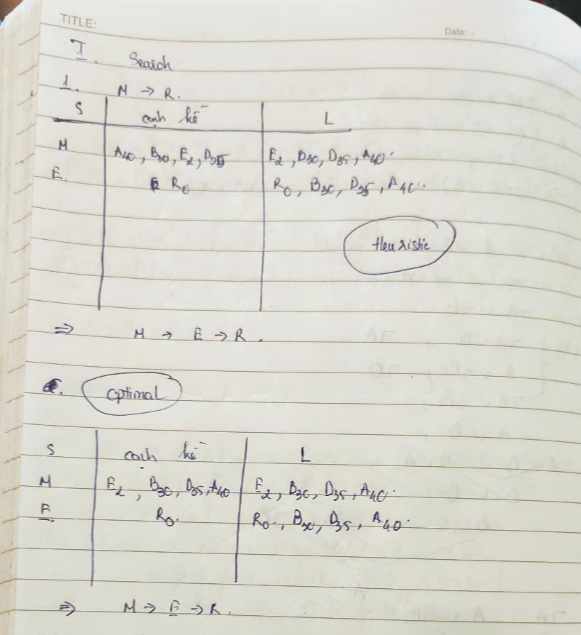
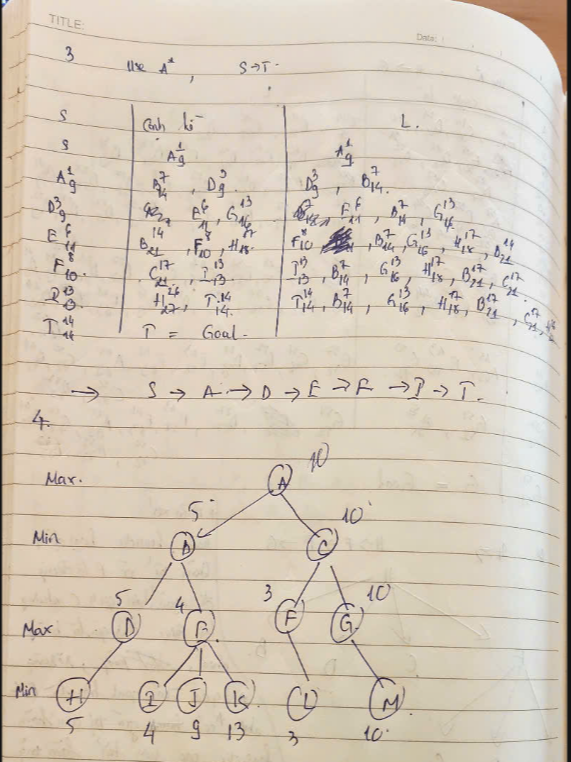
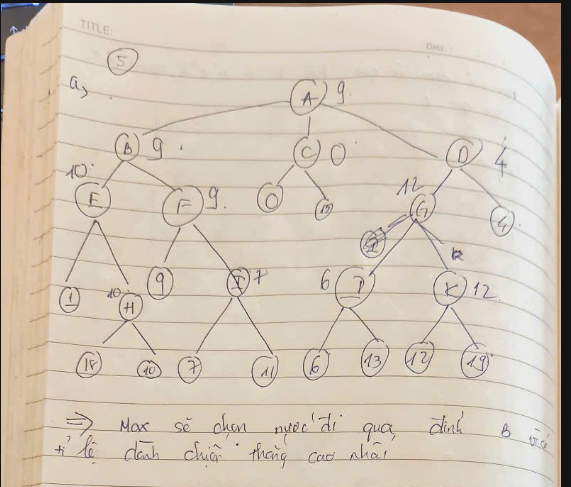
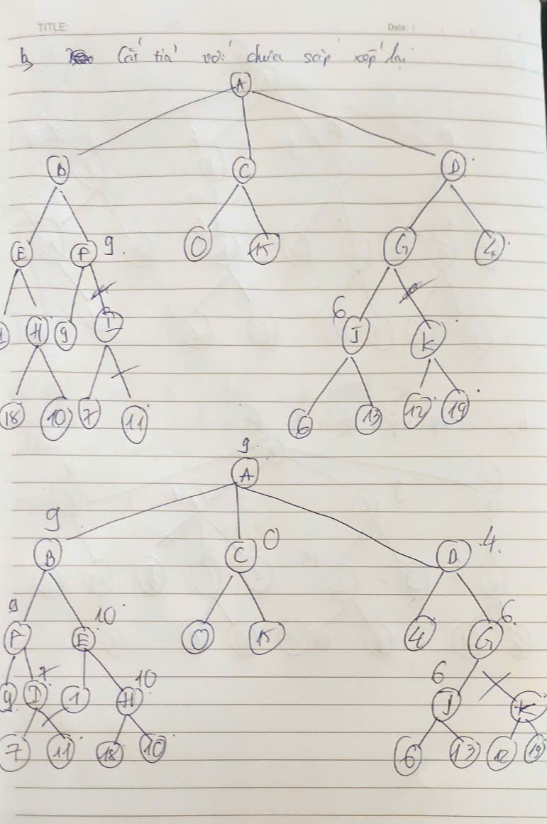
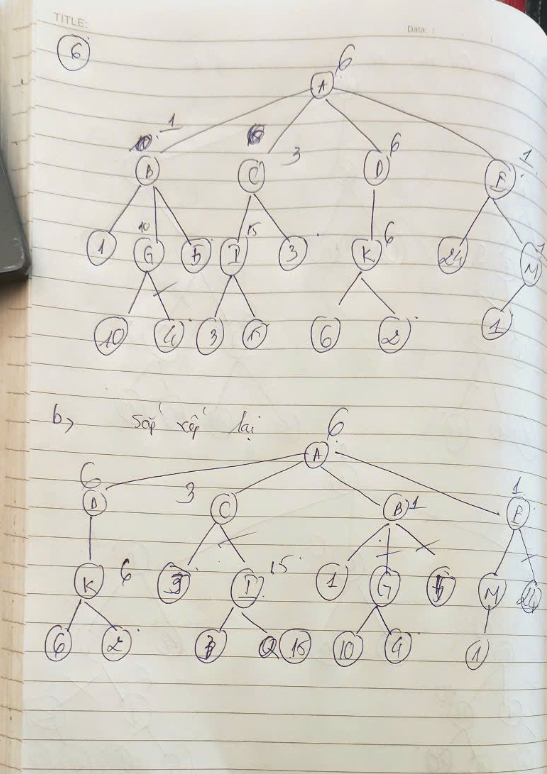
# Lý thuyết

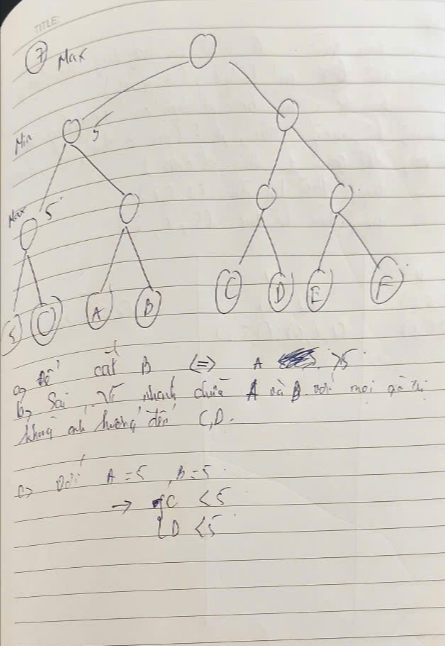












Câu 8

- Không gian trạng thái:  
 + Trạng thái: Vị trí của Pacman trên bản đồ

+ Toán tử: Dịch chuyển của pacman theo chiều lên, xuống, sang trái , sang phải

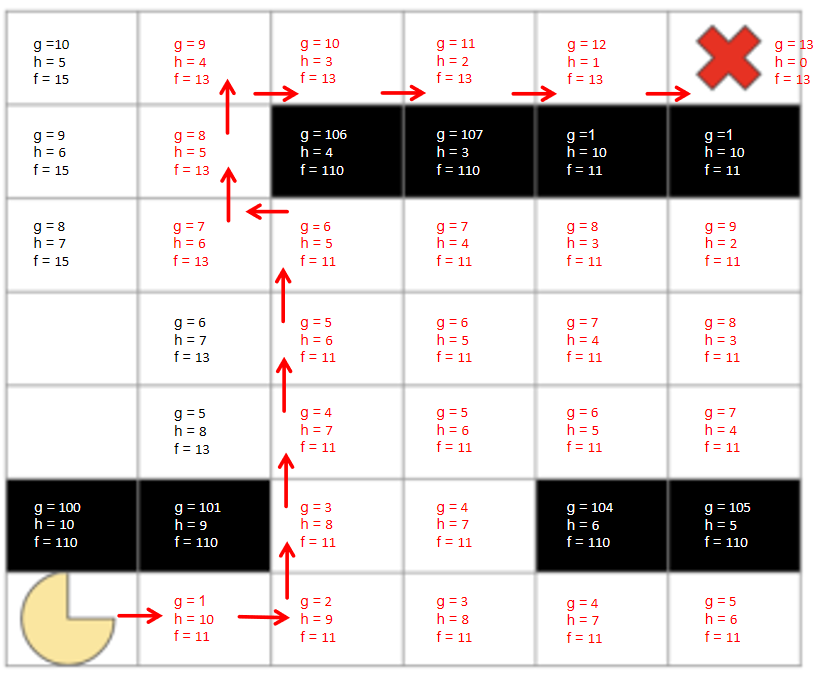
+ Vị trí bắt đầu: Tọa độ (0, 0)

+ Vị trí kết thúc: Tọa độ (5, 6)

+ Hàm heuristic h(n) là khoảng cách Manhattan.

+ Hàm g(n): mỗi lần di chuyển vào ô trắng là 1, mỗi bước di chuyển vào ô đen là 10.

+ Hàm đánh giá: f(n) = g(n) + h(n)



Câu 9:

Không gian trạng thái:

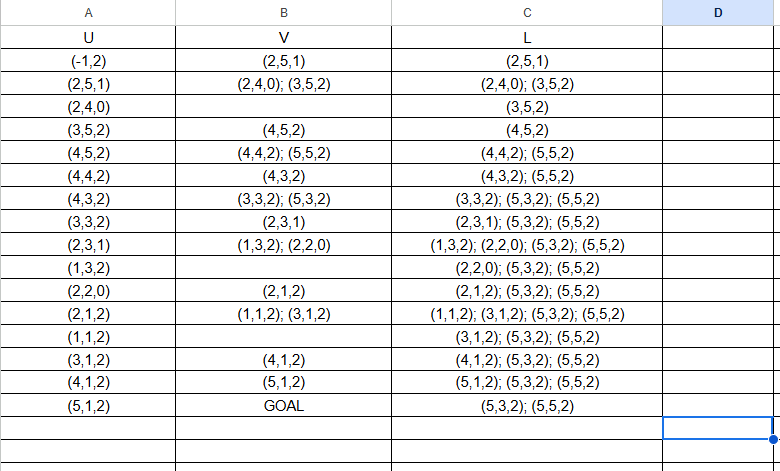
- Trạng thái: là vị trí của Ladybug và lượng không khí của Ladybug (x, y, a) (với a phải lớn hơn hoặc bằng 0)

- Toán tử: mỗi bước di chuyển của Lady bug theo 4 hướng lên, xuống, trái, phải. Với mỗi bước di chuyển vào ô xám thì lượng khí giảm đi 1, ở ô trắng thì lượng khí luôn là 2. Không đi vào ô đen.

- Trạng thái bắt đầu: (1, 5, 2)

- Trạng thái kết thúc: (5, 1, 2)

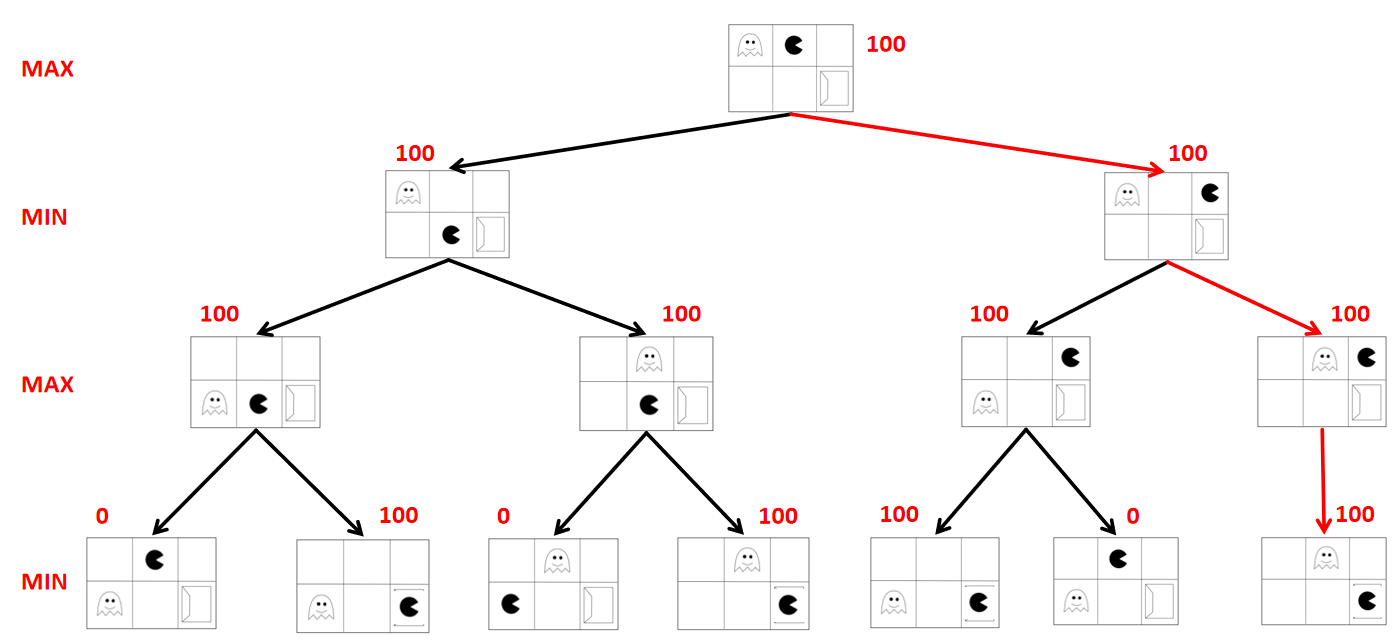
Sử dụng DFS (tìm kiếm theo chiều sâu)



## Bài toán Pacman

Không gian trạng thái:

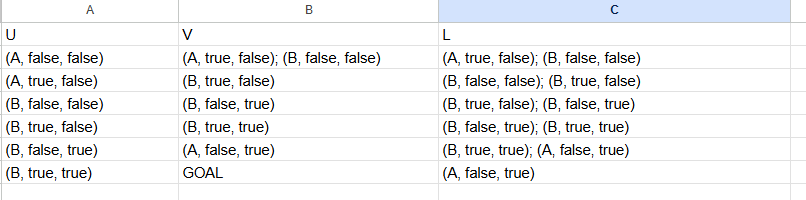
* Trạng thái: Một trạng thái được định nghĩa bởi vị trí hiện tại của Pacman (xP,yP), vị trí của Ghost (xG,yG), và vị trí của Đích (xD,yD) trên bản đồ.
* Toán tử: Các hành động có thể thực hiện bao gồm 4 hướng di chuyển cơ bản (lên, xuống, trái, phải) cho cả Pacman và Ghost trong mỗi bước.
* Trạng thái bắt đầu: Pacman ở (2,1), Ghost ở (1,1), và Đích ở (3,2).
* Trạng thái kết thúc:
  + Pacman thắng: Pacman đến được vị trí của Đích (xP = xDvà yP=yD). Điểm thưởng: f(n)=100.
  + Ghost thắng: Ghost bắt được Pacman (xP=xG và yP=yG). Điểm phạt: f(n)= −100.
* Hàm h(n)=−(∣xP−xD∣+∣yP−yD∣)+(∣xG−xP∣+∣yG−yP∣). Hàm này ưu tiên các trạng thái mà Pacman gần Đích hơn và Ghost xa Pacman hơn.



## 11.

Không gian trạng thái:

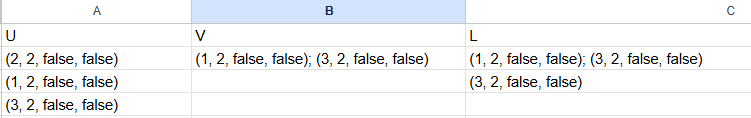
* Trạng thái: Một trạng thái bao gồm vị trí hiện tại của robot (có thể ở vị trí A hoặc B) và trạng thái sạch của hai vị trí: vị trí A (SA, true/false) và vị trí B (SB, true/false).
* Toán tử:
  + L : Di chuyển robot sang trái.
  + R : Di chuyển robot sang phải.
  + S : Hút bụi tại vị trí hiện tại của robot.
* Trạng thái bắt đầu: Robot ở vị trí A, cả hai vị trí A và B đều bẩn (false).
* Trạng thái kết thúc: Cả hai vị trí A và B đều sạch (true), bất kể vị trí cuối cùng của robot.
* Thuật toán tìm kiếm: Tìm kiếm theo chiều rộng (BFS).



## 12.

Không gian trạng thái:

* Trạng thái: Một trạng thái được xác định bởi tọa độ (xP,yP) của robot và trạng thái sạch của hai vị trí (SA, SB). SA và SB có thể là true (sạch) hoặc false (bẩn).
* Toán tử:
  + L : Di chuyển robot sang trái.
  + R : Di chuyển robot sang phải.
  + S : Hút bụi tại vị trí hiện tại.
* Trạng thái bắt đầu: Robot ở tọa độ (2,2), và cả hai vị trí đều bẩn (SA=false, SB=false).
* Trạng thái kết thúc: Bất kỳ trạng thái nào mà cả hai vị trí đều sạch (SA=true, SB=true), không quan tâm đến vị trí cuối cùng của robot.
* Thuật toán tìm kiếm: Tìm kiếm theo chiều rộng (BFS).
* Kết quả: Không tìm thấy đường đi (có thể do cấu trúc bản đồ hoặc ràng buộc).

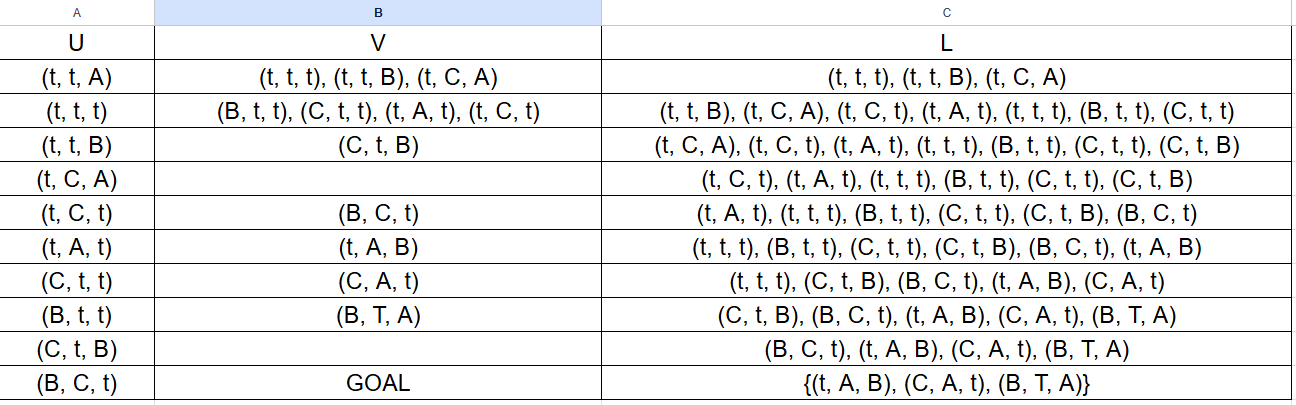


=>Không tìm ra đường đi.

13

Không gian trạng thái:

* Trạng thái: Một trạng thái được định nghĩa bằng vị trí của ba khối A, B, và C. Mỗi khối có thể nằm trên bàn (t) hoặc nằm trên một khối khác. Cụ thể:
  + OnA: Vị trí của khối A (t, B, C).
  + OnB: Vị trí của khối B (t, A, C).
  + OnC: Vị trí của khối C (t, A, B).
* Toán tử: Di chuyển một khối từ vị trí hiện tại của nó (trên bàn hoặc trên một khối khác) sang một vị trí mới. Các ràng buộc bao gồm:
  + Chỉ có thể di chuyển khối trên cùng của một chồng.
  + Một khối không thể nằm trên chính nó.
  + Nhiều hơn một khối không thể nằm trên cùng một vị trí trừ khi chúng tạo thành một chồng (ví dụ: A trên B, B trên bàn).
* Trạng thái bắt đầu: Khối A nằm trên bàn, khối B nằm trên bàn, và khối C nằm trên khối A. (t, t, A).
* Trạng thái kết thúc: Khối B nằm trên khối A, khối C nằm trên khối B, và khối C nằm trên bàn. (B, C, t).
* Thuật toán tìm kiếm: Tìm kiếm theo chiều rộng (BFS).



# THỰC HÀNH

1. **Cài đặt 4 thuật toán heuristic và optimal**
2. **Mô tả chương trình.**

- Dùng ngôn ngữ Java, xây dụng 2 class chính để lưu trữ đồ thị: Graph và Node và 1 class lưu trữ đường đi MyPath

- Class Node : hiện thực Interface Comparable để có thể so sánh giữa hai đối tượng node, lớp có 3 thuộc tính:

+ name: Tên đỉnh.

+ neighbor: danh sách đỉnh kề kèm theo trọng số, HashMap

+ heuristic: Giá trị hàm đánh giá.

- Class Graph: lớp có 2 thuộc tính:

+ nodes: Lưu tất cả các đỉnh trong đồ thị.

+ root: Lưu đỉnh gốc của đồ thị

- Menu chính:

+ Thêm một đỉnh

+ Xóa 1 đỉnh

+ Best first search

+ Hill Climming

+ A\*

+ Branch And Bound

+ In

1. **Mã giả thuật toán**

*- Best first search*

Function findByBFS(start, goal, L):

If start is null or goal is null then

Return (end procedure)

actualStart ← findNodeByName(start.getName())

actualGoal ← findNodeByName(goal.getName())

If actualStart is null or actualGoal is null then

Print "Node does not exist in the graph"

Return

If actualStart.getName() = actualGoal.getName() then

Print "Found vertex " + actualStart.getName()

Return

Print "Current vertex: " + actualStart.getName()

Print "Neighbor list of current vertex: "

Print actualStart's neighbors

If L is null then

workingL ← empty list

Else

workingL ← L

Print "List L: "

For each node in actualStart's neighbor list:

If node is not in workingL then

Add node to workingL

sortedL ← Sort workingL by heuristic value in ascending order

Print sortedL

If sortedL is not empty then

nextNode ← sortedL.removeFirst()

findByBFS(nextNode, actualGoal, sortedL)

- Hill climming

Function findByHillClimbing(start, goal, L):

If start is null or goal is null then

Return (end procedure)

// Find actual nodes in the graph

actualStart ← findNodeByName(start.getName())

actualGoal ← findNodeByName(goal.getName())

If actualStart is null or actualGoal is null then

Print "Node does not exist in the graph"

Return

If actualStart.getName() = actualGoal.getName() then

Print "Found vertex " + actualStart.getName()

Return

Print "Current vertex: " + actualStart.getName()

Print "Neighbor list of current vertex: "

Print actualStart's neighbors

If L is null then

workingL ← empty list

Else

workingL ← L

Print "List L: "

neiForStartSorted ← Get neighbors of actualStart

Filter out any nodes already in workingL

Sort by heuristic value in ascending order

concatList ← empty list

For each node in neiForStartSorted:

If node is not in workingL then

Add node to concatList

For each node in workingL:

If node is not in concatList then

Add node to concatList

Print concatList

If concatList is not empty then

nextNode ← concatList.removeFirst()

findByHillClimbing(nextNode, actualGoal, concatList)

*- A\**

Function findWithAStar(start, goal):

If start is null or goal is null then

Return null

// Find actual nodes in the graph

actualStart ← findNodeByName(start.getName())

actualGoal ← findNodeByName(goal.getName())

If actualStart is null or actualGoal is null then

Print "Node does not exist in the graph"

Return null

// Initialize data structures

gScore ← new HashMap<Node, Integer>() // Cost from start to current node

fScore ← new HashMap<Node, Integer>() // Estimated total cost (g + h)

// Define comparator for priority queue based on fScore

fScoreComparator ← new Comparator that compares nodes based on their fScore values

openSet ← new PriorityQueue<Node>(fScoreComparator) // Nodes to be evaluated

openSetContents ← new HashSet<Node>() // For O(1) lookup in openSet

closedSet ← new HashSet<Node>() // Already evaluated nodes

cameFrom ← new HashMap<Node, Node>() // Path reconstruction map

// Initialize scores for all nodes

For each node in nodes:

gScore[node] ← INFINITY

fScore[node] ← INFINITY

// Initialize start node

gScore[actualStart] ← 0

fScore[actualStart] ← actualStart.getHeuristic()

openSet.add(actualStart)

openSetContents.add(actualStart)

While openSet is not empty:

current ← openSet.poll() // Get node with lowest fScore

openSetContents.remove(current)

Print "Current vertex: " + current.getName() +

" (f = " + fScore[current] +

", g = " + gScore[current] +

", h = " + current.getHeuristic() + ")"

If current equals actualGoal then

Print "Found path to vertex " + actualGoal.getName()

Return reconstructPath(cameFrom, current)

closedSet.add(current)

Print "Neighbor list of current vertex: "

Print current's neighbors

For each neighbor and edgeCost in current's neighbor list:

If neighbor is in closedSet then

Continue to next neighbor

tentativeGScore ← gScore[current] + edgeCost

If tentativeGScore < gScore[neighbor] then

// This path to neighbor is better than any previous one

cameFrom[neighbor] ← current

gScore[neighbor] ← tentativeGScore

fScore[neighbor] ← tentativeGScore + neighbor.getHeuristic()

If neighbor is not in openSetContents then

openSet.add(neighbor)

openSetContents.add(neighbor)

Print "Added " + neighbor.getName() +

" to openSet (f = " + fScore[neighbor] +

", g = " + gScore[neighbor] +

", h = " + neighbor.getHeuristic() + ")"

Else

// Update position in priority queue

openSet.remove(neighbor)

openSet.add(neighbor)

Print "Updated " + neighbor.getName() +

" in openSet (f = " + fScore[neighbor] +

", g = " + gScore[neighbor] +

", h = " + neighbor.getHeuristic() + ")"

Print "Current openSet: " + openSetToString(openSet)

Print "Current closedSet: " + closedSetToString(closedSet)

Print "No path found to vertex " + actualGoal.getName()

Return null

*- Branch and Bound Algorithm*

Function findWithBranchAndBound(start, goal):

If start is null or goal is null then

Return null

// Find actual nodes in the graph

actualStart ← findNodeByName(start.getName())

actualGoal ← findNodeByName(goal.getName())

If actualStart is null or actualGoal is null then

Print "Node does not exist in the graph"

Return null

// Initialize data structures

queue ← new PriorityQueue<MyPath>(sorted by path cost)

visited ← new HashSet<Node>()

bestCost ← INFINITY

bestPath ← null

// Create initial path with just the start node

initialPath ← new MyPath()

initialPath.addNode(actualStart, 0)

queue.add(initialPath)

Print "Starting Branch and Bound algorithm from node " + actualStart.getName()

While queue is not empty:

currentPath ← queue.poll()

currentNode ← currentPath.getLastNode()

Print "Examining node: " + currentNode.getName() +

" (current cost: " + currentPath.getCost() + ")"

// Pruning: Skip paths that exceed best cost found so far

If currentPath.getCost() >= bestCost then

Continue to next path

// Check if we've reached the goal

If currentNode equals actualGoal then

If currentPath.getCost() < bestCost then

bestCost ← currentPath.getCost()

bestPath ← currentPath.getNodes()

Print "Found better path to " + actualGoal.getName() +

" with cost " + bestCost

Continue to next path

visited.add(currentNode)

Print "Neighbor list of node " + currentNode.getName() + ":"

Print currentNode's neighbors

// Explore all neighbors

For each neighbor and edgeCost in currentNode's neighbor list:

// Skip if neighbor is already in the path (avoid cycles)

If currentPath contains neighbor then

Continue to next neighbor

// Create new path by extending current path

newPath ← new MyPath(currentPath) // Clone the current path

newPath.addNode(neighbor, edgeCost)

// Only consider path if it has potential to be better than best found

If newPath.getCost() < bestCost then

queue.add(newPath)

Print "Added new path through node " + neighbor.getName() +

" with total cost " + newPath.getCost()

If bestPath is not null then

Print "Best path: " + pathToString(bestPath) +

" with cost " + bestCost

Return bestPath

Else

Print "No path found to " + actualGoal.getName()

Return null

**2. THUẬT TOÁN MINIMAX VỚI CẮT TỈA ALPHA-BETA**

Thuật toán Minimax là một thuật toán tìm kiếm đệ quy được sử dụng trong lý thuyết trò chơi để tìm nước đi tối ưu cho người chơi trong trò chơi có lượt. Thuật toán này giả định rằng đối thủ của bạn chơi tối ưu.

Alpha-Beta là kỹ thuật cắt tỉa nhằm giảm số lượng nút cần đánh giá trong cây tìm kiếm Minimax, giúp tăng hiệu suất mà không làm mất đi khả năng tìm ra nước đi tối ưu.

=> Dùng thuật toán để xây dựng trog chơi Caro