Homework 4

Bài 1:

1. Tìm đường đi từ A đến B trong mê cung.
    - Một mê cung có n phòng. Đường đi giữa các phòng được mô hình dưới dạng ma trận M là ma trận vuông, đối xứng n * n. Giá trị tại M[i][j] = 1 nếu có đường đi trực tiếp từ phòng i sang phòng j và = 0 nếu ngược lại.
    - Do mỗi bước đi phải sang một phòng khác nên ta đặt M[i][i] = 0 $\forall i \leq n$. Tức là không có đường đi trực tiếp từ phòng i sang phòng i.
    - Lời gọi ban đầu sẽ là Try(1) để phù hợp với chỉ số của mảng bắt đầu từ 0.
    - Thuật toán sẽ cho ra kết quả phù hợp đầu tiên, không phải kết quả tốt nhất.
    - File code maze.java

2. Bài toán người bán hàng.
    - Các thành phố và đường đi giữa các thành phố biểu diễn dưới dạng ma trận vuông.
    - Giá trị M[i][j] = 0 nếu không có đường đi giữa 2 thành phố i và j hoặc là chi phí đi từ thành phố đến thành phố j.
    - Giả sử chi phí đi từ i đến j bằng chi phí đi từ j đến i nên ma trận sẽ là ma trận vuông đôi xứng.
    - Để đơn giản bài toán ta sẽ đặt các phần tử trên đường chéo chính của ma trận bằng 0, tức là sẽ không có đường đi từ i đến i.
    - File code travelingSaleman.java

Bài 2:

1. Liệt kê các hoán vị của n phần tử
    - Phân tích bài toán
        - Input số nguyên n
        - Output: Liệt kê các hoán vị của tập các số nguyên từ 1 đến n ra màn hình.
        - Xét tập S = {1, 2, …, n} có n phần tử.
        - Mỗi số trong tập S chỉ xuất hiện 1 lần trong mỗi hoán vị nên ta dùng mảng used[1 … n] để đánh dấu số i thuộc tập S đã được sử dụng trong hoán vị hiện tại chưa.
          used[i] = 0 nếu số i chưa được sử dụng, = 1 nếu ngược lại.
        - Tại mỗi bước ta sẽ duyệt các các số S từ bé đến lớn, xem số i đã được sử dụng hay chưa, nếu chưa ta thêm i vào mảng hoán vị.

- Do mỗi hoán vị sẽ có n phần tử nên khi mảng chứa hoán vị đầy, ta sẽ in ra mảng đó và quay lui lại để sinh cấu hình ngay trước đó.

- Thuật toán:

```
Try(i){
    for j = 1 to n do
            if(used[j] == 0)
                    permutation[i] = j;
                    used[j] = 1;
                    if (i == n)
                            print(permutation[])
                    else Try(i + 1)
                    used[i] = 0

}
```

- Lời gọi khởi tạo: Try(1)
- File code permutation.java

2. Liệt kê tổ hợp k phần tử của các số từ 1 đến n
    - Phân tích
        - Liệt kê các tập con k phần tử của tập S = {1, 2, …, n}
        - Input: số nguyên n và k
        - Output: Hiển thị các tập con có k phần tử của tập S ra màn hình, mỗi tập con chỉ xuất hiện 1 lần.
        - Do các tổ hợp không lặp lại nê ta sẽ liệt kê các cấu hình $x_1x_2x_3 \cdots x_k$ với $x_i \in S$, và $x_i < x_{i+1}\ \forall\ i$.
        - Tại bước sinh cấu hình thứ i, ta xét các phần tử chưa được dùng thông qua mảng used[1 … n](used[i] = 0 nếu số i chưa được sử dụng , = 1 nếu ngược lại) nếu số j thỏa mãn used[j] = 0 và j > min với min là subSet[i - 1] thì ta đưa j vào cấu hình thứ i.
        - Mỗi tổ hợp chỉ có k phần tử nên khi i = k ta sẽ in ra tổ hợp đã lưu trong mảng và quay lui lại việc sinh cấu hình trước đó.
    - Thuật toán

```
Try(i, min) {
        for j = 1 to n do{
                if( used[j] == 0 and j > min){
                        subSet[i] = j
                        used[j] = 1
                        if(i == n) print(subSet)
                        else Try(i +1, j)
                        used[j] = 0
                }
        }
}
```

- Lời gọi khởi tạo: Try(1,0)
- File code subSet.java

Bài 3:

1. Phương pháp quay lui
    - Bài toán tô màu bản đồ: Cho 1 bản đồ có n quốc gia. Mỗi quốc gia sẽ có đường biên giới với một số nước nhất định. Ta cần tô màu cho bản đồ trên bằng m màu khác nhau, sao cho 2 quốc gia bất kì có đường biên giới chung thì không được tô cùng màu.
    - Phân tích
        - Bản đồ được mô tả dưới dạng ma trận đối xứng, các phần tử là 1 nếu quốc gia thứ i và quốc gia thứ j có chung biên giới, = 0 nếu ngược lại.
        - input: ma trận biểu diễn và số màu cần để tô m.
        - output: 1 cách tô màu hoặc thông báo không có cách tô màu nào thỏa mãn.
        - Tại bước thứ i ta sẽ tô màu cho quốc gia thứ i. ta duyệt các màu lần lượt từ 1 đến m, kiểm tra xem màu đó có thỏa mãn yêu cầu. Nếu có ta tô màu đó cho quốc gia i. Khi tô màu xong cho quốc gia thứ n thì ta in ra màn hình kết quả.
        - Hàm check(int vertex, int color) return true nếu màu color có thể dùng để tô màu cho quốc gia vertex, return false nếu ngược lại.
        - Kết quả sẽ được lưu trong mảng coloring[1 … n], nếu trong mảng chứa bất kì phần tử nào khác các số nguyên từ 1 đến m thì với m màu ta không tìm được cách tô màu thỏa mãn.
    - Thuật toán

```
Try(i) {
        for j = 1 to m do{
                if( check(i, j)){
                        colored[i] = j;
                        if(i == n)
                                print(coloring) // in ra 1 cách tô màu cho bản đồ
```

```
                                return;
                            else Try(i + 1)

                    }
            }
}
```

- Lời gọi ban đầu Try(1)
- Đánh giá thuật toán.
  - Trường hợp xấu nhất: $O(n^m)$. Xảy ra khi không có cách tô màu nào thỏa mãn.
  - Trường hợp tốt nhất: O(n). Xảy ra khi mỗi lần tìm cấu hình, ta tìm được màu phù hợp từ những màu đầu tiên, tức là đồ thị không có bất kì cạnh nào hay ma trận chứa toàn các phần tử 0.
  - Trường hợp bình thường: Bản đồ sẽ được tô đầy đủ bằng m màu, số lần sử dụng của mỗi màu trung bình là n/m. Vậy số bước thực hiên là $\frac{n}{m}\sum_{i=1}^{m} i = \frac{n}{m}\frac{m*(m-1)}{2} = \frac{n*m}{2}$. Tức là thuật toán có độ phức tạp O(n * m).
- File code mapColoring.java

2. Phương pháp nhánh cận
   - Bài toán chia việc 1 - 1: Có n công việc cần được chia cho n công nhân, mỗi công nhân khi thực hiên một công việc bất kì sẽ tốn 1 khoản chi phí. Tìm cách chia việc sao cho mỗi công nhân làm 1 việc và chi phí bỏ ra là nhỏ nhất.
   - Phân tích
     - Danh sách chi phí mà mỗi công nhân thực hiện công việc được biểu diễn dưới dạng ma trận.
     - Phần tử A[i][j] của ma trận là chi phí của công nhân I khi thực hiện công việc j. A[i][j] > 0. Nếu công nhân i không thể thực hiện công việc j thì A[i][j] = ∞, nhưng để thuận tiện khi lập trình ta sẽ để A[i][j] = 0.
     - input: ma trận biểu diễn của chi phí.
     - Output: Cách chia việc hợp lý nhất: Bất kì công nhân nào cũng nhận việc và chi phí bỏ ra là thấp nhất. Kết quả được lưu trong mảng assign[1 … n]. giá trị của assign[i] là công việc được chia cho người thứ i. => tìm hoán vị của dãy từ 1 đến n sao cho chi phí là nhỏ nhất.
     - Chi phí tốt nhất là bestCost. Ban đầu bestCost = ∞.
     - Tại mỗi bước i ta chia việc cho nhân viên thứ i. Việc mà người thứ i nhận phải chưa được ai nhận. Tính chi phí lời giải nhận được. Nếu "tốt hơn" lời giải hiện thời thì chia việc cho người

thứ i và tiếp tục chia việc cho người thứ i + 1. Nếu không có khả năng nào thì lùi lại bước chia việc cho người i - 1.

    - Với mỗi khả năng chia việc j cho người I ta tính chi phí c1 = c + A[i][j]. Nếu c1 > bestCost thì ta quay lui, ngược lại ta tiếp tục sinh cấu hình i + 1. Đến khi gặp nghiệm i = n, cập nhật cách chia việc tốt nhất và chi phí tối thiểu.

- Thuật toán

```
Try(i, c) {
          for j = 1 to n do{
                    if( used[j] == 0){
                              c1 = c + A[i][j]
                              if (c1 < bestCost){
                                        asign[i] = j
                                        used
                                        if(i == n){
                                                  result = assign[1 .. n]
                                                  bestCosst = c1
                                        } else Try(i +1, c1)
                              }
                    }
          }
}
```

- Đánh giá thuật toán
    - Trong mọi trường hợp ta đều cần xét mọi trường hợp có thể chia việc. Cho nên thuật toán sẽ chỉ tiết kiệm khi gặp các trường hợp cần quay lui. Giả sử chi phí mỗi người cần để thực hiện 1 công việc có độ lớn tương đương nhau, khi đó ta sẽ quay lui tại những bước sinh cấu hình cuối cùng. Cho nên dù có cải thiện so với phương pháp vét cạn nhưng độ phức tạp vẫn ở mức $O(n!)$
- File code assignment.java