情報工学実験IIレポート（探索アルゴリズム1）

月曜日　7グループ

2010年12月13日

グループメンバ

（補足：レベル毎に 全員が協力して実施 した上で、レベル毎にレポートをまとめる担当者を決め、全体を一つのレポートとして整理すること。分担方法も自由である。）

- 095701B 青木 史林: 担当 Level1
- 095703J 岩瀬 翔　: 担当 Level3
- 095707B 大城 佳明: 担当 Level2

1 Level1: 探索とは

1.1 Level1.1: コンピュータと人間の違いを述べよ

1.1.1 課題説明

コンピュータが人間より得意とするモノ、その反対に人間より不得手のモノ、両者について 2 つ以上の視点（立場や観点など）を示し、考察する。

1.1.2 考察

- 人間と比較したときのコンピュータにとっての利点
  コンピュータならば高速な演算が可能である。
  人間よりも論理的な演算や処理を実行できる。

- 人間と比較したときのコンピュータにとっての欠点
  人間のような創造性に欠ける。
  その場での適応力・感性を持ち合わせることが難しい。

1.2 Leve1.2: 具体事例を挙げ、「探索」という観点から考察せよ

1.2.1 課題説明

システムが実世界または仮想世界で作業をする問題（具体事例）を 1 つ挙げ、その問題を解決するにあたって必要となる機能・技術等について考察する。

1.2.2 システム概要図

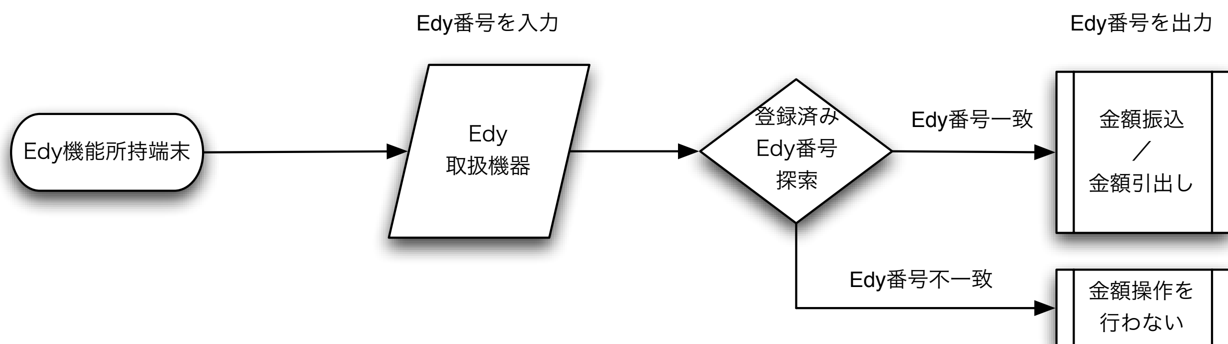


図 1: 入出力と内部モデルのイメージ図

1.2.3 入力・内部モデル・出力の説明

上記は現在広く用いられている Edy のシステムを考察したものである。たとえば Edy にチャージする場合、Edy 取り扱い機器に Edy のチップを読み込ませて固有の Edy 番号を入力する。次に、Edy 取扱機器は登録されている Edy 番号からチャージ申請された Edy 番号と一致するものを探索する。番号が一致した場合、その Edy 番号の金額の残高などの情報を取得することができる。また、番号が一致したことにより Edy 内部の情報操作が可能となり、金額振込／引出操作を行うことができる。

1.2.4 問題空間の定義や説明、この問題ならではの特徴など

- 問題点たとえば Edy を紛失した場合、その Edy を拾った他者はその Edy にチャージされている金額を自由に使うことができる。
- 解決策 Edy を使用する際に Edy 番号とは別の、使用者のみ知るパスワードを入力しなければ Edy を使用できないようにする。これによってセキュリティが向上するため、不本意に Edy を使用される危険性が大幅に減少すると思われる。

1.2.5 探索方法の説明

(b) 推測のパターンで考える。

Edy 番号は一般的には 4 桁区切りの 16 桁で登録されているが、この区切りの利点を考える。16 桁を一度に探索するのは時間にコストがかかるため、4 桁区切りで Edy 番号探索を 4 分割して効率の向上を図っていると考えられる。さらに番号探索をより早くするため、Edy チャージなど頻繁に利用される Edy 番号を優先的に探索の上位に登録する。
以上の探索法が Edy に用いられているのではないかと推測する。

2 Level2: 連続関数における探索手法を検討し、その効率を示せ

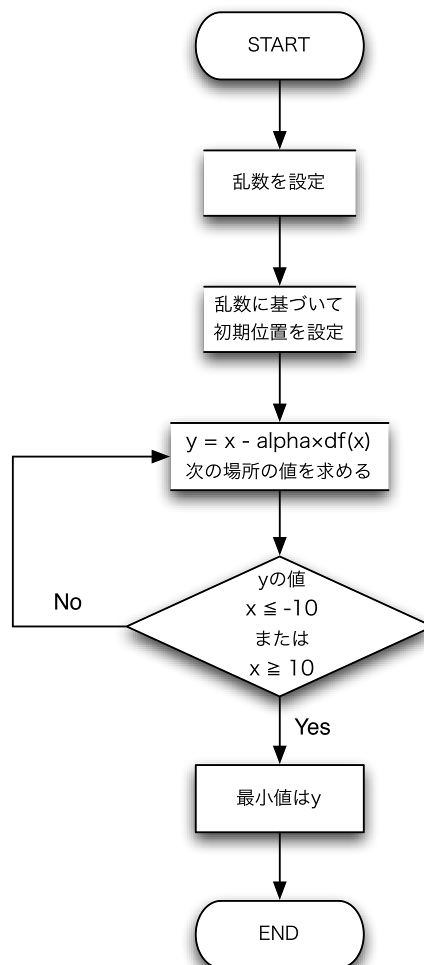
2.1 課題説明

3 種類の連続関数 $y = x$、$y = x^2$、$y = -x \times sin(x)$ について、再急降下法により最小値を求めるプログラムを作成し、計算機実験により評価し、結果を示した上で考察する。

2.1.1 探索の手続き（共通部分）

詳細は以下のフローチャートの図に掲載してある。
まずはランダムに初期位置を設定し、最小値の方向（傾きが負の方向）へ向かって辿って行く。

2.1.2 フローチャート（共通部分）



2.1.3 Level2 で用いたオリジナルシェルスクリプト

以下に Level2 全般で用いた、グラフを作成するシェルスクリプト「p_ave.sh」を載せる。

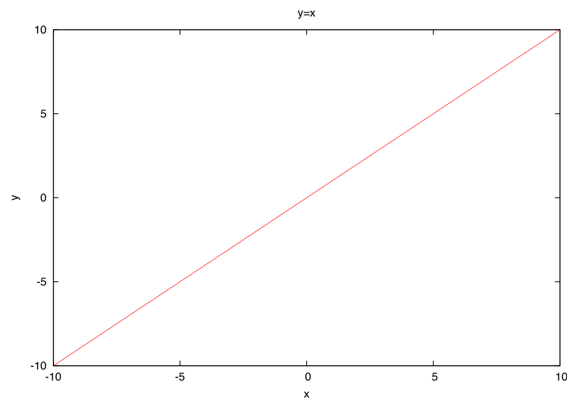
表 1: Level 2.3 にて用いたシェルスクリプトソース

```
#!/bin/sh
rm test.txt
./a.out >> test.txt
#------------gnuplot で出力する際に適宜名前を設定し直す必要がある------------#
#------------出力画像の形式を設定．ここでは eps で出力する-----------#
echo 'set terminal postscript eps color' > test.gnuplot

#------------eps ファイル名を決定------------#
echo 'set output "L3_graph-3.eps"' >> test.gnuplot

#------------グラフのタイトルを設定-----------#
echo 'set title "0.047 <= alpha <= 0.184"' >> test.gnuplot

#------------x 軸のラベルを設定-----------#
echo 'set xlabel "alpha"' >> test.gnuplot

#------------y 軸のラベルを設定----------#
echo 'set ylabel "trial_ave"' >> test.gnuplot



#------------x 成分、y 成分の両方のデータを含んだテキストファイルを読む----------#
#------------実行すると、指定した名前の eps ファイルが出力される ------------#
echo 'plot "L3_graph-3.txt" with lines'>> test.gnuplot

gnuplot < test.gnuplot
#------end for gnuplot
```

2.2 Level2.1: $y = x$

2.2.1 プログラムソース（steepest_decent3_2.c による変更部分）

steepest_decent.c（静的に $alpha$ の値を決めたときのソース）




```
----中略----

double f(double x) {
  double y;

  y = x;   // y = x のグラフである

  return( y );
}
double df(double a) {
  double y_dx;

  y_dx= 1;  // y = x を x で微分した結果である

  return( y_dx );
}

int main(int argc, char **argv) {

----中略----

 for (i = 1; i < term_cond; i++) {
    /* step2. 次の探索場所へ移動 */

    x = x-alpha*df(x);

----中略----
```

2.2.2 実行結果 (steepest_decent1_1.c)

2.2.3 考察

seed の値は 0〜99999 までの結果を調べた。上記の表 1 は静的に *alpha* の値を決めた場合である。
次に、以下に 0〜10 まで *alpha* の値を変えた場合の図 2 を、0.1〜0.9 までの *alpha* の値を変えた場合の図 3 を以下にそ

表 2: y=x

alpha:0.000000―――――――――――
max:9.999870 min:-9.999773 ave:0.000168

alpha:0.100000―――――――――――
max:0.099870 min:-10.099990 ave:-7.474425

alpha:0.200000―――――――――――
max:-9.800130 min:-10.199961 ave:-10.097954

――中略――

alpha:0.900000―――――――――――
max:-10.000003 min:-10.899985 ave:-10.453485

alpha:1.000000―――――――――――
max:-10.000003 min:-10.99996 ave:-10.499962

れぞれ示す。ただし、x 軸は *alpha* 値、y 軸はループ回数の平均を示す。

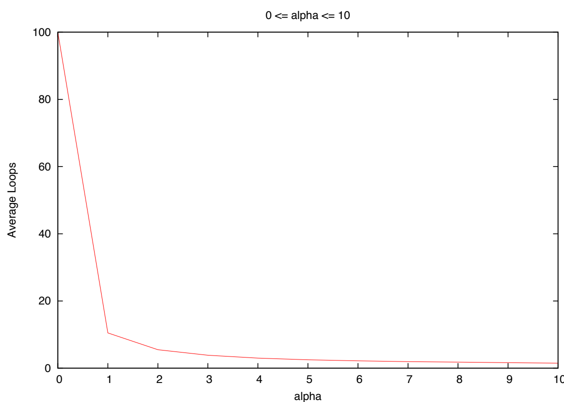


図 2: $0 \leqq alpha \leqq 10$

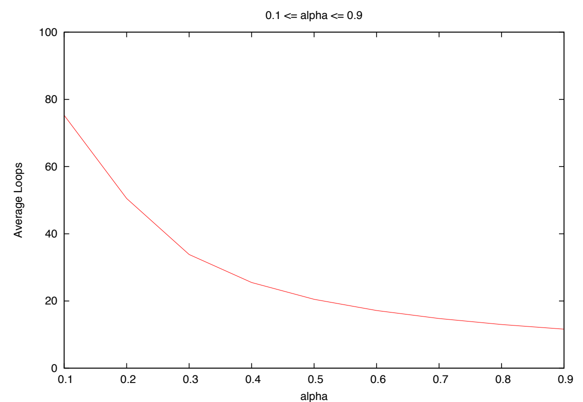


図 3: $0.1 \leqq alpha \leqq 0.9$

図 2 からわかること

A. *alpha* の値が大きくなるほど実行回数は減る。
B. *alpha* の値が 0 のときは終了せずに 100 回実行されている。
C. *alpha* の値が 1 の時と 2 時では 2 倍の差がある。
D. *alpha* の値は n の時 約 $\frac{10}{n}$ の割合で平均が取れていると思われる。
E. つまり反比例の関係であることがわかる。

図 3 からわかること

- 図 2 と同様に alpha の値が大きくなるほど実行回数は減る。

次に、10〜0 まで $alpha$ の値を変えた図 4 と 0.9〜0.1 までの $alpha$ の値を変えた図 5 の結果をそれぞれ示す。ただし、図 4,5 の y 軸は平均 $f(x)$ の最小値を示している。

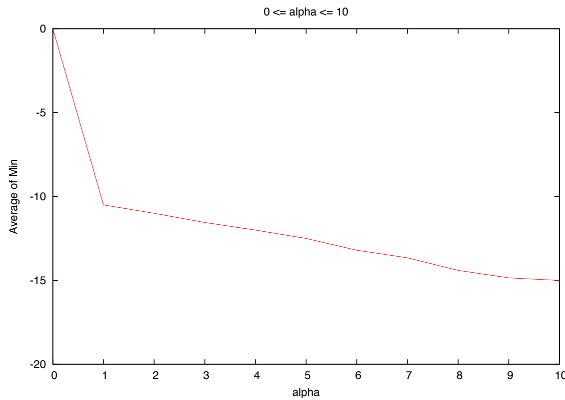


図 4: $0 \leqq alpha \leqq 10$

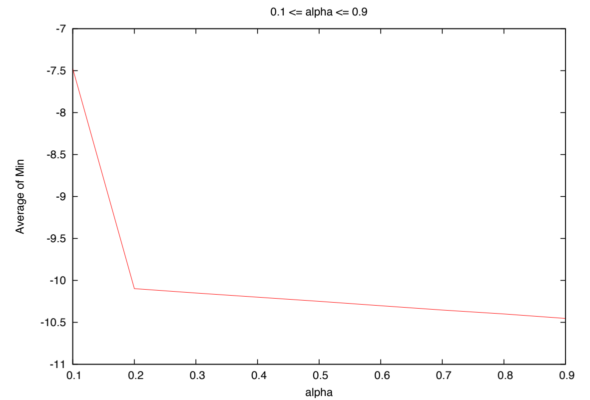


図 5: $0.1 \leqq alpha \leqq 0.9$

図 4 からわかること

A. $alpha$ の値が大きいほど最小値が小さい。
B. しかし、ここでの平均の最小値は-10 に近いほど良いので $alpha$ の値が小さいほど良い。
C. 最小値の平均と-10 の最小値の差は $alpha$ の値を 2 で割った値に近い。

$$|ave_{min} - (-10)| \fallingdotseq \frac{alpha}{2}$$

図5からわかること

A. 図4と同様に *alpha* の値が大きいほど最小値が小さい。
B. しかし、*alpha* の値が0.1の時の最小値が変である。
C. これは *alpha* の値が0.1の時は最小値を求めることができる処理が終了したからである (実行回数が100回を越えた)

次に、計算結果の最大値、最小値、平均値を示す（表1）。

表3:

alpha	Maximum	Minimum	Average
0.1	0.099870	-10.099990	-7.474425
0.2	-9.800130	-10.199961	-10.097954
0.3	-10.000001	-10.299990	-10.150491
0.4	-10.000001	-10.399902	-10.199932
0.5	-10.000003	-10.499988	-10.249997
0.6	-10.000001	-10.599960	-10.301964
0.7	-10.000001	-10.699984	-10.352979
0.8	-10.000001	-10.799902	-10.399936
0.9	-10.000003	-10.899985	-10.453485

E. 表より、*alpha* の値が0.1の場合では最小値が-10より大きい。
F. これは最小値が求めることができずに終了したことがわかる。
G. 平均値が最も-10に近いのはalphaが0.2の時だが、0.2は処理が完全に終了できなかった。
H. つまりalphaの値が0.3の時が最も最小値を求めることに適しているのではないかと思われる。

まとめ

A. *alpha* の値が大きいほど実行回数が少ない。
B. *alpha* の値が小さいほど最小値が正確である。ただし小さ過ぎてはダメ。
C. 実行回数を取るか、正確性を取るかで *alpha* の値が適しているのが変わってくる。

考察に用いたソース

- steepest_decent1_1.c

1. steepest_decent1_1.c は実行回数の平均を求めるプログラムである。
2. steepest_decent.c のソースの main 関数を fx 関数と置き換えた
3. 新しく main 関数を作り引数を変更することで alpha の値を変更できるようにした
4. さらに seed の値も変更できるようにした
5. main 関数では seed 値 0〜99999 の実行回数をすべて足し、平均する。

- steepest_decent1_3.c

1. steepest_decent1_1.c の main 関数を変更したプログラムである。
2. if 文を使い、最大値、最小値を求めている。
3. それぞれの seed の値の最小値の値を加算し、平均を求めている。

- 上記の 2 つのソースは以下に示す（一部省略)。

steepest1_1.c (サンプルソースからの変更点を主に記載)

```
//seed 値と alpha 値の引数をもらい実行する
int fx(int seed,double alpha) {
  double x;
  int    i;
  /** alpha: 学習レート
   *  （課題）正の範囲内で任意に設定し，それに伴う探索点の移動を観察せよ.
   *  （ポイント）alpha を固定にした場合と，静的または動的に決定した場合
   *          とで学習効率に差はでるか？
   */
  int term_cond = 100; /* 終了条件（繰り返し数） */

  srand(seed);
  rand();



  /* step1. 探索の初期位置を設定 */
  x = X_MIN + X_RANGE * (double)rand()/RAND_MAX;

  for (i = 1; i < term_cond; i++) {
    /* step2. 次の探索場所へ移動 */

----省略----

    }
  }
  return i;  //trial の値が戻り値
}
```

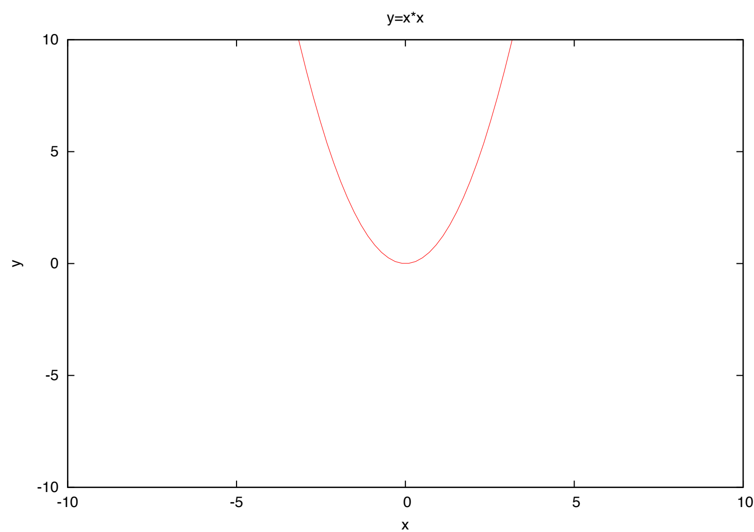
```
int main(){
  float i;
  int num;
  double j;

  //j は alpha の値である。
  for(j=0;j<1;j+=0.1){
    num = 0; //trial 数の合計が入る
    printf("alpha:%f----------------------------\n",j);

    //seed を 0~99999 の計 100000 パターンの trial の値を足している
    for(i=0;i<100000;i++){
      num +=fx(i,j);
    }

    //平均を計算し、出力している
    printf("trial average:%f\n",num/i);
  }
  return 0;
}
```

steepest1_3.c（steepest1_1.c との相違点を主に記載）

```
----省略----

double df(double a) {

----省略----

}



double fx(int seed,double alpha) {

----省略----

}

int main(){
  double i,j,n_min,n_max,n_ave,n_new;
```

```
  for(j=0;j<10;j+=1){
    n_max = -100; //最大値が入るので小さい値を入れている
    n_min = 100;  //最小値が入るので大きい値を入れている
    n_ave = 0; //平均の初期値は 0 である
    printf("alpha:%f---------------------------\n",j);
    for(i=0;i<100000;i++){
      n_new = fx(i,j);  //n_new には実行回数が入る
      //n_new の方が大きいなら、n_max を n_new の値にする
      if(n_max < n_new) n_max = n_new;
      //n_new の方が小さいなら、n_max を n_new の値にする
      if(n_min > n_new) n_min = n_new;
      n_ave += n_new; //n_ave には実行回数が加算されていく
    }
    //最大値、最小値、平均を出力する
    printf("max:%lf\tmin:%lf\tave:%lf\n\n",n_max,n_min,n_ave/i);
  }



  return 0;
}
```

2.3 Level2.2: $y = x^2$



2.3.1 プログラムソース（変更部分）

```
----中略----

/* f(x)/dx
 *     y=f(x) の微分値を求め，返す.
 */
double df(double a) {
  double y_dx;

  /** 作成せよ (2) **/
  y_dx= 2*a;

  return( y_dx );
}



double fx(int seed,double alpha) {
  double x;
  int    i;

----中略----
/* 終了条件 2 */
/* 最小値が 0 なので、x=0 のときも終了する */
    if( (x <= X_MIN) || (x >= X_MAX) || x==0){
      //       printf("seed %d\t:trial %d x rearched to %f f(x) %f\n",seed,i,x,f(x));
      break;
    }
```

2.3.2 実行結果 (steepest_decent2_1.c)

```
% ./a.out
alpha : 0.000000-----------------
average:100.000000

alpha : 0.100000-----------------
average:100.000000

alpha : 0.200000-----------------
average:100.000000

----中略----

alpha : 0.900000-----------------
average:100.000000

alpha : 1.000000-----------------
average:100.000000
```

2.3.3　考察

上記のソースは $y = x^2$ となったため変更した部分を記載した。
その中で注目すべき行を以下に記述する。

L.26　`y = x*x;  //y = x^2` である

L.37　`y_dx= 2*a;  //y = x^2` を微分した結果である

L.72　`if(x == x-alpha*df(x))  break;`

- 26 行目と 37 行目は式の情報である。

- 72 行目は現在の x の値と移動先の x の値が同じなら終了する。

- これは収束してるなら終了するという意味である。

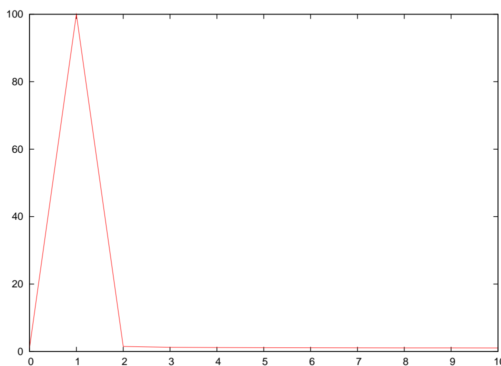


図 6: $0 \leqq alpha \leqq 10$

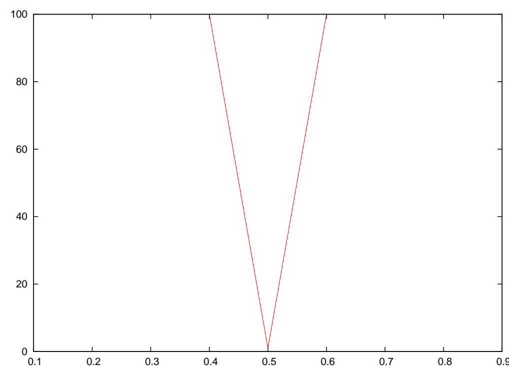


図 7: $0.1 \leqq alpha \leqq 0.9$

1. steepest_decent2.c は静的に $alpha$ の値を決めた。
2. 図 6 からわかること
   A. $alpha$ の値が 2〜10 にかけて平均的に少ない。
   B. なぜならば、alpha の値が大きいと $-10 < x < 10$ の範囲を越えて処理が終了してしまう。
   C. $alpha$ の値を 2 にして実行してみる。

実行結果

```
./steepest_decent2 300
trial 0 x -0.773271 f(x) 0.597948
trial 1 x 2.319813 f(x) 5.381534
trial 2 x -6.959440 f(x) 48.433809
trial 3 x_rearched_to 20.878321 f(x) 435.904280
```

   D. 以上の結果のように $alpha$ の値が大きいと収束せずに発散してしまう。

3. 図 7 からわかること
   A. 結果からわかるように $alpha$ の値が 0.5 の時実行回数が 1 回だけとなっている。

実行結果

```
./steepest_decent2 300
    trial 0 x -0.773271 f(x) 0.597948
    trial 1 x_rearched_to 0.000000 f(x) 0.000000
```

B. 実行結果より 1 回だけで最小値を求めることが出来ている。

C. *alpha* の値を計算で求める。

$$\begin{aligned} y &= x - a \times dx \\ dx &= 2x \text{ かつ} \\ y &= 0 \text{ より} \\ 0 &= x - 2ax \\ a &= \frac{1}{2} \end{aligned}$$

D. 以上の結果より *alpha* の値は 0.5 ならば x の値と関係なく y=0 となる。

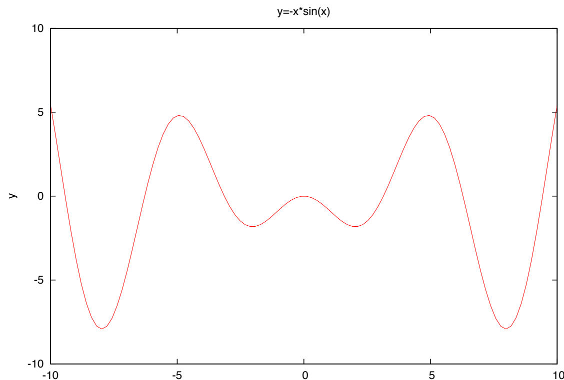
E. したがって、*alpha* の値が 0.5 の時が最も良い。

steepest_decent2_1.c について

1. steepest_decent2.c を変更した
2. steepest_decent1_1.c の時と同様に main 関数を fx 関数に置き換えている。

```
----中略----
/* 終了条件 2 */
/* 最小値が 0 なので、x=0 のときも終了する */
    if( (x <= X_MIN) || (x >= X_MAX) || x==0){
      //      printf("seed %d\t:trial %d x rearched to %f f(x) %f\n",seed,i,x,f(x));

----中略----


int main(){
  int i;
  double j;
  for(j=0;j<=1;j+=0.1){
    double ave=0;
    printf("alpha : %f-----------------\n",j);
    for(i=0;i<10;i++) ave +=fx(i,j);
    printf("average:%lf\n\n",ave/i);
  }
  return 0;
}
```

2.4 Level2.3: $y = -xsinx$



2.4.1 プログラムソース（変更部分）

```
1.   以下は y=-x*sin(x) になったために変更した点である
14.    //trial の値を main に渡す変数 (最小値を取ったとき)
15.    int trial_c=0;

29.    y = -x*sin(x); //y=-x*sin(x) を求める

39.    y_dx= -sin(a)-a*cos(a); //y=-x*sin(x) を x で微分した結果である

82.    trial_c = i; //main 関数に trial の値を渡す
83.    return f(x); //戻り値を f(x) とする

89.   int main(){
90.    int i;
91.    double j=0,k;
92.    double ave,atai,ave_c=100,ave_alpha=0,ave_r=0;
93.    double ratio=0, r_alpha=0;
94.
95.    //alpha の値を k として増やしていく
96.    for(k=0;k<1;k+=0.01){
97.
98.      //trial の平均を求める変数
99.      double trial_ave=0;
100.     printf("percentage:%lf------------------------\n",k);
101.     for(i=1;i<=10000;i++){
102.
103.       //seek の値と alpha の値を渡し、f(x) の値を受け取る
104.       atai = fx(i,k);
105.
106.       //y=-x*sin(x) の最小値は-7.916 より小さく、-7.917 より大きくなるので条件文を加えた
```

```
107.        if( (atai<-7.916) && (atai>-7.917) ){
108.      //正しい最小値が出た回数
109.       j++;
110.      //trial を足していく
111.      trial_ave += trial_c;
112.        }
113.      }
114.
115.      //実行回数の平均を求めている
116.      ave = trial_ave/j;
117.      printf("trial_ave:%lf\t",ave);
118.
119.      //正しい最小値を取ったときの割合を求めている
120.      j = j / (i-1);
121.      printf("average:%lf \n\n" , j);
122.
123.      //割合が最も高い場合を求め、格納している
124.      if( j > ratio ){
125.        ratio = j;
126.        r_alpha = k;
127.      }
128.
129.      //実行回数が少ない値を求めて、格納している
130.      if((ave < ave_c) &&(j > 0.3)){
131.        ave_alpha = k;
132.        ave_c = ave;
133.        ave_r=j;
134.      }
135.    }
136.
137.      printf("最小値を取る割合が最も高かったのは
    alpha=%lf の時、%lf である。\n",r_alpha,ratio);
138.      printf("trial の値の平均が最小なのは
    alpha=%lf の時、平均%lf 回で、%lf ある。\n",ave_alpha,ave_c,ave_r);
139.
140.    return 0;
141.
142.  }
```

2.4.2 実行結果

```
% ./a.out
alpha:0.000000―――――――――
trial_ave:1.000000 average:0.002700

alpha:0.100000―――――――――
trial_ave:23.612605 average:0.508800

―省略―
最小値を取る割合が最も高かったのは alpha=0.100000 の時、0.508800 である。
trial の値の平均が最小なのは alpha=0.1000000 の時、平均 23.612605 回で、0.508800 ある。
```

2.4.3 考察

- 15 行目はグローバル変数を置き、関数間の値の引渡しに使われる。
- 29,39 行目は式の情報である。
- 82 行目は fx 関数の中にあり、main 関数に trial の数を渡すためにある。
- 81 行目も fx 関数の中にあり、最小値を戻り値にしている。
- 87〜131 行は main 関数である。重要な部分だけ説明する。

A.104 行目で fx 関数から最小値をもらい、atai に代入している。
B.107 行目では最小値の値が正しいかを判断している。
C. $-10 < x < 10$ の $y = -x \times sin(x)$ において、-7.916 未満かつ-7.917 より大きいならば正しい最小値の値である。
D.109 行目では正しい最小値が出てきた回数をカウントしている。
E.110 行目ではグローバル変数でもらってきた trial_c(trial の数) を足し、合計を求めている。
F.116 行目では trial_ave(trial の数の合計) から j(正しい最小値が出た回数) を割り、trial の数の平均を求めている。
G.120 行目では正しい最小値を取ったときの割合を求めている。
H.124〜127 行目は割合が最も高い値を求めている。
I.130〜135 行目では ave の比較を行い、trial の数が最も少なく、割合が 0.3 よりも高い時の場合を求めている。

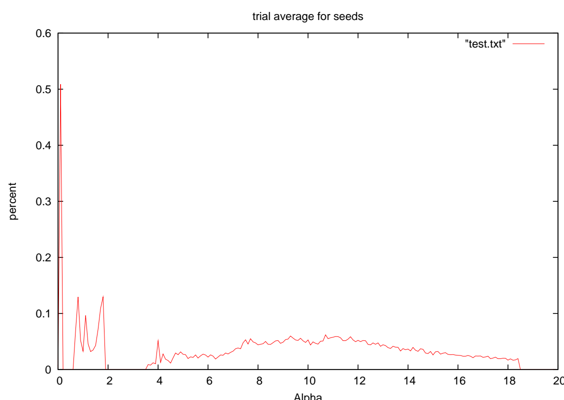


図 8: $0 \leqq alpha \leqq 20$

実行結果 ($0 \leqq alpha \leqq 20$)

A.*alpha* の値が 0.1 で最も高い割合であることがわかる。
B.*alpha* の値が 0.3〜0.6、2.1〜3.5 の時は一度も最小値を取ることがなかった。
C.*alpha* の値が小さい時でも最小値を取ることができない場合がある。
D.*alpha* の値が約 4〜18 の時は割合は低いが最小値を取る場合がある。

- 以上の結果より 0〜0.2 の間で最も良い *alpha* の値が存在すると考えられる。
- 0.1〜0.2 の間で細かく調べてみた。

```
% ./a.out
alpha:0.010000————————
trial_ave:nan average:0.000000

alpha:0.020000————————
trial_ave:nan average:0.000000

alpha:0.030000————————
trial_ave:97.333333 average:0.000300

alpha:0.040000————————
trial_ave:88.906661 average:0.480000

alpha:0.050000————————
trial_ave:68.420236 average:0.508848

—省略—
alpha:0.170000————————
trial_ave:38.009170 average:0.508851

alpha:0.180000————————
trial_ave:46.485131 average:0.508851

alpha:0.190000————————
trial_ave:0.000000 average:0.000051

最小値を取る割合が最も高かったのは alpha=0.080000 の時、0.508851 である。
trial の値の平均が最小なのは alpha=0.120000 の時、平均 11.855536 回で、0.508851 ある
```

実行結果 ($0 \leqq alpha \leqq 0.2$)

A. 割合は 0.508851 よりは大きくならないことがわかる。
B.*alpha* が 0.03 の時から徐々に上がっていき、0.047 の時には最大を取り続ける。
C.0.185 以下からはほぼ割合がほぼ 0 となる。
D. よって *alpha* の値が 0.047〜0.184 の時、最大の割合 0.508851 となる。

5. 0.047〜0.184 をさらに細かく調べてみた。

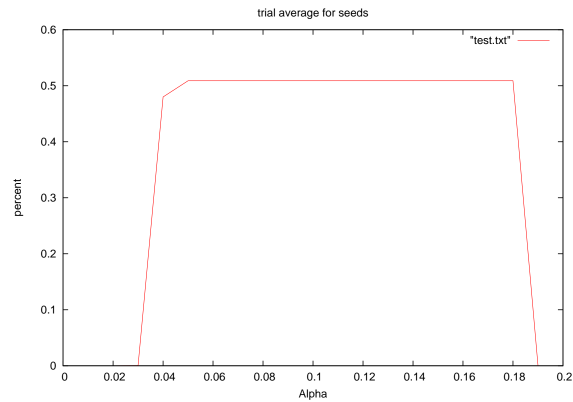


図 9: 0 ≦ $alpha$ ≦ 0.2

```
% ./a.out
alpha:0.122400————————
trial_ave:8.232704 average:0.508800

alpha:0.122500————————
trial_ave:7.829209 average:0.508851

alpha:0.122600————————
trial_ave:8.505635 average:0.508851

最小値を取る割合が最も高かったのは alpha=0.122600 の時、0.508851 である。
trial の値の平均が最小なのは alpha=0.122500 の時、平均 7.829209 回で、0.508851 ある。
```

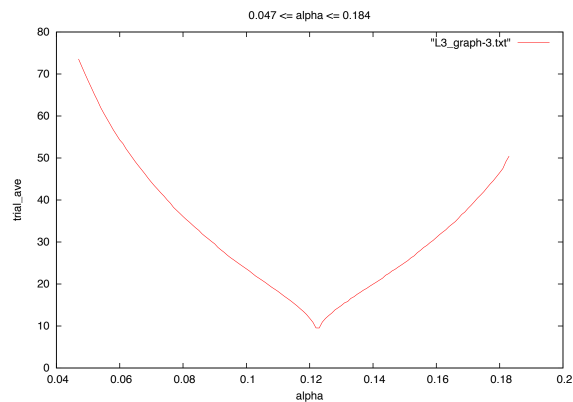


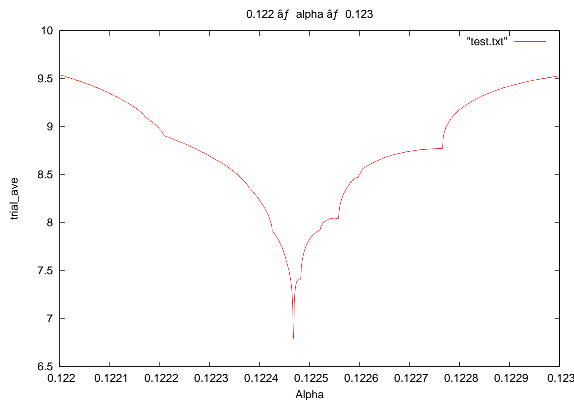
図 10: 0.047 ≦ $alpha$ ≦ 0.184

実行結果 (0.047 ≦ $alpha$ ≦ 0.184)

A. だんだん少なくなっていき、$alpha$ が 0.1225 の時最小を取る。
B.$alpha$ が 0.1225 の値を越えると trial の数も大きくなる。

6. 0.1225 をさらに細かく調べてみた。

```
% ./a.out

alpha:0.122466―――――――――
trial_ave:7.084709 average:0.508800

alpha:0.122467―――――――――
trial_ave:6.795901 average:0.508851

alpha:0.122468―――――――――
trial_ave:6.812605 average:0.508851

―省略―

最小値を取る割合が最も高かったのは alpha=0.122468 の時、0.508851 である。
trial の値の平均が最小なのは alpha=0.122467 の時、平均 6.795901 回で、0.508851 である。
```



実行結果 (0.122 ≦ $alpha$ ≦ 0.123)

A. 図 10 とグラフの形が少し似ている。
B.$alpha$=0.122467 の時、平均 6.795901 回の最小を取る。
C.$alpha$ の値が最小値に近くににつれて傾きが急になる。

8. 以上のことから、さらに細かく調べることで trial の数が小さくなるのではないかと考えられる。
9. 1 回で最小値を取るような $alpha$ の値が存在するのではないかと考えた。
10. しかし、この実験では小数第 6 位までしか求めることができないのであくまで仮説である。

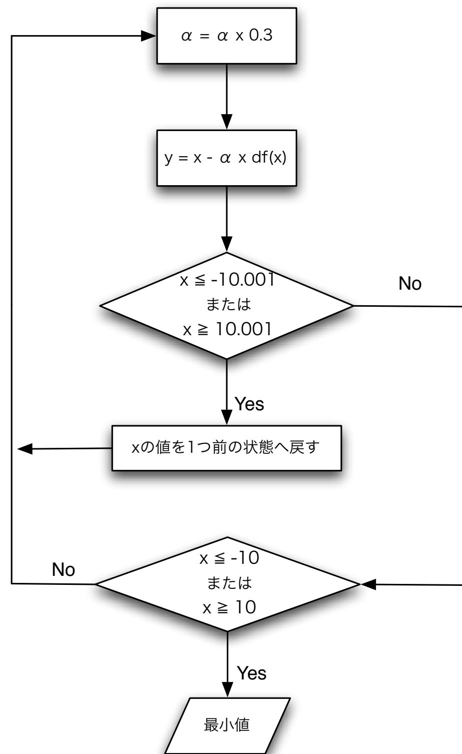
2.5 Level2.x：オプション課題

steepest_decent1.c

- 概要
  $alpha$ の値を動的させ、より効率よくかつ正確に最小値を求める。
  今回考案した探索方法としては、サンプルソースでは一定の割合の $alpha$ 値で探索を行うと、

もし最小値が現地点より遠い位置にあった場合にコストがかかる。
そこで、$alpha$ の初めの値を大きめに設定し、最小値の位置を超えた場合は 1 つ前の地点へ戻り、
今度は $alpha$ 値を一段階小さくした上で再び移動を開始する。
終了地点はきわめて最小値に近い値（定義域の限界値）を検出したいので、
$-10.000 \geqq x \geqq -10.001$ または $10.000 \leqq x \leqq 10.001$ にあるときの値を出力するようにした。

2.5.1 フローチャート



上記のフローチャートでは上記の探索方法を実装するときの流れを示した。
なお、探索範囲を収束する *alpha* はフローチャートにおいて 0.3 と示しているが、この理由は下記にて説明する。

2.5.2　プログラムソース（変更部分のみ）

```
48    double alpha = abs(X_RANGE); //適当に決めている

70      //x の値が範囲より大きく離れていた場合は移動せずに、alpha の値を変更する
71      if( (x <= X_MIN-0.0000001) || (x >= X_MAX+0.0000001) ){
72        x = x+alpha*df(x);
73        alpha = alpha * 0.3; //alpha の値を 0.3 倍する
74      }else{
75
76        /* 終了条件 2 */
77        if( (x <= X_MIN) || (x >= X_MAX)){
78          printf("seed %d\t:trial %d x_rearched_to %f f(x) %f\n",seed,i,x,f(x));
78          break;
79        }
89      }
```

- 48 行目は alpha の初期値を代入する。
- 71 行目では x の値が指定の範囲より離れてないかの判断をしている。
- 72 行目では x の値をもとに戻している。
- 73 行目では alpha の値を減らし、x 移動する範囲を減らしている。
- 71 行目と 77 行目で最小値の範囲を小さくしている。
- 処理が終わる範囲は「$-10.0000001 < x < -10, 10 < x < 10.0000001$」である。

2.5.3　実行結果

```
seed 0  :trial 29 x_rearched_to -10.000000 f(x) -10.000000
seed 1  :trial 34 x_rearched_to -10.000000 f(x) -10.000000

—省略—

seed 98 :trial 41 x_rearched_to -10.000000 f(x) -10.000000
seed 99 :trial 34 x_rearched_to -10.000000 f(x) -10.000000
```

- 実行結果より最小値が-10 になるのがわかる。
- trial の数は 20〜40 で安定している。

steepest_decent1_6.c

```
83    return i;
84  }

86  int main(){
87    double i,j,k,n_min,n_max,n_ave,n_new;
88    float alpha_m=0,ratio_m=0,ave_m=100;
89    //printf("[alpha,ratio]----------------------------\n");
90
91     //変数 j は alpha の初期値である
92     for(j=0.1;j<10;j+=0.1){
93       double a_ratio = 0;
94       double a_ave=100;
95
96      //変数 k は alpha を減らす割合である
97      for(k=0.1;k<0.9;k+=0.1){
98        n_min = 100;
99        n_max = 0;
100       n_ave = 0;
101
102       //変数 i は seed 値である。
103       for(i=0;i<100000;i++){
104     n_new = fx(i,j,k);
105     if(n_min > n_new) n_min = n_new;//戻り値 (trial の数) の最大値を求めている
106     if(n_max < n_new) n_max = n_new;//最小値を求めている
107     n_ave += n_new; //平均を求めるために trail の数を合計している
108       }
109        //printf("%lf\t%lf\n",j,k);
110       //printf("min:%lf\tmax:%lf\tave:%lf\n\n",n_min,n_max,n_ave/i);
111       //printf("%lf\t%lf\n",j,n_ave/i);
112
113
114       //alpha ごとの trial の数が最も少ない時の ratio の値を調べている
115       if(a_ave >= (n_ave/i)){
116     a_ratio = k;
117     a_ave = n_ave/i;
118       }
119       //trial の数が最も少ない組み合わせを調べている
120       if(ave_m >= (n_ave/i)){
121     alpha_m = j;
122     ratio_m = k;
123     ave_m = n_ave/i;
124       }
125     }
```

```
126       printf("(alpha): %lf (ratio): %lf (average): %lf\n",j,a_ratio,a_ave);
127     }
128       printf("最も平均が低い組合せは [%lf,%lf] で、平均は%lf である\n",alpha_m,ratio_m,ave_m);
129
130
131     return 0;
132   }
```

2.5.4 ソースの説明 (重要箇所のみ)

A. 103〜108 行目の for 文では seed の値が 0〜99999 の最大の trial の数と最小の trial の数を求めている。
B. 115〜118 行目と 120〜124 行目は trail の平均数が少ない場合を求めている。
C. 115〜118 行目は各 alpha の値での最小値の時の ratio の値を求めている。
D. 98〜100 行目で alpha が切り替わる度に初期化されている。
E. 120〜124 行目では全体での最小の値と組み合わせを求めている。

2.5.5 実行結果 (alpha、ratio、trail_average の順に表示させている)

```
0.100000    0.300000   85.191290
0.200000    0.300000   71.232280

—省略—

5.700000    0.300000   31.673440
5.800000    0.300000   31.667620
5.900000    0.300000   31.660360
6.000000    0.300000   31.661910
6.100000    0.300000   31.676180
6.200000    0.300000   31.666170
6.300000    0.300000   31.671440
6.400000    0.300000   31.682760
6.500000    0.300000   31.679050

—省略—

9.700000    0.300000   31.912130
9.800000    0.300000   31.933270
9.900000    0.300000   31.963490
10.000000   0.300000   32.008450
最も平均が低い組合せは [5.900000,0.300000] で、平均は 31.660360 である
```

実行結果よりわかること

A. 最も平均が低い組み合わせは alpha が 5.9、raito が 0.3 の時である
B. 各 alpha は ratio が 0.3 の時に最小を取ることがわかる
C. ratio を 0.3 に固定した図を以下に作成した
D. 図より alpha の値が 0 に近い時は trail の数が大きい
F. 5.9 の時に最小となり徐々に上昇する
G. alpha の値が 5.7〜6.5 の時に小さい値が集中している

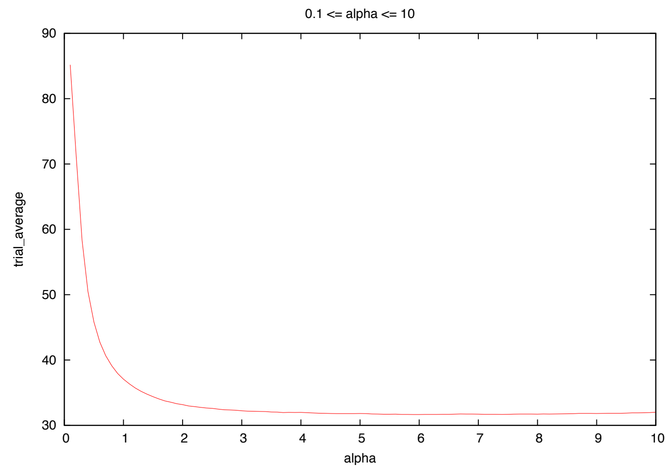


図 11: $0.1 \leqq alpha \leqq 10$

$alpha$ の値を 5.7〜6.5 の間で細かく調べてみた。

2.5.6 実行結果

```
5.700000   0.300000   31.673440
5.710000   0.300000   31.667000

—省略—

6.340000   0.300000   31.667990
6.350000   0.300000   31.615670
6.360000   0.300000   31.674400

—省略—
6.490000   0.300000   31.699710
6.500000   0.300000   31.679050
最も平均が低い組合せは [6.350000,0.300000] で、平均は 31.615669 である
```

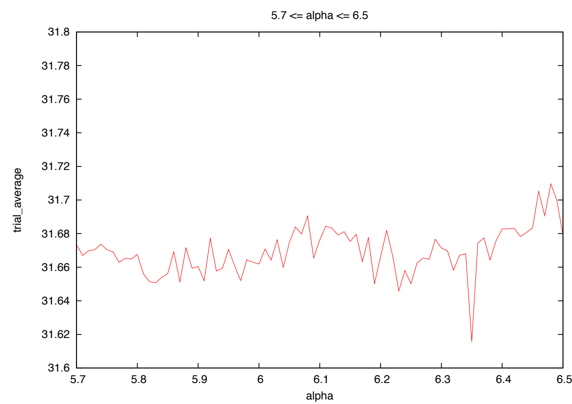


図 12: $5.7 \leqq alpha \leqq 6.5$

- 最も平均が低い組み合わせは $alpha$ が 6.35 の時、平均 31.615669 を取る。
- 図より規則性はあまりみられない。
- 最小値の値がとてもわかりやすい。
- $alpha$ の規則性は見られなかったが最適値のおおよそ値は求めることができた。

3 Level3: 不連続関数における探索手法を検討し、その効率を示せ

3.1 課題説明

Level3.1（ナップサック問題）、Level3.2（巡回セールスマン問題）について、問題空間サイズの特徴について述べ、各々独立した探索手法を検討し、考察する。

3.2 Level3.2: ナップサック問題

3.2.1 問題空間の特徴について

品物の数が 3 個の時の問題空間サイズは 2^3、4 個の時は 2^4、N 個の時は 2^N となる。

3.2.2 探索の手続き、フローチャート

（例）最大容量が $5kg$ の場合を考える。

1. 品物の単位量当たりの金額を計算し、金額が高い順にソートする。

グローブ：	5000yen/0.8kg
かぼちゃ：	2000yen/5.0kg
えんぴつ：	500yen/0.1kg
かなづち：	1000yen/3.0kg
ゴルフグラブ：	10000yen/3.0kg
サッカーボール：	2000yen/0.8kg
南京錠：	3000yen/1.0kg

グローブ：	6250yen/1.0kg
カボチャ：	400yen/1.0kg
えんぴつ：	5000yen/1.0kg
かなづち：	333yen/1.0kg
ゴルフグラブ：	3333yen/1.0kg
サッカーボール：	2500yen/1.0kg
南京錠：	3000yen/1.0kg

グローブ：	6250yen/1.0kg
えんぴつ：	5000yen/1.0kg
ゴルフグラブ：	3333yen/1.0kg
南京錠：	3000yen/1.0kg
サッカーボール：	2500yen/1.0kg
カボチャ：	400yen/1.0kg
かなづち：	333yen/1.0kg

2. 最大の物から順にアクセスする。検索対象が無ければ処理が完了する。

グローブ：	5000yen/0.8kg
えんぴつ：	500yen/0.1kg
ゴルフグラブ：	10000yen/3.0kg
南京錠：	3000yen/1.0kg
サッカーボール：	2000yen/0.8kg
かぼちゃ：	2000yen/5.0kg
かなづち：	1000yen/3.0kg

3. 金額が高い順にナップザックに入れる。この時対象物が容量を超えるなら入れずにリストから削除する。

4. 対象物を入れる。その後、最大容量から入れた品物の容量を引く。そして、2 に戻る。

　グローブが入る（残り 4.2kg）
→えんぴつが入る（残り 4.1kg）
→ゴルフグラブが入る（残り 1.1kg）
→南京錠が入る（残り 0.1kg）
→サッカーボールが入らずリストから削除
→かぼちゃが入らずリストから削除
→かなづちが入らずリストから削除
→終了。

ナップザックに入っている品物の合計金額は 18500yen となった。

3.2.3 提案手法の利点・欠点

- 利点
  全部の品物にアクセスするので、確実に最良のものを選ぶことができる。
- 欠点
  品物の数が膨大になった場合、アクセスする回数も膨大になってしまうので、遅くなってしまう。
- 改善点
  例えば、最大容量が 100kg であり、品物が 99 個、それぞれ 1kg〜99kg だとする。
  ソートした後に最大のもの、つまり 99kg のものが入るとすると 98kg〜2kg のものを調べることが無駄になる。
  この無駄を無くすことが改善に繋がると我々は考える。

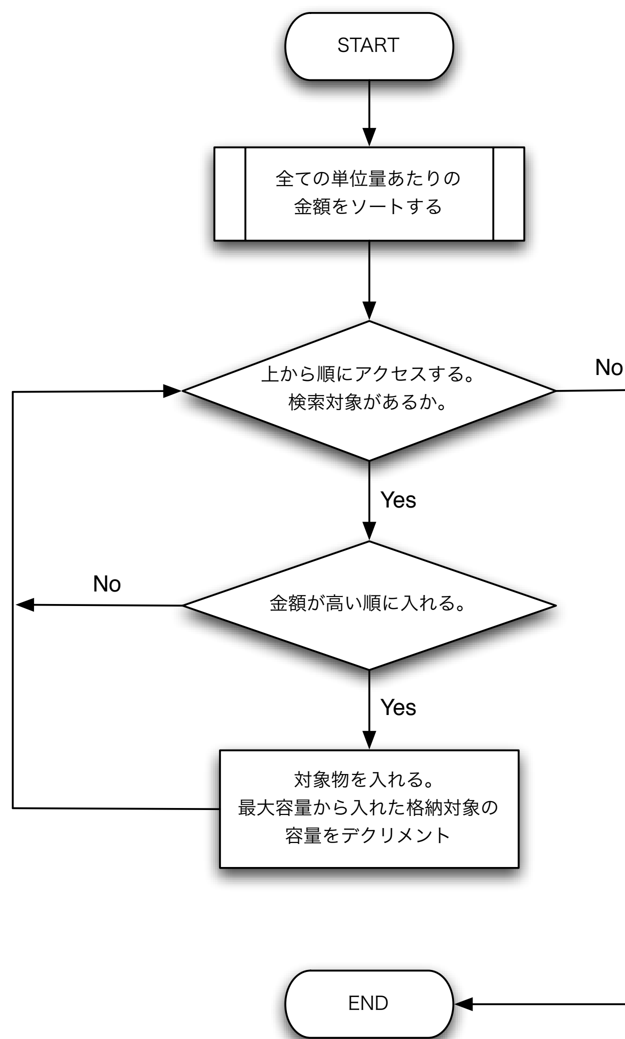


図 13: フローチャート

3.3 Level3.3: 巡回セールスマン問題

3.3.1 問題空間の特徴について

- 問題空間サイズについて
  都市数が3の時、解 x は 123(都市1→都市2→都市3)、213(都市2→都市1→都市3) と表す事ができる。この時の全ての都市を辿るのは『2! 通り』と表せる。都市数が4の時の全ての都市を辿るのは『3! 通り』と表せる。都市数がN個の時、全ての都市を辿るのは『$(N-1)!$ 通り』と表せる。したがって、問題空間サイズは『$(N-1)!$ 個』と表せる。

- 計算量について（ここで言うアクセスとは、都市に関しての情報を調べることを指す。）
  また、都市数が3の時、都市へのアクセスを2回行うので、そのときの計算量を『$2 \times 2!$ 個』と表すことができる。都市数が4の時、都市へのアクセスを3回行うので、そのときの計算量を『$3 \times 3!$ 個』と表すことができる。都市数がNの時、都市へのアクセスをN回行うので、そのときの計算量は『$(N-1)(N-1)!$ 個』と表せる。

ー①

3.3.2 探索の手続き、フローチャート

☒それぞれ最短距離を求める

我々7班は計算量を要素にどれだけアクセスしたかとする。ここで言うアクセス回数とは、座標（都市）を比べた回数を指し示している。

1. 原点を決めて、そこから全ての座標（都市）にアクセスし、距離を求める。
2. 求めた距離が最短のものを選択し、移動する。移動した座標から再び他の都市にアクセスし、距離を求める。このとき一度移動した座標は選択しないようにする。

（例）都市が5カ所あった場合

1. 原点を決め、そこから距離を求める際、原点以外の4都市にアクセスするので、アクセスを4回行う。
2. 最短の都市に移動した後、そこからまだ辿っていない都市にアクセスするので、アクセスを3回行う。
3. さらに移動した後、まだ辿っていない都市は2カ所ある。その2カ所にさらにアクセスするので、アクセスを2回行う。
4. 最後に、辿っていない都市が1カ所あるので、その都市に移動する。つまりアクセスを1回行う。
5. ここまでアクセスを行った回数は『$4+3+2+1=10$』回となる。

よって、この方法を用いたときの計算量は、『$\frac{n(n-1)}{2}$ 個』となる。

ー②

☒一番角度が小さいものを選ぶ

この案は、原点をランダムに選択したものではないが、もし原点をこちらで選択ができるならば、このようなことができると考えたものである。

1. y 座標が一番小さいものを選び、原点とする。
2. 選んだ点を中心とし、$0^\circ \leq \theta \leq 360^\circ$ の範囲の中で一番角度が最小の座標を選び、移動する。
3. 移動した点で2を行う。当然、一度移動した座標は選ばない。
4. 選べる座標が無くなったら、原点に戻る。

（例）都市が５カ所あったとする。

1. y 座標が一番小さいものを選ぶために、全ての都市にアクセスしなければならないので、５回アクセスする。
2. 選んだ点から一番角度が最小の座標を選ぶので、他の点それぞれにアクセスするので、４回アクセスする。
3. 最小の都市に移動した後、さらにその都市から角度が最小のものを選ぶので、3 回アクセスする。
4. 移動後、残った 2 つの都市の角度を計算し、最小のものを選ぶので、2 回アクセスする。
5. 移動後、１つの都市しか残っていないので、アクセスは 1 回となる。
6. 移動できる都市がないので、原点に戻る。
7. ここまでアクセスを行った回数は『5+4+3+2+1 = 15』回となる。

この方法を用いたときの計算量は、『$\frac{n(n+1)}{2}$個』となる。

ー③

3.3.3 提案手法の利点・欠点

- ①計算量 $= (N-1)(N-1)!$ の利点
  全パターンを調べるので、確実に最小の経路を探索することができる。
- ①計算量 $= (N-1)(N-1)!$ の欠点
  全パターンを調べるので、探索時間が長くなってしまう。

- ②計算量 $= \frac{n(n-1)}{2}$ の利点
  全パターンを調べるより速く計算量を計算できる。
- ②計算量 $= \frac{n(n-1)}{2}$ の欠点
  探索した結果が必ずしも最小であるとは限らない。

- ③計算量 $= \frac{n(n+1)}{2}$ の利点
  周りを囲むように経路を辿ることができる。
- ③計算量 $= \frac{n(n+1)}{2}$ の欠点
  これも、探索した結果が必ずしも最小であるとは限らない。

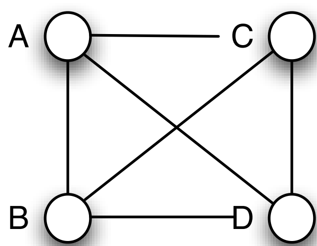


図 14: デフォルト

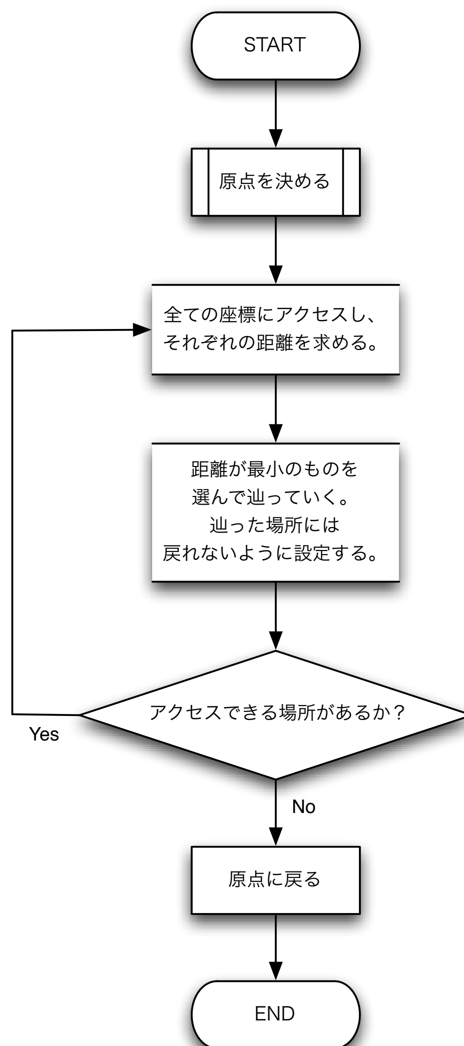


図 15: フローチャート

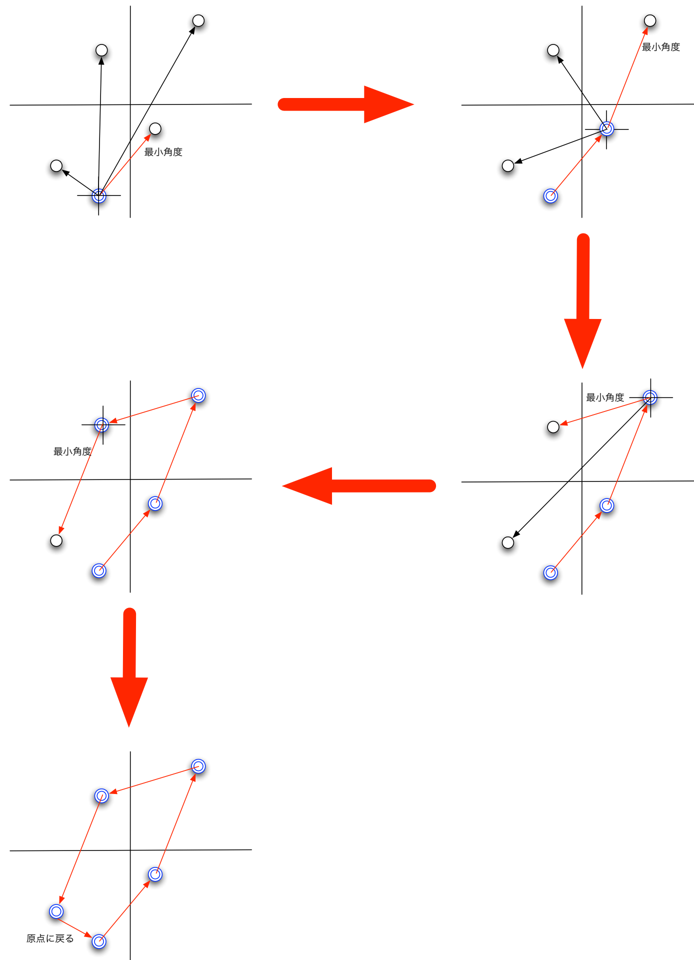


図 16: 一番角度が小さいものを選ぶ

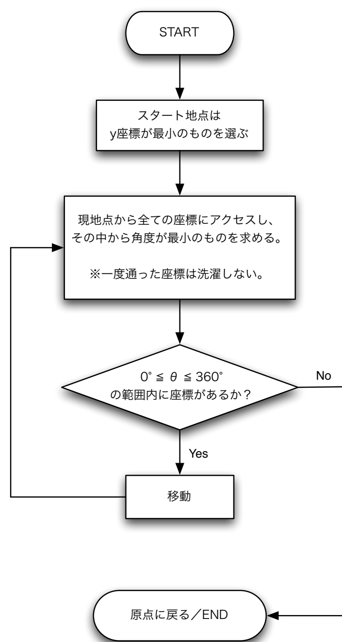


図 17: フローチャート（一番角度が小さいものを選ぶ）

参考文献

[1] 情報工学実験 2: 探索アルゴリズムその 1（當間）
http://www.eva.ie.u-ryukyu.ac.jp/~tnal/2010/info2/search1/