Project 01: Cryptarithmetic Problem

I. Thông tin đồ án

1. Nhóm

- **Thành viên**

  Võ Quang Huy: 19127425
  Hồ Nhật Linh: 19127652
  Nguyễn Ngọc Anh Khoa: 19127445

- **Công việc**

Công việc	Participant	Mức độ hoàn thành
Xây dựng và phân tích Global search (Backtracking search)	VÕ QUANG HUY	Giải quyết thành công 4 level của bài toán. Không gặp lỗi với những test cases đã tạo.
Xây dựng và phân tích Local Search (Beam search)	HỒ NHẬT LINH	Giải quyết thành công 4 level của bài toán. Không gặp lỗi với những test cases đã tạo.
Generate test cases	NGUYỄN NGỌC ANH KHOA	Xây dựng được ít nhất 5 test cases cho mỗi level.

2. Đồ án

Ngôn ngữ: Python 3
Môi trường thực thi: Visual studio code
Cấu trúc folder nộp:

Report.pdf: báo cáo chi tiết
Test Cases:

- **level1.txt:** chỉ có hai số hạng +/- với nhau
- **level2.txt:** nhiều hơn 2 số hạng +/- với nhau
- **level3.txt:** level 2 sử dụng thêm dấu ngoặc đơn "()"
- **level4.txt:** nhiều số hạng * với nhau

Backtracking:

- **input.txt:** input hiện tại muốn test
- **main.py:** dùng để chạy chương trình
- **Utils.py:** chứa một số hàm tiện ích như load file, chuyển đổi infix sang postfix, chuyển đổi string sang số,...

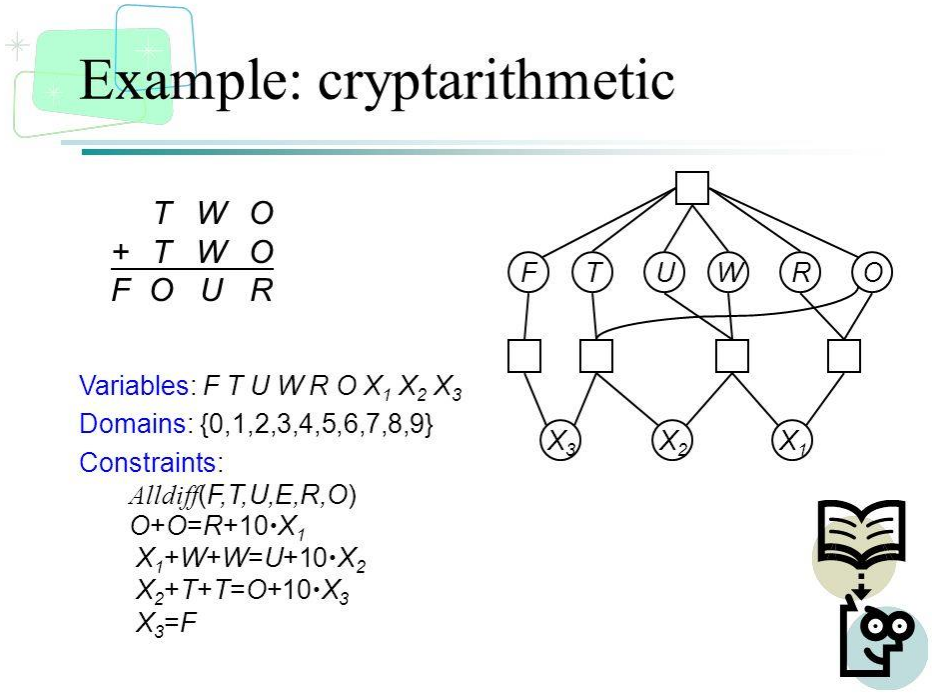
LoalBeam

- **beamSearch.py:** hàm local beam search
- **expression.py**: chứa lớp Expression
- **fileUtilities.py**: chứa một số hàm hỗ trợ việc load file
- **input.txt**: input đang muốn test
- **main.py**: dùng để chạy chương trình
- **operand.py:** chứa lớp Operand
- **priorityQueue.py**: chứa lớp priority queue tự dựng
- **state.py:** chứa lớp State giữ thông tin 1 trạng thái
- **storeTotal.py:** chứa lớp lưu giá trị các toán hạng

- **twoWayDic.py**: chứa lớp dictionary 2 chiều
- **utilities.py**: chứa các hàm làm việc với toán hạng và toán tử (tính giá trị biểu thức, tạo biểu thức ngẫu nhiên,..)

II. Phân tích bài toán

Cryptarithmetic có lẽ không còn là vấn đề xa lạ trong chủ đề CSP. Ngay từ khi tiếp cận bài toán, ta đã nhận ra "sự CSP" với rất nhiều các ràng buộc, đặc biệt là ràng buộc Alldiff giữa các biến.



(mô tả các ràng buộc trong 1 bài toán đơn giản. Nguồn: https://slideplayer.com/slide/9196901/)

Mặc dù ta đã biết đến "người anh em thân thiết" của CSP là backtracking search, với khả năng phục hồi trạng thái và quay lui. Tuy nhiên, bài toán này có rất nhiều chiến lược tiếp cận thú vị. Để có thể đánh giá được bài toán 1 cách toàn diện, trong đồ án này, Nhóm sẽ xây dựng cả 2 chiến lược tìm kiếm là Global search và Local search, với những phân tích điểm yếu, điểm mạnh của từng loại.

Về Global search, Nhóm xây dựng thuật toán Backtracking search (BS). Không chỉ nhờ sự "nổi tiếng" của nó mà đây là chiến lược rất thích hợp cho bài toán này. Ta điểm qua 1 "đối thủ" với cơ chế tìm kiếm trái ngược với BS là Breadth first search và các biến thể của nó (A*, GBFS), bài toán này có tối đa 10! (3628800) trạng thái, với mỗi trạng thái ta cần 1 mảng 10 phần tử cho các biến (có dung lượng 136 bytes trong python3), và 1 mảng 10! phần tử cho explored set, do đó ta cần tối đa 940 MB cho các thuật toán tìm theo chiều rộng, rõ ràng bài toán nhỏ này của chúng ta không cần thiết tiêu tốn nhiều đến vậy. Trong khi đó, BS lại chỉ duy trì 1 đường đi hiện tại trên cây tìm kiếm, hơn nữa khả năng phục hồi trạng thái và quay lui của nó sẽ giúp rẽ đi các nhánh rất tốt mà không tốn quá nhiều chi phí.

Về Local search, dù cho sự gọn nhẹ và nhanh của nó thì local search vẫn rất "nguy hiểm", do đây là bài toán cũng chứa rất nhiều local maxima. Do vậy, Nhóm đã triển khai kết hợp 1 vài thuật toán của Local search để tạo mọi điều kiện cho nó có thể đi đến được trạng thái kết quả. Lõi chỉnh của thuật toán vẫn là Hill Climbing với sự "tham lam" hữu ích của nó, và 1 vài biến thể để hỗ trợ như Random-restart, Beam search, Tabu search.

Đồ án này xây dựng Local search là thuật toán chính, Backtrack chỉ nhằm mục đích so sánh và chỉ ra điểm yếu, điểm mạnh.

III. Xây dựng thuật toán

1. Backtracking: đệ quy

a. Ý tưởng

Xây dựng dựa theo cách tính 1 biểu thức bằng cách dựng các hàng toán hạng và đứng chính xác theo vị trí giá trị của từng con số chục theo chục, đơn vị theo đơn vị,..). Ta tiến chạy thuật toán từ cột phải nhất qua cột trái nhất. Ở mỗi cột ta tiến hành gán giá trị cho các biến của cột đó, nếu các biến vừa gán thỏa mãn cột hiện tại, ta nhảy qua cột tiếp theo để tiếp tục gán, còn nếu không tìm được phép gán hợp lí cho cột hiện tại thì ta phục hồi lại trạng thái trước khi gán trong cột này và nhảy về cột trước để gán lại.
two two four

Ta xét thử ví dụ dưới đây để hiểu rõ hơn về thuật toán:

```
 TWO
   +
 TWO
   =
FOUR
```

Đầu tiên bắt đầu từ cột phải nhất (O O R), ví dụ ta gán O = 1 và R = 2. Ta được giá trị như dưới:

```
 TW1
   +
 TW1
   =
F1U2
```

Ví dụ ta gán W = 2, U = 4. Ta được giá trị như dưới:

```
 T21
   +
 T21
   =
F142
```

Rõ ràng, ở cột này, ta không thể tìm được bộ số thích hợp để gán nữa, ta nhảy trở về cột trước (2 2 4) để gán lại giá trị cho chúng. Nếu cột (2 2 4) tiếp tục không thỏa, ta lại nhảy về (1 1 2). Đồng thời mỗi lần quay ngược ta cũng gỡ những giá trị cho những biến đã gán hiện tại (phục hồi lại trạng thái).

b. Triển khai

b.1) Cách gán cho các biến:

Các biến được gán theo từng cột từ phải sang trái.

Nếu như đang gán các biến đang ở số hạng:

- Nếu biến đã được gán → <u>đệ quy</u> dòng tiếp theo (biến của số hạng tiếp theo).
- Sử dụng vòng lặp for trong domain để gán giá trị cho các biến dựa trên điều kiện mọi biến phải khác giá trị.
- Đối với những biến nằm ngoài cùng bên trái sẽ thêm điều kiện ≠ 0.
- Sau khi gán thành thành công → <u>đệ quy</u> dòng tiếp theo (biến của số hạng tiếp theo).

Nếu như đang gán các biến kết quả (sau khi gán các biến ở số hạng), cần thực hiện phép tính và lấy kết quả theo cột tương ứng.

- Nếu biến này đã được gán trước đó và khớp với kết quả của phép tính thì đệ quy qua cột tiếp theo. Nếu không khớp → return False.
- Nếu chưa được gán trước đó thì kiểm tra kết quả phép tính đã được biến nào sử dụng chưa. Nếu chưa biến nào sử dụng → thực hiện gán và đệ quy cột tiếp theo. Ngược lại, đã có biến sử dụng → return False.

Mọi kết quả trả về nếu True → tiếp tục trả về True. Nếu False thì trả các giá trị đã gán lại cho domain và chọn giá trị khác.

b.2) Cách tính toán:

Sử dụng chuỗi postfix để tính toán. Tính toán theo từng cột.

Để đảm bảo có thể tính toán được (các biến của các số hạng ở cột tương ứng phải được gán hết).

Thuật toán đã tránh sử dụng biến chứa số dư (carry). Xem ví dụ bên dưới.

Ví dụ: TWO+TWO=FOUR.

- Cột phải ngoài cùng: O + O = R → (O = 1, R = 2).
- Cột tiếp theo: WO + WO = UR → (W=3, U = 6). Thuật toán không tính theo từng cột một mà sử dụng thêm các cột cũ để tránh không phải lưu biến carry đồng thời giảm độ phức tạp khi nhân hai số.

2. Hill climbing search

a. Ý tưởng

Đầu tiên để có thể xây dựng thuật toán, ta cần phải xác định giá trị heuristic của 1 trạng thái. Tương tự như trong định nghĩa của heuristic, khoảng cách từ 1 trạng thái đến mục tiêu. Ta xây dựng hàm heuristic bằng cách tính tất cả toán hạng hiện tại trong biểu thức (kể cả kết quả), giá trị heuristic này càng gần 0 chứng tỏ nó càng tốt.

Ví dụ, ta có một biểu thức đơn giản: A + B = CD

- Trạng thái 1: Ta gán A = 1, B = 2, C = 3, D = 4. Lúc này heuristic của nó sẽ là abs(1+2-34) = 31. Ta dùng trị tuyệt đối là do ta đang xét xem trạng thái nào gần 0 hơn.
- Trạng thái 2: Ta gán A = 9, B = 8, C = 1, D = 5. Lúc này, heuristic của nó sẽ là abs(9+8-15)=2
- Lúc này, ta nói trạng thái 2 tốt hơn 1 do nó gần 0 hơn.

Tiếp theo, ta xác định như thế nào là trạng thái kế thừa (successors) của 1 trạng thái ban đầu. Ở đây ta ngầm định rằng, 1 trạng thái sẽ là 1 trạng thái kế thừa nếu nó có sự khác biệt nhiều nhất 1 phép gán hoặc có sự tráo đổi giá trị của 1 cặp biến so với cha của nó. Ví dụ, ta có 1 trạng thái như sau: {A: 1, D: 2, E: 3}, lúc này, nó sẽ các trạng thái con là {A: 5, D: 2, E: 3} (gán A = 5 so với A = 3 ở trạng thái ban đầu), {A: 2, D: 1, E: 3} (tráo đối giữa A và D…

Cách thức tính giá trị của 1 toán hàng cũng khá đơn giản, từ trạng thái với các biến đã gán, ta tính toán giống như cách ta làm trong hệ số thập phân với số thứ i sẽ nhân với 10^i. Ví dụ, ta có toán hạng ABC với trạng thái {A: 1, B: 2, C: 3}, ta tính được giá trị của nó là $3 * 10^0 + 2 * 10^1 + 1 * 10^2 = 123$

Ta tiến hành xây dựng lõi của thuật toán là Hill Climbing. Đầu tiên ta gán ngẫu nhiên 1 trạng thái nào đó và xem đây là trạng thái gốc. Để tìm ra successor, ta lấy ngẫu nhiên 1 biến và cố gắng gán tất cả giá trị (0 – 9, kể cả những giá trị đã bị biến khác chọn) cho nó và lấy giá trị đầu tiên làm cho nó có heuristic nhỏ nhất. Ví dụ, ta có 1 biểu thức A + B = C, với trạng thái {A: 1, B: 4, C: 3}, ta chọn ngẫu nhiên biến B, lúc này ta lần lượt gán các giá trị từ 0 đến 9 cho B, ta thấy giá trị B = 2 mang heuristic = 0 (nhỏ nhất) đem lại

kết quả thuận lợi nhất, nên ta chọn nó làm successor. Điều kiện dừng ở đây là thời gian, nếu thời gian chạy vượt quá thời gian giới hạn ban đầu thì thuật toán cho rằng không có giải pháp. Ta chạy thử thuật toán 100 lần với input TOW + TWO = FOUR

```
running time:  0.001994609832763672
running time:  0.0
running time:  0.0
running time:  0.0
running time:  0.000978708267211914
running time:  0.0029926300048828125
running time:  0.0
running time:  0.0009818077087402344
running time:  0.0009779930114746094
running time:  0.0009641647338867188
running time:  0.0
running time:  0.0009961128234863281
running time:  0.0
running time:  0.0010001659393310547
14
```

(Những lần thành công sẽ in ra thời gian chạy, có 14 lần thành công)

Có thể thấy, tỉ lệ thành công khá thấp, chỉ ra 14 kết quả ở 100 lần chạy. Rõ ràng Hill climbing hiện tại vẫn chưa đủ để giải quyết bài toán, việc rơi vào local maxima là rất cao. Do đó, ta sẽ trạng bị cho nó thêm biến thể Random-restart hill climbing, để nó có thể bắt đầu lại với những trạng thái ngẫu nhiên mới nếu sau số bước tối đa (do ta tự quy định) mà vẫn không tìm ra giải pháp. Cách triển khai cũng rất đơn giản, ở bên ngoài Hill climbing ta lồng thêm 1 vòng lặp để tạo trạng thái ngẫu nhiên mới sau mỗi lần thất bại. Điều kiện dừng của thuật toán này cũng là yếu tố thời gian, sau thời gian tối đa mà vẫn không ra thì có giải pháp. Ta chạy thử thuật toán 100 lần với input TWO + TWO = FOUR

```
running time, random times:  0.0 ,  14
running time, random times:  0.37903809547424316 ,  16
running time, random times:  0.284207820892334 ,  9
running time, random times:  0.17752456665039062 ,  5
running time, random times:  0.0019941329956054688 ,  20
running time, random times:  0.44183778762817383 ,  7
running time, random times:  0.09073925018310547 ,  12
running time, random times:  0.35405397415161133 ,  26
running time, random times:  0.050863027572631836 ,  16
98
```

(Những lần thành công sẽ in ra thời gian chạy và số lần đã restart, có 98 lần thành công)

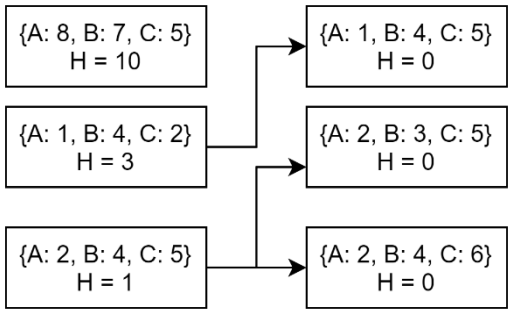
Có thể thấy, tỉ lệ thành công đã được nâng lên rất nhiều, gần như sẽ luôn thành công nếu ta đặt giới hạn thời gian cao hơn (ở đây ta đã đặt 0.5s) đối với input đơn giản này. Ta sẽ tiến hành thử với 1 input thách thức hơn, đương nhiên ta cũng kéo thời gian giới hạn lên 5s:

SO+MANY+MORE+MEN+SEEM+TO+SAY+THAT+THEY+MAY+SOON+TRY+TO+STAY+AT+HOME+SO+AS+TO+SEE+OR+HEAR+THE+SAME+ONE+MAN+TRY+TO+MEET+THE+TEAM+ON+THE+MOON+AS+HE+HAS+AT+THE+OTHER+TEN=TESTS

```
running time, random times:  0.010970830917358398 ,  18
running time, random times:  3.351039409637451 ,  30
running time, random times:  2.934103488922119 ,  47
running time, random times:  3.7339866161346436 ,  14
running time, random times:  2.3478100299835205 ,  36
running time, random times:  1.7373557090759277 ,  26
running time, random times:  2.601097583770752 ,  24
running time, random times:  1.633582353591919 ,  28
running time, random times:  4.986665725708008 ,  38
running time, random times:  2.372605085372925 ,  29
10
```

(Những lần thành công sẽ in ra thời gian chạy và số lần đã restart, có 10 lần thành công)

Có thể thấy mọi chuyện không còn đơn giản với nó nữa. Nếu ta đặt thời gian lớn hơn thì hẳn sẽ có nhiều trường hợp thành công hơn, nhưng ta sẽ cố gắng để chúng lộ ra những điểm yếu để có thể tiến hành cải thiện. Ở đây ta nhận ra điểm yếu cố hữu của thuật toán này là ta sự "đơn độc" của phép tìm kiếm, dù cho được tạo nhiều cơ hội trong các lần restart nhưng việc tìm kiếm cũng chỉ có mình nó thực hiện. Nên ta sẽ tạo cho nó thêm vài người "đồng minh" để quá trình tìm kiếm diễn ra thuận lợi hơn. Ta áp dụng thêm Local beam search, ở mỗi lần restart lại trạng thái ngẫu nhiên, ta không chỉ tạo 1 mà tạo ra K trạng thái (K do ta quy định), ở K trạng thái này ta sẽ cố gắng bóc ra K successors tốt nhất từ chúng để tiếp tục cho lần tìm kiếm kế tiếp, có nghĩa là cùng 1 lúc sẽ có K trạng thái đang hoạt động để đóng góp vào danh sách những trạng thái tốt nhất. Ta cũng thay đổi cách lựa chọn successors lại, thay vì chọn ngẫu nhiên 1 biến để tạo successors như trong phiên bản thường, ta chuyển thành mỗi trạng thái ta quét qua tất cả các biến để cố gắng lấy ra những successors tốt nhất của nó, sẽ có những trạng thái không có successors nào được chọn. Ta cũng sử dụng chức năng đa luồng trong việc tìm successors của các trạng thái, do ta đã hạn định yếu tố thời gian là yếu tố dùng để dừng thuật toán, nên việc cho chúng chạy tuần tự sẽ rất hao tốn, thời gian chạy mỗi lần tìm successors trong K trạng thái lúc này sẽ là thời gian của trạng thái luồng lâu nhất thay vì là tổng thời gian của tất cả các luồng nếu chạy tuần tự. Đương nhiên ta vẫn sẽ giữ Random restart, dù lúc này tỉ lệ thành công đã cao hơn do có nhiều trạng thái được xét hơn, nhưng việc tất cả cùng rơi vào local maxima (danh sách các trạng thái tốt nhất rỗng) là hoàn toàn có thể. Ta xét 1 ví dụ để hiểu rõ hơn thuật toán này: A + B = C với K = 3. Ta tạo ra các 3 trạng thái ngẫu nhiên {A: 8, B: 7, C: 5}, {A:1 , B: 4, C: 2}, {A: 2, B: 4, C: 5}, từ bọn chúng ta thu được 3 trạng thái tốt nhất là {A: 1, B: 4, C: 5} (từ trạng thái 2), {A: 2, B: 3, C: 5} và {A: 2, B: 4, C: 6) (từ trạng thái 3).



(Hình minh họa quá trình sinh successors)

Ta chạy thử thuật toán vừa xây dựng với input mà Random-restart đã rất khó khăn để xoay xở: SO+MANY+MORE +
SO+MANY+MORE+MEN+SEEM+TO+SAY+THAT+THEY+MAY+SOON+TRY+TO+STAY+AT+HOME+SO+AS+TO+SEE+OR+HEAR+THE+SAME+ONE+MAN+TRY+TO+MEET+THE+TEAM+ON+THE+MOON+AS+HE+HAS+AT+THE+OTHER+TEN=TESTS

```
running time, randome times:  0.13509893417358398 , 1
running time, randome times:  0.17753219604492188 , 1
running time, randome times:  0.19547677040100098 , 1
running time, randome times:  0.13238286972045898 , 1
running time, randome times:  0.1593918800354004 , 1
running time, randome times:  0.1398935317993164 , 1
running time, randome times:  0.13858723640441895 , 1
running time, randome times:  0.13891196250915527 , 1
100
```

Rất tuyệt, dù cho gặp 1 input khó thì thuật toán vẫn chạy tốt với 100 lần thành công (giới hạn thời gian 5s), thời gian chạy cũng rất nhỏ. Có vẻ như việc tạo ra 1 "nồi lẩu thập cẩm" của chúng ta đã không vô ích. Ta có thể thấy số lần restart gần như rất ít, tuy nhiên như đã nói, ta phải cố gắng để thuật toán an toàn hơn, dưới đây là những lần dùng đến random restart:

```
running time, randome times:  1.3308897018432617 , 2
```

(cần restart lại 1 lần để có kết quả)

```
running time, randome times:  3.233898639678955 , 4
```

(cần restart lại 3 lần để có kết quả)

Và cuối cùng là câu hỏi ta có nên cho phép những sideways hay không? Câu trả lời là Nhóm có cho phép sideways, mặc dù trong bài toán này thì các trạng thái có heuristic bằng nhau hiếm khi xảy ra, tuy nhiên chúng ta sẽ tạo mọi điều kiện để đạt được mục tiêu nên vẫn sẽ chấp nhận các sideways, nhưng cũng phải tinh tế để không vì 1 tính năng nhỏ mà làm ảnh hưởng hiệu năng cả chương trình. Mà tình huống làm ảnh hưởng nhiều nhất hiệu năng nhiều nhất là các trạng thái cứ nhảy qua nhảy lại với nhau vì chúng có cùng heuristic, để giải quyết tình huống này, ta áp dụng ý tưởng của Tabu search, ta lưu lại 1 danh sách nhỏ (dưới dạng queue) các trạng thái gần nhất vừa đi để không cho successor đi lại vị trí cũ, khi danh sách đầy thì đẩy cái cũ nhất ra để lưu trạng thái mới.

b. Triển khai

Ta sẽ đề cập đến triển khai của thuật toán cuối cùng vừa nêu bên trên (Random restart hill climbing + Beam search).

Đầu tiên ta đề cập đến cách để tính toán các biểu thức trong chương trình. Ta lưu các trạng thái thành 1 lớp bao gồm 2 thành phần. Thành phần đầu tiên là 1 dictionary lưu giá trị đã gán cho các biến (chẳng hạn {A: 2, B: 3, C: 5}), thành phần thứ 2 là dictonary chứa giá trị của từng toán hạng trong biểu thức, ta lưu thứ này nhằm mục đích cải thiện thời gian tính toán trong chương trình, thay vì với mỗi lần gán biến ta phải quy đổi toàn bộ toán hạng từ chữ ra giá trị số nguyên, chẳng hạn ta có 1 phép tính là AB+BC=EA với trạng thái là {A: 2, B: 1, C: 3, E: 4}, ta lưu lại giá trị các toán hạng là {0: 21 (AB), 1: 13 (BC), 2: EA(42)] (key là vị trí của toán hạng trong biểu thức, value là giá trị của toán hạng đó),, ví dụ ta gán được trạng thái mới là {**A: 5**, B: 1, C:3, E: 4), lúc này ta đi vào để chỉnh sửa lại giá trị của những toán hạng bị ảnh hưởng là AB và EA thành (0: **51** (AB), 1: 13 (BC), 2: **45**(EA)), những toán hạng không liên quan sẽ không bị ảnh hưởng gì, rồi tính lại giá trị cho toàn bộ biểu thức. Nếu không lưu lại thì ta sẽ phải quy đổi từng toán hạng ra số nguyên ở mỗi lần tính toàn bộ giá trị biểu thức.

Tiếp theo là về cách tính toán biểu thức, ta tiếp tục áp dụng biểu thức Ba Lan ngược như trong backtracking để tính toán 1 biểu thức, các toán hạng và toán tử sẽ được lưu

dưới dạng 1 list theo kiểu Ba Lan ngược, chẳng hạn [“ABCD”, “HGD”, “+], ở mỗi lần tính, ta truy xuất đến dictionary chứa giá trị biểu thức đã đề cập bên trên để tính, ví dụ ta có biểu thức [“AB”, “CD”, “+”] với dictionary giá trị các biểu thức là {0: 123, 1: 24}, ta thu được [123, 24, “+], và cuối cùng dùng cách tính 1 biểu thức Ba Lan ngược để tính ra giá trị biểu thức.
Về chi tiết chương trình, đầu tiên ta đặt 1 vòng lặp bên ngoài có điều kiện dừng là thời gian chạy nhỏ hơn thời gian tối đa. Đây là vòng lặp đại diện cho các lần restart, do đó ngay khi vừa vào vòng lặp ta sẽ tạo ra K trạng thái ngẫu nhiên.
Đi đến vòng lặp bên trong, nó đại diện cho việc chạy những luồng tìm kiếm từ danh sách các trạng thái tốt nhất hiện tại, nó sẽ dừng lại nếu không còn trạng thái tốt nào để xét nữa hoặc chạy quá số lần tối đa. Khi vào vòng lặp này, ta sẽ tạo ra K luồng tìm kiếm tương ứng với K trạng thái hiện tại đang xét, các luồng này sẽ chạy hàm tìm successors của riêng nó, 1 vòng lặp kết thúc khi mà không còn luồng nào còn chạy.
Mỗi luồng tìm kiếm sẽ chạy hàm tìm những successors tốt nhất tương ứng với trạng thái gốc mà nó mang theo. Nó sẽ tiến hành lấy tất cả các biến, mỗi biến thử lần lượt các giá trị từ 0 đến 9 để tạo ra những successors tốt hơn trạng thái gốc hiện tại.
Các successors sẽ được lưu trữ trong 1 priority queue, do ta đã quy định sẽ không có quá K luồng được chạy trong những vòng lặp. Khi 1 luồng gửi successor của nó vào, nếu queue lúc này đã đầy ta sẽ so sánh nó với trạng thái tệ nhất trong queue (có heuristic lớn nhất), nếu nó tốt hơn thì ta sẽ đẩy trạng thái đó khỏi queue và thêm successor vừa nhận vào, nếu nó tệ hơn thì ta sẽ bỏ qua successor này.

Ta có thể minh họa mã giả chương trình như sau:

```
found = False #Marks that the solution was found or not
bestList #priority queue storing the best successors
Function AddSuccessor(state):
    If (not bestList.Full or (bestList.worstState.heuristic >
            state.heuristic):
            If (bestList.Full)
                bestList.Deque()
            bestList.Enque(state)

Function ThreadSearch(state):
    currentHeuristic = state.Heuristic
    For var in state.variables:
            For i = 0 to 9:
                newState = Assign(var, i)
                If (newState.heuristic = 0)
                    Found = True
                Else If (newState.heuristic < currentState.heuristic)
                    AddSuccessor(newState)

Function BeamSearch(expressions, K, maxTime, maxAttepEachSearch):
    While (currentTime < maxTime):
            For i = 1 to K:
                #A queue storing random states
```

```
                states = empty queue
                randomState = CreateRandomState(expressions)
                states.Enque(randomState)
                For i = 1 to K:
                        #Get a state
                        state = states.Deque()
                        thread = CreateThread(state)
                        thread.Run()
                If Found:
                        Return solution
#Run out of time
Return no solution
```

IV. Thử nghiệm

Một bài toán Cryptarithmetic được đánh giá bởi nhiều yếu tố cấu thành nên nó. Ta sẽ xem xét 1 số yếu tố tiêu biểu để có cái nhìn toàn diện về những thuật toán đã xây dựng. Ở mỗi test case ta sẽ chạy 100 lần và lấy ra thời gian trung bình cho 100 lần chạy.

1. Số lượng biến

No.	Test cases	Number of variables	Average time	
			Backtracking	Hill climbing + Beam
1	ABBAA+BAABB=CCCCC	3	0.001	0.001
2	VIOLIN-CELLO=CORNET	9	0.062	0.07
3	ABBBAABBBAAA +ACCCDAACCCDD +ABCBBDAAADAA -BABBBBBBBBA =BBDBBADAACAB	4	0.001	0.03
4	ALNCAGZHAGLA +LLGZGZRNMAAC +NGGAARMGCZKM -GZHHKKLHNHK =HMLCALARRHZM	10	0.06	0.15
5	LALALA*AA=LCCCCCA	3	0.00004	0.002
6	ALCKMN*ZR=HZGNKZR	10	0.004	0.04

Ở những test cases trên, Nhóm đã cố tình tạo ra những biểu thức có độ khó tương đương nhau nhưng lại số biến khác nhau, Ta có thể thấy tình huống có nhiều biến sẽ chạy chậm hơn nhiều so với những biểu thức có ít biến. Có 1 vài test case (1 so với 2, 5 so với 6) thì độ chênh lệch khá lớn (khoảng từ 20 đến 50 lần)

Điều này cũng dễ hiểu khi mà quá trình thế giá trị cho biến và kiểm tra là quá trình rất tốn thời gian, nếu trong backtracking thì có thể dẫn đến đường đi quá sâu và mở nhiều nhánh, còn trong local search (bị ảnh hưởng bởi yếu tố ngẫu nhiên) thì sẽ phải mở rất nhiều nút. Hơn nữa, không gian trạng thái còn bị ảnh hưởng rất nhiều bởi số lượng biến, chẳng hạn như ở test case 1 thì không gian trạng thái chỉ là 720 (chỉnh hợp chập 3 của 10) trong khi với test case 2 có cùng độ khó thì là 3628800 (chỉnh hợp chập 9 của 10) trạng thái.

Ta cũng thấy ở những test case đơn giản thì backtracking vẫn khá nhanh, do nó rất đơn giản, không cần quá nhiều thứ khác hỗ trợ trong quá trình chạy. Trong khi beam search lại cần nhiều thứ hỗ trợ như là module thread, copy, random… do đó những test case quá đơn giản thì những thứ chúng ta xây dựng đôi khi lại trở thành gánh nặng của thuật toán.

2. Chiều dài toán hạng và số lượng toán hạng

No.	Test case	Number of operands	Average length of operands	Average time	
				Backtracking	Hill climbing + Beam
1	HUI-HJY+JVY-(VHX-FHJ-HOI-XHF-OJU+UIJ)+JOY+JXJ+IJU+IUF-YHX+XQU+OOX-VJX+XVF-XIO+HIX=FOOO	20	3	0.65	0.03
2	TGTTTGQTE-QYYTQGTGJ+TYTGOBNQA+NQQGGYJGB+NOGQENQTQ-NBOTGOYYE-GJTNJOJJO+TABTBTOOE-GGOBBNYTQ+NOTOQNBON+QAGYBYYGO+OEJEEAYGQ-JYTNYQYYE+OTJNEJNOQ+GQOBNTJEG-TJAAYNJEQ-(GJEOEABTQ-QYTJQOONA)+QJYEBEJTO+EYNBYTQTJ=GNONEQOGYO	20	9	23.26	0.07
3	CCCYK+AXATY+KMAML-MCCLA+XTYYA+AKIII+CAPAK+ILPIY-XKAKM-LMATI=XPLTXI	11	5	2.3	0.03
4	BGYOJ-OGNTT+JGEYO-YGOGT-(TNTOA+OTAYB-GJAQG-NAOEA)+JOQTB+GYNQJ+OYGNG-(NGYTB+BANJG)+JQEJN+BAYYB+EOTTG+GAQBA+GGABT-YEYOG-(OGBGG+BEOJB+OTEGQ)-(JNAAY+YGGYG+GTBBJ+OJTOJ-EYTYB)+BBAEG-QOEOY+TBJJQ+QQBQB+TBEGG+JEEYQ+JGGQT+QYTJN+NNTOG-(BEEYJ+NBEAQ+ENOTO-TONBN)-(NATYY+BNYAA)+BBGAQ-NJEYE+OEJEB-BJYTT+OONET+NYYOB-NTEOG=JBQEEJ	50	5	11.75	0.2

Có thể thấy có sự khác biệt lớn giữa 1 biểu thức có độ dài các toán hạng và số lượng toán hạng lớn với những biểu thức có quy mô nhỏ hơn.

Đối với backtracking thì nó sẽ phải đi qua quá nhiều cột và quay lui lại những cột rất xa trong những trường hợp xấu, nó hoàn có thể xảy ra tình huống quay lai từ cột đầu về lại tận cột cuối. Việc tính toán thì không quá ảnh hưởng nhiều do nó phụ thuộc vào số lần quay lui (khiến cho nó phải tính lại), nếu không thì chi phí cho tính toán các cột và hàng không quá nặng nề.

Local search cũng bị ảnh hưởng nhiều bởi vấn đề này. Do nó phải liên tục phải thế biến và tính toán lại giá trị biểu thức, việc tính toán lại là gánh nặng lớn đối với thuật toán khi mà số lượng các toán hạng quá lớn. Chúng ta cũng nghĩ tới tình huống khi mà có quá nhiều toán hạng thì có càng nhiều phần tử dẫn đầu khác nhau (phải khác 0), từ đó làm cho bài toán trở nên ngặt nghèo hơn khi phải liên tục tránh việc thế 0 cho các biến dẫn đầu.

Trong những tình huống test case phức tạp thì backtracking bắt đầu lộ ra điểm yếu chậm chạp của nó, khi mà quá trình tính toán và quay lui rõ ràng là quá trình vét cạn. Trong khi đó, local search vẫn chạy rất nhanh trong những tình huống phức tạp.

3. Sự phức tạp của phép tính

No.	Test case	Average time	
		Backtracking	Hill climbing + Beam
1	MCUULRXWWG+CEILCMXXUX+IIWERILUWR+EEWLREXLCX+CWMGWGGMIR+MIGXWXRMGI+MMUWMIRWLX+CMLMUCWEGE+XUMLLUWCIM+LILRXUXWIU+GRMUUIMMRI+GEIWLUEMGR+RUUMRMXELE+EUCIIGUGLG+RUXIMEIIUC+CLIIIMLUWX+LEXCCILUGI+CLRLERURRM+RXXIUWMEUL+ECUUXGLMUM+CIXMWEXLRR+MWLCLXGCLI+LURXEGCWXR+LLWXCXIMCI+XXIECLXRMX+GIELURWGGC+CUWLMGCERM+MULRMGGRLE+RGLELUWGEU+GRGERGCEUC+GMGLCELXLW+LERXCEMLGW+MIWMGREUXR+IWGEXUUXXW+CUIWCMWMGG+GMUWWRWMIR+LEXLCUUCGM+CEWMLEGUXW+WGCXEMUUCC+XRCWCREXGU+MECGXXGLWW+LXLEGCGIMM+RMXGLUXXCU+LCLIRCELGR+GIWWRIXIMX+RCEILIRXMW+CCUXEWMGGE+LMCGXWIXMG+WRGMMMEXWL+XEUGLUWEWL=RWRMXEGLXMCE	46.22	0.26
2	EGXECGMLXU-XWLGGERWRC+EXXRMGLGWL-RLRUULGELC+RWGCMCGCML+XLUEXIEIUC-RIXUXIEMUW+LGCMCXXMLX-LCEECMICEL-MXUXIMEGUC+XULUMWEERI+RUXELILMWL+GUCGXMULCI+CUXMCRRCLU-CLIWRMIIXL-GWRWGRREGR+MWCLXXECXM+EUWUMWUGXC+XRCWMIGGUI+MEGXMCGXXU+LXGLRELCLR-XWLXLLCRRU-RGGLREGUXU+LXUCLCCWML-	0.8	0.18

	XLRWRIILUG+REXIXUECUX-XULCGRLGRM+GMLIMWRMIC+EXIXILCMGI+LWGEGMWXWM+GWLCGXCCRU-XXIMXLULEW+MCMLXCCGME-LGIUWIMMCX+LRWUXCEGLW+EULMLMUUUG+RWLXLRCMUI-ECXWIUXCEC-MIRMLUCXER+RRLMMCUXWM-EMCMMMUWUL-LEERIIUMUW+IGLECRLXIC-EGXMUGWUGL+ILCMULUWIG+IRIULWMXCW-MLWXREXWRR+EIEEWMXXUI-LECMWUGRMC-EIIWUUWCIE=XRWRWMWCXIR		
3	LELLMIWWCE+XXUXCUCMWU+IRUMLGRILU-GRUEUIMRWM+GMEWLXIRMU-(RGLELGGCLU+RURXRIEMIR)-EURXLLEUEG+LGGURMICEL-WEWUWLUEXW+WXCWEEWGUI-(CIMEXURMEU+CGULMXWMIG-GULRIELGLX)-LRMIGMIGEU+RIICIRUERI+CWGLRLGGLW-(RXILLWMEMX+RIRMGUXXUC+LLWUGGGMLM)-MEEMEGLGER-RCUGWUXEXM-EMEGEUWLRM-WEXIWUMMRU+MMGUREMXMM+ECWWILEMWM+XRLXXLEEUG-(MXELLXCUMX-MEMGGUEWEG-WGCIWIGLCG)-IULRCXLWLG-(CRWURWEIRX-EXEIXGEMCW)+MUWCIRWEWE+ELRULMEUEM-EGICGMCIIR+GILWLREGGL+WCELUUCUWE-RWMLLILLRC+EUWUCRUWGL-XCIMGCMRLM-(GEGLWEIMMR+GWCWUEWEEM)+EEGREIXMXU+XRLGXLURXW+XILRLXWRGX+REWUMGMRIL-(XIGLLWREXI+LGIWWCMMWC)+WWRMLXRGLC=RLWRIGWGIGC	31.56	0.16
4	UEK+I*(UEKUI+IYI)-Y*(YYHY+UEK)+QZAA+ZHQ*Q-T*YYY-I*T+(UEKK+YQHI)*H+UU*(QHQQ-UUK)+UE*(UKH+TT)=UYAIZI	28	0.1

5	ADFASD*BEFEX*BCFDA*ADBXF X*AB*XDSX*DD*AE*FX*ASDDD* EECXA*EXXFA*XDEXXX*EEDCAF *XDEXXA*DIEEDDDX*CEENA*CC CXXXXX*AEEFFCX=CCCXDAEFA AAXXDDBBCFFFA	6.5 (No solution)	10 (No solution)

Ta có thể thấy đối với Local search thì độ phức tạp của biểu thức không ảnh hưởng quá nhiều, do ta đã triển khai nó tính lại toàn bộ biểu thức ở mỗi lần thế, mà các biểu thức con phức tạp bên trong đổi ra cũng chỉ là những phép tính thông thường.
Trường hợp cuối cùng (4) để minh họa cho điểm yếu của Local search, miễn là chưa tìm được giải pháp thì nó sẽ chạy đến khi nào hết thời gian quy định mới thôi (ta không chắc được là có giải pháp hay không), trong khi đó backtracking có thể kiểm tra được tình huống này, nhưng đó là khi nó đã quét qua tất cả các trạng thái có thể, hay nói cách khác là nó đã vét cạn bài toán.

4. Nhận xét chung

1 bài toán có thể bị ảnh hưởng bởi nhiều yếu tố, trong quá trình chạy thử thì ta thấy yếu tố số lượng biến và số lượng toán hạng là ảnh hưởng nhiều nhất đến quá trình thực thi. Trong khi 1 biểu thức có phức tạp hay không thì cũng là tập hợp của những phép tính đơn giản, do đó nó không ảnh hưởng quá nhiều.
Mặc dù được triển khai theo kiểu vét cạn, nhưng những tình huống đơn giản thì backtracking cũng rất hiệu quả, không gian tìm kiếm không quá lớn dẫn đến việc ra kết quả rất nhanh. Đây cũng là 1 thuật toán optimal, nó sẽ đảm bảo cho việc ra kết quả nếu kết quả tồn tại. Tuy nhiên khi mà những tình huống trở nên lớn và phức tạp khi nó lại khá ”ì ạch”, hầu như thời gian chạy rất lâu so với Local search. Kết quả đưa ra bởi nó cũng “kém hấp dẫn”, do mỗi lần thực thi đều cho ra 1 giải duy nhất, và 1 thời gian chạy không đổi.
Local search (Hill climbing + Beam search) chạy rất hiệu quả trong những tình huống phức tạp, khi mà không gian trạng thái quá lớn, nó vẫn có thể giải quyết rất tốt bài toán. Các giải pháp được đưa ra bởi nó cũng rất đa dạng qua những lần chạy khác nhau. Tuy nhiên, điểm yếu cố hữu của thuật toán này vẫn là sự an toàn, khi mà chúng ta đã cố tình xây dựng nó dựa vào yếu tố ngẫu nhiên thì nó sẽ không đảm bảo sẽ cho ra giải pháp cho dù có tồn tại. Nói về yếu tố ngẫu nhiên thì đây vừa có thể được xem là điểm mạnh cũng như điểm yếu của thuật toán, nếu may mắn ta có thể ra giải pháp cực kì nhanh, còn ngược lại có thể sẽ rất lâu, thậm chí là không tìm được giải pháp.

V. Tham khảo

Xây dựng backtracking: https://www.geeksforgeeks.org/solving-cryptarithmetic-puzzles-backtracking-8/

Tabu search, local search cho CSP: sách Artificial Intelligence: A Modern Approach version 3.