ルート検索アルゴリズムの研究
（2007年度最終報告書）
船井電機（株）御中

東京海洋大学
久保 幹雄

目 次

第 1 章

はじめに

本報告書は，2007年度の共同研究（ルート検索アルゴリズム）の成果についてまとめたものである．

最短路問題は，様々な場面で登場する基本的なネットワーク最適化問題であり，次のように定義される．

有向グラフ $G = (V, E)$，枝長（移動時間，費用）を表す写像 $c : E \to \mathrm{R}^+$ を与えたとき，始点 s から終点 t までの最短路（パス）を求めよ．

本プロジェクトでは，この問題に対する最近の研究を総括するとともに，ナビゲーション・システムのコアになる実用的なアルゴリズムを開発することを目標とする．本年度の成果は，以下のようにまとめられる．

- 大規模最短路問題に対する前処理を用いた高速アルゴリズム

  通常の教科書に掲載されている最短路アルゴリズムは，1 つの点から他のすべての点（もしくは他のすべての点から1 つの点）への最短路を求めることを目的としたものである．これについては，研究しつくされた感がある．

  道路ネットワークを想定した最短路問題は，非負の枝長をもつ点対点（Point-to-Point: P2P）最短路問題となる．道路ネットワークの規模はヨーロッパで 2000 万点，北米で 3000 万点のものが整備されており，これらの大規模ネットワークを想定し，高速な最短路アルゴリズムを開発する．

  本研究で開発したアルゴリズムは，階層メッシュ疎化法（Level-wise Mesh Sparsification: LMS）と2次元ビットベクトル法である．また，従来法で最も高速なクエリを達成している経由点通過法（transit node routing）も実装する．本報告書では，これらのアルゴリズムの実装について述べ，さらに長所・短所について分析する．

- メモリ階層を用いた高速化

  従来の前処理を用いた最短路アルゴリズムは，すべての情報がメモリ上に確保されているという状態で競争を行っていた．しかし，実際のナビシステムでは，小さなメモリ空間しか用いることができず，またL1, L2 キャッシュの有効利用も高速化には重要である．ここでは，そのようなメモリ階層を考慮した最短路アルゴ

リズムの高速化について，実用的な観点から検討する．

以下の構成は次の通り．

第 **??** 章では，我が国のナビで用いられている最短路アルゴリズムを概観し，その問題点を指摘する．

第 2 章では，基本となる 1 対全の最短路問題に対するプログラムの解説を行う．

第 3 章では，2 次元ビットベクトル法の改良版について述べ，Python によるプログラムの解説を行う．

第 4 章では，経由点通過法を紹介し，Python によるプログラムの解説を行う．

第 **??** 章では，階層メッシュ疎化法について述べる．

第 chapter:メモリ階層構造を考慮した高速化 章では，メモリ階層を用いた高速化について考え，高速化した C 言語による Dial 法のプログラムを解説する．

また付録として，Python 言語と使用したモジュールの概要を紹介する．

第 2 章

基本プログラム

ここでは，基本となる最短路アルゴリズムの実装法について述べる．インターネットを検索すると，様々な最短路アルゴリズムのプログラムが見つかるが，大規模問題例に対して高速に最短路を求めることができるものは意外と少ない．ここでは，適切なデータ構造を選択した「正しい」実装を行ったプログラムを紹介する．

2.1 Python による Dial 法

簡単な例として，Python による Dial 法を紹介する．グラフを記述するために networkX モジュールを利用する．

networkX は，Python によって記述されたグラフクラスである．networkX は `https://networkx.lanl.gov/wiki` からダウンロードできる．他にも，グラフ描画のためのモジュール Scipy `http://matplotlib.sourceforge.net/` や高速な配列と数値計算のためのモジュール NumPy `http://numpy.scipy.org/` もインストールする必要がある．

DIMACS データを読み込んでグラフを返すルーチンを以下に示す．また，同時に枝長の最大値に 1 を加えたもの C を計算している．これは，Dial 法の巡回リストの長さを規定するときに用いられる．

```
def ReadDimacs(filename):
    '''Read the data from DIMACS shortest path format
       that can be downloaded from DIMACS site'''

    f=open(filename+'.co', 'r')
    f2=open(filename+'.gr', 'r')

    # read data from file
    line=f.readline()
    x_cord=[]
    y_cord=[]
    G=XGraph()
    G.position={}

    while line.find("v 1")==-1:
        line=f.readline()

    (dum,i,x,y)=line.split()
    x_cord.append(float(x))
    y_cord.append(float(y))

    while 1:
```

```
        line=f.readline()
        if line.find("v")==0:
            (dum,i,x,y)=line.split()
            x_cord.append(float(x))
            y_cord.append(float(y))
        else:
            break

    #edge info
    edge_info=[]
    line=f2.readline()

    for i in range(6):
        line=f2.readline()
        #print(line)

    while 1:
        line=f2.readline()
        if line.find("a")==0:
            (dum,i,j,length)=line.split()
            edge_info.append( (int(i)-1,int(j)-1,int(length)) )
            #print ("edge info",i,j,length)
        else:
            break

    C=max(edge_info)[2]
    #print "Max Edge Length=",C
    C+=1

    minx=float(min(x_cord))
    maxx=float(max(x_cord))
    xwidth= maxx-minx+1

    miny=float(min(y_cord))
    maxy=float(max(y_cord))
    ywidth=maxy-miny+1

     # add nodes after scaling coordinates
    n=len(x_cord)
    for i in range(n):
        G.add_node(i)
        x=(x_cord[i]-minx)/xwidth
        y=(y_cord[i]-miny)/ywidth
        G.position[i]=(x,y)
        #print i,G.position[i]

    print "number of nodes=",n

    # add edges
    m=len(edge_info)
    for (i,j,length)in edge_info:
        G.add_edge(i,j,length)
        #print ("add edge",i,j)

    f.close()
    f2.close()

    return G,n,m,C
```

Dial法のメインルーチンは，以下のように書ける．巡回リストは，リストを要素としたリストとして表現している．他のデータは，Numpyモジュールの配列を用いている．これによって，メモリの節約と高速化が実現される．

```
if __name__ == "__main__":
    filename="DC.tmp"
    G,n,m,C = ReadDimacs(filename)
    print "number of nodes =",n

    source=0
    sink =-1 #if sink =-1 then find all shortest paths from source to other nodes

    settled= numpy.zeros(n,numpy.int0) #=1 if node is settled (or has an eternal label or is scanned)
    reached= numpy.zeros(n,numpy.int0) #=1 if node is heap or list (or has a finite potential)
    dist= numpy.zeros(n,numpy.int0)
    prev= numpy.zeros(n,numpy.int0)
    circular=[[-1] for i in range(C) ] #-1 represents the list is empty
    circular[0].append(source)
    reached[source]=1
    prev[source]=-1
    current=0
    #scanned=0
    while 1:
        first=current
        while 1:
            v=circular[current][-1] #v is the node with minimum potential
            if v>=0:
                break
            #the current list is empty, so we increment it
            current+=1
            current= current%C
            if current==first:
                #the circular list is empty, so we quit
                v=-1
                break

        if v==-1: #the circular list is empty, so we quit
            break
        circular[current].pop()
        settled[v]=1
        if v==sink:
            break
        #reached[v]=0
        #scanned+=1
        #print "scanning node",v,scanned,circular
        for w in G.neighbors(v): #scan operation
            if settled[w]==0:
                temp=dist[v]+G.get_edge(v,w)
                if reached[w]== 0: #w is not in the circular list
                    reached[w]=1
                    dist[w]=temp
                    prev[w]=v
                    circular[temp%C].append(w)
                else: #w is in the circular list
                    if temp<dist[w]:
                        circular[dist[w]%C].remove(w)
                        circular[temp%C].append(w)
                        dist[w]=temp
                        prev[w]=v

    if sink>=0:
        path=[sink]
        w=sink
        while 1:
            v=prev[w]
            path.append(v)
            if v==source:
                break
            w=v
        path.reverse()
```

```
        print "Path Length=",dist[sink]
        print "Optimal Path",path
    else:
        print "Distance Array=",dist
        print "Prev Array",prev
```

2.2 CによるDial法

宮本によるDial法のC言語による実装を紹介する．これは単純ではあるが，非負の整数値を枝長とする道路ネットワークに対しては高速である．

まず，グラフをforward-starとよばれるデータ構造で保持するための構造体FSTARと距離を入れるためのバケットに挿入する点の情報を表す構造体DLISTを準備しておく．

```
typedef struct {
      int n;
      int m;
      int *point;
      int *tail;
      int *head;
      int *length;
} FSTAR;

typedef struct {
    int n;        //点の数
      int *val;   //最短距離を保持する配列
      int *next;  //つなぎリストの次の点を保持する配列
      int *prev;  //つなぎリストの前の点を保持する配列
} DLIST;
```

各点から出ている枝の先の点（tail）を距離の短い順に並べ替えておく．そのためのサブルーチンは，以下のように書ける．これは，ソーティングにQuick Sortを用いている．

```
void sortByTail(int start, int end, int min, int max, int tail[], int head[], int length[]) {
      int j, k, half, tmp;

      if (start >= end || min >= max) return;
      j = start;
      k = end;
      half = (min + max)/2;
      for (; j < k;) {
             if (tail[j] <= half) j++;
             else {
                   tmp = tail[k];
                   tail[k] = tail[j];
                   tail[j] = tmp;
                   tmp = head[k];
                   head[k] = head[j];
                   head[j] = tmp;
                   tmp = length[k];
                   length[k] = length[j];
                   length[j] = tmp;
                   k--;
             }
      }
      if (tail[j] <= half) k++;
      else j--;
      sortByTail(start, j, min, half, tail, head, length);
      sortByTail(k, end, half + 1, max, tail, head, length);
```

```
}
```

DIMACS フォーマットのデータを読み込む関数は，以下のように記述できる．この中で forward-star のデータ構造を組み立てる．

また，同時に最大の枝長を計算し，それをメインルーチンに返す．これは，Dial 法でバケットの長さを決めるときに用いられる．

```
#define L_MAX 100 // maximum length of a line of DIMACS graph file

int readDIMACSGraph(char fileName[], FSTAR *g) {
      FILE *fp;
      char line[L_MAX], dummy1[L_MAX], dummy2[L_MAX];
      int m, tail, head, length, a, t, n;
      int maxlen;

      maxlen = 0;
      fp = fopen(fileName, "r");
      m = 0;
      while (fgets(line, L_MAX, fp)) {
             if (strncmp(line, "a ", 2) == 0) {
                   sscanf(line, "%s %d %d %d", dummy1, &g->tail[m], &g->head[m], &g->length[m]);
                   if (maxlen < g->length[m]) maxlen = g->length[m];
                   m++;
             }
             else if (strncmp(line, "p sp ", 5) == 0) {
                   sscanf(line, "%s %s %d %d", dummy1, dummy2, &g->n, &g->m);
                   g->point = malloc(sizeof(int)*(g->n + 2));
                   g->tail = malloc(sizeof(int)*g->m);
                   g->head = malloc(sizeof(int)*g->m);
                   g->length = malloc(sizeof(int)*g->m);
             }
      }
      close(fp);
      sortByTail(0, g->m - 1, 1, g->n, g->tail, g->head, g->length);
      t = 0;
      for (a = 0; a < g->m; a++) {
             if (t < g->tail[a]) {
                   for (n = t + 1; n <= g->tail[a]; n++) g->point[n] = a;
                   t = g->tail[a];
             }
      }
      if (t < g->n + 1) {
             for (n = t + 1; n <= g->n + 1; n++) g->point[n] = g->m + 1;
      }
      return maxlen;
}
```

バケット point に距離 p，点の IDi の点を（バケットの先頭に）挿入するサブルーチンは，以下のように記述される．バケットは，距離 p ごとに保持される（両方向の）つなぎリストである．

```
void add(int *point, DLIST *list, int p, int i) {
      list->next[i] = point[p];
      list->prev[i] = -1;
      list->prev[point[p]] = i;
      point[p] = i;
}
```

バケット point から距離 p，点の IDi の点を削除するサブルーチンは，以下のように書ける．

```
void delete(int *point, DLIST *list, int p, int i) {
      if (list->next[i] > 0) list->prev[list->next[i]] = list->prev[i];
      if (list->prev[i] > 0) list->next[list->prev[i]] = list->next[i];
      else point[p] = list->next[i];
}
```

上のサブルーチン群を用いることによって，Dial法による最短路計算のサブルーチンは，以下のように書ける．

```
int dijkDial(FSTAR *g, int backetsize, int orig, int dest, int backet[], DLIST *list) {
      int min, a, i, fixed, head;

      //すべてのバケットを空に初期設定する．
      for (i = 0; i < backetsize; i++) backet[i] = -1;

      //各点のラベル（=最短距離）の初期設定
      //点が未探索であることを，ラベルに -1 を入れることによって代用している．
      //Transit Node Routingなどで，グラフの一部の最短路を求める際には，永久ラベルがついた点の情報も
      //配列で保持することによって，初期設定を高速化できる．
      for (i = 1; i <= g->n; i++) list->val[i] = -1;

      //始点origを設定し，バケットの番号minを0に初期設定しておく．
      min = 0;
      list->val[orig] = 0;
      add(backet, list, 0, orig);
      for (;;) {
            //ラベルが最小の点fixedをバケットから求め，バケットから除く．
             for (i = min;;) {
                   if (backet[i] > 0) {
                         fixed = backet[i];
                         delete(backet, list, i, fixed);
                         min = i;
                         break;
                   }
                   i = (i + 1)%backetsize;
                  //バケットは巡回リストであり，1周まわったら空であるので，点なし（-1）を返す．
                   if (i == min) {
                         fixed = -1;
                         break;
                   }
             }
            //バケットが空もしくはfixedが終点（dest）なら終了する．
             if (fixed < 0 || fixed == dest) break;

            //点fixedから出ている枝の先の点のラベル更新（いわゆる走査を行う）
             for (a = g->point[fixed]; a < g->point[fixed + 1]; a++) {
                   head = g->head[a];
                   if (list->val[head] < 0) {
                         list->val[head] = list->val[fixed] + g->length[a];
                         add(backet, list, list->val[head]%backetsize, head);
                   }
                   else if (list->val[fixed] + g->length[a] < list->val[head]) {
                         delete(backet, list, list->val[head]%backetsize, head);
                         list->val[head] = list->val[fixed] + g->length[a];
                         add(backet, list, list->val[head]%backetsize, head);
                   }
             }
      }
      return list->val[dest];
}
```

上のサブルーチンを呼び出すメインルーチンの例を示す．

```
int main(int argc, char* argv[])
{
```

```
    FSTAR g;
    DLIST list;
    int *backet, maxlen;
    int o, d, q;
    int *orig, *dest, qnum;
    FILE *fp;

    if (argc < 2) {// usage
             printf("usage: a.out DIMACSFile\n");
             printf("usage: a.out DIMACSFile P2PFile\n");
    }
    else if (argc < 3){// interactive mode
             maxlen = readDIMACSGraph(argv[1], &g);
             backet = malloc(sizeof(int)*(maxlen + 1));
             list.val = malloc(sizeof(int)*(g.n + 1));
             list.next = malloc(sizeof(int)*(g.n + 1));
             list.prev = malloc(sizeof(int)*(g.n + 1));
             for (;;) {
                 printf("o d = ");
                 scanf("%d %d", &o, &d);
                 if (o <= 0 || d <= 0) break;
                 printf("opt = %d\n", dijkDial(&g, maxlen + 1, o, d, backet, &list));
             }
             free(g.point);
             free(g.tail);
             free(g.head);
             free(g.length);
             free(backet);
             free(list.val);
             free(list.next);
             free(list.prev);
    }
    else {// batch mode
             fp = fopen(argv[2], "r");
             fscanf(fp, "%d", &qnum);
             orig = malloc(sizeof(int)*qnum);
             dest = malloc(sizeof(int)*qnum);
             for (q = 0; q < qnum; q++) {
                 fscanf(fp, "%d %d", &orig[q], &dest[q]);
             }
             fclose(fp);
             maxlen = readDIMACSGraph(argv[1], &g);
             backet = malloc(sizeof(int)*(maxlen + 1));
             list.val = malloc(sizeof(int)*(g.n + 1));
             list.next = malloc(sizeof(int)*(g.n + 1));
             list.prev = malloc(sizeof(int)*(g.n + 1));
             for (q = 0; q < qnum; q++) {
                 printf("opt = %d\n", dijkDial(&g, maxlen, orig[q], dest[q], backet, &list));
             }
             free(g.point);
             free(g.tail);
             free(g.head);
             free(g.length);
             free(backet);
             free(list.val);
             free(list.next);
             free(list.prev);
             free(orig);
             free(dest);
    }
}
```

2.3 階層メッシュ疎化法（LMS 法）

ここでは，我々が提案したLMS 法の実装について述べる．LMS 法のプログラムは，様々なサブルーチンから構成されており，複雑であるため，使用法と主要部のみを解説する．

2.3.1 使用法

まず，使用法について述べる．

LMS 法は1つのC 言語のプログラムファイルで構成されており，コンパイラ（ここではgcc を想定）がインストールされた環境で，

```
gcc lms.c
```

とコンパイルできる．これによって，a.out という実行ファイルが生成されたものとする．

これを用いて（たとえば）ワシントンDC の地図データで実験するときには，DIMACS フォーマットのDC.tmp.co（座標が保管されているデータ）, DC.tmp.gr（グラフが保管されているデータ）を用意し，

```
a.out DC.tmp.co DC.tmp.gr DC.lms 8
```

とすると，DC.lms という名前で 8 層からなるLMS 用のデータが生成される．

この時点では疎化ネットワークは作られておらず，続けて

```
a.out addall DC.lms
```

とすると全ての疎化ネットワークが添加される．その後，

```
a.out test DC.lms
```

とするとビットベクトル法の実験を行うことができる．

また，

クエリ数

始点1 終点1

始点2 終点2

…

というフォーマットのクエリファイルを作成し，

```
a.out p2p DC.lms クエリファイル
```

でLMS 法のP2P クエリの計算実験を行うことができる．

2.3.2 Dijkstra法

LMS法は，ヒープを用いたDijkstra法を用いる．通常のヒープの実装との違いは，各点がヒープのどの場所に保管されているかの情報を保持する点である．以下では，ヒープに保管する点のIDを配列heap[]に保管し，その逆写像をpeah[]（heapの逆綴り）に保管するものとする．

ヒープのi番目とj番目の要素を交換するサブルーチンを準備する．

```
void swap(int heap[], int paeh[], int i, int j) {
    int tmp;

    tmp = heap[i];
    heap[i] = heap[j];
    heap[j] = tmp;
    paeh[heap[i]] = i;
    paeh[heap[j]] = j;
}
```

点ID n の点をヒープに入れて，ヒープ構造を保持するように修正するサブルーチンは，以下のように書ける．

```
int updateHeap(int distance[], int heap[], int paeh[], int n, int heapSize) {
    int i;

    if (paeh[n] < 0) {
              heapSize++;
              heap[heapSize] = n;
              paeh[n] = heapSize;
    }
    i = paeh[n];
    for (;i > 1;) {
              if (distance[heap[i/2]] > distance[heap[i]]) {
                  swap(heap, paeh, i/2, i);
                  i = i/2;
              }
              else {
                  break;
              }
    }
    return heapSize;
}
```

ヒープ内の最小値（heap[1]）を削除し，ヒープ構造を保持するように修正するサブルーチンは，以下のように書ける．

```
int deletemin(int heap[], int paeh[], int heapSize, int distance[]) {
    int i;

    paeh[heap[1]] = 0;
    heap[1] = heap[heapSize];
    paeh[heap[1]] = 1;
    heapSize--;
    i = 1;
    for (;;) {
              if (2*i + 1 <= heapSize) {
                  if (distance[heap[2*i]] < distance[heap[2*i + 1]]) {
                            if (distance[heap[2*i]] < distance[heap[i]]) {
                                swap(heap, paeh, i, 2*i);
                                i = 2*i;
                            }
                            else break;
```

```
                }
                else {
                        if (distance[heap[2*i + 1]] < distance[heap[i]]) {
                            swap(heap, paeh, i, 2*i + 1);
                            i = 2*i + 1;
                        }
                        else break;
                }
            }
            else if (2*i <= heapSize) {
                if (distance[heap[2*i]] < distance[heap[i]]) {
                        swap(heap, paeh, i, 2*i);
                        i = 2*i;
                }
                else break;
            }
            else break;
    }
    return heapSize;
}
```

ヒープを組み込んだ（1対全の）Dijkstra法は，以下のように記述できる．LMS法で用いるため，グラフを枝リストARCLIST構造体のarcで入力していること，また，同じ最短距離を達成する「直前の」枝が複数あるときには，それをリストnextPrevに保管しておくことに注意されたい．

```
// Single Source Type Shortest Path
void dijkstraWithHeapSS(ARCLIST *arc, int source, int distance[], int prev[], int nextPrev[]) {
    int a, fixed, n, heapSize, p;
    int *heap, *paeh;

    // initialize
    heap = malloc((arc->n + 1)*sizeof(int));
    paeh = malloc((arc->n + 1)*sizeof(int));
    for (n = 1; n <= arc->n; n++) {
            distance[n] = INFINITY;
            prev[n] = -1;
            paeh[n] = -1;
    }
    for (a = 0; a < arc->m; a++) nextPrev[a] = -1;
    // set start
    heapSize = 1;
    distance[source] = 0;
    heap[1] = source;
    paeh[source] = 1;
    // Iteration
    for (; heapSize > 0;) {//ヒープが空になるまで実行する.
            fixed = heap[1];   //ヒープの先頭の点を fixed とする.
            heapSize = deletemin(heap, paeh, heapSize, distance);//先頭を削除
            if (distance[fixed] >= INFINITY) break;

             //fixedから出ている枝の先の点 n の走査
            for (a = arc->incident[fixed]; a >= 0; a = arc->next[a]) {
                n = arc->head[a];
                if (distance[n] > distance[fixed] + arc->length[a]) {
                        distance[n] = distance[fixed] + arc->length[a];
                        prev[n] = a;
                        heapSize = updateHeap(distance, heap, paeh, n, heapSize);
                }
                else if (distance[n] == distance[fixed] + arc->length[a]) {
                        if (prev[n] >= 0) {
                            for (p = prev[n]; nextPrev[p] >= 0; p = nextPrev[p]);
                            nextPrev[p] = a;
                        }
```

```
                        else prev[n] = a;
                        heapSize = updateHeap(distance, heap, paeh, n, heapSize);
                }
            }
    }
    free(heap);
    free(paeh);
}
```

第 3 章

2 次元ビットベクトル法

ここでは，我々が提案するアルゴリズムのうちの 1 つである 2 次元ビットベクトル法の実装について考える．用いるプログラミング言語は Python であり，主にプロトタイピングの役割を果たす．

3.1 アルゴリズム概要

ここでは，前報告書で提案した 2 次元ビットベクトルを用いた解法の改良版について述べる．なお，以下では簡単のためグラフ（距離行列）は対称（行きと帰りの移動距離が同じ）であると仮定する．

点集合を適当な部分集合（領域）に分割し，始点 s を含む領域 R_s から終点 t を含む領域 R_t の情報を用いて枝上にフラグを立てることを考える．領域数を 10 程度にすれば，100 ビットの情報が枝上に保管されることになる．領域 R_s に含まれるいずれかの点から領域 R_t に含まれるいずれかの点への最短路が，枝 (i,j) を使うなら $b_{ij}(R_s,R_t)=1$ ，それ以外なら 0 とする．これは，前報告書で提案したアルゴリズムによって，領域 R_s,R_t 内の縁の点同士の最短路を事前に求めておくことによって構成できる．

始点領域 R_s と終点領域 R_t を与えたとき，2 次元ビットベクトル $b_{ij}(R_s,R_t)$ が 1 になっている枝 (i,j) で，両端点 i,j の両者が始点領域 R_s もしくは終点領域 R_t 内ではないものから構成されるグラフを考える．このグラフにおいて，次数が 2 の点を抽出し，それらの点を短絡除去しておく．具体的には，点 j の次数が 2 であり，$(i,j),(j,k)$ の枝があるものとする．このとき，枝 (i,k) を新たに引き，枝 $(i,j),(j,k)$ を除去する．枝 (i,k) に対する移動時間などの重みは，元のグラフにおける点 i から点 k への最短移動時間に設定する．なお，多目的を有する最短路問題の場合には，各目的に対する非劣ベクトルの集合を保持する必要があり，時刻依存最短路の場合には，パスの終点 k に到達する時刻を，始点 i を出発する時刻の関数として保持する必要がある．

2 つの相異なる領域 S,T を与えたとき，2 次元ビットベクトル $b_{ij}(S,T)$ が 1 になっている枝 (i,j) を短絡除去したグラフを $G(S,T)$ と記す．

始点領域 S に対する経由点の集合 $TRAN(S)$ を以下のように定義する．始点領域 S 自身と S の前後左右の領域を，S の近傍領域と定義する．S の近傍領域に含まれない点で，S の近傍領域内の点と隣接する点を S の経由点とよぶ．$TRANS(S)$ は経由点をすべて集めた集合である．終点領域 T に対知る経由点の集合 $TRAN(T)$

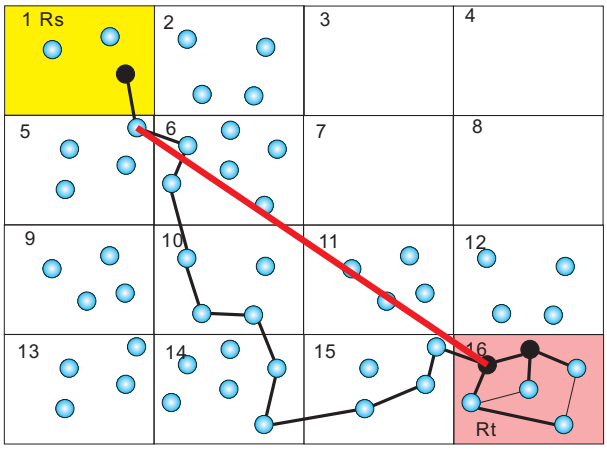

図 3.1: 2 次元ビットベクトル $b_{ij}(R_s, R_t)$ が 1 になっている枝 (i, j)（細線）と，その中で両端点 i, j の両者が終点領域 R_t 内ではないものを短絡除去したグラフ（太線）.

も同様に定義する.

始点領域内のすべての点から経由点への最短距離を保持しておく．これは，各経由点に対する最短路を（S 内のすべての点に永久ラベルがつくまで Dijkstra 法の探索を行うことによって）1 度求めれば良い．同様に，経由点から終点領域内のすべての点への最短距離も保持しておく．これは，方向性をもった経由点通過法とも考えられ，クエリ時には，経由点通過法より少ないメモリで探索が可能になる.

3.2 Python による実装と実験

Python（言語仕様については付録参照）は様々なモジュールが用意されている．ここでは，以下のモジュールを用いて実装した.

networkX: networkX は，Python によって記述されたグラフクラスである．networkX は `https://networkx.lanl.gov/wiki` からダウンロードできる．グラフの描画のためには，他のモジュールもインストールする必要がある．Windows 環境における最も簡単な方法は，Scipy `http://scipy.org/Download` をインストールすることである.

Boost Graph Library: Boost Graph Library は，C++で記述されたグラフアルゴリズムのライブラリであり，ここでは networkX の最短路アルゴリズムを補完する目的で使用した．Python との接続のためには，`http://osl.iu.edu/~dgregor/bgl-python/`をインストールする必要がある．また，`http://boost.cppll.jp/` に日本語の解説がある.

2 次元ビットベクトル法に対する記述をする前に，これらのモジュールと，部品となる関数を記述したモジュールを読み込む.

```
from pylab import *
from networkx import *  # import networkx after pylab
```

```
from boost1 import *
from read_dimacs import *
```

pylabはnetworkXで用いるモジュールである．boost1はBoost Graph Library関連の関数を記述した以下のモジュールである．

```
import boost.graph as boost
class PathFind:
    def __init__(self):
        self.graph = boost.Graph()
        self.weights = self.graph.edge_property_map('float')
        self.vertexesid = self.graph.vertex_property_map('integer')
        self.vertexes = {}

    def __addvertex__(self, id):
        if not self.vertexes.has_key(id):
            v = self.graph.add_vertex()
            self.vertexes[id] = v
            self.vertexesid[v] = id
            return v

        return self.vertexes[id]

    def addEdge(self, srcVertex, dstVertex, cost):
        src = self.__addvertex__(srcVertex)
        dst = self.__addvertex__(dstVertex)

        edge = self.graph.add_edge(src, dst)
        self.weights[edge] = cost

    def findPath(self, src, dst):
        p = self.graph.vertex_property_map('vertex')
        boost.dijkstra_shortest_paths(self.graph, self.vertexes[src],
                                      predecessor_map = p,
                                      weight_map = self.weights)

        path = [dst]
        while src != dst:
            dst = self.vertexesid[p[self.vertexes[dst]]]
            path.append(dst)
        path.reverse()

        return path

    def findAllShortestPath(self, src):
        p = self.graph.vertex_property_map('vertex')
        boost.dijkstra_shortest_paths(self.graph, self.vertexes[src],
                                      predecessor_map = p,
                                      weight_map = self.weights)

        return p

    def findLength(self, src, dst):
        d = self.graph.vertex_property_map('float')
        #v = self.graph.dijkstra_visitor #no property!

        boost.dijkstra_shortest_paths(self.graph, self.vertexes[src],
                                      distance_map = d,
                                      visitor = vis(),
                                      weight_map = self.weights)
        print vis.tree_edge()
        return d
```

```
class vis(boost.dijkstra_visitor):
        def __init__(self):
            self.edges = list()

        def tree_edge(self, e, g):
            self.edges.append(e)

        def examine_vertex(self, vertex, graph):
            pass

        def finish_vertex(self, vertex, graph):
            pass
```

read_dimacs は，DIMACS フォーマットのデータを読み込むための関数を記述したモジュールである．データを読み込むだけではなく，グラフの連結成分を求め，最大の連結成分を返すように設計されている．

```
from pylab import *
from networkx import *  # import networkx after pylab

def dis((x1,y1), (x2,y2)):
    xdiff = x2 - x1
    ydiff = y2 - y1
    return math.sqrt(xdiff*xdiff + ydiff*ydiff)


def ReadDimacs(filename):
    f=open(filename+'.co', 'r')
    f2=open(filename+'.gr', 'r')

    # read data from file
    line=f.readline()
    x_cord=[]
    y_cord=[]
    G=XGraph()   #グラフインスタンスの生成
    G.position={} #座標保管用の辞書

    while line.find("v 1")==-1:
        line=f.readline()

    (dum,i,x,y)=line.split()
    x_cord.append(float(x))
    y_cord.append(float(y))

    while 1:
        line=f.readline()
        if line.find("v")==0:
            (dum,i,x,y)=line.split()
            x_cord.append(float(x))
            y_cord.append(float(y))
        else:
            break

    #edge info
    edge_info=[]
    line=f2.readline()

    for i in range(6):
        line=f2.readline()
        #print(line)

    while 1:
        line=f2.readline()
        if line.find("a")==0:
```

```
            (dum,i,j,length)=line.split()
            edge_info.append( (int(i)-1,int(j)-1,int(length)) )
            #print ("edge info",i,j,length)
        else:
            break

    minx=float(min(x_cord))
    maxx=float(max(x_cord))
    xwidth= maxx-minx+1

    miny=float(min(y_cord))
    maxy=float(max(y_cord))
    ywidth=maxy-miny+1

    # add nodes after scaling coordinates
    n=len(x_cord)
    for i in range(n):
        G.add_node(i)  #点集合の追加
        x=(x_cord[i]-minx)/xwidth
        y=(y_cord[i]-miny)/ywidth
        G.position[i]=(x,y)

    # add edges
    m=len(edge_info)
    for (i,j,length)in edge_info:
        G.add_edge(i,j,length)   #枝集合の追加
        #print ("add edge",i,j)

    print "number of edges=",m

    f.close()
    f2.close()

    #グラフ G の最大の連結成分（0番目）をHに入れる
    H=connected_component_subgraphs(G) [0]

    #点の座標の更新
    H.position={}
    for node in H.nodes_iter():
        (x,y)=G.position[node]
        H.position[node]=(x,y)

    n=len(H)
    m=size(H)

    return H,n,m
```

メインルーチンからデータを読み込むには，以下のように記述すればよい．ここでは，DC.tmpというファイル名をもつ小規模データを読み込んでいる．元のデータはグラフを保管するDC.tmp.grと点の座標を保管するDC.tmp.coであるが，それらは上で記述した関数内で変換され，呼び出される．

```
filename="DC.tmp"
G,n,m = ReadDimacs(filename)
```

まず，点の座標をもとに，バケット（格子）に含まれる点集合を抽出する．これには，$O(n)$（n は点の数）時間を要する．

```
#各バケットにおおよそ100の点を入れるようにバケット数を決める
n_bucket=int(sqrt(n/100))
```

```
#バケット(i,j)に所属する点のリストを準備
bucket=[ [  []  for i in range(n_bucket) ]  for j in range(n_bucket)  ]

G.bucket={}  #点の所属するバケットを保管する辞書を準備

for node in G.nodes_iter():   ↑      (x,y)=G.position[node]
      i=int(x*n_bucket)
      j=int(y*n_bucket)
      bucket[i][j].append(node)   #バケット(i,j)のリストに点を追加
      G.bucket[node]=(i,j)            #辞書にバケット番号を保管
```

次に，各バケットに対して縁の点リストを求める．実験では，始点を含むバケット（stgart_i,start_j) と終点を含むバケット（end_i,end_j) を固定して行う．そのために，縁の点集合を抽出する．ここで，set() はリストから集合を生成する関数であり，これによってリスト内の重複する点を削除することができる．

```
#縁の点集合を保管するリストの準備
border=[[[] for i in range(n_bucket)] for j in range(n_bucket)]

for i in range(n_bucket):
      for j in range(n_bucket):
           if len(bucket[i][j])>0:  #空でないバケットに対して
               for node in bucket[i][j]:
                   for n1 in G.neighbors(node): #隣接点集合（networkX）
                        if G.bucket[node] <> G.bucket[n1]:
                            #隣接点が異なるバケットに含まれているなら縁の点集合に追加
                           border[i][j].append(node)

#Make the border set (delete duplicate nodes)
startnodes=set(border[start_i][start_j])
endnodes=set(border[end_i][end_j])
```

始点と終点を含む2 つのバケット間の最短路を計算する．まず，Boost Graph Lirary の最短路計算ルーチンをよぶ準備して，グラフのコピーを行う．

```
# call the boost graph library to find shortest paths
BoostGraph = PathFind()
for (i,j,length)in G.edges_iter():
    BoostGraph.addEdge(i,j,length)
```

始点を含むバケットの縁からの最短路を計算する際には，終点を含むバケットから遠い順に計算し，最短路木に含まれる点は削除することによって，最短路を解く回数を減らすことができる．

そのために，終点を含むバケットの中心点を求めて，そこへの直線距離の順にソートしておく．

```
#Find the center point of the target node
target_x=1/float(n_bucket)*float(end_i)+1/(2*float(n_bucket))
target_y=1/float(n_bucket)*float(end_j)+1/(2*float(n_bucket))

#Sort the nodes in (start_i, start_j) bucket in the nearest ordering to the target region
list1 = [(dis(G.position[s],(target_x,target_y)), s) for s in startnodes]
list1.sort()

#Make a list (list2) that incudes the node number only
list2=[]
for (d,i) in list1:
    list2.append(i)

paths=[]
i=0
```

```
while 1:
    i+=1
    if len(list2)==0:
        break
    s=list2.pop()
    pred=BoostGraph.findAllShortestPath(s)
    for t in endnodes:
        dst=t
        sp = [dst]
        while s != dst:
            dst = BoostGraph.vertexesid[pred[BoostGraph.vertexes[dst]]]
            sp.append(dst)
            #delete the nodes in the shortest paths from list2
            try:
                list2.remove(dst)
            except:
                pass
        sp.reverse()
        paths.append(sp)
```

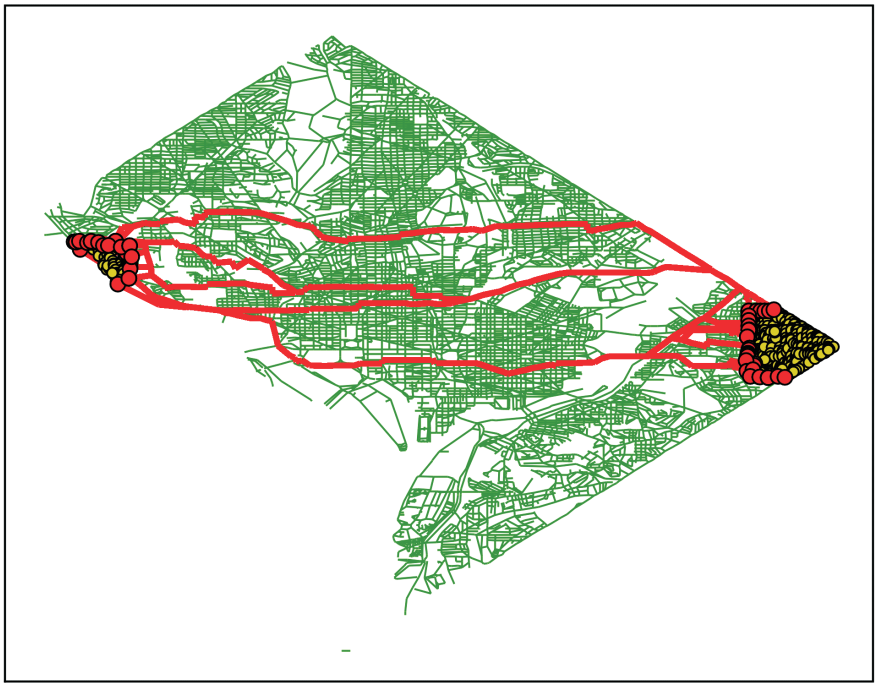

図 3.2: 2 次元ビットベクトル $b_{ij}(R_s, R_t)$ が 1 になっている枝 (i, j) から構成されるグラフ.

結果を描画したものを，図 3.2 に示す．ちなみに，最短路を遠い順に解くことによって，求解回数を約半分に減らすことができた.

縁の点の間の最短路から構成されるグラフは，次数 2 の点がほとんどであること図から見てとれる．確認のため，点の次数のヒストグラムを作成してみる.

```
degseq=G.degree()          #次数を計算してリストを返す関数
dmax=max(degseq)+1   #最大の次数を計算（maxはpythonの関数）
```

```
freq= [ 0 for d in range(dmax) ] #頻度を保管するリストの準備
for d in degseq:
    freq[d] += 1

print "degree frequency histgram",freq
```

結果は $[0, 39, 648, 97, 18]$ となり，次数 2 の点が 648 個と大半を占めていることが確認できる．これらの点は，入って出るだけであるので，削除して前後の点を直結させる（グラフの用語で言うところの点の短絡除去）を行うことができる．

```
# concatenate degree 2 nodes
G1=G.copy()
for node in G1.nodes_iter():
    if G.degree(node)==2:
        nei=G.neighbors(node)
        G.delete_node(node)
        G.add_edge(nei[0],nei[1])
```

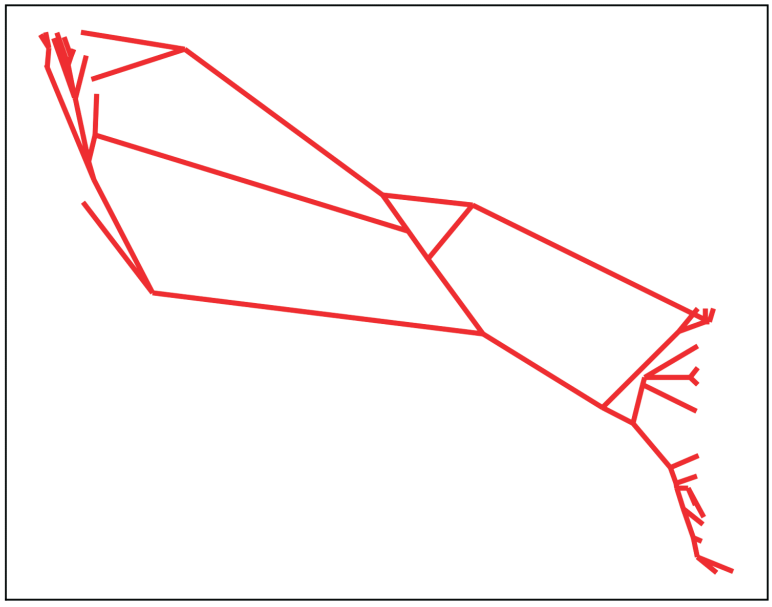

図 3.3: 次数 2 の点の短絡除去後のグラフ．

短絡除去後のグラフを図 3.3 に示す．この図から，始点・終点付近は密であるが，それ以外の中間地点は疎であることが分かる．そこで，始点を含むバケットに隣接するバケットに含まれる点集合を 1 つの点とみなす（グラフの用語で言うところの縮約）を行う．終点を含むバケットに対しても，同様の縮約操作を行うプログラムは，以下のように書ける．

```
#shrink the nodes in the bucket
G2=G.copy()
G.add_node("dummy1")
G.add_node("dummy2")
```

```
for node in G2.nodes_iter():
    if Nearbucket(G.bucket[node],(start_i,start_j)):
        for n1 in G2.neighbors(node):
            if not(Nearbucket(G.bucket[n1],(start_i,start_j))):
                G.add_edge( ("dummy1",n1) )
        try:
            G.position["dummy1"]=G.position[node]
            G.delete_node(node)
        except:
            pass
    if Nearbucket(G.bucket[node],(end_i,end_j)):
        for n1 in G2.neighbors(node):
            if not(Nearbucket(G.bucket[n1],(end_i,end_j))):
                G.add_edge( ("dummy2",n1) )
        try:
            G.position["dummy2"]=G.position[node]
            G.delete_node(node)
        except:
            pass
```

ここで用いた隣接バケットの判定用の関数 Nearbucket() は，boost1 モジュールに記述されている．

```
def Nearbucket((i1,j1), (i2,j2)):
    ret=0
    for k in range(i2-1,i2+2):
        for l in range(j2-1,j2+2):
            if (i1,j1)==(k,l):
                ret=1
    return ret
```

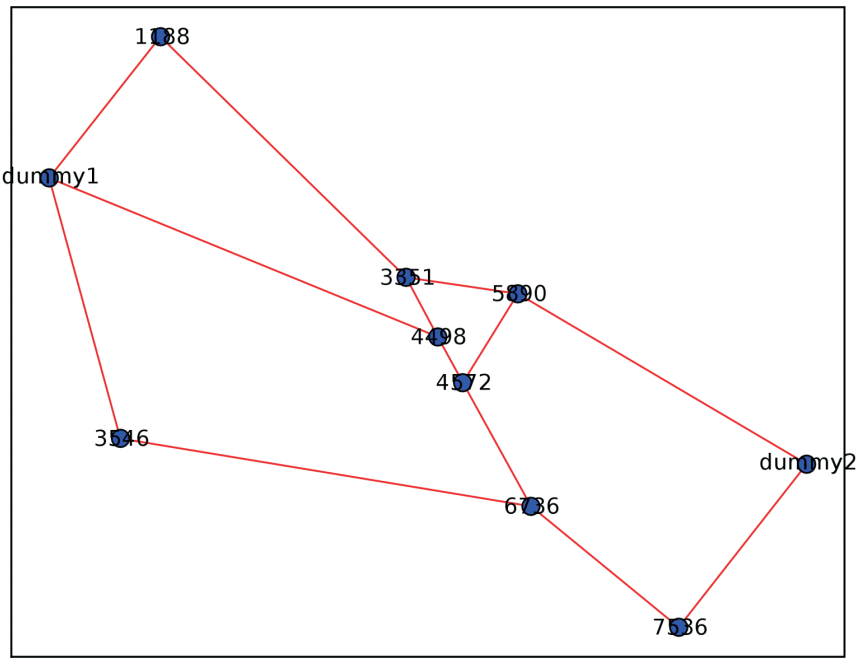

図 3.4: 縮約後のグラフ．

縮約後のグラフを図 3.4 に示す．始点まわりを縮約した点に隣接する点が経由点（transit node）になる．始点を含むバケット内の各点から，経由点への最短距離を保管しておく．また，終点まわりに対しても同様のデータを保管しておく．すると，クエリ時には，このグラフ上だけで探索を行えば良うことになり，経由点通過法とほぼ同じ程度の計算時間が期待できる．また，アクセスするメモリ量は，経由点通過法より小さくできるので，小メモリのPDAなどでも用いることができる．

第 4 章

経由点通過法

ここでは，経由点通過法（transit node routing）を紹介し，その実装について考える．用いるプログラミング言語はPythonであり，主にプロトタイピングの役割を果たす．

4.1 アルゴリズム概要

経由点通過法（transit node routing）は，Bast–Funke–Sanders–Schultes（未発表論文は `http://algo2.iti.uka.de/schult` からダウンロードできる）によって開発された最短路問題に対する前処理法であり，現在では最速のクエリ時間を達成している．ここでは，正方格子（バケット）を用いた方法について考える．

経由点通過法は，以下の観察に基づく．

> 始点から遠く離れた終点への最短路に行く際には，始点からある程度離れた中間地点を経由するが，その中間地点の数はそれほど多くはない．

この観察から，必ず通過する経由点を求めておき，経由点への最短距離と経由点間の最短距離を覚えておけば，最短路を高速に求めることができることが推測される．これを達成するためのアルゴリズムが，経由点通過法である．この方法のアイディアを図 4.1 を用いて解説する．

いま，中央の小さな四角（C と書かれたバケット）内の点から，大きな四角（outer）の外側へ行くことを考える．このとき，中央のバケット C と大きな四角の縁の点間の最短路をすべて計算し，それが中くらいの大きさの四角（inner）を通過する地点をすべて記憶しておく．これを C 内の点のアクセス点とよぶ．すべてのバケットに対して同じことを行うことによって得たアクセス点を合わせたものが経由点の集合になる．

より正確に記述すると，以下のようになる．

経由点通過法は，与えられた最短路問題の問題例（有向グラフ $G = (V, E)$，枝の距離を表す写像 $c : E \to \mathrm{R}^+$）に対して，以下の情報を前処理によって得る．

- 経由点集合 $\mathcal{T} \subseteq V$；遠くの点に行くときに必ず経由する点集合．おおよそ $O(\sqrt{n})$（$n = |V|$ は点の数）となるように決める．

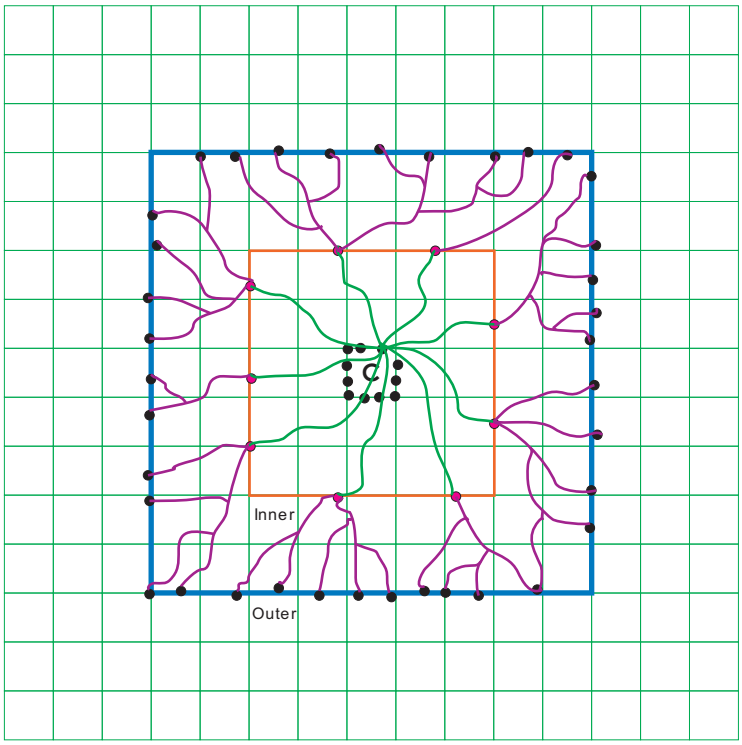

図 4.1: 経由点通過法のアイディア.

- アクセス写像 $A : V \to 2^{\mathcal{T}}$；各点を与えると経由点の部分集合（アクセス点）を返す関数.
- 局所フィルタ $L : V \times V \to \{1, 0\}$；2 つの点を与えたとき，それが近いか（1），遠いか（0）を返す関数.
- 経由点集合間の最短距離行列 d
- 各点とアクセス点間の距離（距離は対称を仮定）d

上の情報が与えられたとき，始点 s から終点 t への最短路を求めるクエリに対するアルゴリズムは，以下のように書ける.

経由点通過法におけるクエリ

```
1 if  L(s,t) = 0 then
2     最短距離 = min{d(s,u) + d(u,v) + d(v,t) : u ∈ A(s), v ∈ A(t)}
3 else  // s と t の間が十分に近い
4     s から t までの最短距離を通常の Dijkstra 法で求める.
```

経由点をより高速に求めるために，以下の sweep 法を用いる.

まず，縦のラインをまたぐ枝の端点（いずれの端点でも良いがここでは左側のバケットに含まれる点）を，経由点の候補 s とする．またぐ枝を求めるのは計算時間がかかるので，バケットの縁の点だけを考慮しても，バケットの大きさが十分に大きければ最適性は保証される.

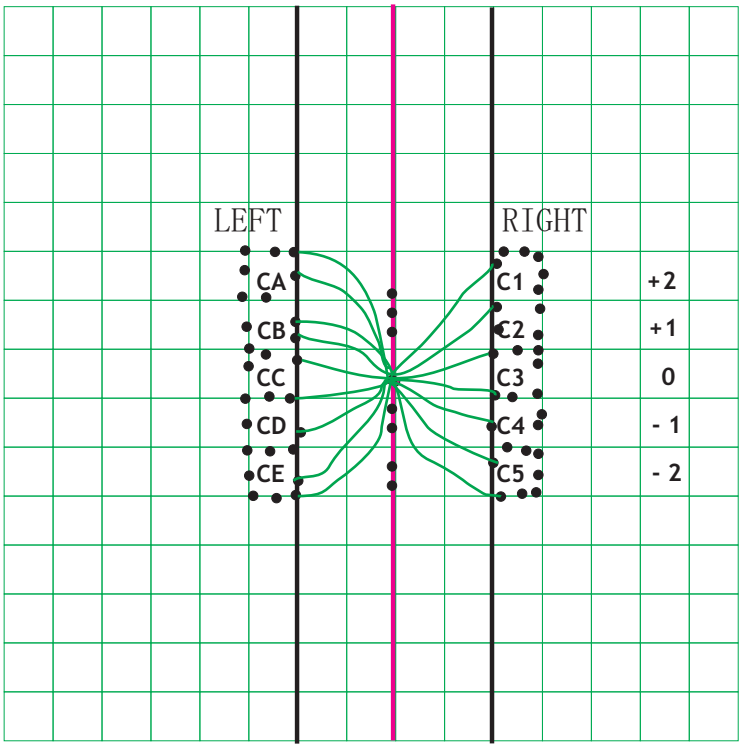

図 4.2: sweep 法の参考図 (1).

こラインより左に 2 セル，右に 2 セル離れたバケットを考え，上に 2 セル，下に 2 セル離れた領域を考える．この領域内で，経由点の候補 s から，左端の領域の縁の点 ℓ と右端の領域の縁の点 r への最短距離を計算する（図 4.2）．原論文では，バケットのすべての縁の点を ℓ, r の候補としているが，左側の線と交差する枝の端点だけで十分である．

ℓ と r 間の最短距離を更新していたら，その距離と点 s を保管する．これをすべてのラインをまたぐ枝の端点に対して計算する．保管された点 s が経由点となる（図 4.3）．これをすべての縦と横のラインに対して行うことによって経由点の集合を求めることができる．

sweep 法の高速化のために，バケット数を大きい方から順に計算する方法が提案されている．バケットの分割数を 2^K と 2 のべき乗とし，(たとえば) $K = 10$ から順に小さくしていくことを考える．この際，ターゲットとするバケット数（たとえば $K = 7$）における sweep するライン上の点のみを経由点の候補として，最短路問題を解く．K が大きいときの最短路は小さな範囲を対象とするので高速に解くことができ，候補は小さな K のライン上だけに限定して行う点がミソである．より大きな K で経由点になっていない点に対しては，候補の対象から外すことができるので，全体として少ない探索で，ターゲットとするバケット数の経由点の集合を求めることができる．図 4.4にバケット数が 1/4 になったときの，sweep 法の様子を示す．

しかし，小さな（K が大きいときの）バケットを跨ぐ枝が存在するので，ラインの前後の点を求める際には注意を要する．また，左右の縁の点に対しても，ターゲットとするバケットの大きさでの縁になっている必要があるので，左右のラインを跨ぐ枝の端点を用いないと，近似的な経由点になる可能性がある．

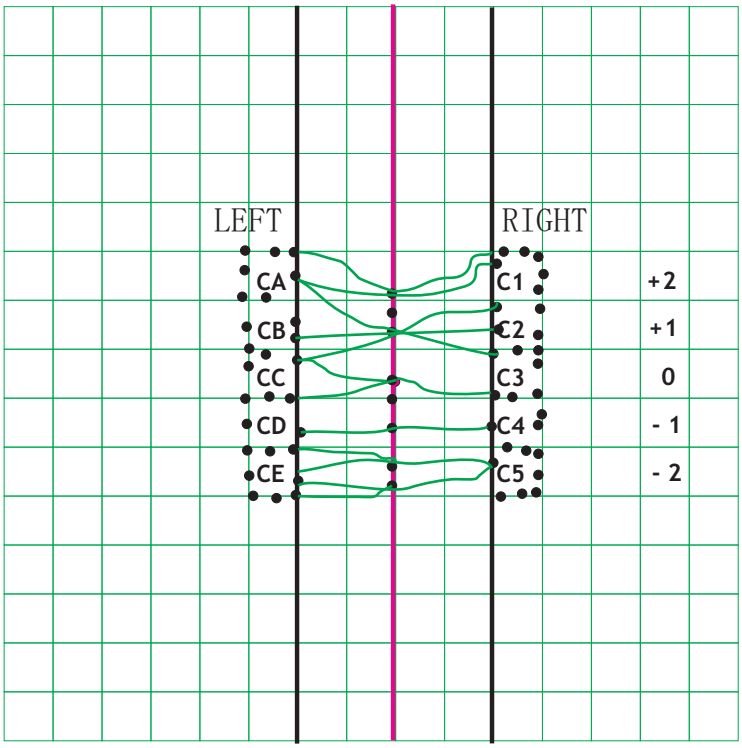

図 4.3: sweep 法の参考図 (2).

次に，アクセス点を求めるための方法について考える．

各経由点から Dijkstra 法の探索を行う．このとき，探索を途中で打ち切るために，被覆（cover）の概念を導入しておく．永久ラベルがついた点が被覆されているとは，その点と始点 s の最短路木のパス内に,（自分を含めて）少なくとも 1 つの経由点があることを指す．ヒープに入っていて（言い換えれば有限のラベルがついていて），その先行点が被覆されているとき，その点は被覆されていると定義する．経由点で，その先行点が被覆されていないものを被覆点（covering node）とよぶ．

これは，Dijkstra 法による探索の際に，経由点に永久ラベルがついたときに，被覆されたことを表すフラグをたて，被覆された点 i から出ている枝の先の点 j に対してラベルを更新したときに，点 j に被覆されたことを表すフラグをたてることによって求めることができる．探索は，ヒープに入っていて，被覆されていない点がなくなったときに，終了すれば良い．

探索後に最短路木に入っていて（つまり永久ラベルがついていて），被覆されていない点に対しては，その点のアクセス点を，始点の経由点と設定し，距離を記憶する．また，被覆点に対しては，始点からその点までの距離を記憶する．

この操作をすべての経由点を始点として行うことによって，経由点間の最短距離を計算すると同時に，アクセス点に関する情報を得ることができる．

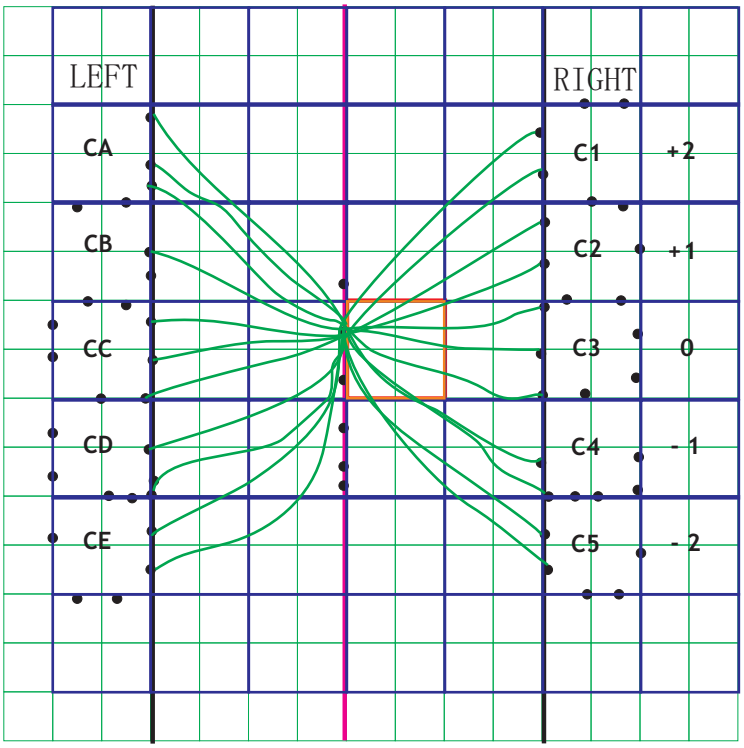

図 4.4: sweep 法の参考図 (3).

4.2 Python による実装と実験

2次元ビットベクトル法と同様に，モジュールと，部品となる関数を記述したモジュールを読み込む．

最短路問題は，Dial 法を用いて求解する．グラフの一部分だけの最短路を解くために，永久ラベルがついたか否かを表す配列 settled を準備しておく．

```
filename="DC.tmp"
G,n,m,C = ReadDimacs(filename)
print "number of nodes in the largest connected componet=",n,C

settled= numpy.ones(n,numpy.int0) #=1 if node is settled (or has an eternal label or is scanned)
reached= numpy.zeros(n,numpy.int0) #=1 if node is heap or list (or has a finite potential)
dist= numpy.zeros(n,numpy.int0)
prev= numpy.zeros(n,numpy.int0)
```

その後で，やはり2次元ビットベクトル法と同様に，バケットを準備する．異なる点は，バケットの縁の点を，右端，左端，上端，下端の4つに分けて保持することである．右端を borderE（East の意味），左端を BorderW（West の意味，上端を borderN（North の意味），下端を borderS（South の意味）とし，点のリストとして保持する．その後で，その和集合を縁の点集合 border として，set() 関数で重複を除いて生成する．

```
n_bucket=int(sqrt(n/50))
bucket=[[[] for i in range(n_bucket)] for j in range(n_bucket)]
borderE=[[[] for i in range(n_bucket)] for j in range(n_bucket)] #右端
borderW=[[[] for i in range(n_bucket)] for j in range(n_bucket)] #左端
borderN=[[[] for i in range(n_bucket)] for j in range(n_bucket)] #上端
borderS=[[[] for i in range(n_bucket)] for j in range(n_bucket)] #下端
border=[[[] for i in range(n_bucket)] for j in range(n_bucket)]
```

```
G.bucket={}
for node in G.nodes_iter():
    (x,y)=G.position[node]
    i=int(x*n_bucket)
    j=int(y*n_bucket)
    bucket[i][j].append(node)
    G.bucket[node]=(i,j)  #bucket number is stored

for i in range(n_bucket):
    for j in range(n_bucket):
        if len(bucket[i][j])>0:
            for node in bucket[i][j]:
                for n1 in G.neighbors(node):
                    if G.bucket[node] <> G.bucket[n1]:
                        (i1,j1)=G.bucket[n1]
                        if i1>=i+1:
                            #right (east) border
                            borderE[i][j].append(node)
                            break
                        if i1<=i-1:
                            #left (west) border
                            borderW[i][j].append(node)
                            break
                        if j1>=j+1:
                            #upper (north) border
                            borderN[i][j].append(node)
                            break
                        if j1<=j-1:
                            #lower (south) border
                            borderS[i][j].append(node)
                            break

for i in range(n_bucket):
    for j in range(n_bucket):
        border[i][j]=set(borderE[i][j]+borderW[i][j]+borderN[i][j]+borderS[i][j])
```

続いて経由点を求める．経由点から左の点集合（Left）と右の点集合（Right）までの最短距離を計算する．経由点の候補集合 Candidateは，対象とするバケット (i,j) の右の縁（borderE）とし，縦の線上にある点に対してsweep操作を行う．同様に，横の線上にある点に対してもsweep操作を行う必要があるが，ここでは省略する．

```
TransitAll=set([])
for i in range(2,n_bucket-3):
    Dic={} #prepare a dictionary (key=(left,right), vaue=(min distance,transit node))
    for j in range(0,n_bucket):
        Candidate=borderE[i][j]
        Left=[]
        Right=[]
        for j2 in range(j-2,j+3):
            if j2>=0 and j2<n_bucket:
                for t in borderE[i-2][j2]:
                    Left.append(t)
        for j2 in range(j-2,j+3):
            if j2>=0 and j2<n_bucket:
                for t in borderW[i+3][j2]:
                    Right.append(t)
        if len(Candidate)>0 and len(Left)>0 and len(Right)>0:
            print "finding transit point in bucket",i,j
            Nodes=Left+Right
            for i2 in range(i-1,i+3):
                for j2 in range(j-2,j+3):
                    if i2>=0 and i2<n_bucket and j2>=0 and j2<n_bucket:
```

```
                Nodes+=bucket[i2][j2]
        #print "unsettled nodes=",Nodes

        for s in Candidate:
            source=s
            sink=-1
            for v in Nodes:
                settled[v]=0
                reached[v]=0
                dist[v]=999999
                prev[v]=0

            DialSub(n,G,settled,reached,dist,prev,C,source,sink)
            print "Finding shortest path from ",source
            #for v in Nodes:
            #    print dist[v]

            for l in Left:
                for r in Right:
                    if (l,r) in Dic:
                        dummy=dist[l]+dist[r]
                        (value,v)=Dic[(l,r)]
                        if dummy <value:
                            Dic[(l,r)]=(dummy,s)
                    else:
                        dummy=dist[l]+dist[r]
                        if dummy<999999:
                            Dic[(l,r)]=(dummy,s)

    for (value,v) in Dic.itervalues():
        TransitAll.add(v)
```

特定のバケットに対する左の点集合，右の点集合，探索する点集合，経由点の候補，ならびに選択された経由点（大きい丸）を表示する．

図 4.5 に左の点集合，右の点集合，経由点の候補と経由点を，図 4.6 に経由点と候補だけを拡大したものを示す．

横側のsweep線に対しても同様の操作を行うことにより，経由点の集合を得ることができる（図 4.7）．この例では，136 点の経由点が抽出されている．

次に，与えられた経由点集合に対して，経由点を始点としたDijkstra探索を行い，すべてのヒープ内の点が被覆されたら終了するプログラムを示す．これによって，アクセス点と経由点間の最短距離を得ることができる．

```
def DialSubForAccess(n,G,settled,reached,covered,transit,dist,prev,C,source,sink):
    ''' Dial method subroutine
     for finding access nodes transit'''
    circular=[[-1] for i in range(C) ] #-1 represents the list is empty
    circular[0].append(source)
    reached[source]=1
    prev[source]=-1
    current=0
    num_uncovered=1
    covered[source]=0

    while 1:

        first=current
        while 1:
            v=circular[current][-1] #v is the node with minimum potential
            if v>=0:
```

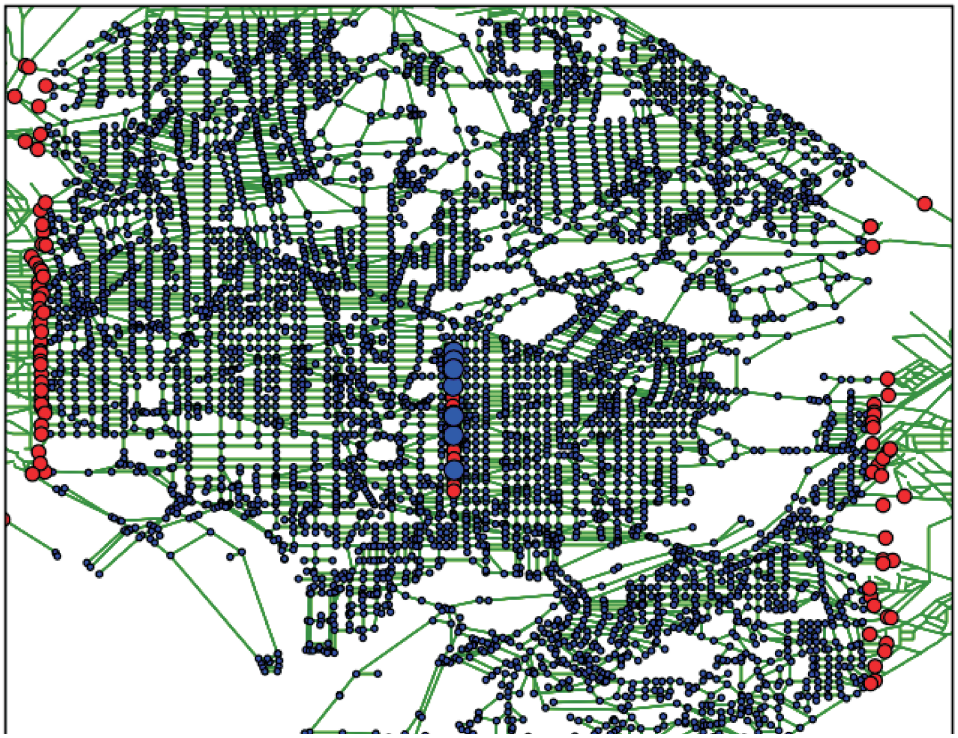

図 4.5: 左の点集合，右の点集合，探索する点集合，経由点の候補，ならびに選択された経由点（大きい丸）.

```
            break
        #the current list is empty, so we increment it
        current+=1
        current= current%C
        if current==first:
            #the circular list is empty, so we quit
            v=-1
            break

    if v==-1: #the circular list is empty, so we quit
        break
    circular[current].pop()
    settled[v]=1

    if transit[v]==1:
        if v !=source and covered[v]==0:
            covered[v]=1
            num_uncovered-=1
    else:
        if covered[v]==0:
            num_uncovered -=1

    if num_uncovered==1 and v !=source:
        break #all the nodes in the heap are covered

    for w in G.neighbors(v): #scan operation
        if settled[w]==0:
            temp=dist[v]+G.get_edge(v,w)
```

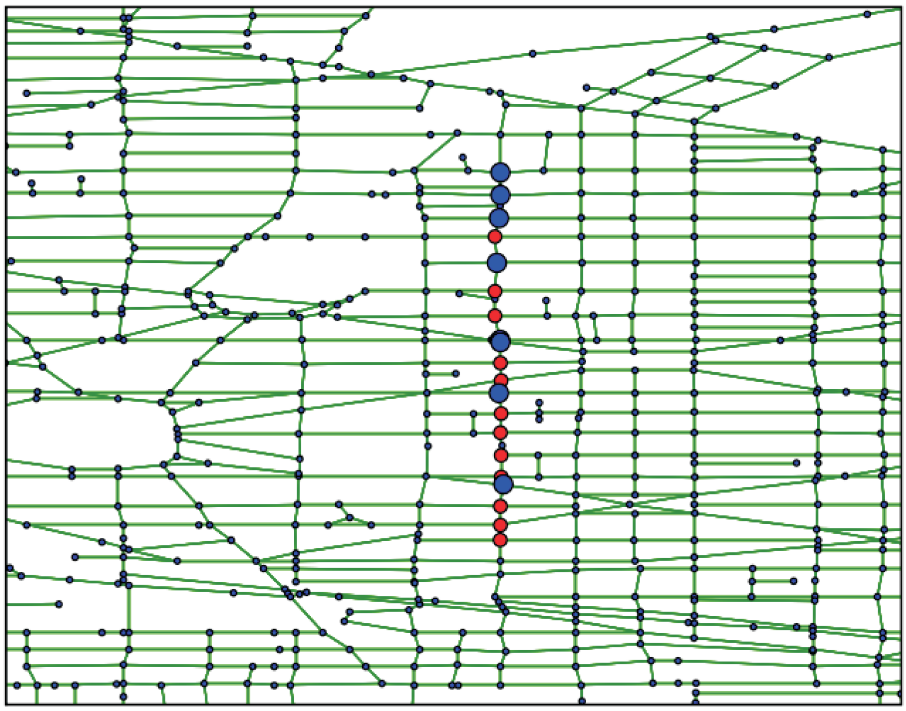

図 4.6: 経由点の候補と経由点（大きい丸）.

```
if reached[w]== 0: #w is not in the circular list
    reached[w]=1
    dist[w]=temp
    prev[w]=v
    circular[temp%C].append(w)

    if covered[v]==1:
        covered[w]=1
    else:
        num_uncovered+=1

else: #w is in the circular list
    if temp<dist[w]:
        circular[dist[w]%C].remove(w)
        circular[temp%C].append(w)
        dist[w]=temp
        prev[w]=v

        if covered[v]==1:
            if covered[w]==0:
                covered[w]=1
                num_uncovered-=1
        else:
            if covered[w]==1:
                covered[w]=0
                num_uncovered+=1
```

図 4.8 にある経由点からの Dijkstra 探索の結果を示す．グラフの一部分だけの探索で，アクセス点，被覆する

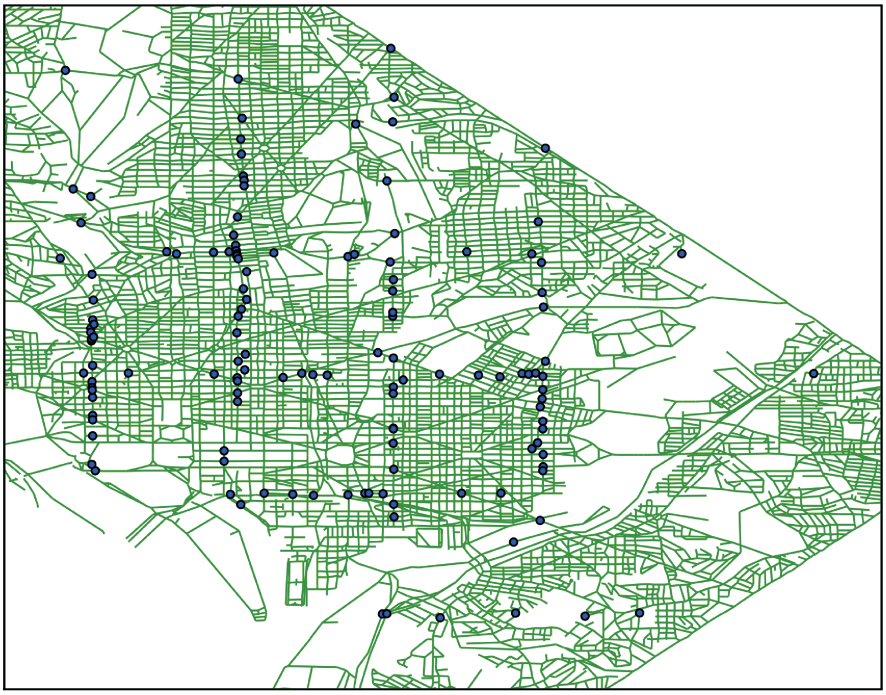

図 4.7: 経由点の集合.

経由点までの距離を求めることができる.

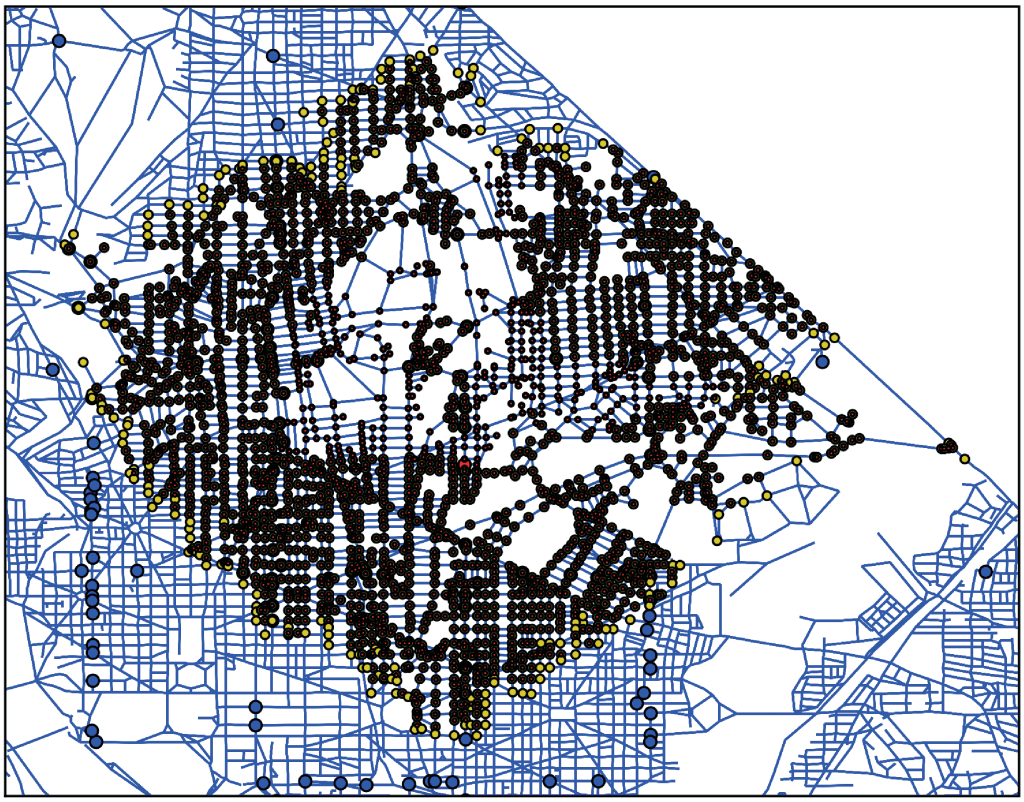

図 4.8: 中央の赤い点からの Dijkstra 探索．大きい点が経由点．塗りつぶされた点が永久ラベルが付いた点，中くらいの点が被覆された点．

第 5 章

メモリ階層構造を考慮した高速化

この章ではメモリの階層構造を利用したダイスクトラ法の高速化について考える。現在の代表的な PC 用の CPU である Intel Core2 系の階層的なメモリモデルは図 5.1 になる。カーナビ用の組み込む系 CPU は、常に PC 用の CPU と比較してキャッシュ&メモリ容量、メモリ ⇔ キャッシュ間のレイテンシ (データ転送を要求してから、実際にデータが転送されてくるまでの遅延時間) やメモリのバンド幅などが劣る傾向がある (PC の方が、予算、スペース、消費電力などの点で余裕があるため)。今回の提案内容は以下の特徴を持っている

1. ダイクストラ法のみに限定されずに他の最適化手法に使用できる。すでに GotoBLAS[1] の GEMM(行列積) などに採用されている。

2. 詳しくは後述するが、モデルはレジスタ、L1(1 次) & L2(2 次) キャッシュ、メモリに限定して考えるので、特定の CPU に依存しない汎用的モデルである。各容量や転送速度はパラメータとして処理できる。

5.1 メモリの階層構造のモデル化

すでに述べたように、代表的なメモリの階層構造は図 5.1になる。この他にも L3(3 次) キャッシュなどを有したり、メモリと HDD 間に SSD(フラッシュメモリ等のソリッドステートで作られた補助記憶装置) が入る場合もある。一般には以下のような性質を持っている。

1. レジスタ上にデータが存在する場合が最も高速で、レジスタ上にデータが無い場合では L1 データキャッシュ ⇒ L2 キャッシュ ⇒ メモリ ⇒ HDD の順番でデータを探しに行く。

2. 概念的に捉えるとレジスタへのアクセスは**瞬時**, L1 キャッシュは **超高速**, L2 キャッシュは **高速**, メモリ は **低速**, HDD は **問題外** になる。しかし、レジスタは 64bit か 128bit 長が数十個が使用できるのみであり、L1 データキャッシュも数十 kbyte であることから、多くの場合ではレジスタ ⇔ L1 キャッシュ間のみで計算を行うことは不可能である。

[1] http://www.tacc.utexas.edu/resources/software/

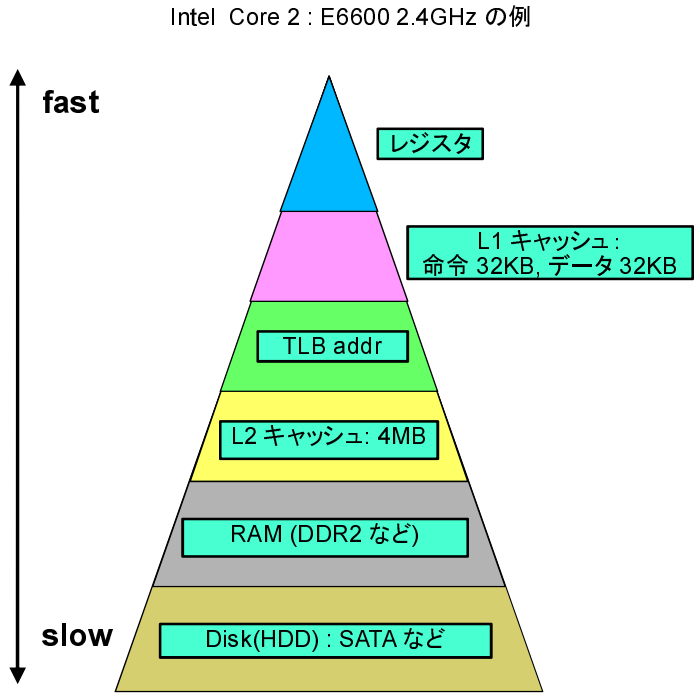

図 5.1: 階層的なメモリモデル

3. L2 キャッシュは現在では 1MB から 4MB 程度であるので (複数のコアで共有されている場合も多い)、ほとんどの計算をレジスタ ⇔ L2 キャッシュ間で行うのが理想的である。

4. その他に TLB (Translation Lookaside Buffer : CPU 内のキャッシュであり、仮想アドレスから物理アドレスへの変換の高速化を図るもの) などもあるが、複雑なモデルになるので今回は省略する。

現在は PC では平均的なメモリ量は 1 〜 4Gbyte であるので (組込み系でも数百 Mbyte)、前処理ではなく第二段階の query においては全データがメモリ上にあると仮定することもできる。しかし L2 キャッシュは 1 プロセスでは多くても全容量の半分ぐらいしか使用できないので全データを L2 キャッシュに収納することは不可能であるという仮定を置いてもよい。

5.2 L2 キャッシュの有効利用によるダイクストラ法の高速化

前節で述べたように query において全データはメモリ上にあるが、L2 キャッシュには一度に収容できないという仮定を置こう。このときダイクストラ法の高速化の基本方針は以下の通りである。

1. 始点と終点が決定したら、次に訪れる可能性があるバケットのデータを先読みしてメモリから L2 キャッシュに移動させる (図 5.2)。ここでいうバケットはイメージ図なので実際には現在地点と終点の方向から次に訪れる部分グラフのデータを特定できれば、他のデータ構造でも可能。

2. L2 キャッシュからメモリへの移動と探索の関係について

   (a) 理想的な状態：ロード&ストア命令と整数や比較演算が同時並行的に行われる。ロード&ストア命令の実行には比較演算等よりも多くのクロック数を要するので、両者のクロック数の比を考慮して命令を交互に発行する。例えば探索が進んできたら、少し先のグラフのデータを L2 キャッシュに入れる命令を先に発行してから探索を行う。探索が終わった時点ですでに次のデータの L2 キャッシュへのロードが終わっているのが望ましい。さらに使用するデータの頻度によってブロックサイズを変えて、より頻繁に使用されるデータは L1 キャッシュに入れるのがさらに理想的である。

   (b) 悪い状態：頻繁にロード&ストア命令が発行されて、しかもこれらの命令が終了するまでの間、探索は中断される。

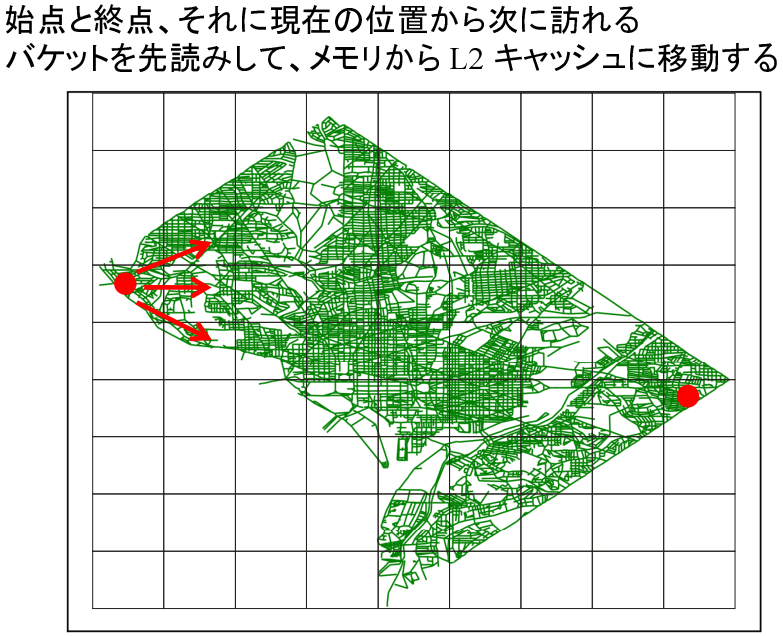

図 5.2: データの先読みと移動

この方法の注意点を以下に記す。

1. L2 キャッシュにデータを入れるときには古いデータを消去する必要がある。非ダーティな (値が変更されていないのでメモリに書き出す必要がない) データはそのまま消去して構わないが、ダーティなデータは消去する前にメモリに書き出す必要がある。これは非常にクロック数を要する作業である。あるいはダーティな部分は触らない方法もある。しかし、他のプロセスによって強制的に書き出される可能性もあるので注意が必要。

2. L2 キャッシュ上に新しくロードしたデータを固定する

3. L2 キャッシュへのロードにはデータをブロッキングする、あるいはレジスタ経由で変数をコピーするなどの方法がある。基本的には C/C++ 言語でも部分的にはアセンブラで実装した方が良い。この場合 128bit の SSE 命令用の XMM レジスタにロードすれば同時に図 5.3のように SIMD 命令による高速化もできる

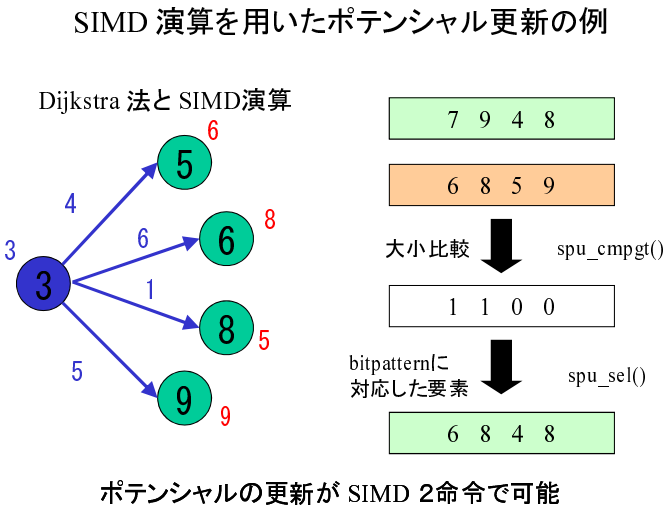

図 5.3: simd 演算による高速化の例

5.3 最適化の基本方針

宮本による Dial 法の C 言語による実装に対し, メモリ階層構造を考慮した最適化を行ったものを紹介する. 最適化前と比べ実行時間は, 環境にもよるが半分程となっている. 基本的な方針としては 2 点ある.

1. 同一の ID を参照する頻度が高いデータは単に複数の配列ではなく, それらをまとめてパッキングし, 構造体の配列として保持する

2. 呼出回数の多い関数は `static inline` 修飾子により, 呼出側に `inline` 展開する

(1) より, メモリアクセスのオーバーヘッドを軽減することができる. また, 実行の際に不必要であると思われるデータ (`tail[]`) はここに含めないようにすることで, より効果を得られる. 構造体 `graph_t` 内では, 新たに構造体 `arc_t` を用いて `head` と `length` をパッキングしている. (2) より, 関数呼び出しにかかってしまうオーバーヘッドを削減することができる.

5.4 最適化に伴うサブルーチン群の変更点

まず, グラフを forward-star で保持するための構造体 `graph_t` について, 宮本による DIAL 法と比較する形で見ていく. また, グラフ読み込み時のみに必要な `tail[]` はここに含めていないが, グラフ読み込みサブルーチ

ン内で動的に確保・開放することとする.

・宮本による DAIL 法でのグラフデータを格納する構造体

```
typedef struct {
      int n;
      int m;
      int *point;
      int *tail;
      int *head;
      int *length;
} FSTAR;
```

・最適化後の DAIL 法でのグラフデータを格納する構造体

```
typedef struct {
    int head;
    int length;
} arc_t;

typedef struct {
    int    node_num;
    int    arc_num;
    int    max_length; // バケットサイズ
    int   *point;
    arc_t *arc;        // head, length をパッキング
} graph_t;
```

データ保持方法の変更に伴い, 様々なサブルーチン群には若干の変更がある. クイックソート・サブルーチン QuickSortTail() ・ DIMACS フォーマットグラフを読み込むサブルーチン ReadDimacsGraph() はそれぞれ以下のように変更されるが, 本質的には変わっていない. head[], length[] がそれぞれ arc[].head, arc[].length に置き換わっているところに注意されたい.

```
void QuickSortTail (int left, int right, int min, int max, arc_t *arc, int *tail);
{
    int i, j;
    int shaft, tmp;
    arc_t arc_tmp;

    if (left >= right || min >= max) return;
    i     = left;
    j     = right;
    shaft = min + (max - min)/2;

    while (i < j) {
        if (tail[i] <= shaft) i++;
        else {
            tmp     = tail[j];
            tail[j] = tail[i];
            tail[i] = tmp;
            arc_tmp = arc[j];
            arc[j]  = arc[i];
            arc[i]  = arc_tmp;
            j--;
        }
    }
    if (tail[i] <= shaft) j++;
    else i--;

    QuickSortTail(left,  i,   min, shaft, arc, tail);
    QuickSortTail(j ,right, shaft+1, max, arc, tail);
}
```

```
void ReadDimacsGraph (graph_t *graph, char *graph_file)
{
    int  m, a, t, n;
    char line[1<<7];
    FILE *graph_fp;
    int  *tail = NULL;

    if ((graph_fp = fopen(graph_file, "r")) == NULL) {
      fprintf(stderr, "Can't open this Graph file ! : \"%s\" \n", graph_file);
      exit(1);
    }

    // パラメータを読み込む
    m  = 0;
    graph->max_length = 0;
    while (fgets(line, 1<<7, graph_fp)) {
        if (strncmp(line, "a ", 2) == 0) {
            sscanf(&line[2], "%d %d %d",
                   &tail[m], &(graph->arc[m].head), &(graph->arc[m].length));
            if (graph->max_length < graph->arc[m].length){
                graph->max_length = graph->arc[m].length;
            }
            m++;
        }
        else if (strncmp(line, "p sp ", 5) == 0) {
            sscanf(&line[5], "%d %d", &(graph->node_num), &(graph->arc_num));

            // グラフを格納する領域の動的確保
            graph->point = (int *)  malloc(sizeof(int)   * (graph->node_num+2));
            graph->arc   = (arc_t *)malloc(sizeof(arc_t) * (graph->arc_num   ));
            tail         = (int *)  malloc(sizeof(int)   * (graph->arc_num   ));
            if (graph->point == NULL || graph->arc == NULL || tail == NULL) {
                fprintf(stderr, "memory cannot alloc ! : GraphMalloc()\n");
                exit(1);
            }
        }
    }

    // forward-star のために tail 順にソート
    QuickSortTail (1, graph->arc_num - 1, 1, graph->node_num, graph->arc, tail);

    // forward-star を構成する
    for (a = t = 0; a < graph->arc_num; a++) {
        if (t < tail[a]) {
            for (n = t + 1; n <= tail[a]; n++) {
                graph->point[n] = a;
            }
            t = tail[a];
        }
    }
    if (t < graph->node_num+1) {
        for (n = t + 1; n < graph->node_num+2; n++)
            graph->point[n] = graph->arc_num;
    }

    // これ以後 tail[] は用いないため，メモリ開放
    if (tail != NULL) {
        free(tail);
        tail = NULL;
    }
    fclose(graph_fp);
}
```

5.5 最適化後のダイクストラ法

ダイクストラ法を実行するためには構造体 sp_run_t を必要とする．ここでもグラフデータを保持している構造体 graph_t の arc_t と同様の実装方法をこれらにも適用しており, list_t, node_t はそれぞれ構造体の配列である．list_t は バケット用の両方向リスト prev と next を, node_t は 各点におけるポテンシャル potential と, 最短路を求めるために必要な直前点を保持する prev_node をそれぞれパッキングしている．bucket[] と list[] は Dail 法のための, node[] 構造体はダイクストラ法のための作業領域である．

```
typedef struct {
    int prev;
    int next;
} list_t;

typedef struct {
    int potential; // 各点における最短距離を保持する配列
    int prev_node; // 各点における直前点，たどることで最短路を求める
} node_t;

typedef struct {
    int    *bucket; // バケット
    list_t *list;   // 両方向リスト用
    node_t *node;   // 最短距離・直前点を保持する
} sp_run_t;
```

次に 1 レベルバケットを用いたダイクストラ法の実装例を示す．呼出回数の多い BucketInsertNode(), BucketDeleteNode(), BucketGetMinNode() ルーチンは, いずれも static inline 修飾子により, 呼出側である DijkstraWithBucket() へ inline 展開される．また, while で記述された部分を do while へ変更することで, 1 回分の BucketGetMinNode() を削減することができ, 可視性も向上している．

```
// バケット bucket にポテンシャル pt、点 id の点を挿入
static inline void BucketInsertNode (int *bucket, list_t *list, int pt, int id)
{
    list[id].next = bucket[pt];
    list[id].prev = -1;
    if (bucket[pt] >= 0) list[bucket[pt]].prev = id;
    bucket[pt] = id;
}


// バケット bucket にポテンシャル pt、点 id の点を削除
static inline void BucketDeleteNode (int *bucket, list_t *list, int pt, int id)
{
    if (list[id].next > 0) list[list[id].next].prev = list[id].prev;

    if (list[id].prev > 0) list[list[id].prev].next = list[id].next;
    else bucket[pt] = list[id].next;
}


// バケット内にある最小ポテンシャル点 ID を返す．
// バケットが空の場合には，-1 を返す．
static inline
int BucketGetMinNode (int *bucket, list_t *list, int *prev_bucket, int size)
{
    int pt, tail;
```

```
    pt = *prev_bucket;
    do {
        if (bucket[pt] > 0) {
        tail = bucket[pt];
        BucketDeleteNode (bucket, list, pt, tail);
        *prev_bucket = pt;
        return tail;
        }
        pt = (pt + 1) % size;
    } while (pt != *prev_bucket);

    return -1;
}


// single-source type にも target を -1 とすることで対応できる
int DijkstraWithBucket (graph_t *graph, int *bucket, list_t *list,
                        node_t  *node,  int  source, int  target)
{
    int i, tail_node, prev_bucket;

    int    size  = graph->max_length;
    int   *point = graph->point;
    arc_t *arc   = graph->arc;

    // すべての点のポテンシャル・直前ノードを, (-1) に初期化する
    for (i = 1; i <= graph->node_num; i++) {
        node[i].potential = node[i].prev_node = -1;
    }
    // すべてのバケットを空 (-1) に初期化する
    for (i = 0; i < size; i++) bucket[i] = -1;

    // 始点を source に設定し, バケット ID を 0 に初期化する
    tail_node              = source;
    node[source].potential = 0;
    prev_bucket            = 0;

    // 探索点 tail_node が終点 target になるか、バケットが空になるまで繰り返す
    do {
        for (i = point[tail_node]; i < point[tail_node+1]; i++) {
            // バケットに格納されていない
            if (node[arc[i].head].potential == -1) {
                node[arc[i].head].potential = node[tail_node].potential + arc[i].length;
                node[arc[i].head].prev_node  = tail_node;
                BucketInsertNode(bucket, list, node[arc[i].head].potential%size, arc[i].head);
            }
            // バケットに格納されている
            else if (node[tail_node].potential + arc[i].length < node[arc[i].head].potential) {
                BucketDeleteNode(bucket, list, node[arc[i].head].potential%size, arc[i].head);
                node[arc[i].head].potential = node[tail_node].potential + arc[i].length;
                node[arc[i].head].prev_node  = tail_node;
                BucketInsertNode(bucket, list, node[arc[i].head].potential%size, arc[i].head);
            }
        }

        // バケットから最小ノードを取り出す
        tail_node = BucketGetMinNode (bucket, list, &prev_bucket, size);
    } while (tail_node != -1 && tail_node != target);

    // p2p : 終点でのポテンシャルを返す
    // ss  : 0 を返す
    return (target != -1) ? node[target].potential : 0;
}
```

以上のサブルーチンを呼び出すメインルーチンは以下のように実装できる.

```
/*============================================================
  main function
  ============================================================*/
int main (int argc, char *argv[])
{
  int q;
  graph_t  graph;
  query_t  query;
  sp_run_t sp_run;

  // for profile
  char *graph_file, *query_file, *report_file;
  double time, read_time, run_time;
  long   distance, check_sum;

  if (argc < 3 || 4 < argc) { // 3 or 4
    printf("Usage: ./sp.xxx graph_file query_file report_file\n");
    printf("   ex) ./sp.xxx USA.gr     USA.p2p    USA.report\n");
    printf("       graph_file  : dimacs graph format\n");
    printf("       query_file  : point-to-point type [.p2p]\n");
    printf("                     single-source  type [.ss]\n");
    exit(1);
  }

  graph_file  = argv[1];
  query_file  = argv[2];
  report_file = (argc == 4) ? argv[3] : NULL;

  printf("============================================================\n");
  printf("shortest path route solver\n");
  printf("============================================================\n");
  printf("algorithm        is dijkstra's algorithm & 1-level bucket\n");
  printf("graph  file      is %s\n", graph_file);
  printf("query  file      is %s\n", query_file);
  printf("report file      is %s\n", report_file);


  // DIMACS format のクエリを読み込む
  ReadDimacsQuery (&query, query_file);

  printf("query            is ");
  if (query.target[0] == -1) printf("single-source x %d\n",  query.query_num);
  else                       printf("point-to-point x %d\n", query.query_num);


  // DIMACS format のグラフを読み込む：読み込み時間は read_time
  read_time = GetRusageSec();
  ReadDimacsGraph (&graph, graph_file);
  read_time = GetRusageSec() - read_time;

  printf("node / arc       is %d / %d\n", graph.node_num, graph.arc_num);
  printf("max length       is %d\n\n", graph.max_length);


  // ダイクストラ法を実行するために必要な作業領域の動的確保
  sp_run.bucket = (int *)     malloc(sizeof(int)      * (graph.max_length+1));
  sp_run.list   = (list_t *)  malloc(sizeof(list_t)   * (graph.node_num+1));
  sp_run.node   = (node_t *)  malloc(sizeof(node_t)   * (graph.node_num+2));
  if (sp_run.bucket == NULL || sp_run.list == NULL || sp_run.node == NULL) {
    fprintf(stderr, "memory cannot alloc ! : sp_run\n");
    exit(1);
  }


  // 距離・実行時間の合計値のための初期化
  check_sum = 0;
```

```
  run_time  = 0.0f;

  printf(" id |  source  |  target  | distance | time[ms]\n");
  printf("----+----------+----------+----------+---------\n");

  // クエリの数だけ、ダイクストラ法を繰り返す
  for (q = 0; q < query.query_num; q++) {
    time = GetRusageSec();
    distance
      = DijkstraWithBucket (&graph, sp_run.bucket, sp_run.list, sp_run.node,
                            query.source[q], query.target[q]);
    time = GetRusageSec() - time;

    // クエリ番号・始点・終点・距離・実行時間を表示
    printf("%3d | %8d | ", q+1, query.source[q]);
    if (query.target[q] == -1) printf("   --    |    --    | ");
    else {
      printf("%8d | ", query.target[q]);
      if (distance == -1) printf("infinity | ");
      else printf("%8ld | ", distance);
    }
    printf("%7.1f \n", time * 1e3);

    // ss type での最短距離・最短経路、p2p type での最短経路を表示
    DisplayReport (&graph, sp_run.node, query.source[q], query.target[q], report_file);

    // 距離・実行時間を合計する
    check_sum += distance;
    run_time  += time;
  }


  // １クエリあたりの距離(平均)・グラフ読み込み時間・総計算時間を表示
  printf("\n");
  printf("average distance is %ld\n", check_sum/query.query_num);
  printf("file read time   is %7.1f[ms]\n", read_time * 1e3);
  printf("computing time   is %7.1f[ms]\n\n", run_time * 1e3);


  // 動的に確保していたメモリを開放
  if (query.source != NULL) {
    free(query.source);
    query.source = NULL;
  }
  if (query.target != NULL) {
    free(query.target);
    query.target = NULL;
  }

  if (graph.arc != NULL) {
    free(graph.arc);
    graph.arc = NULL;
  }
  if (sp_run.node != NULL) {
    free(sp_run.node);
    sp_run.node = NULL;
  }
  if (sp_run.bucket != NULL) {
    free(sp_run.bucket);
    sp_run.bucket = NULL;
  }

  return 0;
}
```

メインルーチン内では GetRusageSec() サブルーチンで実行時間を計測している. ここで得られる実行時間の

の単位は秒なので, msec を得たい場合には 1000 倍すればよい.

```
double GetRusageSec (void)
{
  struct rusage t;
  getrusage(RUSAGE_SELF, &t);
  return (double)(t.ru_utime.tv_sec + t.ru_utime.tv_usec * 1e-6);
}
```

以下のように使用する.

```
double time;
time = GetRusageSec();
/* 計測したい処理 */
time = GetRusageSec() - time;
```

5.6 DIMACS クエリの実行方法

クエリファイルは DIMACS フォーマットに準拠している必要があり, 以下の 2 種類のクエリに対応している.

- single-source type (以下, ss タイプ)

  1 つの始点から, それ以外の点への 1 対全最短路問題. 最短距離・最短路木が出力される.

- point-to-point type (以下, p2p タイプ)

  1 つの始点から, 1 つの終点までの 1 対 1 最短路問題. 最短距離・最短路が出力される.

例として, ss 用クエリファイル sample.ss と, p2p 用クエリファイル sample.p2p を挙げる.

```
[sample.ss]
c 9th DIMACS Implementation Challenge: Shortest Paths
c http://www.dis.uniroma1.it/~challenge9
c Sample single-source problem specification file
c
p aux sp ss 3
c contains 3 source lines
c
s 1
s 3
s 5

[sample.p2p]
c 9th DIMACS Implementation Challenge: Shortest Paths
c http://www.dis.uniroma1.it/~challenge9
c Sample point-to-point problem specification file
c
p aux sp p2p 3
c contains 3 query pairs
c
q 1 5
q 5 1
q 1 2
```

これら DIMACS フォーマットは, パラメータ行, 始点・終点, コメント行のより構成される.

1. p から始まる行はパラメータ行である. ただし, s や q から始まる行より前に記述しなければならない.

   ss タイプ　　:　p aux sp ss クエリ数
   p2p タイプ　:　p aux sp p2p クエリ数

2. s から始まる行は ss タイプ用の始点情報行である.

   `s` 始点 `ID`

3. q から始まる行は p2p タイプ用の始点・終点情報行である.

   `q` 始点 `ID` 終点 `ID`

4. c から始まる行はコメント行であり, c から後は無視される. 任意の行に挿入することができる.

   `c` コメント

DIMACS フォーマットのクエリファイルは次のようなサブルーチンにより、構造体 `query_t` へと格納される. また, クエリが ss タイプの際には終点 target を -1 とすることで, p2p タイプと ss タイプを同一のダイクストラ法・サブルーチン `DijkstraWithBucket()` で処理できるようになる. p2p タイプと ss タイプを混ぜて記述することも可能である (DIMACS フォーマットの拡張).

```
[sample.que]
c 9th DIMACS Implementation Challenge: Shortest Paths
c http://www.dis.uniroma1.it/~challenge9
c
c Sample single-source problem specification file
c Sample point-to-point problem specification file
c
p aux sp ss 6
c contains 6 query lines
c
s 1
q 1 5
s 3
q 5 1
s 5
q 1 2
```

上位のようなクエリを読み込むサブルーチン `ReadDimacsQuery()` は, 以下のように書ける. forward-star のためのクイックソート・サブルーチン `QuickSortTail()` が必要となる.

```
void ReadDimacsQuery (query_t *query, char *query_file)
{
    int i;
    FILE *query_fp;
    char line[1<<7];

    // p2p ファイルの読み込み
    if ((query_fp = fopen(query_file, "r")) == NULL) {
        fprintf(stderr, "Can't open this Query file ! : \"%s\" \n", query_file);
        exit(1);
    }

    // パラメータ行の読み込み
    while (fgets(line, 1<<7, query_fp)) {
        if (strncmp(line, "p aux sp ss ", 12) == 0) {
            sscanf(&line[12], "%d", &(query->query_num));
            break;
```

```
        }
        else if (strncmp(line, "p aux sp p2p ", 13) == 0) {
            sscanf(&line[13], "%d", &(query->query_num));
            break;
        }
    }

    // クエリ数の分だけ動的確保
    query->source = (int *)malloc(sizeof(int) * query->query_num);
    query->target = (int *)malloc(sizeof(int) * query->query_num);
    if (query->source == NULL || query->target == NULL) {
        fprintf(stderr, "memory cannot alloc ! : query\n");
        exit(1);
    }

    // クエリの読み込み
    for (i = 0; fgets(line, 1<<7, query_fp); ) {
        // ss type：target = -1
        if (strncmp(line, "s ", 2) == 0) {
            sscanf(&line[2], "%d", &(query->source[i]));
            query->target[i] = -1;
            i++;
        }
        // p2p type
        else if (strncmp(line, "q ", 2) == 0) {
            sscanf(&line[2], "%d %d", &(query->source[i]), &(query->target[i]));
            i++;
        }
    }

    fclose(query_fp);
}
```

次にコンパイルの例を挙げる. 様々なコンパイルオプションがあるが, あまりあれこれ付けてしまうと過剰な最適化により速度が低下しかねないため, 以下のように `-O2`, `-mach=nocona` の 2 つ程度で十分である. `-O2` は自動最適化オプション, `-mach=nocona` は CPU アーキテクチャに応じた最適化を行ってくれる. Core2 系 CPU では, nocona となる.

```
gcc -O2 -march=nocona sample.c
```

できあがったバイナリ `./a.out` は次のように実行することができる. 3 つめの引数 `report file` は最短距離・最短路の出力先ファイル名であるが, なくても正常に動作する.

```
./a.out [dimacs graph data] [dimacs query data] [report file]
```

```
[xxxxx@xxxx xxx]$ ./a.out USA-road-d.USA.gr USA10.p2p
============================================================
shortest path route solver
============================================================
algorithm        is dijkstra's algorithm & 1-level buclet
graph  file      is USA-road-d.USA.gr
query  file      is USA10.p2p
report file      is (null)
query            is point-to-point x 10
node / arc       is 23947347 / 58333344
max length       is 368855
```

```
 id |  source  |  target  | distance | time[ms]
----+----------+----------+----------+---------
  1 |   957498 | 10453327 | 39503661 |  3074.5
  2 | 19200797 |  7727679 |  6652463 |   667.9
  3 | 13006257 |    40639 | 15806743 |  2389.6
  4 |  4314559 | 22779984 | 31900664 |  1802.7
  5 | 17261435 |  8424294 |  2928315 |   195.0
  6 |  8077810 | 13186938 | 33411726 |  2877.6
  7 |  3048748 |  1475829 | 13958246 |  1010.8
  8 | 21869636 |  3531883 | 18762398 |   728.9
  9 | 13446936 |  4981527 | 36604539 |  2664.6
 10 | 18549540 |  3230879 | 20809134 |  2961.5

average distance is 22033788
file read time   is 39223.0[ms]
computing time   is 18373.2[ms]
```

AppendixA

Python解説

ここでは，超高水準プログラミング言語 Python（パイソン）について概説する．もちろんPythonのすべてをこの限られた付録のスペースで紹介することはできないので，本文でメタ解法を設計する際に用いたPythonの文法に限定して簡単に解説を行う．Pythonについての詳細は，以下のサイトを参照されたい．

日本語版のPythonは`http://www.python.jp/` から，英語版のPythonは`http://www.python.org/`からダウンロードできる．これらのページへのリンク，ならびに本書で用いたPythonのモジュールに対する情報は，本書のサポートページ`http://modern-heuristics.com/`から得ることができる．

A.1 なぜPythonか？

Pythonは初学者に優しいプログラミング言語である．まず，Pythonの予約語（キーワード）は，他の高級プログラミング言語とくらべて圧倒的に少ない．よって，初学者が覚える命令が少なくて済む．同時に，C(++) 言語のようにおまじないを覚える必要がないという意味でも敷居が低い．

たとえば，C++で画面に “Hello, world!” と出力しようと思ったら，

```
#include <iostream>
int main() {
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```

と，さっそく初学者に解説するのが困難な幾つかのキーワードが登場するが，Pythonだと，

```
print "Hello, world!"
```

と書くだけである．

Pythonでは字下げ（インデント）によりブロックを指定し，実行文をグループ化する文法を採用している．これによって，否が応でも綺麗にプログラミングを行うことが強制され，万人が同じように読みやすいプログラムを書くことが可能になる．たとえば，C++ だと

```
if (x > 1) { y=x+1;
  z=x+y; }  else { y=0; z=0; }
```

とも書ける（行儀の悪い）プログラムも，Pythonだと誰が書いても，

```
if x > 1:
   y=x+1
   z=x+y
else:
   y=z=0
```

となる．

開発を短時間で終わらせたい実務家にとってもPythonは有効である．まず，同じ内容のプログラミングをする際，Pythonで書かれたプログラムの行数は圧倒的に少なくなる．（これが本書でPythonを採用した一番の理由である．）これは，モジュールとよばれる部品がたくさん準備されているためである．

他にも，Pythonは「お気楽」言語として，様々な便利な機能を搭載している．たとえば，変数の宣言が必要なく，メモリ管理も必要なく，多くのプラットフォームで動作し，オブジェクト指向であり，しかもフリーソフトである．以下，その文法を簡単に解説していこう．

A.2　データ型

Pythonでは，整数型，長整数型，ブール型，浮動小数点数型，文字列型など，他のプログラミング言語でもお馴染みの基本データ型の他に，幾つかの複合型（リスト，タプル，辞書，集合）が準備されている．

A.2.1　数

最も基本的な数を表す型は，整数型である．この整数型は，32 ビットで表現される範囲の整数を表現する．Pythonでは無限長の整数を保管するための長整数型も準備されている．長整数型の数値は，数字の後ろにLをつけて表す．たとえば，$5L$ は長整数型の数値である．ブール型は，真（True）もしくは偽（False）を表現する．浮動小数点数型は，Pythonでは倍精度の浮動小数を表現し，対応する数値は数字の後ろに小数点を付加して区別する．たとえば，10. や 3.14 は浮動小数点数型の数値である．

A.2.2　文字列

文字列は，文字から成る**順序型**（sequence type；シーケンス型）である．文字列は，‘abcd’のようにシングルクオートで囲むか，"abcd"のようにダブルクオートで囲むか，

"""　解を評価する関数．以下の値を返す．

- 目的関数値

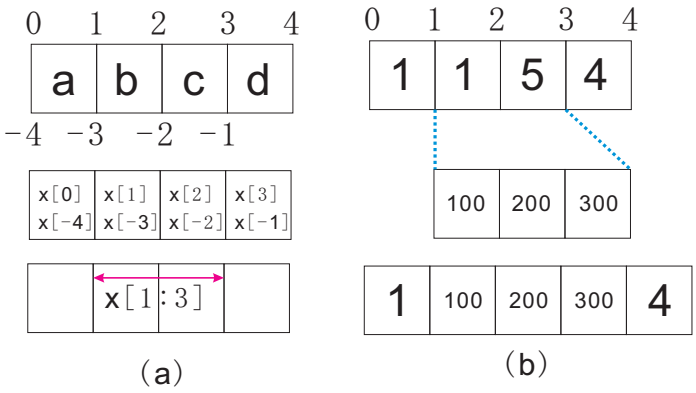

図 A.1: スライス表記の参考図．(a) 文字列と添え字．(b) リスト L=[1,1,5,4] に対してスライス表記による代入 L[1:3]=[100,200,300] を行った結果．

- 実行可能解からの逸脱量 """

のようにトリプルクオートで囲むことによって記述される．最後のトリプルクオートは，複数行にまたがる文字列を記述することができる．これは，関数定義（A.5 節）の直後に記述することによって，オンラインドキュメントを自動生成するときに便利である．

文字列は順序が付けられた型であるので，左から $0, 1, 2, \cdots$ と添え字付きの配列のように表記できる．たとえば，x= ‘abcd’ に対して，x[1] は‘b’ となる．

文字列などの順序型に対しては，コロン（:）で区切られた 2 つの添え字で，区間を表す**スライス表記**（slicing）が適用できる．（文字列の他の標準の順序型には，後述するリストやタプルがある．）スライス表記 $i:j$ は，$i \leq k < j$ を満たす整数 k を表す．右端の j は含まないことに注意されたい．これは，$j-i$ を対応する区間の要素数にするためである．また，i が省略されると最初から，j が省略されると最後までの区間を表す．たとえば，x= ‘abcd’ に対して，x[1:3] は‘bc’，x[1:] は‘bcd’ となる．‘abcd’ 全体は x[0:4] もしくは x[:] で表される．要素数を返す関数 len() を用いると，x= ‘abcd’ に対して len(x[1:3]) は 2 となり，$3-1$ に等しいことが確認できる．

末尾の文字は，-1 の添え字をもつ．添え字は，文字と文字の間の番号であると考えると分かりやすい．この関係を図 A.1 (a) に示す．

A.2.3 リスト

リストは任意の要素から成る順序型である．リストは，[ ] の中にカンマに区切って要素を並べることによって生成される．要素は異なる型でも良いし，入れ子にしても良い．たとえば，L= [1,5,"a"] や L= [1,[5,1,6],["a","b"]] はリストである．リストは，その中身を変更できる**可変**（mutable；変更可能）型である．ちなみに，文字列は中身を変えることのできない**不変**（immutable；変更不能）型である．

リストに対しても文字列と同じように，添え字を適用できる．たとえば，L=[1,1,5,4] に対して，L[2] は 5 であり，L[-1] は 4 である．スライス表記についても同様に，L[1:3] は [1,5] であり，L[2:] は [5,4] となる．

Table A.1: リストLに対する主要なメソッド．例としてL=[1,1,5,4] を用いる．

メソッド	説明と例
L.count(x)	L 内での x の生起回数を返す． L.count(1) ⇒ 2
L.index(x)	L 内で x が最初に発生する添え字を返す．ない場合には例外を発生． L.index(5) ⇒ 2
L.append(x)	L の最後に要素 x を追加する． L+=[x] と同じ効果． L.append(8) ⇒ L=[1,1,5,4,8]
L.insert(i,x)	L の i 番目の添え字に要素 x を挿入する．L[i:i] =[x] と同じ効果． L.insert(2,8) ⇒ L=[1,1,8,5,4]
L.remove(x)	L で最初に発生する x を削除する． L.remove(5) ⇒ L=[1,1,4]
L.pop(i)	L の i 番目の要素を削除する．i が省略された場合には最後の要素を削除． L.pop(3) もしくは L.pop() ⇒ L=[1,1,5]
L.reverse()	L を逆順にする． L.reverse() ⇒ L=[4,5,1,1]
L.sort()	L を小さい順に並べ替える． L.sort() ⇒ L=[1,1,4,5]

リストは（文字列とは異なり）可変型であるので，中身を代入によって変更することができる．たとえば，L=[1,1,5,4] に対して，L[1]=100 とすると，リストは L=[1,100,5,4] と変更される．スライス表記を用いると，指定した区間にリストの代入が可能である．たとえば，L=[1,1,5,4] に対して，L[1:3]=[100,200,300] とすると，リストは L=[1,100,200,300,4] と変更される（図 A.1 (b) 参照）．長さ 2の区間に長さ 3のリストを代入したので，リストの長さが1 つ増えていることに注意されたい．

リストに対する主要なメソッド（“.”の後ろにキーワードを記述するリストに対する関数）を，表 A.1 に示す．

A.2.4 タプル

タプル（tuple；組）もリストと同様に任意の要素から成る順序型である．タプルは，カンマ（,）で区切って要素を並べることによって生成される．たとえば，T= 1,5,"a" はタプルである．タプルを入れ子で定義する際には，( ) の中にまとめて記述する．たとえば，T= ((1,5),("a","b",6)) のように記述する．タプルはリストと違って不変型であり，中身を変えることはできない．

本文で巡回セールスマン問題の点の (X, Y) 座標を表す際には，

```
coord = [(4,0),(5,6),(8,3),(4,4),(4,1),(4,10),(4,7),(6,8),(8,1)]
```

のように，タプルのリストとして表現した．また，関数の返値として複数のものを返したいときにもタプルを用いた．このように，タプルは，複数のものをまとめて扱う際に用いられ，C 言語の構造体のような役目を果たす．

タプルの概念を使うと，変数の交換が簡単に書ける．たとえば，a と b を交換したいとき，通常は，

```
temp=a
```

```
a=b
b=temp
```

と一時保管用の変数tempを用いて書く必要があるが，Pythonでは，タプルb,aのタプルa,bへの代入と考え，

```
a,b = b,a
```

と 1 行で書くことができる．

A.2.5 辞書

辞書（dictionary）は，**キー**（key）と**値**（value）の組から構成される型である．辞書はリストやタプルとは異なり，順序型ではない．値は任意の型をとることができるが，キーになれるのは不変型のみである．したがって，数値，文字列,（不変な要素から成る）タプルなどをキーとして辞書は構築される．辞書は，{ }の中にカンマで区切って (キー : 値) を並べることによって生成される．たとえば，名前をキーとし，身長を値とした辞書は，以下のようになる．

```
D={ "Mary": 126, "Jane": 156, "Sara": 170}
```

辞書に格納されている値は，キーを添え字として抽出することができる.（そのため，辞書のキーは一意でなくてはならない.）たとえば，上の身長を格納した辞書で，D["Sara"] は 170 を返す．キーを添え字として値を変更する際には，D["Sara"]=130 と代入すれば良い．

A.2.6 集合

集合（set）は，要素の重複を削除したり，和集合（union），共通部分（intersection），差集合（difference）などの集合に対する演算を行うときに用いられる型である．可変（変更可能な）型である set と不変（変更不能な）型である fronzenset が用意されている．不変型の frozenset は，辞書のキーとして用いることができる．可変型の set は，リストと同様に要素を追加・削除することができる．順序型ではないので，追加はリストのように最後に追加（append）ではなく，add になる．削除は同じ remove である．

集合は，リストや文字列などの順序型から関数set を用いて生成できる．たとえば，set("abcde") は文字列の個々の要素から成る集合set(['a', 'c', 'b', 'e', 'd']) を返す．

集合に対する主要な（set と fronzenset の両者に共通の）メソッド（“.”の後ろにキーワードを記述するリストに対する関数）を，表 A.2 に示す．

A.3 演算子

演算は他のプログラミング言語と同様に，括弧 ()，べき乗（指数演算）**，乗算 * もしくは除算 /，加算 + もしくは減算 − の順で行われる．

Table A.2: 集合Sに対する主要な（setとfronzensetの両者に共通の）メソッド．例としてS=set([1,2,3,4]), T=set([1,2]) を用いる．

メソッド	説明と例
S.issubset(T)	集合 S が集合 T の部分集合であるとき真（True），そうでないとき偽（False）を返す． S <= T と同じ効果．S.issubset(T) ⇒ False
S.issuperset(T)	集合 T が集合 S の部分集合であるとき真（True），そうでないとき偽（False）を返す． S >= T と同じ効果． S.issuperset(T) ⇒ True
S.union(T)	集合 S と T の和集合を返す． S | T と同じ効果． S.union(T) ⇒ set([1, 2, 3, 4])
S.intersection(T)	集合 S と T の共通部分を返す． S & T と同じ効果． S.intersection(T) ⇒ set([1, 2])
S.difference(T)	集合 S と T の差集合（S に含まれるが T に含まれない要素から成る集合）を返す． S - T と同じ効果． S.difference(T) ⇒ set([3,4])

割る数と割られる数が両方とも整数の場合には，除算 / は整数除算で行われる．この場合には，答えは小数ではなく，整数に切り捨てられた結果を返す．たとえば，4/5は0を返す．いずれか片方の数値が浮動小数点数型なら，結果は小数となる．たとえば，4./5 や4/5.0は0.80000000000000004を返す．

文字列やリストに対しても数値と同様の加算や乗算を行うことができる．たとえば，S="abc"に対して，S+"d"は"abcd"を返し，S*2は"abcabc"を返す．

if文やwhile文の中では，比較や論理条件を表す演算子が用いられる．<= は以下，>= は以上， == は等しい，!= は等しくないときに真になり，それ以外のとき偽になる演算子である．in は集合やリストなどの要素であるとき，not in は要素でないときに真になり，それ以外のとき偽になる演算子である．ブール演算子and やor は，それぞれ論理積と論理和をとる演算子である．真を 1，偽を 0 としたとき，論理積は乗算，論理和は加算に対応する．たとえば，(1<4) or (5<4)は1<4が真なので，$1+0=1$ となるので，真になる．

文字列，リスト，タプル，辞書，集合に対しては，それに含まれる要素数（長さ，位数）を返す関数lenがある．文字列のスライス表記のところで紹介したがlen('abcd') は 4 を返す．リストに対しても同様に，len([1,4,3,6]) は 4 を返し，タプルでも同様にlen((1,4,3,6)) は 4 を返す．

A.4 制御フロー

ある程度長いプログラムは，分岐や反復などの制御フローから構成される．ここでは，そのような制御フローに関する文法について解説する．

A.4.1 分岐

「もし」このような条件を満たしていたら「… せよ」という命令によって，プログラムは分岐した処理を行うことができる．Pythonにおける分岐のための予約語（キーワード）は，if である．文型は以下のように書く．

```
if 条件文:
```

「… せよ」(条件文が真のときに実行される命令)

Pythonの特徴として，字下げ（インデント）で命令群の開始や終了を表していることに注意されたい．たとえば，変数 x が負のときに “赤字だよ” と印刷するためには，以下のように記述すれば良い．

```
if x<0:
    print "赤字だよ！"
```

条件を満たさないときにも，何か処理をしたい場合には，`if` の後に `else` を入れた文型になる．

```
if 条件文:
```

「… せよ」(条件文が真のときに実行される命令)

```
else:
```

「… せよ」(条件文が偽のときに実行される命令)

A.4.2 反復

計算機に仕事をやらせる一番の動機は，人間には面倒な反復操作を行わせることだろう．そのため，プログラミングでは反復を行うための命令は，頻繁に利用される．

最も良く使う反復は，`for` 文である．Pythonにおける `for` 文は，リストや辞書などの反復可能な型から，1つずつ要素を取り出して反復を行う．たとえば，リスト `L =[ ‘こぶた’, ‘たぬき’, ‘きつね’, ‘ねこ’]` を順番に書き出したい場合には，以下のように記述する．

```
for x in L:
    print x
```

一般に，リストに対する反復は，以下のように記述される．

```
for 反復ごとに代入される変数 in リスト:
```

繰り返しをしたい文

反復したいものが単なる整数の並びである場合には，`range()` 関数を用いて，連続する数から構成されるリストを生成して，`for` 文とあわせて用いる．たとえば，`range(5)` はリスト `[0,1,2,3,4]` を返すので，$0, 1, 2, 3, 4$ を出力したいときには，

```
for x in range(5):
    print x
```

と書けばよい．一般には range(i,j) は，$i \leq k < j$ を満たす整数 k から成るリストを返す．スライス表記と同様に，j 未満の数を返すことに注意されたい．たとえば，range(1,3) は，[1,2] を返す．

for 文はリストの中に記述することもできる．これは，**リスト内包表記**（list comprehension）とよばれ，リストをコンパクトに定義するためにしばしば用いられる．たとえば，[(x,x**2,2**x) for x in range(5)] は，$x, x^2, 2^x$ から成るタプルのリスト [(0, 0, 1), (1, 1, 2), (2, 4, 4), (3, 9, 8), (4, 16, 16)] を出力する．

もう 1 つの反復を表すキーワードとして，while がある．while は，一般に以下のように記述する．

```
while 真か偽を判定される式:
    繰り返しをしたい文
```

例として，変数 x が正の間だけ x^2 を出力するプログラムを示す．

```
x=10
while x>0:
   print x**2
   x =x-1
```

反復の途中でから抜けるためのキーワードとして break と continue がある．両者とも必ず反復の中に記述し，break が実行されると反復から抜け，continue が実行されると次の反復処理に飛ばされる．

上の while の例題と同じプログラムを break を用いて記述すると，以下のようになる．

```
x=10
while True:
   print x**2
   x =x-1
   if x<=0:
     break
```

A.5 関数

関数を定義するには，キーワード def を用い，以下のように記述する．

```
def 関数名(引数):
    関数内で行う処理
```

また，関数から何らかの返値を得たい場合には，キーワード return の後ろに返したい値を記述する．たとえば，以下の関数は，与えられた文字列を合体させてから，3 回繰り返したものを返す関数である．

```
def concatenate3(a,b):
      c=a+b
      return 3*c
```

上の関数を concatenate3("a","b") と呼び出すと，ababab が返される．

関数は入れ子にして（つまり自分で自分を呼び出して）用いることもできる．これを**再帰**（recursion）とよぶ．例として，$n!$（n の階乗）を返す関数を考える．$n!$ は，非負の整数 n に対して，$n\cdot(n-1),\cdots,3\cdot2\cdot1$ を表し，急激に増大することで知られている．$n!$ は，$0!=1$ という初期条件の下で，$n!=n\times(n-1)!$ と再帰的に定義できる．これは，再帰を用いて以下のように記述できる．

```
def factorial(n):
      if n==0:
         return 1
      else:
         return n*factorial(n-1)
```

A.6　モジュール

Pythonで長いプログラムを書く際には，ファイルを分割して保管しておくことが望ましい．分割して保管されたPythonのプログラムファイル（これをモジュールとよぶ）を読み込むには，キーワード import を用いて，

import ファイル名

と記述する．モジュールは，必ず接尾語 ".py" をつける必要がある．読み込みの際には，".py" は付けずに呼び出す．

自分が書いたプログラムだけでなく，他人の書いた便利なプログラムも同様に読み込んで使うことができる．本文のプログラムでは，数値演算用のモジュール NumPy と乱数発生用のモジュール random，数学用のモジュール math を読み込んで用いた．例として，数学用モジュール math を読み込んで，モジュール内の平方根を返す関数 sqrt() を呼び出して，$\sqrt{2}$ の計算結果（1.41421356237）を印刷してみる．

```
import math
print math.sqrt(2)
```

import 文ではモジュールを読み込むだけなので，平方根を計算するためには，math.sqrt(2) と記述する必要がある．これをさぼって sqrt(2) と書くだけで済ますには，

from ファイル名 import 関数名1，関数名2，…

と記述すればよい．関数名を書くのが面倒なときには，ワイルドカード $*$ を用いて，

Table A.3: randomモジュールの主要な関数.

関数	説明と例
`seed(x)`	x を用いて乱数の初期化を行う．x を省略した場合には現在のシステム時間で初期化される． `seed(156)`
`random()`	$[0.0, 1.0)$ の一様ランダムな浮動小数点型の数を返す． `random()` ⇒ `0.48777947886`
`randint(i,j)`	整数 i, j $(i \leq j)$ に対して $i \leq k \leq j$ の一様ランダムな整数 k を返す． `randint(-5,1)` ⇒ `-3`
`shuffle(L)`	リスト L を順序をランダムに混ぜる． `L=[1,2,3,4], shuffle(L)` ⇒ `L=[4, 1, 3, 2]`
`choice(L)`	リスト L からランダムに1つの要素を選択する． `L=[1,2,3,4], choice(L)` ⇒ `3`

```
from ファイル名 import *
```

と書いても良い．たとえば，平方根の計算は，

```
from math import *
print sqrt(2)
```

と簡略化される．

以下では，本文のプログラムで用いたrandomモジュールとNumPyモジュールについて簡単に解説する．

A.6.1 擬似乱数発生モジュール random

randomは擬似乱数発生のための便利な関数から成る標準モジュールである．本文のプログラムで用いた関数を表 A.3 に示す．

A.6.2 科学計算モジュール numpy

numpy（Numeric Python）は科学計算のための便利な関数から成るモジュールである．numpyは標準モジュールではないので，使用する環境に応じてインストールする必要がある．numpyは`http://numpy.scipy.org/`からダウンロードすることができる．本文のプログラムで用いているのは，数値を要素とした行列と配列の生成である．行列も配列もリストを用いて記述できるが，大規模な問題に対して高速なプログラムを作成するためには，numpyを用いた方が良い．

numpyで長さ 5 の整数を要素とした配列 `a` を宣言し，各要素の初期値を 0 とするには，

```
from numpy import *
a=zeros(5,int)
```

とすれば良い．

2 行，5 列の配列 `A` で各要素が 1.0 の実数を生成するには，

```
A=ones((2,5),int)
```

とすれば良い.

行列の2 行，3 列目の要素を参照するには，A[2,3] とする（リストを用いた場合には，A[2][3] となるので注意）.

A.7 networkX モジュール

ここでは，グラフを処理するためのPython のモジュールである networkX について解説する．networkX についての詳細は，https://networkx.lanl.gov を参照されたい.

A.7.1 グラフの生成と基本操作

networkX におけるグラフには，以下の種類がある.

- Graph: 自己ループや並列枝をもたない無向グラフ．新しい空のグラフを生成するためには，G=Graph() とする.
- DiGraph: 自己ループや並列枝をもたない有向グラフ．新しい空のグラフを生成するためには，G=DiGraph() とする.
- XGraph: 自己ループ，並列枝を許し，さらに枝上に任意のデータをもつことができる無向グラフ．新しい空のグラフを生成するためには，G=XGraph() とする.
- XDiGraph: 自己ループ，並列枝を許し，さらに枝上に任意のデータをもつことができる有向グラフ．新しい空のグラフを生成するためには，G=XDiGraph() とする.

グラフ G に，点 i を追加するためには，G.add_node(i) とする．グラフ G に，枝上のデータ length をもつ枝 (i,j) を追加するためには，G.add_edge(i,j,length) とする．グラフを初期化するための関数は，G.clear() である．グラフにパスを追加するには，G.add_path(点のリスト) とする.

点 i の次数（接続する枝の本数）を得るには，**G.degree(i)** とする．点 i に隣接する点のリストを返す関数は，G.neighbors(i) である．これは，G[i] と書いても良い．枝 (i,j) 上の情報は，G.get_edge(i,j) で得ることができる.

A.7.2 ファイルの読み書き

networkX には，グラフをファイルに保存したり，読み込んだりするための関数が準備されている.

枝上にデータが付加されたグラフを隣接リスト形式での保存は，以下のように行われる.

```
write_multiline_adjlist(G, path)
```

ここで，pathは保存するファイル名である．ファイル名が .gz もしくは .bz2 で終わる場合には，圧縮形式で保存される．

保存したデータの読み込みは，以下のように行われる．

```
read_multiline_adjlist(path, nodetype, edgetype)
```

ここで，pathは読み込みを行うファイル名である．また，nodetypeとedgetypeは，それぞれ点と枝上のデータの型であり，省略可能である．省略された場合には，文字列として読み込まれる．