第7章　組合せ最適化

7.1　目的

組合せ最適化問題（離散最適化問題）とは，順序や割当のように「組合せ的な構造」によって解が表現される最適化問題を指す．現実の多くの場面において自然に現れる問題であるが，問題の構造をうまく捉えなければ効率よく解くことは難しい．本実験の目的は**最短経路問題**を通して組合せ最適化問題の難しさを体感し，代表的な解法の一つである分枝限定法について学ぶことである．

7.2　準備

実数の集合を $\mathbb{R}$ とする．節点集合 $V = \{1, 2, \ldots, n\}$ と有向枝集合 $E \subseteq V \times V$ から成る有向グラフ $G = (V, E)$ が与えられたとする．任意の二節点 $s, t \in V$ について，節点の系列 $P = \langle v_1 = s, v_2, \ldots, v_k = t \rangle$ が

(条件 1) $(v_i, v_{i+1}) \in E,\ i \in \{1, 2, \ldots, k-1\}$

を満たすとき，P を s から t への**有向路** (あるいは単に**路**[1]) と呼ぶ．また $s = v_1$ を P の**始点**，$t = v_k$ を P の**終点**と呼ぶ．さらに P が

(条件 2) すべての節点 $v_1, v_2, \ldots, v_k$ が相異なる

とき，P を**単純有向路** (あるいは単に**単純路**) と呼ぶ．ただし $v_1 = v_k$ のみ成り立つとき，これを**閉路**と呼ぶ．

有向グラフ $G = (V, E)$ における各枝 $(u, v) \in E$ の長さ $d(u, v) \in \mathbb{R}$ が与えられたとき，s から t への路 $P = \langle v_1 = s, v_2, \ldots, v_k = t \rangle$ に含まれる枝の長さの総和を路の**長さ**と呼び，$d(P)$ と書く．すなわち $d(P)$ は，

$$d(P) \triangleq \sum_{i=1}^{k-1} d(v_i, v_{i+1})$$

と定義される．二つの節点 s, t をそれぞれ始点，終点とするような路のうち，長さが最小となるようなものを s から t への**最短経路**と呼ぶ．

[1] **道**と呼ぶ場合もある．

7.3　最短経路問題

有向グラフと枝の長さが与えられたとき，最短経路問題は文字通り最短経路を問う問題だが，始点と終点の指定の仕方によって以下の四通りの問題が考えられる．

- 単一頂点対最短経路問題: 指定された始点と終点の間の最短経路を問う問題．
- 単一始点最短経路問題: 始点のみが指定され，始点からそれ以外の節点すべてへの最短経路を問う問題．
- 単一終点最短経路問題: 終点のみが指定され，終点を除くすべての節点を始点として最短経路を問う問題，
- 全点対最短経路問題: すべての二節点間について最短経路を問う問題．

最短経路問題は乗り換え検索システムやカーナビの設計，PERT 図などのタスクスケジューリングに応用を持つ．**本実験では単一頂点対最短経路問題を取り扱う．**

最短経路問題を考えるとき，枝の長さを非負の値に限定することもあるが，**本実験では枝の長さが負の値を取ることも許容する．**すると長さ $d(C)$ が負になるような閉路 C が存在し得る（そのような閉路を**負閉路**と呼ぶ）．この場合最適解は存在しない．なぜならその C を繰り返し回ることで，路の長さをいくらでも小さくできてしまうからである．そこで**本実験では，単純路に限定して最短経路問題を考える**ことにする．すなわち始点と終点が与えられたとき，(条件 1), (条件 2) の両方を満たす節点の系列のうち，長さが最小となる

ものを問う．この問題は一般に NP 困難 [1] であることが知られている[2]．本実験では，この問題に対して分枝限定法に基づいた厳密解法を設計して実装し，簡単な性能評価を行う．

メモ． 単一始点最短経路問題に関して，枝の長さが非負に限定されている場合，ダイクストラ法を用いれば $O(m + n \log n)$ 時間で最適解を得ることができる（ただし $m = |E|, n = |V|$）．また枝の長さが負の値を取ることを許容し，長さが最小となる路を問う場合，ベルマン・フォード法を用いれば $O(nm)$ 時間で以下を行うことができる :「負閉路が存在するならばその旨を出力し，そうでなければ最適解を出力」．詳しくは [2, 5] などを参照せよ．

7.4 分枝限定法

分枝限定法は，組合せ最適化問題全般に用いることのできる厳密解法（厳密な意味での最適解を求める解法）の枠組みである．その動作原理は，組織的な場合分け（分枝）に基づく列挙法に，不要な場合の排除機能（限定）を付したものとなっている．

本稿では [4] に基づく説明を示す．興味のある者は [2, 3, 5] なども参照されたい．

分枝限定法は，問題をいくつかの小規模な問題に分割し，その全てを解くことで等価的に元の問題を解くという考え方に基づいている．小規模な問題への分割は，例えば，ある一つの決定変数の値を固定し（グラフ探索の場合，枝や節点を決め），それぞれの場合を個別に考察することによって実現できる．このように問題を分割する操作を**分枝操作**（branching operation）という．分枝操作を繰り返し行うことで，すべての場合を列挙することができるが，その過程は生成木と呼ばれる根付き木を用いて表現できる．生成木において，根は元の問題に対応し，その二つの子はそれぞれある変数の値を 0 または 1 に固定した 2 つの場合に対応する．その他の節点も同様である．よって，内部の節点は根からその節点への路に対応していくつかの変数の値を 0 か 1 に固定した問題に対応し，葉は全ての変数の値が定まった解に対応する．一部の変数の値が固定された問題を**部分問題**（partial problem）と呼び，この例の場合，部分問題も元の問題と等しい構造である．

分枝限定法では，実際に全ての葉を列挙するわけではなく，その一部分だけを生成する．生成木のうち実際に生成された節点のみから成るものを**探索木**（search tree）という．ある部分問題（節点）に対して，

(i) その最適値，

(ii) その部分問題が実行不可能であること，

(iii) その部分問題の最適解が元の問題の最適解とはならないこと，

のいずれかが分かれば，その部分問題をこれ以上調べる必要はなく，その下の節点（子孫と呼ぶ）の探索を省略できる（**終端する**（terminate）という）．これを**限定操作**（bounding operation）と呼ぶ．探索の過程で，まだ終端されていない部分問題を活性（active）であるといい，全ての部分問題が終端されたとき，つまり活性部分問題がなくなったとき，分枝限定法は厳密な最適解を与える（もしくは実行可能解が存在しないことを証明する）．

限定操作の代表例である**下界値テスト**について簡単に述べる．近似解法等を用いて実行可能解を一つでも得ることができれば，その目的関数値は最適値の上界（upper bound）値となる．また，探索が生成木の葉に到達したときにも実行可能解が得られる場合がある．これまでの探索で得られている最良の上界値を暫定値（incumbent value）という．一方，制約を緩めた緩和問題を解くなどして得られた最適値は，元の問題の最適値の下界（lower bound）値となる．このようにして得られた上下界値を用い，「ある部分問題の下界値が暫定値以上であれば，その子孫を調べる必要はない」ことが結論できる．その部分問題の子孫を全て調べても，暫定値よりもよい目的関数値を持つ解は見つからないからである．以上が下界値テストで

[2] ある問題が NP 困難であるとき，その問題には多項式時間アルゴリズムは存在しないと予想される．

ある.(最大化問題の場合は上界と下界の役割が入れ替わる.)

なお，計算時間の都合などにより活性部分問題が残っている状態で探索を終了し，その時点での暫定値を出力する方法もしばしば用いられる．このとき，出力した値が最適である保証はなく，分枝限定法を近似解法として用いることになるが，この場合でも，活性部分問題の下界値の最小値が元の問題の下界値となり，出力値の精度を測る指標としてこれを用いることができる．

7.5 最短路問題に対する分枝限定法

有向グラフ $G = (V, E)$，枝重み $d : E \to \mathbb{R}$，始点 $s \in V$ および終点 $t \in V$ $(s \neq t)$ が与えられたとする．以下では始点 s から終点 t への最短路を求めるための分枝限定法を設計するが，本節では s から t へ至る有向路をすべて列挙するためのアルゴリズムを示す．次節において，そのアルゴリズムを実装して最短路を求めることと，限定操作の導入によって計算を高速化することを実験課題とする．

単純路 $P = \langle v_1, v_2, \ldots, v_k \rangle$ の最後に節点 v を追加する操作を $P.\mathsf{push}(v)$，$P = \langle v_1, v_2, \ldots, v_k \rangle$ の最後尾の節点を削除する操作を $P.\mathsf{pop}()$ のように書くことにする．この二つの操作の結果，P はそれぞれ $P = \langle v_1, v_2, \ldots, v_k, v \rangle$，$P = \langle v_1, v_2, \ldots, v_{k-1} \rangle$ となる．このような操作は，単純路をスタックによって表現することで実現できる．また文脈に応じて単純路 P を非順序集合とみなし，その記法を用いる．たとえば，節点 $v \in V$ が P に属することを $v \in P$，属さないことを $v \notin P$ と表す．また空の単純路を $\emptyset$ で表す．

始点 s から終点 t へ至るすべての単純路を得るには，アルゴリズム 1 の擬似コードで示される再帰アルゴリズムにおいて OutputAllPaths$(\emptyset, s)$ を実行すればよい．

7.6 実験課題

あとに述べる注意事項をよく読んだ上で，次の課題 1 から課題 4 に取り組むこと．

アルゴリズム 1 OutputAllPaths(P, u)

入力: 単純路 P および節点 $u \in V \setminus P$;
出力: 単純路 P に属する節点を順に訪れた後に u を訪れ，やがて t へ至るすべての単純路.

```
1: P.push(u);
2: if u = t then
3:    P を出力
4: else
5:    for each (u, v) ∈ E, v ∉ P: do
6:       OutputAllPaths(P, v)
7:    end for
8: end if
9: P.pop()
```

課題 1
有向グラフ $G = (V, E)$，節点 $s, t \in V$ 及び各枝 $(u, v) \in E$ の長さ $d(u, v)$ が与えられたとき，点 s から点 t へ至る最短路を求める分枝アルゴリズムを設計し，その擬似コードを書け．アルゴリズム 1 を用いて設計するとよい．

課題 2
課題 1 で設計したアルゴリズムを実装し，グラフの節点数及び枝数に対して実行時間及び探索したノード数を調べ，表にまとめよ．

- 節点数の目安はおおむね 5 から 15 程度だが，数十分経っても計算が終わらないなど，実行時間があまりに長すぎる場合は適当な基準を設けて計算を打ち切ること．
- 枝をどこに張るか，および枝の重みは疑似乱数を使って決定するとよい．この場合，乱数によって計算結果が変わり得るので，各節点数に対して最低でも五回程度の計算を行うこと．表にまとめるべき実行時間やノード数は，その平均と標準偏差を示すこと．
- 枝の張り方について，ちょうど m 本の枝をランダムに張る方法として，すべての可能な有向枝の集合 $\{(1,2),(2,1),(1,3),\ldots,(n-1,n),(n,n-1)\}$ をシャッフルし (n は節点数)，最初の m 本を採用する，という手順が考えられる．このとき始点 s と終点 t の間

に路の存在を保証したいのであれば，必ず有向枝 (s,t) を含めるようにするとよい．

- 枝の重みは，適当な上限と下限を定めた上でランダムに決めるとよい．

課題 3
課題 1 で設計したアルゴリズムに関して限定操作を導入し，その妥当性を説明せよ．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書け．

課題 4
限定操作を実装し，実行時間及び探索したノード数を課題 2 の実装によるものと比較せよ．結果を表にまとめよ．

注意事項 (1) 擬似コード　擬似コードを書く時以下の点に注意すること：

- 代入 (substitution) には,「:=」を使う (「←」を使うこともある)．
- **goto** 文又は **break** 文は使わないようにする．
- 構造を分かりやすくするため，
  if ... **then** ... **end if**
  if ... **then** ... **else** ... **end if**
  for ... **do** ... **end for**
  while ... **do** ... **end while**
  の形を守り，字下げ (indentation) を深さに応じて使う．とくに入れ子構造が分かりやすくなるようにする．
- 関数の返す出力には **return**，印字・ファイルへの出力には **output** を使う．
- 連続するコマンドは「;」 で区切る（機械的にコマンドの終わりに付けるのではない)．

注意事項 (2) プログラムの実行時間の計測方法　ここでは，`gettimeofday()` を使用する方法を説明する．`gettimeofday()` を使用するにはプログラムの最初で以下のように記述し，`time.h` と `sys/time.h` をインクルードし，関数 `gettimeofday_sec()` を定義する必要がある．

```
#include <time.h>
#include <sys/time.h>

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

下の記述のように実行時間を計測したい操作の前後で `gettimeofday_sec()` を呼び出して値を保存すると，その差に 1000 をかけたものが実行時間（ミリ秒）になる．

```
/* 時間計測開始 */
double start =
gettimeofday_sec();

(計測したい操作の記述)

/* 時間計測終了 */
double end = gettimeofday_sec();

printf("time = %lf msec.\n",
(end-start)*1000);
```

この方法で計測される時間は実時間ではなく，プロセスによって消費された CPU 時間であるので，計算機で同時に実行されている他のプロセスによる影響は受けない．

参考文献

[1] M. R. Garey, D. S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W. H. Freeman, 1979.

[2] 浅野孝夫, 情報の構造 (上) (下), 日本評論社, 1994.

[3] 茨木俊秀, C によるアルゴリズムとデータ構造（改訂 2 版）, オーム社, 2019.

[4] 柳浦睦憲，野々部 宏司，分枝限定法— さらなる計算効率の希求 —,「システム／制御／情報」 第 50 巻 第 9 号，350-356，2006.

[5] T. コルメン，C. ライザーソン，R. リベスト，C. シュタイン，アルゴリズムイントロダクション第 3 版 総合版，近代科学社, 2013.

[Aleksandar Shurbevski, 原口 和也]