ĐẠI HỌC ĐÀ NẴNG

TRƯỜNG ĐẠI HỌC BÁCH KHOA

TRÍ TUỆ NHÂN TẠO

Khoa Công Nghệ Thông Tin

TS. Nguyễn Văn Hiệu

Khoa Công Nghệ Thông Tin
TS. Nguyễn Văn Hiệu

TRÍ TUỆ NHÂN TẠO

Chương 3: Kỹ thuật tìm kiếm có sử dụng thông tin

Nội dung

- Giới thiệu
- Tìm kiếm tham lam tốt nhất
- Tìm kiếm A*
- Bài tập
- Tìm kiếm leo đồi

Giới thiệu

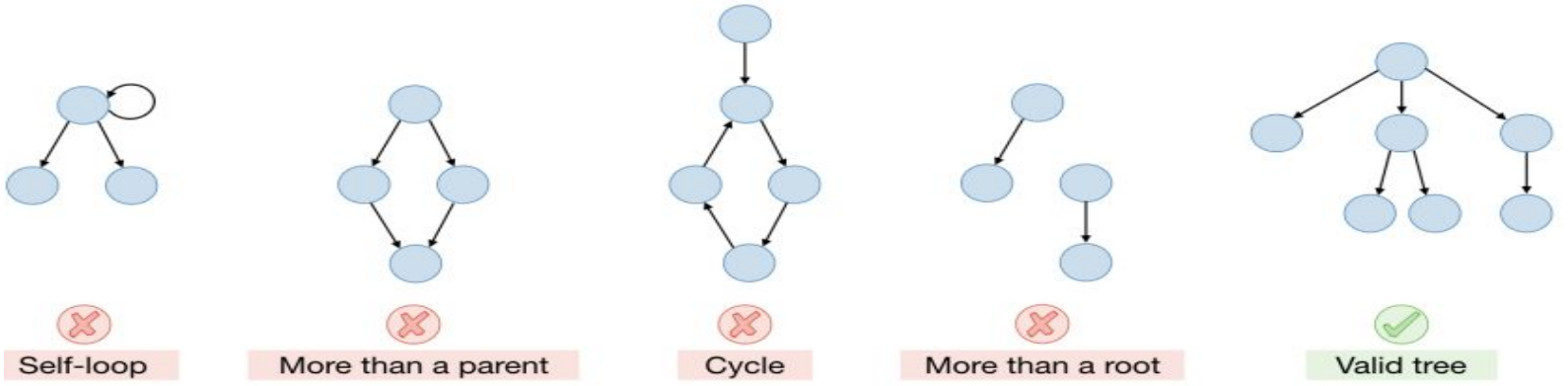
Tối ưu tìm kiếm (Search Optimization):

- Trạng thái **s**
- Hành động **a** từ trạng thái s dẫn đến trạng thái **Suss(s,a)**
- Mục tiêu xây dựng chuỗi hành động (a_1 ,a_2,a_n ,...,) từ trạng thái ban đầu đến trạng thái kết thúc.
- **Vấn đề**: Cần tìm đường đi với chi phí bé nhất

Giới thiệu

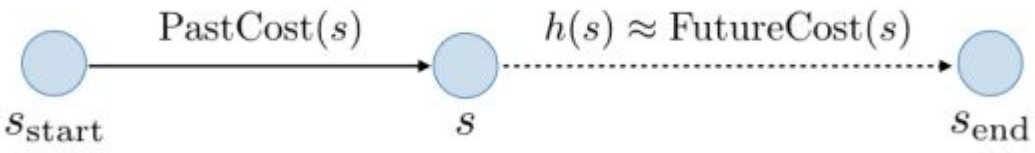
Tìm kiếm mù hay tìm kiếm không có thông tin

- Khám phá tất cả các trạng thái và hành động có thể có (trên xuống, trái - phải, …)
- Kỹ thuật tìm với không gian bộ nhớ phù hợp (như DFS **O**(b.m))
- Hạn chế về thời gian thực thi thường khai triển theo cấp số nhân (như BFS và DFD **O**(b^d)).



Giới thiệu

- Kỹ thuật tìm kiếm không có thông tin
  - Kỹ thuật BFS và DFS kém hiệu quả.
- Kỹ thuật tìm kiếm có thông tin
  - Sử dụng ngữ cảnh, tri thức và hiểu biết để tìm kiếm hiệu quả
  - Tìm trạng thái đang hướng tới trạng thái đích và tìm hàm đo khoảng cách đến trạng thái đích.
- Hàm kinh nghiệm (Heuristic function)
  - Hàm h ước lượng tại trạng thái s: h(s) = FutureCost(s)



Giới thiệu

- Kỹ thuật tìm kiếm có thông tin
  - Xây dựng hàm đánh giá có vai trò then chốt.
  - Hiệu quả của thuật toán phụ thuộc vào hàm đánh giá.
- Hàm đánh giá trong bài toán tìm đường đi:
  - Hàm đánh giá là đường chim bay.
  - Hàm đánh giá là khoảng cách thực giữa hai địa điểm.
  - Hàm đánh giá là khoảng cách thực và trọng số bổ sung trên đường đi.
- **Tóm lại: Hàm đánh giá tùy thuộc vào vấn đề cần giải.**

Giới thiệu

- Kỹ thuật tìm kiếm có thông tin được dạy
  - **Tìm kiếm ăn tham tốt nhất đầu tiên (Greedy best-first search).**
  - **Tìm kiếm A* (A star search).**
  - Thuật toán leo đồi (Hill-climbing search).

Tìm kiếm tham lam tốt nhất đầu tiên (Greedy best-first search)

Tìm kiếm tham lam tốt nhất đầu tiên

- Ý tưởng:
  - sử dụng hàm đánh giá.
  - sử dụng tìm kiếm theo chiều rộng.
- Hàm đánh giá **h(u)**
  - Hàm ước lượng đến trạng thái kết thúc.
- Tìm kiếm tham lam tốt nhất đầu tiên
  - Các trạng thái được phát dựa vào hàm đánh giá.
  - Các trạng thái không được phát sinh lần lượt theo BFS.
- Hướng cài đặt
  - Dùng hàng đợi có sự ưu tiên dựa vào hàm đánh giá.

Tìm kiếm tham lam tốt nhất đầu tiên

- Ý tưởng: **h(u) và Queue**

```
function GREEDY-BEST-FIRST-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :  /* Cost f(n) = h(n) */

    frontier = Heap.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.deleteMin()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.insert(neighbor)
            else if neighbor in frontier:
                frontier.decreaseKey(neighbor)

    return FAILURE
```

khoảng cách ước lượng đến đích

Cập nhật

Tìm kiếm tham lam tốt nhất đầu tiên

- Quá trình biến đổi **frontier ?**

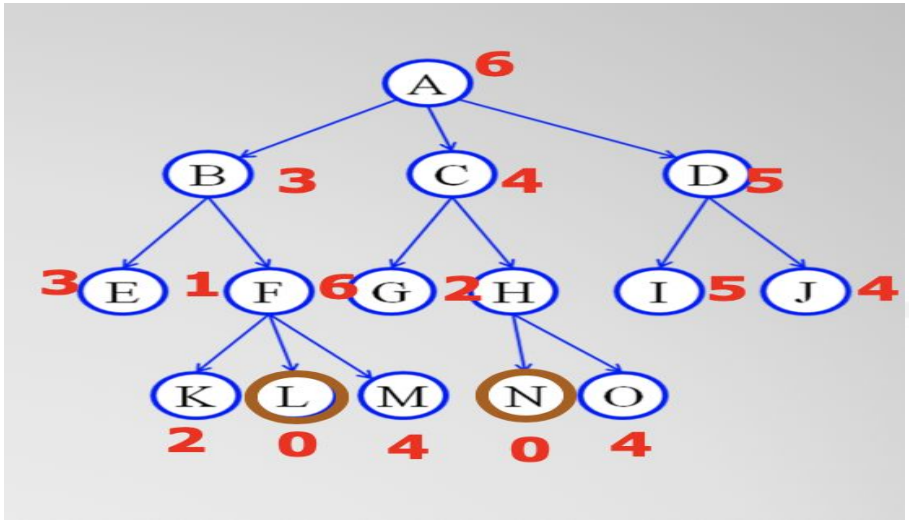
Frontier = {A(6)}

Frontier = {B(3), C(4),D(5)}

Frontier = {F(1),E(3), C(4), D(5)}

Frontier = {L(0),K(2),M(4),E(3), C(4), D(5)}}

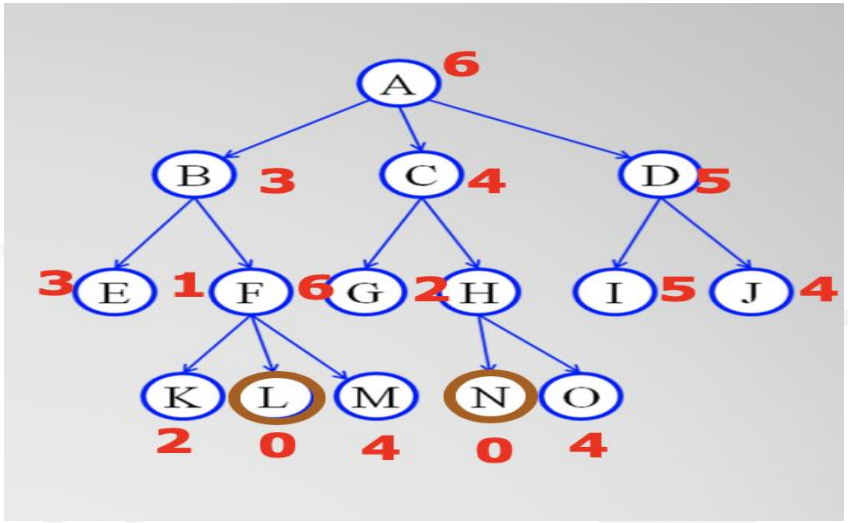
Frontier = ….




```
function GREEDY-BEST-FIRST-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :  /* Cost f(n) = h(n) */

    frontier = Heap.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.deleteMin()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.insert(neighbor)
            else if neighbor in frontier:
                frontier.decreaseKey(neighbor)

    return FAILURE
```

Demo(10 phút)

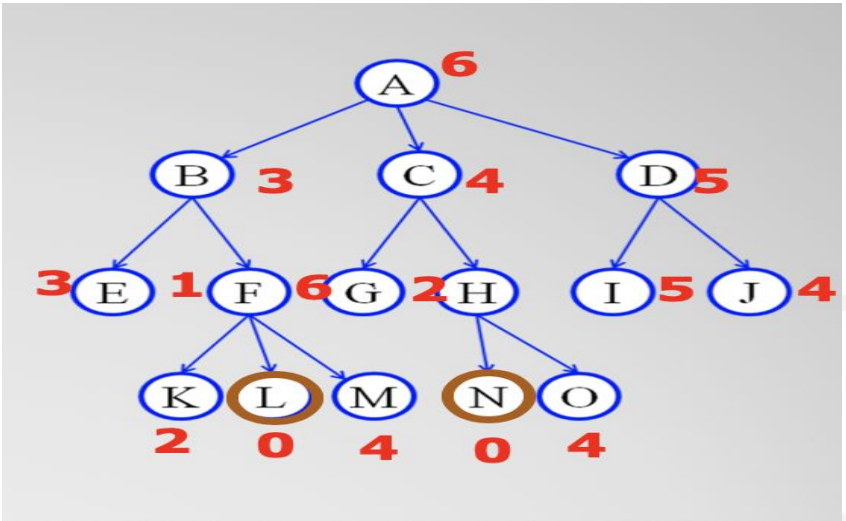
```python
import heapq
from TreeNode import Tree
def update_frontier(frontier, new_node):
    for idx, n in enumerate(array_of_node):
        if n == new_node:
            if frontier[idx].goal_cost > new_node.goal_cost:
                frontier[idx] = new_node
def GBF_search(initial_state, goalTest):
    frontier = []
    explored = []
    heapq.heapify(frontier)
    heapq.heappush(frontier, initial_state)
    while len(frontier) > 0:
        state = heapq.heappop(frontier)
        explored.append(state)
        if state == goalTest:
            return explored;
        for neighbor in state.get_children():
            if neighbor not in (frontier and explored):
                heapq.heappush(frontier, neighbor)
            elif neighbor in frontier:
                update_frontier(frontier=frontier, new_node=neighbor)
    return False
```




```
function GREEDY-BEST-FIRST-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :  /* Cost f(n) = h(n) */

    frontier = Heap.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.deleteMin()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.insert(neighbor)
            else if neighbor in frontier:
                frontier.decreaseKey(neighbor)

    return FAILURE
```

Demo(10 phút)

```
24  if __name__ == '__main__':
25      A = Tree("A",6)
26      B = Tree("B",3)
27      C = Tree("C",4)
28      D = Tree("D",5)
29      E = Tree("E",3)
30      F = Tree("F",1)
31      G = Tree("G",6)
32      H = Tree("H",2)
33      I = Tree("I",5)
34      J = Tree("J",4)
35      K = Tree("K",2)
36      L = Tree("L",0)
37      M = Tree("M",4)
38      N = Tree("N",0)
39      O = Tree("O",4)
40      A.add_child(B)
41      A.add_child(C)
42      A.add_child(D)
43      B.add_child(E)
44      B.add_child(F)
45      C.add_child(G)
46      C.add_child(H)
47      D.add_child(I)
48      D.add_child(J)
49      F.add_child(K)
50      F.add_child(L)
51      F.add_child(M)
52      H.add_child(N)
53      H.add_child(O)
54      result = GBF_search(A,L)
55      if result:
56          s = 'explored: '
57          for i in result:
58              s+=i.data + " "
59              print(s)
60      else:
61          print('404 not Found!')
```



Tìm kiếm A *
(A Star search)

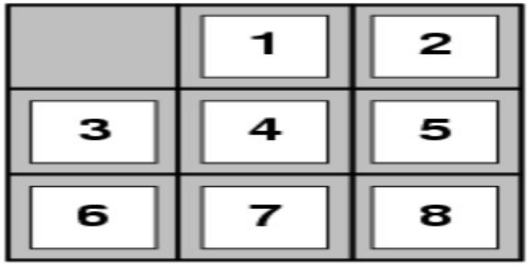
Tìm kiếm A *

- Tìm kiếm đường đi tốt nhất từ trạng thái đầu đến trạng thái đích
- Ý tưởng:
  - sử dụng hàm kinh nghiệm chấp nhận được
  - kỹ thuật tìm kiếm **BFS**
- Khám phá trạng thái s
  - g(u) - độ dài từ trạng thái đầu đến trạng thái u
  - h(u) - hàm heuristic
- Hàm h(u) chấp nhận được
  - **h(u) <= h*(u)**,
  - h*(u)- giá trị thực tế từ u đến trạng thái đích.
- Đề tăng hiệu quả tìm kiếm: **f(u) = g(u) + h(u)**
  - **f(u)** - lượng giá từ trạng thái đầu tới đích qua trạng thái u

Tìm kiếm A *

- Ví dụ hàm heuristic chấp nhận được của bài toán 8 số
  - $h_1(u)$ - số lượng ô bị sai.
  - $h_2(u)$- tổng số lượng ô từ ô hiện tại đến trạng thái đích

Start State

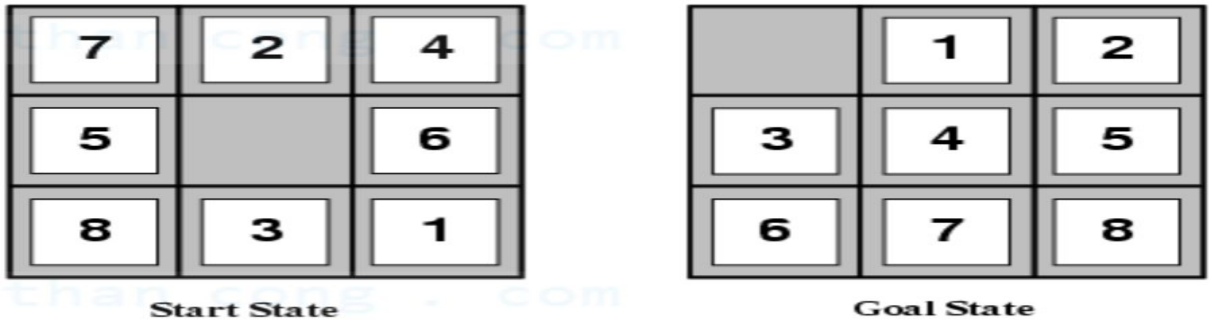


Goal State

* **$h_1(u)$ = ? $h_2(u)$ = ?**

Tìm kiếm A *

- Ví dụ hàm Heuristic chấp nhận được của bài toán 8 số



* $h_1(u) = 8$

* $h_2(u) = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18$

- Nếu $h_2(u) >= h_1(u)$ với mọi u.

  ⇒ $h_2(u)$ mạnh hơn $h_1(u)$ có nghĩa **$h_2(u)$ tốt hơn $h_1(u)$**

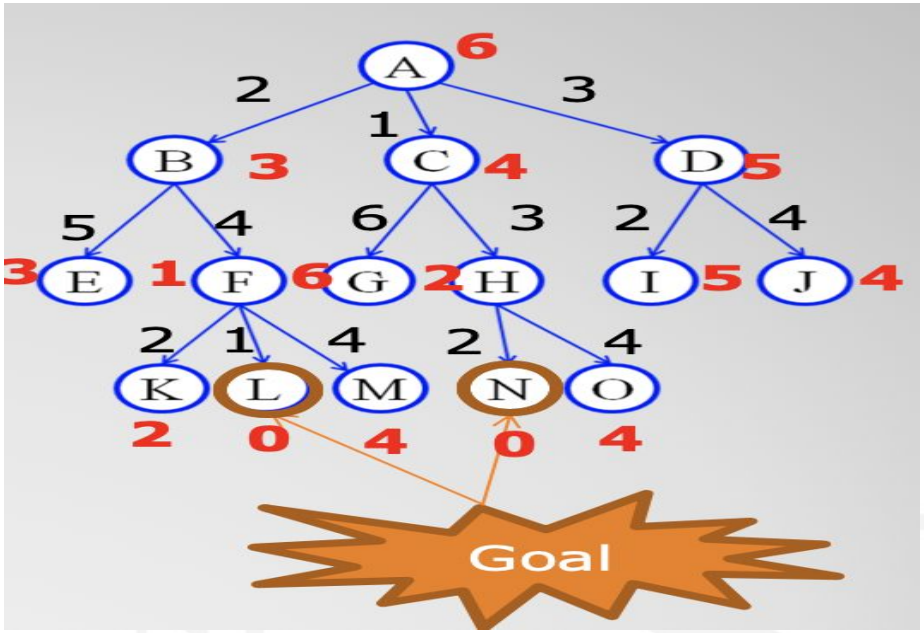
Tìm kiếm A *

- Kết hợp giữa hàm heuristic và BFS.

```
function A-STAR-SEARCH(initialState, goalTest)
      returns SUCCESS or FAILURE :  /* Cost f(n) = g(n) + h(n)  */

      frontier = Heap.new(initialState)
      explored = Set.new()

      while not frontier.isEmpty():
            state = frontier.deleteMin()
            explored.add(state)

            if goalTest(state):
                  return SUCCESS(state)

            for neighbor in state.neighbors():
                  if neighbor not in frontier ∪ explored:
                        frontier.insert(neighbor)
                  else if neighbor in frontier:
                        frontier.decreaseKey(neighbor)

      return FAILURE
```

Tìm kiếm A*

- Quá trình biến đổi **frontier ?**



frontier = {**A(0+6)**}

frontier = {**B(2+3)**, C(1+4),D(3+5)}

frontier = {E(7+3), F(6+1),**C(1+4)**,D(3+5)}

frontier = {E(10), F(7),G(7+6),**H(4+2)**,D(8)}

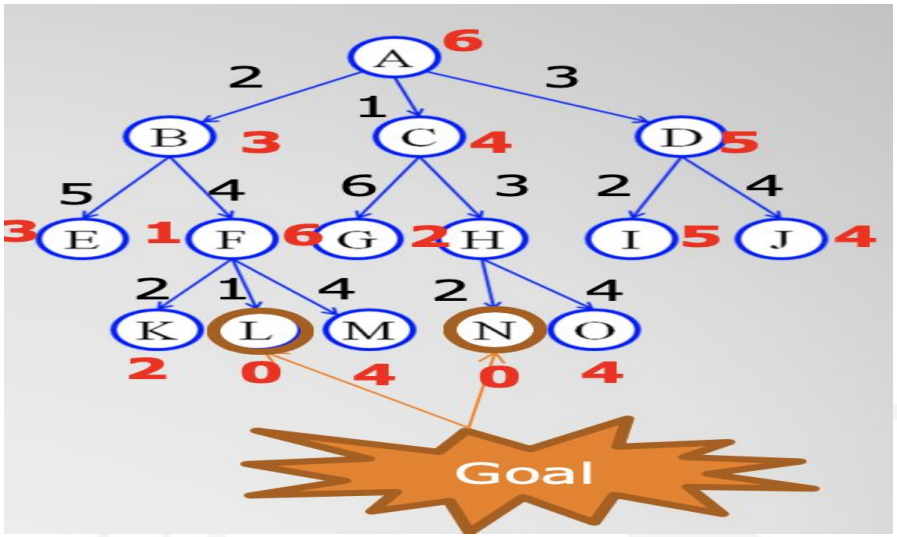
frontier = {E(10),
F(7),G(7+6),**N(6+0)**,O(8+4),D(8)}

frontier = …

```
function A-STAR-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :  /* Cost f(n) = g(n) + h(n) */

    frontier = Heap.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.deleteMin()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.insert(neighbor)
            else if neighbor in frontier:
                frontier.decreaseKey(neighbor)

    return FAILURE
```

Demo(15 phút)

```
90  if __name__ == "__main__":
91      tree = Tree()
92      tree.add_nodes([
93          Node("A", 6),
94          Node("B", 3),
95          Node("C", 4),
96          Node("D", 5),
97          Node("E", 3),
98          Node("F", 1),
99          Node("G", 6),
100         Node("H", 2),
101         Node("I", 5),
102         Node("J", 4),
103         Node("K", 2),
104         Node("L", 0),
105         Node("M", 4),
106         Node("N", 0),
107         Node("O", 4)
108     ])
109     tree.add_edges([
110         ("A", "B", 2),
111         ("A", "C", 1),
112         ("A", "D", 3),
113         ("B", "E", 5),
114         ("B", "F", 4),
115         ("C", "G", 6),
116         ("C", "H", 3),
117         ("D", "I", 2),
118         ("D", "J", 4),
119         ("F", "K", 2),
120         ("F", "L", 1),
121         ("F", "M", 4),
122         ("H", "N", 2),
123         ("H", "O", 4),
124     ])
125     tree.nodes[0].cost = 0
126     #print(tree.edges)
127     result = A_Star(tree, tree.nodes[0], tree.nodes[14])
128     if result:
129         s = 'explored: '
130         for i in result:
131             s += i.label + " "
132             print(s);
133     else:
134         print('404 Not Found!')
135
```




```
function A-STAR-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :  /* Cost f(n) = g(n) + h(n) */

    frontier = Heap.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.deleteMin()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.insert(neighbor)
            else if neighbor in frontier:
                frontier.decreaseKey(neighbor)

    return FAILURE
```

Demo(15 phút)

```
69  def update_cost(tree, current_node, prev_node):
70      if tree.get_edge(prev_node, current_node) is not None:
71          #print("OK")
72          if current_node.cost > prev_node.cost + tree.get_edge(prev_node, current_node)[2]:
73              current_node.cost = prev_node.cost + tree.get_edge(prev_node, current_node)[2]
74  def A_Star(tree, start, end):
75      frontier = [start]
76      heapq.heapify(frontier)
77      explored = []
78      while len(frontier) > 0:
79          state = heapq.heappop(frontier)
80          explored.append(state)
81          print(state)
82          if state == end:
83              return explored
84          for child in state.neighbors():
85              update_cost(tree, child, state)
86              if child.get_label() not in list(set(node.get_label() for node in frontier + explored)):
87                  heapq.heappush(frontier, child)
88      return False
```

Demo(15 phút)

```
2  import heapq
3  class Node:
4      def __init__(self, label, goal_cost):
5          self.label = label
6          self.cost = 10000
7          self.goal_cost = goal_cost
8          self.save_cost = None
9          self.pr = []
10         self.chld = []
11     def __repr__(self):
12         return str(dict({
13             "label": self.label,
14             "cost" : self.cost,
15             "goal cost": self.goal_cost
16         }))
17     def __eq__(self, other):
18         return self.label == other.label
19     def __lt__(self, other):
20         if self.save_cost == 10000:
21             return self.goal_cost + self.cost < other.goal_cost + other.cost
22         else:
23             return self.cost < other.cost
24     def get_label(self):
25         return self.label
26     def neighbors(self):
27         return self.chld + self.pr
28
```

Demo(15 phút)

```
29  class Tree:
30      def __init__(self):
31          self.nodes = []
32          self.edges = []
33      def add_nodes(self, data):
34          for node in data:
35              self.nodes.append(node)
36      def add_node(self, node):
37          self.nodes.append(node)
38
39      def get_index(self, node):
40          for i, n in enumerate(self.nodes):
41              if n.get_label() == node.get_label():
42                  return i
43          return -1
44      def add_edges(self, tuple_edges):
45          for t in tuple_edges:
46              start_label = t[0]
47              end_label = t[1]
48              w = t[2]
49              index_start_label = self.get_index(Node(start_label, None))
50              index_end_label = self.get_index(Node(end_label, None))
51
52              self.nodes[index_start_label].chld.append(self.nodes[index_end_label])
53              self.nodes[index_end_label].pr.append(self.nodes[index_start_label])
54              self.edges.append((self.nodes[index_start_label], self.nodes[index_end_label],t[2]))
55
56      def show_nodes(self):
57          return [node.get_data() for node in self.nodes]
58      def get_edge(self, start_node, end_node):
59          try:
60              return [edges for edges in self.edges if edges[0] == start_node
61                      and edges[1] == end_node][0]
62          except:
63              return None
```

Bài tập 1

- Xây dựng chương trình cài đặt kỹ thuật tìm kiếm tham lam tốt nhất đầu tiên và kỹ thuật tìm kiếm A*

-

Bài tập 2

- Mỗi sinh viên chọn 01 kỹ thuật tìm kiếm bên dưới để nghiên cứu để hiểu kỹ thuật tìm kiếm đã chọn:
  - Tìm kiếm nhánh và cận (Branch and Bound)
  - Tìm kiếm cục bộ (Local search algorithms)
  - Tìm kiếm mô phỏng luyện kim (Simulated annealing search)
  - Thuật toán gen (Genetic algorithms)

-