情報工学実験

-探索アルゴリズム 1-

グループ：A

メンバー：j05002　池野谷克俊 (Level 2, Level 3 担当)

j05019　鴻池宏輝 (Level 2 担当)

j05066　與久田龍一 (Level 1 担当)

提出日:2006 年 11 月 20 日 月曜日

1 Level 1

1.1 Level 1.1

コンピュータが人間より得意とするものとしては, 計算処理能力の高さ, 記憶能力, 単純作業を繰り返しできるなどということが挙げられる。コンピュータは人間と違い飽きるということが無いので, 上記のことが得意と言える。

逆にコンピュータが人間より苦手とするものとしては, あやふやな物事 (定性) に対する判断, 予想外の結果の対応などが挙げられる。コンピュータはあらかじめ定義されたものに対して処理を行っているので, 定義することが極めて難しいあやふやな物事や, 定義されてない結果に関しての対応が苦手と言える。

1.2 Level 1.2

- クルマを目的地にまでたどり着くために (カーナビゲーションシステム)

  自分がまだいった事の無い土地に行く時カーナビゲーションシステムは地図以上に役に立つ。なぜなら、このシステムは初めに目的地さえ入力してしまえば目的地までの最短の経路をしめしてくれるからだ。ここでは最短経路探索 (Dijkstra 法) について書いて行く。この最短経路問題は重みつきグラフ G=(V,E) が与えられた時に、任意の 2 頂点を結ぶ経路の中から、辺の重みの総和が最小のものを求めるものである。
  Dijkstra 法のアルゴリズムは出発点 A から V (頂点集合) の中の各頂点への最短経路のコストを全て求める。

- 最短経路探索 (Dijkstra 法)

  - 1 節点から全節点への最短経路を求めるアルゴリズム
  - 1959 年に E.Dijkstra によって提案された
  - 全ての枝の重みが非負の場合にのみ適用できる
  - 手順：
    １．始点のラベルを 0、それ以外の点のラベルを無限大とする
    ２．最短路の長さが確定した点のラベルを 確定ラベル、それ以外の点を一時ラベルと置き、次に一時ラベルの中で最小の点 x をみつける
    ３．２で見つけた点を確定ラベルに変更し、隣接する点の一時ラベルを、「一時ラベルの値と、点 x のラベルの値とその間の枝の重みを足したものの小さい ほう」の値に変更し、確定ラベルにす

る
４．１，２，３を繰り返し、 すべての点が確定ラベルになったら終了．その結果，各点の確定ラベルが始点からの最短路の長さになっている。

下図の有向グラフにおいて，[13] から [4] までの最短経路を、Dijkstra 法を用いて求めてみる。

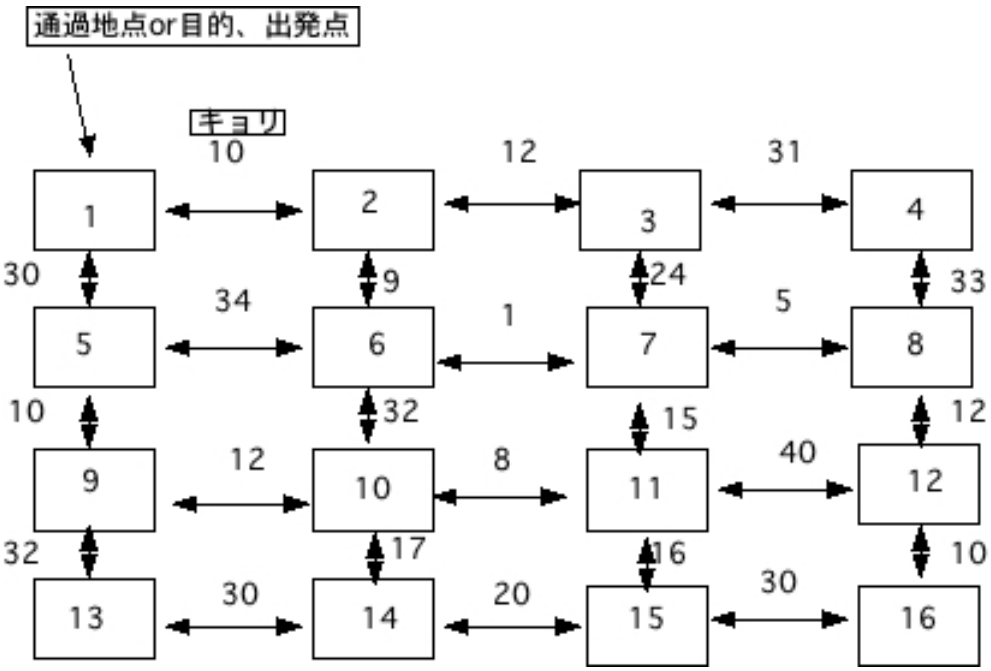

図 1: グラフ

まず、このグラフについてのデータを与える
(始点 ⋯ Ns=13, 終点 ⋯ Ng=4)
存在する区間データを与える

以下の 3 つのリストを用意する.

- リスト A (未調査リスト)
- リスト B (調査中リスト)
- リスト C (調査済リスト)
- 区間データからノードデータ (Ni,Ns からの最短距離) を生成しリスト A に登録、最短距離の初期値は∞
  リスト A:$(1,\infty)(2,\infty)(3,\infty)$ ⋯ $(16,\infty)$

区間データ		
1	2	10
2	1	10
2	3	12
3	2	12
・	・	・
・	・	・
15	16	30

- 始点 Ns に関するノードデータを未調査リストから調査済リストへ移動、その際ノードデータの最短距離を 0 に更新
  リスト A:$(1,\infty)(2,\infty)(3,\infty)\cdots(16,\infty)$
  リスト B
  リスト C(13,0)
- 区間データを元に、始点 Ns から直接到達可能なノードを調べ、そのノードに関するノードデータをリスト A からリスト B に移動しノードデータを更新
  リスト A:$(1,\infty)(2,\infty)(3,\infty)\cdots(16,\infty)$
  リスト B(9,32)(14,30)
  リスト C(13,0)
- 以下の項目をリスト B が空になるまで繰り返す。
  (a) リスト B から最短距離最小のノード Nm を選び C へ移動
  リスト A:$(1,\infty)(2,\infty)(3,\infty)\cdots(16,\infty)$
  リスト B(9,32)
  リスト C(13,0)(14,30)
  (b)Nm から直接到達可能なノード Ni を探し、Ni がリスト A にあればリスト B へ移動
  リスト A:$(1,\infty)(2,\infty)(3,\infty)\cdots(16,\infty)$
  リスト B(9,32)$(10,\infty)(15,\infty)$
  リスト C(13,0)(14,30)
  (c)Ni が B にあれば、Ns から Nm の最短距離に区間距離 Nm,Ni を加えた値と、既知の最短距離 Ns,Ni を比べ、より小さい方を新たな最短距離とする
  リスト A:$(1,\infty)(2,\infty)(3,\infty)\cdots(16,\infty)$
  リスト B(9,32)(10,30+17)(15,30+20)
  リスト C(13,0)(14,30)

- リスト B が空に成った時点で A,B,C について以下のような　リス

トが得られる。
リスト A(4,120)(8,102)(12,90)(3,89)(16,80)(2,77)(1,72)
リスト B(6,68)(7,67)(11,52)(15,50)(10,44)(5,42)(9,32)
リスト C(13,0)(14,30)

ここでリスト C に、始点 Ns から各ノードへの最短距離が入っていることになり、ノード 13 から 4 までの最短距離が 120 であることが判明する
ノードデータに対して一歩手前のノード番号を記憶させておき、終点から順に遡ることで最短距離だけでなく，最短経路も求めることができる

- 考察

上で述べた例題のようにクルマのカーナビゲーションシステムも最短距離及び最短経路を求める探索を行っている。ここで挙げた例は単純だが最近のカーナビゲーションシステムにはこの他にはもっと色々な探索機能がついている。例えば、出発地から目的地まで、交通情報を加味した時間短縮ルートを常時探索する機能や、通常の探索対象外の細街路に入っても、目的地方向のルートをすばやく探索し直しルートを変更し表示する機能等がある。

2 Level 2

検索の手続きについては $Level2.1 \sim Level2.3$ は同じ手続きを行っているので, ここで, まとめて記述する

- 検索の手続き

1. 引数で与えられた数字からランダムに初期位置を設定
2. 現在の位置から $-alpha * df(x)$ だけずれた地点に移動 (傾きがマイナスなら右へ, プラスなら左へ移動)
3. 繰り返し数 (今回は 100 回) を超えるか,x の値が定義された範囲 (今回は $-10 \sim 10$) を超えるまで, 2 の動作を繰り返す

- フローチャート

以下の図は検索の手続きを表したフローチャートである。

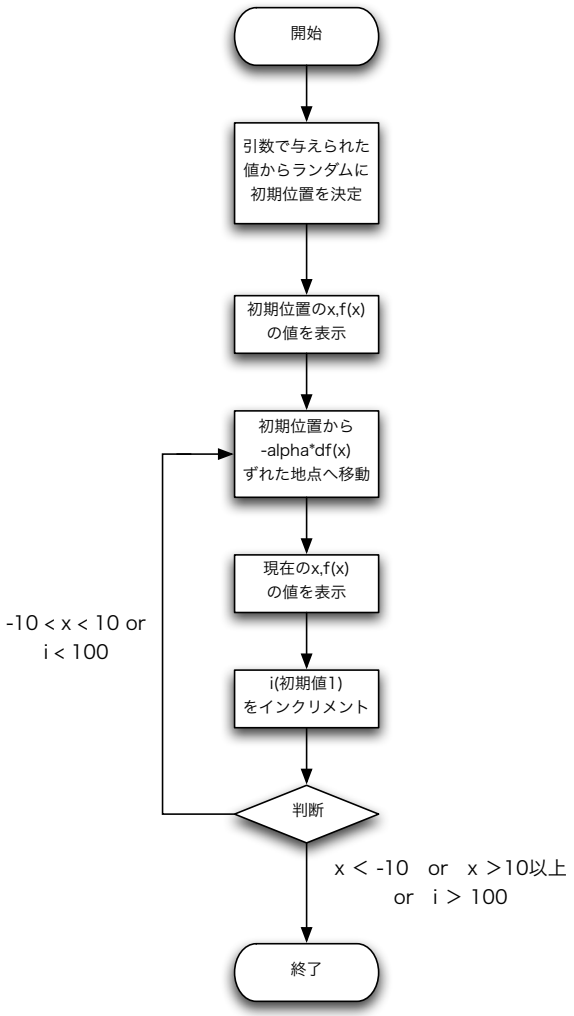

図 2: フローチャート

2.1 Level 2.1

- プログラムソース

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define X_MAX 10.0    /* 定義域の最大値 */
#define X_MIN -10.0   /* 定義域の最小値 */
#define X_RANGE (abs(X_MAX)+abs(X_MIN))
```

```
void usage();
double f(double x);
double df(double x);

省略

/* 関数 f(x)
 *    入力された x に対する y=f(x) の値を求め，返す.
 */
double f(double x) {
  double y;

  /** 作成せよ (1) **/
  y = x;
  return( y );
}

/* 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 = 1;
  return( y_dx );
}


int main(int argc, char **argv) {
  double x,dy;
  int    i;
  double alpha = 0.1;

省略

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

    /** 作成せよ (3) **/
    x = x - alpha*df(x);

省略

}
```

- 実行結果

  alpha が 0.1 の時

```
[j05002@search1-sample]% ./steepest_decent 1
trial 0 x -7.369244 f(x) -7.369244
trial 1 x -7.469244 f(x) -7.469244
trial 2 x -7.569244 f(x) -7.569244
trial 3 x -7.669244 f(x) -7.669244
trial 4 x -7.769244 f(x) -7.769244
trial 5 x -7.869244 f(x) -7.869244
trial 6 x -7.969244 f(x) -7.969244
trial 7 x -8.069244 f(x) -8.069244
trial 8 x -8.169244 f(x) -8.169244

省略

trial 23 x -9.669244 f(x) -9.669244
trial 24 x -9.769244 f(x) -9.769244
trial 25 x -9.869244 f(x) -9.869244
trial 26 x -9.969244 f(x) -9.969244
trial 27 x_rearched_to -10.069244 f(x) -10.069244
```

alpha が 1 の時

```
[j05002@search1-sample]% ./steepest_decent 1
trial 0 x -7.369244 f(x) -7.369244
trial 1 x -8.369244 f(x) -8.369244
trial 2 x -9.369244 f(x) -9.369244
trial 3 x_rearched_to -10.369244 f(x) -10.369244
```

alpha が 0.001 の時

```
[j05002@search1-sample]% ./steepest_decent 1
trial 0 x -7.369244 f(x) -7.369244
trial 1 x -7.370244 f(x) -7.370244
trial 2 x -7.371244 f(x) -7.371244
trial 3 x -7.372244 f(x) -7.372244
trial 4 x -7.373244 f(x) -7.373244
trial 5 x -7.374244 f(x) -7.374244
trial 6 x -7.375244 f(x) -7.375244
trial 7 x -7.376244 f(x) -7.376244
trial 8 x -7.377244 f(x) -7.377244
```

省略

```
trial 95 x -7.464244 f(x) -7.464244
trial 96 x -7.465244 f(x) -7.465244
trial 97 x -7.466244 f(x) -7.466244
trial 98 x -7.467244 f(x) -7.467244
trial 99 x -7.468244 f(x) -7.468244
```

2.2 Level 2.2

- プログラムソース

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define X_MAX 10.0    /* 定義域の最大値 */
#define X_MIN -10.0   /* 定義域の最小値 */
#define X_RANGE (abs(X_MAX)+abs(X_MIN))


void usage();
double f(double x);
double df(double x);
```

省略

```
/* 関数 f(x)
 *    入力された x に対する y=f(x) の値を求め，返す.
 */
double f(double x) {
  double y;

  /** 作成せよ (1) **/
  y = x*x;
  return( y );
}

/* 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 );
}


int main(int argc, char **argv) {
  double x,dy;
  int    i;
  double alpha = 0.1;

省略

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

    /** 作成せよ (3) **/
    x = x - alpha*df(x);

省略

}
```

- 実行結果

  alpha が 0.1 の時

```
[j05002@search1-sample]% ./steepest_decent2 1
trial 0 x -7.369244 f(x) 54.305761
trial 1 x -5.895395 f(x) 34.755687
trial 2 x -4.716316 f(x) 22.243640
trial 3 x -3.773053 f(x) 14.235929
trial 4 x -3.018442 f(x) 9.110995
trial 5 x -2.414754 f(x) 5.831037
trial 6 x -1.931803 f(x) 3.731863
trial 7 x -1.545443 f(x) 2.388393
trial 8 x -1.236354 f(x) 1.528571

省略

trial 94 x -0.000000 f(x) 0.000000
trial 95 x -0.000000 f(x) 0.000000
trial 96 x -0.000000 f(x) 0.000000
trial 97 x -0.000000 f(x) 0.000000
trial 98 x -0.000000 f(x) 0.000000
trial 99 x -0.000000 f(x) 0.000000
```

  alpha が 1 の時

```
[j05002@search1-sample]% ./steepest_decent2 1
trial 0 x -7.369244 f(x) 54.305761
trial 1 x 7.369244 f(x) 54.305761
trial 2 x -7.369244 f(x) 54.305761
trial 3 x 7.369244 f(x) 54.305761
trial 4 x -7.369244 f(x) 54.305761
trial 5 x 7.369244 f(x) 54.305761
trial 6 x -7.369244 f(x) 54.305761
trial 7 x 7.369244 f(x) 54.305761
trial 8 x -7.369244 f(x) 54.305761

省略
```

```
trial 93 x 7.369244 f(x) 54.305761
trial 94 x -7.369244 f(x) 54.305761
trial 95 x 7.369244 f(x) 54.305761
trial 96 x -7.369244 f(x) 54.305761
trial 97 x 7.369244 f(x) 54.305761
trial 98 x -7.369244 f(x) 54.305761
trial 99 x 7.369244 f(x) 54.305761
```

alpha が 0.001 の時

```
[j05002@search1-sample]% ./steepest_decent2 1
trial 0 x -7.369244 f(x) 54.305761
trial 1 x -7.354506 f(x) 54.088755
trial 2 x -7.339797 f(x) 53.872616
trial 3 x -7.325117 f(x) 53.657341
trial 4 x -7.310467 f(x) 53.442926
trial 5 x -7.295846 f(x) 53.229368
trial 6 x -7.281254 f(x) 53.016664
trial 7 x -7.266692 f(x) 52.804809
trial 8 x -7.252158 f(x) 52.593801

省略

trial 94 x -6.105115 f(x) 37.272426
trial 95 x -6.092905 f(x) 37.123486
trial 96 x -6.080719 f(x) 36.975140
trial 97 x -6.068557 f(x) 36.827387
trial 98 x -6.056420 f(x) 36.680225
trial 99 x -6.044307 f(x) 36.533651
```

2.3 Level 2.3

- プログラムソース

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define X_MAX 10.0    /* 定義域の最大値 */
#define X_MIN -10.0   /* 定義域の最小値 */
#define X_RANGE (abs(X_MAX)+abs(X_MIN))


void usage();
double f(double x);
double df(double x);

省略

/* 関数 f(x)
 *    入力された x に対する y=f(x) の値を求め，返す.
 */
double f(double x) {
  double y;

  /** 作成せよ (1) **/
  y = -x*sin(x);
  return( y );
}

/* 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 = -sin(a) - a*cos(a);
  return( y_dx );
}


int main(int argc, char **argv) {
  double x,dy;
  int    i;
  double alpha = 0.1;

省略

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

    /** 作成せよ (3) **/
    x = x - alpha*df(x);

省略

}
```

- 実行結果

  解を発見できなかった実行結果

```
[j05002@search1-sample]% ./steepest_decent3 11
trial 0 x -1.061687 f(x) -0.927042
trial 1 x -1.200751 f(x) -1.119473
trial 2 x -1.337408 f(x) -1.301149
trial 3 x -1.465628 f(x) -1.457530
trial 4 x -1.580461 f(x) -1.580387
trial 5 x -1.678929 f(x) -1.669123
trial 6 x -1.760225 f(x) -1.728738
trial 7 x -1.825292 f(x) -1.766500
trial 8 x -1.876118 f(x) -1.789348

省略

trial 95 x -2.028758 f(x) -1.819706
trial 96 x -2.028758 f(x) -1.819706
trial 97 x -2.028758 f(x) -1.819706
trial 98 x -2.028758 f(x) -1.819706
trial 99 x -2.028758 f(x) -1.819706
```

  解を発見できた実行結果

```
[j05002@search1-sample]% ./steepest_decent3 7
trial 0 x 8.415290 f(x) -7.124042
trial 1 x 8.052005 f(x) -7.894648
trial 2 x 7.991642 f(x) -7.916039
trial 3 x 7.981030 f(x) -7.916705
trial 4 x 7.979099 f(x) -7.916727
trial 5 x 7.978745 f(x) -7.916727
trial 6 x 7.978680 f(x) -7.916727
trial 7 x 7.978668 f(x) -7.916727
trial 8 x 7.978666 f(x) -7.916727

省略

trial 95 x 7.978666 f(x) -7.916727
trial 96 x 7.978666 f(x) -7.916727
```

```
trial 97 x 7.978666 f(x) -7.916727
trial 98 x 7.978666 f(x) -7.916727
trial 99 x 7.978666 f(x) -7.916727
```

試行 10 回中 7 回最適解を発見できた。

2.4 考察

- alpha はどれくらいの刻み幅で移動していくかを定めるものなので, 実行結果からも分かるように,alpha の値が大きいと, 探索の精度が悪くなってしまう。しかし,alpha の値が小さ過ぎると, 最小値を見つける前に繰り返し回数を超してしまい解を見つけられないことが起きる場合がある。以上のことから探索の精度を上げ, 解を発見できるようにするには,alpha の値を小さくし, 繰り返し回数を増やすという方法が良いと考えられる。

- Level 2.3 の解の発見の成功確率を上げる方法としては, 検索の範囲を狭めるという方法がある。Level 2.3 のグラフは偶関数なので,x の範囲を $-10 \sim 10$ では無く $0 \sim 10$ とすれば解の発見の成功確率を上げることができる。この方法は偶関数全てに用いる事ができるので,$f(x) = f(-x)$ が成り立てば上記の方法が有効である。

3 Level 3

3.1 Level3.2

要素の数が 3 個の時, 解 x は 001(1 つ目をナップサックに入れる）, 011(1 つ目と 2 つ目をナップサックに入れる）という表現で表すことができる．したがって要素の数が n の時, 問題空間サイズは
2^n
で表すことができる.

問題空間サイズの増加の具合を表す図を以下に示した。

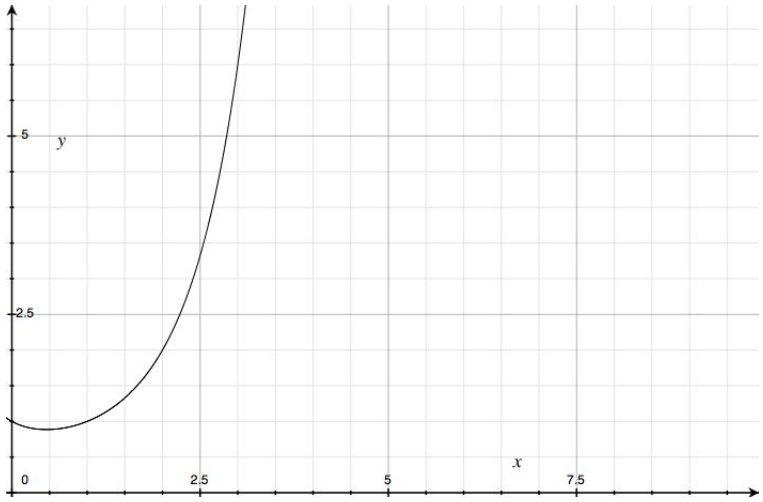

図 3: 問題空間サイズのグラフ

ここで要素数に対する問題空間サイズと 1 秒で 2000 個の探索を行える能力を持った計算機を用いたときの, 所要時間を以下に表で示す.

要素数	問題空間サイズ	所要時間
2	4	0.002s
3	8	0.004s
4	16	0.008s
5	32	0.016s
6	64	0.032s
7	128	0.064s
8	256	0.128s
9	512	0.256s
・ ・ ・	・ ・ ・	・ ・ ・

検索の方法として, まず全ての要素の 1kg あたりの金額を計算し, 金額が高いものから重量オーバーになるまで入れていくという方法がある. この方法は, 重さと値段のバランスがよいものを優先的に入れていくため,(順) 最適解を導くことができると言える. 以下の図はこの方法のフローチャートである.

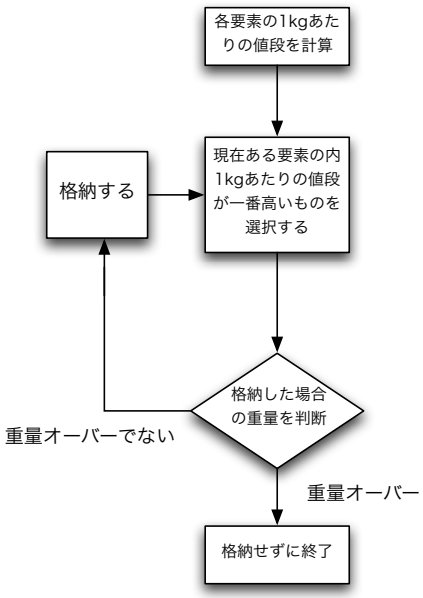

図 4: フローチャート

3.2 Level3.3

要素の数が 3 つ (都市を A,B,C とおく) の時, 解は ABC,ACB という表現ができる. したがって問題空間サイズは, 要素数 n の時
$(n \times n-1 \times \cdots 2 \times 1) \div 2$ となる.

問題空間サイズの増加の具合を表す図を以下に示した。

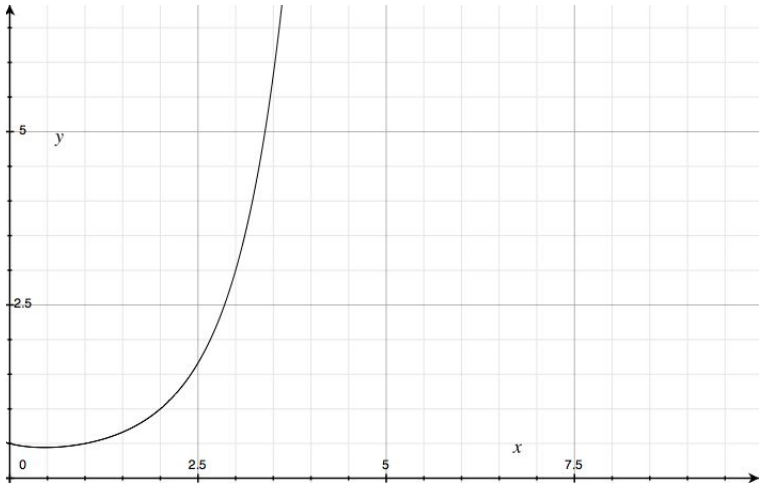

図 5: 問題空間サイズのグラフ

ここで要素数に対する問題空間サイズと 1 秒で 2000 個の探索を行える能力を持った計算機を用いたときの, 所要時間を以下に表で示す.

要素数	問題空間サイズ	所要時間
2	1	0.0005s
3	3	0.0015s
4	12	0.006s
5	60	0.03s
6	360	0.18s
7	2520	1s
8	20160	10s
9	181440	1.5m
・ ・ ・	・ ・ ・	・ ・ ・

探索の方法として, 現在の都市から, まだ行っていない都市への距離を計算し, 最も近い都市へ移動するということを繰り返す方法がある. この方法は, 計算量は全探索に比べると, だいぶ少なくなるが, 一番近い都市に移動してあと, その都市から一番近い都市へ移動したとき, その合計が小さくならなくなり,(順) 最適解としてふさわしくない場合がある. 以下の図はこの方法のフローチャートである.

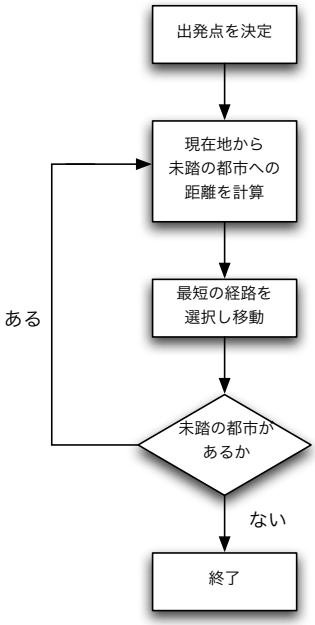

図 6: フローチャート