**ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH**

**TRƯỜNG ĐẠI HỌC BÁCH KHOA**

**BỘ MÔN: NHẬP MÔN TRÍ TUỆ NHÂN TẠO**

A blue and white logo

Description automatically generated

**BÁO CÁO BÀI TẬP LỚN 1**

**ĐỀ TÀI:**

**ÁP DỤNG GIẢI THUẬT DFS, HILL CLIMBING**

**VÀ GENETIC VÀO CÁC TRÒ CHƠI**

**SUDOKU VÀ MINESWEEPER**

**Giáo viên bộ môn: Vương Bá Thịnh**

**Lớp: L02**

**Thành viên nhóm:**

**Nguyễn Phúc Gia Khiêm 2211573**

**Lê Ngọc Vinh ???????**

**THÀNH PHỐ HỒ CHÍ MINH, THÁNG 4 NĂM 2024**

**MỤC LỤC**

[**A. CƠ SỞ LÝ THUYẾT.** 1](#_Toc164560407)

[**I. Giải thuật DFS (Depth First Search).** 1](#_Toc164560408)

[**II. Giải thuật Hill Climbing dốc nhất.** 1](#_Toc164560409)

[**III. Giải thuật Genetic.** 2](#_Toc164560410)

[**B. CÁC BÀI TOÁN.** 3](#_Toc164560411)

[**I. Trò chơi Sudoku.** 3](#_Toc164560412)

[**1. Giới thiệu trò chơi.** 3](#_Toc164560413)

[**2. Giải thuật Depth First Search.** 3](#_Toc164560414)

[**3. Giải thuật Genetic.** 5](#_Toc164560415)

[**II. Trò chơi Minesweeper.** 10](#_Toc164560416)

[**1. Giới thiệu trò chơi.** 10](#_Toc164560417)

[**2. Quá trình tìm hiểu.** 11](#_Toc164560418)

[**3. Giải thuật Depth First Search.** 11](#_Toc164560419)

[**4. Giải thuật Hill Climbing dốc nhất.** 15](#_Toc164560420)

[**C. TỔNG KẾT.** 18](#_Toc164560421)

[**I. Ưu điểm.** 18](#_Toc164560422)

[**II. Nhược điểm.** 18](#_Toc164560423)

# **A. CƠ SỞ LÝ THUYẾT.**

## **I. Giải thuật DFS (Depth First Search).**

Thuật toán DFS (Depth-First Search) là một thuật toán tìm kiếm trong đồ thị, được sử dụng để duyệt một đồ thị theo chiều sâu. Thuật toán này bắt đầu từ một đỉnh bất kỳ và tiếp tục theo các cạnh để tìm các đỉnh kề, sau đó di chuyển xuống đỉnh kề tiếp theo và tiếp tục quá trình này cho đến khi không còn đỉnh nào mới để thăm. Khi gặp một đỉnh chưa được thăm, thuật toán sẽ tiếp tục đi theo một nhánh nào đó cho đến khi đạt được đỉnh cuối cùng trên nhánh đó trước khi quay lại và tiếp tục từ đỉnh trước đó.

Cách triển khai DFS trên một cây:

1. Tạo một mảng visited để đánh dấu một đỉnh đã được duyệt qua hay chưa.
2. Tạo một stack stk để phục vụ cho quá trình duyệt cây bằng DFS.
3. Bắt đầu từ một đỉnh, ta chèn đỉnh đó vào stk.
4. Ta lấy đỉnh trên cùng của stk ra và xem xét các nút có liên kết với đỉnh đó.
5. Nếu nút con đã ở trong visited thì bỏ qua, nếu nút con chưa ở trong visited ta thêm nó vào visited và chèn nó vào stack.
6. Nếu stack rỗng thì tiến tới bước 7, nếu stack chưa rỗng thì quay lại bước 4.
7. Kết thúc thuật toán.

## **II. Giải thuật Hill Climbing dốc nhất.**

Giải thuật Hill Climbing là một thuật toán tìm kiếm cục bộ (local search algorithm) được sử dụng trong các bài toán tối ưu hóa, nhất là trong các bài toán tìm kiếm không gian trạng thái (state space search). Mục tiêu của thuật toán là di chuyển từ một trạng thái hiện tại đến một trạng thái kế tiếp tốt hơn (tối ưu hơn) trong không gian trạng thái.

Thuật toán Hill Climbing dốc nhất hoạt động như sau:

1. Khởi tạo: Bắt đầu từ một trạng thái bất kỳ trong không gian trạng thái.
2. Đánh giá: Sử dụng một hàm đánh giá (evaluation function) để đánh giá chất lượng của trạng thái hiện tại. Hàm đánh giá thường trả về một giá trị số.
3. Tạo lân cận: Tạo ra các trạng thái lân cận của trạng thái hiện tại bằng cách thay đổi một phần của nó.
4. Lựa chọn lân cận tốt nhất: Đánh giá chất lượng của các trạng thái lân cận được tạo ra trong bước trước bằng cách sử dụng hàm đánh giá. Chọn ra trạng thái lân cận có giá trị đánh giá cao nhất (hoặc thấp nhất, tùy thuộc vào bài toán).
5. Kiểm tra điều kiện dừng: Kiểm tra xem trạng thái lân cận tốt nhất có tốt hơn trạng thái hiện tại không. Nếu có, di chuyển đến trạng thái lân cận tốt nhất và quay lại bước 2. Nếu không, dừng lại và trả về trạng thái hiện tại là giải pháp.
6. Kết thúc: Kết thúc thuật toán và trả về giải pháp tìm được.

Thuật toán Hill Climbing là một giải thuật tham lam (greedy algorithm), do vậy không tránh khỏi một số hạn chế như:

1. Dễ bị mắc kẹt ở điểm cực tiểu cục bộ (local minimum) mà không phải là điểm cực tiểu toàn cục (global minimum).
2. Bước vào trạng thái “cao nguyên” tức là các trạng thái lân cận có giá trị lượng giá bằng với trạng thái hiện tại và không thể đi tiếp.

Phương pháp giải quyết các hạn chế này sẽ được đề cập khi tiếp cận bài toán.

## **III. Giải thuật Genetic.**

Thuật toán genetic là thuật toán phỏng theo quá trình chọn lọc tự nhiên trong sinh học. Ta có một quần thể gồm nhiều cá thể (là các trạng thái của bài toán). Cá thể được đánh giá dựa trên fitness score. Cá thể có fitness cao hơn thì khả năng được chọn làm parent và tạo ra cá thể con cao hơn.

Thuật toán bao gồm 5 giai đoạn:

1. Khởi tạo quần thể ban đầu bằng cách các trạng thái ngẫu nhiên.
2. Tính fitness score của từng cá thể trong quần thể.
3. Chọn lọc (selection): Thông thường người ta lấy fitness score làm trọng số để chọn parent. Tuy nhiên 1 cách làm khác là lấy ngẫu nhiên n - p (n: số cá thể trong quần thể, p < n) cá thể từ quần thể, sau đó lấy p cá thể có fitness score lớn nhất.
4. Crossover: thông thường người ta dùng 2 parent chọn 1 điểm crossover để nối chuỗi và tạo nên chuỗi con.
5. Biến dị (mutation): Thay giá trị 1 vị trí bất kỳ trong chuỗi bằng một giá trị mới.

# **B. CÁC BÀI TOÁN.**

## **I. Trò chơi Sudoku.**

### **1. Giới thiệu trò chơi.**

Sudoku được chơi trên 1 bảng 9x9, được chia thành 9 khối 3x3, điền các giá trị vào ô trống sao cho:

- Không có 2 số giống nhau nằm trên cùng 1 hàng.

- Không có 2 số giống nhau nằm trên cũng 1 cột.

- Không có 2 số giống nhau nằm trong cùng 1 khối.

Đối với bài toán sudoku, nhóm chúng em đã sử dụng 2 phương pháp: depth first search và genetic algorithm.

### **2. Giải thuật Depth First Search.**

**2.1 Cách tiếp cận của nhóm:**

Nhóm sử dụng toán tử bitwise cho bài toán thay vì dùng mảng do tốc độ tính toán nhanh hơn.

Định nghĩa không gian trạng thái của bài toán:

* Không gian trạng thái: dùng 3 dictionary để biểu diễn cột, hàng, khối; dùng 1 số nguyên để biểu diễn các ô có giá trị 0.
* Trạng thái khởi đầu: Duyệt bảng ban đầu, và điền vào các dictionary để biểu diễn số x có tồn tại trên hàng i, cột j, block k hay không.
* Trạng thái kết thúc: Khi số ô trống bằng 0 và bảng sudoku đầu vào là hợp lệ.
* Các bước di chuyển hợp lệ: điền 1 số vào ô trống có index nhỏ nhất sao cho không vi phạm luật chơi.

Với cách tiếp cận của nhóm thì độ sâu của lời giải luôn bằng m, với m là số ô trống.

**2.2 Tiêu tốn bộ nhớ:**

Nhóm sử dụng thư viện chuẩn của python tracemalloc để tính toán tiêu tốn bộ nhớ. Thư viện này giúp ta truy vết cấp phát bộ nhớ.

Ta gọi hàm tracemalloc.get\_traced\_memory(). Hàm này trả về 1 tuple gồm 2 số là bộ nhớ được cấp phát tại thời điểm hiện tại và vào lúc đỉnh điểm. Ta quan tâm đến bộ nhớ được cấp phát vào lúc đỉnh điểm.

Ta sẽ cho k chạy từ 0 tới 81. Với k là số ô bị khuyết trên bảng sudoku. Thực hiện lấy nhiều lần nhóm nhận thấy kết quả tiêu tốn bộ nhớ trả về là như nhau. Nên ứng với mỗi k nhóm sẽ lấy kết quả 1 lần duy nhất.

A graph with a line drawn on it

Description automatically generated

Khi lấy kết quả về sự tiêu tốn bộ nhớ của thuật toán, ta thấy được tiêu tốn bộ nhớ của chương trình tăng theo k. Điều này cũng khá dễ hiểu do độ phức tạp về không gian của thuật toán DFS là b\*m với b là breanching factor, m là độ sâu lớn nhất (và với cách tiếp cận của nhóm thì m = k).

**2.3 Thời gian chạy của thuật toán:**

Thực hiện tương tự như khi tính tiêu tốn bộ nhớ, tuy nhiên, ta cho k chạy từ 0 tới 60. Ứng với mỗi k, ta sinh ra 100 puzzle ngẫu nhiên và cho thuật toán giải.

Lý do cho việc lấy tới k = 60 là do khi k tăng thì nhóm gặp 1 số trường hợp giải khá lâu (> 1 giờ). Điều này có thể giải thích là do thuật toán DFS có độ phức tạp thời gian là b\*d. Với b đươc định nghĩa như phần 1.2 và d là độ sâu của lời giải. Và ở đây d = k. Trong tình huống tệ nhất thì thuật toán của chúng ta có thể phải duyệt trường hợp nên việc thuật toán chạy lâu với 1 số trường hợp cũng là điều dễ hiểu.

Trong hầu hết trường hợp, thuật toán chạy với thời gian khá ngắn (< 0.1 giây và có thể là mili giây).

A graph with a line

Description automatically generated

Từ k = 0 tới k = 52 thời gian trung bình của thuật toán gần như bằng nhau. Thuật toán chạy trong thời gian mili giây. Từ k = 53 trờ đi ta thấy thời gian chạy trung bình có xu hướng tăng nhanh, đó là do thuật toán gặp phải những trường hợp phải giải trong thời gian lâu hơn.

A graph with numbers and a triangle

Description automatically generated

Tuy nhiên khi ta cho k chạy từ 75 tới 81 thì thời gian chạy giảm xuống còn mili giây. Đó là do khi số ô trống quá nhiều thì bài toán trở nên vô cùng đơn giản.

Từ dữ liệu thu thập được về thời gian chạy cũng như tiêu tốn bộ nhớ của thuật toán DFS, ta kết luận rằng thuật toán DFS nhìn chung là hiệu quả đối với bài toán Sudoku.

### **3. Giải thuật Genetic.**

**3.1. Cách tiếp cận của nhóm**

Nhóm sử dụng 1 chuỗi có độ dài 81 để biểu diễn bảng sudoku. (1 trạng thái là 1 chuỗi có độ dài 81 chứa giá tri từ 1 đến 9).

Cách tính fitness: Ban đầu gán fitness = MAX\_FITNESS = 216 (do 1 bảng sudoku với tất cả các ô có cùng 1 giá trị sẽ có fitness = 0). Ta vẫn sử dụng 3 dictionary như DFS.

* Duyệt chuỗi
* Nếu số x tồn tại trên hàng i rồi thì giảm fitness 1 đơn vị, ngược lại đánh dấu x tồn tại trên hàng i.
* Làm tương tự đối với cột và khối.

Thuật toán tìm được nghiệm khi fitness = MAX\_FITNESS

Cho đến nay, nhóm đã thử sử dụng 9 cách tiếp cận đối với thuật toán này. Ngoài ra nhóm đã thử chỉnh kích thước quần thể khác nhau, chỉnh mutation rate... tuy nhiên, không có cách nào có thể tìm được lời giải bài toán sudoku.

**3.1.1 Các cách tiếp cận.**

* + - * Cách 1 – 3:

Chuỗi được tạo thành bằng cách biến ma trận 9x9 ban đầu thành 1 mảng 1 chiều.

* Khởi tạo quần thể ban đầu, ta điền các giá trị ngẫu nhiên vào các ô trống.
* Selection: chọn ngẫu nhiên từ quần thể, lấy fitness làm trọng số (fitness càng cao thì khả năng được chọn cao hơn)
* Crossover:
* Cách 1: chọn 1 điểm ngẫu nhiên bất kỳ (gọi là x). Ta thực hiện crossover bằng cách ghép chuỗi parent1[:x] với parent2[x:]
* Cách 2 - 3: dùng nhiều parent hơn. Cách ghép tương tự cách 1.
* Mutation: duyệt chuỗi.
* Cách 1 - 2: nếu tại vị trí i trong bảng ban đầu là trống thì ta thay giá trị tại vị trí i bằng 1 giá trị hợp lệ (nếu có nhiều giá trị hợp lệ thì chọn bằng random).
  + - * Cách 4:

Cách tạo chuỗi tương tự từ cách 1 tới cách 3. Tuy nhiên, ta sẽ giúp thuật toán bằng cách khi khởi tạo quần thể ban đầu ta sẽ random các giá trị sao cho trên mỗi hàng, không có vị trí nào bị vi phạm.

* Crossover tương tự cách 1.
* Mutation: swap 2 giá trị tại 2 vị trí bất kỳ với điều kiện 2 vị trí đó là 2 vị trí trống trên bảng sudoku ban đầu.
  + - * Cách 5:

Đổi cách crossover, ta chọn m hàng từ parent1 và 9 - m hàng từ parent2

* + - * Cách 6:

Dùng nhiều parent hơn khi crossover, logic tương tự cách 5

* + - * Cách 7:

Dùng mutation rate thay đổi theo thời gian. Nhóm đã tiếp cận theo 2 cách:

* Dùng mutation rate tăng theo thời gian
* Mutation rate cao tại những iteration đầu, tăng độ đa dạng của quần thể.
  + - * Cách 8:

Nhóm đổi cách tạo chuỗi. Chuỗi sẽ được tạo thành bằng cách ghép 9 khối lại với nhau, trong đó mỗi khối 3x3 sẽ được trải phẳng thành mảng 1 chiều 9 phần tử.

Logic cách tạo quần thể ban đầu tương tự cách 4, tạo chuỗi sao cho trong mỗi khối, mỗi giá trị chỉ xuất hiện 1 lần.

Nhóm đổi cách Selection. Thay vì dùng fitness làm trọng số, nhóm sẽ chọn 100\*p% cá thể có fitness cao nhất trong quần thể, 100\*(1-p)% còn lại được chọn ngẫu nhiên.

* Crossover: dùng 2 parent. Cũng tương tự cách 5, ta chọn m khối từ parent1 và 9 - m khối từ parent2.
* Mutation: Không thay đổi so với cách gần nhất.

Với việc thay đổi cách Selection, nhóm nhận thấy rằng chất lượng của quần thể giảm theo thời gian. Nguyên nhân có thể là do 100 \* (1 - p)% còn lại được chọn hoàn toàn ngẫu nhiên dẫn đến việc những cá thể có fitness thấp được chọn làm parent.

* + - * Cách 9:

Đổi cách selection trở lại giống các phiên bản trước.

Các siêu tham số của hàm:

* n\_iters: số lần lặp trước khi kết thúc giải thuật. Thuật toán genetic sẽ dừng khi tìm được nghiệm, để thuật toán dừng trong thời gian hữu hạn ta sẽ thiết lập sao cho thuật toán dừng sau 1 số lần lặp nhất định.
* population\_size: kích thước quần thể, mặc định 1000
* mutation\_rate: tỉ lệ đột biến, mặc định 0.2
  + - * Cách 10 (Hiện tại):

Crossover: ứng với mỗi khối ta chọn ngẫu nhiên 1 của của 1 trong 2 parent

Ta thử cho thuật toán với tham số mặc định, số lần lặp bằng 5000 và kết quả thu được:

A graph of blue and orange lines

Description automatically generated

**3.1.2 Giải thích nguyên nhân thuật toán không tìm được nghiệm**

Dưới đây là một số dự đoán của nhóm về nguyên nhân thuật toán không tìm được lời giải:

* Thuật toán phụ thuộc vào hàm fitness và dễ rơi vào các điểm tối ưu cục bộ. Hàm mutation không đủ tốt để thoát khỏi các điểm tối ưu cục bộ.
* Sudoku là 1 bài toán phức tạp đối với thuật toán này. Các cách tiếp cận khác như DFS, constraint satisfaction problem (CSP)... là phù hợp hơn.

**3.2 Thời gian chạy và tiêu tốn bộ nhớ của thuật toán.**

Vì thuật toán không tìm được nghiệm nên ta sẽ chỉ phân tích thời gian chạy và sự tiêu tốn của bộ nhớ trên câu đố này:

A screen shot of a computer screen

Description automatically generated

**3.2.1 Thời gian chạy của thuật toán**

Ta sẽ chọn các tham số khác là tham số mặc định và xét ảnh hưởng của số lần lặp lên thời gian chạy của thuật toán. Ta lần lượt thử số lần lặp bằng 1000, 2000, 3000, 4000, 5000:

A graph with a line

Description automatically generated

Hiển nhiên, thời gian chạy của thuật toán là thời gian từ lúc bắt đầu đến khi kết thúc thuật toán (và do đó tỉ lệ với số lần lặp).

Bây giờ, chúng ta sẽ để số lần lặp bằng 1000, vẫn giữ các tham số khác là mặc định và xét ảnh hưởng của kích thước quần thể lên thời gian chạy:

A graph with a line

Description automatically generated

Từ đồ thị ta thấy được thời gian chạy của thuật toán tỉ lệ thuận với kích thước quần thể. Điều này khá dễ hiểu do ở mỗi lần lặp ta phải duyệt cả quần thể để tính fitness score và sinh quần thể mới.

Mutation rate thay đổi sẽ không làm tăng đáng kể thời gian chạy của thuật toán. Giả sử mutation\_rate = 1 thì ứng với mỗi cá thể con được tạo ra ta tốn O(1) => cả quần thể sẽ tốn O(population\_size).

Kết luận: độ phức tạp thời gian là O(n\_iters \* population\_size)

**2.2.2 Tiêu tốn bộ nhớ của thuật toán**

Ta sẽ chọn các tham số khác là tham số mặc định và xét ảnh hưởng của số lần lặp lên sự tiêu tốn bộ nhớ của thuật toán. Ta lần lượt thử số lần lặp bằng 1000, 2000, 3000, 4000, 5000:

A graph with a line

Description automatically generated

Thực chất, sự tiêu tốn bộ nhớ của thuật toán không phụ thuộc vào thời gian chạy. Tuy nhiên, vì nhóm có lưu lại thông tin là fitness của cá thể tốt nhất và tệ nhất trong mỗi lần lặp nên ta thấy bộ nhớ là 1 hàm tuyến tính theo số lần lặp. Sự tiêu tốn bộ nhớ thực sự của thuật toán này nằm ở kích thước quần thể:

A graph with a line

Description automatically generated

Tương tự như độ phức tạp thời gian, mutation rate không gây ảnh hưởng nhiều đến độ phức tạp không gian của thuật toán.

Kết luận: Độ phức tạp không gian của thuật toán là O(population\_size + n\_iters) nếu lưu thông tin về fitness score, O(population\_size) nếu trái lại.

## **II. Trò chơi Minesweeper.**

### **1. Giới thiệu trò chơi.**

Trò chơi bao gồm một bản đồ kích thước gồm n dòng và m cột, trong đó có các ô được che đậy (masked) và các ô chữ số đã được hiển thị, chữ số trên một ô thể hiện số bom có xung quanh ô đó (8 ô), theo đó nhiệm vụ của người chơi là quyết định xem các ô được che đậy sẽ là ô trống (không chứa bom) hay ô chứa bom (được thể hiện bằng cờ đỏ) sao cho tất cả các ô số đều được thỏa mãn về số bom xung quanh nó.

Để giải quyết bài toán về trò chơi này, nhóm hiện thực hai giải thuật là DFS (Depth First Search) và Hill Climbing dốc nhất.

Trước khi đi vào các giải thuật sử dụng để giải trò chơi, nhóm xin trình bày cách tổ chức trò chơi trong mã nguồn:

* Các dữ liệu về trạng thái của trò chơi (bản đồ) và các phương thức để tương tác với bản đồ được tổ chức thành class minesweeper\_core.
* Bản đồ của trò chơi được tổ chức bằng mảng hai chiều.
* Các phương thức được tổ chức thành ba loại chính:
  + Phương thức tạo bản đồ và kiểm tra trạng thái của trò chơi: Các phương thức này dùng để tạo ra bản đồ dựa trên tham số đầu vào là số dòng, số cột và số bom, được sự dụng trong nội bộ class. Bản đồ sau khi được tạo ra sẽ được che đậy lại nhằm phục vụ cho việc giải.
  + Phương thức official: Các phương thức được đánh dấu bằng official\_<name> là các phương thức dùng để giải trò chơi, tương tác trực tiếp với bản đồ đã được che đậy sau khi tạo ra.
  + Phương thức dfs\_solution và hill\_climbing\_solution: Sử dụng các phương thức official để giải quyết trò chơi và xuất ra kết quả, có tùy chọn hiển trị quá trình hoặc không.
* Giá trị của các ô trong bản đồ được quy ước như sau:
  + Giá trị -3: Là ô được che đậy.
  + Giá trị -2: Là ô trống.
  + Giá trị -1: Là ô chứa bom.
  + Giá trị gồm số nguyên thuộc đoạn [0; 8]: Thể hiện số bom xung quanh.
* Cần lưu ý rằng số bom ban đầu được đưa vào dưới dạng tham số chỉ phục vụ cho mục đích tạo ra bản đồ, khi bản đồ được che lại chỉ còn các ô số được hiển thị, do vậy kết quả cuối cùng sau khi giải có thể có osos bom không đúng với số bom ban đầu đã truyền vào. Điều này xuất phát từ hạn chế của thuật toán tạo bản đồ không thể tạo ra bản đồ sao cho lời giải là duy nhất, chỉ có thể đảm bảo rằng sẽ có lời giải do đó lời giải cho ra có thể không phải là trạng thái ban đầu của bản đồ (cũng là một lời giải).

### **2. Quá trình tìm hiểu.**

Để tổ chức trò chơi trong mã nguồn cũng như hiện thực giải thuật, nhóm đã thực hiện các bước tìm hiểu như sau:

1. Tìm hiểu cách chơi, luật chơi trên trang web <https://www.puzzle-minesweeper.com/minesweeper-5x5-hard/> bằng cách chơi thử trò chơi ở các độ khó khác nhau.
2. Thiết kế giải thuật nhằm tạo ra bản đồ cho trò chơi, tham khảo từ trò chơi Mine Sweeper gốc cũng như từ suy luận trong quá trình tìm hiểu ở bước 1, kết quả thu được giải thuật tạo bản đồ nhưng vẫn còn hạn chế vì không thể tạo ra bản đồ với lời giải duy nhất, chỉ có thể là giải được.
3. Thiết kế giải thuật DFS và Hill climbing, ở đây nhóm sử dụng tham khảo là mã nguồn JavaScript ở trang web trò chơi, trong đó hiện thực các hàm có chức năng như quét lỗi, xác định ô đã hoàn thành, …. Các hàm này dẫn dắt nhóm đến ý tưởng hiện thực các chức năng trong mã nguồn và logic chạy của trò chơi trong mã nguồn.
4. Kiểm thử và vẽ biểu đồ, nhóm kiểm tra giải thuật nhiều lần bằng các input khác nhau, sau đó tiến hành vẽ biểu đồ và đánh giá biểu đồ, các yếu tố khách quan khác ảnh hưởng đến thời gian chạy và bộ nhớ tiêu tốn.

### **3. Giải thuật Depth First Search.**

**3.1 Cách tiếp cận của nhóm:**

**3.1.1. Các luật di chuyển của trò chơi.**

Để giải trò chơi, nhóm đã xây dựng các luật di chuyển, trong đó có hai nhóm luật chính:

* Luật *move*: Là luật di chuyển thông thường khi mà với tham số đầu vào là tọa độ của một ô và chỉ thị, nếu chỉ thị là ‘l’ (left click) thì ô đó được quyết định là ô trống, nếu là ‘r’ (right click) thì được quyết định là ô có cờ.
* Luật cố định: Là những luật cố định được xây dựng dựa trên cách chơi trò minesweeper trong thực tế, theo đó xét một ô chữ số:
  + Nếu một ô đã được thỏa mãn về số lượng bom xung quanh, ta sẽ mở tất cả các ô được che đậy còn lại là ô trống.
  + Nếu số lượng ô được che đậy còn lại đúng bằng số bom còn thiếu của ô chữ số, ta sẽ đánh dấu toàn bộ các ô che đậy là bom.

Luật di chuyển đóng vai trò trong các bước rẽ nhánh, còn luật cố định giúp giải mã bản đồ nhanh hơn và khiến việc giải trò chơi trở nên “con người” hơn.

**3.1.2. Hiện thực giải thuật DFS.**

Dựa trên khái niệm về giải thuật DFS, các bước cụ thể khi áp dụng DFS vào trò chơi như sau:

1. Áp dụng luật cố định lên các ô số để mở các ô có thể mở.
2. Kiểm tra xem bản đồ đã được hoàn thành chưa, nếu đã hoàn thành thì ta kết thúc, nếu chưa chuyển sang bước 3.
3. Ở bước này ta bắt đầu sinh ra các trường hợp và đi theo nhánh của trường hợp đó, trước tiên chọn một ô số chưa được hoàn thành (vẫn còn ô bị che đậy xung quanh).
4. Gọi r là số bom còn lại cần đặt để hoàn thành ô số đã chọn (là hiệu của số trên ô và số bom xung quanh đã được mở), ta chọn r ô được che đậy ngẫu nhiên quanh ô số và đánh dấu chúng là bom.
5. Áp dụng luật cố định lên ô vừa chọn và tất cả các ô số nhằm mở rộng bản đồ hết mức có thể.
6. Kiểm tra bản đồ, nếu bản đồ có lỗi, trả lại trạng thái bản đồ cũ và quay trở lại bước 4 và chọn một tổ hợp khác, nếu bản đồ không có lỗi nhưng chưa được hoàn thành, ta không trả lại trạng thái cũ mà quay lại bước 3, nếu bản đồ đã được hoàn thành ta kết thúc thuật toán. Trong quá trình kiểm tra, nếu phát hiện một ô bị che đậy nhưng xung quanh không có ô số nào thì ta sẽ cho nó là ô trống vì dù nó là ô trống hay ô có bom cũng không ảnh hưởng đến sự hợp lệ của lời giải, trường hợp này xảy ra vì khi số lượng bom đầu vào lớn, thuật toán tạo bản đồ có thể ngẫu nhiên xếp một bom được vây quanh bởi 8 quả bom khác và khi che đậy thì nó sẽ trở thành một ô bị che đậy mà xung quanh không có ô số nào.

Nhìn chung thuật toán của chúng ta đúng trên tinh thần giải thuật DFS, tuy nhiên để tối ưu cho thuật toán cũng như đưa các yếu tố “con người” vào trò chơi, nhóm quyết định ở bước 3, thay vì chọn ngẫu nhiên một ô số ta sẽ có cách chọn như sau:

*Gọi v là tỉ lệ giữa số bom còn thiếu xung quanh một ô số và số ô được che đậy còn lại xung quanh ô đó, ta sẽ ưu tiên chọn ô có v cao nhất để thực hiện bước 3. Nếu có các ô có cùng giá trị v, ta chọn ô có số nhỏ nhất. Điều này được lấy ý tưởng từ cách chơi thực tế, khi mà phải chia trường hợp con người thường cố gắng sao cho tỉ lệ đúng là cao nhất và số trường hợp cần thử là ít nhất. Việc lựa chọn tỉ lệ v thay vì tổ hợp các cách để đặt bom (số lượng tổ hợp r ô được đề cập ở bước 4) làm giá trị ưu tiên nhằm giảm tải khối lượng tính toán vì tính toán các tổ hợp rất tốn thời gian.*

**3.2. Kết quả thu được.**

a) Thuật toán đã giải được trò chơi Mine Sweeper đã đề ra.

A black and white screen with white lines and numbers

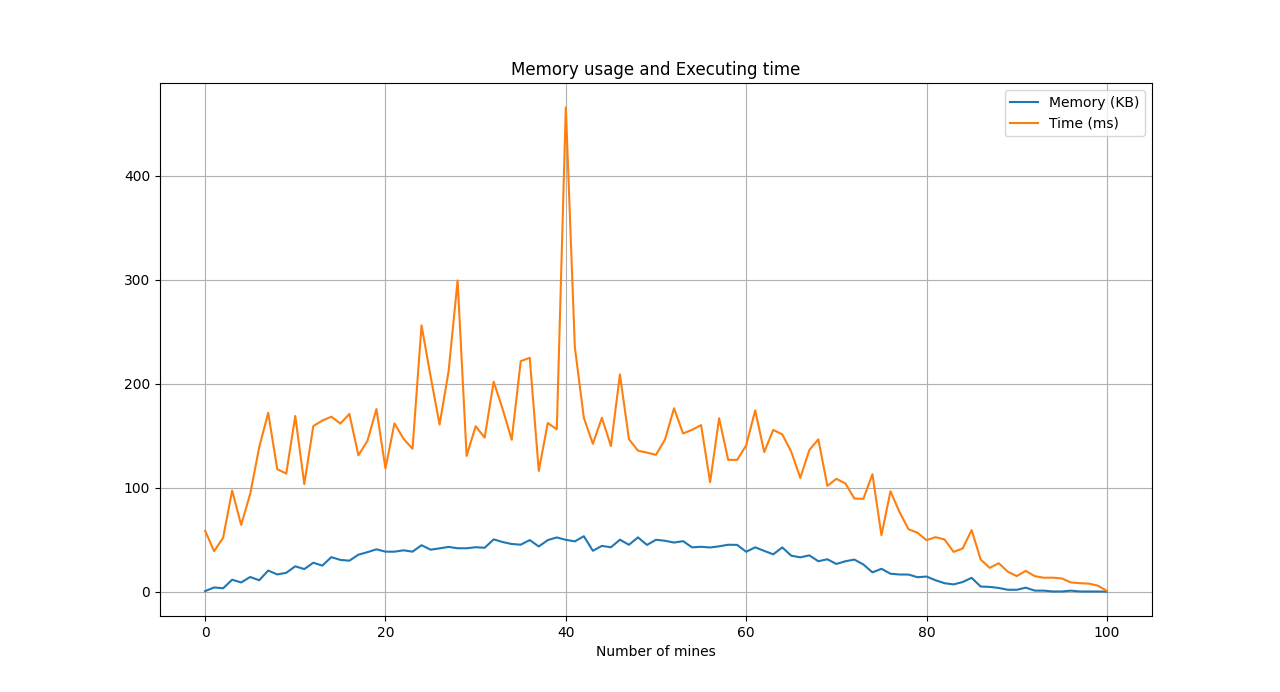
Description automatically generated

*Kết quả sau khi chạy thuật toán với bản đồ kích thước 10 x 10 và số bom khởi tạo là 50*

Ta thấy rằng dù số bom khởi tạo là 50 nhưng khi giải ra thì số bom chỉ là 41 dù rằng lời giải này vẫn hợp lệ, đó là do hạn chế không thể tạo ra bản đồ có lời giải duy nhất trong phần tạo bản đồ.

b) Thời gian chạy và bộ nhớ tiêu tốn của thuật toán.

Nhóm dùng phép thử với bản đồ có kích thước 10 x 10, và cho số bom khởi tạo tăng liên tục từ 1 đến 100, mỗi số bom khởi tạo sẽ chạy thuật toán 5 lần rồi rút ra thời gian chạy trung bình cũng như bộ nhớ tiêu tốn trung bình để khảo sát sự thay đổi của thời gian chạy thuật toán cũng như sự tiêu tốn bộ nhớ, kết quả được trình bày ở biểu đồ sau:



*Biểu đồ thể hiện sự biến động của thời gian chạy và sự tiêu tốn bộ nhớ của thuật toán DFS biến động theo số bom khởi tạo.*

Từ biểu đồ có thể đưa ra các nhận xét sau về thời gian chạy:

* Thời gian chạy của thuật toán biến động khá thất thường tuy nhiên vẫn có thể nhận thấy thời gian chạy đặc biệt lâu hơn ở số bom từ 20 đến dưới 60. Việc thời gian chạy biến động do tính chất ngẫu nhiên ở bước 4 trong hiện thực của DFS, khi chọn sai nhánh đi việc quay trở lại cũng tốn tương đối thời gian và bộ nhớ, với số bom từ 15 đến 50 thì lời bản đồ sẽ cố số ô số ở mức trung bình, không quá nhiều và không quá ít đặt ra nhiều yêu cầu hơn cho vị trí đặt bom nên khả năng chọn sai và quay lại cao hơn dẫn đến thời gian chạy lâu hơn.
* Ở các mức bom thấp, các ô số sẽ chiếm vị trí rất ít và đa phần là số 0 (sẽ mở ngay lập tức 8 ô trống xung quanh khi dùng luật cố định) do vậy thời gian thuật toán chạy sẽ ít hơn.
* Ở các mức bom từ 60 đến gần 100 thời gian chạy sẽ giảm dần do các ô số lúc này sẽ có giá trị lớn (khả năng xuất hiện ô có số 8 cũng cao) nên việc mở bản đồ sẽ dễ dàng hơn.
* Ở mức bom 100 thì thời gian chạy là 0 vì thuật bản đồ đã hoàn thành ngay từ đầu.

Từ biểu đồ có thể đưa ra các nhận xét sau về sự tiêu tốn bộ nhớ:

* Sự tiêu tốn bộ nhớ nhìn chung cũng sẽ biến động giống như thời gian (thời gian tăng thì bộ nhớ tăng) vì khi đó nghĩa là ta phải rẽ nhánh nhiều, mỗi lần rẽ nhánh ta phải lưu một bản sao của trạng thái trước đó dẫn đến sự tăng về bộ nhớ, tuy nhiên sẽ có những trường hợp bộ nhớ sử dụng giảm nhưng thời gian chạy thì tăng, điều này đến từ việc thời gian còn phụ thuộc vào việc quay lại, mà việc quay lại sẽ không làm tăng bộ nhớ và các yếu tố khách quan như thiết bị cũng có thể làm tăng thời gian chạy mà không tăng bộ nhớ.
* Bộ nhớ tiêu tốn cũng tăng dần từ số bom 0 đến dưới 60, sau đó thì giảm dần vì sự đơn giản trong việc tìm lời giải đã trình bày ở phần thời gian.

### **4. Giải thuật Hill Climbing dốc nhất.**

**4.1 Cách tiếp cận của nhóm:**

**4.1.1. Các luật di chuyển của trò chơi.**

Các luật của trò chơi vẫn được hiện thực như giải thuật DFS.

**4.1.2. Hàm lượng giá và giải quyết các trường hợp thuật toán không thể tìm được đường đi.**

Hàm lượng giá: Hàm lượng giá mà nhóm sử dụng là số các ô số đã được hoàn thành ở trên bản đồ.

Giải quyết các trường hợp thuật toán không thể tìm trạng thái tiếp theo:

* Chấp nhận các trạng thái mới mà giá trị lượng giá bằng trạng thái hiện tại để giải quyết trường hợp “cao nguyên”.
* Cách lấy hàm lượng giá như trên sẽ luôn cho ra trạng thái mới có giá trị lượng giá cao hơn hoặc bằng trạng thái hiện tại nên không gây ra tình trạng tối ưu cục bộ.
* Nếu đường đi theo trạng thái tốt nhất dẫn đến một lời giải sai, ta sẽ quay lại và tiếp tục với đường đi của trạng thái có giá trị lượng giá tốt thứ hai. Việc này giúp ta giải quyết được hạn chế của giải thuật tham lam.

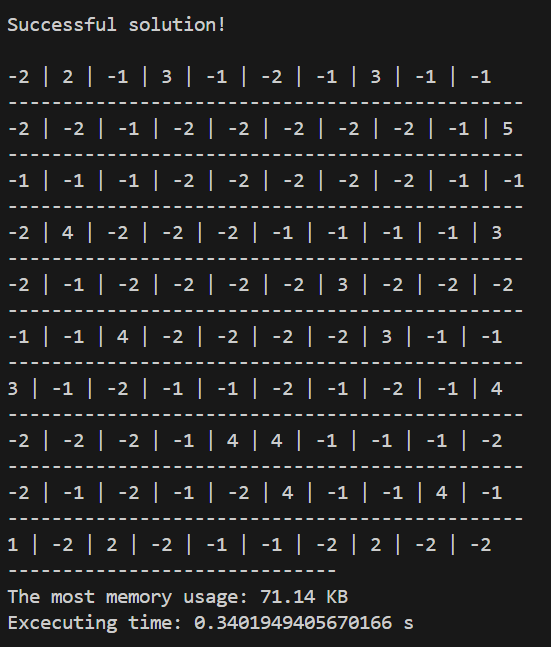
**4.1.3. Hiện thực giải thuật Hill Climbing dốc nhất.**

Dựa trên khái niệm về giải thuật Hill climbing dốc nhất, các bước cụ thể khi áp dụng vào trò chơi như sau:

1. Áp dụng luật cố định lên các ô số để mở các ô có thể mở.
2. Kiểm tra xem bản đồ đã được hoàn thành chưa, nếu đã hoàn thành thì ta kết thúc, nếu chưa chuyển sang bước 3.
3. Ở bước này ta bắt đầu sinh ra các trường hợp và đi theo nhánh của trường hợp đó, trước tiên chọn một ô số chưa được hoàn thành (vẫn còn ô bị che đậy xung quanh), cách chọn được xác định giống như DFS dựa theo tỉ lệ v.
4. Gọi r là số bom còn lại cần đặt để hoàn thành ô số đã chọn (là hiệu của số trên ô và số bom xung quanh đã được mở), ta xét tất cả các tổ hợp r ô được lấy từ các ô che đậy còn lại. Ta giả định trường hợp quyết định các tổ hợp đó là bom, mỗi tổ hợp sẽ sinh ra một trạng thái mới, ta dùng luật cố định để mở rộng tại vị trí ô ta đã xác định ở bước 3 và các ô số lân cận bị ảnh hưởng. Với mỗi trạng thái mới đã được mở rộng ta xác định hàm lượng giá và sắp xếp các tổ hợp ở trên theo mức lượng giá tương ứng giảm dần (tổ hợp nào cho ra lượng giá cao hơn được xếp cao hơn). Cuối cùng ta sẽ chọn tổ hợp có lượng giá cao nhất và quyết định các ô đó là bom.
5. Áp dụng luật cố định lên ô vừa chọn và tất cả các ô số nhằm mở rộng bản đồ hết mức có thể.
6. Kiểm tra bản đồ, nếu bản đồ có lỗi, trả lại trạng thái bản đồ cũ và quay trở lại bước 4 và chọn một tổ hợp có giá trị lượng giá lớn kế tiếp, nếu bản đồ không có lỗi nhưng chưa được hoàn thành, ta không trả lại trạng thái cũ mà quay lại bước 3, nếu bản đồ đã được hoàn thành ta kết thúc thuật toán. Trong quá trình kiểm tra, nếu phát hiện một ô bị che đậy nhưng xung quanh không có ô số nào thì ta sẽ cho nó là ô trống vì dù nó là ô trống hay ô có bom cũng không ảnh hưởng đến sự hợp lệ của lời giải, trường hợp này xảy ra vì khi số lượng bom đầu vào lớn, thuật toán tạo bản đồ có thể ngẫu nhiên xếp một bom được vây quanh bởi 8 quả bom khác và khi che đậy thì nó sẽ trở thành một ô bị che đậy mà xung quanh không có ô số nào.

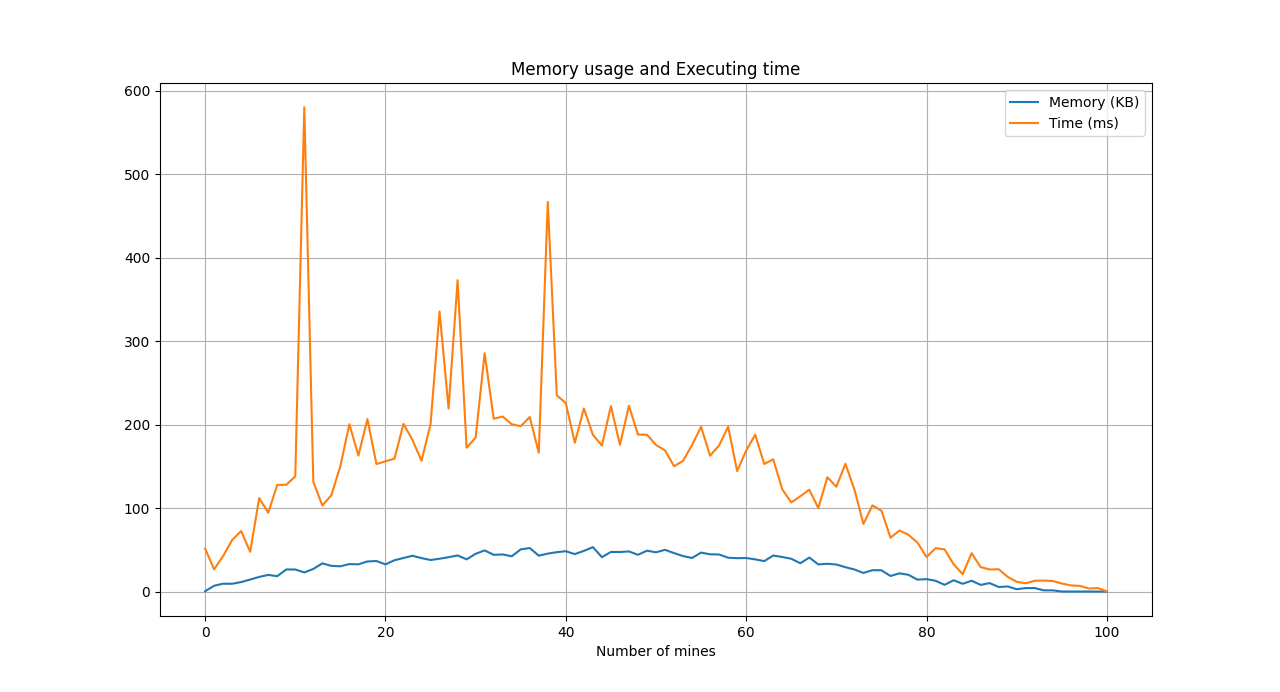
**4.2. Kết quả thu được.**

a) Thuật toán đã giải được trò chơi Mine Sweeper đã đề ra.



*Kết quả sau khi chạy thuật toán với bản đồ kích thước 10 x 10 và số bom khởi tạo là 50*

b) Thời gian chạy và bộ nhớ tiêu tốn của thuật toán.

Nhóm dùng phép thử với bản đồ có kích thước 10 x 10, và cho số bom khởi tạo tăng liên tục từ 1 đến 100, mỗi số bom khởi tạo sẽ chạy thuật toán 5 lần rồi rút ra thời gian chạy trung bình cũng như bộ nhớ tiêu tốn trung bình để khảo sát sự thay đổi của thời gian chạy thuật toán cũng như sự tiêu tốn bộ nhớ, kết quả được trình bày ở biểu đồ sau:

*Biểu đồ thể hiện sự biến động của thời gian chạy và sự tiêu tốn bộ nhớ của thuật toán Hill Climbing dốc nhất biến động theo số bom khởi tạo.*

Từ biểu đồ có thể đưa ra các nhận xét sau về thời gian chạy: Thời gian chạy của thuật toán biến động khá thất thường tuy nhiên vẫn có thể nhận thấy thời gian chạy đặc biệt lâu hơn ở số bom từ 20 đến dưới 60 và giảm ở dần ở phần còn lại, điều này được lí giải tương tự như giải thuật DFS. Tuy nhiên ta thấy rằng tồn tại nhiều giá trị cao bất thường trong biểu đồ (có điểm chênh lệch hơn 400 ms so với các điểm xung quanh) trên khác với biểu đồ của DFS khi mà các giá trị không có chênh lệch quá xa như vậy, điều này là do sự trả giá trong việc tính toán các trường hợp lượng giá ở bước 4, khi ta đệ quy quá trình hoặc quay lại và rẽ nhánh đều sẽ tốn nhiều tài nguyên và thời gian hơn. Đây là tính chất của giải thuật tham lam khi ta đi đúng hướng thì sẽ nhanh hơn nhưng với chi phí tính toán cao hơn thì nếu phải quay lại sẽ tốn thời gian hơn.

Từ biểu đồ có thể đưa ra các nhận xét sau về sự tiêu tốn bộ nhớ:

* Sự tiêu tốn bộ nhớ nhìn chung cũng sẽ biến động giống như thời gian (thời gian tăng thì bộ nhớ tăng) vì khi đó nghĩa là ta phải rẽ nhánh nhiều, mỗi lần rẽ nhánh ta phải lưu một bản sao của trạng thái trước đó dẫn đến sự tăng về bộ nhớ, tuy nhiên sẽ có những trường hợp bộ nhớ sử dụng giảm nhưng thời gian chạy thì tăng, điều này đến từ việc thời gian còn phụ thuộc vào việc quay lại, mà việc quay lại sẽ không làm tăng bộ nhớ và các yếu tố khách quan như thiết bị cũng có thể làm tăng thời gian chạy mà không tăng bộ nhớ.
* Bộ nhớ tiêu tốn cũng tăng dần từ số bom 0 đến dưới 60, sau đó thì giảm dần vì sự đơn giản trong việc tìm lời giải đã trình bày ở phần thời gian.

# **C. TỔNG KẾT.**

## **I. Ưu điểm.**

Thông qua bài tập lớn này, nhóm đã đạt được một số kết quả sau:

* Củng cố kiến thức về giải thuật blind search và heuristic search.
* Hiện thực được các giải thuật blind search DFS và heuristic search Hill Climbing và Genetic. Đối với giải thuật Hill Climbing thì thể hiện được phần nào ưu thế về mặt thời gian ở một vài trường hợp tuy nhiên không thực sự rõ ràng.
* Rèn luyện kĩ năng tìm và đọc mã nguồn.
* Rèn luyện kĩ năng suy luận và sáng tạo trong giải quyết vấn đề về tạo bản đồ trò chơi và thiết kế OOP, thiết kế logic của giải thuật.
* Rèn luyện kĩ năng lập trình bằng Python (OOP, Memory tracing, random, …).
* Rèn luyện kĩ năng làm việc nhóm, sử dụng Github để chia sẻ mã nguồn.
* Rèn luyện kĩ năng phân tích biểu đồ, củng cố các kiến thức về xác suất thống kê.

## **II. Nhược điểm.**

* Thiết kế bản đồ trò chơi chưa tốt, chưa tạo ra được bản đồ có lời giải duy nhất.
* Chưa tối ưu được giải thuật heuristic:
  + Không tìm được lời giải ở trò chơi Sudoku.
  + Việc chọn và đánh giá hàm lượng giá tương đối tốn tài nguyên dẫn tới giải thuật chưa thể hiện được ưu thế so với DFS một cách rõ ràng.