

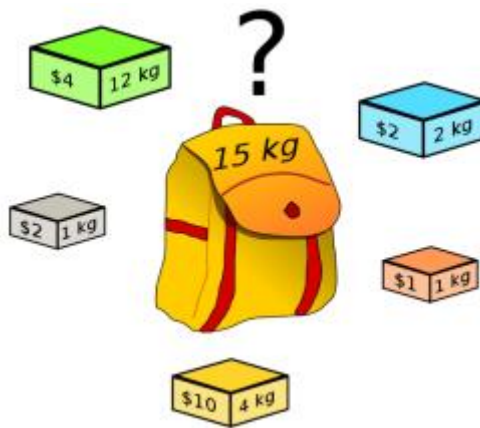


**TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN - ĐHQG  
TP.HCM  
VNUHCM - UNIVERSITY OF SCIENCE**

**KHOA CÔNG NGHỆ THÔNG TIN**

**PROJECT 1**

**THE KNAPSACK PROBLEM**



**MÃ MÔN HỌC: CSC-14003\_21CLC04**

**THỰC HIỆN:**

**NHÓM 18**

**GVHD:**

**LÊ NGỌC THÀNH**

**NGUYỄN NGỌC THẢO**

**HỒ THỊ THANH TUYẾN**

**Tp. Hồ Chí Minh, tháng 4 năm 2023**

## GIỚI THIỆU THÔNG TIN THÀNH VIÊN

Tên nhóm: 18

Tên thành viên	Mã số sinh viên
Trịnh Minh Trung (Nhóm trưởng)	21127711
Thành Thiện Nhân	21127535
Cao Nguyễn Khánh	21127627
Nguyễn Văn Tuấn Kiệt	21127331

## BIÊN BẢN PHÂN CÔNG CÔNG VIỆC NHÓM

Công việc	Thành viên	Tiến độ
Algorithm 1	Thành Thiện Nhân	100%
Algorithm 2	Cao Nguyễn Khánh	100%
Algorithm 3	Nguyễn Văn Tuấn Kiệt	100%
Algorithm 4	Trịnh Minh Trung	100%
Tạo Dataset với size 10-40	Nhóm	100%
Tạo Dataset với size 50-1000	Nhóm	100%
Report	Nhóm	100%

# MỤC LỤC

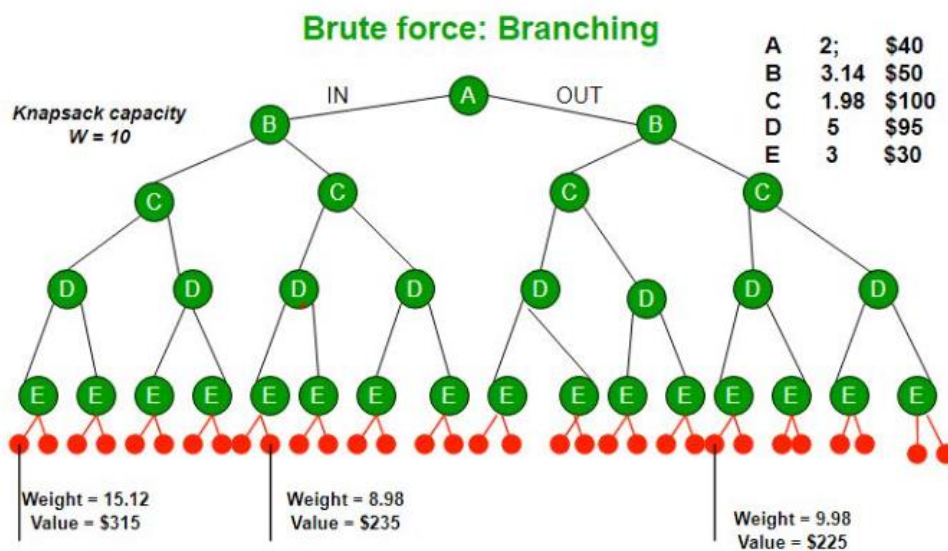
<b>1. Brute Force Searching.....</b>	<b>4</b>
Ý tưởng thuật toán: .....	4
Mã giả:.....	4
Tóm tắt chương trình:.....	5
Kết quả thu được:.....	6
Nhận xét: .....	7
<b>2. Branch &amp; Bound.....</b>	<b>8</b>
Ý tưởng thuật toán: .....	8
Mã giả:.....	8
Tóm tắt chương trình:.....	10
Nhận xét: .....	11
Đánh giá: .....	11
Kết luận: .....	12
<b>3. Local Beam Search .....</b>	<b>13</b>
Ý tưởng thuật toán: .....	13
Mã giả:.....	14
Tóm tắt chương trình:.....	14
Nhận xét: .....	14
Minh họa:.....	15
<b>4. Genetic Algorithms.....</b>	<b>17</b>
Ý tưởng thuật toán: .....	17
Mã giả:.....	20
Nhận xét: .....	21
<b>5. SO SÁNH CÁC THUẬT TOÁN.....</b>	<b>22</b>

# NỘI DUNG BÁO CÁO

## 1. Brute Force Searching

### Ý tưởng thuật toán:

Brute Force Searching là một phương pháp tìm kiếm một cách trực tiếp và đơn giản. Nó được sử dụng để xét tất cả các trường hợp có thể xảy ra trong việc sắp xếp vật dụng (items), sau đó sẽ kiểm tra từng trường hợp một xem nó có thỏa mãn yêu cầu bài toán hay không, nếu thỏa mãn thì sẽ xét giá trị tổng của trường hợp đó. Sau khi xét tất cả các trường hợp và so sánh các giá trị tổng của mỗi trường hợp thì ta sẽ có một giá trị tối ưu max\_value mà bài toán cần tìm, cùng với chuỗi số 0,1 để đại diện cho việc từng vật dụng có được chọn hay không.



*Khai triển tất cả các trường hợp như tương tự như một cái cây phân nhánh*

*(Nguồn: <https://www.geeksforgeeks.org/0-1-knapsack-using-branch-and-bound/>)*

### Mã giả:

- Bước 1: Tạo ra tất cả các trường hợp sắp xếp vật dụng.

- Bước 2: Nếu đã xét tất cả trường hợp thì nhảy đến bước 6. Nếu chưa xét tất cả thì chọn một trường hợp (chưa được xét), kiểm tra xem có ít nhất một vật dụng được chọn cho mỗi loại (class) hay chưa, nếu có thì sang bước 3, nếu không thì sang bước 5.
- Bước 3: Kiểm tra xem cân nặng tổng có nằm trong sức chứa của balo không. Nếu có thì sang bước 4, nếu không thì sang bước 5.
- Bước 4: Kiểm tra xem giá trị tổng của trường hợp này có lớn hơn max\_value hay không, nếu có thì gán max\_value với giá trị tổng của trường hợp này.
- Bước 5: Nhảy về bước 2 (kiểm tra trường hợp tiếp theo).
- Bước 6: Đưa ra kết quả cuối cùng mà bài toán cần tìm

## Tóm tắt chương trình:

### Về thư viện:

Đầu tiên em sẽ thêm thư viện “intertools” để sử dụng hàm combination(), giúp tạo ra các tổ hợp cho việc chọn các vật dụng, và thêm thư viện “os” để dùng hàm os.path.isfile(), kiểm tra xem đường dẫn tới file có tồn tại hay không.

### Về hàm *main*:

Người dùng sẽ được yêu cầu để nhập một số nguyên x (x là chỉ số của test case) để có thể sử dụng thuật toán cho file INPUT\_x.txt, nếu người dùng nhập x=0 thì chương trình sẽ dừng, nếu không nhập một số hay người dùng nhập số nhưng không tồn tại file INPUT tương ứng thì sẽ thông báo và yêu cầu nhập lại.

Còn nếu nhập thành công chỉ số và file INPUT cho chỉ số ấy có tồn tại thì file INPUT sẽ được đọc theo từng dòng bởi hàm fin.readline(), được lược bỏ dấu cách ở đầu và cuối dòng bởi hàm strip(), được phân ra thành từng phần tử bởi hàm split() và được ép kiểu dữ liệu với hàm map cho từng phần tử (vì dữ liệu đọc vào ban đầu sẽ có dạng string)

*Ví dụ: weights = list(map(float, fin.readline().strip().split(' ')))*

Sau đó hàm brute\_force\_search sẽ được gọi để giải bài toán, giá trị trả về sẽ là giá trị tối ưu (max\_value) và chuỗi các số 0,1 thể hiện việc chọn vật dụng (best\_subset)

Sau khi có được kết quả cần tìm thì chương trình sẽ tiến hành xuất kết quả qua file OUTPUT\_x.txt

### Về hàm *brute\_force\_search*:

Thuật toán sẽ có:  $n$  (số lượng vật dụng được xét),  $\text{max\_value}$  (giá trị tối ưu) được khai báo với giá trị 0,  $\text{best\_value}$  (chuỗi kết quả) được khai báo ở dạng list

Dùng hai vòng lặp for để xét tất cả các tổ hợp có thể có khi sắp xếp vật dụng với hàm `intertools.combinations()`

Với mỗi tổ hợp, kiểm tra xem có ít nhất một vật dụng được chọn cho mỗi loại hay chưa:

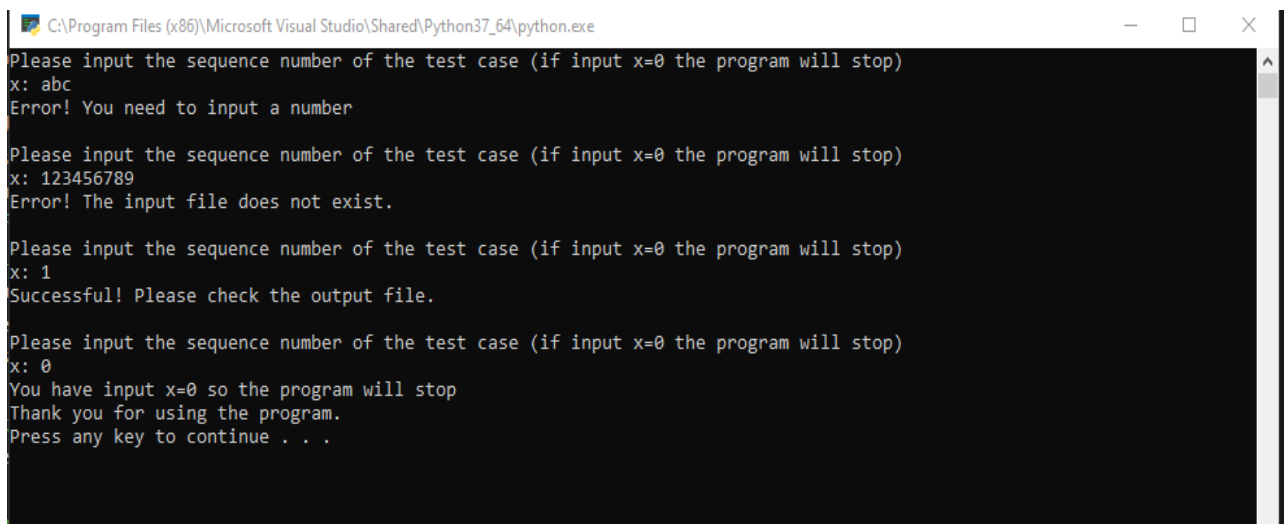
*if len(set(class\_index[i] for i in subset)) == c:*

Nếu thỏa mãn, thì kiểm tra tiếp xem tổng cân nặng có vừa với balo hay không:

*if sum(weights[i] for i in subset) <= W:*

Nếu thỏa mãn, thì sẽ tiến hành tính  $\text{subset\_value}$  (giá trị tổng của trường hợp này) và so sánh với  $\text{max\_value}$ , nếu  $\text{subset\_value} > \text{max\_value}$  thì sẽ tiến hành cập nhật  $\text{max\_value}$  và  $\text{best\_subset}$ .

## Kết quả thu được:



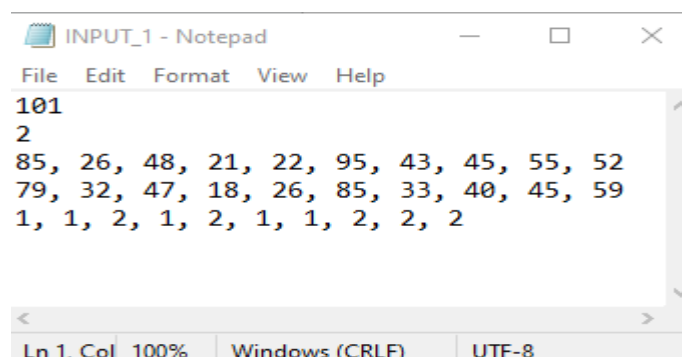
```
C:\Program Files (x86)\Microsoft Visual Studio\Shared\Python37_64\python.exe
Please input the sequence number of the test case (if input x=0 the program will stop)
x: abc
Error! You need to input a number

Please input the sequence number of the test case (if input x=0 the program will stop)
x: 123456789
Error! The input file does not exist.

Please input the sequence number of the test case (if input x=0 the program will stop)
x: 1
Successful! Please check the output file.

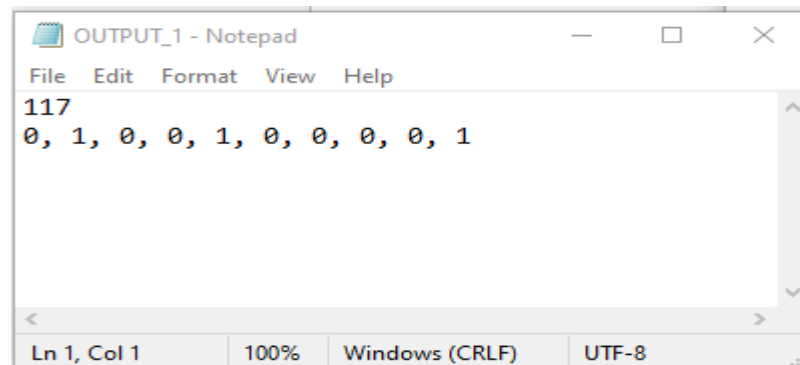
Please input the sequence number of the test case (if input x=0 the program will stop)
x: 0
You have input x=0 so the program will stop
Thank you for using the program.
Press any key to continue . . .
```

Màn hình console



```
INPUT_1 - Notepad
File Edit Format View Help
101
2
85, 26, 48, 21, 22, 95, 43, 45, 55, 52
79, 32, 47, 18, 26, 85, 33, 40, 45, 59
1, 1, 2, 1, 2, 1, 1, 2, 2, 2
```

*File INPUT\_1.txt được xét*



*File OUTPUT\_1.txt được xuất ra*

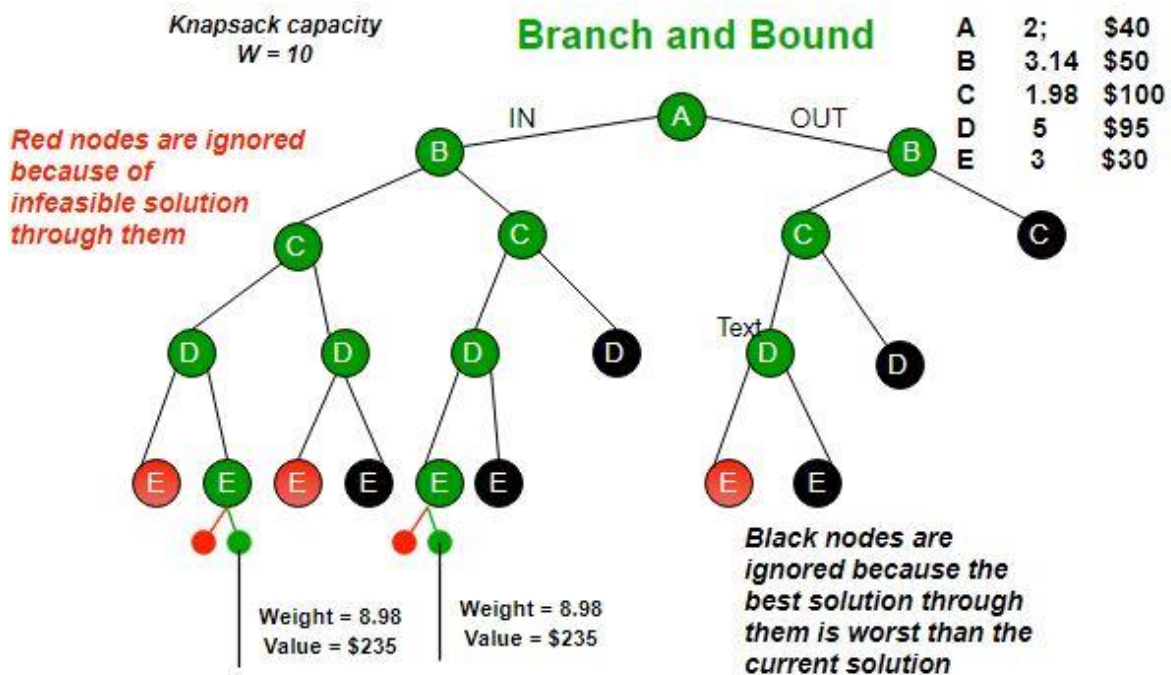
### **Nhận xét:**

- Brute Force Search có độ phức tạp về thời gian là  $O(2^n)$  với tất cả  $2^n$  trường hợp sắp xếp vật dụng (items) sẽ được xét, và kiểm tra xem trong mỗi trường hợp thì cân nặng tổng có vượt quá sức chứa của balo hay không (nếu không vượt quá sức chứa của balo thì xét tiếp rằng giá trị tổng có lớn hơn giá trị max\_value hay không).
- Brute Force Search có độ phức tạp về không gian là  $O(n)$  với sự phụ thuộc vào số lượng vật dụng được xét cho bài toán và các thông tin về giá trị, cân nặng và loại vật dụng.
- Ưu điểm của thuật toán Brute Force Search là có tính đơn giản và dễ hiểu, dễ cài đặt, và theo lý thuyết thì thuật toán sẽ đảm bảo việc tìm ra kết quả tối ưu nhất. Nhưng đánh đổi với nó là việc độ phức tạp của thuật toán sẽ tăng lên rất đáng kể khi phạm vi tìm kiếm tăng, vì vậy với một tập hợp có quá nhiều vật dụng (items) để xét thì việc dùng thuật toán này sẽ mất rất nhiều thời gian.

## 2. Branch & Bound

### Ý tưởng thuật toán:

Thuật toán bắt đầu bằng việc sắp xếp các vật phẩm theo giá trị / khối lượng vào mảng tạm. Vì mảng đã được sắp xếp nên vật phẩm đầu tiên thuộc 1 class sẽ là vật phẩm tối ưu nhất. Ta lấy ra 1 vật phẩm tối ưu của mỗi class trước. Sau đó sẽ sắp xếp lại các vật phẩm theo giá trị / khối lượng vào mảng để tối ưu hóa giới hạn về khối lượng. Sau đó, nó tạo ra một nút gốc, thêm nó vào cây và thêm nó vào ngăn xếp BB\_stack. Tiếp theo, ta bắt đầu lặp lại việc tạo các nút con, kiểm tra xem chúng có cải thiện giá trị tối ưu hoá hay không, và thêm chúng vào BB\_stack cho đến khi cây được duyệt hết. Cuối cùng, thuật toán truy xuất lại các vật phẩm đã chọn và không chọn bằng cách duyệt cây từ nút giải pháp tốt nhất về nút gốc. Kết quả là một bộ giá trị tốt nhất và một mảng cho biết vật phẩm nào được chọn và không được chọn. Bộ giá trị tốt nhất này không phải là kết quả chính xác vì khi chọn vật phẩm tối ưu từ mỗi loại, em đã thay đổi giá trị của chúng lên cực lớn để chúng sẽ được ưu tiên chọn trước. Thông qua mảng cho biết vật phẩm nào được chọn và không được chọn em tính lại bộ giá trị tốt nhất một lần nữa.



(Nguồn: <https://www.geeksforgeeks.org/branch-and-bound-algorithm/>)



## Mã giả:

- Function BranchAndBound\_Slove(max\_weight, listV, listW, listC, numClass): // đây là hàm chính thực hiện chương trình
  - ❖ BB\_solution = None
  - ❖ BB\_stack = []
  - ❖ BB\_tree = []
  - ❖ items = len(values) # Get number of items
  - ❖ New listtaken = [0] \* items # Allocate memory for taken
  - ❖
  - ❖ New listTemp = listV[:]
  - ❖ New listArr(index, value/weights, class).sort(value/weights)
  - ❖ for i in range(1, numclasses + 1):
    - Tìm vật phẩm có giá trị value/weights lớn nhất của mỗi class .
    - Gán giá trị value = Số Cực Lớn, cho những vật phẩm được tìm thấy.
  - ❖ New listvalue\_per\_weight(index, value/weights).sort(value/weights)
  - ❖ Đổi vị trí các phần tử của listV, listW theo listValuePerWeights
  - ❖ New rootnode, tính giới hạn dưới của rootnade
  - ❖ BB\_tree.append(rootnode)
  - ❖ BB\_stack.append(rootnode)
  - ❖
  - ❖ // tạo các nút con và cập nhật các giới hạn giá trị. Nếu BB\_stack đã rỗng, giải thuật kết thúc.
  - ❖ while BB\_stack:
    - ❖ Branch()
  - ❖ // thực hiện việc tạo các node con từ node gốc Root, tạo các node con tương ứng với việc chọn hoặc không chọn
  - ❖
  - ❖ Node = BB\_solution

- ❖ // tìm nút lá có giá trị lớn nhất và truy xuất lại danh sách các đồ vật đã chọn (taken) thông qua các nút cha của nút đó.
- ❖ while Node.ParentNode:
- ❖     taken[value\_per\_weight[Node.ObjectID][0]] = Node.Taken
- ❖     Node = Node.ParentNode
- ❖
- ❖ // vì ta đã thay đổi các giá trị value khi chọn các phần tử tối ưu nhất của mỗi class nên giá trị maxvalue mà hàm tìm ra là không đúng nên ta cần tìm lại maxvalue thông qua listtaken.
- ❖
- ❖ maxvalue = 0
- ❖     for i in range(len(vtemp)):
- ❖         if taken[i] == 1:
- ❖             maxvalue += vtemp[i]
- ❖
- ❖ return (maxvalue, taken)
  - Function Branch(items, values, weights, value\_per\_weight):
- ❖ Lấy node gốc từ BB\_stack
- ❖ Nếu giá trị giới hạn của node gốc nhỏ hơn giá trị tốt nhất hiện tại thì quay lại
- ❖ Ngược lại nếu Root là node lá thì quay lại
- ❖ Tạo node tiếp theo và tính toán giới hạn trên giá trị của túi cho node đó
- ❖ Nếu Node vừa tạo là node lá và có giá trị tốt hơn BB\_solution thì cập nhật BB\_solution
- ❖ Thêm node đó vào BB\_tree

## Tóm tắt chương trình:

- ❖ Hàm *load\_input\_file(input\_file)* đọc dữ liệu từ file đầu vào và trả về các giá trị cần thiết để giải quyết bài toán.
- ❖ *BTreeNode*: Là một lớp đại diện cho một nút trong cây nhị phân. Nó có một số thuộc tính như parent node, relaxation value, objective value, object ID, và liệu đối tượng đó có được chọn hay không.

- ❖ *BB\_solver(capacity, weights, values, classes, and number of classes)*: Là hàm chính. trả về giá trị tối đa có thể đạt được và danh sách chỉ ra đối tượng nào được chọn.
- ❖ *Branch(số lượng vật phẩm, values, weights và values/weights)*: Nó cập nhật cây nhị phân.
- ❖ *getBound(số lượng vật phẩm, rootid, root\_room, root\_objective, weights, values, value\_per\_weight)*: Là một hàm tính toán giới hạn giá trị của một nút đã cho trong cây nhị phân.

## Nhận xét:

- Branch and Bound là một giải thuật tốt trong việc giải quyết các bài toán tối ưu hóa. Giải thuật này được thiết kế để khám phá không gian giải pháp và cắt tỉa các nhánh không có triển vọng để cải thiện hiệu quả.
- Độ hiệu quả của Branch and Bound phụ thuộc vào chất lượng của các giới hạn dưới được sử dụng để ước lượng giải pháp. Một giới hạn dưới tốt có thể loại bỏ các nhánh không có triển vọng.

## Đánh giá:

- Branch and Bound là một thuật tốt để giải quyết bài toán Knapsack. Một trong những lợi ích của Branch and Bound là đảm bảo tìm được giải pháp tối ưu cho bài toán Knapsack.
- Nhưng khi thêm yêu cầu mỗi Class phải có ít nhất 1 vật phẩm thì chương trình code của em đôi khi đã trả ra 1 số kết quả không tối ưu. Lý do:
  - ❖ Giải thuật của em là đầu tiên ta sẽ chọn những vật phẩm có giá trị (value/weights) lớn nhất mỗi Class để thêm vào túi trước dẫn đến sai lệch về kết quả. Kết quả trả ra có thể không tối ưu nhưng thời gian chạy là rất nhanh.
  - ❖ Giải thuật không thể đưa ra kết quả tối ưu khi, giá trị (value/weights) của 1 vật phẩm a trong lớp A là rất lớn nhưng tổng số weights của vật này cũng rất lớn. Làm tốn rất nhiều trọng lượng tối đa của túi, trong khi ở Class khác có nhiều vật khác có (value/weights) tốt hơn vật a nhưng không còn đủ trọng lượng của túi để chứa đựng.
  - ❖ VD: cho test case
    - 30
    - 2
    - 20, 10, 10, 10

40, 15, 30, 35

1, 1, 2, 2

Đáp án tối ưu cho bài toán là : 80, [0, 1, 1, 1].

Nhưng kết quả mà chương trình của em trả ra là: 75, [1, 0, 0, 1]. Vì ở Class 1 vật phẩm đầu có giá trị (value/weights) =2 “lớn nhất trong Class 1, nên sẽ bị ưu tiên được thêm vào túi dẫn đến sai lệch kết quả.”

- ❖ Nhưng thuật toán này cũng có 1 số ưu điểm lớn. Với những test case có nhiều vật phẩm và Class thì thời gian chạy của chương trình là rất nhỏ. Bên cạnh đó cũng có 1 số khuyết điểm như hàm thực thi tốn thêm 2 mảng để lưu và chọn ra những vật phẩm có (value/weights) lớn. Và tốn thêm 11 hàm sort khi thực thi chương trình.

## Kết luận:

- Branch and Bound là một thuật tốt để giải quyết bài toán Knapsack. Một trong những lợi ích của Branch and Bound là đảm bảo tìm được giải pháp tối ưu cho bài toán Knapsack.
- Nhưng trong trường hợp này thì giải thuật của nhóm em sẽ phù hợp cho những test case lớn mà cần thời gian thực thi nhanh.

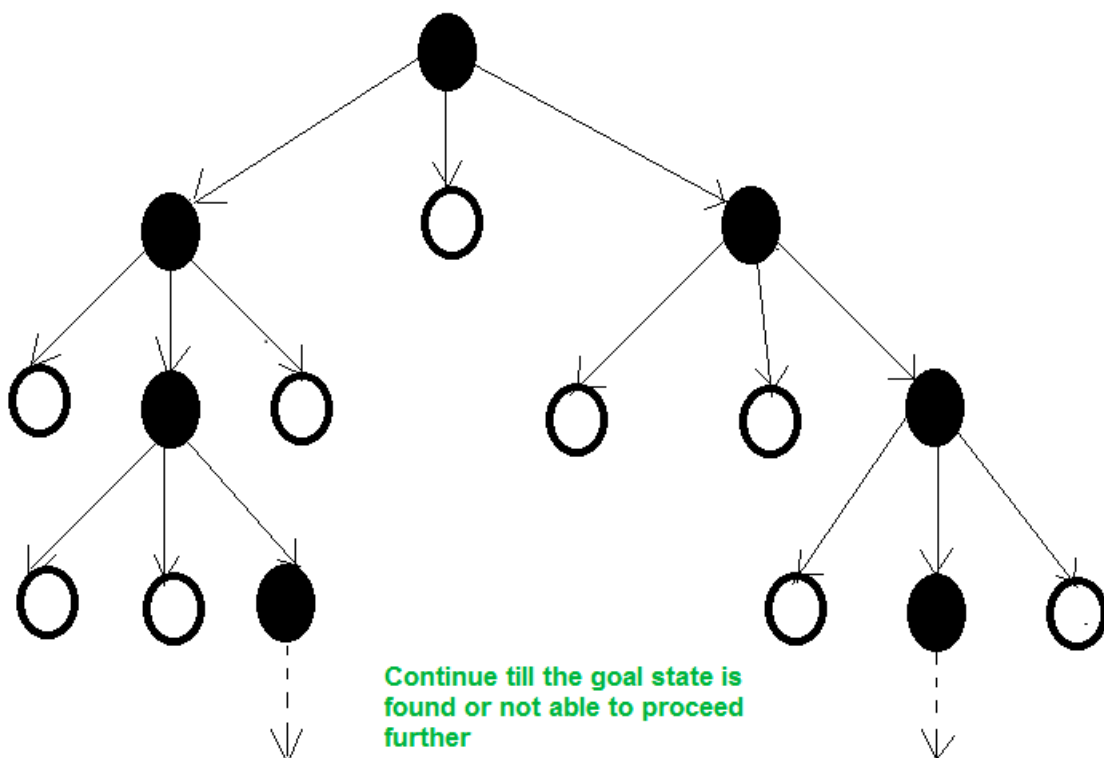
### 3. Local Beam Search

#### Ý tưởng thuật toán:

Bắt đầu bằng việc khởi tạo  $k$  các trạng thái ngẫu nhiên, sau đó tiếp tục tính trạng thái tiếp theo bằng việc thực hiện các thao tác như thêm hoặc bỏ đi một hoặc nhiều món hàng, hoặc sắp xếp lại thứ tự của các món hàng. Sau khi có được trạng thái mới, chúng ta tiến hành đánh giá chúng rồi chọn tiếp  $k$  trạng thái cho lần tính tiếp theo. Quá trình sẽ kết thúc khi ta có được giải pháp tối ưu hoặc hoàn thành số vòng lặp đã được cài đặt từ trước.

Tuy nhiên, thuật toán có thể trả về kết quả không tối ưu vì nó sẽ lựa chọn  $k$  trạng thái tốt nhất gần bằng nhau hoặc nằm gần nhau trong không gian tìm kiếm, có thể thuật toán sẽ bị mắc kẹt ở local optimum.

Ta có thể tránh việc này bằng nhiều cách khác nhau, như tăng  $k$  để giảm khả năng rơi vào trường hợp trên, thêm các yếu tố ngẫu nhiên vào quá trình tìm kiếm (tìm kiếm lại ở một trạng thái ngẫu nhiên khác,..) hoặc sử dụng thuật toán tối ưu hơn như simulated annealing, genetic algorithm.



*Tạo ra các trạng thái mới, lựa chọn và tiếp tục tạo mới cho tới khi đạt được goal hoặc hết vòng lặp*

(Nguồn <https://www.geeksforgeeks.org/introduction-to-beam-search-algorithm/>)

## Mã giả:

- Bước 1: Khởi tạo k trạng thái ngẫu nhiên.
- Bước 2: Lặp lại quá trình sau cho đến khi tìm được trạng thái tối ưu:
  - Bước 2.1: Tạo ra tất cả các trạng thái con của k trạng thái hiện tại.
  - Bước 2.2: Nếu bất kỳ trạng thái nào trong các trạng thái mới tạo được bằng với trạng thái tối ưu, thì trả về trạng thái đó.
  - Bước 2.3: Chọn k trạng thái con tốt nhất và sử dụng chúng để tạo ra k trạng thái mới cho lần lặp tiếp theo.
- Bước 3: Trả về trạng thái tối ưu tìm được.

## Tóm tắt chương trình:

- ❖ Hàm `read_input_file(input_file)` đọc dữ liệu từ file đầu vào và trả về các giá trị cần thiết để giải quyết bài toán.
- ❖ Hàm `write_output_file(output_file, best_fitness, best_state)` ghi kết quả vào file đầu ra.
- ❖ Hàm `check_types(new_solution, types)` kiểm tra điều kiện đủ class ở mỗi trạng thái
- ❖ Hàm `minimum_list_types(numType, weights, types, capacity)` kiểm tra tính khả dĩ của lời giải khi lấy ít nhất mỗi class 1 loại
- ❖ Hàm `knapsack_local_beam(numType, weights, values, types, capacity, beam_width, max_iterations)` là hàm điều khiển chính của thuật toán, bắt đầu bằng việc khởi tạo 1 danh sách thể hiện việc lựa chọn đồ vật (0 hoặc 1), sau đó tiến hành loại bỏ hoặc thêm bớt và kiểm tra tính hợp lệ của trạng thái. Chúng được sắp xếp lại theo giá trị giảm dần và k trạng thái tốt nhất sẽ được lựa chọn cho lần sản sinh tiếp theo. Nếu có trạng thái tốt hơn, chúng sẽ thay thế cho trạng thái hiện tại hoặc chương trình sẽ dừng.
- ❖ Hàm `main()` là hàm chạy chính của chương trình. Nó đọc dữ liệu từ file đầu vào, khởi tạo quần thể ban đầu, thực hiện thuật toán tìm kiếm cục bộ và ghi kết quả tối ưu vào file đầu ra.

## Nhận xét:

Độ phức tạp của thuật toán Local Beam Search trong bài toán Knapsack sẽ phụ thuộc vào các yếu tố như kích thước của quần thể (population), số lần lặp lại (num\_iterations), và độ rộng của bước beam (beam\_width).

- Trong trường hợp xấu nhất, chương trình có thể phải kiểm tra tất cả các tổ hợp của các đồ vật, vì vậy độ phức tạp là  $O(2^n)$
- Trong hàm knapsack\_local\_beam, ta có vòng lặp bên ngoài chạy tối đa max\_iterations lần. Trong mỗi lần lặp này, ta thực hiện beam\_width lần tìm kiếm. Trong mỗi lần tìm kiếm, ta sẽ tạo ra numType phiên bản khác nhau của giải pháp hiện tại bằng cách chọn một item và đổi trạng thái của nó (tức là bỏ vào hoặc bỏ ra khỏi knapsack). Do đó, độ phức tạp của vòng lặp này là  $O(\text{numType} * n * \text{beam\_width})$ .
- Do đó, tổng độ phức tạp của chương trình sẽ là  $O(\text{numType} * n * \text{beam\_width} * \text{max\_iterations})$ .
- **Worst case** của thuật toán sẽ xảy ra với ví dụ với input gồm 3 items và capacity = 10 như sau:
  - Item 1: weight=5, value=10
  - Item 2: weight=6, value=12
  - Item 3: weight=7, value=14

Với trường hợp ta đặt giá trị của  $k = 2$ , và 2 trạng thái đầu tiên sinh ra chỉ chứa vật phẩm 1, sau đó thuật toán tiến hành sinh trạng thái mới từ 2 trạng thái ban đầu (không thỏa mãn vì sẽ vi phạm yêu cầu đề bài,  $\text{weight} > \text{capacity}$ )

Trong trường hợp xấu nhất, thuật toán sẽ tiếp tục sinh trạng thái mới của  $k$  giải pháp chỉ có vật phẩm 1, mà không tìm thấy trạng thái nào thỏa mãn ràng buộc dung lượng. Việc này sẽ tiếp tục cho đến khi đạt đến số lần lặp tối đa, và thuật toán sẽ trả về trạng thái tốt nhất được tìm thấy, đó là chỉ có vật phẩm 1.

- **Best case** của thuật toán sẽ xảy ra với ví dụ: capacity = 10 và chỉ có một item với trọng lượng là 5 và giá trị là 10, giải thuật local beam

search sẽ tìm được giải pháp tối ưu với một bước duy nhất là chọn món hàng đó để đưa vào knapsack. Kết quả là giải pháp tối ưu với giá trị là 10 và trọng lượng là 5.

## Minh họa:

với input như sau,  $k=3$  và  $\text{max\_iteration} = 3$ :

50.0

20

5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0

10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20

Bước 0: Khởi tạo ngẫu nhiên  $k=3$  giải pháp khác nhau:

{1, 2, 3, 4, 5}

{6, 7, 8, 9, 10}

{11, 12, 13, 14, 15}

Bước 1: Tính toán hàm mục tiêu cho từng giải pháp:

{1, 2, 3, 4, 5}: Trọng lượng: 25.0, giá trị: 50

{6, 7, 8, 9, 10}: Trọng lượng: 25.0, giá trị: 50

{11, 12, 13, 14, 15}: Trọng lượng: 25.0, giá trị: 50

Bước 2: Tìm  $k$  giải pháp tốt nhất và sinh ra các giải pháp mới từ chúng:

{1, 2, 3, 4, 5}: sinh ra {1, 2, 3, 4, 6}, {1, 2, 3, 4, 7}, {1, 2, 3, 4, 8},  
{1, 2, 3, 4, 9}, {1, 2, 3, 4, 10}

{6, 7, 8, 9, 10}: sinh ra {6, 7, 8, 9, 11}, {6, 7, 8, 9, 12}, {6, 7, 8, 9,  
13}, {6, 7, 8, 9, 14}, {6, 7, 8, 9, 15}

.....

Vòng lặp tiếp theo sẽ tiếp tục tính trọng lượng và giá trị của từng trạng thái, sau đó chọn ra trạng thái tốt nhất .



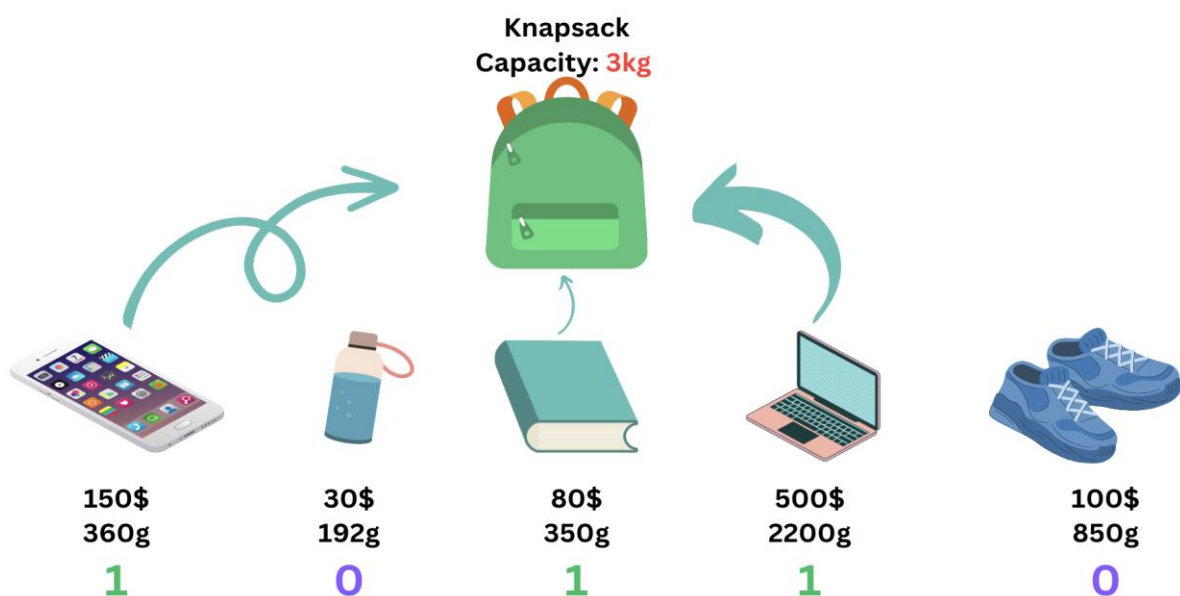
## 4. Genetic Algorithms

### Ý tưởng thuật toán:

Mở đầu với một số lời giải được tạo ra một cách ngẫu nhiên (gọi là các cá thể). Cùng với nhau, tạo nên một quần thể. Quần thể mở đầu được gọi là thế hệ thứ nhất (*The Zeroth Generation*) giữ trọng trách duy trì thế hệ và sau đó thực hiện một số cuộc cách mạng tiến hóa như các quá trình chọn lọc (*Selection*), đột biến (*Mutation*),... thông qua sự đánh giá mức độ xứng đáng của mỗi cá thể (*Fitness Evaluation*) để chọn ra những cá thể phù hợp cho quá trình tạo ra các thế hệ tiếp theo. Tiếp tục vòng lặp cho đến khi tìm được cá thể (lời giải) phù hợp cho yêu cầu bài toán đưa ra.

❖ Biểu hiện cho một cá thể (*Individual Representation*): Để cho dễ hình dung, ta biểu hiện lời giải của bài toán Knapsack là một dãy bit có độ dài bằng với số đồ vật để ở bên ngoài với mỗi bit là trạng thái “có xuất hiện trong ba lô hay không”:

- Bit 0: tượng trưng cho việc ta không chọn đồ vật đó
- Bit 1: tượng trưng cho việc ta chọn đồ vật đó bỏ vào ba lô



❖ Tạo ra thế hệ quần thể đầu tiên (*Initial Generation Creation*): Ngẫu nhiên tạo các lời giải (cá thể) bằng cách đánh ngẫu nhiên các bit trong một cá thể sao cho đủ số lượng các cá thể trong quần thể và các cá thể đó phải khác nhau. Để các cá thể được chọn

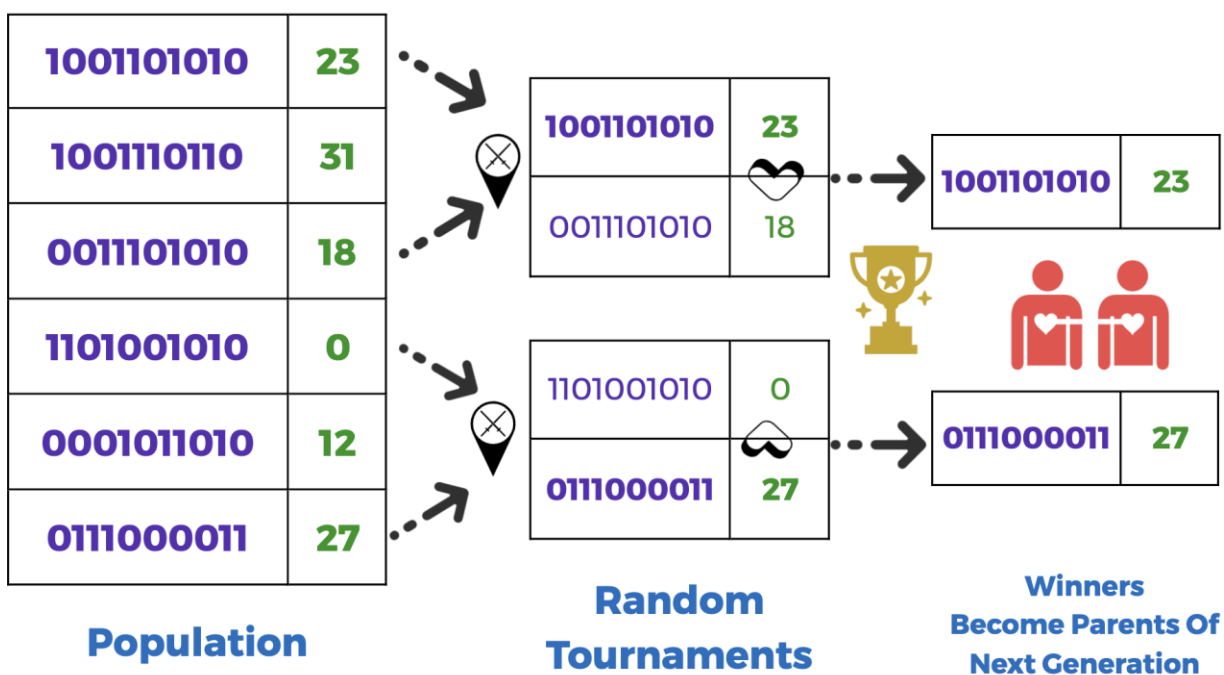
hữu ích cho các thế hệ quần thể tiếp theo (thỏa yêu cầu bài toán đưa ra), ta chỉ chọn các cá thể có mức độ phù hợp khác 0.

❖ Đánh giá mức độ phù hợp của một cá thể (*Fitness Evaluation*): Đối với bài toán Knapsack được yêu cầu, một cá thể (lời giải) được coi là xứng đáng khi thỏa các điều kiện sau đây:

- Tổng khối lượng của các đồ vật trong ba lô hiển nhiên không được vượt quá dung lượng chứa của chiếc ba lô, nếu vi phạm thì điểm phù hợp (fitness score) sẽ bằng 0.
- Do yêu cầu đề bài có 1 ràng buộc rằng ba lô phải chứa ít một đồ vật của mỗi loại vì vậy nếu trong lời giải chứa thiếu đồ vật của một loại đồ vật thì sẽ vi phạm và sẽ đạt được điểm phù hợp bằng 0.
- Tổng số điểm phù hợp sẽ là tổng giá trị của các đồ vật mang trong ba lô nếu như không vi phạm 2 điều kiện trên.

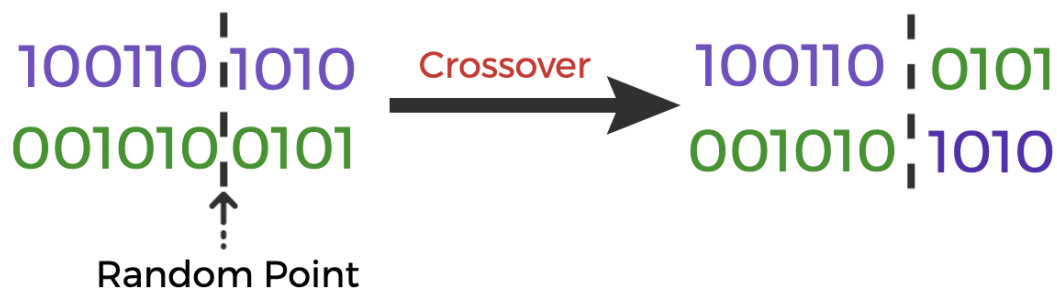
❖ Quá trình chọn lọc (*Selection*): Đối với thuật toán này, ta sẽ chọn lọc bằng cách thi đấu (*Tournament Selection*) thông qua số điểm phù hợp (fitness score) được tính như trên.

- Chọn ngẫu nhiên 2 cặp cá thể trong thế hệ quần thể để đấu nhau.
- Mỗi cặp thi đấu với nhau bằng cách xem fitness score của ai cao hơn thì sẽ giành chiến thắng.
- Người chiến thắng (lời giải) của mỗi cặp đấu sẽ được chọn để làm cha mẹ cho các thế hệ sau.



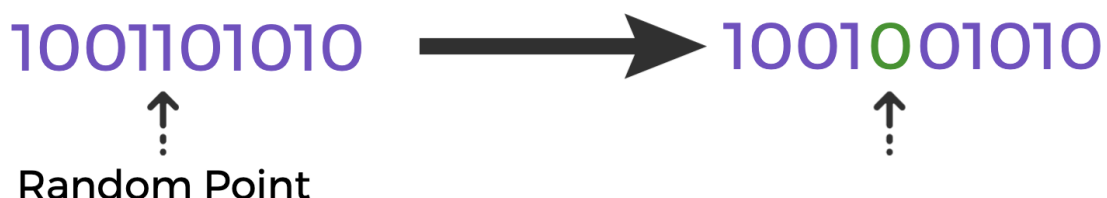
❖ Quá trình hoán đổi gen của một cặp cá thể (**Crossover**): Nếu như xem lời giải (một dãy bit) của bài toán là một đoạn gen thì đây là quá trình hoán đổi gen của 2 cá thể được chọn. Tất nhiên, điều này có thể dẫn đến việc biến thiên các giá trị fitness score của các thế hệ (tăng hoặc giảm).

- Chọn ngẫu nhiên một điểm trên 2 đoạn gen của từng cá thể.
- Cắt đoạn gen của từng cá thể ra thành 2 ở điểm được chọn.
- Bắt chéo 4 đoạn gen vừa mới được hình thành, nối đầu của đoạn này với đuôi của đoạn kia và ngược lại.
- Ta nhận được 2 đoạn gen (lời giải) mới cũng chính là 2 cá thể con cho thế hệ tiếp theo.



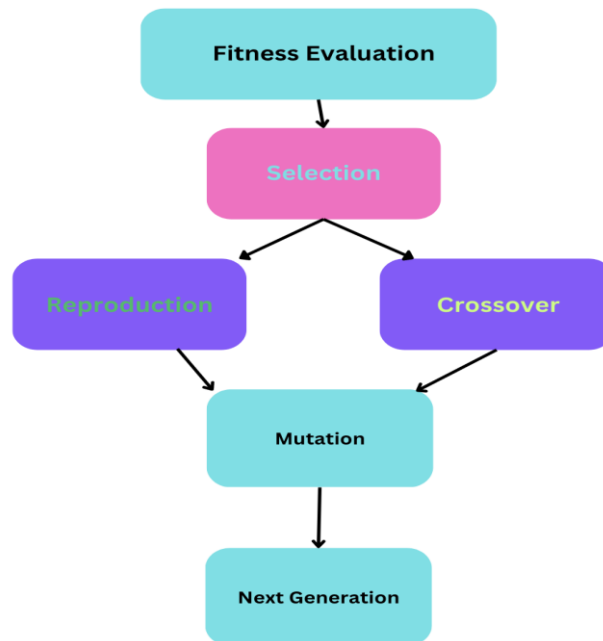
❖ Sự đột biến (**Mutation**): Là quá trình thay đổi trạng thái của một đồ vật được chọn ngẫu nhiên trong ba lô dưới một mức độ đột biến xác định (Mutation Probability).

- Chọn ngẫu nhiên một bit trong cá thể (lời giải được biểu diễn dưới dạng một dãy bit).
- Lật ngược bit đó lại so với trạng thái ban đầu, nếu bit đó là 0 (chưa chọn đồ vật đó) thì trở thành 1 (thêm đồ vật đó vào ba lô) và ngược lại.



❖ Tái sử dụng các cá thể trong quần thể (**Reproduction**): Do quá trình hoán đổi gen (Crossover) hay sự đột biến (Mutation) như trên có thể gây ra sự biến thiên đối với mức độ phù hợp (Fitness Score) được đánh giá cho các thế hệ sau, ta sẽ tránh sự việc này bằng cách duy trì các cá thể trong thế hệ quần thể cũ ở thế hệ mới.

- ❖ Quá trình tạo thế hệ mới (*Generations Creation*): là quá trình lặp đi lặp lại các quá trình chọn lọc (Selection), tái sử dụng (Reproduction), hoán đổi gen (Crossover) và đột biến (*Mutation*) theo trình tự cho đến khi tạo đủ các cá thể cho thế hệ quần thể mới.



## Mã giả:

- Bước 1: Tạo ra một thế hệ quần thể ban đầu (*The Zeroth Generation*) bằng cách ngẫu nhiên tạo ra các lời giải khác nhau dựa trên việc đánh giá sự phù hợp sao cho đủ kích thước quần thể.
- Bước 2: Bước vào vòng lặp tạo ra các thế hệ mới cho đến mức giới hạn số thế hệ quần thể được quy định bởi lập trình viên.
  - Bước 2.1: Lần lượt bốc 1 cặp cá thể thông qua chọn lọc thi đấu (*Tournament Selection*) của 2 cá thể trong 1 thế hệ.
  - Bước 2.2: Cặp cá thể được chọn sẽ có 2 lựa chọn thông qua xác suất ngẫu nhiên là giữ nguyên thế hệ cũ (*Reproduction*) hay hoán đổi gen cho nhau để tạo ra cặp cá thể mới (*Crossover*).
  - Bước 2.3: Cặp cá thể mới này sẽ thực hiện sự đột biến (*Mutation*) thông qua xác suất ngẫu nhiên.
  - Bước 2.4: Thêm cặp cá thể mới này vào thế hệ quần thể mới và kết thúc vòng lặp.
- Bước 3: Thế hệ cuối cùng được xem là có những cá thể (lời giải) xứng đáng (phù hợp hay có fitness score cao) nhất.

- Bước 4: Chọn trong thế hệ này ra được cá thể phù hợp nhất và đây cũng là lời giải của bài toán.

### Nhận xét:

- Độ phức tạp thời gian của thuật toán Genetic Search trong bài toán Knapsack sẽ phụ thuộc vào một số yếu tố nhất định như kích cỡ của quần thể (Pop\_Size), mức giới hạn của số thế hệ quần thể (Gen\_Num) và độ phức tạp của hàm đánh giá độ phù hợp (Fitness\_Func\_Complexity). Độ phức tạp của cả thuật toán là  $O(\text{Pop\_Size} * \text{Gen\_Num} * \text{Fitness\_Func\_Complexity})$ . Tuy nhiên, việc tăng các đối số như Pop\_Size và Gen\_Num sẽ giúp ích rất nhiều trong việc tạo ra lời giải tốt nhất do càng tạo ra nhiều thế hệ đồng nghĩa với việc các cá thể sẽ càng phù hợp hơn.
- Cách cài đặt thuật toán này sẽ dài hơn hầu hết các thuật toán phía trên và được xem là khá phức tạp. Tuy nhiên, sẽ rất dễ hiểu cách cài đặt này khi nhân hóa bài toán.

## 5. SO SÁNH CÁC THUẬT TOÁN

**Knapsack** là bài toán tìm lời giải tối ưu nổi tiếng, được sử dụng để đo đặc sức mạnh tính toán của hệ thống và tính tối ưu của giải thuật. Trong 4 giải thuật của project (*Brute-force*, *Branch and Bound*, *Local Beam Search*, *Genetic Algorithm*) thì:

- khi làm việc với dataset nhỏ, Brute-force sẽ là thuật toán tối ưu vì nó có khả năng tìm kiếm toàn bộ trong không gian tìm kiếm, từ đó sẽ đưa ra lời giải tối ưu nhất trong một khoảng thời gian chấp nhận.
- local beam search và genetic algorithm sẽ là thuật toán tối ưu khi chúng ta không cần yêu cầu độ chính xác cao của lời giải và thời gian bị giới hạn
- branch and bound sẽ là thuật toán tối ưu nhất khi làm việc với dataset nhỏ, nó có khả năng tìm kiếm của brute-force và tối ưu tốt hơn bởi khả năng loại bỏ những lời giải không tốt -> giảm thời gian thực thi nhưng đảm bảo độ chính xác.
- ngược lại khi làm việc với dataset lớn, brute-force và branch and bound sẽ gặp khó bởi không gian tìm kiếm quá rộng
- local beam search và genetic algorithm sẽ là lựa chọn tối ưu hơn bởi phương pháp tìm kiếm heuristic sẽ giảm thời gian và không gian tìm kiếm đi đáng kể, nhưng sẽ đánh đổi đi độ chính xác của lời giải

Các liên kết của các video chạy các thuật toán với các data set đã được tạo:

1. Brute Force: <https://www.youtube.com/watch?v=NTRAN98ksMo>
2. Branch & Bound: [https://www.youtube.com/watch?v=hPzoBVb\\_5gE](https://www.youtube.com/watch?v=hPzoBVb_5gE)
3. Local Beam Search: <https://www.youtube.com/watch?v=Ap3JtDx4DBc>
4. Genetic Search: <https://www.youtube.com/watch?v=G1axwmItemE>