# 数理工学実験レポート

第 7 章（連続最適化）

学籍番号 1029366161　　中塚一瑳

2026 年 1 月 8 日

## 目次

# はじめに

今回は、分枝限定法を用いて有向グラフ上の最短単純路問題を解くアルゴリズムの設計、実装、評価を行った。課題 1,2 では分枝アルゴリズムの設計とその実装および実行時間の測定を行い、課題 3,4 では限定操作（下界値テスト）の設計とその組込み、さらに限定操作付きアルゴリズムの実装と実行時間の比較を行った。

# 1　課題 1：最短単純路を求める分枝アルゴリズムの設計

## 1.1　課題の概要

有向グラフ $G = (V, E)$、始点 $s$、終点 $t$、各有向枝 $(u, v) \in E$ の長さ $d(u, v)$ が与えられたとき、**単純路（同一節点を 2 回通らない路）**に限定して、$s$ から $t$ へ至る路のうち路長の総和が最小となるもの（最短路）を求める。

本課題では、単純路を列挙する再帰アルゴリズム（アルゴリズム 1）を基礎として、**探索木を分枝（branching）により生成し、葉（$t$ 到達）で解を評価して最良解を更新する**厳密解法を設計する。

## 1.2　アルゴリズム設計（分枝の考え方）

アルゴリズム 1 は「現在の単純路 $P$ の末尾に節点を追加し、到達可能な次節点へ再帰的に進む」ことで、始点から終点へ至る**すべての単純路**を列挙する。[1] これを最短路探索に拡張するため、以下を追加する。

- 探索中の経路 $P$ をスタックとして保持する（`P.push`, `P.pop`）。
- 現在までの累積路長を $L$ として引数で持ち回る。
- $u = t$ に到達した時点で、累積路長 $L$ を用いて暫定最良値（incumbent）を更新する。
- 全探索終了時点で保持している最良経路が最短単純路となる。

本課題 1 では限定操作（枝刈り）を要求していないため、**分枝により探索木を全て生成する**（＝全単純路列挙に基づく）設計とする。

## 1.3　擬似コード

以下に、最短路を求める手続き `ShortestSimplePath` と、再帰探索手続き `Explore` を示す。

**Algorithm: ShortestSimplePath**$(G = (V, E),\ d,\ s,\ t)$
**Input:** 有向グラフ $G = (V, E)$, 枝長 $d : E \to \mathbb{R}$, 始点 $s$, 終点 $t$
**Output:** $s$ から $t$ への最短単純路 `bestPath` とその長さ `bestValue`

1. $P := \emptyset$
2. `bestValue` $:= +\infty$
3. `bestPath` $:= \emptyset$
4. `Explore`$(P, s, 0)$

5. **return** (`bestPath`, `bestValue`)

**Procedure: Explore**($P$, $u$, $L$)
**Input:** 単純路 $P$, 現在節点 $u \in V \setminus P$, 現在までの累積長 $L$
**Effect:** $P$ を前置として $u$ から始める分枝探索を行い、最良解を更新する

1. $P$.`push`($u$)
2. **if** $u = t$ **then**
   (a) **if** $L <$ `bestValue` **then**
      (i) `bestValue` $:= L$
      (ii) `bestPath` $:=$ `Copy`($P$)
3. **else**
   (a) **for each** $(u, v) \in E$, $v \notin P$ **do**
      (i) `Explore`($P, v, L + d(u, v)$)
4. $P$.`pop`()
5. **return**

ここで `Copy`($P$) は、探索中に $P$ が変更されても最良経路を保持できるように $P$ **の複製**を作る操作を表す。`output` は本アルゴリズムでは用いず、関数の出力は `return` により行う（規則に従う）。

# 2 課題 2：アルゴリズムの実装と実行時間の測定

## 2.1 実装

課題 1 で設計した分枝アルゴリズム（全単純路列挙に基づく最短路探索）を Python で実装した。探索は深さ優先探索（DFS）で行い、現在の経路 $P$ をスタックとして保持する。次に進む候補頂点 $v$ について「$v \notin P$（経路上で未訪問）」を満たすもののみに再帰的に進むことで単純路制約を満たす。

探索ノード数は、DFS の再帰呼び出し回数（すなわち「探索木において生成された部分問題の数」）として定義し、DFS 関数に入るたびにカウンタを 1 増やすことで計測した。実行時間は `time.process_time()` により CPU 時間を測定し、ミリ秒に変換した。

グラフは節点数 $n$ と枝数 $m$ を指定して疑似乱数で生成した。候補となる有向枝（自己ループ除く）の集合をシャッフルし、その先頭から $m$ 本を採用することで「ちょうど $m$ 本の枝をランダムに張る」手続きを実現した。始点 $s = 0$、終点 $t = n - 1$ とし、$(s, t)$ が含まれない場合は追加して路の存在を保証した。枝重みは整数の一様乱数（例：$[-10, 10]$）で与えた。乱数によるばらつきが生じるため、各 $(n, m)$ について 5 回試行し、実行時間と探索ノード数の平均および標準偏差を算出した。

## 2.2 結果

枝数は $m = \frac{n(n-1)}{2}$ とし、$n = 5$ から 15 まで計測した。結果を表 1 に示す。

節点数が増加するにつれて探索ノード数が急増し、それに伴って実行時間も急激に増大した。特に $n = 12$ 以降で増加が顕著であり、$n = 15$ では探索ノード数が約 $7.4 \times 10^6$、実行時間が約 4.6 秒となった。

**表 1:** 実行時間と探索ノード数（平均 ± 標準偏差、試行回数 5）

| $n$ | time_mean (ms) | time_std (ms) | nodes_mean | nodes_std |
|---|---|---|---|---|
| 5 | 0.00 | 0.00 | 7.4 | 2.2 |
| 6 | 0.00 | 0.00 | 18.4 | 3.6 |
| 7 | 0.00 | 0.00 | 34.8 | 14.8 |
| 8 | 0.00 | 0.00 | 119.8 | 51.8 |
| 9 | 0.00 | 0.00 | 414.0 | 251.0 |
| 10 | 3.12 | 6.25 | 1276.6 | 215.5 |
| 11 | 3.12 | 6.25 | 6103.2 | 1687.0 |
| 12 | 15.62 | 0.00 | 34643.0 | 13605.0 |
| 13 | 109.38 | 46.35 | 160508.6 | 65819.5 |
| 14 | 534.38 | 142.32 | 820563.0 | 248673.9 |
| 15 | 4556.25 | 1465.38 | 7437024.2 | 2220241.9 |

## 2.3 考察

### 2.3.1 探索ノード数が急増する理由

本実装は「単純路をすべて列挙し、その中で最短を選ぶ」方式であり、探索量はグラフ構造に強く依存する。今回 $m = \frac{n(n-1)}{2}$ と比較的密なグラフを用いたため、各頂点からの分岐数が大きくなりやすい。単純路の候補は「次に進める未訪問頂点」の選択の組合せで増えるため、節点数が増えると探索木の葉（候補経路）の数が指数的に増大する。表 1 でも $n$ の増加に対して nodes_mean が桁を跨いで増えており、全探索が現実的でなくなる様子が定量的に確認できる。

### 2.3.2 実行時間と探索ノード数の関係

DFS 呼び出し回数（探索ノード数）は「列挙過程で実際に生成した部分問題数」に対応しており、実行時間は概ね探索ノード数に比例して増加すると考えられる。実測でも $n = 10$ から 15 で nodes_mean が約 $1.3 \times 10^3 \to 7.4 \times 10^6$ と増えるにつれて、time_mean も 3.12ms → 4556.25ms と大きく増加している。したがって、計算時間のボトルネックは主に探索量そのもの（列挙した部分問題数）であると言える。

# 3 課題 3：限定操作の設計とアルゴリズムへの組込み

## 3.1 限定操作の方針

課題 1 で設計した全単純路探索（分枝）に対し，本課題では限定操作として **下界値テスト**を導入する。すなわち，探索途中の部分問題（探索木の節点）に対して，下界が暫定最良値（incumbent）以上であるとき，その部分問題の子孫探索を省略（終端）することで探索木を縮小する。[1]

## 3.2 下界（LB）の定義：事前計算による弱い下界（辺数制限付き）

単純路では同一節点を 2 回訪れないため，始点から終点までに使用できる辺数は高々 $|V|-1$ である。この点を下界にも反映させるため，本課題では各節点 $v \in V$ に対し

$$\mathrm{LB}_k[v] \quad (k = 0, 1, \ldots, |V|-1)$$

を次の意味で定義する：

> $\mathrm{LB}_k[v]$：「**単純路制約を無視**した緩和問題において，$v$ から $t$ へ **高々 $k$ 本の辺**で到達する歩道（頂点重複を許す）を考えたときの最短距離」。

単純路制約を無視することは制約緩和であるため，任意の単純路の残余コストは $\mathrm{LB}_k[v]$ 以上となり，$\mathrm{LB}_k$ は元の問題に対する下界として安全である。

### 3.2.1 （正規の Bellman–Ford 法との違い：負閉路検出の省略）

本課題では枝の長さが負でもよいので，$\mathrm{LB}_k$ の計算は Bellman–Ford 法の反復緩和と同型の手続きで行える。ただし **正規の Bellman–Ford 法が行う負閉路検出は本実装では省略する**。理由は以下の通りである。

- 元問題は単純路であり，使用可能な辺数は高々 $|V|-1$ である。本課題の下界も $\mathrm{LB}_k$ として**辺数を $k$ で制限して定義**しているため，閉路を何度でも回って距離を無限に下げる状況（$-\infty$）は，この $k$ 制限の枠内では本質的に問題になりにくい。
- 下界値テストで必要なのは「過小評価（楽観的）な下界」である。負閉路検出は「さらに反復すれば距離が下がり続け得る」ことを示すが，本課題では $k$ を固定して $\mathrm{LB}_k$ を用いるため，負閉路検出の追加ステップを入れても枝刈り効果の改善には直結しにくい（むしろ下界が過度に小さくなる方向に働き得る）。

以上より，本課題では $k = 0, 1, \ldots, |V|-1$ **の範囲での緩和結果**を下界として利用し，負閉路検出は省略する。

## 3.3 下界値テストの妥当性

探索中，現在節点を $u$，始点 $s$ から $u$ までの累積路長を $L$ とする。現在までに訪れた節点数が $|P|$ のとき，単純路制約の下で今後使用できる辺数は高々

$$r := |V| - |P|$$

である（これ以上進むと必ず頂点重複が生じるため）。よって，$u$ から $t$ への任意の単純路の追加コストは，制約を緩めた緩和問題の「高々 $r$ 本の辺での最短距離」$\mathrm{LB}_r[u]$ 以上である。したがって，この部分問題から得られる任意の $s \to t$ 単純路の長さは

$$L + \mathrm{LB}_r[u]$$

以上である。ここで暫定最良値（上界）を `bestValue` とすると，

$$L + \mathrm{LB}_r[u] \;\geq\; \texttt{bestValue}$$

が成り立つ場合，この部分問題の子孫をどれだけ探索しても `bestValue` より良い解は得られない。よってこの条件を満たすとき探索を終端してよい（下界値テスト）。また $\mathrm{LB}_r[u] = +\infty$（$r$ 本以内では $u$ から $t$ に到達不能）の場合も実行可能解が存在しないため，同様に終端してよい。

## 3.4 擬似コード

### 3.4.1 前処理：下界配列 $\mathrm{LB}_k$ の計算（負閉路検出は省略）

以下は Bellman–Ford 法と同様に緩和を反復するが，**各反復 $k$ の結果を保存**し，$\mathrm{LB}_k$ を得る手続きである。

**Algorithm:** **PrecomputeLBSteps**$(G = (V, E),\ d,\ t)$
**Input:** 有向グラフ $G = (V, E)$, 枝長 $d : E \to \mathbb{R}$, 終点 $t$
**Output:** 下界配列 $\mathrm{LB}_k[v]$（$k = 0, \ldots, |V| - 1$）

1. **for each** $v \in V$ **do** $\mathrm{LB}_0[v] := +\infty$
2. $\mathrm{LB}_0[t] := 0$
3. **for** $k := 1$ **to** $|V| - 1$ **do**
   (a) **for each** $v \in V$ **do** $\mathrm{LB}_k[v] := \mathrm{LB}_{k-1}[v]$
   (b) **for each** $(u, v) \in E$ **do**
      (i) **if** $\mathrm{LB}_{k-1}[v] \neq +\infty$ **and** $d(u, v) + \mathrm{LB}_{k-1}[v] < \mathrm{LB}_k[u]$ **then**
         A. $\mathrm{LB}_k[u] := d(u, v) + \mathrm{LB}_{k-1}[v]$
4. // 注：正規の Bellman–Ford 法の負閉路検出（追加の緩和チェック）は省略する
5. **return** $\{\mathrm{LB}_k\}_{k=0}^{|V|-1}$

### 3.4.2 限定操作：下界値テスト

**Procedure:** **LowerBoundTest**$(u,\ L,\ r)$
**Input:** 現在節点 $u$, 累積長 $L$, 残り許容辺数 $r$
**Output:** 枝刈りするなら **true**，続行するなら **false**

1. **if** $\mathrm{LB}_r[u] = +\infty$ **then return true**
2. **if** $L + \mathrm{LB}_r[u] \geq$ `bestValue` **then return true**
3. **return false**

### 3.4.3 全体：限定操作込み探索

**Algorithm:** **ShortestSimplePathBB**$(G = (V, E),\ d,\ s,\ t)$
**Input:** $G = (V, E)$, $d : E \to \mathbb{R}$, 始点 $s$, 終点 $t$
**Output:** 最短単純路 `bestPath` と長さ `bestValue`

1. $\{\mathrm{LB}_k\} :=$ `PrecomputeLBSteps`$(G, d, t)$
2. $P := \emptyset$
3. `bestValue` $:= +\infty$
4. `bestPath` $:= \emptyset$
5. `ExploreBB`$(P, s, 0)$

6. **return** (`bestPath`, `bestValue`)

**Procedure: ExploreBB**($P$, $u$, $L$)
**Input:** 単純路 $P$, 現在節点 $u \in V \setminus P$, 累積長 $L$

1. $P$.`push`($u$)
2. $r := |V| - |P|$ // 残り許容辺数
3. **if** `LowerBoundTest`($u, L, r$) **then**
   (a) $P$.`pop`()
   (b) **return**
4. **if** $u = t$ **then**
   (a) **if** $L <$ `bestValue` **then**
      (i) `bestValue` $:= L$
      (ii) `bestPath` $:=$ `Copy`($P$)
5. **else**
   (a) **for each** $(u, v) \in E$, $v \notin P$ **do**
      (i) `ExploreBB`($P, v, L + d(u, v)$)
6. $P$.`pop`()
7. **return**

# 4 課題 4：限定操作付きアルゴリズムの実装と実行時間の比較

## 4.1 実装

課題 3 で設計した限定操作（下界値テストおよび到達不能判定）を Python で実装した。下界値 $\mathrm{LB}[u]$ は、$u$ から終点 $t$ までの「単純路制約を無視した最短距離」として Bellman–Ford 法により前処理で計算する。探索中は、現在節点 $u$ と累積路長 $L$ に対し、$L + \mathrm{LB}[u] \geq$ `bestValue` が成り立つ場合に部分問題を打ち切る（下界値テスト）。また、$\mathrm{LB}[u] = +\infty$ の場合（$u$ から $t$ へ到達不能）も同様に打ち切る。

探索ノード数と枝刈り数は、DFS 関数に入った回数（探索ノード数）と、限定操作により打ち切った回数（枝刈り数）としてそれぞれカウンタを用いて計測した。実行時間は `time.process_time()` により CPU 時間を測定し、ミリ秒に変換した。

グラフ生成方法および試行回数は課題 2 と同一とし、各 $(n, m)$ について 5 回試行して平均および標準偏差を算出した。枝数は $m = \frac{n(n-1)}{2}$、$n = 5$ から 15 まで計測した。

## 4.2 結果

限定操作なし（課題 2）と限定操作あり（課題 4）の実行時間と探索ノード数を表 2 に示す。

限定操作の導入により、探索ノード数が大幅に削減され、それに伴い実行時間も劇的に短縮された。特に $n = 15$ では、限定なしで約 4163ms、探索ノード約 $8.47 \times 10^6$ であったのに対し、限定ありでは約 47ms、探索ノード約 $6.51 \times 10^4$ となり、探索ノード数が約 130 分の 1、実行時間が約 88 分の 1 に削減された。

**表 2:** 限定操作の有無による実行時間・探索ノード数の比較（平均 ± 標準偏差、試行回数 5）

| $n$ | 限定なし (Plain) | | | | 限定あり (BB) | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | time (ms) | | nodes | | time (ms) | | nodes | | pruned | |
| | mean | std | mean | std | mean | std | mean | std | mean | std |
| 5 | 0.016 | 0.024 | 4.4 | 2.1 | 0.024 | 0.007 | 3.2 | 2.0 | 0.8 | 0.4 |
| 6 | 0.008 | 0.002 | 10.6 | 3.0 | 0.038 | 0.006 | 8.6 | 2.6 | 2.8 | 1.2 |
| 7 | 0.039 | 0.029 | 69.8 | 33.3 | 0.058 | 0.009 | 26.6 | 11.3 | 11.6 | 5.5 |
| 8 | 0.033 | 0.019 | 73.4 | 45.4 | 0.101 | 0.033 | 40.4 | 22.1 | 17.8 | 15.0 |
| 9 | 0.122 | 0.083 | 240.8 | 160.3 | 0.177 | 0.037 | 109.0 | 50.4 | 41.8 | 30.2 |
| 10 | 0.468 | 0.255 | 1283.0 | 609.1 | 0.306 | 0.112 | 336.8 | 166.0 | 140.8 | 46.1 |
| 11 | 2.355 | 0.781 | 5725.0 | 1741.0 | 0.742 | 0.223 | 805.2 | 318.2 | 419.6 | 175.4 |
| 12 | 14.980 | 5.121 | 26190.0 | 8652.0 | 1.598 | 0.826 | 1315.0 | 703.0 | 689.6 | 365.6 |
| 13 | 86.410 | 43.880 | 192400.0 | 92210.0 | 10.450 | 9.409 | 14090.0 | 13080.0 | 7991.0 | 7887.0 |
| 14 | 497.500 | 254.500 | 1093000.0 | 487600.0 | 14.380 | 4.662 | 19770.0 | 6362.0 | 11560.0 | 3256.0 |
| 15 | 4163.000 | 1294.000 | 8474000.0 | 2836000.0 | 47.330 | 55.020 | 65050.0 | 75650.0 | 41510.0 | 47960.0 |

## 4.3 考察

### 4.3.1 限定操作の効果

下界値テストの導入により、暫定解を超える可能性のない部分問題を早期に打ち切ることができ、探索木の大部分を枝刈りできた。表 2 の “pruned” 列は、限定操作により打ち切られた部分問題数を示しており、$n$ が大きくなるほど枝刈り数も増加している。例えば $n = 15$ では約 41510 個の部分問題が枝刈りされ、全体の探索ノード数（約 65050）のうち約 64% が枝刈りによって削減されたことになる。

### 4.3.2 実行時間の削減

探索ノード数の削減に伴い、実行時間も大幅に短縮された。$n = 10$ までは両手法ともミリ秒単位で完了するが、$n \geq 11$ からは差が顕著になる。$n = 13$ 以降では、限定なしでは秒単位の計算時間を要するのに対し、限定ありではミリ秒単位で完了している。これは、限定操作により実質的に解けるグラフのサイズが拡大したことを意味する。

### 4.3.3 限定操作のオーバーヘッド

$n$ が非常に小さい場合（例：$n = 5, 6$）、限定ありの方が実行時間が長くなることがある。これは、下界値計算（Bellman–Ford 法）や下界値テストの判定に伴うオーバーヘッドが、枝刈りによる削減効果を上回るためである。しかし、$n \geq 7$ では枝刈り効果がオーバーヘッドを上回り、限定操作の導入が明確に有効となる。

# 結論

本実験により、限定操作（下界値テスト）の導入が探索効率を劇的に改善することが実証された。特に大規模なグラフに対しては、限定操作なしでは現実的な時間内に解を得ることが困難であるが、

限定操作を導入することで実用的な計算時間で最適解を求めることが可能となる。これは、分枝限定法における限定操作の重要性を示す典型的な例である。

# 付録 A　ソースコード

コード作成、レポート作成の一部に GitHub Copilot を使用した。

## A.1　課題 2 の実装コード

**Listing 1:** 課題 2 の実装コード

```python
import random
import time
import math

def shortest_path_branching(G, d, s, t):
    P = []
    best_length = float("inf")
    best_path = None
    node_count = 0

    def dfs(u, current_length):
        nonlocal best_length, best_path, node_count
        node_count += 1
        P.append(u)

        if u == t:
            if current_length < best_length:
                best_length = current_length
                best_path = P.copy()
        else:
            for v in G.get(u, []):
                if v not in P:
                    dfs(v, current_length + d[(u, v)])

        P.pop()

    dfs(s, 0.0)
    return best_path, best_length, node_count


def generate_random_graph(n, m, weight_low=-10, weight_high=10):
    V = list(range(n))
    all_edges = [(i, j) for i in V for j in V if i != j]
    random.shuffle(all_edges)
    E = all_edges[:m]

    G = {i: [] for i in V}
    d = {}

    for (u, v) in E:
        G[u].append(v)
        d[(u, v)] = random.randint(weight_low, weight_high)

    return G, d
```

```
45
46
47  def run_experiment(n, m, trials=5):
48      times = []
49      nodes = []
50
51      for _ in range(trials):
52          G, d = generate_random_graph(n, m)
53          s = 0
54          t = n - 1
55
56          if (s, t) not in d:
57              G[s].append(t)
58              d[(s, t)] = random.randint(-10, 10)
59
60          start = time.process_time()
61          _, _, node_count = shortest_path_branching(G, d, s, t)
62          end = time.process_time()
63
64          times.append((end - start) * 1000)
65          nodes.append(node_count)
66
67      def mean_std(arr):
68          mean = sum(arr) / len(arr)
69          std = math.sqrt(sum((x - mean) ** 2 for x in arr) / len(arr))
70          return mean, std
71
72      return mean_std(times), mean_std(nodes)
73
74
75  # 実行（節点数5〜15で実験）
76  print("n | time_mean(ms) | time_std | nodes_mean | nodes_std")
77  print("-" * 55)
78
79  for n in range(5, 16):
80      m = n * (n - 1) // 2
81      (t_mean, t_std), (n_mean, n_std) = run_experiment(n, m)
82      print(f"{n:2d} | {t_mean:12.2f} | {t_std:8.2f} | {n_mean:11.1f} | {n_std:9.1f}")
```

## A.2　課題 4 の実装コード

**Listing 2:** 課題 4 の実装コード

```
1  import random
2  import time
3  import math
4
5  # ------------------------------
6  # 課題2：限定なし（全単純路DFS）
7  # ------------------------------
8  def shortest_path_branching(G, d, s, t):
9      P = []
10     best_length = float("inf")
11     best_path = None
12     node_count = 0
13
```

```
    def dfs(u, current_length):
        nonlocal best_length, best_path, node_count
        node_count += 1
        P.append(u)

        if u == t:
            if current_length < best_length:
                best_length = current_length
                best_path = P.copy()
        else:
            for v in G.get(u, []):
                if v not in P:
                    dfs(v, current_length + d[(u, v)])

        P.pop()

    dfs(s, 0.0)
    return best_path, best_length, node_count


# ----------------------------
# 課題4：限定あり（前処理LB + 下界値テスト）
#  - LB_k[u] = 「辺数 <= k」で u->t の最短歩道（単純路制約は無視）
#  - Bellman-Ford の各反復の dist を保存して作る（計算量 O(nm) のまま）
# ----------------------------
def precompute_lb_by_steps(G, d, t, n):
    """
    LB[k][u] = u->t の最短距離（歩道OK）ただし使用辺数 <= k
    k=0..n-1 を前処理で全部作る。
    """
    INF = float("inf")
    LB = [[INF] * n for _ in range(n)]  # LB[k][u]
    LB[0][t] = 0.0

    # k 回目 = 辺数 <= k の最短歩道
    for k in range(1, n):
        prev = LB[k - 1]
        cur = LB[k]

        # まず「何もしない（辺を増やさない）」を許す：cur[u] <= prev[u]
        for u in range(n):
            cur[u] = prev[u]

        # 1 辺伸ばす遷移：cur[u] <= d(u,v) + prev[v]
        #  (u->t を求めたいのでこの形が自然)
        for u in range(n):
            for v in G.get(u, []):
                if prev[v] != INF:
                    cand = d[(u, v)] + prev[v]
                    if cand < cur[u]:
                        cur[u] = cand

    return LB


def greedy_initial_path(G, d, s, t, LB, n):
    """
    LB をガイドにして、とりあえずの実行可能単純路を作る（上界を早く得る用）
```

```
    失敗したら (None, inf)。
    """
    INF = float("inf")
    visited = [False] * n
    path = [s]
    visited[s] = True
    u = s
    length = 0.0

    while u != t:
        remaining = n - len(path)  # これから使える最大辺数（単純路なので）
        if remaining <= 0:
            return None, INF

        # 次候補（単純路制約のみ）
        cand = []
        for v in G.get(u, []):
            if not visited[v] and LB[remaining - 1][v] != INF:
                cand.append(v)

        if not cand:
            return None, INF

        # d(u,v) + 下界 の小さい順に選ぶ（雑だが速い）
        v = min(cand, key=lambda x: d[(u, x)] + LB[remaining - 1][x])

        length += d[(u, v)]
        path.append(v)
        visited[v] = True
        u = v

    return path, length


def shortest_path_branching_bb(G, d, s, t):
    """
    Branch & Bound 版（課題4）
    - LB[k][u] を前処理で作る（Bellman-Ford の途中結果）
    - 途中で remaining edges を使って強めの下界 LB[remaining][u] を参照
    - 下界値テスト：current_length + LB[remaining][u] >= best_length なら枝刈り
    - 探索順を d(u,v)+LB でソートして上界を早く更新しやすくする
    """
    n = len(G)
    INF = float("inf")

    LB = precompute_lb_by_steps(G, d, t, n)

    # 上界（incumbent）を早めに用意（貪欲で1本作る）
    best_path, best_length = greedy_initial_path(G, d, s, t, LB, n)
    if best_path is None:
        best_length = INF
        best_path = None

    P = []
    inP = [False] * n
    node_count = 0
    pruned_count = 0

```

```
    def dfs(u, current_length):
        nonlocal best_length, best_path, node_count, pruned_count
        node_count += 1
        P.append(u)
        inP[u] = True

        # 残りの最大辺数（単純路なので s->t 全体で <= n-1 辺）
        remaining = n - len(P)
        # u->t を「辺数 <= remaining」で行けないならここで終了
        if remaining < 0 or LB[remaining][u] == INF:
            pruned_count += 1
            inP[u] = False
            P.pop()
            return

        # 下界値テスト（限定操作）
        if current_length + LB[remaining][u] >= best_length:
            pruned_count += 1
            inP[u] = False
            P.pop()
            return

        if u == t:
            if current_length < best_length:
                best_length = current_length
                best_path = P.copy()
        else:
            # 探索順を工夫：見込みが良さそうな v から先に（上界更新を早める）
            cand = [v for v in G.get(u, []) if not inP[v]]
            # 次に 1 辺進むので remaining-1 の下界を使う
            if remaining - 1 >= 0:
                cand.sort(key=lambda v: d[(u, v)] + LB[remaining - 1][v])
            for v in cand:
                dfs(v, current_length + d[(u, v)])

        inP[u] = False
        P.pop()

    dfs(s, 0.0)
    return best_path, best_length, node_count, pruned_count


# ----------------------------
# グラフ生成（あなたのまま）
# ----------------------------
def generate_random_graph(n, m, weight_low=-10, weight_high=10):
    V = list(range(n))
    all_edges = [(i, j) for i in V for j in V if i != j]
    random.shuffle(all_edges)
    E = all_edges[:m]

    G = {i: [] for i in V}
    d = {}

    for (u, v) in E:
        G[u].append(v)
        d[(u, v)] = random.randint(weight_low, weight_high)

```

```
    return G, d


def mean_std(arr):
    mean = sum(arr) / len(arr)
    std = math.sqrt(sum((x - mean) ** 2 for x in arr) / len(arr))
    return mean, std


# ----------------------------
# 同一グラフで比較（これ大事：別々に回すと差がノイズになる）
# ----------------------------
def run_experiment_compare(n, m, trials=5):
    times_plain, nodes_plain = [], []
    times_bb, nodes_bb, prunes_bb = [], [], []

    for _ in range(trials):
        G, d = generate_random_graph(n, m)
        s = 0
        t = n - 1

        # s->t を必ず入れる（実行可能解の保証）
        if (s, t) not in d:
            G[s].append(t)
            d[(s, t)] = random.randint(-10, 10)

        # 限定なし
        st = time.perf_counter()
        _, _, nc_plain = shortest_path_branching(G, d, s, t)
        ed = time.perf_counter()
        times_plain.append((ed - st) * 1000)
        nodes_plain.append(nc_plain)

        # BB（限定あり）
        st = time.perf_counter()
        _, _, nc_bb, pr_bb = shortest_path_branching_bb(G, d, s, t)
        ed = time.perf_counter()
        times_bb.append((ed - st) * 1000)
        nodes_bb.append(nc_bb)
        prunes_bb.append(pr_bb)

    return (mean_std(times_plain), mean_std(nodes_plain),
            mean_std(times_bb), mean_std(nodes_bb), mean_std(prunes_bb))


# ----------------------------
# 実行
# ----------------------------
print("n | time_plain(ms)        | nodes_plain        || time_BB(ms)           |
    nodes_BB           | pruned_BB")
print("  | mean      std         | mean      std      || mean      std         | mean
        std      | mean      std")
print("-" * 110)

for n in range(5, 16):
    m = n * (n - 1) // 2
    (tp_mean, tp_std), (np_mean, np_std), (tb_mean, tb_std), (nb_mean, nb_std), (
        pr_mean, pr_std) = run_experiment_compare(n, m)
```

```
243
244     print(f"{n:2d} | "
245           f"{tp_mean:11.3e} {tp_std:11.3e} | {np_mean:11.3e} {np_std:11.3e} || "
246           f"{tb_mean:11.3e} {tb_std:11.3e} | {nb_mean:11.3e} {nb_std:11.3e} | "
247           f"{pr_mean:11.3e} {pr_std:11.3e}")
```

## 参考文献

## 参考文献

[1] 数理工学実験（2025 年度配布資料）.