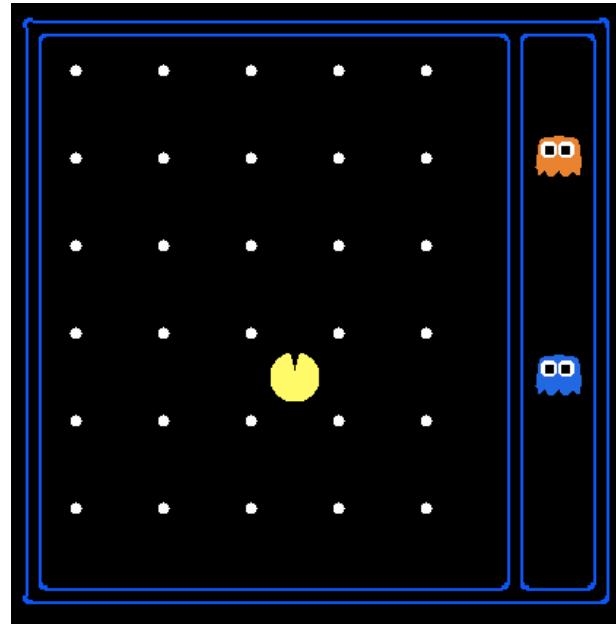
ตัวอย่าง: eat-all-dots

- States: (x, y) , dot booleans
- Actions: NSEW
- Transition: อัปเดตตำแหน่งและอาจจะอัปเดต dot boolean
- Goal test: dots ทั้งหมดเป็น false



ขนาดของ State space

- World state:
 - ตำแหน่ง Agent: 120
 - จำนวนที่พบร: 30
 - ตำแหน่ง Ghost: 12
 - ทิศทาง Agent: NSEW
- มีกี่สถานะ?
 - World states?
 - $120 \times 2^{30} \times 12^2 \times 4$
 - States สำหรับ eat-all-dots?
 - 120×2^{30}

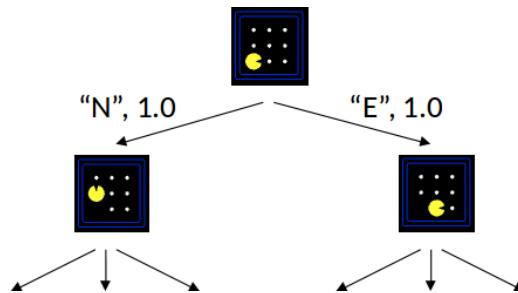


Search trees

เขตของลำดับที่ยอมรับได้ที่เริ่มต้นจาก initial state สร้าง search tree

- Nodes สอดคล้องกับ states ใน state space โดย initial state เป็น root node
- Branches สอดคล้องกับ actions ที่ใช้ได้ โดย child nodes สอดคล้องกับ successors

สำหรับปัญหาส่วนใหญ่ เราไม่สามารถสร้าง tree ทั้งหมดได้จริง แต่เราต้องการหา branch ที่เหมาะสมที่สุด!



Tree search algorithms

```
function TREE-SEARCH(problem, fringe) returns a solution, or failure
    fringe  $\leftarrow$  INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node  $\leftarrow$  REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE(node)) then return node
        fringe  $\leftarrow$  INSERTALL(EXPAND(node, problem), fringe)
```

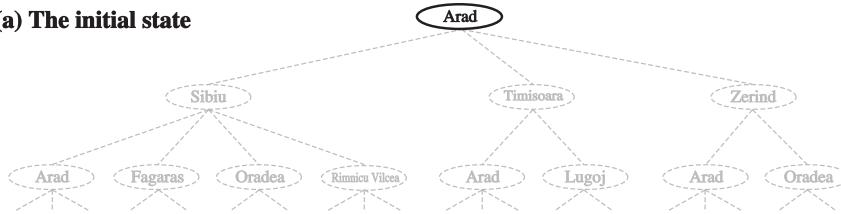
แนวคิดสำคัญ

- Fringe (หรือ frontier) ของแผนบังส่วนที่อยู่ระหว่างการพิจารณา
- Expansion
- Exploration

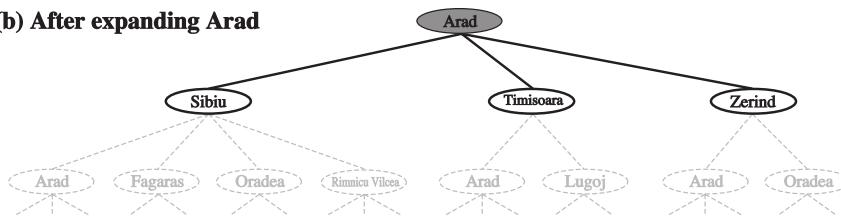
แบบฝึกหัด

จะสำรวจ fringe nodes ได้ก่อน? จะ expand nodes ให้น้อยที่สุดเพื่อบรรลุเป้าหมายได้อย่างไร?

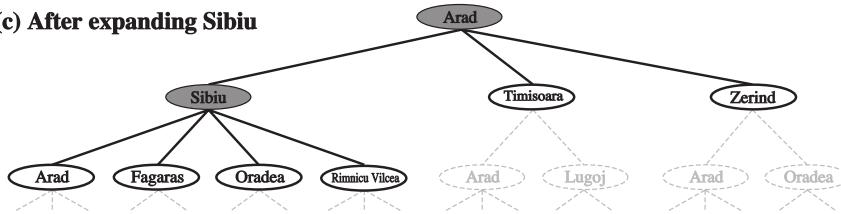
(a) The initial state



(b) After expanding Arad



(c) After expanding Sibiu



Uninformed search strategies

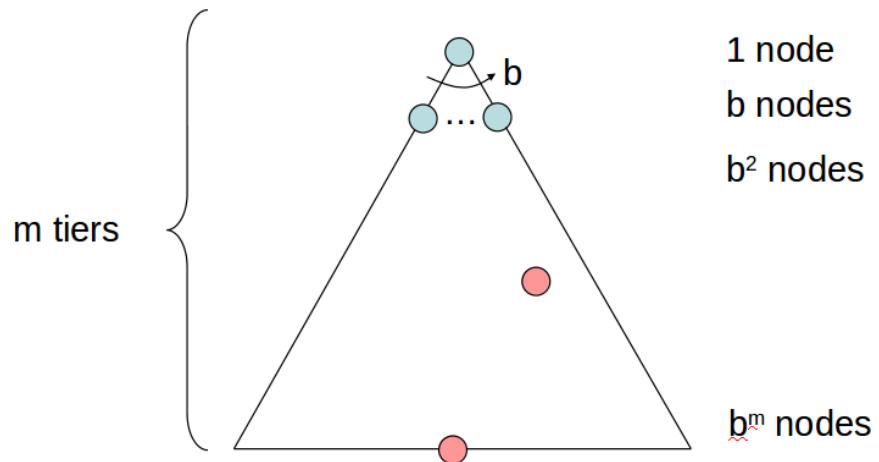
Uninformed search strategies ใช้เฉพาะข้อมูลที่มีในคำจำกัดความของปัญหา พวกลมันไม่รู้ว่าสถานะใดดูมีแนวโน้มดีกว่าสถานะอื่น

กลยุทธ์

- Depth-first search
- Breadth-first search
- Uniform-cost search
- Iterative deepening

คุณสมบัติของ search strategies

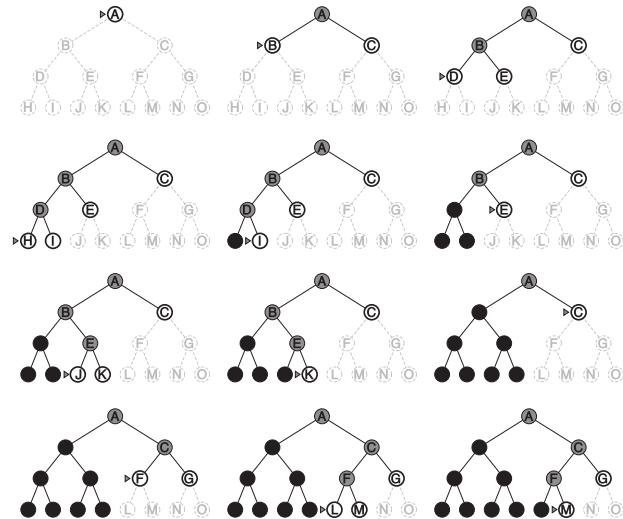
- กลยุทธ์กำหนดโดยการเลือกลำดับของการ expansion
- กลยุทธ์ประเมินตามมิติต่อไปนี้:
 - Completeness: หากลัพธ์ได้เสมอหรือไม่ถ้ามีคำตอบ?
 - Optimality: หากลัพธ์ที่มีต้นทุนน้อยที่สุดเสมอหรือไม่?
 - Time complexity: ใช้เวลาในการหาผลลัพธ์?
 - Space complexity: ใช้หน่วยความจำเท่าไหร่ในการค้นหา?
- Time และ space complexity วัดในแบบของ
 - b: branching factor สูงสุดของ search tree
 - d: ความลึกของ least-cost solution
 - ความลึกของ s คือจำนวน actions จาก initial state ถึง s
 - m: ความยาวสูงสุดของ path ใดๆ ใน state space (อาจเป็น infinity)

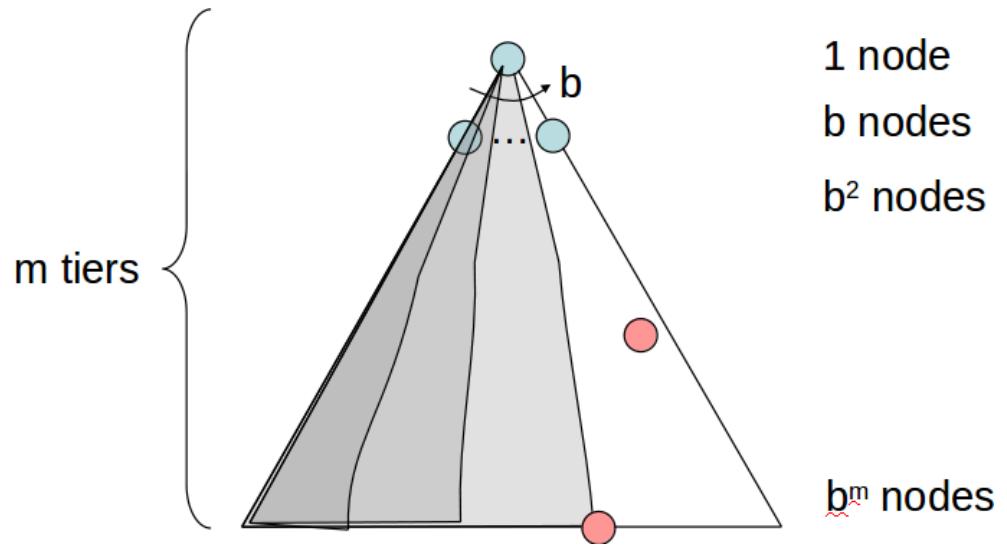


Depth-first search



- กลยุทธ์: expand node ที่ลีกที่สุดใน fringe
- การใช้งาน: fringe เป็น LIFO stack

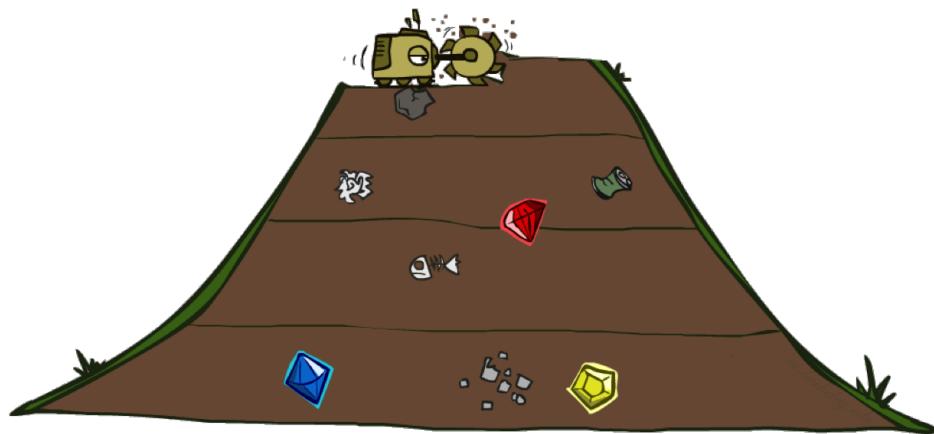




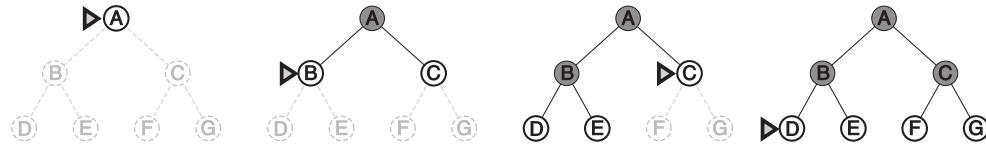
คุณสมบัติของ DFS

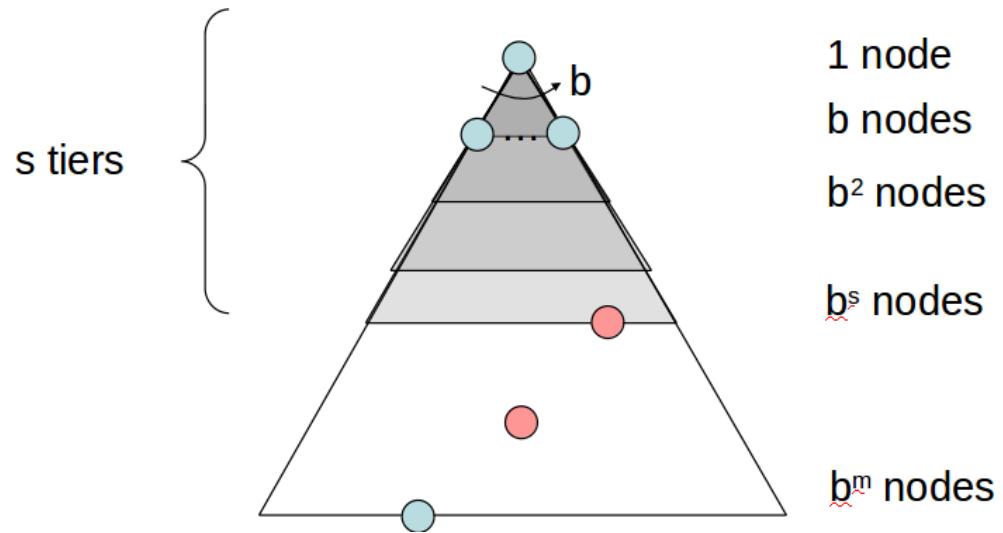
- **Completeness:**
 - m อาจเป็นอนันต์ ดังนั้นจึงสำเร็จก็ต่อเมื่อเราป้องกัน cycles (รายละเอียดเพิ่มเติมในภายหลัง)
- **Optimality:**
 - ไม่ DFS หา solution ทางช้ายสุด ไม่ว่าความลึกหรือต้นทุน
- **Time complexity:**
 - อาจสร้าง tree ทั้งหมด (หรือส่วนใหญ่ ไม่กว่า d) ดังนั้น $O(b^m)$ ซึ่งอาจมากกว่าขนาดของ state space 许多!
- **Space complexity:**
 - เก็บเฉพาะ siblings บน path ไป root ดังนั้น $O(bm)$
 - เมื่อ descendants ทั้งหมดของ node ถูกเยี่ยมชมแล้ว node สามารถลบออกจากหน่วยความจำได้

Breadth-first search



- กลยุทธ์: expand node ที่ตื้นที่สุดใน fringe
- การใช้งาน: fringe เป็น FIFO queue





คุณสมบัติของ BFS

- **Completeness:**

- ถ้า goal node ที่ต้นที่สุดอยู่ที่ความลึก d BFS จะหามันได้ในที่สุดหลังจากสร้าง nodes ที่ต้นกว่าทั้งหมด (ถ้า b เป็นจำกัด)

- **Optimality:**

- Goal ที่ต้นที่สุดไม่จำเป็นต้องเป็น optimal
- BFS เป็น optimal เนื่องจาก path cost เป็นฟังก์ชันที่ไม่ลดลงของความลึกของ node

- **Time complexity:**

- ถ้า solution อยู่ที่ความลึก d จำนวน nodes ทั้งหมดที่สร้างก่อนหา node นี้คือ $b + b^2 + b^3 + \dots + b^d = O(b^d)$

- **Space complexity:**

- จำนวน nodes ที่ต้องเก็บในหน่วยความจำคือขนาดของ fringe ซึ่งจะใหญ่ที่สุดที่ tier สุดท้าย คือ $O(b^d)$

(demo)

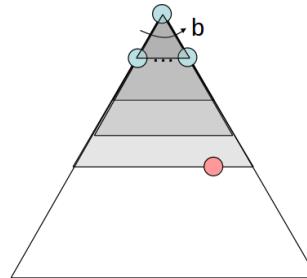
Iterative deepening

แนวคิด: ได้ข้อดีด้าน space ของ DFS กับข้อดีด้าน time/shallow solution ของ BFS

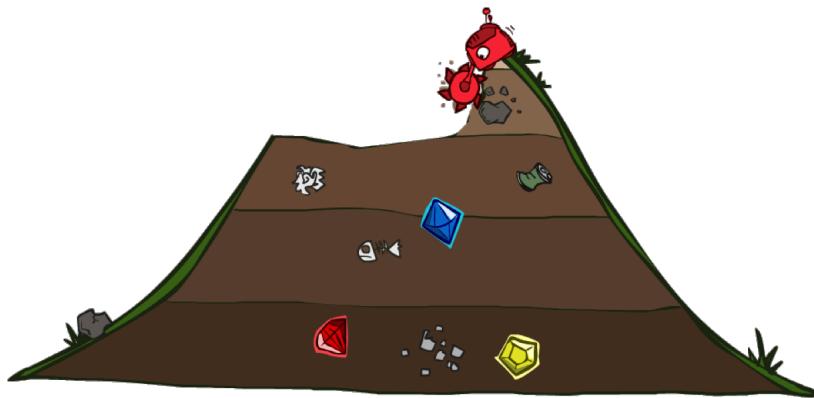
- รัน DFS ด้วย depth limit 1
- ถ้าไม่มี solution รัน DFS ด้วย depth limit 2
- ถ้าไม่มี solution รัน DFS ด้วย depth limit 3
 - ...

แบบฝึกหัด

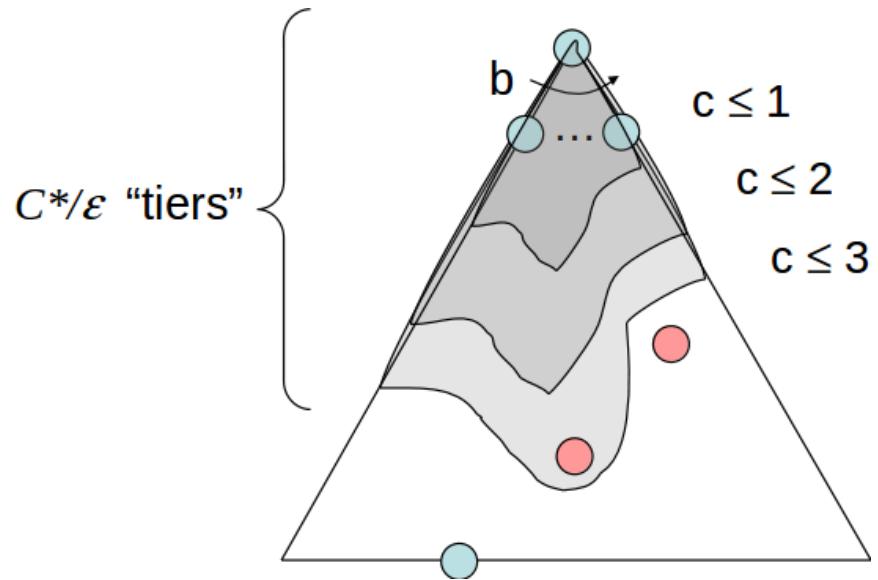
- คุณสมบัติของ iterative deepening คืออะไร?
- กระบวนการนี้เมื่อเสียเวลาโดยไม่จำเป็นหรือ?



Uniform-cost search



- กลยุทธ์: expand node ที่มีต้นทุนต่ำสุดใน fringe
- การใช้งาน: fringe เป็น priority queue ใช้ต้นทุนสะสม $g(n)$ จาก initial state ถึง node n เป็น priority

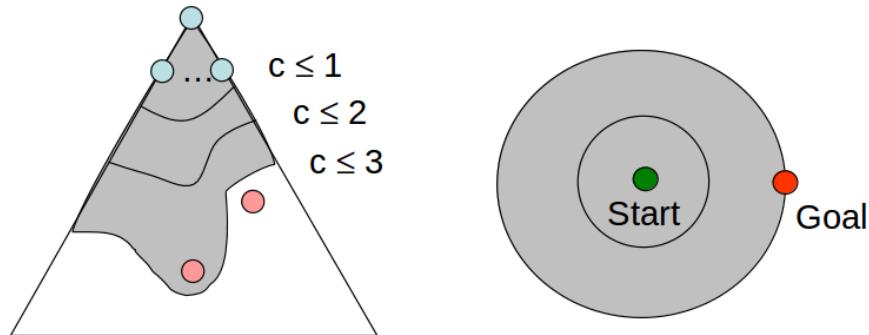


คุณสมบัติของ UCS

- Completeness:
 - ใช่ ถ้า step costs ทั้งหมดเป็น $c(s, a, s') \geq \epsilon > 0$ (ทำไง?)
- Optimality:
 - ใช่ เนื่องจาก UCS expand nodes ตามลำดับของ optimal path cost
- Time complexity:
 - สมมติ C^* เป็นต้นทุนของ optimal solution และ step costs ทั้งหมด $\geq \epsilon$
 - "Effective depth" คือประมาณ C^*/ϵ
 - Time complexity ในกรณีเลวร้ายที่สุดคือ $O(b^{C^*/\epsilon})$
- Space complexity:
 - จำนวน nodes ที่ต้องเก็บคือขนาดของ fringe เท่ากับใน tier สุดท้าย $O(b^{C^*/\epsilon})$

(demo)

Informed search strategies



หนึ่งในปัญหาของ UCS คือมันสำรวจ state space ไปทุกทิศทาง โดยไม่ใช้ประโยชน์จากข้อมูลเกี่ยวกับตำแหน่ง (ที่เป็นไปได้) ของ goal node

Informed search strategies มีจุดประสงค์เพื่อแก้ปัญหานี้โดยการ expand nodes ใน fringe ตามลำดับความต้องการที่ลดลง

- Greedy search
- A*

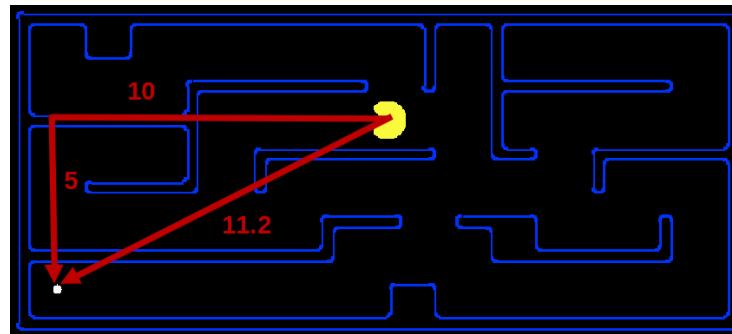
Greedy search



Heuristics

Heuristic (หรือ evaluation) function $h(n)$ คือ:

- พังก์ชันที่ **ประมาณต้นทุนของ path** ที่ถูกทิ划จาก node n ไป goal state
 - $h(n) \geq 0$ สำหรับ nodes ทั้งหมด n
 - $h(n) = 0$ สำหรับ goal state
- ถูกออกแบบสำหรับ search problem [เฉพาะ](#)



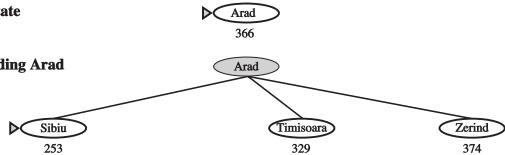
Greedy search

- กลยุทธ์: expand node n ใน fringe ที่ $h(n)$ ต่ำที่สุด
- การใช้งาน: fringe เป็น priority queue ใช้ $h(n)$ เป็น priority

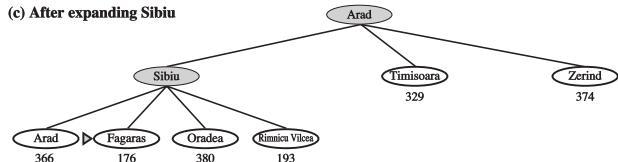
(a) The initial state



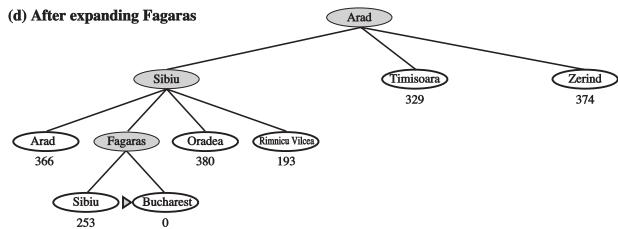
(b) After expanding Arad



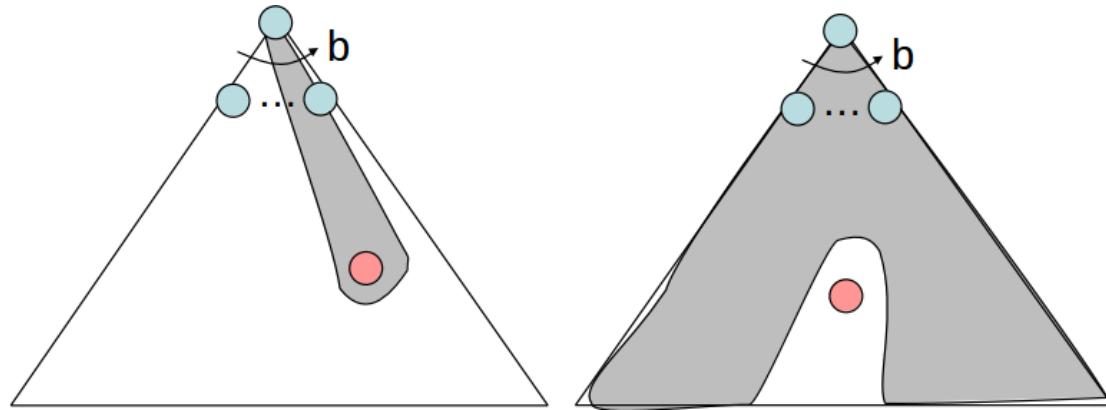
(c) After expanding Sibiu



(d) After expanding Fagaras



$h(n)$ = ระยะทางเส้นตรงถึง Bucharest



ในกรณีดีที่สุด greedy search นำคุณตรงไปสู่เป้าหมาย
ในกรณีเลวร้ายที่สุด มันเหมือน BFS ที่ถูกนำทางผิด

คุณสมบัติของ greedy search

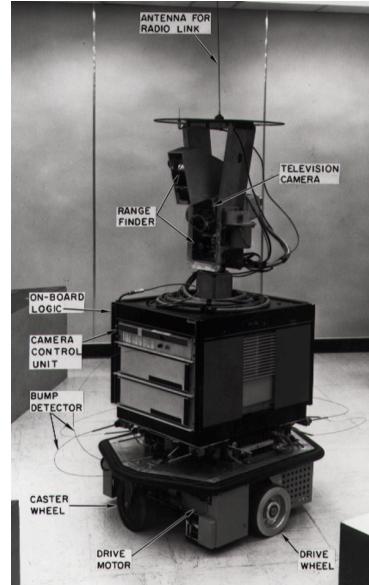
- **Completeness:**
 - ไม่เว้นแต่เราป้องกัน cycles (รายละเอียดเพิ่มเติมในภายหลัง)
- **Optimality:**
 - ไม่ เช่น path ผ่าน Sibiu และ Fagaras ยาวกว่า path ผ่าน Rimnicu Vilcea และ Pitesti 32km
- **Time complexity:**
 - $O(b^m)$ เว้นแต่เราจะมี heuristic function ที่ดี
- **Space complexity:**
 - $O(b^m)$ เว้นแต่เราจะมี heuristic function ที่ดี

A*



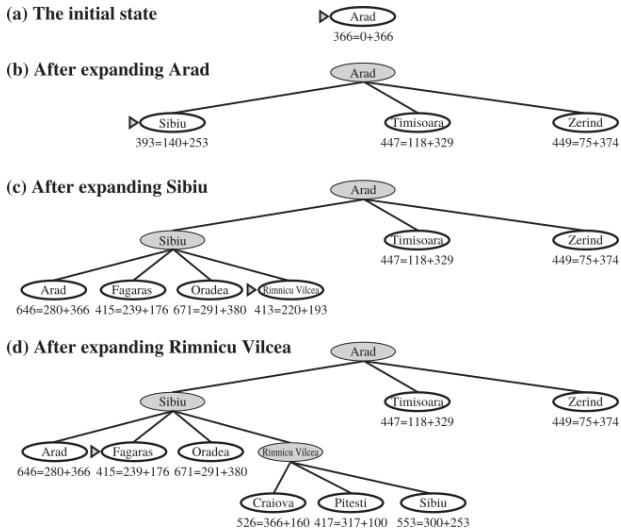
Shakey the Robot

- A* ถูกเสนอครั้งแรกใน **1968** เพื่อปรับปรุงการวางแผนของหุ่นยนต์
- เป้าหมายคือการนำทางผ่านห้องที่มีสิ่งกีดขวาง

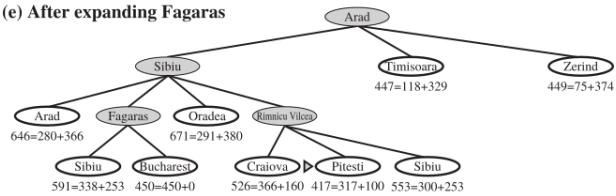


A*

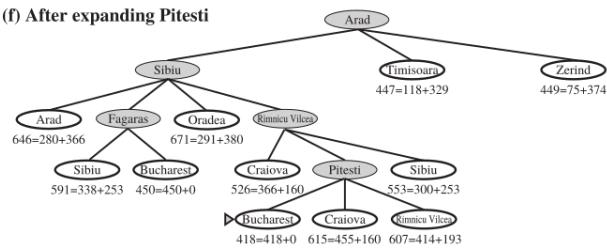
- Uniform-cost เรียงลำดับตาม path cost หรือ **backward cost** $g(n)$
- Greedy เรียงลำดับตามความใกล้เป้าหมาย หรือ **forward cost** $h(n)$
- A* รวมสองอัลกอริทึมและเรียงลำดับตามผลรวม $f(n) = g(n) + h(n)$
- $f(n)$ คือต้นทุนประมาณของ solution ที่ถูกที่สุดผ่าน n



(e) After expanding Fagaras



(f) After expanding Pitesti

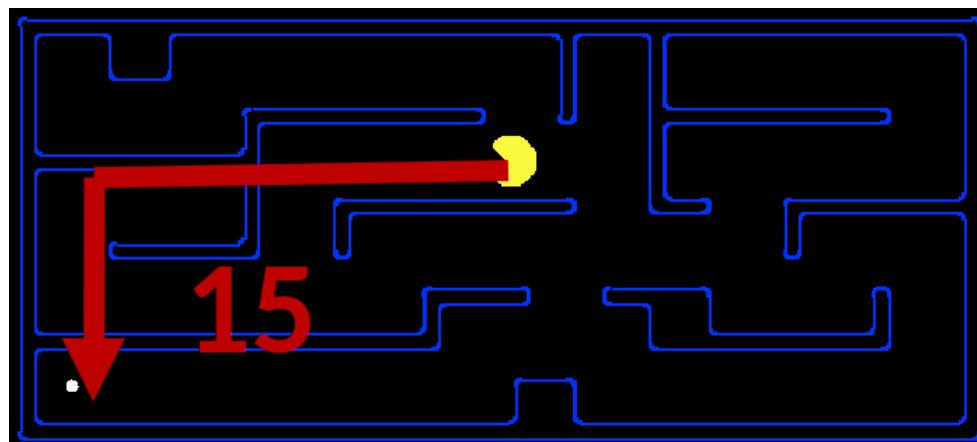


แบบฝึกหัด

ทำไม A* ไม่หยุดที่ชั้นตอน (e) ถึงแม้ Bucharest อยู่ใน fringe?

Admissible heuristics

Heuristic h เป็น **admissible** ถ้า $0 \leq h(n) \leq h^*(n)$ โดย $h^*(n)$ คือต้นทุนจริงไปยัง goal ที่ใกล้ที่สุด



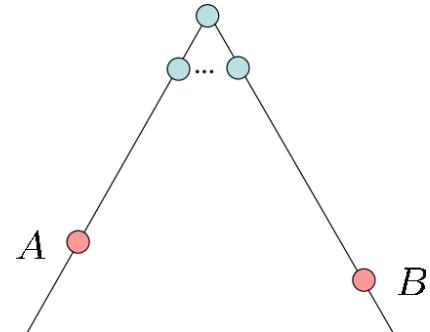
Manhattan distance เป็น admissible

Optimality ของ A*

สมมติฐาน:

- **A** เป็น optimal goal node
- **B** เป็น suboptimal goal node
- **h** เป็น admissible

ข้อกล่าวอ้าง: **A** จะออกจาก fringe ก่อน **B**

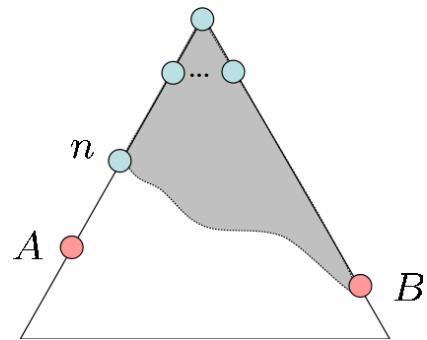


การพิสูจน์

สมมติ B อยู่บน fringe บรรพบุรุษ n ของ A อยู่บน fringe ด้วย

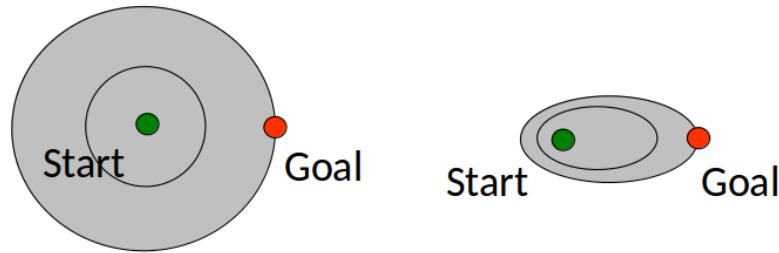
- $f(n) \leq f(A)$
 - $f(n) = g(n) + h(n)$ (ตามคำจำกัดความ)
 - $f(n) \leq g(A)$ (admissibility ของ h)
 - $f(A) = g(A) + h(A) = g(A)$ ($h = 0$ ที่ goal)
- $f(A) < f(B)$
 - $g(A) < g(B)$ (B เป็น suboptimal)
 - $f(A) < f(B)$ ($h = 0$ ที่ goal)
- ดังนั้น n expand ก่อน B
 - เมื่อจาก $f(n) \leq f(A) < f(B)$

ในทำนองเดียวกัน บรรพบุรุษทั้งหมดของ A expand ก่อน B รวมถึง A ดังนั้น A^* เป็น optimal



A* contours

- สมมติ f -costs ไม่ลดลงตาม path ใดๆ
- เราสามารถกำหนด contour levels t ใน state space ที่รวม nodes ทั้งหมด n ที่ $f(n) \leq t$



สำหรับ UCS ($h(n) = 0$) สำหรับ n ทั้งหมด bands เป็นวงกลมรอบจุดเริ่มต้น

สำหรับ A* ที่มี heuristics ที่แม่นยำ bands ยืดไปทางเป้าหมาย



Greedy search



UCS



A*

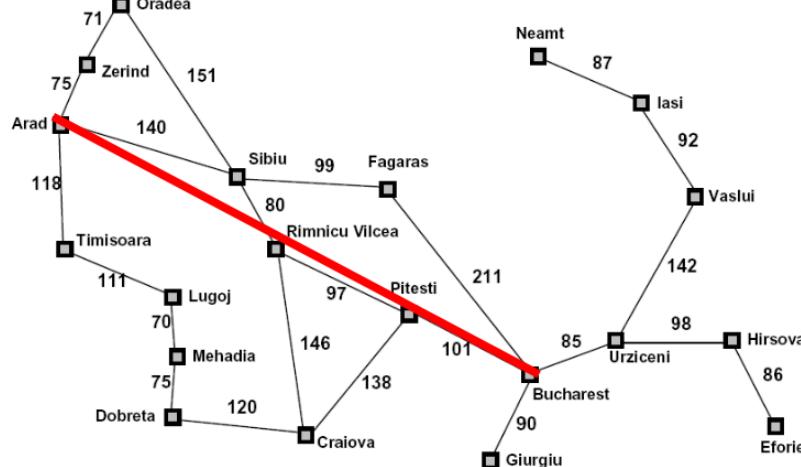
(demo)

การสร้าง admissible heuristics

งานส่วนใหญ่ในการแก้ปัญหาการค้นหาที่ยกอ่าย่างเหมาะสมที่สุดอยู่ที่การหา admissible heuristics

Admissible heuristics สามารถมาจากการคลัพหรือที่แน่นอนของ relaxed problems โดยมี actions ใหม่

366



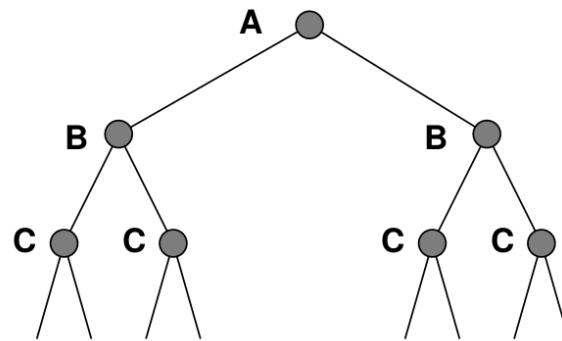
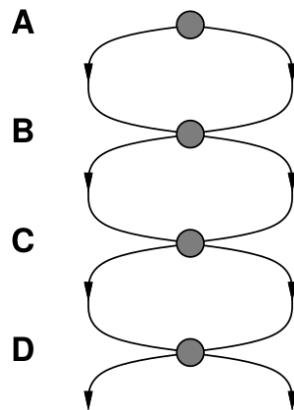
Dominance

- ถ้า h_1 และ h_2 เป็น admissible ทั้งคู่ และถ้า $h_2(n) \geq h_1(n)$ สำหรับ n ทั้งหมด แล้ว h_2 dominates h_1 และดีกว่าสำหรับการค้นหา
- ให้ admissible heuristics h_a และ h_b ใดๆ $h(n) = \max(h_a(n), h_b(n))$ ก็เป็น admissible และ dominates h_a และ h_b

การเรียนรู้ heuristics จากประสบการณ์

- สมมติสภาพแวดล้อมแบบ **episodic** agent สามารถ **เรียนรู้** heuristics ที่ได้โดยการเล่นเกมหลายครั้ง
- แต่ละ optimal solution s^* ให้ **training examples** ที่ $h(n)$ สามารถเรียนรู้ได้
- แต่ละตัวอย่างประกอบด้วยสถานะ n จาก solution path และต้นทุนจริง $g(s^*)$ ของ solution จากจุดนั้น
- การแปลง $n \rightarrow g(s^*)$ สามารถเรียนรู้ด้วยอัลกอริทึม **supervised learning**
 - Linear models, Neural networks, ฯลฯ

Graph search



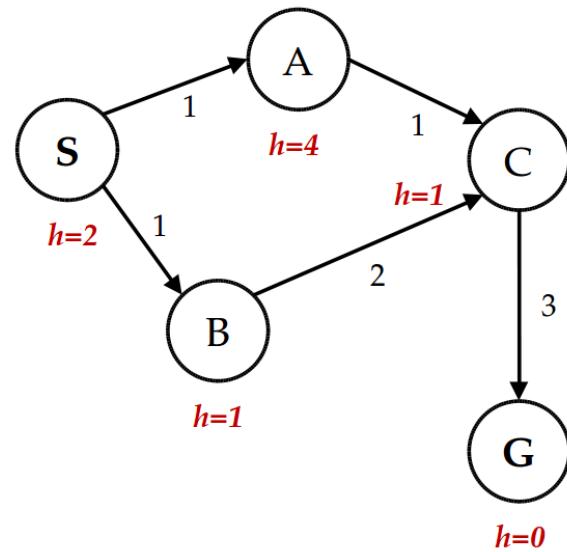
ความล้มเหลวในการตรวจจับ **repeated states** สามารถเปลี่ยนปัญหาเชิงเส้นให้เป็นแบบเลขซึ่งกำลังได้ และอาจนำไปสู่การค้นหาที่ไม่ถึงสุด

Redundant paths และ cycles สามารถหลีกเลี่ยงได้โดย **การติดตาม** states ที่ได้รับการ **สำรวจแล้ว** นี้เท่ากับการปลูก tree โดยตรงบน state-space graph

```
function GRAPH-SEARCH(problem, fringe) returns a solution, or failure
    closed  $\leftarrow$  an empty set
    fringe  $\leftarrow$  INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node  $\leftarrow$  REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe  $\leftarrow$  INSERTALL(EXPAND(node, problem), fringe)
    end
```

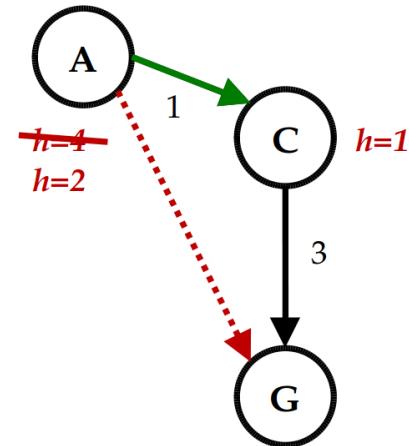
A* graph-search ผิดพลาด?

- เราเริ่มที่ **S** และ **G** เป็น goal state
- Graph search หา path ได้?



Consistent heuristics

Heuristic h เป็น consistent ถ้าสำหรับทุก n และทุก successor n' ที่สร้างโดย action a ได้ $h(n) \leq c(n, a, n') + h(n')$.



ผลลัพธ์ของ consistent heuristics:

- $f(n)$ ไม่ลดลงตาม path ใดๆ
- $h(n)$ เป็น admissible
- ด้วย consistent heuristic, graph-search A* เป็น optimal

สรุปตัวอย่าง: Super Mario



- Task environment?
 - performance measure, environment, actuators, sensors?
- ประเภท ของ environment?
- Search problem?

Infinite Mario AI - Long Level



A* ในการทำงาน

สรุป

- การกำหนดปัญหามักต้องการการทำ abstraction รายละเอียดโลกจริงเพื่อกำหนด state space ที่สามารถสำรวจได้
- วิธีการ uninformed search หลากหลาย ([DFS](#), [BFS](#), [UCS](#), [Iterative deepening](#))
- Heuristic functions ประมาณต้นทุนของ shortest paths Heuristic ที่ดีสามารถลดต้นทุนการค้นหาได้อย่างมาก
- Greedy best-first search expand h ต่อสุด ซึ่งแสดงให้เห็นว่าไม่สมบูรณ์และไม่เหมาะสมเสมอไป
- A^* search expand $f = g + h$ ต่อสุด กลยุทธ์นี้สมบูรณ์และเหมาะสม
- Graph search สามารถมีประสิทธิภาพมากกว่า tree search แบบเลขชี้กำลัง

ฉบับ