TỔNG LIÊN ĐOÀN LAO ĐỘNG VIỆT NAM
TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG
KHOA CÔNG NGHỆ THÔNG TIN

BÁO CÁO CUỐI KỲ MÔN
PHÂN TÍCH THIẾT KẾ VÀ GIẢI THUẬT

TÌM HIỂU VỀ MỘT SỐ KỸ THUẬT PHÂN TÍCH VÀ THIẾT KẾ GIẢI THUẬT

Giảng viên giảng dạy: **ThS. Trần Lương Quốc Đại**
Sinh viên thực hiện: **Tô Vĩnh Khang - 51800408**
Bùi Quang Khải - 51800785
Du Thuận Long - 51800429
Nhóm: 04
Khóa: 22
Lớp: 18050203

THÀNH PHỐ HỒ CHÍ MINH, NĂM 2020

TỔNG LIÊN ĐOÀN LAO ĐỘNG VIỆT NAM
TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG
KHOA CÔNG NGHỆ THÔNG TIN

BÁO CÁO CUỐI KỲ MÔN
PHÂN TÍCH THIẾT KẾ VÀ GIẢI THUẬT

TÌM HIỂU VỀ MỘT SỐ KỸ THUẬT PHÂN TÍCH VÀ THIẾT KẾ GIẢI THUẬT

Giảng viên giảng dạy: **ThS. Trần Lương Quốc Đại**
Sinh viên thực hiện: **Tô Vĩnh Khang - 51800408**
Bùi Quang Khải - 51800785
Du Thuận Long - 51800429
Nhóm: 04
Khóa: 22
Lớp: 18050203

THÀNH PHỐ HỒ CHÍ MINH, NĂM 2020

LỜI CẢM ƠN

Chúng em xin chân thành cảm ơn Khoa Công nghệ thông tin và Trường Đại học Tôn Đức Thắng đã tạo điều kiện cho chúng em được học tập trong suốt thời gian qua. Nhờ có sự giảng dạy tận tình, nhiệt huyết của thầy Trần Lương Quốc Đại đã giúp chúng em có thêm kiến thức, hiểu rõ hơn về cách phân tích một thuật toán thông qua việc đọc hiểu mã giả, tính toán độ phức tạp, tối ưu thuật toán,… Giúp chúng em mở mang kiến thức về một số mô hình như tìm kiếm toàn diện, chia để trị, thuật toán tham lam, quy hoạch động, thuật toán xấp xỉ cũng như các thuật toán tìm kiếm đồ thị và cây. Chân thành cảm ơn thầy.

BÀI BÁO CÁO CUỐI KỲ NÀY ĐƯỢC HOÀN THÀNH TẠI TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG

Chúng em xin cam đoan đây là sản phẩm của riêng chúng em. Các nội dung, kết quả được trình bày đều là trung thực.

Nếu phát hiện có bất kỳ sự gian lận nào chúng em xin hoàn toàn chịu trách nhiệm về nội dung bài cuối kỳ của mình. Trường đại học Tôn Đức Thắng không liên quan đến những vi phạm tác quyền, bản quyền do chúng em gây ra trong quá trình thực hiện (nếu có).

TP. Hồ Chí Minh, ngày 22 tháng 10 năm 2020

Tác giả

(ký tên và ghi rõ họ tên)

Tô Vĩnh Khang (Trưởng nhóm)

Bùi Quang Khải

Du Thuận Long

PHẦN XÁC NHẬN VÀ ĐÁNH GIÁ CỦA GIẢNG VIÊN

Phần xác nhận của GV hướng dẫn

_____

_____

_____

_____

_____

_____

_____

Tp. Hồ Chí Minh, ngày tháng năm

(kí và ghi họ tên)

Phần đánh giá của GV chấm bài

_____

_____

_____

_____

_____

_____

_____

Tp. Hồ Chí Minh, ngày tháng năm

(kí và ghi họ tên)

MỤC LỤC

DANH MỤC HÌNH

CHƯƠNG I: KỸ THUẬT TÌM KIẾM TOÀN DIỆN (BRUTE-FORCE)

Là kỹ thuật giải quyết vấn đề rất chung và mô hình thuật toán là liệt kê một cách có hệ thống tất cả các ứng viên có thể có và kiểm tra xem mỗi ứng viên có đáp ứng yêu cầu của bài toán hay không. Nó có thể được coi là siêu mô phỏng đơn giản nhất. Ở chương này, thuật toán Sắp xếp chọn (Selection Sort), Sắp xếp nổi bọt (Bubble Sort) và Tìm kiếm tuần tự (Sequence Search) được sử dụng làm ví dụ minh họa cho kỹ thuật tìm kiếm toàn diện này.

1.1 Ý tưởng thuật toán

1.1.1 Thuật toán Sắp xếp chọn (Selection Sort)

Dựa trên việc so sánh tại chỗ. Chọn phần tử nhỏ nhất trong N phần tử ban đầu, đưa phần tử này về vị trí đúng là đầu tiên của mảng hiện tại. Sau đó không quan tâm đến nó nữa, xem mảng hiện tại chỉ còn N-1 phần tử của mảng ban đầu, bắt đầu từ vị trí thứ 2. Lặp lại quá trình trên cho mảng hiện tại đến khi mảng hiện tại chỉ còn 1 phần tử. Tóm lại, thuật toán sẽ thực hiện N-1 lượt việc đưa phần tử nhỏ nhất trong mảng hiện tại về vị trí đúng ở đầu dãy.

Mô tả thuật toán:

Bước 1 : Đặt min thành vị trí đầu tiên

Bước 2 : Tìm kiếm phần tử nhỏ nhất trong danh sách

Bước 3 : Hoán đổi với giá trị tại vị trí min

Bước 4 : Tăng min lên 1 để trỏ đến phần tử tiếp theo

Bước 5 : Lặp lại cho đến khi danh sách được sắp xếp

Mã giả:

```
SelectionSort(A):
    N = len(A)
    For i=0 to N-1 do:
        min = i
        For j=i+1 to N-1 do:
            If(A[min] > A[j]) then:
                min = j
            EndIf
        EndFor
        Swap(A[i], A[min])
    EndFor
End
```

Chi tiết cách thức hoạt động của thuật toán Selection Sort:

Giả sử ta có mảng A = [7, 4, 8, 2, 1]

|	7	4	8	2	**1**
1	|	4	8	**2**	7
1	2	|	8	**4**	7
1	2	4	|	8	**7**
1	2	4	7	|	**8**
1	2	4	7	8	|

1.1.2 Thuật toán Sắp xếp nổi bọt (Bubble Sort)

So sánh 2 phần tử kề nhau, nếu chúng chưa đứng đúng thứ tự thì đổi chỗ với nhau thông qua mỗi lần duyệt. Nếu trong một lần duyệt nào đó mà không phải đổi chỗ bất cứ cặp phần tử nào thì danh sách đã được sắp xếp xong.

Mô tả thuật toán:

Bước 1: Bắt đầu từ phần tử i ngoài cùng bên trái của danh sách và so sánh với phần tử kế tiếp nó.

Bước 2: Nếu lớn hơn, hoán vị chúng.

Ngược lại, tăng i lên 1 để trỏ đến phần tử tiếp theo.

Bước 3: Lặp lại đến khi không còn sự hoán vị ta được danh sách đã sắp xếp.

Mã giả:

```
BubbleSort(A):
    N = len(A)
    For i=0 to N-2 do:
        For j=0 to N-2-i do:
            If(A[j] > A[j+1]) then:
                Swap(A[j], A[j+1])
            EndIf
        EndFor
    EndFor
End
```

Chi tiết cách thức hoạt động của thuật toán Bubble Sort:

Giả sử ta có mảng A = [7, 4, 8, 2, 1]

7 <?> 4 8 2 1 |

4 7 <?> 8 <?> 2 1 |

4 7 2 8 <?> 1 |

4 7 2 1 | 8

4 <?> 7 <?> 2 1 | 8

4 2 7 <?> 1 | 8

4 2 1 | 7 8

4 <?> 2 1 | 7 8

2 4 <?> 1 | 7 8

2 1 | 4 7 8

2 <?> 1 | 4 7 8

1 | 2 4 7 8

| 1 2 4 7 8

1.1.3 Thuật toán Tìm kiếm tuần tự (Sequential Search)

Là một phương pháp tìm kiếm một phần tử cho trước trong một danh sách bằng cách duyệt lần lượt từng phần tử của danh sách đó cho đến lúc tìm thấy giá trị mong muốn hay đã duyệt qua toàn bộ danh sách. Giải thuật này tỏ ra khá hiệu quả khi cần tìm kiếm trên một danh sách đủ nhỏ hoặc một danh sách chưa sắp thứ tự đơn giản.

Mô tả thuật toán:

Bước 1: Bắt đầu từ phần tử ngoài cùng bên trái của danh sách (i=0) và so sánh giá trị cần tìm key với giá trị của phần tử thứ i.

Bước 2: Nếu key khớp phần tử thứ i trong danh sách, trả về i.

Ngược lại, tăng i lên 1 để trỏ đến vị trí tiếp theo

Bước 3: Lặp lại bước 1 đến khi duyệt hết phần tử thứ i=N. Nếu không tìm thấy phần tử thứ i nào khớp với key, trả về kết quả không tìm được. Kết thúc.

Mã giả:

```
SequentialSearch(A,key):
    i = 0
    N = len(A)
    While (i < N) do:
        If(A[i] = = key) then:
            Return i
        Else:
            i+=1
        EndIf
    EndWhile
    Return Notfound
End
```

Chi tiết cách thức hoạt động của thuật toán Sequential Search:

Giả sử ta có mảng A = [7, 4, 8, 2, 1] và cần tìm giá trị key = 2

7	4	8	2	1

i=0; key?

7	4	8	2	1

i=1; key?

7	4	8	2	1
		i=2; key?		
7	4	8	**2**	1
			i=3; key!!	

1.2 Triển khai bằng Python3

1.2.1 Thuật toán Sắp xếp chọn (Selection Sort)

Yêu cầu:

Đầu vào: Mảng A cần sắp xếp

Đầu ra: Mảng A đã sắp xếp

Mục đích: Sắp xếp mảng theo thứ tự tăng dần bằng cách duyệt chọn phần tử nhỏ hơn vị trí phần tử đang xét và đổi chỗ nó. Giúp xác định rõ không gian chiếm dụng bộ nhớ trước và sẽ không chiếm thêm không gian bộ nhớ nữa. Đơn giản dễ sử dụng cũng như hiện thực nó.

Hiện thực bằng code:

```
def SelectionSort(A):
    N=len(A)
    for i in range(N):
        MIN = i
        for j in range(i+1,N):
            if(A[MIN] > A[j]):
                MIN = j
        A[i], A[MIN] = A[MIN], A[i]
```

```
    return A
```

1.2.2 Thuật toán Sắp xếp nổi bọt (Bubble Sort)

Yêu cầu:

Đầu vào: Mảng A cần sắp xếp

Đầu ra: Mảng A đã sắp xếp

Mục đích: Sắp xếp theo thứ tự tăng dần bằng cách lần lượt duyệt so sánh từng phần tử kế tiếp phần tử đang xét và sẽ hoán vị khi cần thiết để tạo sự ổn định cho thuật toán. Đồng thời giúp xác định rõ không gian chiếm dụng bộ nhớ trước và sẽ không chiếm thêm không gian bộ nhớ nữa. Đơn giản dễ dàng hiện thực và sử dụng.

Hiện thực bằng code:

```
def BubbleSort(A):
    N = len(A)
    for i in range(N-1):
        for j in range(N-1):
            if(A[j] > A[j+1]):
                A[j], A[j+1] = A[j+1], A[j]
    return A
```

1.2.3 Thuật toán Tìm kiếm tuần tự (Sequential Search)

Yêu cầu:

Đầu vào: Mảng A bất kỳ và giá trị key cần tìm

Đầu ra: Vị trí của giá trị key cần tìm hoặc thông báo không tìm thấy được

Mục đích: Đảm bảo duyệt không sót thông tin, thích hợp duyệt cho những danh sách nhỏ. Đơn giản dễ sử dụng và hiện thực.

Hiện thực bằng code:

```
def SequentialSearch(A,key):
    i = 0
    N = len(A)
    while (i<N):
        if(A[i]==key):
            return i
        else:
            i+=1
    return "Notfound"
```

1.3 Demo

1.3.1 Thuật toán Sắp xếp chọn (Selection Sort)

- **Mô phỏng nhỏ:**

```
import timeit
A1 = [2,8,6,9]
start1 = timeit.default_timer()
SelectionSort(A1)
end1 = timeit.default_timer()
res1 = end1-start1
A2 = [7,4,8,2,1,22,2,10,9,4,99]
start2 = timeit.default_timer()
SelectionSort(A2)
end2 = timeit.default_timer()
```

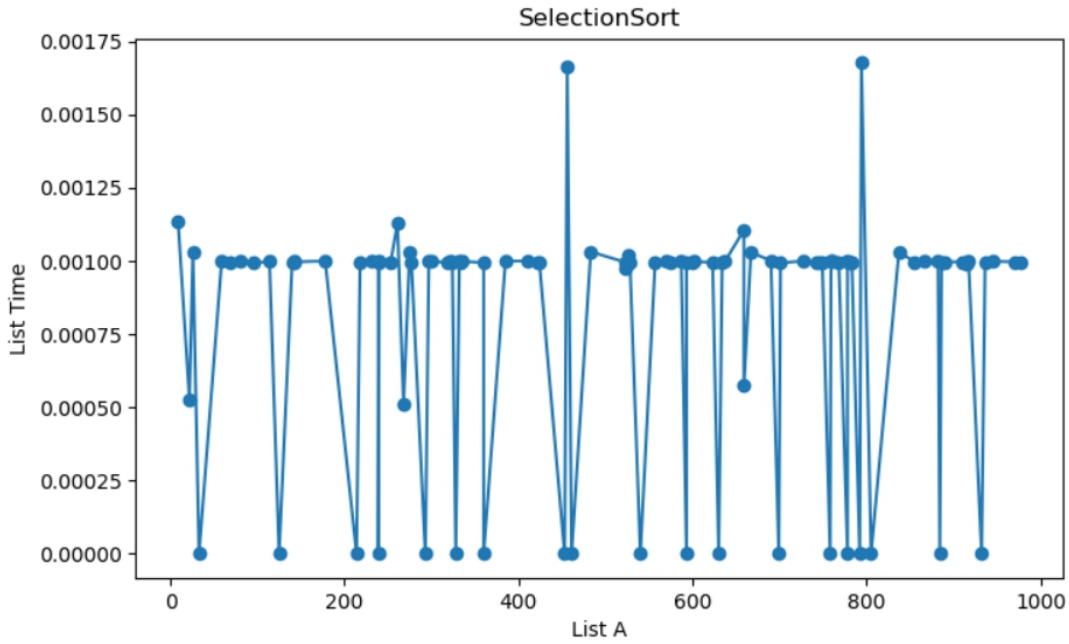
```
res2 = end2-start2
print('Thời gian sắp xếp mảng A1 :', res1)
print('Thời gian sắp xếp mảng A2 :', res2)
print('Thời gian chênh lệch      :', abs(res2-res1))
```

Kết quả từ màn hình Command Prompt:

```
Thời gian sắp xếp mảng A1 : 0.0009870000000000018
Thời gian sắp xếp mảng A2 : 0.0029560000000000003
Thời gian chênh lệch      : 0.0019689999999999985
```

- **Tổng quát hóa:**

  Cho một danh sách các số ngẫu nhiên từ 0-1000 gồm 100 phần tử, tiến hành tính thời gian chạy của từng lần sắp xếp bằng thuật toán SelectionSort. Xuất ra biểu đồ tổng quát về thời gian chạy của thuật toán sắp xếp này.



Hình 1.1 Demo thời gian chạy của thuật toán SelectionSort

Nhận xét:

- *Thời gian chạy khi sắp xếp mảng A1 nhỏ hơn khi sắp xếp mảng A2. Vậy thuật toán Selection Sort này thích hợp cho những mảng nhỏ.*
- *Thời gian chạy của thuật toán đa phần rơi vào khoảng 0.001*

1.3.2 Thuật toán Sắp xếp nổi bọt (Bubble Sort)

- **Mô phỏng nhỏ:**

  import timeit

  A1 = [2,8,6,9]

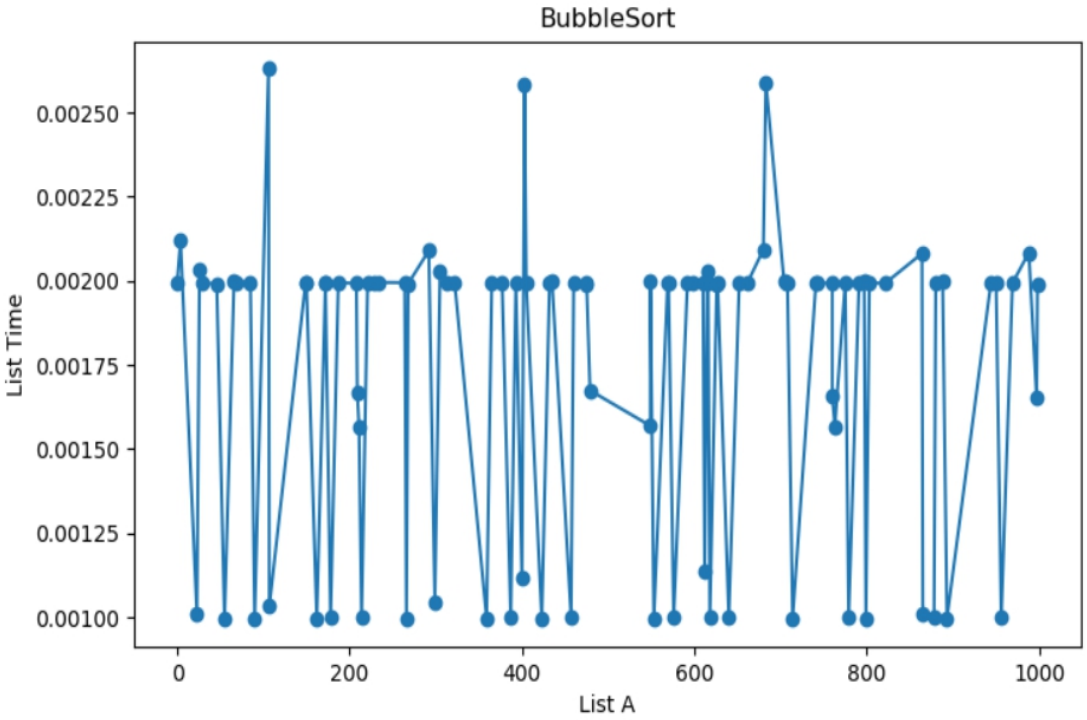
```
start1 = timeit.default_timer()
BubbleSort(A1)
end1 = timeit.default_timer()
res1 = end1-start1
A2 = [7,4,8,2,1,22,2,10,9,4,99]
start2 = timeit.default_timer()
BubbleSort(A2)
end2 = timeit.default_timer()
res2 = end2-start2
print('Thời gian sắp xếp mảng A1 :', res1)
print('Thời gian sắp xếp mảng A2 :', res2)
print('Thời gian chênh lệch      :', abs(res2-res1))
```

Kết quả từ màn hình Command Prompt:

Thời gian sắp xếp mảng A1 : 0.0035941999999999953
Thời gian sắp xếp mảng A2 : 0.0011977999999999989
Thời gian chênh lệch : 0.0023963999999999964

- **Tổng quát hóa:**

  Cho một danh sách các số ngẫu nhiên từ 0-1000 gồm 100 phần tử, tiến hành tính thời gian chạy của từng lần sắp xếp bằng thuật toán BubbleSort. Xuất ra biểu đồ tổng quát về thời gian chạy của thuật toán sắp xếp này.



Hình 1.2 Demo thời gian chạy của thuật toán BubbleSort

Nhận xét:

- *Thời gian chạy khi sắp xếp mảng A1 lớn hơn khi sắp xếp mảng A2. Vậy thuật toán Bubble Sort này thích hợp cho những mảng lớn.*
- *Thời gian chạy của thuật toán đa phần rơi vào khoảng 0.002*

1.3.3 Thuật toán Tìm kiếm tuần tự (Sequential Search)

- **Mô phỏng nhỏ:**

  import timeit

  A = [7,4,8,2,1,22,2,10,9,4,99]

  start1 = timeit.default_timer()

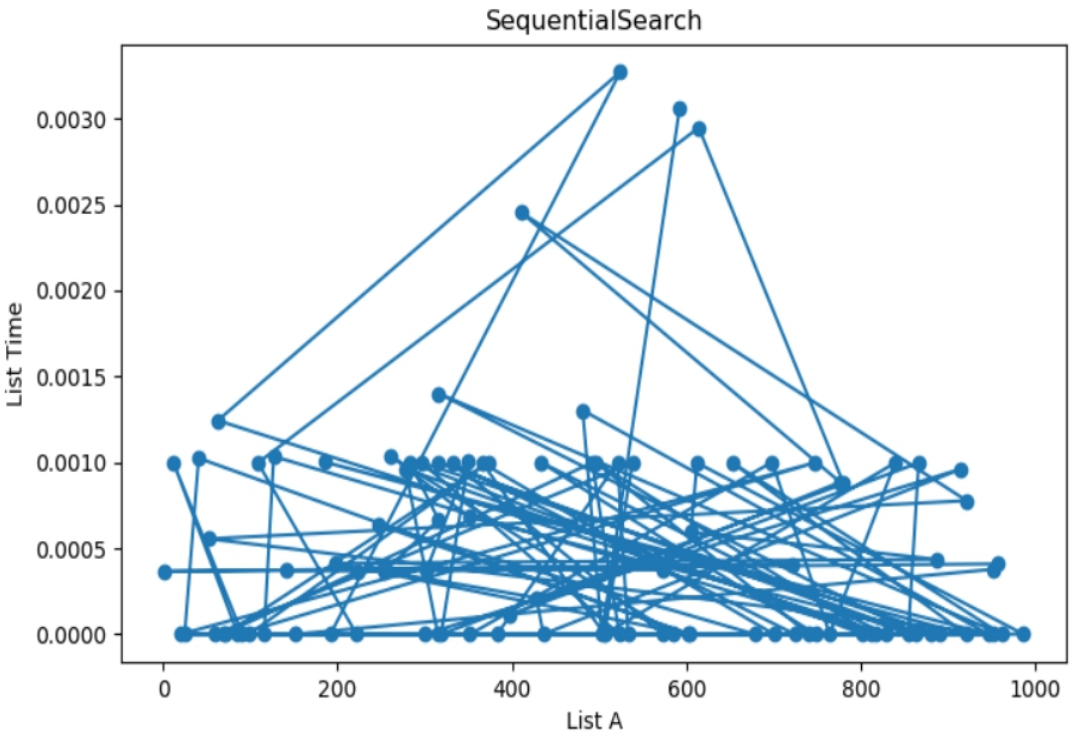
```
SequentialSearch(A,7)
end1 = timeit.default_timer()
res1 = end1-start1
A = [7,4,8,2,1,22,2,10,9,4,99]
start2 = timeit.default_timer()
SequentialSearch(A,99)
end2 = timeit.default_timer()
res2 = end2-start2
print('Thời gian tìm kiếm key = 7   :', res1)
print('Thời gian tìm kiếm key = 99 :', res2)
print('Thời gian chênh lệch          :', abs(res2-res1))
```

Kết quả từ màn hình Command Prompt:

```
Thời gian tìm kiếm key = 7   : 1.0000000000001e-06
Thời gian tìm kiếm key = 99 : 1.300000000002688e-06
Thời gian chênh lệch          : 3.000000000016878e-07
```

- **Tổng quát hóa:**

  Cho một danh sách các số ngẫu nhiên từ 0-1000 gồm 100 phần tử, tiến hành tính thời gian chạy của từng lần tìm kiếm bằng thuật toán SequentialSearch. Xuất ra biểu đồ tổng quát về thời gian chạy của thuật toán tìm kiếm này.



Hình 1.3 Demo thời gian chạy của thuật toán SequentialSearch

Nhận xét:

- *Thời gian chạy khi tìm kiếm khi key ở phía đầu mảng sẽ nhanh hơn key ở phía cuối mảng. Vậy trường hợp xấu nhất là trường hợp duyệt đến phần tử cuối cùng của mảng.*
- *Thời gian chạy của thuật toán trường hợp tốt nhất rơi vào khoảng (0;0.001) và trường hợp xấu nhất có khi vượt qua con số 0.003*

CHƯƠNG II: KỸ THUẬT CHIA ĐỂ TRỊ (DIVIDE-AND-CONQUER)

Là một kỹ thuật thiết kế thuật toán quan trọng dựa trên đệ quy với nhiều phân nhánh. Nó hoạt động bằng cách chia bài toán thành nhiều bài toán nhỏ hơn thuộc cùng thể loại, cứ như vậy lặp lại nhiều lần, cho đến khi bài toán thu được đủ đơn giản để có thể giải quyết trực tiếp. Sau đó, lời giải của các bài toán nhỏ được tổng hợp lại thành lời giải cho bài toán ban đầu. Ở chương này, thuật toán Sắp xếp trộn (Merge Sort), Sắp xếp nhanh (Quick Sort) và Duyệt cây nhị phân (Binary Tree Traversals) sẽ được dùng để làm ví dụ minh họa cho kỹ thuật chia để trị này.

2.1 Ý tưởng thuật toán

2.1.1 Thuật toán Sắp xếp trộn (Merge Sort)

Chia mảng thành những mảng con nhỏ hơn bằng cách chia đôi mảng lớn và tiếp tục chia đôi các mảng con cho đến khi mảng con nhỏ nhất chỉ còn 1 phần tử. Sau đó, vừa so sánh 2 mảng con có cùng mảng cơ sở (khi chia đôi mảng lớn thành 2 mảng con thì mảng lớn đó gọi là mảng cơ sở của 2 mảng con đó) vừa sắp xếp vừa ghép 2 mảng con đó lại thành mảng cơ sở. Tiếp tục lặp lại đến khi còn lại mảng duy nhất, đó là mảng đã được sắp xếp.

Mô tả thuật toán:

Bước 1: Tìm điểm giữa để chia danh sách thành 2 nửa đến khi các danh sách con chỉ còn 1 phần tử

Bước 2: So sánh giá trị trong danh sách con có cùng danh sách cơ sở và sắp xếp theo thứ tự

Bước 3: Hợp nhất sắp xếp cho nửa đầu

Bước 4: Hợp nhất sắp xếp cho nửa đầu

Bước 5: Hợp nhất 2 nửa được sắp xếp ở bước 3,4.

Mã giả:

MergeSort(A[], l, r)

if r > 1

1. *Tìm điểm giữa để chia mảng thành 2 phần:*

Điểm giữa m = (l + r) / 2

2. *Thực hiện giải thuật cho phần thứ nhất:*

MergeSort(A, l, m)

3. *Thực hiện giải thuật cho phần thứ hai:*

MergeSort(A, m + 1, r)

4. *Nối 2 phần đã sắp xếp:*

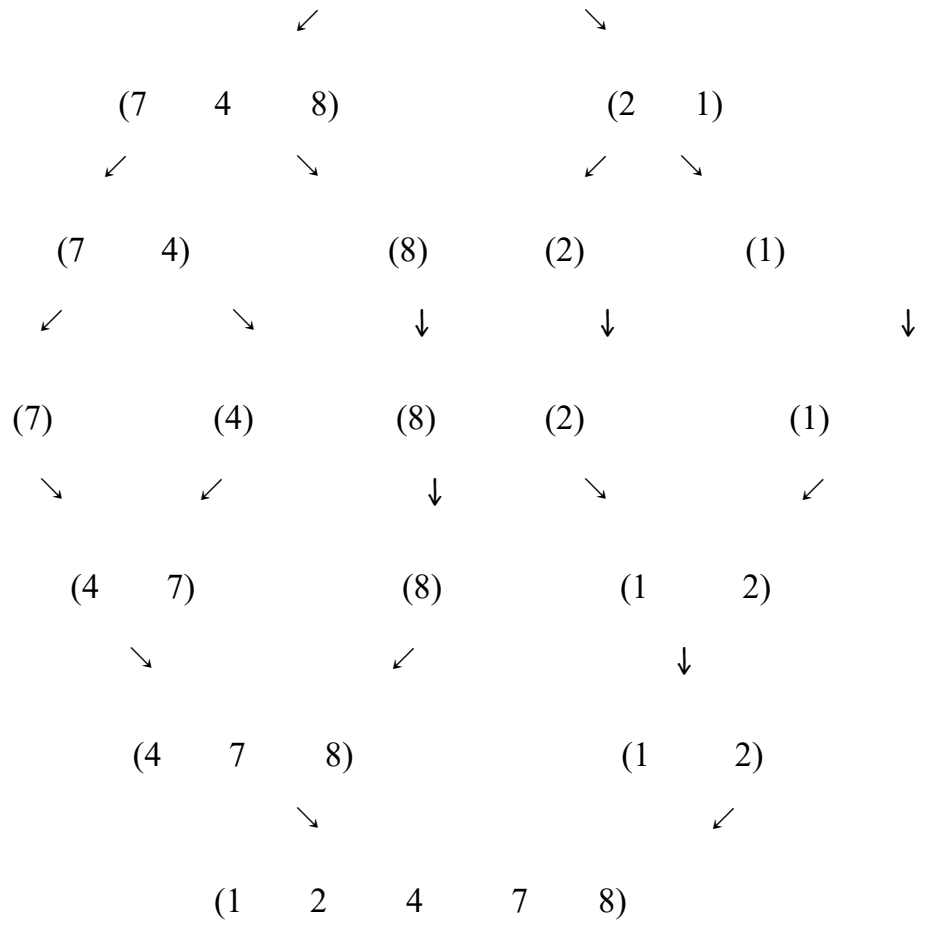
Merge(A, l, m, r)

endif

Chi tiết cách thức hoạt động của thuật toán Merge Sort:

Giả sử ta có mảng A = [7, 4, 8, 2, 1]

(7 4 8 2 1)



2.1.2 Thuật toán Sắp xếp nhanh (Quick Sort)

Chọn một phần tử làm trục và phân vùng mảng đã cho xung quanh trục đã chọn. Phân vùng bằng cách cho một mảng và một phần tử pivot của mảng làm trụ, đặt x vào đúng vị trí của nó trong mảng đã sắp xếp và đặt tất cả các phần tử nhỏ hơn pivot sẽ đứng trước pivot và đặt tất cả các phần tử lớn hơn pivot sẽ đứng sau pivot.

Mô tả thuật toán:

Bước 1 - Chọn giá trị chỉ mục cao nhất là pivot

Bước 2 - Lấy 2 biến trỏ sang trái và phải của danh sách không bao gồm trục

Bước 3 - Bên trái trỏ đến chỉ số thấp

Bước 4 - Bên phải trỏ đến chỉ số cao

Bước 5 - Trong khi giá trị ở bên trái nhỏ hơn pivot, di chuyển sang phải

Bước 6 - Trong khi giá trị ở bên phải lớn hơn pivot, di chuyển sang trái

Bước 7 - Nếu bước 5 và 6 không khớp thì hoán vị trái và phải

Bước 8 - Nếu trái lớn hơn hoặc bằng phải, điểm họ gặp nhau là pivot mới

Mã giả:

```
QuickSort(A[], low, high)
  if (low < high)
    p_index = Partition(A, low, high)
    QuickSort(A, low, p_index - 1)
    QuickSort(A, p_index + 1, high)
  endif
```

Chi tiết cách thức hoạt động của thuật toán Quick Sort:

Giả sử ta có mảng A = [7, 4, 8, 2, 1]

7 4 8 2 1

↙ ↘

4 2 1 **8**

↙ ↘ ↘

2 1 {} {}

↙ ↘

1 {}

=> 1 2 4 7 8

2.1.3 Thuật toán Duyệt cây nhị phân (Binary Tree Traversals)

Có 3 cách thường được sử dụng để duyệt cây nhị phân trong kỹ thuật chia để trị:

- Duyệt nút gốc giữa - Inorder Traversal (LNR)

  Mô tả thuật toán:

  Bước 1. Di chuyển qua cây con bên trái.

  Bước 2. Ghé thăm gốc.

  Bước 3. Đảo qua cây con bên phải.

  Mã giả:

  Tạo class Node cùng phương thức khởi tạo với các thuộc tính left, right, value và nhận vào tham số key. Sau đó viết hàm đệ quy để xuất ra các giá trị của cây nhị phân theo phương pháp duyệt nút gốc giữa.

  ```
  InOrderTraversal(root):
    If root do:
      InOrderTraversal(root.left)
      Print(root.value)
      InOrderTraversal(root.right)
    EndIf
  ```

- Duyệt nút gốc trước - Preorder Traversal (NLR)

  Mô tả thuật toán:

  Bước 1. Ghé thăm gốc.

  Bước 2. Di chuyển qua cây con bên trái

  Bước 3. Đảo qua cây con bên phải

  Mã giả:

Tạo class Node cùng phương thức khởi tạo với các thuộc tính left, right, value và nhận vào tham số key. Sau đó viết hàm đệ quy để xuất ra các giá trị của cây nhị phân theo phương pháp duyệt nút gốc trước.

```
PreOrderTraversal(root):
    If root do:
        Print(root.value)
        PreOrderTraversal(root.left)
        PreOrderTraversal(root.right)
    Endif
```

- Duyệt nút gốc sau - Postorder Traversal (LRN)

Mô tả thuật toán:

Bước 1. Đảo qua cây con bên trái

Bước 2. Duyệt qua cây con bên phải

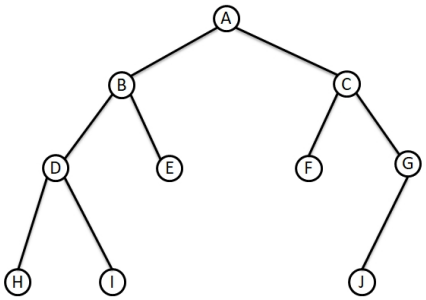
Bước 3. Ghé thăm gốc.

Mã giả:

Tạo class Node cùng phương thức khởi tạo với các thuộc tính left, right, value và nhận vào tham số key. Sau đó viết hàm đệ quy để xuất ra các giá trị của cây nhị phân theo phương pháp duyệt nút gốc sau.

```
PostOrderTraversal(root):
    If root do:
        PostOrderTraversal(root.left)
        PostOrderTraversal(root.right)
        Print(root.value)
    Endif
```

Giả sử ta có cây nhị phân như Hình 2.1:



Hình 2.1 Hình ảnh ví dụ về cây nhị phân

Sử dụng thuật toán Binary Tree Traversals cho Hình 2.1 ta được:

LNR: H,D,I,B,E,A,F,C,J,G

NLR: A,B,D,H,I,E,C,G,F,J

LRN: H,I,D,E,B,F,,J,G,C,A

2.2 Triển khai bằng Python3

2.2.1 Thuật toán Sắp xếp trộn (Merge Sort)

Yêu cầu:

Đầu vào: Mảng A cần sắp xếp

Đầu ra: Mảng A đã sắp xếp

Mục đích: Sắp xếp theo thứ tự tăng dần các phần tử trong mảng bằng cách chia đôi liên tục thành những mảng con rồi so sánh và sắp xếp, sau đó ghép lại thành mảng kết quả cuối cùng đã được sắp xếp chính xác. Thuật toán Merge Sort

mang lại hiệu quả nhanh chóng khi được áp dụng với các mảng nhỏ hoặc lớn đều được.

Hiện thực bằng code:

```
def MergeSort(A):
    if len(A) > 1:
        mid = len(A) // 2

        firstPart = A[:mid]
        secondPart = A[mid:]

        MergeSort(firstPart)
        MergeSort(secondPart)

        i = 0
        j = 0
        k = 0

        while i < len(firstPart) and j < len(secondPart):
            if firstPart[i] < secondPart[j]:
                A[k] = firstPart[i]
                i = i + 1

            else:
                A[k] = secondPart[j]
                j = j + 1
```

```
    k = k + 1

while i < len(firstPart):
    A[k] = firstPart[i]
    i = i + 1
    k = k + 1

while j < len(secondPart):
    A[k] = secondPart[j]
    j = j + 1
    k = k + 1
```

2.2.2 Thuật toán Sắp xếp nhanh (Quick Sort)

Yêu cầu:

Đầu vào: Mảng A cần sắp xếp

Đầu ra: Mảng A đã sắp xếp

Mục đích: Sắp xếp theo thứ tự tăng dần các phần tử trong mảng bằng cách chọn một phần tử làm trục và phân vùng mảng đã cho xung quanh trục đã chọn. Thuật toán Quick Sort mang lại hiệu quả cực kỳ nhanh chóng khi được áp dụng với mảng hoặc chuỗi dữ liệu nhỏ.

Hiện thực bằng code:

```
def Partition(A, low, high):
        pivot = A[high]
        i = low - 1
        for j in range(low, high):
```

```
                if A[j] <= pivot:
                        i = i + 1
                        A[i], A[j] = A[j], A[i]
        A[i + 1], A[high] = A[high], A[i + 1]
    return i + 1

def QuickSort(A, low, high):
        if (len(A) == 1):
                return A
        if low < high:
                part = Partition(A, low, high)
                QuickSort(A, low, part - 1)
                QuickSort(A, part + 1, high)
        return A
```

2.2.3 Thuật toán Giao dịch cây nhị phân (Binary Tree Traversals)

Yêu cầu:

Đầu vào: cây nhị phân

Đầu ra: cây đã được duyệt

Mục đích:

- *Duyệt nút gốc giữa: tìm những nút có bậc không giảm dần.*
- *Duyệt nút gốc trước: tạo ra một bản sao chép của cây, đồng thời cũng cho thấy biểu thức tiền tố của cây.*
- *Duyệt nút gốc sau: được sử dụng để xóa cây, đồng thời cũng cho thấy biểu thức hậu tố của cây.*

Hiện thực bằng code:

```
class Node:
    def __init__(self, name):
            self.left = None
            self.right = None
            self.value = name

def InOrderTraversal(root):
        if root:
                InOrderTraversal(root.left)
                print(root.value)
                InOrderTraversal(root.right)

def PostOrderTraversal(root):
        if root:
                PostOrderTraversal(root.left)
                PostOrderTraversal(root.right)
                print(root.value)

def PreOrderTraversal(root):
        if root:
                print(root.value)
                PreOrderTraversal(root.left)
                PreOrderTraversal(root.right)
```

2.3 Demo

2.3.1 Thuật toán Sắp xếp trộn (Merge Sort)

- **Mô phỏng nhỏ:**

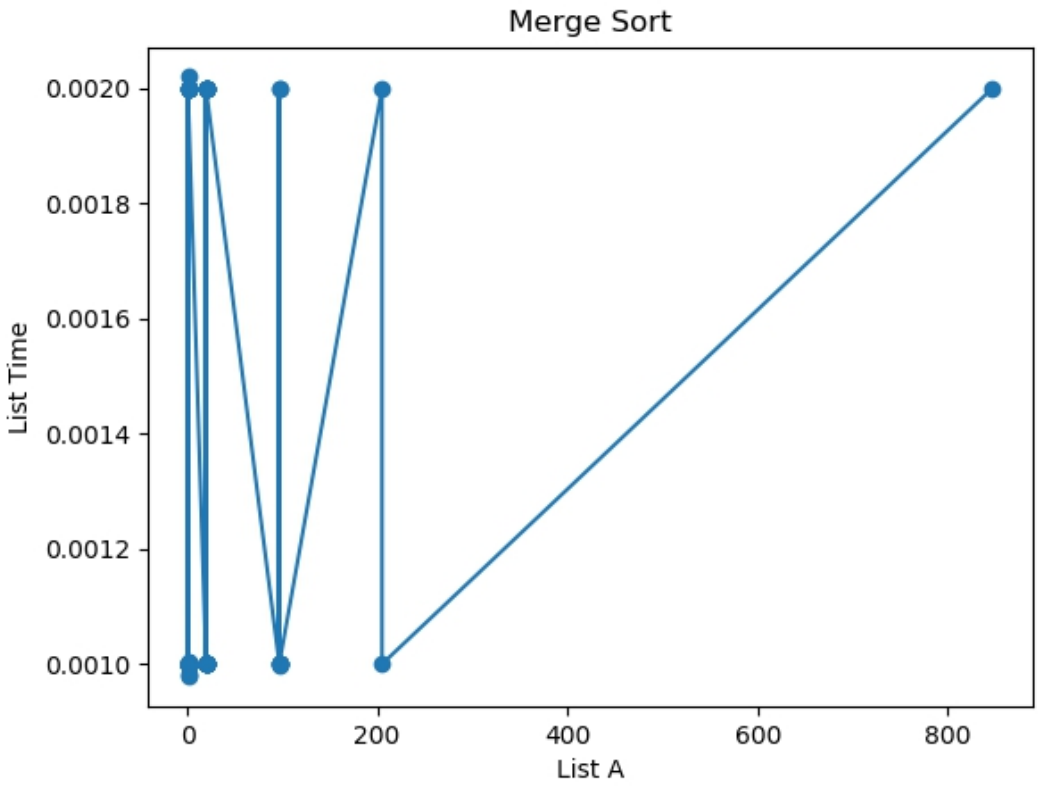
```
import timeit

A1 = [4, 123, 12, 1]
start1 = timeit.default_timer()
MergeSort(A1)
end1 = timeit.default_timer()
res1 = end1 - start1
A2 = [123, 1, 5, 67, 0, 9, 12, 11]
start2 = timeit.default_timer()
MergeSort(A2)
end2 = timeit.default_timer()
res2 = end2 - start2

print('Thời gian sắp xếp mảng A1 :', res1)
print('Thời gian sắp xếp mảng A2 :', res2)
print('Thời gian chênh lệch      :', abs(res2 - res1))
```

Kết quả từ màn hình Command Prompt:

```
Thời gian sắp xếp mảng A1 : 1.5800000255694613e-05
Thời gian sắp xếp mảng A2 : 2.2900000658410136e-05
Thời gian chênh lệch      : 7.100000402715523e-06
```

- **Tổng quát hóa:**

Cho một danh sách các số ngẫu nhiên từ 0-1000 gồm 100 phần tử, tiến hành tính thời gian chạy của từng lần tìm kiếm bằng thuật toán Merge Sort. Xuất ra biểu đồ tổng quát về thời gian chạy của thuật toán sắp xếp này.



Hình 2.2 Demo thời gian chạy của thuật toán Merge Sort

Nhận xét:

- *Thuật toán Merge Sort thực thi có tính ổn định và hiệu quả nhanh chóng dù mảng nhỏ hay lớn.*

2.3.2 Thuật toán Sắp xếp nhanh (Quick Sort)

- **Mô phỏng nhỏ:**

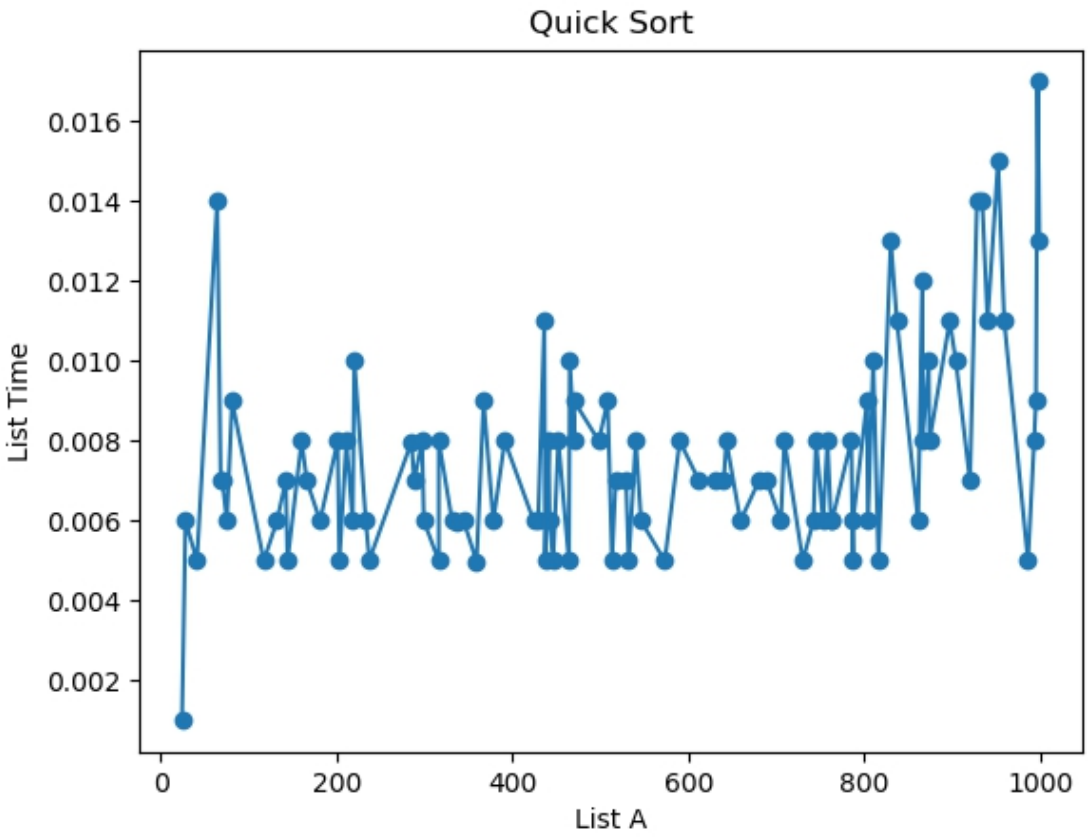
```
import timeit

A1 = [4, 123, 12, 1]
start1 = timeit.default_timer()
QuickSort(A1, 0, len(A1) - 1)
end1 = timeit.default_timer()
res1 = end1 - start1

A2 = [123, 1, 5, 67, 0, 9, 12, 11, 19, 3]
start2 = timeit.default_timer()
QuickSort(A2, 0, len(A2) - 1)
end2 = timeit.default_timer()
res2 = end2 - start2

print('Thời gian sắp xếp mảng A1 :', res1)
print('Thời gian sắp xếp mảng A2 :', res2)
print('Thời gian chênh lệch      :', abs(res2-res1))
```

Kết quả từ màn hình Command Prompt:

```
Thời gian sắp xếp mảng A1 : 4.8400001105619594e-05
Thời gian sắp xếp mảng A2 : 3.730000025825575e-05
Thời gian chênh lệch      : 1.1100000847363845e-05
```

- **Tổng quát hóa:**

Cho một danh sách các số ngẫu nhiên từ 0-1000 gồm 100 phần tử, tiến hành tính thời gian chạy của từng lần tìm kiếm bằng thuật toán Quick Sort. Xuất ra biểu đồ tổng quát về thời gian chạy của thuật toán sắp xếp này.



Hình 2.3 Demo thời gian chạy thuật toán Quick Sort

Nhận xét:

- *Có thể thấy khi chạy thuật toán Quick Sort cho mảng có nhiều phần tử hơn như mảng A2 thì tốc độ thực thi sẽ nhanh hơn.*
- *Thuật toán Quick Sort thích hợp để sử dụng cho các mảng lớn.*

2.3.3 Thuật toán Giao dịch cây nhị phân (Binary Tree Traversals)

- **Mô phỏng nhỏ:**

```
import timeit
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(6)
root.left.right = Node(7)
root.left.right.left = Node(8)
root.right.left = Node(4)
root.right.right = Node(5)

print("Inorder Traversal: ")
start1 = timeit.default_timer()
InOrderTraversal(root)
end1 = timeit.default_timer()
res1 = end1 - start1

print("Preorder Traversal: ")
start2 = timeit.default_timer()
PreOrderTraversal(root)
end2 = timeit.default_timer()
```

```
res2 = end2 - start2

print("Postorder Traversal: ")
start3 = timeit.default_timer()
PostOrderTraversal(root)
end3 = timeit.default_timer()
res3 = end3 - start3

print('Thời gian duyệt nút gốc giữa :', res1)
print('Thời gian duyệt nút gốc trước :', res2)
print('Thời gian duyệt nút gốc sau :', res3)
```

Kết quả từ màn hình Command Prompt:

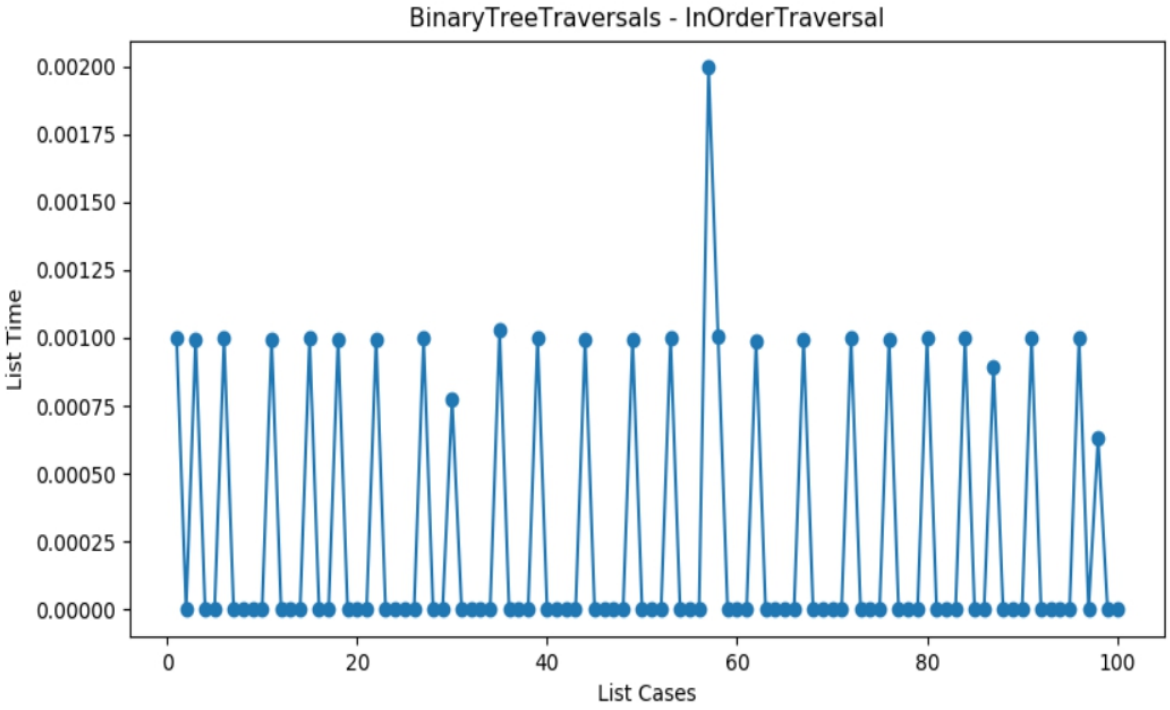
Thời gian duyệt nút gốc giữa : 0.013409499999397667

Thời gian duyệt nút gốc trước : 0.0030211999983293936

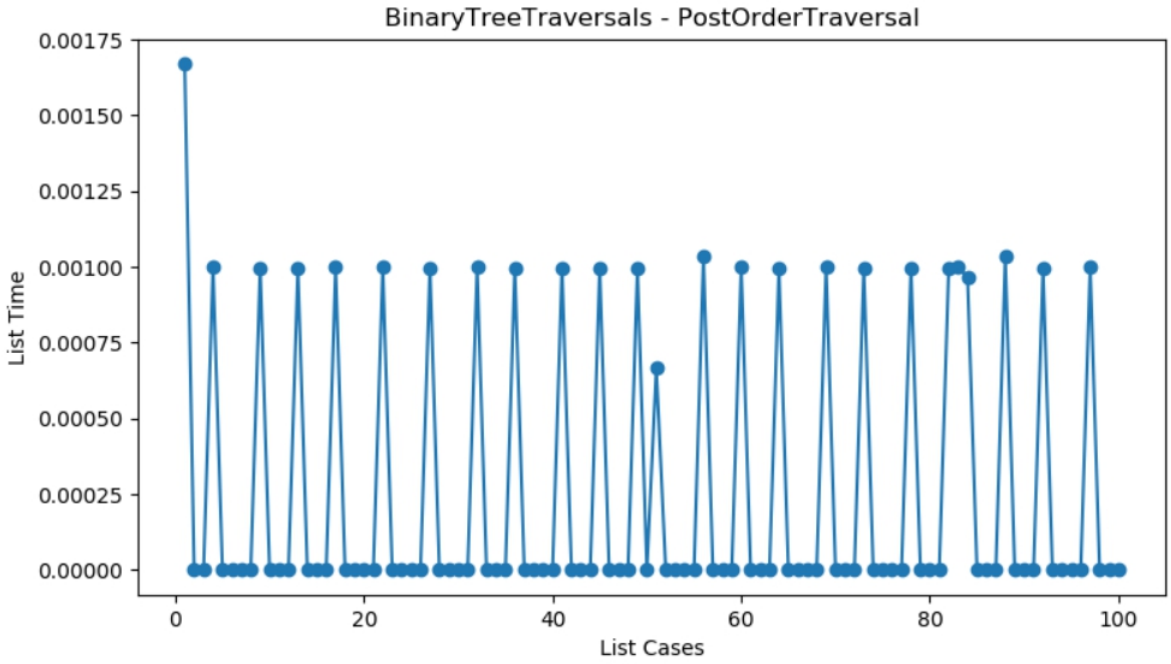
Thời gian duyệt nút gốc sau : 0.002746100002696039

- **Tổng quát hóa:**

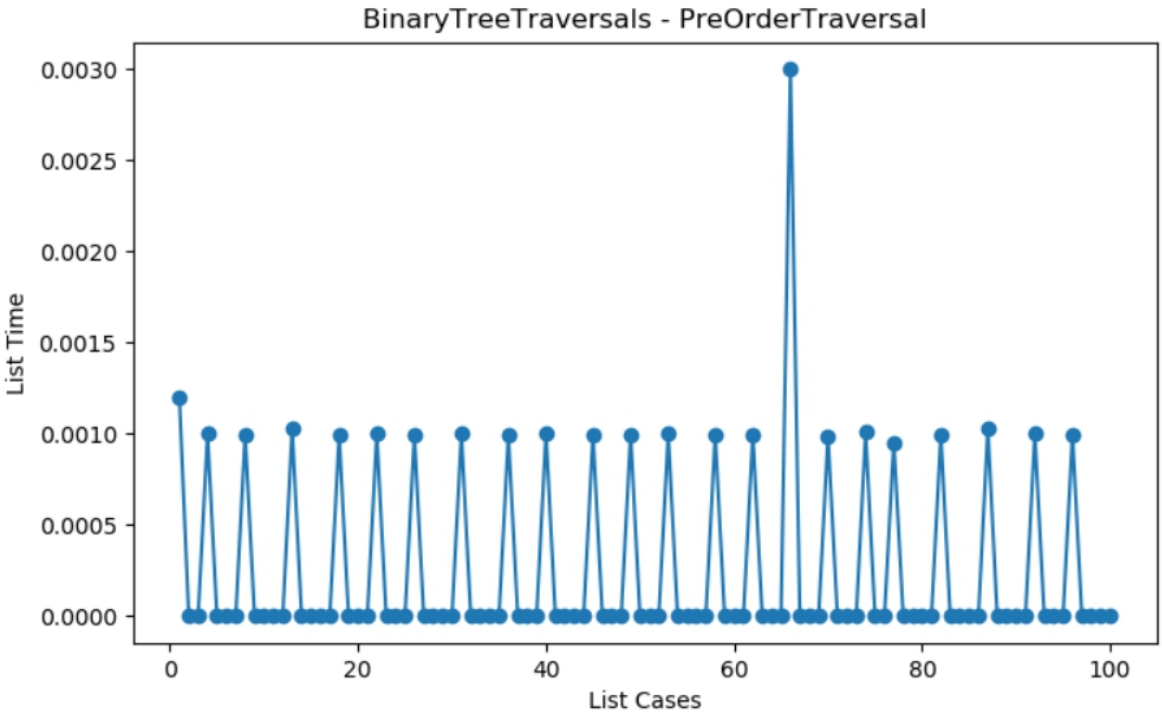
  Cho một danh sách gồm 100 trường hợp thử duyệt cây ngẫu nhiên với chiều cao là 2, tương ứng với lần lượt 3 cách duyệt cây là nút gốc giữa, nút gốc trước và nút gốc sau. Tiến hành tính thời gian chạy của từng trường hợp bằng thuật toán BinaryTreeTraversals. Xuất ra biểu đồ tổng quát về thời gian chạy của thuật toán duyệt cây này.



Hình 2.4 Demo thời gian chạy thuật toán BinaryTreeTravelsals - LNR



Hình 2.5 Demo thời gian chạy thuật toán BinaryTreeTravelsals - NLR



Hình 2.6 Demo thời gian chạy thuật toán BinaryTreeTravelsals - LRN

Nhận xét:

- *Có thể thấy khi chạy các thuật toán duyệt nút gốc của cây đều mang lại kết quả khá nhanh chóng và chính xác.*
- *Thời gian chạy trung bình của thuật toán là 0.001*

CHƯƠNG III: KỸ THUẬT THAM LAM (GREEDY ALGORITHMS)

Là kỹ thuật tìm kiếm lựa chọn tối ưu địa phương ở mỗi bước đi với hy vọng tìm được tối ưu toàn cục. Các thuật toán tham lam khá thành công trong một số vấn đề, chẳng hạn như Mã hóa Huffman (Huffman Coding) được sử dụng để nén dữ liệu, hoặc bài toán người bán hàng (Travelling Salesman Problem),.. Ở chương này, thuật toán Prim, Kruskal và Dijkstra sẽ được sử dụng để làm ví dụ minh họa về việc tìm cây bao trùm nhỏ nhất bằng kỹ thuật tham lam.

3.1 Ý tưởng thuật toán

3.1.1 Thuật toán Prim (Prim's Algorithms)

Bắt đầu từ một đỉnh bất kỳ. Cung cấp cho mỗi đỉnh thông tin về cạnh ngắn nhất nối đỉnh với một đỉnh cây thông qua việc gắn 2 nhãn lần lượt vào một đỉnh gồm tên đỉnh cây gần nhất và độ dài cạnh tương ứng. Đối với các đỉnh không liền kề với bất kỳ đỉnh cây nào sẽ được gắn nhãn ∞ (khoảng cách vô tận của nó đến đỉnh cây) và nhãn rỗng ∅ (đỉnh cây gần nhất).

Mô tả thuật toán:

Bước 1: Bắt đầu với bất kỳ đỉnh nào trong đồ thị.

Bước 2: Trong tất cả các cạnh đối với đỉnh này.Chọn cạnh có trọng số nhỏ nhất.

Bước 3: Lặp lại bước 2 bằng cách sử dụng biến cố các cạnh với đỉnh mới và đỉnh chưa được vẽ. Lặp lại cho đến khi một cây bao trùm được tạo.

Mã giả:

PrimAlgorithm(G):

$V_T = \{x\}$

$E_T = \{\}$

For(i=1 to |V|-1) do:

$e' = (v', u') = min(weight\ all\ (v, u))\ \ (v\ in\ V_T\ ;\ u\ in\ V-V_T)$

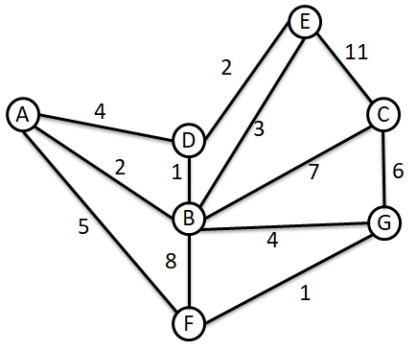
$V_T = V_T \cup \{u'\}$

$E_T = E_T \cup \{e'\}$

EndFor

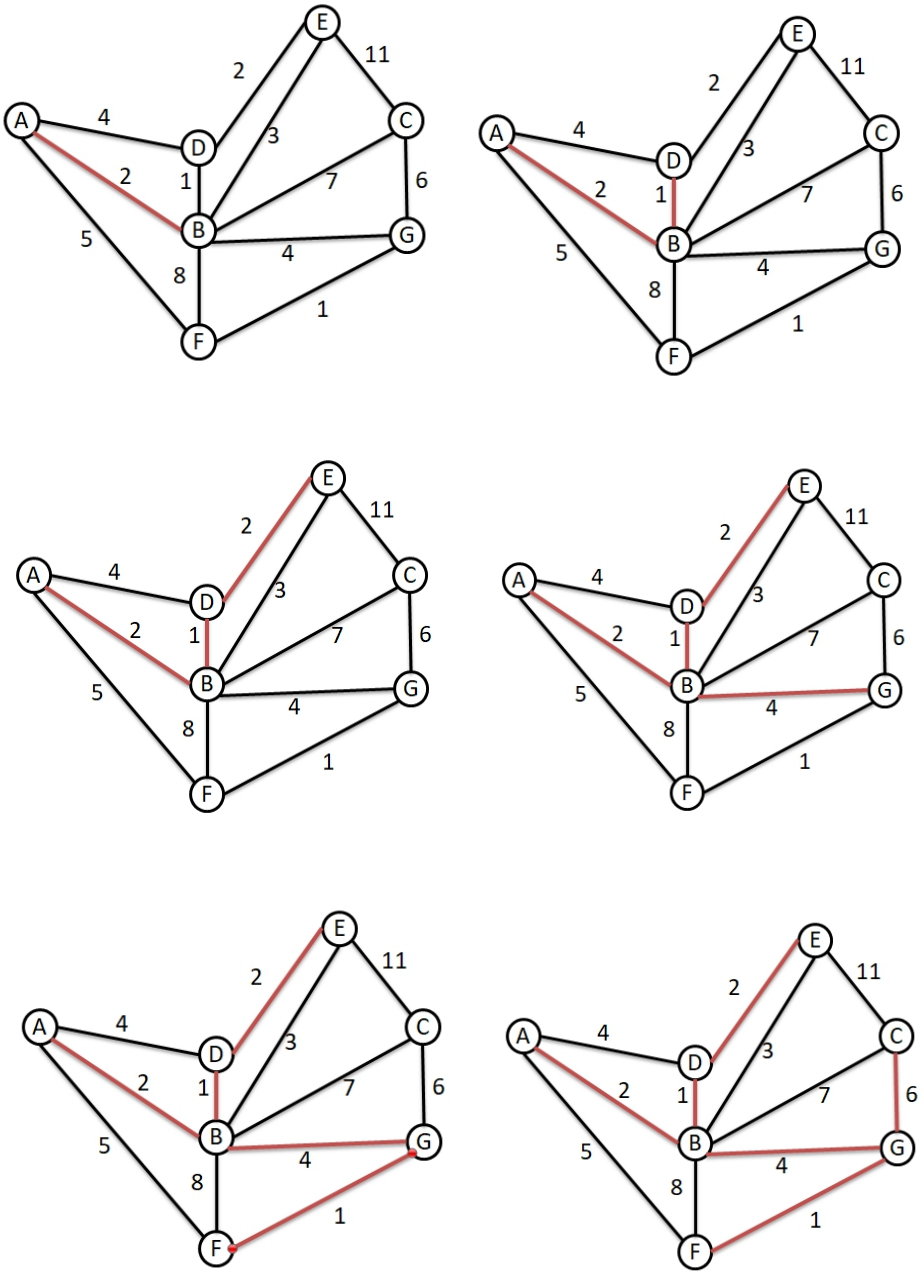
End

Giả sử ta có đồ thị G như Hình 3.1:



Hình 3.1 Hình ảnh ví dụ về một đồ thị vô hướng

Áp dụng thuật toán Prim tìm cây bao trùm nhỏ nhất cho Hình 3.1:



Hình 3.2 Hình ảnh các bước tìm cây bao trùm sử dụng thuật toán Prim

3.1.2 Thuật toán Kruskal (Kruskal's Algorithm)

Thay vì bắt đầu từ một đỉnh, nó bắt đầu bằng cách sắp xếp các cạnh của biểu đồ theo thứ tự không giảm về trọng số của chúng. Sau đó, bắt đầu với đồ thị con trống, nó sẽ quét danh sách đã sắp xếp này, thêm cạnh tiếp theo trên danh sách vào đồ thị con hiện tại nếu việc đưa vào như vậy không tạo ra một chu trình và chỉ cần bỏ qua cạnh đó.

Mô tả thuật toán:

Bước 1: Sắp xếp tất cả các cạnh theo thứ tự không giảm dần về trọng số.

Bước 2: Chọn cạnh nhỏ nhất. Kiểm tra xem nó có tạo thành một chu trình với cây khung đã hình thành cho đến nay hay không.

Bước 3: Nếu chu kỳ không được hình thành thì bao gồm cạnh này.

Ngược lại thì loại bỏ nó.

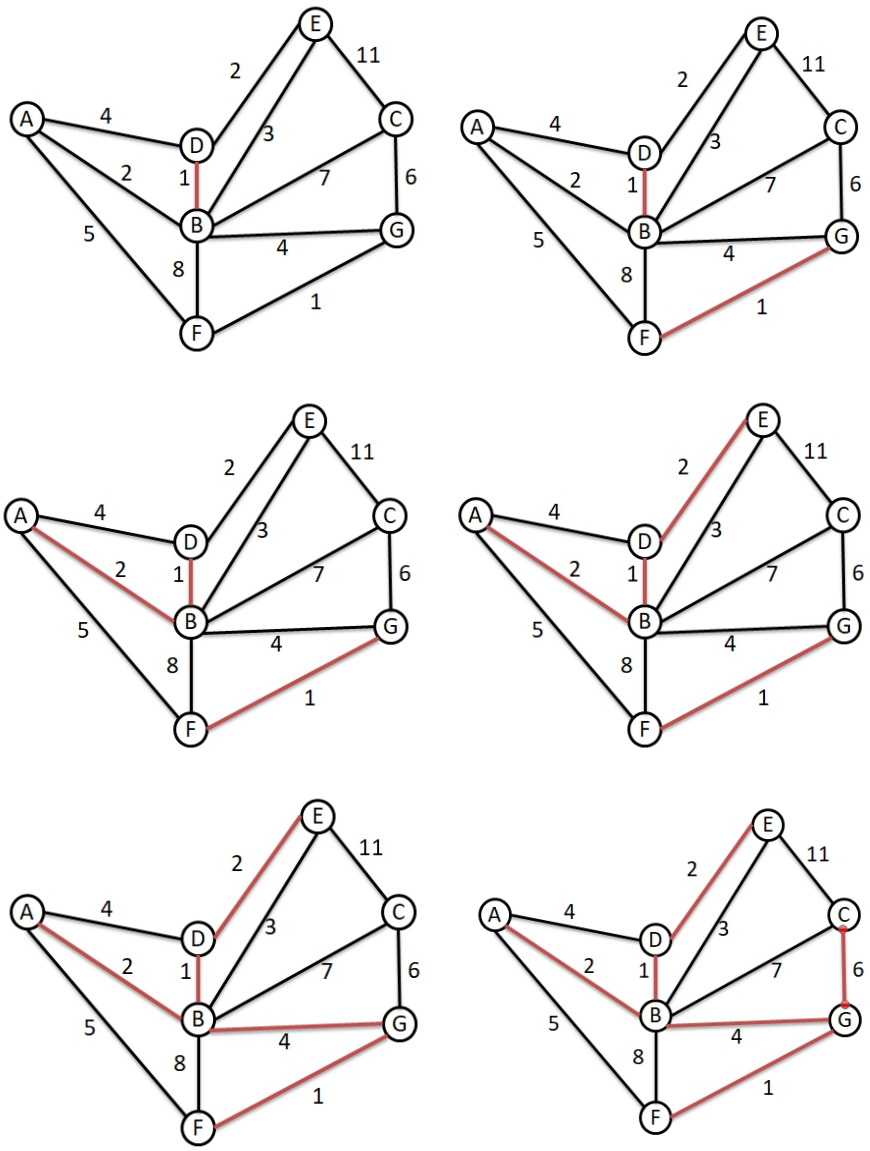
Bước 4: Lặp lại bước 2 cho đến khi có V-1 cạnh trong cây khung.

Mã giả:

```
KruskalAlgorithm(G):
    k = count = 0
    E_T = {}
    While(count < |V|-1) do:
        k+=1
        If(E_T ∪ {e_k} is acyclic) then:
            E_T = E_T ∪ {e_k}
            count+=1
        EndIf
    EndWhile
```

End

Áp dụng thuật toán Kruskal tìm cây bao trùm nhỏ nhất cho Hình 3.1:



Hình 3.3 Hình ảnh các bước tìm cây bao trùm sử dụng thuật toán Kruskal

3.1.3 Thuật toán Dijkstra (Dijkstra's Algorithm)

Mô tả thuật toán:

Bước 1: Tạo một tập hợp cây đường đi ngắn nhất theo dõi các đỉnh có trong cây đường đi ngắn nhất, tức là khoảng cách tối thiểu từ nguồn được tính toán và hoàn thiện. Ban đầu, tập hợp này là trống.

Bước 2: Gán giá trị khoảng cách là 0 cho đỉnh nguồn để nó được chọn đầu tiên và khoảng cách là ∞ cho tất cả các đỉnh còn lại.

Bước 3: Trong khi tập hợp không bao gồm tất cả các đỉnh. Chọn một đỉnh u không có trong tập hợp và có giá trị khoảng cách nhỏ nhất. Bao gồm u vào tập hợp. lặp qua tất cả các đỉnh liền kề để cập nhật giá trị khoảng cách của tất cả các đỉnh liền kề của u.

Bước 4: Đối với mọi đỉnh liền kề v, nếu tổng giá trị khoảng cách của u (từ nguồn) và trọng số của cạnh u-v, nhỏ hơn giá trị khoảng cách của v, thì cập nhật giá trị khoảng cách của v.

Mã giả:

```
DijkstraAlgorithm(G,s):
    Queue = {};
    For(every v in V) do:
        dist_v = INF; p_v = null; Insert(Queue, v, dist_v)
    EndFor
    dist_s=0; Decrease(Queue, s, dist_s); V_T = {}
    For(i = 0 to |V|-1) do:
        u' = Queue.remove(Min(Queue)) ; V_T = V_T ∪ {u'}
```

For(every u in V-V_T adjacencyMatrixacent to u') do:

If(($dist_{u'}$+ w(u',u)) < $dist_u$) then:

$dist_u$ = $dist_{u'}$+ w(u',u) ; p_u = u';

Decrease(Queue, u, $dist_u$)

EndIf

EndFor

EndFor

End

3.2 Triển khai bằng Python3

3.2.1 Thuật toán Prim (Prim's Algorithms)

Yêu cầu:

Đầu vào: một tập đồ thị gồm các đỉnh và trọng số tương ứng

Đầu ra: cây bao trùm nhỏ nhất

Mục đích: Tìm cây bao trùm nhỏ nhất cho đồ thị vô hướng có trọng số.

Hiện thực bằng code:

```
class Graph():
    def __init__(self, nodes):
        self.N = nodes
        self.inputGraph = [[0 for c in range(nodes)] for r in range(nodes)]
    def minIndex(self, keyPick, setNodes):
        minValue = 99999
```

```
        for n in range(self.N):
            if keyPick[n] < minValue and setNodes[n] == False:
                minValue = keyPick[n]
                index = n
        return index
    def displayResult(self, resultTree):
        for i in range(1, self.N):
            print(resultTree[i], i, "\t", self.inputGraph[i][resultTree[i]])
    def PrimAlgorithm(self):
        keyPick = [99999] * self.N
        keyPick[0] = 0
        resultTree = [None] * self.N
        resultTree[0] = -1
        setNodes = [False] * self.N
        for i in range(self.N):
            minDistanceNode = self.minIndex(keyPick, setNodes)
            setNodes[minDistanceNode] = True
            for n in range(self.N):
                if self.inputGraph[minDistanceNode][n] > 0 and setNodes[n] ==
False and keyPick[n] > self.inputGraph[minDistanceNode][n]:
                    keyPick[n] = self.inputGraph[minDistanceNode][n]
                    resultTree[n] = minDistanceNode
        self.displayResult(resultTree)
```

3.2.2 Thuật toán Kruskal (Kruskal's Algorithm)

Yêu cầu:

Đầu vào: một tập đồ thị gồm các đỉnh và trọng số tương ứng

Đầu ra: cây bao trùm nhỏ nhất

Mục đích: Tìm cây bao trùm nhỏ nhất, giải thuật Kruskal xem mỗi đỉnh như là một cây độc lập và kết nối từng đỉnh này với đỉnh khác nếu nó có giá trị so sánh thấp nhất trong số các lựa chọn còn lại.

Hiện thực bằng code:

```
from collections import defaultdict

class Graph():
    def __init__(self, nodes):
        self.N = nodes
        self.inputGraph = []
    def Edge(self, v1, v2, weight):
        self.inputGraph.append([v1, v2, weight])
     def findSetOfElement(self, parent, element):
        if parent[element] == element:
            return element
        return self.findSetOfElement(parent, parent[element])
    def Union(self, parent, degree, x, y):
        xRoot = self.findSetOfElement(parent, x)
        yRoot = self.findSetOfElement(parent, y)
        if degree[xRoot] < degree[yRoot]:
            parent[xRoot] = yRoot
        elif degree[xRoot] > degree[yRoot]:
            parent[yRoot] = xRoot
         else:
            parent[yRoot] = xRoot
```

```
            degree[xRoot] += 1

    def KruskalAlgorithm(self):
        result = []
        parent = []
        degree = []
        element = 0
        index = 0
        self.inputGraph = sorted(self.inputGraph, key = lambda item: item[2])
        for node in range(self.N):
            parent.append(node)
            degree.append(0)
        while index < self.N - 1:
            v1, v2, weight = self.inputGraph[element]
            element = element + 1
            x = self.findSetOfElement(parent, v1)
            y = self.findSetOfElement(parent, v2)
            if x != y:
                index = index + 1
                result.append([v1, v2, weight])
                self.Union(parent, degree, x, y)
        lowestCost = 0
        print("Cạnh \tTrọng số")
        for v1, v2, weight in result:
            lowestCost += weight
            print("%d %d \t %d" % (v1, v2, weight))
        print("--> Giá trị của cây bao trùm nhỏ nhất =", lowestCost)
```

3.2.3 Thuật toán Dijkstra (Dijkstra's Algorithm)

Yêu cầu:

Đầu vào: một tập đồ thị gồm các đỉnh và trọng số tương ứng

Đầu ra: giá trị đường đi ngắn nhất từ đỉnh gốc đến các đỉnh còn lại

Mục đích: Tìm đường đi ngắn nhất từ đỉnh gốc đã chọn đến các đỉnh còn lại.

Hiện thực bằng code:

```
class Graph():
    def __init__(self, nodes):
        self.N = nodes
        self.inputGraph = [[0 for c in range(nodes)] for r in range(nodes)]
    def minDistanceValue(self, result, shortestPathSet):
        minDistance = 99999
        for n in range(self.N):
            if result[n] < minDistance and shortestPathSet[n] == False:
                minDistance = result[n]
                minIndex = n
        return minIndex
    def displayResult(self, result):
        for node in range(self.N):
            print (node, "\t", result[node])
    def DijkstraAlgorithm(self, source):
        result = [99999] * self.N
        result[source] = 0
        shortestPathSet = [False] * self.N
```

```
        for i in range(self.N):
            pickMinDistance = self.minDistanceValue(result, shortestPathSet)
            shortestPathSet[pickMinDistance] = True
            for n in range(self.N):
                if self.inputGraph[pickMinDistance][n] > 0 and shortestPathSet[n]
    == False and result[n] > result[pickMinDistance] +
    self.inputGraph[pickMinDistance][n]:
                    result[n] = result[pickMinDistance] +
    self.inputGraph[pickMinDistance][n]
  self.displayResult(result)
```

3.3 Demo

3.3.1 Thuật toán Prim (Prim's Algorithms)

- **Mô phỏng nhỏ:**

```
import timeit
G = Graph(6)
G.inputGraph = [
   [0, 1, 2, 8, 5, 3],
   [1, 1, 1, 1, 2, 8],
   [0, 3, 2, 2, 5, 7],
   [6, 8, 1, 1, 1, 8],
   [2, 5, 2, 8, 8, 4],
   [2, 0, 7, 7, 8, 1]
]
print("Cạnh \tTrọng số")
start = timeit.default_timer()
```

G.PrimAlgorithm()

end = timeit.default_timer()

res = end - start

print("Thời gian chạy thuật toán Prim :", res)

Kết quả từ màn hình Command Prompt:

Cạnh Trọng số

0 1 1

1 2 3

1 3 8

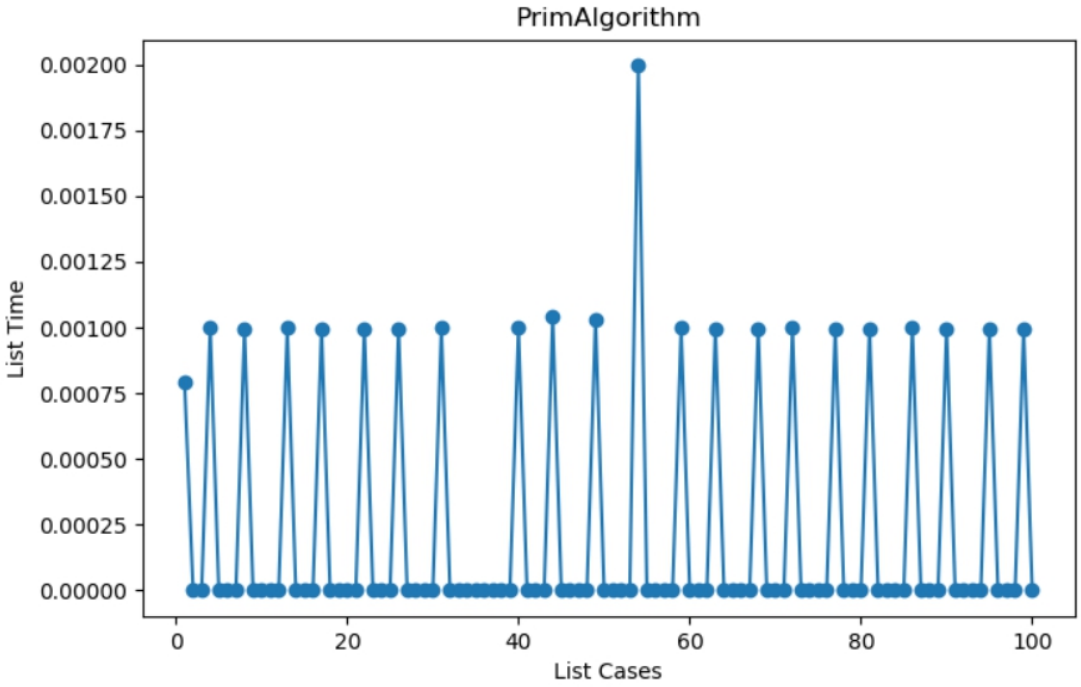
3 4 8

0 5 2

Thời gian chạy thuật toán Prim : 0.07171479999669828

- **Tổng quát hóa:**

  Cho danh sách gồm 100 trường hợp thử bằng ma trận đầu vào ngẫu nhiên tương ứng. Tiến hành tính thời gian chạy của từng trường hợp bằng thuật toán Prim. Xuất ra biểu đồ tổng quát về thời gian chạy của thuật toán tìm cây bao trùm nhỏ nhất này.



Hình 3.1 Demo thời gian chạy thuật toán Prim

Nhận xét:

- *Thuật toán Prim trả về kết quả cây bao trùm nhỏ nhất khá nhanh chóng và chính xác.*
- *Thời gian chạy của thuật toán vào khoảng (0-0.001). Trong một số trường hợp xấu, Thời gian chạy vượt lên con số 0.002*

3.3.2 Thuật toán Kruskal (Kruskal's Algorithm)

Mô phỏng nhỏ:

import timeit

G = Graph(6)

G.Edge(0, 1, 2)

G.Edge(2, 3, 1)

G.Edge(3, 5, 2)

G.Edge(1, 5, 1)

G.Edge(4, 0, 2)

start = timeit.default_timer()

G.KruskalAlgorithm()

end = timeit.default_timer()

res = end - start

print("--> Thời gian chạy thuật toán Kruskal :", res)

Kết quả từ màn hình Command Prompt:

Cạnh Trọng số

2 3 1

1 5 1

0 1 2

3 5 2

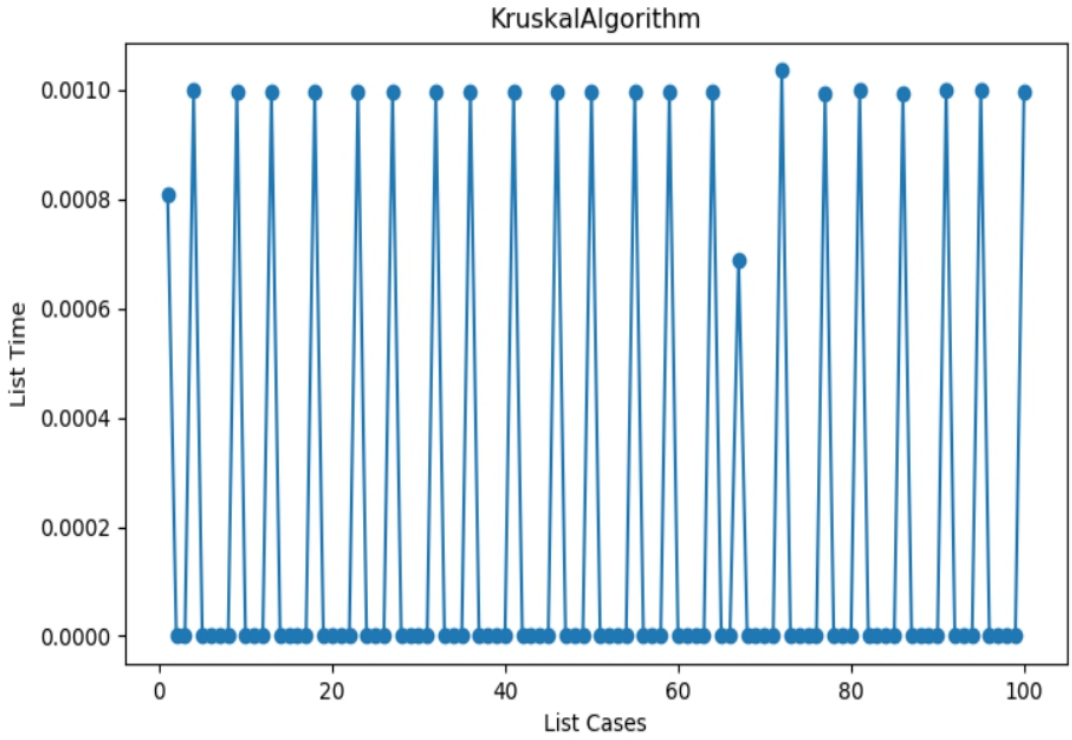
4 0 2

--> Giá trị của cây bao trùm nhỏ nhất = 8

--> Thời gian chạy thuật toán Kruskal : 0.004902100001345389

- **Tổng quát hóa:**

  Cho danh sách gồm 100 trường hợp thử bằng ma trận đầu vào định sẵn. Tiến hành tính thời gian chạy của từng trường hợp bằng thuật toán Prim. Xuất ra biểu đồ tổng quát về thời gian chạy của thuật toán tìm cây bao trùm nhỏ nhất này.



Hình 3.2 Demo thời gian chạy thuật toán Kruskal

Nhận xét:

- *Thuật toán Kruskal trả về kết quả cây bao trùm nhỏ nhất khá nhanh chóng và chính xác cùng với tổng giá trị của cây.*
- *Thời gian chạy khá tương đồng với thuật toán Prim (0-0.001)*

3.3.3 Thuật toán Dijkstra (Dijkstra's Algorithm)

- **Mô phỏng nhỏ:**

import timeit

G = Graph(6)

G.inputGraph = [

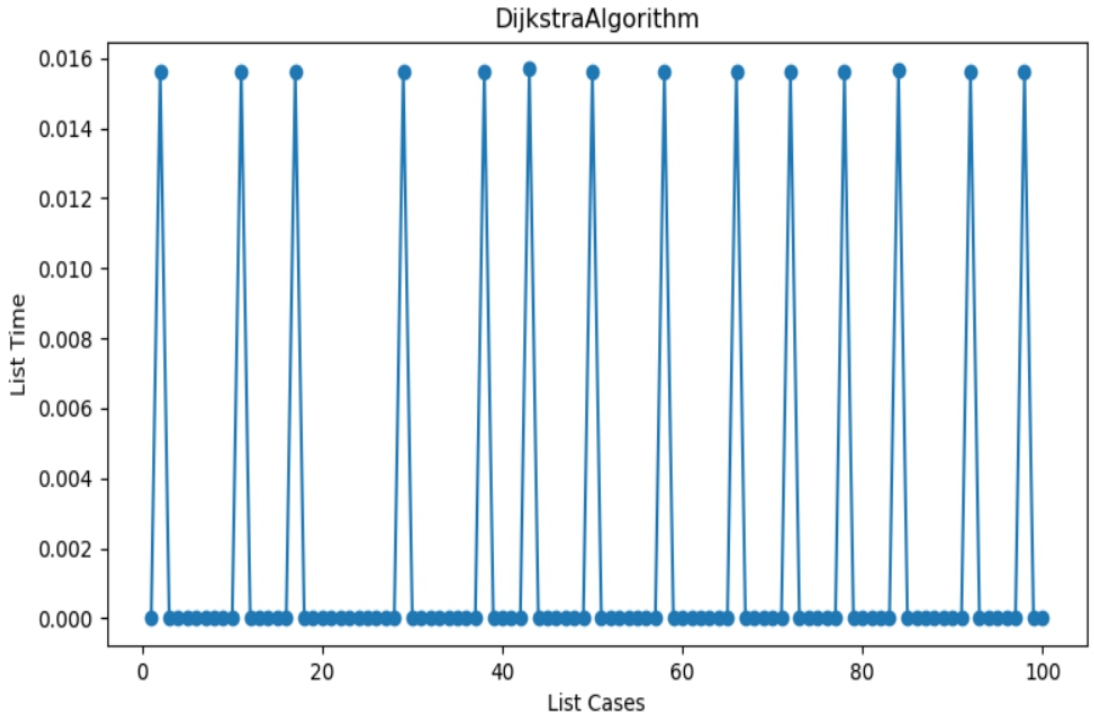
```
    [0, 1, 2, 8, 5, 3],
    [1, 1, 1, 1, 2, 8],
    [0, 3, 2, 2, 5, 7],
    [6, 8, 1, 1, 1, 8],
    [2, 5, 2, 8, 8, 4],
    [2, 0, 7, 7, 8, 1]
];
print("Đỉnh \t Khoảng cách tới đỉnh gốc")
start = timeit.default_timer()
G.DijkstraAlgorithm(0);
end = timeit.default_timer()
res = end - start
print("Thời gian chạy thuật toán Dijkstra :", res)
```

Kết quả từ màn hình Command Prompt:

```
Đỉnh    Khoảng cách tới đỉnh gốc
0       0
1       1
2       2
3       2
4       3
5       3
Thời gian chạy thuật toán Dijkstra : 0.022671300001093186
```

- **Tổng quát hóa:**

  Cho danh sách gồm 100 trường hợp thử bằng ma trận đầu vào ngẫu nhiên tương ứng. Tiến hành tính thời gian chạy của từng trường hợp bằng thuật toán Dijkstra. Xuất ra biểu đồ tổng quát về thời gian chạy của thuật toán tìm đường đi ngắn nhất này.



Hình 3.3 Demo thời gian chạy thuật toán Dijkstra

Nhận xét:

- *Thuật toán Dijkstra trả về kết quả các đường đi ngắn nhất của các đỉnh trong đồ thị đến đỉnh gốc đã chọn rất nhanh chóng và chính xác.*
- *Thời gian chạy trong trường hợp xấu có thể lên đến 0.016*

CHƯƠNG IV:KỸ THUẬT QUY HOẠCH ĐỘNG (DYNAMIC PROGRAMMING)

Là kỹ thuật giúp làm giảm thời gian chạy của các thuật toán thông qua việc chia nhỏ nó thành một tập hợp các bài toán con đơn giản hơn, giải từng bài toán con đó chỉ một lần và lưu trữ kết quả của chúng bằng cách sử dụng cấu trúc dữ liệu dựa trên bộ nhớ. Mỗi giải pháp bài toán con được lập chỉ mục theo một cách nào đó, thường dựa trên các giá trị của các tham số đầu vào của nó để thuận tiện cho việc tra cứu để lần tới khi vấn đề con tương tự xảy ra, thay vì tính toán lại giải pháp của nó thì chỉ cần tra cứu giải pháp đã tính toán trước đó. Ở chương này, kỹ thuật quy hoạch động sẽ được minh họa rõ nét thông qua bài toán về Dãy Fibonacci, Sắp xếp chiếc túi (Knapsack Problem) và Đổi xu (Change-making Problem).

4.1 Ý tưởng thuật toán

4.1.1 Bài toán Dãy Fibonacci (Fibonacci Problem)

Fibonacci là một chuỗi các chữ số, bắt đầu là 0 và 1, các chữ số phía sau là tổng của hai chữ số liền trước nó.

Mô tả thuật toán:

Bước 1: Tạo mảng Fibonacci bắt đầu với hai giá trị 0 và 1.

Bước 2: Tính và thêm các giá trị tiếp theo của dãy số vào mảng.

Mã giả:

```
Fibonacci(n):
    fibArray = [0, 1]
    For i = 2 to n do
        fibArray[i] = fibArray[i - 1] + fibArray[i - 2]
    EndFor
    return fibArray
```

4.1.2 Bài toán Đổi xu (Change-making Problem)

Bài toán Đổi xu dùng để tìm ra số lượng xu tối thiểu với những mệnh giá nhất định để tạo thành một lượng tiền lớn.

Mô tả thuật toán:

Tính số xu quy đổi của giá trị từ 0 đến tổng giá trị S theo mệnh giá từ nhỏ đến lớn. Chỉ tính lại số xu của mệnh giá lớn hơn đối với giá trị lớn hơn hoặc bằng mệnh giá xu.

Mã giả:

```
ChangeMaking(coinValueList, changeAmount, minCoins, coinsUsed)
    For cents = 0 to changeAmount do
        coinCount = cents
        newCoin = 1
        For c in coinValueList do
            If c <= cents do
                For j in c do
                    If minCoins[cents-j] + 1 < coinCount
                        coinCount = minCoins[cents-j] + 1
                        newCoin = j
                    EndIf
                EndFor
            EndIf
            minCoins[cents] = coinCount
            coinsUsed[cents] = newCoin
        EndFor
    EndFor
```

4.1.3 Bài toán Sắp xếp chiếc túi (Knapsack Problem)

Knapsack là bài toán sắp xếp đồ vật có giá trị V[v_1, v_2, …, v_n] và trọng lượng W[w_1, w_2, …, w_n] vào một túi có sức chứa C sao cho tổng giá trị có được là lớn nhất.

Mô tả thuật toán:

Tính toán tất cả các giá trị lớn nhất có thể đạt của sức chứa từ 0 đến C.

F[i, c] chứa giá trị lớn nhất có thể đạt được sức chứa c, điền tổng giá trị tối đa của các vật với trọng lượng w_i vào F[i, c].

Mã giả:

```
Knapsack(C, W, V, n)
    For i = 0 to n do
        For c = 0 to C do
            F[i, c] = 0
        EndFor
    EndFor
    For i = 0 to n:
        For c = 0 to C:
            If W[i-1] <= c:
                F[i, c] = max(V[i-1] + F[i-1, c-W[i-1]], F[i-1, c])
            Else:
                F[i, c] = F[i-1, c]
            EndIf
        EndFor
    EndFor
    return F[n, C]
```

4.2 Triển khai bằng Python3

4.2.1 Bài toán Dãy Fibonacci (Fibonacci Problem)

Yêu cầu:

Đầu vào: Vị trí cuối cùng của dãy số Fibonacci muốn tính

Đầu ra: Dãy Fibonacci có vị trí từ 0 đến n

Mục đích: Tính toán tất cả các số Fibonacci từ 0 đến n

Hiện thực bằng code:

```
def fibonacci(n):
    FibArray = [0, 1]
    while len(FibArray) < n + 1:
        FibArray.append(0)
    for i in range(2, n+1):
        FibArray[i] = FibArray[i - 2] + FibArray[i - 1]
    return FibArray
```

4.2.2 Bài toán Đổi xu (Change-making Problem)

Yêu cầu:

Đầu vào: Danh sách giá trị của xu để quy đổi và Tổng giá trị cần đổi

Đầu ra: Danh sách quy đổi của xu lớn nhất

Mục đích: Tìm ra số lượng xu tối thiểu để quy đổi ra tổng giá trị cần đổi

Hiện thực bằng code:

```
    def ChangeMaking(d, S, coins, used):
        for s in range(S + 1):
            count = s
```

```
            newCoin = 1
            for j in [c for c in d if c <= s]:
                if coins[s-j] + 1 < count:
                    count = coins[s-j] + 1
                    newCoin = j
            coins[s] = count
            used[s] = newCoin
        return coins
```

4.2.3 Bài toán Sắp xếp chiếc túi (Knapsack Problem)

Yêu cầu:

Đầu vào:

Sức chứa tối đa C

Danh sách W trọng lượng các vật

Danh sách V giá trị các vật

Số lượng các vật

Đầu ra: Tổng giá trị tối ưu nhất có thể chứa được

Mục đích: Tìm ra tổng giá trị tối ưu nhất có thể chứa với sức chứa C

Hiện thực bằng code:

```
def Knapsack(C, W, V, n):
    F = [[0 for x in range(C + 1)] for x in range(n + 1)]
    for i in range(n + 1):
        for c in range(C + 1):
            if W[i-1] <= c:
                F[i][c] = max(V[i-1] + F[i-1][c-W[i-1]], F[i-1][c])
```

```
                else:
                        F[i][c] = F[i-1][c]
    return F[n][C]
```

4.3 Demo

4.3.1 Bài toán Dãy Fibonacci (Fibonacci Problem)

- **Mô phỏng nhỏ:**

  ```
  import timeit
  n = 10
  print("Fibonacci của", n, "là:")
  start = timeit.default_timer()
  print(fibonacci(n))
  end = timeit.default_timer()
  res = end - start
  print("Thời gian chạy thuật toán Fibonacci:", res)
  ```

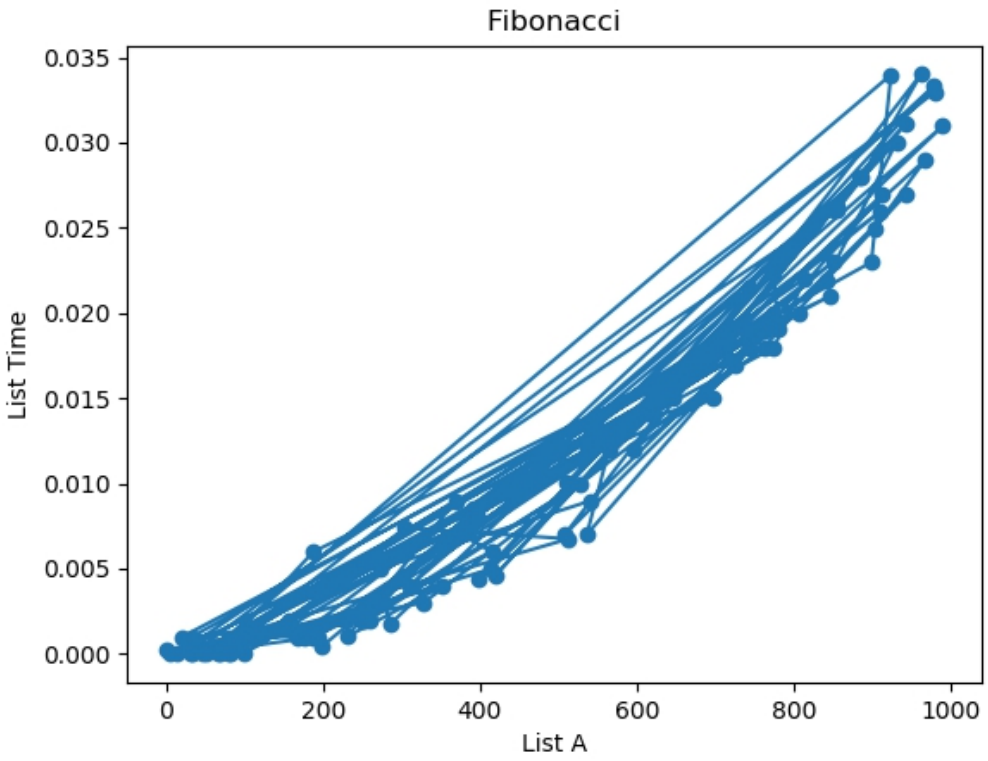
  Kết quả từ màn hình Command Prompt:

  Fibonacci của 10 là: 55

  Thời gian chạy thuật toán Fibonacci: 0.00047060000000000157

- **Tổng quát hóa:**

  Cho danh sách gồm 100 kết quả thử nghiệm thuật toán. Tính thời gian chạy của từng trường hợp. Xuất ra biểu đồ tổng quát về thời gian chạy của thuật toán giải quyết bài toán Fibonacci này.



Hình 4.1 Demo thời gian chạy của bài toán Fibonacci

Nhận xét:

- *Thời gian chạy của thuật toán càng tăng khi giá trị kết quả càng tăng*
- *Có thể xem đây là sự đồng biến (tăng)*

4.3.2 Bài toán Đổi xu (Change-making Problem)

- **Mô phỏng nhỏ:**

```
import timeit
d = [1, 2, 5, 10]
S = 57
coins = [0]*(S + 1)
used = [0]*(S + 1)
print("Bảng quy đổi coins của giá trị", S)
```

start = timeit.default_timer()

print(ChangeMaking(d, S, coins, used))

end = timeit.default_timer()

res = end - start

print("Thời gian chạy thuật toán Change-Making:", res)

Kết quả từ màn hình Command Prompt:

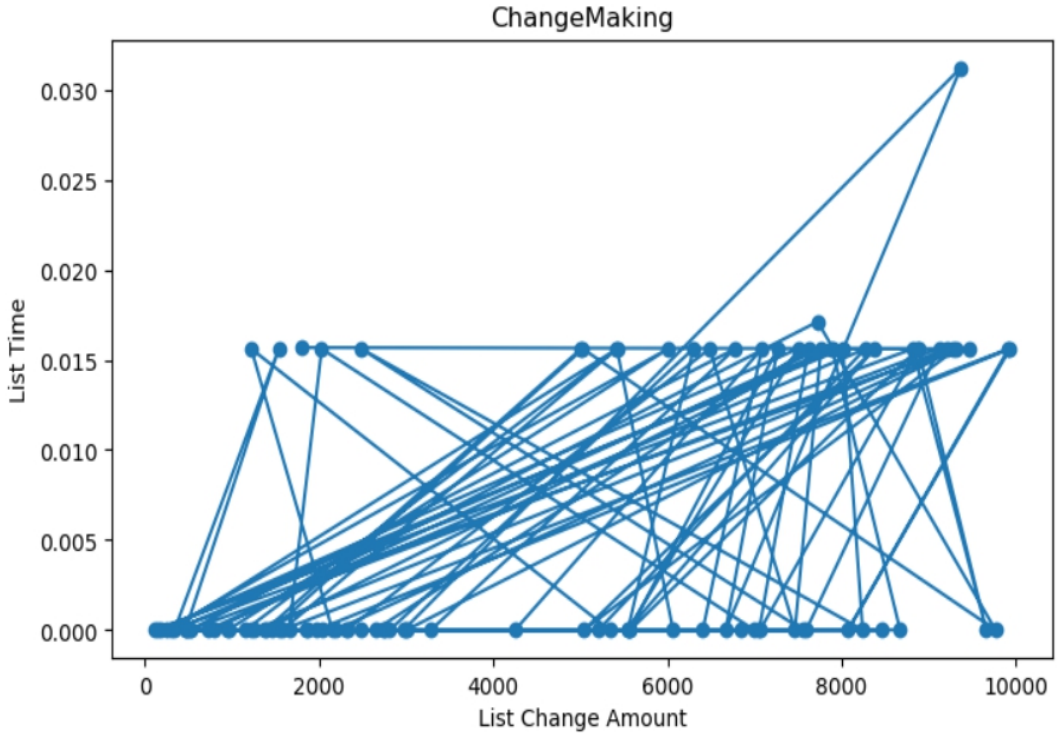
Bảng quy đổi coins của giá trị 57

[0, 1, 1, 2, 2, 1, 2, 2, 3, 3, 1, 2, 2, 3, 3, 2, 3, 3, 4, 4, 2, 3, 3, 4, 4, 3, 4, 4, 5, 5, 3, 4, 4, 5, 5, 4, 5, 5, 6, 6, 4, 5, 5, 6, 6, 5, 6, 6, 7, 7, 5, 6, 6, 7, 7, 6, 7, 7]

Thời gian chạy thuật toán Change-Making: 0.0013642999999999988

- **Tổng quát hóa:**

  Cho danh sách gồm 100 trường hợp thử với giá trị xu mong muốn quy đổi được để ngẫu nhiên trong khoảng từ 100 đến 10000. Tính thời gian chạy của từng trường hợp. Xuất ra biểu đồ tổng quát về thời gian chạy của thuật toán giải quyết bài toán đổi xu này.



Hình 4.2 Demo thời gian chạy của bài toán ChangeMaking

Nhận xét:

- *Thời gian chạy của thuật toán rơi vào khoảng (0-0.016)*
- *Trong trường hợp xấu xảy ra có thể vượt qua mức 0.032*

4.3.3 Bài toán Sắp xếp chiếc túi (Knapsack Problem)

- **Mô phỏng nhỏ:**

import timeit

C = 50

W = [10, 30, 40]

V = [20, 50, 110]

n = len(V)

print("Giá trị có được khi xếp vào túi có sức chứa", C)

start = timeit.default_timer()

print(Knapsack(C, W, V, n))

end = timeit.default_timer()

res = end - start

print("Thời gian chạy thuật toán Change-Making:", res)

Kết quả từ màn hình Command Prompt:

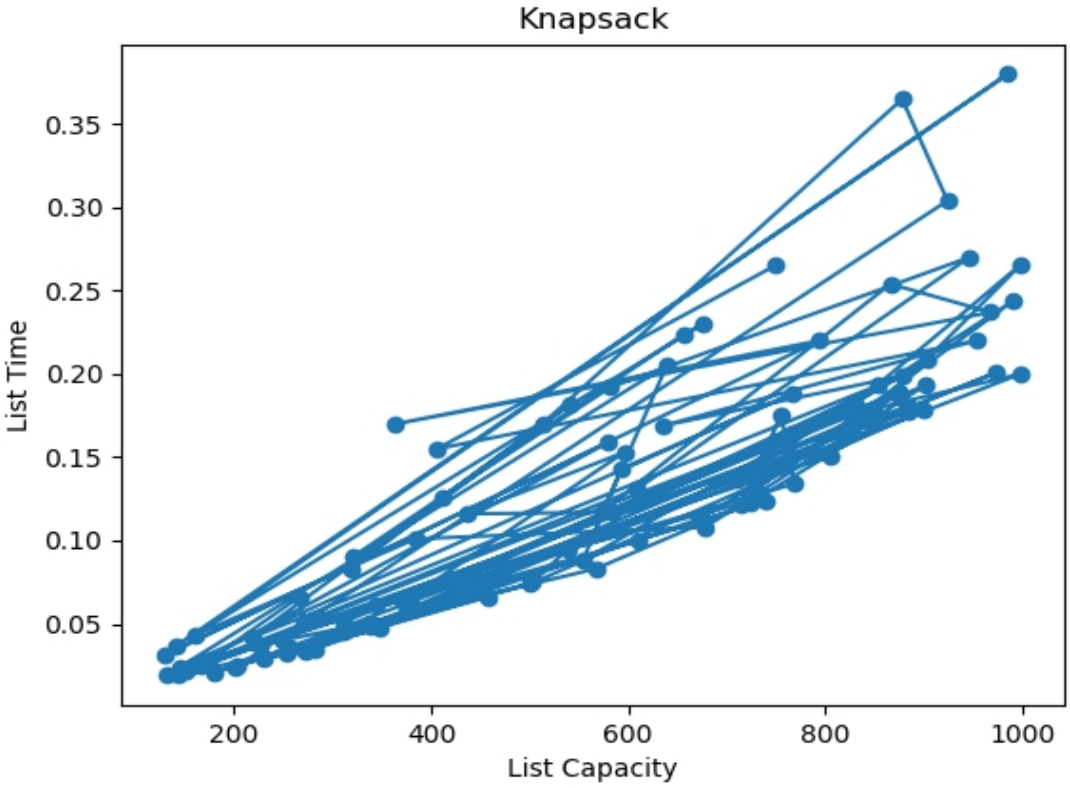
Giá trị có được khi xếp vào túi có sức chứa 50

130

Thời gian chạy thuật toán Change-Making: 0.0006963999999999998

- **Tổng quát hóa:**

  Cho danh sách gồm 100 trường hợp tương ứng là sức chứa từ 0 đến 1000. Tính thời gian chạy của từng trường hợp. Xuất ra biểu đồ tổng quát về thời gian chạy của thuật toán giải quyết bài toán đổi xu này.



Hình 4.3 Demo thời gian chạy của bài toán Knapsack

Nhận xét:

- *Thời gian chạy của thuật toán càng tăng khi sức chứa của túi càng tăng*
- *Có thể xem đây là sự đồng biến (tăng)*

CHƯƠNG V: KỸ THUẬT NHÁNH VÀ RÀNG BUỘC (BRANCH-AND-BOUND)

Là một mô hình thiết kế thuật toán cho các bài toán tối ưu hóa tổ hợp và rời rạc , cũng như tối ưu hóa toán học. Gồm liệt kê một cách có hệ thống các giải pháp ứng viên bằng phương pháp tìm kiếm không gian trạng thái : tập hợp các ứng viên được coi như tạo thành một cây gốc với tập hợp đầy đủ ở gốc. Thuật toán khám phá các "nhánh" (branch) của cây này, đại diện cho các tập con của tập giải pháp. Trước khi liệt kê các giải pháp ứng viên của một nhánh, nhánh được kiểm tra dựa trên các "ràng buộc" (bound) ước tính trên và dưới của giải pháp tối ưu và bị loại bỏ nếu nó không thể tạo ra giải pháp tốt hơn giải pháp tốt nhất được tìm thấy cho đến nay bằng thuật toán. Thuật toán phụ thuộc vào việc ước tính hiệu quả giới hạn dưới và giới hạn trên của các vùng hoặc nhánh của không gian tìm kiếm. Nếu không có giới hạn nào khả dụng, thuật toán sẽ chuyển thành tìm kiếm toàn diện (Brute-Force). Ở chương cuối cùng này sẽ sử dụng 3 bài toán kinh điển để minh họa gồm bài toán về vấn đề Phân công công việc (Job Assignment Problem), Người bán hàng (Traveling Salesman Problem) và N Quân hậu (N Queen Problem).

5.1 Ý tưởng thuật toán

5.1.1 Bài toán Phân công công việc (Job Assignment Problem)

Bài toán Phân công công việc được áp dụng để giải quyết vấn đề về việc sắp xếp và chỉ định các phần công việc được đặt ra để sao cho thời gian thực thi toàn bộ được tối thiểu nhất có thể.

Mô tả thuật toán:

Cho N công nhân và N công việc, bất kì người công nhân nào cũng có thể được chỉ định để thực hiện một công việc nào đó. Yêu cầu đặt ra là phải thực hiện tất

cả công việc được chỉ định một chính xác mỗi người công nhân sẽ thực hiện một công việc sao cho tổng chi phí bỏ ra để thực hiện được tối thiểu nhất.

Mã giả:

```
JobAssignmentProblem():
        e : nodepointer
        e = new(node)
        h: heap
        While(True) do:
                If(e is final leaf) then: Print(path from e to the root) Return
                EndIf
                Expand(E);
                If(h is empty) then: Print(No solution) Return
                EndIf
                E = E.remove(top(H))
        EndWhile
End
Expand(e):
        i = i ->e
        x,p: nodepointer
        S[1:N]: boolean
        p = e
        While (p is not the root) do:
                S[p->j] = 1
                p = p -> parent
        EndWhile
        For job=1 to N do:
                If(S[job] == 0) then:
```

x= new(node)

x->i+=1

x->j=job

x->parent=e

x->c = e->c + $a_{x->i,x->j}$ - $m_{x->i}$ /$m_{x->i}$ is min(row x->i)*/*

Insert(x,h)

EndIf

EndFor

End

5.1.2 Bài toán Người bán hàng (Traveling Salesman Problem)

Bài toán Người bán hàng được áp dụng để tìm đường đi ngắn nhất mà nó có thể đi qua tất cả các điểm và quay lại điểm bắt đầu mà không lặp lại hay tạo chu trình trong đồ thị hoặc trong các bài toán thực tế.

Mô tả thuật toán:

Bước 1: Chọn một đỉnh bắt đầu V.

Bước 2: Từ đỉnh hiện tại chọn cạnh nối có chiều dài nhỏ nhất đến các đỉnh chưa đến. Đánh dấu đã đến đỉnh vừa chọn.

Bước 3: Nếu còn đỉnh chưa đến thì quay lại bước 2.

Bước 4: Quay lại đỉnh V.

Mã giả:

```
def findMinimumEdgeCost(adjacencyMatrix, i):
    minCost = infinity
    For j = 0 to N do
```

```
            If (adjacencyMatrix[i, j] < minCost and i != j):
                minCost = adjacencyMatrix[i, j]
            EndIf
        EndFor
        return minCost

def findSecondMinimumEdgeCost(adjacencyMatrix, i):
        firstCost, secondCost = infinity, infinity
        For j = 0 to N do
            If (i == j)
                continue

            If (adjacencyMatrix[i, j] <= firstCost)
                secondCost = firstCost
                firstCost = adjacencyMatrix[i, j]

            Else If(adjacencyMatrix[i, j] != firstCost and adjacencyMatrix[i, j]
            <= secondCost):
                secondCost = adjacencyMatrix[i, j]
            EndIf
        EndFor
        return secondCost

def SolveTravelingSalesManProblem(currentPath, currentBound,
currentWeight, adjacencyMatrix, level, visited):
        If (level == N)
            If (adjacencyMatrix[currentPath[level - 1], currentPath[0]] != 0)
                currentResult = currentWeight +
                adjacencyMatrix[currentPath[level - 1], currentPath[0]]
            EndIf
            If (currentResult < finalResult)
                copySolutionToFinal(currentPath)
                finalResult = currentResult
            return
        For i = 0 to N do
```

```
        If (adjacencyMatrix[currentPath[level-1], i] != 0 and visited[i]
    == False)
                temp = currentBound
                currentWeight += adjacencyMatrix[currentPath[level - 1],
                i]
                If (level == 1)
                        currentBound -=
                        ((findMinimumEdgeCost(adjacencyMatrix,
                        currentPath[level - 1]) +
                        findMinimumEdgeCost(adjacencyMatrix, i)) / 2)
                Else:
                        currentBound -=
                        ((findSecondMinimumEdgeCost(adjacencyMatrix,
                        currentPath[level - 1])
                        +findMinimumEdgeCost(adjacencyMatrix, i)) / 2)
                EndIf
                If currentBound + currentWeight < finalResult:
                        currentPath[level] = i
                        visited[i] = True
                        SolveTravelingSalesManProblem(currentPath,
                        currentBound, currentWeight, adjacencyMatrix,
                        level + 1, visited)
                EndIf
                currentWeight -= adjacencyMatrix[currentPath[level - 1],
                i]
                currentBound = temp
                visited = [False] * len(visited)
                For j = 0 to level do
                        If currentPath[j] != -1:
                                visited[currentPath[j]] = True
                        EndIf
                EndFor

def TravelingSalesmanProblem(adjacencyMatrix):
        currentPath = [-1] * (N + 1)
```

```
visited = [False] * N
currentBound = 0
For i = 0 to N do
        currentBound += (findMinimumEdgeCost(adjacencyMatrix, i) +
        findSecondMinimumEdgeCost(adjacencyMatrix, i))
EndFor
currentBound = math.ceil(currentBound / 2)
visited[0] = True
currentPath[0] = 0
SolveTravelingSalesManProblem(currentPath, currentBound, 0,
adjacencyMatrix, 1, visited)
```

5.1.3 Bài toán N quân hậu (N Queen Problem)

N quân hậu là bài toán mô phỏng về việc đặt N quân hậu lên một bàn cờ có kích thước N x N mà không có hai quân hậu nào có thể gây xung đột lẫn nhau. Do đó, giải pháp được đưa ra để thực thi bài toán đó là sẽ không có hai quân hậu nào cùng nằm chung một hàng, cột hoặc đường chéo.

Mô tả thuật toán:

Đặt từng quân hậu vào các cột khác nhau bắt đầu từ phía ngoài cùng bên trái của bàn cờ, khi đặt chúng ta phải kiểm tra được xem các quân hậu có gây xung đột lẫn nhau hay không. Nếu trong một cột đang xét mà hàng của cột đó không có sự xung đột nào giữa các quân hậu với nhau thì ta sẽ đánh dấu cột và hàng đó. Ngược lại, nếu không tìm được bất kì cột và hàng nào mà không xảy ra xung đột thì ta quay lại và trả về kết quả của thuật toán là false.

Cần phải đảm bảo 3 yếu tố khi thực hiện thuật toán:

1. *Không có 2 quân hậu cùng chung một cột.*
2. *Không có 2 quân hậu cùng chung một hàng.*
3. *Không có 2 quân hậu cùng nằm trên cùng đường chéo.*

Mã giả:

```
PlacingChess(row, col)
    for i = 1 to row - 1 do:
        if (temp[i] == col || (abs(temp[i] - col) == abs(i - row))) do:
            return false
        endif
        else return true
    endfor
NQueenAlgorithm(row, N)
    for i = 1 to N do:
        if (PlacingChess(row, i)) do:
            temp[row] = i
        endif
    endfor
    if (row == N) do:
        print(temp[1...N])
    else do: NQueenAlgorithm(row + 1, N)
    endif
```

5.2 Triển khai bằng Python3

5.2.1 Bài toán Người bán hàng (Traveling Salesman Problem)

Yêu cầu:

Đầu vào: Ma trận kề của đồ thị.

Đầu ra: Đường đi với chi phí tối thiểu.

Mục đích: Tìm ra đường đi ngắn nhất từ một điểm bắt đầu đi qua hết tất cả các điểm và quay về điểm xuất phát.

Hiện thực bằng code:

```
infinity = 99999999999999999
def copySolutionToFinal(currentPath):
    finalPath[:N + 1] = currentPath[:]
    finalPath[N] = currentPath[0]
def findMinimumEdgeCost(adjacencyMatrix, i):
    minCost = infinity
    for j in range(N):
        if adjacencyMatrix[i][j] < minCost and i != j:
            minCost = adjacencyMatrix[i][j]
    return minCost
def findSecondMinimumEdgeCost(adjacencyMatrix, i):
    firstCost, secondCost = infinity, infinity
    for j in range(N):
        if i == j:
            continue
        if adjacencyMatrix[i][j] <= firstCost:
            secondCost = firstCost
            firstCost = adjacencyMatrix[i][j]

        elif(adjacencyMatrix[i][j] != firstCost and adjacencyMatrix[i][j]
        <= secondCost):
            secondCost = adjacencyMatrix[i][j]
    return secondCost
```

```
def SolveTravelingSalesManProblem(currentPath, currentBound,
currentWeight, adjacencyMatrix, level, visited):
    global finalResult
    if level == N:
        if adjacencyMatrix[currentPath[level - 1]][currentPath[0]] != 0:
            currentResult = currentWeight +
            adjacencyMatrix[currentPath[level - 1]][currentPath[0]]
        if currentResult < finalResult:
            copySolutionToFinal(currentPath)
            finalResult = currentResult
        return
    for i in range(N):
        if (adjacencyMatrix[currentPath[level-1]][i] != 0 and visited[i]
    == False):
            temp = currentBound
            currentWeight += adjacencyMatrix[currentPath[level -
            1]][i]
            if level == 1:
                currentBound -=
                ((findMinimumEdgeCost(adjacencyMatrix,
                currentPath[level - 1]) +
                findMinimumEdgeCost(adjacencyMatrix, i)) / 2)
            else:
                currentBound -=
                ((findSecondMinimumEdgeCost(adjacencyMatrix,
                currentPath[level - 1])
                +findMinimumEdgeCost(adjacencyMatrix, i)) / 2)
            if currentBound + currentWeight < finalResult:
                currentPath[level] = i
                visited[i] = True
                SolveTravelingSalesManProblem(currentPath,
                currentBound, currentWeight, adjacencyMatrix,
                level + 1, visited)
            currentWeight -= adjacencyMatrix[currentPath[level -
        1]][i]
```

```
                currentBound = temp
                visited = [False] * len(visited)
                for j in range(level):
                        if currentPath[j] != -1:
                                visited[currentPath[j]] = True

def TravelingSalesmanProblem(adjacencyMatrix):
        currentPath = [-1] * (N + 1)
        visited = [False] * N
        currentBound = 0
        for i in range(N):
                currentBound += (findMinimumEdgeCost(adjacencyMatrix, i) +
                findSecondMinimumEdgeCost(adjacencyMatrix, i))
        currentBound = math.ceil(currentBound / 2)
        visited[0] = True
        currentPath[0] = 0
        SolveTravelingSalesManProblem(currentPath, currentBound, 0,
        adjacencyMatrix, 1, visited)
        print("Path Taken : ", end = ' ')
        for i in range(N + 1):
                print(finalPath[i], end = ' ')
        print()
```

5.2.2 Bài toán N quân hậu (N Queen Problem)

Yêu cầu:

Đầu vào: Ma trận NxN(Bàn cờ)

Đầu ra: Ma trận đã giải quyết N Queen Problem

Mục đích: Tìm ra vị trí các quân hậu trên bàn cờ sao cho không quân hậu nào đe dọa nhau

Hiện thực bằng code:

```
N = 0
```

```
def checkToPlaceQueen(slashCode, backslashCode, row, col, rowCheck, slashCodeCheck, backslashCheck):
    if (rowCheck[row] or slashCodeCheck[slashCode[row][col]] or backslashCheck[backslashCode[row][col]]):
        return False

    return True

def solveNQueensProblem(chessboard, col, slashCode, backslashCode, rowCheck, slashCodeCheck, backslashCheck):
    if(col >= N):
        return True

    for i in range(N):
        if(checkToPlaceQueen(slashCode, backslashCode, i, col, rowCheck, slashCodeCheck, backslashCheck)):
            slashCodeCheck[slashCode[i][col]] = True
            backslashCheck[backslashCode[i][col]] = True
            chessboard[i][col] = 1
            rowCheck[i] = True

            if(solveNQueensProblem(chessboard, col + 1, slashCode, backslashCode, rowCheck, slashCodeCheck, backslashCheck)):
                return True

            slashCodeCheck[slashCode[i][col]] = False
```

```
                backslashCheck[backslashCode[i][col]] = False
                chessboard[i][col] = 0
                rowCheck[i] = False
        return False
def NQueensProblem_BrandAndBound():
        chessboard = [[0 for i in range(N)] for j in range(N)]
        slashCode = [[0 for i in range(N)] for j in range(N)]
        backslashCode = [[0 for i in range(N)] for j in range(N)]
        rowCheck = [False] * N
        slashCodeCheck = [False] * (2 * N - 1)
        backslashCheck = [False] * (2 * N - 1)
        for rs in range(N):
                for cs in range(N):
                        slashCode[rs][cs] = rs + cs
                        backslashCode[rs][cs] = rs - cs + 7
        if(solveNQueensProblem(chessboard, 0, slashCode, backslashCode,
        rowCheck, slashCodeCheck, backslashCheck) == False):
                print("Can't solve")
                return False
        for i in range(N):
                for j in range(N):
                        print(chessboard[i][j], end = " ")
                print()
        return True
```

5.3 Demo

5.3.1 Bài toán Người bán hàng (Traveling Salesman Problem)

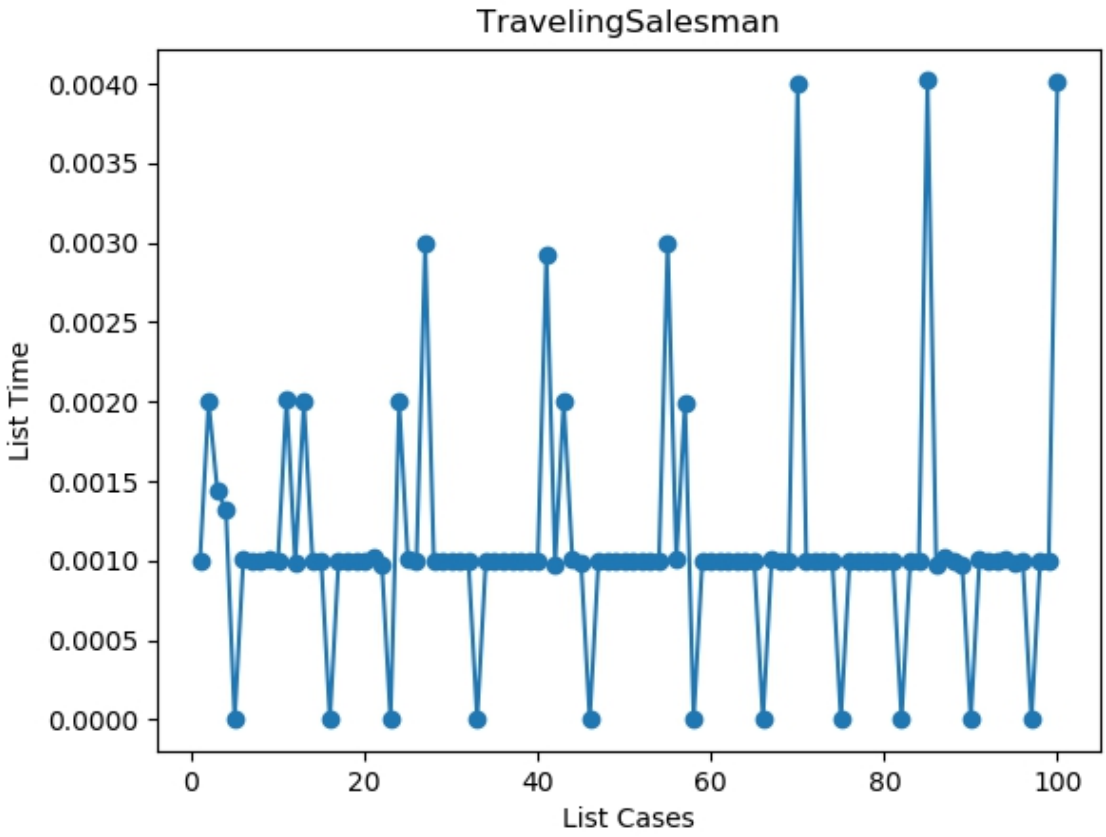
- **Mô phỏng nhỏ:**

```
adjacencyMatrix = [
        [0, 5, 7, 2, 10],
        [5, 0, 14, 3, 13],
        [7, 14, 0, 21, 6],
        [2, 3, 21, 0, 1],
        [10, 13, 6, 1, 0]
]
N = 5
finalPath = [None] * (N + 1)
visited = [False] * N
finalResult = maxsize
import timeit
start = timeit.default_timer()
TSP(adjacencyMatrix)
end = timeit.default_timer()
res = end - start
print("Thời gian chạy thuật toán Traveling Salesman:", res)
```

Kết quả từ màn hình Command Prompt:

```
Path Taken :  0 1 3 4 2 0
Thời gian chạy thuật toán Traveling Salesman: 0.0002524999999999958
```

- **Tổng quát hóa:**



Hình 5.2 Demo thời gian chạy của bài toán TravelingSalesman

Nhận xét:

- *Thời gian chạy trung bình của thuật toán để đi hết 5 địa điểm là khoảng 0.001s*
- *Trường hợp tốt nhất có thể đạt 0.00001s*
- *Trường hợp tệ nhất lên đến 0.004s*

5.3.2 Bài toán N quân hậu (N Queen Problem)

- **Mô phỏng nhỏ:**

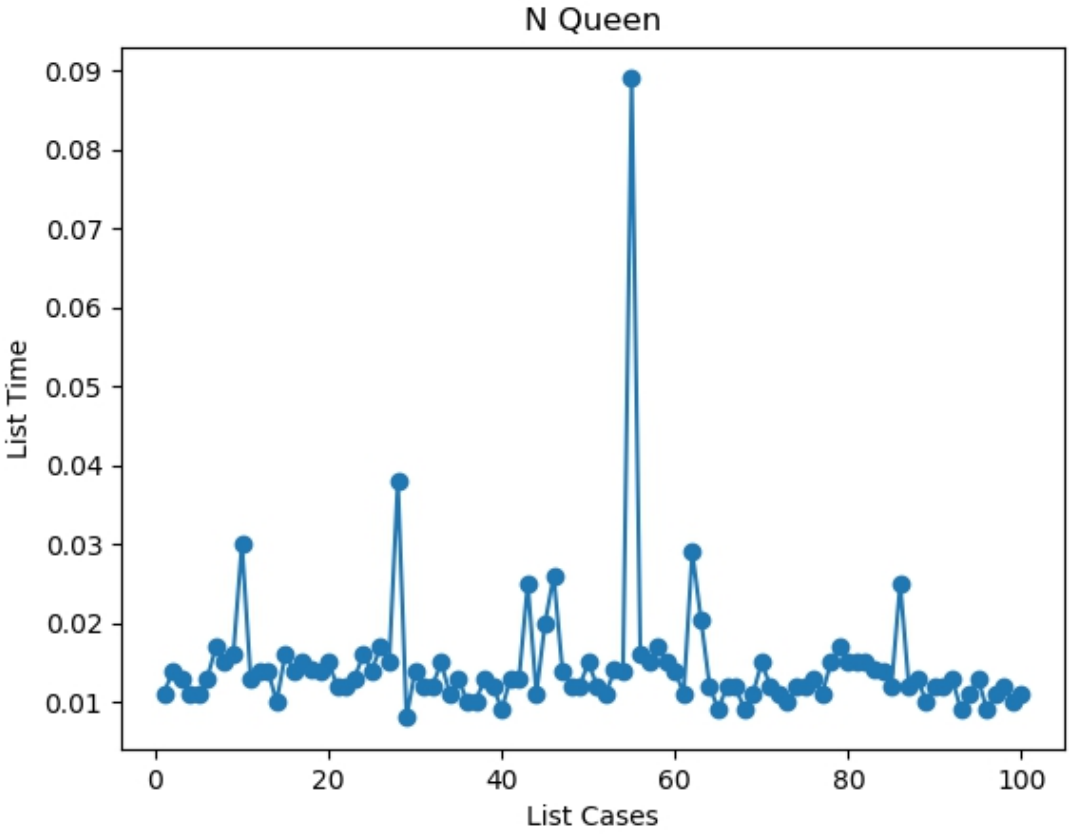
  $N = 8$

```
import timeit
print("N Queen solution with N = ", N)
start = timeit.default_timer()
SolveNQueens()
end = timeit.default_timer()
res = end - start
print("Thời gian chạy thuật toán N Queen Problem:", res)
```

Kết quả từ màn hình Command Prompt:

```
N Queen solution with N =  8
1 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1
0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0
0 0 1 0 0 0 0 0
Thời gian chạy thuật toán N Queen Problem: 0.0029681999999999972
```

- **Tổng quát hóa:**



Hình 5.3 Demo thời gian chạy của bài toán NQueen

Nhận xét:

- *Thời gian chạy trung bình của thuật toán để giải quyết bài toán của bàn cờ 8x8 rơi vào khoảng từ 0.008s đến 0.018s*
- *Trường hợp tệ nhất lên đến 0.09s*

CHƯƠNG VI – TỔNG KẾT

Bài báo cáo này đã giới thiệu về 5 kỹ thuật phổ biến nhất gồm kỹ thuật tìm kiếm toàn diện, kỹ thuật chia để trị, kỹ thuật thuật toán tham lam, kỹ thuật quy hoạch động, kỹ thuật nhánh và ràng buộc. Với mỗi kỹ thuật thiết kế trên đều được phân tích chi tiết về thông qua những bài toán cụ thể.

Tương ứng là những mô tả, phân tích về các yêu cầu đầu vào, đầu ra, mục đích được triển khai bằng ngôn ngữ lập trình Python. Cuối cùng là demo chi tiết bằng cách vẽ thời gian chạy dưới dạng một hàm của kích thước đầu vào. Qua đó, hiểu rõ hơn về quá trình phân tích và thiết kế một thuật toán. Có cái nhìn tổng quát hóa hơn về thuật toán thông qua thời gian chạy.

PHÂN CÔNG CÔNG VIỆC

Tên thành viên	Thời gian	Công việc cần làm
Tô Vĩnh Khang	22/10/2020 - 23/10/2020	Trình bày bố cục các phần trong báo cáo và phân công công việc cho thành viên nhóm.
	23/10/2020 - 03/11/2020	Trình bày Ý tưởng thuật toán (Chương I, II, III).
	03/11/2020 - 09/11/2020	Trình bày Triển khai bằng Python3, Demo (Chương I) và Slide thuyết trình (Phần 1).
	09/11/2020 - 10/11/2020	Tổng hợp tất cả nội dung và Nộp bài cuối kỳ.
Bùi Quang Khải	22/10/2020 - 27/10/2020	Trình bày Ý tưởng thuật toán (Chương IV).
	27/10/2020 - 09/11/2020	Trình bày Triển khai bằng Python3, Demo (Chương II, III) và Slide thuyết trình (Phần 2).
Du Thuận Long	22/10/2020 - 27/10/2020	Trình bày Ý tưởng thuật toán (Chương V).
	27/10/2020 - 09/11/2020	Trình bày Triển khai bằng Python3, Demo (Chương IV, V) và Slide thuyết trình (Phần 3).

TÀI LIỆU THAM KHẢO

[1] Anany Levitin, Introduction to The Design and Analysis of Algorithms 3rd edition, [2012] , Villanova University, 123-466.