ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA CÔNG NGHỆ THÔNG TIN

Phạm Nhật Duy - 21120058

CƠ SỞ TRÍ TUỆ NHÂN TẠO

Đồ án 1: Tìm kiếm trong đồ thị

DFS, BFS, UCS, Dijkstra, A-Star

GIÁO VIÊN HƯỚNG DẪN

GS.TS. Lê Hoài Bắc

CN. Nguyễn Bảo Long

Tp. Hồ Chí Minh, tháng 10/2003

Lời cảm ơn

Báo cáo Đồ án – Các thuật toán tìm kiếm trên đồ thị là kết quả của quá trình cố gắng không ngừng của bản thân và được sự giúp đỡ của các thầy cô, bạn bè. Qua trang viết này tác giả xin gửi lời cảm ơn tới những người đã giúp đỡ cả nhóm trong thời gian học tập - nghiên cứu khoa học.

Xin tỏ lòng kính trọng và biết ơn sâu sắc đối với thầy Lê Hoài Bắc - giảng viên lý thuyết và thầy Nguyễn Bảo Long - hướng dẫn thực hành đã trực tiếp tận tình hướng dẫn cũng như cung cấp tài liệu thông tin khoa học cần thiết cho Đồ án này.

Em xin chân thành cảm ơn!

Mục lục

Danh sách hình

Danh sách bảng

Chương 1

Thông tin cá nhân

1.1 Thông tin

Họ tên	MSSV
Phạm Nhật Duy	21120058

1.2 Đánh giá đồ án

- Nghiên cứu và trình bày thuật toán: 100%
- So sánh thuật toán: 100%
- Cài đặt thuật toán: 100%

Chương 2

Nội dung đồ án

2.1 Vấn đề tìm kiếm

Tham khảo từ sách *Artificial Intelligence - A Modern Approach* [8] và giáo trình *Cơ sở Trí tuệ nhân tạo* [1].

2.1.1 Bài toán tìm kiếm

Việc tối ưu hóa các biện pháp giải quyết vấn đề đôi khi có thể được đơn giản hóa chỉ bằng cách xác định một đích đến và di chuyển trực tiếp tới đích đến ấy.

Môi trường tìm kiếm cơ bản sẽ là các môi trường mà người ta đã biết đủ trạng thái, dễ quan sát và có tính xác định (tức là mỗi hành động đều có chính xác một kết quả tương ứng). Để tổ chức chuỗi hành động một cách hiệu quả nhất, người ta thường giới hạn các đối tượng có thể trở thành mục tiêu.

Quá trình đi tìm một chuỗi hành động để đạt mục tiêu được gọi là thuật toán tìm kiếm. Kiểu thuật toán này sẽ lấy một vấn đề hay một điểm đầu và điểm đích làm đầu vào, sau đó trả về cho bạn giải pháp dưới dạng chuỗi hành động để di chuyển từ trạng thái đầu đến trạng thái đích cũng tức là phương thức để giải quyết vấn đề.

2.1.2 Các thành phần của bài toán tìm kiếm

Một bài toán tìm kiếm có thể định nghĩa bằng 5 thành phần:

- **Trạng thái bắt đầu** (Initial state):
  Là trạng thái mà từ đó tác nhân bắt đầu.

- **Hành động** (Action):
  Mô tả về các hành động khả thi của tác nhân. Với trạng thái s,

ACTIONS(s) trả về một tập hợp các hành động hữu hạn có thể được thực thi trong s. Chúng ta nói rằng mỗi hành động này có thể áp dụng được trong s.

- **Mô hình di chuyển** (Transition model):
  Mô tả kết quả của các hành động. RESULT(s, a) trả về trạng thái là kết quả của việc thực hiện hành động a ở trạng thái s.

  Thuật ngữ **SUCCESSOR** tương ứng với các trạng thái có thể di chuyển được với một hành động duy nhất.

  Trạng thái bắt đầu, hành động và mô hình di chuyển định nghĩa **không gian trạng thái** (State space) của bài toán. Không gian trạng thái hình thành nên một **đồ thị** có hướng với đỉnh là các trạng thái và cạnh là các hành động.

  Một **đường đi** (Path) trong không gian trạng thái là một chuỗi các trạng thái được kết nối bằng một chuỗi các hành động.

- **Kiểm tra đích** (Goal test):
  Xác định một trạng thái có là trạng thái đích.

- **Chi phí đường đi** (Path cost):
  Hàm chi phí đường đi gán chi phí với một giá trị số cho mỗi đường đi. Chi phí đường đi khi thực hiện hành động a từ trạng thái s đến trạng thái s' là COST(s, a, s').

Một lời giải (solution) là một **chuỗi hành động** di chuyển từ trạng thái bắt đầu cho đến trạng thái đích. Một **lời giải tối ưu** có chi phí đường đi thấp nhất trong tất cả các lời giải.

2.1.3 Mã giả

Tham khảo từ [2].

```
1:  function SIMPLE-PROBLEM-SOLVING-AGENT(percept) return an action
2:      persistent: seq, an action sequence, initially empty
3:                  state, some description of the current world state
4:                  goal, a goal, initially null
5:                  problem, a problem formulation
6:      state ← UPDATE-STATE(state, percept)
7:      if i > seq is empty then
8:          goal ← FORMULATE-GOAL(state)
9:          problem ← FORMULATE-PROBLEM(state, goal)
10:         seq ← SEARCH(problem)
11:         if seq = failure then return a null action
12:     action ← FIRST(seq).
13:     seq ← REST(seq).
14:     return action
```

2.1.4 Phân biệt tìm kiếm không có thông tin và tìm kiếm có thông tin

Tìm kiếm không có thông tin hay **tìm kiếm mù** là kỹ thuật tìm kiếm không có thông tin bổ sung về khoảng cách từ trạng thái hiện tại đến trạng thái đích.

Tìm kiếm có thông tin là một kỹ thuật khác có thông tin bổ sung về khoảng cách ước tính từ trạng thái hiện tại đến trạng thái đích.

Cơ sở so sánh	Tìm kiếm không có thông tin	Tìm kiếm có thông tin
Tri thức cơ bản	Không sử dụng tri thức	Vận dụng tri thức để tìm ra các bước giải quyết vấn đề
Hiệu quả	Trung bình	Cao vì tiêu tốn ít thời gian và chi phí
Chi phí	Tương đối cao	Thấp
Hiệu suất	Tốc độ chậm	Tìm ra giải pháp nhanh
Thuật toán	Depth-first search, breadth-first search, and uniform-cost search	Heuristic greedy best-first search, and A* search

Bảng 2.1: Sự khác nhau giữa tìm kiếm không có thông tin và tìm kiếm có thông tin [4]

2.2 Thuật toán tìm kiếm

Tham khảo từ hai bài giảng [7] và [6].

2.2.1 Tìm kiếm theo chiều sâu (Depth-first search - DFS)

Ý tưởng chung

Tìm kiếm theo chiều sâu luôn mở rộng nút nằm sâu nhất trong nhánh hiện tại đang xét trên cây tìm kiếm. Thuật toán tiến hành tìm kiếm ở mức sâu nhất của cây tìm kiếm, tại đó các nút không có nút con nào. Với những nút đã được mở rộng sẽ được loại khỏi nhánh tìm kiếm, khi đó công việc tìm kiếm sẽ quay trở lại nút gần nhất có các nút con chưa được xét đến. Thuật toán luôn mở rộng nút sâu mãi theo một hướng, chỉ khi nào không mở rộng được nữa thì chuyển sang hướng khác.

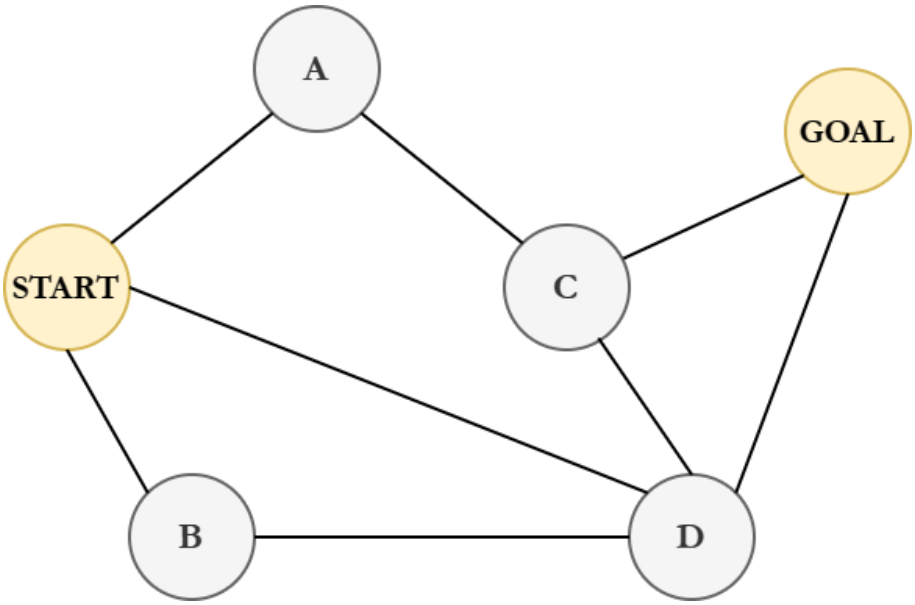
Mã giả

```
1:  function DEPTH-FIRST SEARCH(problem) return a solution or failure
2:      frontier ← node                          ▷ frontier is LIFO queue (stack)
3:      while frontier ≠ ∅ do
4:          node ← POP(frontier)
5:          if problem.GOAL-TEST(node.STATE then return node
6:          else if not IS-CYCLE(node) then
7:              for each action in problem.ACTIONS(node.STATE) do
8:                  child ← CHILD-NODE(problem, node, action)
9:                  frontier ← INSERT(child, frontier)
10:     return failure
```

Phân tích thuật toán

- Tính đầy đủ: Không khi không gian có độ sâu vô hạn hoặc không gian có vòng (chỉ có thể hoàn thành thuật toán trong không gian hữu hạn).
- Tính tối ưu: Không (bất kể độ sâu hay chi phí).
- Độ phức tạp về thời gian: $O(b^m)$, tệ nếu m lớn hơn nhiều so với b. Đối với những vấn đề có nhiều lời giải thì DFS thực sự nhanh hơn BFS.
- Độ phức tạp về không gian: $O(bm)$.

Ví dụ minh họa



Hình 2.1: Đồ thị minh họa DFS

Open	Close
START	Ø
A	START
C	START, A
D	START, A, C
GOAL	START, A, C, D
Ø	START, A, C, D, GOAL

Bảng 2.2: Minh họa thuật toán DFS

Đường đi: $START \rightarrow A \rightarrow C \rightarrow D \rightarrow GOAL$

2.2.2 Tìm kiếm theo chiều rộng (Breadth-first search - BFS)

Ý tưởng chung

Tìm kiếm theo chiều rộng ứng với việc xây dựng cây tìm kiếm theo chiều rộng, nút gốc được mở rộng đầu tiên, sau đó tất cả những nút con có được từ nút gốc sẽ được mở rộng, tiếp theo và sau đó là nút con của chúng và cứ tiếp tục. Tổng quát, ta có thể thực hiện như sau: tất cả các nút ở độ sâu là d trong cây tìm kiếm sẽ được mở rộng trước những nút ở độ sâu $d + 1$. Do chưa xác định được số bước để đi tới đích nên sẽ phải xem xét hết các đường đi có thể lần lượt từ $n = 1, 2, 3...$ tới khi nào gặp đích thì dừng.

Mã giả

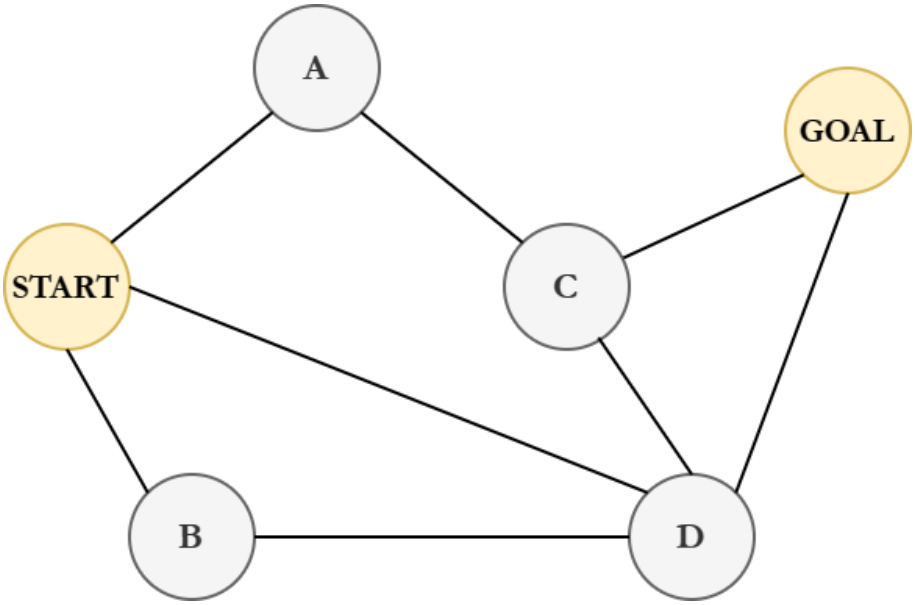
```
1:  function BREADTH-FIRST SEARCH(problem) return a solution or failure
2:      node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
3:      if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
4:      frontier ← node                          ▷ frontier is FIFO queue
5:      explored ← ∅
6:      while frontier ≠ ∅ do
7:          node ← POP(frontier)
8:          add node.STATE to explored
9:          for each action in problem.ACTIONS(node.STATE) do
10:             child ← CHILD-NODE(problem, node, action)
11:             if child.STATE is not in explored or frontier then
12:                 if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
13:                 frontier ← INSERT(child, frontier)
14:     return failure
```

Phân tích thuật toán

- Tính đầy đủ: Có (nếu b hữu hạn).
- Tính tối ưu: Có (nếu chi phí di chuyển như nhau).

- Độ phức tạp về thời gian: $1 + b^2 + b^3 + ... + b^d = O(b^d)$.
- Độ phức tạp về không gian: $O(b^{d-1})$ cho tập mở và $O(b^d)$ cho biên.

Ví dụ minh họa



Hình 2.2: Đồ thị minh họa BFS

Open	Close
START	Ø
A, B, D	START
B, D, C	START, A
D, C, GOAL	START, A, B
C, GOAL	START, A, B, D
GOAL	START, A, B, D, C
Ø	START, A, B, D, C, GOAL

Bảng 2.3: Minh họa thuật toán BFS

Đường đi: $START \rightarrow D \rightarrow GOAL$

2.2.3 Tìm kiếm chi phí đồng nhất (Uniform-cost search - UCS)

Ý tưởng chung

Thuật toán này phù hợp với những đồ thị trạng thái có trọng số, tức là những bài toán tìm kiếm mà các bước biến đổi có chi phí. Thay vì mở rộng nút gần nhất như thuật toán BFS, UCS sẽ mở rộng đến nút có chi phí đường đi tới nó thấp nhất. Để tránh trường hợp lặp vô hạn, thuật toán chi phí đồng nhất cần có điều kiện ràng buộc: chi phí đường đi tại mỗi bước biến đổi phải lớn hơn hoặc bằng một hằng số c dương.

Mã giả

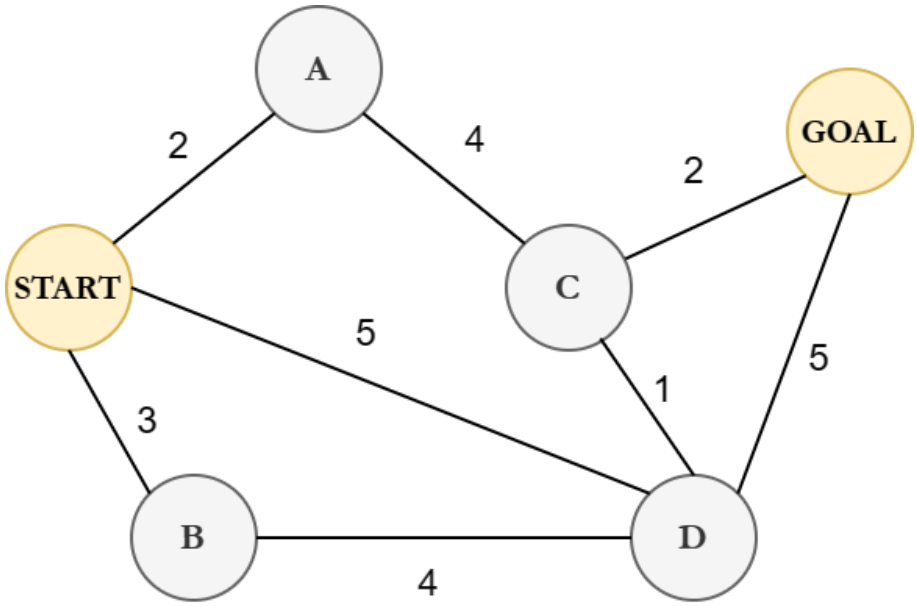
```
1:  function UNIFORM-COST SEARCH(problem) return a solution or failure
2:      node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
3:      frontier ← node
    ▷ frontier is a priority queue ordered by PATH-COST
4:      explored ← ∅
5:      while frontier ≠ ∅ do
6:          node ← POP(frontier)                    ▷ choose the lowest-cost node
7:          if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
8:          add node.STATE to explored
9:          for each action in problem.ACTIONS(node.STATE) do
10:             child ← CHILD-NODE(problem, node, action)
11:             if child.STATE is not in explored or frontier then
12:                 frontier ← INSERT(child, frontier)
13:             else if child.STATE is in frontier with higher PATH-COST then replace that frontier node with child
14:     return failure
```

Phân tích thuật toán

- Tính đầy đủ: Có nếu chi phí di chuyển thấp nhất ϵ là một hằng số dương đủ nhỏ.

- Tính tối ưu: Có. Do các nút được mở rộng theo thứ tự tăng dần của hàm chi phí nên khi bắt gặp nút đích đầu tiên thì đó cũng là lời giải tối ưu.
- Độ phức tạp về thời gian: $O(b^{1+\lfloor C^*/\epsilon \rfloor})$, C^* là chi phí của lời giải tối ưu.
- Độ phức tạp về không gian: $O(b^{1+\lfloor C^*/\epsilon \rfloor})$.

Ví dụ minh họa



Hình 2.3: Đồ thị minh họa UCS

Open	Close
(START, 0)	Ø
(A, 2), (B, 3), (D, 5)	START
(B, 3), (D, 5), (C, 6)	START, A
(D, 5), (C, 6), (GOAL, 8)	START, A, B
(C, 6), (GOAL, 8)	START, A, B, D
(GOAL, 8)	START, A, B, D, C
Ø	START, A, B, D, C, GOAL

Bảng 2.4: Minh họa thuật toán UCS

Đường đi: $START \rightarrow A \rightarrow C \rightarrow GOAL$

Chi phí: $2 + 4 + 2 = 8$

2.2.4 Tìm kiếm A* (A-Star)

Ý tưởng chung

A* sẽ tích hợp *heuristic* vào quá trình tìm kiếm và tránh các đường đi có chi phí lớn. Dựa trên một **hàm đánh giá** để chọn ra nút tốt nhất để mở rộng (thông thường hàm này sẽ tính khoảng cách tới đích, như vậy sẽ chọn được nút có giá trị khoảng cách nhỏ nhất). Thuật toán A* đánh giá gần như tương tự với thuật toán UCS, tuy nhiên A* sử dụng hàm đánh giá là $f(n) = g(n) + h(n)$ thay vì chỉ sử dụng $f(n) = g(n)$ (với $g(n)$ là chi phí đường đi từ nút gốc đến nút n, $h(n)$ là ước lượng chi phí đường đi ngắn nhất từ nút n đến nút đích, $f(n)$ là ước lượng chi phí tốt nhất của lời giải qua n).

Mã giả

Tham khảo từ wikipedia [3].

```
1: procedure RECONSTRUCT_PATH(came_from, current)
2:     total_path ← [current]
3:     while current in came_from do
4:         current ← came_from[current]
5:         total_path.append(current)
6:     return total_path
```

```
1: procedure A*(start, goal)
2:     closedset ← the empty set
3:     openset ← {start}
4:     came_from ← the empty map
5:     g[start] ← 0
6:     f[start] ← g[start] + heuristic_cost_estimate(start, goal)
7:     while openset ≠ ∅ do
8:         current ← the node in openset having the lowest f[] value
9:         if current = goal then
10:            return RECONSTRUCT_PATH(came_from, goal)
11:        remove current from openset
12:        add current to closedset
13:        for all neighbor in neighbor_nodes(current) do
14:            if neighbor in closedset then continue
15:            tentative_g ← g[current] + dist(current, neighbor)
16:            if neighbor not in openset or tentative_g < g[neighbor]
    then
17:                came_from[neighbor] ← current
18:                g[neighbor] ← tentative_g
19:                f[neighbor] ← g[neighbor] +
    heuristic_cost_estimate(neighbor, goal)
20:                if neighbor not in openset then
21:                    add neighbor to openset
22:    return failure
```

Phân tích thuật toán

- Tính đầy đủ: Có, trừ khi có vô số nút có $f < f(GOAL)$.
- Tính tối ưu: Có nếu *heuristic* hợp lý và nhất quán.
- Độ phức tạp về thời gian: cấp số mũ.
- Độ phức tạp về không gian: cấp số mũ, lưu trữ tất cả các nút trong bộ nhớ.

Một vài hàm *heuristic*: [5] Đầu tiên tìm chi phí D tối thiểu để di chuyển từ trạng thái này sang trạng thái liền kề. Trong trường hợp đơn giản, bạn có thể đặt D là 1.

- Khoảng cách **Manhattan** phù hợp khi di chuyển 4 hướng.

```
function heuristic(node):
    dx = abs(node.x - goal.x)
    dy = abs(node.y - goal.y)
    return D * (dx + dy)
```

- Khoảng cách **đường chéo** phù hợp khi di chuyển 8 hướng ($D2$ là chi phí di chuyển theo đường chéo).

```
function heuristic(node):
    dx = abs(node.x - goal.x)
    dy = abs(node.y - goal.y)
    return D * (dx + dy) + (D2 - 2 * D) * min(dx, dy)
```

- Khoảng cách **Euclidean** phù hợp khi di chuyển mọi góc độ.

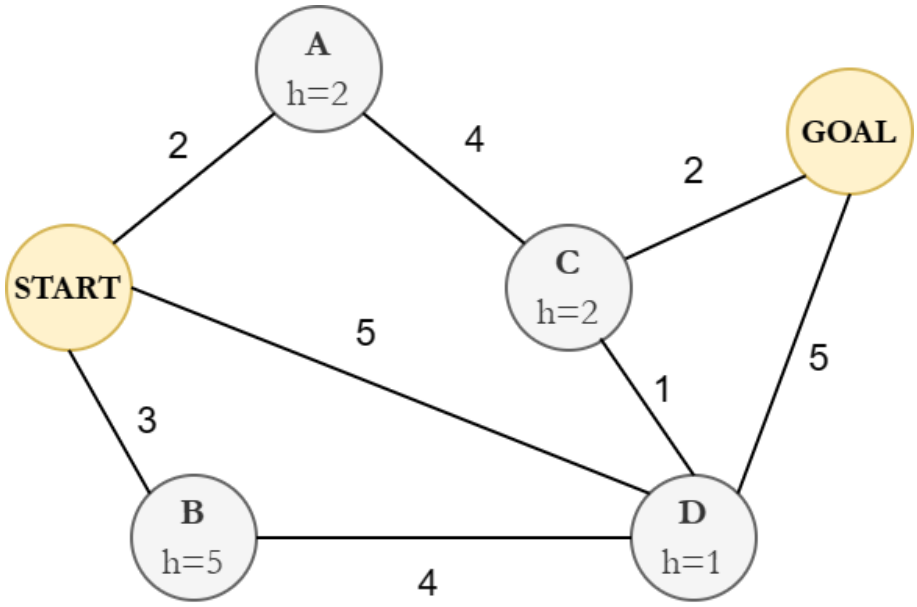
```
function heuristic(node):
    dx = abs(node.x - goal.x)
    dy = abs(node.y - goal.y)
    return D * sqrt(dx * dx + dy * dy)
```

Hàm *heuristic* chấp nhận được [9] nếu nó không bao giờ đánh giá quá cao chi phí thực sự cho mục tiêu gần nhất. Ta xây dựng ràng buộc về khả năng chấp nhận về mặt toán học bằng cách xác định $h^*(n)$ là chi phí di chuyển thực sự tối ưu để đạt được trạng thái mục tiêu gần nhất từ một nút n nhất định:

$$\forall n, 0 \leq h(n) \leq h^*(n)$$

.

Ví dụ minh họa



Hình 2.4: Đồ thị minh họa A-Star

Open	Close
(START, 0)	Ø
(A, 4), (B, 8), (D, 6)	START
(B, 8), (D, 6), (C, 8)	START, A
(B, 8), (C, 8), (GOAL, 8)	START, A, D
(C, 8), (GOAL, 8)	START, A, D, B
(GOAL, 8)	START, A, D, B, C
Ø	START, A, D, B, C, GOAL

Bảng 2.5: Minh họa thuật toán A-Star

Đường đi: $START \rightarrow A \rightarrow C \rightarrow GOAL$

Chi phí: $2 + 4 + 2 = 8$

2.3 So sánh các thuật toán

2.3.1 So sánh: UCS, Greedy, A-Star

Greedy và **A-Star** là những dạng **Best-first search** (tìm kiếm tối ưu) phổ biến nhất. Giữa chúng không có sự khác biệt quá lớn về ý tưởng thuật toán, nên đầu tiên ta sẽ so sánh giữa **UCS** và **Best-first search** để có một góc nhìn trực quan và cụ thể. 2.7

Sự khác nhau giữa **UCS**, **Greedy** và **A-Star** chủ yếu nằm ở hàm ước lượng chi phí của lời giải tối ưu f. Nếu ta xác định $f(n) = g(n)$ (chi phí từ nút gốc đến nút n), ta sẽ được **UCS**. Nếu ta sử dụng *heuristic* $h(n)$ (ước tính chi phí từ nút n hiện tại đến đích) làm hàm $f(n)$, ta nhận được thuật giải **Greedy**.

Khi kết hợp hàm g của **Uniform-cost search** và hàm h của **Greedy Best-first search** ta sẽ có được thuật giải **A-Star** nổi tiếng: $f(n) = g(n) + h(n)$. Do đó, nếu ta định tìm lời giải tốt nhất thì cách dễ thấy nhất là lấy nút có giá trị $g(n) + h(n)$ nhỏ nhất. Hơn nữa nếu hàm *heuristic* chấp nhận được (thỏa điều kiện không bao giờ ước lượng quá chi phí để đến đích thực sự) thì thuật giải tìm kiếm **A-Star** sẽ đầy đủ và tối ưu.

Cơ sở so sánh	UCS	Best-first search
Chiến lược	Tìm kiếm mù, quan tâm chi phi đường đi, không sử dụng hàm đánh giá	Tìm kiếm *heuristic*, có sử dụng hàm đánh giá
Chi phí	Tính từ nút bắt đầu đến nút hiện tại	Tính từ nút bắt đầu đến nút hiện tại
Độ phức tạp theo thời gian	$O(b^d)$	$O(b^m)$
Độ phức tạp theo không gian	$O(b^d)$	$O(b^m)$
Tính tối ưu	Có, khi UCS mở rộng nút mục tiêu, nó sẽ tìm thấy đường dẫn tối ưu đến trạng thái của nó. Bởi vì chi phí biên không âm và UCS mở rộng các nút biên giới theo thứ tự giá trị g của chúng, nên bất kỳ nút mục tiêu nào được mở rộng sau nút mục tiêu đầu tiên sẽ nằm trên đường dẫn ít nhất có chi phí bằng mục tiêu đầu tiên.	Không, vì không đảm bảo đưa ra giải pháp tốt nhất, nhưng với một hàm *heuristic* đủ tốt có thể cái thiện đảng kể
Tính đầy đủ	Có trong trường hợp không gian là hữu hạn và ngay cả nếu vô hạn nhưng không có nút nào có vô hạn lân cận và tất cả chi phí đều dương thì **UCS** cũng sẽ hoàn tất. Điều kiện chi phí biên đều dương đảm bảo rằng sẽ không có một đường dẫn vô hạn với các cạnh có chi phí bằng 0 mà các nút **UCS** sẽ tiếp tục mở rộng mãi dẫn đến mắc kẹt trong vòng lặp.	Không, vì chỉ đi theo một con đường để đến đích nếu gặp ngõ cụt thì sẽ đi sang hướng khác (nó có thể đi vào con đường vô tận và không thể quay lại được)

Bảng 2.6: Sự khác nhau giữa **UCS** và **Best-first search** [10]

2.3.2 So sánh: UCS, Dijkstra

Cả thuật toán **Dijkstra** và thuật toán **Uniform-cost search** đều có thể giải được bài toán tìm đường đi ngắn nhất với cùng độ phức tạp về thời gian. Chúng có cấu trúc mã code tương tự nhau. Ngoài ra, sử dụng cùng một công thức để cập nhật giá trị khoảng cách của mỗi đỉnh: $dist[v] = min(dist[v], dist[u] + w(u, v))$. Có thể phân biệt một cách đơn giản rằng **Dijkstra** là thuật toán tìm đường đi ngắn nhất từ nút này đến mọi nút khác trong đồ thị trong khi **UCS** tìm đường đi ngắn nhất giữa 2 nút.

Sự khác biệt chính giữa hai thuật toán này là **cách lưu trữ các đỉnh** trong Queue. Thuật toán **Dijkstra** yêu cầu nhiều bộ nhớ hơn khi cần lưu trữ toàn bộ đồ thị, do đó thuật toán **Dijkstra** chỉ áp dụng cho các đồ thị tường minh. Ngược lại, thuật toán **UCS** chỉ lưu trữ đỉnh nguồn ở đầu, dần dần đi qua các phần đồ thị cần thiết và ngừng mở rộng khi đến đỉnh đích, vì vậy nó có thể áp dụng cho cả đồ thị tường minh và đồ thị không tường minh.

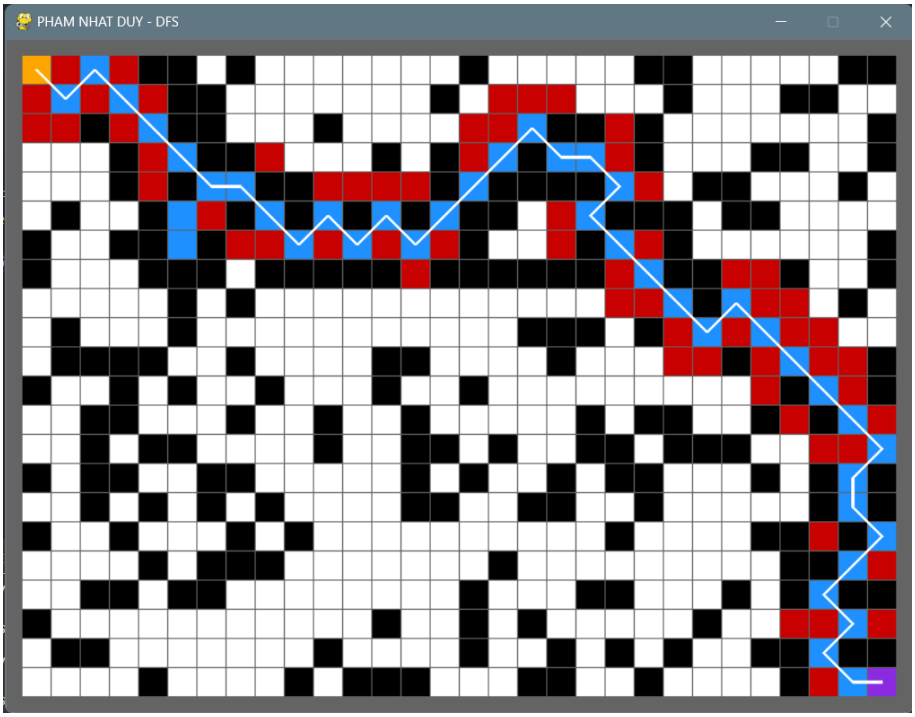
Một điểm khác biệt nhỏ nữa là **giá trị khoảng cách trên các đỉnh không thể tiếp cận được từ đỉnh nguồn**. Trong thuật toán **Dijkstra**, nếu không có đường đi giữa đỉnh nguồn s và đỉnh v thì giá trị khoảng cách của nó $dist[v] = +\infty$. Tuy nhiên, trong thuật toán **UCS**, khi không thể tìm thấy giá trị đó, tức là $dist[v]$ không tồn tại.

Cơ sở so sánh	UCS	Dijkstra
Khởi tạo	Một tập hợp các đỉnh nguồn	Tất cả các đỉnh của đồ thị
Bộ nhớ	Chỉ lưu trữ những đỉnh cần thiết	Lưu trữ cả đồ thị
Thời gian chạy	Nhanh	Chậm do yêu cầu lớn về bộ nhớ
Ứng dụng	Đồ thị tường minh và không tường minh	Đồ thị tường minh

Bảng 2.7: Sự khác nhau giữa **UCS** và **Dijkstra** [11]

2.4 Cài đặt thuật toán

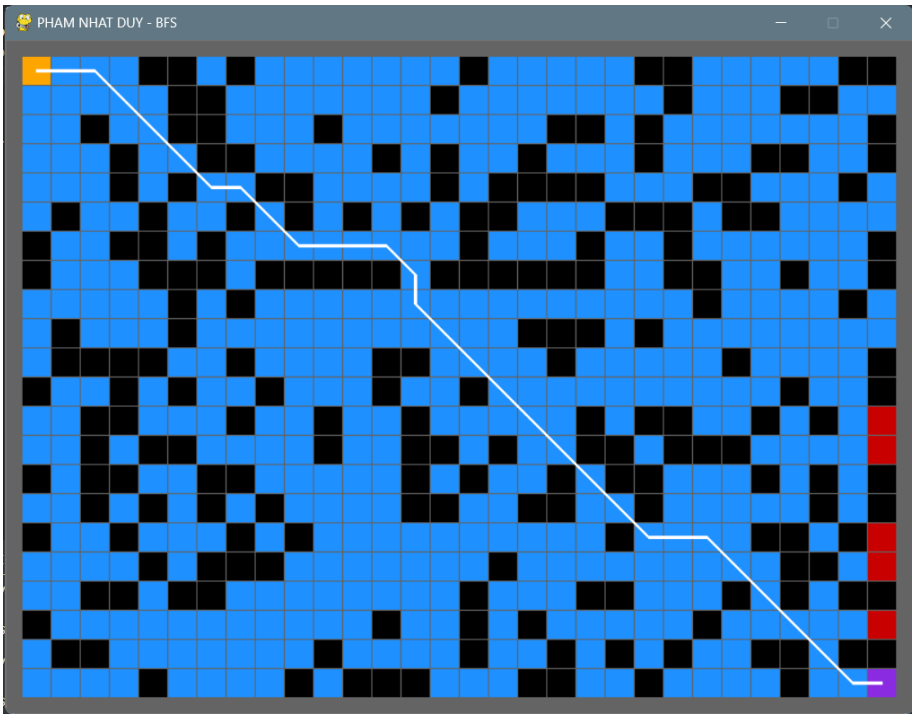
2.4.1 DFS



Hình 2.5: Demo DFS

Hàm **DFS** (Tìm kiếm theo chiều sâu) là một tìm kiếm trên đồ thị bằng cách đi qua tất cả các đường đi có thể từ nút gốc đến nút mục tiêu. Thuật toán hoạt động bằng cách duy trì một ngăn xếp các nút sẽ được truy cập, bắt đầu từ nút gốc và thêm nó vào ngăn xếp. Sau đó, thuật toán truy cập từng nút lân cận của nút gốc, thêm chúng vào ngăn xếp. Nếu thuật toán đến được nút mục tiêu, nó sẽ trả về đường dẫn đến nút đó. Ngược lại, thuật toán sẽ quay lại nút trước đó trên ngăn xếp và lặp lại quy trình cho đến khi ngăn xếp trống hoặc tìm thấy nút mục tiêu.

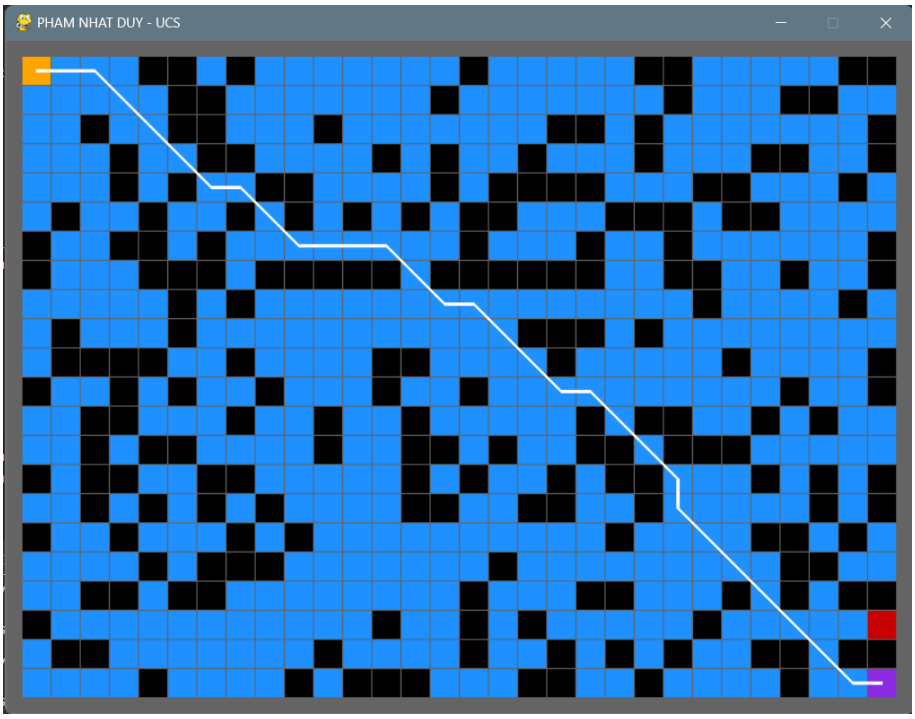
2.4.2 BFS



Hình 2.6: Demo BFS

Hàm **BFS** (Tìm kiếm theo chiều rộng) duyệt đồ thị bằng cách truy cập tất cả các nút trong đồ thị theo từng lớp. Thuật toán hoạt động bằng cách duy trì một hàng đợi các nút sẽ được truy cập, bắt đầu tại nút gốc và thêm nó vào hàng đợi. Sau đó, thuật toán loại bỏ nút đầu tiên khỏi hàng đợi và truy cập tất cả các nút lân cận của nó. Thuật toán sẽ thêm các lân cận vào hàng đợi nếu chúng chưa được truy cập. Thuật toán lặp lại quá trình này cho đến khi hàng đợi trống hoặc tất cả các nút trong đồ thị đã được truy cập.

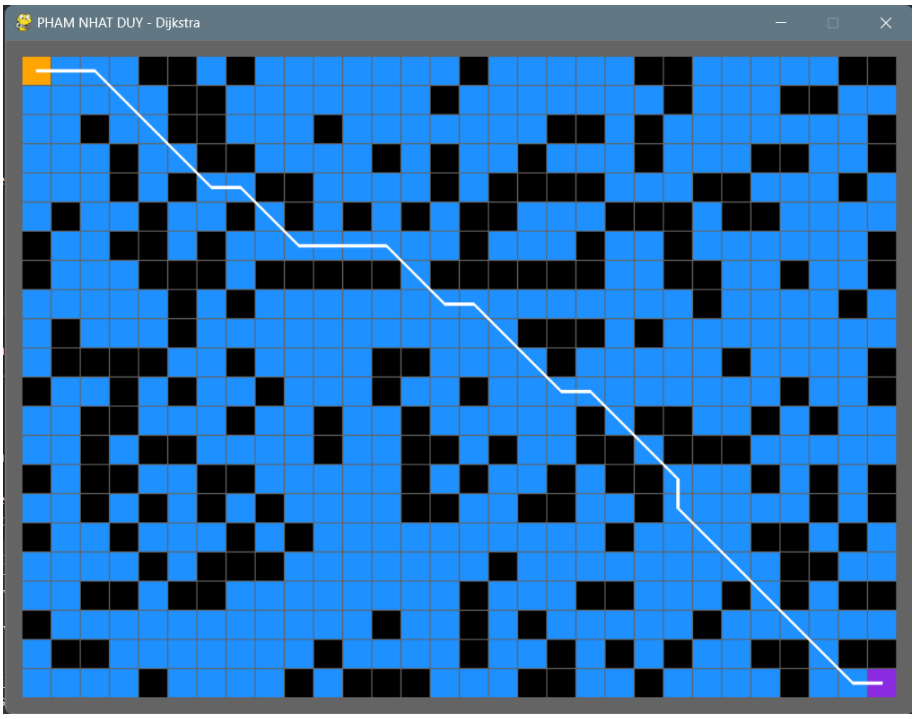
2.4.3 UCS



Hình 2.7: Demo UCS

Hàm **UCS** (Tìm kiếm chi phí đồng nhất) tìm đường đi ngắn nhất giữa hai nút trong đồ thị. Thuật toán hoạt động bằng cách duy trì hàng đợi ưu tiên của các nút sẽ được truy cập. Hàng đợi ưu tiên được sắp xếp theo chi phí của đường đi từ nút bắt đầu đến mỗi nút. Thuật toán bắt đầu tại nút bắt đầu và thêm nó vào hàng đợi ưu tiên. Sau đó, thuật toán loại bỏ nút có chi phí thấp nhất khỏi hàng đợi ưu tiên và truy cập tất cả các nút lân cận của nó. Nếu thuật toán đến được nút mục tiêu, nó sẽ trả về đường dẫn đến nút mục tiêu. Mặt khác, thuật toán sẽ thêm các lân cận vào hàng ưu tiên nếu chúng chưa được truy cập. Thuật toán lặp lại quá trình này cho đến khi hàng đợi ưu tiên trống hoặc tìm thấy nút mục tiêu.

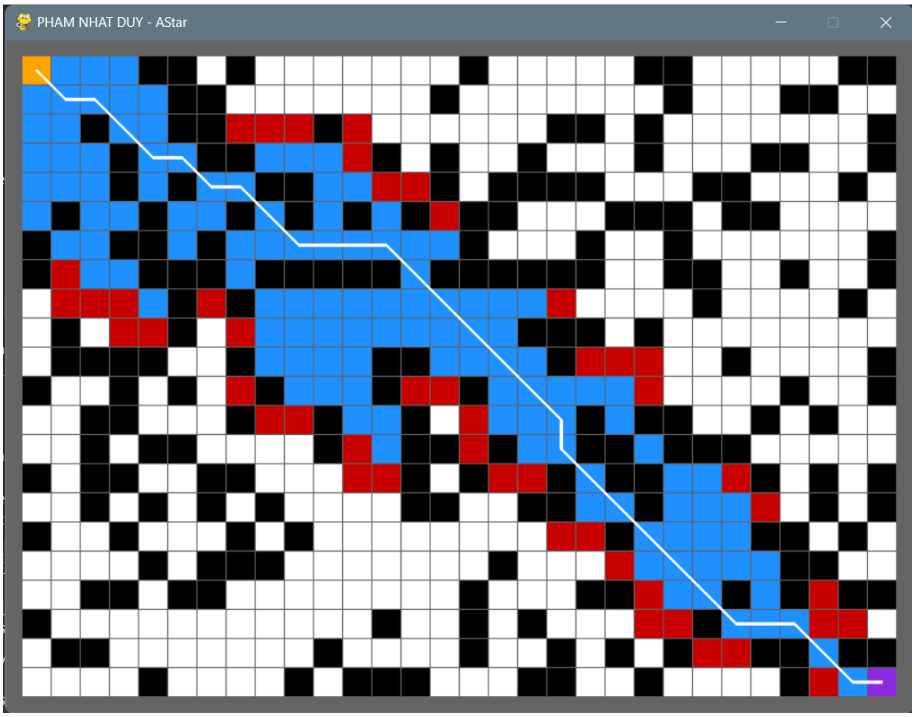
2.4.4 Dijkstra



Hình 2.8: Demo Dijkstra

Dijkstra là thuật toán tìm đường đi ngắn nhất giữa một nút nguồn duy nhất và tất cả các nút khác trong đồ thị. Quá trình tìm kiếm đường đi giữa hai nút hoàn toàn tương tự như thuật toán **UCS** tuy nhiên thuật toán này sẽ kết thúc khi tất cả các nút trong đồ thị đã được duyệt qua hoặc hàng đợi ưu tiên trống.

2.4.5 A-Star



Hình 2.9: Demo A-Star

A* đi tìm đường đi ngắn nhất giữa hai nút trong đồ thị. Với **A*** hàng đợi ưu tiên được sắp xếp theo giá trị f của mỗi nút, là tổng của g và h của nút.

Quá trình tìm kiếm bắt đầu tại nút bắt đầu và thêm nó vào hàng đợi ưu tiên. Sau đó, thuật toán loại bỏ nút có f thấp nhất khỏi hàng đợi ưu tiên và truy cập tất cả các nút lân cận của nó. Nếu thuật toán đến được nút mục tiêu, nó sẽ trả về đường dẫn đến nút mục tiêu. Mặt khác, thuật toán sẽ tính g và h cho mỗi nút lân cận và thêm vào hàng đợi ưu tiên nếu nó chưa có trong hàng đợi ưu tiên hoặc nếu giá trị g mới thấp hơn g hiện tại của lân cận. Thuật toán lặp lại quá trình này cho đến khi hàng đợi ưu tiên trống hoặc tìm thấy nút mục tiêu.

Tài liệu tham khảo

Tiếng Việt

[1] Lê Hoài Bắc and Tô Hoài Việt. *Cơ sở Trí tuệ nhân tạo*. Nhà xuất bản Khoa học và Kỹ thuật, 2014.

Tiếng Anh

[2] Domingos, Pedro. *Chapter 3: Problem-solving and search, CSE 573: Artificial Intelligence I, Spring 2014*. URL: `https://courses.cs.washington.edu/courses/cse573/14sp/slides/chapter03-6pp.pdf` (visited on 10/19/2023).

[3] Online. *A* search algorithm*. URL: `https://en.wikipedia.org/wiki/A*_search_algorithm` (visited on 10/19/2023).

[4] Online. *What is the difference between informed and uninformed searches?* URL: `https://intellipaat.com/community/3654/what-is-the-difference-between-informed-and-uninformed-searches` (visited on 10/19/2023).

[5] Patel, Amit. *Heuristics from Amit's Thoughts on Pathfinding*. URL: `http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html` (visited on 10/19/2023).

[6] Rhodes, Anthony D. *Part 7: Solving problems by searching, A.I. Course Lectures*. URL: `https://web.pdx.edu/~arhodes/ai7.pdf` (visited on 10/19/2023).

[7] Rhodes, Anthony D. *Part 8: Informed Search Algorithms, A.I. Course Lectures*. URL: `https://web.pdx.edu/~arhodes/ai8.pdf` (visited on 10/19/2023).

[8] Russell, Stuart J. and Norvig, Peter. *Artificial Intelligence: A Modern Approach, 4th Edition*. Pearson Education, 1984.

[9] Sharma, Nikhil. *Note 1, CS 188: Introduction to Artificial Intelligence, Spring 2019*. URL: https://inst.eecs.berkeley.edu/~cs188/sp19/assets/notes/n1.pdf (visited on 10/19/2023).

[10] Simic, Milos. *Uniform-Cost Search vs. Best-First Search*. URL: https://www.baeldung.com/cs/uniform-cost-search-vs-best-first-search (visited on 10/19/2023).

[11] Wu, Gang. *Comparison Between Uniform-Cost Search and Dijkstra's Algorithm*. URL: https://www.baeldung.com/cs/uniform-cost-search-vs-dijkstras (visited on 10/19/2023).

Chương A

Các ký hiệu được sử dụng

- b: hệ số phân nhánh hoặc số lượng nút kế thừa của một nút cần được xem xét $b > 1$.
- d: độ sâu hoặc số lượng hành động trong lời giải tối ưu (chi phí hoặc số bước ít nhất).
- m: độ sâu hoặc số lượng hành động tối đa trên bất kỳ đường đi nào.
- C^*: chi phí của lời giải tối ưu.
- ϵ: giới hạn dưới về chi phí của mỗi hành động ($\epsilon > 0$).
- $g(n)$: chi phí đường đi từ nút gốc đến nút n.
- $h(n)$: ước lượng chi phí đường đi từ nút n đến điểm đích.
- $f(n)$: ước lượng chi phí lời giải tốt nhất qua n.