

容量制約付きネットワーク設計問題に対する AI による コード生成

AI-Assisted Code Generation for Capacitated Network Design Problem

片山 直登 流通経済大学 流通情報学部

1 はじめに

ネットワーク設計問題は、ネットワーク全体の費用を最小化するようにノードまたはアークを選択してネットワークを構成するとともに、そのネットワーク上のフローを決定する問題である (Crainic et al. 2021). 容量制約付きネットワークデザイン問題 (*CND*) はその一種であり、アークの容量制約の下で、使用するアークとネットワーク上のフローを同時に決定する問題である. Katayama et al. (2009) は、*CND* に対してアーク容量をスケールリングしながら線形緩和問題を反復的に解くことにより、デザイン変数を 0 または 1 へ収束させる容量スケールリング法を開発している. Katayama (2015) は、容量スケールリング法によってデザイン変数が取り得る領域を限定し、その限定問題を局所分枝法により解くマシユールスティクスを提案している. さらに、片山直登 (2017a) および Katayama (2020) は、限定された問題を MIP 近傍探索法により解くマシユールスティクスを提案している.

近年、Codex に代表されるプログラムコード生成 AI の実用化が進んでいる. 本論文では、*CND* に対するアークフローによる定式化、列生成法を伴うパスフローによる定式化、容量スケールリング法および MIP 近傍探索法に関する先行研究の論文と関連データをプログラムコード生成 AI である Codex に入力し、論文に記述された定式化および疑似コードに基づいて Python のプログラムコードを生成させる. さらに、生成された Python コードを用いて数値実験を行い、生成されたコードの妥当性を検証する.

2 *CND* の定式化

本節では、アークフローによる定式化およびパスフローによる定式化を示す.

2.1 アークフローによる定式化

ノード集合を N 、アーク候補集合を A 、品種の集合を K とし、アーク (i, j) を選択するか否かを表すデザイン変数を y_{ij} 、アーク (i, j) 上の品種 k のフロー量を表すアークフロー

変数を x_{ij}^k とする. アーク (i, j) の容量を C_{ij} , デザイン費用を f_{ij} , アーク (i, j) 上の品種 k の単位当たりのフロー費用を c_{ij}^k とする. 品種 k の需要量を d^k , 品種 k の始点を O^k , 終点を D^k とする. また, ネットワーク上でノード n に入るアークをもつノード集合を N_n^+ , ノード n から出るアークをもつノード集合を N_n^- とする.

このとき, アークフロー変数を用いた CND の定式化 $CNDA$ は, 次のように表される.
 $CNDA$

$$\min \sum_{(i,j) \in A} \sum_{k \in K} c_{ij}^k x_{ij}^k + \sum_{(i,j) \in A} f_{ij} y_{ij} \quad (1)$$

subject to

$$\sum_{i \in N_n^+} x_{in}^k - \sum_{j \in N_n^-} x_{nj}^k = \begin{cases} -d^k & \text{if } n = O^k \\ d^k & \text{if } n = D^k \\ 0 & \text{otherwise} \end{cases} \quad \forall n \in N, k \in K, \quad (2)$$

$$\sum_{k \in K} x_{ij}^k \leq C_{ij} y_{ij} \quad \forall (i, j) \in A, \quad (3)$$

$$x_{ij}^k \leq d^k y_{ij} \quad \forall k \in K, (i, j) \in A, \quad (4)$$

$$y_{ij} \in \{0, 1\} \quad \forall (i, j) \in A, \quad (5)$$

$$x_{ij}^k \geq 0 \quad \forall (i, j) \in A, k \in K. \quad (6)$$

定式化の詳細については, 片山 (2017) を参照されたい.

2.2 パスフローによる定式化

品種 k が取り得るパス集合を P^k , 品種 k のパス p 上のフロー量を表すパスフロー変数を z_p^k とし, パス p がアーク (i, j) を含む場合に 1, そうでない場合に 0 となる定数を δ_{ij}^p とする. このとき, パスフロー変数を用いた CND の定式化 $CNDP$ は, 次のように表される.

$CNDP$

$$\min \sum_{(i,j) \in A} \sum_{k \in K} c_{ij}^k \sum_{p \in P^k} \delta_{ij}^p z_p^k + \sum_{(i,j) \in A} f_{ij} y_{ij} \quad (7)$$

subject to

$$\sum_{p \in P^k} z_p^k = d^k \quad \forall k \in K, \quad (8)$$

$$\sum_{k \in K} \sum_{p \in P^k} \delta_{ij}^p z_p^k \leq C_{ij} y_{ij} \quad \forall (i, j) \in A, \quad (9)$$

$$\sum_{p \in P^k} \delta_{ij}^p z_p^k \leq d^k y_{ij} \quad \forall (i, j) \in A, k \in K, \quad (10)$$

$$y_{ij} \in \{0, 1\} \quad \forall (i, j) \in A, \quad (11)$$

$$z_p^k \geq 0 \quad \forall p \in P^k, k \in K. \quad (12)$$

パスフロー変数の数はアーク数に対して指数的に増加するため、双対変数を用いた列生成法により、必要なパスフロー変数を逐次生成する必要がある。同時に、行生成法により、必要な強制制約である式 (10) も逐次生成する必要がある。パスフローによる定式化と列生成法の詳細については、Katayama et al. (2009) を参照されたい。

3 近似解法

本節では、容量スケールリング法と MIP 近傍探索法を取り上げ、それぞれの概要を示す。

3.1 容量スケールリング法

容量スケールリング法は、デザイン変数を連続緩和した問題を反復的に解き、デザイン変数値に基づいてアーク容量を更新することにより、デザイン変数を 0 または 1 へ収束させる方法である。

l 回目の反復におけるアーク (i, j) の容量 C_{ij}^l は、次式により更新する。

$$C_{ij}^l := \lambda C_{ij}^{l-1} \tilde{y}_{ij} + (1 - \lambda) C_{ij}^{l-1}. \quad (13)$$

ここで、 $\tilde{y}_{ij}$ は $l-1$ 回目の反復で得られたデザイン変数の値であり、 λ は 0 から 1 のスムージングパラメータである。

0 に収束していないデザイン変数に対応するアーク集合と生成済みパス集合からなる限定問題を分枝限定法によって求解し、近似解を得る。

詳細については、Katayama et al. (2009) または片山 (2017) を参照されたい。

3.2 MIP 近傍探索法

MIP 近傍探索法は、アークフローによる定式化 $CNDA$ に現在の試行解を探索対象から除外する近傍制約を追加し、得られた問題を反復的に解くことによって解を改善する方法である。

MIP 近傍探索法では、次の 3 式を $CNDA$ に追加する。

$$\sum_{(i,j) \in A | \bar{y}_{ij}=1} y_{ij} \leq L - 1, \quad (14)$$

$$\Phi \leq UB - \zeta, \quad (15)$$

$$\sum_{(i,j) \in A | \bar{y}_{ij}=1} y_{ij} \geq L - M. \quad (16)$$

L は現在の試行解 $\bar{\mathbf{y}}$ において $\bar{y}_{ij} = 1$ であるデザイン変数の個数, Φ は試行解の目的関数値, UB は $CNDA$ における現時点での最良上界値, ζ は十分に小さい非負の定数, M は実行可能領域を制限するためのパラメータである.

ここでは, 容量スケール法で得られた実行可能解を MIP 近傍探索法の初期の試行解として用いる. 詳細については, 片山 (2017) および Katayama (2020) を参照されたい.

4 プログラムコードの生成

プログラムコード生成 AI である Codex を用いて, 2 種類の定式化を解く Python プログラムコードおよび 2 種類の近似解法を実装した Python プログラムコードを生成する. なお, 本論文では Codex 5.6 Sol(非常に高い) を使用する.

4.1 アークフローによる定式化

アークフローによる定式化 $CNDA$ を MIP ソルバーである Gurobi で直接解く Python コードを生成する. 以下の枠内には, 入力したプロンプトおよびファイル, Codex によるレスポンス, 説明ファイルの内容, ならびに実行時の画面出力を示す. なお, これらは一部を省略または編集している.

以下に, 入力したプロンプトとファイルを示す.

プロンプト

添付した論文ファイルの (1) 式から (6) 式を, Gurobi を使って解く Python のコードを作成する.

入力ファイル

- 論文ファイル: 片山 (2017)
- データファイル: c100_400_30_V_T_10.dow, c64.dow, r18.9.dow

Codex に論文の PDF ファイルとデータファイルを与え, プロンプトにより「論文内の (1) 式から (6) 式を解くコードを作成する」ことを指示した.

片山 (2017) は MIP 近傍探索法に関する論文であるが, アークフローによる定式化も記述されている. また, CND の中規模ベンチマークデータにも対応できることを確認するため, C 問題と R 問題 (Katayama et al. 2009) の中で比較的規模の大きい 3 種類のベンチマークのデータファイルを入力し, これらを求解対象とした.

続いて, Codex からのレスポンスとして, 進捗状況, 最終回答および主なパラメータを示す.

進捗表示

- 論文の (1)~(6) 式と 3 種類の .dow 形式を確認し、データを引数で読み込む Gurobi コード、説明書、実行結果を作成します。まず数式・入力形式・Gurobi 実行環境を調べ、実データ 3 件で検証します。
- 数式は、論文どおり「固定費+多品種フロー費の最小化、各品種のフロー保存、共有容量、アーク選択との連結、0-1・非負条件」で確認できました。続いて .dow の 7 列の意味を資料・周辺ファイルと照合します。ここを誤ると解けても目的値が別問題になるため、列対応を確定してから実装します。
- 入力形式が確定しました。実装は 3 データすべてに適用できます。コード、入力検査、モデル規模確認、実行不能を明記した result.sol まで整えます。

最終回答

作成しました。solve.cnd.py は、論文の式 (1)~(6) をそのまま Gurobi モデル化し、上界・下界・計算時間・誤差を result.sol に追記します。

solve_cnd.py, 説明.md, result.sol, Response.md, requirements.txt

3 データの形式と件数は検証済みです。実行例は次のとおりです。

```
Python solve.cnd.py c64.dow -result result.sol
```

主なパラメータ

- -time-limit: MIP 計算時間制限
- -mip-gap-percent: 相対ギャップ終了条件

この試行では、約 10 分で 247 行のプログラムコードが生成された。

次に、説明ファイルに記述された入力形式と (1)~(6) 式の解釈を示す。

入力形式

.dow ファイルは次の順序で読み込む、

ノード数・アーク数・品種数、始点・終点・フロー単価・容量・固定費、始点・終点・需要量

ノード番号はファイル上では 1 始まり、コード内部では 0 始まりに変換する、入力行数、ノード番号、費用・容量・需要量も検査する。

以上の内容から、Codex が入力データの形式を適切に解釈していることが確認できる。

数理モデルとの対応

アークを $a = (i, j)$, 品種を k と表す, .dow には品種別ではなくアーク別のフロー単価が格納されているため, 論文の c_{ij}^k には全品種で同じアーク単価を使用する,

- 式 (1): 全品種のフロー費用と, 選択アークの固定費の和を最小化する,
- 式 (2): 各品種・各ノードについて「流入量 - 流出量」を, 始点では -需要量, 終点では +需要量, その他では 0 とする,
- 式 (3): アーク上の全品種のフロー合計を「容量 $\times$ デザイン変数」以下にする,
- 式 (4): 品種ごとに「フロー量 $\leq$ 需要量 $\times$ デザイン変数」とする,
- 式 (5): デザイン変数 y を 0 - 1 変数とする,
- 式 (6): アークフロー変数 x を非負連続変数とする,

同じ始点・終点を持つ並行アークがあっても区別できるよう, 変数はアークの行番号を添字にしている,

以上の内容から, 定式化における各式の意味についても, Codex が適切に解釈していることが確認できる.

生成されたプログラムコードは, 各入力データを読み込み, 論文の (1)~(6) 式に基づいて Gurobi モデルを構築し, 求解する構成となっている. コードを確認したところ, Codex のレスポンスおよび説明ファイルの内容に沿って, データの読み込み, アークフローによる定式化に基づくモデルの構築, および Gurobi による求解処理が実装されていた.

続いて, データ c100_400_10_FL_10 を用いて求解したときの画面出力の一部を示す.

画面出力

```
Optimize a model with 5400 rows, 4400 columns and 20400 nonzeros (Min)
Model has 4400 linear objective coefficients
Variable types: 4000 continuous, 400 integer (400 binary)
Found heuristic solution: objective 256915.00000
Root relaxation: objective 2.008652e+04, 2461 iterations, 0.10 seconds (0.15 work units)
Nodes — Current Node — Objective Bounds — Work Expl Unexpl — Obj Depth
IntInf — Incumbent BestBd Gap — It/Node Time
  0 0 20086.5234 0 52 256915.000 20086.5234 92.2% - 0s
...
146 26 23864.0417 11 10 23949.0000 23575.4542 1.56% 494 10s
Optimal solution found (tolerance 1.00e-04)
Best objective 2.394900000000e+04, best bound 2.394900000000e+04, gap 0.0000%
```

この画面出力は Gurobi による最適化モデルの求解過程を示しており, コード内でモデルが生成され, 最適解が求められていることが分かる.

同一または類似のプロンプトを用いて複数回コード生成を行ったところ, 進捗表示や最終回答などのレスポンス内容, 作業時間, 生成コードのパラメータ名, 関数構成および行数は, 試行ごとに差がみられた. また, 生成コードの実行時にエラーが発生する場合もあり, 追加のプロンプトでエラー内容を示して修正を指示する必要があった. ただし, 本論

文で実施した範囲では、いずれの試行においても最終的に実行可能なプログラムコードを生成することができた。

4.2 パスフローによる定式化

パスフローによる定式化 *CNDP* を Gurobi で解くプログラムコードを生成する。アーキフローによる定式化 *CNDA* とは異なり、各品種の始点・終点間を結ぶパスに対応するパスフロー変数を特定する必要がある。パスの総数は一般にデータのサイズに従って指数的に増加するため、必要なパス変数を逐次生成する列生成法が必要となる。列生成法では、制約式に対応する双対変数を用いた価格付け問題と、パス生成のための最短路問題を組み込む必要がある。また、強制制約を逐次追加する行生成を行う必要がある。このようなことから、論文中の定式化や疑似コードの解釈に加えて、これらを実装するための Gurobi の機能を適切に対応させる必要がある。

次に、入力したプロンプトとファイル、Codex からのレスポンスおよび主なパラメータ等を示す。

プロンプト

添付した論文ファイルの (7) 式から (12) 式を、Gurobi を使って解く Python のコードを作成する。

入力ファイル

- 論文ファイル：片山 (2017)
- データファイル：c100_400_30_V_T_10.dow, c64.dow, r18.9.dow

片山 (2017) は MIP 近傍探索法に関する論文であるが、パスフローによる定式化および列生成法・行生成法についても記述されている。

進捗表示

- (7)～(12) 式のパスフロー定式化へ切り替えます。全単純パスの列挙は c64.dow では現実的でないため、Gurobi で線形緩和と最短路価格付けを反復してパス集合を生成し、その列だけで式 (7)～(12) の最終 MIP を解くコードにします。
- パス版コードを作成しました。初期実行可能パスの生成、式 (7)～(12) の LP 列生成、最終 0 – 1MIP, result.sol 追記までを 1 本にまとめています。いま、3 データの行数整合と、需要・容量・強制制約、出力項目の実装漏れがないかを静的に確認しています。

最終回答

作成しました. 新しい `solve_cnd_path.py` は, `.dow` を読み, 生成したパス集合上で式 (7)~(12) の MIP を解きます. 指定された 6 項目は `result.sol` に追記されます.

`solve_cnd_path.py`, 説明.md, `result.sol`, `Response.md`, `requirements.txt`

3 サンプルの入力件数とコード上の式 (7)~(12) の対応を静的検証しました. `c64.dow` では全単純パス列挙が非現実的なため, 初期 LP のフロー分解と最短路価格付けによる列生成を行い, その制限パス集合で最終 MIP を解きます. Gurobi の Column API を使ってパス変数を追加しています.

これは生成パス集合上では式 (7)~(12) そのものですが, 厳密な branch-and-price ではありません. 上界・下界・誤差は制限パス MIP に対する値です.

主なパラメータ

- `-time-limit`: MIP 計算時間制限
- `-mip-gap-percent`: 相対ギャップ終了条件
- `-cg-max-iterations`: 列生成の最大反復回数

以上の内容から, パスを逐次生成する列生成が必要であることや, 得られる解が全パス版に対する厳密解であることを保証するものではないこと等を Codex が適切に解釈していることが確認できる.

この試行では, 約 7 分で 578 行のプログラムコードが生成された. なお, 本コードは前節の *CNDA* 用コードを生成した対話に続けて生成したため, それまでの対話内容や生成済みコード等が参照されている.

次に, 説明文に記述された (7)~(12) 式の解釈とパス集合の生成方法を示す.

式 (7)~(12) との対応

- 式 (7): 各パスを構成するアークのフロー単価とパスフロー量の積および選択アークの固定費の合計を最小化する,
- 式 (8): 品種ごとの全パスフロー量を需要量に等しくする,
- 式 (9): 各アークを通る全品種・全パスのフロー量を「容量 $\times y_{ij}$ 」以下にする,
- 式 (10): 各アーク・各品種について, 該当パスのフロー量を「需要量 $\times y_{ij}$ 」以下にする,
- 式 (11): y_{ij} を 0-1 変数とする,
- 式 (12): x_p^k を非負連続変数とする,

コードではパスをアーク行番号のタプルとして保持する, このため, 同じ始点・終点をもつ並行アークも区別できる,

パス集合の作り方

全単純パスの個数は指数的に増える．特に c64.dow は 30 ノード・683 アーク・400 品種であり，全パスの事前列挙は実用的でない．このコードは次の方法で制限パス集合を作る．

1. 0-1 デザイン変数を連続緩和したアークフロー LP を Gurobi で解く．
2. 各品種の実行可能なアークフローを始点-終点パスへ分解し，初期パス列とする．
3. 式 (7)~(12) の線形緩和を制限主問題として解く．
4. 需要制約，共有容量制約，品種別強制制約の双対値を使い，品種ごとの最短路価格付け問題を解く．
5. 被約費用が負のパスを追加し，負の列がなくなるか最大反復回数に達するまで反復する．
6. 生成されたパス集合上で y_{ij} を 0-1 に戻し，式 (7)~(12) の最終 MIP を解く．

最終 MIP は，生成された P^k 上では式 (7)~(12) そのものである．ただし，Gurobi は MIP 探索中に新しい変数列を追加できないため，これは全単純パスを用いる厳密な branch-and-price ではない．したがって，出力する上界・下界・誤差は「生成された制限パス集合上の MIP」に対する値である．全パス版の大域最適性を保証するものではない．

以上の内容から，Codex が各式の意味および列 (パス) 生成法の手順を適切に解釈していることが確認できる．

片山 (2017) には，定式化 $CNDP$ に対する列生成法および価格付け問題の詳細な記述がないため，Codex は WEB 上にある関連資料も検索・参照しながら，列生成法と価格付け問題を含むコードを生成している．また，「パス集合の作り方」の手順 4 に示すように，双対解に基づいて価格付け問題を構成し，これを最短路問題として求解している．生成コードを確認したところ，Codex のレスポンスおよび説明ファイルの内容に沿って，パスフローによる定式化に基づく Gurobi モデルの構築，双対解の利用，および最短路問題のダイクストラ法コードが実装されている．

続いて，データ c100_400_10.F.L.10 を用いて求解したときの画面出力の一部を示す．

画面出力

```
初期パス列数: 32
列生成: 反復 1, 追加列=10, パス列数=42
...
列生成: 反復 73, 追加列=1, パス列数=601
列生成: 74 反復で収束, LP 目的値=19748.12753, パス列数=601
Optimize a model with 1650 rows, 1001 columns and 11679 nonzeros (Min)
Model has 1001 linear objective coefficients
Variable types: 601 continuous, 400 integer (400 binary)
Found heuristic solution: objective 73450.000000
Nodes — Current Node — Objective Bounds — Work Expl Unexpl — Obj Depth
IntInf — Incumbent BestBd Gap — It/Node Time
  0 0 19781.6530 0 49 73450.0000 19781.6530 73.1% - 0s
...
  0 2 23608.3998 0 187 23949.0000 23608.3998 1.42% - 3s
Optimal solution found (tolerance 1.00e-04)
Best objective 2.3949000000000e+04, best bound 2.3949000000000e+04, gap 0.0000%
```

初期パス数は32であり、列生成は74回の反復で収束し、最終的に601本のパスが生成されていることが分かる。6行目以降はGurobiによる最適化モデルの求解過程を示している。この画面出力からも、コード内でパス生成とGurobiモデルの求解が実際に行われていることが確認できる。なお、入力データに対する最適解を算出している。

4.3 容量スケーリング法

パスフローによる定式化CNDPを基礎として、容量スケーリング法を組み合わせた解法を実装するプログラムコードを生成する。

以下に、入力したプロンプトとファイル、Codexからのレスポンスおよび主なパラメータを示す。

プロンプト

添付した論文ファイルの(1)式から(6)式を、Gurobiを使って、COLUMNAND ROWGENERATIONTECHNIQUEとCAPACITY SCALING HEURISTICを用いて解を求めるPythonのコードを作成する。

入力ファイル

- 論文ファイル：Katayama(2009)
- データファイル：c100_400_30_V_T_10.dow, c64.dow, r18.9.dow

Katayama et al. (2009) は容量スケーリング法に関する論文である。

進捗表示

- (1)～(6) 式と、論文中の Column-and-Row Generation / Capacity Scaling の手順を先に正確に抽出します。その定義に合わせて、.dow 入力・Gurobi 求解・result.sol 追記までを新しい Python コードにまとめ、説明書とこの画面表示を Response.md に追記します。
- 論文では、(1)～(6) はパスフロー型 MCND です。Capacity Scaling は反復ごとに容量を「 $\lambda \times$ アークフロー + $(1 - \lambda) \times$ 前回容量」へ更新し、ほぼ 0/1 のデザイン変数を固定して、自由変数数が閾値以下なら分枝限定を行う手順です。数式画像でも符号・上限式を確認してから実装します。
- 実装は論文どおり、初期パスを $c + f/C$ 最短路で作り、必要なパス列と forcing 行だけを増やす二段階の列・行生成を各 Capacity Scaling 反復で実行する構成にしました。初期制限主問題が容量超過で実行不能にならないよう、Phase-I 人工フローを 0 にしてから実費用の価格付けへ移る補助手順も入れています。これからコード構造と 3 データの対応を最終検査します。

最終回答

作成しました。solve_cnd_scaling.py は、.dow を読み、式 (1)～(6) を次の手順で解きます。

- $c + f/C$ 最短路による初期パス生成
- Phase-I を含む Column-and-Row Generation
- 双対価格 $c + u + w$ による Dijkstra 価格付け
- 新しいパス列と forcing 行の追加
- Capacity Scaling と、収束したデザイン変数の部分固定 MIP
- 上界・初期強化 LP 下界・全計算時間・誤差の result.sol 追記

成果物はこちらです。

solve_cnd_scaling.py, 説明.md, result.sol, Response.md, requirements.txt

3 サンプルは、ヘッダーと実データ行数の一致およびコード内の式・価格付け・容量更新・結果項目を静的検証済みです。

主なパラメータ

- -mip-time-limit : MIP 計算時間制限
- -lambda : smoothing パラメータ
- -free-variable-limit : 自由変数制限

この試行では、約 9 分で 648 行のプログラムコードが生成された。

続いて、説明文に記述された Column-and-Row Generation Technique と Capacity Scaling Heuristic の解釈を示す。

Column-and-Row Generation Technique

全パスを列挙するとパス変数が指数的に増え、forcing 制約も $|A| \times |K|$ 本になる。コードは論文の Section 4 に従って次を反復する。

1. 制限パス集合 $\bar{P}^k$ だけで制限主問題 $RLR(C'(l), \bar{P})$ を Gurobi で解く。
2. 需要制約の双対値 s^k ，共有容量制約の非負価格 u_{ij} ，存在する forcing 行の非負価格 w_{ij}^k を得る。
3. 各品種について，アーク長 $c_{ij}^k + u_{ij} + w_{ij}^k$ の Dijkstra 最短路を解く。
4. 最短路長が s^k より小さければ，負の被約費用をもつパス変数を新しい列として追加する。
5. 新しいパスが，それまで当該品種で使われていなかったアークを含む場合，対応する forcing 制約を新しい行として追加する。
6. 負の被約費用をもつ新列がなくなるまで反復する。

実装ではモデルを反復ごとに再構築することで，追加後と同じ列・行集合を安全に表現する。初期パスだけで共有容量を満たせない場合に備え，Phase-I では品種ごとの人工パスフローを最小化する。人工フローが 0 になった後，実費用の列生成へ移る。これは論文に明記されていない，初期制限主問題の実行可能性を保証するための実装上の補助である。

初期パスは $c_{ij} + f_{ij}/C_{ij}$ をアーク長とした最短路で作る。

「論文の Section 4 に従って」とあるように，Codex は Katayama et al. (2009) の Section 4 に示された疑似コードを解釈し，Column-and-Row Generation Technique を実装したコードを生成している。また，論文に明記されていない初期パスが実行不可能な場合の人工パスフローによる対応が記述されている。なお，この部分は前節の「パス集合の作り方」に対応しているが，前節とは全く異なる表現およびコードとなっている。

次に，Capacity Scaling Heuristic の解釈を示す。

Capacity Scaling Heuristic

論文の Section 3 に従う。

1. 初期スケーリング容量を $C'_{ij}(1) = C_{ij}$ とする。
2. y_{ij} を連続緩和し，共有容量を $C'_{ij}(l)y_{ij}$ ， y_{ij} の上限を $C_{ij}/C'_{ij}(l)$ とした強化 LP を Column-and-Row Generation で解く。
3. LP 解が ϵ 未満の y を 0， $1 - \epsilon$ より大きい y を 1，それ以外を自由変数とする。
4. 自由変数数が B 未満なら，0/1 候補を固定し，自由変数だけを Gurobi の分枝限定で解いて実行可能上界を得る。
5. アークフローを $\tilde{X}_{ij}$ として， $C'_{ij}(l+1) = \lambda \times \tilde{X}_{ij} + (1 - \lambda) \times C'_{ij}(l)$ に更新する。
6. 規定反復数に達し，上界が得られたら終了する。

既定値は $\lambda = 0.10$ ， $\epsilon = 0.01$ ， $MAXN = 20$ ，追加反復 10， $B = 75$ である。論文は λ として 0.025 から 0.150 を試し， $B = 75$ を使用している。

最大反復後も上界がなければ，生成済み列上で全デザイン変数を自由な 0-1 変数にした MIP を 1 回解く。

「論文の Section 3 に従う」とあるように、Codex は Katayama et al. (2009) の Section 3 に示された疑似コードを解釈し、Capacity Scaling Heuristic を実装したコードを生成している。また、必要なパラメータ値を論文より抽出している。

生成コードによって実行可能解を得ることはできたものの、同様の条件で複数回生成した他のコードと比較すると、本コードの計算時間は数倍から 100 倍程度となった。そこで、計算時間を短縮するため、Codex に次の指示を与え、コードを改善させる。

プロンプト

計算時間がかかりすぎるので、計算を効率化する。

入力ファイル

- プログラムコード：solve_cnd_scaling.py

進捗表示

- 対象スクリプトを計測・解析し、ボトルネックを特定してから互換性を保った高速化を実装・検証します。まずコードと周辺の実行条件を確認します。
- コードを見る限り、反復ごとに Gurobi モデルをゼロから再構築している点と、各品種で毎回同じ基礎重みを作り直している点が主な候補です。実データ、構文状態、Gurobi 環境を確認し、結果が変わらない最適化に絞って比較します。変更対象フォルダーへの書き込み権限も、この作業範囲に限定して取得します。
- 実データ群と実行スクリプトも確認できました。特に c64.dow は 683 アーク×400 品種なので、列生成 LP を反復ごとに再構築する現在の方式では Python 側のモデル生成だけでも急増します。ここを、同一 Gurobi モデルへ列・行を差分追加し、容量更新後もモデルと LP 基底を再利用する方式へ変更します。最終フォールバックの巨大モデルから冗長な品種別制約を除くことも併せて行います。
- 主要な書き換えは反映済みです。さらに、制限 LP 解のパスフローを MIP 開始解へ渡し、完全アークフローの「開始 LP → MIP」を同一モデル内で切り替えるようにしたため、巨大モデルの二重構築もなくなりました。構文検査と小規模な数値同値テストを行える隔離ランタイムを作業フォルダー内だけに用意して検証します。
- 静的検証では、更新ファイルが正しい UTF-8 であること、文字列と括弧が閉じていること、変更部の係数・添字が元の定式化と一致することを確認しました。実行対象データもヘッダーと実行数が一致しています。

最終回答

高速化を実装し、指定ファイルを更新しました。

- 主な高速化は、制限主問題を 1 回だけ構築して列・forcing 行・容量係数を差分更新する方式への変更、LP 基底の再利用、品種別価格の疎表現、LP パスフローによる MIP 開始解、完全アークフロー・フォールバックの二重構築と大量の冗長行の削減です。c64.dow の完全フォールバックでは 273,200 本の品種別 forcing 行を削減します。
- UTF-8、括弧・文字列、変更部の係数と添字、データのヘッダーと行数整合を静的検証しました。
- 実行環境では、まず同じ小規模データを旧版・新版で各 1 回実行し、上界・下界と計算時間を比較してください。

この試行では、約 30 分でコードの改善が行われ、923 行のプログラムコードが生成された。

続いて、説明文に記述された主な改善内容を示す。

改善内容

(1) 制限主問題を 1 回だけ構築

変更前は、Column-and-Row Generation の各価格付け反復で `path_incidence` を全パスについて作り直し、Gurobi モデル、変数、制約をすべて再生成していた。Capacity Scaling の次の反復へ進むたびに同じ処理を繰り返していたため、パス列と forcing 行が増えるほど Python 側の生成時間とメモリ割当が増えていた。

変更後は `RestrictedMaster` が 1 個の制限主問題を保持し、次だけを差分更新する。

- 負の被約費用を持つパス変数は `Gurobi.Column` で既存モデルへ追加する。
- 新しく必要になったアーク・品種の forcing 制約だけを追加する。
- Capacity Scaling 後の容量は、 y の上限と容量制約内の係数だけを変更する。
- Phase-I と実費用最小化は、人工変数の上限と目的係数の変更で切り替える。
- LP は双対シンプレックス法を使用し、直前のモデルと基底を再利用する。

これにより、モデル全体の再構築回数は「LP を解く回数」から「プログラム実行中に 1 回」へ減る。特に品種数・生成列数・Column-and-Row 反復数が大きいデータほど効果大きい。

(2) 価格付け用重みの生成量を削減

共有容量価格とルーティング費用は全品種に共通なので、価格付け反復につき 1 回だけ密な配列を作る。品種ごとに異なる forcing 価格は、正の値を持つアークだけを疎な辞書として Dijkstra へ渡す。変更前の「品種数×アーク数」のリスト再生成を避けている。最短路の判定式と被約費用は変更していない。

(3) MIP 開始解を改善

制限 LP のパスフローを部分固定 MIP へ渡す。0 固定されるアークと矛盾しない場合は、正のフローが通るアークを開いた整数デザインと LP パスフローを組み合わせた実行可能開始解にする。固定候補と矛盾する場合は従来どおりデザイン変数だけを開始値として与える。実行可能開始解が使える反復では、Gurobi が上界を早く得やすくなる。

(4) 完全アークフロー・フォールバックを軽量化

変更前は全アーク開放 LP と完全 MIP を別々に構築し、さらに全アーク・全品種の forcing 制約を追加していた。変更後は 1 個のモデルで全開放 LP を解いた後、その変数型・上限・目的係数を変更して MIP として再利用する。

完全アークフロー定式化では共有容量制約 $\sum_k x[k, a] \leq C[a]y[a]$ が、アークを閉じたとき全品種のフローを 0 にするため、パス定式化用の品種別 forcing 行は省略した。各フロー変数には品種需要量の上限も設定して LP 緩和を強化する。たとえば `c64.dow` では、このフォールバックの 273,200 本の forcing 行を削減できる。

進捗状況、最終回答および改善内容の記述から、Codex は先に生成したプログラムコードを分析し、冗長な処理の削減に加えて、モデルの再利用や縮約などにより計算効率を改善していることが分かる。

入力した論文においても同様の計算効率化が図られているが、生成コードには、大規模モデルへの対応を扱った片山直登 (2017b) で採用されている改善も含まれている。一方、生成コードにはなお改善の余地がみられるため、適切な追加指示を与えることにより、さ

らなる効率化が可能であると考え.

続いて, データ c64 を用いて求解したときの画面出力の一部を示す.

画面出力

Capacity Scaling 反復 1

C&R Phase-I: 3 反復で収束, 列=801, forcing 行=1375

C&R cost: 27 反復で収束, 列=3349, forcing 行=6404

LP=127572.1129, 自由 y=117, C&R 反復=30, 列=3349

...

Capacity Scaling 反復 20

C&R Phase-I: 1 反復で収束, 列=4882, forcing 行=8732

C&R cost: 3 反復で収束, 列=4887, forcing 行=8740

LP=132015.9003, 自由 y=4, C&R 反復=4, 列=4887

B&B 実行可能解=130661.4, 現在上界=130278

Capacity Scaling は 20 回反復され, 4887 個の列と 8740 本の forcing 行が生成されている. 最終反復では, 自由なデザイン変数が 4 個に限定された状態で分枝限定法が実行されていることが分かる.

4.4 MIP 近傍探索法

容量スケールリング法によって得られた解を MIP 近傍探索法によって改善するプログラムコードを生成する.

以下に, 入力したプロンプトとファイル, Codex からのレスポンスおよび主なパラメータを示す.

プロンプト

添付した論文ファイルの (7) 式から (12) 式を, Gurobi を使って, Capacity Scaling Procedure, Column generation procedure と MIP Neighborhood Search を用いて解を求める Python のコードを作成する.

入力ファイル

- 論文ファイル: Katayama(2020)
- データファイル: c100_400_30_V_T_10.dow, c64.dow, r18.9.dow

進捗表示

- 式 (7)~(12) と, Capacity Scaling Procedure, Column Generation Procedure, MIP Neighborhood Search の擬似コード・パラメータを抽出します. 論文の順序と停止条件に合わせて, .dow 入力, Gurobi 求解, result.sol 追記までを新しい Python コードにまとめ, この表示文も Response.md に追記します.
- 論文の構成を確認しました. Capacity Scaling と Column Generation はパスフロー式 (7)~(12) に適用し, 得られた縮約アーク集合で制限 MIP を解いた後, 固定デザインのアークフロー問題で上界を確定します. その上界とデザインを初期解として, 式 (14)~(16) の近傍制約を付けたアークフロー MIP を反復します. 下界には, 初回容量 C で列生成を収束させた強化 LP 値を使います.
- コードを作成しました. 現在, 式 (7)~(12) の 6 要素, Algorithm 1~3 の停止・更新条件, 3 データの構造, 結果 6 項目を静的検査しています.
- 行番号で照合し, 式 (7)~(12) の目的関数・需要・容量・強化制約・変数領域および 3 手法の実装箇所を確認しました. 3 サンプルも宣言行数 (400/30, 683/400, 315/200) と実データ行数が一致しています. 次に, 利用可能なローカル Python 実行系を再確認し, 結果記録と文書の最終整合性を確認します.
- 最終検査は完了しました. 3 ファイルとも入力形式が整合し, コードには論文の式 (7)~(12), Capacity Scaling, 双対価格による Column Generation, 式 (14)~(16) の MIP Neighborhood Search, 6 項目の追記処理が揃っています.

最終回答

作成しました. solve_cnd_neighbor.py は, .dow を読み, 式 (7)~(12) を Capacity Scaling, Column Generation, MIP Neighborhood Search で解きます. 実行すると, 指定された 6 項目を result.sol に追記します.

solve_cnd_scaling.py, 説明.md, result.sol, Response.md, requirements.txt
3 サンプルは入力構造と実装内容を静的検証済みです.

主なパラメータ

- -lambda: smoothing パラメータ
- -tlr: 1 近傍探索あたりの MIP 時間制限
- -tc: 全 MIP 計算時間制限
- -neighborhood-size: 近傍探索サイズ
- -neighborhood-total-time: 近傍探索時間制限
- -beta: 近傍探索サイズの縮小係数

この試行では, 約 10 分で 828 行のプログラムコードが生成された.
続いて, 説明文に記述された MIP Neighborhood Search の解釈を示す.

MIP Neighborhood Search

Capacity Scaling のデザインを初期解として、完全アークフロー MIP に次の近傍制約を追加する。

1. 式 (14)：現在開いているアークを少なくとも 1 本閉じる。
2. 式 (15)：目的値を現在上界より ζ 以上改善する。
3. 式 (16)：現在開いているアークのうち閉じる本数を最大 M に制限する。

最初は式 (14)・(15) だけで時間 TC で解く。その後は式 (14)～(26) を付け、新しい実行可能解が得られれば *incumbent* と上界を更新する。実行不可能が証明されれば終了し、時間切れなどで実行可能解がなければ $M = \lfloor M/\beta \rfloor$ として近傍を縮小する。安全のため、論文にはない総反復上限も設けている。

以上の内容から、Codex が Katayama (2020) にある MIP Neighborhood Search を適切に解釈していることが確認できる。

容量スケールリング法の場合と同様に、Codex に次の効率化の指示を与え、コードを改善させる。

プロンプト

計算時間がかかりすぎるので、計算を効率化する。

入力ファイル

- プログラムコード：solve_cnd_neighbor.py

作業中の表示

- 対象の近傍探索コードを実データ・実行スクリプトと合わせて解析し、計算結果の意味を保ちながらボトルネックを減らします。変更後は静的検証と、可能なら実行比較を行い、変更内容を説明.md、この表示を含む応答全文を Response.md に保存します。まず現行コードと周辺ファイルを確認します。
- 主な重複は、列生成の各反復で制限 LP を全再構築していること、各品種ごとに全アーク長配列を作ること、近傍探索で巨大な全アーク×全品種モデルに品種別 forcing 制約を追加していることです。加えて、-tlr は現在「列生成全体」ではなく直近 1 回の LP 時間だけを見ており、意図した打ち切りとして機能しにくい状態です。対象フォルダーだけに書き込み権限を取得し、これらをまとめて修正します。
- 高速化の中心は、Capacity Scaling 全体で 1 個の制限 LP を保持する方式です。削除アークは対応する y とパス列の上限を 0 へ変更し、新しい列・forcing 行だけを追加します。これにより最大 250 回の Capacity Scaling と各 Column Generation 反復で繰り返していたモデル再生成をなくし、直前の LP 基底も再利用できます。近傍探索には全体時間上限も設け、-tc 3600 が最大 51 回それぞれに適用され得る長時間化を防ぎます。
- 実装を反映しました。特に c64.dow の近傍 MIP は 273,883 変数に対して従来 285,883 行あり、そのうち 273,200 行が品種別 forcing でした。フロー変数へ min(需要, 容量) の上限を直接設定し、共有容量制約と二値 y で同じ整数実行可能集合を保つことで、近傍モデルを 12,683 行まで縮小しました。Column Generation 側は列・行・LP 基底を全反復で保持します。これから説明書と表示記録を更新します。
- 最終検査では、更新コードが正しい UTF-8 で、文字列・括弧・インデント構造に不整合がないこと、実行対象データすべてでヘッダーと行数が一致することを確認しました。数理面では、永続 LR の需要・容量・forcing 係数、削除アークの上限 0、近傍 MIP の二値解における従来 forcing 行との同値性を照合済みです。

最終回答

高速化を実装し、指定フォルダーの 3 ファイルを更新しました。

- 主な変更は、制限 LR の永続化と LP 基底再利用、列・forcing 行・容量係数の差分更新、価格付け重みの疎表現、TLR の累積時間化、固定デザイン LP の隣接リスト事前計算、PFR 開始解の追加です。
- 近傍 MIP は、フロー変数上限と共有容量制約で同じ整数実行可能集合を維持しつつ、全アーク×全品種の forcing 行を省略しました。c64.dow では基礎制約が 285,883 行から 12,683 行になります。また、近傍探索全体の既定時間上限を -tc と同じ値にしました。必要なら -neighborhood-total-time で別指定できます。

この試行では、約 14 分でコードの改善が行われ、1057 行のプログラムコードが生成された。容量スケリング法の場合と同様に、Codex は先に生成したプログラムコードを分

析し、計算効率の改善を行っている。入力した論文においても同様の計算効率化が図られているが、生成コードには、大規模モデルへの対応を扱った片山 (2017b) で採用されている改善も含まれている。なお、適切な追加指示を与えることにより、さらなる効率化が可能であると考ええる。

続いて、データ r18.9 を用いて求解したときの画面出力の一部を示す。

画面出力

```
初回 LR を完全 Column Generation で解いて下界を計算します。  
初回強化 LP 下界=6088503.717, 列数=2165  
Capacity Scaling 1: active=141, deleted=174, LP=6132819.141, CG 反復=6, CG 収  
束=True  
...  
Capacity Scaling 10: active=114, deleted=1, LP=6347463.596, CG 反復=4, CG 収  
束=True  
Capacity Scaling 初期上界=6401575  
Neighborhood 初回改善: UB=6361906  
Neighborhood 全体時間上限に到達しました。
```

Column Generation により 2165 個の列が生成され、下界値が算出されている。続いて、Capacity Scaling を 10 回反復することにより自由変数が 114 個まで限定され、限定された問題を分枝限定法で解くことにより上界値が得られている。さらに、MIP 近傍探索法によって上界値が改善されていることが分かる。なお、この入力データに対する最適解を算出している。

5 数値実験

容量制約付きネットワークデザイン問題の中規模ベンチマークデータ (Katayama et al. 2009) を用いて数値実験を行い、生成されたプログラムコードによって妥当な実行可能解を求められるかを検証した。

数値実験には、C 問題から c100_400_30_V_T_10 と c64, R 問題から r18.9 を用いた。さらに、同一グループに属する小規模データとして、c100_400_10_F_L_10, c33 および r10.1 も使用した。これら 6 データに対して、生成した 4 種類のプログラムコードを適用し、上界値を求めた。

数値実験に使用した計算環境を以下に示す。

- 使用 OS および言語：Ubuntu 24.04, Python 3
- 最適化ソルバー：Gurobi 13.02
- CPU AMD Ryzen9-3950X, 16Cores, 3.5GHz
- 使用コア数：LP 1 コア, MIP 16 コア
- メモリ 64GByte

アークフローによる定式化とパスフローによる定式化では、MIP の時間制限を 3600 秒とした。一方、容量スケール法および MIP 近傍探索法では MIP を反復的に解くため、1 回あたりの MIP 時間制限を 600 秒とした。また、これら 2 つの解法ではスミージングパ

ラメータ λ が計算結果に大きく影響するため、0.05 刻みで複数の値を設定して求解し、得られた結果のうち最良の上界値を採用した。なお、この他にも複数のパラメータが存在するが、ここでは記載を省略する。

使用した 6 データについては、先行研究により上界値および下界値が得られているため、これらの値と各手法によって得られた上界値を比較する。また、先行研究で報告された上界値と計算時間も併記する。

表 1 に各定式化・各解法によって得られた上界値および下界値を示し、表 2 に既知の最良下界値に対する誤差を示す。表中の *CNDA* はアークフローによる定式化、*CNDP* はパスフローによる定式化、*SCALING* は容量スケールリング法、*NEIGH* は MIP 近傍探索法によって得られた値である。また、*BEST UB* は先行研究で得られている最良上界値、*LB* は先行研究で得られている最良下界値、*P-SCALING* は Katayama et al. (2009) による容量スケールリング法、*P-NEIGH* は片山直登 (2017a) による MIP 近傍探索法の結果を表す。なお、c64 以外の 5 データにおいて最適解が求められている。

表 1 より、Codex によって生成された 4 種類のプログラムコードは、いずれも先行研究で得られている最良上界値に近い実行可能解が求められていることが分かる。また、得られたすべての実行可能解の目的関数値は既知の最良下界値以上となっており、下界値を下回る不整合な目的関数値は算出されていない。これらの結果から、少なくとも入力したデータに対しては、生成された各コードに大きな不整合はなく、妥当な実行可能解が得られていることが確認できる。

表 2 より、容量スケールリング法では先行研究よりも誤差の小さい解を算出することができている。また、MIP 近傍探索法では先行研究と同等の誤差の解を算出することができている。

表 3 に各定式化・各解法による計算時間を示す。先行研究とは使用したプログラム言語、CPU、Gurobi のバージョンならびに開発・実行環境等が異なるため、計算時間を単純に比較することはできない。*CNDA* と *CNDP* の定式化では MIP モデルを直接求解するため、データによっては計算時間が長くなり、c64 および r18.9 では MIP の計算時間制限である 3600 秒前後までの計算を要している。

一方、容量スケールリング法では、6 データすべてを 50 秒以内に求解している。MIP 近傍探索法についても、c64 および r18.9 の計算時間はそれぞれ 644 秒および 606 秒である。なお、先行研究で使用した言語は C++ であり、Python に比べて極めて高速に処理できる言語である。また、2 つの近似解法では MIP の計算時間制限をパラメータとして設定できるため、計算時間と解の精度を考慮しながら調整することが可能である。MIP 近傍探索法の計算時間は先行研究より長いですが、MIP の計算時間制限に加えて、近傍の大きさや反復回数などの複数のパラメータを適切に調整することにより、解の精度を維持しつつ計算時間を短縮できる可能性がある。

6 おわりに

本論文では、容量制約付きネットワークデザイン問題を対象として、アークフローによる定式化、パスフローによる定式化、容量スケールリング法および MIP 近傍探索法を記述した先行研究の論文とベンチマークデータをプログラムコード生成 AI である Codex に与

え、それぞれに対応する Python プログラムコードを生成した。さらに、生成されたコードをベンチマークデータに適用して数値実験を行い、先行研究で報告された結果と比較した。その結果、生成されたコードによって先行研究と同等水準の実行可能解が得られることを確認した。

プログラムコード生成 AI である Codex は、数理計画モデルの直接的な実装だけでなく、列生成法、容量スケールリング法および MIP 近傍探索法などのアルゴリズムを含む解法についても、論文中の記述や疑似コードに加え、WEB 上にある論文等に基づいてプログラムコードとして具体化できることが確認された。また、計算時間の短縮化等のコード改善の指示により、適切な改善ができることも確認された。

本研究は科学研究費基盤研究 C(課題番号 23K04273) による成果の一部である。

参考文献

- T.G. Crainic, M. Gendreau, and B. Gendron, editors. *Network Design with Applications to Transportation and Logistics*. Springer, Switzerland, 2021.
- N. Katayama. A combined capacity scaling and local branching approach for capacitated multi-commodity network design problem. *Far East Journal of Applied Mathematics*, 92(1):1–30, 2015.
- N. Katayama. A combined matheuristic for the piecewise linear multicommodity network flow problem. *Asia-Pacific Journal of Operational Research*, 34(6), 2017.
- N. Katayama. MIP neighborhood search heuristics for a capacitated fixed-charge network design problem. *Asia-Pacific Journal of Operational Research*, 37(3), 2020.
- N. Katayama, M.Z. Chen, and M. Kubo. A capacity scaling procedure for the multi-commodity capacitated network design problem. *Journal of Computational and Applied Mathematics*, 232(2):90–101, 2009.
- 片山直登. 容量制約をもつネットワーク設計問題に対する MIP 近傍探索法. **流通経済大学流通情報学部紀要**, 22(1):1–18, 2017a.
- 片山直登. 容量制約をもつネットワーク設計の大規模問題に対する近似解法. **流通経済大学流通情報学部紀要**, 22(1):19–34, 2017b.

表 1: Upper / Lower Bounds

Data	CNDA	CNDP	SCALING	NEIGH
c100_400_10	23949	23949	24459	23949
c100_400_30	384802	384890	384809	384802
c33	423848	423933	424075	423848
c64	130581	131252	130261	130157
r10.1	200087	200383	200617	200087
r18.9	6361906	6512535	6424547	6361906
Data	BEST UB	LB	P-SCALING ^a	P-NEIGH ^b
c100_400_10	23949 ^c	23949 ^c	28426	23949
c100_400_30	384802 ^c	384802 ^c	384883	384802
c33	423848 ^c	423848 ^c	424075	423848
c64	129816 ^c	129396 ^c	130148	129982
r10.1	200087 ^d	200087 ^d	200087	200087
r18.9	6361906 ^d	6361906 ^d	6409475	6370685

a:Katayama et al. (2009) b: 片山 (2017) c: 片山 (2025) d:Katayama (2017)

表 2: Gaps (%)

Data	CNDA	CNDP	SCALING	NEIGH
c100_400_10	0.00	0.00	2.13	0.00
c100_400_30	0.00	0.02	0.00	0.00
c33	0.00	0.02	0.05	0.00
c64	0.92	1.43	0.67	0.59
r10.1	0.00	0.15	0.26	0.00
r18.9	0.00	2.37	0.98	0.00
Average	0.15	0.67	0.68	0.10
Data	BEST UB	LB	P-SCALING	P-NEIGH
c100_400_10	0.00	-	18.7	0.00
c100_400_30	0.00	-	0.02	0.00
c33	0.00	-	0.05	0.00
c64	0.32	-	0.58	0.45
r10.1	0.00	-	0.00	0.00
r18.9	0.00	-	0.75	0.14
Average	0.05	-	3.35	0.10

表 3: Computational Times(s)

Data	CNDA	CNDP	SCALING	NEIGH
c100_400_10	10	66	5	54
c100_400_30	11	51	2	39
c33	0	1	0	1
c64	3600	3622	49	644
r10.1	0	1	0	1
r18.9	2924	3609	12	606
Average	1091	1125	11	224
Data	BEST UB	LB	P-SCALING ^a	P-NEIGH ^b
c100_400_10	-	9 ^c	1	0
c100_400_30	-	10 ^c	101	20
c33	-	0 ^c	75	1
c64	-	900000 ^c	3857	12
r10.1	-	108000 ^d	2	1
r18.9	-	108000 ^d	287	68
Average	-	-	721	35

a:Katayama et al. (2009) b: 片山 (2017) c: 片山 (2025) d:Katayama (2017)