TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN – ĐHQG HCM

KHOA CÔNG NGHỆ THÔNG TIN

BÁO CÁO NHÓM
CƠ SỞ TRÍ TUỆ NHÂN TẠO
Lab 01 - Search

Lớp: Cơ sở Trí tuệ nhân tạo 22_4

GV LT: Bùi Duy Đăng

Giảng viên TH: Nguyễn Thanh Tình

MSSV	Họ và tên
22120271	Dương Hoàng Hồng Phúc
22120286	Lê Võ Minh Phương
22120293	Võ Hoàng Anh Quân
18120159	Nguyễn Chẩn

CHƯƠNG TRÌNH CHÍNH QUY - KHÓA 22

MỤC LỤC

1. Bảng phân công công việc và tự đánh giá

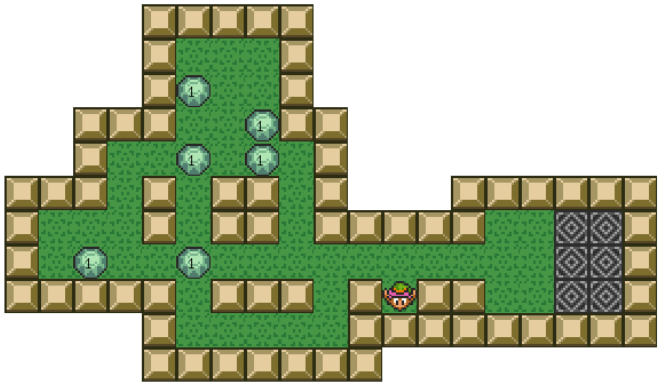
Bảng phân công công việc:

Thành viên	Công việc	Mức độ hoàn thành
Dương Hoàng Hồng Phúc	– Cài đặt thuật toán UCS, AStar – Cài đặt các phương thức trung gian giữa GUI và các thuật toán	100%
Lê Võ Minh Phương	– Cài đặt GUI với Pygame – Test chương trình – Viết báo cáo	100%
Võ Hoàng Anh Quân	– Cài đặt thuật toán BFS – Cài đặt thuật toán DFS – Quay video demo	100%
Nguyễn Chẩn	Không liên hệ để nhận công việc	0%

Bảng chức năng, trọng số và tự đánh giá:

Chức năng	Trọng số	Tự đánh giá
Cài đặt BFS đúng	10%	10%
Cài đặt DFS đúng	10%	10%
Cài đặt UCS đúng	10%	10%
Cài đặt A* đúng	10%	10%
Tạo ít nhất 10 trường hợp kiểm thử với các thuộc tính khác nhau	10%	10%
Tạo được các tập tin output_*.txt và giao diện đồ họa (GUI)	15%	15%
Video demo chương trình một vài trường hợp kiểm thử	10%	10%
Báo cáo các thuật toán, các thử nghiệm và tự đánh giá kết quả	25%	20%
	Tổng điểm tự đánh giá	95%

2. Bài toán Sokoban



Một màn chơi trong Sokoban

Sokoban là một trò chơi giải đố trong đó mục tiêu của người chơi là đẩy các thùng gỗ vào các ô đích trong một mê cung. Trò chơi được tạo ra vào năm 1981 bởi Hiroyuki Imabayashi xuất bản vào 1982 bởi Thinking Rabbit. Trò chơi ban đầu bao gồm 20 màn chơi, sau đó, nhiều hậu bản của trò chơi được ra mắt, bao gồm các màn chơi mới lẫn những bản sao của các màn chơi cũ. Đến nay đã có rất nhiều màn chơi được tạo ra,

từ các màn chơi chính thức đến các màn tạo bởi cộng đồng. Tuy vậy, lối chơi cốt lõi của Sokoban vẫn được giữ nguyên,

Một màn chơi của Sokoban là một bảng các ô vuông, mỗi ô vuông ứng với 1 trong 5 loại vật thể: người chơi, thùng gỗ, tường, sàn và đích. Sokoban có luật chơi như sau:

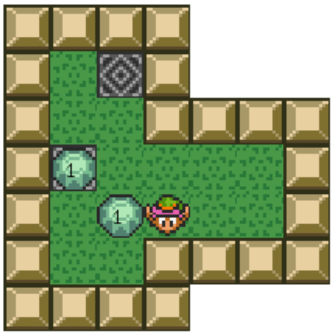
+ Số lượng thùng gỗ luôn bằng số lượng ô đích
+ Người chơi có thể di chuyển tự do theo 4 hướng: lên, xuống, trái, phải trên sàn
+ Người không thể đi xuyên qua tường hay thùng gỗ
+ Trong một lúc, người chơi chỉ có thể đẩy một thùng gỗ theo hướng di chuyển
+ Người chơi không thể kéo thùng gỗ
+ Người chơi không thể đẩy một thùng gỗ vào tường hoặc một thùng gỗ khác
+ Trò chơi kết thúc khi mỗi thùng gỗ đều ở một ô đích.

Sokoban có thể được hình dung trong nhiều ngữ cảnh khác nhau, tùy vào đó mà ta có nhiều cách gọi tên cho 5 loại vật thể trên. Trong giới hạn bài lab này, các "thùng gỗ" được xem như các "hòn đá", các "đích" xem là các "công tắc" và "người chơi" có tên là "Ares".

Độ khó của Sokoban nằm ở việc trong quá trình chơi, người chơi rất dễ gặp các trường hợp các thùng gỗ bị đẩy vào vị trí khiến chúng không thể nào bị di chuyển được nữa. Trong trường hợp khác, các thùng có thể nằm ở vị trí mà chúng không bao giờ có thể được đẩy đến đích. Các trường hợp như vậy gọi là *deadlocks*:

Người chơi không thể đẩy bất kỳ viên đá nào được nữa

Viên đá chỉ có thể đẩy được theo chiều ngang, nó không thể nào đến được đích bên góc trái



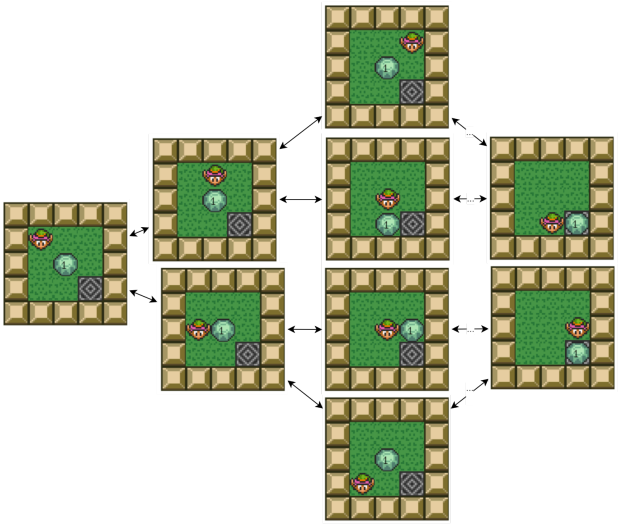
Một màn chơi bị deadlock; bất kỳ chuỗi bước đi tiếp theo nào đều dẫn tới deadlock

Trò chơi Sokoban là một công cụ thú vị để học tập các thuật toán và nghiên cứu về Trí tuệ nhân tạo. Thoạt nhìn luật chơi có vẻ đơn giản nhưng vấn đề cần phải giải quyết thì rất phức tạp. Trò chơi này thuộc loại NPHard và PSPACE-complete, những level đơn giản nhất cũng cần rất nhiều bước tính toán.

Sự phức tạp là kết quả của độ sâu cần khám phá để đạt được mục tiêu (thông thường lên đến hàng trăm) và việc phân nhánh tìm kiếm, cũng như sự tồn tại của các trường hợp dẫn đến bài toán không thể giải (*deadlock*). Mặc dù vậy, Sokoban vẫn có thể được xem là một sự đơn giản hóa cho việc điều khiển robot sắp xếp hàng hóa trong nhà kho, nên nếu có thể vượt qua những khó khăn tồn tại và tìm được thuật toán tối ưu để giải Sokoban thì sẽ mang lại nhiều lợi ích cho các bài toán tương tự khác trong cuộc sống hằng ngày và các lĩnh vực liên quan trong Khoa học Máy tính.

3. Mô hình hóa bài toán Sokoban

Không gian trạng thái (state space) của Sokoban có thể xem như một đồ thị (*graph*). Mọi bước đi của người chơi làm thay đổi màn chơi từ trạng thái này sang trạng thái khác. Do vậy, việc tìm kiếm lời giải cho bài toán là việc phải tìm kiếm một đường đi, thường là tối ưu, giữa các trạng thái dẫn đến một trạng thái kết quả.



Đồ thị biểu diễn không gian trạng thái của một màn chơi Sokoban

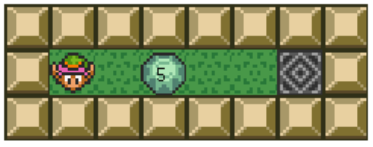
Có nhiều cách định nghĩa trạng thái (*state*), và tùy vào định nghĩa mà hiệu năng, xét về tốc độ và bộ nhớ, của thuật toán có thể thay đổi. Trong bài này, trạng thái được định nghĩa theo hướng đơn giản nhất, đó là *vị trí và nội dung của tất cả các ô* tại một thời điểm. Do Ares và các viên đá chỉ có thể ở trên các ô không phải tường, kích thước của không gian trạng thái có thể tính bằng: *(Diện tích màn chơi - Số ô tường)!*

Theo định nghĩa đó, trạng thái đầu (*initial state*) là vị trí và nội dung của tất cả các ô ở đầu màn chơi. Trạng thái kết quả (*goal state*) của Sokoban cũng định nghĩa là trạng thái mà vị trí của các viên đá trùng với vị trí của các ô đích ban đầu, hay theo cách cài đặt của bài tập này, là trạng thái mà không có viên đá không nằm trên ô đích.

Các hành động (*actions*) làm thay đổi và chuyển tiếp trạng thái trong Sokoban là hành động di chuyển theo 4 hướng của Ares. Tùy vào tình huống, bước đi của Ares có thể là di chuyển đơn thuần trên các ô trống, tức làm thay đổi vị trí của bản thân Ares, hoặc là vừa di chuyển vừa đẩy một viên đá.

Để tìm được đường đi tối ưu, việc tìm kiếm đường đi phải xét đến số bước mà Ares phải đi, và trọng lượng của các viên đá Ares phải đẩy. Ta có định nghĩa chi phí (*cost*) như sau: mỗi bước di chuyển của Ares tốn 1 đơn vị; nếu bước đi của Ares đẩy một viên đá, chi phí thêm vào trọng lượng của viên đá đó vào mỗi lần đẩy. Đồng nghĩa, tổng chi phí của lời giải tính bởi: *Số bước đi của Ares (steps) + Tổng trọng lượng mà Ares đã đẩy(weight).*

Các thuộc tính của môi trường tác vụ (*task environment*) của Sokoban cũng dễ được xác định:



*Ares phải di chuyển liên tục sang phải để đẩy viên đá vào công tắc. Tổng số bước Ares đi là 4 và Ares phải đẩy viên đá 3 lần một viên đá có trọng lượng là 5. Do đó tổng chi phí sẽ là: 4 + 3*5 = 19*

Fully observable	Ares có mọi thông tin về về vị trí và nội dung của các ô trong màn chơi
Single agent	Chỉ có Ares
Deterministic	Trạng thái kế tiếp được xác định hoàn bởi trạng thái hiện tại và hành động của Ares
Sequential	Bước đi này của Ares ảnh hưởng đến quyết định của các bước đi tiếp theo
Static	Các viên đá, các ô tường đều không đổi khi Ares thực hiện tính toán
Discrete	Số trạng thái của môi trường là đếm được
Known	Kết quả của mọi bước đi là biết trước (deadlock hoặc goal)

4. Tổ chức chương trình

Ban đầu, nhóm nhắm tới sử dụng và tổ chức hiệu quả các lớp phong cách lập trình hướng đối tượng (OOP). Trong quá trình thực hiện, cấu trúc chương trình của nhóm có qua nhiều thay đổi, dẫn đến biểu diễn theo lớp đối tượng bằng mô hình UML không thực sự hợp lý. Tuy vậy, nhóm đã cố gắng phân chia các lớp, mô-đun và phân bổ chức năng như sau:

- **bfs.py**, **dfs,py**, **ucs.py**, **astar.py**: lần lượt chứa các lớp **BFSSolver**, **DFSSolver**, **UCSSolver** và **AStarSolver**, về sau gọi chung là các lớp *solver*. Đây là các lớp chứa thuật toán để giải một đầu vào **Sokoban**
- **solver_utils.py** : là mô-đun cung cấp các hàm làm việc chung trong cho các lớp solver
- **sokoban.py** : chứa 3 lớp sau:
    + **Sokoban**: là lớp đọc và lưu trữ các thông tin về trạng thái đầu của màn chơi
    + **SokobanStateDrawingData**: là lớp lưu trữ thông tin cho việc để giao diện vẽ lại các trạng thái từ một lời giải nhằm mục đích trực quan hóa
    + **Record**: là lớp dữ liệu lưu trữ các thống kê kết quả thuật toán, bao gồm số bước Ares phải đi, tổng trọng lượng đá Ares phải đẩy, thời gian chạy và bộ nhớ đã sử dụng
- **runner.py**: chứa duy nhất lớp **Runner**. Lớp này chịu trách nhiệm tạo các thể hiện **SokobanStateDrawingData** từ một lời giải của solver
- **sokoban_gui.py**: là đầu vào của cả chương trình, đây là tệp cài đặt giao diện đồ họa và hiển thị nội dung các màn chơi và lời giải một các trực quan
- **widgets.py**: cung cấp các thành phần đồ họa cho tệp **sokoban_gui.py**
- **widgets_events.py**: chứa các hằng số sự kiện mà pygame sẽ bắt trong quá trình chạy giao diện đồ họa, chủ yếu là các sự kiện với các nút bấm

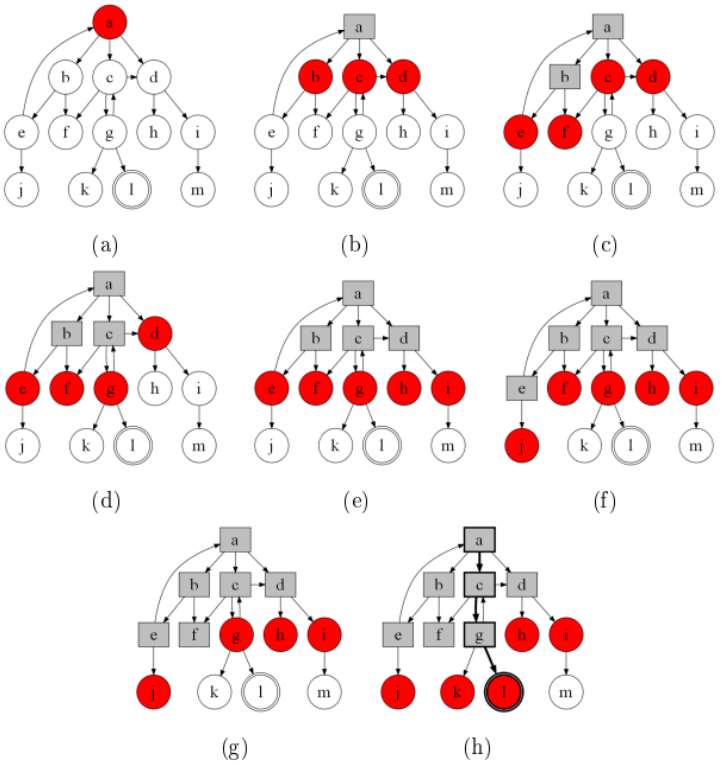
5. Các thuật toán tìm kiếm

a. Breadth-first search (BFS)

Breadth-First Search là thuật toán tìm kiếm đường đi thuộc nhóm tìm kiếm mù (*Uniformed Search*). Ý tưởng của BFS rất đơn giản, bắt đầu từ nút gốc, ta mở rộng các nút con của nó, sau đó với từng nút con ta lại tiếp tục mở rộng các nút con tiếp theo, và cứ như vậy cho đến khi tìm được nút kết quả. BFS cố gắng mở rộng các nút mức độ sâu (*depth*) thấp nhất trước. Để tránh lặp lại các nút đã được xử lý, vì các trạng thái của Sokoban có thể lặp lại trong quá trình chơi, ta lưu trữ các nút đã qua trong một tập hợp gọi là *explored*. Để thực hiện mở rộng theo từng độ sâu, ta mở rộng, lưu trữ và lấy các nút trên một danh sách các nút gọi là *frontier*. Frontier của BFS phải là một *queue*, hoạt động theo quy tắc First-In-First-Out (FIFO) - thêm các nút mở rộng vào đuôi của danh sách và lấy ra từ đầu danh sách.

Xét ví dụ trên một đồ thị dưới đây, ở từng bước ta liệt kê được nội dung của *explored* và *frontier* như bảng bên:

Frontier	Explored
a	
b c d	a
c d e f	a b
d e f g	a b c
e f g h i	a b c d
f g h i j	a b c d e
g h i j	a b c d e f
h i j k l	a b c d e f g



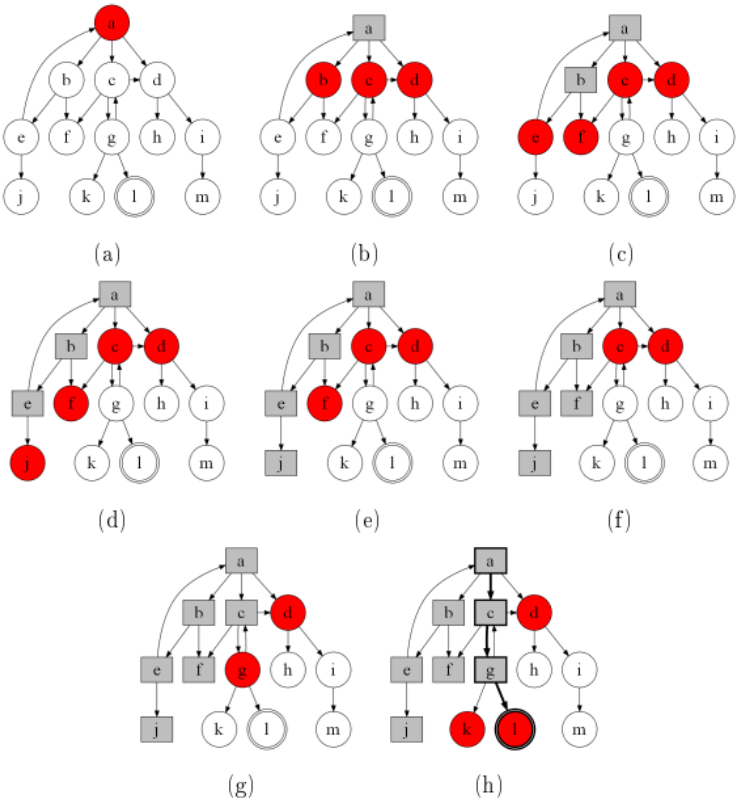
BFS là Hoàn thiện (*Complete*) và Tối ưu (*Optimal*). Nghĩa là nó bảo đảm tìm được kết quả, với điều kiện thời gian đủ lâu và bộ nhớ đủ lớn, và nó sẽ tìm được đường đi ngắn nhất, với điều kiện các chi phí trên từng cạnh là giống nhau và không âm

b. Thuật toán Depth-First Search (DFS):

Depth-First Search (DFS) cũng là một thuật toán tìm kiếm mù. Khác với BFS, ý tưởng của DFS là tìm kiếm toàn bộ trong một nhánh phụ trước khi chuyển sang một nhánh khác. DFS ưu tiên mở rộng nút đang ở mức độ sâu (*depth*) sâu nhất trước. DFS có *frontier* là một danh sách dạng *stack*, làm việc theo nguyên tắc Last-In-First-Out (LIFO) - thêm và lấy các nút mở rộng vào đuôi của danh sách. Cũng như BFS, ta lưu các nút đã xử lý trong một danh sách *explored* và kiểm tra trước khi mở rộng nút.

Xét ví dụ trên một đồ thị dưới đây, ở từng bước ta liệt kê được nội dung của *explored* và *frontier* như bảng bên:

Frontier	Explored
a	
d c b	a
d c f e	a b
d c f j	a b e
d c f	a b e j
d c	a b e j f
d g	a b e j f c
d l k	a b e j f c g



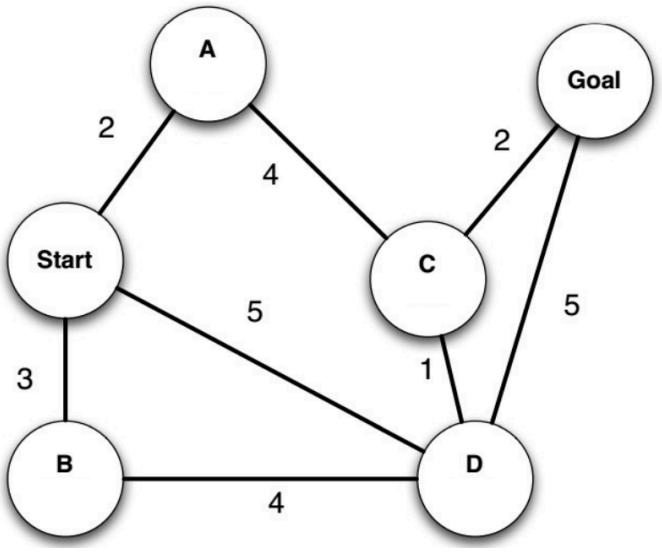
DFS là Hoàn thiện (Complete) nhưng không Tối ưu (Optimal). Nó bảo đảm tìm ra nút kết quả, nhưng không đảm bảo rằng không có đường đến nút kết quả đó (hoặc một nút kết quả khác) ở mức thấp hơn

c. Thuật toán Uniform Cost Search (UCS):

Uniform Cost Search (UCS) là thuật toán tìm kiếm mù. Khác với BFS và DFS, UCS có tính đến chi phí cạnh giữa các nút, dựa vào đó UCS tìm kiếm đường đi đến đích với chi phí thấp nhất. UCS sẽ ưu tiên mở rộng các nút có tổng chi phí đường đi hiện tại là thấp nhất. *Frontier* của UCS là một Priority Queue, các nút được đánh độ ưu tiên theo chi phí đường đi hiện tại đến nút đó, các chi phí bằng nhau sẽ sắp xếp theo thứ tự FIFO. UCS cũng ghi nhớ các nút đã xử lý bằng danh sách *explored*. Điều đáng chú ý là ta chỉ có thể kết thúc thuật toán khi nút kết quả dược lấy ra khỏi *frontier*, khác với BFS hay DFS khi mà ta có thể xác nhận kết quả ngay khi thấy nút kết quả. Ngoài ra ta phải cập nhật độ ưu tiên trong *frontier* khi phát hiện các nút có chi phí đường đi thấp hơn.

Xét ví dụ trên một đồ thị dưới đây, ở từng bước ta liệt kê được nội dung của *explored* và *frontier* như bảng bên. Ký hiệu: *Nút* : *chi phí*

Frontier	Explored
Start	
A:2,B:3, D:5	Start
B:3, D:5, C:6	A:2
D:5, C:6	A:2, B:3
C:6, Goal:10	A:2, B:3, D:5
Goal : 10	A:2, B:3, D:5, C:6
Goal bị lấy ra khỏi frontier và thuật toán kết thúc	



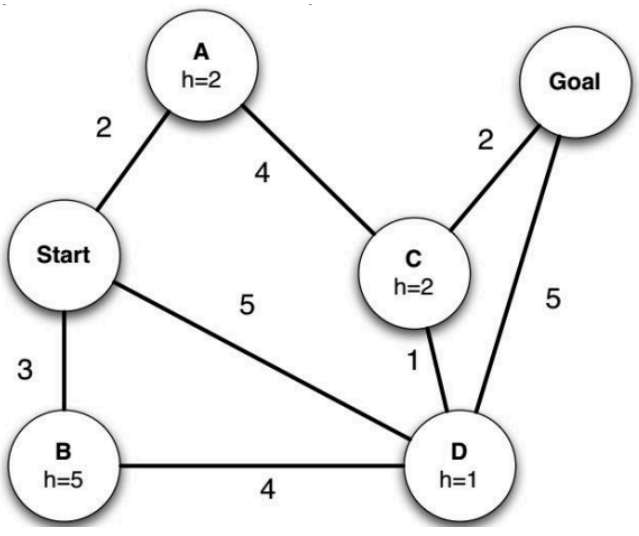
UCS đảm bảo tính Hoàn thiện (*Complete*). Với điều kiện các chi phí cạnh là không âm, UCS là Tối ưu (*Optimal*).

d. Thuật toán A* (A Star):

A* thuộc nhóm thuật toán tìm kiếm dựa kinh nghiệm (Informed Search hay Heuristic Search). A* tìm đường đi ngắn nhất trong đồ thị dựa trên của tổng của hai yếu tố: chi phí đường đi đến nút hiện tại và chi phí heuristic của nút. Cách tiếp cận của A Star với chi phí của đường đi giống với UCS, và chi phí này được gọi là *g cost*. Chi phí heuristic của nút đóng vai trò là một giá trị xấp xỉ chi phí thực đến nút kết quả, giúp định hướng thuật toán mở rộng các nút có xu hướng gần nút kết quả hơn, đây gọi là *h cost*. Chi phí tổng *f cost* của hai giá trị này là cơ sở sắp độ ưu tiên của các nút trong *frontier* Priority Queue.

Xét ví dụ trên một đồ thị dưới đây, ở từng bước ta liệt kê được nội dung của *explored* và *frontier* như bảng bên. Ký hiệu: *Nút-f cost|g cost-Nút cha*

Frontier	Explored
Start	
A-4|2-Start, D-6|1-Start, B-8|3-Start	Start
D-6|1-Start, B-8|3-Start, C-8|6-A	Start, A
B-8|3-Start, C-8|6-A, Goal-10|10-D	Start, A, D
C-8|6-A, Goal-10|10-D	Start, A, D, B
Goal-8|8-C	Start, A, D, B, C
Goal bị lấy ra khỏi frontier và thuật toán kết thúc	



A* đảm bảo tính Hoàn thiện (*Complete*). Đối với tính Tối ưu (*Optimal*), *hàm heuristic* của thuật toán phải đẩm bảo 2 tính chất sau:

+ Tính chấp nhận được (*Admissible*): giá trị heuristic của bất kỳ nútnào phải luôn không âm và không lớn hơn độ dài đường đi thực sự từ node đó đến node đích.
+ Tính nhất quán (*Consistent*): sai khác giá trị heuristic giữa hai nút liên tiếp phải không lớn hơn chi phí đi từ nút này đến nút còn lại

6. Cài đặt

a. Cách lưu trữ trạng thái trong cài đặt

Theo yêu cầu của bài lab, ta có tệp văn bản đầu (*input-*.txt*) vào biểu diễn nội dung của màn chơi, hay trạng thái ban đầu, như quy tắc sau:

- Hàng đầu tiên của văn bản chứa các số nguyên, cách nhau bằng dấu cách, chỉ ra trọng lượng của từng viên đá theo thứ tự xuất hiện của trạng thái ban đầu
- Các dòng tiếp theo biểu diễn từng dòng của màn chơi theo quy ước:
+ “**#**” cho các ô là tường
+ “ “ (khoảng trắng) cho các ô trống
+ “**$**” cho các viên đá
+ “**.**” cho các ô là công tắc
+ “**+**” thể hiện Ares đang ở trên công tắc
+ “*****” thể hiện một viên đá đang ở trên công tắc

```
1 1
####
# .#
#  ####
#*@   #
#  $  #
#  ####
####
```

Ví dụ về tệp đầu vào

Như đã đề cập, tại một thời điểm, trạng thái của Sokoban được lưu trữ là toàn bộ vị trí và nội dung của các ô trong màn chơi. Lý do cho cách lưu trữ đơn giản này là vì ta chỉ cần lưu toàn bộ trạng thái màn chơi bằng một chuỗi ký tự (*string*) đơn giản, bổ sung thêm số dòng và số cột (hai thông số này chỉ cần lưu một lần). Ta có thể sử dụng vị trí của ký tự trên chuỗi và sử dụng phép chia số nguyên (*//*) và phép chia dư (*%*) để suy ra vị trí 2 chiều của ô trên màn chơi; nội dung của ô cũng là nội dung ký tự. Lưu trữ dưới dạng chuỗi giúp đơn giản hóa quá trình so sánh trạng thái trong *frontier* và *explored* (sẽ đề cập sau), khi ta chỉ cần so sánh hai chuỗi trạng thái với phép so sánh bằng (==). Bằng cách này, trạng thái ban đầu cho ví dụ trên có thể được lưu bằng chuỗi:

“#### # .# # #####*@ ## $ ## ######## ”

Một hạn chế của cách tiếp cập này, trong trường hợp của bài lab, đó là khi nhìn vào một trạng thái, ta không xác định được viên đá nào là viên đá nào. Mỗi viên đá có trọng lượng riêng, và cũng vì tính toán chi phí để lựa chọn đường đi hợp lý phụ thuộc vào trọng lượng của từng viên đá, ta phải có cách định danh từng viên đá trong một trạng thái.

Để làm được điều này, trong quá trình chạy thuật toán, thay vì chỉ đánh dấu nội dung một ô chứa đá với “$” và “*”, ta có thể dùng các ký tự chữ cái A-Z cho các viên đá trên khoảng trống, và a-z cho các viên đá trên công tắc, và đánh dấu theo thứ tự xuất hiện của chúng khi bắt đầu tính toán. Cách làm này đương nhiên chỉ giải quyết cho trường hợp với tối đa 26 viên đá, nhưng nó là đủ cho hầu hết các màn chơi Sokoban thông thường. Như vậy, thay vì lưu như chuỗi phía trên, ví dụ trước đó có thể lưu bằng chuỗi:

“#### # .# # #####b@ ## A ## ######## ”

Trong các mục tiếp theo, thuật ngữ nút và trạng thái có nghĩa như nhau.

b. Cấu trúc chung của giải thuật

Tất cả các thuật toán được cài đặt về sau đều có một cấu trúc chung. Ta có *frontier* để lưu trữ các trạng thái sắp mở rộng và *explored* để lưu trữ các trạng thái đã xử lý. Thứ tự nút được mở rộng sẽ phụ thuộc cách cài đặt của *frontier* ở mỗi thuật toán. Để mở rộng một trạng thái, ta đơn giản cho Ares thử đi trong 4 hướng di chuyển và nhận về các trạng thái tiếp theo (nếu hợp lệ). Thêm vào đó, để giúp giảm số lần rẽ nhánh, các thuật toán thực hiện kiểm tra *deadlock* cho trạng thái trả về và loại bỏ chúng nếu chúng khớp một trong các *mẫu deadlock*.

Quá trình tính toán trên các trạng thái đòi hỏi ta phải lưu trữ nhiều hơn một chuỗi ký tự trạng thái trong *frontier*, ta luôn cần thêm các biến bổ sung như: *ares position* - dùng để lưu trữ vị trí của Ares từ trong trạng thái để tính các bước di chuyển tiếp theo; *current cost* chứa chi phí đường đi hiện tại của trạng thái; *path* để lưu lại các bước đã đi để đạt được trạng thái. Để lưu nhiều gía trị như vậy trong cùng một biến, trong Python ta dùng *tuple* hoặc *list*. Phần cài đặt của bài tập này sử dụng *tuple*. Các giá trị nói trên có thể được phân lưu như một tuple và phân rã để lấy giá trị khi bắt đầu tính toán.

Để cập nhật *path* theo đầu ra của đề bài, ta ánh xạ 4 hướng di chuyển với các ký tự tương ứng: “l”, “r”, “u”, “d”. Nếu trong quá trình di chuyển thử theo 4 hướng đó, Ares đẩy một viên đá, cờ *pushed* trả về True và ta chỉ cần viết hoa ký tự được ánh xạ.

Dưới đây là pseudo code chung cho các thuật toán:

```
frontier = initialize with (initial state, initial ares position, zero path cost, empty path)
explored = empty set

while frontier is not empty:
  state, ares postion, move cost, path = get from frontier
  add state to explored

  for move in four directions:
    new state, move cost, pushed = can move from ares position to (ares position + move) in state

    if new state is null:
      continue
    if new state in frontier or new state in explored:
      continue
    if new state is deadlock
      continue

    if pushed:
      new state path = path + (map move to char) to lowercase
       else:
         new state path = path + (map move to char) to uppercase

    new state cost = current cost + move cost
    new ares position = ares position + move

    if new state is solution:
      return new state path

    add (new state, new ares position, new state cost, new state path) to frontier
```

c. Các deadlock

Có rất nhiều mẫu deadlock khác nhau. Mục đích của việc tạo nhiều deadlock là để bỏ đi những nhánh không cần thiết, dù Ares vẫn có thể di chuyển và thực hiện các di chuyển hợp lệ, giúp tiết kiệm bộ nhớ. Tuy nhiên, quá nhiều deadlock cũng dẫn đến thời gian chạy tăng do mỗi trạng thái con phải qua so khớp với nhiều mẫu. Thông thường, đánh đổi này là chấp nhận được vì số nhánh được bỏ đi là đủ lớn để cải thiện bộ nhớ lẫn thời gian chạy một cách đáng kể, hầu hết là tốt hơn so với khi không dùng kiểm tra deadlock.

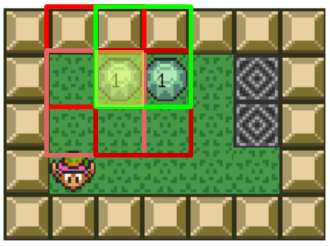
Phần cài đặt của nhóm sử dụng các mẫu deadlock thường gặp sau:

+ Deadlock góc (*Corner deadlock*): Các vị trí mà một viên đá bị kẹt trong góc tường

Deadlock dạng này có thể được kiểm tra bằng cách lặp qua từng viên đá và kiểm tra tra liệu có một ô theo chiều ngang và chiều dọc bên cạnh viên đá là ô tường hay không

+ Deadlock nhiều viên đá (*Multi-boxes deadlocks*): Các trường hợp 2x2 của các deadlock này bao gồm từ đến 4 viên đá nằm sát nhau và cản trở khả năng bị dịch chuyển của nhau

Ý tưởng để xác deadlock theo mẫu này, đó là với mỗi ô chứa viên đá, ta lặp qua 4 khung 2x2 có chứa viên đá và xem xét nội dung của khung có trùng vào một trong các mẫu deadlock hay không.



Ta dùng 4 khung 2x2 xung quanh mỗi viên đá để xác định deadlock

Trong mã nguồn, hàm **is_deadlock()** trong **solver_utils.py** giúp kiểm tra trạng thái vừa được mở rộng có chứa deadlock hay không bằng cách so khớp mẫu như vừa trình bày phía trên

d. Di chuyển khả thi

Để thực sự mở rộng các trạng thái, ta di chuyển Ares theo 4 hướng và xem xét các trạng thái trả về. Di chuyển là khả thi và trả về trạng thái khác None nếu nó không vi phạm 2 luật cơ bản của Sokoban: không đi qua tường và không đẩy một viên đá vào tường hay một viên đá khác.

Để xem xét hai điều kiện này trong một trạng thái, ta xét hai vị trí: *vị trí tiếp theo của Ares* theo hướng di chuyển và *vị trí liền ngay sau đó*. Tùy vào nội dung của 2 ô này ta có thể xét tính hợp lệ và trạng thái tiếp theo sau khi di chuyển

Vị trí kế tiếp của Ares	Vị trí liền sau vị trí kế tiếp	Trạng thái kế tiếp
		
Đá	Khoảng trống	Đẩy
Đá	Công tắc	Đẩy
Đá	Tường	Không hợp lệ
Đá	Đá	Không hợp lệ
Khoảng trống	<Bất kỳ>	Di chuyển
Tường	<Bất kỳ>	Không hợp lệ

Khi trạng thái kế tiếp là hợp lệ thì nó được trả về cho để thuật toán thực hiện các bước kiểm tra tiếp theo. Trong mã nguồn, hàm này là hàm **can_move()** trong **solver_utils.py**

e. Cài đặt các thuật toán

Các solver cần được khởi tạo với một thể hiện **Sokoban**, là thể hiện lưu thông tin ban đầu của màn chơi. Mỗi solver đều có hàm **solve()** để thực hiện giải màn chơi và trả về đường đi dưới dạng chuỗi ký tự của các hướng di chuyển, nếu không có đáp án, **solve()** trả về None.

Bản thân lớp **Sokoban** nhận vào đầu vào là đường dẫn đến một tệp *input-*.txt*. Lớp **Sokoban** sẽ đọc và khởi tạo trạng thái ban đầu, số dòng, số cột của màn chơi, vị trí công tắc và trọng lượng các viên đá.

Ví dụ, để chạy thuật toán BFS trên đầu vào "input-01.txt" ta thực hiện các lệnh Python:

```
sokoban_game = Sokoban("input-01.txt")
sokoban_solver = BFSSolver(sokoban)
solution = sokoban_solver.solve()
```

i. Cài đặt Breadth-First-Search (BFS)

Cài đặt của BFS nằm trong tệp **bfs.py** và trong lớp **BFSSolver**.

Như đã đề cập các thuật toán cấu trúc khá tương tự nhau, điểm khác biệt nằm ở cách cài đặt các *frontier*. BFS sử dụng một danh sách dạng *queue* để lưu trữ các trạng thái sẽ mở rộng tiếp theo. Các trạng thái được thêm vào đuôi và được lấy từ đầu danh sách. BFS không quan tâm đến chi phí cạnh, hay chí phí chuyển đổi giữa các trạng thái, tuy nhiên để ghi lại thông số thuật toán, ở mỗi trạng thái ta vẫn lưu một giá trị *current cost*.

```
frontier = initialize with (initial state, initial ares position, zero path cost, empty path)
explored = empty set

while frontier is not empty:
  state, ares postion, move cost, path = frontier→pop front
  add state to explored

  for move in four directions:
    new state, move cost, pushed = can move from ares position to (ares position + move) in state

    if new state is null:
      continue
    if new state in frontier or new state in explored:
      continue
    if new state is deadlock
      continue

    if pushed:
      new state path = path + (map move to char) to lowercase
       else:
         new state path = path + (map move to char) to uppercase

    new state cost = current cost + move cost
    new ares position = ares position + move

    if new state is solution:
      return new state path

    frontier←push back (new state, new ares position, new state cost, new state path)
```

ii. Cài đặt Depth-First-Search (DFS)

Cài đặt của DFS nằm trong tệp **dfs.py** và trong lớp **DFSSolver**.

DFS sử dụng một danh sách dạng *stack* để lưu trữ các trạng thái sẽ mở rộng tiếp theo. Các trạng thái được thêm vào đuôi và được lấy từ đầu danh sách. DFS không quan tâm đến chi phí cạnh, hay chí phí chuyển đổi giữa các trạng thái, tuy nhiên để ghi lại thông số thuật toán, ở mỗi trạng thái ta vẫn lưu một giá trị *current cost*.

```
frontier = initialize with (initial state, initial ares position, zero path cost, empty path)
explored = empty set

while frontier is not empty:
  state, ares postion, move cost, path = frontier→pop back
  add state to explored

  for move in four directions:
    new state, move cost, pushed = can move from ares position to (ares position + move) in state

    if new state is null:
      continue
    if new state in frontier or new state in explored:
      continue
    if new state is deadlock
      continue

    if pushed:
      new state path = path + (map move to char) to lowercase
    else:
        new state path = path + (map move to char) to uppercase

    new state cost = current cost + move cost
    new ares position = ares position + move

    if new state is solution:
      return new state path

    frontier←push back (new state, new ares position, new state cost, new state path)
```

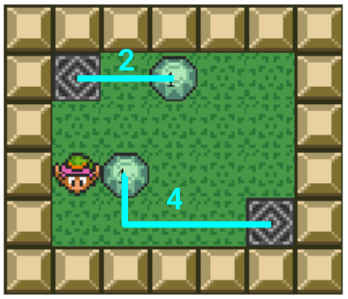
iii. Cài đặt A Star (A*)

Cài đặt của A*nằm trong tệp **astar.py** và trong lớp **AStarSolver**.

AStar sử dụng một danh sách dạng Priority Queue (sử dụng Heap) để lưu trữ các trạng thái sẽ mở rộng tiếp theo. Các trạng thái được thêm vào danh sách và các nút có chi phí đường đi ngắn nhất hiện có sẽ được ưu tiên mở rộng trước

A* sử dụng phương pháp tính *tổng khoảng cách Manhattan* để tính gía trị heuristic của một tạng thái (*h cost)*, sau đó cộng thêm chi phí đường đi thực tế, lưu trong là *g cost*, từ trạng thái bắt đầu đến vị trí hiện tại để tính tổng chi phí đường đi, lưu trong *current f cost*. Khi lấy nút từ *frontier*, A* ưu tiên lấy các nút có *current f cost* nhỏ nhất. Nếu tồn tại các chi phí *current f cost* bằng nhau thì chọn nút để mở rộng theo nguyên tắc FIFO. Ngoài ra, ở mỗi trạng thái ta vẫn lưu một giá trị *current g cost* và cập nhật giá trị trong *frontier* khi phát hiện đường đi ngắn hơn.

Hàm heuristic dựa trên tổng Manhattan sẽ tính tổng khoảng cách từ mỗi viên đá đến công tắc gần nó nhất. Trong quá trình gán các viên đá với công tắc gần nhất, ta không nên gán nhiều viên đá cho một công tắc, vì điều này làm giảm độ chính xác của hàm heuristic.



Viên đá trên gần với công tắc phải nhất nên tính vào h cost là 2;
Viên đá dưới gần công tắc trên hơn, nhưng do công tắc đó đã được gán cho viên đá phía trên, khoảng cách từ nó đến công tắc còn lại, là 4, sẽ được tính vào h cost.

Tổng khoảng cách Manhattan thỏa tính chất Chấp nhận được (*Admissible*) và Tính nhất quán (*Consistent*):

- *Admissible*: Heuristic này không bao giờ đánh giá quá cao chi phí thực sự để đến đích. Trong trường hợp này, hàm heuristic tính toán khoảng cách Manhattan từ mỗi viên đá đến công tắc gần nhất. Khoảng cách Manhattan luôn nhỏ hơn hoặc bằng chi phí thực sự để di chuyển viên đá đến công tắc, vì nó không tính đến các chướng ngại vật hoặc các đường đi phức tạp hơn

- *Consistent*: Heuristic này thỏa mãn điều kiện tam giác, nghĩa là chi phí ước lượng từ trạng thái hiện tại đến đích không lớn hơn chi phí di chuyển từ trạng thái hiện tại đến trạng thái kế tiếp cộng với chi phí ước lượng từ trạng thái kế tiếp đến đích. Điều này đảm bảo rằng giá trị heuristic không giảm khi tiến gần đến đích.

```
frontier = initialize with (zero f cost, zero g cost, initial state, initial ares position,
empty path)
explored = empty set

while frontier is not empty:
 current fcost, current gcost, state, ares position, path = frontier→pop smallest f cost
 add state to explored

 if state is solution:
  return path

 for move in four directions:
  new state, move cost, pushed = can move from ares position to (ares position + move) in state

  if new state is null:
    continue
  if new state in frontier or new state in explored:
    continue
  if new state is deadlock
    continue

  if pushed:
     new state path = path + (map move to char) to lowercase
  else:
     new state path = path + (map move to char) to uppercase

     new gcost = current gcost + move cost
     new fcost = new gcost + manhattan distance of new state
     new ares position = ares position + move

  if new state in frontier and new gcost < cost of gstate in frontier:
     set gcost of (new state) in frontier to (new gcost)
     set fcost of (new state) in frontier to (new fcost)
     set path of (new state) in frontier to (new state path)
     continue

frontier←push (new fcost, new gcost, new state, new ares position, new state path)
```

Cách tính khoảng cách Manhattan:

```
function manhattan_distance(state):
    g = self.g
    stones = get_stones(g, state)
    assigned = empty set
    total_cost = 0

    for each stone in stones:
        min_cost = infinity
        min_switch = 0
        for i from 0 to length of g.switch_pos - 1:
            if i in assigned:
                continue
            switch = convert g.switch_pos[i] to 2D position
            distance = absolute value of (stone.x - switch.x) + absolute value of (stone.y -
switch.y)
            if distance < min_cost:
                min_cost = distance
                min_switch = i
        total_cost = total_cost + min_cost
        add min_switch to assigned

    return total_cost
```

iv. Cài đặt Uniformed-Cost-Search (UCS)

Cài đặt của DFS nằm trong tệp **ucs.py** và trong lớp **UCSSolver**.

UCS sử dụng một danh sách dạng *priority queue* để lưu trữ các trạng thái sẽ mở rộng tiếp theo. UCS tính đến chi phí khi chuyển đổi của các trạng thái và được tính theo quy ước về chi phí trong mục mô hình hóa Sokoban. Độ ưu tiên của mỗi trạng thái xét bằng chi phí đường đi của trạng thái đó. Khi lấy trạng thái ra *frontier*, ta lựa chọn trạng thái có độ ưu tiên *nhỏ nhất*, cũng là trạng thái có chi phí đường đi đến đó là thấp nhất. Các trạng thái có cùng độ ưu tiên, tức cùng chi phí đường đi, sẽ được lấy vào và ra theo thứ tự FIFO.

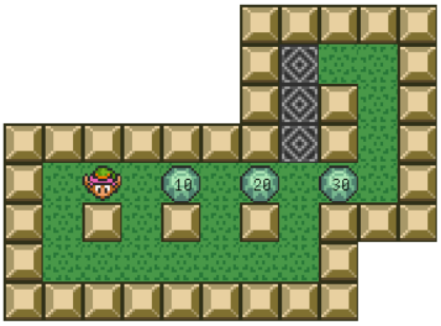
Để đảm bảo lời giải là tối ưu, ta chỉ dừng lại khi ta một trạng thái kết quả được lấy ra khỏi *frontier*. Bên cạnh đó, trong quá trình mở rộng, nếu ta tìm trạng thái được mở đã nằm trong *frontier* và chi phí đường đi hiện tại nhỏ hơn chi phí được ghi nhận trong *frontier*, ta phải cập nhật chi phí đường đi (*current cost*)và đường đi (*path*) đến trạng thái đó.

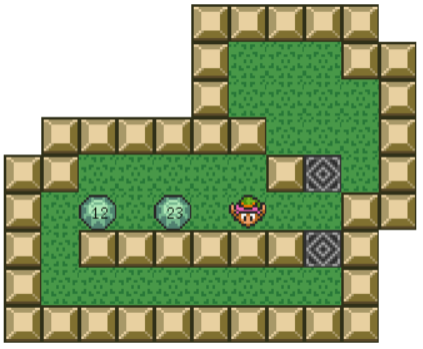
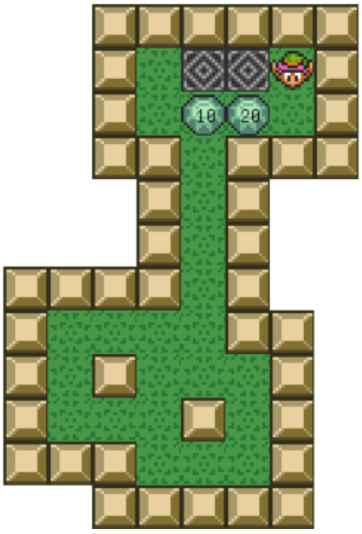
```
frontier = initialize with (initial state, initial ares position, zero path cost, empty path)
explored = empty set

while frontier is not empty:
  state, ares position, move cost, path = frontier→pop smallest path cost
  add state to explored

  if state is solution:
    return path

  for move in four directions:
    new state, move cost, pushed = can move from ares position to (ares position + move) in state

    if new state is null:
      continue
    if new state in explored:
      continue
    if new state is deadlock:
      continue

    if pushed:
      new state path = path + (map move to char) to lowercase
    else:
      new state path = path + (map move to char) to uppercase

    new state cost = current cost + move cost
    new ares position = ares position + move

    if new state in frontier and new state cost < cost of new state in frontier:
        set cost of (new state) in frontier to (new state cost)
        set path of (new state) in frontier to (new state path)
        continue

    frontier←push (new state, new ares position, new state cost, new state path)
```

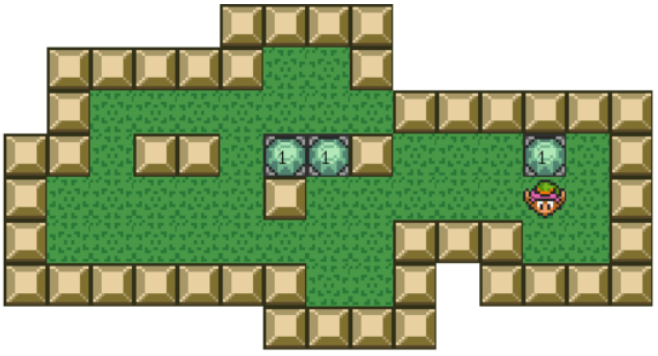
7. Kiểm thử và đánh giá hiệu năng

Dưới đây là 10 trường hợp kiểm thử cho chương trình:

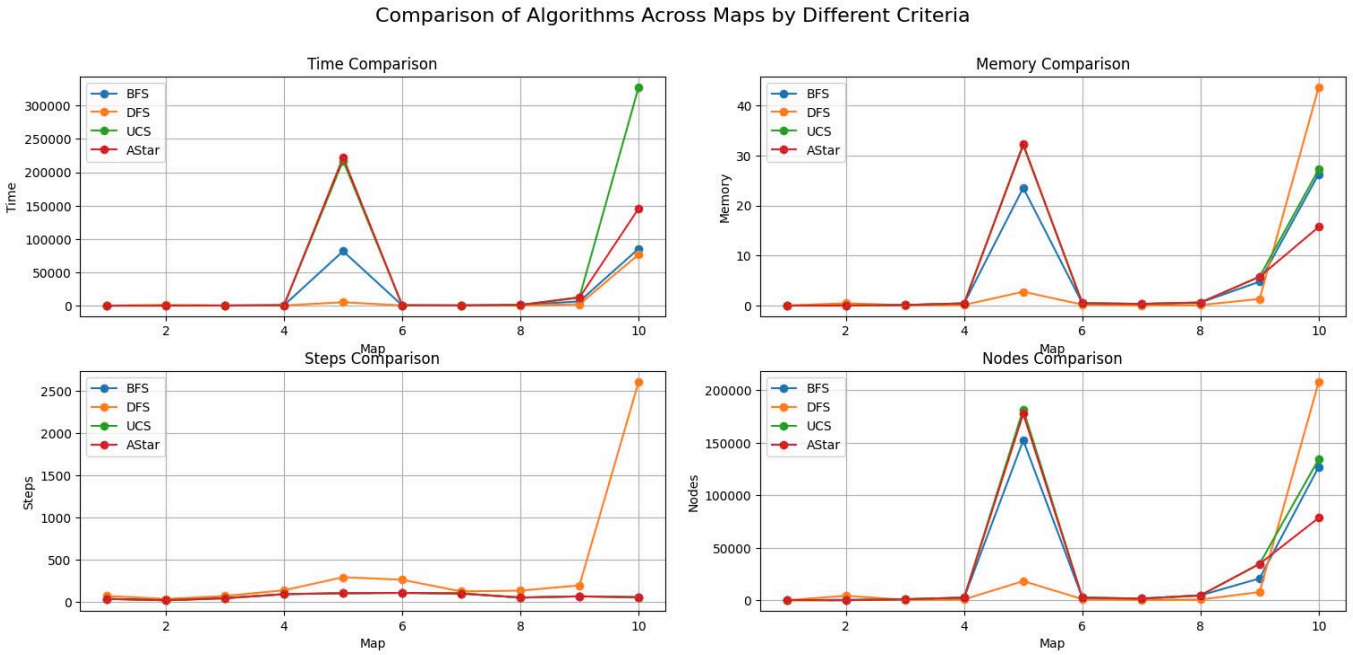


` Chạy chương trình với 10 trường hợp cho 4 thuật toán ta được biểu đồ thống kê thời gian chạy, bộ nhớ sử dụng, số trạng thái đã tạo và số bước của lời giải:



8. Đường dẫn tới video demo chương trình:

https://www.youtube.com/watch?v=2Vmf-EX8Lgw

TÀI LIỆU THAM KHẢO

1. Timo Virkkal (2011), *Solving Sokoban, Sokoban.dk*, https://sokoban.dk/wp-content/uploads/2016/02/Timo-Virkkala-Solving-Sokoban-Masters-Thesis.pdf [Truy cập lần cuối ngày 6/11/2024]

2. Tim Wheeler (2022), *Basic Search Algorithms on Sokoban, Tim Wheeler.com*, https://timallanwheeler.com/blog/2022/01/19/basic-search-algorithms-on-sokoban/ [Truy cập lần cuối ngày 6/11/2024]

3. Lê Trung Hiếu, Đoàn Tây Đô, Nguyễn Chí Trung (2021), *Nhập môn Trí Tuệ Nhân Tạo - Bài tập lớn - Áp dụng giải thuật tìm kiếm vào game SOKOBAN - HCMUT*, https://youtu.be/ev6XPeJ6mnw?si=idtDHH-mciSapFov [Truy cập lần cuối ngày 6/11/2024]

4. Borgar Þorsteinsson (2009), *Sokoban: levels*, *Borget.net*, https://borgar.net/programs/sokoban/#Intro [Truy cập lần cuối ngày 6/11/2024]