数理工学実験
テーマ:組合せ最適化

2 回生　田中風帆 (1029321151)
実施場所:自宅

実施:2021 年 12 月 27 日
提出:2022 年 1 月 16 日

目 次

図 目 次

表 目 次

第 I 部
概要

このレポートでは、分枝限定法を用いて負閉路を持つグラフに関する最短経路問題を解くアルゴリズムを実装した。限定操作を行わない場合 (課題 1,2) と行った場合 (課題 3,4) で同じ問題を解き、かかった時間および探索したノードの数を比較した。結果は表とグラフにまとめ、それらに対して考察を行った。

第 II 部
課題 1

ここでは、グラフ $G = (V, E)$、節点 $s, t \in V$ および各枝 $(u, v) \in E$ の長さ $w(u, v)$ が与えられた時、点 s から t へ至る最短路を求める分枝アルゴリズムを設計し、その擬似コードを与えた。ただし、経路は単純なもの (同じ節点を二度以上通らない) に限るものとした。

1　アルゴリズムの説明

設計したアルゴリズムの説明を行う。
再帰関数 $calcMinDist1$ は、節点 x、節点 s から x へ至る有向路に含まれる節点集合 F、その有向路の経路長 sum を引数とする。出力はグラフ G 上の点 s から点 t に至り、かつ節点集合 F を含む有向路の経路およびその経路長である。また、その時点で求まっている暫定の s から t への最短経路長と比較し、求まった経路長がそれより短かった場合、求めた経路および経路長を暫定の最短経路および最短経路長として更新する。これにより、s から t への単純な有向路とその経路長を全て調べることができ、結果として解が求まる。

2　擬似コード

Algorithm 1 calcMinDist1(x,F,sum)

Input: グラフ G 上の節点 x , 節点 s から x へ至る有向路に含まれる節点集合 F,s から x へ至る有向路の経路長

Output: グラフ G 上の点 s から点 t に至り、かつ節点集合 F を含む有向路の経路およびその経路長. また、暫定の最短路 ($minPath$) およびその経路長 ($minDist$) と比較し、代入作業を行う.

```
for 全ての有向枝 (x, y) ∈ E, y ∉ F について do
  if y=t then
    output F ∪ {y}, sum + w(x, y)
    if sum + d(x, y) < それまでに求まっている s − t 間の最短経路長 minDist then
      minDist ← sum + w(x, y);minPath ← F ∪ {y}
    end if
  else
    calcMinDist1(y, F ∪ {y})
  end if
end for
```

第 III 部

課題 2

ここでは、課題 1 で設計したアルゴリズムを実装し、グラフの節点数および枝の数に対して実行時間および探索したノード数を調べ、表とグラフにまとめた。ただし、探索の始点はノード 0、終点はノード $N-1$(N はグラフを構成するノードの数) とした。

3 課題 2 の回答

以下はノード数 6 のグラフに対して、枝数を 12,14,16,18,20,22,24,26,28,30 として、探索にかかった時間および探索したノード数をまとめたものである。グラフは横軸が枝の本数、縦軸が時間または探索したノード数である。

表 1: ノード数 6 のグラフ探索にかかった時間と探索ノード数の表. 枝の数ごとにまとめてある.

枝の本数	探索ノード数	時間 (msec)
12	9	0.0669956
14	11	0.082016
16	15	0.111103
18	25	0.187159
20	25	0.29397
22	43	0.336885
24	43	0.403881
26	43	0.424862
28	54	0.628948
30	65	0.688076

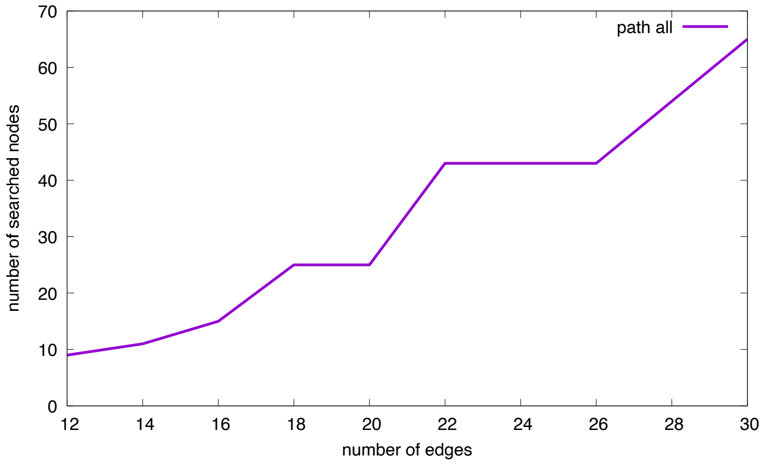

図 1: ノード数が 6 の時のグラフ. 横軸が枝数、縦軸が探索したノードの数.

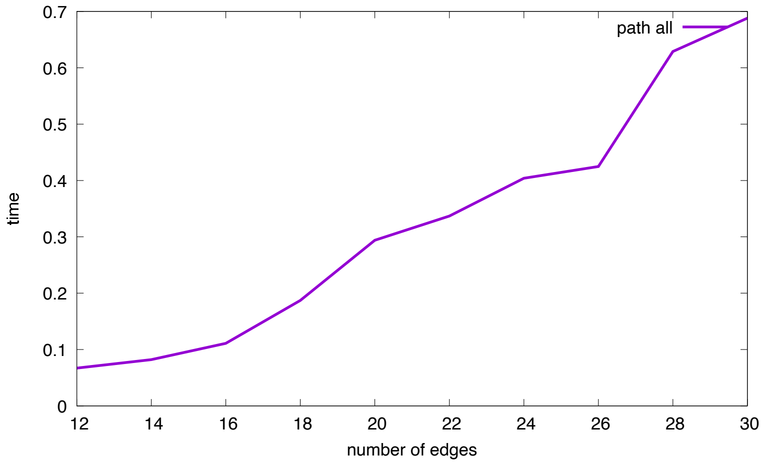

図 2: ノード数が 6 の時のグラフ. 横軸が枝数、縦軸がかかった時間 (cpu 時間で、単位は msec).

以下はノード数 10 のグラフに対して、枝数を 20,30,40,50,60,70,80,90 として、探索にかかった時間および探索したノード数をまとめたものである。グラフは横軸がエッジの本数、縦軸が時間または探索したノード数である。

表 2: ノード数 10 のグラフ探索にかかった時間と探索ノード数の表. 枝の数ごとにまとめてある.

枝の本数	探索ノード数	時間 (msec)
20	8	0.0858307
30	100	1.17207
40	459	6.12497
50	1690	17.6229
60	7047	65.5611
70	19971	188.124
80	42070	399.007
90	109601	1050.97

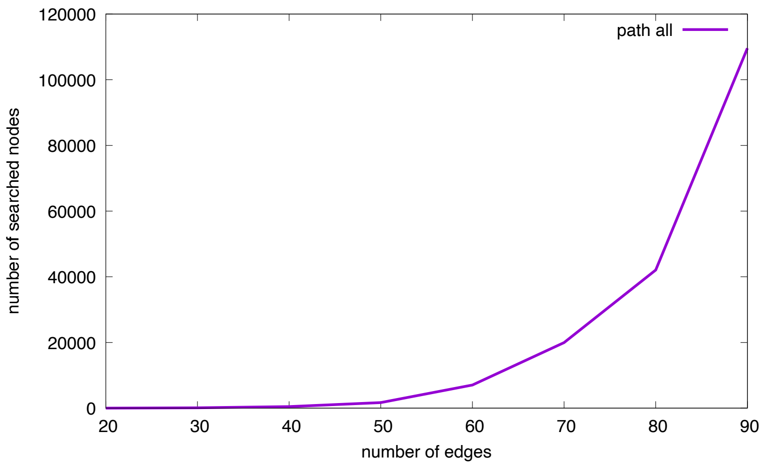

図 3: ノード数が 10 の時のグラフ. 横軸が枝数、縦軸が探索したノードの数.

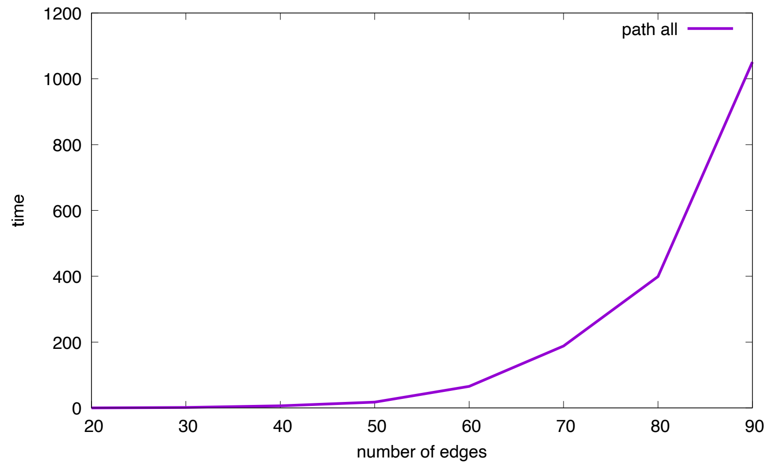

図 4: ノード数が 10 の時のグラフ. 横軸が枝数、縦軸がかかった時間 (cpu 時間で、単位は msec).

以下はノード数 14 のグラフに対して、枝数を 30,40,50,60,...,120 として、探索にかかった時間および探索したノード数をまとめたものである。グラフは横軸がエッジの本数、縦軸が時間または探索したノード数である。

表 3: ノード数 14 のグラフ探索にかかった時間と探索ノード数の表. 枝の数ごとにまとめてある.

枝の本数	探索ノード数	時間 (msec)
30	30	0.311852
40	258	2.70605
50	1284	14.0021
60	9357	93.3471
70	42910	447.985
80	135860	1385.51
90	628661	6485.87
100	1733836	18406.3
110	5437492	60884.5
120	13019591	150919

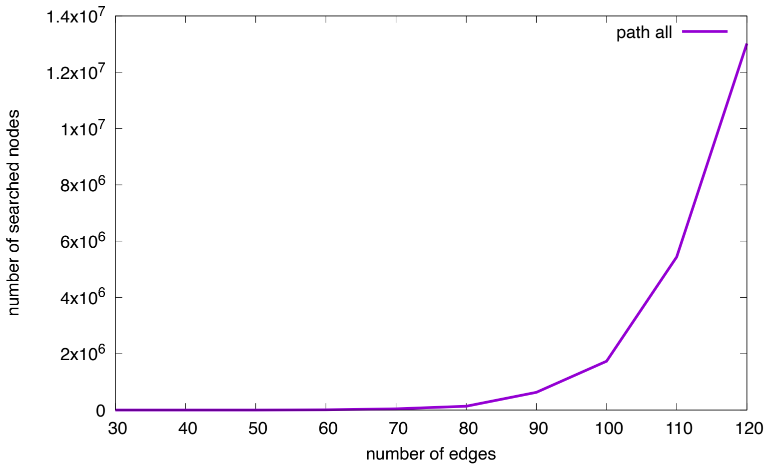

図 5: ノード数が 14 の時のグラフ. 横軸が枝数、縦軸が探索したノードの数.

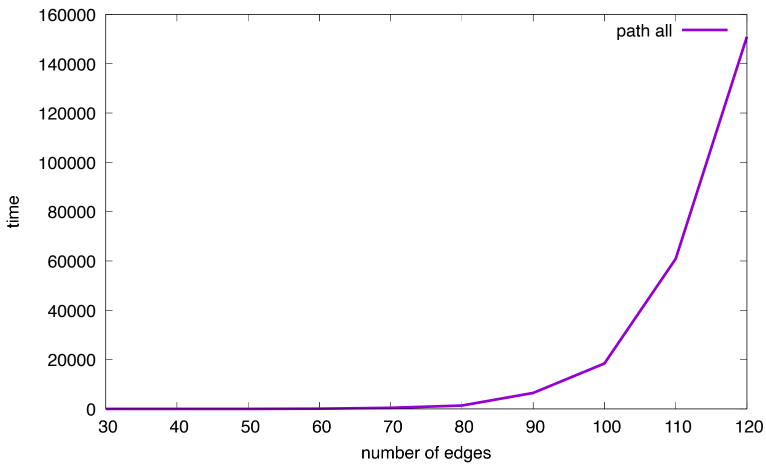

図 6: ノード数が 14 の時のグラフ. 横軸が枝数、縦軸がかかった時間 (cpu 時間で、単位は msec).

以下はノード数 18 のグラフに対して、枝数を 40,60,80,100,120 として、探索にかかった時間および探索したノード数をまとめたものである。グラフは横軸がエッジの本数、縦軸が時間または探索したノード数である。

表 4: ノード数 18 のグラフ探索にかかった時間と探索ノード数の表. 枝の数ごとにまとめてある.

枝の本数	探索ノード数	時間 (msec)
40	24	0.245094
60	6579	67.374
80	315507	3493.82
100	8382312	93073.5
120	133827589	1.57905e+06

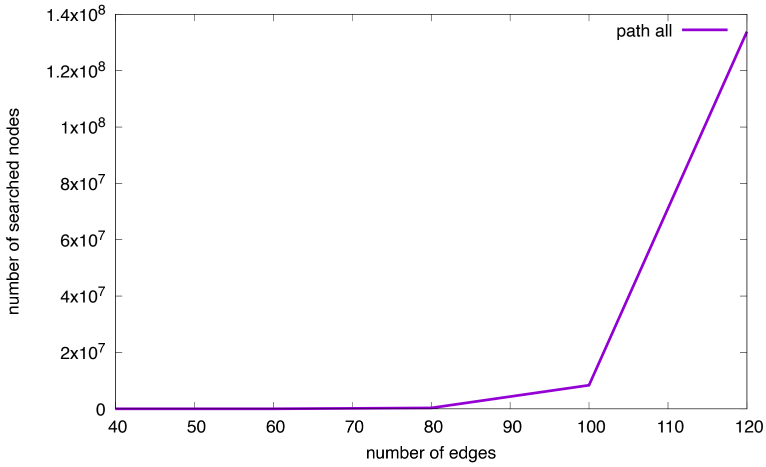

図 7: ノード数が 18 の時のグラフ. 横軸が枝数、縦軸が探索したノードの数.

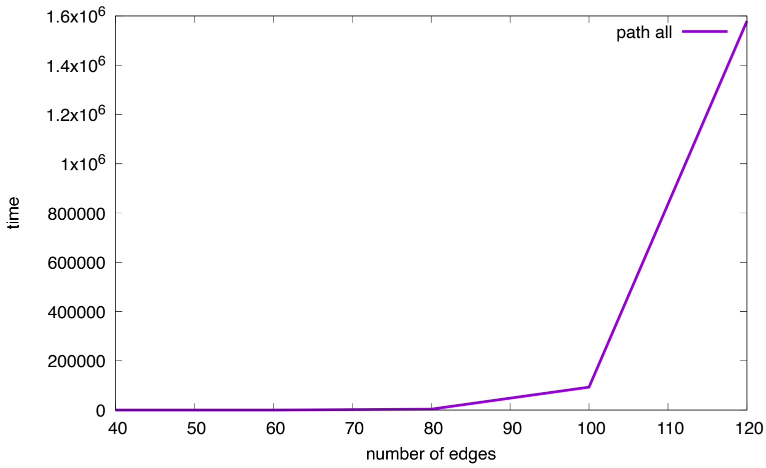

図 8: ノード数が 18 の時のグラフ. 横軸が枝数、縦軸がかかった時間 (cpu 時間で、単位は msec).

4　考察

グラフが密になるにしたがい、探索時間、探索ノード数共に爆発的に増加することが分かった。全ての有向経路を調べており、組合せ爆発が起こるためこの結果は妥当であると言える。また、全ての有向経路を比較することからこの計算にかかる計算量はノードの数を N として $O(N!)$ となるので、理論的にはノードの数の増加に伴い計算にかかる時間や探索ノードも増加するといえる。この実験結果はこの理論通りのものであることから、正しいものであると結論づけられる。

5　課題 2 のコード

コード 1: 課題 2 のコード

```
#include <vector>
#include <iostream>
#include <fstream>
#include <cmath>
#include <algorithm>
#include <time.h>
#include <sys/time.h>
using namespace std;

double gettimeofday_sec(){
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return tv.tv_sec+(double) tv.tv_usec*1e-6;
```

```
}

struct Edge {
    int to; // 隣接頂点番号
    int w; // 重み
    Edge(int to, int w) : to(to), w(w) {}
};

double min_weight = pow(10,6);
vector<int> minPath;
int node_num = 0;//探索ノード数

//再帰関数
void PathAll(int x, int t, vector<int> F, vector<vector<Edge> > G, int sum){
    node_num += 1;
    //x から伸びる有向枝に対して
    for(auto e : G[x]){
        int y = e.to;
        bool include = false;
        for(int i=0; i<F.size(); i++){
            if(F[i] == y) include = true;
        }
        if(include) continue;
        else{
            vector<int> F_copy;
            int sum_tmp;
            copy(F.begin(), F.end(),back_inserter(F_copy));
            //s から x までの距離に w(x,y) を足す
            sum_tmp = sum + e.w;
            F_copy.push_back(y);
            if(y == t){
                if(sum_tmp < min_weight){
                    min_weight = sum_tmp;
                    minPath.clear();
                    copy(F_copy.begin(), F_copy.end(), back_inserter(minPath));
                }
            }
            else{
                PathAll(y, t, F_copy, G,sum_tmp);
            }
        }
    }
}

int main(){
    //ファイルの読み込み
    int N, M;
    string filename("Graphs/n_14/n_14_m_90.txt");
    int number;

    ifstream input_file(filename);
    if (!input_file.is_open()) {
        cerr << "Could not open the file - '"<< filename << "'" << endl;
        return EXIT_FAILURE;
    }

    //グラフの作成
    int i=0;
    vector<int> From, To, W;
    while (input_file >> number) {
```

```
            i++;
            if(i==1) N = number;
            else if(i==2) M = number;
            else{
                if (i%3==0) From.push_back(number);
                if (i%3==1) To.push_back(number);
                if (i%3==2) W.push_back(number);
            }
        }
        input_file.close();

        vector<vector<Edge> > G(N);

        for(int k=0; k<M; k++){
            int from = From[k];
            int to = To[k];
            int w = W[k];
            G[from].push_back(Edge(to, w));
            }
        vector<int> F_;
        int s = 0;
        int t = N-1;
        F_.push_back(s);

        //時間の計測と pathall の実行
        double start = gettimeofday_sec();
        PathAll(s,t,F_,G,0);
        double end = gettimeofday_sec();
        cout << "実行にかかった時間は
            ␣" << (end-start)*1000 << "␣msec"<< endl;
        cout << "探索したノードの数は" << node_num <<endl;

        cout << "最短経路は" << endl;
        for(int i=0; i<minPath.size(); i++){
            if(i != minPath.size()-1) cout << minPath[i] << "␣->␣";
            else{
                cout << minPath[i]<< endl;
            }
        }
        cout << "その距離は␣" << min_weight << endl;
}
```

第 IV 部

課題 3

ここでは、課題 1 で設計したアルゴリズムに限定操作を導入し、どの妥当性を説明および証明した。さらに限定操作別の擬似コードを書き、最後に全体の限定操作アルゴリズムの擬似コードを与えた。

6 アルゴリズムの説明

ここでは今回設計した限定操作および、限定操作を導入した分枝限定アルゴリズムの説明を行い、その正当性を証明した。

6.1 全体の流れ

節点 s から節点 x を経由し、節点 t へ至る経路に限定して考える。s から x への有向経路に含まれる節点の集合 F およびその経路長はすでに求まっているとする。この時、節点 x から節点 t へ至る有向経路 (ただし F に含まれる節点は通らない) とその経路長を求める部分問題を考える。
F に含まれない x の子節点が n 個存在するとし、それらを $v_1, ..., v_n$ とする。節点 $v_i(i = 1, ..., n)$ から節点 s へ至る経路長の上界と下界をそれぞれ求める。ただし、上界は v_i から t への実際の最短経路長以上であることが保証されている値、下界は v_i から t への実際の最短経路長以下であることが保証されている値である。以下、この上界の値を $upper(v_i)$、下界の値を $lower(v_i)$ と表す。この時、ある $i, j(i, j \in \{1, ..., n\})$ について、$upper(v_i)+w(x, v_i) < lower(v_j)+w(x, v_j)$ が成立する時節点 v_j を探索する必要はないと判定し、枝刈りを行う (限定操作 1)。また、その時点での s から t への暫定の最短距離を $minDist$ と表すこととした時、$minDist < sum + w(x, v_j) + lower(v_j)$ が成立する場合も同様に枝刈りを行う (限定操作 2)。さらに下界を求めた際、v_i から t までの部分グラフに負閉路がないとわかった場合、求めた下界を与える経路は v_i から t までの厳密な最短経路となっていると判断し、v_i からの探索を打ち切る (限定操作 3)(下界の求値にベルマンフォード法を用いるためこれは正当である。詳細は後ほど述べる)。この時、求めた経路長および $w(x, v_i)$ の値を F により構成される s から x までの経路長に加え、その値が $minDist$ 未満であれば $minDist$ にその値を代入する。またその経路も外部の変数に保存する。以上の限定操作を行った上で、活性状態のノードを探索することで探索ノード数を大幅に減らした求解が可能となる。

6.2 上界について

6.2.1 上界の求め方

ここでは x から t へ至る最短経路 (ただし集合 F に含まれる節点は通らない) の長さ以上であることが保証されている値を出力する関数 $Upper(x, F)$ を定義する。関数 $Upper$ は x から t までの部分問題に存在する負辺を全て 0 と置換したグラフ G' に対してダイクストラ法を適用し、G' における x から t までの有向経路を求め、さらにその経路の G' 上における経路長を求める。

このようにして求められた経路長は、G 上における x から t までの最短経路長以上の値となっている。

6.2.2 ダイクストラ法

ダイクストラ法は、重みが非負のグラフにおける最短路問題を解くための手法である。そのアルゴリズムは次のグラフの性質を帰納的に用いるというものである [1]。
性質：非負の重み w を持つ有向ネットワーク $N = [G = (V, E), w]$ において、点の部分集合 $A \subseteq V$ と始点 $s \in A$ が選ばれている。この時、$E(A, V \backslash A)$ のなかで $dist(s, u) + w(u, v)$ の値を最小にする枝を $(u', v') \in E(A, V \backslash A)$ とすると、$dist(s, v') = dist(s, u') + w(u', v')$ が成り立つ
証明：$dist(s, v') \leq dist(s, u') + w(u', v')$ は明らか。
$dist(s, v') \geq dist(s, u') + w(u', v')$ を示す。P を始点 s から点 v' への最短路の一つとすると、$dist(s, v') = w(P)$ であり、P と $E(A, V \backslash A)$ は少なくとも一本の枝 (x, y) を共有する。ここで $P_{s,x}, P_{y,v'}$ をそれぞれ P 上で s から x まで、y から v' までの部分パスとする。最短路長の定義から $dist(s, x) \leq w(P_{s,x})$ が成り立ち、重みが非負であることから $0 \leq w(P_{y,v'})$ が成り立つ。枝 (u', v') の選択基準から、$dist(s, u') + w(u', v') \leq dist(s, x) + w(x, y)$ が成り立つ、以上より、$dist(s, v') = w(P)w(P_{s,x}) + w(x, y) + w(P_{y,v'}) \leq dist(s, u') + w(u', v') + 0 \leq dist(s, x) + w(x, y)$ が示された。(証明終わり)

6.2.3 正当性の証明

上界の求め方の正当性を示す。最小化問題において、一つでも実行可能解があった場合それは上界となる。これは、x から t までの最短経路問題の場合、x から t までの到達可能な経路が存在すればその距離は必ず実際の解 (l_{real} とする) 以上の値となることを意味する (＊)。また、グラフ G において負辺の重さを 0 にしたとしても、辺の向きが変わらなければ x から t までの到達可能な経路の存在は保存される。辺が全て非負のグラフに対してはダイクストラ法を適用することができ、これにより x から t への単純な最短経路を得ることができる。得られた経路のグラフ G 上における長さを l、G' 上における長さを l' とする。この時、(＊) より $l_{real} \leq l \leq l'$ となる。これより、l' は G 上における x から t への最短経路長以上となっているといえ、これは上界として機能する。擬似コードは以下である。

6.2.4 x から t までの部分問題の上界を求める関数 $Upper$ の擬似コード

Algorithm 2 Upper(x,F)

Input: グラフ G 上の節点 x , 節点 s から x へ至る有向路に含まれる節点集合 F

Output: グラフ G 上の点 x から t に至り、F に含まれる節点を通らない最短経路の長さの上界

$B = \{x\}, l(x) = 0, A = \phi$,$dist(x, v) = 0$ と初期化;x 以外の節点 x' については $l(x') = INF$,$dist(x, x') = INF$(ただし $dist(x, x')$ は x から x' までの最短距離)

while B が空でない **do**

  B の中で $l(v)$ を最小にする v を選ぶ;$dist(x, v) \leftarrow l(v)$

  if $v = t$ **then**

    output $l(v)$

  end if

  v を B から除き A に加える

  for v から出る各有向枝 $(v, z) \in E, z \notin F$ **do**

    if $z \in V \backslash (A \cup B)$ **then**

      z を B に加え、$l(z) = dist(x, v) + max\{w(v, z), 0\}$ と設定する

    else if $z \in B$ and $dist(x, v) + max\{w(v, z), 0\} < l(v)$ **then**

      $l(z) = dist(x, v) + max\{w(v, z), 0\}$

    else

      (v, z) を走査済みとする

    end if

  end for

end while

6.3 下界について

6.3.1 下界の求め方

ここでは x から t へ至る最短経路 (ただし F に含まれる節点は通らない) の長さ以下であることが保証されている値を出力する関数 $Lower(x, F)$ を定義する。なお、求めた経路が実際の最短経路であることがわかった場合は、その経路も出力する。関数 $Lower$ は、x から t までの部分グラフ G'(全体のグラフ G のノードから F に含まれるノードを除いたもの。ノード数は N' とする) において、ベルマンフォード法を $N' - 1$ 回繰り返し、その結果として得られた x から t までの距離を下界として出力する。また、N' 回目の反復において、ベルマンフォード法で値の更新が行われなかった場合、G' 上に負閉路は存在しないと判別できる。G' 上に負閉路が存在しなかった場合は求めた x から t への最短経路を同時に出力する。

6.3.2 ベルマンフォード法

ベルマンフォード法のアルゴリズムを述べる。
グラフ上で始点と終点を決め、それぞれのノードの始点からの距離を無限で初期化する (始点は 0 とする)。あとは以下の操作を収束するまで繰り返す。
距離が無限でない頂点全てに対して、(基点となる頂点の最短距離) + (隣接頂点に到達するためのコスト) < (隣接頂点の最短距離) が成立するかどうかを調べ、成立する場合は隣接している頂点の最短距離を (基点となる頂点の最短距離) + (隣接頂点に到達するためのコスト) に更新する。

6.3.3 正当性の証明

まず、ベルマンフォード法において、G' に x から到達可能な負閉路が存在しない場合高々$N' - 1$ 回の反復で最短経路が求まること、およびグラフ G' 上に到達可能な負閉路が存在するならば N' 回目の反復で値の更新が起こることを示す [2]。
到達可能な負閉路をもたないグラフの場合、路中に含まれる閉路を除去することで道とする操作を行っても長さが増加することはない。よって路のうちそれに含まれる辺の本数が高々$N' - 1$ 以下であるもののみを考えれば良い。これは「各辺について一通り緩和する」という処理を最大でも $N' - 1$ 回反復すれば始点から到達可能な全頂点に対する単純な経路の最短路長が求まっていることを意味する。また、x から到達可能な負閉路 P の各頂点を $v_0, v_1, ..., v_{k-1}, v_0$ とする。もし P に含まれる全ての辺について更新が行われなかったと仮定すると、

$$l(P) = \sum_{i=0}^{k-1} l((v_i, v_{i+1})) \geq \sum_{i=0}^{k-1} (d[v_{i+1}] - d[v_i]) = 0 \tag{6.1}$$

が成立する。これは P が負閉路であることに矛盾する。よって x から到達可能な負閉路がある場合は N' 回目の反復時に必ず更新が発生することが示される。
次に、x を始点とするノード数 N' の部分グラフ G' において $N'-1$ 回ベルマンフォード法の反復を行えば x から t への最短経路長の下界値が求まることを示す。
グラフ G' 上に負閉路が存在しない場合、ベルマンフォード法の正当性より x から t への最短路が求まる。グラフ G' 上に x から到達可能な負閉路が存在する場合についても、任意の頂点 v について、どの反復回においても、緩和操作の性質より実際の x から v へ最短経路長よりも大きい値に更新されることはないと言える。以上より、ベルマンフォード法を $N'-1$ 回繰り返せば、x から t への単純な最短経路長以下の値が x から t への最短経路長として求まることが示される。

以上より、上で与えたアルゴリズムによりグラフ G' 上における x から t への単純な経路長の下界値が求まることが示される。

6.3.4 x から t までの部分問題の下界を求める関数 $Lower$ の擬似コード

擬似コードは以下である。

Algorithm 3 Lower(x,F)

Input: グラフ G 上の節点 x , 節点 s から x へ至る有向路に含まれる節点集合 F

Output: グラフ G 上の点 x から t へ至る最短経路の長さの下界. それが厳密な最短経路長かどうかを表す変数 $boolean$. それが厳密な最短経路長の場合は求めた経路も出力する.

$dist(x) = 0$;x 以外の節点 x' について、$dist(x') = INF, N' \leftarrow N - |F|$(ただし N は G の頂点数)

$i \leftarrow 0; boolean \leftarrow false$

for $i < N' - 1$ **do**

  for $y \in \{v|v \in V \backslash F\}$ **do**

    for 有向枝 $(y, z), z \notin F, dist(y) \neq INF$ **do**

      if $dist(y) + w(y, z) < dist(z)$ **then**

        $dist(z) \leftarrow dist(y) + w(y, z); parent(z) = y$

      end if

    end for

  end for

  $i = i + 1$

end for

if N' 回目の反復で $dist$ が更新されない **then**

  $path$ ←(z が x に至るまで $parent$ を辿り復元した経路);$boolean \leftarrow true$

end if

output $path, dist(t), boolean$

6.4 限定操作 1 の正当性の証明

限定操作 1 の正当性について証明する。
x から子節点 v_i を経由し t に至る経路のうち本来の最短経路長を l_i とする。この時、$upper, lower$ の決め方から、任意の $i \in \{1, ..., n\}$ について $l_i \leq w(x, v_i) + upper(v_i), lower(v_i) + w(x, v_i) \leq l_i$ が成り立つ。よって、ある $i, j (i, j \in \{1, ..., n\})$ について、$upper(v_i) + w(x, v_i) < lower(v_j) + w(x, v_j)$ が成立する時、$l_i \leq upper(v_i) + w(x, v_i) < lower(v_j) + w(x, v_j) \leq l_j$、つまり $l_i < l_j$ が成り立つ。このことから、s から枝 x, v_j を経由し t に至る経路が s から t への真の最短経路となることはあり得ず、この x からの部分問題において枝 (x, v_j) を通る経路の探索は打ち切っても良いことが示される。

6.5 限定操作 1 の擬似コード

Algorithm 4 $bound1(x, F)$

Input: グラフ G 上の節点 x , 節点 s から x へ至る有向路に含まれる節点集合 F
Output: x の子節点に至る枝を枝刈りする

```
枝刈りする x の子節点を格納する cutNode{}
for 全ての有向枝 (x, y) ∈ E, y ∉ F について do
    y から t へ至る有向路長の上界を関数 Upper(y, F) で計算
    y から t へ至る有向路長の下界を関数 Lower(y, F) で計算
end for
for 全ての有向枝 (x, y_i) ∈ E, y_i ∉ F do
    for 全ての有向枝 (x, y_j) ∈ E, y_j ∉ F ∪ cutNode do
        if w(x, y_i) + Upper(y_i, F) < w(x, y_j) + Lower(y_j, F) then
            y_j に至る枝を枝刈り;cutNode ← y_j
        end if
    end for
end for
```

6.6 限定操作 2 の正当性の証明

x から子節点 v_i を経由し t に至る経路のうち本来の最短経路長を l_i とする。この時、$lower$ の決め方から、任意の $i \in \{1, ..., n\}$ について $lower(v_j) + w(x, v_j) \leq l_i$ が成り立つ。よってその時点での s から t への暫定の最短距離を $minDist$ と表すこととした時、$minDist < sum + w(x, v_j) + lower(v_j)$ が成立する場合、$minDist < sum + w(x, v_j) + lower(v_j) \leq l_i + sum$ が成り立つ。これは s から枝 x, v_j を経由し t に至る経路が s から t への真の最短経路とはなり得ないことを意味するため、x からの部分問題において枝 (x, v_j) を通る経路の探索は打ち切って良いことが示される。

6.7 限定操作 2 の擬似コード

Algorithm 5 $bound2(x, F, sum, minDist)$

Input: グラフ G 上の節点 x , 節点 s から x へ至る有向路に含まれる節点集合 F,s から x へ至る有向路の経路長 sum, その時点での s から t への暫定の最短距離 $minDist$

Output: x から x の子節点に至る枝を枝刈りする

```
for 全ての有向枝 (x, y) ∈ E, y ∉ F について do
  y から t へ至る有向路長の下界を関数 Lower(y, F) で計算
  if minDist < Lower(y, F)+w(x,y)+sum then
    枝 (x, y) を枝刈りする
  end if
end for
```

6.8 限定操作 3 の正当性の証明

v_i から t に至る経路長の下界を求める際、ベルマンフォード法を $N'-1$ 回繰り返す。この時、ベルマンフォード法の性質から、v_i から t までの部分グラフに負閉路がない場合には v_i から t への最短経路およびその経路長が求まっていると言える。よって x からの部分問題において枝 (x, v_i) をとおる経路の探索はこれ以上行う必要がないと言える。具体的には、$sum+w(x,v_i)+Lower(v_i,t)$ が $minDist$ より小さい場合は外部の変数にその経路を格納し $minDist$ に経路長を保存した上で、v_i 以降の探索を打ち切れば良い。

6.9 限定操作 3 の擬似コード

Algorithm 6 $bound3(x, F, sum, minDist, minPath)$

Input: グラフ G 上の節点 x , 節点 s から x へ至る有向路に含まれる節点集合 F,s から x へ至る有向路の経路長 sum, その時点での s から t までの最短経路長 $minDist$, その経路長を与える経路 $minPath$
Output: x から x の子節点に至る枝の枝刈りを行う
  for 全ての有向枝 $(x, y) \in E, y \notin F$ について **do**
    y から t へ至る有向路長の下界を関数 $Lower(y, F)$ で計算
    if y から t への部分グラフに負閉路がない **then**
      if $sum + Lower(y, F) + w(x, y) < minDist$ **then**
        $minDist \leftarrow sum + Lower(y, F) + w(x, y)$
        $minPath \leftarrow F \cup$(求めた下界を与える経路に含まれる節点の集合)
      end if
      枝 (x, y) を枝刈り
    end if
  end for

7 限定操作の擬似コード

以上をまとめた全体の限定操作アルゴリズムを表す擬似コードは以下である。

Algorithm 7 $Bound(x, F, sum, minDist, minPath)$

Input: グラフ G 上の節点 x , 節点 s から x へ至る有向路に含まれる節点集合 F,s から x へ至る有向路の経路長 sum, その時点での s から t までの最短経路長 $minDist$, その経路長を与える経路 $minPath$

Output: x の子節点 y に関して、枝 (x, y) の枝刈りを行う

```
枝刈りする x の子節点を格納する cutNode{}
for 全ての有向枝 (x, y) ∈ E, y ∉ F について do
  y から t へ至る有向路長の上界を関数 Upper(y, F) で計算
  y から t へ至る有向路長の下界を関数 Lower(y, F) で計算
  if y から t への部分グラフに負閉路がない then
    if sum + Lower(y, F) + w(x, y) < minDist then
      minDist ← sum + Lower(y, F) + w(x, y)
      minPath ← F∪(求めた下界を与える経路に含まれる節点集合)
    end if
    cutNode ← y(枝刈り)
  end if
  if minDist < Lower(y, F) + w(x, y) + sum then
    cutNode ← y(枝刈り)
  end if
end for
for 全ての有向枝 (x, y_i) ∈ E, y_i ∉ F do
  for 全ての有向枝 (x, y_j) ∈ E, y_j ∉ F, cutNode do
    if w(x, y_i) + Upper(y_i, F) < w(x, y_j) + Lower(y_j, F) then
      cutNode ← y_j(枝刈り)
    end if
  end for
end for
```

8　全体の擬似コード

以上までで述べたアルゴリズムをまとめた擬似コードは以下である。

Algorithm 8 再帰的関数 $branchAndBound(x, F, sum)$

Input: グラフ G 上の節点 x , 節点 s から x へ至る有向路に含まれる節点集合 F,s から x へ至る有向路の経路長

Output: グラフ G 上の点 s から点 t に至り、かつ節点集合 F を含む有向路の経路およびその経路長. また、暫定の最短路およびその経路長と比較し、場合によっては代入を行う.

枝刈りする x の子節点を格納する $cutNode\{\}$

その時点での暫定の s から t への最短経路は外部の変数 $minPath$ に格納されている

for 全ての有向枝 $(x, y) \in E, y \notin F$ について **do**
  y から t へ至る有向路長の上界を関数 $Upper(y, F)$ で計算
  y から t へ至る有向路長の下界を関数 $Lower(y, F)$ で計算
  if y から t への部分グラフに負閉路がない **then**
    if $sum + Lower(y, F) + w(x, y) < minDist$ **then**
      $minDist \leftarrow sum + Lower(y, F) + w(x, y)$
      $minPath \leftarrow F \cup$(求めた下界を与える経路に含まれる節点集合)
    end if
    $cutNode \leftarrow y$(枝刈り)
  end if
  if $minDist < Lower(y, F) + w(x, y) + sum$ **then**
    $cutNode \leftarrow y$(枝刈り)
  end if
end for
for 全ての有向枝 $(x, y_i) \in E, y_i \notin F$ **do**
  for 全ての有向枝 $(x, y_j) \in E, y_j \notin F, cutNode$ **do**
    if $w(x, y_i) + Upper(y_i, F) < w(x, y_j) + Lower(y_j, F)$ **then**
      $cutNode \leftarrow y_j$(枝刈り)
    end if
  end for
end for
for 全ての有向枝 $(x, y) \in E, y \notin F \cup cutNode$ **do**
  if y=t **then**
    output $F \cup \{y\}, sum + w(x, y)$
    if $sum + w(x, y) < minDist$ **then**
      $minDist \leftarrow sum + w(x, y)$
      $minPath \leftarrow F \cup \{y\}$
    end if
  else
    $branchAndBound(y, F \cup \{y\}, sum + w(x, y))$
  end if
end for

第 V 部
課題 4

ここでは課題 3 で与えた限定操作を実装し、課題 2 と同じグラフを用いて最短路問題を解き、両者の実行時間 (ただし cpu 時間で、単位は msec とする) と探索ノード数を比較した。ただし探索の始点はノード 0、終点はノード $N-1$(N はグラフを構成するノードの数) とした。結果は表とグラフにまとめ、それらに対して考察を行った。

9 課題 4 の回答

以下はノード数 6 のグラフに対して、枝数を 12,14,16,18,20,22,24,26,28,30 として、探索にかかった時間および探索したノード数をまとめたものである。グラフは横軸が枝の本数、縦軸が時間または探索したノード数である。ただし紫の線が限定操作なし、緑の線が限定操作ありの結果である。

表 5: ノード数 6 のグラフ探索にかかった時間と探索ノード数の表. 枝の数ごとに、限定操作あり、なしのそれぞれの結果をまとめてある.

枝の本数	枝刈り	探索ノード数	時間 (msec)
12	あり	1	0.0829697
	なし	9	0.0669956
14	あり	1	0.084877
	なし	11	0.082016
16	あり	1	0.134945
	なし	15	0.111103
18	あり	1	0.135899
	なし	25	0.187159
20	あり	1	0.138044
	なし	25	0.29397
22	あり	1	0.19908
	なし	43	0.336885
24	あり	8	0.746012
	なし	43	0.403881
26	あり	8	0.792027
	なし	43	0.424862
28	あり	9	1.12891
	なし	65	0.628948
30	あり	9	1.45102
	なし	65	0.688076

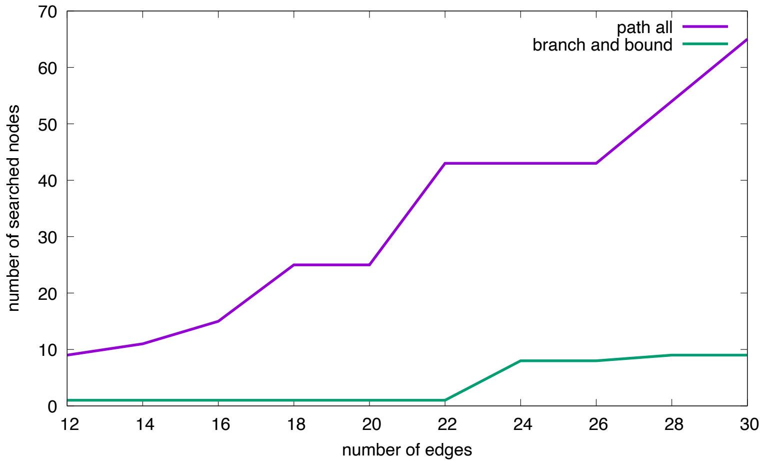

図 9: ノード数が 6 の時のグラフ. 横軸が枝数、縦軸が探索したノードの数.

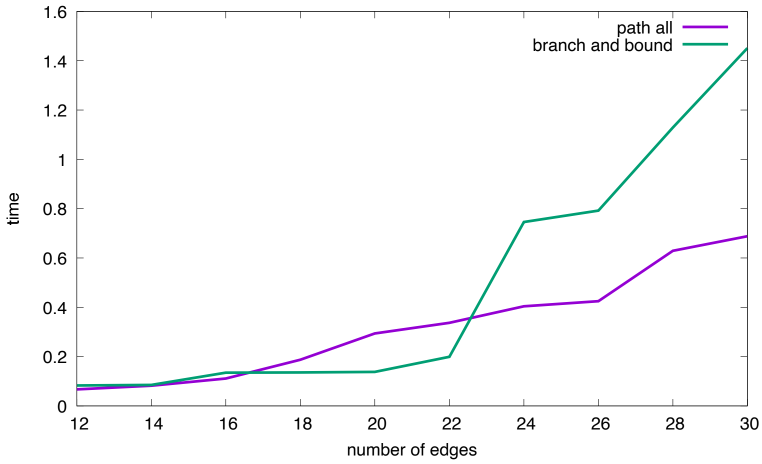

図 10: ノード数が 6 の時のグラフ. 横軸が枝数、縦軸がかかった時間.

以下はノード数 10 のグラフに対して、枝数を 20,30,40,50,60,70,80,90 として、探索にかかった時間および探索したノード数をまとめたものである。グラフは横軸が枝の本数、縦軸が時間または探索したノード数である。ただし紫の線が限定操作なし、緑の線が限定操作ありの結果である。

表 6: ノード数 10 のグラフ探索にかかった時間と探索ノード数の表. 枝の数ごとに、限定操作あり、なしのそれぞれの結果をまとめてある.

枝の本数	枝刈り	探索ノード数	時間 (msec)
20	あり	1	0.0619888
	なし	8	0.0858307
30	あり	20	2.32697
	なし	100	1.17207
40	あり	34	6.20008
	なし	459	6.12497
50	あり	49	10.7648
	なし	1690	17.6229
60	あり	194	49.983
	なし	7047	65.5611
70	あり	436	132.833
	なし	19971	188.124
80	あり	917	308.118
	なし	42070	399.007
90	あり	1370	545.272
	なし	109601	1050.97

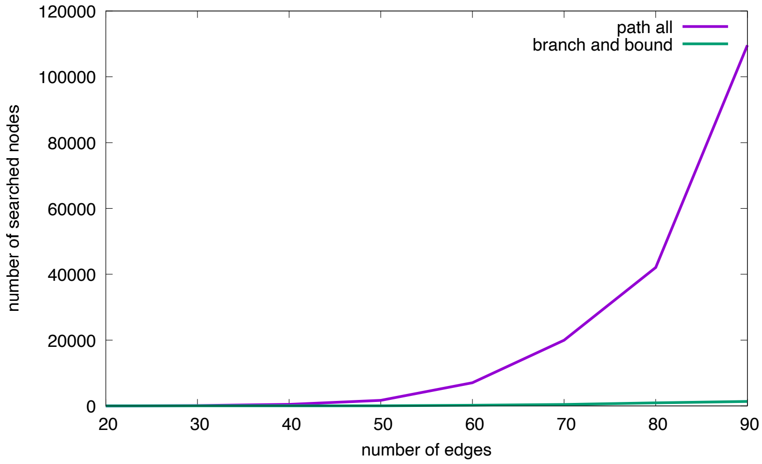

図 11: ノード数が 10 の時のグラフ. 横軸が枝数、縦軸が探索したノードの数.

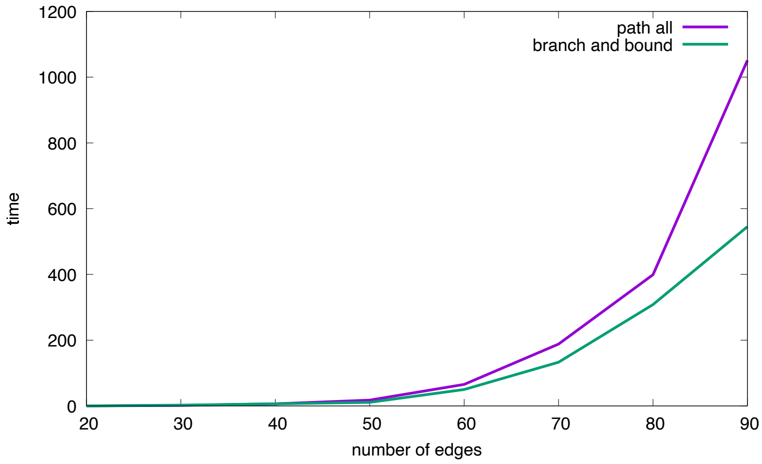

図 12: ノード数が 10 の時のグラフ. 横軸が枝数、縦軸がかかった時間.

以下はノード数 14 のグラフに対して、枝数を 30,40,50,60,...,120 として、探索にかかった時間および探索したノード数をまとめたものである。グラフは横軸が枝の本数、縦軸が時間または探索したノード数である。ただし紫の線が限定操作なし、緑の線が限定操作ありの結果である。

表 7: ノード数 14 のグラフ探索にかかった時間と探索ノード数の表. 枝の数ごとに、限定操作あり、なしのそれぞれの結果をまとめてある.

枝の本数	枝刈り	探索ノード数	時間 (msec)
30	あり	8	1.32513
	なし	30	0.311852
40	あり	66	12.8109
	なし	258	2.70605
50	あり	148	35.4831
	なし	1284	14.0021
60	あり	578	145.075
	なし	9357	93.3471
70	あり	1267	385.964
	なし	42910	447.985
80	あり	4815	1462.48
	なし	135860	1385.51
90	あり	11325	4079.32
	なし	628661	6485.87
100	あり	55093	18306
	なし	1733836	18406.3
110	あり	112038	44583.9
	なし	5437492	60884.5
120	あり	225471	93897
	なし	13019591	150919

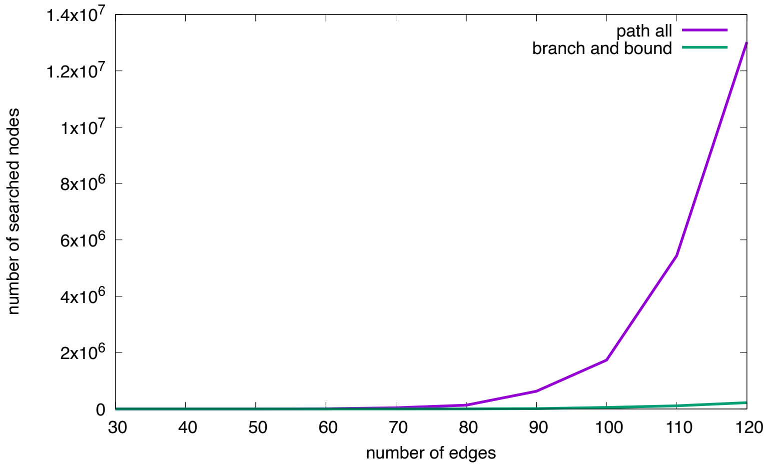

図 13: ノード数が 14 の時のグラフ. 横軸が枝数、縦軸が探索したノードの数.

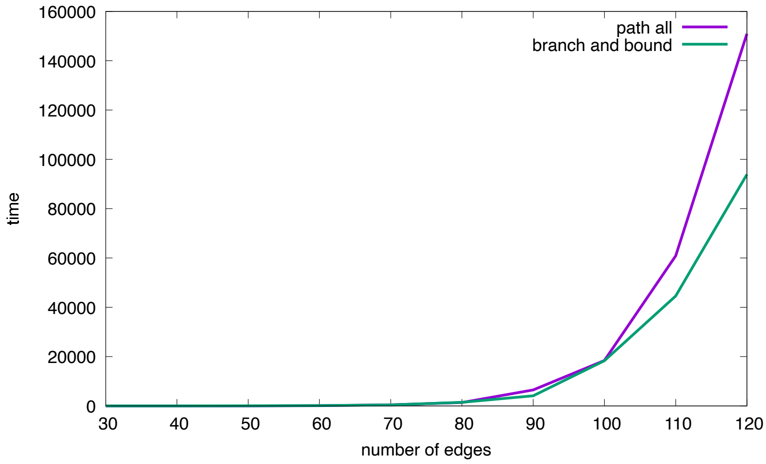

図 14: ノード数が 14 の時のグラフ. 横軸が枝数、縦軸がかかった時間.

以下はノード数 18 のグラフに対して、枝数を 40,60,80,100,120 として、探索にかかった時間および探索したノード数をまとめたものである。グラフは横軸が枝の本数、縦軸が時間または探索したノード数である。ただし紫の線が限定操作なし、緑の線が限定操作ありの結果である。

表 8: ノード数 18 のグラフ探索にかかった時間と探索ノード数の表. 枝の数ごとに、限定操作あり、なしのそれぞれの結果をまとめてある.

枝の本数	枝刈り	探索ノード数	時間 (msec)
40	あり	18	2.80786
	なし	24	0.245094
60	あり	359	108.683
	なし	6579	67.374
80	あり	4652	1897.16
	なし	315507	3493.82
100	あり	63403	32764.8
	なし	8382312	93073.5
120	あり	724101	445814
	なし	133827589	1.57905e+06

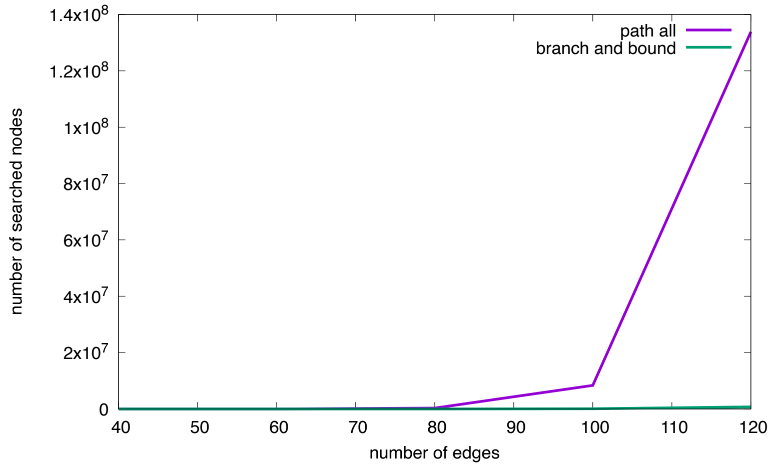

図 15: ノード数が 18 の時のグラフ. 横軸が枝数、縦軸が探索したノードの数.

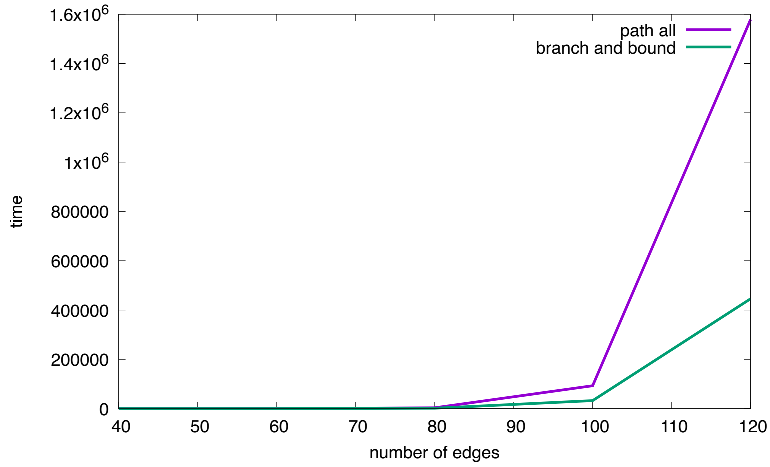

図 16: ノード数が 18 の時のグラフ. 横軸が枝数、縦軸がかかった時間.

10 考察

課題 2 と同様、限定操作を導入した場合についてもグラフが密になるにつれてかかる時間や探索ノード数が増加することが見て取れた。ただし、限定操作を行った場合はそうでない場合に比べて時間、探索ノード数共に少なくて済む傾向にあり、またグラフが密であれば密であるほど限定操作の優位性が顕著に現れる結果となった。

しかし、ノードの数が 6 の場合および枝の数がノードの数に対して少ない場合 (グラフが疎である場合) については限定操作を導入した場合の方が探索に時間がかかる傾向にあった。これは限定操作を導入した場合には調べないパスを全て調べるのにかかる時間と比べて、限定操作を行うのにかかる時間の方が長くなることに起因すると推察する。限定操作にはベルマンフォード法が含まれており、最大計算量は $O(VE)$ である。これを探索するノードの子節点の数だけ行うため、ノードの数に対して枝の本数が少なければ当然限定操作にかかる時間はパスを調べるのにかかる時間より長くなることが理論的にも説明できる。また、ノードの数に対して枝の本数が多くなれば多くなるほど、探索しなければならないパスの本数は増加する。この時、ベルマンフォード法にかかる時間と比べパスを調べる時間の方が長くなり、限定操作の導入による優位性が探索時間の面にも現れるようになることが説明できる。

11　課題 4 のコード

コード 2: 課題 4 のコード

```
#include <vector>
#include <iostream>
#include <fstream>
#include <cmath>
#include <algorithm>
#include <time.h>
#include <sys/time.h>
#include <queue>
using namespace std;

const int INF = pow(10,6);
int min_weight = INF, node_num = 0, s, t;
vector<int> minPath, F_;

//時間を計る関数
double gettimeofday_sec(){
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return tv.tv_sec+(double) tv.tv_usec*1e-6;
}

// 緩和を実施する関数
bool chmin(int a, int b) {
    if (a > b) {
        return true;
    }
    else return false;
}

//枝を表す
struct Edge {
    int to; // 隣接頂点番号
    int w; // 重み
    Edge(int to, long long w) : to(to), w(w) {}
};

```

```
//ベクトルにある要素が含まれているかどうか
bool included(int a, vector<int> vec){
    bool include = false;
    for(int i=0; i<vec.size(); i++){
        if(vec[i]==a){
            include = true;
            break;
        }
    }
    return include;
}

//x から t への部分問題において上界を計算
int upperBound(int x, int t, vector<int> F, vector<vector<Edge> > G, int sum, int N){
    if(x == t) return 0;
    vector<int> dist(N, INF);
    dist[x] = 0;
    vector<int> real_dist(N, INF);

    // (d[v], v) のペアを要素としたヒープを作る
    priority_queue<pair<int, int>,
                vector<pair<int, int> >,
                greater<pair<int, int> > > que;
    que.push(make_pair(dist[x], x));

    // ダイクストラ法の反復を開始
    while (!que.empty()) {
        // v: 使用済みでない頂点のうちd[v] が最小の頂点
        // d: v に対するキー値
        int v = que.top().second;
        if(v == t) break;
        int d = que.top().first;
        que.pop();

        // d > dist[v] は，(d, v) がゴミであることを意味する
        if (d > dist[v]) continue;

        // 頂点v を始点とした各辺を緩和
        for (auto e : G[v]) {
            if(included(e.to, F)) continue;
            if (chmin(dist[e.to], dist[v] + max(e.w,0))) {
                // 更新があるならヒープに新たに挿入
                dist[e.to] = dist[v] + max(e.w, 0);
                real_dist[e.to] = real_dist[v] + e.w;
                que.push(make_pair(dist[e.to], e.to));
            }
        }
    }
    return real_dist[t];//上界を返す
}

//下界、負閉路が存在しないかどうか、負閉路が存在しない時の経路最短路 ()
struct forLowerBound {
    int distance;
    bool canCut;
    vector<int> path;
} typedef forLowerBound;

//下界を求めるための関数.
forLowerBound lowerBound(int x, int t, vector<int> F, vector<vector<Edge> > G, int sum, int N){
    //cout << xに対する<<"が実行されたよ lower" << endl;
```

```
    vector<int> path;
    vector<int> prev(N, -1);
    if(x==t){
        forLowerBound a;
        a.distance = 0;
        a.canCut = true;
        a.path = path;
        return a;
    }

    bool exist_negative_cycle = false; // 負閉路をもつかどうか
    vector<int> dist(N, INF);
    dist[x] = 0;
    bool can_cut = false;
    for (int iter = 0; iter < N-F.size()+1; iter++) {
        bool update = false; // 更新が発生したかどうかを表すフラグ
        for (int v = 0; v < N; ++v) {
            // dist[v] = INF のときは頂点v からの緩和を行わない
            if (dist[v] == INF) continue;
            if(v != x && included(v, F)) continue;

            for (auto e : G[v]) {
                // 緩和処理を行い，更新されたらupdate をtrue にする
                if (chmin(dist[e.to], dist[v] + e.w) && !included(e.to, F)) {
                    dist[e.to] = dist[v] + e.w;
                    update = true;
                    prev[e.to] = v;
                }
            }
        }

        // 更新が行われなかったら，すでに最短路が求められている
        if (!update) break;

        // N-F.size() 回目の反復で更新が行われたならば，負閉路をもつ
        if (iter == N-F.size() && update){
            exist_negative_cycle = true;
            break;
        }
    }
    //負閉路を持たない場合、経路を復元
    if(exist_negative_cycle==false && dist[t]!=INF){
        can_cut = true;
        int t_ = t;
        for(; t_!=-1; t_=prev[t_]) path.push_back(t_);
        reverse(path.begin(), path.end());
    }
    forLowerBound a;
    a.distance = dist[t];
    a.canCut = can_cut;
    a.path = path;
    return a;
}

//メイン関数の返す型
struct forBranchAndBound {
    int distance;
    int node;
    vector<int> path;
} typedef forBranchAndBound;

//枝刈りを行う
```

```
void branch_and_bound(int x, int t, vector<int> F, vector<vector<Edge> > G, int sum, int N){
    node_num += 1;
    vector<int> upper_vec;
    vector<int> lower_vec;
    vector<Edge> search_edge;
    vector<forBranchAndBound> edge_cut_vec;
    for(auto e : G[x]){
        int y = e.to;
        if(included(y, F)) continue;
        vector<int> F_tmp;
        copy(F.begin(), F.end(), back_inserter(F_tmp));
        F_tmp.push_back(y);
        forLowerBound lower_tmp = lowerBound(y, t, F_tmp, G, sum, N);

        //下界、上界を計算
        int lower = lower_tmp.distance + e.w;
        int upper = upperBound(y, t, F_tmp, G, sum, N) + e.w;

        //下界を求めた際、負閉路がなかった場合
        if(lower_tmp.canCut){
            upper = lower;
            forBranchAndBound b;
            b.distance = lower;
            b.node = y;
            b.path = lower_tmp.path;
            edge_cut_vec.push_back(b);
        }
        upper_vec.push_back(upper);
        lower_vec.push_back(lower);
        search_edge.push_back(e);
    }

    vector<int> erase_num;//枝刈りする枝の行き先

    for(int i=0; i<upper_vec.size(); i++){
        for(int j=0; j<upper_vec.size(); j++){
            if(upper_vec[i] < lower_vec[j] || min_weight<lower_vec[j]+sum){
                erase_num.push_back(j);
            }
        }
    }

    int k=-1;
    //調査する
    if(search_edge.size()!=0){
        for(auto e : search_edge){
            int y = e.to;
            k += 1;
            bool include = false;

            //枝刈りするかどうか
            int I;
            for(int i=0; i<edge_cut_vec.size(); i++){
                if(edge_cut_vec[i].node == y){
                    include = true;
                    I = i;
                }
            }

```

```
            //下界がその部分問題の最短経路長である場合、枝刈りして暫定の最短距離と比較
            vector<int> F_copy;
            if(include){
                copy(F.begin(), F.end(), back_inserter(F_copy));
                F_copy.insert(F_copy.end(), edge_cut_vec[I].path.begin(), edge_cut_vec[I].path.end());
                int sum_tmp = edge_cut_vec[I].distance+sum;
                if(sum_tmp < min_weight){
                    min_weight = sum_tmp;
                    minPath.clear();
                    copy(F_copy.begin(), F_copy.end(), back_inserter(minPath));
                }
                continue;
            }

            //上界 < 下界の時、その下界を与える部分問題を枝刈り
            if(included(k, erase_num)) continue;

            int sum_tmp;
            copy(F.begin(), F.end(),back_inserter(F_copy));
            sum_tmp = sum + e.w;
            F_copy.push_back(y);
            //暫定の最短経路長と比較
            if(y == t){
                if(sum_tmp < min_weight){
                    min_weight = sum_tmp;
                    minPath.clear();
                    copy(F_copy.begin(), F_copy.end(), back_inserter(minPath));
                }
            }
            //再帰的に関数を呼び出す
            else{
                branch_and_bound(y, t, F_copy, G, sum_tmp, N);
            }
        }
    }
}

//実行
int main(){
    string filename("Graphs/n_18/n_18_m_120.txt");
    int number;

    ifstream input_file(filename);
    if (!input_file.is_open()) {
        cerr << "Could not open the file - '"<< filename << "'" << endl;
        return EXIT_FAILURE;
    }

    //ファイルからグラフ情報を読み取る
    int i=0;
    vector<int> From, To;
    vector<long long> W;
    int N, M;
    while (input_file >> number) {
        i++;
        if(i==1) N = number;
        else if(i==2) M = number;
        else{
```

```
                if (i%3==0) From.push_back(number);
                if (i%3==1) To.push_back(number);
                if (i%3==2) W.push_back(number);
            }
        }
        input_file.close();

        vector<vector<Edge> > G(N);
        vector<vector<Edge> > G_rev(N);

        //グラフを構成
        for(int k=0; k<M; k++){
            int from = From[k];
            int to = To[k];
            int w = W[k];
            G[from].push_back(Edge(to, w));
            }
        s = 0;
        t = N-1;
        F_.push_back(s);

        //出力
        double start = gettimeofday_sec();
        branch_and_bound(s,t,F_,G, 0,N);
        double end = gettimeofday_sec();
        std::cout << "実行にかかった時間は
              ␣" << (end-start)*1000 << "␣msec"<< endl;
        std::cout << "探索したノードの数は" << node_num <<endl;

        std::cout << "最短経路は" << endl;
        for(int i=0; i<minPath.size(); i++){
            if(i != minPath.size()-1) std::cout << minPath[i] << "␣->␣";
            else{
                std::cout << minPath[i]<< endl;
            }
        }
        std::cout << "その距離は␣" << min_weight << endl;
}
```

第 VI 部
このレポートのまとめ

このレポートでは分枝限定法を用いて負閉路を持つグラフに関する最短経路問題を解くアルゴリズムを実装した。限定操作を行わない場合 (課題 1,2) と行った場合 (課題 3,4) で同じ問題を解き、かかった時間および探索したノードの数を比較した。結果は表とグラフにまとめ、それらに対して考察を行った。結果、限定操作を行った場合は行わない場合と比べて探索ノード数が減少することがわかった。また、実行時間に関しても、ノードに対してエッジの数が増えるほど限定操作の有効性が顕著に現れるという結果となった。

参考文献

[1] 京都大学工学部情報学科配当科目「グラフ理論」講義資料

[2] 大槻兼資、秋葉拓哉「アルゴリズムとデータ構造」