Wydział Podstawowych Problemów Techniki
Politechnika Wrocławska

Iteracyjne metody w optymalizacji kombinatorycznej na grafach

Szymon Wojtaszek
Nr indeksu: 236592

Praca inżynierska napisana
pod kierunkiem
Dr. hab. Pawła Zielińskiego

Politechnika
Wrocławska

Wrocław 2020

Spis treści

Wstęp

Optymalizacją kombinatoryczna to dział matematyki znajdujący coraz więcej zastosowań w dzisiejszych technicznym świecie. Modele optymalizacji kombinatorycznej używa się między innymi w:

- planowaniu produkcji, czy tworzeniu rozkładów lotów,
- budowie optymalnych sieci transportowych i sieci połączeń kablowych między miastami,
- organizacji pracy złożonych systemów informatycznych.

Niestety większość rzeczywistych problemów okazuje się zadaniami NP-trudnymi, co wyklucza możliwość znalezienia dla nich szybkich i dokładnych algorytmów. W pracy zaprezentujemy metodę konstrukcji algorytmów aproksymacyjnych nazywaną podejściem iteracyjnym. Następnie pokażemy kilka jej zastosowań w konstrukcji algorytmów aproksymacyjnych dla problemów optymalizacyjnych określonych na grafach oraz zaprezentujemy wyniki przeprowadzonych na nich testów.

Poza częścią pisemną praca składa się z oprogramowania załączonego na płycie w CD opisanego w rozdziale 5.

Wybrane pojęcia z optymalizacji

Na początku przedstawimy podstawowe pojęcia z optymalizacj, z których będziemy korzystać w późniejszych rozdziałach. Większość rozdziału została przygotowana na podstawie książki [5] poza podrozdziałem dotyczącym rozwiązywania dużych programów liniowych, o czym można więcej poczytać w monografii [11].

1.1 Programowanie liniowe

Zagadnienie programowania liniowego (w skrócie LP, ang. linear programming problem) to zadanie optymalizacyjne polegające na znalezieniu zbioru wartości zmiennych, minimalizujących liniową funkcję celu i spełniających pewien zbiór ograniczeń liniowych. Zadanie to można przedstawiać w kilku różnych postaciach.

Kanoniczna postać programowania liniowego jest następująca:

$$\begin{aligned} &\text{Znaleźć minimum} && z = \boldsymbol{c}^T\boldsymbol{x} \\ &\text{przy warunkach} && \boldsymbol{A}\boldsymbol{x} \geqslant \boldsymbol{b}, \\ & && \boldsymbol{x} \geqslant 0. \end{aligned} \tag{1.1}$$

gdzie $\boldsymbol{A} \in \mathbb{R}^{m\times n}$ jest daną macierzą, $m \leqslant n$, $\boldsymbol{c} \in \mathbb{R}^n$ jest wektorem kosztów, $\boldsymbol{b} \in \mathbb{R}^m$ jest wektorem ograniczeń, a $\boldsymbol{x} \in \mathbb{R}^n$ jest wektorem zmiennych decyzyjnych.

Czasami wygodniej jest ograniczenia programu wyrazić za pomocą równań, co przedstawia postać standardowa programowania liniowego:

$$\begin{aligned} &\text{Znaleźć minimum} && z = \boldsymbol{c}^T\boldsymbol{x} \\ &\text{przy warunkach} && \boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}, \\ & && \boldsymbol{x} \geqslant 0. \end{aligned} \tag{1.2}$$

Istotne jest to, że obie przedstawione postacie (jak również postać mieszana) są sobie równoważne. Model w postaci kanonicznej łatwo można przekształcić do postaci standardowej. Dla każdego wiersza i macierzy $\boldsymbol{A}$ nierówność

$$a_{i1}x_1 + a_{i2}x_2 + ... + a_{in}x_n \geqslant b_i$$

może być sprowadzona do równości przez wprowadzenie zmiennej uzupełniającej $x_{n+1} \geqslant 0$, co tworzy wiersz postaci kanonicznej:

$$a_{i1}x_1 + a_{i2}x_2 + ... + a_{in}x_n - x_{n+1} = b_i.$$

Rozwiązaniem nazywamy dowolne wartościowanie wektora zmiennych decyzyjnych $\boldsymbol{x}$. Rozwiązanie nazywamy *dopuszczalnym*, jeżeli spełnia wszystkie ograniczenia programu liniowego. Dla problemu minimalizacji rozwiązanie dopuszczalne $\boldsymbol{x}$ nazywamy optymalnym, jeżeli dla dowolnego innego rozwiązania dopuszczalnego $\boldsymbol{x'}$ zachodzi:

$$\boldsymbol{c}^T\boldsymbol{x} \leqslant \boldsymbol{c}^T\boldsymbol{x'}.$$

Rozważmy model LP w postaci standardowej (1.2) i załóżmy, że macierz $\boldsymbol{A}$ jest rzędu m. Niech $\boldsymbol{B}$ będzie dowolną nieosobliwą podmacierzą stopnia m macierzy $\boldsymbol{A}$, a $\boldsymbol{N}$ będzie pozostałą podmacierzą $\boldsymbol{A}$. Ograniczenia $\boldsymbol{Ax} = \boldsymbol{b}$ możemy zapisać w postaci:

$$\begin{bmatrix}\boldsymbol{B}, & \boldsymbol{N}\end{bmatrix}\begin{bmatrix}\boldsymbol{x_B}\\ \boldsymbol{x_N}\end{bmatrix} = \boldsymbol{b},$$

gdzie $\boldsymbol{x_B}$ i $\boldsymbol{x_N}$ mają odpowiednio, m i $m-n$ składowych.

Jeżeli $\boldsymbol{x_N} = 0$ i $\boldsymbol{x_B} = \boldsymbol{B}^{-1}\boldsymbol{b}$ to rozwiązanie $\begin{bmatrix}\boldsymbol{x_B}, & \boldsymbol{x_N}\end{bmatrix}^T$ nazywamy *rozwiązaniem bazowym* . Jeśli ponadto $\boldsymbol{x} \geqslant 0$, to mówimy, że $\boldsymbol{x}$ jest *bazowym rozwiązaniem dopuszczalnym.*

Twierdzenie 1. *Jeżeli istnieje rozwiązanie dopuszczalne to istnieje również bazowe rozwiązanie dopuszczalne. Co więcej, jeżeli istnieje rozwiązanie optymalne, to istnieje optymalne rozwiązanie bazowe.*

Twierdzenie to jest podstawą działania najbardziej popularnego algorytmu do rozwiązywania programów liniowych, algorytmu sympleks. Zaczyna on od obliczenia pierwszego rozwiązania bazowego. Następnie w umiejętny sposób przechodzi on przez rozwiązania bazowe, tak aby w każdym przejściu zmniejszyć wartość funkcji celu, do czasu znalezienia rozwiązania optymalnego, bądź stwierdzenia, że dany program ma rozwiązanie nieograniczone.

Górnym ograniczeniem iteracji potrzebnych do znalezienia rozwiązania optymalnego jest liczba rozwiązań bazowych programu, która wynosi $\binom{n}{m}$. Wynika z tego, że metoda sympleks ma ponad wielomianową złożoność co zazwyczaj dyskwalifikuje użyteczność algorytmu. Zaskakujące jest, że dla większości praktycznych przykładów liczba iteracji jest zdecydowanie mniejsza (waha się pomiędzy m, a $3m$), co w połączeniu z prostotą algorytmu decyduje o powszechnym stosowaniu algorytmu sympleks.

Prezentowana w pracy iteracyjna metoda bazuje na geometrycznej interpretacji zagadnienia programowania liniowego. Zbiór $\mathbb{U} \in \mathbb{R}$ nazywamy *wypukłym*, jeżeli dla każdej pary punktów $x, y \in \mathbb{U}$ zbiór $\mathbb{U}$ zawiera łączące je odcinek. *Punkt ekstremalny* zbioru wypukłego to punkt nie leżący wewnątrz żadnego odcinka na tym zbirze. Zbiór rozwiązań dopuszczalnych programu liniowego jest wielościanem wypukłym. Oznaczmy zbiór rozwiązań dopuszczalnych przez $\mathbb{X}$.

Twierdzenie 2. *Rozwiązanie $\boldsymbol{x} \in \mathbb{X}$ jest punktem ekstremalnym zbioru $\mathbb{X}$ wtedy i tylko wtedy, gdy $\boldsymbol{x}$ jest rozwiązaniem bazowym dopuszczalnym układu $\boldsymbol{Ax} = \boldsymbol{b}$.*

Rozwiązanie będące punktem ekstremalnym zbioru rozwiązań dopuszczalnych nazywa się czasem *rozwiązaniem ekstremalnym.* Na mocy powyższego twierdzenia optymalne rozwiązanie ekstremalne można efektywnie obliczać za pomocą algorytmu sympleks.

1.2 Rozwiązywanie dużych programów liniowych

Czas obliczeń algorytmu sympleks zależy od rozmiaru modelu programowania liniowego. Modelując problem za pomocą programowania liniowego dochodzi do sytuacji, w której pewne cechy problemu reprezentuje wykładnicza, względem rozmiaru problemu, liczba ograniczeń. Rozwiązanie takiego programu algorytmem sympleks ma wykładniczy czas działania, co jest zazwyczaj nieużyteczne. Problem wykładniczej liczby ograniczeń można rozwiązać na dwa sposoby.

Pierwszym podejściem jest próba znalezienia alternatywnego modelu programowania liniowego opisującego rozważany problem przy pomocy wielomianowej liczby ograniczeń. Jeżeli dwa programy określają ten sam zbiór rozwiązań dopuszczalnych, to można jednego używać do rozważań teoretycznych, drugiego natomiast do efektywnych obliczeń. Wykorzystujemy tą technikę w podrozdziale poświęconym minimalnym drzewom rozpinającym, jednak nie dla każdego problemu jesteśmy w stanie znaleźć model o wielomianowej liczbie ograniczeń.

Drugie podejście wywodzi się z wielomianowego algorytmu rozwiązywania modeli programowania liniowego metody elipsoidalnej. Mimo, że metoda elipsoidalna ma zastosowania tylko teoretyczne wprowadziła ona ważne narzędzie do omijania wykładniczej liczby ograniczeń.

Wyrocznia separacyjna (ang. separation oracle) to funkcja, która przyjmuje rozwiązanie $\boldsymbol{x}$, a następnie albo potwierdza, że $\boldsymbol{x}$ jest rozwiązaniem dopuszczalnym, albo zwraca ograniczenie niespełnione przez $\boldsymbol{x}$.

Jeśli dla modelu programowania liniowego o wykładniczej liczbie ograniczeń jesteśmy w stanie skonstruować wielomianową wyrocznię separacyjną, to możemy wykorzystać ją do znalezienia rozwiązania optymalnego. Upraszczamy model zostawiając w nim tylko pewien podzbiór ograniczeń. Następnie do czasu, aż wyrocznia separacyjna orzeknie, że podane rozwiązanie jest dopuszczalne, rozwiązujemy uproszczony model, przekazujemy rozwiązanie do wyroczni i dodajemy zwrócone przez nią ograniczenie do modelu. Obliczone w ten sposób rozwiązanie jest optymalne. Mimo, że nie ma gwarancji, że za pomocą opisanej metody znajdziemy rozwiązanie dopuszczalne w czasie wielomianowym, dla większości przypadków okazuje się ona niezwykle efektywna.

1.3 Programowanie całkowitoliczbowe

Programowanie całkowitoliczbowe (w skrócie IP, ang. integer programming) to bliźniacze zadanie do programowania liniowego, różniące się warunkiem, aby zmienne decyzyjne przyjmowały tylko wartości całkowite.

Wiele problemów kombinatorycznych można dokładnie modelować za pomocą programowania całkowitoliczbowego. Zazwyczaj nie prowadzi to do znalezienia rozwiązania optymalnego, gdyż obliczenie optymalnego rozwiązania programu całkowitoliczbowego jest zadaniem NP-trudnym.

Powszechną techniką jest relaksacja modelu programowania całkowitego do modelu programowania liniowego. Pozwala to uzyskać dobre ograniczenie na poszukiwaną wartość optymalną. Ponadto, kiedy zależy bardziej na czasie niż na dokładności, zaokrąglenie ułamkowego

rozwiązania może być wystarczające. Trzeba jednak uważać, ponieważ zaokrąglone rozwiązanie może nie tylko nie być optymalne, ale również może nie być rozwiązaniem dopuszczalnym.

Na znalezienie rozwiązania optymalnego są znane dokładne algorytmy o skończonym czasie działania. Jednym z nich jest metoda wykorzystująca branch and bound, rozwiązująca wielokrotnie relaksację pierwotnego problemu, dodając do niego ograniczenia na zmienne, którym wcześniej przypisano wartości niecałkowite. Kolejnym z nich jest przegląd sterowany, polegający na przejściu po zbiorze rozwiązań dopuszczalnych, których liczba jest skończona. W celu poprawy efektywności zazwyczaj implementuje się algorytmy będące kombinacją kilku różnych metod do rozwiązywania problemów programowania całkowitoliczbowego. Każdą z tych metod łączy wykładniczy czas działania, który sprawia, że przestają być użyteczne nawet dla średniej wielkości problemów.

1.4 Algorytmy aproksymacyjne

Rozwiązując dowolny problem optymalizacyjny szukamy algorytmu zwracającego wartość optymalną. Według obecnej wiedzy dla problemów NP-trudnych dokładne efektywne algorytmy nie istnieją. Możemy zrezygnować z dokładności znajdowanego rozwiązania na rzecz szybszego czasu wykonania. Rozwiązujemy wtedy problem za pomocą algorytmu aproksymacyjnego.

Dla pewnego problemu optymalizacji, niech A będzie algorytmem zwracającym rozwiązanie dopuszczalne. Dla dowolnego egzemplarza J problemu, niech $P_{opt}(J)$ oznacza wartość optymalnego rozwiązania, a $P_A(J)$ wartość rozwiązanego obliczonego przez algorytm A. Powiemy, że algorytm A jest *α-aproksymacyjny*, jeżeli

$$\max\left\{\frac{P_{opt}(J)}{P_A(J)}, \frac{P_A(J)}{P_{opt}(J)}\right\} \leqslant \alpha$$

dla dowolnego egzemplarza J rozważanego problemu.

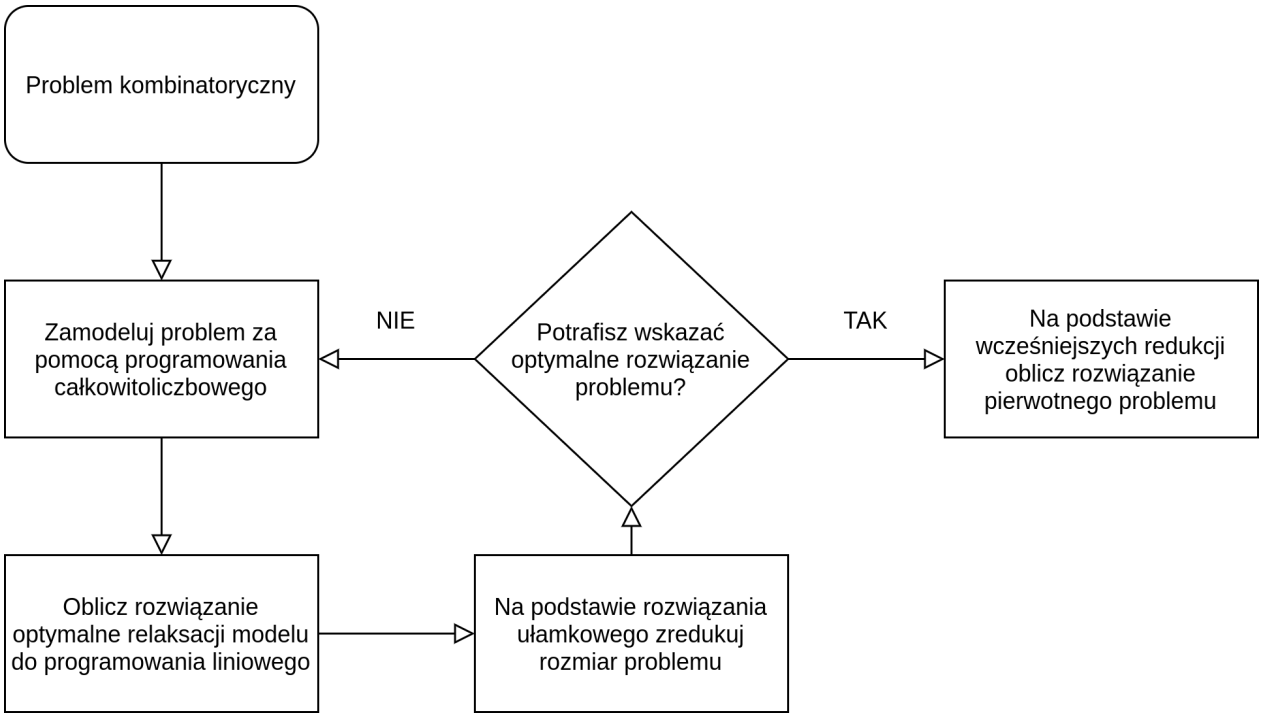
Drugim podejściem jest konstrukcja algorytmu zwracającego rozwiązanie o wartości nie gorszej od rozwiązania optymalnego, ale nie będące rozwiązaniem dopuszczalnym. Często rozluźniając delikatnie warunki zadania można wyjść z klasy problemów NP-trudnych, co skutkuje znaczącym przyśpieszeniem obliczeń. Ważne jest jednak, aby kontrolować jak mocno zwracane rozwiązanie odbiega od warunków zadania.

Iteracyjne podejście do rozwiązywania problemów kombinatorycznych

W rozdziale tym najpierw zaprezentujemy schemat iteracyjnego rozwiązywania problemów kombinatorycznych. Następnie dokładnie przeanalizujemy jego zastosowanie na dwóch prostych zagadnieniach *problem skojarzenia o maksymalnej wadze* oraz *minimalnego drzewa rozpinającego*. Algorytmy prezentowane w tym rozdziale wraz z twierdzeniami pochodzą z książki [3].

2.1 Schemat iteracyjnego podejścia do problemów

Prezentowane w pracy iteracyjne podejście polega na wielokrotnym redukowaniu rozmiaru problemów kombinatorycznych do czasu, aż staną się trywialne, bądź łatwe do rozwiązania. Redukcje dokonujemy z wykorzystaniem rozwiązań ułamkowych modeli programowania liniowego. Poniższy rysunek 2.1 wyodrębnia wspólną strukturę wszystkich przedstawionych w pracy algorytmów.



Rysunek 2.1: Schemat blokowy metody iteracyjnej.

Przedstawione w pracy algorytmy dotyczą problemów optymalizacyjnych w których szuka się podgrafów minimalizujących, bądź maksymalizujących sumę wag krawędzi. Warto zauważyć, że dla dowolnego problemu minimalizacji wartość rozwiązania optymalnego modelu liniowego, będzie zawsze nie większa niż wartość rozwiązania optymalnego modelu całkowitoliczbowego. W każdej iteracji kluczowym krokiem jest wykorzystanie ułamkowego rozwiązania do uproszczenia modelu. Zazwyczaj w tym kroku będziemy też dodawać elementy do grafu F, na którym będziemy budować całkowitoliczbowe rozwiązanie. Skuteczność metody opiera się na dokonywaniu takich redukcji, aby suma wartości optymalnego rozwiązania modeli programowania liniowego i suma wag krawędzi grafu F była nierosnąca wraz z upływem iteracji. Pozwala to indukcyjnie udowodnić graf F ma koszt nie większy niż rozwiązanie optymalne problemu.

Punktem wyjścia do poszukiwania własności rozwiązań ułamkowych jest prezentowany poniższej lemat o rzędzie. Opisuje on rozwiązania ekstremalne modeli programowania liniowego.

Lemat 1. *(Lemat o rzędzie) Niech $\boldsymbol{x}$ będzie ekstremalnym rozwiązaniem programu liniowego o zbiorze rozwiązań dopuszczalnych $P = \{\boldsymbol{x} : \boldsymbol{Ax} \leqslant \boldsymbol{b}, \boldsymbol{x} \geqslant \boldsymbol{0}\}$ takim, że każda zmienna decyzyjna x_i jest większa od zera. Wtedy liczba ograniczeń w których zachodzi równość jest równa liczbie zmiennych. Co więcej wiersze macierzy $\boldsymbol{A}$ w tych ograniczeniach tworzą zbiór wektorów liniowo niezależnych.*

2.2 Problem skojarzenia o maksymalnej wadze

Dla grafu dwudzielnego $G = (V_1 \cup V_2, E)$ z określonymi wagami w_e problem skojarzenia o maksymalnej wadze to zadanie znalezienia skojarzenia o największej łącznej wadze krawędzi. Problem dokładnie opisuje poniższy model programowania całkowitoliczbowego.

$$\begin{aligned} &\text{Znaleźć maksimum} && \sum_{e \in E} w_e x_e \\ &\text{przy warunkach} && \sum_{e \in \delta(v)} x_e \leqslant 1, \quad \forall v \in V_1 \cup V_2, \\ & && x_e \in \{0,1\}, \qquad \forall e \in E. \end{aligned} \tag{2.1}$$

Posiada on $|E|$ zmiennych decyzyjnych. Każde rozwiązanie dopuszczalne reprezentuje pewne skojarzenie. Przypisanie zmiennej x_e wartości 1 odpowiada występowaniu krawędzi e w skojarzeniu, wartość 0 wskazuje na jej pominięcie. Wyrażenie $\delta(v)$ oznacza zbiór wszystkich krawędzi incydentnych z wierzchołkiem v.

Przytoczymy iteracyjne rozwiązanie problemu przedstawione w pracy [7].

Relaksując powyższy program do liczb rzeczywistych nieujemnych dostajemy model programowania liniowego $LP_{BM}(G)$.

$$\begin{aligned} &\text{Znaleźć maksimum} && \sum_{e \in E} w_e x_e \\ &\text{przy warunkach} && \sum_{e \in \delta(v)} x_e \leqslant 1, \quad \forall v \in V_1 \cup V_2, \\ & && x_e \geqslant 0, \qquad \forall e \in E. \end{aligned} \tag{2.2}$$

Za pomocą optymalnych ułamkowych rozwiązań modeli LP_{BM} budujemy skojarzenie o maksymalnej wadze jak przedstawiono w pseudokodzie 2.1.

Niech F będzie grafem pustym o wierzchołkach tożsamych z wierzchołkami grafu G. Następnie tak długo jak w G są krawędzie następujące czynności powtarzamy w pętli. Rozwiązujemy $LP_{BM}(G)$. Usuwamy z G każdą krawędź, której zmiennej została przypisana wartość 0. Jeśli jest w G krawędź e, której zmiennej x_e została przypisana wartość 1, to dodajemy e do grafu F i usuwamy ją z G. Przechodzimy do kolejnej iteracji. Po wyjściu z pętli graf F zawiera tylko krawędzie wchodzące w maksymalne skojarzenie.

Pseudokod 2.1: Problem maksymalnego skojarzenia.

Input: Graf dwudzielny $G = (V, E_G)$ z wagami.
Output: Graf dwudzielny $F = (V, E_F)$.

```
1 F ← Graf pusty o |V| wierzchołkach.
2 while E_G ≠ ∅ do
3 |   Oblicz optymalne ekstremalne rozwiązanie x dla LP_BM(G).
4 |   Usuń krawędzie e z x_e = 0 z G.
5 |   Jeżeli istnieje krawędź e z x_e = 1, dodaj krawędź e do F oraz usuń ją z G.
6 Zwróć F.
```

Algorytm przedstawiony w pseudokodzie 2.1 poprawnie redukuje rozmiar problemu, jeżeli w każdej iteracji rozwiązanie modelu programowania liniowego przypisuje jednej zmiennej wartość 0 lub wartość 1.

Dla podzbioru krawędzi $F \subseteq E$, niech $\chi(F)$ będzie wektorem charakterystycznym F. (Tzn. wektorem w $\{0,1\}^{|E|}$, który ma współrzędną odpowiadającą każdej krawędzi $e \in E$. Współrzędna ta wynosi 1, jeżeli krawędź e jest w F, w przeciwnym przypadku wynosi 0.)

Przy bezpośrednim zastosowaniu lematu 1 do programu $LP_{BM}(G)$ wynika poniższa charakterystyka rozwiązań.

Lemat 2. *Dla dowolnego ekstremalnego rozwiązania $\boldsymbol{x}$ modelu $LP_{BM}(G)$, w którym wszystkie zmienne decyzyjne są niezerowe istnieje zbiór $W \subseteq V_1 \cup V_2$ taki, że wszystkie poniższe warunki są spełnione.*

1. $\sum_{e \in \delta(v)} x_e = 1$ *dla każdego* $v \in W$;
2. *Wektory w* $\{\chi(\delta(v)) : v \in W\}$ *są liniowo niezależne;*
3. $|W| = |E|$.

Łącząc charakterystykę rozwiązań z ich interpretacją na grafach dwudzielnych pokazano poniższy lemat zapewniający płynne działanie algorytmu.

Lemat 3. *Dowolne ekstremalne rozwiązanie $\boldsymbol{x}$ modelu programowania liniowego $LP_{BM}(G)$ musi zawierać zmienną x_e o wartości 0 lub 1.*

Twierdzenie 3. *Algorytm zawarty w pseudokodzie 2.1 zwraca skojarzenie F o maksymalnej wadze.*

2.3 Minimalne drzewo rozpinające

Zagadnienie minimalnego drzewa rozpinającego to bardzo dobrze znany problem kombinatoryczny. Mając graf spójny poszukujemy drzewa rozpinającego na tym grafie o najmniejszej możliwej sumie wag.

Dla grafu spójnego $G = (V,E)$ z określonymi wagami w_e problem minimalnego drzewa rozpinającego dokładnie opisuje poniższy model programowania całkowitoliczbowego.

$$\begin{aligned} &\text{Znaleźć minimum} && \sum_{e\in E} w_e x_e \\ &\text{przy warunkach} && \sum_{e\in E} x_e = |V| - 1, \\ & && \sum_{e\in E(S)} x_e \leqslant |S| - 1, \quad \forall S \subseteq V, |S| \geqslant 2, \\ & && x_e \in \{0,1\}, \qquad\qquad \forall e \in E. \end{aligned} \tag{2.3}$$

Jest to model o $|E|$ zmiennych decyzyjnych, których wartości odpowiadają występowaniu poszczególnych krawędzi w poszukiwanym drzewie. Dla zbioru wierzchołków $S \subseteq V$, $E(S)$ oznacza podzbiór wszystkich krawędzi z dwoma wierzchołkami w S, a $\delta(S)$ oznacza podzbiór wszystkich krawędzi z dokładnie jednym wierzchołkiem w S. Przyjmujemy konwencje $\delta(\{v\})$ oznaczać przez $\delta(v)$.

Przytoczymy iteracyjne rozwiązanie problemu przedstawione w pracy [7].

Po pominięciu ograniczenia, że zmienne mogą przyjmować tylko liczby całkowite dostajemy poniższy model programowania liniowego $LP_{MST}(G)$.

$$\begin{aligned} &\text{Znaleźć minimum} && \sum_{e\in E} w_e x_e \\ &\text{przy warunkach} && \sum_{e\in E} x_e = |V| - 1, \\ & && \sum_{e\in E(S)} x_e \leqslant |S| - 1, \quad \forall S \subseteq V, |S| \geqslant 2, \\ & && x_e \geqslant 0, \qquad\qquad \forall e \in E. \end{aligned} \tag{2.4}$$

Drugie ograniczenie programu $LP_{MST}(G)$ wprowadza wykładniczo wiele nierówności względem liczby wierzchołków. Możemy jednak efektywnie rozwiązać powyższy model liniowy przy pomocy wyroczni separacyjnej albo skorzystać z alternatywnego zwięzłego modelu.

T.L.Magnanti i L.Wolsey w [9] przedstawili model opisujący minimalne drzewo rozpinającego jako przypadek znajdowania przepływu wielotowarowego w sieci. Prowadzi do tego następująca konstrukcja. Wyróżniamy pierwszy wierzchołek grafu jako źródło, a resztę wierzchołków oznaczamy jako ujścia. Do każdego wierzchołka $k \neq 1$ wysyłamy jedną jednostkę unikatowego towaru k ze źródła, tak aby użyć przy tym krawędzi o minimalnym koszcie.

Dla grafy $G = (V,E)$ określmy zbiór łuków $A = \{(i,j) \in V \times V : \{i,j\} \in E\}$. Szukamy drzewa rozpinającego grafu skierowanego $D = (V, A)$. W rozważanym modelu występują dwa rodzaje zmiennych. Dla każdej pary $(i,j) \in A$ zmienna $y_{i,j}$ określa istnienie przepływu z wierzchołka i do j. Dla każdej pary $(i,j) \in A$ oraz towaru $k \in V \setminus \{1\}$ zmienna $f^k_{i,j}$ niech określa przepływ towaru k po strzałce (i,j).

$$
\begin{aligned}
&\text{Znaleźć minimum} && \sum_{\{i,j\}\in E} w_{i,j}(y_{ij}+y_{ji}) && \\
&\text{przy warunkach} && \sum_{(1,j)\in A} f^k_{1,j} - \sum_{(j,1)\in A} f^k_{j,1} = 1, && \forall k \in V\setminus\{1\}, \\
& && \sum_{(j,i)\in A} f^k_{j,i} - \sum_{(i,j)\in A} f^k_{i,j} = 0, && \forall i,k \in V\setminus\{1\} \wedge i \neq k, \\
& && \sum_{(j,k)\in A} f^k_{j,k} - \sum_{(k,j)\in A} f^k_{k,j} = 1, && \forall k \in V\setminus\{1\}, \\
& && \sum_{(i,j)\in A} y_{ij} = n-1, && \\
& && f^k_{ij} \leqslant y_{i,j}, && \forall (i,j) \in A \wedge \forall k \in V\setminus\{1\}, \\
& && f^k_{ij} \geqslant 0 \wedge y_{i,j} \geqslant 0, && \forall (i,j) \in A \wedge \forall k \in V\setminus\{1\}.
\end{aligned}
\tag{2.5}
$$

Mając rozwiązanie modelu programowania liniowego $\boldsymbol{x}$ określmy $N(x)$ jako *nośnik funkcji* (ang. support of x), to jest $N(x) = \{e \in E : x_e > 0\}$.

Redukcja modelu opiera się na następującym lemacie.

Lemat 4. *Dla każdego ekstremalnego rozwiązania $\boldsymbol{x}$ modelu programowania liniowego LP_{MST} istnieje pewien wierzchołek $v \in V$ taki, że maksymalnie jedna krawędź w $N(x)$ jest do niego incydentna.*

Powyższy wynik pozwala na obliczanie minimalnego drzewa rozpinającego w następujący sposób. Na początku tworzymy graf pusty F powstały z G pozbawionego krawędzi. Następnie tak długo jak G ma co najmniej dwa wierzchołki powtarzamy w pętli kolejne czynności. Rozwiązujemy $LP_{MST}(G)$. Usuwamy z G każdą krawędź e opatrzoną zmienną x_e, której została przypisana wartość 0. Następnie znajdujemy wierzchołek $v \in V$ z jedną krawędzią incydentną. Lemat 4 gwarantuje nam jego istnienie. Dodajemy tą krawędź do F. Usuwamy ją wraz z wierzchołkiem v z G i przechodzimy do kolejnej iteracji pętli.

Pseudokod 2.2: Minimalne drzewo rozpinające.

Input: Graf spójny $G = (V, E_G)$ z wagami.
Output: Graf $F = (V, E_F)$.

```
F ← Graf pusty o |V| wierzchołkach.
while V(G) ⩾ 2 do
    Oblicz optymalne ekstremalne rozwiązanie x dla LP_MST(G).
    Usuń z grafu G krawędzie e z x_e = 0.
    Znajdź wierzchołek v z jedną krawędzią incydentną e = uv.
    Dodaj krawędź e do F.
    Usuń krawędź e wraz z wierzchołkiem v z G.
Zwróć F.
```

Twierdzenie 4. *Algorytm zawarty w pseudokodzie 2.2 zwraca minimalne drzewo rozpinające.*

Zastosowanie iteratywnego podejścia do problemów NP-trudnych

3.1 Uogólniony problem przydziału

Uogólniony problem przydziału (skrót GAP, ang. generalized assignment problem) to zadanie optymalizacyjne polegające na przypisaniu zadań do maszyn, tak aby minimalizować koszt produkcji. Dodatkowo każde zadanie ma określony indywidualny czas wykonania na poszczególnych maszynach, a łączny czas pracy każdej maszyny jest ograniczony.

Formalnie problem opisuje się następująco. Dany jest zbiór zadań $J = \{1,...,n\}$ oraz zbiór maszyn $M = \{1,...,m\}$. Dla każdego zadania $j \in J$ oraz każdej maszyny $i \in M$ jest określony czas p_{ij} oraz koszt c_{ij} wykonania zadania na tej maszynie. Ponadto dla każdej maszyny $i \in M$ jest określony maksymalny czas T_i jaki maszyna może być łącznie użyta.

Problem ten dokładnie opisuje poniższy model programowania całkowitoliczbowego.

$$\begin{aligned} &\text{Znaleźć minimum} && \sum_{i=1}^{m}\sum_{j=1}^{n} c_{ij}x_{ij} \\ &\text{przy warunkach} && \sum_{i=1}^{m} x_{ij} = 1, \quad j = 1,...,n, \\ & && \sum_{j=1}^{n} x_{ij} \leqslant T_i, \quad i = 1,...,m, \\ & && x_{i,j} \in \{0,1\}, \quad i = 1,...,m, \quad j = 1,...,n. \end{aligned} \tag{3.1}$$

Uogólniony problem przydziału jest problemem NP-trudnym. W artykule [4] jest dowód twierdzenia, że jeśli klasy P i NP nie są równe to nie istnieje algorytm p-aproksymacyjny dla $p \leqslant \frac{3}{2}$. Jest tam również przedstawiony algorytm 2-aproksymacyjny rozwiązujący problem.

Odmienne obszedł ograniczenia złożoności M. Singh w swojej pracy doktorskiej [7]. Użył on iteratywnego podejścia, aby znaleźć rozwiązanie uogólnionego problemu przydziału o koszcie nie większym jak koszt dokładnego rozwiązania, dopuszcza on użycie każdej maszyny nie dłużej jak dwukrotność nałożonego ograniczenia.

W kolejnych sekcjach przytoczymy wyniki jego pracy. Dowody twierdzeń można znaleźć w pracy doktorskiej, bądź w wydanej później książce o tym samym tytule [3].

Uogólniony problem przypisania modelujemy za pomocą grafów dwudzielnych. Niech $G = (J \cup M, E)$ będzie pełnym grafem dwudzielnym z wagami, jego wierzchołki odpowiadają zbiorom zadań i maszyn. Wagi krawędzi odzwierciedlają zarówno koszt jak i czas wykonania. Zadanie sprowadza się do znalezienia grafu F będącego minimalnym skojarzeniem grafu G uwzględniającym ograniczenia czasu używania poszczególnych maszyn.

Aby osiągnąć założone wymagania względem czasu używania maszyn, musimy delikatnie oczyścić model. Dla każdej maszyny $i \in M$ należy usunąć wszystkie krawędzie łączące z zadaniami, których czas wykonania jest dłuższy, niż maksymalny czas użycia T_i tej maszyny. Operacja jest bezstratna, usuwane połączenia nie mogłyby należeć do dokładnego rozwiązania.

W trakcie obliczeń używamy poniższego modelu programowania liniowego $PL_{GAP}(G, T, M')$. Ograniczenie czasu dotyczy tylko maszyn z zbioru $M' \subseteq M$ pierwotnie inicjalizowanego przez M. W trakcie obliczeń będziemy usuwać z M' maszyny, dla których czas działania wszystkich możliwych do przypisania zadań jest mniejszy niż dwukrotność ich dopuszczalnego czasu.

$$\begin{aligned} &\text{Znaleźć minimum} && \sum_{e=(i,j)\in E} c_{ij} x_e \\ &\text{przy warunkach} && \sum_{e\in\delta(j)} x_e = 1, \quad \forall j \in J, \\ & && \sum_{e\in\delta(i)} x_e \leqslant T_i, \quad \forall i \in M', \\ & && x_e \geqslant 0, \qquad\quad \forall e \in E. \end{aligned} \tag{3.2}$$

Dla dowolnego wierzchołka $i \in J \cup M$ symbol $\delta(i)$ oznacza zbiór krawędzi do niego incydentnych.

Przy wykorzystaniu lematu 1 oraz struktury problemu można udowodnić poniższy lemat. Symbol $d(i)$ oznacza stopień wierzchołka i.

Lemat 5. *Dla dowolnego ekstremalnego rozwiązania $\boldsymbol{x}$ modelu programowania liniowego $PL_{GAP}(G,T,M')$ zachodzi jeden z poniższych warunków.*

1. *Istnieje zmienna x_e o wartości 0 lub 1;*
2. *Istnieje maszyna $i \in M'$, taka że $d(i) = 1$ lub $d(i) = 2$ i $\sum_{j\in J} x_{ij} \geqslant 1$.*

Powyższy wynik prowadzi do następującego algorytmu. Modelujemy problem za pomocą dwudzielnego grafu $G = (J \cup M, E)$. Tworzymy graf pusty F z zbiorem wierzchołków $J \cup M$. Wyodrębniamy zbiór wierzchołków M', dla których będziemy uwzględniać ograniczenie maksymalnego czasu używania maszyn, początkowo do M' przypisujemy wszystkie wierzchołki z M. Następnie do czasu, aż wszystkie zadania nie mają przypisanej maszyny, powtarzamy w pętli następujące czynności. Obliczamy optymalne rozwiązanie $\boldsymbol{x}$ modelu $PL_{GAP}(G, T, M')$. Usuwamy z grafu G każdą krawędź e, której odpowiada zmienna x_e z przypisaną wartością 0. Jeżeli w rozwiązaniu $\boldsymbol{x}$ jest zmienna x_{ij} z przypisaną wartością 1, to dodajemy odpowiadającą jej krawędź $e = (i,j)$ do F, usuwamy e z grafu F, usuwamy zadanie j z J, a dopuszczalny czas i-tej maszyny zmniejszamy o p_{ij}. Jeżeli w M' jest maszyna i o stopniu 1 lub maszyna i o stopniu 2 dla której zachodzi $\sum_{j\in J} x_{ij} \geqslant 1$ to usuwamy ją z zbioru M'. Wychodzimy z pętli. Po zakończeniu obliczeń graf F przedstawia rozwiązanie problemu.

Lemat 5 gwarantuje nam, że w każdej iteracji możemy uprościć zadanie przez znalezienie krawędzi do usunięcia z G, albo maszyny do usunięcia z M'. Dzięki wcześniejszemu oczyszczeniu danych, każdy z dwóch warunków pozwalających na usunięcie maszyny z zbioru M' zapewnia, że suma czasów pozostałych prac możliwych do przypisania maszynie jest mniejsza niż nałożone na nią ograniczenie czasowe.

Twierdzenie 5. *Algorytm zawarty w pseudokodzie 3.1 zwraca rozwiązanie uogólnionego problemu przypisania, które używa każdą maszynę i nie dłużej niż $2T_i$, a koszt rozwiązania jest nie większy niż wartość rozwiązania optymalnego.*

Pseudokod 3.1: Uogólniony problem przypisania - Singh.

Input: Graf dwudzielny $G = (V = J \cup M, E_G)$ z kosztami c_{ij} oraz czasami p_{ij}, wektor czasów dostępu maszyn T.
Output: Graf dwudzielny $F = (V, E_F)$.

1 $F \leftarrow$ Graf pusty o $|V|$ wierzchołkach.
2 $M' \leftarrow M$.
3 Dla każdej maszyny i usuń połączone nią krawędzie z grafu o czasie wykonania większym niż jej dopuszczalny czas T_i.
4 **while** $J \neq \emptyset$ **do**
5 Oblicz optymalne ekstremalne rozwiązanie $\boldsymbol{x}$ dla $LP_{GAP}(G, T, M')$.
6 Usuń z grafu G krawędzie e z $x_e = 0$.
7 Dla każdej krawędzi $e = (j,i)$ z $x_e = 1$, dodaj krawędź e do F oraz usuń e z G, usuń zadanie j z J oraz zaktualizuj $T_i \leftarrow T_i - p_{ij}$.
8 Jeżeli jest maszyna $i \in M'$ z $d(i) = 1$ lub maszyna $i \in M'$ z $d(i) = 2$ i $\sum_{j \in J} x_{ij} >= 1$, usuń ją z zbioru M'.
9 Zwróć F.

3.2 Minimalne drzewo rozpinające z ograniczeniami na stopnie wierzchołków

W tym rozdziale będziemy zajmowali się problemem znalezienia drzewa rozpinającego minimalizującego koszt, tak aby stopień każdego wierzchołka nie był większy, niż pewne określone ograniczenie.

Dany jest nieskierowany graf $G = (V,E)$, z określonym kosztem c_e dla każdej krawędzi $e \in E$ oraz z określonymi ograniczeniami b_v na stopnie wierzchołków $v \in W$, gdzie W jest pewnym podzbiorem wierzchołków. Problem minimalnego drzewa rozpinającego z ograniczeniami na stopnie wierzchołków (skrót MBST, ang. minimum-cost bounded spanning tree) to zadanie znalezienia minimalnego drzewa rozpinającego grafu G takiego, aby żaden wierzchołek $v \in W$ nie miał stopnia większego niż b_v.

Problem ten jest dokładnie modelowany przez poniższy model programowania całkowitoliczbowego.

$$\begin{aligned}
\text{minimalizuj} \quad & \sum_{e\in E} c_e x_e \\
\text{względem} \quad & \sum_{e\in E} x_e = |V| - 1, \\
& \sum_{e\in E(S)} x_e \leqslant |S| - 1, \quad \forall S \subseteq V, |S| \geqslant 2, \\
& \sum_{e\in \delta(v)} x_e \leqslant b_v, \qquad \forall v \in W, \\
& x_e \in \{0,1\}, \qquad \forall e \in E.
\end{aligned} \tag{3.3}$$

Dla dowolnego wierzchołka $v \in V$ symbol $\delta(v)$ oznacza zbiór krawędzi do niego incydentnych. Zaś dla zbioru wierzchołków $S \subseteq V$ symbol $E(S)$ oznacza zbiór krawędzi o dokładnie jednym wierzchołku w zbiorze S.

Przedstawiony problem jest problemem NP-trudnym. Zadanie obliczenia minimalnego drzewa rozpinającego z ograniczeniem na każdy wierzchołek do stopnia 2 jest zadaniem znalezienia minimalnej ścieżki hamiltona, co jest znanym problemem NP-trudnym. Pokażemy dwa różne sposoby wykorzystania metody iteracyjnej do skonstruowania algorytmu aproksymacyjnego dla powyższego problemu. Relaksację powyższego modelu całkowitoliczbowego do programowania liniowego oznaczamy $LP_{MBST}(G, B, W)$.

Pierwszy algorytm pochodzi z pracy [1]. Przedstawiono tam wielomianowy algorytm znajdujący drzewo rozpinające o koszcie nie większym, niż optymalne rozwiązanie problemu, ale dopuszczający przekroczenie ograniczeń na stopnie wierzchołków o 2. Opisana metoda w każdej iteracji dokonuje redukcji opartej na poniższym lemacie.

Lemat 6. *Dla każdego ekstremalnego rozwiązania $\boldsymbol{x}$ modelu programowania liniowego PL_{MBST} zachodzi jeden z poniższych warunków.*

1. *Istnieje wierzchołek $v \in V$ taki, że maksymalnie jedna krawędź w $N(x)$ jest do niego incydentna.*

2. *Istnieje wierzchołek $v \in W$ taki, że maksymalnie trzy krawędzie w $N(x)$ są do niego incydentne.*

Symbol $N(x)$ oznacza nośnik funkcji, to jest $N(x) = \{e \in E : x_e > 0\}$.

Własności rozwiązań ekstremalnych wynikające z powyższego lematu pozwalają na konstrukcje iteracyjnego algorytmu zaprezentowanego w pseudokodzie 3.2. W każdej iteracji redukcja rozmiaru modelu następuję albo przez znalezienie wierzchołka połączonego pozostałą częścią grafu tylko jedną krawędzią, która musi przez to należeć do budowanego drzewa, albo przez zdjęcie ograniczeń dotyczących stopnia wierzchołka, tak aby stopień tego wierzchołka w rozwiązaniu nie był większy niż wartość ograniczenia plus 2.

Twierdzenie 6. *Algorytm 3.2 zwraca drzewo rozpinające F, takie że dla każdego wierzchołka $v \in W$ stopień v jest nie większy niż $b_v + 2$, a suma wag krawędzi F jest nie większa, niż wartość optymalnego rozwiązania.*

Pseudokod 3.2: Minimalne drzewo rozpinające z ograniczeniami - Goemans.

Input: Graf $G = (V, E_G)$ z wagami, wektor ograniczeń B,
zbiór wierzchołków W.
Output: Graf $F = (V, E_F)$.

```
F ← Graf pusty o |V| wierzchołkach.
while V(G) ⩾ 2 do
    Oblicz optymalne ekstremalne rozwiązanie x dla LP_MBST(G, B, W).
    Usuń krawędzie e z x_e = 0 z G.
    if Istnieje wierzchołek v z dokładnie jedną krawędzią e = (u,v) incydentną then
        Dodaj krawędz e do F, usuń ją z G. Usuń wierzchołek v z G oraz z W.
        Jeśli u ∈ W, to zmniejsz ograniczenie b_u o jeden.
    else
        Znajdź wierzchołek v ∈ W stopnia ⩽ 3 i usuń go z W.
Zwróć F.
```

Alternatywne podejście do rozwiązania omawianego problemu zaprezentowano w pracy [8]. Pokazano tam algorytm znajdujący rozwiązanie problemu o wartości nie gorszej od optymalnego, ale dopuszczone jest przekroczenie ograniczeń na stopnie wierzchołków o 1. Punktem wyjścia do tego algorytmu jest następujący lemat.

Lemat 7. *Dla każdego ekstremalnego rozwiązania $\boldsymbol{x}$ modelu programowania liniowego PL_{MBST}, w którym $W \neq \emptyset$, istnieje wierzchołek $v \in W$ taki, że maksymalnie $b_v + 1$ krawędzi w $E(x)$ jest do niego incydentych.*

Zaproponowany algorytm jest dokładnie przedstawiony w pseudokodzie 3.3. W tym przypadku redukcja rozmiaru problemu sprowadza się do usuwania krawędzi, którym zostały przypisane wartości 0, a następnie usunięcie wierzchołków z zbioru W, których stopień jest mniejsza niż wartość nałożonego ograniczenia + 1.

Pseudokod 3.3: Minimalne drzewo rozpinające z ograniczeniami - Singh i Lau.

Input: Graf $G = (V, E_G)$ z wagami, wektor ograniczeń B,
zbiór wierzchołków W.
Output: Graf $F = (V, E_F)$.

```
F ← Graf pusty o |V| wierzchołkach.
while W ≠ ∅ do
    Oblicz optymalne ekstremalne rozwiązanie x dla LP_MBST(G, B, W).
    Usuń krawędzie e z x_e = 0 z G.
    Znajdź wierzchołek v ∈ W stopnia ⩽ b_v + 1 i usuń go z W.
Oblicz minimalne drzewo rozpinające na G, dodaj jego krawędzie do F.
Zwróć F.
```

Twierdzenie 7. *Algorytm 3.3 zwraca drzewo rozpinające F, w którym dla każdego wierzchołka $v \in W$ stopień v jest nie większy niż $b_v + 1$, a suma wag krawędzi F jest nie większa, niż wartość optymalnego rozwiązania.*

3.3 Budowa sieci odpornych na awarie

Budowa sieci odpornych na awarie (skrót SND, ang. survivable network design) to problem optymalizacyjny wywodzący się z telekomunikacji. Dany jest nieskierowany graf $G = (V,E)$ z określonym kosztem $c_e \geqslant 0$ dla każdej krawędzi $e \in E$. Ponadto dla każdej pary wierzchołków $i,j \in V$ jest podana minimalna ilość rozłącznych ścieżek r_{ij}, które muszą przebiegać pomiędzy tymi wierzchołkami. Ograniczenie to nazywamy warunkiem łączności. Zadaniem jest znaleźć minimalizujący koszt zbiór wierzchołków $F \subset E$ spełniający warunki łączności.

Opisany problem jest dokładnie modelowany przez poniższy model programowania całkowitoliczbowego.

$$\begin{aligned} &\text{Znaleźć minimum} && \sum_{e\in E} c_e x_e && \\ &\text{przy warunkach} && \sum_{e\in\delta(S)} x_e \geqslant \max_{i\in S, j\notin S} r_{ij}, && \forall S \subseteq V, \\ & && x_e \in \{0,1\}, && \forall e \in E. \end{aligned} \tag{3.4}$$

Zagadnienie budowy sieci odpornych na awarie jest uogólnieniem problemu drzewa Steinera, przez co zalicza się do klasy problemów NP-trudnych. Zaprezentujemy 2-aproksymacyjny algorytm rozwiązujący powyższy problem pochodzący z pracy [2]. Znajdują się tam również dowody przytoczonych twierdzeń.

W celu zrozumienia algorytmu niezbędne jest zrozumienia w jaki sposób powyższy program ogranicza zbiór rozwiązań dopuszczalnych tylko do rozwiązań reprezentujących sieci spełniające warunki łączności. Weźmy dowolne wierzchołki $i,j \in V$ oraz dowolne rozwiązanie $\boldsymbol{x}$. Interpretując wartości $\boldsymbol{x}$ jako dopuszczalną przepustowość na poszczególnych krawędziach, warunek aby pomiędzy i a j, było co najmniej r_{ij} rozłącznych ścieżek, oznacza że maksymalny przepływ pomiędzy i a j, musi być co najmniej r_{ij}. Co z kolei na mocy twierdzenia o maksymalnym przepływie i minimalnym przekroju, oznacza że dla dowolnego przekroju S rozdzielającego i oraz j suma zmiennych określających wchodzące w niego krawędzie jest nie mniejsza niż r_{ij}. Innymi słowy rozwiązanie $\boldsymbol{x}$ jest dopuszczalne wtedy i tylko wtedy, gdy $\sum_{e\in\delta(S)} x_e \geqslant \max_{i\in S, j\notin S} r_{ij}$ dla wszystkich $S \subseteq V$.

W każdej iteracji algorytmu będziemy wybierali część krawędzi grafu G i zapisywali je w zbiorze F. Aby modelować problem w kolejnych iteracji możemy ponownie użyć porównania do sieci przepływowej, gdzie rozwiązanie $\boldsymbol{x}$ opisuje przepustowość na krawędziach w $E \setminus F$, a krawędzie z F będą miały przepustowość 1. Dokonując podobnego rozumowania co wcześniej dochodzimy do poniższego modelu programowania liniowego $LP_{SND}(G,r,F)$.

$$\begin{aligned} &\text{Znaleźć minimum} && \sum_{e\in E} c_e x_e && \\ &\text{przy warunkach} && \sum_{e\in\delta(S)} x_e \geqslant \max_{i\in S, j\notin S} r_{ij} - |\delta(S)\cap F|, && \forall S \subset V, \\ & && x_e \geqslant 0, && \forall e \in E \setminus F. \end{aligned} \tag{3.5}$$

Dla zbioru wierzchołków $S \subseteq V$ symbol $\delta(S)$ oznacza zbiór krawędzi z dokładnie jednym wierzchołkiem w S. Warto zauważyć, że dla początkowej sytuacji kiedy zbiór $F = \emptyset$, powyższy model jest relaksacją modelu całkowitoliczbowego opisującego problem.

Korzystając z powyższego modelu dostajemy bardzo prosty algorytm zaprezentowany w pseudokodzie 3.4. Zaczynając od pustego zbioru krawędzi F, do czasu aż F nie jest rozwiązaniem dopuszczalnym, obliczamy rozwiązanie optymalne modelu $LP_{SND}(G,r,F)$. Następnie, krawędzie którym zostanie przypisana wartość większa niż $\frac{1}{2}$, dodajemy do zbioru F.

Możliwość dodania nowych krawędzi do F w każdej iteracji zapewnia nam poniższe twierdzenie.

Twierdzenie 8. *Dla każdego ekstremalnego rozwiązania $\boldsymbol{x}$ modelu $LP_{SND}(G,r,F)$ istnieje krawędź $e \in E$, dla której $x_e \geqslant \frac{1}{2}$.*

Pseudokod 3.4: Budowa sieci odpornych na awarie - Jain.

Input: Graf spójny $G = (V, E_G)$ z wagami, macierz wymagań spójności $\boldsymbol{r}$.
Output: Graf $F = (V, E_F)$.

```
1 F ← Graf pusty o |V| wierzchołkach.
2 while F nie jest rozwiązaniem dopuszczalnym do
3 |   Oblicz optymalne ekstremalne rozwiązanie x dla LP_MST(G,r,F).
4 |   Dla każdej krawędzi e z grafu G, jeżeli x_e ⩾ 1/2, dodaj krawędź e do F, a następnie
  |     usuń ją z G
5 Zwróć F
```

Twierdzenie 9. *Pseudokod 3.4 jest algrytmem 2-aproksymacyjnym dla problemu budowy sieci odpornych na awarie.*

Wyniki eksperymentów

W tym rozdziale zostaną przedstawione wyniki przeprowadzony eksperymentów na algorytmach aproksymacyjnych z poprzedniego rozdziału. Dla każdego algorytmu zaprezentujemy średni czas działania, średnie relatywne odchylenie (skót ARPD, ang. average relative percentage deviation) oraz przyjrzymy się jak rozkłada się upraszczanie modelu w poszczególnych iteracjach.

Niech J będzie zbiorem egzemplarzy, dla $j \in J$ niech $Z_h(j)$ oznacza wartość obliczonego rozwiązania, a $Z_{opt}(j)$ wartość rozwiązania optymalnego:

$$ARPD = \frac{1}{|J|} \sum_{j \in J} 100 \times \frac{Z_h(j) - Z_{opt}(j)}{Z_{opt}(j)}.$$

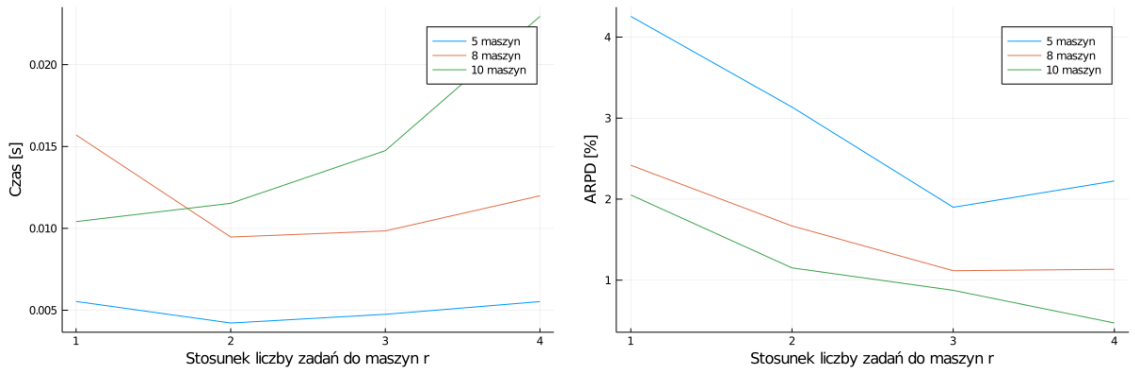
Eksperymenty zostały przeprowadzone korzystając z procesora Intel(R) Core(TM) i5-7200U o taktowaniu 2.50GHz. Algorytmy zostały zaimplementowane w języku Julia, do rozwiązywania modeli programowania liniowego użyto solvera GLPK.

4.1 Uogólniony problem przydziału

W tym podrozdziale zaprezentujemy wyniki testów przeprowadzonych na algorytmie rozwiązującym uogólniony problem przydziału. Testy zostały przeprowadzone na podstawie danych pochodzących z OR-Library [6]. Ze względu na rodzaj dostępnych danych testy zostały przeprowadzone dla maksymalizacyjnej wersji problemu.

Algorytm 3.1 został przetestowany na zbiorze 60 egzemplarzy maksymalizacyjnej wersji problemu. Egzemplarze są przedstawione w 12 różnych konfiguracjach ilości maszyn i zadań, po 5 egzemplarzy dla każdej konfiguracji. Liczba maszyn m przyjmuje wartości: 5, 8, 10, a współczynniki ilości zadań do maszyn $r = \frac{n}{m}$ wynoszą: 3, 4, 5, 6, co determinuje liczbę zadań do przypisania.

Na rysunku 4.1 zostały zaprezentowane dwa wykresy. Wykres 4.1a pokazuje średni czas obliczeń na różnych konfiguracjach egzemplarzy. Widać po nim, że zgodnie z przytoczonym twierdzeniem, algorytm działa w czasie wielomianowym zarówno pod względem wzrostu liczby maszyn, jak i zadań do przydziału. Wykres 4.1b przedstawia obliczone ARPD na każdej z badanych konfiguracji. Dodatnie wartości wynikają z przeprowadzenia eksperymentu na maksymalizacyjnej wersji problemu. Widać, że wraz z wzrostem rozmiaru danych otrzymywane wyniki są coraz bliższe wartości optymalnej. Oczywiście zgodnie z twierdzeniem 5, dostosowanym do problemu maksymalizacji, wartość wszystkich obliczonych rozwiązań była nie mniejsza, jak wartość rozwiązań optymalnych.



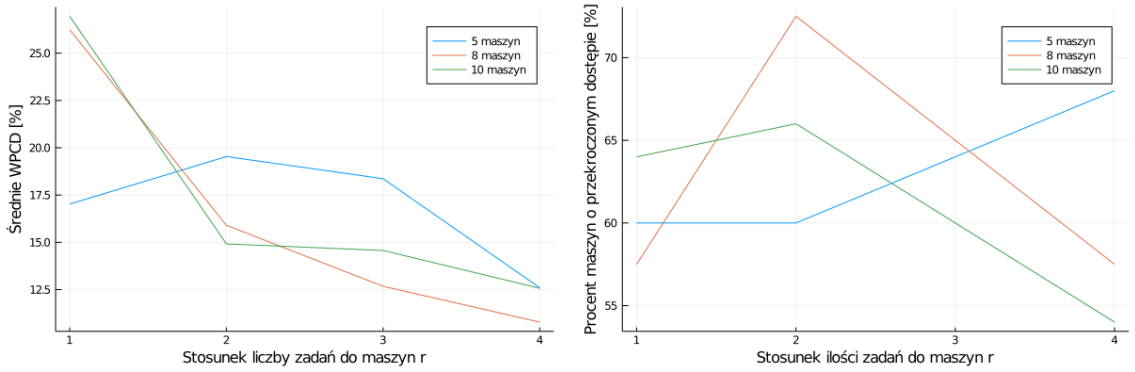
(a) (b)

Rysunek 4.1: Średni czas obliczeń oraz ARPD względem rozmiaru problemu.

Na rysunku 4.2 przedstawiono wykresy obrazujące jak mocno obliczone rozwiązanie różni się od wymagań problemu. Dla maszyny i, której zostały przypisane zadania oznaczone $\delta(i)$ względne przekroczenie czasu dostępu (skrót WPCD) określa wzór:

$$WPCD = 100 \times \frac{\sum_{j \in \delta(i)} p_{ji} - T_i}{T_i}.$$

Przypomnijmy, twierdzenie 5 daje gwarancje, że testowany algorytm zwraca rozwiązanie, w którym dla każdej maszyny WPCD wynosi mniej niż 1. Widać, że praktyce ta wartość okazuje się znacznie niższa. W wszystkich przeprowadzonych eksperymentach żadna maszyna nie miała WPCD wyższego od 40%. Licząc tylko maszyny o przekroczonym czasie działania średnie przekroczenie czasu dostępu maszyn wynosiło 17%. Z wykresu wynika, że wraz ze wzrostem liczby zadań do przydziału WPCD maleje, trudno jednak dopatrzyć się zależności WPCD od liczby maszyn. Podobnie nie widać zależności pomiędzy procentem maszyn o przekroczonym dostępie, a konfiguracją problemu.

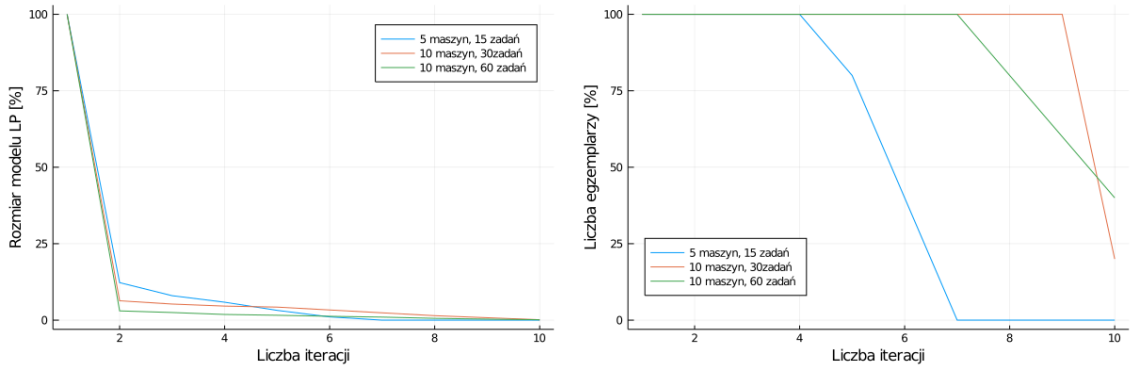


(a) (b)

Rysunek 4.2: Średnie WPCD liczone tylko z maszyn o przekroczonym dostępie oraz procent maszyn o przekroczonym dostępie względem rozmiaru problemu.

Interesujące jest jak rozmiar modelu zmienia się podczas upływu iteracji. Na rysunku 4.3 znajduje się wykres 4.3a przedstawiający średnią liczbę zmiennych decyzyjnych w modelu

na początku każdej iteracji dla trzech konfiguracji problemu. Dla czytelniejszego porównania między konfiguracjami, liczba zmiennych w danej iteracji została przedstawiona w stosunku do liczby zmiennych w modelu pierwszej iteracji. Z wykresu wynika, że niezależnie od konfiguracji problemu ponad 80% krawędzi zostaje usuwana w dwóch pierwszych iteracjach, po czym tempo upraszczania modelu znacznie spada i utrzymuję się bliskie liniowemu do końca obliczeń. Wykres 4.3b przedstawia procent liczby egzemplarzy w badanych konfiguracjach, które dochodzą do poszczególnych iteracji. Pozwala on lepiej odczytać poprzedni wykres. Mimo znacznego zmniejszenia średniego rozmiaru modelu w dwóch pierwszych iteracjach dla każdego egzemplarza algorytm wkracza w fazę powolnego dobierania rozwiązania, a liczba iteracji algorytmu dla różnych egzemplarzy danej konfiguracji jest do siebie zbliżona.



(a) (b)

Rysunek 4.3: Średni procentowy rozmiar modelu na początku każdej iteracji oraz procent egzemplarzy dochodzących do poszczególnych iteracji.

4.2 Minimalne drzewo rozpinające z ograniczeniami na stopnie wierzchołków

W tym podrozdziale zaprezentujemy wyniki testów przeprowadzonych na algorytmach znajdujących minimalne drzewo rozpinające z ograniczeniami na stopnie wierzchołków.

Dwa omówione wcześniej algorytmy zostały przetestowane na zbiorze 165 egzemplarzy problemu. Przypomnijmy, że zgodnie z twierdzeniami 6 i 7, algorytm Goemans'a (pseudokod 3.2) zwraca rozwiązanie dopuszczające przekroczenie o 2 ograniczeń na stopienie wierzchołków, a algorytm Singh'a i Lau'a (pseudokod 3.3) zwraca rozwiązanie dopuszczające przekroczenie ograniczeń o 1. Dane do doświadczeń zostały wygenerowanych w następujący sposób.

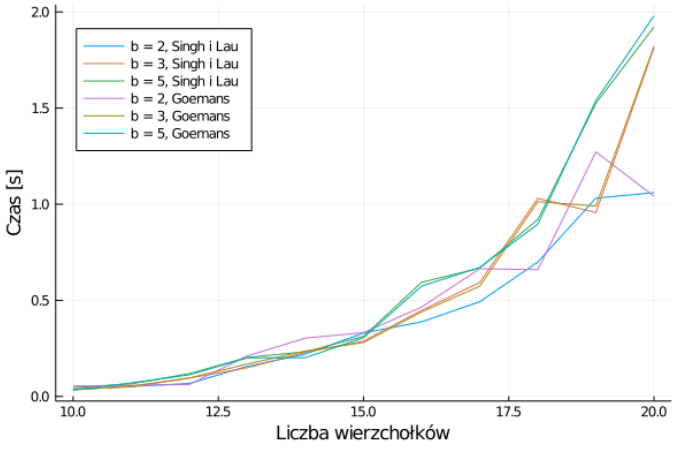
1. Dla $n \in \{10,11,12,13,14,15,16,17,18,19,20\}$ powstało 5 grafów pełnych o n wierzchołkach;

2. Każdej krawędzi została przypisana waga będąca losową liczba całkowita z przedziału [1,20];

3. Każdy z powyższych grafów został użyty w trzech egzemplarzach dla stałego ograniczeniach stopni na wszystkie wierzchołki $b \in \{2, 3, 5\}$;

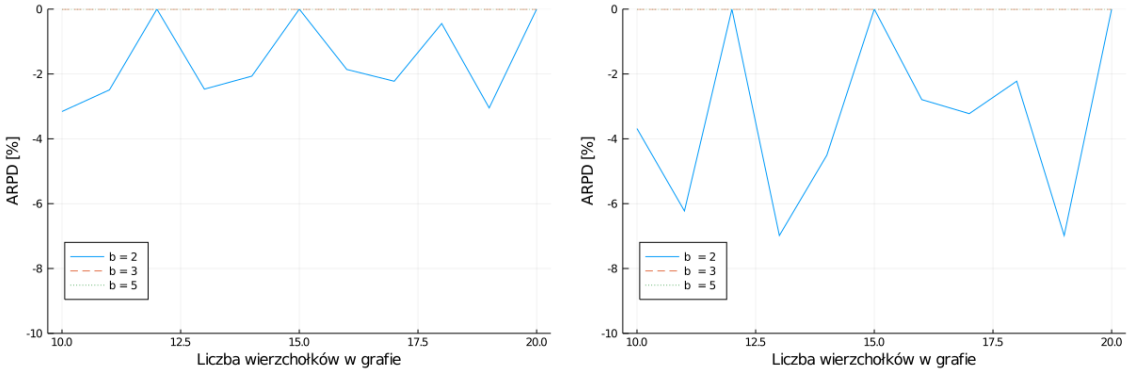
4. Wartości optymalne wygenerowanych egzemplarzy zostały obliczone za pomocą programowania całkowitoliczbowego.

Na rysunku 4.4 został zaprezentowany średni czas obliczeń w stosunku do liczby wierzchołków w grafie. Widać, że oba algorytmy cechują się zbliżonym czasem działania. Trudno dostrzec jakikolwiek związek pomiędzy wartością nałożonych ograniczeń, a czasem obliczeń.



Rysunek 4.4: Średni czas obliczeń względem rozmiaru problemu.

Na rysunku 4.5 umieszczono dwa wykresy przedstawiające ARPD rozwiązań zwracanych przez oba algorytmy na wszystkich badanych konfiguracjach. Interesujące jest, że dla wszystkich egzemplarzy problemu w których ograniczenie na stopnie wierzchołków jest stałe wynoszące 3 lub 5, wartość ARPD wynosi 0. Oznacza to, że otrzymane rozwiązanie ma wartość rozwiązania optymalnego. Jak pokazały późniejsze testy dla stałych ograniczeń wynoszących 3 lub 5, wszystkie wierzchołki zwracanych drzew miały stopnie spełniające to ograniczenie. Oznacza to, że dla tych egzemplarzy oba algorytmy zwracały rozwiązania istotnie optymalne.



(a) Przy użyciu algorytmu Goemans'a. (b) Przy użyciu algorytmu Singh'a i Lau'a.

Rysunek 4.5: Wykresy przedstawiające wartość ARPD względem rozmiaru problemu.

Tabele zaprezentowane na rysunkach 4.6 i 4.7 przedstawiają liczbę wierzchołków stopnia 3 w rozwiązaniach obliczonych dla wersji problemu z stałym ograniczeniem wynoszącym 2. Pierwszy wiersz tabel określa liczbę wierzchołków w użytych grafach. Mimo, że algorytm Goemans'a dopuszcza taką możliwość, nie było żadnego rozwiązania z wierzchołkiem stopnia 4. Analizując tabelę widać, że większość z obliczonych rozwiązań jest rozwiązaniami dopuszczalnym. Rozwiązania, które nie są rozwiązaniami dopuszczalnymi, mają znikomą liczbę wierzchołków o przekroczonych stopniach.

10	11	12	13	14	15	16	17	18	19	20
0	0	0	1	1	0	2	0	0	2	0
0	1	0	0	1	0	0	2	0	0	0
0	0	0	0	0	0	0	0	1	0	0
1	1	0	1	0	0	0	0	0	1	0
0	0	0	0	0	0	0	1	0	0	0

Rysunek 4.6: Liczba wierzchołków stopnia 3 w rozwiązaniach uzyskanych przy użyciu algorytmu Goemans'a dla wersji problemu z stałym ograniczeniem na wierzchołki 2.

10	11	12	13	14	15	16	17	18	19	20
0	0	0	1	2	0	5	0	0	4	0
0	1	0	0	1	0	0	2	0	0	0
0	0	0	0	0	0	0	0	3	0	0
3	1	0	2	0	0	0	0	0	4	0
0	0	0	0	0	0	0	2	0	0	0

Rysunek 4.7: Liczba wierzchołków stopnia 3 w rozwiązaniach uzyskanych przy użyciu algorytmu Singh'a i Lau'a dla wersji problemu z stałym ograniczeniem na wierzchołki 2.

4.3 Budowa sieci odpornych na awarie

W tym podrozdziale zaprezentujemy wyniki testów przeprowadzonych na algorytmie Jain'a (pseudokod 3.4) rozwiązującym problem budowy sieci odpornych na awarie.

Dane do testów zostały skonstruowane na podstawie archiwum „directed-germany50-DFN-aggregated-1month-over-1year.tgz" pochodzących z strony [10]. Archiwum składa się z plików xml zwierających dane o miesięcznym przepływie pakietów IP pomiędzy miastami w Niemczech. Ponadto dla każdego miasta określone są jego współrzędne geograficzne. Do doświadczeń użyto 55 egzemplarzy problemu, wygenerowanych w sposób następujący.

1. Powstał graf pełny G o 50 wierzchołkach określających miasta, wagi krawędzi zostały nadane na podstawie odległości między miastami;

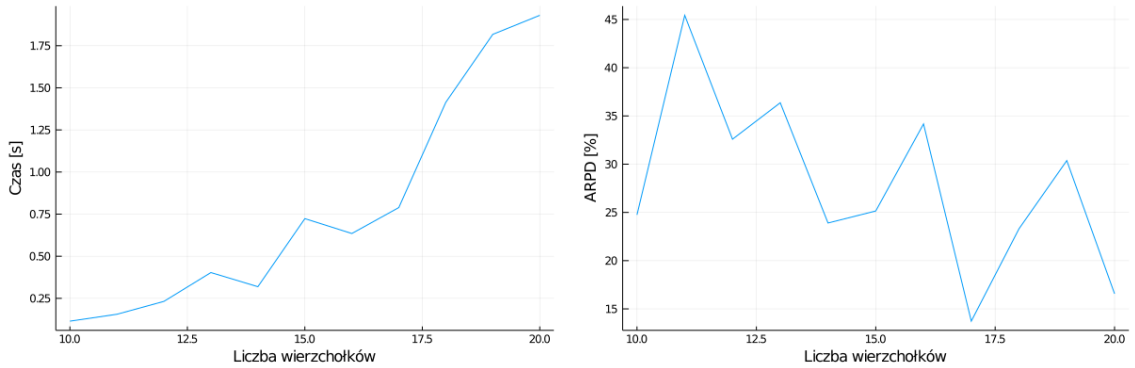
2. Na podstawie miesięcznych przepływów pakietów pomiędzy miastami zostały określone wymagania spójności pomiędzy wierzchołkami;

3. Dla każdego $n \in \{10,11,12,13,14,15,16,17,18,19,20\}$ zostało wybranych 5 pełnych podgrafów G o n losowych wierzchołkach;

4. Wartości optymalnych rozwiązań zostały obliczone za pomocą programowania całkowitoliczbowego.

.

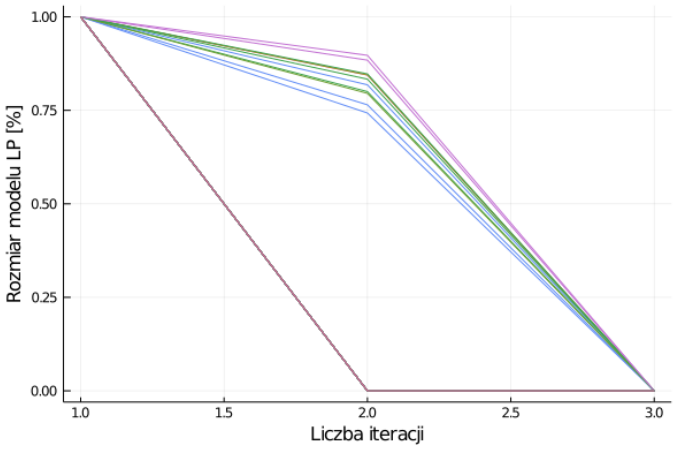
Rysunek 4.8 przedstawia średni czas obliczeń oraz ARPD wyników przeprowadzonych eksperymentów. Wykres 4.8a reprezentuje średni czas obliczeń w stosunku do ilości wierzchołków grafu, na którym budujemy sieć. Krzywa wykresu jest wyraźnie podobna do krzywej z rysunku 4.4 co może wynikać z faktu, że większość czasu obliczeń zajmuje rozwiązanie modeli programowania liniowego. Wykres 4.8b prezentuje ARPD obliczonych rozwiązań. Widać, że odchylenie wartości otrzymywanych rozwiązań od wartości rozwiązań optymalnych jest znacząco większe, niż we wcześniej prezentowanych problemach, mimo to średnie odchylenia na wykresie są znacznie mniejsze niż 100% z twierdzenia 9. Spośród wszystkich przebadanych egzemplarzy największe odchylenie wynosiło 76%, a średnia wynosiła 28%.



(a) Czas (b) ARPD

Rysunek 4.8: Średni czas obliczeń oraz ARPD względem rozmiaru problemu.

Wykres na rysunku 4.9 przedstawia stosunek liczby zmiennych modelu LP użytego w każdej iteracji względem liczby zmiennych w pierwszej iteracji. Dla zdecydowanej większości egzemplarzy problemu (oznacza je zbiorczo bordowa krzywa) wystarczyła jedna iteracja, aby obliczyć rozwiązanie. Druga iteracja konieczna była tylko dla 11 przypadków. Po wykresie widać, że szybsze uproszczenie modelu występuje na końcu obliczeń, co jest zachowaniem odwrotnym do wcześniej badanych algorytmów.



Rysunek 4.9: Rozmiar modelu LP w poszczególnych iteracjach.

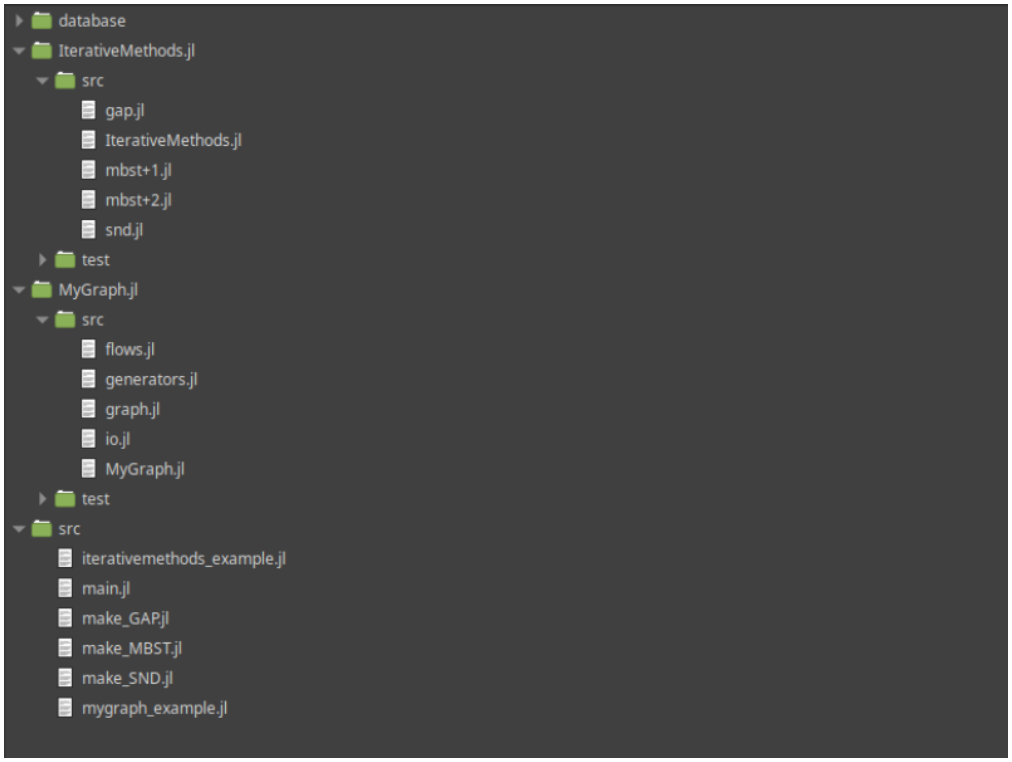
Implementacja

Wszystkie przedstawione w pracy algorytmy zostały zaimplementowane w języku Julia i zamknięte w pakiecie nazwanym `IterativeMethods.jl`. Z powodu zawodności oficjalnej biblioteki grafowej powstał też pakiet `MyGraph.jl` udostępniający możliwość operowania na grafach. Do rozwiązywania programów liniowych użyto biblioteki `JuMP` wraz z solverem `GLPK`.

Struktura plików na załączonej płycie CD została przedstawiona na rysunku 5.1. W folderze `database` znajdują się dane użyte w przeprowadzonych eksperymentach wraz z kodami programów je generujących oraz obliczających wartości rozwiązań optymalnych. W folderze `src` są trzy pliki: `make_GAP.jl`, `make_MBST.jl`, `make_SND.jl`. Jest w nich zamieszczony kod przeprowadzonych eksperymentów. Aby powtórzyć eksperymenty wystarczy przy pomocy Juli uruchomić jeden z tych plików. Można też skorzystać z pliku `main.jl`, który uruchamia je po kolei, co czyni poniższa instrukcja.

```
\$ julia ./src/main.jl
```

Wyniki testów będą się znajdować nowo utworzonym katalogu `results`.



Rysunek 5.1: Struktura katalogów.

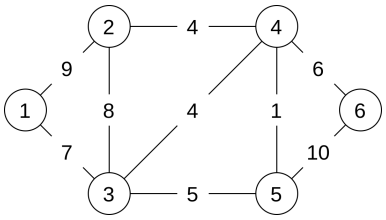
5.1 Opis pakietu MyGraph.jl

W pakiecie `MyGraph.jl` jest definicja struktury reprezentującej graf w postaci listy sąsiedztwa oraz obszerna ilość funkcji pozwalających nią zarządzać. Interfejs części funkcji najczęściej używanych w implementacji algorytmów został zaprezentowany w listingu 1.

```
# Konstruktor zwracający obiekt reprezentujący graf pusty o n wierzchołkach.
function Graph(n::Int, is_directed::Bool=false)

# Funkcje zwracające odpowiednio liczbę krawędzi i liczbę wierzchołków.
function ne(g::Graph)::Int
function nv(g::Graph)::Int

# Funkcja testująca czy graf ma krawędź z v1 do v2.
function has_edge(g::Graph, v1::Int, v2::Int)::Bool

# Funkcja dodająca krawędź pomiędzy v1 a v2 o wadze w.
function add_edge!(g::Graph, v1::Int, v2::Int, w::Float64)

# Funkcja usuwająca krawędź pomiędzy v1 a v2.
function rem_vertex!(g::Graph, v::Int)

# Funkcja zwracająca zbiór wierzchołków incydentnych do wierzchołka v.
function delta(g::Graph, v::Int)::Vector{Int}

# Funkcja zwracająca zbiór wierzchołków grafu.
function vertices(g::Graph)::Vector{Int}

# Funkcja zwracająca stopień wierzchołka v.
function degree(g::Graph, v::Int)::Int

# Funkcja testująca czy graf jest spójny.
function is_connected(g::Graph, r::Int = first(vertices(g)))::Bool

# Funkcja zwracająca wartość minimalnego przekroju pomiędzy s i t
# wraz z wierzchołkami, które znajdują się po stronie s.
function fordfulkerson(g::Graph, s::Int, t::Int)::Tuple{Float64, Array{Int, 1}}
```

Listing 1: Interfajs pakietu MyGraph.jl.

Korzystanie z pakietu prezentuje przykład w listingu 2. Kod w nim zamieszczony tworzy graf przedstawiony na rysunku 5.2. Następnie jest obliczana suma jego krawędzi oraz maksymalny możliwy przepływ pomiędzy wierzchołkami 1 i 6. Na koniec są wypisane wszystkie wierzchołki o stopniu równym 4.



Rysunek 5.2: Graf użyty w przykładzie.

```
push!(LOAD_PATH, pwd())
using MyGraph

g = Graph(6)
add_edge!(g, 1, 2, 9.0)
add_edge!(g, 1, 3, 7.0)
add_edge!(g, 2, 4, 4.0)
add_edge!(g, 2, 3, 8.0)
add_edge!(g, 3, 4, 4.0)
add_edge!(g, 3, 5, 5.0)
add_edge!(g, 4, 5, 1.0)
add_edge!(g, 4, 6, 6.0)
add_edge!(g, 5, 6, 10.0)

println("Suma wag krawędzi grafu g wynosi:")
w = weight(g)
println(w)

println("Maksymalny przepływ pomiędzy 1, a 6 wynosi:")
(flow, s_part) = fordfulkerson(g, 1, 6)
println(flow)

println("Wierzchoły stopnia 4 to:")
for v in vertices(g)
    if degree(g, v) == 4
        println(v)
    end
end
```

Listing 2: Przykład korzystania z pakietu MyGraph.jl.

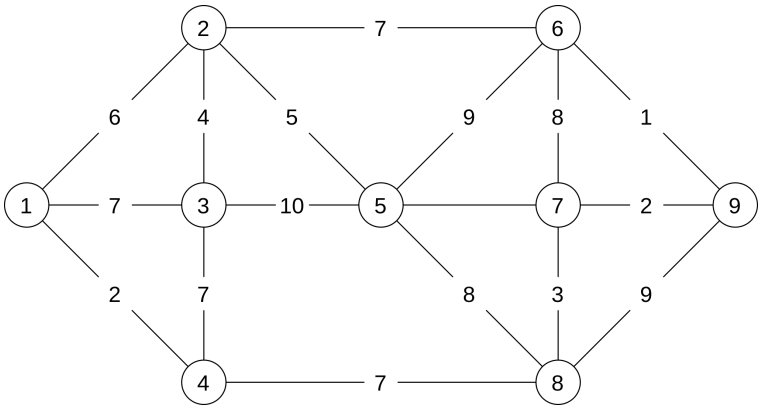
5.2 Opis pakietu IterativeMethods.jl

Pakiet `IterativeMethods` składa się z czterech niezależnych od siebie funkcji implementujących algorytmy aproksymacyjne przedstawione w tej pracy. Interfejs tych funkcji jest przedstawiony w listingu 3. Każda z tych funkcji zwraca dwójkę składającą się z grafu reprezentującego rozwiązanie oraz wektora przedstawiającego ilość zmiennych decyzyjnych w każdej iteracji algorytmu, który był użyty podczas przeprowadzania doświadczeń.

```
# Uogólniony problem przydziału
function gap(h::Graph, jobs_n::Int, processing_times::Array{Float64, 2},
     machines_times::Vector{Float64})::Tuple{Graph, Vector{Int}}

# MBST algorytm Singh'a i Lau'a
function mbst_additive_one(h::Graph, w::Set{Int}, b::Dict{Int, Int})
    ::Tuple{Graph, Vector{Int}}

# MBST algorytm Geomens'a
function mbst_additive_two(h::Graph, w::Set{Int}, b::Dict{Int, Int})
    ::Tuple{Graph, Vector{Int}}

# Budowa sieci odpornych na awarie
function snd(h::Graph, r::Array{Int, 2})::Tuple{Graph, Vector{Int}}
```

Listing 3: Interfejs pakietu IterativeMethods.jl.

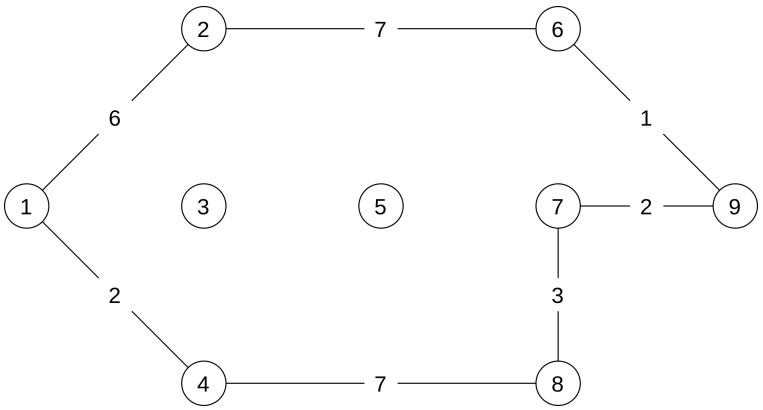
Listing 4 przedstawia wykorzystanie pakietu w celu znalezienia sieci odpornej na awarie. Budujemy w nim graf przedstawiony na rysunku 5.3, a następnie obliczamy minimalną sieć o co najmniej dwóch niezależnych połączeniach pomiędzy wierzchołkiem 1 a 9. Obliczony wynik został przedstawiony na rysunku 5.4.



Rysunek 5.3: Graf użyty w przykładzie.

```
push!(LOAD_PATH, pwd())
using MyGraph
using IterativeMethods

g = Graph(9)
add_edge!(g, 1, 2, 6.0)
add_edge!(g, 1, 3, 7.0)
add_edge!(g, 1, 4, 2.0)
add_edge!(g, 2, 3, 4.0)
add_edge!(g, 2, 6, 7.0)
add_edge!(g, 2, 5, 5.0)
add_edge!(g, 3, 4, 7.0)
add_edge!(g, 3, 5, 10.0)
add_edge!(g, 4, 8, 7.0)
add_edge!(g, 5, 6, 9.0)
add_edge!(g, 5, 7, 8.0)
add_edge!(g, 5, 8, 8.0)
add_edge!(g, 6, 7, 8.0)
add_edge!(g, 6, 9, 1.0)
add_edge!(g, 7, 8, 3.0)
add_edge!(g, 7, 9, 2.0)
add_edge!(g, 8, 9, 9.0)

r = Int.(zeros(9, 9))
r[1,9] = 2
(f, sizes) = snd(g, r)
```

Listing 4: Przykład obliczający sieć odporną na awarie.



Rysunek 5.4: Wynik obliczeń z przykładu.

Podsumowanie

W pracy przedstawiono iteracyjna metodę rozwiązywania problemów kombinatorycznych na przykładzie problemów grafowych, a następnie przedstawiono trzy jej zastosowania do konstrukcji algorytmów aproksymacyjnych dla problemów NP-trudnych. Każdy z prezentowanych algorytmów wykorzystywał rozwiązania ułamkowe uzyskane przy pomocy programowania liniowego, w celu konstrukcji rozwiązań całkowitoliczbowych o pożądanych własnościach. Następnie zostały przeprowadzone testy pokazujące efektywność opisanej metody. Wszystkie rozwiązania całkowitoliczbowe zostały znajdowane w zaledwie kilku iteracjach. Ważną obserwacją jest, że odchylenie otrzymywanych rozwiązań od rozwiązań optymalnych było znacząco mniejsze, niż teoretyczne ograniczenia podane w twierdzeniach poprawności algorytmów, co istotnie zwiększa zakres ich stosowania.

W pracy zostały zawarte tylko przykładowe algorytmy korzystające z iteratywnego podejścia. W celu pogłębienia tematu warto zajrzeć do monografii [3].

Bibliografia

[1] Michel X. Goemans. „Minimum Bounded Degree Spanning Trees". W: *47th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2006), 21-24 October 2006, Berkeley, California, USA, Proceedings*. IEEE Computer Society, 2006, s. 273–282. DOI: 10.1109/FOCS.2006.48. URL: https://doi.org/10.1109/FOCS.2006.48.

[2] Kamal Jain. „A Factor 2 Approximation Algorithm for the Generalized Steiner Network Problem". W: *Combinatorica* 21.1 (2001), s. 39–60. DOI: 10.1007/s004930170004. URL: https://doi.org/10.1007/s004930170004.

[3] Lap Chi Lau, R. Ravi i Mohit Singh. *Iterative Methods in Combinatorial Optimization*. Cambridge Texts in Applied Mathematics. Cambridge University Press, 2011. DOI: 10.1017/CBO9780511977152.

[4] Jan Karel Lenstra, David B. Shmoys i Éva Tardos. „Approximation Algorithms for Scheduling Unrelated Parallel Machines". W: *Math. Program.* 46 (1990), s. 259–271. DOI: 10.1007/BF01585745. URL: https://doi.org/10.1007/BF01585745.

[5] J. Kowalik M. Sysło N. Deo. *Algorytmy optymalizacji dyskretnej*. Wydawnictwo Naukowe PWN, 1995.

[6] *OR-Library*. URL: http://people.brunel.ac.uk/~mastjjb/jeb/info.html.

[7] Mohit Singh. „Iterative Methods in Combinatorial Optimization". 2008. URL: https://www.microsoft.com/en-us/research/publication/iterative-methods-in-combinatorial-optimization/.

[8] Mohit Singh i Lap Chi Lau. „Approximating Minimum Bounded Degree Spanning Trees to within One of Optimal". W: *J. ACM* 62.1 (mar. 2015). ISSN: 0004-5411. DOI: 10.1145/2629366. URL: https://doi.org/10.1145/2629366.

[9] L. Wolsey T.L. Magnanti. *Network Models, Handbook in Operations Research and Management Science*. Elsevier Science, 1995.

[10] *Trace Collection*. URL: https://trace-collection.net/trace-download.

[11] David P. Williamson i David B. Shmoys. *The Design of Approximation Algorithms*. Cambridge University Press, 2011. ISBN: 978-0-521-19527-0. URL: http://www.cambridge.org/de/knowledge/isbn/item5759340/?site%5Clocale=de%5CDE.