第三期JAVA課題

種市隼兵
学籍番号：6313067

平成２６年7月１６日

1　課題

2　アルゴリズムの説明

2.1　更なる速さを求めて

提出が遅くなりました。なかなかコードが書く気が起きず、だらだらと過ごしていたら、提出日当日になり、その日の内に書き上げようと思いましたが思った以上に手こずってしまい提出する事が出来ませんでした。なので、遅れて出すからにはちゃんと速さにこだわろうと思い、色々試行錯誤してみる事にしました。条件として次のことを決めました。

2.1.1　flag を使わない

ナップサックの問題で、flag を用いて解いていましたが、あれはアイテムに対しフラグを折る、折らないをしました。しかし今回の場合余り深くは考えていませんがフラグを折る、折らないをするのは今いる駅から次の到着駅へのレールに対してです。そうした場合駅数が 5 駅の場合は自分から自分への移動も含めると 5 × 5 の 25 通り、100 駅あった場合は 100 × 100 の 10000 通りもフラグがある事になります。これでは処理速度も遅いし、無駄なフラグもたくさん出てきてしまいます。そのため、これの使用はやめました。

2.1.2　for 文は極力使わない

再帰的メソッドは、あたかもそれ自身がイテレータによってまわされているかのような振る舞いをします。そのため、その再帰的メソッドの内部で複数回 for 文をまわすのはどこか気持ち悪く、ナンセンスに感じました。なので、for 文の使用頻度は必要最低限に押さえようと思いました。

2.1.3　デバッグはしやすく

再帰的なプログラムは、僕が知っているなかで 1,2 位を争うわかりにくさです。このブロックに入ったら、次にどこにいって、そのときの状態がどんなものなのかが想像しにくいです。そのため、本質的には余り関係のないところ、真偽判定の所などをわかりやすく書く事を求めました。

2.1.4　自分の想像に忠実に

自分が考えたアルゴリズムにあくまで忠実に振る舞ってくれるコードを考えました。コードを書いているとき、一番やってはいけないことは、”自分が

何をやっているのかわからない、でもなんかできたからいいや” という思考です。これではヒエラルキー的立場が真逆になってしまいます。コードであそぶのではなく、コードにあそばれてしまっているのです。

以上が、僕がこのコードを書く為に心がけたことです。次は具体的なアルゴリズムの説明をしようと思います。

2.2 何を引き継いでいくべきか

さて、アルゴリズムの具体的な説明です。次のようなものを考えました。

- 最寄り駅がゴールならば、そのゴールまでにかかったコストを他のルートでゴールしたときのものと比較し、そのコストより小さかった場合新しく最小コストのルートとして記録する。
- 最寄り駅から、通った駅以外に行ける駅が無いとき、一つ前の駅にもどり自分以外でまだ行ってない駅が無いか探す。
- 最寄り駅まで来るコストが、他のルートで来たときのコストよりも高い場合、そこで計算をやめる。

以上この三つが、僕がコードを書くときに考えた事です。２番は、一番最初から呼び直すより、一つ上に戻ったほうが無駄が無いと思ったからです。３番は、そこまでくるのに他のルートの方がコストが低かったら、そちらの方が最短に決まっているよね、という考えのもとで条件付けしました。

このアルゴリズムの一番の味噌となるところは、その駅を通った事があるか否か、ということです。これをどうやって表現するかどうかで、迷いましたが、結果的に ArrayList を用いる事で解決しました。ArrayList を上手く用いてあげる事によって、再帰的なプログラムで引き渡す値がコストと現在いる場所だけでよくなったのは嬉しい誤算でした。

これらの考えのもと、完成させたプログラムは、最後のサンプルを僕の PC でおよそ一秒で解いてくれるプログラムとなりました。五秒以内、というのが目標でしたが、それより速く処理してくれたのは驚きでした。

3 プログラムの説明

プログラムの説明をしたいと思います。可読性向上のため、複数のコンストラクタをつくり、それを順に読込む事によってなるべくモジュール化し、互いに疎になるようにつとめました。最初に大まかなコンストラクタのおおまかな役割説明をします。

main -> Search を呼び出す。
Search -> 今回のプログラム全ての源。
DefaultSettings -> 実行時に入力された第二引数を元に配列の要素数を定義
EntryValue -> した二つのコンストラクタを呼ぶユーザー入力にたいしての操作
AcceptValueFromUser -> ユーザーに最寄り駅と、最終地点の入力を求める
InsertValueAddedByUser -> 入力された値を元に変数に値を代入していく
StationWay -> 最短距離を求めるプログラム

それでは一つずつみていきます。

3.1 main

```
public static void main(String args[]){
  new Search(args);
}
```

言うまでもないですね。java コマンドを入力したときの引数を args という配列で受け取って、それを Search に渡しています。

3.2 Search

```
public Search(String args[]){
  fileName = args[0];
  n = Integer.parseInt(args[1]);
  DefaultSettings();
  EntryValue();
  StationWay(startNumber,0);
  System.out.println(minRoute);
  System.out.println(minCost);
}
```

第一引数で受け取ったものを fileName という変数に代入。また、第二引数で受け取ったものを n という変数に代入しています。fileName は open するファイルの名前、n は駅の総数を受け取る事になっています。そして他のコンストラクタを呼び出した後、処理結果である最短距離と、コストを表示してくれます。

3.3 DefaultSettings

```
private void DefaultSettings(){
  data = new int[n][n];
  station = new String[n];
  stationCost = new int[n];
}
```

data には二次元配列電車の相関マップデータが代入されます。それの配列の要素数を定義しました。station 変数には、それぞれの行（列）番号の駅名、stationCost にはそれぞれの行（列）番号のコストを受け取ります。

3.4 EntryValue

```
private void EntryValue(){
  do{
    AcceptValueFromUser();
    InsertValueAddedByUser();
  }while(!isGoalExist || !isStartExist);
}
```

ここではユーザー入力を求めること、その値をもとに変数に value を記録していく事をします。isGoalExist, isStartExist は共に真偽値を返す引数で、もし入力された値が存在していなかった場合、もう一度これらを呼び出すようになっています。

3.5 AcceptValueFromUser

```
private void AcceptValueFromUser(){
  try{
    BufferedReader input;
    System.out.println("サンプルデータ内に存在する駅名を入力して
下さい。");
    System.out.println("From?");
    input = new BufferedReader (new InputStreamReader (System.in));
    start = input.readLine( );
    System.out.println("To?");
    input = new BufferedReader (new InputStreamReader (System.in));
    goal = input.readLine( );
  }catch(IOException e){
    e.printStackTrace();
```

```
  }
}
```

BufferReader を用いてユーザー入力を求めます。start, goal に値を代入します。

3.6 AcceptValueFromUser

```
private void InsertValueAddedByUser(){
  try{
    BufferedReader br;
    FileReader fr;
    int row = 0;
    int i;
    fr = new FileReader(fileName);
    br = new BufferedReader(fr);
    while (br.ready()) {
      String str = br.readLine();
      if(row < n){
        station[row] = str;
        if(isStationStartName(row)){
          startNumber = row;
          isStartExist = true;
        }
        if(isStationGoalName(row)){
          goalNumber = row;
          isGoalExist = true;
        }
      }else{
        String[]tmp = str.split(" ");
        for(i=0; i<n; i++){
          data[row-n][i] = Integer.parseInt(tmp[i]);
         }
      }
      row++;
    }
  }catch(IOException e){
    e.printStackTrace();
  }
}
```

```
  private boolean isStationStartName(int row){
    return start.equals(station[row]) || start.equals(station[row].toLowerCase());
  }
  private boolean isStationGoalName(int row){
    return goal.equals(station[row]) || goal.equals(station[row].toLowerCase());
  }
```

ユーザーから入力された goal と start の値、及び file 内のデータを全て変数に代入しています。br.ready() を用いて文字が読込める間ひたすらまわし、行が n(駅数) 未満の時は、station[n] に駅名を登録し、n 行目からはマップデータになるので、それを data[n][n] に代入しています。また、station[n] を登録するついでに、それが goal,start の文字列と等しいかを判断します。それを判断しているのが、した二つの isStationStartName() と isStationGoalName の真偽判断メソッドです。小文字で打ったときや、大文字で打ってしまったときも、駅名が正しければ true を返すようになっています。

3.7 StationWay

```
private void StationWay(int stationNumber, int cost){
    int i;
    if(isStationGoal(stationNumber)){
      if(isCostLowerThanMinCost(cost)){
        minCost = cost;
        route = "";
        for(i = 0; i < passedStationNumber.size(); i++){
          route += station[passedStationNumber.get(i)] + "->";
        }
        minRoute = route + goal;
      }
      return;
    }

    for(i=0;i<n;i++){
      if(isStationExist(stationNumber,i) && !isStationAlreadyThrough(i)){
        if(isStationCostNullOrHigher(stationNumber, cost, i)){
          stationCost[stationNumber] = cost;
          cost += data[stationNumber][i];
          passedStationNumber.add(stationNumber);
```

```
          StationWay(i, cost);
          cost -= data[stationNumber][i];
          passedStationNumber.remove(passedStationNumber.size() - 1);
        }
      }
    }
  }

  private boolean isCostLowerThanMinCost(int cost){
    return minCost == 0 || minCost > cost;
  }

  private boolean isStationGoal(int from){
    return from == goalNumber;
  }

  private boolean isStationExist(int i, int j){
    return data[i][j] != 0;
  }

  private boolean isStationAlreadyThrough(int i){
    return passedStationNumber.indexOf(i) > -1;
  }

  private boolean isStationCostNullOrHigher(int stationNumber, int cost, int i){
    return stationCost[i] == 0 || cost + data[stationNumber][i] <= stationCost[i];
  }
```

これが最短を求めるコードです。まず真偽判断メソッドからみていきましょう。

`isCostLowerThanMinCost` -> ゴールにたどり着いたコストが、
最小のコストよりも大きいかそうでないかを調べる

最初は 0 なので、そのときも真を返す。

`isStationGoal` -> 最寄り駅が、ゴールかどうか判断する。

`isStationExist` -> 最寄り駅から、その駅に行けるかどうかを判断する

isStationAlreadyThrough -> その経路を既に通ったかどうかを判断する。

isStationCostNullOrHigher -> 行き先である駅を一度利用した事があるか、
その場合ここまでかかっているコストより
高いコストか否か

アルゴリズムのところで話したように、メソッド名で何をしたいかわかるようにしました。for 分の中身を見てみましょう。

```
for(i=0;i<n;i++){
  if(isStationExist(stationNumber,i) && !isStationAlreadyThrough(i)){
    if(isStationCostNullOrHigher(stationNumber, cost, i)){
      stationCost[stationNumber] = cost;
      cost += data[stationNumber][i];
      passedStationNumber.add(stationNumber);
      StationWay(i, cost);
      cost -= data[stationNumber][i];
      passedStationNumber.remove(passedStationNumber.size() - 1);
    }
  }
}
```

最初の if で今いる駅からその先の駅に行けるかの判断及び、既に通ったか否かを判断しています。次に、そこまでかかったコストが今までできた他のルートよりも高かったか高くないかを判断し、真が帰ってきたときのみブロック内に入ります。まず、今いる自分の駅までのコストを記憶、次に今いる駅から次の駅に行くまでかかるコストを記憶とおった駅として今いる駅を記憶します。そして再び自分自身を呼び出します。そして i が回りきった (どこにもいけなくなった)、もしくはゴールにたどり着いたときに、自分自身をよんだ一つ上のメソッドに戻り、その呼び出したコードの下にある

```
cost -= data[stationNumber][i];
passedStationNumber.remove(passedStationNumber.size() - 1);
```

が実行されます。一つ上のメソッドに戻るということは、つまり、一つ駅を巻き戻す、ということです。なので、下の層に行ったときに足した cost 及び通った駅を削除しました。

再帰メソッドは、例えばイテレータが 4 回まで回ったときに次のメソッドがよばれ奥に入り込んで、そこでのイテレータが回りきって上に戻ってきたとき、イテレータは 5 から再び回り始めます。この性質を上手く用いました。

```
  if(isStationGoal(stationNumber)){
    if(isCostLowerThanMinCost(cost)){
      minCost = cost;
      route = "";
      for(i = 0; i < passedStationNumber.size(); i++){
        route += station[passedStationNumber.get(i)] + "->";
      }
      minRoute = route + goal;
    }
    return;
  }
```

そして最後にゴールにたどり着いたときのコードです。真偽判断をしたあと、minCost に cost 、そして minRoute に今まで通ってきた駅名に→をたし、最後に goal の駅名を足してあげれば完了です。

これでコードの説明は終わりです

4 実行例

```
└─$ time java Search sample4.txt 101
サンプルデータ内に存在する駅名を入力して下さい
From?
mitaka
To?
abiko
Mitaka->Ogikubo->Nakano->Takadanobaba->Ikebukuro->Sugamo->Komagome->Nishinippori->Kitasen
61
java -Dfile.encoding=UTF-8 Search sample4.txt 101  1.15s user 0.43s system 39% cpu 3.993
```

5 考察

真偽判断メソッドのようなものを作らず、平文 (地の文) ですべて記述したらもう少し速くなるのかなっと思いました。ですが、僕は速さよりも可読性が大事だと考えているので、分けました。stationWay の中身もさらにこまかくメソッドとしてわけてもよさそうでした。一番上からか、はたまた一つ上の階層に戻って処理を実行するか処理速度に雲泥の差がでるなとおもいました。flag ではやりませんでしたが、ArrayList を用いて正しかったと思います。