Các Thuật Toán Thông Minh Nhân Tạo & Ứng Dụng

Chương 2

Solving Problems by Searching

Giảng viên: Thái Hùng Văn

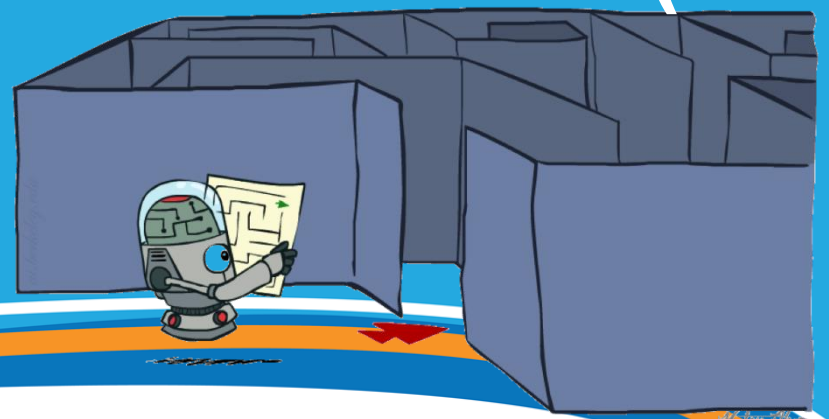
Email: thvan@fit.hcmus.edu.vn

Khoa Công Nghệ Thông Tin
Trường Đại Học Khoa Học Tự Nhiên
ĐHQG-HCM

Các Thuật Toán Thông Minh Nhân Tạo & Ứng Dụng

2.1

CÁC CHIẾN LƯỢC TÌM KIẾM KHÔNG CÓ THÔNG TIN (TÌM KIẾM MÙ)



Khoa Công Nghệ Thông Tin
Trường Đại Học Khoa Học Tự Nhiên
VNU-HCMC

Nội dung

* Bài toán tìm kiếm

  + Giới thiệu

    - Các thành phần
    - Phát biểu dưới dạng toán
    - Dạng tổng quát

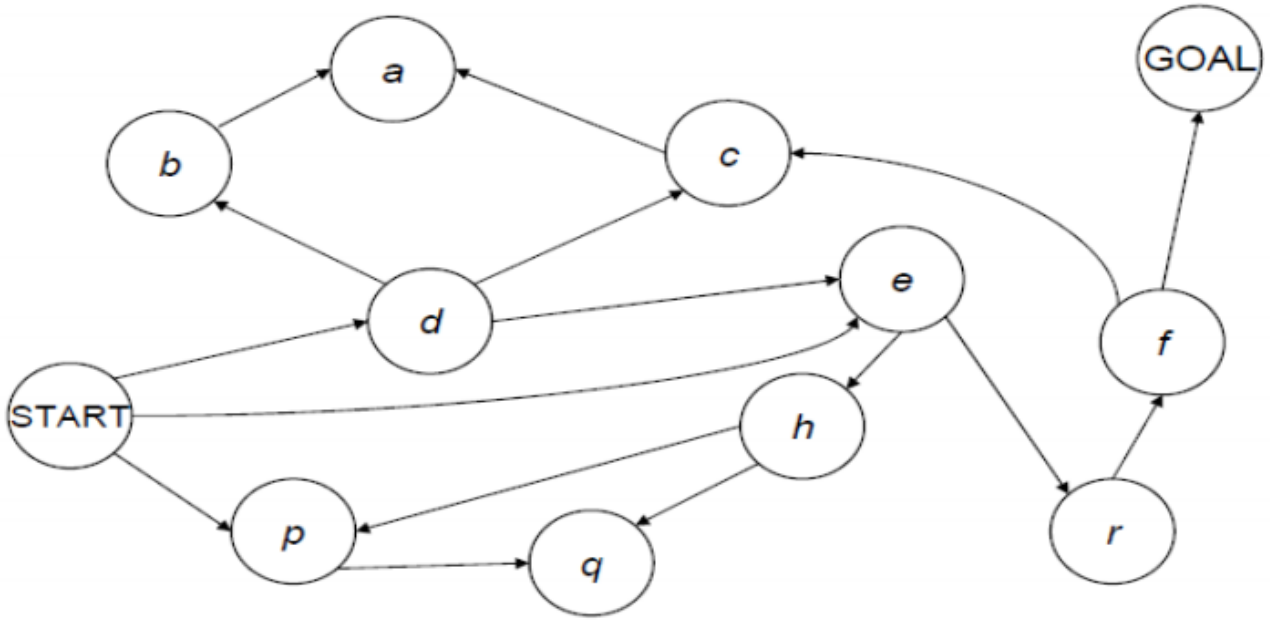
  + Đồ thị không gian trạng thái
  + Cây tìm kiếm
  + Trạng thái và Nút
  + Tạo cây tìm kiếm từ đồ thị trạng thái
  + Tìm kiếm & mở rộng cây

* Những phương pháp tìm kiếm mù

  - Tìm kiếm theo chiều sâu (DFS – Depth First Search)
  - Tìm kiếm theo chiều rộng (BFS – Breadth First Search)
  - Tìm kiếm chi phí đồng nhất (UCS – Uniform Cost Search)

Bài toán tìm kiếm – giới thiệu

- Rất nhiều vấn đề /bài toán phức tạp trên thực tế có thể quy về dạng "tìm đường đi trong đồ thị"
- Thông thường, bài toán sẽ trở thành: "xuất phát từ 1 đỉnh trong đồ thị, tìm đường hiệu quả nhất đến 1 đỉnh nào đó".



Bài toán tìm kiếm – các thành phần

- Mỗi trạng thái có thể xảy ra trong thế giới bài toán được biểu diễn thành một đỉnh trên đồ thị. Không gian trạng thái của bài toán sẽ là một đồ thị.
- Cung nối từ đỉnh X đến đỉnh Y tương ứng với hành động để đạt được trạng thái Y từ trạng thái X, chi phí thực hiện (nếu có) là trọng số của cung. *(giữa 2 đỉnh bất kỳ có thể có hoặc không có cung nối)*
- Lời giải của bài toán là 1 đường đi trong không gian trạng thái có điểm đầu là trạng thái đầu và điểm cuối là trạng thái đích

Bài toán tìm kiếm - phát biểu dưới dạng toán

Bài toán tìm kiếm tổng quát có thể phát biểu: Cho trước 2 trạng thái T0 & TG, xác định chuỗi trạng thái T0, T1, T2, T3, …, Tn=TG sao cho $\sum_1^n cost(T_{i-1},\ Ti)$ thỏa 1 điều kiện cho trước (*thường là nhỏ nhất*)

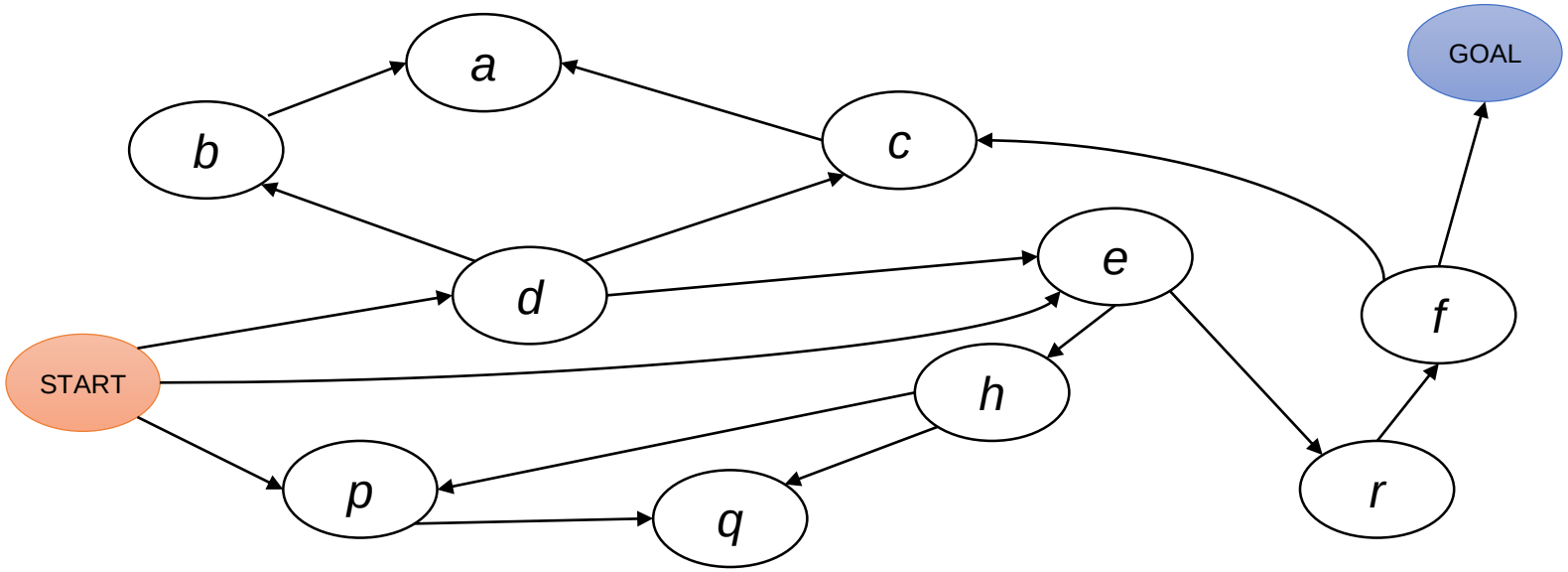
- $\mathbf{T_i}$ thuộc tập hợp không gian trạng thái **Q** bao gồm tất cả các trạng thái có thể có, và **cost($\mathbf{T_{i-1}}$, $\mathbf{T_i}$)** là chi phí để biến đổi từ trạng thái $\mathbf{T_{i-1}}$ sang $\mathbf{T_i}$
- Không gian trạng thái **Q** là 1 đồ thị có hướng, với mỗi nút (đỉnh) $\mathbf{T_i}$ là 1 trạng thái, và chi phí **cost($\mathbf{T_{i-1}}$, $\mathbf{T_i}$)** là trọng số của cung nối $\mathbf{T_{i-1}}$ sang $\mathbf{T_i}$

Bài toán tìm kiếm – dạng tổng quát

Tổng quát hơn, bài toán giải quyết vấn đề bằng phương pháp tìm kiếm bao gồm:

- **Q** : một tập hữu hạn các trạng thái - Không gian trạng thái
- $\mathbf{S} \subset \boldsymbol{Q}$: một tập khác rỗng các trạng thái bắt đầu (không nhất thiết chỉ là 1 trạng thái)
- $\mathbf{G} \subset Q$ một tập khác rỗng các trạng thái đích.
- **Succs** : $\mathbf{Q} \rightarrow \mathbf{P}(\mathbf{Q})$ là một hàm nhận một trạng thái làm đầu vào và trả về kết quả là một tập trạng thái. **Succs(s)** là tập các trạng thái có thể đến từ **s** (trong một bước)
- **cost**: $\mathbf{Q} \rightarrow$ **giá trị dương** , là một hàm nhận hai trạng thái **s** và **s'** làm đầu vào và trả về chi phí của việc chuyển từ **s** đến **s'** (trong một bước)

Bài toán tìm kiếm – ví dụ



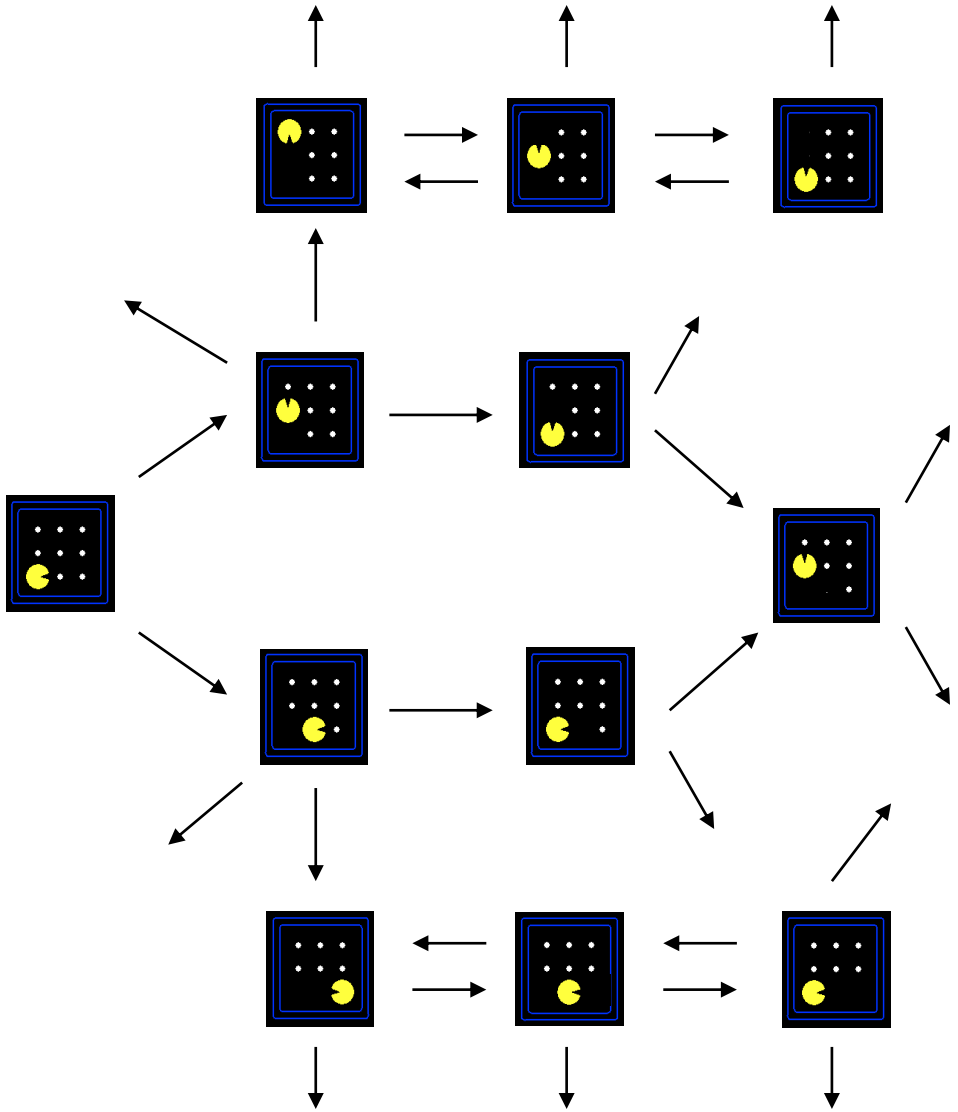
- $Q = \{START, a, b, c, d, e, f, h, p, q, r, GOAL\}$
- $S = \{START\}$
- $G = \{GOAL\}$
- Succs: $succs(b) = \{a\}$, $succs(e) = \{h, r\}$, $succs(a) = NULL$ ...
- $cost(s, s') = 1$ cho tất cả các biến đổi

Đồ thị không gian trạng thái – khái niệm

- Nhiều vấn đề /bài toán phức tạp trên thực tế có thể quy về bài toán tìm đường đi trong đồ thị - gọi tắt là bài toán tìm kiếm (TK).
- Bài toán TK trước tiên phải được biểu diễn dưới dạng toán học, cụ thể là phải xây dựng được đồ thị không gian trạng thái (KGTT), trong đó:
  - Các đỉnh là cấu hình thế giới (thế giới của bài toán), mỗi đỉnh là 1 TT (chỉ cần chứa các thông tin cần thiết cho bài toán - ko cần tới các chi tiết ngoài thế giới thực)
  - Các cung đi ra từ 1 đỉnh ứng với hàm Succs tại đỉnh đó (mỗi cung ứng với 1 hành động để đạt được TT kế)
  - Các goal test là một tập hợp các nút đích (có thể là một hoặc nhiều nút)
- Trong một đồ thị KGTT, mỗi TT chỉ xảy ra một lần!
- Trong thực tế, chúng ta hiếm khi có thể xây dựng đồ thị đầy đủ trong bộ nhớ (vì nó quá lớn)

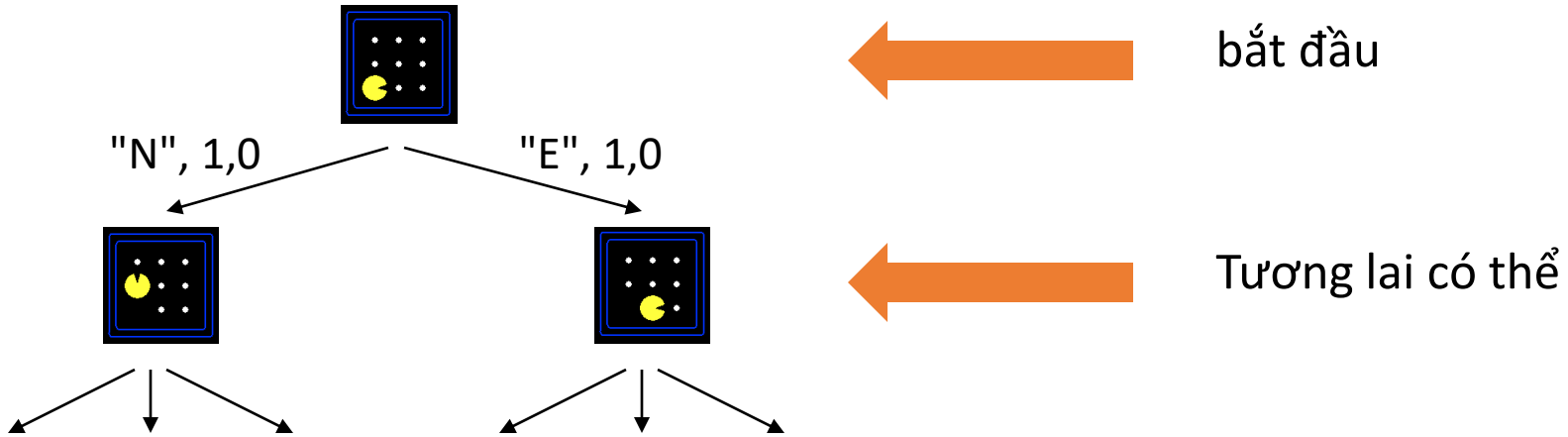
Đồ thị không gian trạng thái – ví dụ

Đồ thị KGTT thái Pacman (1 phần)



Hãy vẽ Đồ thị KGTT ứng với các giao lộ xung quanh trường KHTN (~15 đỉnh)

Cây tìm kiếm



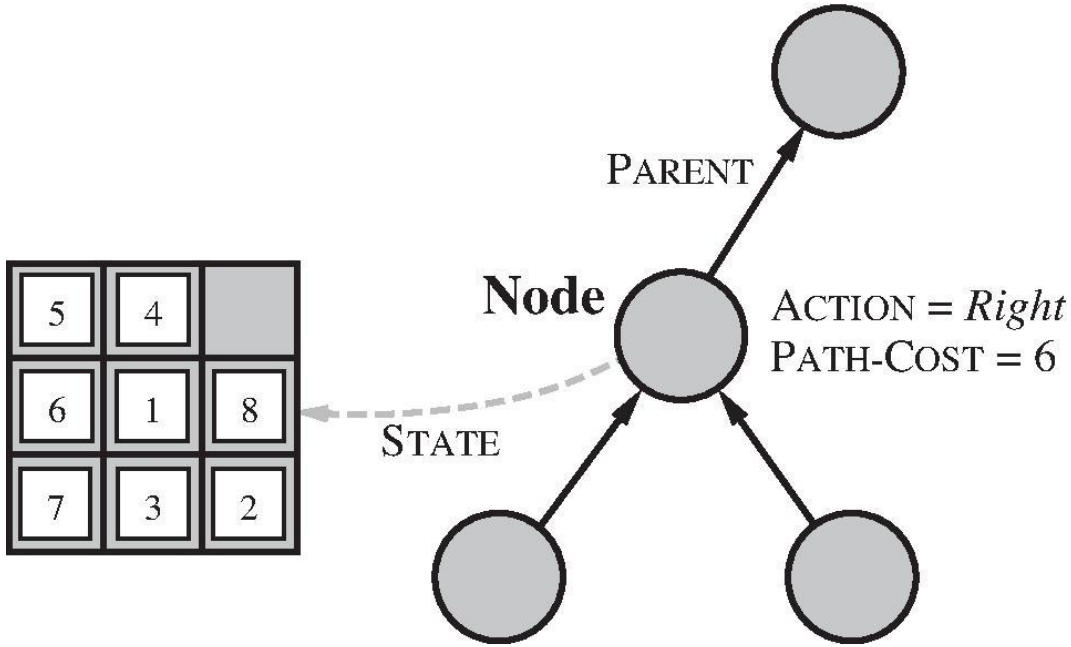
Cây TK sẽ phát sinh khi thực hiện hiện thao tác TK trên đồ thị, với:

- Nút gốc tương ứng với TT bắt đầu, là 1 đỉnh bất kỳ nào đó trên đồ thị.
- Các nút con của 1 nút X tương ứng với các TT kế tiếp của X, có được từ hàm successor(X), cũng chính là các đỉnh được nối đến bởi các cung đi ra từ đỉnh X trên đồ thị

// Đối với hầu hết các bài toán, chúng ta có thể không bao giờ thực sự xây dựng toàn bộ cây

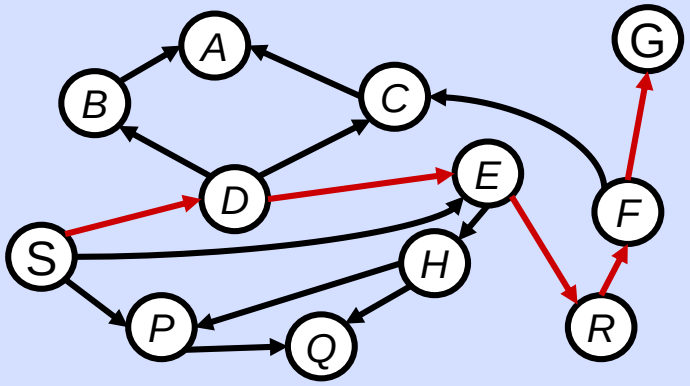
Trạng thái và Nút

- Một trạng thái (**state**) là một tình trạng trong thế giới bài toán, ứng với 1 đỉnh trên đồ thị KGTT.
- Một nút (**node**) trong cây TK là một cấu trúc dữ liệu thường chỉ chứa các thông tin về nút cha, các nút con, độ sâu, chi phí để đến nó (tính từ gốc), liên kết đến đỉnh tương ứng. (TT không có những thông tin này)

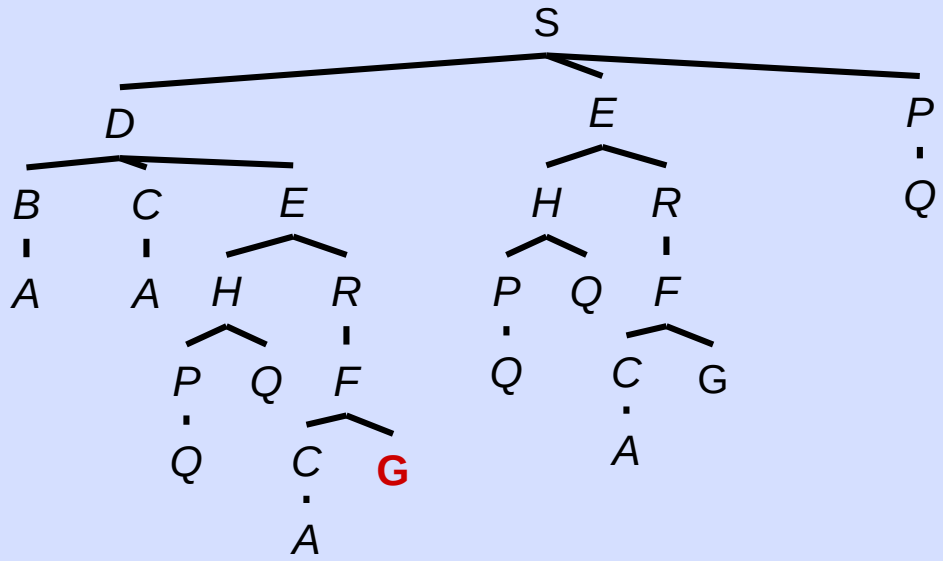


Tạo cây tìm kiếm từ Đồ thị trạng thái

Đồ thị không gian trạng thái



Cây tìm kiếm tương ứng với nút gốc là S

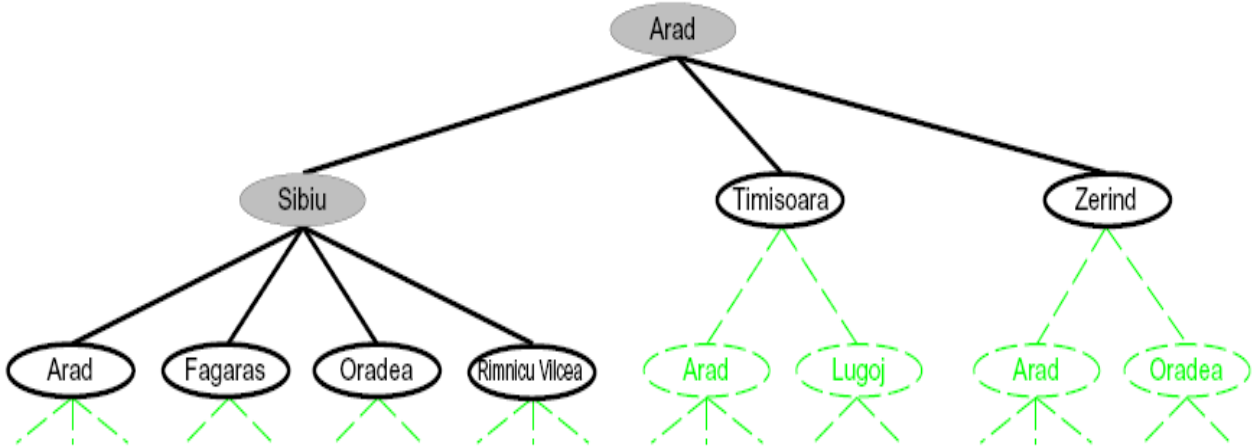


Mỗi nút trong cây thực ra không phải là 1 TT, mà là 1 kế hoạch /đường đi từ gốc đến nó. (*và vì vậy 1 TT trên đồ thị có thể ứng với nhiều nút trên cây)*

Cơ chế tìm kiếm & mở rộng cây

* Cây có thể không có sẵn mà sẽ hình thành dần trong quá trình TK – cho đến khi tìm ra nút kết quả (hoặc xác định được là không có)

* Cách thực hiện:
  - Để TK, cần có biến **frontier** (or fringe) lưu danh sách các kế hoạch tiềm năng (ứng với các node lá của cây đang xây dựng).
  - Mỗi lần lặp, lấy ra từ frontier 1 kế hoạch theo 1 chiến lược nào đó và mở rộng ra (dựa vào hàm successor), rồi thêm các kế hoạch mới này vào frontier (tức phát triển cây)
  - Lặp đến khi kế hoạch được lấy ra là TT đích, hoặc frontier rỗng (không có lời giải)



Thuật toán tìm kiếm & mở rộng cây

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

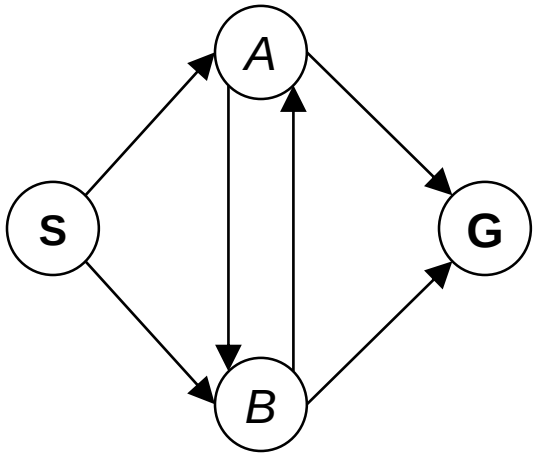
choose a leaf node and remove it from the frontier

if the node contains a goal state **then return** the corresponding solution

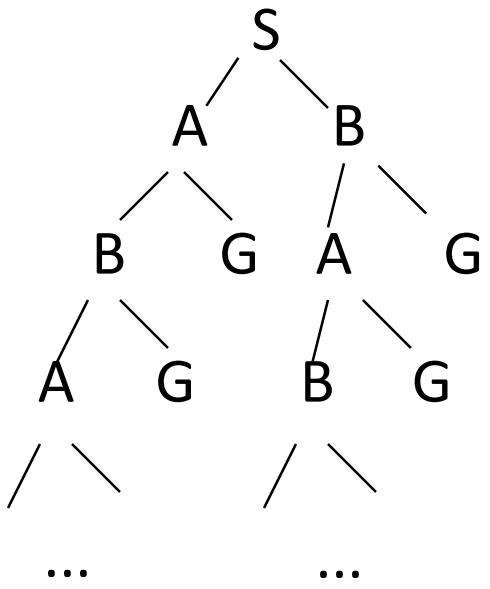
expand the chosen node, adding the resulting nodes to the frontier

Vấn đề của cây tìm kiếm

Đồ thị KGTT



Cây TK của nó (từ S) lớn thế nào?



Rất nhiều nút lặp lại trong cây TK!

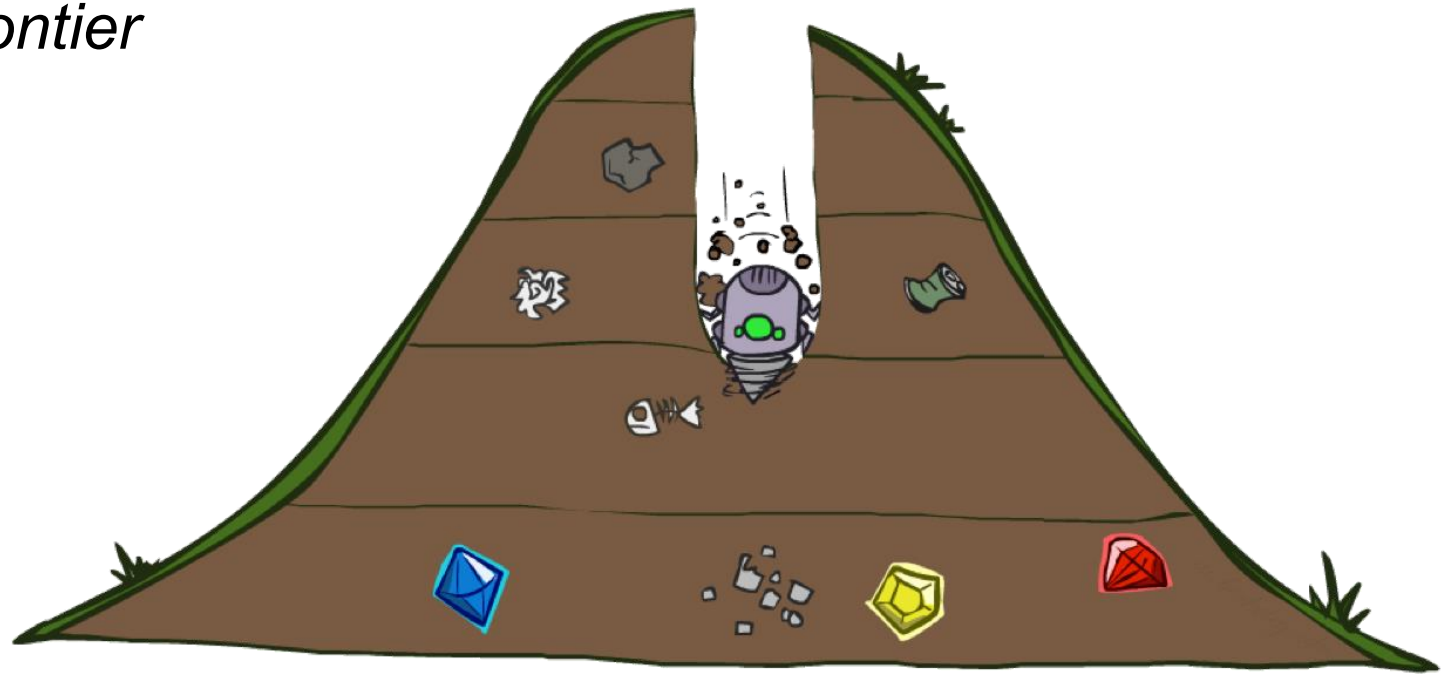
Các thuật toán tìm kiếm trên cây

* Tư tưởng: từ TT bắt đầu (nút gốc), ta phát triển dần cây TK bằng cách phát sinh các nút con theo **một chiến lược nào đó** cho đến khi tìm được TT đích. (mỗi chiến lược sẽ ứng với 1 thuật toán).

* Tùy thuộc vào từng bài toán cụ thể mà lựa chọn chiến lược /thuật toán phù hợp để đạt hiệu quả tìm kiếm tốt nhất.

Tìm kiếm theo chiều sâu (DFS)

Chiến lược chọn kế hoạch ở frontier để mở rộng: chọn kế hoạch "sâu" nhất.
DFS duyệt cây theo hướng từ trái qua phải (hoặc từ phải qua trái tùy cài đặt)
Có thể dùng stack để cài đặt frontier



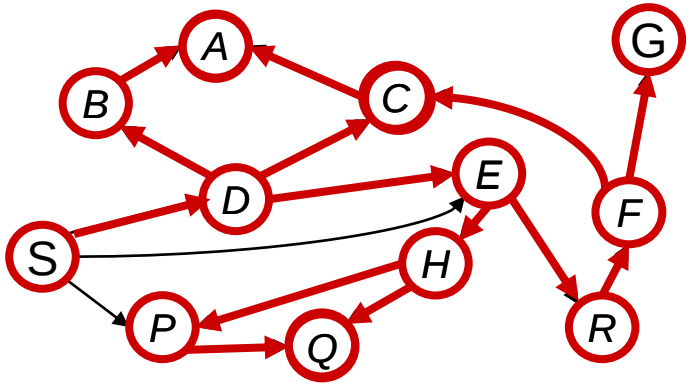
Thuật toán DFS

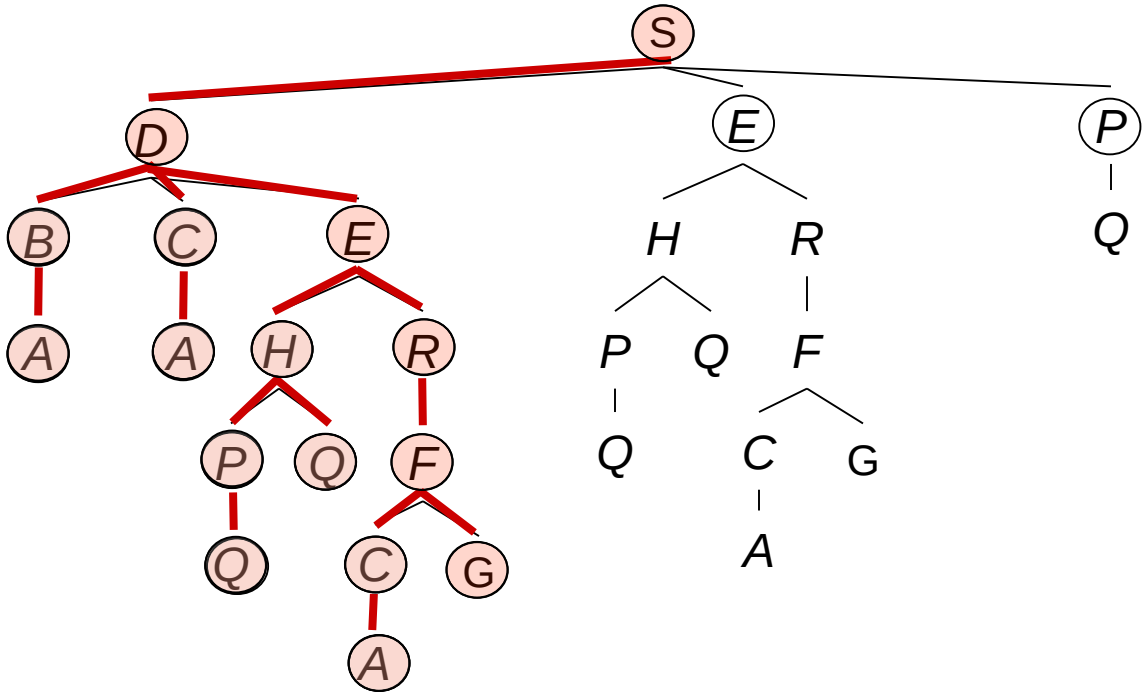
```
function DEPTH-FIRST-SEARCH(problem) returns a solution, or failure
    if problem's initial state is a goal then return empty path to initial state
    frontier ← a LIFO stack initially containing one path, for the problem's initial state
    reached ← a set of states; initially empty
    solution ← failure
    while frontier is not empty do
        parent ← the first node in frontier
        for child in successors(parent) do
            s ← child.state
            if s is a goal then
                return child
            if s is not in reached then
                add s to reached
                add child to the end of frontier
    return solution
```

Ví dụ thuật toán DFS

Chiến lược: mở rộng một nút sâu nhất trước tiên

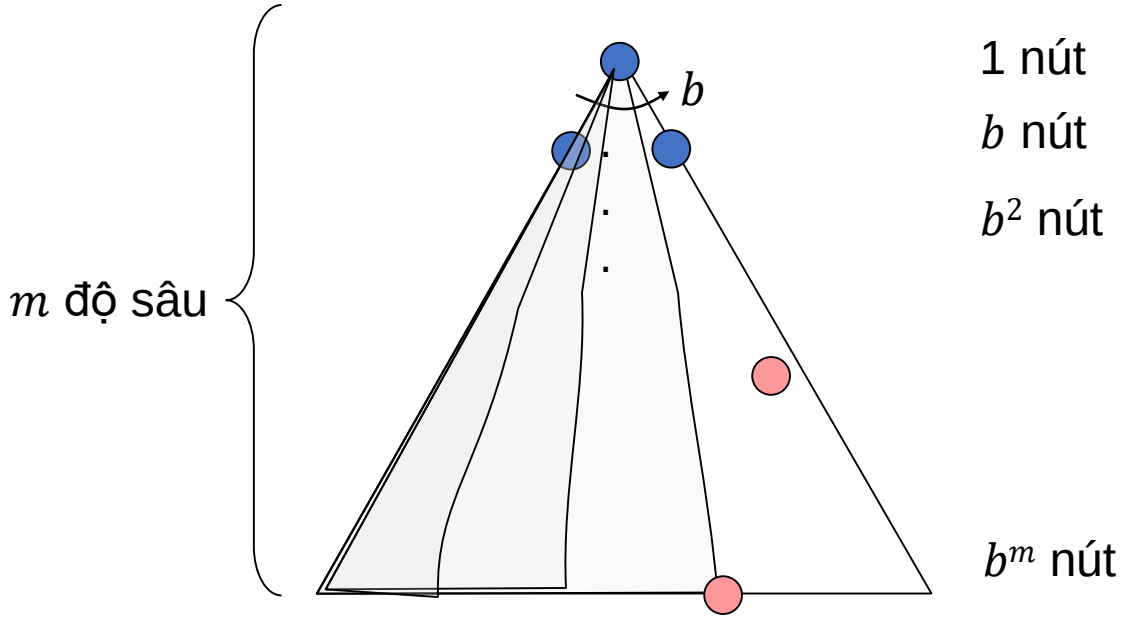
Thực hiện: frontier là một ngăn xếp LIFO

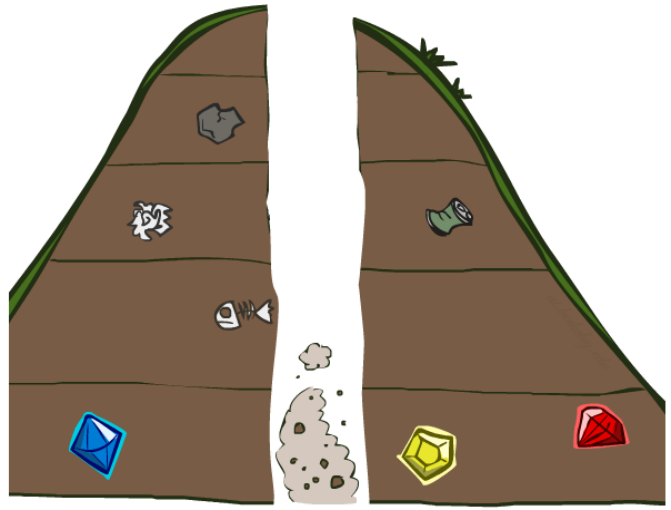




Đánh giá DFS

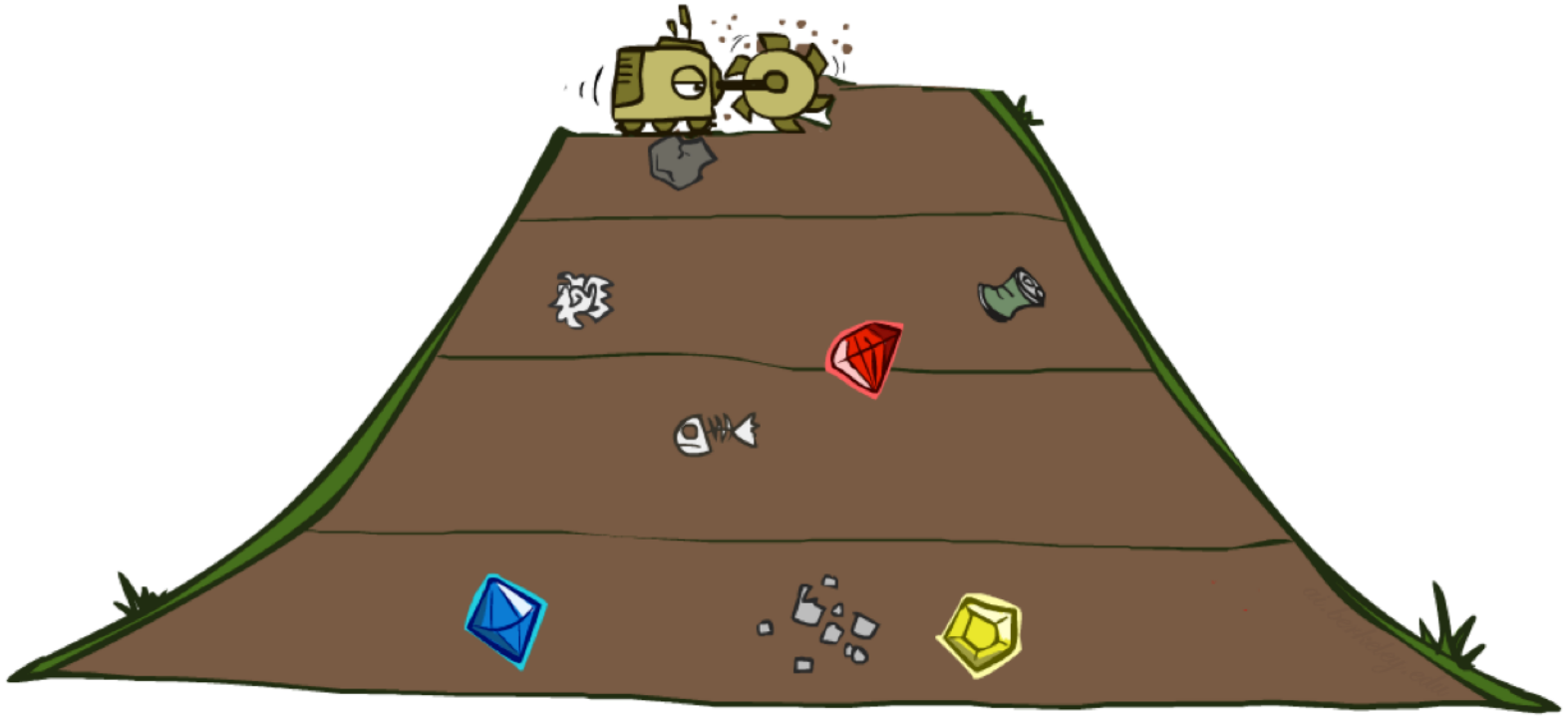
- DFS mở rộng nút nào?
  - Mở các nút phía bên trái.
  - Có thể xử lý toàn bộ cây!
  - Nếu m là hữu hạn, mất thời gian $O(bm)$
- Frontier cần không gian bao nhiêu?
  - Chỉ cần nút trên đường dẫn đến gốc, vì vậy $O(bm)$
- Hoàn chỉnh?
  - Không nếu như m là vô hạn (đồ thị có vòng lặp)
- Có tối ưu không?
  - Không, nó tìm thấy lời giải "bên trái" nhất, bất kể chiều sâu hoặc chi phí





Tìm kiếm theo chiều rộng (BFS)

Chiến lược chọn kế hoạch ở frontier để mở rộng: chọn kế hoạch "nông" nhất.
BFS duyệt cây theo hướng từ trên xuống dưới.
Có thể dùng queue để cài đặt frontier



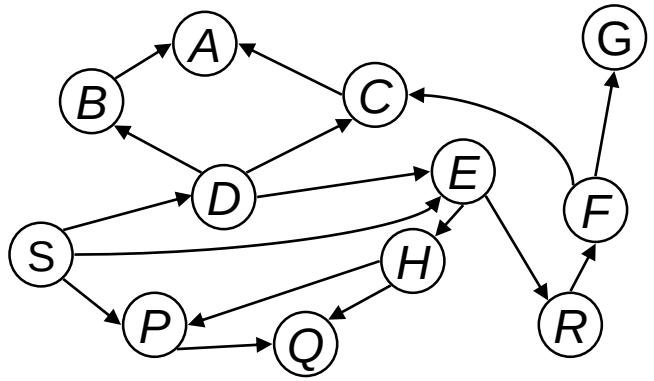
Thuật toán BFS

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
    if problem's initial state is a goal then return empty path to initial state
    frontier ← a FIFO queue initially containing one path, for the problem's initial state
    reached ← a set of states; initially empty
    solution ← failure
    while frontier is not empty do
        parent ← the first node in frontier
        for child in successors(parent) do
            s ← child.state
            if s is a goal then
                return child
            if s is not in reached then
                add s to reached
                add child to the end of frontier
    return solution
```

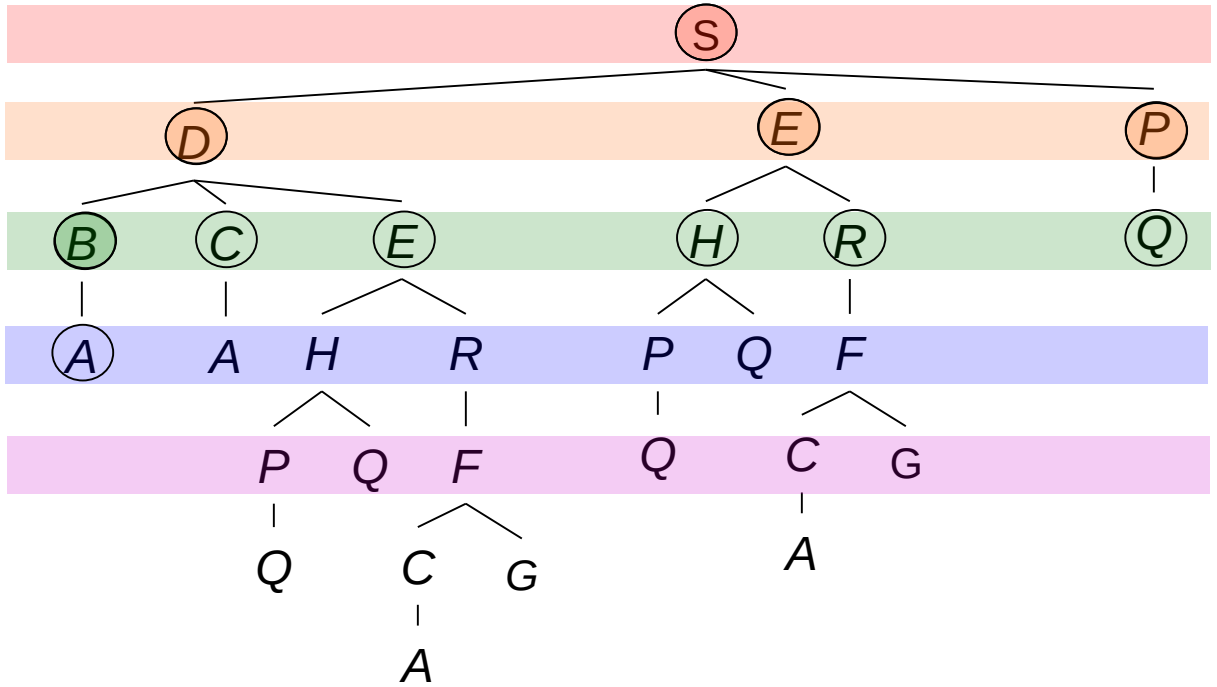
Minh họa BFS

Chiến lược: mở rộng nút nông nhất đầu tiên

Thực hiện: frontier là một hàng đợi FIFO

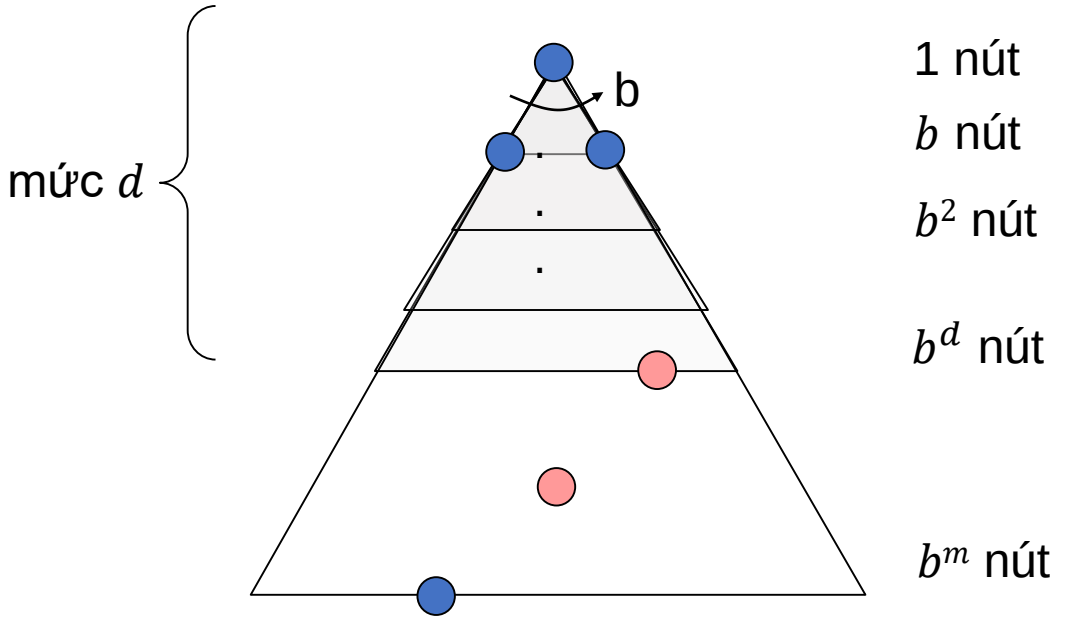


Mức tìm kiếm

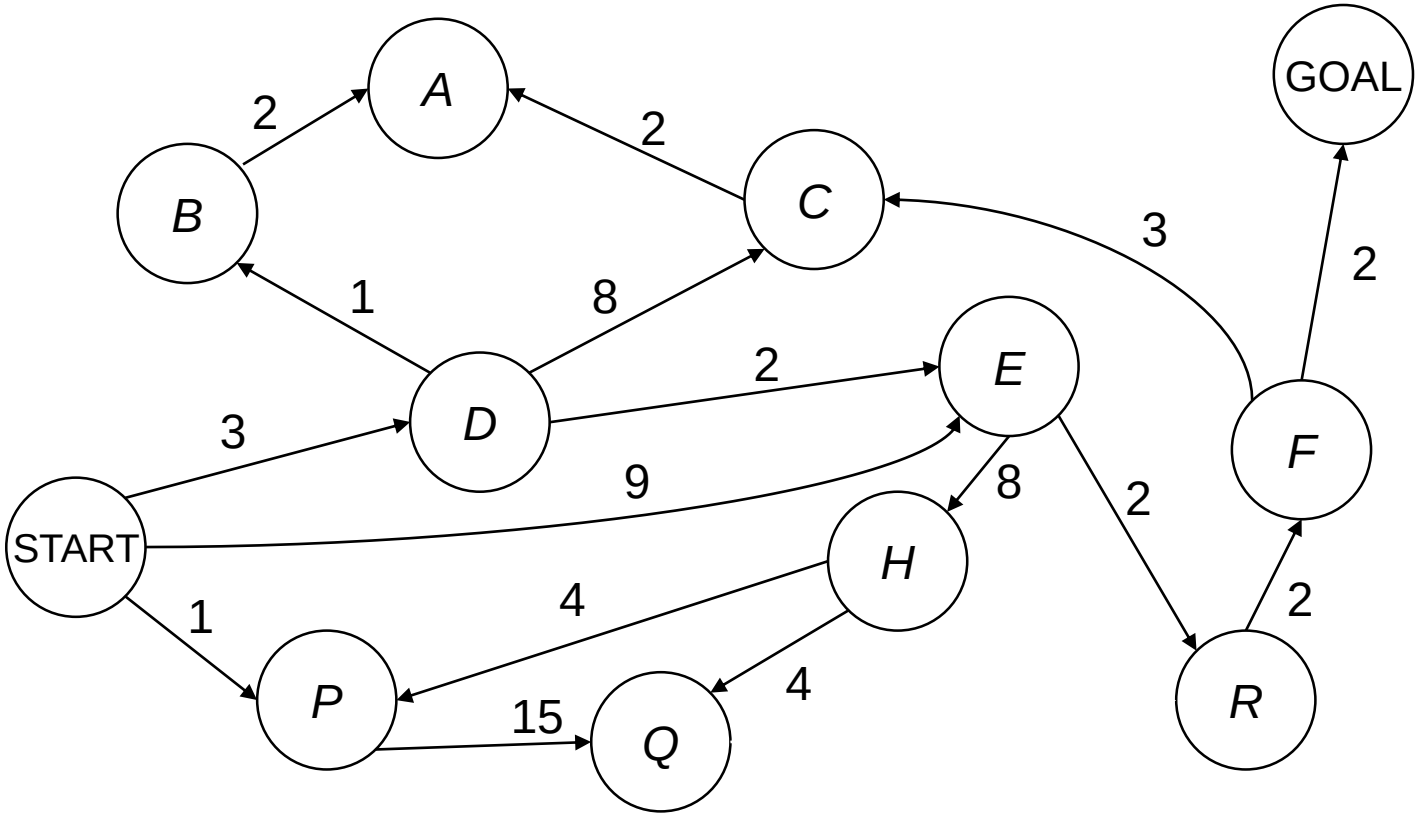


Đánh giá BFS

- BFS mở rộng những nút nào?
  - Xử tất cả các nút trên lời giải nông nhất
  - Độ sâu của giải pháp nhỏ nhất là d
  - Tìm kiếm mất thời gian $O(b^d)$
- frontier cần không gian bao nhiêu?
  - Khoảng tầng cuối, do đó, $O(b^d)$
- Hoàn chỉnh?
  - d phải là hữu hạn nếu tồn tại giải pháp
- Có tối ưu không?
  - Có - nếu chi phí là tất cả 1



Vấn đề của BFS



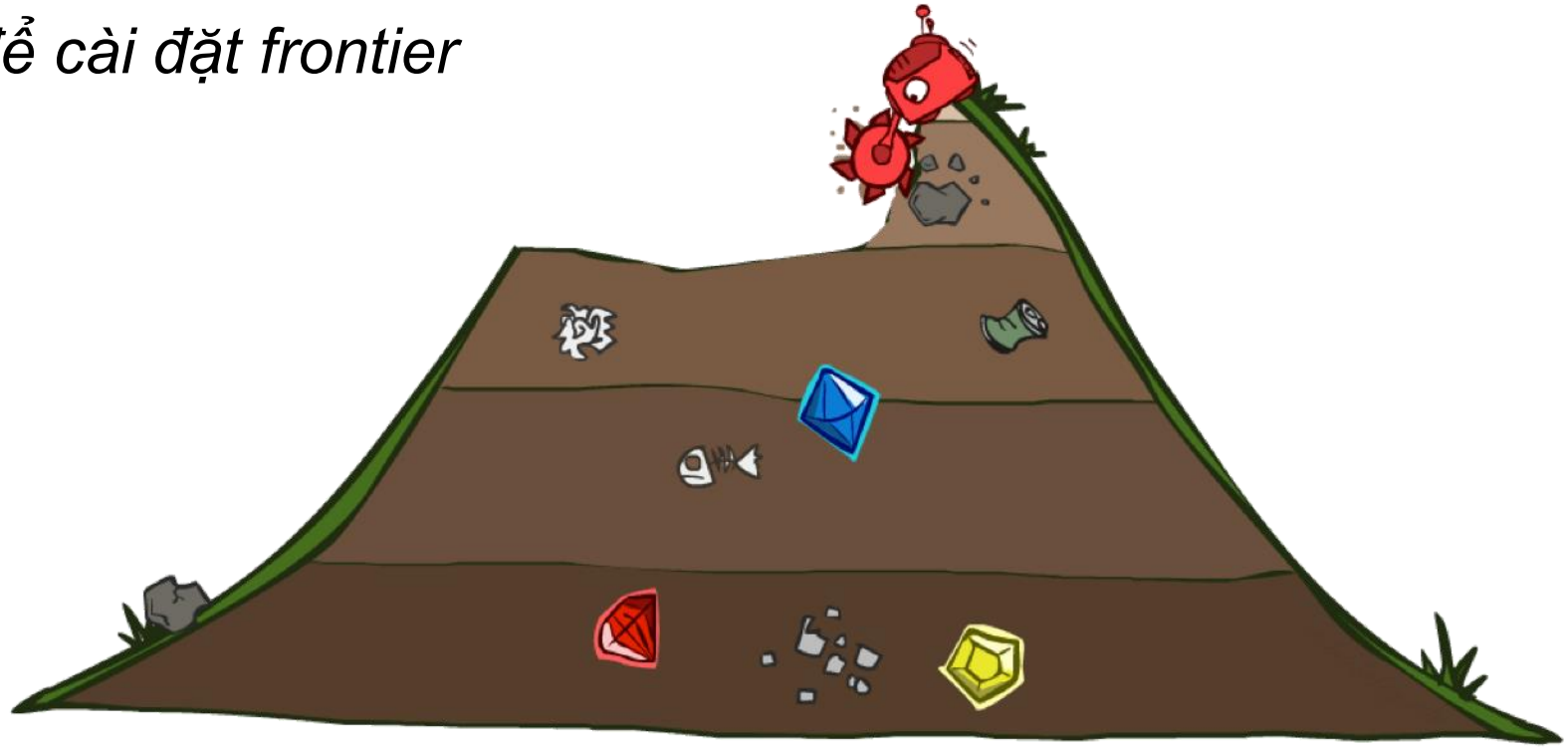
Thuật toán BFS tìm thấy đường đi với số lượng hành động ít nhất. Tuy nhiên, nó không tìm thấy đường đi với chi phí ít nhất.

Tìm kiếm với chi phí đồng nhất UCS

Chiến lược chọn kế hoạch ở frontier để mở rộng: chọn kế hoạch "ít chi phí" nhất.

UCS duyệt cây tương tự như BFS, nhưng mở rộng theo chi phí tăng dần chứ ko phải chiều sâu tăng dần

Có thể dùng priority queue để cài đặt frontier



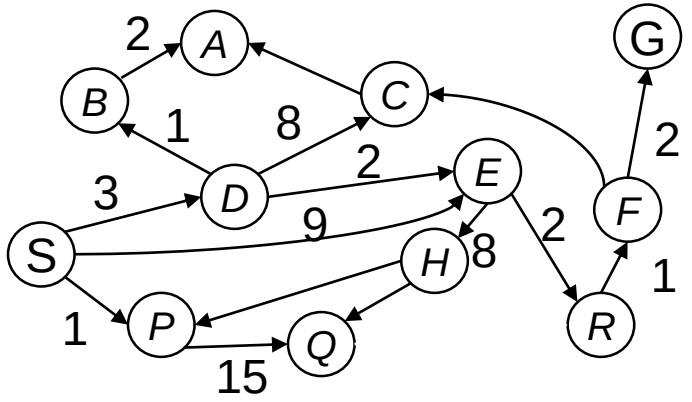
Thuật toán UCS

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
    if problem's initial state is a goal then return empty path to initial state
    frontier ← a priority queue ordered by pathCost, with a node for the initial state
    reached ← a table of {state: the best path that reached state}; initially empty
    solution ← failure
    while frontier is not empty and top(frontier) is cheaper than solution do
        parent ← pop(frontier)
        for child in successors(parent) do
            s ← child.state
            if s is not in reached or child is a cheaper path than reached[s] then
                reached[s] ← child
                add child to the frontier
                if child is a goal and is cheaper than solution then
                    solution = child
    return solution
```

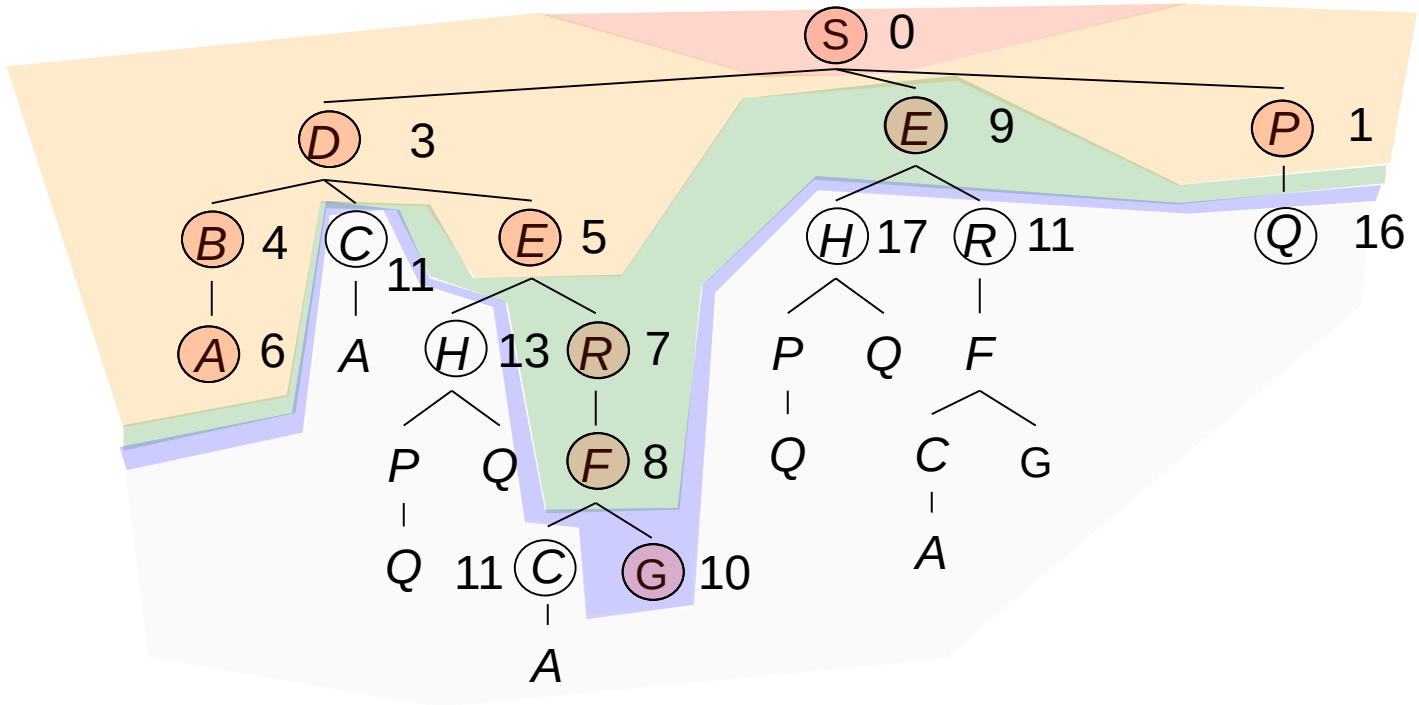
Minh họa UCS

Chiến lược: mở rộng một nút có pathCost nhỏ nhất

Thực hiện: frontier là hàng đợi ưu tiên theo pathCost

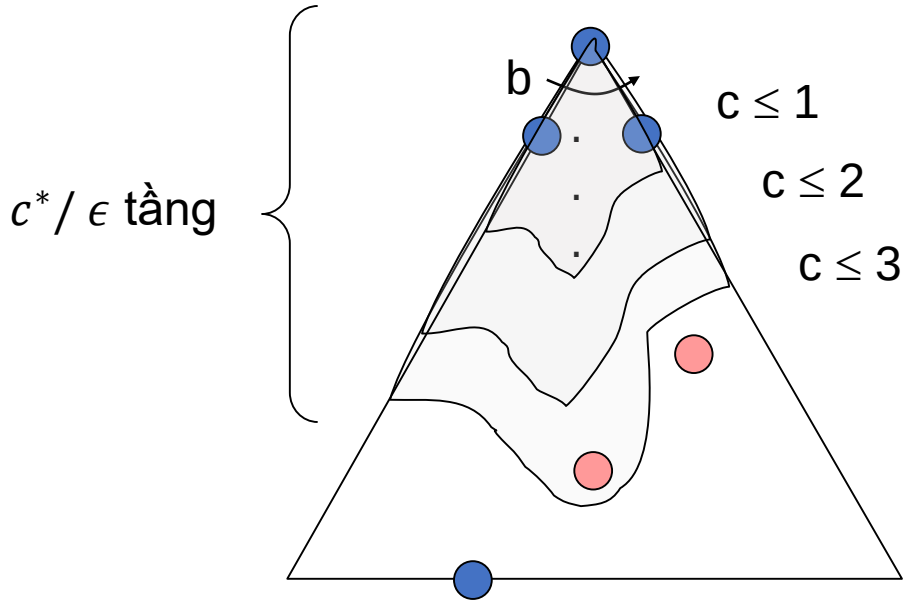


contour chi phí



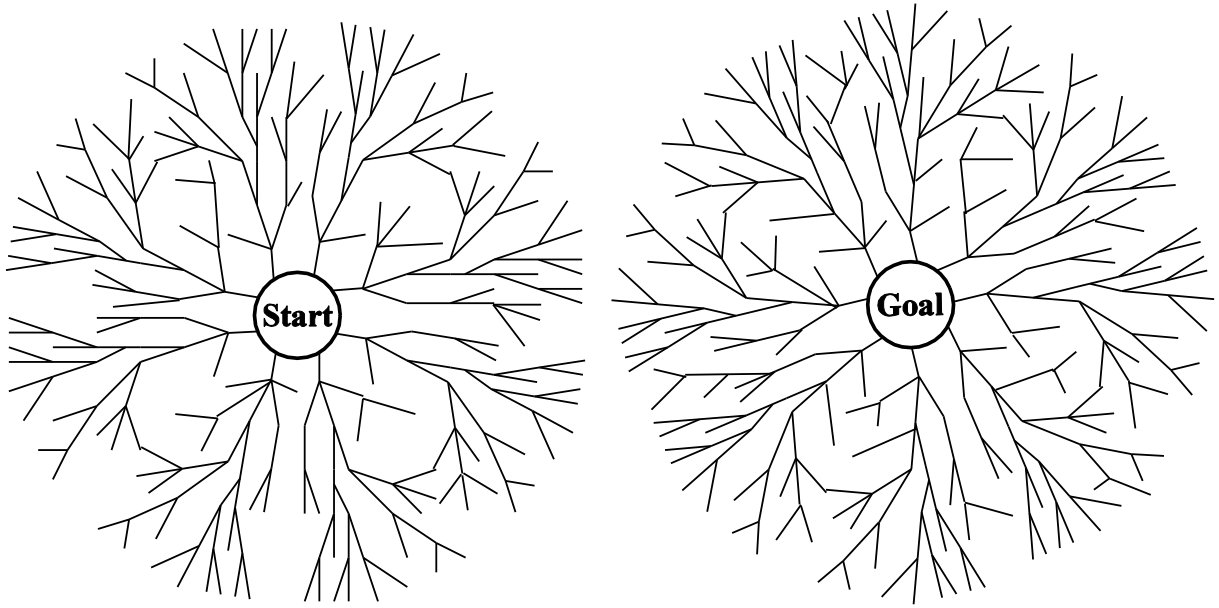
Đánh giá UCS

- UCS mở rộng nút nào?
  - Xử tất cả các nút với chi phí ít hơn K!
  - Nếu giải pháp đó chi phí c và có chi phí ít nhất ϵ, thì "độ sâu hiệu quả" là khoảng c^*/ϵ
  - Mất thời gian $O(b^{c^*/\epsilon})$ (hàm mũ ở độ sâu hiệu quả)
- Frontier cần không gian bao nhiêu?
  - Khoảng tầng cuối, do đó, $O(b^{c^*/\epsilon})$
- UCS đảm bảo tìm được lời giải (nếu tồn tại), và lời giải tìm được là tối ưu.



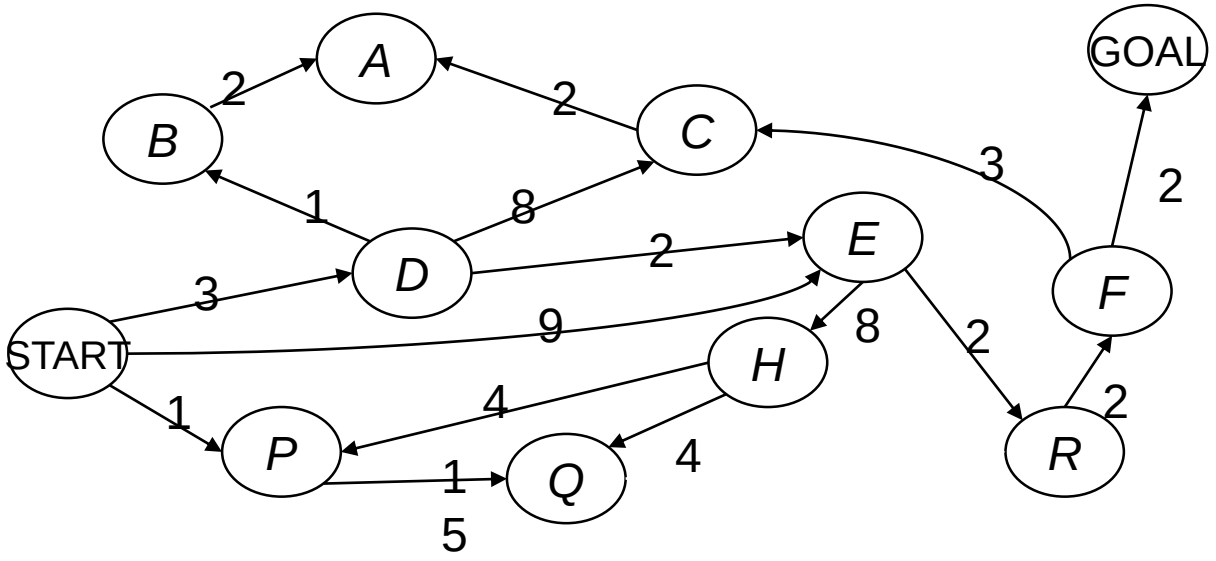
Mở rộng: Tìm kiếm hai chiều (BiS)

Chạy đồng thời **hai lượt tìm kiếm**, một lượt đi tới từ TT bắt đầu (Start) và một lượt đi lui từ TT đích (Goal), dừng khi cùng gặp nhau tại một nút.



Bài tập

1. Chạy DFS, BFS, UCS với đồ thị bên dưới



2. So sánh các thuật toán DFS, BFS và UCS