TRƯỜNG ĐẠI HỌC
BÁCH KHOA HÀ NỘI
HANOI UNIVERSITY
OF SCIENCE AND TECHNOLOGY

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG
SCHOOL OF INFORMATION AND COMMUNICATIONS TECHNOLOGY

IT3160
Nhập môn Trí tuệ nhân tạo

Artificial Intelligence

PGS.TS Lê Thanh Hương & TS Đỗ Tiến Dũng

ONE LOVE. ONE FUTURE.

Nội dung môn học

- Chương 1 - Giới thiệu về Trí tuệ nhân tạo
- Chương 2 - Tác tử
- Chương 3 - Giải quyết vấn đề
  - Các kỹ thuật tìm kiếm cơ bản
  - Tìm kiếm với tri thức bổ sung (Informed search)
  - Các kỹ thuật tìm kiếm có đối thủ
  - Tìm kiếm dựa trên thỏa mãn ràng buộc
- Chương 4 - Logic và suy diễn
- Chương 5 - Học máy

Nhắc lại: Tìm kiếm theo cấu trúc cây

- Một chiến lược (phương pháp) tìm kiếm = Một cách xác định thứ tự xét các nút của cây

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

Tìm kiếm với tri thức bổ sung

- Các chiến lược tìm kiếm cơ bản (uninformed search strategies) chỉ sử dụng các thông tin chứa trong định nghĩa của bài toán
  - Không phù hợp với nhiều bài toán thực tế (do đòi hỏi chi phí quá cao về thời gian và bộ nhớ)

- Các chiến lược tìm kiếm với tri thức bổ sung (informed search strategies) sử dụng *các tri thức cụ thể của bài toán* $\rightarrow$ Quá trình tìm kiếm hiệu quả hơn
  - Các giải thuật tìm kiếm best-first (Greedy best-first, A*)
  - Các giải thuật tìm kiếm cục bộ (Hill-climbing, Simulated annealing, Local beam, Genetic algorithms)
  - Các giải thuật tìm kiếm đối kháng (MiniMax, Alpha-beta pruning)

Best-first search

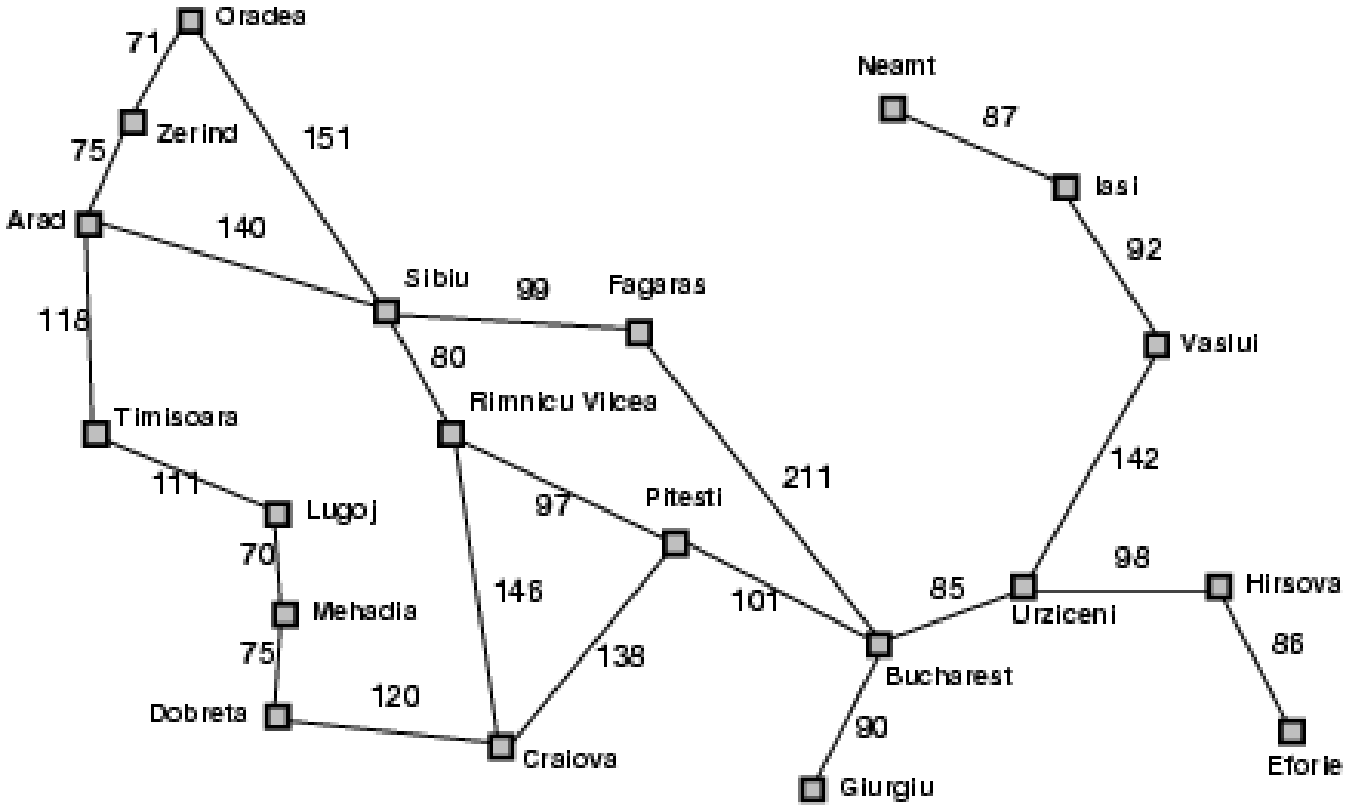
- Ý tưởng: Sử dụng một *hàm đánh giá f(n)* cho mỗi nút của cây tìm kiếm
  - Để đánh giá mức độ "phù hợp" của nút đó

  →Trong quá trình tìm kiếm, ưu tiên xét các nút có mức độ phù hợp cao nhất
- Cài đặt giải thuật
  - Sắp thứ tự các nút trong cấu trúc fringe theo trật tự giảm dần về mức độ phù hợp
- Các trường hợp đặc biệt của giải thuật Best-first search
  - Greedy best-first search
  - A* search

Greedy best-first search

- Hàm đánh giá $f(n) = h(n)$
  - Hàm heuristic $h(n)$ đánh giá chi phí để đi từ nút hiện tại n đến nút đích (mục tiêu)
- Ví dụ: Trong bài toán tìm đường đi từ Arad đến Bucharest, sử dụng: $h_{SLD}(n)$ = Ước lượng khoảng cách đường thẳng ("chim bay") từ thành phố hiện tại n đến Bucharest
- Phương pháp tìm kiếm Greedy best-first search sẽ xét (phát triển) nút "có vẻ" gần với nút đích (mục tiêu) nhất

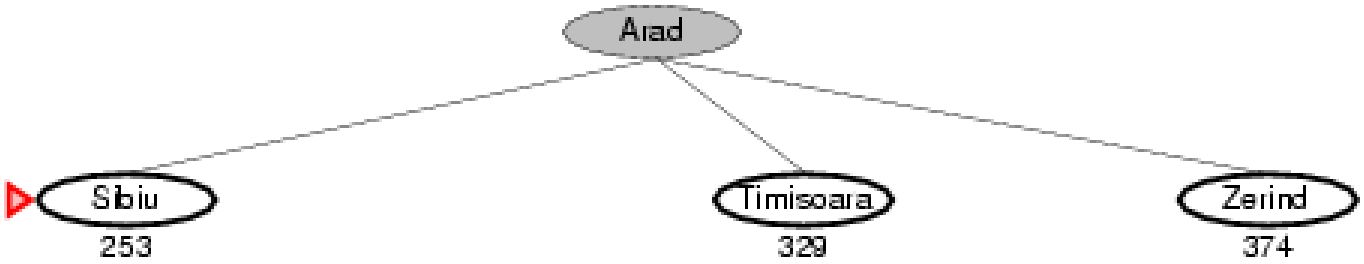
Greedy best-first search – Ví dụ (1)



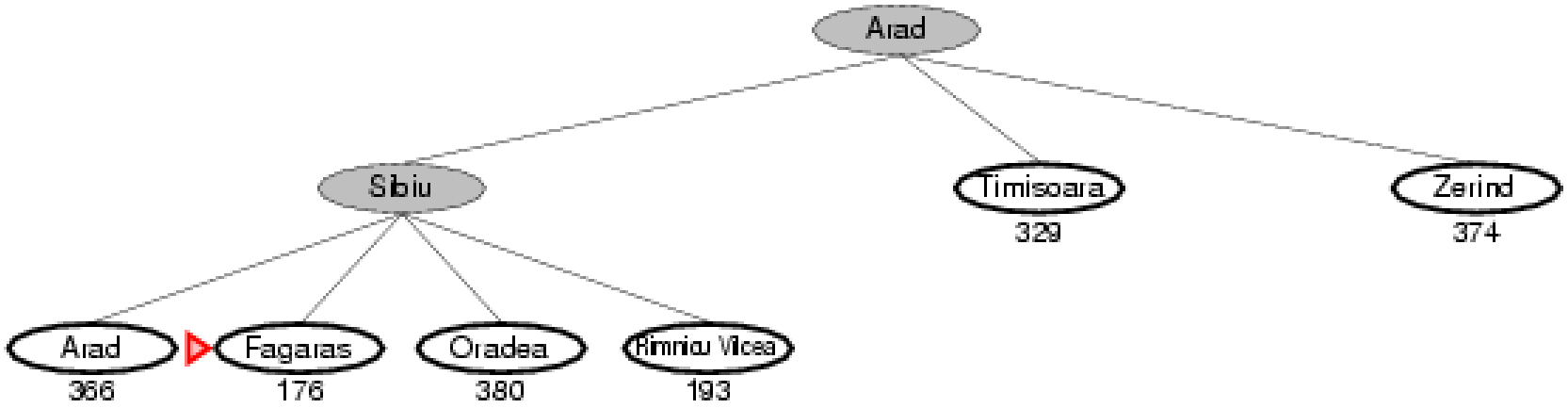
Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy best-first search – Ví dụ (2)

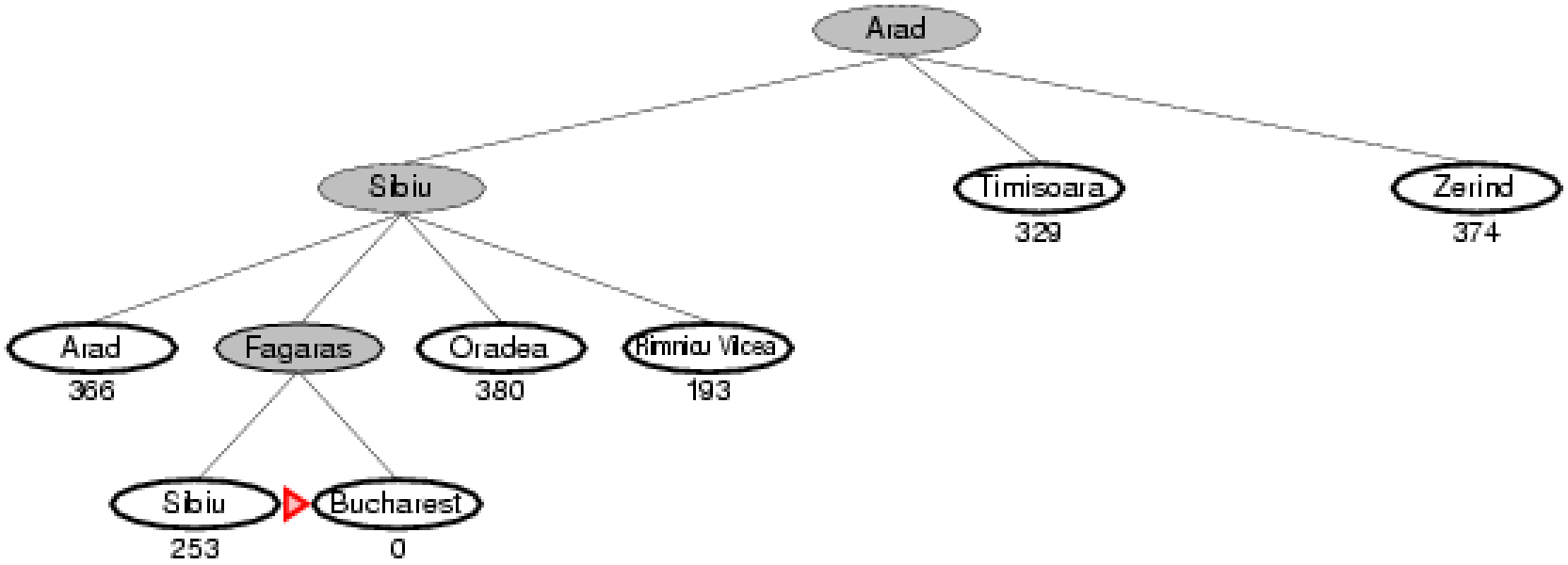
Greedy best-first search – Ví dụ (3)



Greedy best-first search – Ví dụ (4)

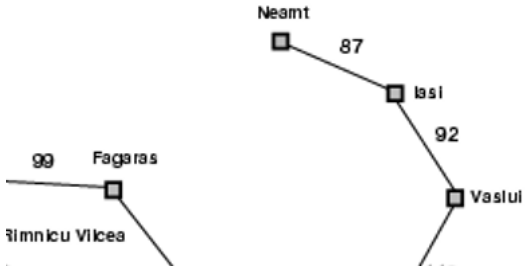


Greedy best-first search – Ví dụ (5)



Greedy best-first search – Các đặc điểm

- Tính hoàn chỉnh?
  - Không – Vì có thể vướng (chết tắc) trong các vòng lặp kiểu như: Neamt → Iasi → Neamt →…



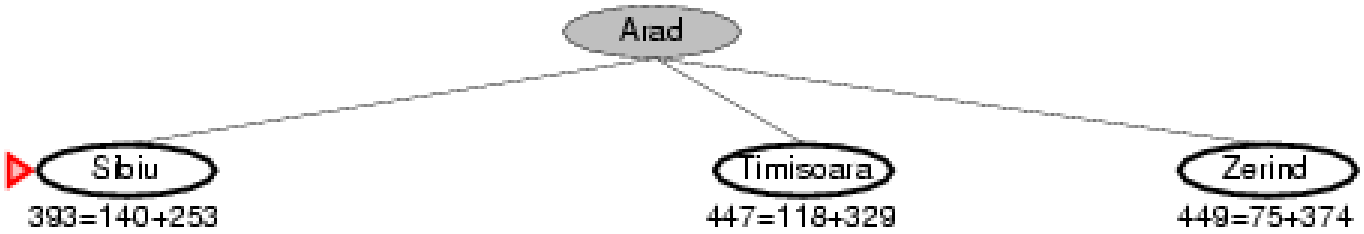
- Độ phức tạp về thời gian?
  - $O(b^m)$
  - Một hàm heuristic tốt có thể mang lại cải thiện lớn
- Độ phức tạp về bộ nhớ?
  - $O(b^m)$ – Lưu giữ tất cả các nút trong bộ nhớ
- Tính tối ưu?
  - Không

A* search

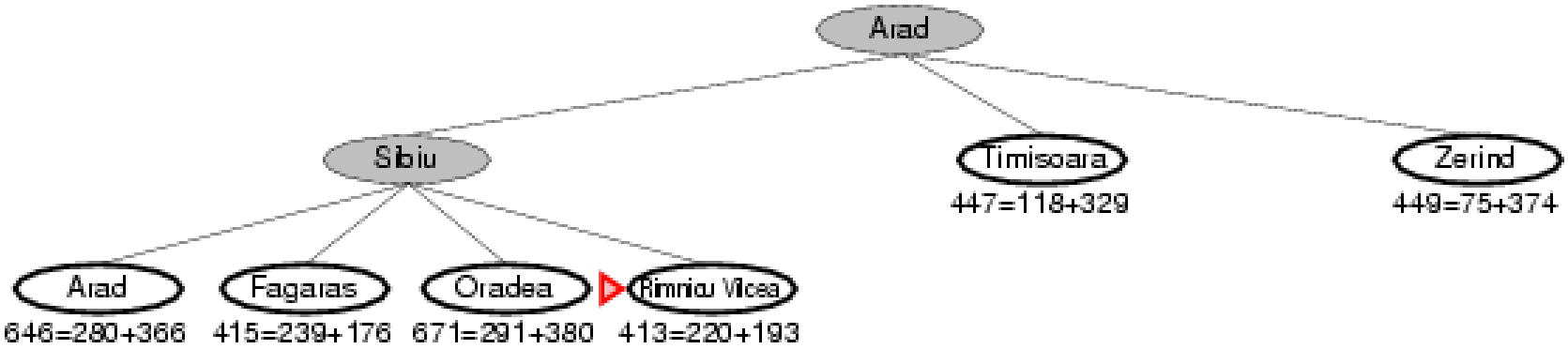
- Ý tưởng: Tránh việc xét (phát triển) các nhánh tìm kiếm đã xác định (cho đến thời điểm hiện tại) là có chi phí cao
- Sử dụng hàm đánh giá $f(n) = g(n) + h(n)$
  - $g(n)$ = chi phí từ nút gốc cho đến nút hiện tại n
  - $h(n)$ = chi phí ước lượng từ nút hiện tại n tới đích
  - $f(n)$ = chi phí tổng thể ước lượng của đường đi qua nút hiện tại n đến đích

A* search – Ví dụ (1)

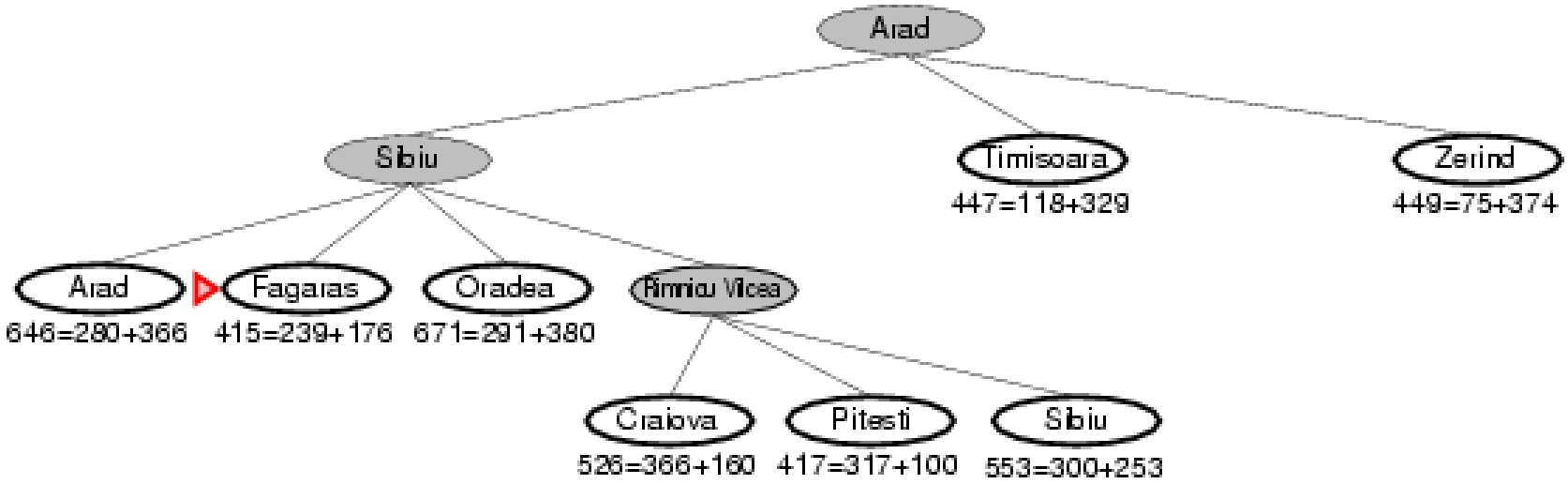
A* search – Ví dụ (2)



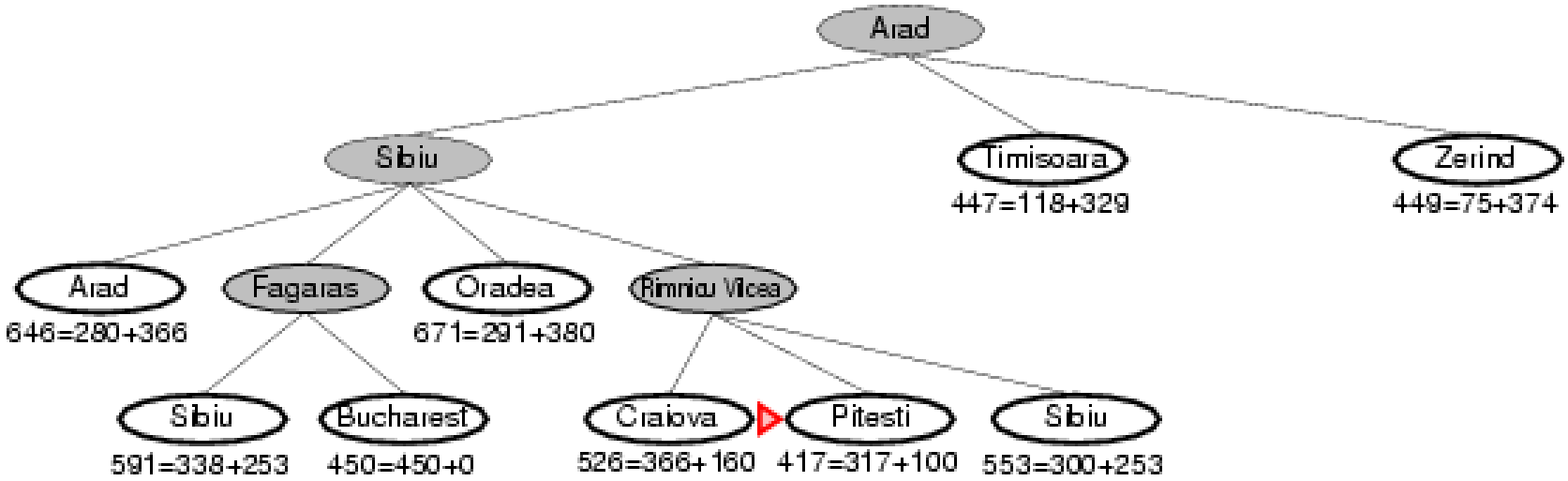
A* search – Ví dụ (3)



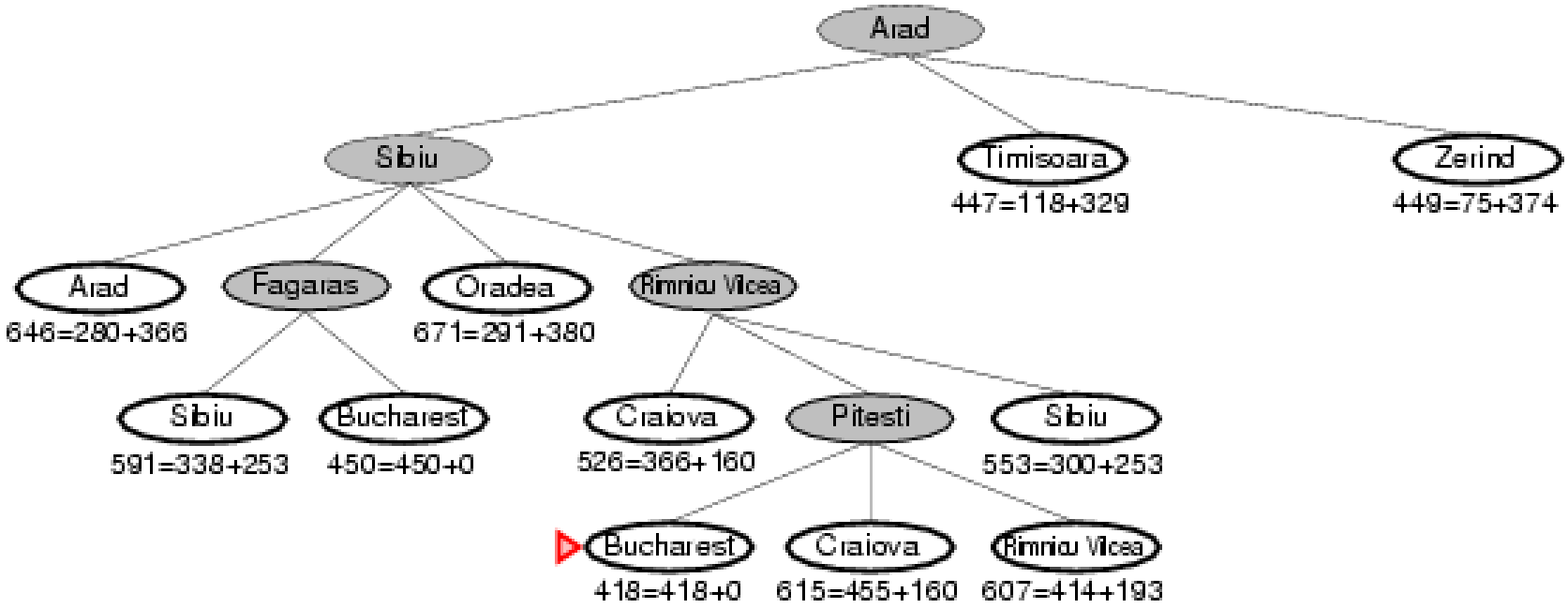
A* search – Ví dụ (4)



A* search – Ví dụ (5)



A* search – Ví dụ (6)



A* search: các đặc điểm

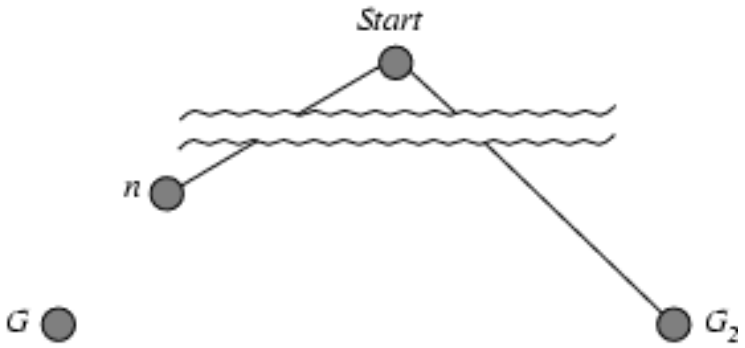
- Tính hoàn chỉnh?
  - *Không gian trạng thái là hữu hạn* và *có giải pháp để tránh việc xét (lặp) lại các trạng thái:* A* là hoàn chỉnh (tìm được lời giải)
  - K*hông gian trạng thái là hữu hạn* và *không có giải pháp để tránh việc xét (lặp) lại các trạng thái:* A* là không hoàn chỉnh
  - Nếu *không gian các trạng thái là vô hạn:* A* là không hoàn chỉnh
- Khi nào thì A* tối ưu?

Các ước lượng chấp nhận được

- Ước lượng chấp nhận được H(n):
  - $0 \leq h(n) \leq h^*(n)$, $h^*(n)$ là chi phí thật (thực tế) từ nút *n* đến đích
  - Ước lượng chấp nhận được không đánh giá quá cao (overestimate) đối với chi phí để đi tới đích (đánh giá "lạc quan")
  - Ví dụ ước lượng $h_{SLD}(n)$ đánh giá thấp hơn khoảng cách đường đi thực tế
- **Định lý:** Nếu *h(n)* là đánh giá chấp nhận được, thì phương pháp tìm kiếm A^* sử dụng giải thuật `TREE-SEARCH` là tối ưu

Tính tối ưu của A* - Chứng minh (1)

- Giả sử có một đích không tối ưu (suboptimal goal) G_2 được sinh ra và lưu trong cấu trúc *fringe*. Gọi *n* là một nút chưa xét trong cấu trúc *fringe* sao cho *n* nằm trên một đường đi ngắn nhất đến một đích tối ưu (optimal goal) *G*
  $\rightarrow f(n) \geq f(G_2)$ (*)



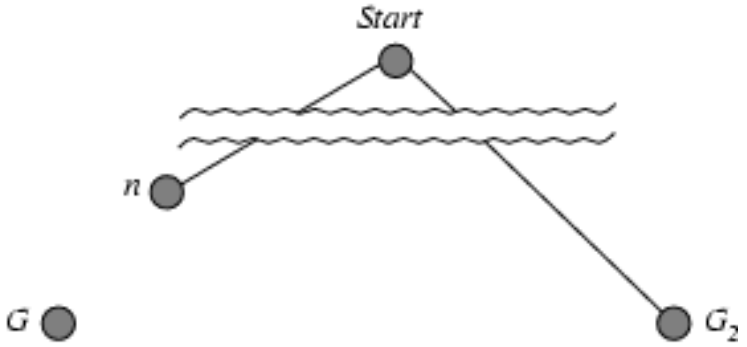
$h(G) = 0$
$h(G_2) = 0$
$g(G) < g(G_2)$

$f(x) = g(x) + h(x)$
$h(x) \leq h^*(x)$

- Ta có: 1) $f(G_2) = g(G_2)$ vì $h(G_2) = 0$
- Ta có: 2) $g(G_2) > g(G)$ vì G_2 là đích không tối ưu
- Ta có: 3) $f(G) = g(G)$ vì $h(G) = 0$
- Từ 1)+2)+3) suy ra: 4) $f(G_2) > f(G)$ (**)

Tính tối ưu của A* - Chứng minh (2)

- Ta có: 5) $h(n) \le h^*(n)$ vì *h* là ước lượng chấp nhận được
- Từ 5) suy ra: 6) $g(n) + h(n) \le g(n) + h^*(n)$
- Ta có: 7) $g(n) + h^*(n) = f(G)$ vì *n* nằm trên đường đi tới *G*
- Từ 6)+7) suy ra: 8) $f(n) \le f(G)$ (***)



$f(n) \ge f(G_2)$ (*)
$f(G_2) > f(G)$ (**)
$f(n) \le f(G)$ (***)

$\rightarrow$ *Mâu thuẫn*

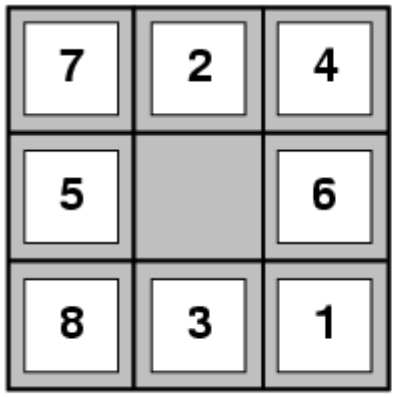
(Tức là, giải thuật A* không xét G_2 trước G)

Các ước lượng chấp nhận được (1)

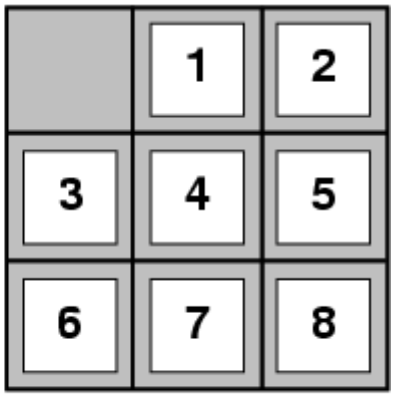
Ví dụ đối với trò chơi ô chữ 8 số:

- $h_1(n)$ = số các ô chữ nằm ở sai vị trí (so với vị trí của ô chữ đấy ở trạng thái đích)
- $h_2(n)$ = khoảng cách dịch chuyển ($\leftarrow,\rightarrow,\uparrow,\downarrow$) ngắn nhất để dịch chuyển các ô chữ nằm sai vị trí về vị trí đúng

- $h_1(S)$ = ?
- $h_2(S)$ = ?



Start State



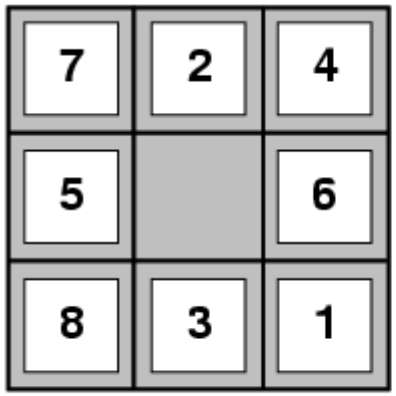
Goal State

Các ước lượng chấp nhận được (2)

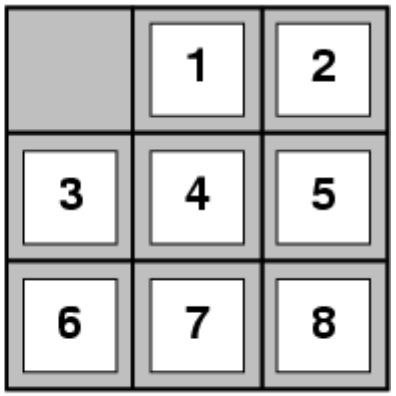
Ví dụ đối với trò chơi ô chữ 8 số:

- $h_1(n)$ = số các ô chữ nằm ở sai vị trí (so với vị trí của ô chữ đấy ở trạng thái đích)
- $h_2(n)$ = khoảng cách dịch chuyển ($\leftarrow,\rightarrow,\uparrow,\downarrow$) ngắn nhất để dịch chuyển các ô chữ nằm sai vị trí về vị trí đúng

- $h_1(S) = 8$
- $h_2(S) = 3+1+ 2+2+ 2+3+ 3+2 = 18$



Start State



Goal State

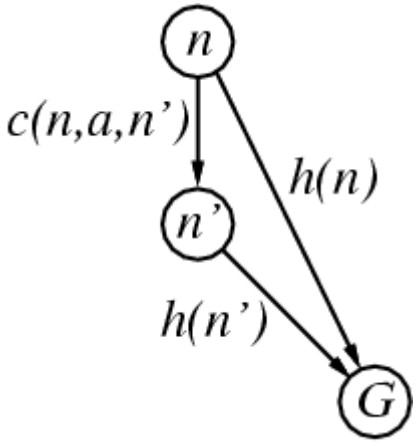
Ước lượng ưu thế

- Ước lượng h_2 được gọi là ưu thế hơn / trội hơn (dominate) ước lượng h_1 nếu:
  - $h^*(n) \geq h_2(n) \geq h_1(n)$ đối với tất cả các nút n
- Nếu ước lượng h_2 ưu thế hơn ước lượng h_1, thì h_2 tốt hơn (nên được sử dụng hơn) cho quá trình tìm kiếm
- Trong ví dụ (ô chữ 8 số) ở trên: Chi phí tìm kiếm = Số lượng trung bình của các nút phải xét:
  - Với độ sâu d =12
    - IDS (Tìm kiếm sâu dần): 3.644.035 nút phải xét
    - A*(sử dụng ước lượng h_1): 227 nút phải xét
    - A*(sử dụng ước lượng h_2): 73 nút phải xét
  - Với độ sâu d =24
    - IDS (Tìm kiếm sâu dần): Quá nhiều nút phải xét
    - A*(sử dụng ước lượng h_1): 39.135 nút phải xét
    - A*(sử dụng ước lượng h_2): 1.641 nút phải xét

Các ước lượng kiên định

- Một ước lượng *h* được xem là kiên định (consistent), nếu với mọi nút *n* và với mọi nút tiếp theo *n'* của *n* (được sinh ra bởi hành động *a*):

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



- Nếu ước lượng *h* là kiên định, ta có:

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

  $= g(n) + c(n,a,n') + h(n')$

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

  Nghĩa là: *f(n)* không giảm trong bất kỳ đường đi (tìm kiếm) nào đi qua *n*

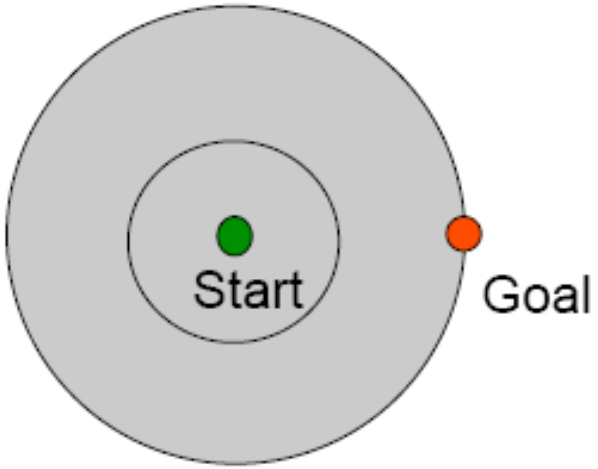
- **Định lý:** Nếu *h(n)* là kiên định, thì phương pháp tìm kiếm A* sử dụng giải thuật `GRAPH-SEARCH` là tối ưu

Các đặc điểm của A*

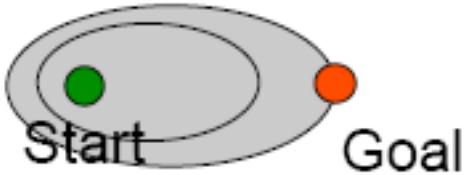
- Tính hoàn chỉnh?
  - Có (trừ khi có rất nhiều các nút có chi phí f ≤ *f(G)*)
- Độ phức tạp về thời gian?
  - Bậc của hàm mũ – Số lượng các nút được xét là hàm mũ của độ dài đường đi của lời giải
- Độ phức tạp về bộ nhớ?
  - Lưu giữ tất cả các nút trong bộ nhớ
- Tính tối ưu?
  - Có (đối với điều kiện đặc biệt)

A* vs. UCS

- Tìm kiếm với chi phí cực tiểu (UCS) phát triển theo mọi hướng



- Tìm kiếm A* phát triển chủ yếu theo hướng tới đích, nhưng đảm bảo tính tối ưu

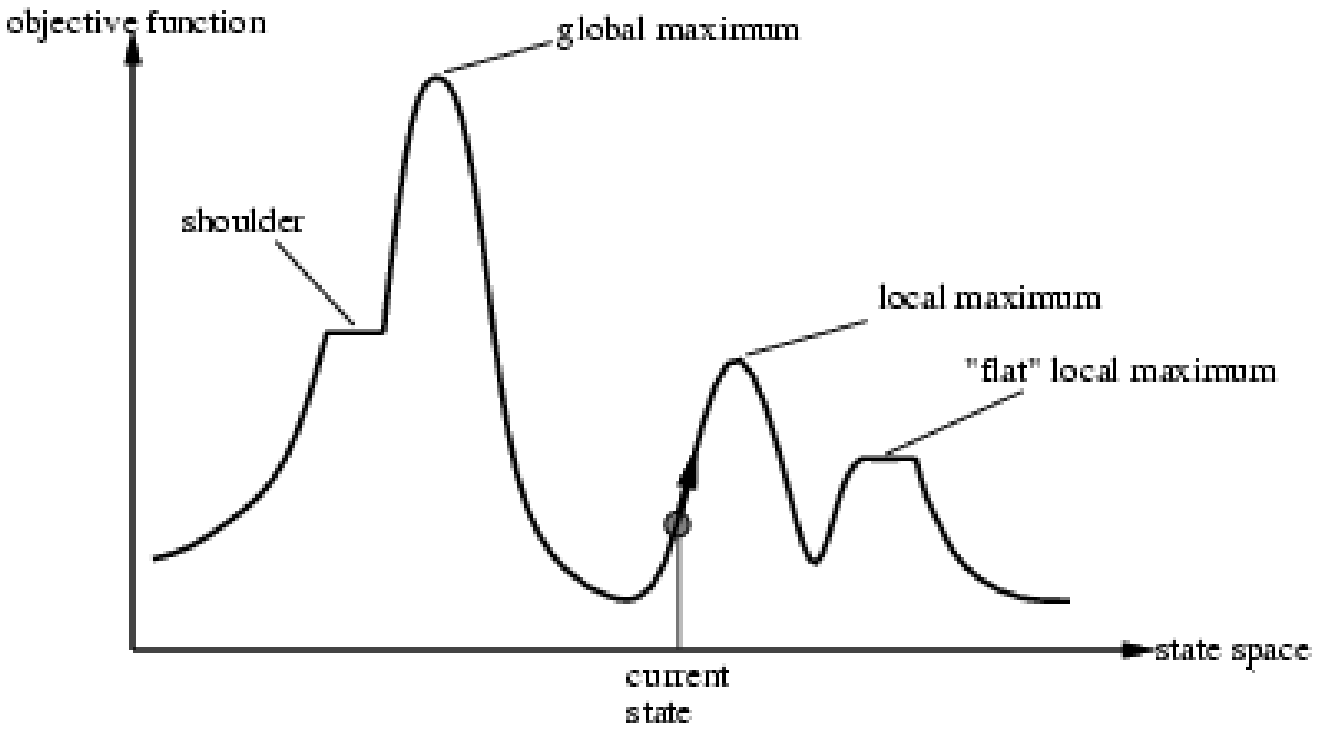


Các giải thuật tìm kiếm cục bộ

- Trong nhiều bài toán tối ưu, các điều kiệu thường phức tạp, khó tìm được lời giải bằng sử dụng đường đi để khảo sát không gian tìm kiếm (như các thuật toán đã xem xét):
  - Tìm trạng thái đích (không phải tìm các bước, hành động đến đích).
  - Bài toán tối ưu có nhiều điểm cực trị địa phương.
- Các giải thuật tìm kiếm cục bộ
  - Có thể sử dụng cho những bài toán kể trên
  - Chỉ lưu một trạng thái, không lưu hành động dẫn đến trạng thái. Mục tiêu: cải thiện trạng thái hiện thời theo một tiêu chí nào đó.
  - Bài toán 8 quân hậu: tìm vị trí các quân mà không quan tâm tới đường đi dẫn tới trạng thái đó.

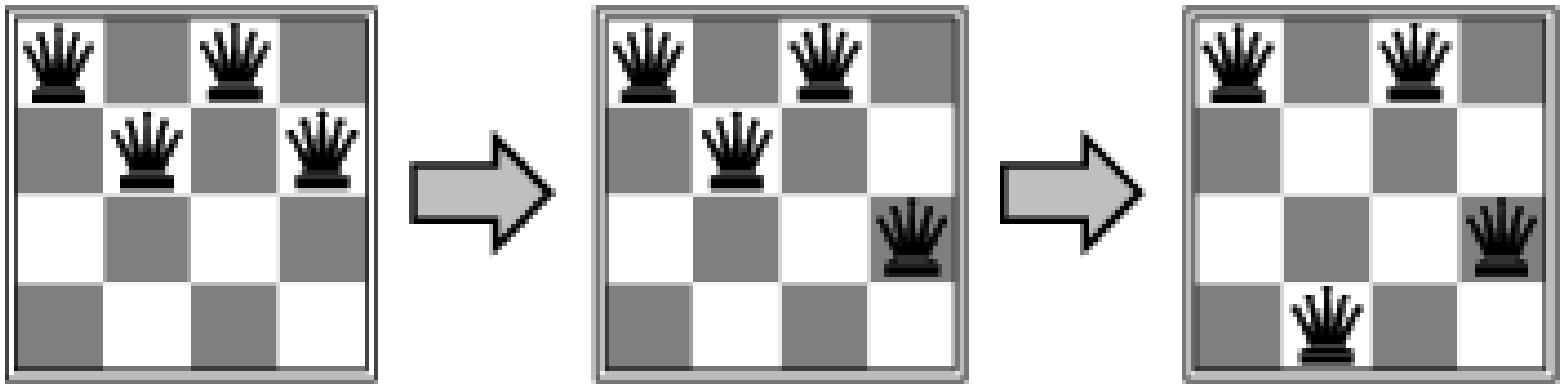
Tìm kiếm leo đồi: minh họa

- Cơ chế: tạo ra hàng xóm cho trạng thái hiện thời và di chuyển sang hàng xóm có hàm mục tiêu tốt hơn
- Nhược điểm:
  - Có thể "tắc" ở các điểm cực đại cục bộ (local maxima)
  - Không tìm được lời giải tối ưu toàn cục (global optimal solution)



Ví dụ: Bài toán *n* quân hậu

- Bố trí *n* (=4) quân hậu trên một bàn cờ có kích thước *n*×*n*, sao cho không có 2 quân hậu nào trên cùng hàng, hoặc trên cùng cột, hoặc trên cùng đường chéo

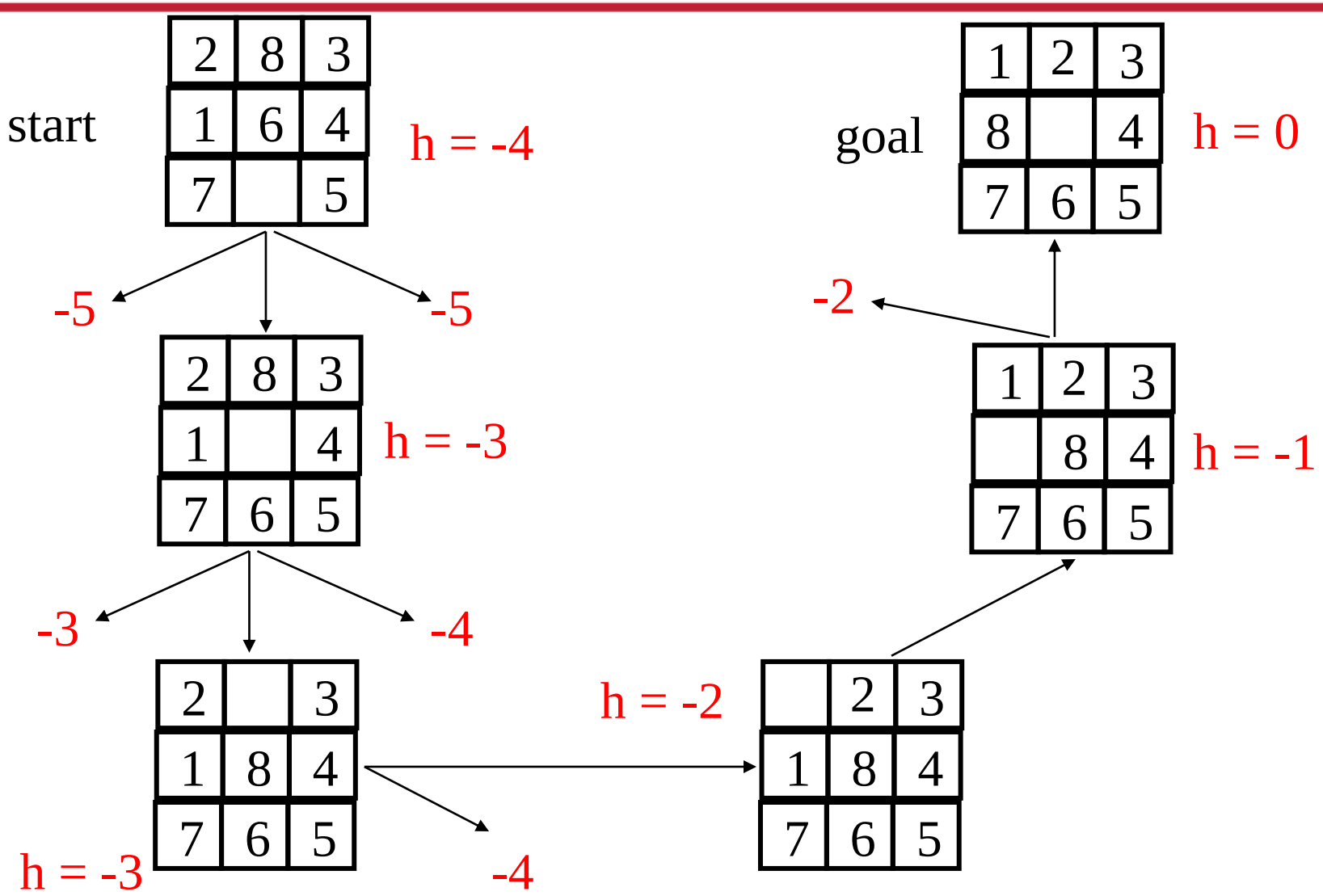


Tìm kiếm leo đồi: giải thuật

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
    inputs: problem, a problem
    local variables: current, a node
                     neighbor, a node

    current ← MAKE-NODE(INITIAL-STATE[problem])
    loop do
        neighbor ← a highest-valued successor of current
        if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
        current ← neighbor
```

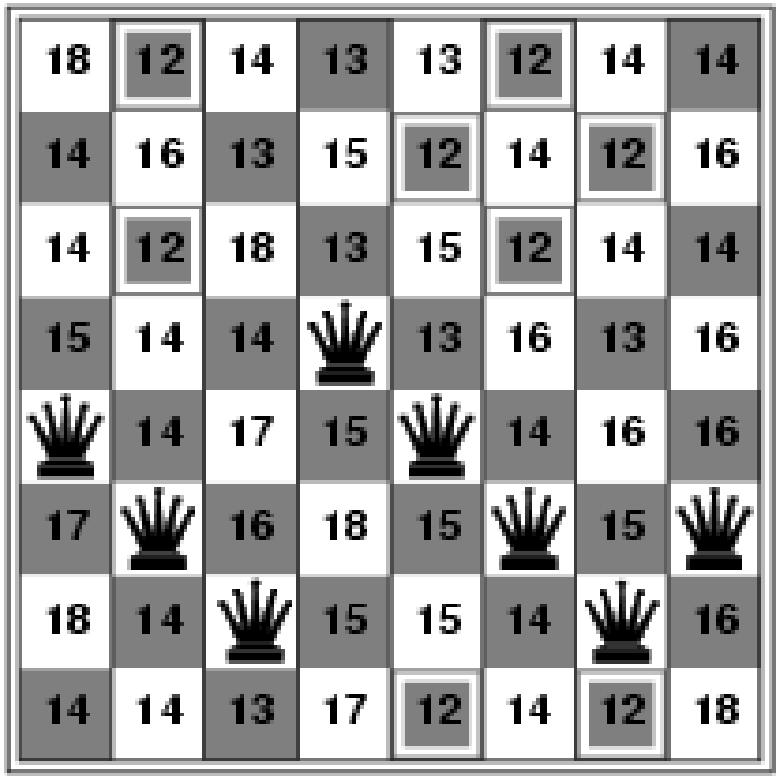
Tìm kiếm leo đồi: bài toán ô chữ



f(n) = -(Số lượng các ô chữ nằm sai vị trí)

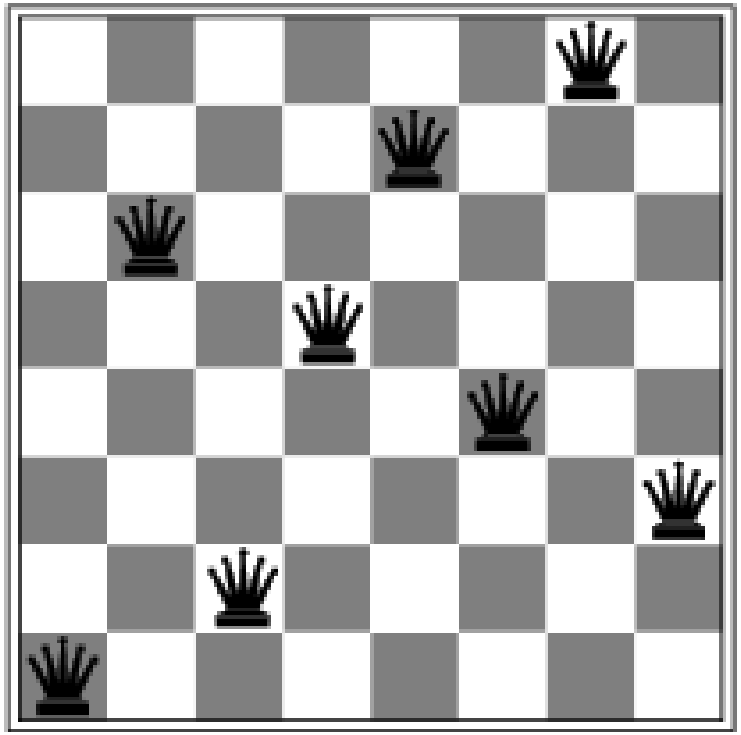
Tìm kiếm leo đồi: bài toán 8 quân hậu (1)

- Ước lượng *h* = tổng số các cặp quân hậu ăn nhau (f(n) = -h)
- Trong trạng thái (bàn cờ) trên: *h* =17



Qua mỗi thế hệ, các cá thể được đánh giá, xắp xếp. Các cá thể thích nghi cao nhất được giữ lại và sinh ra thế hệ tiếp theo.

Tìm kiếm leo đồi: bài toán 8 quân hậu (2)



- Trạng thái bàn cờ trên là một giải pháp tối ưu cục bộ (a local minimum)
  - Với ước lượng *h* =1 (vẫn còn 1 cặp hậu ăn nhau)

Simulated annealing search

- Dựa trên quá trình tôi ủ (annealing process): Kim loại nguội đi và lạnh cứng lại thành cấu trúc kết tinh
- Phương pháp tìm kiếm Simulated Annealing có thể tránh được các điểm tối ưu cục bộ (local optima)
- Phương pháp tìm kiếm Simulated Annealing sử dụng chiến lược tìm kiếm ngẫu nhiên, trong đó chấp nhận các thay đổi làm tăng giá trị hàm mục tiêu (i.e., cần tối ưu) và *cũng chấp nhận (có hạn chế) các thay đổi làm giảm*
- Phương pháp tìm kiếm Simulated Annealing sử dụng một tham số điều khiển *T* (như trong các hệ thống nhiệt độ)
  - Bắt đầu thì *T* nhận giá trị cao, và giảm dần về 0

Simulated annealing search: giải

- Ý tưởng: Thoát khỏi (vượt qua) các điểm tối ưu cục bộ bằng cách cho phép cả các dịch chuyển "tồi" từ trạng thái hiện thời, nhưng giảm dần tần xuất của các di chuyển tồi này

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    inputs: problem, a problem
            schedule, a mapping from time to "temperature"
    local variables: current, a node
                     next, a node
                     T, a "temperature" controlling prob. of downward steps

    current ← MAKE-NODE(INITIAL-STATE[problem])
    for t ← 1 to ∞ do
        T ← schedule[t]
        if T = 0 then return current
        next ← a randomly selected successor of current
        ΔE ← VALUE[next] – VALUE[current]
        if ΔE > 0 then current ← next
        else current ← next only with probability e^(Δ E/T)
```

Simulated annealing search: các đặc điểm

- (Có thể chứng minh được) Nếu giá trị của tham số *T* (xác định mức độ giảm tần xuất đối với các di chuyển tồi) giảm chậm, thì phương pháp tìm kiếm Simulated Annealing sẽ tìm được lời giải tối ưu toàn cục với xác suất xấp xỉ 1

- Phương pháp tìm kiếm Simulated Annealing Search rất hay được sử dụng trong các lĩnh vực: thiết kế sơ đồ bảng mạch VLSI, lập lịch bay, …

Local beam search

- Ở mỗi thời điểm (trong quá trình tìm kiếm), luôn lưu giữ *k* – thay vì chỉ 1 – trạng thái tốt nhất
- Bắt đầu giải thuật: Chọn *k* trạng thái ngẫu nhiên
- Ở mỗi bước tìm kiếm, sinh ra tất cả các trạng thái kế tiếp của *k* trạng thái này
- Nếu một trong số các trạng thái là trạng thái đích, thì giải thuật kết thúc (thành công); nếu không, thì chọn *k* trạng thái tiếp theo tốt nhất (từ tập các trạng thái tiếp theo), và lặp lại bước trên

Giải thuật di truyền: giới thiệu

- Dựa trên (bắt chước) quá trình tiến hóa tự nhiên trong sinh học
- Áp dụng phương pháp tìm kiếm ngẫu nhiên (stochastic search) để tìm được lời giải (vd: một hàm mục tiêu, một mô hình phân lớp, …) tối ưu
- Giải thuật di truyền (Genetic Algorithm – GA) có khả năng tìm được các lời giải tốt thậm chí ngay cả với các không gian tìm kiếm (lời giải) không liên tục rất phức tạp
- Mỗi khả năng của lời giải được biểu diễn bằng một chuỗi nhị phân (vd: 100101101) – được gọi là **nhiễm sắc thể (chromosome)**
    - Việc biểu diễn này phụ thuộc vào từng bài toán cụ thể

Giải thuật di truyền: mô tả

- Xây dựng (khởi tạo) **quần thể (population) ban đầu**
  - Tạo nên một số các giả thiết (khả năng của lời giải) ban đầu
  - Mỗi giả thiết khác các giả thiết khác (vd: khác nhau đối với các giá trị của một số tham số nào đó của bài toán)
- Đánh giá quần thể
  - Đánh giá (cho điểm) mỗi giả thiết (vd: bằng cách kiểm tra độ chính xác của hệ thống trên một tập dữ liệu kiểm thử)
  - Trong lĩnh vực sinh học, điểm đánh giá này của mỗi giả thiết được gọi là **độ phù hợp (fitness)** của giả thiết đó
  - Xếp hạng các giả thiết theo mức độ phù hợp của chúng, và chỉ giữ lại các giả thiết tốt nhất (gọi là **các giả thiết phù hợp nhất – survival of the fittest**)
- Sản sinh ra **thế hệ tiếp theo (next generation)**
  - Thay đổi ngẫu nhiên các giả thiết để sản sinh ra thế hệ tiếp theo (gọi là **các con cháu – offspring**)
- Lặp lại quá trình trên cho đến khi ở một thế hệ nào đó có giả thiết tốt nhất có độ phù hợp cao hơn giá trị phù hợp mong muốn (định trước)

GA(Fitness, θ, n, r_{co}, r_{mu})

Fitness: A function that produces the score (fitness) given a hypothesis

θ: The desired fitness value (i.e., a threshold specifying the termination condition)

n: The number of hypotheses in the population

r_{co}: The percentage of the population influenced by the *crossover* operator at each step

r_{mu}: The percentage of the population influenced by the *mutation* operator at each step

Initialize the population: H ← Randomly generate n hypotheses

Evaluate the initial population. For each h∈H: compute Fitness(h)

while ($\max_{\{h \in H\}}$ Fitness(h) < θ) do

$H^{next} \leftarrow \varnothing$

Reproduction (Replication). Probabilistically select $(1-r_{co}).n$ hypotheses of H to add to H^{next}.

The probability of selecting hypothesis h_i from H is:

$$P(h_i) = \frac{Fitness(h_i)}{\sum_{j=1}^{n} Fitness(h_j)}$$

GA(Fitness, θ, n, r_{co}, r_{mu})

…

Crossover.

Probabilistically select (r_{co}.n/2) pairs of hypotheses from H, according to the probability computation $P(h_i)$ given above.

For each pair (h_i, h_j), produce two offspring (i.e., children) by applying the crossover operator. Then, add all the offspring to H^{next}.

Mutation.

Select (r_{mu}.n) hypotheses of H^{next}, with uniform probability.

For each selected hypothesis, invert one randomly chosen bit (i.e., 0 to 1, or 1 to 0) in the hypothesis's representation.

Producing the **next generation**: $H \leftarrow H^{next}$

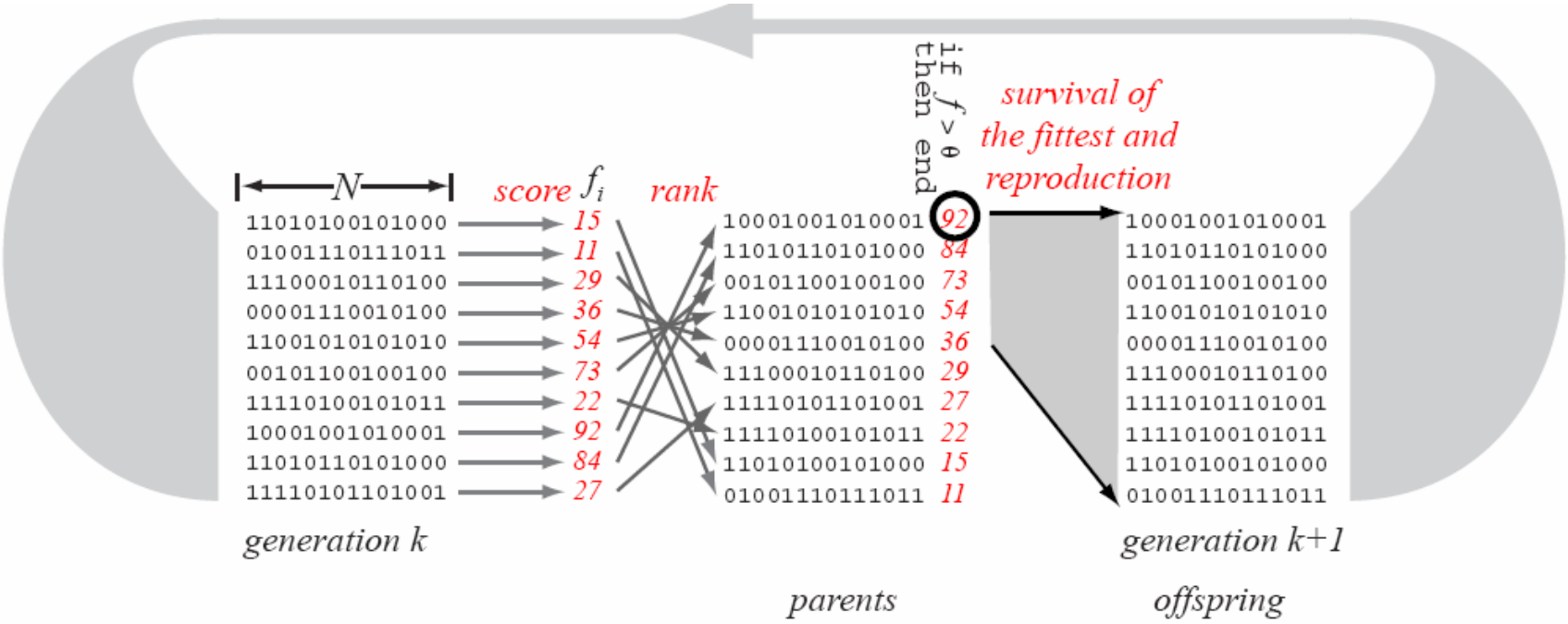
Evaluate the new population. For each $h \in H$: compute Fitness(h)

end while

return $\text{argmax}_{\{h \in H\}}$Fitness(h)

Giải thuật di truyền: minh họa

[Duda et al., 2000]



Các toán tử di truyền

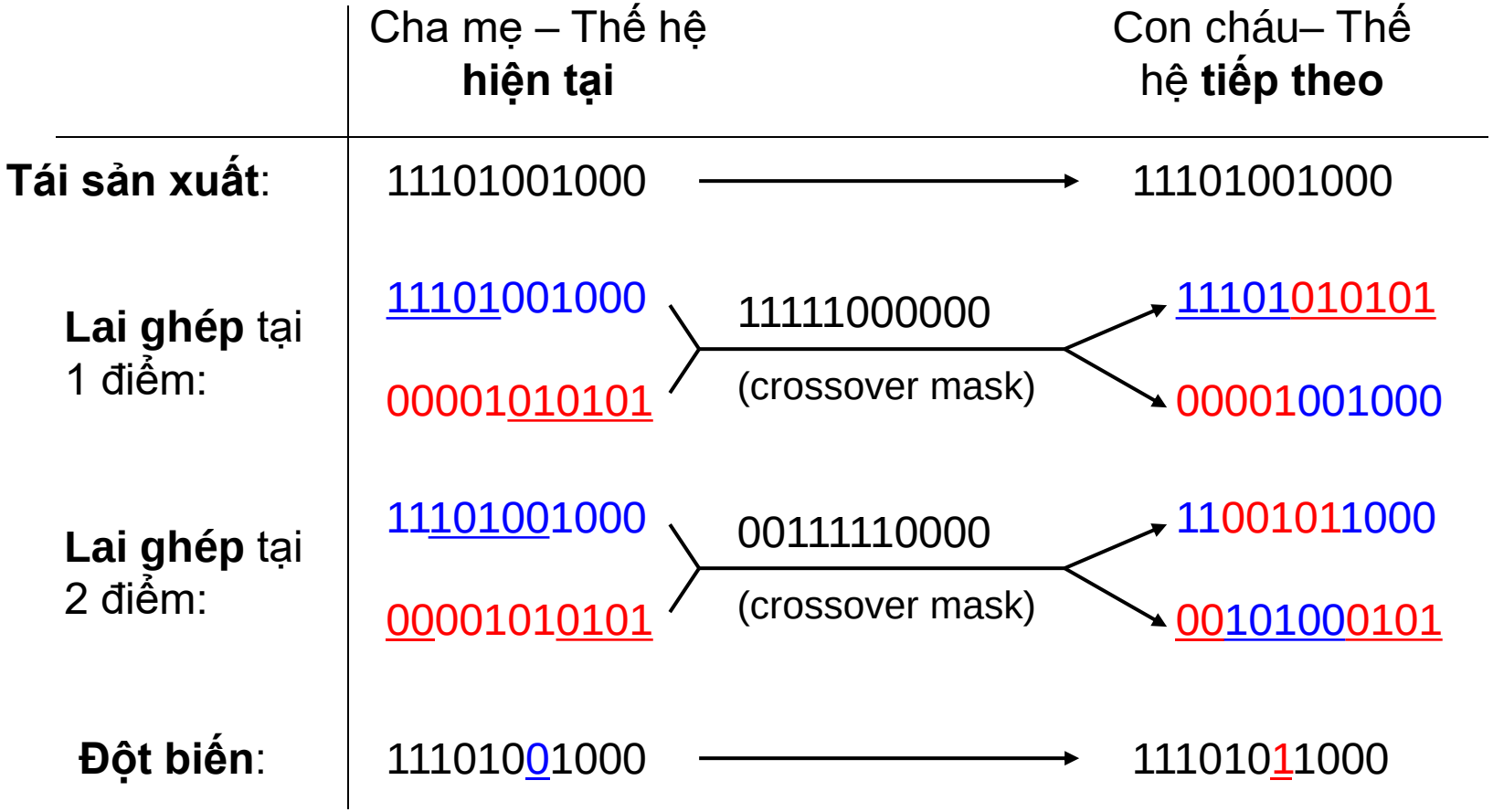
- 3 toán tử di truyền được sử dụng để sinh ra các cá thể con cháu (offspring) trong thế hệ tiếp theo
  - Nhưng chỉ có 2 toán tử lai ghép (*crossover*) và đột biến (*mutation*) tạo nên sự thay đổi
- **Tái sản xuất (Reproduction)**

  → Một giả thiết được giữ lại (không thay đổi)
- **Lai ghép (Crossover)** để sinh ra 2 cá thể mới

  → Ghép ("phối hợp") của hai cá thể cha mẹ
  - Điểm lai ghép được chọn ngẫu nhiên (trên chiều dài của nhiễm sắc thể)
  - Phần đầu tiên của nhiễm sắc thể h_i được ghép với phần sau của nhiễm sắc thể h_j, và ngược lại, để sinh ra 2 nhiễm sắc thể mới
- **Đột biến (Mutation)** để sinh ra 1 cá thể mới

  →Chọn ngẫu nhiên một bit của nhiễm sắc thể, và đổi giá trị (0→1 / 1→0)
  - Chỉ tạo nên một thay đổi nhỏ và ngẫu nhiên đối với một cá thể cha mẹ!

Các toán tử di truyền: ví dụ



[Mitchell, 1997]

Toán tử lai ghép: ví dụ

Bài toán bố trí 8 quân hậu trên bàn cờ - Toán tử lai ghép (crossover)