

WYDZIAŁ PODSTAWOWYCH PROBLEMÓW TECHNIKI
POLITECHNIKI WROCŁAWSKIEJ

WYBRANE PROBLEMY
ODPORNEJ OPTYMALIZACJI DYSKRETNEJ
Z MOŻLIWOŚCIĄ MODYFIKACJI

TOMASZ STRZAŁKA

Praca magisterska napisana
pod kierunkiem
dr hab. Pawła Zielińskiego, prof. PWr



Politechnika
Wrocławska

WROCŁAW 2016

Podziękowania chciałbym skierować do mojego promotora,
dr hab. Pawła Zielińskiego, prof. nadzw. Politechniki Wrocławskiej,
za nakład włożonej pracy w korektę tekstu,
poświęcony na konsultacjach czas oraz zaangażowanie,
pływające z chęci pomocy w przygotowaniu jak najlepszej pracy dyplomowej,

oraz podziękować wszystkim koleżankom i kolegom,
z którymi miałem przyjemność studiować
przez cały okres studiów magisterskich
na Politechnice Wrocławskiej, szczególnie tym paru najbliższym,
których wierzę, że z imienia nie trzeba wymieniać,
gdyż doskonale wiedzą, że to o nich mowa.



Spis treści

1 Wstęp	1
2 Podstawy	3
2.1 Grafy a drzewa rozpinające	3
2.1.1 Drzewo rozpinające	4
2.2 Problem minimalnego drzewa rozpinającego	5
2.3 Znane algorytmy rozwiązujące problem MST	7
2.3.1 Algorytm Kruskala	7
2.3.2 Algorytm Prima	9
2.4 Podsumowanie rozdziału	11
3 Odporna optymalizacja	13
3.1 Scenariusze dyskretne a ciągłe	13
3.2 Problemy minimaksowe	14
3.2.1 Przypadek ciągły	14
3.2.2 Przypadek dyskretny	15
3.2.3 Sposoby aproksymacji problemu	17
3.3 Problemy minimaksowe ze względnią funkcją optymalności	19
3.3.1 Przypadek dyskretny	19
3.3.2 Przypadek ciągły	21
3.4 Optymalizacja z możliwością poprawy	24
3.4.1 Problem przyrostowy	24
3.4.2 Zagadnienia oparte na problemie INCREMENTAL	28
3.5 Podsumowanie rozdziału	31
4 Problemy programowania liniowego	33
4.1 Programowanie liniowe	33
4.1.1 Problem minimalnego drzewa rozpinającego	34
4.1.2 Naiwne podejście	35
4.1.3 Przepływy	36
4.1.4 Przepływy z całkowitymi punktami ekstremalnymi	38
4.2 Odporne minimalne drzewo rozpinające	39
4.2.1 Relaksacja Lagrange'a	42
4.3 Podsumowanie rozdziału	45
5 Odporna optymalizacja przyrostowa	49
5.1 Konstrukcja algorytmu i warunki optymalności	49
5.2 Struktura drzewa i koszty krawędzi	51
5.3 Binarne poszukiwanie rozwiązania	55
5.4 Pełny algorytm przeszukiwania binarnego	59
5.4.1 Lepiej, szybciej	60
5.5 Analiza poprawności	63
5.6 Podsumowanie rozdziału	67

6 Odporna optymalizacja w sąsiedztwie	69
6.1 Odporna optymalizacja z możliwością poprawy	69
6.2 Algorytm zachłanny	70
6.2.1 Sąsiedztwo	70
6.2.2 Wady algorytmu zachłannego	71
6.3 Symulowane wyżarzanie	72
6.4 Przeszukiwanie z listą Tabu	73
6.4.1 Ocena ruchu	73
6.4.2 Sąsiedztwo	74
6.4.3 Losowe drzewo rozpinające i strategia dywersyfikacji	75
6.4.4 Lista ruchów zakazanych i kryterium końca	76
6.4.5 Podsumowanie rozdziału	77
7 Wyniki eksperymentalne	79
7.0.1 Środowisko testowe	79
7.1 Problem INCREMENTAL	79
7.2 Problem adwersarza	81
7.3 Tabu Search	82
7.3.1 Heurystyka a rozwiązywanie optymalne	83
7.3.2 Próg tolerancji	84
8 Zakończenie	89
A Biblioteka: RIT	95
A.1 Wymagania i instalacja	95
A.1.1 Struktura projektu	95
A.1.2 Pomocnicze skrypty	98
A.1.3 Biblioteka: log4cxx	100
A.1.4 Pozostałe biblioteki i skrypty grafowe	101
A.2 Możliwości biblioteki	103
A.2.1 Budowanie grafu	103
A.2.2 Rozwiązywanie problemów grafowych	106
A.3 Przykładowa sesja	109

Wstęp

W poniższej pracy zajmujemy się NP-trudnym, nieaproksymowalnym [13, twierdzenie 6] problemem odpornej optymalizacji dyskretnej dla minimalnego drzewa rozpinającego w wersji INCREMENTAL z możliwością poprawy rozwiązania dla dyskretnego zbioru scenariuszy. W związku z klasą, do której powyższy problem należy, będziemy stosowali podejście heurystyczne, w celu przybliżenia odnajdywanych rozwiązań do ich optymalnych odpowiedników dla konkretnych instancji problemów. Naszym celem było zbadanie sposobu zachowania się jednego z takich algorytmów (TABU SEARCH, którego opis znajdziemy w rozdziale 6) dla tak zadanego problemu, co zostało wykonane w części eksperymentalnej (patrz rozdział 7). Główną ideą rozpatrywanego problemu jest odnalezienie takiego drzewa rozpinającego w grafie, które dla małej liczby zmian (w obliczu pojawienia się nowych kosztów krawędzi) nadal pozostaje rozwiązaniem najlepszym.

Aby zaprezentować całość problemu, kolejno w rozdziałach 2 oraz 3 wyjaśniamy podstawowe pojęcia dotyczące zagadnienia minimalnego drzewa rozpinającego oraz optymalizacji odpornej, aby w następnych trzech zbudować algorytmy, służące nam ostatecznie do heurystycznego wyznaczania rozwiązań, będących możliwie najbliżej maksymalnych dolnych ograniczeń dla zadanego instancji problemu. W pracy przedstawiamy także zagadnienia dotyczące programowania liniowego (rozdział 4), na ich podstawie konstruujemy ideę, której implementacje przedstawiamy w rozdziale 5, który jest poświęcony szybkiemu algorytmowi, opartemu na przeszukiwaniu binarnym, rozwiązującemu problem INCREMENTAL MINIMUM SPANNING TREE. Tak skonstruowany algorytm, wykorzystujemy następnie w rozdziale 6, gdzie skupiamy się na głównym problemie tej pracy.

Część implementacyjna pracy magisterskiej została napisana w języku C++, zgodnego ze standardem ISO/IEC 14882:2014, z wykorzystaniem środowiska programistycznego ECLIPSE w wersji 4.5.1 (MARS), debuggerów GDB oraz VALGRIND (w wersji 3.10.1). Tekst poniższej pracy został złożony w systemie L^AT_EX, między innymi z wykorzystaniem podstawowych pakietów do reprezentacji kodu programistycznego: MINTED oraz ALGORITHM2E. Do składu wszelkich ilustracji użyto aplikacji internetowej DRAW.IO¹ oraz programu INKSCAPE w wersji 0.91. Do wygenerowania, przedstawionych w pracy, wykresów dwu- i trójwymiarowych zastosowano programy: OCTAVE (w wersji od 3.8.2 do 4.0.0) oraz CROPPDF (do ich korekty). Całość została skompilowana pod kontrolą systemu LINUX UBUNTU 15.10 (WILY WEREWOLF).

¹ Adres strony internetowej: <https://www.draw.io/>.



Podstawy

Aby w pełni móc zrozumieć przekrój problemów jaki poruszamy, niezbędne jest uprzednie zapoznanie się z podstawami, na których są one oparte. Jako że będziemy rozważali problemy odpornej optymalizacji na przykładzie minimalnych drzew rozpinających, w tym rozdziale zajmiemy się przedstawieniem podstawowych definicji dotyczących zagadnień bezpośrednio z nimi związanych. Przytoczymy definicje samego **minimalnego drzewa rozpinającego** (ang. *minimum spanning tree*), **grafów**, terminów im pochodnych oraz sposobów ich reprezentacji. Na samym końcu dokonamy przeglądu algorytmów do rozwiązywania problemu minimalnego drzewa rozpinającego (dalej będziemy często wykorzystywać skrót od ich angielskiej nazwy — **MST**) oraz przyjrzymy się właściwościom tych specyficznych struktur grafowych, na podstawie których wspomniane algorytmy funkcjonują i zwracają poprawne rozwiązania.

2.1 Grafy a drzewa rozpinające

Chcąc mówić o problemie **minimalnego drzewa rozpinającego**, musimy najpierw przypomnieć sobie definicję podstawowych zagadnień związanych z grafami. **Grafem** $G = (V, E)$ będziemy zatem nazywać zbiór punktów $v_i \in V$ opcjonalnie ze sobą połączonych krawędziami $e_{ij} \in E$, gdzie przyjmujemy następującą definicję krawędzi: $e_{ij} \equiv v_i \xrightarrow{1} v_j$, która wyraża fakt istnienia łuku (krawędzi) pomiędzy wierzchołkami (punktami) grafu (v_i oraz v_j), które są nim bezpośrednio połączone¹. Same zaś wierzchołki będziemy numerować za pomocą indeksów $i \in \{1, \dots, |V|\}$, gdzie $|V| = n$ i wyraża **moc** (liczbę elementów) zbioru wierzchołków grafu. Analogicznie będziemy przyjmować, że $|E| = m$.

Przy omawianiu problemu **minimalnego drzewa rozpinającego** będziemy rozważać tylko specyficzną rodzinę grafów: grafów nieskierowanych (ang. *undirected graphs*), które charakteryzują się tym, że fakt istnienia krawędzi e_{ij} pomiędzy wierzchołkami grafu v_i oraz v_j nie przesąduje o **kierunku** danego łuku — w przypadku grafu nieskierowanego nasza definicja łuku tak naprawdę powinna wyglądać następująco: $e_{ij} \equiv v_i \xrightarrow{1} v_j \wedge v_j \xrightarrow{1} v_i$ (zatem w grafie nieskierowanym $e_{ij} \equiv e_{ji}$, zaś stosowana w zapisie kolejność indeksów jest tylko umowna). Będziemy wymiennie, w zależności od sytuacji, stosować aż cztery rodzaje oznaczeń krawędzi e :

- e_{ij} , gdy będziemy chcieli podkreślić fakt wystąpienia krawędzi pomiędzy wierzchołkami grafu o indeksach i oraz j ,
- (i, j) , gdy będziemy chcieli położyć szczególny nacisk na połączone ze sobą wierzchołki grafu,
- e_i , gdzie w tym przypadku $i \in \{1, \dots, |E|\}$ oznacza indeks krawędzi w grafie, oraz
- $v_i \xrightarrow{1} v_j$, gdy będziemy chcieli podkreślić rozpatrywany przez nas „kierunek” krawędzi nieskierowanej.

Dodatkowo każda krawędź będzie posiadała dodatkowy atrybut, zwany przez nas dalej **kosztem** (lub **wagą**) krawędzi. Jako że bierzemy pod uwagę tylko grafy nieskierowane, koszty krawędzi e_{ij} oraz e_{ji} z definicji

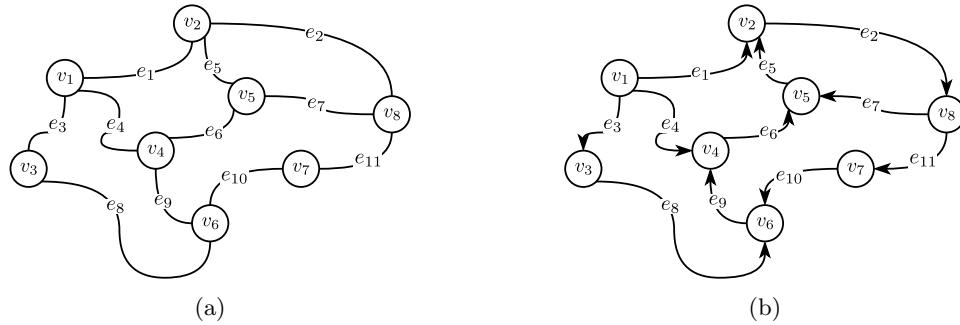
¹ Wyrażeniem $v_i \xrightarrow{k} v_j$ często też będziemy chcieli zaznaczać fakt istnienia **ścieżki** pomiędzy wierzchołkami v_i a v_j , gdzie k oznaczać będzie dokładną liczbę krawędzi, która znajduje się na takiej ścieżce między wymienionymi wierzchołkami (jeśli $k = *$, wtedy ich liczba może być dowolna, dla $k = 1$ będziemy tę liczbę pomijać). **Ścieżką** pomiędzy wierzchołkami v_i oraz v_j będziemy zaś nazywać następujący ciąg krawędzi: $\left\{ v_i \xrightarrow{1} v_{a_1}, v_{a_1} \xrightarrow{1} v_{a_2}, \dots, v_{a_b} \xrightarrow{1} v_{a_{b+1}}, v_{a_{b+1}} \xrightarrow{1} v_j \right\}$.



są takie same. Wagi krawędzi e_{ij} będziemy oznaczać jako $c_{e_{ij}}$ (ang. *cost*) lub jako c_e (w przypadku, gdy nie będą nas interesowały wierzchołki, które dana krawędź łączy), lub jako c_k (gdy będziemy chcieli odwoać się do kosztu krawędzi e o indeksie k). Pełną zatem definicję krawędzi w grafie nieskierowanym przedstawiła równość:

$$e_{ij} \equiv v_i \xrightarrow{1,c_{ij}} v_j \wedge v_j \xrightarrow{1,c_{ij}} v_i. \quad (2.1)$$

Różnicę między grafami skierowanymi a nieskierowanymi przedstawiają rysunki 2.1a oraz 2.1b — w drugim przypadku widzimy, że wiele ścieżek, możliwych do skonstruowania dla grafu nieskierowanego, jest niesiągalnych w przypadku nadania krawędziom kierunku. Dodatkowo ważnym założeniem, które przyjmiemy, będzie brak występowania w grafie wielu krawędzi o wspólnym wierzchołku początkowym oraz końcowym, czyli: $(e_i \equiv v_{s_i} \rightsquigarrow v_{t_i} \neq e_j \equiv v_{s_j} \rightsquigarrow v_{t_j}) \rightarrow s_i \neq s_j \vee t_i \neq t_j$, gdzie s_i, s_j oznaczają odpowiednio wierzchołki początkowe (źródła — ang. *sources*) krawędzi e_i oraz e_j , zaś t_i, t_j — ich węzły końcowe (cele — ang. *targets*). Grafy, które nie spełniają tej własności (posiadają więcej niż jedną krawędź prowadzącą bezpośrednio z wierzchołka początkowego v_s do węzła końcowego v_t) nazywamy **multigrafami** i nie będziemy się nimi zajmować.



Rysunek 2.1: (a) Nieskierowany graf $G = (V, E)$, gdzie $V = \{v_1, v_2, \dots, v_8\}$ i $E = \{e_1, e_2, \dots, e_{11}\}$. (b) Skierowana wersja tego samego grafu.

2.1.1 Drzewo rozpinające

Rozpoczniemy od definicji drzewa, które jest specyficzny rodzajem grafu:

Definicja 2.1.1 Drzewo jest spójnym grafem nie zawierającym żadnych cykli.

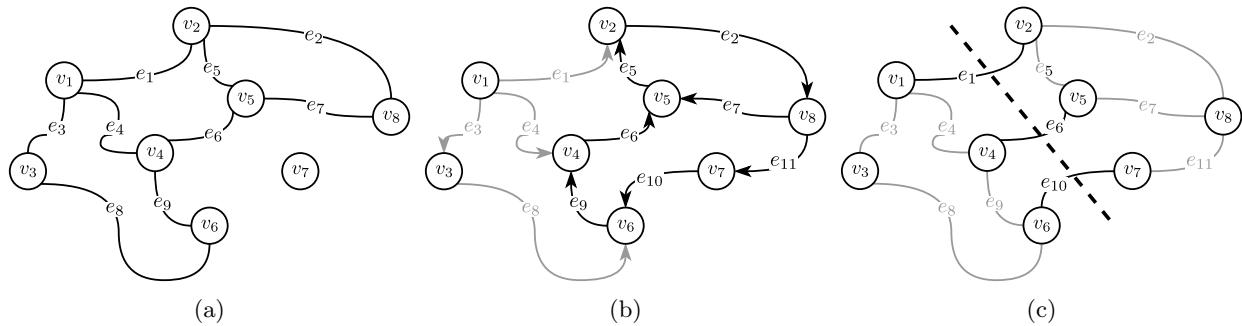
Grafem spójnym z kolei nazywamy graf, którego wszystkie wierzchołki są ze sobą w dowolny sposób połączone — do wszystkich z nich jesteśmy w stanie dojść z wykorzystaniem pewnej liczby krawędzi grafu. **Cyklem** zaś nazywamy taką ścieżkę w grafie, której wierzchołek początkowy jest jednocześnie węzłem na końcu tej ścieżki.

Drzewem rozpinającym dany graf $G = (V, E)$ będziemy nazywać taki najmniej liczny zbiór krawędzi T , który łączy ze sobą wszystkie wierzchołki w grafie. Formalnie:

$$T = \left\{ e \in E : |T| = |V| - 1 \wedge (\forall v, v' \in V : v \neq v') !\exists v \xrightarrow{*^T} v' \right\}, \quad (2.2)$$

gdzie $|T|$ symbolizuje liczbę krawędzi w zbiorze T , $v \xrightarrow{*^T} v'$ wyraża dowolnej długości ścieżkę, składającą się tylko z krawędzi generowanego zbioru T . Zbiór ten, jak widać z powyższej definicji, powinien mieć tę własność, że dla dowolnych dwóch różnych wierzchołków, należących do grafu, istnieje dokładnie jedna ścieżka pomiędzy tymi wierzchołkami. Aby przekonać się, że liczba krawędzi należących do zbioru T rzeczywiście powinna wynosić $|V| - 1$, możemy posłużyć się następującą konstrukcją:

- poprowadźmy w grafie ścieżkę, która przechodzi przez wszystkie wierzchołki dokładnie raz i kończy się w wierzchołku początkowym (zbudujmy **cykl Hamiltona**) — nie trudno zauważyc, że aby połączyć ze sobą wszystkie wierzchołki, potrzebujemy z każdego kolejnego wierzchołka poprowadzić nową krawędź. Otrzymujemy zatem cykl złożony z dokładnie $|V|$ krawędzi.



Rysunek 2.2: (a) Graf nieskierowany $G' = (V, E)$, gdzie $V = \{v_1, v_2, \dots, v_8\}$ i $E = \{e_1, e_2, \dots, e_{11}\}$, zawierający 6 cykli: $\{e_1, e_2, e_7, e_6, e_9, e_8, e_3\}$, $\{e_2, e_7, e_5\}$, $\{e_2, e_7, e_6, e_4, e_1\}$, $\{e_1, e_5, e_6\}$, $\{e_1, e_5, e_6, e_9, e_8, e_3\}$ oraz $\{e_3, e_4, e_9, e_8\}$. Graf jest niespójny — wierzchołek v_7 nie jest w żaden sposób połączony z pozostałymi wierzchołkami grafu. (b) Graf skierowany posiadający tylko dwa cykle (zaznaczone czarnym kolorem): $\{e_2, e_7, e_5\}$ oraz $\{e_2, e_{11}, e_{10}, e_9, e_6, e_5\}$. (c) Przykład cięcia w grafie.

- Usuimy teraz dowolną krawędź z cyklu. Ta operacja powoduje oczywiście jego przerwanie, zaś ze sposobu jego konstrukcji wynika, że pozostała ścieżka przechodzi kolejno przez wszystkie węzły w grafie — tworzy drzewo rozpinające o liczbie krawędzi równej $|V| - 1$.

2.2 Problem minimalnego drzewa rozpinającego

Niech \mathcal{T}_G oznacza zbiór wszystkich drzew rozpinających dla grafu G . Problem **minimalnego drzewa rozpinającego** (dalej MST) polega na znalezieniu takiego zbioru krawędzi $T^* \in \mathcal{T}_G$, że ich całkowity koszt jest najmniejszy spośród wszystkich możliwych rozwiązań. Zanim jednak bezpośrednio przejdziemy do omawiania algorytmów, które służą do odnajdywania konstrukcji o takich właściwościach, przedstawimy warunki jakie musi spełniać drzewo, abyśmy mieli pewność, że suma kosztów jego krawędzi rzeczywiście jest najmniejsza.

Pierwszym takim podejściem do zdefiniowania warunków optymalności rozwiązania T jest warunek **optymalnych cięć** (ang. *cut optimality conditions*). Znajdujący się na rysunku 2.2c przykład prezentuje cięcie grafu G — w przypadku cięcia drzew rozpinających, takie posunięcie spowoduje jego podział na dwie części, tak jak to pokazano na rysunkach 2.3a–2.3c, gdyż w wyniku zastosowania cięcia przez wybraną krawędź, jest ona usuwana z ciętego zbioru. Optymalnym cięciem natomiast będziemy nazywali takie cięcie, w wyniku którego z grafu usuwana jest krawędź o jak najmniejszym (w przypadku problemów minimalizacyjnych) koszcie ze wszystkich, znajdujących się na drodze takiego cięcia (np. na rysunku 2.2c cięcie przechodzi przez krawędzie $\{e_1, e_6, e_{10}\}$). W przypadku cięcia drzewa rozpinającego T (zobacz rysunek 2.3) przez krawędź e_6 , zbiór taki będziemy oznaczać przez $\mathcal{Q}(T, e_6)$ i będą do niego należeć wszystkie krawędzie $e' \in E$ łączące ze sobą dwa, powstałe w wyniku cięcia, zbiory wierzchołków, zgodnie z poniższą definicją.

$$\mathcal{Q}(T, e) = \{(i, j) : v_i \in V_1 \wedge v_j \in V_2\} \quad (2.3)$$

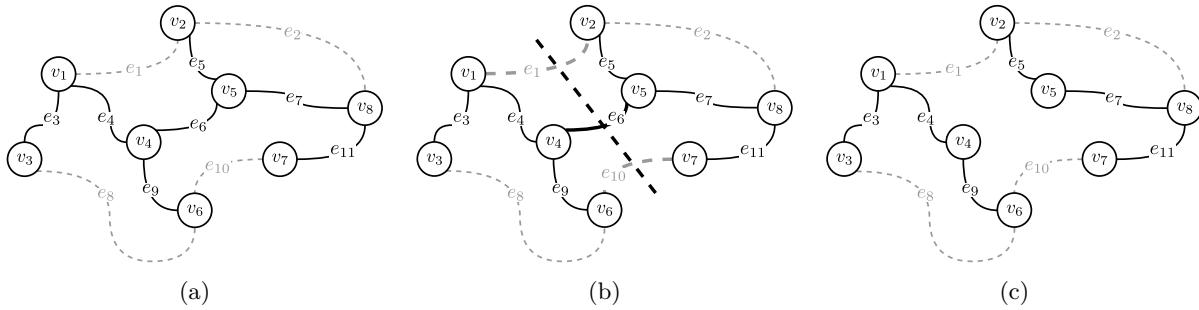
Twierdzenie 2.2.1 (Kryterium optymalnych cięć) [1, 516–518] Dla grafu $G = (V, E)$, drzewo rozpinające T^* jest minimalnym drzewem rozpinającym dany graf wtedy i tylko wtedy, gdy dla każdej krawędzi $e_{ij} \in T^*$ jej koszt c_{ij} jest najmniejszy spośród wszystkich krawędzi zbioru $\mathcal{Q}(T^*, e_{ij})$, powstałego w wyniku cięcia drzewa T^* przez krawędź e_{ij} .

Innymi słowy, przyglądając się rysunkowi 2.3b, koszt krawędzi, przez którą dokonujemy cięcia (e_6), musi być mniejszy bądź równy wagom krawędzi e_1 oraz e_{10} — własność ta powinna zachodzić dla każdego możliwego cięcia w drzewie rozpinającym (dla przykładu z omawianego rysunku, tych cięć jest jeszcze 6 — każde takie cięcie usuwa z drzewa rozpinającego inną jego krawędź). W ogólnym przypadku liczba cięć równa się liczbie krawędzi należących do drzewa rozpinającego (jedno cięcie nie może przebiegać przez wiele krawędzi $e \in T$ naraz). Argument, przemawiający za takim kryterium optymalności rozwiązania, jest łatwy do zauważenia — jeżeli krawędź e_6 (patrz 2.3b) nie miałaby najmniejszego kosztu spośród wszystkich krawędzi



należących do $\mathcal{Q}(T, e_6)$ (to jest: albo $c_1 < c_6$, albo $c_{10} < c_6$), wtedy, usuwając z drzewa rozpinającego krawędź e_6 (zrywając połączenie między wierzchołkami grafu) a dodając do niego krawędź e_1 lub e_{10} (zależnie od przypadku), stworzylibyśmy nowe drzewo rozpinające T' o koszcie oczywiście mniejszym niż koszt drzewa T . Zatem drzewo T w takiej sytuacji z pewnością nie byłoby szukanym rozwiązaniem optymalnym.

Dowód. [1, 518] Dowód w pierwszą stronę, pokazujący, że jeżeli drzewo T jest minimalnym drzewem rozpinającym to musi spełniać podane kryterium, jest bardzo prosty. Jego główną ideę zdążyliśmy już przedstawić, zatem skupimy się na pokazaniu odwrotnej zależności — jeżeli drzewo T^* spełnia warunki optymalnych cięć, musi być minimalnym drzewem rozpinającym. Założmy zatem, że drzewo T jest minimalnym drzewem rozpinającym i jest różne od T^* . Z połączenia faktów, że $|T| = n - 1 = |T^*|$ i $T \neq T^*$ otrzymujemy wniosek, że do drzewa T^* musi należeć choć jedna krawędź (niech będzie to łuk $e_{ij} \in T^*$), która nie należy do drugiego z drzew ($e_{ij} \notin T$). Usuńmy tą krawędź z T^* . Tym samym stworzymy podział drzewa T^* na dwa poddrzewa: T_1 oraz T_2 , których wierzchołki, łączone przez ich krawędzie, podzielimy na zbiory: V_1 i V_2 . Spójrzmy teraz na drzewo T . Jego krawędzie oczywiście łączą ze sobą te same zbiory wierzchołków (składa się tylko z innych krawędzi). Z definicji zaś drzewa rozpinającego wiemy, że dodając do niego jeszcze jedną krawędź, stworzymy w nim cykl. Dodajmy do niego zatem łuk, o którym wiemy, że nie należy do tego drzewa — e_{ij} . Dodając do tego drzewa dodatkową krawędź, stworzyliśmy cykl, którego elementy przynajmniej dwukrotnie przechodzą pomiędzy wierzchołkami należącymi do V_1 oraz V_2 (możemy tu odwołać się do rysunku 2.3b, gdzie przedstawione na nim drzewo to T^* , zaś usuwana z niego krawędź e_{ij} to łuk e_6 — z założenia, że $e_6 \notin T$ wiemy, że do T musi należeć przynajmniej jedna z krawędzi $\{e_1, e_{10}\}$, tak aby zachować połączenie między wierzchołkami, więc dodanie do niego łuku e_6 owocuje obecnością dwóch takich krawędzi, które łączą ze sobą węzły zbioru V_1 z tymi należącymi do V_2). Niech ta drugą krawędzią będzie krawędź e_{kl} . Drzewo T^* z założenia spełnia warunek optymalnych cięć, tak więc $c_{ij} \leq c_{kl}$ (cięższy je wzduż krawędzi c_{ij} , więc z założenia wszystkie krawędzie należące do zbioru $\mathcal{Q}(T^*, e_{ij})$ — w tym e_{kl} — mają koszty nie mniejsze od wagi e_{ij}). Dodatkowo, na początku dowodu założyliśmy, że drzewo T jest minimalnym drzewem rozpinającym graf (jest rozwiązaniem optymalnym) — jego optymalność wymusza, aby koszt krawędzi $e_{kl} \in T$ spełniał warunek $c_{kl} \leq c_{ij}$ (inaczej z pierwszej części dowodu natychmiast otrzymalibyśmy wynik, że T nie jest optymalne). Z tych dwóch nierówności otrzymujemy, że $c_{ij} = c_{kl}$. Możemy zatem bezkarnie wymienić krawędź $e_{ij} \in T^*$ na łuk $e_{kl} \notin T^*$ — otrzymane drzewo nadal będzie mieć takie same koszty (pozostanie rozwiązaniem optymalnym) a przy okazji liczba krawędzi różniących go od drzewa T (optymalnego z założenia) ulegnie zmniejszeniu. Kontynuując powyższe czynności (wymieniając krawędzie drzewa T^* , które nie należą do T na te, które są jego częścią), na pewnym etapie konstrukcji takiego drzewa okaże się, że skonstruowane drzewo jest drzewem zawierającym te same krawędzie co T — jako że po drodze ani razu nie zmienialiśmy kosztów konstruowanego drzewa, możemy wyciągnąć wniosek, że drzewo T^* (od którego wyszliśmy) od początku było optymalne, co mieliśmy udowodnić. ♦



Rysunek 2.3: (a) Drzewo rozpinające dla grafu $G = (V, E)$, gdzie $V = \{v_1, v_2, \dots, v_8\}$ i $E = \{e_1, e_2, \dots, e_{11}\}$. (b) Cięcie przez krawędź drzewa rozpinającego e_6 w grafie. Krawędzie, leżące na cięciu, zostały pogrubione. (c) Zbiory wierzchołków $V_1 = \{v_1, v_3, v_4, v_6\}$ oraz $V_2 = \{v_2, v_5, v_7, v_8\}$ powstałe w wyniku podziału drzewa rozpinającego T na mniejsze poddrzewa. Zbiór krawędzi $\mathcal{Q}(T, e_6)$, definiowany przez to cięcie, zawiera elementy: $\{e_1, e_{10}\}$.

Alternatywnym warunkiem, określającym optymalność rozwiązania problemu minimalnego drzewa rozpinającego, jest kryterium optymalnych ścieżek (ang. *path optimality conditions*), które definiujemy jako:



Twierdzenie 2.2.2 (Kryterium optymalnych ścieżek) [1, 519] Drzewo rozpinające T^* jest minimalnym drzewem rozpinającym wtedy i tylko wtedy, gdy dla każdej krawędzi spoza tego drzewa $e_{kl} \in E \setminus T^*$, dla każdej krawędzi $e_{ij} \in T^*$ należącej do ścieżki $v_k \xrightarrow{*} v_l$, zachodzi $c_{ij} \leq c_{kl}$.

Dowód. [1, 519] Pokażmy, że jeśli drzewo T^* jest minimalnym drzewem rozpinającym, to spełnia warunek optymalnych ścieżek. Dowód ten częściowo wynika z poprzedniego — jeżeli drzewo T^* jest optymalne i założymy, że istnieje taka krawędź e_{ij} na ścieżce pomiędzy wierzchołkami v_k a v_l taka, że $c_{ij} > c_{kl}$, to dodając do drzewa krawędź e_{ij} , otrzymamy cykl, którego nie wszystkie koszty krawędzi są mniejsze od wagi nowo dodanego łuku. Aby na powrót otrzymać drzewo rozpinające musimy przerwać cykl, wyrzucając z rozwiązania jedną z jego krawędzi — jako że własność optymalnej ścieżki nie była spełniona, wśród krawędzi cyklu na pewno jest łuk e , którego koszt jest większy od kosztu nowej krawędzi, zatem naturalnym posunięciem będzie usunąć właśnie ten łuk, by otrzymać jak najlepsze rozwiązanie. Usuwając go jednak otrzymujemy nowe rozwiązanie, którego koszt jest mniejszy od kosztu pierwotnego drzewa T , które było optymalne. Otrzymaliśmy zatem sprzeczność. Możemy także pokazać równoważność powyższych dwóch twierdzeń — zauważmy, że jeśli dane drzewo T^* podzielimy poprzez wykonanie cięcia wzdłuż krawędzi e_{ij} (tak jak w poprzednim dowodzie, tworząc zbiory T_1, T_2, V_1, V_2), wybierzymy dowolną krawędź e_{kl} taką, że $v_k \in V_1$ i $v_l \in V_2$, to z warunku optymalności ścieżki otrzymujemy natychmiast, że $c_{ij} \leq c_{kl}$, gdzie krawędź e_{kl} jest dowolną krawędzią należącą do $\mathcal{Q}(T^*, e_{ij})$ — przedstawiona własność jest własnością optymalnego cięcia, którą drzewo T^* spełnia, zatem na mocy poprzedniego twierdzenia jest optymalne. ♦

2.3 Znane algorytmy rozwiązuające problem MST

Problem minimalnego drzewa rozpinającego jest bardzo dobrze znany, toteż istnieje wiele algorytmów (ich wariacji) radzących sobie z danym problemem. W tej części skupimy się na dwóch podstawowych: algorytmie Josepha Kruskala [1, 520–522] i Vojtěcha Jarníka (znanego bardziej jako algorytm Prima) [1, 523–525]. Inne sposoby podejścia do problemu o jakich wspomnijmy w następnych rozdziałach to: algorytm Chazelle'ego i, również sobie z nim radzące, modele programowania liniowego oraz całkowitoliczbowego (o tych ostatnich więcej opowiemy w rozdziale 4). Algorytmami, którymi się nie będziemy zajmować są natomiast: algorytm Tarjana oraz Borůvki.

2.3.1 Algorytm Kruskala

Pierwszym algorytmem, którego schemat działania omówimy, będzie algorytm Kruskala, który w bardzo dużym stopniu polega na, udowodnionym przez nas, kryterium optymalności drzewa rozpinającego — optymalnych ścieżek (zobacz twierdzenie 2.2.2). Jak pamiętamy, zgodnie z podanym kryterium, drzewo rozpinające T jest minimalnym drzewem rozpinającym grafu G tylko wtedy, gdy wszystkie krawędzie nienależące do tego drzewa, zaś należące do ścieżki między dwoma wierzchołkami, które łączy krawędź należąca do T , mają koszt nie większy niż waga tej ostatniej. Definicja ta bezpośrednio przekłada się na ideę algorytmu: będziemy chcieli kolejno dodawać do naszego rozwiązania krawędzie w kolejności od ich najmniejszego kosztu do największej wagi, konstruując przy tym coraz to dłuższe ścieżki, aż do momentu, w którym na ścieżkach nie zaczną pojawiać się cykle. Dzięki uprzedniemu posortowaniu krawędzi względem ich kosztów, mamy w tym przypadku pewność, że na takiej ścieżce znajdują się tylko krawędzie o najniższych kosztach, zaś wszystkie pozostałe łuki, które leżały na tej ścieżce (a których nie możemy już dodać ze względu na pojawienie się cyklu), mają większy koszt krawędzi niż ostatni łuk dodany do ścieżki. Naszym głównym celem zatem jest konstruowanie ścieżek — w linii 8 pseudokodu 1 sprawdzamy, czy oba wierzchołki, które łączy analizowana przez nas krawędź nie należą do tego samego zbioru. Jeśli tak jest — krawędź, którą chcemy dodać, utworzyłaby cykl na ścieżce, do której należą oba te wierzchołki, zatem nie chcemy dodawać takiej krawędzi. W przypadku odwrotnym (gdy krawędź łączy różne zbiory — $V \neq V'$) musimy połączyć ze sobą zbiory V i V' , które od tej chwili będą reprezentować nową, większą ścieżkę. Oczywiście, aby algorytm działał poprawnie, zakładamy, że zbiór krawędzi, po którym iterujemy (7–11) jest odpowiednio posortowany, co zapewniamy sobie w linii 6. Całość prezentuje się w formie pseudokodu zamieszczonego w 1. Czas działania takiego algorytmu oczywiście zależy od sposobu zaimplementowania linii 6 oraz 8–11, lecz dolna jego granica to $O(m + n \cdot \log(n))$ [1, 522].

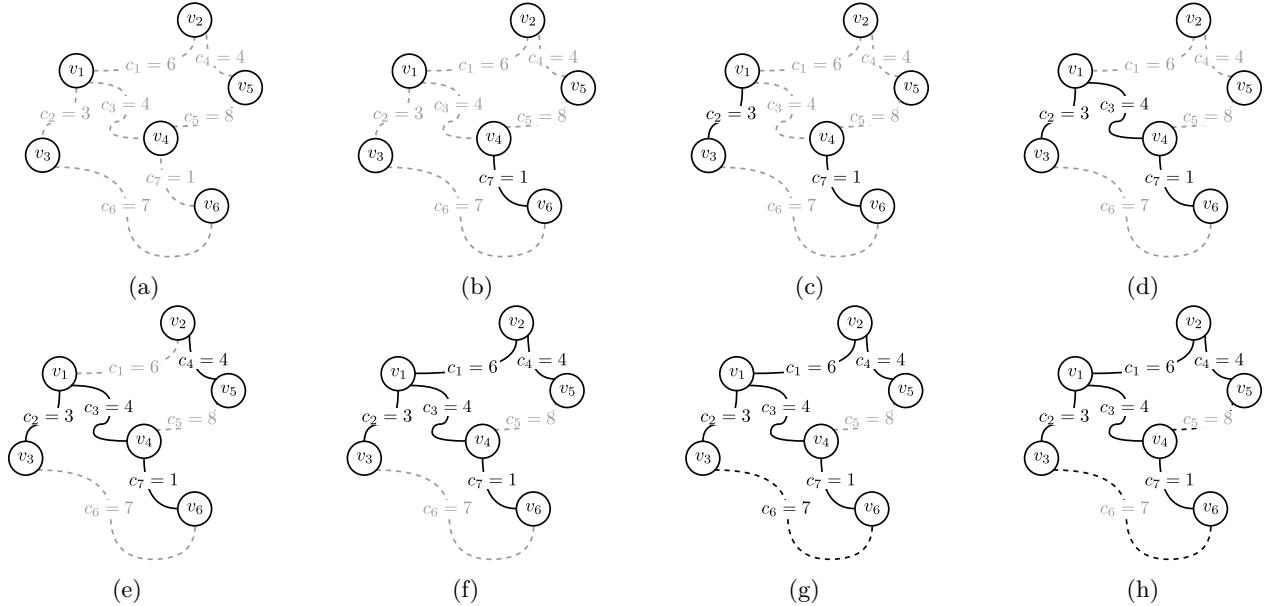
**Pseudokod 1: KRUSKAL-MST (G)**

Wejście: $G = (V, E)$ — graf wejściowy,
Wyjście: T^* — minimalne drzewo rozpinające.

```

1 begin
2    $T^* \leftarrow \emptyset$ 
3    $\mathcal{V} \leftarrow \emptyset$ 
4   foreach  $v \in V$  do
5      $\mathcal{V} \leftarrow \mathcal{V} \cup \{v\}$ 
6   INC-ORDER ( $E$ )
7   foreach  $e_{ij} \in E$  do
8     if  $(V : v_i \in V) \neq (V' : v_j \in V')$  then
9        $T^* \leftarrow T^* \cup e_{ij}$ 
10       $\mathcal{V} \leftarrow \mathcal{V} \setminus \{V, V'\}$ 
11       $\mathcal{V} \leftarrow \mathcal{V} \cup \{v : v \in V \cup V'\}$ 
12   return  $T^*$ 

```



Rysunek 2.4: Przebieg algorytmu Kruskala dla grafu $G = (V, E)$, gdzie $|V| = 6$, $|E| = 7$. (a) Początkowa sytuacja w grafie G . (b) Wybór krawędzi $e_7 \equiv e_{46}$ o najmniejszym koszcie ze zbioru nie wybranych jeszcze łuków (zaznaczone szarymi przerywanymi liniami). Usunięcie ze zbioru \mathcal{V} zbiorów $V_4 = \{v_4\}$ oraz $V_6 = \{v_6\}$ i utworzenie nowego zbioru, będącego sumą tych usuniętych: $V_7 = \{v_4, v_6\}$. (c) Krawędź o najmniejszym koszcie w zbiorze $E \setminus T^*$ ($e_2 \equiv e_{13}$) została dodana do rozwiązania. Równolegle $V_8 = \{v_1, v_3\}$ został dodany do zbioru zbiorów \mathcal{V} . (d) Krawędzie $e_3 \equiv e_{14}$ oraz $e_4 \equiv e_{25}$ mają ten sam koszt — algorytm losowo wybiera jedną z nich (e_3) a następnie dodaje ją do rozwiązania. Wierzchołki, które łączy dodana do rozwiązania krawędź, należą do zbiorów V_7 oraz V_8 — analogicznie jak w poprzednich przypadkach, zbiorów te są usuwane z \mathcal{V} a na ich miejsce wstawiany jest zbiór $V_9 = \{v_3, v_1, v_4, v_6\}$. (e) Wybranie drugiej krawędzi o tym samym koszcie i dodanie jej do rozwiązania. $\mathcal{V} = \{V_9, V_{10}\}$, gdzie $V_{10} = \{v_2, v_5\}$. (f) Następna w kolejności krawędzią, wedle rosnących kosztów, jest krawędź e_1 łącząca wierzchołki należące do zbiorów V_9 oraz V_{10} ($V_9 \ni v_1 \rightsquigarrow v_2 \in V_{10}$). W wyniku usunięcia obu zbiorów wierzchołków $\mathcal{V} = \{V_{11}\}$, gdzie V_{11} zawiera już wszystkie wierzchołki należące do grafu. W tym momencie, w przypadku zauważenia takiej własności powstałego zbioru, można zakończyć algorytm. (g) Krawędź e_6 łączy ze sobą wierzchołki v_3 oraz v_6 , z czegooba należą do zbioru V_{11} , w związku z czym krawędź jest pomijana. (h) Kolejna z analizowanych krawędzi łączy wierzchołki należące do tego samego zbioru. Dodanie jej do rozwiązania spowodowałoby powstanie cyklu w budowanym drzewie.

2.3.2 Algorytm Prima

Kolejnym algorytmem, którego sposób działania bezpośrednio wynika z warunków optymalności drzewa rozpinającego (zobacz dowody twierdzeń 2.2.1 i 2.2.2), jest algorytm Prima. W odróżnieniu od poprzednio przedstawionego algorytmu, ten w działaniu opiera się na kryterium optymalnych cięć a jego implementacja jest równie nieskomplikowana jak w poprzednim przypadku (zobacz pseudokod 2). Przypomnijmy, że według kryterium optymalnych cięć, minimalnym drzewem rozpinającym T^* jest takie drzewo, do którego należą tylko takie krawędzie e , które spełniają warunek $e = \min \arg_{e_i} \{c_i : e_i \in \mathcal{Q}(T^*, e)\}$ — innymi słowy, do drzewa T^* należą tylko takie krawędzie, których koszt spośród wszystkich krawędzi dla cięcia, które same generują, jest najmniejszy. Przekłada się to na prosty algorytm zachłanny (tak jak w przypadku algorytmu Kruskala), który — rozpoczynając konstrukcję rozwiązania od arbitralnie wybranego wierzchołka v_s — sekwencko dołącza do rozwiązania krawędzie o jak najmniejszym koszcie, wybierając je ze zbioru definiowanego wraz z postępowaniem algorytmu ($\{e_{ij} : v_i \in S \wedge v_j \in \bar{S}\}$, gdzie S jest zbiorem wierzchołków, które są połączone krawędziami należącymi do konstruowanego rozwiązania, zaś $\bar{S} = \{v_i : v_i \in V \setminus \{v_i, v_j : e_{ij} \in T^*\}\}$). Do zapewnienia szybkiego dostępu do elementu e_{ij} o najmniejszym koszcie krawędzi oraz sprawnego budowania kolejnych zbiorów łuków, z których aktualnie możemy wybierać, posłużymy się strukturą **kopca** H . Zdefiniujmy dla niego następujące operacje, które następnie wykorzystamy do przedstawienia pseudokodu 2.

- **CREATE-HEAP** (G) — tworzy kopiec, którego elementami są trójki²: (v, k, p) , gdzie v jest wierzchołkiem, k wartością **klucza**³ elementu, p jest wskazaniem na poprzedni element w grafie⁴,
- **INSERT** (v, H) — wstawia element $(v, v.k, v.p)$ do kopca H ,
- **FIND-MIN** (H) — zwraca wierzchołek, którego wartość $v.k$ jest najmniejsza ze wszystkich wartości $v'.k$ dla elementów $v' \in H$,
- **DELETE-MIN** (H) — usuwa element z kopca H o najmniejszej wartości klucza,
- **DEC-KEY** (v, c, H) — zmniejsza wartość $v.k$ wierzchołka $v \in H$ tak, że nowa jego wartość wynosi c .

Tak zdefiniowany kopiec będziemy wykorzystywać do przechowywania wszystkich wierzchołków, których nasz algorytm jeszcze nie przeanalizował (zatem na samym początku algorytmu kopiec H zawiera wszystkie wierzchołki $v \in V$). Aby rozpocząć konstrukcję naszego rozwiązania, musimy wybrać wierzchołek początkowy — niech to będzie węzeł v_s (zatem v_s należy do zbioru wierzchołków już przez algorytm przetworzonych S). Teraz, zgodnie z kryterium optymalnych cięć, musimy wybrać taką krawędź, która będzie miała najmniejszy koszt ze wszystkich krawędzi definiowanych przez to cięcie. Niech wierzchołki, które należą do H , definiuje zbiór \bar{S} . Nasz zbiór krawędzi, określonych przez pierwsze cięcie, jest zatem zdefiniowany jako $\{e_{sj} : v_j \in \bar{S}\}$ ($v_s \in S$ i $v_j \in \bar{S}$). Wybrany spośród nich krawędź o najmniejszym koszcie (niech będzie to łuk e_{ss_1})⁵, aktualizujemy dane wierzchołka v_{s_1} ($v_{s_1}.p = v_s$, $v_{s_1}.k = \min \{v_{s_1}.k, c_{ss_1}\}$), przenosimy go do zbioru S (usuwając jednocześnie z kopca \bar{S}) i przechodzimy do wyznaczania kolejnych optymalnych cięć (dla większego zbioru S i mniejszego \bar{S} , które wyznaczają nowy zbiór krawędzi dla następnego cięcia). Opisane kroki kontynuujemy aż do osiągnięcia w konstruowanym drzewie rozpinającym wymaganej liczby krawędzi — $|V| - 1$. Powyższy opis stanowi główną część algorytmu Prima (patrz linie 12–20 pseudokodu 2).

Analizując schemat tak przedstawionego algorytmu, możemy zauważyc, że wstawianie wszystkich wierzchołków do kopca H nie jest potrzebne — wiedząc jaki wybrałyśmy wierzchołek początkowy v_s , jesteśmy w stanie ręcznie obliczyć wartości atrybutów wszystkich wierzchołków, do których krawędzie wychodzące z v_s bezpośrednio prowadzą (czyli dla wszystkich $v_s \sim v_j$ $v_j.p = v_s \wedge v_j.k = c_{sj}$). Oszczędzamy w ten sposób czas potrzebny na inicjalizację atrybutów wierzchołków, których wartości i tak ulegną zmianie w pierwszej iteracji, przeprowadzonej przez omawiany algorytm (gdyż na początku dla wszystkich wierzchołków v —

² Do elementów kopca będziemy się odwoływać tylko po wierzchołku v , zakładając, że pozostałe atrybuty są z nim związane.

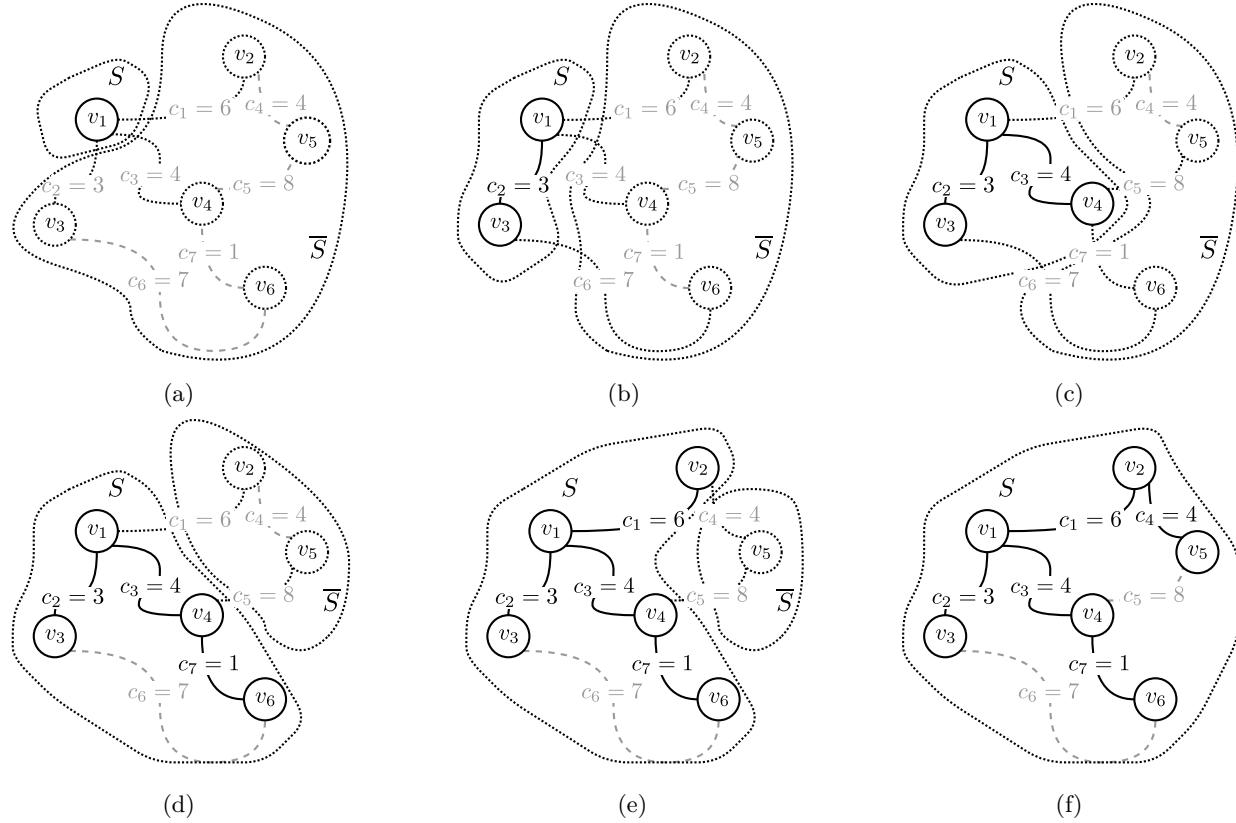
³ Kluczem w kopcu nazywamy parametr elementu należącego do kopca, który determinuje w nim jego pozycję. W tym wypadku jesteśmy zainteresowani stworzeniem kopca, którego pierwszymi elementami zawsze są wierzchołki, których wartość klucza jest najmniejsza.

⁴ Dla przykładu: gdy z danego wierzchołka v_i decydujemy się iść krawędzią $v_i \sim v_j$, poprzednikiem węzła v_j staje się wierzchołek v_i .

⁵ Zauważmy, że wartości kluczowe wierzchołków w grafie zawsze odpowiadają wagomiarom krawędzi, która bezpośrednio do niego prowadzi, toteż wybór odpowiedniej krawędzi sprawdza się nie do porównywania wszystkich ich wag, lecz do prostego wybrania pierwszego elementu z kopca (na jego szczytce znajduje się wierzchołek v o najmniejszej wartości $v.k$).



poza początkowym $v_s — v.k = \infty$ a $v.p$ są niezdefiniowane). Innymi słowy — poprzez taką obserwację potrafimy „ręcznie” zapisać informacje (linie 2–7), które byśmy uzyskali po pierwszym wykonaniu się głównej pętli algorytmu (linie 12–20).



Rysunek 2.5: Przebieg algorytmu Prima dla gafu $G = (V, E)$, gdzie $|V| = 6$, $|E| = 7$, o wierzchołku początkowym v_1 . (a) Wybrano wierzchołek początkowy v_1 . Pozostałe wierzchołki zostały umieszczone w kopcu H , definiującym zbiór \bar{S} ($v_2.p = v_4.p = v_3.p = v_1$ oraz $v_2.k = c_1$, $v_4.k = c_3$, $v_3.k = c_2$ — dla pozostałych wierzchołków v : $v.k = \infty$). (b) Na podstawie kolejności występowania wierzchołków w kopcu (wedle ich klucza — $v_3.k < v_4.k < v_2.k < \dots$) wybrano — spośród krawędzi zdefiniowanych przez $\{e_{ij} : v_i \in S \wedge v_j \in \bar{S}\}$ (krawędzie należące do aktualnego cięcia są zaznaczone czarnymi wykropkowanymi liniami) — łuk o najmniejszej wadze: e_2 (wierzchołek o najmniejszym kluczu, do którego prowadzi dana krawędź z wierzchołka v_1 — $e_2 \equiv v_1 \rightsquigarrow v_3$). Element v_3 został usunięty z kopca i dodany do zbioru S . Teraz zdefiniowany zbiór krawędzi należących do następnego cięcia: $\{e_1, e_3, e_6\}$. Jednocześnie wszystkie elementy w kopcu, odpowiadające wierzchołkom, do których można dojść z wybranego wierzchołka ($\{v_j : e_{3j} \in E\}$), zostały uaktualnione ($v_6.k = 7$, $v_6.p = v_3$). (c) Na podstawie wierzchołka pobranego ze szczytu kopca H ($v_4.k < v_2.k < v_6.k \dots$) poszerzono zbiór S o wierzchołek v_4 , zaś do konstruowanego minimalnego drzewa rozpinającego dodano krawędź $v_1 \rightsquigarrow v_4$ ($v_4.p = v_1$). Pozostałe elementy w kopcu zostały uaktualnione: $H = \{(v_6, 1, v_4), (v_2, 6, v_1), (v_5, 8, v_4)\}$ (po-grubioną czcionką zaznaczono nowe elementy). Klucz oraz poprzednik wierzchołka v_6 , który znajdował się już w kopcu, uległy zmianie, gdyż podczas analizy węzła v_4 okazało się, że koszt krawędzi prowadzącej bezpośrednio do v_6 jest dużo niższy niż dotychczasowy ($c_{36} = 7 > 1 = c_{46}$). (d) Kolejnym elementem pobranym ze stosu jest wierzchołek v_6 , którego poprzednik uległ zmianie w poprzednim kroku. Wszystkie wierzchołki bezpośrednio połączone z wybranym węzłem nie należą już do \bar{S} (nie ma ich w kopcu), tak więc wierzchołek zostaje usunięty z kopca bez żadnych, powodowanych przez siebie, zmian. W kopcu pozostały dwa elementy: v_2 (o wartości klucza $v_2.k = 6$) oraz v_5 ($v_5.k = 8$). (e) Wybrano kolejny najmniejszy (w sensie wartości klucza) element z kopca. Do konstruowanego rozwiązania dodano krawędź $v_2.p \rightsquigarrow v_2$ i uaktualniono wartość atrybutów elementów kopca, do których istnieje bezpośrednie połączenie z v_2 — $v_5.k = 4$, $v_5.p = v_2$. (f) Po pobraniu ostatniego elementu z kopca, algorytm kończy działanie.

Widzimy, że czas pracy danego algorytmu jest silnie zależny od sposobu implementacji użytej do przechowywania danych struktury — w naszym przypadku kopca, gdzie zależnie od jego implementacji możemy uzyskać czas działania pomiędzy $O(m \cdot \log(n))$ (dla zwykłego kopca binarnego), $O(m + n \cdot \log(n))$ (dla kopca Fibonacciego) nawet do $O(m \cdot \log(\log(C)))$ (struktura Johnsona [12]) [1, strona 525].

Pseudokod 2: PRIME-MST (G, v_1)

Wejście: $G = (V, E)$ — graf wejściowy,

v_1 — węzeł początkowy, od którego rozpocznie się konstrukcja rozwiązania.

Wyjście: T^* — minimalne drzewo rozpinające.

```
1 begin
2   foreach  $v \in V \setminus (v_1 \cup \{v' : v_1 \rightsquigarrow v'\})$  do
3      $v.k \leftarrow \infty$ 
4    $v_1.k \leftarrow 0$ 
5   foreach  $v_i : v_1 \rightsquigarrow v_i$  do
6      $v_i.k \leftarrow c_{1i}$ 
7      $v_i.p \leftarrow v_1$ 
8    $H \leftarrow \text{CREATE-HEAP}(G)$ 
9   foreach  $v \in V$  do
10     $\text{INSERT}(v, H)$                                 // Dodaj do kopca wszystkie wierzchołki w grafie.
11    $T^* \leftarrow \emptyset$ 
12   while  $|T^*| < |V| - 1$  do
13      $v_i \leftarrow \text{FIND-MIN}(H)$                   // Znajdź i wyciągnij węzeł  $v_i$  o najmniejszej wartości klucza  $v_i.k$ .
14      $\text{DELETE-MIN}(H)$                           // Usuń z kopca wyciągnięty przed chwilą wierzchołek  $v_i$ .
15      $T^* \leftarrow T^* \cup (v_i.p \rightsquigarrow v_i)$ 
16     foreach  $j : v_i \rightsquigarrow v_j \wedge v_j \in H$       // Dla każdego wierzchołka  $v_j \in \bar{S}$ , który jest sąsiadem węzła  $v_i$ .
17       do
18         if  $v_j.k > c_{ij}$  then
19            $v_j.c \leftarrow c_{ij}$ 
20            $v_j.p \leftarrow v_i$ 
21            $\text{DEC-KEY}(v_j, c_{ij}, H)$ 
22   return  $T^*$ 
```

2.4 Podsumowanie rozdziału

Problem minimalnego drzewa rozpinającego jest jednym z fundamentalnych problemów optymalizacyjnych, pojawiających się w wielu dziedzinach codziennego życia. Wszędzie tam, gdzie może nam zależeć na np. jak największym uproszczeniu badanej struktury grafowej, pozbyciu się redundantnych rozwiązań, bardzo prawdopodobnym jest, że napotkamy właśnie problem minimalnego drzewa rozpinającego. Uogólniając, zależnie od sposobu interpretacji elementów grafu możemy znaleźć wiele zastosowań dla wspomnianego problemu. Warto też zauważać, że nie musimy ograniczać się tylko do problemu znalezienia rozwiązania o najmniejszej sumie kosztów — jeżeli nasze koszty będą skonstruowane w ten sposób, że każdy z nich przybierać będzie postać logarytmu o ustalonej podstawie, ich suma tak naprawdę będzie wyrażać iloczyn rzeczywistych kosztów jakie chceliśmy w problemie zatrzymać. Tworzy nam to jeszcze więcej możliwości zastosowania dla tak postawionego problemu [1, 512–516].

Zapoznawszy się z podstawowymi pojęciami dotyczącymi problemów grafowych, naszą uwagę w następnych rozdziałach poświęcimy problemom bardziej złożonym, u podstaw których znajdziemy właśnie problem minimalnego drzewa rozpinającego (widzimy zatem, że jego zastosowania nie kończą się tylko na bezpośrednim zidentyfikowaniu problemu jako minimalnego drzewa rozpinającego, lecz jest on również podstawą do



rozwiązywania wielu innych, bardziej złożonych problemów). W tym zaś przedstawiliśmy podstawowe narzędzia pozwalające nam na uporanie się z nim, przedstawiliśmy warunki optymalności takiej konstrukcji oraz przytoczyliśmy liczne definicje, z których będziemy korzystać we wszystkich następnych rozdziałach.

Odporna optymalizacja

W poprzednim rozdziale zapoznaliśmy się z podstawowymi informacjami na temat problemu odnajdywania minimalnych drzew rozpinających w grafach oraz poznaliśmy sposoby radzenia sobie z nim. Do tej pory jednak nie poruszyliśmy najistotniejszego dla nas tematu: **optymalizacji odpornej** na czynniki zewnętrzne. Co przez to należy rozumieć? Do tej pory poznaliśmy algorytmy, które potrafią rozwiązywać problem w środowisku zamkniętym — idealnym, w którym nic nie ma prawa się zmieniać przez cały czas pracy algorytmu. W tym rozdziale omówimy modele problemów nie posiadających takiego ograniczenia — w modelach tych dane nie tylko będą mogły się zmieniać, ale będziemy także dopuszczać możliwość bycia nieświadomym tych zmian. W charakterze podsumowania, formalnie zapiszemy model jednego z wariantów przedstawionych problemów, wskazując tym samym ogólne podejście do jego rozwiązania.

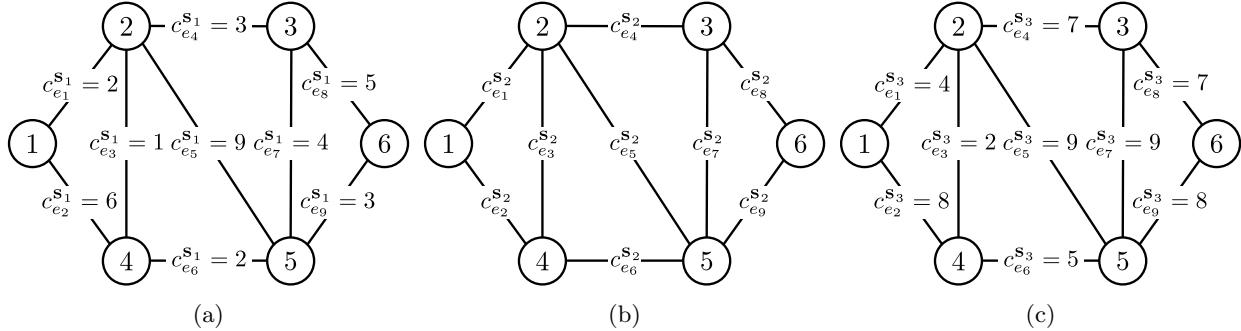
3.1 Scenariusze dyskretnie a ciągłe

W poprzednim rozdziale skupialiśmy swoją uwagę na problemie wyszukiwania minimalnego drzewa rozpinającego w grafie $G = (V, E)$, czyli takiego zbioru krawędzi T^* , który łączył ze sobą wszystkie wierzchołki w grafie, a jednocześnie którego suma kosztów krawędzi $e \in T^*$ była możliwie najmniejsza. Formalnie:

$$T_{G=(V,E)}^* = \left\{ e \in E : T_{G=(V,E)}^* \in \mathcal{T}_G \wedge (\forall T' \in \mathcal{T}_G) \sum_{e \in T'} c_e \geq c_{T_{G=(V,E)}^*} \right\}, \quad (3.1)$$

gdzie \mathcal{T}_G oznaczał zbiór wszystkich drzew możliwych do skonstruowania w grafie G , c_e — koszt krawędzi $e \in E$, zaś $\sum_{e \in T'} c_e = c_{T'}$ — sumę kosztów wszystkich krawędzi, która wynika ze **scenariusza**. Scenariuszem \mathbf{s} zatem nazwiemy zbiór definiujący koszt dla każdej krawędzi w grafie: $\mathbf{s} = \{c_{e_i}^s : i \in \{1, \dots, |E|\}\}$. Na rysunku 3.1 odpowiednio widzimy przykłady trzech różnych scenariuszy, zastosowanych do tego samego drzewa. Rzecz jasna, zależnie od kosztów jakie zostaną narzucone krawędziom przez dany scenariusz, zwarcane przez dotychczas poznane algorytmy rozwiązania będą się różnić. Zwrócić uwagę na rysunek 3.1b, gdzie celowo nie zostały ujawnione koszty scenariusza — jedyne co o nim wiemy to to, że żaden koszt krawędzi w tym scenariuszu nie jest mniejszy od kosztu odpowiadającej mu krawędzi w \mathbf{s}_1 , ani większy od kosztu tej samej krawędzi w scenariuszu \mathbf{s}_3 . Taki rodzaj scenariusza nazywamy **scenariuszem ciągłym**, zaś scenariusze, których poszczególne koszty krawędzi są najmniejsze (bądź największe) spośród wszystkich scenariuszy, nazywamy **scenariuszami granicznymi/krytycznymi** (odpowiednio **scenariuszem najlepszego/najgorszego przypadku**).

Przedstawione rodzaje scenariuszy można rozważyć dla wszystkich problemów odpornej optymalizacji dyskretnej jakie tutaj wymienimy. W przypadku problemu minimalizacyjnego, jakim jest omawiany przez nas problem znajdowania minimalnego drzewa rozpinającego, krytyczne scenariusze najlepszego i najgorszego przypadku będące oznaczać odpowiednio przez $\underline{\mathbf{s}}$ oraz $\bar{\mathbf{s}}$, gdzie $0 \leq \underline{\mathbf{s}} \leq \bar{\mathbf{s}}$. Dla problemów maksymalizacyjnych znaczenie przedstawionych symboli byłoby zgoła odwrotne: $\bar{\mathbf{s}}$ oznaczałby krytyczny scenariusz o największych współczynnikach (najlepszego przypadku), zaś $\underline{\mathbf{s}}$ — najgorszego, gdzie każdy jego współczynnik nie byłby większy od dowolnego z odpowiadających mu współczynników w pozostałych scenariuszach. **Scenariuszem dyskretnym** (ang. *discrete scenario*) będziemy natomiast nazywać każdy scenariusz, który ma zdefiniowane koszty dla każdej krawędzi. Takimi scenariuszami są na przykład scenariusze krytyczne (ang. *extreme scenarios*).



Rysunek 3.1: Scenariusze s_1 , s_2 , s_3 dla grafu nieskierowanego $G = (V, E)$, $V = \{1, 2, \dots, 6\}$, $E = \{e_i : i \in \{1, \dots, 9\}\}$. (a) Scenariusz najlepszego przypadku $s_1 = [2, 6, 1, 3, 9, 2, 4, 5, 3]$. Dla scenariusza zachodzi: $(\forall i \in \{1, \dots, 9\}) c_{e_i}^{s_1} \leq c_{e_i}^{s_2}$. Scenariusz o takich właściwościach będziemy też oznaczać jako \underline{s} . (b) Scenariusz ciągły s_2 . Koszty tego scenariusza zdefiniowane są następująco: $(\forall i \in \{1, \dots, 9\}) c_{e_i}^{\underline{s}} \leq c_{e_i}^{s_2} \leq c_{e_i}^{\bar{s}}$. (c) Krytyczny scenariusz najgorszego przypadku $s_3 = [4, 8, 2, 7, 9, 5, 9, 7, 8]$. Dla scenariusza zachodzi: $(\forall i \in \{1, \dots, 9\}) c_{e_i}^{s_2} \leq c_{e_i}^{s_3}$. Scenariusz o takich właściwościach będziemy też oznaczać jako \bar{s} .

3.2 Problemy minimaksowe

Problemami minimaksowymi [2, 428] nazywamy problemy, których celem jest odnalezienie najlepszego rozwiązania przy założeniu wystąpienia najgorszych z możliwych scenariuszy dla danych rozwiązań. Możemy rozróżnić tutaj problemy typu MIN-MAX (3.2) oraz MAX-MIN (3.3):

$$\min_{\mathbf{x} \in X} \max_{\mathbf{s} \in S} v(\mathbf{x}, \mathbf{s}) \quad (3.2)$$

oraz

$$\max_{\mathbf{x} \in X} \min_{\mathbf{s} \in S} v(\mathbf{x}, \mathbf{s}), \quad (3.3)$$

przy czym \mathbf{x} jest rozwiązaniem problemu, wybranym ze zbioru wszystkich możliwych rozwiązań X , $v(\mathbf{x}, \mathbf{s})$ reprezentuje koszt rozwiązania problemu dla scenariusza \mathbf{s} i wybranego zbioru krawędzi $\{e \in E : x_e = 1\}$ ¹, zaś S reprezentuje zbiór dostępnych scenariuszy. Oczywiście w przypadku gdy $|S| = 1$, problem MIN-MAX SPANNING TREE sprowadza się do klasycznego problemu minimalnego drzewa rozpinającego.

3.2.1 Przypadek ciągły

Niech $S = \{\mathbf{s} : (\forall i \in \{1, \dots, |E|\}) c_e^s \in [c_e^s, c_e^{\bar{s}}]\}$, gdzie scenariusze \underline{s} i \bar{s} są odpowiednio scenariuszami przedstawionymi na rysunkach 3.1a i 3.1c, zaś c_e^s oznacza koszt krawędzi e dla scenariusza \mathbf{s} . Innymi słowy: zbiór scenariuszy S składa się z takich schematów kosztów \mathbf{s} , że każdy koszt krawędzi c_e^s w tym scenariuszu może przyjmować wartość należącą do przedziału $[c_e^s, c_e^{\bar{s}}]$. Naszym celem niech będzie rozwiązanie problemu INTERVAL MIN-MAX SPANNING TREE.

Jak łatwo zauważać, w przypadku ciągłym zachodzi następująca prawidłowość:

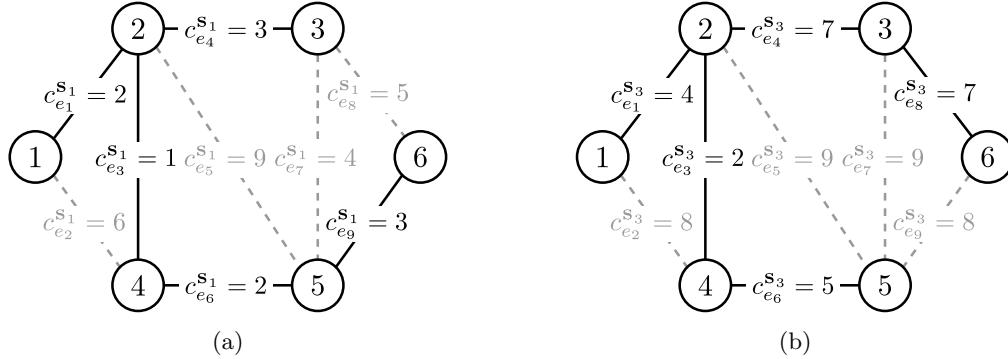
$$\min_{\mathbf{x} \in X} \left(\max_{\mathbf{s} \in S} v(\mathbf{x}, \mathbf{s}) \right) = \min_{\mathbf{x} \in X} \left(\max_{\mathbf{s} \in S} \sum_{e \in E} x_e \cdot c_e^s \right) = \min_{\mathbf{x} \in X} \left(\sum_{e \in E} x_e \cdot c_e^{\bar{s}} \right) = \min_{\mathbf{x} \in X} v(\mathbf{x}, \bar{s}), \quad (3.4)$$

jako że dla scenariusza \bar{s} i każdego definiowanego przez niego kosztu, dla dowolnego scenariusza $\mathbf{s} \in S$ zachodzi: $c_e^{\bar{s}} \geq c_e^s$. Podobne rozumowanie możemy przeprowadzić dla problemu MAX-MIN. Widzimy zatem, że w przypadku tak postawionego problemu, dla scenariuszy o rozmytych/niepewnych kosztach bardzo łatwo możemy znaleźć jego rozwiązanie, redukując problem do klasycznego problemu z jednym, krytycznym scenariuszem.

¹ Często dla własnej wygody będziemy zamiennie stosować oznaczenia: x_e oraz x_{e_i} (lub x_i) dla zmiennej decyzyjnej w wektorze \mathbf{x} , będącym wybranym rozwiązaniem problemu minimalnego drzewa rozpinającego, czy też c_e i c_{e_i} (lub c_i) dla oznaczeń kosztów krawędzi, chyba że zostanie napisane inaczej. Oznaczenia x_{e_i}/x_i oraz c_{e_i}/c_i będąmi stosować w przypadku, gdy będzie wymagane zachowanie kolejności oznaczeń np. dla krawędzi występujących w zbiorze kolejno po sobie ($x_{e_i}, x_{e_{i+1}}, \dots$).

3.2.2 Przypadek dyskretny

Powróćmy raz jeszcze do rysunku 3.1. Niech tym razem naszym zbiorem scenariuszy będzie $S = \{\mathbf{s}_1, \mathbf{s}_3\}$, gdzie \mathbf{s}_1 i \mathbf{s}_3 odnoszą się odpowiednio do 3.1a oraz 3.1c. **Dyskretnym** zbiorem scenariuszy nazwiemy taki zbiór, którego elementy jesteśmy w stanie ponumerować (będący zbiorem przeliczalnym) — tutaj nasz zbiór scenariuszy posiada dokładnie dwa elementy.



Rysunek 3.2: Dyskretnie scenariusze \mathbf{s}_1 , \mathbf{s}_3 dla grafu nieskierowanego $G = (V, E)$, $V = \{1, 2, \dots, 6\}$, $E = \{e_i : i \in \{1, \dots, 9\}\}$, z zaznaczonymi minimalnymi drzewami rozpinającymi dla każdego z nich.

Pomimo tak prosto zdefiniowanego zadania przekonamy się, że jego rozwiązanie wcale nie jest tak intuicyjne jak oczekujemy — w tabeli 3.1 przedstawiono kilka z 55 możliwych rozwiązań zadanego problemu². Na uwagę w niej zasługują rozwiązania \mathbf{x}_1 oraz \mathbf{x}_2 , które są rozwiązaniami optymalnymi dla scenariuszy \mathbf{s}_3 i \mathbf{s}_1 , rozpatrywanych osobno. Jeżeli spojrzymy na ostatnią kolumnę, zauważymy, że optymalne rozwiązanie dla pierwszego z wymienionych scenariuszy jest także rozwiązaniem optymalnym problemu minimaksowego — tak jak w przypadku scenariuszy ciągłych, tak i tutaj mogliśmy skorzystać z posiadanej wiedzy o kosztach definiowanych przez każdy ze scenariuszy; wiedząc, że dla scenariusza \mathbf{s}_3 żaden koszt nie jest mniejszy od odpowiadającej mu wagi w drugim scenariuszu, nie powinno być dla nas zaskoczeniem, że optymalność rozwiązania problemu dla scenariusza o wyższych kosztach pociąga za sobą optymalność rozwiązania problemu MIN-MAX. Należy jednak podkreślić, że taka sytuacja jest zazwyczaj mało prawdopodobna i bardzo łatwo możemy stworzyć scenariusze, dla których ta prawidłowość nie będzie zachodziła.

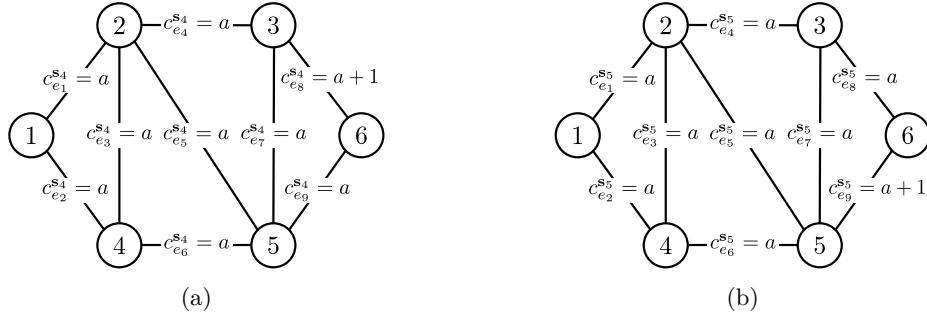
Tablica 3.1: Tabela przedstawiająca część z dopuszczalnych (ang. *feasible*) rozwiązań dla problemu minimalnego drzewa rozpinającego dla scenariuszy \mathbf{s}_1 oraz \mathbf{s}_3 (3.2a i 3.2b), koszty dla każdego z proponowanych rozwiązań dla podanych scenariuszy oraz wartość rozwiązania dla problemu minimaksowego. Wiersze w tabeli zostały posortowane w kolejności rosnących wartości rozwiązań.

X	Scenariusze											
	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	e_9	$v(\mathbf{x}, \mathbf{s}_1)$	$v(\mathbf{x}, \mathbf{s}_3)$	$\max\{v(\mathbf{x}, \mathbf{s}_1), v(\mathbf{x}, \mathbf{s}_3)\}$
\mathbf{x}_1	1	0	1	1	0	1	0	1	0	13	25	25
\mathbf{x}_2	1	0	1	1	0	1	0	0	1	11	26	26
\mathbf{x}_3	1	0	1	0	0	1	0	1	1	13	26	26
\mathbf{x}_4	1	0	1	0	0	1	1	1	0	14	27	27
...
\mathbf{x}_{53}	1	1	0	0	1	0	1	0	1	24	38	38
\mathbf{x}_{54}	0	1	0	0	1	1	1	1	0	26	38	38
\mathbf{x}_{55}	0	1	0	0	1	1	1	0	1	24	39	39

² Rozważane przykłady grafów posiadają 9 krawędzi łączących jego wierzchołki — spośród wszystkich możliwych 512 kombinacji zbiorów krawędzi, należących do rozwiązania (każda z nich może do niego niezależnie należeć bądź nie, co daje nam 2^9 możliwości wyboru rozwiązania), tylko 55 z nich zawiera krawędzie, które tworzą drzewo rozpinające dla danego grafu. Rozwiązania zostały wygenerowane poprzez modyfikację wszystkich kosztów w grafie (ustawieniu ich na 0) i pobranie wszystkich optymalnych rozwiązań dla tak zadanego problemu (w takim przypadku każde dopuszczalne rozwiązanie było także rozwiązaniem optymalnym — więcej o zagadniению optymalności opowiemy w rozdziale 4, poświęconym programowaniu liniowemu).

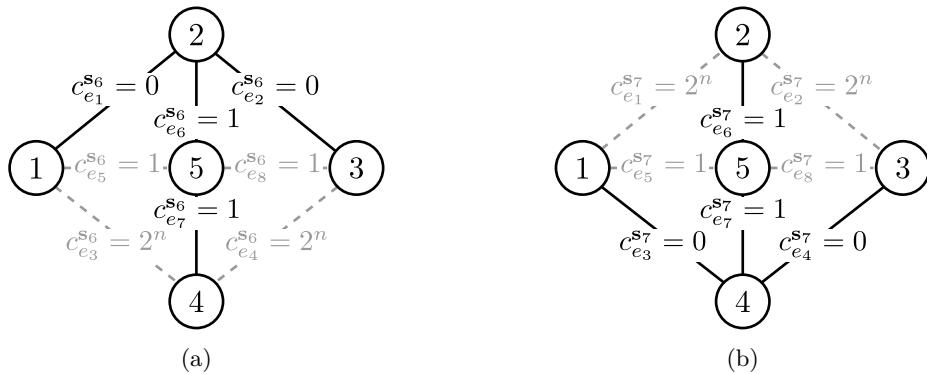


Niech scenariusze s_4 i s_5 będą takie, że dla każdego e_i , $i \in \{1, \dots, |E| - 2\}$, $c_e^{s_4} = c_e^{s_5}$ oraz niech $c_{e_{|E|-1}}^{s_4} = c_{e_{|E|-1}}^{s_5} + 1$ i $c_{e_{|E|}}^{s_4} = c_{e_{|E|}}^{s_5} - 1$, tak jak to pokazano na rysunku 3.3. Użyując rozwiązań postawiony przed nami problem postaci: $\min_{\mathbf{x} \in X} \max_{\mathbf{s} \in S} v(\mathbf{x}, \mathbf{s})$, szybko zauważymy, że w tym przypadku odnalezienie optymalnych rozwiązań dla każdego ze scenariuszy i próba wybrania spośród nich jednego, jako rozwiązania problemu MIN-MAX SPANNING TREE, nie jest dobrym pomysłem: dla scenariusza s_4 optymalnym wyborem jest każdy podzbiór zbioru krawędzi grafu tworzący drzewo rozpinające, który nie zawiera łuku e_8 o koszcie $a + 1$ (koszt takiego rozwiązania wynosi $a \cdot (|V| - 1)$), dla scenariusza s_5 uzyskamy podobne wyniki, tym razem wykluczając ze zbioru rozwiązań te podzbiory, do których należy krawędź e_9 . W obu przypadkach wartość optymalnego rozwiązania wynosi tyle samo, zaś prawidłowym rozwiązaniem problemu $\min_{\mathbf{x} \in X} \max_{\mathbf{s} \in S} v(\mathbf{x}, \mathbf{s})$ jest rozwiązanie dowolne — analizowany graf i jego koszty zostały tak dobrane, że wartość wyrażenia $\max_{\mathbf{s} \in S} v(\mathbf{x}, \mathbf{s}) = 5 \cdot a + 1$ dla każdego dopuszczalnego rozwiązania. Wystarczy zauważać, że jedyną istotną decyzją jaką musimy podjąć, jest wybór ostatniej krawędzi, zaś — niezależnie od dokonanego wyboru — maksymalny koszt wszystkich wybranych krawędzi nie ulegnie zmianie.



Rysunek 3.3: Dyskretnie scenariusze s_4 , s_5 dla grafu nieskierowanego $G = (V, E)$, $V = \{1, 2, \dots, 6\}$, $E = \{e_i : i \in \{1, \dots, 9\}\}$. Dla tak określonych scenariuszy dłużej już nie zachodzi własność: $\forall e \in E : c_e^{s_4} \leq c_e^{s_5}$.

Problem jest tym bardziej widoczny, im większe różnice będą zachodzić pomiędzy poszczególnymi scenariuszami. Za przykład niech posłuży nam ilustracja 3.4, gdzie pomimo występowania nadal jedynie dwóch scenariuszy, opieranie się na dotychczasowej intuicji policzenia optymalnych rozwiązań dla każdego z nich i wybraniu jednego, prowadzi do znacznie poważniejszych błędów, niż miało to miejsce w poprzednim przykładzie, gdzie pomimo zastosowania błędnej logiki, otrzymaliśmy poprawny rezultat. Wprowadźmy kolejne scenariusze: $s_6 = [0, 0, 2^n, 2^n, 1, 1, 1, 1]$ oraz $s_7 = [2^n, 2^n, 0, 0, 1, 1, 1, 1]$, gdzie n jest dowolne³.



Rysunek 3.4: Skrajny przykład problemu DISCRETE MIN-MAX SPANNING TREE, gdzie optymalne rozwiązania dla poszczególnych scenariuszy okazują się być bardzo złe (dla dużych n) w porównaniu z rozwiązaniem głównego problemu. Podobny przykład [2, 429–430] można skonstruować dla minimaksowego problemu najkrótszej ścieżki.

³ Chcemy aby funkcje zmiennej n przyjmowały wartość większą od jedynki, co w przypadku funkcji wykładniczej jest spełnione dla dowolnego $n \in \mathbb{N}^+$.

Optymalnymi rozwiązaniami dla poszczególnych scenariuszy są oczywiście: $T_{\mathbf{s}_6}^* = \{e_1, e_2, e_6, e_7\}$, o całkowitym koszcie równym 2, oraz $T_{\mathbf{s}_7}^* = \{e_3, e_4, e_6, e_7\}$, gdzie $v(\mathbf{s}_7, x_{\mathbf{s}_7}^*)$ ma tą samą wartość co poprzednie rozwiązanie⁴. Oczywiście po podstawieniu otrzymanych danych do wzoru: $\min_{\mathbf{x} \in X^*} \max_{\mathbf{s} \in S} v(\mathbf{x}, \mathbf{s})$, gdzie $X^* = \{\mathbf{x}_{\mathbf{s}_6}^*, \mathbf{x}_{\mathbf{s}_7}^*\}$ otrzymamy:

$$\min \left\{ \max_{\mathbf{s} \in S} v(\mathbf{x}_{\mathbf{s}_6}^*, \mathbf{s}), \max_{\mathbf{s} \in S} v(\mathbf{x}_{\mathbf{s}_7}^*, \mathbf{s}) \right\} = \min \left\{ \max \{v(\mathbf{x}_{\mathbf{s}_6}^*, \mathbf{s}_6), v(\mathbf{x}_{\mathbf{s}_6}^*, \mathbf{s}_7)\}, \max \{v(\mathbf{x}_{\mathbf{s}_7}^*, \mathbf{s}_6), v(\mathbf{x}_{\mathbf{s}_7}^*, \mathbf{s}_7)\} \right\}.$$

Latwo zauważać, że $v(\mathbf{x}_{\mathbf{s}_6}^*, \mathbf{s}_6) < v(\mathbf{x}_{\mathbf{s}_6}^*, \mathbf{s}_7)$ oraz $v(\mathbf{x}_{\mathbf{s}_7}^*, \mathbf{s}_7) < v(\mathbf{x}_{\mathbf{s}_7}^*, \mathbf{s}_6)$. Stąd:

$$\max \{v(\mathbf{x}_{\mathbf{s}_6}^*, \mathbf{s}_6), v(\mathbf{x}_{\mathbf{s}_6}^*, \mathbf{s}_7)\} = v(\mathbf{x}_{\mathbf{s}_6}^*, \mathbf{s}_7) = 2 \cdot 2^n + 2 = 2^{n+1} + 2, \quad (3.5)$$

$$\max \{v(\mathbf{x}_{\mathbf{s}_7}^*, \mathbf{s}_6), v(\mathbf{x}_{\mathbf{s}_7}^*, \mathbf{s}_7)\} = v(\mathbf{x}_{\mathbf{s}_7}^*, \mathbf{s}_6) = 2 \cdot 2^n + 2 = 2^{n+1} + 2, \quad (3.6)$$

a całe wyrażenie skracia się do $\min \{2^{n+1} + 2, 2^{n+1} + 2\} = 2^{n+1} + 2$, podczas gdy optymalną wartością dla problemu MIN-MAX SPANNING TREE dla tych scenariuszy wynosi $v(x^*) = v^* = 4$, gdzie $x^* = [0, 0, 0, 0, 1, 1, 1, 1]$ (zbiór $T^* = \{e_5, e_6, e_7, e_8\}$). Uzyskaliśmy zatem błąd rzędu wykładniczego.

3.2.3 Sposoby aproksymacji problemu

Aby nie musieć uciekać się do rozwiązywania problemu MIN-MAX ze zbiorem scenariuszy S prosto z definicji tj.

- dla każdego dopuszczalnego rozwiązania $\mathbf{x} \in X$ wygenerować zbiór $V_S^{\mathbf{x}} = \{v(\mathbf{x}, \mathbf{s}) : \mathbf{s} \in S\}$ (zawierający wartości rozwiązań dla ustalonego wektora \mathbf{x} dla wszystkich scenariuszy w S),
- z tak powstałego zbioru $\mathcal{V}_S^X = \{\max_{y \in V_S^{\mathbf{x}}} \{y\} : \mathbf{x} \in X\}$ wybrać rozwiązanie $\mathbf{x}^* \in X$, dla którego $v(\mathbf{x}^*, S) = \min_{z \in \mathcal{V}_S^X} \{z\}$,

możemy próbować aproksymować problem na podstawie jednego scenariusza, który jest składową wszystkich rozpatrywanych scenariuszy [18] [2, 430]. Przyjrzymy się dwóm podejściom do rozpatrywanego przez nas problemu, w obu przypadkach otrzymamy jego k -aproksymację, gdzie k będzie liczbą scenariuszy, jakie bierzemy pod uwagę. Choć będziemy skupiać się tylko na problemie MIN-MAX SPANNING TREE, przedstawione dowody da się z łatwością uogólnić na całą klasę problemów MIN-MAX.

Uśrednianie scenariuszy

Pierwszym, najprostszym do udowodnienia pomysłem jest potraktowanie problemu minimalizacyjnego \mathcal{P} ze zbiorem scenariuszy S ($|S| = k$) jako problemu z pojedynczym scenariuszem, zdefiniowanym w sposób podany w poniższym twierdzeniu.

Twierdzenie 3.2.1 [2, 430] Niech \mathcal{I} będzie instancją problemu DISCRETE MIN-MAX \mathcal{P} . Niech problem zawiera k scenariuszy w zbiorze S i niech każdy $\mathbf{s}_j \in S$ będzie zdefiniowany jako $\mathbf{s}_j = [c_1^{s_j}, \dots, c_m^{s_j}]$. Dodatkowo niech \mathcal{I}' będzie instancją problemu \mathcal{P} z jednym scenariuszem \mathbf{s}' , którego koszty są średnią kosztów scenariuszy $\mathbf{s} \in S$. Wtedy, jeżeli istnieje optymalne rozwiązanie \mathbf{x}' dla \mathcal{I}' , to $\max_{\mathbf{s} \in S} v(\mathbf{x}', \mathbf{s}) \leq k \cdot \text{opt}(\mathcal{I})$, gdzie $\text{opt}(\mathcal{I})$ oznacza optymalną wartość rozwiązania dla \mathcal{I} .

Dowód. [2, 430] Koszty scenariusza \mathbf{s}' dla \mathcal{I}' są zdefiniowane następująco:

$$c_i^{\mathbf{s}'} = \sum_{j=1}^k \frac{c_i^{s_j}}{k} \quad \forall i \in \{1, \dots, m\}. \quad (3.7)$$

⁴ Z oznaczeniem $v(\mathbf{s}, \mathbf{x})$ spotkaliśmy się już wcześniej, przy omawianiu tabeli 3.1, gdzie wyrażenie to oznaczało całkowity koszt dopuszczalnego rozwiązania problemu optymalizacyjnego dla zadanego scenariusza \mathbf{s} oraz wybranego zbioru krawędzi, reprezentowanego przez wektor \mathbf{x} . Analogicznie poprzez $v(\mathbf{s}, x_{\mathbf{s}}^*)$ oznaczać będziemy koszt **optymalnego** rozwiązania dla scenariusza \mathbf{s} , gdzie $x_{\mathbf{s}}^*$ to binarny wektor reprezentujący optymalne rozwiązanie dla danego scenariusza ($T^* = \{e : x_e = 1\}$). Często będziemy skracać ten zapis do $v_{\mathbf{s}}$.



Rozpisując wyrażenie $v(\mathbf{x}, \mathbf{s}') = \sum_{e \in E} x_e \cdot c_e^{\mathbf{s}'}$ otrzymujemy: $\sum_{e \in E} x_e \cdot \left(\sum_{j=1}^k \frac{c_e^{\mathbf{s}_j}}{k} \right) = \frac{1}{k} \cdot \sum_{j=1}^k \sum_{e \in E} x_e \cdot c_e^{\mathbf{s}_j} = \frac{1}{k} \cdot \sum_{e \in E} x_e \cdot c_e^{\mathbf{s}}$. Pokażemy, że optymalne rozwiązanie \mathbf{x}' dla scenariusza \mathbf{s}' nie jest gorsze od optymalnego rozwiązania oryginalnego problemu (w przeciwnym wypadku rozwiązanie takiego scenariusza byłoby bezcelowe, gdyż nawet jego optymalne rozwiązanie było by gorsze od pożądanej). Mamy zatem:

$$L = v(\mathbf{x}', \mathbf{s}') = \min_{\mathbf{x} \in X} v(\mathbf{x}, \mathbf{s}') = \min_{\mathbf{x} \in X} \frac{1}{k} \cdot \sum_{\mathbf{s} \in S} v(\mathbf{x}, \mathbf{s}) \leq \min_{\mathbf{x} \in X} \frac{1}{k} \cdot k \cdot \max_{\mathbf{s} \in S} v(\mathbf{x}, \mathbf{s}) = \min_{\mathbf{x} \in X} \max_{\mathbf{s} \in S} v(\mathbf{x}, \mathbf{s}) = \text{opt}(\mathcal{I}). \quad (3.8)$$

W czasie obliczeń skorzystaliśmy z własności $\sum_{\mathbf{s} \in S} v(\mathbf{x}, \mathbf{s}) \leq k \cdot \max_{\mathbf{s} \in S} v(\mathbf{x}, \mathbf{s})$ (liczba scenariuszy $|S| = k$). Wartość L nazywamy **dolnym ograniczeniem**. **Górnym ograniczeniem** na wartość rozwiązania jest oczywiście $\max_{\mathbf{s} \in S} v(\mathbf{x}', \mathbf{s}) = U$. Próbując ograniczać otrzymaną wartość U otrzymamy:

$$\text{opt}(\mathcal{I}) = \min_{\mathbf{x} \in X} \max_{\mathbf{s} \in S} v(\mathbf{x}, \mathbf{s}) \leq \max_{\mathbf{s} \in S} v(\mathbf{x}', \mathbf{s}) \leq \sum_{\mathbf{s} \in S} v(\mathbf{x}', \mathbf{s}) = k \cdot \frac{1}{k} \cdot \sum_{\mathbf{s} \in S} v(\mathbf{x}', \mathbf{s}) = k \cdot L \leq k \cdot \text{opt}(\mathcal{I}), \quad (3.9)$$

gdzie w dwóch ostatnich równościach skorzystaliśmy z tego, że $L = \min_{\mathbf{x} \in X} \frac{1}{k} \cdot \sum_{\mathbf{s} \in S} v(\mathbf{x}, \mathbf{s}) = \frac{1}{k} \cdot \sum_{\mathbf{s} \in S} v(\mathbf{x}', \mathbf{s})$ oraz, że wcześniej udowodniliśmy nierówność $L \leq \text{opt}(\mathcal{I})$. Pokazaliśmy zatem, że $\max_{\mathbf{s} \in S} v(\mathbf{x}', \mathbf{s}) \leq k \cdot \text{opt}(\mathcal{I})$, co oznacza, że wybierając optymalne rozwiązanie \mathbf{x}' dla scenariusza \mathbf{s}' , w najgorszym możliwym przypadku (wystąpienia takiego scenariusza, że wartość rozwiązania dla wybranego \mathbf{x}' będzie najbardziej oddalona od minimum) otrzymane rozwiązanie nie będzie gorsze niż k -krotność wartości optymalnej. ♦

Scenariusz najgorszych wartości

Scenariuszem najgorszych wartości będziemy nazywać taki scenariusz, którego każdy koszt, który definiuje, jest największym z możliwych, odpowiadających mu, kosztów scenariuszy, z których ten scenariusz powstaje. Innymi słowy, powracając do rysunku 3.4, jeżeli mamy dwa scenariusze: $\mathbf{s}_6 = [0, 0, 2^n, 2^n, 1, 1, 1, 1]$ oraz $\mathbf{s}_7 = [2^n, 2^n, 0, 0, 1, 1, 1, 1]$, to scenariuszem najgorszych wartości będzie scenariusz $\mathbf{s}' = [2^n, 2^n, 2^n, 2^n, 1, 1, 1, 1]$ — tak samo jak w przypadku ciągłym, gdy najgorszy z możliwych scenariuszy był definiowany za pomocą \bar{s} , tak wszystkie koszty scenariusza \mathbf{s}' są nie mniejsze niż odpowiadające im wartości w scenariuszach pozostałych. Co więcej, jeżeli wykorzystalibyśmy tak stworzony scenariusz \mathbf{s}' do rozwiązania problemu przedstawionego na ilustracji 3.4, otrzymalibyśmy rozwiązanie optymalne, podobnie jak miało to miejsce w przypadku ciągłym. Pokażemy jednak, że — pomimo obiecującego wyniku — taka próba podejścia do problemów MIN-MAX nie zapewnia nam optymalnego rozwiązania, a co najwyżej jego k -aproksymację, podobnie jak poprzednio zaprezentowana metoda.

Twierdzenie 3.2.2 [2, 430] Niech \mathcal{I} będzie instancją problemu DISCRETE MIN-MAX \mathcal{P} . Niech problem zawiera k scenariuszy w zbiorze S i niech każdy $\mathbf{s} \in S$ będzie zdefiniowany jako $\mathbf{s} = [c_1^{\mathbf{s}}, \dots, c_m^{\mathbf{s}}]$. Dodatkowo niech \mathcal{I}' będzie instancją problemu \mathcal{P} z pojedynczym scenariuszem najgorszych wartości \mathbf{s}' . Wtedy, jeżeli istnieje optymalne rozwiązanie \mathbf{x}' dla \mathcal{I}' , to $\max_{\mathbf{s} \in S} v(\mathbf{x}', \mathbf{s}) \leq k \cdot \text{opt}(\mathcal{I})$, gdzie $\text{opt}(\mathcal{I})$ oznacza optymalną wartość rozwiązania dla \mathcal{I} .

Dowód. [2, 430] Koszty scenariusza \mathbf{s}' dla \mathcal{I}' są zdefiniowane następująco:

$$c_i^{\mathbf{s}'} = \max_{\mathbf{s} \in S} c_i^{\mathbf{s}} \quad \forall i \in \{1, \dots, m\}. \quad (3.10)$$

Pokażemy ciąg przekształceń, który da nam górne oszacowanie na wartość wyrażenia $\max_{\mathbf{s} \in S} v(\mathbf{x}', \mathbf{s})$. Niech \mathbf{x}^* będzie optymalnym rozwiązaniem oryginalnego problemu \mathcal{I} :

$$\max_{\mathbf{s} \in S} v(\mathbf{x}', \mathbf{s}) \leq \sum_{i=1}^m c_i^{\mathbf{s}'} \cdot x'_i \leq \sum_{i=1}^m c_i^{\mathbf{s}'} \cdot x_i^* \leq \sum_{i=1}^m \sum_{\mathbf{s} \in S} c_i^{\mathbf{s}} \cdot x_i^* = \sum_{\mathbf{s} \in S} \sum_{i=1}^m c_i^{\mathbf{s}} \cdot x_i^* \leq k \cdot \max_{\mathbf{s} \in S} \sum_{i=1}^m c_i^{\mathbf{s}} \cdot x_i^* = k \cdot \text{opt}(\mathcal{I}). \quad (3.11)$$

Chcąc dokładniej przyjrzeć się przeprowadzonemu przez nas rozumowaniu:



- $\max_{\mathbf{s} \in S} v(\mathbf{x}', \mathbf{s}) = \sum_{i=1}^m c_i^{\mathbf{s}} \cdot x'_i \leq \sum_{i=1}^m c_i^{\mathbf{s}'} \cdot x'_i$ otrzymujemy prosto z definicji kosztów scenariusza s' — dla każdego $i \in \{1, \dots, m\}$ zachodzi $c_i^{\mathbf{s}'} \leq \max_{\mathbf{s} \in S} c_i^{\mathbf{s}} = c_i^{\mathbf{s}'}$, zaś reszta wyrażenia nie ulega zmianie,
- $\sum_{i=1}^m c_i^{\mathbf{s}'} \cdot x'_i \leq \sum_{i=1}^m c_i^{\mathbf{s}'} \cdot x_i^*$ — rozwiązanie opisywane przez wektor \mathbf{x}' jest optymalne dla scenariusza z kosztami $c_i^{\mathbf{s}'}$ zatem każde inne rozwiązanie jest nie lepsze od niego (\mathbf{x}^* jest optymalne dla kosztów oryginalnych, nie dla tych, które występują w nierówności),
- $\sum_{i=1}^m c_i^{\mathbf{s}'} \cdot x_i^* \leq \sum_{i=1}^m \sum_{\mathbf{s} \in S} c_i^{\mathbf{s}} \cdot x_i^*$ w oczywisty sposób wynika z faktu, że dla każdego $i \in \{1, \dots, m\}$ zachodzi $c_i^{\mathbf{s}'} = \max_{\mathbf{s} \in S} c_i^{\mathbf{s}} \leq \sum_{\mathbf{s} \in S} c_i^{\mathbf{s}}$,
- $\max_{\mathbf{s} \in S} \sum_{i=1}^m c_i^{\mathbf{s}} \cdot x_i^*$ jest równe wartości rozwiązania dla najgorszego możliwego przypadku, zakładając, że podstawiane do wzoru rozwiązanie \mathbf{x}^* jest optymalne — $opt(\mathcal{I})$.

Pokazaliśmy zatem, tak samo jak w przypadku poprzedniego twierdzenia, że takie podejście do problemu MIN-MAX zapewnia nam jedynie k -aproksymację rozwiązania. ♦

3.3 Problemy minimaksowe ze względną funkcją optymalności

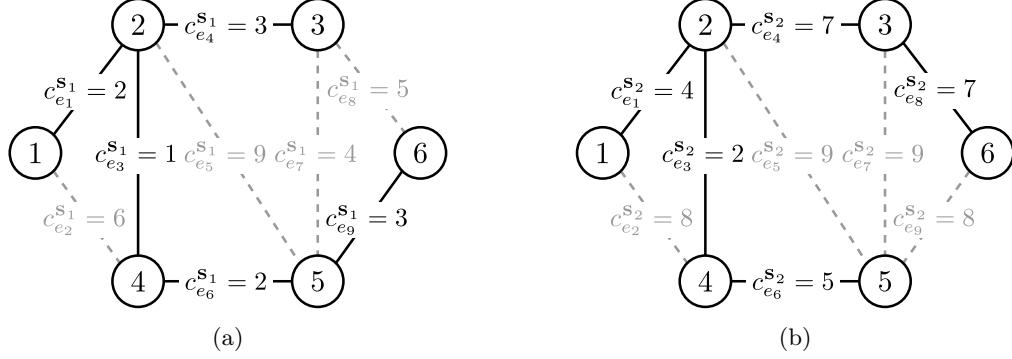
W poprzednim dziale rozważaliśmy problem, który w naturalnym języku wyraża się pytaniem „Jakie wybrać rozwiązanie aby w najgorszym przypadku było ono najlepsze z możliwych?”. Skupiało się zatem ono tylko na niwelowaniu strat, których można było się spodziewać, nie zaś na faktycznym szukaniu rozwiązania, które byłoby możliwe bliskie optymalnego. W tym dziale zmienimy nieco nasze podejście do postawionego problemu i zadamy sobie pytanie: „Jakie rozwiązanie wybrać, aby w najgorszym przypadku było ono tak blisko rozwiązania optymalnego, jak to tylko możliwe?”, Aby to osiągnąć, wprowadzimy pojęcie **żalu** (ang. *regret*). Niech $v_{\mathbf{s}}^*$ oznacza wartość optymalnego rozwiązania dla scenariusza \mathbf{s} . Wektorem, zapewniającym to rozwiązanie optymalne, jest wektor $x_{\mathbf{s}}^*$. Równocześnie oznacza to, że dla każdego innego wektora $\mathbf{x} \neq \mathbf{x}_{\mathbf{s}}^*$ $v(\mathbf{x}, \mathbf{s}) \geq v(x_{\mathbf{s}}^*, \mathbf{s})$. Różnicę tych dwóch wartości nazywamy żalem i reprezentuje on strategię jaką ponieśliśmy w konsekwencji wyboru innego rozwiązania niż optymalne dla danego scenariusza. Istotą problemu MIN-MAX REGRET [14, 595] jest wybranie takiego rozwiązania \mathbf{x} , aby zminimalizować tę różnicę dla wszystkich scenariuszy, w szczególności dla tego najgorszego, bo to od niego zależy wynik poniższego wyrażenia:

$$\min_{\mathbf{x} \in X} \max_{\mathbf{s} \in S} (v(\mathbf{x}, \mathbf{s}) - v(x_{\mathbf{s}}^*, \mathbf{s})). \quad (3.12)$$

Możemy także rozpatrywać przypadek MAX-MIN REGRET, którego definicja nieco różni się od tej podanej wyżej i nie jest analogiczna do problemu MAX-MIN: $\min_{\mathbf{x} \in X} \max_{\mathbf{s} \in S} (v(x_{\mathbf{s}}^*, \mathbf{s}) - v(\mathbf{x}, \mathbf{s}))$. Tak samo jak w przypadku poprzednich problemów, tutaj też możemy wyróżnić podział na scenariusze ciągłe oraz dyskretnie.

3.3.1 Przypadek dyskretny

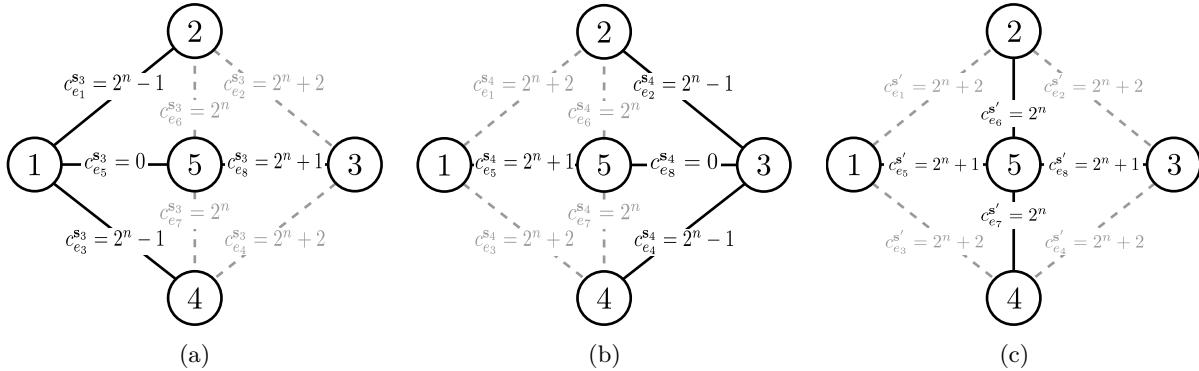
Tym razem zaczniemy od omówienia przypadku dyskretnego, gdyż — jak się będziemy mogli przekonać — wiele z tego, co do tej pory powiedzieliśmy o problemach MIN-MAX, powtórzy się dla problemów MIN-MAX REGRET. Podobnie jak to miało miejsce w przypadku poprzedniej klasy problemów, zaczniemy od omówienia prostego przykładu, który przedstawia rysunek 3.5, a na którym zaznaczono optymalne rozwiązania dla problemu MIN-MAX REGRET MINIMUM SPANNING TREE dla dwóch scenariuszy: \mathbf{s}_1 oraz \mathbf{s}_2 , natomiast w tabeli 3.2 — 7 przykładowych wyników, jakie daje zastosowanie różnych drzew rozpinających. Już dwa pierwsze rozwiązania: \mathbf{x}_1 oraz \mathbf{x}_2 wskazują na to, że kierowanie się przy wyborze rozwiązania dla tego problemu tylko kryterium optymalności dla pojedynczego scenariusza jest błędne i nie należy go stosować; obydwa rozwiązania są optymalne odpowiednio dla scenariuszy \mathbf{s}_1 oraz \mathbf{s}_2 , lecz nie dają tego samego rezultatu przy rozpatrywaniu właściwego problemu.



Rysunek 3.5: Dyskretne scenariusze $\mathbf{s}_1, \mathbf{s}_2$ dla grafu nieskierowanego $G = (V, E)$, $V = \{1, 2, \dots, 6\}$, $E = \{e_i : i \in \{1, \dots, 9\}\}$. (a) Optymalnym rozwiązaniem dla scenariusza \mathbf{s}_1 jest zbiór wierzchołków $T_{\mathbf{s}_1}^* = \{e_1, e_3, e_4, e_6, e_9\}$, któremu odpowiada wektor $\mathbf{x}_{\mathbf{s}_1}^* = [1, 0, 1, 1, 0, 1, 0, 0, 1]$. Wartość tego rozwiązania wynosi $v_{\mathbf{s}_1}^* = 11$. (b) Optymalnym rozwiązaniem dla scenariusza \mathbf{s}_2 jest zbiór wierzchołków $T_{\mathbf{s}_2}^* = \{e_1, e_3, e_4, e_6, e_8\}$, któremu odpowiada wektor $\mathbf{x}_{\mathbf{s}_2}^* = [1, 0, 1, 1, 0, 1, 0, 1, 0]$. Wartość tego rozwiązania wynosi $v_{\mathbf{s}_2}^* = 25$.

Tablica 3.2: Tabela przedstawiająca część z osiągalnych rozwiązań dla problemu minimalnego drzewa rozpinającego dla scenariuszy \mathbf{s}_1 oraz \mathbf{s}_2 (3.5a i 3.5b), koszty dla każdego z proponowanych rozwiązań dla podanych scenariuszy, poniesione straty względem optymalnych rozwiązań oraz wartość rozwiązania dla problemu MIN-MAX REGRET. Wiersze w tabeli zostały posortowane w kolejności rosnących wartości rozwiązań.

X	Scenariusze													
	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	e_9	$v(\mathbf{x}, \mathbf{s}_1)$	$r_{\mathbf{s}_1}$	$v(\mathbf{x}, \mathbf{s}_2)$	$r_{\mathbf{s}_2}$	$\max \{r_{\mathbf{s}_1}, r_{\mathbf{s}_2}\}$
\mathbf{x}_1	1	0	1	1	0	1	0	0	1	11	0	26	1	1
\mathbf{x}_2	1	0	1	1	0	1	0	1	0	13	2	25	0	2
\mathbf{x}_3	1	0	1	0	0	1	0	1	1	13	2	26	1	2
\mathbf{x}_4	1	0	1	0	0	1	1	0	1	12	1	28	3	3
...
\mathbf{x}_{53}	0	1	1	0	1	0	1	1	0	25	14	35	10	14
\mathbf{x}_{54}	0	1	0	0	1	1	1	1	0	26	15	38	13	15
\mathbf{x}_{55}	1	1	0	0	1	0	1	1	0	26	15	37	12	15



Rysunek 3.6: Dyskretne scenariusze $\mathbf{s}_3, \mathbf{s}_4$ i \mathbf{s}' dla grafu nieskierowanego $G = (V, E)$, $V = \{1, 2, \dots, 5\}$, $E = \{e_i : i \in \{1, \dots, 8\}\}$, gdzie \mathbf{s}' jest sztucznie wygenerowanym scenariuszem najgorszych kosztów. (a) Optymalne rozwiązanie $T_{\mathbf{s}_3}^* = \{e_1, e_3, e_5, e_8\}$ o całkowitym koszcie: $3 \cdot 2^n - 1$. (b) Optymalne rozwiązanie $T_{\mathbf{s}_4}^* = \{e_2, e_4, e_5, e_8\}$ o identycznym koszcie, co $v_{\mathbf{s}_3}^*$. (c) Optymalne rozwiązanie dla scenariusza \mathbf{s}' o kosztach $c_i^{s'} = \max_{\mathbf{s} \in S} c_i^{\mathbf{s}}$. Koszt całkowity rozwiązania optymalnego wynosi $v(\mathbf{x}_{\mathbf{s}'}, \mathbf{s}') = 4 \cdot 2^n + 2$.

Aproksymacja rozwiązania

Chcąc otrzymać aproksymację dla problemu MIN-MAX REGRET, możemy odwołać się do udowodnionego twierdzenia 3.2.1 — jego dowód przebiega analogicznie dla tej klasy problemów [2, 430] i zostanie tu pominięty. Dużo ciekawszy okazuje się natomiast przypadek, w którym tworzyliśmy dodatkowy scenariusz s' , którego koszty definiowaliśmy jako $c'_i = \max_{s \in S} c_i^s \forall i \in \{1, \dots, m\}$. Pokażemy, że nawet dla dwóch scenariuszy (gdzie spodziewalibyśmy się wyniku co najwyżej 2 razy gorszego niż optymalny) otrzymany rezultat jest dalece gorszy od spodziewanego. Stwórzmy dwa scenariusze: $s_3 = [2^n - 1, 2^n + 2, 2^n - 1, 2^n + 2, 0, 2^n, 2^n, 2^n + 1]$ oraz $s_4 = [2^n + 2, 2^n - 1, 2^n + 2, 2^n - 1, 2^n + 1, 2^n, 2^n, 0]$, tak jak na rysunku 3.6. Będziemy chcieli pokazać, że optymalna wartość rozwiązania problemu MIN-MAX REGRET MINIMUM SPANNING TREE dla taki konfiguracji scenariuszy wynosi 0, toteż każda inna wartość (różna od zera) jaką otrzymamy przy rozwiązywaniu problemu z wykorzystaniem scenariusza s' , będzie błędna (zgodnie z twierdzeniem 3.2.2 $\max_{s \in S} v(\mathbf{x}', s) \leq k \cdot \text{opt}(\mathcal{I})$, zatem w przypadku, gdy $\text{opt}(\mathcal{I}) = 0$, gdzie \mathcal{I} to instancja oryginalnego problemu, jedyne optymalne rozwiązanie \mathbf{x}' dla scenariusza s' , które spełnia tą nierówność, także musi mieć wartość równą 0⁵.

Jak łatwo zauważać, koszty obydwu scenariuszy dla problemu zostały tak dobrane, aby oba optymalne rozwiązania dla każdego scenariusza nie różniły się pod względem wartości, która wynosi $v_{s_3}^* = v_{s_4}^* = 3 \cdot 2^n - 1$. Jest to o tyle istotne, że podstawiając te dane do ogólnego wzoru, otrzymujemy:

$$\min_{\mathbf{x} \in X} \max_{s \in \{s_3, s_4\}} v(\mathbf{x}, s) - v(\mathbf{x}_s^*, s) = \min \left\{ \max_{s \in \{s_3, s_4\}} v(\mathbf{x}_{s_3}^*, s) - v(\mathbf{x}_s^*, s), \max_{s \in \{s_3, s_4\}} v(\mathbf{x}_{s_4}^*, s) - v(\mathbf{x}_s^*, s) \right\}. \quad (3.13)$$

Na tym etapie możemy zauważyć, że bez względu na to, które rozwiązanie wybierzemy (czy $\mathbf{x}_{s_3}^*$ czy $\mathbf{x}_{s_4}^*$), dla obu rozpatrywanych scenariuszy otrzymamy identyczną wartość żalu równą zera. Podstawiając dane do wzoru 3.13, otrzymamy zatem: $\min \{\max \{0, 0\}, \max \{0, 0\}\} = 0$. Innych rozwiązań nie rozpatrywaliśmy, gdyż z samej definicji problemu MIN-MAX REGRET wynika, że $\forall \mathbf{x} \in X v(\mathbf{x}, s) - v(\mathbf{x}_s^*, s) \geq 0$, toteż wartości otrzymane dla pozostałych rozwiązań z pewnością nie mogą być lepsze.

Zakładając, że analogiczne twierdzenie dla MIN-MAX REGRET do 3.2.2 jest prawdziwe, spodziewamy się, że wartość optymalnego rozwiązania dla dodatkowego scenariusza s' będzie nie gorsza niż k -krotność powyżej otrzymanego rozwiązania (będzie równa 0). Z rysunku 3.6c natomiast jednoznacznie wynika, że rozwiązaniem dla tego scenariusza jest zbiór krawędzi $T_{s'}^* = \{e_5, e_6, e_7, e_8\}$ o całkowitym koszcie równym $2^{n+2} + 2$, gdzie $\mathbf{x}_{s'}^* = [0, 0, 0, 0, 1, 1, 1, 1]$, zaś po podstawieniu danych do wzoru otrzymamy:

$$\begin{aligned} \min_{\mathbf{x} \in \{x_{s'}^*\}} \max_{s \in \{s_3, s_4\}} v(\mathbf{x}, s) - v(\mathbf{x}_s^*, s) &= \max_{s \in \{s_3, s_4\}} v(x_{s'}^*, s) - v(\mathbf{x}_s^*, s) = \\ &= \max \{v(x_{s'}^*, s_3), v(x_{s'}^*, s_4)\} - (3 \cdot 2^n - 1) = \max \{3 \cdot 2^n + 1, 3 \cdot 2^n + 1\} - 3 \cdot 2^n + 1 = 2. \end{aligned}$$

Wynikiem możemy dowolnie manipulować — przykładem takiej manipulacji są scenariusze s'_3, s'_4 oraz s'' , przedstawione (wraz z optymalnymi rozwiązaniami dla każdego z osobna) na rysunkach 3.7a, 3.7b i 3.7c, gdzie optymalną wartością rozwiązania problemu MIN-MAX REGRET dla tych pierwszych jest ponownie wartość 0, zaś dla trzeciego z nich: $2^{n+1} \cdot (2^n - 1)$.

Pokazaliśmy zatem, że w tym przypadku nie możemy zastosować analogicznego twierdzenia.

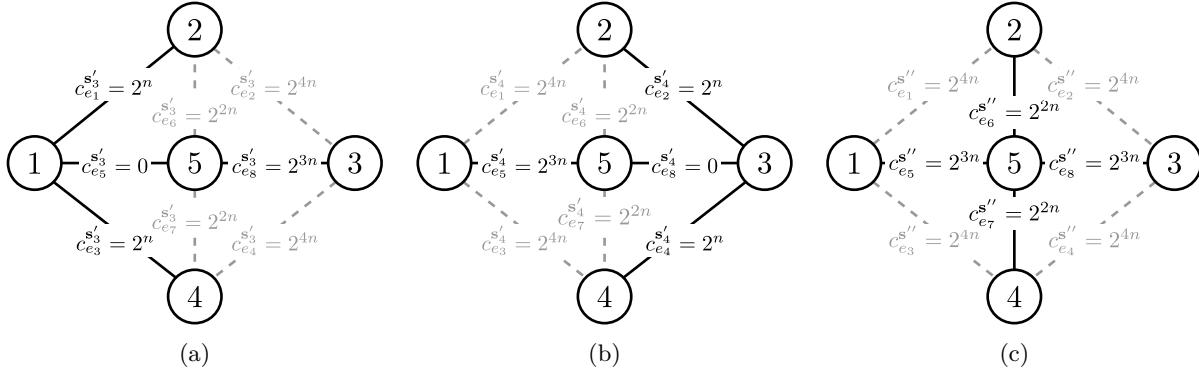
3.3.2 Przypadek ciągły

Zacznijmy od następującego twierdzenia:

Twierdzenie 3.3.1 [2, 431] Niech dany będzie problem minimalizacyjny \mathcal{P} . Niech \underline{s} i \bar{s} będą krytycznymi scenariuszami (odpowiednio najgorszym i najlepszym), definiującymi przestrzeń scenariuszy S . Wartość żalu dla dowolnego rozwiązania $\mathbf{x} \in X$ jest największa dla, zależnego od \mathbf{x} , scenariusza $s^-(\mathbf{x})$, którego koszty są zdefiniowane następująco:

$$c_i^{s^-(\mathbf{x})} = \begin{cases} c_i^{\bar{s}} & \text{gdy } e_i \text{ należy do rozwiązania } (\mathbf{x}_i = 1), \\ c_i^{\underline{s}} & \text{w przeciwnym przypadku } (\mathbf{x}_i = 0), \end{cases} \quad \forall i \in \{1, 2, \dots, m\}. \quad (3.14)$$

⁵Rozpatrywane przez nas grafy nie posiadają ujemnych wag krawędzi.

Rysunek 3.7: Scenariusze s'_3 i s'_4 , dla których scenariusz najgorszych kosztów s'' daje bardzo złe rozwiązanie.

(a) Minimalne drzewo rozpinające dla scenariusza s'_3 o całkowitym koszcie $v_{s'_3}^* = 2^{3n} + 2 \cdot 2^n$. (b) Minimalne drzewo rozpinające dla scenariusza s'_4 o takim samym koszcie. (c) Optymalne drzewo rozpinające dla scenariusza s'' .

Dowód. [2, 431] Dla danego rozwiązania $\mathbf{x} \in X$ niech będzie dany zbiór $\mathbb{1}(\mathbf{x}) = \{i : i \in \{1, \dots, m\} \wedge x_i = 1\}$. Wartość żalu dla scenariusza $\mathbf{s} \in S$ i dowolnego rozwiązania \mathbf{x} wynosi:

$$\begin{aligned} R(\mathbf{x}, \mathbf{s}) &= v(\mathbf{x}, \mathbf{s}) - v_{\mathbf{s}}^* \stackrel{(1)}{=} \sum_{i \in \mathbb{1}(\mathbf{x}) \setminus \mathbb{1}(\mathbf{x}_s^*)} c_i^{\mathbf{s}} - \sum_{i \in \mathbb{1}(\mathbf{x}_s^*) \setminus \mathbb{1}(\mathbf{x})} c_i^{\mathbf{s}} \stackrel{(2)}{\leq} \sum_{i \in \mathbb{1}(\mathbf{x}) \setminus \mathbb{1}(\mathbf{x}_s^*)} c_i^{\bar{\mathbf{s}}} - \sum_{i \in \mathbb{1}(\mathbf{x}_s^*) \setminus \mathbb{1}(\mathbf{x})} c_i^{\mathbf{s}} \stackrel{(3)}{=} \\ &\stackrel{(3)}{=} v(\mathbf{x}, \mathbf{s}^- (\mathbf{x})) - v(\mathbf{x}_s^*, \mathbf{s}^- (\mathbf{x})) \stackrel{(4)}{\leq} v(\mathbf{x}, \mathbf{s}^- (\mathbf{x})) - v(\mathbf{x}_{s^- (\mathbf{x})}^*, \mathbf{s}^- (\mathbf{x})) = \\ &= v(\mathbf{x}, \mathbf{s}^- (\mathbf{x})) - v_{\mathbf{s}^- (\mathbf{x})}^* = R(\mathbf{x}, \mathbf{s}^- (\mathbf{x})), \end{aligned}$$

co dowodzi poprawności twierdzenia. Aby lepiej zrozumieć ciąg logicznego rozumowania jaki został podjęty, skupimy się teraz na kilku, kluczowych dla dowodu, przekształceniach.

- (1) Niech $\emptyset(\mathbf{x}) = \{i : i \in \{1, \dots, m\} \wedge x_i = 0\}$. Łatwo zauważyc, że $\emptyset(\mathbf{x})$ i $\mathbb{1}(\mathbf{x})$ są rozłączne, zaś ich suma $\emptyset(\mathbf{x}) \cup \mathbb{1}(\mathbf{x})$ generuje cały zbiór $\{i : i \in \{1, \dots, m\}\}$. Rozpisując wzór na wartość rozwiązania \mathbf{x} dla danego scenariusza \mathbf{s} mamy: $v(\mathbf{x}, \mathbf{s}) = \sum_{e_i \in E} x_i \cdot c_i^{\mathbf{s}} = \sum_{i \in (\emptyset \cup \mathbb{1})(\mathbf{x})} x_i \cdot c_i^{\mathbf{s}} = \sum_{i \in \emptyset(\mathbf{x})} 0 \cdot c_i^{\mathbf{s}} + \sum_{i \in \mathbb{1}(\mathbf{x})} 1 \cdot c_i^{\mathbf{s}} = \sum_{i \in \mathbb{1}(\mathbf{x})} c_i^{\mathbf{s}}$. Analogicznie możemy postąpić dla $v_{\mathbf{s}}^* = v(\mathbf{x}_s^*, \mathbf{s}) = \sum_{i \in \mathbb{1}(\mathbf{x}_s^*)} c_i^{\mathbf{s}}$. Zauważmy, że w wyrażeniu $v(\mathbf{x}, \mathbf{s}) - v_{\mathbf{s}}^*$ wszystkie elementy sumy, które należą do obu zbiorów, zredukują się tj.

$$\begin{aligned} v(\mathbf{x}, \mathbf{s}) - v_{\mathbf{s}}^* &= \sum_{i \in \mathbb{1}(\mathbf{x})} c_i^{\mathbf{s}} - \sum_{i \in \mathbb{1}(\mathbf{x}_s^*)} c_i^{\mathbf{s}} = \left(\sum_{i \in \mathbb{1}(\mathbf{x}) \setminus \mathbb{1}(\mathbf{x}_s^*)} c_i^{\mathbf{s}} + \sum_{i \in \mathbb{1}(\mathbf{x}_s^*) \setminus \mathbb{1}(\mathbf{x})} c_i^{\mathbf{s}} - \sum_{i \in \mathbb{1}(\mathbf{x}_s^*) \cap \mathbb{1}(\mathbf{x})} c_i^{\mathbf{s}} \right) - \sum_{i \in \mathbb{1}(\mathbf{x}_s^*)} c_i^{\mathbf{s}} = \\ &= \sum_{i \in \mathbb{1}(\mathbf{x}) \setminus \mathbb{1}(\mathbf{x}_s^*)} c_i^{\mathbf{s}} - \sum_{i \in \mathbb{1}(\mathbf{x}_s^*) \setminus \mathbb{1}(\mathbf{x})} c_i^{\mathbf{s}}. \end{aligned}$$

- (2) Łatwo zauważyc, że suma po prawej stronie nierówności jest większa ze względu na definicje kosztów scenariuszy $\bar{\mathbf{s}}$ oraz \mathbf{s} ($\sum_{i \in \mathbb{1}(\mathbf{x}) \setminus \mathbb{1}(\mathbf{x}_s^*)} c_i^{\mathbf{s}} \leq \sum_{i \in \mathbb{1}(\mathbf{x}) \setminus \mathbb{1}(\mathbf{x}_s^*)} c_i^{\bar{\mathbf{s}}}$ oraz $\sum_{i \in \mathbb{1}(\mathbf{x}_s^*) \setminus \mathbb{1}(\mathbf{x})} c_i^{\mathbf{s}} \geq \sum_{i \in \mathbb{1}(\mathbf{x}_s^*) \setminus \mathbb{1}(\mathbf{x})} c_i^{\bar{\mathbf{s}}}$).

- (3) Przeprowadzone rozumowanie jest analogiczne do tego z punktu (2) — na uwagę zasługuje fakt, iż:

$$\sum_{i \in \mathbb{1}(\mathbf{x}) \setminus \mathbb{1}(\mathbf{x}_s^*)} c_i^{\bar{\mathbf{s}}} - \sum_{i \in \mathbb{1}(\mathbf{x}_s^*) \setminus \mathbb{1}(\mathbf{x})} c_i^{\mathbf{s}} = \sum_{i \in \mathbb{1}(\mathbf{x}) \setminus \mathbb{1}(\mathbf{x}_s^*)} c_i^{\mathbf{s}^-} - \sum_{i \in \mathbb{1}(\mathbf{x}_s^*) \setminus \mathbb{1}(\mathbf{x})} c_i^{\mathbf{s}^-},$$

co bezpośrednio wynika z definicji scenariusza $\mathbf{s}^- (\mathbf{x})$ — dla wszystkich elementów pierwszej sumy (indeksowanej $i \in \mathbb{1}(\mathbf{x}) \setminus \mathbb{1}(\mathbf{x}_s^*)$) wartości kosztów dla tych elementów x_i przyjmują wartość $c_i^{\bar{\mathbf{s}}}$ ($x_i = 0$). Analogicznie druga suma jest indeksowana po i , które w szczególności nie należą do zbioru $\mathbb{1}(\mathbf{x})$, tak więc dla każdego elementu tej sumy spełniony jest warunek $x_i = 0$. Poczyniwszy tę obserwację, dalsze przekształcanie formuły odbywa się identycznie jak przedstawiono w kroku (2).



(4) Wynika bezpośrednio z właściwości optymalnego rozwiązania $\mathbf{x}_{\mathbf{s}^-(\mathbf{x})}^*$ dla scenariusza $\mathbf{s}^-(\mathbf{x})$:

$$v(\mathbf{x}_{\mathbf{s}}^*, \mathbf{s}^-(\mathbf{x})) \geq v(\mathbf{x}_{\mathbf{s}^-(\mathbf{x})}^*, \mathbf{s}^-(\mathbf{x})) = v_{\mathbf{s}^-(\mathbf{x})}^*. \quad \diamond$$

Udowodniliśmy zatem, że dla dowolnego rozwiązania $\mathbf{x} \in X$ wartość $R(\mathbf{x}, \mathbf{s}) = v(\mathbf{x}, \mathbf{s}) - v_{\mathbf{s}}^*$ osiąga swoje maksimum dla $\mathbf{s} = \mathbf{s}^-(\mathbf{x})$. W oparciu o twierdzenie 3.3.1 możemy wyciągnąć następujący wniosek, od którego tylko krok dzieli nas od podania metody na rozwiązywanie zagadnienia INTERVAL MIN-MAX REGRET.

Wniosek 3.3.1 *Dla problemu INTERVAL MIN-MAX REGRET \mathcal{P} , gdzie \mathcal{P} jest problemem minimalizacyjnym, optymalnym rozwiązaniem \mathbf{x}^* jest jedno z rozwiązań $\mathbf{x} \in X$ dla scenariusza $\mathbf{s}^-(\mathbf{x})$.*

Znaczy to dla nas tyle, że zamiast rozwiązywać problem z definicji: $\min_{\mathbf{x} \in X} \max_{\mathbf{s} \in S} R(\mathbf{x}, \mathbf{s})$, możemy równoważnie rozważać problem $\min_{\mathbf{x} \in X} R(\mathbf{x}, \mathbf{s}^-(\mathbf{x}))$. Tak więc, jeśli istnieje optymalne rozwiązanie problemu $\min_{\mathbf{x} \in X} R(\mathbf{x}, \mathbf{s}^-(\mathbf{x}))$, jest ono także optymalnym rozwiązaniem dla $\min_{\mathbf{x} \in X} \max_{\mathbf{s} \in S} R(\mathbf{x}, \mathbf{s})$.

Choć powyższe twierdzenie da się łatwo uogólnić na klasę problemów MIN-MAX REGRET (co też uczyliśmy), pierwotny pomysł wywodzi się z rozważań nad problemem MIN-MAX REGRET MINIMUM SPANNING TREE [2, 429–430] [25], który szczególnie nas interesuje. Podobnie możemy postąpić z następującym twierdzeniem, którego potrzebujemy aby udowodnić prawidłowość w drugą stronę: że istnienie optymalnego rozwiązania dla problemu INTERVAL MIN-MAX REGRET pociąga za sobą istnienie optymalnego rozwiązania dla problemu minimalizacyjnego \mathcal{P} dla jednego z ekstremalnych scenariuszy. Udowodnijmy zatem: $\exists \mathbf{x}^* = \min \arg_{\mathbf{x} \in X} \max_{\mathbf{s} \in S} R(\mathbf{x}, \mathbf{s}) \Rightarrow \exists \mathbf{s}(\mathbf{x}) \in S : \min \arg_{\mathbf{x} \in X} R(\mathbf{x}, \mathbf{s}(\mathbf{x})) = \mathbf{x}^*$.

Twierdzenie 3.3.2 [2, 432] *Niech dany będzie problem minimalizacyjny \mathcal{P} i optymalne rozwiązanie problemu INTERVAL MIN-MAX REGRET \mathcal{P} \mathbf{x}^* . Rozwiązanie optymalne \mathbf{x}^* odpowiada optymalnemu rozwiązaniu problemu \mathcal{P} dla co najmniej jednego scenariusza ekstremalnego, w szczególności dla $\mathbf{s}^+(\mathbf{x}^*)$, którego koszty są zdefiniowane następująco:*

$$c_i^{s^+(\mathbf{x}^*)} = \begin{cases} c_i^{\mathbf{s}} & \text{gdy } e_i \text{ należy do optymalnego rozwiązania } (\mathbf{x}_i^* = 1), \\ c_i^{\bar{\mathbf{s}}} & \text{w przeciwnym przypadku } (\mathbf{x}_i^* = 0), \end{cases} \quad \forall i \in \{1, 2, \dots, m\}. \quad (3.15)$$

Będziemy chcieli zatem udowodnić, że o ile istnieje optymalne rozwiązanie dla problemu INTERVAL MIN-MAX REGRET \mathcal{P} , musi istnieć co najmniej jeden ekstremalny scenariusz (jest nim $\mathbf{s}^+(\mathbf{x}^*)$), który możemy wykorzystać do wygenerowania szukanego rozwiązania (na podstawie twierdzenia 3.3.1) — samo udowodnienie 3.3.1 gwarantuje nam bowiem następującą własność: $\exists \mathbf{s}(\mathbf{x}) \in S : \min \arg_{\mathbf{x} \in X} R(\mathbf{x}, \mathbf{s}(\mathbf{x})) = \mathbf{x}^* \Rightarrow \exists \mathbf{x}^* = \min \arg_{\mathbf{x} \in X} \max_{\mathbf{s} \in S} R(\mathbf{x}, \mathbf{s})$. Dopiero gdy udowodnimy powyższe twierdzenie, będziemy mogli wymiennie stosować obydwie techniki, gdyż wtedy: $\exists \mathbf{s}(\mathbf{x}) \in S : \min \arg_{\mathbf{x} \in X} R(\mathbf{x}, \mathbf{s}(\mathbf{x})) = \mathbf{x}^* \Leftrightarrow \exists \mathbf{x}^* = \min \arg_{\mathbf{x} \in X} \max_{\mathbf{s} \in S} R(\mathbf{x}, \mathbf{s})$.

Dowód. [2, 432] Niech \mathbf{x}^* będzie oznaczało optymalne rozwiązanie dla problemu INTERVAL MIN-MAX REGRET \mathcal{P} oraz niech $\mathbb{1}(\mathbf{x}) = \{i : i \in \{1, \dots, m\} \wedge x_i = 1\}$. Dla dowolnego scenariusza $\mathbf{s} \in S$ mamy:

$$\begin{aligned} v(\mathbf{x}^*, \mathbf{s}) - v(\mathbf{x}_{\mathbf{s}^+(\mathbf{x}^*)}^*, \mathbf{s}) &= \sum_{i \in \mathbb{1}(\mathbf{x}^*) \setminus \mathbb{1}(\mathbf{x}_{\mathbf{s}^+(\mathbf{x}^*)}^*)} c_i^{\mathbf{s}} - \sum_{i \in \mathbb{1}(\mathbf{x}_{\mathbf{s}^+(\mathbf{x}^*)}^*) \setminus \mathbb{1}(\mathbf{x}^*)} c_i^{\mathbf{s}} \geq \sum_{i \in \mathbb{1}(\mathbf{x}^*) \setminus \mathbb{1}(\mathbf{x}_{\mathbf{s}^+(\mathbf{x}^*)}^*)} c_i^{\mathbf{s}} - \sum_{i \in \mathbb{1}(\mathbf{x}_{\mathbf{s}^+(\mathbf{x}^*)}^*) \setminus \mathbb{1}(\mathbf{x}^*)} c_i^{\bar{\mathbf{s}}} = \\ &= v(\mathbf{x}^*, \mathbf{s}^+(\mathbf{x}^*)) - v(\mathbf{x}_{\mathbf{s}^+(\mathbf{x}^*)}^*, \mathbf{s}^+(\mathbf{x}^*)). \end{aligned}$$

Analogiczny do powyższego proces przekształcania wyrażeń został wykorzystany przy dowodzie twierdzenia 3.3.1, tak więc nie będziemy jeszcze raz przytaczać uczynionych w nim spostrzeżeń. Co nas teraz interesuje, to fakt pokazania, że dla optymalnego rozwiązania problemu INTERVAL MIN-MAX REGRET \mathcal{P} \mathbf{x}^* nie istnieje inny scenariusz, którego wartość żalu byłaby mniejsza niż dla $\mathbf{s}^+(\mathbf{x}^*)$. Dodatkowo zauważmy, że jeśli \mathbf{x}^* jest rozwiązaniem optymalnym, to nie istnieje inne rozwiązanie \mathbf{x} takie, że $v(\mathbf{x}, \mathbf{s}^+(\mathbf{x})) \leq v(\mathbf{x}^*, \mathbf{s}^+(\mathbf{x}^*))$, w szczególności $v(\mathbf{x}^*, \mathbf{s}^+(\mathbf{x}^*)) = v(\mathbf{x}_{\mathbf{s}^+(\mathbf{x}^*)}^*, \mathbf{s}^+(\mathbf{x}^*))$ (wartość $v_{\mathbf{s}^+(\mathbf{x}^*)}^*$ jest optymalna z definicji i nie może istnieć od niej inne rozwiązanie o mniejszym koszcie). Stąd dochodzimy do wniosku, że jeśli \mathbf{x}^* jest optymalnym rozwiązaniem dla INTERVAL MIN-MAX REGRET \mathcal{P} , to: $v(\mathbf{x}^*, \mathbf{s}) - v(\mathbf{x}_{\mathbf{s}^+(\mathbf{x}^*)}^*, \mathbf{s}) \geq v(\mathbf{x}^*, \mathbf{s}^+(\mathbf{x}^*)) - v(\mathbf{x}_{\mathbf{s}^+(\mathbf{x}^*)}^*, \mathbf{s}^+(\mathbf{x}^*)) = 0$.



Załóżmy teraz, że \mathbf{x}^* nie jest optymalnym rozwiązaniem problemu \mathcal{P} dla scenariusza $\mathbf{s}^+(\mathbf{x}^*)$. W związku z tym wyrażenie: $v(\mathbf{x}^*, \mathbf{s}) - v(\mathbf{x}_{\mathbf{s}^+(\mathbf{x}^*)}^*, \mathbf{s})$ staje się ostro większe od zera, stąd:

$$\begin{aligned} v(\mathbf{x}^*, \mathbf{s}) - v(\mathbf{x}_{\mathbf{s}^+(\mathbf{x}^*)}^*, \mathbf{s}) &> 0 \\ v(\mathbf{x}^*, \mathbf{s}) &> v(\mathbf{x}_{\mathbf{s}^+(\mathbf{x}^*)}^*, \mathbf{s}) \\ v(\mathbf{x}^*, \mathbf{s}) - v_{\mathbf{s}}^* &> v(\mathbf{x}_{\mathbf{s}^+(\mathbf{x}^*)}^*, \mathbf{s}) - v_{\mathbf{s}}^* \end{aligned}$$

dla dowolnego scenariusza $\mathbf{s} \in S$, więc:

$$\max_{\mathbf{s} \in S} (v(\mathbf{x}^*, \mathbf{s}) - v_{\mathbf{s}}^*) > \max_{\mathbf{s} \in S} (v(\mathbf{x}_{\mathbf{s}^+(\mathbf{x}^*)}^*, \mathbf{s}) - v_{\mathbf{s}}^*).$$

Stoi to w oczywisty sposób w sprzeczności z tym, że \mathbf{x}^* jest optymalnym rozwiązaniem problemu INTERVAL MIN-MAX REGRET \mathcal{P} (wybranie rozwiązania $\mathbf{x}_{\mathbf{s}^+(\mathbf{x}^*)}^* \neq \mathbf{x}^*$ zagwarantowałoby mniejsze odchylenie od optymalnego rozwiązania w przypadku wystąpienia najgorszego scenariusza). ♦

Obydwa powyższe twierdzenia (3.3.1 oraz 3.3.2) pozwalają nam na efektywniejsze poszukiwania rozwiązań dla omawianej klasy problemów. Jak już zdążyliśmy wspomnieć po zakończeniu dowodu pierwszego z wymienionych twierdzeń, zamiast szukać rozwiązania w oparciu o wzór: $\min_{\mathbf{x} \in X} \max_{\mathbf{s} \in S} R(\mathbf{x}, \mathbf{s})$, możemy zastąpić go tym: $\min_{\mathbf{x} \in X} R(\mathbf{x}, \mathbf{s}^-(\mathbf{x}))$. Problemem nadal oczywiście zostaje liczba ekstremalnych scenariuszy, która w wielu przypadkach może być ponad wielomianowa.Więcej na ten temat można przeczytać w [15], gdzie pokazano, że rozwiązanie problemu \mathcal{P} dla odpowiedniego scenariusza zapewnia nam dobre górne ograniczenia na wartość INTERVAL MIN-MAX REGRET \mathcal{P} (rozwiązaniem tym jest wartość, zwrócona przez, zaprezentowany we wspomnianym artykule, algorytm 2-aproxymacyjny) oraz [3], gdzie bliżej przyjrzano się omawianemu przez nas podejściu dla stałej liczby ekstremalnych scenariuszy, bądź ograniczonej w taki sposób, aby złożoność obydwu problemów była taka sama. W [17] możemy zobaczyć podsumowanie stopnia złożoności szerokiej gamy problemów MIN-MAX oraz MIN-MAX REGRET dla obu przypadków (ciągłych oraz dyskretnych), dodatkowo osobno rozpatrywanych dla stałych liczb scenariuszy i dla ich liczby, która jest częścią wejścia (zmienna). Dla wszystkich przypadków problemu minimalnego drzewa rozpinającego uzyskujemy jednak stopień trudności NP-trudny, bądź NP-zupełny [18] [3].

3.4 Optymalizacja z możliwością poprawy

Dotychczas omówione zagadnienia ogólnie nazwaliśmy problemami **optymalizacji odpornej**, to jest niewrażliwej na niekorzystne dla siebie zmiany — bez względu na scenariusz, który okazywał się tym, który zaszedł w rzeczywistości, stosując podejście minimaksowe, mogliśmy zapewnić sobie, że otrzymany wynik będzie zawsze ten sam, niezależnie od tego, czy zrealizowany scenariusz był tym najgorszym, czy definiował koszty w najlepszy dla nas sposób. Warunek był jeden — musieliśmy z wyprzedzeniem znać wszystkie możliwe scenariusze, by móc chociaż zacząć rozważać tego typu problemy (np. przy omawianiu problemu INTERVAL MIN-MAX lub INTERVAL MIN-MAX REGRET, swoje rozważania opieraliśmy w głównej mierze na znajomości krytycznych wartości kosztów scenariuszy). W tej części omówimy problemy, które tych ograniczeń się pozbywają.

3.4.1 Problem przyrostowy

Pierwszym tego typu problemem, na jaki zwróciśmy uwagę, będzie **problem przyrostowy** (ang. *incremental problem*), z którym zetknimy się we wszystkich podejściach do problemu odpornej optymalizacji, jakie wymienimy w tej części [20, 1-2][26, 586]. Swoją nazwę problem zawdzięcza swojej specyfice działania — tak jak w przypadku problemów z rodziny MIN-MAX mogliśmy odnieść wrażenie, że nasza odpowiedzialność za podjęcie optymalnej decyzji zaczyna się w momencie otrzymania kompletnej informacji o wszystkich scenariuszach $\mathbf{s} \in S$ i dozwolonych wyborach $\mathbf{x} \in X$ i jednocześnie kończy wraz z momentem ich przeanalizowania i zwróceniu rozwiązania, tak w tym przypadku możemy obserwować specyficzne „narastanie” problemu

i stopniowy wzrost skomplikowania podejmowanych decyzji. Założymy, że na podstawie posiadanych informacji o kosztach opisujących początkowy stan problemu \mathcal{P} , wyznaczyliśmy optymalne rozwiązanie \mathbf{x}^* . Niech po pewnym czasie koszty te ulegną na tyle znaczącym zmianom, iż rozwiązanie \mathbf{x}^* przestanie być optymalne⁶. W ramach definicji problemu przyrostowego, pod wpływem pojawienia się nowych danych, możemy zmienić naszą decyzję, lecz nie może to być zupełnie nowe rozwiązanie — wybrany wektor \mathbf{x}^{**} musi być w pewnym określonym stopniu podobny do rozwiązania poprzedniego \mathbf{x}^* . Niech po pewnym czasie koszty te ulegną na tyle znaczącym zmianom, iż rozwiązanie \mathbf{x}^{**} przestanie być optymalne. Nasz wybór kolejnego rozwiązania podlega tym samym zasadom co poprzednio — widzimy więc, że rozwiązanie \mathbf{x}^{**} zależy od początkowo wybranego rozwiązania \mathbf{x}^* , \mathbf{x}^{**} zależy od dwóch poprzednich rozwiązań, każde kolejne od wszystkich swoich poprzedników (poprzednio lub nie). Aby formalnie zapisać problem INCREMENTAL, musimy wprowadzić kilka dodatkowych pojęć:

- $\mathbf{x}', \mathbf{x}'', \dots$ — oznaczają kolejne rozwiązania problemu, wybierane z **otoczenia** poprzedniego,
- $X_{\mathbf{x}}^k$ — **otoczenie** wektora \mathbf{x} charakteryzuje zbiór rozwiązań, z którego jesteśmy zobligowani wybrać na sze nowe rozwiązanie, w przypadku chęci zmiany rozwiązania pierwotnego (\mathbf{x}). Parametrem k będziemy oznaczać stopień, w jakim dane dwa rozwiązania mogą się od siebie różnić; w ogólnym przypadku wartość ta będzie determinowała zbiór rozwiązań za pomocą specjalnej funkcji $f : X \times X \rightarrow \mathbb{Z}^+$, która dla dwóch wybranych rozwiązań zwracać będzie ich „odległość” od siebie. W przypadku rozważania problemów grafowych, funkcja zwykle zwraca liczbę krawędzi, które nie należą jednocześnie do obu rozwiązań. Formalna definicja zbioru: $X_{\mathbf{x}}^k = \{\mathbf{y} \in X : f(\mathbf{y}, \mathbf{x}) \leq k\}$.

Naszym celem, przy rozwiązywaniu problemu INCREMENTAL dla zadanego rozwiązania początkowego \mathbf{x} , jest znalezienie wartości wyrażenia (a właściwie rozwiązania, dla którego jest ono przyjmowane):

$$\min_{\mathbf{y} \in X_{\mathbf{x}}^k} v(\mathbf{y}, \mathbf{s}'), \quad (3.16)$$

gdzie \mathbf{s}' reprezentuje nowe koszty, dla których poszukujemy optymalnego rozwiązania. Niech **miarą unikalności** rozwiązania będzie funkcja $f(T', T) = |T' \setminus T|$, gdzie T' oraz T są zbiorami krawędzi, przedstawiającymi minimalne drzewo rozpinające, odpowiednio dla scenariuszy \mathbf{s}' i \mathbf{s} , tak jak przedstawiono na rysunkach 3.8a oraz 3.8b. Dodatkowo niech $k = 1$. Zgodnie z definicją naszej funkcji $f(T', T)$, zbiorem dopuszczalnych rozwiązań dla tak zdefiniowanego problemu INCREMENTAL MINIMUM SPANNING TREE są wszystkie te zbioru krawędzi, które zawierają co najwyżej jeden element nie należący do zbioru krawędzi, który charakteryzuje początkowe rozwiązanie T (wyrażenie $|T' \setminus T|$ oznacza **moc**⁷ zbioru $T' \setminus T$, czyli liczbę wszystkich krawędzi, które należą do T' , zaś nie znajdują się w zbiorze T). Łatwo zauważać, że w tym konkretnym przypadku zachodzi $|T \setminus T'| = |T' \setminus T|$, jako że obydwa zbioru są równoliczne (z definicji liczba ich elementów wynosi $|V| - 1$) a ich elementy nie powtarzają się (np. zgodnie z rysunkiem 3.8: $T = T_{\mathbf{s}}^* = \{e_2, e_5, e_6, e_8, e_9\}$, $T' = T_{\mathbf{s}'}^* = \{e_1, e_2, e_6, e_7, e_8\}$, $|T_{\mathbf{s}}^* \setminus T_{\mathbf{s}'}^*| = |\{e_2, e_5, e_6, e_8, e_9\} \setminus \{e_1, e_2, e_6, e_7, e_8\}| = |\{e_5, e_9\}| = 2 = |\{e_1, e_7\}| = |\{e_1, e_2, e_6, e_7, e_8\} \setminus \{e_2, e_5, e_6, e_8, e_9\}| = |T_{\mathbf{s}'}^* \setminus T_{\mathbf{s}}^*|$). Niestety, podane przez nas nowe rozwiązanie T' , będące minimalnym drzewem rozpinającym dla scenariusza \mathbf{s}' , nie jest optymalnym rozwiązaniem w myśl definicji rozpatrywanego przez nas problemu INCREMENTAL MINIMUM SPANNING TREE (dalej IMST) — co więcej, nie jest ono nawet rozwiązaniem dopuszczalnym, jako że $= f(T', T) \not\leq k = 1$.

Naiwne rozwiązanie

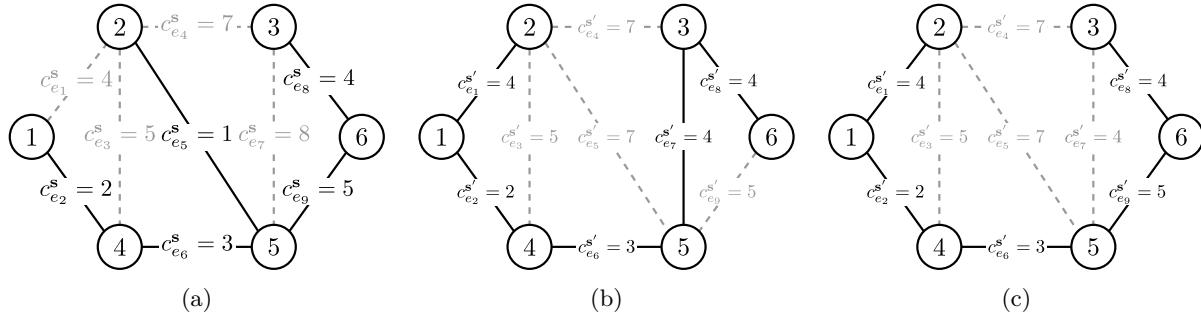
Aby otrzymać poprawne rozwiązanie T^* problemu INCREMENTAL MINIMUM SPANNING TREE — \mathcal{P} , spełniającego ograniczenie $f(T^*, T) \leq 1$ — naturalnym wydaje się rozpoczęć jego konstrukcję od uzyskania zbioru krawędzi, który jest optymalny dla zadanego scenariusza (T' — niedopuszczalne w \mathcal{P}), a następnie przekształcić je w rozwiązanie optymalne, tak jak to pokazano na rysunkach 3.8b i 3.8c — z uzyskanego zbioru krawędzi T' (na rysunku $T_{\mathbf{s}'}^*$), będącego optymalnym rozwiązaniem dla scenariusza \mathbf{s}' , ze zbioru krawędzi nie należących do pierwotnego rozwiązania (będącego przyczyną jego niedopuszczalności, $T' \setminus T = \{e_1, e_7\}$) usunięto krawędź o najwyższym koszcie (e_7) i zastąpiono ją krawędzią o najniższym koszcie ze zbioru $T \setminus T' =$

⁶Warunek ten oczywiście nie musi zachodzić w ogólnym przypadku — gdybyśmy dopuścili taką sytuację, że po zmianie kosztów stare rozwiązanie nadal jest optymalne, niepotrzebne byłoby szukanie następnego rozwiązania.

⁷Liczba elementów w zbiorze.



$\{e_5, e_9\}$. Tym samym otrzymane rozwiązanie T'' spełnia $f(T', T) - 1 = f(T'', T) = k$, zaś sposób jego konstrukcji gwarantuje nam jego optymalność, co w danym przypadku jest trywialne do zauważenia. Zatem $T'' = T^*$ dla tak zdefiniowanego problemu \mathcal{P} .



Rysunek 3.8: Różnice między rozwiązaniami problemów MINIMUM SPANNING TREE i INCREMENTAL MST z parametrem $k = 1$. (a) Optymalne rozwiązanie problemu minimalnego drzewa rozpinającego T_s^* dla scenariusza $s = \{4, 2, 5, 7, 1, 3, 8, 4, 5\}$ o całkowitym koszcie rozwiązania $v_{\text{MST}}(T_s^*, s) = 15$. (b) Niedopuszczalne rozwiązanie $T_{s'}^* = \{e_1, e_2, e_6, e_7, e_8\}$ problemu INCREMENTAL MINIMUM SPANNING TREE, będące jednocześnie optymalnym dla scenariusza s' w problemie MST o koszcie $v_{\text{MST}}(T_{s'}^*, s') = 17$ (c) Optymalne rozwiązanie problemu INCREMENTAL MST i scenariusza s' : $T^* = \{e_1, e_2, e_6, e_8, e_9\}$ o całkowitym koszcie $v_{s'}^* = 18$.

Niestety wraz ze wzrostem grafu oraz przede wszystkim parametru k , rozwiązanie to staje się nieefektywne — jak mogliśmy zauważyć, opisany algorytm działa sekwencyjnie, co każdą iterację zmniejszając wartość funkcji $f(T^i, T)$, gdzie T^i to rozwiązanie uzyskane podczas i -tej iteracji. Ogólnie: $f(T^{i-1}, T) - 1 = f(T^i, T)$ ⁸. Naszym zadaniem zatem jest konstrukcja takiego ciągu drzew rozpinających T^1, \dots, T^l , aby $f(T^l, T^0) = k$, gdzie $T^0 = T$ i jest początkowym rozwiązaniem problemu \mathcal{P} dla pierwotnego scenariusza, zaś $l = f(T^1, T^0) - k + 1$ ($k = f(T^l, T^0) = f(T^{l-1}, T^0) - 1 = \dots = f(T^2, T^0) - (l-2) = f(T^1, T^0) - (l-1) = k$ — ogólnie $f(T^{l-i}, T^0) - i = k$). Każde kolejne rozwiązanie powinno zatem składać się z coraz to większej liczby krawędzi należących do T^0 . Aby to osiągnąć, dla każdej i -tej iteracji musielibyśmy rozpatrzyć wszystkie możliwe podziały drzewa T^{i-1} , powstałe w wyniku usunięcia jednej z krawędzi $e^{i-1} \in T^{i-1} \setminus T^0$ — innymi słowy wszystkie takie pary drzew (T_1^{i-1}, T_2^{i-1}) , których łączna liczba krawędzi nie należących do drzewa T^0 jest o 1 mniejsza niż dla drzewa T^{i-1} . Takich par w i -tej iteracji jest dokładnie $f(T^1, T^0) - k - (i - 1)$ (iteracje numerujemy rozpoczynając od $i = 1$). Dla każdej takiej pary musimy następnie znaleźć wszystkie krawędzie należące do zbioru $Q(T^{i-1}, e^{i-1}) \cap T^0$, łączące ze sobą dane poddrzewa⁹, gdzie krawędź $e^{i-1} \in T^{i-1}$ jest krawędzią usuwaną z T^{i-1} , nienależącą do T^0 . Na samym końcu musimy stworzyć nowe drzewo, do którego dołączymy krawędź $e' \in Q(T^{i-1}, e^{i-1}) \cap T^0$ o najkorzystniejszym stosunku jej wagi do kosztu usuniętej krawędzi. Formalnie: $T^i = (\{e : e \in T^{i-1} \setminus e^{i-1}\} \cup e') : e' = \min arg_{e'} \frac{c_{e'}}{c_{e^{i-1}}}$.

Złożoność takiego rozwiązania to $O(|V| \cdot (l^3 + |V|))$, gdzie l to liczba iteracji algorytmu (miara tego, jak daleko od dopuszczalności dla INCREMENTAL MST jest optymalne rozwiązanie dla problemu MST). W pierwszej iteracji liczba potencjalnych krawędzi do usunięcia wynosi $l - 1$ i stopniowo maleje. Dla każdej takiej krawędzi musimy zaś zdeterminować zbiór możliwych do dodania łuków. Bez straty ogólności możemy założyć, że po uzyskaniu podziału drzewa T^{i-1} na (T_1^{i-1}, T_2^{i-1}) , aby ustalić właściwy zbiór krawędzi, będziemy szukać odpowiednich łuków wychodzących z wierzchołków, należących do drzewa o mniejszej ich liczbie (będziemy analizować krawędzie wychodzące z wierzchołków $v \in T_1^{i-1}$ jeśli $|\{v_i : e_{ij} \in T_1^{i-1}\}| \leq |\{v_i : e_{ij} \in T_2^{i-1}\}|$ ¹⁰, z wierzchołków $v \in T_2^{i-1}$ w przeciwnym przypadku). Możemy założyć, że w najgorszym z możliwych przypadków wszystkie krawędzie, tworzące podział na poddrzewa T_1^{i-1} oraz T_2^{i-1} , dzieli oryginalne drzewo T^{i-1} na możliwie równe części — każdorazowe ustalenie zbioru krawędzi $Q(T^{i-1}, e^{i-1})$ (a tym

⁸W rozdziale 5 omówimy algorytm, który także działa iteracyjnie ze względu na wartość tej funkcji, lecz przedstawiona własność nie jest zachowana, co przekłada się na dużo lepsze otrzymywane czasy działania.

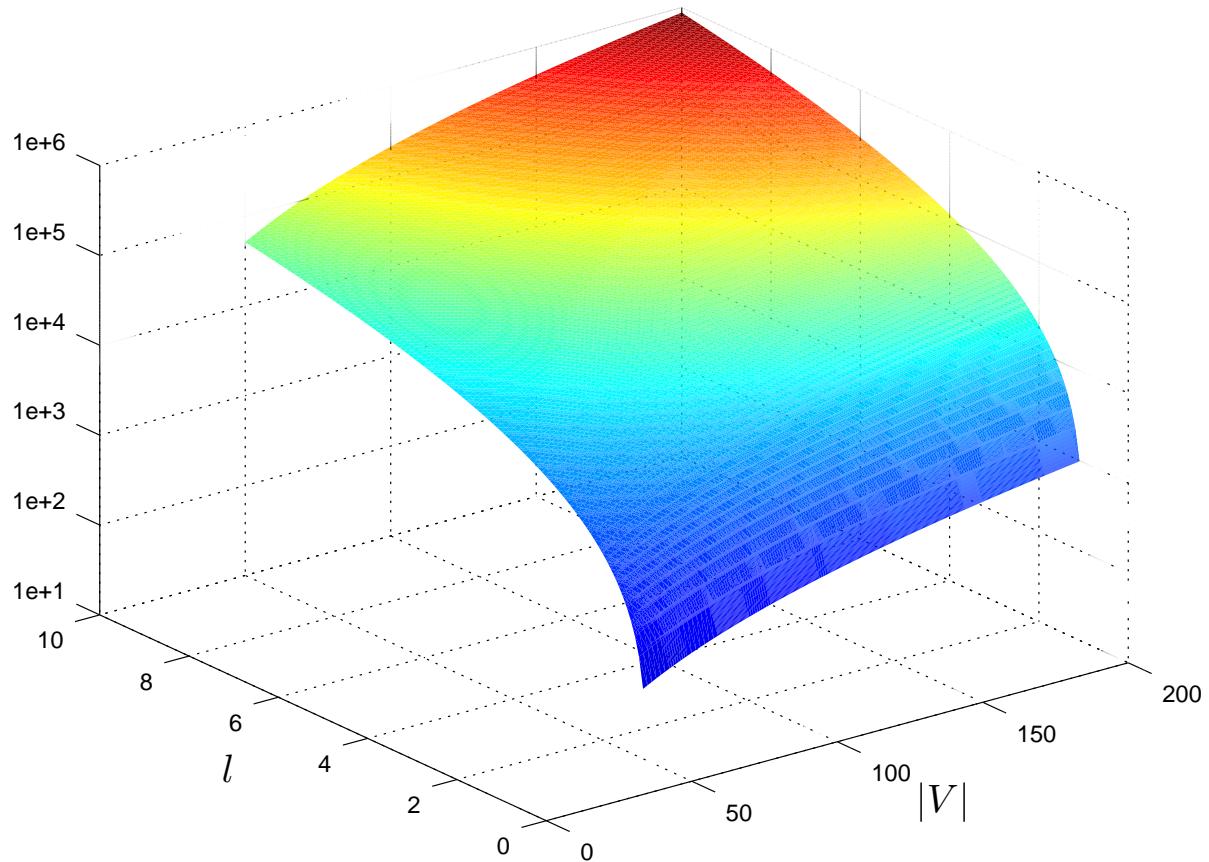
⁹O zbiorze krawędzi łączącym dwa poddrzewa rozpinające powstałe w wyniku cięcia oryginalnego drzewa pisaliśmy w poprzednim rozdziale (2.2).

¹⁰Wyrażenie $|\{v_i : e_{ij} \in T\}|$ oznacza po prostu liczbę wierzchołków, które łączą ze sobą krawędzie należące do drzewa T — jako że rozpatrujemy tylko grafy nieskierowane, $\{v_i : e_{ij} \in T\} \equiv \{v_j : e_{ij} \in T\}$ (kolejność wierzchołków w zbiorach nie jest istotna).

samym $\mathcal{Q}(T^{i-1}, e^{i-1}) \cap T^0)$ zatem wymagać będzie od nas przeglądnienia wszystkich krawędzi wychodzących z $\frac{|V|-1}{2} - \frac{i-j}{2}$ wierzchołków, gdzie j to indeks krawędzi e_j , według której dokonujemy podziału. Możemy także założyć, że w pesymistycznym przypadku będziemy mieli do czynienia z grafem pełnym, gdzie stopień każdego wierzchołka $v \in V$ wynosi $d(v) = |V| - 1$. Stąd otrzymujemy wzór na ogólną liczbę porównań wybranych par krawędzi (e^{i-1}, e'), gdzie e^{i-1} jest krawędzią wychodzącą z rozwiązania, zaś e' — wchodzącą:

$$\begin{aligned} \sum_{i=1}^{l-1} \sum_{j=1}^i (|V| - 1) \cdot \left(\frac{|V| - 1}{2} - \frac{i-j}{2} \right) &= \frac{1}{12} \cdot (l-3) \cdot (|V|-1) \cdot (l^2 - 3 \cdot l \cdot |V| + 2) = \\ &= \frac{l^3 \cdot (|V|-1)}{12} - \frac{l^2 \cdot (|V|^2 - 1)}{4} + \frac{3 \cdot l \cdot |V|^2}{4} - \frac{7 \cdot l \cdot |V|}{12} - \frac{l}{6} - \frac{|V|}{2} + \frac{1}{2}. \end{aligned}$$

Dodając do tego koszt odnalezienia potencjalnych krawędzi, które będziemy usuwać ($|V| - 1$ — wystarczy raz przejść po wszystkich krawędziach drzewa T^0 ; z każdą następną iteracją zbiór ten pomniejsza się o ustaloną krawędź, więc nie musimy za każdym razem rozpoczynać analizy krawędzi drzewa od nowa), otrzymujemy — wymieniony na samym początku — stopień złożoności, który ilustruje wykres 3.9.



Rysunek 3.9: Stopień złożoności naiwnego algorytmu, rozwiązującego problem minimalnego drzewa rozpinającego w wersji INCREMENTAL, głównie zależy od liczby krawędzi, o które optymalne rozwiązanie dla problemu MST różni się od dopuszczalnej liczby nowych krawędzi względem starego rozwiązania w problemie INCREMENTAL MINIMUM SPANNING TREE.

Najprostszym z możliwych opisów tak zdefiniowanego problemu jest model przedstawiony w 3.17, gdzie T_0 oznacza nasze drzewo początkowe, T^* — dla problemu IMST, gdzie $x_e = 1$ oznacza należenie do niego krawędzi $e \in E$.



$$\min \sum_{e \in E} c_e \cdot x_e, \quad (3.17a)$$

$$x_e = 1, \quad \forall e \in T_0, \quad (3.17b)$$

$$\sum_{e \in T^* \setminus T_0} x_e \leq k, \quad (3.17c)$$

$$x_e \in \{0, 1\}, \forall e \in E, \quad (3.17d)$$

3.4.2 Zagadnienia oparte na problemie INCREMENTAL

Problem adwersarza

Przyjrzymy się teraz nieco bardziej rozbudowanemu problemowi, znanego jako **problem adwersarza** (ang. adversarial problem). W tym przypadku zamiast skupiać się na minimalizacji rozwiązania, naszym celem (celem adwersarza) jest maksymalizowanie jego wartości [20, 2]. Osiągnąć cel możemy poprzez wybór takiego scenariusza, dla którego najlepsza wartość rozwiązania problemu INCREMENTAL będzie najgorsza:

$$\max_{\mathbf{s}' \in S} \min_{\mathbf{y} \in X_{\mathbf{x}}^k} v(\mathbf{y}, \mathbf{s}') \quad (3.18)$$

Latwo zauważyc, że oba te wybory — adwersarza i osoby wybierającej rozwiązanie dla problemu INCREMENTAL — nie zależą od siebie, przez co problem sprowadza się do omawianego wcześniej, z tą różnicą, że zamiast posiadania dwóch scenariuszy (gdzie na podstawie pierwszego wybieraliśmy rozwiązanie w postaci wektora \mathbf{x}), mamy ich większą liczbę i adwersarz dla każdego z nich musi policzyć wartość $\min_{\mathbf{y} \in X_{\mathbf{x}}^k} v(\mathbf{y}, \mathbf{s}')$, by móc wybrać najbardziej sprzyjający mu scenariusz (skutkujący zwróceniem największej wartości). Problem można zatem bez trudu urównoleglić (rozwiązywać jednocześnie klasyczny problem IMST dla wielu scenariuszy naraz), przez co nie da się o nim napisać wiele więcej ponad to, co już napisaliśmy przy omawianiu wcześniejszego problemu.

Odporny problem przyrostowy z możliwością poprawy rozwiązania

Kolejnym rozszerzeniem poprzednio omawianego problemu jest problem **odpornej optymalizacji przyrostowej** (ang. *robust incremental optimization* [20, 2]). W odróżniu od problemów typu INCREMENTAL oraz problemu adwersarza, w tym przypadku chcemy cofnąć się o jeden krok w procesie poszukiwania rozwiązania i zminimalizować jego wartość, uwzględniając dodatkowo naszą pierwszą decyzję — tą odnoszącą się do wektora \mathbf{x} , będącą podstawą do rozwiązania problemu INCREMENTAL. Jak możemy się domyślać, problem ten definiujemy w następujący sposób:

$$\min_{\mathbf{x} \in X} \left(v(\mathbf{x}, \mathbf{s}) + \max_{\mathbf{s}' \in S} \min_{\mathbf{y} \in X_{\mathbf{x}}^k} v(\mathbf{y}, \mathbf{s}') \right) \quad (3.19)$$

Niech początkowym scenariuszem, na podstawie którego wybierzemy wektor \mathbf{x} , będzie $\mathbf{s}_0 = \{1, 1, 1, 1, 1, 1\}$, zaś scenariuszami, spośród których będzie mógł wybierać adwersarz:

- $\mathbf{s}_1 = \{4, 6, 1, 9, 3, 7\}$ oraz
- $\mathbf{s}_2 = \{7, 3, 9, 1, 9, 4\}$.

Oczywiście wybór scenariusza \mathbf{s}_0 nie jest przypadkowy — naszym celem jest pokazanie możliwej do osiągnięcia różnicy w jakości rozwiązania dla dwóch różnych sposobów podejścia do zadanego problemu: klasycznego, polegającego na prostym wybraniu rozwiązania o najmniejszym koszcie w pierwszym kroku, z pominięciem problemu adwersarza, tożsamym z rozwiązaniem zagadnienia minimalnego drzewa rozpinającego dla pojedynczego scenariusza ($\min_{\mathbf{x} \in X} v(\mathbf{x}, \mathbf{s})$), oraz odpornego. Latwo zauważyc, że w przypadku tak dobranych scenariuszy (tak jak to pokazano na rysunkach od 3.10a do 3.10f oraz 3.11), na pozór nic nie znacząca

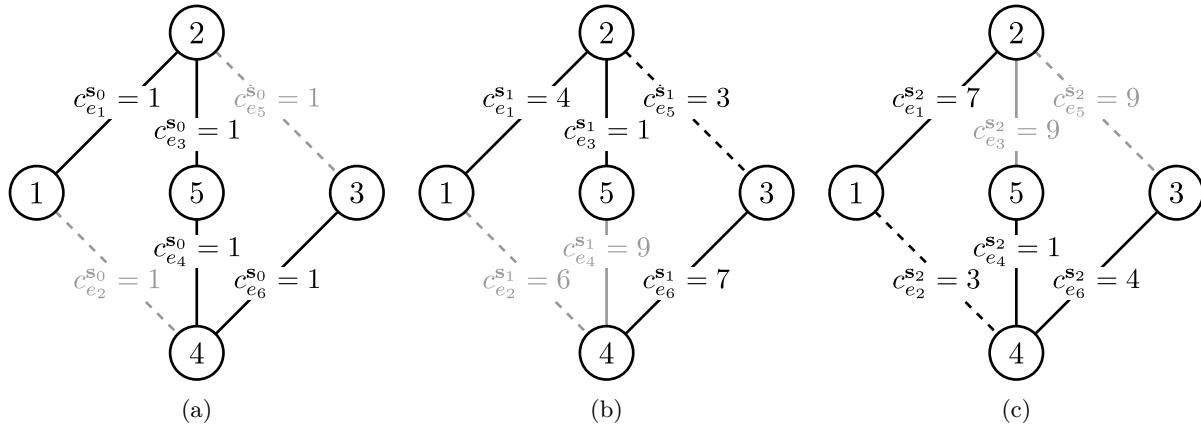
decyzja o pierwszym wyborze rozwiązania (jako że wszystkie koszty w scenariuszu s_0 są sobie równe, dowolnie wybrane drzewo rozpinające charakteryzuje ten sam koszt — będzie zatem wyborem optymalnym), w przypadku pojawienia się odpowiednich zmian kosztów, może zaważyć na optymalności całego rozwiązania w myśl 3.19. Dodatkowo niech parametr dla problemu INCREMENTAL MINIMUM SPANNING TREE wynosi $k = 1$. Przedstawiany na rysunkach 3.10a–3.10f, 3.11 graf jest na tyle nieduży, że z powodzeniem możemy rozwiązać ten problem z użyciem czystej siły — dla każdego możliwego drzewa rozpinającego, reprezentowanego przez wektor \mathbf{x} , możemy wyliczyć wartość $v(\mathbf{x}, \mathbf{s}) + \max_{\mathbf{s}' \in S} \min_{\mathbf{y} \in X_{\mathbf{x}}^k} v(\mathbf{y}, \mathbf{s}')$, rozwiązując po drodze 24 egzemplarze problemu INCREMENTAL MINIMUM SPANNING TREE (po dwa dla każdej z 12 instancji problemu ADVERSARIAL INCREMENTAL MINIMUM SPANNING TREE — $X = \{\mathbf{x}_1, \mathbf{x}_1, \dots, \mathbf{x}_{12}\}$). Otrzymane wyniki zgromadzono w tabeli 3.3, która reprezentuje wartości optymalnych rozwiązań poszczególnych składowych głównego problemu. Tabela nie przedstawia natomiast wyników, uzyskanych dla problemu, przedstawionego w równaniu (3.19) — możemy dostrzec, że dla tak specyficznie zdefiniowanego scenariusza s_0 , suma kosztów wszystkich krawędzi w dowolnie wybranym drzewie rozpinającym jest taka sama (wynosi 4), więc wartości te możemy z powodzeniem obliczyć sami (dodając tą wartość do przedstawionych wyników dla problemu adwersarza).¹¹

Tablica 3.3: Tabela przedstawiająca wszystkie, możliwe do skonstruowania dla grafu prezentowanego na rysunkach od 3.10a do 3.10f oraz 3.11, drzewa rozpinające, reprezentowane przez wektory od \mathbf{x}_1 do \mathbf{x}_{12} , wartości optymalnych rozwiązań problemów INCREMENTAL MINIMUM SPANNING TREE dla wszystkich scenariuszy adwersarza (s_1 oraz s_2) z uwzględnieniem parametru $k = 1$ oraz (w ostatniej kolumnie) wartość optymalnego rozwiązania problemu adwersarza dla zbioru jego scenariuszy oraz danego wektora początkowego \mathbf{x} . Rozwiązania, reprezentowane przez wektory od \mathbf{x}_1 do \mathbf{x}_{12} , są posortowane według rosnącej wartości dla odpowiadającego im rozwiązania problemu ADVERSARIAL INCREMENTAL MINIMUM SPANNING TREE.

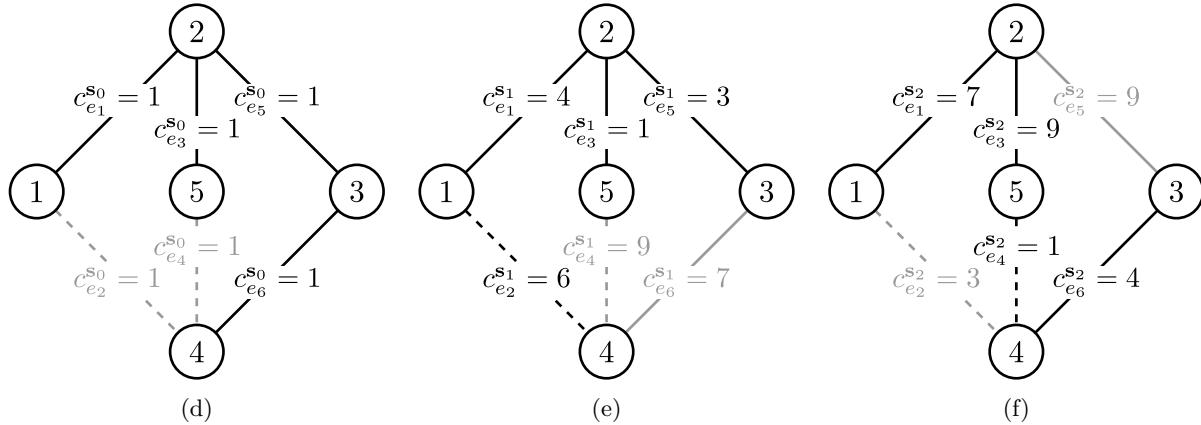
X	e_1	e_2	e_3	e_4	e_5	e_6	Scenariusze		
							$v_{s_1}^{\text{IMST}} = \min_{\mathbf{y} \in X_{\mathbf{x}}^k} v(\mathbf{y}, s_1)$	$v_{s_2}^{\text{IMST}} = \min_{\mathbf{y} \in X_{\mathbf{x}}^k} v(\mathbf{y}, s_2)$	$\max \{v_{s_1}^{\text{IMST}}, v_{s_2}^{\text{IMST}}\}$
\mathbf{x}_1	1	1	0	1	1	0	14	15	15
\mathbf{x}_2	1	0	0	1	1	1	15	15	15
\mathbf{x}_3	0	1	0	1	1	1	17	15	17
\mathbf{x}_4	0	1	1	0	1	1	14	17	17
\mathbf{x}_5	1	0	1	1	0	1	15	15	15
\mathbf{x}_6	0	1	1	1	0	1	17	15	17
\mathbf{x}_7	1	1	1	0	0	1	14	15	15
\mathbf{x}_8	1	0	1	0	1	1	14	21	21
\mathbf{x}_9	1	0	1	1	1	0	14	20	20
\mathbf{x}_{10}	0	1	1	1	1	0	14	17	17
\mathbf{x}_{11}	1	1	1	0	1	0	14	20	20
\mathbf{x}_{12}	1	1	0	1	0	1	18	15	18

Widzimy zatem jak bardzo skomplikowanym i trudnym obliczeniowo procesem jest generowanie rozwiązań dla problemu RECOVERABLE ROBUST INCREMENTAL MINIMUM SPANNING TREE i jak ciężko musielibyśmy pracować, aby dojść do tego punktu (po drodze poznaliśmy dwa inne problemy, które są tylko jego częścią). Poziom złożoności opisanego przez nas problemu dodatkowo potęguje fakt, że przedstawiony przez nas przykład tyczył się grafu o bardzo niewielkich rozmiarach — liczył on sobie zaledwie 6 krawędzi i 5 wierzchołków. Jego konstrukcja z kolei w bardzo znaczący sposób ograniczyła liczbę możliwych rozwiązań (liczliwość zbioru X), my zaś jeszcze bardziej uproszciliśmy problem, wybierając do problemu adwersarza najmniejszą liczbę scenariuszy, jaka ma jeszcze dla tego problemu sens (s_1, s_2). W ogólnym przypadku nie możemy mieć nadzieję, że takie podejście do problemu (rozwiązywanie siłowe) będzie możliwe.

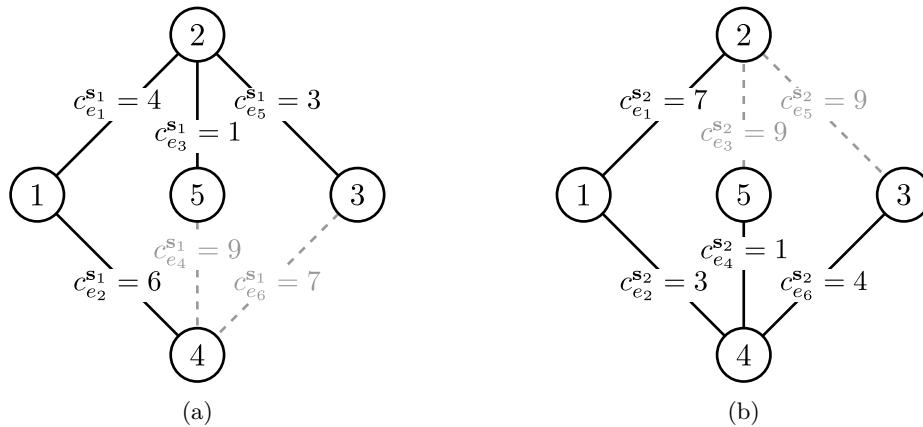
¹¹ Ze względu na dość ogólne znaczenie, jakie ma zwrot odpornej optymalizacji przyrostowej, będziemy chcieli od niego uciec i od tej pory odnosić się do naszego problemu, u którego podstawa leży problem minimalnego drzewa rozpinającego w wersji INCREMENTAL, jako do problemu RECOVERABLE ROBUST INCREMENTAL MINIMUM SPANNING TREE (dalej RRIMST), gdzie takie oznaczenie bezpośrednio wskazuje na podstawowy problem, z jakim się będziemy zmagać. W literaturze często możemy się dodatkowo spotkać z inną jego nazwą: k-DIST RECOVERABLE ROBUST SPANNING TREE [13].



Rysunek 3.10: Sytuacja dla problemu RECOVERABLE ROBUST INCREMENTAL MINIMUM SPANNING TREE dla wybranego rozwiązania $T = T_{\mathbf{x}_5} = \{e_1, e_3, e_4, e_6\}$, gdzie dwa ostatnie rysunki prezentują drzewa, z których rozwiązanie wybierać będzie adwersarz. **(a)** Drzewo rozpinające wybrane przez podejmującego decyzję (nie przez adwersarza), reprezentowane przez wektor $\mathbf{x}_5 = [1, 0, 1, 1, 0, 1]$ (patrz: tabela 3.3) o sumie kosztów wybranych krawędzi $v(T, s_0) = 4$. **(b)** Minimalne drzewo rozpinające $T_{\mathbf{s}_1}^{\text{IMST}}$ o najmniejszym koszcie dla problemu typu INCREMENTAL z parametrem $k = 1$ oraz drzewem początkowym T . Krawędzią, która weszła do rozwiązania nowego drzewa, jest e_5 ($T_{\mathbf{s}_1}^{\text{IMST}} \setminus T = \{e_5\}$), zaś usunięta — e_4 . Koszt rozwiązania wynosi 15. **(c)** Rozwiązanie problemu INCREMENTAL MINIMUM SPANNING TREE dla drugiego scenariusza (s_2) i tego samego początkowego drzewa rozpinającego T . Liczba krawędzi nie należących do drzewa T , jednocześnie zawierających się w nowym rozwiązaniu $T_{\mathbf{s}_2}^{\text{IMST}}$, wynosi 1. W obu przypadkach koszt optymalnych rozwiązań danego problemu jest taki sam ($v(T_{\mathbf{s}_1}^{\text{IMST}}, s_1) = v(T_{\mathbf{s}_2}^{\text{IMST}}, s_2) = 15$), toteż wybór scenariusza (s_1 lub s_2), który zostanie podjęty przez adwersarza, nie ma żadnego znaczenia (dla rozwiązania reprezentowanego przez \mathbf{x}_5).



Rysunek 3.10: Wybrane drzewo rozpinające $T_{\mathbf{x}_8}$ (patrz: tabela 3.3) dla problemu RECOVERABLE ROBUST INCREMENTAL MINIMUM SPANNING TREE wraz z dwoma drzewami, będącymi optymalnymi rozwiązaniami problemu INCREMENTAL MINIMUM SPANNING TREE dla scenariuszy s_1 oraz s_2 . **(d)** Wybrane rozwiązanie, reprezentowane przez wektor $\mathbf{x}_8 = [1, 0, 1, 0, 1, 1]$, o koszcie 4. **(e)** Optymalne rozwiązanie problemu minimalnego drzewa rozpinającego w wersji INCREMENTAL dla scenariusza s_1 (początkowego s_0) o wartości rozwiązania równej 14. **(f)** Minimalne drzewo rozpinające, które jest optymalnym rozwiązaniem dla tego samego problemu, dla scenariusza s_2 . Jako że koszt tego rozwiązania jest największy spośród wszystkich rozwiązań problemu IMST dla scenariuszy adwersarza (koszt tego rozwiązania wynosi 21), zostanie ono przez niego wybrane. Poprzednie rozwiązanie (o koszcie 14) zostanie odrzucone jako nieopłacalne.



Rysunek 3.11: Optymalne rozwiązania problemów INCREMENTAL MINIMUM SPANNING TREE dla podwyższonowej wartości parametru $k = 2$, która dla konkretnego grafu jest wystarczająca, aby zwracane rozwiązania dla poszczególnych scenariuszy były takie same jak dla klasycznego problemu minimalnego drzewa rozpinającego
(a) Minimalne drzewo rozpinające dla scenariusza $s_1 = [4, 6, 1, 9, 3, 7]$. **(b)** Optymalne rozwiązanie problemu minimalnego drzewa rozpinającego w wersji INCREMENTAL dla scenariusza s_2 .

3.5 Podsumowanie rozdziału

Widzimy, że sposobów podejścia do problemów optymalizacyjnych jest bardzo dużo, zaś te przez nas przedstawione stanowią już dużo bardziej zaawansowane spojrzenie na ten problem, niż pokazaliśmy w poprzednim rozdziale, omawiając podstawowe pojęcia związanego z problemem minimalnego drzewa rozpinającego. Wraz z omawianiem kolejnych zagadnień w tym rozdziale, mogliśmy zauważać naszą potrzebę konstrukcji takich rozwiązań, które pozwalały nam będą wybierać najlepsze dla nas rozwiązania nie tylko wtedy, gdy zawsze przewidzimy wszystkie możliwe sytuacje (im większy zbiór zdarzeń zostanie przez nas uwzględniony, tym bardziej ogólne rozwiązanie najprawdopodobniej wybierzymy), ale i na bieżąco reagować na zaistniałe sytuacje (co pozwala na wybór znacznie lepszych rozwiązań, uwzględniających tylko najbliższą przyszłość). Dodatkowo chcieliśmy aby tak konstruowane rozwiązania łatwo oddawały się zmianom — aby wybór jednego (optymalnego w czasie jego wyboru) nie powodował konieczności jego całkowitej rekonstrukcji w przypadku zajścia przewidzianego przez nas scenariusza. W związku z tym podaliśmy formułę, która wybiera dla nas takie rozwiązania, które nie tylko łatwo modyfikować (poprawiać), ale i zapewniają najlepsze rezultaty w najbliższej, przewidzianej przez nas, przyszłości. Niestety wraz z rosnącymi oczekiwaniami oczywiście rósł poziom skomplikowania opisywanych problemów i dla ostatniego z nich (RECOVERABLE ROBUST INCREMENTAL MINIMUM SPANNING TREE) zwyczajnie nie jesteśmy już w stanie podać wielomianowego algorytmu, który radziłby sobie z danym problemem (w przypadku dwóch pozostałych: ADVERSARIAL INCREMENTAL MINIMUM SPANNING TREE oraz INCREMENTAL MINIMUM SPANNING TREE podamy je w następnych rozdziałach). W związku z tym, w rozdziale 6 przedstawimy inną metodę podejścia do danego problemu (aproksymacyjną). Dodatkowo warto w tym miejscu podkreślić, że wszystkie do tej pory omówione przypadki problemów koncentrowały się na znanej, ograniczonej przez daną stałą, liczbie przypadków, co w zdecydowanie upraszcza sposób patrzenia się na te problemy — analizę przez nas omawianych przypadków (tyle że dla nieograniczonej liczby scenariuszy) możemy znaleźć w [16]. Następnym rozdziałem odejdziemy nieco od omawianych jak dotąd problemów — wyjaśnimy znaczenie modelu (równania 3.17), przedstawiając podstawowe pojęcia z nieco innego obszaru algorytmiki — programowania LP, MIP oraz IP.



Problemy programowania liniowego

Chcąc podjąć próbę formalnego opisania przedstawionych wcześniej problemów, należy najpierw zapoznać się z ogólnymi zasadami ich zapisywania. Jak mogliśmy zaobserwować w poprzednim rozdziale, opisując problem typu **INCREMENTAL** (3.17), uciekliśmy się do przedstawienia go za pomocą serii nierówności oraz równości — innymi słowy zapisaliśmy go w postaci **modelu programowania liniowego**. Możliwość przekształcenia dowolnie zadanego problemu do takiej postaci może nam zagwarantować odnalezienie jego istniejącego rozwiązania w czasie wielomianowym i takie narzędzie może stanowić doskonały punkt wyjściowy do dalszych rozważań. My także z niego skorzystamy do konstrukcji optymalnego algorytmu, rozwiązującego problem odpornej optymalizacji dla wyszukiwania minimalnego drzewa rozpinającego, w rozdziale następnym, a opierając się na jego własnościach, wykażemy poprawność takiego podejścia.

4.1 Programowanie liniowe

Mając na uwadze, że nie chcemy zajmować się zagadnieniem programowania liniowego samym w sobie, gdyż nie jest to naszym celem, w poniższym rozdziale zostaną przytoczone tylko podstawowe fakty, niezbędne do zrozumienia dalszej jego części. Punktem wyjścia do pozyskania głębszej wiedzy na poruszany w tym rozdziale temat może być książka [22], która omawia wszystkie aspekty tej dziedziny, począwszy od programowania wkleślego, przez liniowe, do całkowitoliczbowego, tłumacząc zasady działania każdego z tych modeli.

Przez pojęcie **programowania liniowego** będziemy rozumieli zbiór liniowych równań oraz nierówności wraz z określona dla nich **funkcją celu**, która także musi przybrać formę liniowego wyrażenia arytmetycznego. Zbiór tych ograniczeń wraz z funkcją celu będziemy nazywali **modelem prymalnym**¹ programowania liniowego. Schemat takiego modelu wygląda następująco:

$$\min \quad f(\mathbf{x}, \dots), \quad (4.1a)$$

$$\text{takie, że } g_1(\mathbf{x}, \dots) \diamondsuit_1 b_1, \quad (\text{ang. } \textit{subject to}) \quad (4.1b)$$

$$g_2(\mathbf{x}, \dots) \diamondsuit_2 b_2, \quad (4.1c)$$

$$\dots \quad (4.1d)$$

$$g_m(\mathbf{x}, \dots) \diamondsuit_m b_m, \quad (4.1e)$$

$$x_i \geq x_i^{\text{LB}}, \quad x_i \in \mathbf{x}, \quad (4.1f)$$

$$x_i \leq x_i^{\text{UB}}, \quad x_i \in \mathbf{x}, \quad (4.1g)$$

gdzie znak \diamondsuit_i zastępuje dowolny znak ze zbioru $\{=, \leq, \geq\}$, funkcja $f(\mathbf{x}, \dots)$ jest funkcją celu i przyjmuje postać $\sum_{i=1}^n c_i \cdot x_i$, \mathbf{x} jest wektorem symbolizującym rozwiązanie, \mathbf{c} — zbiorem kosztów ($|\mathbf{x}| = |\mathbf{c}| = n$). Zbiór funkcji $\{g_1(\mathbf{x}, \dots), \dots, g_m(\mathbf{x}, \dots)\}$ oraz, odpowiadające im, wartości $\{b_1, \dots, b_m\}$ będziemy zaś odpowiednio nazywać **ograniczeniami** modelu oraz **wartościami prawych stron**. O spełnieniu danej relacji \diamondsuit_i przez parę $(g_i(\mathbf{x}, \dots), b_i)$ będziemy mówić, że zaszło ograniczenie i — oczywiście rozwiązanie dowolnego

¹Terminem „prymalny” będziemy określać modele programowania liniowego/całkowitoliczbowego, których funkcja celu jest minimalizacyjna. Problemem **dualnym** nazwiemy model, który skupia się na maksymalizowaniu jej wartości. Zagadnienia związane z obydwooma modelami zostały dokładniej opisane w książce [22, 67–73].



modelu jest dopuszczalne tylko wtedy, gdy spełnia ono je wszystkie. Wektor \mathbf{x} będziemy nazywać zbiorem **zmiennych decyzyjnych** modelu. Dla wszystkich do tej pory opisanych problemów w poprzednim rozdziale wykorzystywaliśmy podobne oznaczenia — zmienna decyzyjna x_i przyjmowała odpowiednio wartość 0, gdy krawędź e_i nie należała do poszukiwanego rozwiązania, oraz 1 w przeciwnym przypadku. Tutaj generalizujemy tę ideę, dopuszczając przyjmowanie przez te zmienne wartości z zakresu od x_i^{LB} do x_i^{UB} , gdzie odpowiednio x_i^{LB} oznacza dolne ograniczenie na wartość zmiennej x_i (ang. *lower bound*), zaś x_i^{UB} — górne (ang. *upper bound*). Od rodzaju wartości jakie mogą przyjmować te zmienne zależy z jakiego typu problemem mamy do czynienia — w przypadku, gdy wartości $x_i \in \langle x_i^{\text{LB}}, x_i^{\text{UB}} \rangle$, model, w którym występują tylko takie zmienne, będziemy nazywali modelem programowania liniowego. W przeciwnym przypadku, gdy w modelu pojawią się zmienne o wartościach dyskretnych ($x_i \in [x_i^{\text{LB}}, x_i^{\text{UB}}]$), mówimy o modelu **całkowitoliczbowym** (IP). Ma to o tyle duże znaczenie, że w pierwszym przypadku jesteśmy w stanie zagwarantować, że o ile dla zdefiniowanego modelu istnieje **rozwiązanie optymalne** (minimalizujące funkcję celu $f(\mathbf{x}, \dots)$), otrzymamy je w czasie wielomianowym od wielkości modelu. Informację o braku takiego rozwiązania również uzyskamy w takim czasie (często dużo szybciej), gdy algorytm służący do rozwiązywania tak zdefiniowanych problemów znajdzie się w stanie, który będzie tożsamy z brakiem istnienia jakiegokolwiek rozwiązania²). W przypadku zaś modeli całkowitoliczbowych nie mamy takiej gwarancji³ i często czas, potrzebny do znalezienia rozwiązania, wykracza poza wielomianowy. W przypadku gdy w modelu występują oba typy zmiennych, mamy do czynienia z modelem mieszanym **MIP** (ang. *Mixed Integer Programming*), którego czas rozwiązania również jest ponad-wielomianowy. Aby zilustrować schemat działania tak opisanego modelu, posłużymy się prostym przykładem (4.1).

$\begin{array}{ll} \min & 2 \cdot x_1 + 3 \cdot x_2, \\ \text{s.t.} & 3 \cdot x_1 + 2 \cdot x_2 \leqslant 12, \\ & 1 \cdot x_1 - 4 \cdot x_2 \geqslant 2, \\ & 0 \leqslant x_1 \leqslant 3, \quad x_i \in \mathbf{x}, \\ & 0 \leqslant x_2 \leqslant 4, \quad x_i \in \mathbf{x}. \end{array}$	$f(\mathbf{x}, \mathbf{c}) = \sum_{i=1}^{i=2} c_i \cdot x_i,$ $g_1(\mathbf{x}, \mathbf{a}_1) = \sum_{i=1}^{i=2} a_{1,i} \cdot x_i,$ $g_2(\mathbf{x}, \mathbf{a}_2) = \sum_{i=1}^{i=2} a_{2,i} \cdot x_i.$	$\begin{array}{ll} \min & f(\mathbf{x}, \mathbf{c}), \\ \text{s.t.} & g_1(\mathbf{x}, \mathbf{a}_1) \leqslant b_1, \\ & g_2(\mathbf{x}, \mathbf{a}_2) \geqslant b_2, \\ & 0 \leqslant x_1 \leqslant 3, \quad x_i \in \mathbf{x}, \\ & 0 \leqslant x_2 \leqslant 4, \quad x_i \in \mathbf{x}. \end{array}$
(a)	(b)	(c)

Rysunek 4.1: Przykładowy problem sformułowany jako model programowania liniowego. (a) Jawnie zapisany model — w takiej formie będzie rozwiązywany przez wyspecjalizowany algorytm do rozwiązywania zagadnień programowania liniowego. (b) Reprezentacja wszystkich elementów modelu w zwartej formie funkcji z następującymi zdefiniowanymi zbiorami wartości: $\mathbf{x} = \{x_1, x_2\}$, $x_1 \in \langle 0, 3 \rangle$, $x_2 \in \langle 0, 4 \rangle$, $\mathbf{c} = [2, 3]$, $\mathbf{b} = [12, 2]$, $\mathbf{a}_1 = [3, 2]$ i $\mathbf{a}_2 = [1, -4]$. (c) Zwarty zapis modelu.

Oczywistym rozwiązaniem problemu 4.1 jest para wartości $(2, 0)$. Przypisanie takich liczb do zmiennych x_1 oraz x_2 spełnia wszystkie ograniczenia zawarte w modelu, wartości przypisane zmiennym należą do ich dziedzin, zaś wartość funkcji celu dla tak zadanych zmiennych osiąga swoje minimum. Zwykle możemy spotkać się z macierzowym zapisem modelu, gdzie przedstawione przez nas wektory \mathbf{a}_1 oraz \mathbf{a}_2 traktuje się jako macierz $A = [\mathbf{a}_1, \mathbf{a}_2]^T$ a układ ograniczeń zadanego modelu prezentuje się w postaci $A \cdot \mathbf{x} = \mathbf{b}$.

4.1.1 Problem minimalnego drzewa rozpinającego

W dalszej części rozdziału poznamy szereg różnych modeli przedstawiających problem minimalnego drzewa rozpinającego, poczynając od bezpośredniej próby przełożenia jego definicji na model matematyczny, poprzez

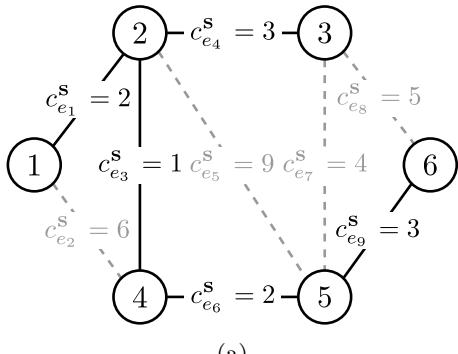
²Dla przykładu: ograniczenia w postaci funkcji $g(\mathbf{x}, \dots)$ mogą tak ograniczyć zbiór wartości, które może przyjąć dana zmienna decyzyjna x_i , że w pewnym momencie $x_i \in \emptyset$ (co jest sygnałem dla algorytmu, że dalsze obliczenia są bezcelowe).

³Algorytm rozwiązuający dany problem w pierwszej kolejności znajdzie rozwiązanie problemu ignorując ograniczenia na całkowitoliczbowość zmiennych decyzyjnych, a następnie zacznie szukać takiego rozwiązania, które będzie je spełniać, będąc jednocześnie możliwe najlepszym. W wielu przypadkach kończy się to na inteligentnym przeglądaniu wszystkich możliwych rozwiązań metodą np. podziału i ograniczeń (ang. *branch and bound*) [22, 433–448].

modele całkowitoliczbowe aż do modelu programowania liniowego, którego specyficzna budowa zapewni nam całkowitoliczbowe rozwiązanie. Na tej podstawie będziemy chcieli skonstruować modele odpowiadające problemowi minimalnego drzewa rozpinającego w wersji INCREMENTAL, które posłużą nam jako punkt wyjścia do dalszych rozważań.

4.1.2 Naiwne podejście

Pierwszym i zarazem najbardziej naturalnym sposobem podejścia do zamodelowania interesującego nas problemu jest model przedstawiony poniżej. Opiera się on na ogólnie znanych właściwościach drzew rozpinających i ich podstruktury — jeśli dane drzewo T jest drzewem rozpinającym dla grafu $G = (V, E)$ (między innymi składa się z $|V| - 1$ krawędzi), to każdy podzbiór wierzchołków $S \subseteq V$ grafu jest połączony ze sobą dokładnie $|S| - 1$ krawędziami (zobacz rysunek 4.2a). Niech zbiór $E(S)$ będzie zdefiniowany jako $E(S) = \{ \{i, j\} : v_i \xrightarrow{1} v_j \in E \wedge v_i \in S \wedge v_j \in S \}$ i oznacza zbiór wszystkich krawędzi łączących dowolne dwa różne wierzchołki ze zbioru S . Niech dodatkowo dla każdej krawędzi w grafie $e \in E$ będzie zdefiniowana zmienna decyzyjna x_e , której wartość odpowiada decyzji o przynależności danej krawędzi do minimalnego drzewa rozpinającego, będącego rozwiązaniem T dla modelu 4.2b (innymi słowy, minimalne drzewo rozpinające T składa się z krawędzi e , dla których $x_e = 1$ — $T = \{e : x_e = 1\}$). W takiej sytuacji, równanie 4.4c w naturalny sposób opisuje wskazaną przez nas własność. W szczególności dla zbioru $S = V$ mamy $E(S) = E$ a wspomniane równanie sprowadza się do wyrażenia 4.4b, które w dobitny sposób wyraża podstawową własność, posiadaną przez drzewa rozpinające dowolny graf. Teraz, aby uzyskać minimalne takie drzewo, wystarczy byśmy zdefiniowali funkcję celu, która będzie dążyć do zminimalizowania kosztów znalezionego rozwiązania (4.4a). Niestety model, którego idea wydaje się być bardzo prosta, jest równie intuicyjny co niepraktyczny w zastosowaniu — wystarczy spojrzeć na definicję zbioru $E(S)$ bądź na rysunek 4.2a oraz jego opis, by dojść do wniosku, że liczba wymaganych przez model ograniczeń jest ponad-wielomianowa — znacznie większa niż czas, w którym spodziewalibyśmy się otrzymać rozwiązanie takiego modelu programowania liniowego (przypomnijmy, że modelem LP nazywamy model, którego zmienne decyzyjne nie są ograniczone do dyskretnego zbioru liczb)⁴



(a)

$$\min \sum_{e \in E} c_e \cdot x_e, \quad (4.4a)$$

$$\text{s.t. } \sum_{e \in E} x_e = |V| - 1, \quad (4.4b)$$

$$\sum_{e \in E(S)} x_e \leq |S| - 1, \quad S \subseteq V, \quad (4.4c)$$

$$x_e \geq 0, \quad e \in E. \quad (4.4d)$$

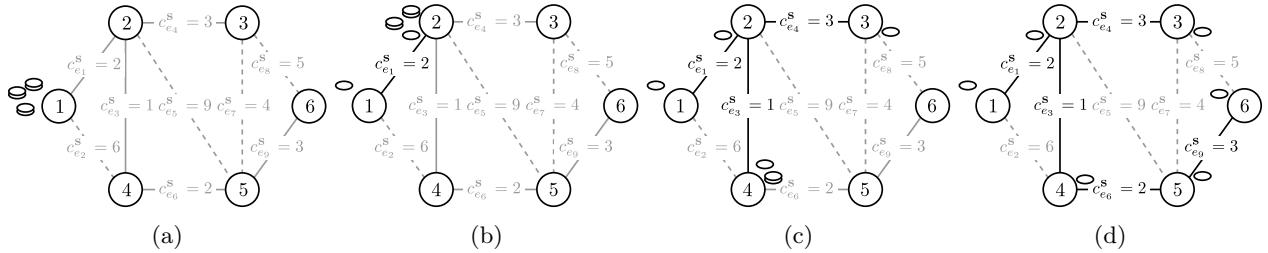
(b)

Rysunek 4.2: (a) Przykładowe drzewo rozpinające $T = \{e_1, e_3, e_4, e_6, e_9\}$ dla grafu $G = (V, E)$. Dla dowolnego podzbioru $S \subseteq V$ liczba krawędzi $\{e_{ij} : e_{ij} \in T \wedge v_i \in S \wedge v_j \in S\}$ jest równa $|S| - 1$. (b) Model liniowego programowania dla problemu minimalnego drzewa rozpinającego.

⁴Z wiedzy ogólnej wiemy, że wszystkich podzbiorów zbioru, składającego się z n elementów, jest 2^n ($\sum_{i=0}^n \binom{n}{i}$). Wzór ten doskonale opisuje naszą sytuację — chcemy mieć ograniczenie dla wszystkich możliwych zbiorów S jedno-, dwu-, ..., n -elementowych. Z drugiej strony powiedzieliśmy (rozdzielając pojęcia programowania liniowego od całkowitoliczbowego), że istnienie modelu LP gwarantuje nam możliwość otrzymywania jego rozwiązania w czasie wielomianowym **od rozmiaru problemu**. W związku z czym algorytm rozwiążający dany model będzie zwracał interesujący nas wynik w nieakceptowalnym dla nas czasie. Aby mimo wszystko uzyskać chciane rozwiązanie w czasie wielomianowym, musimy zrezygnować z algorytmów, które czasem działania płacą za swoją uniwersalność (taki jak przyjęty algorytm SIMPLEX [8], którym potrafimy rozwiązać dowolne zagadnienie z LP), na rzecz bardziej złożonych metod, które do działania nie potrzebują pełnego opisu problemu, tak jak wspomniany przez nas algorytm (a zatem ich czas działania nie jest na samym poczatku ograniczony z dolu wielkością zadanego modelu, lecz występującą w nim liczbą zmiennych decyzyjnych, a tych w omawianym przypadku jest niewiele — $|E|$) [4].

4.1.3 Przepływy

Odmiennym podejściem do problemu minimalnego drzewa rozpinającego jest jego zdefiniowanie za pomocą **przepływów** [19, 38–44]. W tej interpretacji węzłami są miejsca, w których dozwolony jest skład towarów, zaś krawędzie między nimi symbolizują ścieżki, którymi dane towary możemy przemieszczać pomiędzy dwoma sąsiednimi punktami, ponosząc tego koszty proporcjonalne do wagi łączących ich krawędzi. Założymy, że mamy dany graf G , taki jak przedstawiono na rysunku 4.3. Dodatkowo niech jeden z węzłów w grafie (v_1) będzie wyróżniony i posiada tyle jednostek towarów ile w grafie jest wierzchołków. Naszym celem jest dystrybucja wszystkich towarów z wykorzystaniem dostępnych krawędzi w taki sposób, aby każdy z transportów dotarł w efekcie końcowym do innego wierzchołka — biorąc pod uwagę, że życzymy sobie, aby przy okazji poniesć tego jak najmniejsze koszty, okaże się że droga, która zostanie w ten sposób wytyczona z punktu startowego v_1 do wszystkich $v_j \in V \setminus \{v_1\}$, jest minimalnym drzewem rozpinającym ten graf, a fakt rozdzielenia towarów pomiędzy wszystkie węzły w oczywisty sposób świadczy o spójności tak wygenerowanego drzewa.



Rysunek 4.3: Rysunek ilustrujący wygenerowany przepływ dla problemu znalezienia minimalnego drzewa rozpinającego, gdzie szarymi ciągłymi liniami zostało zaznaczone optymalne rozwiązanie problemu, przerywanymi — pozostałe krawędzie w grafie, zaś czarnymi oznaczone są łuki, które do tej pory były wykorzystane do transportu towarów. (a) Sytuacja początkowa z wszystkimi pięcioma elementami zgromadzonymi przy węźle v_1 . (b) Z wierzchołka v_1 została przekazana piątka elementów z wykorzystaniem krawędzi e_1 . (c) Z wierzchołkiem v_2 połączone są jeszcze dwie krawędzie należące do minimalnego drzewa rozpinającego danego grafu przy tak zdefiniowanych kosztach — wykorzystując krawędzie e_3 oraz e_4 , do nowych wierzchołków zostały w sumie przemieszczone cztery elementy, pozostawiając jeden w wierzchołku v_2 . (d) W wyniku rozdysybuowania wszystkich elementów tak, aby każdy wierzchołek posiadał dokładnie jeden spośród nich, otrzymano minimalne drzewo rozpinające dla prezentowanego grafu.

Niech teraz z każdą krawędzią e_{ij} ($e_{ij} \equiv v_i \xrightarrow{1} v_j \wedge v_j \xrightarrow{1} v_i$) będą związane dwie dodatkowe zmienne: f_{ij} oraz f_{ji} , symbolizujące przepływ (ang. *flow*) na tej krawędzi w danym kierunku (wartość zmiennej f_{ij} oznacza więc liczbę elementów, które z wierzchołka v_i przesunieliśmy do v_j przy wykorzystaniu krawędzi e_{ij} , zaś zmienna f_{ji} symbolizuje odwrotną zależność — liczbę transportowanych elementów po tym samym łuku, lecz z v_j do v_i). Tak jak to sobie powiedzieliśmy i jak mogliśmy zaobserwować na rysunku 4.3a, naszym celem jest wyjście z sytuacji, gdzie w jednym wierzchołku znajduje się $n = |V|$ elementów, i zakończenie w takiej, w której do każdego wierzchołka przyporządkowany jest jeden z nich — innymi słowy chcemy, aby z wyróżnionego wierzchołka wysłanych zostało $n - 1$ posiadanych przez niego elementów, przy jednoczesnym upewnieniu się, że żaden z nich do niego nie wróci. W równaniu 4.5b możemy wyróżnić dwie sumy, które kolejno oznaczają:

- $\sum_{(i,j) \in E} f_{ij}$ — sumę wartości wszystkich przepływów, których kierunek jest wyznaczony w stronę wierzchołków innych niż v_i (mówimy wtedy o sumie przepływów wyjściowych wierzchołka v_i),
- $\sum_{(j,i) \in E} f_{ji}$ — sumę wartości przepływów skierowanych w stronę wierzchołka v_i (sumę przepływów przychodzących do danego wierzchołka).

Innymi słowy wartość pierwszego wyrażenia będzie równa liczbie ładunków, które wysłaliśmy z wierzchołka v_i do pozostałych węzłów, zaś wartość drugiej sumy oznacza liczbę tych ładunków, które do wierzchołka v_i dotarły. Widzimy zatem, że wartość wyrażenia z równania 4.5b dla parametru $i = 1$ odpowiada

opisanej przez nas sytuacji — wierzchołek początkowy wysyła w kierunku pozostałych wierzchołków łącznie $n - 1$ ładunków (sobie pozostawiając jeden — $\sum_{(1,j) \in E} f_{1j} = n - 1$), zaś sam nie przyjmuje żadnych — $\sum_{(j,1) \in E} f_{j1} = 0$. Dla wszystkich pozostałych przypadków, tak jak to mogliśmy zaobserwować na rysunkach 4.3a–4.3d, każdy wierzchołek, z otrzymanych od sąsiada elementów, wybierał dla siebie jeden a resztę przekazywał dalej, co natychmiast prowadzi do równości $\sum_{(j,i) \in E} f_{ji} - \sum_{(i,j) \in E} f_{ij} = 1$ (i oczywiście $\sum_{(i,j) \in E} f_{ij} - \sum_{(j,i) \in E} f_{ji} = -1$, gdyż w takiej formie najwygodniej jest nam zastosować tą równość w prezentowanym modelu 4.5 [25, 35]).

$$\min \quad \sum_{e \in E} c_e \cdot x_e, \quad (4.5a)$$

$$\text{s.t.} \quad \sum_{(i,j) \in E} f_{ij} - \sum_{(j,i) \in E} f_{ji} = \begin{cases} n-1 & \text{jeżeli } i = 1, \\ -1 & \text{w przeciwnym przypadku,} \end{cases} \quad (4.5b)$$

$$f_{ij} \leq (n-1) \cdot x_{ij}, \quad \forall \{i,j\} \in E, \quad (4.5c)$$

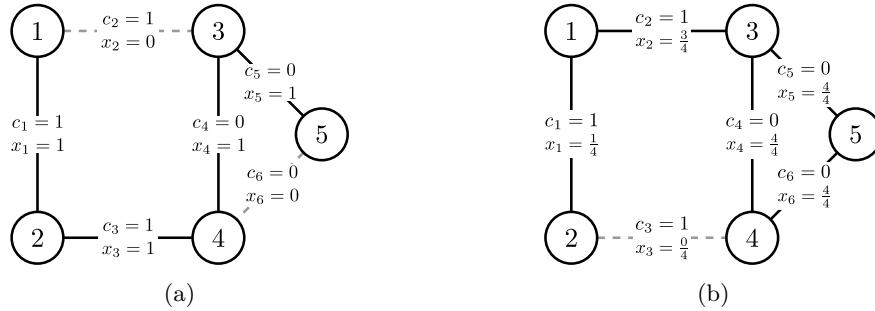
$$f_{ji} \leq (n-1) \cdot x_{ij}, \quad \forall \{i,j\} \in E, \quad (4.5d)$$

$$\sum_{e \in E} x_e = n-1, \quad (4.5e)$$

$$f_e \geq 0, \quad \forall e \in E, \quad (4.5f)$$

$$x_e \in \{0, 1\}, \quad \forall e \in E. \quad (4.5g)$$

Reszta ograniczeń w modelu 4.5 wydaje się być intuicyjna — poprzez ograniczenia takie jak 4.5c czy 4.5d gwarantujemy sobie, że w przypadku krawędzi e , należącej do minimalnego drzewa rozpinającego ($x_e = 1$), liczba przesyłanych przezeń elementów wynosić może maksymalnie $n - 1$ (czyli tyle, ile przesyłałby wierzchołek początkowy, gdyby posiadał tylko jednego sąsiada), w przeciwnym zaś razie wykorzystanie tej krawędzi jest zabronione ($x_{ij} = 0 \rightarrow f_{ij} = 0$). Dodatkowo posiadamy ograniczenie, które determinuje liczbę krawędzi tworzących rozwiązanie, oraz — wreszcie — ograniczenia wartości samych zmiennych decyzyjnych. Te ostatnie widzimy, że ograniczają ich dziedzinę do jedynie dwóch wartości (0 oraz 1), co wskazuje na to, że przedstawiony w 4.5 model opisuje problem programowania całkowitoliczbowego, w związku z czym oczekiwany czas otrzymania rozwiązania tak postawionego problemu, przy wykorzystaniu naszego modelu 4.5, nie jest w żaden sposób ograniczony funkcją wielomianową od wielkości problemu. Możemy oczywiście zignorować ograniczenia całkowitoliczbowości, licząc na prawidłowe zachowanie się modelu bez tego ograniczenia — choć brzmi to niedorzecznie, w rzeczywistości (np. dla grafu i kosztów zaprezentowanych na rysunkach 4.3a–4.3d) może się okazać, że otrzymamy identyczne rozwiązanie co w przypadku zastosowania ograniczeń na całkowitoliczbowość, co może sugerować, że tak zbudowany model również jest poprawny.



Rysunek 4.4: Optymalne rozwiązania dla modelu przedstawionego w 4.5. (a) Minimalne drzewo rozpinające o całkowitym koszcie 2, zwrócone przez model opisany ograniczeniami 4.5a–4.5g. Wartości zmiennych decyzyjnych $\{x_i : e_i \in E\}$ są ograniczone do wartości 0 (krawędź nie należy do rozwiązania) albo 1. (b) Rozwiązanie nie będące drzewem rozpinającym, zwrócone przez model opisany ograniczeniami 4.5a–4.5f (gdzie dodatkowo $\forall e_i \in E : x_i \geq 0$).



Niestety, jak widzimy na rysunkach 4.4a oraz 4.4b [19, 41], nawet w najprostszych przypadkach zniesienie wspomnianego ograniczenia może spowodować, że zwracane rozwiązania będą niedopuszczalne w myśl problemu minimalnego drzewa rozpinającego (choć oczywiście wszystkie ograniczenia tak zbudowanego modelu są spełnione — po prostu nie modeluje on już więcej problemu, którego rozwiązania byśmy oczekiwali [19, 39]). W związku z tym będziemy chcieli zaproponować trzeci wariant rozwiązania, który w znacznym stopniu opiera się na idei powyższego modelu, jednak w odróżnieniu od swojego poprzednika, optymalne rozwiązania przez niego zwracane będą poprawne nawet, jeżeli zaniechamy ograniczenia na całkowitoliczbowość zmiennych decyzyjnych $\{x_i : e_i \in E\}$.

4.1.4 Przepływy z całkowitymi punktami ekstremalnymi

W tej części będziemy chcieli pokazać wariacje na temat poprzednio prezentowanego modelu (patrz: model 4.5), która pozwoli nam na stwierdzenie, że tak zbudowany model, jego rozwiązania (punkty ekstremalne [22]) są liczbami całkowitymi [19, 42–46], bez względu na to, jakie ograniczenia zostaną nałożone na zmienne decyzyjne tego modelu. To z kolei pozwoli nam na zdefiniowanie dziedzin wszystkich zmiennych decyzyjnych jako nieograniczonego zakresu liczb rzeczywistych większych od zera — z omówionych wcześniej własności problemu modelowania wiemy zaś, że w takim przypadku mamy do czynienia z modelem programowania liniowego, rozwiązywalnego w czasie wielomianowym (a to jest naszym celem — pokazania modelu, dla którego rozwiązanie otrzymamy właśnie w takim czasie). Zanim jednak przedstawimy pełen model (4.6), zaproponowany w [19, 42–46], omówimy kolejno znaczenie wszystkich najistotniejszych jego ograniczeń, które wymagają pochylenia się nad nimi.

Wielotowarowy model przepływu dla grafów skierowanych

Zasadniczą różnicą pomiędzy tym (ang. *directed multicommodity flow model*) a poprzednio przedstawionym modelem jest odmienny sposób organizacji przepływu pomiędzy wierzchołkami grafu. Spójrzmy na rysunki 4.5a–4.5d — przedstawiają one niemal tę samą sytuację co na rysunku 4.3, lecz skupiają się one jedynie na pojedynczym elemencie do przetransportowania, w odróżnieniu od przytoczonego przykładu.

$$\min \sum_{(i,j) \in E} c_{ij} \cdot (y_{ij} + y_{ji}), \quad (4.6a)$$

$$\text{s.t. } \sum_{(j,s) \in E} f_{js}^k - \sum_{(s,j) \in E} f_{sj}^k = -1, \quad \forall k \in V \setminus \{v_s\}, \quad (4.6b)$$

$$\sum_{(j,i) \in E} f_{ji}^k - \sum_{(i,j) \in E} f_{ij}^k = 0, \quad \forall i, k \in V \setminus \{v_s\} \wedge i \neq k, \quad (4.6c)$$

$$\sum_{(j,k) \in E} f_{jk}^k - \sum_{(k,j) \in E} f_{kj}^k = 1, \quad \forall k \in V \setminus \{v_s\}, \quad (4.6d)$$

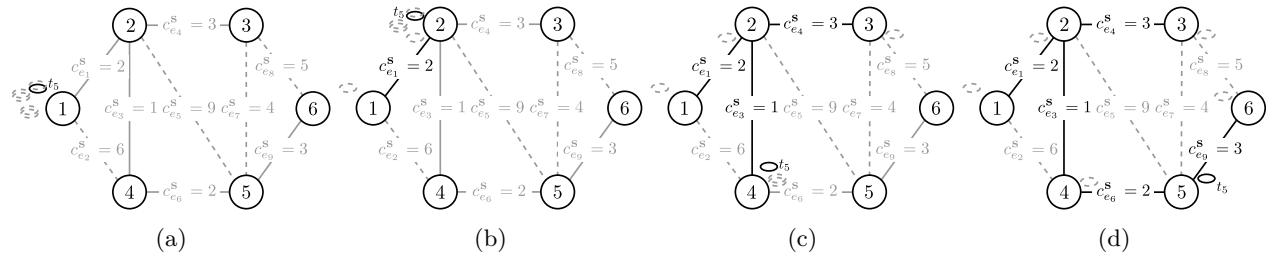
$$f_{ij}^k \leq y_{ij}, \quad \forall (i,j) \in E \wedge \forall k \in V \setminus \{v_s\}, \quad (4.6e)$$

$$\sum_{(i,j) \in E} y_{ij} = n - 1, \quad (4.6f)$$

$$f_{ij} \geq 0, \quad \forall (i,j) \in E, \quad (4.6g)$$

$$y_{ij} \geq 0, \quad \forall (i,j) \in E. \quad (4.6h)$$

Na wspomnianych rysunkach widzimy zatem wyraźnie sposób konstrukcji ograniczeń 4.6b, 4.6c oraz 4.6d. Niech zmienne decyzyjne f_{ij} oznaczają przepływ tak jak w poprzednim modelu. Wprowadźmy teraz nowe zmienne f_{ij}^k , które wykorzystamy do konstrukcji modelu 4.6. Niech wyrażenie f_{ij}^k oznacza przepływ skierowany pomiędzy wierzchołkami v_i oraz v_j dla towaru t_k ($v_i \xrightarrow{1,t_k} v_j$), gdzie $i, j, k \in \{1, \dots, n\}$ (oczywiście $\sum_{k=1}^n f_{ij}^k = f_{ij}$). Suma przepływów wchodzących do wierzchołka początkowego równa się 0, jako że to właśnie w nim na początku znajdują się wszystkie towary, które wierzchołek tylko przekazuje dalej. Jasnym



Rysunek 4.5: Przepływ towarów z perspektywy wyróżnionego elementu t_5 . (a) Sytuacja początkowa: wierzchołkiem startowym jest wierzchołek $v_1 = v_s$, końcowym dla wyróżnionego towaru — v_5 . (b) Towar t_5 został przesunięty w kierunku następnego wierzchołka. Suma przepływów wychodzących dla danego towaru z wierzchołka startowego wynosi 1. Wszystkie wierzchołki pośrednie (nie będące ani początkowym, ani końcowym dla branego pod uwagę towaru), które otrzymują towar t_5 w czasie jego drogi do v_5 , otrzymawszy go, (c) natychmiast przesyłają go dalej — bilans przepływu dla takiego wierzchołka względem towaru t_5 wynosi 0. (d) Suma przepływów wychodzących oraz wchodzących do wierzchołka będącego odbiorcą swojego towaru wynosi 1 — jeśli wierzchołek nie jest wierzchołkiem początkowym, akceptuje on tylko przeznaczony dla siebie element, resztę zaś przesyła dalej w chwili ich otrzymania.

jest, że w przypadku rozpatrywania każdego z towarów osobno, mamy $\sum_{(s,j) \in E} f_{sj}^k = 1$ (gdy $j \neq s$, gdyż towar t_s z założenia znajduje się już na swoim miejscu). Analogicznie wszystkie wierzchołki v_i , nie będące wierzchołkiem początkowym, oczekują na przyjęcie tylko jednego towaru (t_i), toteż ich bilans przepływów dla konkretnego towaru t_i także sumuje się do jedynki. Dla wszystkich pozostałych wierzchołków, które nie są ani wierzchołkiem początkowym, ani docelowym węzłem danego towaru t_i , suma przepływów wchodzących do danego wierzchołka i z niego wychodzących dla towaru t_i powinna być taka sama. Własności te są zachowane dla wszystkich towarów, poza tym znajdującym się w wierzchołku początkowym (t_s). Oczywiście jest również, że w przypadku, gdy dana krawędź nie należy do rozwiązania, przepływ dla tej krawędzi i dowolnego towaru nie występuje ($y_{ij} = 0 \rightarrow \forall f_{ij}^k = 0$), zaś ograniczenie 4.6f redukuje nam przestrzeń dopuszczalnych przez model rozwiązań do drzew rozpinających zadany graf. Dzięki funkcji celu spośród nich wybierane jest zaś rozwiązanie optymalne — minimalne drzewo rozpinające. Warto zwrócić uwagę na fakt, że do problemu, który zdefiniowaliśmy tylko dla grafów nieskierowanych⁵, wykorzystujemy model, który rozróżnia kierunki przepływu na krawędziach (który jest przystosowany do grafów skierowanych). Wiąże się to z koniecznością zastosowania funkcji celu, która za wartość zmiennej decyzyjnej x_{ij} (z poprzedniego modelu) przyjmuje sumę zmiennych y_{ij} oraz y_{ji} — dzięki tak prostemu zabiegowi otrzymujemy wyrażenie, którego wartość jest niezależna od kierunku przepływów i może być traktowana jako wartość rozwiązania problemu minimalnego drzewa rozpinającego dla grafu nieskierowanego.

4.2 Oporne minimalne drzewo rozpinające

Naszym następnym naturalnym krokiem, po zapoznaniu się z podstawowymi schematami modelującymi problem minimalnego drzewa rozpinającego, jest postawienie pytania o modele, z których pomocą moglibyśmy rozwiązywać bardziej złożone zagadnienia, które są oparte o wyżej omówiony problem. W tej części postaramy się zatem skonstruować modele rozwiązujące problem minimalnego drzewa rozpinającego w wersji INCREMENTAL, a następnie wykorzystamy uzyskane rezultaty do sformułowania twierdzeń, które będą dla nas punktem wyjściowym do stworzenia „tradycyjnego”, iteracyjnego algorytmu dla tego problemu.

Przypomnijmy, że problem minimalnego drzewa rozpinającego w wersji INCREMENTAL (IMST) stawał przed nami zadanie odnalezienia takiego drzewa T^* (w obliczu pojawiennia się nowych kosztów krawędzi w grafie), którego część wspólną ze starym rozwiązaniem jest dostatecznie duża (jest regulowana przez parametr

⁵ Problem minimalnego drzewa rozpinającego dla grafów skierowanych nosi nazwę optymalnego rozgałęziania (ang. *optimum branching*) i jest także rozwiązywalny w czasie porównywalnym do czasu działania algorytmu Prima dla grafów nieskierowanych [23].



k , należącego do obowiązkowego opisu problemu). Zatem, tak jak zdefiniowaliśmy to w 3.4.1, naszym celem jest znalezienie drzewa T^* , które spełnia

$$v(T^*, \mathbf{s}') = \min_{T' \in \mathcal{T}_{T_s^*}^k} v(T', \mathbf{s}'), \quad (4.7)$$

gdzie za T_s^* przyjmujemy drzewo będące minimalnym drzewem rozpinającym grafu G , którego koszty krawędzi definiowane są przez scenariusz \mathbf{s} , $\mathcal{T}_{T_s^*}^k$ jest zbiorem zbiorów krawędzi (drzew), zdefiniowanym jako $\{T : f(T, T_s^*) \leq k\}$ (gdzie $f : X \times X \rightarrow \mathbb{N}$ była zdefiniowana jako $f(T^0, T^1) = |T^0 \setminus T^1|$ ⁶), zaś my szukamy pośród tego zbioru takiego drzewa T^* , którego suma kosztów krawędzi dla nowego scenariusza \mathbf{s}' będzie jak najmniejsza. Przyjmując, że pierwsze rozwiązanie problemu minimalnego drzewa rozpinającego T_s^* opisują zmienne decyzyjne x_{ij} ($T_s^* = \{e_{ij} : x_{ij} = 1\}$), nowego zaś — y_{ij} ($T_{\mathbf{s}'}^* = \{e_{ij} : y_{ij} = 1\}$), to bez głębszego zastanowienia możemy napisać następujący model:

$$\min \sum_{e \in E} c_e \cdot y_e, \quad (4.8a)$$

$$\text{s.t. } A \cdot \mathbf{y} = \mathbf{b}, \quad (4.8b)$$

$$\sum_{e \in E} |x_e - y_e| \leq 2 \cdot k, \quad (4.8c)$$

$$y_e \geq 0, \quad \forall e \in E, \quad (4.8d)$$

gdzie równanie 4.8b symbolizuje dowolny układ ograniczeń służący nam w poprzednim rozdziale do modelowania problemu minimalnego drzewa rozpinającego. Nierówność 4.8c w naturalny sposób opisuje dodatkowe ograniczenie w problemie typu INCREMENTAL — jesteśmy zainteresowani rozwiązaniami, które nie zawierają więcej niż k krawędzi, których nie ma w rozwiązaniu bazowym ($x_{ij} = 0 \wedge y_{ij} = 1 \rightarrow |x_e - y_e| = 1$)⁷. Przyglądając się tak zdefiniowanemu modelowi, od razu możemy dostrzec pierwszy poważny problem — funkcji $g(x_{ij}, y_{ij}) = |x_{ij} - y_{ij}|$ nie jesteśmy w stanie przedstawić za pomocą dostępnych nam narzędzi, gdyż w zależności od wartości zmiennych decyzyjnych, zmianom ulega nasza macierz współczynników dla modelu (gdy $x_{ij} \geq y_{ij}$, współczynnikami dla tego ograniczenia są $[1, -1]$, zaś w przypadku gdy $x_{ij} < y_{ij}$ — sytuacja jest odwrotna), co oczywiście nie może mieć miejsca (nie możemy uzależniać struktury modelu od jego własnego rozwiązania). Aby rozwiązać ten problem wprowadźmy dodatkowe zmienne decyzyjne:

- z_{ij}^+ — dla każdej krawędzi jej wartość wynosi 1, gdy dana krawędź $e_{ij} \in T_{\mathbf{s}'}^*$ ($y_{ij} = 1$), lecz $e_{ij} \notin T_s^*$ ($x_{ij} = 0$). W przeciwnym wypadku $z_{ij}^+ = 0$. Wartość tak zdefiniowanej zmiennej zatem będzie udzielać informacji czy dana krawędź nie występowała w oryginalnym rozwiązaniu i została dodana do nowego rozwiązania w wyniku zmiany kosztów w grafie,
- z_{ij}^- — analogicznie do wyżej zdefiniowanej zmiennej wartość tej wskazuje na krawędzi, które zostały usunięte z rozwiązania na rzecz tych, dla których $z_{ij}^+ = 1$ (w przypadku, gdy dana krawędź $e_{ij} \notin T_{\mathbf{s}'}^*$, ale $e_{ij} \in T_s^*$ — $z_{ij}^- = 1$),

które mają następujące własności:

- jeśli dana krawędź e_{ij} należy do pierwotnego rozwiązania (dla starych kosztów grafu), wtedy $z_{ij}^+ = 0$ (nie możemy ponownie dodać do rozwiązania tego samego łuku — 4.9h),
- jeśli krawędź e_{ij} nie należy do drzewa T_s^* , wtedy $z_{ij}^- = 0$ (analogicznie nie możemy z rozwiązania usunąć krawędzi, której w nim nie ma — 4.9i),
- nowa zmienna decyzyjna, generująca nam optymalne rozwiązanie problemu IMST ($T^* = \{e_{ij} : z_{ij} = 1\}$), jest zależna od obu tych zmiennych i przybiera formę $z_e = z_e^+ - z_e^- + x_e$.

⁶Zbiorem X w tym przypadku będziemy oznaczać przestrzeń wszystkich możliwych drzew rozpinających graf.

⁷Weźmy pod uwagę, że skoro oba rozwiązania zawierają z definicji tą samą liczbę krawędzi, to fakt pojawiения się łuku należącego tylko do jednego z nich pociąga za sobą fakt pojawiения się w drugim rozaniu krawędzi, która nie jest częścią rozwiązania pierwszego. Innymi słowy jesteśmy zmuszeni do podwojenia wartości parametru k ($x_{ij} = 0 \wedge y_{ij} = 1 \rightarrow \exists e_{kl} \in E \ x_{kl} = 1 \wedge y_{kl} = 0$).

Widzimy, że w łatwy sposób możemy związać ze sobą informacje o zmianach w zbiorze krawędzi będących rozwiążaniem, z informacjami niesionymi przez wektor zmiennych \mathbf{x} — do modelu przedstawionego w 4.5 dołączymy dodatkowe ograniczenia: 4.9f, 4.9g, 4.9h oraz 4.9i a także za wektor zmiennych, wyznaczających rozwiązanie dla modelu, przyjmemy \mathbf{z} (wektor \mathbf{x} w tym przypadku symbolizuje stare rozwiązanie). Jak wskazaliśmy wyżej, jedynym miejscem, w którym wiążemy nowe zależności ze starym modelem, jest zbiór równości 4.9g (gdybyśmy je usunęli, otrzymywane rozwiązania dla tego modelu niczym by się nie różniły od tych otrzymywanych dla 4.5 — ograniczenia od 4.9f do 4.9i po prostu nie miałyby na rozwiązanie żadnego wpływu). Bardzo łatwo też można dostrzec, że jeżeli $\forall e_{ij} \in E z_{ij}^+ = 0 \wedge z_{ij}^- = 0$, to otrzymany model niczym nie różni się od przedstawionego wyżej 4.5 — wartości zmiennych decyzyjnych z_{ij} będą po prostu przerównane do starych wartości a model bez zastanowienia powinien zwrócić nam identyczne rozwiązanie. W zależności od parametru k dla problemu IMST wiele ograniczeń zajdzie właśnie w takiej formie — $z_{ij} = 0 - 0 + x_{ij}$ (jeśli k jest małe w stosunku do liczby krawędzi w grafie). Ogólnie $\sum_{e \in E} z_e^+ = \sum_{e \in E} z_e^- = k$.

$$\min \sum_{e \in E} c_e \cdot z_e, \quad (4.9a)$$

$$\text{s.t. } \sum_{(i,j) \in E} f_{ij} - \sum_{(j,i) \in E} f_{ji} = \begin{cases} n-1 & \text{jeżeli } i=1, \\ -1 & \text{w przeciwnym przypadku,} \end{cases} \quad (4.9b)$$

$$f_{ij} \leq (n-1) \cdot z_{ij}, \quad \forall \{i,j\} \in E, \quad (4.9c)$$

$$f_{ji} \leq (n-1) \cdot z_{ij}, \quad \forall \{i,j\} \in E, \quad (4.9d)$$

$$\sum_{e \in E} z_e = n-1, \quad (4.9e)$$

$$\sum_{e \in E} z_e^+ + z_e^- \leq 2 \cdot k, \quad (4.9f)$$

$$z_e = z_e^+ - z_e^- + x_e, \quad \forall e \in E, \quad (4.9g)$$

$$z_e^+ \leq 1 - x_e, \quad \forall e \in E, \quad (4.9h)$$

$$z_e^- \leq x_e, \quad \forall e \in E, \quad (4.9i)$$

$$f_e, z_e^+, z_e^- \geq 0, \quad \forall e \in E, \quad (4.9j)$$

$$z_e \in \{0, 1\}, \quad \forall e \in E. \quad (4.9k)$$

Wracając zaś do problemu wyrażenia wartości bezwzględnej w modelu programowania LP/MIP, widzimy, że dla tak zdefiniowanych zmiennych zachodzi $|x_{ij} - y_{ij}| = z_{ij}^+ - z_{ij}^-$ — wystarczy zauważyc, że w przypadku gdy $x_{ij} < y_{ij}$ bądź $y_{ij} < x_{ij}$ (odpowiednio $e_{ij} \notin T_s^*$ i $e_{ij} \in T_s^*$ lub $e_{ij} \notin T_s^*$ i $e_{ij} \in T_s^*$ — krawędź e_{ij} należy tylko do pierwszego bądź tylko do drugiego rozwiązania), suma zmiennych $z_{e_{ij}}^+$ oraz $z_{e_{ij}}^-$ wynosi dokładnie 1 — dzięki ograniczeniom 4.9h oraz 4.9i mamy pewność, że gdy jedna z tych zmiennych równa się 1, w tym czasie wartość drugiej jest z góry ograniczona przez 0 ($z_{e_{ij}}^+$ i $z_{e_{ij}}^-$ są zmiennymi o wartościach nieujemnych). Dodatkowo aby któraś z nierówności ($x_{e_{ij}} < y_{e_{ij}}$ lub $y_{e_{ij}} < x_{e_{ij}}$) mogła zajść, wartość którejkolwiek ze zmiennych $z_{e_{ij}}^+$ lub $z_{e_{ij}}^-$ musi się różnić od zera — zakładając zaś, że wartości zmiennych $x_{ij} \in \mathbf{x}$ ze starego rozwiązania należą do przedziału $\{0, 1\}$, oraz że domeny zmiennych należących do wektora \mathbf{z} wymuszają te same zakresy dopuszczalnych wartości (4.9k), dochodzimy do wniosku, że jedynymi dopuszczalnymi w takim wypadku wartościami zmiennych $z_{e_{ij}}^+$ lub $z_{e_{ij}}^-$ są te należące do $\{0, 1\}$ ⁸. Oczywiście, gdy $|x_e - y_e| = 0$, $x_e = y_e$, co pociąga za sobą $z_e^+ = z_e^- = 0$.

Możemy całkowicie zrezygnować z ograniczenia 4.9g, pozbywając się w ten sposób nadmiaru zmiennych z modelu oraz znacznie redukując liczbę potrzebnych ograniczeń, tak jak to pokazano poniżej (4.11), gdzie w miejsce zmiennych z_e dla każdej krawędzi pojawiły się jego podwyrażenia z poprzedniego modelu —

⁸Dla przykładu: jeśli $z_{ij} = 1$, $x_{ij} = 0$, to wiemy, że $z_{e_{ij}}^+ - z_{e_{ij}}^- = 1$, zaś z ograniczenia 4.9i wiemy dodatkowo, że $z_{e_{ij}}^- \leq 0$, co razem z 4.9j ($z_{e_{ij}}^- \geq 0$) doprowadza nas do jedynego słusznego wniosku, że $z_{e_{ij}}^+ = 0$.



$\sum_{e \in E} c_e \cdot x_e + \sum_{e \in E} c_e \cdot (z_e^+ - z_e^-) = \sum_{e \in E} c_e \cdot z_{ij}$, gdzie z_{ij} to zmienne odnoszące się do poprzedniego modelu (4.9g). Równość 4.11b także pokrywa się z równością 4.8b dla modelu 4.8:

$$A \cdot \mathbf{x} = b \wedge A \cdot \mathbf{z} = b \leftrightarrow A \cdot (\mathbf{z}^+ - \mathbf{z}^- + \mathbf{x}) = b \leftrightarrow A \cdot (\mathbf{z}^+ - \mathbf{z}^-) = \mathbf{0}, \quad (4.10)$$

zaś pozostałe ograniczenia wyrażają te same zależności co wcześniej.

$$\min \sum_{e \in E} c_e \cdot x_e + \sum_{e \in E} c_e \cdot (z_e^+ - z_e^-), \quad (4.11a)$$

$$\text{s.t. } A \cdot (z_e^+ - z_e^-) = \mathbf{0}, \quad (4.11b)$$

$$\sum_{e \in E} z_e^+ + z_e^- \leq 2 \cdot k, \quad (4.11c)$$

$$z_e^- \leq x_e, \quad \forall e \in E, \quad (4.11d)$$

$$z_e^+, z_e^- \geq 0, \quad \forall e \in E. \quad (4.11e)$$

4.2.1 Relaksacja Lagrange'a

W celu efektywnego rozwiązywania modeli programowania liniowego/całkowitoliczbowego, możemy posłużyć się jeszcze jednym bardzo ciekawym narzędziem, jakim jest **relaksacja** ograniczeń. Przyglądając się modelowi zdefiniowanemu w 4.8 (przytoczymy go poniżej)

$$\min \sum_{e \in E} c_e \cdot y_e, \quad (4.12a)$$

$$\text{s.t. } A \cdot \mathbf{y} = \mathbf{b}, \quad (4.12b)$$

$$\sum_{e \in E} |x_e - y_e| \leq 2 \cdot k, \quad (4.12c)$$

$$y_e \geq 0, \quad \forall e \in E, \quad (4.12d)$$

możemy zauważyć, że gdyby nie obecność ograniczenia 4.12c, prezentowany model niczym nie różniłby się od dowolnego modelu rozwiązywającego problem minimalnego drzewa rozpinającego (byłyby dokładnie jego uogólnionym zapisem — jak powiedzieliśmy przy omawianiu rzeczonego modelu, równość 4.12b symbolizuje ograniczenia dla tego problemu). Naturalnym pytaniem zatem jest: „Czy możemy za pomocą posiadanego już modelu rozwiązywać inne, podobne do siebie problemy?”. Zanim odpowiemy na to pytanie, przekształćmy jeszcze model 4.12 do najprostszej, najbardziej wymownej postaci jaką możemy uzyskać — zauważmy, że zgodnie z definicją funkcji $f : X \times X \rightarrow \mathbb{N}$, jej wartość zależy tylko od dwóch zbiorów krawędzi (w naszym przypadku od drzew T_s^* oraz T^* , gdzie pierwszym z nich jest optymalne (minimalne) drzewo rozpinające dla grafu z kosztami krawędzi zgodnymi ze scenariuszem początkowym s , drugie zaś jest optymalnym rozwiązaniem problemu IMST). Skoro zaś $f(T_s^*, T^*) = |T^* \setminus T_s^*|$ (oraz $f(T_s^*, T^*) = |T_s^* \setminus T^*|^9$), możemy dojść do wniosku, że wyliczanie wartości $|x_e - y_e|$ dla wszystkich krawędzi nie jest najszybszym sposobem narzucenia ograniczeń zgodnie z wymogami problemu minimalnego drzewa rozpinającego w wersji INCREMENTAL. Szybko możemy zauważyć, że tak naprawdę jesteśmy zainteresowani tylko wartościami zmiennych decyzyjnych dla dowolnego z podanych zbiorów krawędzi: $T^* \setminus T_s^*$ lub $T_s^* \setminus T^*$, a dokładniej ich sumą:

$$\sum_{e \in E} |x_e - y_e| \leq 2 \cdot k \leftrightarrow \sum_{e_i \in E \setminus T_s^*} z_i \leq k \leftrightarrow \sum_{e_i \in T^* \setminus T_s^*} z_i \leq k.^{10} \quad (4.13)$$

⁹Zatem kolejność argumentów funkcji $f : X \times X \rightarrow \mathbb{N}$ nie ma żadnego znaczenia.

¹⁰Zauważliśmy tutaj, że sumy $\sum_{e_i \in E \setminus T_s^*} z_i$ oraz $\sum_{e_i \in T^* \setminus T_s^*} z_i$ są identyczne — wynika to z prostego faktu, że dla wszystkich krawędzi $e \in E \setminus T^*$ $z_i = 0$.

Przyjrzyjmy się zatem następującemu modelowi¹¹ [26, 588]:

$$\min \sum_{e \in E} c_e \cdot x_e, \quad (4.14a)$$

$$\text{s.t. } A \cdot \mathbf{x} = \mathbf{b}, \quad (4.14b)$$

$$\sum_{e \in T^* \setminus T_s^*} x_e \leq k, \quad (4.14c)$$

$$x_e \geq 0, \forall e \in E, \quad (4.14d)$$

a następnie dokonajmy **relaksacji** jego ograniczeń — innymi słowy usuniemy z niego ograniczenie 4.14c (tak aby powstały w ten sposób model był identyczny, co model rozwiążający problem minimalnego drzewa rozpinającego) a następnie dodamy to ograniczenie do wartości funkcji celu:

$$\min \sum_{e \in E} c_e \cdot x_e + \lambda \cdot \left(\sum_{e \in T^* \setminus T_s^*} x_e - k \right), \quad (4.15a)$$

$$\text{s.t. } A \cdot \mathbf{x} = \mathbf{b}, \quad (4.15b)$$

$$x_e \geq 0, \quad \forall e \in E, \quad (4.15c)$$

gdzie λ jest parametrem. Idea stojąca za takim postępowaniem jest bardzo prosta. Założymy, że ograniczenie 4.14c nie zaszło (czyli $\sum_{e \in T^* \setminus T_s^*} x_e > k$) — sami na to zezwoliliśmy, usuwając dane ograniczenie z modelu. Bezpośrednią konsekwencją niespełnienia danego ograniczenia jest wzrost wartości rozwiązania, jaką zwróci nam tak zmodyfikowana funkcja celu — zwracana wartość będzie tym większa, im bardziej będziemy oddalać się od przestrzeni rozwiązań dopuszczalnych ($\sum_{e \in T^* \setminus T_s^*} x_e$ będzie coraz bardziej oddalone od k) lub im większą wartość parametru λ przyjmiemy. Naszą motywacją w tak podjętym działaniu jest naturalne wymuszenie na modelu odrzucania niedopuszczalnych rozwiązań (ściślej mówiąc — traktowania ich jako rozwiązanie nieoptymalne, dla których funkcja celu nie zwraca najmniejszej wartości spośród wszystkich dostępnych rozwiązań) przy jednoczesnym zachowaniu jego uproszczonej struktury. Pokażemy teraz słuszność takiego postępowania w ogólnym przypadku.

Lemat 4.2.1 [1, 607] Niech dany będzie problem optymalizacyjny \mathcal{P} , do którego zastosowano relaksację Lagrange'a, relaksując ograniczenia w postaci $A \cdot \mathbf{x} \leq b$. Niech \mathbf{x}^* będzie optymalnym rozwiązaniem takiego problemu dla pewnego wektora $\boldsymbol{\lambda} \geq \mathbf{0}$. Jeśli \mathbf{x}^* jest rozwiązaniem dopuszczalnym w \mathcal{P} i spełnia założenia warunku komplementarności $\boldsymbol{\lambda} \cdot (A \cdot \mathbf{x}^* - b) = \mathbf{0}$, wtedy rozwiązanie \mathbf{x}^* jest rozwiązaniem optymalnym dla problemu \mathcal{P} .

W naszym przypadku, jak można było zauważyć w modelu 4.15, użyty przez nas parametr λ nie jest wektorem (zwykliście je oznaczać pogrubioną czcionką), co ma swoje uzasadnienie w tym, że wartość, przez którą go mnożymy, też nim nie jest (usunięte z modelu 4.14 ograniczenie nie jest aggregatorem dla większej liczby ograniczeń — tyczy się pojedynczej sumy). Naszym celem zatem jest pokazać, że o ile model 4.15 zwróci rozwiązanie spełniające podane w lematce kryteria, będzie ono szukanym przez nas rozwiązaniem optymalnym, co pozwoli nam się następnie skupić na sposobie obliczania samego parametru λ dla minimalizującej funkcji celu.

Dowód. Niech dany będzie model w postaci macierzowej:

$$v^* = \min \mathbf{c} \cdot \mathbf{x}, \quad (4.16a)$$

$$\text{s.t. } A \cdot \mathbf{x} = \mathbf{b}, \quad (4.16b)$$

$$x_i \in X, \quad (4.16c)$$

¹¹Jako że nie będziemy już rozróżniać zmiennych z wektorów \mathbf{x} , \mathbf{y} i \mathbf{z} (odpowiednio jako zmienne wskazujące krawędzie należące do zbiorów T_s^* , $T_{s'}^*$ — dla problemu MST — oraz T^* — dla problemu IMST), w dalszej części będziemy na powrót posługiwać się oznaczeniami \mathbf{x} , by podkreślić odrębność omawianego zagadnienia.



którego postać po relaksacji Lagrange'a jest następująca:

$$L(\boldsymbol{\lambda}) = \min_{\mathbf{x} \in X} \mathbf{c} \cdot \mathbf{x} + \boldsymbol{\lambda} \cdot (A \cdot \mathbf{x} - \mathbf{b}), \quad (4.17a)$$

$$\text{s.t. } x_i \in X. \quad (4.17b)$$

Zauważmy oczywisty fakt, że zbiory opisane przez 4.16 oraz 4.17 łączy wzajemna relacja zawierania się, to jest zbiór rozwiązań opisany przez $\{\mathbf{x} : A \cdot \mathbf{x} = \mathbf{b}\} \subseteq \{\mathbf{x}\}$ (drugi model nie narzuca nam żadnych ograniczeń co do rozwiązania). Wiemy zatem, że pomiędzy wartościami dla rozwiązań \mathbf{x}^* a $L(\boldsymbol{\lambda})$ zachodzi prawidłowość:

$$v^* = \min_{\mathbf{x} \in X} \{\mathbf{c} \cdot \mathbf{x} : A \cdot \mathbf{x} = \mathbf{b}\} \geq \min_{\mathbf{x} \in X} \{\mathbf{c} \cdot \mathbf{x} + \boldsymbol{\lambda} \cdot (A \cdot \mathbf{x} - \mathbf{b})\} = L(\boldsymbol{\lambda})^{12}. \quad (4.18)$$

Możemy zatem o wartości $L(\boldsymbol{\lambda})$ powiedzieć, że jest dolnym ograniczeniem wartości v^* , jako że zawsze $L(\boldsymbol{\lambda}) \leq v^*$. Wartość wyrażenia v^* jest najmniejsza spośród wszystkich innych rozwiązań ($v^* = \mathbf{c} \cdot \mathbf{x}^*$, gdzie \mathbf{x}^* jest optymalnym rozwiązaniem problemu opisanego przez model 4.16), zatem $L(\boldsymbol{\lambda})$ — jeśli jest mniejsza od v^* — jest rozwiązaniem na pewno niedopuszczalnym. Co nas w związku z tym powinno interesować, to jak najokładniejsze dolne ograniczenie na wartość v^* : $L^* = \max_{\boldsymbol{\lambda}} L(\boldsymbol{\lambda})$, które spełnia $L(\boldsymbol{\lambda}) \leq L^* \leq v^*$. Dodatkowo z faktu optymalności rozwiązania \mathbf{x}^* wiemy, że dla dowolnego $\mathbf{x} \in X$ zachodzi także $L(\boldsymbol{\lambda}) \leq L^* \leq v^* = \mathbf{c} \cdot \mathbf{x}^* \leq \mathbf{c} \cdot \mathbf{x}$. Założymy teraz, że rozwiązanie $\mathbf{x} \in X$ spełnia warunek komplementarności, to znaczy $\boldsymbol{\lambda} \cdot (A \cdot \mathbf{x} - \mathbf{b}) = \mathbf{0}$, w związku z tym wyrażenie $L(\boldsymbol{\lambda})$ upraszcza się do $\mathbf{c} \cdot \mathbf{x}$. Jeśli, zgodnie z lematem, przedstawione rozwiązanie \mathbf{x} o takich własnościach jest dopuszczalne, to znaczy, że jego wartość na pewno nie jest mniejsza od wartości rozwiązania reprezentowanego przez wektor zmiennych decyzyjnych \mathbf{x}^* . W związku z tym mamy, że:

$$L(\boldsymbol{\lambda}) = \mathbf{c} \cdot \mathbf{x} + \boldsymbol{\lambda} \cdot (A \cdot \mathbf{x} - \mathbf{b}) = \mathbf{c} \cdot \mathbf{x}, \quad (4.19)$$

$$L(\boldsymbol{\lambda}) \leq L^* \leq v^* = \mathbf{c} \cdot \mathbf{x}^* \leq \mathbf{c} \cdot \mathbf{x} = L(\boldsymbol{\lambda}). \quad (4.20)$$

Z powyższego zaś wynika, że drugi z podanych wzorów przedstawia szereg równości, tak więc ostatecznie otrzymujemy, że $L(\boldsymbol{\lambda}) = v^* = \mathbf{c} \cdot \mathbf{x}$, gdzie $\mathbf{x} = \mathbf{x}^*$. ♦

Powyższy lemat jest bardzo potężnym narzędziem — okazuje się bowiem, że dzięki spełnieniu warunku komplementarności możemy bezkarnie uprościć nasz model rozwiązywający problem minimalnego drzewa rozpinającego w wersji INCREMENTAL¹³. W naszym przypadku zastosowanie tej metody jest bardzo proste, gdyż liczba relaksowanych przez nas ograniczeń to zaledwie 1 — w związku z tym przedstawimy teraz cały proces transformacji modelu rozwiązywającego problem IMST do prostszego, radzącego sobie z problemem MST, zwracającego zaś te same rozwiązania.

Minimalne drzewo rozpinające w wersji INCREMENTAL

W modelu 4.15 w wyniku relaksacji ograniczeń otrzymaliśmy następujące wyrażenie:

$$\sum_{e \in E} c_e \cdot x_e + \lambda \cdot \left(\sum_{e \in T^* \setminus T_s^*} x_e - k \right) .. \quad (4.21)$$

Aby móc je uprościć, by miało dla nas większą wartość merytoryczną, musimy pozbyć się z niego wyrażenia T^* , które definiuje nam zbiór krawędzi będących elementami minimalnego drzewa rozpinającego w wersji INCREMENTAL (w funkcji celu, pozwalającej nam na odnalezienie optymalnego rozwiązania problemu IMST, pojawia nam się samo jego rozwiązanie). Aby wyeliminować tą zależność między powyższym wzorem a rozwiązaniem T^* , spójrzmy jeszcze raz na ograniczenie, które wciążliśmy do wyrażenia funkcji celu dla modelu. Możemy zauważyć, że

¹²Optymalne rozwiązanie (o najmniejszej wartości) może znajdować się w zbiorze większym, zaś nie należeć do zbioru będącego tylko jego podzbiorem.

¹³Powstało bardzo dużo prac naukowych na temat użyteczności zaprezentowanego lematu i polach do zastosowania relaksacji Lagrange'a, których nie będziemy tu przytaczać — głównym polem, na którym eksplotowana jest ta metoda, jest problem transformacji modeli programowania IP/MIP do postaci LP tak, aby bez pogorszenia stopnia użyteczności danego modelu, móc uzyskiwać rozwiązania problemów w rozsądny, wielomianowym czasie.



$$\sum_{e \in T^* \setminus T_s^*} x_e \leq k \rightarrow \sum_{e \in T_s^*} x_e \geq n - 1 - k. \quad (4.22)$$

Powyzsza implikacja wynika z faktu, że suma zmiennych x_i dla dowolnego drzewa rozpinającego graf o n wierzchołkach jest równa co najwyżej $n - 1$ (dla zmiennych binarnych), zatem jeżeli liczba krawędzi e należących do takiego rozwiązania, gdzie $e \notin T_s^*$ nie przekracza k , to pozostały krawędzi w rozwiązaniu ($e \in T_s^*$) musi być tyle, aby suma ich liczebności dawała $n - 1$. Bardziej formalnie:

$$k \geq \sum_{e \in T^* \setminus T_s^*} x_e \leftrightarrow |e : e \in T^* \setminus T_s^* \wedge x_e = 1| \leftrightarrow \quad (4.23)$$

$$\leftrightarrow |e : e \in T^* \wedge x_e = 1| - |e : e \in T_s^* \wedge x_e = 1| \quad (4.24)$$

$$|e : e \in T^* \wedge x_e = 1| = n - 1 \quad (4.25)$$

$$\sum_{e \in T_s^*} x_e = |e : e \in T_s^* \wedge x_e = 1| \leftrightarrow \quad (4.26)$$

$$\leftrightarrow |e : e \in T^* \wedge x_e = 1| - |e : e \in T^* \setminus T_s^* \wedge x_e = 1| \geq (n - 1) - k. \quad (4.27)$$

Korzystając zatem z równoważności powyższych warunków, możemy przekształcić równanie 4.21 w następujący sposób:

$$\sum_{e \in E} c_e \cdot x_e + \lambda \cdot \left(\sum_{e \in T^* \setminus T_s^*} x_e - k \right) \leftrightarrow \quad (4.28)$$

$$\leftrightarrow \sum_{e_i \in E} c_i \cdot x_i + \lambda \cdot \left((n - 1 - k) - \sum_{e_i \in T_s^*} x_i \right) \stackrel{(1)}{\leftrightarrow} \quad (4.29)$$

$$\stackrel{(1)}{\leftrightarrow} \left[\sum_{e_i \in E \setminus T_s^*} c_i \cdot x_i + \sum_{e_i \in T_s^*} c_i \cdot x_i \right] + \lambda \cdot \left(- \sum_{e_i \in T_s^*} x_i \right) \leftrightarrow \quad (4.30)$$

$$\leftrightarrow \sum_{e_i \in E \setminus T_s^*} c_i \cdot x_i + \sum_{e_i \in T_s^*} c_i \cdot x_i - \sum_{e_i \in T_s^*} \lambda \cdot x_i \leftrightarrow \quad (4.31)$$

$$\leftrightarrow \sum_{e_i \in E \setminus T_s^*} c_i \cdot x_i + \sum_{e_i \in T_s^*} (c_i - \lambda) \cdot x_i, \quad (4.32)$$

gdzie w kroku oznaczonym jako (1) pozbyliśmy się stałego wyrażenia $\lambda \cdot (n - 1 - k)$, które nie wpływa w żaden sposób na rozwiązanie (dla każdego wektora \mathbf{x} , który możemy podstawić do otrzymanego wzoru, relatywna wartość funkcji dla rozwiązań \mathbf{x} i \mathbf{x}' się nie zmienia), oraz rozobiliśmy sumę zmiennych decyzyjnych dla zbioru e na dwie części.

4.3 Podsumowanie rozdziału

Otrzymaliśmy zatem dokładny przepis na rozwiązanie problemu minimalnego drzewa rozpinającego w wersji INCREMENTAL. Tak jak zapowiadaliśmy na początku tego rozdziału, w kolejnym wykorzystamy zgromadzone informacje do implementacji algorytmu nie polegającego na zagadnieniu programowania całkowito-liczbowego. Wykorzystując podane przez nas warunki optymalności rozwiązania:

$$f(T^*, T_s^*) \leq k \wedge \lambda = 0 \text{ lub} \quad (4.33)$$

$$f(T^*, T_s^*) = k \wedge \lambda \neq 0, \quad (4.34)$$



oraz policzone wyżej wyrażenie, będące definicją funkcji celu dla zrelaksowanego problemu IMST, podamy algorytm, który następnie posłuży nam do rozwiązania kolejnych, bardziej złożonych problemów: adwersarza (dalej AIMST) oraz odpornego drzewa rozpinającego w wersji INCREMENTAL z możliwością poprawy (RRIMST). W tym rozdziale natomiast mieliśmy okazję zapoznać się z terminem liniowego programowania, poznać jego zalety, wady jak i ograniczenia. Przygotowaliśmy dla siebie naprawdę potężne narzędzia do pracy z modelami i wykorzystaliśmy je, by otrzymać przedstawione powyżej wyniki.





Odporna optymalizacja przyrostowa

W tym rozdziale skonstruujemy efektywny algorytm rozwiązujący jedno z zagadnień odpornej optymalizacji dyskretnej — problem minimalnego drzewa rozpinającego dla okresowo zmieniających się kosztów z ograniczoną możliwością modyfikacji pierwotnego rozwiązania (problem INCREMENTAL MINIMUM SPANNING TREE). Mając w ręku odpowiednie narzędzia przekonamy się, że konstrukcja takiego algorytmu jest prosta — w oparciu o wcześniej przedstawione pomysły będziemy mogli sprowadzić całe zagadnienie do prostego wyszukiwania binarnego, pokazać słuszność takiego postępowania oraz sposoby na jego udoskonalenie.

5.1 Konstrukcja algorytmu i warunki optymalności

Konstruowanie algorytmu, rozwiązującego zagadnienie typu INCREMENTAL dla minimalnego drzewa rozpinającego, rozpoczęmy od powtórzenia wyniku 4.32, otrzymanego w poprzednim rozdziale po zastosowaniu relaksacji Lagrange'a dla modelu 4.8:

$$\min \left\{ \sum_{e_i \in E \setminus T_s^*} c_i \cdot x_i + \sum_{e_i \in T_s^*} (c_i - \lambda) \cdot x_i \right\}, \quad (5.1)$$

gdzie przypomnijmy, że T_s^* oznaczał zbiór krawędzi należący do optymalnego rozwiązania problemu minimalnego drzewa rozpinającego dla starych kosztów w grafie, zdefiniowanych przez scenariusz s .

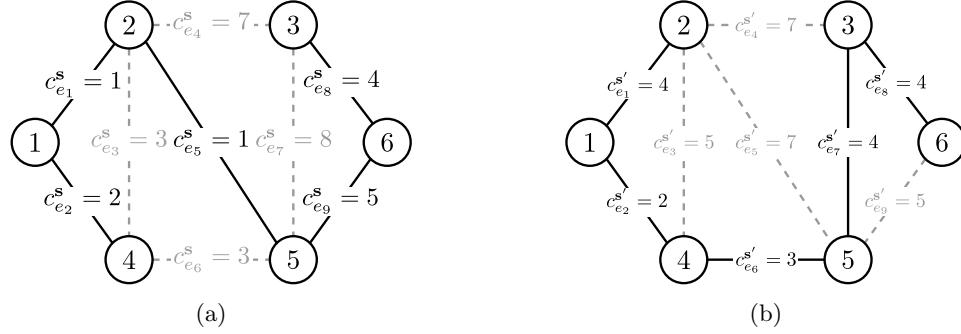
Pokazaliśmy, że dla problemu minimalnego drzewa rozpinającego w wersji INCREMENTAL z początkowym minimalnym drzewem rozpinającym T_s^* i parametrem k , przedstawiającym liczbę dopuszczalnych zmian krawędzi względem pierwotnego rozwiązania, w wyniku zmiany kosztów w grafie, z twierdzenia o różnicach dopełniających zastosowanego do tak opisanego problemu (4.3), wynika, że optymalnym jego rozwiązaniem jest zbiór krawędzi, który zawiera dokładnie k łuków nie należących do początkowego rozwiązania T_s^* . Otrzymany wynik jest zgodny z naszą intuicją — niech $N(T, k)$ oznacza zbiór wszystkich drzew rozpinających różniących się od drzewa T co najwyżej k krawędziami, w szczególności $N(T, 0) = \{T\}$. Niech dane będzie $k' < k$. Nietrudno zauważyc, że $N(T, k') \supseteq N(T, k)$, zatem jeżeli optymalne rozwiązanie problemu IMST T^* znajduje się w zbiorze $N(T, k')$, tym bardziej zawiera się w zbiorze większym — $N(T, k)$. Odwrotna relacja oczywiście nie zachodzi — rozwiązanie należące do zbioru $N(T, k)$, niekoniecznie musi należeć także do zbioru mniejszego. Aby zatem mieć pewność, że znajdziemy rozwiązanie optymalne dla problemu IMST z parametrem k , musimy szukać go wśród drzew rozpinających graf o takiej samej liczbie nowych krawędzi względem starego rozwiązania (innymi słowy, co wydaje się oczywiste, wśród wszystkich możliwych, dopuszczalnych rozwiązań problemu, a te należą do zbioru $N(T, k)$ — rozwiązania, które charakteryzują się większą liczbą łuków niż k są niedopuszczalne). Opierając się zatem o wcześniej przypomniany wzór 5.1, możemy zauważyc, że rozsądna strategią na szukanie optymalnego rozwiązania dla badanego zagadnienia, jest taka manipulacja parametrem λ , aby koszty grafu, definiowane przez scenariusz s' :

$$c_i^{s'(\lambda)} = \begin{cases} c_i^{s'} & \text{gdy } e_i \text{ nie należy do } T_s^*, \\ c_i^{s'} - \lambda & \text{gdy } e_i \text{ jest krawędzią należącą do oryginalnego rozwiązania,} \end{cases} \quad (5.2)$$

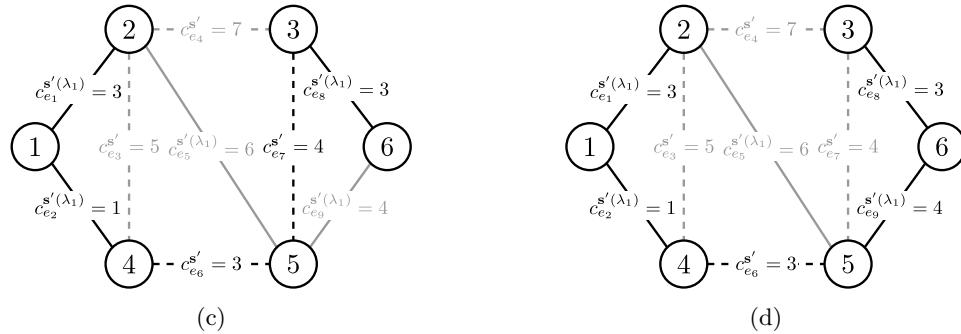
stopniowo wymuszały na algorytmie rozwiązującym problem minimalnego drzewa rozpinającego rozwiązania, których część wspólna ze starym rozwiązaniem jest coraz większa. Innymi słowy zmniejszając koszty poszczególnych krawędzi w grafie, podnosimy ich szansę na to, że znajdą się w zbiorze krawędzi minimalnego drzewa



rozpinającego, jako że ich koszty w pewnym momencie staną się mniejsze od wag tych łuków, które do tej pory trafiały do tego zbioru. Taką sytuację (oraz problemy związane z takim podejściem) doskonale odzwierciedlają rysunki od 5.1a do 5.1f. Przypomnijmy tylko, że liniami ciągłymi dla problemu IMST oznaczaliśmy oryginalne minimalne drzewo rozpinające T_s^* , zaś kolorem czarnym — aktualnie przedstawiane na rysunku rozwiązanie (to znaczy, że czarną linią przerywaną są zaznaczone te krawędzie, które w wyniku zmiany kosztów krawędzi w grafie weszły do rozwiązania problemu IMST T^* i nie należą do T_s^* , zaś szara linia ciągła oznacza krawędź, która z tych samych przyczyn nie należy do znalezionej rozwiązania, jest natomiast elementem zbioru T_s^*). Pozwoli nam to na łatwe stwierdzenie, czy przedstawiane rozwiązanie jest dopuszczalne, optymalne, czy nie należące do zbioru dopuszczalnych rozwiązań (odpowiednio $f(T^*, T_s^*) = |T^* \setminus T_s^*| = |T_s^* \setminus T^*| \leq k$, $f(T^*, T_s^*) = k$, $f(T^*, T_s^*) > k$).



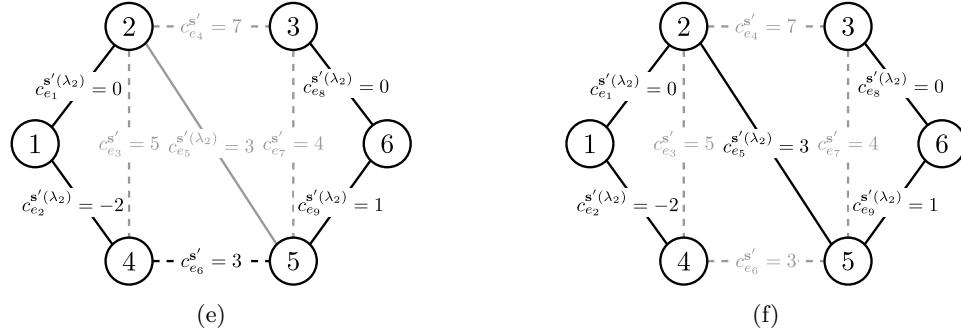
Rysunek 5.1: (a) Optymalne rozwiązanie problemu MST dla scenariusza $s = [1, 2, 3, 7, 1, 3, 8, 4, 5]$. Całkowity jego koszt wynosi 13. Krawędzie do niego należące tworzą początkowe rozwiązanie dla problemu IMST z parametrem $k = 1$: $T_s^* = \{e_1, e_2, e_5, e_8, e_9\}$. (b) Nieograniczone parametrem k , optymalne rozwiązanie problemu IMST dla scenariusza $s' = [4, 2, 5, 7, 7, 3, 4, 4, 5]$, gdzie pochyloną czcionką zaznaczono wagi krawędzi, które uległy zmianie względem scenariusza s . Wartość takiego rozwiązania wynosi 17 i jest tym samym rozwiązaniem, jakie otrzymalibyśmy dla scenariusza $s'(\lambda)$ w przypadku, gdy $\lambda = 0$. Należące do tego rozwiązania krawędzie tworzą zbiór $T_{s'}^* = \{e_1, e_2, e_6, e_7, e_8\}$, gdzie krawędzie e_6 i e_7 nie należą do T_s^* ($f(T_{s'}^*, T_s^*) = 2 > k$).



Rysunek 5.1: (c) Rozwiązanie optymalne problemu MST dla wygenerowanego scenariusza $s'(\lambda_1) = [3, 1, 5, 7, 6, 3, 4, 3, 4]$, gdzie pogrubioną czcionką zaznaczono, obniżone o współczynnik $\lambda_1 = 1$ względem scenariusza s' i rozwiązania początkowego T_s^* , koszty na podstawie których algorytm rozwiązujący problem minimalnego drzewa rozpinającego zwrócił nam przedstawione rozwiązanie $T^1 = \{e_1, e_2, e_6, e_7, e_8\}$. Fikcyjny koszt takiego rozwiązania wynosi 14 (dla rzeczywistych kosztów, tych definiowanych przez scenariusz s' , jego wartość to 18). (d) Inne optymalne rozwiązanie dla scenariusza $s'(\lambda_1)$.

Po przyjrzeniu się bliżej rysunkom 5.1c oraz 5.1d, zauważamy pierwszy poważny problem dotyczący obranej strategii — algorytm do rozwiązywania problemu minimalnego drzewa rozpinającego (w tym przypadku do wygenerowania rysunków celowo zastosowaliśmy algorytm Prima) ma skłonność do zwracania różnych odpowiedzi. Co gorsza, w kontekście problemu IMST rozwiązania te nie są takie same, pomimo że ich koszt jest identyczny. Zauważmy, że o ile w przypadku 5.1c dla scenariusza $s'(1)$ zostało zwrócone identyczne

rozwiązań co na rysunku 5.1b (niedopuszczalne, jako że zawierające więcej niż jedną krawędź nienależącą do T_s^*), to już rozwiązanie zwrócone na rysunku 5.1d jest rozwiązaniem dopuszcjalnym (co więcej, na mocy wcześniejszych lematów jest ono rozwiązaniem optymalnym). Skąd wzięła się ta różnica? Jeśli wróćmy do, przedstawionego w pierwszym rozdziale, pseudokodu algorytmu Prima (Pseudokod 2), zobaczymy, że rzeczywisty algorytm jako parametr wejściowy przyjmuje arbitralnie wybrany przez nas węzeł początkowy. Jako że w przypadku klasycznego problemu minimalnego drzewa rozpinającego jesteśmy zainteresowani jedynie sumą kosztów krawędzi zwracanego rozwiązania, nie ma dla nas różnicy od którego węzła algorytm zacznie jego konstruowanie (zwykle po prostu jest to pierwszy węzeł, jeśli wierzchołki grafu przechowujemy w dowolny sposób uporządkowanej strukturze danych). Skutkuje to, tak jak pokazano na rysunkach 5.1c i 5.1d, w przypadku których wierzchołkami startowymi są odpowiednio wierzchołki v_1 i v_6 , dwoma różnymi rozwiązaniami. Jakby tego było mało, na rysunkach 5.1e i 5.1f przedstawiono podobną sytuację, w której dla tego samego parametru $\lambda_2 = 4$, znalezione rozwiązaniami są odpowiednio: rozwiązanie optymalne $T^2 = \{e_1, e_2, e_6, e_8, e_9\}$ o rzeczywistym (patrz rysunek 5.1b) koszcie równym 18 oraz dopuszcjalne, o sumie kosztów krawędzi należących do rozwiązania równej 21 (koszty obydwu rozwiązań dla tymczasowego scenariusza $s'(\lambda_2)$ są identyczne i wynoszą 2). Ponownie otrzymaliśmy dwa różne rozwiązania tylko dlatego, że za początkowy wierzchołek obraliśmy v_1 (rysunek 5.1e) oraz v_2 (5.1f).



Rysunek 5.1: (e) Minimalne drzewo rozpinające graf dla parametru $\lambda_2 = 4$ i wygenerowanego na jego podstawie scenariusza $s'(\lambda_2) = [0, -2, 5, 7, 3, 3, 4, 0, 1]$. (f) Inne optymalne rozwiązanie dla scenariusza $s'(\lambda_2)$. Koszt optymalnego rozwiązania wynosi 2.

Widzimy zatem, że tak opisana strategia szukania rozwiązania dla problemu IMST posiada wadę, która czyni ją zupełnie bezużyteczną, jako że typ zwracanego rozwiązania (optymalny, dopuszcjalny bądź niedopuszczalny) nie zależy w pełni od świadomie wybieranego parametru λ , lecz od pośredniczącego algorytmu, wyliczającego minimalne drzewo rozpinające. Należy podkreślić, że problem ten nie dotyczy tylko algorytmu Prima — zdecydowaliśmy się na ten algorytm, gdyż na jego przykładzie najdokładniej widać istotę problemu. Problem ten będzie występował (może występować) zawsze, gdy w grafie, dla którego chcemy znaleźć minimalne drzewo rozpinające, będą występować co najmniej dwie krawędzie o takich samych kosztach, niezależnie od wybranego algorytmu.

5.2 Struktura drzewa i koszty krawędzi

Aby rozwiązać powyższy problem, będziemy chcieli zaburzyć koszty grafu w taki sposób, aby zapewnić sobie unikalność wag wszystkich krawędzi w grafie, nie zaburzając jednocześnie ich kolejności występowania tj. dla grafu $G = (V, E)$ i dla kosztów krawędzi w grafie $s = [c_i : \forall e_i \in E]$, gdzie założymy dodatkowo, że $\exists e, e' \in E : c_e = c_{e'} \wedge e \neq e'$, będziemy chcieli wygenerować nowe koszty $s' = [c'_i : \forall e_i \in E]$ takie, że $\forall e, e' \in E : e \neq e' \implies c'_e \neq c'_{e'}$ a dodatkowo $\forall e, e' \in E : c_e < c_{e'} \implies c'_e < c'_{e'}$. Zanim jednak zdefiniujemy takie przekształcenie, poczytajmy inną obserwację.

Lemat 5.2.1 [26] Niech drzewa T_s^* i $T_{s'}^*$ oznaczają odpowiednio optymalne rozwiązania dla problemu MST i scenariuszy s oraz s' problemu minimalnego drzewa rozpinającego w wersji INCREMENTAL z parametrem k .



Dla dowolnego $T_{s'}^*$ w grafie $G = (V, E)$ istnieje minimalne drzewo rozpinające dla problemu INCREMENTAL T^k takie, że wszystkie jego krawędzie należą do zbioru $E^* = T_{s'}^* \cup T_s^*$.

Z lematu zatem wynika, że aby znaleźć optymalne rozwiązanie dla problemu IMST, możemy zamiast grafu $G = (V, E)$, wziąć pod uwagę dużo rzadszy graf $G^* = (V, E^*)$. Wynika to z faktu, że o ile istnieje optymalne rozwiązanie T^k dla omawianego problemu w grafie G , takie samo rozwiązanie istnieje w G^* , zaś po drodze do rozwiązania ani razu nie będziemy brali pod uwagę krawędzi spoza zbioru E^* (upraszczając, naszym celem jest wyjście od rozwiązania optymalnego $T_{s'}^*$ dla nowego scenariusza s' i stopniowe zamienianie wybranych jego krawędzi na te należące do drzewa T_s^* — nie ma więc mowy, abyśmy potrzebowali krawędzi spoza zbioru $T_{s'}^* \cup T_s^* = E^*$). Ta obserwacja pozwala nam często w znacznym stopniu zredukować strukturę grafu, gdyż w najgorszym przypadku moc nowo otrzymanego zbioru krawędzi będzie wynosić $|E^*| = 2 \cdot n - 2$ (gdy oba zbiorы krawędzi są rozłączne, zaś każdy z nich, z definicji bycia drzewem, zawiera dokładnie $|V| - 1$ krawędzi, gdzie $|V| = n$). Nawet w takim przypadku nasza obserwacja może okazać się wielce pomocna, na przykład gdy mamy do czynienia z grafem pełnym — wtedy możemy mówić o zredukowaniu mocy zbioru krawędzi grafu do co najmniej $\frac{2 \cdot (n-1)}{\binom{n}{2}} = \frac{4}{n}$ części jego oryginalnej liczebności.

Dowód. Niech drzewa T_s^* i $T_{s'}^*$ będą zdefiniowane tak jak w lemacie. Dodatkowo niech T^k będzie optymalnym rozwiązaniem dla problemu minimalnego drzewa rozpinającego w wersji INCREMENTAL, takim że zawiera ono największą liczbę krawędzi ze zbioru E^* spośród pozostałych rozwiązań. Jeśli liczba takich krawędzi w drzewie T^k równa się jego mocy ($|T^k| = n - 1$), to dowód możemy zakończyć, gdyż $\forall e \in T^k \quad (e \in T^k \implies e \in E^*)$, więc $T^k \subseteq E^*$.

Załóżmy zatem, że tak nie jest, czyli że istnieje taka krawędź $e \in T^k$, która nie należy do zbioru E^* — $e \in T^k \setminus E^*$. Aby $T^k \subseteq E^*$, usuńmy z niego tą krawędź. W wyniku usunięcia krawędzi e z drzewa T^k otrzymamy podział na dwa zbiorы krawędzi będące drzewami: S i \bar{S} oraz zbiór $\mathcal{Q}(T^k, e)$ (oznaczenie zdefiniowałyśmy w 2.3 i oznaczało ono zbiór wszystkich takich krawędzi e_{ij} , gdzie $v_i \in S$ oraz $v_j \in \bar{S}$). Skorzystajmy teraz z własności optymalnego cięcia (2.2.1), które mówi, że drzewo jest minimalnym drzewem rozpinającym T wtedy i tylko wtedy, gdy należące do niego krawędzie e' mają najmniejszy koszt spośród wszystkich innych krawędzi, które należą do zbiorów $\mathcal{Q}(T, e')$ (dla każdej krawędzi $e' \in T$). Wybierzmy zatem ze zbioru $\mathcal{Q}(T^k, e)$ krawędź e^* o najmniejszym koszcie w tym zbiorze. Bezpośrednio z powyższej własności wiemy, że wybrana krawędź należy do minimalnego drzewa rozpinającego $T_{s'}^*$. W związku z tym, że usuwana z drzewa T^k krawędź $e \notin E^*$, z definicji $e \notin T_{s'}^* \wedge e \notin T_s^*$. Z faktu zaś, że $e \notin T_{s'}^*$ otrzymujemy natychmiast prosty wniosek, że $e \neq e^*$ ($e \notin T_{s'}^*, e^* \in T_{s'}^*$). W związku z tym nic nie stoi na przeszkodzie, aby z drzewa T^k stworzyć nowe drzewo $T^{k'}$, różne od T^k , poprzez usunięcie krawędzi $e \in T^k \setminus E^*$ a dodanie do niego $e^* \in T_{s'}^* \in E^*$. Zauważmy, że z własności e^* wynika, że suma kosztów krawędzi w drzewie $T^{k'}$ jest co najwyżej równa tej samej wartości dla drzewa T^k . Jako że drzewo T^k było rozwiązaniem optymalnym, nie istnieje drzewo o mniejszej sumie kosztów krawędzi do niego należących. Stąd prosty wniosek, że suma wag łuków w drzewie $T^{k'}$ jest taka sama jak w T^k . Dodatkowo oczywistym jest, że otrzymane drzewo nadal jest rozwiązaniem dopuszczalnym dla problemu IMST. Z założenia optymalności T^k i definicji problemu INCREMENTAL mamy, że $f(T^k, T_s^*) = k$. Usuwana z drzewa T^k krawędź $e \in T^k \setminus E^*$ z założenia nie należy do zbioru $T_{s'}^*$, więc $f(T^k \setminus \{e\}, T_s^*) = k - 1 < k$ (zatem bez względu na dodawaną krawędź e^* , drzewo $T^{k'}$ jest rozwiązaniem dopuszczalnym). Skoro zatem nowe drzewo jest również rozwiązaniem dopuszczalnym i zarazem optymalnym, a przy okazji zawiera o jedną krawędź, wspólną ze zbiorzem E^* , więcej niż drzewo T^k , otrzymaliśmy sprzeczność z założeniem, że T^k zawiera największą liczbę takich krawędzi spośród wszystkich optymalnych rozwiązań, co kończy dowód. ♦

Tym samym od razu możemy zredukować wyprowadzone przez nas równanie 5.1, zauważając, że zbiór $E \setminus T_s^* = (T_s^* \cup T_{s'}^*) \setminus T_s^* = T_{s'}^* \setminus T_s^*$:

$$\min \left\{ \sum_{e_i \in E \setminus T_s^*} c_i \cdot x_i + \sum_{e_i \in T_s^*} (c_i - \lambda) \cdot x_i \right\} = \min \left\{ \sum_{e_i \in T_{s'}^* \setminus T_s^*} c_i \cdot x_i + \sum_{e_i \in T_s^*} (c_i - \lambda) \cdot x_i \right\}. \quad (5.3)$$

Pokażemy teraz jak dla tak zredukowanego zbioru krawędzi (z E do E^*) stworzyć sztuczny scenariusz, w którym wszystkie koszty krawędzi będą unikalne i jednocześnie zachowywały oryginalną relację w stosunku do pozostałych kosztów.

Lemat 5.2.2 [7] Niech dane będą koszty krawędzi c_{e_i} w grafie $G = (V, E)$. Dla zmodyfikowanych kosztów, zdefiniowanych jako $c'_{e_i} = c_{e_i} + \phi(i)$, gdzie $\phi(i) = \frac{m \cdot i^2 + i}{(m+1)^3}$, dla każdego $i \in \{1, \dots, |E|\}$ zachodzi:

$$c_{e_i} < c'_{e_i} < c_{e_i} + 1, \quad (5.4)$$

gdzie $m = |E|$.

Dla tak zdefiniowanych kosztów dodatkowo pokażemy, że dla dowolnych i oraz j , takich że $i \neq j$, zachodzi dodatkowa własność:

$$\forall e_i, e_j \in E \left(i \neq j \implies c'_{e_i} \neq c'_{e_j} \right). \quad (5.5)$$

Dzięki tym dwóm własnościom będziemy mieli pewność, że po nałożeniu na wszystkie krawędzie w grafie takiego wzoru, każda krawędź będzie miała unikalny koszt, zaś po uporządkowaniu krawędzi względem ich kosztów oryginalnych i zaburzonych, otrzymamy dwa identyczne ciągi (w wyniku zaburzenia kosztów, koszt żadnej z krawędzi nie przewyższy ani nie zostanie przewyższony przez koszt innej, której oryginalna waga przewyższała lub była przewyższana przez koszt pierwszej z nich). Taka modyfikacja rozwiąże obydwa problemy, o których wspomnieliśmy przy omawianiu rysunków od 5.1c do 5.1f. Oczywiście takie podejście niesie za sobą pewne ograniczenia — oryginalne koszty krawędzi muszą być całkowitoliczbowe, gdyż w przeciwnym przypadku własność 5.4 nie będzie dawała nam żadnej gwarancji, że dla zmodyfikowanych kosztów będziemy otrzymywać te same rozwiązania problemu drzewa rozpinającego co dla kosztów oryginalnych. Jeżeli zależy nam na zachowaniu zmienoprzecinkowych wag na krawędziach — możemy albo dodatkowo zmodyfikować koszty początkowe, poprzez pomnożenie ich przed odpowiednią potęgą liczby 10, bądź też wprowadzić odgórny porządek na krawędziach, co wiązać się dodatkowo będzie z koniecznością modyfikacji każdej części naszych algorytmów, w których stają one przed wyborem jednej z dwóch lub większej liczby krawędzi o tym samym koszcie.

Dowód. Dowód własności 5.4 sprowadzimy do problemu pokazania, że dla dowolnego parametru i , wartość $\phi(i)$ zawiera się w przedziale otwartym $(0, 1)$. Niech zatem $\phi(i) = \frac{m \cdot i^2 + i}{(m+1)^3}$, gdzie $m = |E|$.

$$\begin{aligned} c_{e_i} &< c'_{e_i} < c_{e_i} + 1, \\ 0 &< (c_{e_i} + \phi(i)) - c_{e_i} < 1, \\ 0 &< \frac{m \cdot i^2 + i}{(m+1)^3} < 1, \\ 0 &< m \cdot i^2 + i < (m+1)^3, \\ 0 &< m \cdot i^2 + i \leq m^3 + m < m^3 + 3 \cdot m^2 + 3 \cdot m + 1 = (m+1)^3. \end{aligned} \quad (5.6)$$

W ostatniej nierówności skorzystaliśmy z faktu, że i oznacza numer porządkowy krawędzi w grafie ($i \in \{1, \dots, m\}$). Pokażemy teraz dodatkową własność 5.5 dla dowolnego $i \neq j$:

$$\begin{aligned} c'_{e_i} &\neq c'_{e_j}, \\ c_{e_i} + \phi(i) &\neq c_{e_j} + \phi(j), \\ c_{e_i} + \frac{m \cdot i^2 + i}{(m+1)^3} &\neq c_{e_j} + \frac{m \cdot j^2 + j}{(m+1)^3}. \end{aligned} \quad (5.7)$$

Jeżeli $c_{e_i} = c_{e_j}$, 5.7 w oczywisty sposób jest spełnione. W przeciwnym przypadku mamy spełniony warunek $c_{e_i} + 1 \leq c_{e_j}$ (zakładając bez straty ogólności, że uporządkowaliśmy koszty rosnąco według numeru porządkowego krawędzi, $i < j$ a kosztami są liczby całkowite) i stosując 5.6 otrzymujemy:

$$c_{e_i} < c_{e_i} + \frac{m \cdot i^2 + i}{(m+1)^3} < c_{e_i} + 1 \leq c_{e_j} < c_{e_j} + \frac{m \cdot j^2 + j}{(m+1)^3}.$$

Pokazaliśmy zatem poprawność jednego ze sposobów na rozwiązanie problemu z nieunikatowymi kosztami krawędzi w grafie, a co za tym idzie — zapewniliśmy jednoznaczność rozwiązań zwracanych przez algorytm



Prima (bądź inny rozwiązujący zagadnienie minimalnego drzewa rozpinającego) dla dowolnego scenariusza i dowolnej wartości parametru λ . Przyglądając się formule 5.1, doszliśmy też do wniosku, że wraz ze stopniowym zwiększeniem wartości tego parametru, otrzymujemy kolejne rozwiązania, którym coraz bliżej do spełnienia warunków, które określiliśmy w poprzednim rozdziale (4.33, 4.34). Pokażemy teraz formalnie, że takie postępowanie istotnie doprowadzi nas do rozwiązania.

Twierdzenie 5.2.1 [26, 589] *Drzewo rozpinające T^* jest unikalnym minimalnym drzewem rozpinającym dla problemu INCREMENTAL wtedy i tylko wtedy, gdy koszty rozpatrywanego grafu G są unikalne¹ oraz istnieje parametr $\lambda \geq 0$, dla którego T^* jest optymalne dla zrelaksowanego problemu oraz:*

$$f(T^*, T_s^*) \leq k \wedge \lambda = 0 \quad \text{lub} \quad (5.8)$$

$$f(T^*, T_s^*) = k \wedge \lambda \neq 0. \quad (5.9)$$

Dowód. Unikalność rozwiązania T^* bezpośrednio wynika z własności 5.5, zaś pierwsza część dowodu (jeśli istnieje $\lambda \geq 0$, dla której T^* jest optymalne dla tak zrelaksowanego problemu w oparciu o ten parametr i spełnione są warunki 5.8 i 5.9, wtedy T^* jest unikalnym minimalnym drzewem rozpinającym dla problemu IMST) jest w zasadzie powtórzeniem twierdzenia o różnicach dopełniających, którego słuszność pokazaliśmy w poprzednim rozdziale (4.3). Pozostaje nam zatem do udowodnienia druga część dowodu (tylko wtedy, gdy istnieje $\lambda \geq 0$, dla której T^* jest optymalne i spełnione są warunki 5.8 i 5.9, T^* jest unikalnym minimalnym drzewem rozpinającym dla problemu IMST). Oczywiście gdy $\lambda = 0$, drzewo $T^* = T_s^*$ jest minimalnym drzewem rozpinającym. Co więcej, jeżeli spełnia nierówność $f(T^*, T_s^*) \leq k$, jest także optymalnym rozwiązaniem dla problemu IMST. Założymy zatem, że tak nie jest i $f(T^*, T_s^*) = k' > k$. Co powinno być już dla nas oczywiste, jeżeli zaczniemy zwiększać wartość parametru λ , równolegle do niego będziemy zmniejszać koszty wszystkich krawędzi $e \in T_s^*$. W pewnym momencie parametr λ osiągnie taką wartość, że koszt pewnej pojedynczej krawędzi e_0 zrówna się z kosztem innego łuku e_0^* , który należy do aktualnego rozwiązania. Niech wartość tą reprezentuje wyrażenie λ_1 , zaś drzewo o kosztach krawędzi zależnych od niego — $T(\lambda_1)$. Skoro $c_{e_0} - \lambda_1 = c_{e_0^*}$, to zamieniając ze sobą krawędzie miejscami ($e_0 \in T_s^* \setminus T(\lambda_1)$ dołączając do rozwiązania, zaś $e_0^* \in T(\lambda_1) \setminus T_s^*$ z niego usuwając), otrzymamy nowy zbiór krawędzi $E'(\lambda_1)$ ², który będzie spełniał równość $f(E'(\lambda_1), T_s^*) = k' - 1$, zaś suma kosztów wszystkich krawędzi w nowym zbiorze będzie równa tej samej sumie dla $T(\lambda_1)$ (z punktu widzenia algorytmu rozwiązywanego problem MST nic więc się nie zmieni). Dodatkowo warto zauważać, że ze względu na własności 5.4 i 5.5, które założyliśmy, że spełniają wszystkie krawędzie w grafie³, taka para krawędzi (e_0, e_0^*) jest unikalna, więc w każdym kroku będziemy wymieniać dokładnie jedną krawędź pomiędzy zbiorami. Powtarzając więc powyżej opisane kroki, podnosząc sukcesywnie wartość parametru λ , otrzymamy zbiór wartości $\lambda_1, \lambda_2, \dots, \lambda_{k'-k}$ oraz odpowiadający im ciąg zbiorów $E'(\lambda_1), E'(\lambda_2), \dots, E'(\lambda_{k'-k})$, gdzie każdy z nich spełnia równość $f(E'(\lambda_i), T_s^*) = k' - i$. To oznacza, że w którymś momencie (dokładnie dla $\lambda_{k'-k}$) otrzymamy $f(E'(\lambda_{k'-k}), T_s^*) = k' - (k' - k) = k$ (widzimy, że funkcja $f : X \times X \rightarrow \mathbb{N}$ jest monotonicznie malejąca), co oznaczać będzie, że spełniliśmy jeden z dwóch opcjonalnych warunków optymalności rozwiązania (5.9). ◆

Mamy zatem pewność, że nasza strategia stopniowego zwiększenia wartości parametru λ i równoległego obniżania kosztów krawędzi, należących do oryginalnego rozwiązania T_s^* , doprowadzi nas do znalezienia optymalnego rozwiązania problemu minimalnego drzewa rozpinającego w wersji INCREMENTAL z parametrem k . Pozostaje nam zatem już tylko kwestia odpowiedniego dobierania wartości wspomnianego parametru, jako że do tej pory milcząco zakładaliśmy, że zawsze jesteśmy w stanie znaleźć odpowiednie jego wartości (dla których każde kolejne rozwiązanie będzie zawierało malejącą liczbę krawędzi $e \notin T_s^*$, aż do momentu, gdy ich liczba wyniesie dokładnie k). Pierwszym naiwnym pomysłem jest oczywiście zadanie pewnego interwału $\Delta > 0$ i rozpoczęcie generowania kolejnych wartości parametru λ , począwszy od λ_{\min} , kończąc na pewnym

¹Na przykład niech spełniają własności 5.4 i 5.5.

²Trzeba tutaj zaznaczyć, że operacja wymiany krawędzi między zbiorami nic nie wspomina o własnościach dodawanego łuku względem starego — nie jesteśmy zatem w stanie powiedzieć, czy dodawana krawędź zapewni nam to, że nowy zbiór będzie nadal drzewem (np. należy do zbioru $\mathcal{Q}(T^*(\lambda_1), e_0)$, gdzie e_0 to krawędź usuwana, gdzie w takim przypadku na pewno otrzymalibyśmy nowe drzewo). Aby zapewnić sobie, że nowy zbiór będzie nadal drzewem, powinniśmy wartość parametru λ_1 (i każdy następny) podniść jeszcze wyżej (np. o ϵ), a następnie poprosić algorytm dla problemu MST o nowe rozwiązanie.

³Graf może zawierać już tylko łuki $E = T_{s'}^* \cup T_s^*$.



λ_{\max} (gdzie dla przykładu λ_{\min} może równać się zeru⁴, zaś za λ_{\max} możemy przyjąć chociażby koszt krawędzi w grafie o największej wadze)⁵. Wtedy będziemy generować kolejne drzewa: $T(\lambda_{\min})$, $T(\lambda_{\min} + \Delta)$, \dots , $T(\lambda_{\min} + i \cdot \Delta) = T^*$, gdzie po i iteracjach możemy natknąć się na pierwsze optymalne rozwiązanie T^* dla badanego problemu. Problemy, które natychmiastowo pojawiają się przy takim rozwiązyaniu, są dwa. Nie potrafimy określić jakiego rzędu wielkości powinien być parametr Δ , by przypadkiem nie „przeskoczyć” odpowiednich wartości λ . Z drugiej strony, zmniejszanie tego pierwszego bardzo szybko zwiększa nam liczbę minimalnych drzew rozpinających, które musielibyśmy policzyć, co z kolei przekłada się negatywnie na czas działania takiego algorytmu.

5.3 Binarne poszukiwanie rozwiązania

Pierwszą próbą udoskonalenia przedstawionego pomysłu jest skorzystanie z własności funkcji f , która zwraca liczbę krawędzi będących częścią jednego z drzew, będącego parametrem tej funkcji, zaś nie należących do drzewa przekazanego jako jej drugi parametr. W dowodzie twierdzenia 5.2.1 pokazaliśmy, że wspomniana funkcja jest funkcją monotoniczną — dla dwóch różnych parametrów λ_i, λ_j , gdzie $\lambda_i < \lambda_j$ (niech $i < j$), zachodzi $f(T(\lambda_i), T_s^*) \geq f(T(\lambda_j), T_s^*)$. W naszym przypadku $f(T(\lambda_i), T_s^*) \geq f(T(\lambda_i + \Delta), T_s^*)$, zaś w przytoczonych dowodach zakładaliśmy, że każdy kolejny parametr λ spełniał $f(T(\lambda_i), T_s^*) > f(T(\lambda_j), T_s^*)$. Fakt takiego zachowania się funkcji f prowadzi nas do pomysłu zastosowania, dla badanego problemu, wyszukiwania binarnego — mając ciąg parametrów $\lambda_{\min} = \lambda_0 < \lambda_1 < \dots < \lambda_{i-1} < \lambda_i < \lambda_{i+1} < \dots < \lambda_{\max}$ (i odpowiadający im ciąg wartości funkcji $f(T(\lambda_0), T_s^*) \geq f(T(\lambda_1), T_s^*) \geq \dots \geq f(T(\lambda_{i-1}), T_s^*) \geq f(T(\lambda_i), T_s^*) \geq f(T(\lambda_{i+1}), T_s^*) \geq \dots \geq f(T(\lambda_{\max}), T_s^*)$) łatwo zauważać, że:

- jeśli $f(T(\lambda_i), T_s^*) > k$, wtedy dla każdego drzewa $T(\lambda_j)$, gdzie $j < i$ ($\lambda_j < \lambda_i$), zachodzić będzie $f(T(\lambda_j), T_s^*) > k$. Zatem z góry możemy powiedzieć, że minimalne drzewa rozpinające, liczne dla wszystkich scenariuszy $\mathcal{S} = \{s'(\lambda_j) : j \leq i\}$, okażą się rozwiązaniami, które nie są nawet dopuszczalne.
- Jeśli $f(T(\lambda_i), T_s^*) < k$, wtedy dla każdego drzewa $T(\lambda_j)$, gdzie $j > i$ ($\lambda_j > \lambda_i$), zachodzić będzie $f(T(\lambda_j), T_s^*) < k$. Zatem ponownie możemy powiedzieć, że minimalne drzewa rozpinające, tym razem liczne dla wszystkich scenariuszy $\mathcal{S} = \{s'(\lambda_j) : j \geq i\}$, będą rozwiązaniami dopuszczalnymi, lecz nie optymalnymi (ze względu na niespełnienie warunków twierdzenia 5.2.1).
- Jeśli $f(T(\lambda_i), T_s^*) = k$, to znalezliśmy rozwiązanie optymalne.

Dzięki takiemu podejściu, natychmiastowo redukujemy liczbę wymaganych obliczeń do logarytmu z tej liczby (tak jak to pokazano w pseudokodzie 3). Nadal jednak nie mamy żadnej gwarancji, że dobrana wartość parametru pomocniczego Δ zapewni nam prawidłowe działanie naszego algorytmu — może się okazać, że podążając ścieżką wytyczoną przez algorytm 3, dojdziemy do takiego momentu, w którym w rozpatrywanym przedziale $[\lambda_l, \lambda_l + \Delta, \dots, \lambda_h - \Delta, \lambda_h]$ nie ma takiej wartości parametru λ_i^* , dla której otrzymalibyśmy rozwiązanie optymalne (takiego, że $f(T(\lambda_i^*), T_s^*) = k$), chociaż zgodnie z algorytmem są spełnione warunki: $f(T(\lambda_l), T_s^*) \geq k$ oraz $f(T(\lambda_h), T_s^*) \leq k$. Częściowym rozwiązaniem tego problemu jest pozwolenie algorytmowi w takiej sytuacji na porzucenie dyskretnego zbioru parametrów i samodzielne rozpoczęcie generowania nowych, będących średnimi arytmetycznymi wartości skrajnych kolejnych przedziałów. Takie rozwiązanie gwarantuje nam zatrzymanie się algorytmu w chwili znalezienia optymalnego rozwiązania $T(\lambda^*)$, lecz w żaden sposób nie określa liczby kroków, jakie wykona nasz algorytm zanim na to rozwiązanie „wpadnie”.

Aby rozwiązać i ten problem, przedstawimy dodatkowy lemat, będący uzupełnieniem poprzednich dwóch (5.4 i 5.5).

Lemat 5.3.1 [7] *Dla danych kosztów krawędzi c_{e_i} w grafie $G = (V, E)$, dla zmodyfikowanych kosztów,*

⁴Wtedy pierwsze wygenerowane drzewo $T(\lambda_{\min})$ odpowiadać będzie oczywiście drzewu T_s^* .

⁵Jeśli dla zadanych danych problem minimalnego drzewa rozpinającego w wersji INCREMENTAL ma rozwiązanie, najprawdopodobniej przerwiemy obliczenia poniżej wartości λ_{\max} , jednakże w przeciwnym przypadku musimy takie górne ograniczenie przyjąć.



zdefiniowanych jako $c'_{e_i} = c_{e_i} + \phi(i)$, gdzie $\phi(i) = \frac{m \cdot i^2 + i}{(m+1)^3}$, dla każdego $i \in \{1, \dots, |E|\}$ zachodzi:

$$c_{e_i} < c'_{e_i} < c_{e_i} + 1 \text{ oraz} \quad (5.10)$$

$$\forall (i, j, k, l : i \neq j \wedge i \neq k) \quad c'_{e_i} - c'_{e_j} \neq c'_{e_k} - c'_{e_l} \quad (5.11)$$

gdzie $m = |E|$.

Będziemy chcieli skorzystać z powyższego lematu w celu stworzenia następującego zbioru parametrów: $\Lambda = \left\{ c'_{e_i} - c'_{e_j} : e_i \in T_s^* \setminus T_{s'}^*, e_j \in T_{s'}^* \setminus T_s^* \right\}$. Jeśli się przyjrzymy jego definicji, zobaczymy, że zawiera on tylko te parametry λ , których wartość jest różnicą kosztów wszystkich par krawędzi (e, e^*) , z których pierwsza należy do oryginalnego rozwiązania T_s^* , druga zaś do optymalnego rozwiązania dla problemu MST i nowego scenariusza s' . Z naszego punktu widzenia oznacza to, że odejmując dowolny taki parametr $d(i, j) \in \Lambda$ ($d(i, j) = c'_{e_i} - c'_{e_j}$) od kosztu krawędzi $e_i \in T_s^*$ (zgodnie z definicją scenariusza $s'(d(i, j))$), otrzymamy nowy koszt tej krawędzi równy wadze łuku $e_j \in T_{s'}^*$ ($c'_j - d(i, j) = c'_j$, zaś w przypadku zrównania się kosztów pojedynczych krawędzi ze zbiorów T_s^* i $T_{s'}^*$, wedle przeprowadzonego w 5.2.1 dowodu, możemy takie krawędzie zamienić miejscami, zbliżając się tym samym do optymalnego rozwiązania). Dowód lematu, na którym chcemy się opierać przebiega następująco:

Dowód. Na początku rozwińmy wzór, którego prawdziwość chcemy udowodnić, do jego najdłuższej postaci i pogrupujmy wszystkie wyrażenia wedle ich znaków:

$$c'_{e_i} - c'_{e_j} \neq c'_{e_k} - c'_{e_l} \leftrightarrow \quad (5.12)$$

$$\leftrightarrow c_{e_i} + \frac{m \cdot i^2 + i}{(m+1)^3} - c_{e_j} - \frac{m \cdot j^2 + j}{(m+1)^3} \neq c_{e_k} + \frac{m \cdot k^2 + k}{(m+1)^3} - c_{e_l} - \frac{m \cdot l^2 + l}{(m+1)^3} \leftrightarrow \quad (5.13)$$

$$\leftrightarrow c_{e_i} + \frac{m \cdot i^2 + i}{(m+1)^3} + c_{e_l} + \frac{m \cdot l^2 + l}{(m+1)^3} \neq c_{e_k} + \frac{m \cdot k^2 + k}{(m+1)^3} + c_{e_j} + \frac{m \cdot j^2 + j}{(m+1)^3}. \quad (5.14)$$

Zgodnie z naszymi ustaleniami, oryginalne koszty grafu są całkowite, zatem:

$$c_{e_i} + c_{e_l} = C_{il} \in \mathbb{Z} \quad (5.15)$$

$$c_{e_k} + c_{e_j} = C_{kj} \in \mathbb{Z} \quad (5.16)$$

$$[c_{e_i} + c_{e_l}] + \frac{m \cdot i^2 + i}{(m+1)^3} + \frac{m \cdot l^2 + l}{(m+1)^3} \neq [c_{e_k} + c_{e_j}] + \frac{m \cdot k^2 + k}{(m+1)^3} + \frac{m \cdot j^2 + j}{(m+1)^3} \leftrightarrow \quad (5.17)$$

$$\leftrightarrow C_{il} + \frac{m \cdot i^2 + i}{(m+1)^3} + \frac{m \cdot l^2 + l}{(m+1)^3} \neq C_{kj} + \frac{m \cdot k^2 + k}{(m+1)^3} + \frac{m \cdot j^2 + j}{(m+1)^3}. \quad (5.18)$$

Zgodnie z lematem, oczekujemy, że powyższa nierówność jest spełniona dla wszystkich indeksów i, j, k, l takich, że $i \neq j$ oraz $i \neq k$ ($i, j, k, l \in \{1, 2, \dots, m\}$). Pokażmy rzecz odwrotną: że wzór 5.18 może zajść w postaci równości tylko wtedy, gdy powyższe warunki nie zachodzą ($i = j$ lub $i = k$), zaś w innych przypadkach nie może być o równości dwóch stron powyższego wyrażenia mowy.

$$C_{il} + \frac{m \cdot i^2 + i + m \cdot l^2 + l}{(m+1)^3} = C_{kj} + \frac{m \cdot k^2 + k + m \cdot j^2 + j}{(m+1)^3} \leftrightarrow \quad (5.19)$$

$$\leftrightarrow C_{il} + \frac{m \cdot (i^2 + l^2) + (i + l)}{(m+1)^3} = C_{kj} + \frac{m \cdot (k^2 + j^2) + (k + j)}{(m+1)^3}. \quad (5.20)$$

Jak możemy zauważyc, aby mieć w ogóle możliwość rozważania, czy obie strony równości są sobie równe, między nimi musi zachodzić co najmniej jeden z poniższych warunków (oznaczmy lewą stronę równości jako L , prawą — P):

- $C_{il} = C_{kj} \wedge \frac{m \cdot (i^2 + l^2) + (i + l)}{(m+1)^3} = \frac{m \cdot (k^2 + j^2) + (k + j)}{(m+1)^3}$ lub

- $C_{il} \neq C_{kj} \wedge \frac{m \cdot (i^2 + l^2) + (i + l)}{(m+1)^3} + K = \frac{m \cdot (k^2 + j^2) + (k + j)}{(m+1)^3}$, gdzie $K \in \mathbb{Z}$ oraz $K \neq 0$.

Innymi słowy $L = P$ tylko wtedy, gdy wartości obu wyrażeń jednocześnie należą albo do liczb rzeczywistych, albo całkowitych. W związku z tym, jeżeli $C_{il} \neq C_{kj}$ (sumy kosztów krawędzi nie są sobie równe, to jest $c_i + c_l - K = c_k + c_j$), pozostałe członki wyrażeń L oraz P muszą być w odpowiedniej relacji między sobą i różnić się dokładnie o K . Warto podkreślić, że nie zakładamy tutaj nic o tym, czy koszty krawędzi są równe, czy $c_i \neq c_j \neq c_k \neq c_l$, jako że bierzemy pod uwagę oba przypadki: gdy $C_{il} = C_{kj}$ lub $C_{il} \neq C_{kj}$, gdzie pierwszego z nich nie będziemy omawiać, gdyż jest trywialny — jeśli $C_{il} = C_{kj}$, to L może równać się P tylko wtedy, gdy warunek $i = j \vee i = k$ jest spełniony⁶. Wraz z dowodem lematu 5.2.2 pokazaliśmy nierówność 5.6, z której wynikało, że $0 < \frac{m \cdot i^2 + i}{(m+1)^3} < 1$, tak więc dla naszego przypadku mamy:

$$0 < \frac{m \cdot i^2 + i}{(m+1)^3} < 1 \wedge 0 < \frac{m \cdot j^2 + j}{(m+1)^3} < 1 \wedge 0 < \frac{m \cdot k^2 + k}{(m+1)^3} < 1 \wedge 0 < \frac{m \cdot l^2 + l}{(m+1)^3} < 1 \rightarrow \quad (5.21)$$

$$\rightarrow 0 < \frac{m \cdot (i^2 + l^2) + (i + l)}{(m+1)^3} < 2 \wedge 0 < \frac{m \cdot (k^2 + j^2) + (k + j)}{(m+1)^3} < 2. \quad (5.22)$$

W związku z tym, pamiętając, że w przypadku drugim (gdy $C_{il} \neq C_{kj}$) chcieliśmy otrzymać równość $\frac{m \cdot (i^2 + l^2) + (i + l)}{(m+1)^3} + K = \frac{m \cdot (k^2 + j^2) + (k + j)}{(m+1)^3}$, widzimy, że jedyna wartość parametru, dla którego powyższa równość może być spełniona, to $K = 1$. W związku z tym będziemy chcieli pokazać, że:

$$C_{il} + \frac{m \cdot (i^2 + l^2) + (i + l)}{(m+1)^3} + 1 = C_{kj} + \frac{m \cdot (k^2 + j^2) + (k + j)}{(m+1)^3} \leftrightarrow \quad (5.23)$$

$$\leftrightarrow [C_{il} - C_{kj}] + 1 = \frac{m \cdot (k^2 + j^2) + (k + j)}{(m+1)^3} - \frac{m \cdot (i^2 + l^2) + (i + l)}{(m+1)^3} \quad (5.24)$$

$$\frac{m \cdot [(k^2 + j^2) - (i^2 + l^2)] + [(k + j) - (i + l)]}{(m+1)^3} \in \{-1, 0, 1\} \leftrightarrow \quad (5.25)$$

$$\leftrightarrow \left| \frac{m \cdot Z_1 + Z_2}{(m+1)^3} \right| \in \{0, 1\}. \quad (5.26)$$

Co się okazuje, ostatnie wyrażenie nie posiada rozwiązań w liczbach całkowitych innych niż trywialne ($m \cdot Z_1 + Z_2 = 0$), toteż nie jest możliwa do spełnienia zadana przez nas równość w przypadku, gdy nie jest spełniony warunek $i = k \vee i = l$. W związku z tym, o ile tylko $i \neq k \wedge i \neq l$, nierówność z podanego lematu zachodzi. ♦

Nasz nowo zdefiniowany zbiór Λ oczywiście spełnia założenia własności 5.11 — zbiór $\{e_i : d(i, j) \in \Lambda\}$ z definicji jest rozłączny ze zbiorem $\{e_j^* : d(i, j) \in \Lambda\}$ ($e_i \in T_s^* \setminus T_{s'}^* \wedge e_j^* \in T_{s'}^* \setminus T_s^*$), zatem dla wyrażenia z lematu: $c'_{e_i} - c'_{e_j} \neq c'_{e_k} - c'_{e_l}$, spełnione są własności: $i \neq j$ oraz $k \neq l$. Zarówno na samym początku omawiania podejścia bazującego na wyszukiwaniu binarnym do wskazanego problemu, jak i w algorytmie 3 dla argumentu wejściowego Λ , wskazaliśmy potrzebę uporządkowania dyskretnego zbioru parametrów λ w porządku rosnącym tak, aby wykorzystanie algorytmu wyszukiwania binarnego było w ogóle możliwe. Aby spełnić i tę własność, musimy przyjrzeć się sposobi generowania kolejnych wartości $d(i, j) = c'_{e_i} - c'_{e_j}$. Przypomnijmy, że $\Lambda = \{c'_{e_i} - c'_{e_j} : e_i \in T_s^* \setminus T_{s'}^* \wedge e_j \in T_{s'}^* \setminus T_s^*\}$. Rozpatrzmy następujący sposób konstrukcji tego zbioru:

- stworzymy zbiór E_{\geqslant} , do którego należeć będą wszystkie krawędzie $e_i \in T_s^* \setminus T_{s'}^*$, a dodatkowo niech będą one posortowane względem swoich kosztów w kolejności od największego do najmniejszego ($c_{i_1} \geqslant c_{i_2} \geqslant \dots \geqslant c_{i_l}$, gdzie $l = |E_{\geqslant}| = |T_s^* \setminus T_{s'}^*|$).

⁶Nawet, jeśli $i \neq j$ oraz $i \neq k$, a $i + l$ mimo to równa się $k + j$ (wartości l jest odpowiednio dobrana), to okazuje się, że dla wyrażenia $i^2 + l^2 = k^2 + j^2$, gdy $i \neq j$ oraz $i \neq k$, nie istnieje inne niż trywialne ($i = j = k = l = 0$) rozwiązanie, gdzie $i, j, k, l \in \mathbb{Z}$. Oznacza to, że gdy $i \neq j$ oraz $i \neq k$, to $m \cdot (i^2 + l^2) + (i + l) \neq m \cdot (k^2 + j^2) + (k + j)$.



- Analogicznie ze wszystkich krawędzi $e_j^* \in T_{\mathbf{s}'}^* \setminus T_{\mathbf{s}}^*$ stwórzmy zbiór E_{\leqslant} , którego elementy dodatkowo posortujemy rosnąco względem ich kosztów tak, że krawędź o największym koszcie będzie ostatnim elementem zbioru E_{\leqslant} ($c_{i_1} \leqslant c_{i_2} \leqslant \dots \leqslant c_{i_l}$, gdzie $l = |E_{\leqslant}| = |T_{\mathbf{s}'}^* \setminus T_{\mathbf{s}}^*|$).
- Stwórzmy, równoważny⁷ ze zbiorem Λ , zbiór $\Lambda' = \{c'_{e_i} - c'_{e_j} : e_i \in E_{\geqslant} \wedge e_j \in E_{\leqslant}\}$.

Zauważmy, że elementy tak zdefiniowanego zbioru spełniają poniższe nierówności (innymi słowy funkcja zwracająca ich wartości — $d : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{R}$ — jest monotoniczna i rosnąca względem któregokolwiek z parametrów):

$$\forall i, i' \in \{1, \dots, |E^*|\} \quad d(i, j) < d(i', j) \Leftrightarrow i < i' \text{ oraz} \quad (5.27)$$

$$\forall j, j' \in \{1, \dots, |E^*|\} \quad d(i, j) < d(i, j') \Leftrightarrow j < j'. \quad (5.28)$$

Pozwala to nam w łatwy sposób przeglądać kolejne elementy danego zbioru w pożądanej dla nas (patrz argumenty pseudokodu 3) kolejności. Tak stworzony zbiór ma jeszcze jedną, bardzo wygodną dla nas, własność — opierając się na jego sposobie konstrukcji (monotoniczności względem obu indeksów i oraz j), z łatwością możemy zliczać elementy, które w danej chwili nas interesują, co wykorzystamy w następnym rozdziale przy konstrukcji algorytmów 4 oraz 5.

Pseudokod 3: IMST-BINARY-SEARCH ($G^*, T_{\mathbf{s}'}^*, \mathbf{s}', \Lambda, k$)

Wejście: $G^* = (V, E^*)$ — graf ze zbiorem krawędzi $T_{\mathbf{s}'}^* \cup T_{\mathbf{s}}^*$,
 $T_{\mathbf{s}'}^*$ — początkowe minimalne drzewo rozpinające dla scenariusza \mathbf{s} ,
 \mathbf{s}' — nowe koszty krawędzi grafu,
 Λ — posortowany rosnąco, dyskretny zbiór parametrów λ ,
 k — parametr problemu IMST.

Wyjście: $T(\lambda^*)$ — optymalne rozwiązanie problemu IMST.

```

1 begin
2   lowerIdx ← 0                                // Dolny indeks zbioru  $\Lambda$ .
3   upperIdx ←  $|\Lambda| - 1$                       // Górnny indeks zbioru  $\Lambda$ .
4   while  $lowerIdx \leqslant upperIdx$  do
5     lambdaIdx ←  $\frac{lowerIdx+upperIdx}{2}$ 
6      $\lambda \leftarrow \Lambda[\lambdaIdx]$ 
7      $\mathbf{s}'(\lambda) \leftarrow \begin{cases} c_i^{\mathbf{s}'} & e_i \notin T_{\mathbf{s}}^* \\ c_i^{\mathbf{s}'} - \lambda & e_i \in T_{\mathbf{s}}^* \end{cases}$ 
8      $T(\lambda) \leftarrow \text{GET-MST}(G^*, \mathbf{s}'(\lambda))$     // Rozwiąż problem MST dla grafu  $G^*$ , którego koszty krawędzi tymczasowo zostaną ustalone zgodnie ze scenariuszem  $\mathbf{s}'(\lambda)$ .
9     kDiff ←  $\text{GET-DIFF}(T(\lambda), T_{\mathbf{s}}^*)$         // Zwróć liczbę krawędzi  $e \in T(\lambda) \setminus T_{\mathbf{s}}^*$ .
10    if  $kDiff = k$  then
11      return  $T(\lambda)$ 
12    else if  $kDiff > k$  then
13      lowerIdx ← lambdaIdx + 1                    // Podniesienie dolnego ograniczenia na wartość  $\lambda$ , jako że zwrócona liczba krawędzi nienależących do  $T_{\mathbf{s}}^*$  była za duża, co oznacza konieczność dalszego obniżania kosztów krawędzi  $e \in T_{\mathbf{s}}^*$ .
14    else
15      upperIdx ← lambdaIdx - 1                  // Obniżenie górnego ograniczenia na wartość  $\lambda$ , jako że zwrócona liczba krawędzi nienależących do  $T_{\mathbf{s}}^*$  była zbyt mała. Koszty krawędzi  $e \in T_{\mathbf{s}}^*$  zostały zbyt obniżone.

```

⁷Pod względem posiadanych elementów.

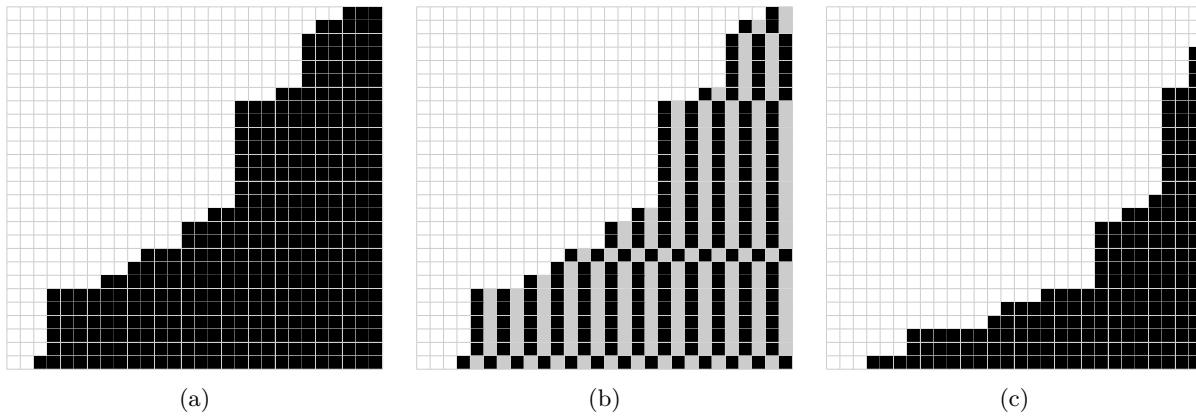
5.4 Pełny algorytm przeszukiwania binarnego

Pełen pseudokod naszego algorytmu rozwiązującego problem minimalnego drzewa rozpinającego w wersji INCREMENTAL, bazujący na algorytmie wyszukiwania binarnego, przedstawiamy poniżej (zobacz pseudokod 4). Większość problemów, które stawia przed nami do rozwiązania prezentowany algorytm, została już przez nas omówiona, toteż zamiast jeszcze raz przytaczać sposoby ich rozwiązywanie, w tej części skupimy się na implementacyjnych szczegółach podanego pseudokodu. Na początku warto zwrócić uwagę na schemat omainiowego algorytmu, który swoją konstrukcją mocno przypomina poprzednio przedstawiany algorytm. Bardzo łatwo zauważać, że tak samo jak w przypadku IMST-BINARY-SEARCH ($G^*, T_s^*, s', \Lambda, k$), także tutaj główną osią algorytmu jest podejmowanie decyzji dotyczących zawężania dolnego i górnego parametru ograniczającego przestrzeń rozwiązań, które należy sprawdzić w celu wyłonienia tego optymalnego (linijki 20–25 dla INCREMENTAL-MST (G^*, T_s^*, s', k, L, U) odpowiadają liniom 9–15 w IMST-BINARY-SEARCH ($G^*, T_s^*, s', \Lambda, k$)). Jedyną różnicą w tej części kodu jest sposób wybierania wartości nowego górnego bądź dolnego ograniczenia dla następnej iteracji oraz sposób jej wywołania (choć nic nie stoi na przeszkodzie, aby — na wzór drugiego z wymienionych algorytmów — przekształcić ten pierwszy, usuwając wywołania rekurencyjne, tak jak to pokazano na pseudokodzie 5). Sam zaś algorytm ma głównie na celu generowanie kolejnych zbiorów parametrów λ , które są dopuszczalne w myśl aktualnie wybranych ograniczeń (takimi wartościami parametru λ będą zatem wszystkie wartości $d(i, j)$ spełniające $L \leq d(i, j) \leq U$). Same zaś ograniczenia L i U będziemy oczywiście chcieć tak dobierać, aby z każdą następną iteracją zbiór $F = \{(i, j) : L \leq d(i, j) \leq U\}$ był coraz mniejszy — naszym celem bowiem jest doprowadzenie do sytuacji, w której liczba elementów tego zbioru jest na tyle niewielka, że wykonanie procedury IMST-BINARY-SEARCH ($G^*, T_s^*, s', \Lambda, k$) nie będzie nas wiele kosztować (linie 11–13).

Na początek jednak przyjrzyjmy się fragmentowi, który odpowiada za zliczanie liczby elementów (i, j) , które spełniają zadane ograniczenia (2–9). Dla każdej pierwszej krawędzi z pary łuków (e_i, e_j^*) , których różnica kosztów jest wyliczana wyrażeniem $d(i, j)$, chcemy znaleźć parę indeksów (j_l, j_u) taką, że zachodzi warunek $L \leq d(i, j_l) \leq \dots \leq d(i, j_u) \leq U$. Nasze poszukiwania tych parametrów możemy zacząć w dowolny sposób, jednak już w tym kroku napotykamy pierwszy problem — co jeśli dowolne z wyrażeń $\min \{j : L \leq d(i, j)\}$ lub $\max \{j : d(i, j) \leq U\}$ nie zwróci żadnego wyniku? Może bowiem się zdarzyć, że dla zadanego parametru i , tak zadana wartość (koszt krawędzi $e_i \in T_s^* \setminus T_{s'}^*$) dla wyrażenia $d(i, j)$, dla dowolnego $j \in \{j : e_j^* \in T_{s'}^* \setminus T_s^*\}$, nie będzie spełniać zadanych nierówności. Zwróćmy uwagę, że w takim przypadku, jeżeli byśmy rozpoczęli wyszukiwanie indeksu i od najniżej do najwyższej jego wartości dla $\text{MinIndex}[i]$ (oczekujemy, że szukając najmniejszego indeksu spełniającego zadany warunek, najszybciej na niego natrafimy, rozpoczynając przeszukiwanie w takiej kolejności) i dla żadnego $j \in \{1, \dots, m'\}$ ($m' = |T_s^* \setminus T_{s'}^*| = |T_{s'}^* \setminus T_s^*|$) warunek $L \leq d(i, j)$ nie byłby spełniony, aby zachować poprawność algorytmu, powinniśmy zwrócić wartość większą niż m' , tak aby w linii 6 zapewnić sobie niespełnienie znajdującego się w niej warunku (wartość $\text{MaxIndex}[i]$ nigdy nie będzie większa od m' , o czym się zaraz przekonamy). Analogicznie do $\text{MinIndex}[i]$ wartość wyrażenia $\text{MaxIndex}[i]$ możemy ustalić rozpoczynając wyszukiwanie indeksu $j_u = \max \{j : d(i, j) \leq U\}$ od $j = m'$ (znówmy oczekujemy, że skoro szukamy maksymalnej jego wartości, rozsądnie będzie zacząć od największego indeksu, dla którego wyrażenie $d(i, j)$ ma sens). Tutaj także musimy obsłużyć przypadek, w którym $\forall j \in \{m', \dots, 1\} d(i, j) > U$ — naturalnym wyborem indeksu krawędzi j w takiej sytuacji jest 0 (jako że od samego początku indeksujemy krawędzie począwszy od jedynki).

Tak poprawnie wyliczona wielkość zbioru, w przypadku jej liczebności przekraczającej zadaną granicę (tutaj jest to $12 \cdot n$), powinna zostać w następnym kroku zmniejszona, co dzieje się w liniach od 15 do 19 pseudokodu 4, w których to „przesiewamy” dopuszczalny zbiór parametrów na podstawie wybranej wartości K . Doskonałą ilustrację schematu działania, znajdującego się w tym fragmencie kodu, mechanizmu, przedstawiają rysunki od 5.2a do 5.2c, gdzie w wyniku tylko dwóch iteracji zredukowaliśmy przestrzeń przeszukiwania o ponad 75%. Na uwagę zasługuje też fakt, że w przedstawionym przypadku ani razu nie ograniczyliśmy wartości $d(i, j)$ z góry — widzimy to po specyficzny kształcie, widocznym na wszystkich trzech prezentowanych rysunkach.

Aby uzyskać jeszcze lepszy obraz sytuacji, do której doprowadził nas wykorzystywany algorytm, spójrzmy na rysunki od 5.3a do 5.3c — przedstawiono na nich wszystkie możliwe wartości funkcji $d(i, j)$, dla każdego $i, j \in \{1, \dots, |E|\}$ (nie tylko dla ograniczonego zbioru $\{(i, j) : e_i \in T_s^* \setminus T_{s'}^* \wedge e_j \in T_{s'}^* \setminus T_s^*\}$). Pierwszy rysunek (5.3a) rozpoczyna się tam, gdzie skończyliśmy poprzednio, analizując ilustrację 5.2c — na drugiej iteracji algorytmu INCREMENTAL-MST (G^*, T_s^*, s', k, L, U), dla grafu z 15 wierzchołkami i 42 krawędziami,



Rysunek 5.2: Tablice przedstawiające przykładowy, rzeczywisty rozkład dopuszczalnych wartości funkcji $d(i, j)$ dla par (i, j) i grafu G z 15 wierzchołkami, o gęstości na poziomie 40% grafu pełnego (który ma $\binom{15}{2}$ krawędzi). Z 42 luków grafu, dla dwóch scenariuszy \mathbf{s} i \mathbf{s}' , zostały wygenerowane minimalne drzewa rozpinające $T_{\mathbf{s}}^*$ oraz $T_{\mathbf{s}'}^*$, których część wspólną stanowi 28 krawędzi grafu G . Kolumny odpowiadają pierwszemu indeksowi z pary (i, j) . (a) Zbór $\Lambda = \left\{ c'_{e_i} - c'_{e_j} : e_i \in T_{\mathbf{s}}^* \setminus T_{\mathbf{s}'}^* \wedge e_j \in T_{\mathbf{s}'}^* \setminus T_{\mathbf{s}}^* \right\}$ (cała przestrzeń) oraz dopuszczalne pary $F = \{(i, j) : L' \leq d(i, j) \leq U\}$ (zaznaczone kolorem czarnym) po pierwszej iteracji algorytmu INCREMENTAL-MST($G^*, T_{\mathbf{s}}^*, \mathbf{s}', k, L, U$) (dla ograniczeń L' i U , gdzie $L < L' \leq U$). Wstępna liczba dopuszczalnych par została zredukowana z 1764 do 784 (dzięki ograniczeniu się do par ze zbioru Λ), a następnie do liczby 380 (po zastosowaniu nowego dolnego ograniczenia L'). (b) „Przesiany” zbiór dopuszczalnych rozwiązań, na podstawie którego wyliczona zostanie mediana, będąca nową wartością λ (która w następnej iteracji będzie albo nowym górnym, albo nowym dolnym ograniczeniem na dopuszczalność par (i, j)). (c) Pary $F = \{(i, j) : L'' \leq d(i, j) \leq U\}$ dla nowego dolnego ograniczenia L'' ($L < L' < L'' \leq U$). Liczba par została zredukowana z 380 do 189.

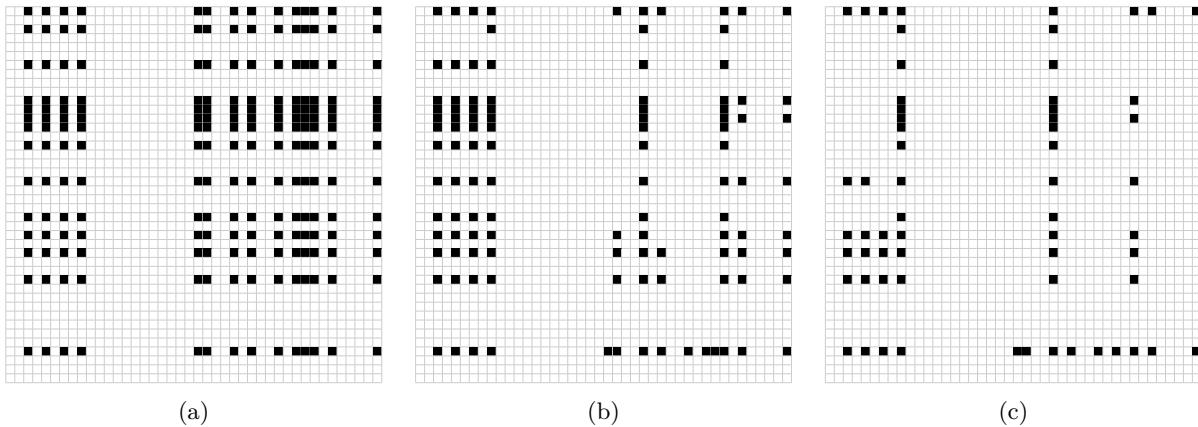
których liczebność wynika z ządanej gęstości generowanego losowo grafu (40%). Należy podkreślić, że obydwa grafy, których tyczą się ilustracje 5.2a oraz 5.3a, choć posiadają te same parametry, są różne względem siebie (procedura wykorzystana do ich generowania jest losowa). Rozmiar danych jednak jest na tyle mały, że liczby dopuszczalnych par (i, j) po drugiej iteracji algorytmu w obu przypadkach są zbliżone do siebie (189 w pierwszym przypadku, 196 w drugim).

To co od razu możemy dostrzec, patrząc zwłaszcza na pierwszy z rysunków (5.3a), to wyraźna regularność zaznaczonych elementów, spowodowana wykorzystaniem wcześniej udowadnianych własności (5.4, 5.5, 5.3.1), które pozwoliły nam na skonstruowanie zbioru Λ — na wspomnianym rysunku widać wyraźnie, że dla niektórych par krawędzi (dla wierszy oraz kolumn symbolizujących krawędzie, które są wspólną częścią drzew $T_{\mathbf{s}}^*$ oraz $T_{\mathbf{s}'}^*$) nie istnieje żadna komórka, która byłaby wliczana w zbiór par dopuszczalnych rozwiązań.

5.4.1 Lepiej, szybciej

Tak jak to zostało już wcześniej podkreślone, niewielkim nakładem pracy jesteśmy w stanie przekształcić pseudokod 4 do postaci nie korzystającej z rekurencji, tak aby do złudzenia swoją konstrukcją przypominał tradycyjny algorytm przeszukiwania binarnego (pseudokod 3). Jedną z niewątpliwych zalet takiego rozwiązania jest nie tylko poprawiona czytelność prezentowanego kodu, co niejednokrotnie dużo szybszy czas jego działania, idącego w parze z oszczędnością pamięci, której algorytm wymaga coraz więcej z każdym wywołaniem rekurencyjnym. W tej części skupimy się na rozwiązaniach, jakie możemy zastosować w celu poprawienia ogólnej wydajności wcześniejszego omówionego algorytmu, krok po kroku go analizując.

Do tej pory nic nie wspominaliśmy o złożoności prezentowanego rozwiązania — jak pamiętamy, podstawową koncepcją algorytmu jest zmniejszanie liczby elementów (i, j) o zadanych właściwościach do momentu, w którym jest ich na tyle niewielu, że decydujemy się na binarne wyszukanie rozwiązania przy wykorzystaniu IMST-BINARY-SEARCH($G^*, T_{\mathbf{s}}^*, \mathbf{s}', F, k$). Owocuje to natychmiastowo złożonością na poziomie $O(I + K + S + \log(n) \cdot B)$, gdzie:



Rysunek 5.3: Macierz, której komórki przedstawiają wartości wyrażeń $c_i - c_j$ dla dowolnej kombinacji krawędzi e_i oraz e_j w grafie $G = (V, E, q)$, gdzie $|V| = 15$, $|E| = 42$, q jest gęstością grafu ($q = 0.4$). Zaznaczone na czarno elementy należą do zbioru $\{(i, j) : L \leq c_i - c_j \leq U\}$, gdzie wartości L i U odpowiadają aktualnym w drugiej iteracji algorytmu INCREMENTAL-MST (G^*, T_s^*, s', k, L, U) ograniczeniom. (a) Dopuszczalne pary w liczbie 196. (b) Zredukowany zbiór dopuszcjalny par (i, j) , dla których wartość $c_i - c_j$ jest nie mniejsza niż dolne i nie większa niż górne ograniczenie, wybrane przez z algorytm w kolejnej iteracji. Liczba zaznaczonych elementów zmalała do 110. (b) Pomniejszony w wyniku kolejnej iteracji algorytmu INCREMENTAL-MST (G^*, T_s^*, s', k, L, U) zbiór, który liczy już tylko 62 elementy.

- I oznacza czas potrzebny na znalezienie rozwiązań T_s^* i $T_{s'}^*$ dla problemu minimalnego drzewa rozpinającego, dla początkowego scenariusza s oraz nowego — s' , i jest równoznaczne z czasem wykonania się podstawowego algorytmu rozwiązującego ten problem ($O(m \cdot \alpha(m, n))$ [5]),
- K czasu musimy przeznaczyć na skonstruowanie parametrów, których wartości do tej pory oznaczaliśmy poprzez $d(i, j)$,
- S oznacza czas potrzebny algorytmowi IMST-BINARY-SEARCH (G^*, T_s^*, s', F, k) na wstępne posortowanie otrzymanego zbioru F (w czasie $O(n \cdot \log(n))$), zaś
- $B = I$, gdyż wspomniany wcześniej algorytm w najgorszym przypadku będzie musiał obliczyć $O(\log(n))$ minimalnych drzew rozpinających, zanim zwróci rozwiązanie.

Podsumowując, otrzymujemy złożoność na poziomie $O(m\alpha(m, n) + K + n \log(n) + \log(n)m\alpha(m, n))$. Wartym uświadomienia jest fakt, że przywoływana tutaj **odwrócona funkcja Ackermann'a** jest funkcją ekstremalnie wolno rosnącą, w związku z czym $O(m\alpha(m, n) + K + n \log(n)\alpha(n, n)) \approx O(m + K + n \log(n))$ (dla dowolnie dużych, lecz jednocześnie sensownie małych w kontekście rozpatrywanego problemu danych, funkcja zwraca wartości na tyle małe, że w notacji dużego O możemy uznać je za stałe). Przy redukowaniu powyższego wzoru skorzystaliśmy dodatkowo z faktu, że o ile czas trwania algorytmu Chazelliego wynosi $O(m \cdot \alpha(m, n))$, to biorąc pod uwagę, że operujemy na grafie G^* , którego maksymalna liczba krawędzi to $2 \cdot (n - 1)$, rzeczywista otrzymana złożoność wynosi już tylko $O(n \cdot \alpha(n, n))$. Nie tyczy się to oczywiście pierwszego członu wyrażenia w notacji dużego O , gdyż w przypadku wyliczania drzew T_s^* oraz $T_{s'}^*$ operowaliśmy jeszcze na pełnym grafie G (dopiero obliczenie tych drzew pozwoli nam na zredukowanie liczby jego krawędzi). To zaś wskazuje na istotność w jaki sposób wyliczane są parametry λ (jedyna niewiadoma, która pozostała w wyprowadzonym wzorze na złożoność obliczeniową algorytmu to K) — możemy przedstawić dwa podejścia do tego problemu.

- Naiwne policzenie wszystkich możliwych wartości — nawet dla bardzo okrojonego zbioru jakim jest $\{(i, j) : e_i \in T_s^* \setminus T_{s'}^* \wedge e_j \in T_{s'}^* \setminus T_s^*\}$, chęć wyliczenia wszystkich możliwych parametrów natychmiast ograniczy naszą złożoność do poziomu $\Omega(n^2)$ ⁸.

⁸Chcąc być dokładniejszym — do $\Omega(|T_s^* \setminus T_{s'}^*|^2)$, czyli w najgorszym przypadku właśnie do $\Omega((|V| - 1)^2) = \Omega(n^2)$.

**Pseudokod 4:** INCREMENTAL-MST ($G^*, T_s^*, \mathbf{s}', k, L, U$)

Wejście: $G^* = (V, E^*)$ — graf ze zbiorem krawędzi $T_{\mathbf{s}'}^* \cup T_s^*$,
 T_s^* — początkowe minimalne drzewo rozpinające dla scenariusza \mathbf{s} ,
 \mathbf{s}' — nowe koszty krawędzi grafu,
 k — parametr problemu IMST,
 L — dolne ograniczenie na wartość parametru λ^* ,
 U — górne ograniczenie na wartość parametru λ^* ,

Wyjście: $T(\lambda^*)$ — optymalne rozwiązanie problemu IMST.

```

1 begin
2   for  $i \in \{1, \dots, m'\}$                                 //  $m' = |T_{\mathbf{s}'}^* \setminus T_s^*|$ 
3     do
4       MinIndex[i]  $\leftarrow \min \{j : L \leq d(i, j)\}$ 
5       MaxIndex[i]  $\leftarrow \max \{j : d(i, j) \leq U\}$ 
6       if  $MinIndex[i] \leq MaxIndex[i]$  then
7         Count[i]  $\leftarrow MaxIndex[i] - MinIndex[i] + 1$ 
8       else
9         Count[i]  $\leftarrow 0$ 
10      TotalCount  $\leftarrow \sum_{i=1}^{m'} Count[i]$ 
11      if  $TotalCount \leq 12 \cdot n$  then
12        F  $\leftarrow \{(i, j) : L \leq d(i, j) \leq U\}$ 
13        return IMST-BINARY-SEARCH ( $G^*, T_s^*, \mathbf{s}', F, k$ )
14    else
15      K  $\leftarrow \lfloor \frac{TotalCount}{6 \cdot n} \rfloor$ 
16      for  $i \in \{1, \dots, m'\}$  do
17        H[i]  $\leftarrow \{(i, j) : L \leq d(i, j) \leq U \wedge j \leq MinIndex[i] + r \cdot K \wedge r \in \mathbb{Z}^+\}$ 
18      H  $\leftarrow \bigcup_{i=1}^{m'} H[i]$ 
19      λ  $\leftarrow \text{MEDIAN}(\mathcal{H})$ 
20      if  $f(T(\lambda), T_s^*) = k$  then
21        return  $T(\lambda)$ 
22      else if  $f(T(\lambda), T_s^*) > k$  then
23        return INCREMENTAL-MST ( $G^*, T_s^*, \mathbf{s}', k, \lambda, U$ )
24      else if  $f(T(\lambda), T_s^*) < k$  then
25        return INCREMENTAL-MST ( $G^*, T_s^*, \mathbf{s}', k, L, \lambda$ )

```

- Wyliczanie wartości $d(i, j)$ na bieżąco (tak jak to właśnie zakładamy w przedstawianych pseudokodach). Skutkuje to koniecznością wygenerowania dwóch zbiorów ($E_0 = \{e \in T_s^* \setminus T_{\mathbf{s}'}^*\}$ i $E_1 = \{e \in T_{\mathbf{s}'}^* \setminus T_s^*\}$ oraz ich posortowania, co wykonamy w czasie co najwyżej $O(m + n \cdot \log(n))$ ($O(m)$ zajmie nam wybór odpowiednich krawędzi, $O(n \cdot \log(n))$ — posortowanie ich, pierwszego rosnąco, drugiego malejąco). Od tej pory odwołanie się do funkcji $d : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{R}$ zwracać nam będzie pożądaną wartość w czasie stałym.

Zajmijmy się teraz sposobem w jaki zminimalizować możemy liczbę wywołań wspomnianej funkcji. Spoglądając na linijki od 2 do 9 w pseudokodzie 4, widzimy, że dla każdego i od 1 do m' wyliczamy dwie wartości z podstawie funkcji $\min(\bullet)$ oraz $\max(\bullet)$. Powiedzieliśmy sobie, że dla każdego $i \in \{1, \dots, m'\}$ najprzypadkniej z naszej strony będzie rozpoczynać wyszukiwanie tej pierwszej od $j = 1$, zaś drugiej od $j = m'$ — w najgorszym przypadku (gdy dla każdego i , $\min\{j : L \leq d(i, j)\} = m'$ a $\max\{j : d(i, j) \leq U\} = 1$) daje nam to złożoność na poziomie $O(m' \cdot m')$ (pamiętamy, że m' jest liczbą krawędzi dowolnego ze zbiorów $T_{\mathbf{s}'}^* \setminus T_s^*$ lub $T_s^* \setminus T_{\mathbf{s}'}$, $m' \leq n - 1$). Widzimy zatem, że taka strategia może zaowocować nawet złożonością na poziomie $O(n^2)$, gdzie podobny rezultat uzyskaliśmy, stosując naiwne podejście.

Pokażemy teraz jak w prosty sposób, wykorzystując własności funkcji $d : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{R}$ (monotoniczność względem dowolnego z jej parametrów), możemy przyspieszyć ten proces, ograniczając go do co najwyżej



m' wywołań powyższej funkcji. W 5.27 pokazaliśmy, że wartość funkcji $d(\bullet, \bullet)$ rośnie wraz z dowolnym jej parametrem, to znaczy zachodzą⁹:

- $\forall i, j \ d(i, j) < d(i, j + 1)$,
- $\forall i, j \ d(i, j) < d(i + 1, j)$
- $\forall i, j \ d(i, j) < d(i + 1, j + 1)$.

Przyjrzymy się teraz wyrażeniom $\min\{j : L \leq d(i, j)\}$ oraz $\max\{j : d(i, j) \leq U\}$. Przypomnijmy jeszcze raz, że w celu znalezienia indeksu j o najmniejszej wartości, postanowiliśmy rozpoczęć jego poszukiwania od $j = 1$. Niech teraz $i = m'$. Niech $j = j_1$ będzie takim indeksem, że $j_1 = \min\{j : L \leq d(m', j)\}$. Zgodnie z powyższymi własnościami zachodzi: $d(m' - 1, j_1) < d(m', j_1)$ — mamy teraz prawo podejrzewać, że skoro $j = j_1$ było najmniejszym indeksem, dla którego wyrażenie $d(m', j)$ było większe niż L , to wartość $d(m' - 1, j)$ może nie spełniać już tej własności. W takim przypadku, by znaleźć kolejny indeks, tym razem dla $i = m' - 1$, musimy wybrać taki parametr j , aby zwiększyć poprzednią wartość funkcji (by na nowo $d(i, j)$ przewyższało wartość dolne ograniczenie L). Z własności przedstawionych wyżej wiemy, że jedynym sposobem na dokonanie tego jest zwiększenie drugiego z parametrów ($d(m' - 1, j_1) < d(m' - 1, j_1 + 1) < \dots$). Co także wiemy, w pewnym momencie, dla pewnego $j = j_2 > j_1$, zajdzie $L \leq d(m' - 1, j)$, tak więc indeks j_2 jest tym najmniejszym, który dla kolejnej wartości indeksu i spełnia wszystkie założenia. Proces ten możemy kontynuować aż do momentu, w którym $j = m' + 1$, bądź policzyliśmy wartości $\text{MinIndex}[i]$ dla wszystkich $i \in \{m', \dots, 1\}$. W tym pierwszym przypadku (linie 13–14 algorytmu 5) możemy zauważać, że nie ma potrzeby kontynuować procesu dla wszystkich takich i' , których wartość jest mniejsza od ostatniego i , dla którego zachodziła nierówność $L \leq d(i, m')$ — ponownie, opierając się na własnościach monotoniczności funkcji $d(\bullet, \bullet)$, wiemy, że dla wszystkich pozostałych wartości $i' < i$, aby utrzymać wartość wyrażenia $d(\bullet, \bullet)$ powyżej dolnego ograniczenia L , jesteśmy zmuszeni do sukcesywnego podwyższania drugiego z jej parametrów, ten zaś nie może być dalej zwiększany, gdyż osiągnął już swoją graniczą wartość, dla której $d(\bullet, \bullet)$ ma dla nas sens. Może się oczywiście zdarzyć, że dla pewnego ciągu $i - k, i - (k - 1), \dots, i$ będzie zachodzić $L \leq d(i - k, j) < d(i - (k - 1), j) < \dots < d(i, j)$ — w takim przypadku po prostu wstrzymujemy się na pewien czas z podwyższaniem wartości parametru j .

Powyższą strategię doskonale ilustruje rysunek 5.4a, gdzie wyraźnie widzimy w których momentach wartość indeksu j dla wyrażenia $d(i, j)$ została na tyle podwyższona, aby przekroczyła wartość dolnego ograniczenia L , co pociąga za sobą obniżenie pierwszego parametru i rozpoczęcie procesu od nowa. Analogicznie do schematu wyszukiwania wszystkich wartości $\text{MinIndex}[i]$ zachowuje się proces obliczania wyrażeń $\text{MaxIndex}[i]$ (przedstawiony na rysunku 5.4b) — opierając się o to samo rozumowanie oraz te same własności co powyżej, zaczynając od $i = 1$ oraz $j = m'$, możemy tak manipułować obydwoema wartościami, aby całkowity czas operacji nie przekroczył czasu liniowego, zależnego od m' , podobnie jak miało to miejsce wcześniej. W przypadku rozpoczętym od $j = m'$, możemy poczynić niewielką obserwację, że gdy pętla 17–21 w algorytmie 5 zostanie przerwana, zanim policzymy $\text{MaxIndex}[i]$ dla wszystkich $i \in \{1 \dots, m'\}$, do pozostałych zmiennych możemy przypisać stałą 0, gdyż taką wartość przyjmie wtedy indeks j . Razem z następnymi linijkami, fragment 5–26 stanowi eleganckie usprawnienie bloku 2–10 poprzedniego algorytmu, już znacznie go usprawniając.

Dalsze różnice pomiędzy algorytmami 4 a 5 mają charakter bardziej kosmetyczny (usunięcie wywołań rekurencyjnych, zasugerowanie sposobu konstrukcji zbiorów dopuszczalnych par F , przeredzonego zbioru \mathcal{H}) i nie wymagają poświęcania im większej uwagi.

5.5 Analiza poprawności

Ostatnią rzeczą, o której będziemy chcieli się przekonać, jest poprawność skonstruowanych przez nas algorytmów — pokażemy, że dla dowolnych danych zatrzymują się one i zwracają oczekiwane przez nas roz-

⁹Na mocy lematu 5.3.1 wiemy także, że we wszystkich trzech przypadkach zachodzi ostra nierówność.

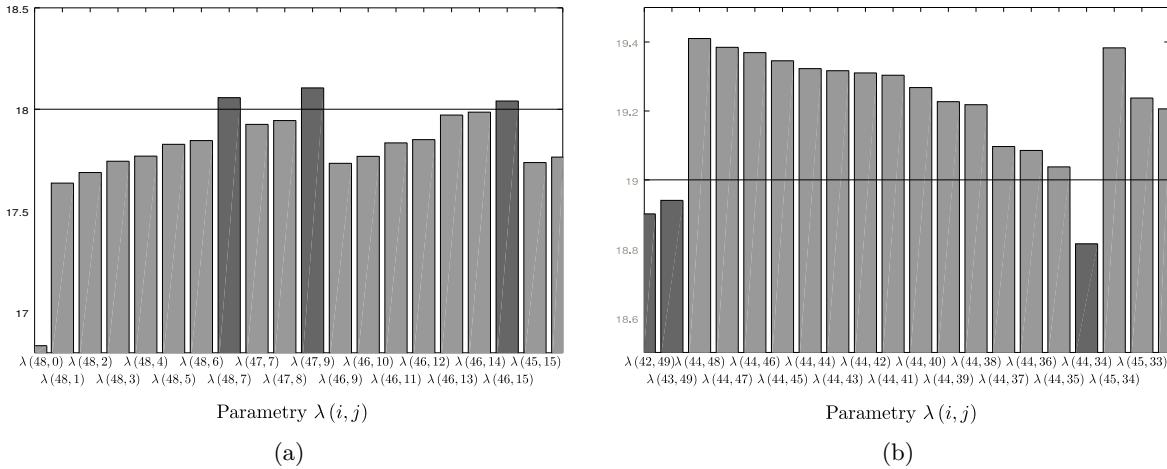
¹⁰Gdybyśmy numerowali nasze krawędzie począwszy od 0, takie przypisanie też zachowywałoby swoje funkcje, gdyż jest ono wykonywane tylko wtedy, gdy $U < d(i, 0)$ (dla dowolnego i oraz najbliższego indeksu krawędzi, którym jest teraz 0). To z kolei pociąga fakt, że $L \leq d(i, 0)$ ($L \leq U$), więc $\text{MinIndex}[i] \geq 0$, co zaś prowadzi do wniosku, że zbiór $\{(i, j) : L \leq d(i, j) \leq U\}$ dla danego i jest co najmniej jednoelementowy (nie potrzebujemy zatem sztucznego dolnego indeksu).

**Pseudokod 5:** INCREMENTAL-MST' (G^*, T_s^*, s', k, L, U)

```

1 begin
2   MaxIndex [0] ←  $m'$ 
3   MinIndex [ $m'$ ] ← 0
4   while  $L \leq U$  do
5     TotalCount ← 0
6      $i \leftarrow m'$ 
7      $j \leftarrow \text{MinIndex}[i]$ 
8     while  $j \leq m' \wedge i \geq 1$  do
9       while  $L > d(i, j) \wedge j \leq m'$  do
10         $j \leftarrow j + 1$ 
11        MinIndex [ $i$ ] ←  $j$ 
12         $i \leftarrow i - 1$ 
13     for  $i' \in \{i, \dots, 1\}$  do
14       MinIndex [ $i$ ] ←  $j$ 
15      $i \leftarrow 1$ 
16      $j \leftarrow \text{MaxIndex}[i]$ 
17     while  $j \geq 1 \wedge i \leq m'$  do
18       while  $d(i, j) > U \wedge j \geq 1$  do
19          $j \leftarrow j - 1$ 
20         MaxIndex [ $i$ ] ←  $j$ 
21          $i \leftarrow i + 1$ 
22     for  $i' \in \{i, \dots, 1\}$  do
23       MaxIndex [ $i$ ] ← 0 // Krawędzie indeksujemy od 1, więc 0 oznacza brak
24     for  $i \in \{1, \dots, m'\}$  do
25       if  $\text{MinIndex}[i] \leq \text{MaxIndex}[i]$  then
26         TotalCount ← TotalCount + MaxIndex [ $i$ ] - MinIndex [ $i$ ] + 1
27     if  $TotalCount \leq 12 \cdot n$  then
28        $F \leftarrow \emptyset$ 
29       for  $i \in \{1, \dots, m'\}$  do
30         for  $j \in \{\text{MinIndex}[i], \dots, \text{MaxIndex}[i]\}$  do
31            $F \leftarrow F \cup (i, j)$ 
32       return IMST-BINARY-SEARCH ( $G^*, T_s^*, s', F, k$ )
33     else
34        $K \leftarrow \lfloor \frac{\text{TotalCount}}{6 \cdot n} \rfloor$ 
35        $\mathcal{H} \leftarrow \emptyset$ 
36       for  $i \in \{1, \dots, m'\}$  do
37          $j_{\max} \leftarrow \text{MaxIndex}[i]$ 
38         for  $j \leftarrow \text{MinIndex}[i] ; j < j_{\max} ; j \leftarrow j + K$  do
39            $\mathcal{H} \leftarrow \mathcal{H} \cup (i, j)$ 
40        $\lambda \leftarrow \text{MEDIAN}(\mathcal{H})$ 
41        $kDiff \leftarrow \text{GET-DIFF}(T(\lambda), T_s^*)$  // Zwróć liczbę krawędzi  $e \in T(\lambda) \setminus T_s^*$ .
42       if  $kDiff = k$  then
43         return  $T(\lambda)$ 
44       else if  $kDiff > k$  then
45          $L \leftarrow \lambda$ 
46       else
47          $U \leftarrow \lambda$ 

```



Rysunek 5.4: Graficzne zaprezentowanie wartości funkcji $d : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{R}$ dla wycinka algorytmu 5, obliczającego $\min \{j : L \leq d(i, j)\}$ dla wybranego zakresu wartości indeksów. (a) Początek procesu obliczania kolejnych zmiennych $\text{MinIndex}[i]$ dla $i \in \{48, \dots, 45\}$ ($m' = 49$), gdzie jasnym szarym kolorem oznaczono te wartości $d(i, j)$, dla których nie zachodził warunek $L \leq d(i, j)$. Miejsca oznaczone kolorem ciemniejszym stanowią punkty, w których dla danego i odnaleziono takie j , że $j = \min \{j : L \leq d(i, j)\}$ ($L = 18$). (b) Graficzna reprezentacja fragmentu algorytmu obliczającego wyrażenie $\max \{j : d(i, j) \leq U\}$ (od $i = 42$ do $i = 45$). Ciemniejszym kolorem oznaczono wartości funkcji $d(i, j)$, która dla tak dobranych parametrów, spełnia powyższą nierówność ($U = 19$).

wiązania. W tym celu podsumujemy kroki, jakie wykonuje algorytm, który przedstawiliśmy poniżej (zobacz pseudokod 5). Będziemy przede wszystkim chcieli pokazać, że albo algorytm zwróci optymalne rozwiązanie, albo w pewnym momencie zajdzie $U < L$, co spowoduje przerwanie głównej pętli algorytmu (4–47) i zakończenie jego pracy. Sytuacja taka może mieć miejsce w przypadku, gdy np. zbyt pospiesznie odrzucamy dopuszczalne parametry λ (w linii 34 zastosujemy inną, dużo większą wartość parametru K , co spowoduje znacznie szybsze zawężanie zbioru dopuszczalnych parametrów λ , jednak jednocześnie niesie za sobą ryzyko, że odrzucimy wszystkie te wartości $\lambda \in \Lambda^*$, dla których $f(T(\lambda), T_s^*) = k$, co uniemożliwi algorytmowi zakończenie pracy w wyniku znalezienia optymalnego, w myśl podanych przez nas warunków, rozwiązania — linie 42–43). Jak już pokazaliśmy wyżej — kroki, mające na celu wygenerowanie zbiorów Λ , charakteryzują się liniowym czasem działania; bez trudu możemy zauważyc, że pętle, zawarte pomiędzy liniami od 6 aż do 26, wykonują się w czasie $O(m')$, jako że $i, j \in \{1, \dots, m'\}$, zaś każde wykonanie się dowolnej z trzech wymienionych pętli pociąga za sobą niezmiennie wzrost albo spadek jednej z tych dwóch wartości. Założymy teraz, że po wykonaniu się fragmentu 5–26, całkowita liczba dopuszczalnych parametrów λ jest większa niż wybrana przez nas wartość $(12 \cdot n$, gdzie n to liczba wierzchołków w grafie). Będziemy chcieli pokazać, że kroki, jakie podejmujemy w 34–47, za każdym razem prowadzą do zmniejszania się zbioru $F = \{(i, j) : L \leq d(i, j) \leq U\}$, \mathcal{H} oraz wyliczenia nowej wartości λ tak, że w pewnym momencie liczba dopuszczalnych rozwiązań jest mniejsza od $12 \cdot n$. To z kolei spowoduje przejście do drugiej części algorytmu, uruchomienie procedury $\text{IMST-BINARY-SEARCH}(G^*, T_s^*, s', F, k)$, która zwróci wynik i zakończy działanie algorytmu. Założymy zatem, że $\text{TotalCount} > 12 \cdot n$, tak abyśmy mogli oszacować faktyczną wielkość zbioru \mathcal{H} , który w tym przypadku zostanie wygenerowany, a na podstawie którego wybierzemy nowy parametr λ , będący albo następnym górnym, albo dolnym ograniczeniem dla zbioru F .

Na początek zauważmy, że jeśli $\text{TotalCount} > 12 \cdot n$, zaś parametr $K = \frac{\text{TotalCount}}{6 \cdot n}$, to liczba elementów w zbiorze \mathcal{H} jest z dołu ograniczona przez $6 \cdot n$. TotalCount reprezentuje liczbę tych wszystkich par (i, j) , które spełniają $L \leq d(i, j) \leq U$, $6 \cdot n \leq \frac{\text{TotalCount}}{K}$. Dla każdego indeksu i , do zbioru \mathcal{H} możemy maksymalnie dodać co K 'ty element $d(i, j)$ (gdy różnica $\text{MaxIndex}[j] - \text{MinIndex}[j]$ dzieli się przez K). Moc zbioru \mathcal{H} w takim przypadku jest równa $\frac{\text{TotalCount}}{K}$ i wynosi dokładnie $6 \cdot n$, gdy $K = \left\lfloor \frac{\text{TotalCount}}{6 \cdot n} \right\rfloor = \frac{\text{TotalCount}}{6 \cdot n}$, czyli gdy mianownik poprzedniego wyrażenia jest największy. Na pożytek tej analizy przyjęliśmy stałe: 12 oraz 6, ale bez trudu można zauważyc, że zwiększąc mianownik dla parametru K , wymuszymy na algorytmie coraz gwałtowniejszą eliminację dopuszczalnych parametrów λ (co może w konsekwencji prowadzić do niepo-



żądanej, wspomnianej wcześniej, sytuacji), zaś decydując się na modyfikację warunku $TotalCount > 12 \cdot n$ — wpływamy tylko na moment, w którym algorytm zadecyduje o zaprzestaniu zawężania zbioru Λ i uruchomimy procedurę wyszukiwania binarnego IMST-BINARY-SEARCH (G^*, T_s^*, s', F, k). Co warto podkreślić, wspomniany algorytm działa w czasie $O(m \cdot \alpha(m, n))$, gdzie m w tym przypadku równa się $TotalCount$, tak więc, zmieniając tylko stałą w wyrażeniu $12 \cdot n$, asymptotycznie nie wpływamy na czas wykonywania się tego algorytmu. Aby ograniczyć moc zbioru \mathcal{H} z góry, posłużymy się tymi samymi argumentami: najpierw zauważmy, że $K > 2$, gdyż $TotalCount > 12 \cdot n$ (zatem dla tak dobranych parametrów $K > \frac{12 \cdot n}{6 \cdot n}$). Z definicji funkcji $\lfloor \bullet \rfloor$ wiemy, że $\lfloor \frac{TotalCount}{6 \cdot n} \rfloor = \frac{TotalCount}{6 \cdot n}$, gdy $TotalCount = 12 \cdot n + 6 \cdot n \cdot k$, gdzie $k \in \mathbb{N}^+$ ($TotalCount$ musi być większe od $12 \cdot n$, toteż k musi być większe od 0). W związku z tym wartość wyrażenia $\frac{TotalCount}{\lfloor \frac{TotalCount}{6 \cdot n} \rfloor}$ osiągnie swoje maksimum wtedy, gdy $TotalCount = 12 \cdot n + 6 \cdot n \cdot k - 1$ (dla $t \in \{6 \cdot n \cdot k - 1, \dots, 6 \cdot n \cdot k - 6 \cdot n\}$ mamy $\lfloor \frac{t}{6 \cdot n} \rfloor = k - 1$, tak więc, chcąc maksymalizować wyrażenie $\frac{t}{\lfloor \frac{t}{6 \cdot n} \rfloor}$, wybierzemy $t = \max \{6 \cdot n \cdot k - 1, \dots, 6 \cdot n \cdot k - 6 \cdot n\} = 6 \cdot n \cdot k - 1$). Otrzymujemy zatem:

$$\frac{TotalCount}{K} \leqslant \frac{12 \cdot n + (6 \cdot n \cdot k - 1)}{\left\lfloor \frac{12 \cdot n + (6 \cdot n \cdot k - 1)}{6 \cdot n} \right\rfloor} = \frac{12 \cdot n + 6 \cdot n \cdot k - 1}{2 + \left\lfloor \frac{6 \cdot n \cdot k - 1}{6 \cdot n} \right\rfloor} = \frac{12 \cdot n + 6 \cdot n \cdot k - 1}{2 + (k - 1)} = \frac{n \cdot (12 + 6 \cdot k) - 1}{k + 1}.$$

Będziemy chcieli teraz znaleźć takie l , które spełnia $l = \min \left\{ l' : \frac{n \cdot (12 + 6 \cdot k) - 1}{k + 1} \leqslant l' \cdot n \right\}$, tak aby otrzymać stałą na górnego oszacowanie $|\mathcal{H}| \leqslant l \cdot n$ (do tej pory pokazaliśmy, że $6 \cdot n \leqslant |\mathcal{H}|$). Rozwiązuając to równanie otrzymamy, że najmniejszą taką wartością jest $l = 9$, gdzie nierówność $\frac{n \cdot (12 + 6 \cdot k) - 1}{k + 1} \leqslant 9 \cdot n$ zachodzi dla każdego $k \geqslant 1$ oraz $n > 0$, gdzie powiedzieliśmy wcześniej, że $k \in \mathbb{N}^+$ (z uwagi na nierówność $TotalCount > 12 \cdot n$). Zatem otrzymaliśmy, że $6 \cdot n \leqslant |\mathcal{H}| \leqslant 9 \cdot n$. Pamiętając, że $TotalCount > 12 \cdot n$, widzimy, że potrafimy zredukować tymczasowy zbiór dopuszczalnych parametrów λ (moc zbioru \mathcal{H}) o co najmniej $\frac{1}{4}$ wielkości otrzymanego zbioru Λ . Nie znaczy to jednak, że o taki współczynnik potrafimy zredukować zbiór, który będzie wyliczany w następnej iteracji dla nowych ograniczeń, z których jedno z nich przyjmie wartość parametru λ , będącego medianą z wartości $d(i, j)$ ze wszystkich par (i, j) w zbiorze \mathcal{H} .

Niech $\tilde{\lambda}$ oznacza wybraną przez algorytm medianę. Rozpatrzymy teraz dwa przypadki: w pierwszym z nich założymy, że $f(T(\tilde{\lambda}), T_s^*) > k$ (drzewo $T(\tilde{\lambda})$, obliczone dla scenariusza $s'(\tilde{\lambda})$, nie jest optymalnym rozwiązaniem problemu INCREMENTAL MINIMUM SPANNING TREE, gdyż zawiera za dużo nowych krawędzi), to znaczy, że wybrany parametr $\tilde{\lambda}$ okazał się za mały ($\tilde{\lambda} < \lambda^*$). Wiedząc, że $\tilde{\lambda}$ jest medianą wartości $d(i, j)$ ze wszystkich par $(i, j) \in \mathcal{H}$ (gdzie $\forall (i, j) \in \mathcal{H} L \leqslant d(i, j) \leqslant U$), możemy powiedzieć, że co najmniej połowa z par $(i, j) \in \mathcal{H}$ spełnia $d(i, j) \leqslant \tilde{\lambda}$ (w przypadku, gdy $f(T(\tilde{\lambda}), T_s^*) > k$, jesteśmy zainteresowani tym, by do następnego zbioru parametrów λ trafiały tylko takie pary (i, j) , dla których wartość $d(i, j)$ jest większa od $\tilde{\lambda}$, więc chcemy policzyć jaką część parametrów będziemy mogli odrzucić). Z drugiej strony, wiedząc, że moc zbioru \mathcal{H} jest z dołu ograniczona przez $6 \cdot n$, możemy wyciągnąć wniosek, że par, którymi nie jesteśmy zainteresowani, jest co najmniej $\frac{1}{2} \cdot 6 \cdot n = 3 \cdot n$. Pamiętając, że zbiór \mathcal{H} powstał w wyniku wybierania co K 'tych elementów ze zbioru $F = \{(i, j) : L \leqslant d(i, j) \leqslant U\}$, możemy oszacować liczbę par w następnym zbiorze F' na co najmniej $3 \cdot n \cdot K$ (jako że będziemy brać pod uwagę wszystkie możliwe pary, nie co K 'tą, tak jak w przypadku zbioru \mathcal{H}). To z kolei daje nam możliwość oszacowania tej wartości z dołu:

$$3 \cdot n \cdot K = 3 \cdot n \cdot \left\lfloor \frac{TotalCount}{6 \cdot n} \right\rfloor \geqslant \frac{TotalCount}{2} - 3 \cdot n \geqslant \frac{TotalCount}{4}, \quad (5.29)$$

gdzie skorzystaliśmy z faktu, iż $TotalCount > 12 \cdot n$:

$$\begin{aligned} & 3 \cdot n \cdot \left\lfloor \frac{TotalCount}{6 \cdot n} \right\rfloor \geqslant \frac{TotalCount}{2} - 3 \cdot n \leftrightarrow \\ \leftrightarrow & 3 \cdot n \cdot \left\lfloor \frac{TotalCount}{6 \cdot n} \right\rfloor - \frac{TotalCount}{2} + 3 \cdot n \geqslant \frac{3 \cdot n \cdot TotalCount}{6 \cdot n} - 3 \cdot n - \frac{3 \cdot TotalCount}{6} + 3 \cdot n \geqslant 0 \leftrightarrow \\ & \leftrightarrow \frac{3 \cdot n \cdot TotalCount}{6 \cdot n} - \frac{3 \cdot n \cdot TotalCount}{6 \cdot n} \geqslant 0. \end{aligned}$$



Ostatecznie zatem otrzymujemy, że w następnej iteracji algorytmu zbiór dopuszczalnych wartości parametru λ będzie co najmniej o $\frac{1}{4}$ mniejszy (parametrów λ takich, że $\lambda < \tilde{\lambda} < \lambda^*$ jest co najmniej $\frac{\text{TotalCount}}{4}$).

Z drugiej zaś strony założymy, że $f\left(T\left(\tilde{\lambda}\right), T_{\mathbf{s}}^*\right) < k$. Opierając się niemal na identycznym rozumowaniu, możemy dojść do wniosku, że minimalna liczba par (i, j) , dla których $d(i, j) \geq \tilde{\lambda} > \lambda^*$, wynosi $2 \cdot n \cdot K$, gdzie następnie możemy z dołu ograniczyć tę wielkość przez:

$$2 \cdot n \cdot K \geq \frac{\text{TotalCount}}{3} - 3 \cdot n \geq \frac{\text{TotalCount}}{6},$$

wyrażając w ten sposób ich liczbę w zależności od dotychczasowej liczby parametrów TotalCount , korzystając w międzyczasie z podobnej argumentacji co dla 5.29. Z tego zaś ograniczenia łatwo możemy odczytać, że z iteracji na iterację, wielkość zbioru dopuszczalnych parametrów λ będzie za każdym razem malała o co najmniej $\frac{1}{6}$ mocy zbioru F z poprzedniej iteracji, co ostatecznie prowadzi nas do wniosku, że w pewnym momencie na pewno spełniony zostanie warunek $\text{TotalCount} \leq 12 \cdot n$, co spowoduje przejście algorytmu do końcowej jego fazy, która na pewno zakończy działanie po pewnym skończonym, określonym wielkością pozostałego zbioru, czasie.

Podsumowując, zaprezentowany algorytm 5 w najgorszym przypadku jest zmuszony rozwiązać $O(\log(n))$ problemów minimalnego drzewa rozpinającego, gdzie każde jego rozwiązanie zajmuje $O(n \cdot \alpha(n, n))$. Na samym początku musimy także wyliczyć początkowe drzewa $T_{\mathbf{s}}^*$ oraz $T_{\mathbf{s}'}^*$ (w celu konstrukcji grafu $G^* = (V, E^*)$), co zajmuje nam $O(m \cdot \alpha(m, n))$. Reszta operacji algorytmu, jak pokazaliśmy, wykonuje się w czasie co najwyżej liniowym od rozmiaru zadanych danych, więc w notacji dużego O są pochłonięte przez inne czynniki. Ostatecznie otrzymujemy, że czas działania algorytmu $\text{INCREMENTAL-MST}'(G^*, T_{\mathbf{s}}^*, \mathbf{s}', k, L, U)$ wynosi: $O(m \cdot \alpha(m, n) + \log(n) \cdot n \cdot \alpha(n, n))$, w przypadku zastosowania algorytmu Chazelliego, gdzie konstruowanie grafu G^* wliczamy w czas trwania tego algorytmu (traktujemy jako wstępne przetwarzanie danych na jego potrzeby).

5.6 Podsumowanie rozdziału

Wychodząc od funkcji celu, którą otrzymaliśmy w części poświęconej programowaniu liniowemu, w tym rozdziale bardzo dokładnie omówiliśmy proces konstrukcji algorytmu rozwiązującego problem minimalnego drzewa rozpinającego w wersji INCREMENTAL, pochyłając się nad wszystkimi problemami, jakie w tym czasie napotkaliśmy i sugerując sposoby ich rozwiązania. Pozwoliło nam to otrzymać bardzo wydajny algorytm, z którego to będziemy korzystać przy próbie przyjrzenia się następnemu problemowi: optymalizacji odpornej z możliwością poprawy rozwiązania, który zdefiniowaliśmy w 3.19. O ile zagadnienie, któremu poświęciliśmy cały ten rozdział, znajduje się jeszcze w zasięgu naszych możliwości obliczeniowych (należy do problemów klasy P — ma wielomianowy algorytm go rozwiązujący), to ten, któremu będziemy się chcieli przyjrzeć, wykracza daleko poza nie [20], w związku z czym nasze dalsze rozważania nie skupią się na konstrukcji algorytmu dającego dokładne jego rozwiązania — będziemy chcieli zaprezentować nieco odmienny punkt widzenia, przedstawiając algorytmy lokalnego przeszukiwania.



Odporna optymalizacja w sąsiedztwie

Posiadamy już wszystkie potrzebne nam narzędzia, aby móc pochylić się nad głównym problemem, który interesował nas od samego początku, a który wymagał od nas stopniowego wprowadzania coraz to bardziej złożonych instancji problemów: od problemu minimalnego drzewa rozpinającego, przez problemy: IMST, AIMST, aż po problem odpornej optymalizacji z możliwością poprawy rozwiązania, którym to zajmiemy się w poniższym rozdziale. Nic jednak to nastąpi, będziemy chcieli krótko podsumować dotychczas zebrane informację, by później przejść do opisu, wspomnianych już wcześniej, algorytmów lokalnego przeszukiwania dla problemu RRIMST. Jedną z podstawowych i bardzo wydajnych metod konstrukcji rozwiązań problemów optymalizacyjnych jest metoda oparta na przeszukiwaniu bezpośrednich **sąsiadów** aktualnie posiadanego rozwiązania z nadzieją, że takie krokowe zachowanie się algorytmu będzie skutkowało szybkim zbliżaniem się do optymalnego rozwiązania. W poniższym rozdziale wskażemy trzy takie metody, przy czym jedną z nich — tą najefektywniejszą — omówimy bardziej szczegółowo. Choć wydawać by się mogło, że taka metoda rozwiązywania problemów optymalizacyjnych nie ma prawa się dobrze zachowywać (ze względu na bardzo silne założenia dotyczące sposobu zachowywania się rozwiązań sąsiednich względem obecnie badanego), to właśnie takie algorytmy — szczególnie **Tabu Search** — są najogólniejszym sposobem na rozwiązywanie problemów optymalizacyjnych.

6.1 Odporna optymalizacja z możliwością poprawy

Tak jak to wielokrotnie przytaczaliśmy, matematyczny opis problemu RECOVERABLE ROBUST INCREMENTAL MINIMUM SPANNING TREE, składa się z wielu, dotychczas przez nas omówionych, komponentów, a u którego podstawa leży, przedstawiony w rozdziale 2, problem minimalnego drzewa rozpinającego dla grafu $G = (V, E)$. Następnym naszym krokiem było zwrócenie uwagi na nieco bardziej złożony problem przyrostowy dla grafu G , gdzie ograniczaliśmy się do poszukiwań takich rozwiązań, które nie różniły się w znaczący sposób od początkowego rozwiązania, które wybieraliśmy na samym początku. Zalożyliśmy wtedy, iż takim scenariuszem, dla którego wybieraliśmy wspomniane rozwiązanie, będzie scenariusz s_0 , i że to od minimalnego drzewa rozpinającego $T_{s_0}^*$ będzie zależeć rozwiązanie problemu IMST, wybierane na podstawie kolejnych scenariuszy, oryginalnego rozwiązania $T_{s_0}^*$ oraz parametru k , który określa jak wiele krawędzi może zawierać nowe rozwiązanie, których $T_{s_0}^*$ nie zawiera. Następnym naszym krokiem było przedstawienie schematu konstrukcji rozwiązania dla problemu adwersarza (patrz Rozdział 3) z dyskretnym zbiorem scenariuszy $S = \{s_1, s_2, \dots, s_l\}$, gdzie zadaniem adwersarza było wybranie takiego scenariusza, z dostępnego dla niego zbioru S , aby najbardziej korzystne rozwiązanie problemu minimalnego drzewa rozpinającego w wersji INCREMENTAL ze scenariuszem początkowym s_0 było jak najgorsze z jego perspektywy. Ustaliliśmy, że dzięki specyficznej definicji tego problemu, istnieje możliwość jego błyskawicznego rozwiązania (dzięki możliwości podzielenia zadań na osobne wątki procesora), w czasie niewiele większym niż wcześniej wspomnianego problemu IMST, którego rozwiązanie potrafieliśmy znaleźć w czasie wielomianowym (sposób, w jaki to robiliśmy, przedstawiony został w rozdziale 5). Następnym naszym krokiem będzie budowa algorytmu, który umożliwi nam poradzenie sobie z problemem, którego matematyczny opis przypominamy w równaniu 6.1 — jak zostało pokazane, problem ten jest już na tyle złożony, że nie tylko jest NP-trudny, lecz nie jest on w żaden sposób aproksymowalny [13, twierdzenie 6]. Oznacza to, że nie istnieje algorytm, gdzie zwracane przez niego rozwiązania moglibyśmy traktować jako co najwyżej p razy gorsze niż rozwiązanie optymalne (tak jak to miało miejsce w przypadku problemu MIN-MAX dla dyskretnych zbiorów scenariuszy — zobacz twierdzenia: 3.2.1 i 3.2.2). W związku z tym musimy znaleźć inny sposób na radzenie sobie z postawionym przed nami



problemem.

6.2 Algorytm zachłanny

Algorytm zachłanny jest najbardziej podstawową i naturalną strategią poszukiwania rozwiązania dla problemów optymalizacyjnych w myśl zasady „jeśli wartość mojego rozwiązania jest niewiele gorsza od wartości optymalnej, za pomocą niewielkich zmian powiniensem tą wartość osiągnąć”. Fundament, na którym zbudowane jest to zdanie, zawiera bardzo ważne założenie, określające charakter zachowania się poszczególnych rozwiązań problemu optymalizacyjnego. Zakładamy bowiem, że proporcjonalnie do wielkości zmian, wprowadzonych w znalezionym rozwiązańiu, rośnie (bądź maleje) koszt takiego rozwiązania. Takim problemem jest na przykład rozpatrywane przez nas zagadnienie minimalnego drzewa rozpinającego, który leży u podstaw problemu, na którego rozwiązanie poświęcimy cały poniższy rozdział:

$$\min_{\mathbf{x} \in X} \left(v(\mathbf{x}, \mathbf{s}) + \max_{\mathbf{s}' \in S} \min_{\mathbf{y} \in X_{\mathbf{x}}^k} v(\mathbf{y}, \mathbf{s}') \right). \quad (6.1)$$

6.2.1 Sąsiedztwo

Rozpoczniemy wyjątkowo od przedstawienia pseudokodu algorytmu zachłannego (zobacz Pseudokod 6)¹, by następnie na jego podstawie wytlumaczyć pojęcie **sąsiedztwa**. Przypomnijmy, że funkcja $v(\bullet, \mathbf{s})$, zdefiniowana w drugim rozdziale, reprezentuje koszt rozwiązania problemu dla scenariusza \mathbf{s} (gdzie jako pierwszy parametr przekazywaliśmy jej binarny wektor \mathbf{x} , reprezentujący to rozwiązanie). W odniesieniu do problemu minimalnego drzewa rozpinającego, zamiast wektora, będziemy przekazywać do niej drzewo — wartością zaś tej funkcji będzie suma kosztów krawędzi należących do tego drzewa. Dla rozróżnienia kosztu rozwiązania problemu minimalnego drzewa rozpinającego w wersji INCREMENTAL od tego dla wyrażenia 6.1, wprowadźmy oddzielne oznaczenia: $v_{IMST}(\bullet, \mathbf{s})$ dla tego pierwszego oraz $v_{RRIMST}(\bullet, S)$ dla problemu odpornej optymalizacji przyrostowej z możliwością poprawy (ang. RRIMST — *Robust Recoverable Incremental Minimum Spanning Tree*). Należy wziąć pod uwagę, że policzenie wartości tego drugiego wymaga od nas w między czasie rozwiązania szeregu problemów IMST oraz problemu adwersarza — drugi z nich, wraz ze sposobem jego rozwiązania, przedstawiliśmy w 3.4.2. Dlatego też zamiast przekazywać do funkcji pojedynczy scenariusz, tak jak to robiliśmy do tej pory, do wyrażenia $v_{RRIMST}(\bullet, \bullet)$ będziemy przekazywać zbiór scenariuszy dla problemu adwersarza. Mając na uwadze dotychczas omówione zagadnienia, nie będziemy więcej wracać do problemu wyliczania wartości tej funkcji, skupimy się zaś na algorytmach zwracających aproksymację wyrażenia $\min_{\mathbf{x} \in X} v_{RRIMST}(\mathbf{x}, S) = \min_{\mathbf{x} \in X} (v(\mathbf{x}, \mathbf{s}) + \max_{\mathbf{s}' \in S} \min_{\mathbf{y} \in X_{\mathbf{x}}^k} v(\mathbf{y}, \mathbf{s}'))$.

W przedstawionym pseudokodzie, linii 4, jesteśmy zainteresowani pozyskaniem ze zbioru $N(T, T_s^*, k)$ najlepszego rozwiązania dopuszczalnego dla problemu IMST. Jednocześnie, przekazując do niego drzewo T , chcemy, aby zwrócone rozwiązanie było do niego w określony sposób podobne. Skalę tego podobieństwa dla różnych problemów definiujemy inaczej, dla problemu minimalnego drzewa rozpinającego jakim się zajmujemy, zbiór, generowany przez powyższe wyrażenie, przybierze formę:

$$N(T, T_s^*, k) = \{T' : f(T', T_s^*) \leq k \wedge f(T', T) = 1\} \quad (6.2)$$

i oznacza zbiór wszystkich tych drzew będących dopuszczalnymi rozwiązaniami problemu IMST ($f(T', T_s^*) \leq k$), które jednocześnie są **sąsiadami** drzewa T . W naszym przypadku sąsiadem T nazwiemy dowolne drzewo T' , które różni się od tego pierwszego dokładnie jedną krawędzią. Wybór takiego drzewa T' spośród zbioru drzew $N(T, T_s^*, k)$ będziemy nazywać **ruchem** — przejściem z jednego rozwiązania dopuszczalnego dla danego problemu do drugiego. Główna pętla algorytmu zachłannego 6 będzie powtarzana do momentu, w którym żadne sąsiednie rozwiązanie nie okaże się lepsze od tego, na podstawie którego wygenerowaliśmy otoczenie (stąd sama nazwa algorytmu — zachłanny). Powstaje naturalne pytanie — jak generować kolejne

¹Ten i pozostałe pseudokody, chociaż dotyczące uniwersalnych algorytmów, będą przedstawiane z perspektywy problemu minimalnego drzewa rozpinającego (tak więc np. zamiast punktu startowego \mathbf{x} z dziedziny dopuszczalnych rozwiązań X , będziemy mieli początkowe drzewo rozpinające T_s^*). Dodatkowo, gdy będzie to uzasadnione, będziemy korzystać z wcześniej udowodnionego lematu 5.2.1 (zamiast grafu $G = (V, E)$, będziemy odwoływać się do $G^* = (V, E^*)$).

Pseudokod 6: LOCAL-SEARCH (G^*, T_s^*, S, k)

Wejście: $G^* = (V, E^*)$ — graf ze zbiorem krawędzi $T_{s'}^* \cup T_s^*$,
 T_s^* — początkowe minimalne drzewo rozpinające dla scenariusza s ,
 S — zbiór scenariuszy adwersarza,
 k — parametr problemu IMST.

Wyjście: T^* — lokalnie optymalne rozwiązanie problemu IMST.

```
1 begin
2   Wybierz dowolne drzewo  $T$ , będące dopuszczalnym rozwiązaniem problemu IMST.
3   while znaleziono lepsze rozwiązanie do
4      $T \leftarrow \min arg T' \{v_{RRIMST}(T', S) : T' \in N(T, T_s^*, k)\}$ 
5     if  $v_{RRIMST}(T, S) < v_{RRIMST}(T_s^*, S)$  then
6        $T_s^* \leftarrow T$ 
7     else
8       return  $T_s^*$ 
```

drzewa rozpinające, które na dodatek są sąsiadami innego, wskazanego przez nas drzewa T ? Jednym z pomysłów na realizację tego zadania może być konstrukcja nowych drzew poprzez dodawanie do nich krawędzi nie należących do T a usuwaniu tych, które razem z dodaną przed chwilą krawędzią tworzą cykl [14].

Pseudokod 7: NEIGHBORHOOD (G, T)

Wejście: $G = (V, E)$,
 T — minimalne drzewo rozpinające dla grafu G .

Wyjście: $N(T)$ — otoczenie drzewa T .

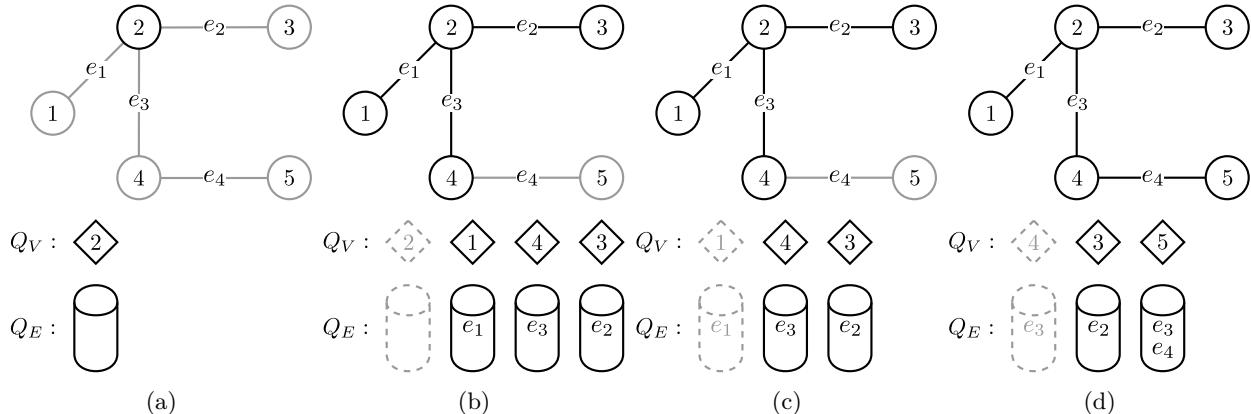
```
1 begin
2    $N(T) \leftarrow \emptyset$ 
3   foreach  $(i, j) \in E \setminus T$  do
4     foreach  $e \in v_i \overset{T}{\rightsquigarrow} v_j$  do
5        $N(T) \leftarrow N(T) \cup (T \cup e_{ij} \setminus e)$ 
6   return  $N(T)$ 
```

Przez wyrażenie $e \in v_i \overset{T}{\rightsquigarrow} v_j$ (w linii 4 pseudokodu 7) rozumiemy wszystkie takie krawędzie e , które należą do ścieżki, której początek znajduje się w wierzchołku v_i a kończy w v_j . Dodatkowo jesteśmy zainteresowani tylko takimi krawędziami, które należą do wskazanego przez nas drzewa T . Tą własność możemy bardzo łatwo wymusić, szukając ścieżek tylko w obrębie danego drzewa rozpinającego (jako że z definicji łączy ono ze sobą wszystkie wierzchołki, na pewno istnieje choć jedna ścieżka pomiędzy wskazanymi wierzchołkami v_i oraz v_j). Algorytmem, służącym nam do generowania takich zbiorów, może być na przykład zmodyfikowany algorytm przeszukiwania wszerz [6, 604–606]. Nasza modyfikacja polegałaby na tym, że zamiast jednej kolejki priorytetowej, posiadałibyśmy takich kolejek dwie, z czego pierwsza zachowałaby swoje oryginalne własności, elementami drugiej byłyby zbiory krawędzi reprezentujące ścieżki, które doprowadziły do konkretnego elementu z pierwszej listy, tak jak to pokazano na rysunkach od 6.1a do 6.1d.

6.2.2 Wady algorytmu zachłannego

Jak każdy algorytm zachłanny, i ten boryka się z tymi samymi problemami: lokalny charakter jego decyzji powoduje, że bardzo łatwo zatrzymać mu się w **lokalnym optimum** $T^{(*)}$, które charakteryzuje się tym, że dla dowolnego drzewa $T' \in N(T, T_s^*, k)$ mamy $v_{RRIMST}(T^{(*)}, S) \leq v_{RRIMST}(T', S)$. W takiej sytuacji algorytm

²Jako że dodawany wierzchołek jest wierzchołkiem końcowym ścieżki $v_s \rightsquigarrow v_t$, zamiast aktualizować obydwie kolejki, możemy odpowiadający mu (wierzchołkowi v_t) zbiór krawędzi od razu przenieść do puli znalezionych ścieżek bez modyfikacji kolejek i przejść do następnego elementu.



Rysunek 6.1: Kolejne kroki zmodyfikowanego algorytmu BFS (T, v_s, v_t) (ang. *Breadth-First Search*), gdzie T jest drzewem rozpinającym graf G , v_s oraz v_t są odpowiednio wierzchołkami początkowym i końcowym ścieżki $v_s \rightsquigarrow v_t$, dla której chcemy wyznaczyć zbiór krawędzi należących do T i do tej ścieżki. (a) Początkowa sytuacja algorytmu dla $v_s = v_2$ i $v_t = v_5$. Na koniec kolejki Q_V został dodany wierzchołek początkowy. Odpowiadający mu element w kolejce V_E nie zawiera żadnych elementów. (b) W wyniku zdjęcia z kolejki pierwszego elementu v_2 , algorytm dodał do Q_V wszystkie wierzchołki, do których prowadziły krawędzie ze zdążytego wierzchołka. Odpowiednio dla każdego takiego wierzchołka, do kolejki Q_E zostały dodane krawędzie, dzięki którym algorytm do nich dotarł. (c) Zgodnie z kolejnością ściągania elementów z kolejek, aktualnie badanymi elementami (zaznaczonymi w kolejce jasnym kolorem) są: wierzchołek v_1 oraz zbiór krawędzi $\{e_1\}$. Z tego wierzchołka nie prowadzi jednak żadna nowa krawędź, badany wierzchołek nie jest też tym końcowym, toteż algorytm usuwa wskazane elementy. (d) W wyniku wykonania operacji na kolejnej parze elementów: $(v_4, \{e_3\})$, do kolejki priorytetowej Q_V została dodany² nowy, jeszcze nieodwiedzony wierzchołek $v_5 = v_t$. Równolegle, do kolejki Q_E dla dodanego wierzchołka, został dodany zbiór krawędzi będący sumą zbioru łuków, po których algorytm dotarł do analizowanego wierzchołka v_4 , oraz łuku, po którym przeszedł on do nowego wierzchołka (zbiór $\{e_3, e_4\}$). Fakt, że dodawany do kolejki Q_V element jest wierzchołkiem końcowym w ścieżce $v_s \rightsquigarrow v_t$, oznacza, że podany zbiór krawędzi jest jedną z możliwych ścieżek (w tym przypadku jedyną) pomiędzy tymi wierzchołkami. Algorytm kontynuujemy dopóki obie kolejki nie są puste, zapisując po drodze elementy z Q_E , które odpowiadają wierzchołkowi v_t , dodawanym do kolejki Q_V (zapisujemy te ścieżki, których ostatnia krawędź prowadzi do v_t).

zachłanny oczywiście zakończy działanie z przekonaniem, że znalezione przez niego rozwiązanie jest optymalne globalnie, choć w rzeczywistości przeglądnięta przez niego przestrzeń rozwiązań dopuszczalnych jest bardzo mała i mogła „nie otrzymać” się nawet o prawidłowe rozwiązanie. Oczywiście konsekwencją powyższego jest w pełni deterministyczny schemat działania takiego algorytmu, tak więc jakość otrzymanego rozwiązania w pełni zależy od zdanego przez nas początkowego drzewa T_s^* . Dodatkowo algorytm nie uczy się — raz przeglądnięte przez niego rozwiązanie może być przeglądane dowolną liczbę razy, jeżeli zdarzy się, że dane rozwiązanie jest najlepsze w sąsiedztwie wielu innych drzew rozpinających. Jak zobaczymy w części poświęconej algorytmowi lokalnego przeszukiwania z listą ruchów zakazanych (ang. *Tabu Search*), mimo tak licznych i poważnych wad, algorytm ten jeszcze nam posłuży do konstrukcji dużo lepszego algorytmu.

6.3 Symulowane wyżarzanie

Algorytm **symulowanego wyżarzania** (ang. *Simulated annealing*) jest algorymem mocno zbliżonym do poprzednio omawianego, jednak eliminuje pewne wady wspomnianego rozwiązania. Przede wszystkim rezygnuje on z przeglądania wszystkich sąsiadów zadanego drzewa T (co wymuszało na nas wyrażenie $\min_{\text{arg}T'} \{v_{\text{RRIMST}}(T', S) : T' \in N(T, T_s^*, k)\}$ w linii 4 pseudokodu 6). Zamiast tego algorytm wprowadza pojęcie **temperatury**, która jest składową miary prawdopodobieństwa wybrania konkretnego sąsiada drzewa T z otoczenia $N(T, T_s^*, k)$. Na jej podstawie, zamiast przeglądać całość otoczenia $N(T, T_s^*, k)$, algorytm lo-



suje z niego drzewo T' , oblicza wartość takiego rozwiązania ($v_{RRIMST}(T', \mathbf{s})$), oraz na podstawie tej danej oraz temperatury decyduje się na wybór takiego drzewa, bądź losuje następne, do momentu, w którym dla wylosowanego drzewa T' i przekazanego do funkcji wyliczającej prawdopodobieństwo jego wybrania, zwrocona przez nią wartość jest dostatecznie wysoka. Algorytm rozpoczyna pracę od pewnej temperatury maksymalnej, a następnie ją sukcesywnie obniża, powodując spadek wartości obliczanych prawdopodobieństw [21] [11].

Zaletami takiego rozwiązania w porównaniu do poprzedniego algorytmu są niewątpliwie: brak konieczności przeglądania wszystkich sąsiednich rozwiązań (przeglądamy kolejne losowe rozwiązania do momentu, w którym zdefiniowane wcześniej reguły pozwolą nam na wybranie jednego z nich) oraz większa odporność na pozostanie w obszarze lokalnego optimum (ze względu na wprowadzoną losowość wyboru następnego rozwiązania z sąsiedztwa). Nadal jednak taka szansa istnieje, dlatego skupimy się teraz na trzecim i najbardziej złożonym rozwiążaniu, które w zamian nie jest obarczone wadami poprzedników — algorytmem lokalnego przeszukiwania z listą ruchów zakazanych.

6.4 Przeszukiwanie z listą Tabu

Algorytm **Tabu Search** [9] swoją nazwę zawdzięcza, wprowadzanemu przez siebie, dodatkowemu elementowi jakim jest **lista ruchów zakazanych** (ang. *tabu list*). Taką listą będziemy nazywać zbiór ruchów, których z różnych względów bronimy algorytmowi wykonać w trakcie poszukiwania rozwiązania — w ten sposób, jeżeli **kadencja** takiego elementu na liście *tabu* będzie wystarczająco długa (przez pojęcie kadencji rozumiemy liczbę iteracji algorytmu jaką ten musi wykonać, aby mieć możliwość ponownego wyboru zakazanego ruchu), zapewniamy sobie brak wpadania przez algorytm w cykle podczas poszukiwania rozwiązania³. Dodatkowo będziemy chcieli rozwiązać kwestię groźby trafienia i pozostania w lokalnym minimum — aby temu zapobiegać, będziemy co jakiś czas resetować nasz algorytm, który w takiej formie (upraszczając) można przyrównać do wielokrotnego uruchamiania algorytmu zachłannego, za każdym razem dla innych drzew początkowych. Przy tym wszystkim będziemy oczywiście chcieli, abyśmy w przypadku odnalezienia najlepszego rozwiązania do tej pory, mogli ruch do niego prowadzący, ignorując listę ruchów zakazanych, wykonać. O rozwiązaniu, będącym efektem wykonania zabronionego ruchu, będziemy mówić, że spełniało ono **kryterium aspiracji**.

Aby zapewnić sobie przegląd jak największej przestrzeni rozwiązań dopuszczalnych problemu IMST, będziemy chcieli, aby każde następne drzewo, będące punktem wyjścia po restarcie algorytmu, było jak najbardziej oddalone od drzew do tej pory przeanalizowanych. Innymi słowy będziemy chcieli zaradzić kolejnej wadzie, którą wykazaliśmy przy okazji analizy algorytmu zachłannego — stosunkowo niewielkiego obszaru rozwiązań, które ten algorytm badał, zanim zatrzymywał się w lokalnym optimum. Dla algorytmu TABU SEARCH nie jest inaczej — podczas jednego pełnego przebiegu jesteśmy w stanie przeanalizować niewielką tylko część dopuszczalnych rozwiązań, dlatego sztucznie będziemy wymuszać rozpoczęwanie wyszukiwania rozwiązania od nowa, zaczynając szukać w innych miejscach, tak jak to zaprezentowano na pseudokodzie 8.

6.4.1 Ocena ruchu

Chcąc w trakcie działania algorytmu wykonywać ruchy pomiędzy kolejnymi rozwiązaniami, musimy sobie odpowiedzieć na pytanie: jakie chcemy wybrać kryterium, które będzie decydowało o tym, który ruch będący wybierali. W najprostszym przypadku, tak jak to ma miejsce w algorytmie zachłannym, naszym kryterium jest po prostu funkcja $v_{RRIMST}(T, \mathbf{s})$, która zwraca wartość wyrażenia $v(T, \mathbf{s}) + \max_{\mathbf{s}' \in S} \min_{T' \in \mathcal{T}_T^k} v(T', \mathbf{s}')$ dla konkretnego drzewa T . Optymalnym ruchem w tym przypadku nazwiemy taki wybór $T_1 \in N(T, T_{\mathbf{s}}^*, k)$, który zaowocuje najmniejszą wartością powyższego wyrażenia, gdzie jego wartość zależy tylko od wybieranego drzewa. W kolejnym rozdziale pokażemy (patrz wykresy 7.2), że wybór tej strategii dla algorytmu TABU SEARCH, w istocie nadaje mu zachowanie identyczne, co w przypadku algorytmu zachłannego (taki też jego opis, bardzo uproszczony, przytoczyliśmy wcześniej). Aby wykorzystać potencjał, drzemiący w omawianym

³Analogicznie jak w omawianym wcześniej, zmodyfikowanym algorytmie BFS (T, v_s, v_t) (patrz rysunek 6.1), tak w przypadku algorytmu TABU SEARCH, algorytm nie podejmie decyzji powtórzenia pewnych ruchów, gdyż będzie wiedział, że taki ruch już wykonał (można powiedzieć, że lista *tabu* jest częściowym zapisem historii wykonywanych ruchów).



algorytmie, będziemy stosować inną funkcję, stanowiącą kryterium wyboru ruchów. Niech dana będzie zatem funkcja $Mval(T, T_1)$, gdzie $T_1 \in N(T, T_s^*, k)$, o następującej definicji:

$$Mval(T, T_1) = \alpha_1 \cdot (v_{RRIMST}(T, S) - v_{RRIMST}(T_1, S)) + \alpha_2 \cdot \frac{R[i, j]}{it} + \alpha_3 \cdot \frac{R[k, l]}{it} + \alpha_4 \cdot MR[i, j] + \alpha_5 \cdot MR[k, l], \quad (6.3)$$

gdzie jej kolejnymi elementami składowymi są:

- $v_{RRIMST}(T, S) - v_{RRIMST}(T_1, S)$ — wskaźnik poprawy wartości rozwiązania dla wybranego drzewa T_1 względem starej, liczonej dla drzewa T , z którego szukamy najlepszego dla nas ruchu,
- $R[i, j]$ oraz $R[k, l]$ przechowują informacje na temat liczby wystąpień krawędzi e_{ij} oraz e_{kl} w dotychczas wybieranych rozwiązaniach (gdzie ruch polega na usunięciu z drzewa T krawędzi e_{ij} a dodaniu e_{kl}),
- $MR[i, j]$ oraz $MR[k, l]$ oznaczają odpowiednio średnie wartości funkcji $v_{RRIMST}(T_{ij}, S)$ oraz $v_{RRIMST}(T_{kl}, S)$, gdzie T_{ij} oraz T_{kl} oznaczają te wybrane wcześniej rozwiązania, w których występuje krawędź e_{ij} lub e_{kl} . Parametr it wskazuje na numer aktualnej iteracji, zaś
- parametry $\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5$ są dowolne i mogą być eksperymentalnie zdeterminowane. Z uwagi na fakt, że w tak zdefiniowanej funkcji, większe jej wartości odpowiadają lepszym wybieranym rozwiązaniami, wyszukanie najlepszego ruchu w sąsiedztwie drzewa T oznaczać będzie taki wybór T_1 , dla którego wartość $Mval(T, T_1)$ jest największa.

Widzimy, że tak zdefiniowana funkcja oceny ruchu daje nam dużo większe możliwości wpływu na rozwiązania: zależnie od dobranych parametrów możemy położyć większy nacisk na każdorazowe poprawianie wyników (gdy α_1 jest odpowiednio duża w stosunku do pozostałych parametrów), bądź inne właściwości dotychczasowych rozwiązań. Jeśli zatem sprawą drugorzędną jest dla nas to, by wartość rozwiązania dla każdego następnego drzewa była lepsza ($v_{RRIMST}(T, S) - v_{RRIMST}(T_1, S) > 0$), manipulując odpowiednio parametrami $\alpha_1, \alpha_2, \alpha_3, \alpha_4$ czy α_5 , możemy albo położyć większy nacisk na częstotliwość pojawiania się danej krawędzi wchodzącej do rozwiązań (im częściej będzie się pojawiać, bądź tym lepsze średnie wartości dzięki niej będziemy otrzymywać, tym bardziej może nam się opłacić wybierać takie następne rozwiązania, które daną krawędź także zawierają), albo z nich usuwanej (α_3, α_5). Z drugiej strony zaś, jeżeli wartość α_1 będzie na tyle znacząca, by mieć wpływ na całą wartość funkcji $Mval(T, T_1)$, ryzyko pozostania przez algorytm TABU SEARCH w lokalnym optimum jest niewielkie (jeśli wartość α_1 dominuje nad pozostałymi, wtedy — w przypadku, gdy algorytm „utknął” w takim punkcie — wartości członu $\alpha_1 \cdot (v_{RRIMST}(T, S) - v_{RRIMST}(T_1, S))$ są bardzo małe, bądź można je przyrównać do zera⁴). To właśnie tej funkcji oceny ruchu poświęcimy dłuższy fragment rozdziału, w którym będziemy testować zachowanie się algorytmu dla różnych parametrów.

6.4.2 Sąsiedztwo

Analizując pseudokod 8, nietrudno nie zauważyc, że w linijce 9 nagromadziliśmy bardzo dużą liczbę warunków — w tym miejscu założyliśmy również, że skoro jesteśmy zainteresowani tylko takim sąsiadem T' drzewa T , dla którego wartość $Mval(T, T')$ jest największa spośród wszystkich wartości $Mval(T, T'')$ dla drzew $T'' \in N(T, T_s^*, k)$, to nie potrzebujemy pamiętać jednocześnie pozostałych drzew w sąsiedztwie⁵. W naszym przypadku zatem drzewem T_N jest drzewo które:

- należy do bezpośredniego sąsiedztwa drzewa T (różni się od niego dokładnie jedną krawędzią) oraz ma nie więcej niż k krawędzi, które nie należą do drzewa początkowego T_s^* (czyli $T_N \in N(T, T_s^*, k)$),
 - ruch polegający na przejściu od T do innego dopuszczalnego rozwiązania T' nie znajduje się na liście TABU ($T' \notin TABU$),

⁴Poprzez zwrot „utknął” mamy na myśli sytuację, w której algorytm na przemian wykonuje powtarzające się ruchy, gdyż dla nich funkcja $v_{RRIMST}(\bullet, \bullet)$ zwraca najmniejsze wartości spośród całego otoczenia dla rozwiązań, pomiędzy którymi algorytm się cyklicznie porusza — jedynym ratunkiem w takiej sytuacji byłoby „zresetowanie” algorytmu.

⁵Tak jak na przykład pseudokod algorytmu TABU SEARCH w [14], gdzie są rozdzielone kroki: wygenerowanie całego zbioru $N(T, T_s^*, k)$ oraz wybranie z niego drzewa o jak największej wartości funkcji oceny ruchu.



- lub wartość rozwiązania problemu RRIMST dla drzewa T' jest lepsza (mniejsza) niż najmniejsza taka wartość znaleziona do tej pory (kryterium aspiracji — $v_{RRIMST}(T', S) < C^{\Delta*}$),
- $Mval(T, T')$ zwraca największą wartość spośród wszystkich drzew, które spełniają wszystkie powyższe kryteria ($T_N \leftarrow \max arg_{T'} \{Mval(T, T') : \dots\}$).

W przypadku, gdy zbiór $N(T, T_s^*, k)$ jest na tyle duży, że wykonanie powyższych obliczeń dla każdego drzewa należącego do tego zbioru w sposób nieakceptowalny wpływa na czas wykonywania się algorytmu, możemy zastosować dodatkowo **kryterium aspiracji plus**, które przyjmuje parametry:

- \min — najmniejsza liczba drzew z sąsiedztwa, które musimy przeglądać ($\min \{ \min, |N(T, T_s^*, k)| \}$),
- \max — liczba drzew ze zbioru $N(T, T_s^*, k)$, po których przeglądnięciu musimy zwrócić drzewo, dla którego wartość wyrażenia $Mval(T, \bullet)$ była największa do tej pory — reszty drzew rozpinających nie przeglądamy ($\max = \min \{ \max, |N(T, T_s^*, k)| \}$),
- v_{\min} — próg aspiracji, po którego przekroczeniu przez wartość $Mval(T, \bullet)$ dla pierwszego drzewa (drzewo T^+ i niech będzie ono i tym przeglądanym drzewem), będziemy przeglądać jeszcze p następnych w kolejności sąsiadów drzewa T ze zbioru $N(T, T_s^*, k)$, gdzie
- $p = \min \{i + p, \max\}$.

W takiej sytuacji, zamiast wykonywać obliczenia dla wszystkich drzew w sąsiedztwie, przerywamy je w chwili, gdy zajdą następujące warunki:

- przeglądnijmy \max drzew lub
 - wartość funkcji $Mval(T, \bullet)$ dla pewnego drzewa T^+ jest większa bądź równa niż ustalony z góry parametr v_{\min} oraz
 - wykonamy potrzebne obliczenia jeszcze dla p następnych drzew, należących do $N(T, T_s^*, k)$.

W tym przypadku zwróconym drzewem T_N będzie albo najlepsze spośród pierwszych \max drzew, albo pierwsze takie drzewo (T^+), dla którego $Mval(T, \bullet) \geq v_{\min}$, gdzie v_{\min} jest parametrem stosowanego kryterium aspiracji, albo najlepsze znalezione po nim ($T_N = \max arg_{T'} \{Mval(T, T') : T' \in \{T^+, T^{+2}, \dots, T^{+p}\}\}$).

6.4.3 Losowe drzewo rozpinające i strategia dywersyfikacji

Omawiając cele jakie chcemy zrealizować implementując algorytm TABU SEARCH, wspomnialiśmy także o części resetowania algorytmu i rozpoczętania jego pracy na nowo, lecz z rozwiązaniem początkowym T jak najdalej oddalonym od któregokolwiek drzewa, które pojawiło się w procesie wykonywania się iteracji poprzedniej. W tym celu wprowadziliśmy następujące oznaczenia: E^c oraz T^c , gdzie ten pierwszy jest konstruowany z sumy tych drugich. T^c zaś będziemy nazywali **alternatywą najgorszego przypadku** dla drzewa T — innymi słowy będzie to minimalne drzewo rozpinające (znalezione przez zwykły algorytm rozwiązyjący problem MST) dla scenariusza $S^-(T)$ o następujących kosztach:

$$c_i^{S^-(T)} = \begin{cases} \max \{c_i^s : s \in S\} & e_i \in T, \\ \min \{c_i^s : s \in S\} & e_i \notin T, \end{cases} \quad \forall i \in \{1, 2, \dots, m\}. \quad (6.4)$$

Celem takiego zabiegu jest wymuszenie zwrócenia przez algorytm minimalnego drzewa rozpinającego, którego część wspólna z drzewem T jest jak najmniejsza (które jest jak najbardziej oddalone od zbioru sąsiadów drzewa T — $\{T' : f(T', T) = 1\}$). Widzimy, że ilekroć poprawiamy dotychczas odnalezione rozwiązanie (poza sytuacją, w której resetujemy algorytm i „przez przypadek” nowo wygenerowane drzewo okazuje się być tym najlepszym — linie 20–25), do zbioru E^c dołączamy krawędzie wygenerowanych alternatyw dla tych rozwiązań ($E^c \leftarrow E^c \cup T_N^c$). Taka strategia skutkuje tym, że po it_{max} iteracjach rozpoczętym poszukiwanie rozwiązania w zupełnie innym fragmencie przestrzeni rozwiązań dopuszczalnych, przez co mamy większą szansę „natknąć” się na lokalne optimum, które jest również optymalne globalnie.

Generowanie losowych drzew rozpinających dany graf (linie 2 oraz 20) jest zadaniem prostym — w tym przypadku również możemy skorzystać ze zmodyfikowanej wersji algorytmu BFS, bądź dowolnego innego,

**Pseudokod 8:** TABU-SEARCH (G, T, S, k, it_{max})

Wejście: $G = (V, E)$,
 T_s^* — początkowe minimalne drzewo rozpinające dla scenariusza s ,
 S — zbiór scenariuszy adwersarza,
 k — parametr problemu IMST.
 it_{max} — liczba iteracji, po której następuje zresetowanie algorytmu.

Wyjście: $T^{\Delta*}$ — najlepsze znalezione do tej pory rozwiązanie problemu RRIMST.

```

1 begin
2    $T \leftarrow \text{RANDOM-MST}(G)$                                 // Wybierz początkowe, losowe drzewo rozpinające graf  $G$ .
3    $T^{\Delta*} \leftarrow T$ 
4    $C^{\Delta*} \leftarrow v_{\text{RRIMST}}(T, S)$ 
5    $TABU \leftarrow \emptyset$ 
6    $E^c \leftarrow T^c$ 
7    $it \leftarrow 0$ 
8   while nie zaszło kryterium zatrzymania do
9      $T_N \leftarrow$ 
10     $\max arg_{T'} \{ Mval(T, T') : T' \in N(T, T_s^*, k) \wedge (T' \notin TABU \vee v_{\text{RRIMST}}(T', S) < C^{\Delta*}) \}$ 
11     $C_N \leftarrow v_{\text{RRIMST}}(T_N, S)$ 
12     $it \leftarrow it + 1$ 
13    if  $C_N < C^{\Delta*}$                                 // Znaleziono lepsze rozwiązanie.
14       $T^{\Delta*} \leftarrow T_N$ 
15       $C^{\Delta*} \leftarrow C_N$ 
16       $E^c \leftarrow E^c \cup T_N^c$ 
17       $it \leftarrow 0$ 
18    if  $it > it_{max}$                                 // Przekroczono liczbę iteracji.
19    then
20       $T \leftarrow \text{RANDOM-MST}((V, E^c))$ 
21       $C \leftarrow v_{\text{RRIMST}}(T, S)$ 
22      if  $C < C^{\Delta*}$                                 // Jeśli losowo wybrane, nowe początkowe drzewo rozpinające jest najlepsze.
23      then
24         $T^{\Delta*} \leftarrow T$ 
25         $C^{\Delta*} \leftarrow C$ 
26       $TABU \leftarrow \emptyset$                                 // Resetowanie algorytmu.
27       $E^c \leftarrow T^c$ 
28       $it \leftarrow 0$ 
29    else
30       $T \leftarrow T_N$ 
31      UPDATE-TABU()
32 return  $T^{\Delta*}$ 
```

błędzącego po grafie w sposób losowy (choć istnieją algorytmy od nich efektywniejsze [24]). Najprostsza implementacja, wraz z odwiedzaniem kolejnych krawędzi, po prostu losowo wybiera wychodzącą z niego krawędź spośród dostępnych, dbając o to, aby zakończyć ten proces w chwili wybrania ostatniej, $(|V| - 1)$ ej krawędzi (odwiedzając przy tym każdy wierzchołek tylko raz).

6.4.4 Lista ruchów zakazanych i kryterium końca

Ostatnim elementem, wymagającym omówienia, jest lista TABU. Dzięki niej jesteśmy w stanie uniknąć zbyt częstego powtarzania tych samych ruchów czy wymusić na algorytmie przegląd coraz to dalszego sąsiedztwa danego drzewa początkowego w kolejnych iteracjach. Niestety, zastosowanie listy ruchów zakazanych



ma swoje wady: fakt istnienia takiej listy może oznaczać, że w pewnych sytuacjach najbardziej optymalny ruch jest zakazany i nie będziemy mogli go wykonać (prawdopodobieństwo takiego zdarzenia jest tym większe im na dłuższe kadencje się zdecydujemy). Przeciwdziałać takim przypadkom ma, wprowadzone przez nas, kryterium aspiracji, które w sytuacjach takich jak opisana pozwala nam całkowicie zignorować listę TABU. Wokół samego wyboru długości kadencji poszczególnych elementów listy gromadzi się wiele problemów; z jednej strony powiedzieliśmy, że zbyt długie kadencje mogą powodować pomijanie pewnych optymalnych rozwiązań, lecz dzięki temu, że niektóre ruchy są zakazane przez bardzo długi okres, mamy szansę przeglądając szerszy obszar rozwiązań dopuszczalnych wokół punktu startowego. Pociąga to za sobą również wady — przeszukując coraz większy obszar przy niezmiennej liczbie iteracji, tracimy na jakości uzyskiwanych wyników (badany zbiór rozwiązań jest rzadszy, przez co większe jest prawdopodobieństwo pominięcia rozwiązania, które powinniśmy wybrać). Z kolei skłanianie się ku krótkim kadencjom podnosi co prawda naszą szansę na znalezienie optymalnego rozwiązania, lecz wraz z nią rośnie zagrożenie, że zwróconym rozwiązaniem będzie optimum lokalne, bądź algorytm wpadnie w cykl.

Z każdą następną iteracją, wraz z wywołaniem procedury UPDATE-TABU, pozostały czas kadencji każdego z elementów powinien ulec zmniejszeniu, zaś ruchy, których czas kadencji dobiegł końca — usunięte z listy, co czyni je ponownie możliwymi do wyboru. Dodatkowo, jako najprostsze kryterium końca algorytmu, możemy przyjąć warunek sprawdzający, czy całkowita liczba iteracji nie została przekroczona. Jeżeli tak, zwrócimy najlepsze do tej pory odnalezione rozwiązanie.

6.4.5 Podsumowanie rozdziału

Algorytm TABU SEARCH jest tylko ideą — większa część pokazanego pseudokodu jest opcjonalna i tylko od tego, do jakiego problemu się tę ideę zastosuje, zależy finalny kształt tego algorytmu. Wszystkie podane parametry, takie jak okres kadencji, liczba iteracji, kryterium zatrzymania się algorytmu czy też parametry aspiracji plus, czy choćby poszczególne wskaźniki funkcji oceny ruchu (jak i też sama funkcja) — powinny być rozpatrywane dla każdego problemu oddzielnie i nie istnieje ogólna formuła wskazująca ich wartości. W przypadkach takich jak ten, którym się zajmujemy, gdy nie istnieje żaden algorytm wielomianowy, który zapewniałbym nam zwracanie poprawnego rozwiązania (czy to optymalnego, czy gorszego od niego o stały współczynnik), algorytm TABU SEARCH jest jednak bardzo dobrą alternatywą, o której efektywności przekonamy się w kolejnym rozdziale, testując działanie algorytmu dla różnych wartości jego parametru, pokazując związki między nimi oraz analizując złożoności obliczeniowe problemów, które leżą u podstaw zagadnienia RECOVERABLE ROBUST INCREMENTAL MINIMUM SAPNNING TREE.



Wyniki eksperymentalne

W tym rozdziale pochylimy się nad eksperymentalnymi wynikami, które uzyskaliśmy dla wybranych przypadków testowych. Jako że problem RECOVERABLE ROBUST INCREMENTAL MINIMUM SPANNING TREE sam składa się z wielu podproblemów, część eksperymentalną zaczniemy właśnie od nich, by przekonać się między innymi o różnicach, które wynikają z zastosowania różnych algorytmów, do rozwiązywania tych samych instancji problemu. W rozdziale 5 przedstawialiśmy dwa różne podejścia do problemu INCREMENTAL MINIMUM SPANNING TREE, nie mówiąc już o rozwiązaniu, które zaproponowaliśmy pod koniec omawiania zagadnień, związanych z programowaniem liniowym — porównamy eksperymentalnie czasy działania dwóch pierwszych. Jak się okaże, czas, po którym dowolny z przytoczonych przez nas modeli programowania liniowego/całkowitoliczbowego zwraca optymalne rozwiązanie, jest zbyt duży, by jego porównywanie z poprzednimi dwoma algorytmami miało dla nas jakąkolwiek wartość (zobacz wykresy 7.1a oraz 7.1a). W przypadku problemu adwersarza, który również jest częścią składową dla RECOVERABLE ROBUST INCREMENTAL MINIMUM SPANNING TREE, będziemy chcieli przekonać się, czy podział zadań na osobne wątki (tak jak to sugerowaliśmy w rozdziale 3) rzeczywiście jest dla nas opłacalny. Wreszcie na końcu będziemy chcieli sprawdzić, jak dla różnych instancji grafów zachowuje się, przedstawiony w poprzednim rozdziale, algorytm TABU SEARCH dla szerokiej gamy konfiguracji jego parametrów.

7.0.1 Środowisko testowe

O ile dla samego algorytmu TABU SEARCH nie jesteśmy zainteresowani jego rzeczywistą złożonością obliczeniową (wiemy, że algorytm działa w czasie wielomianowym, co automatycznie czyni go szybszym od wszelkich innych prób dokładnego rozwiązania NP-trudnego problemu, jakim jest RECOVERABLE ROBUST INCREMENTAL MINIMUM SPANNING TREE), tak w przypadku algorytmów przez nas zaproponowanych, służących do rozwiązywania problemów minimalnego drzewa rozpinającego w wersji INCREMENTAL jak i problemu adwersarza, będziemy chcieli uzyskać nadające się do interpretacji wyniki. W związku z tym nasze wszystkie eksperymenty będące przeprowadzać na, użycznym do tego celu, serwerze Politechniki Wrocławskiej o nazwie OTRYT, wspieranego przez 80 jednostek INTEL® XEON® CPU E7- 4850 @ 2.00GHz, zaopatrzonych w 256 GB pamięci RAM i 6,3 TB przestrzeni dyskowej, działającego pod kontrolą systemu operacyjnego LINUX DEBIAN w wersji 7.7.

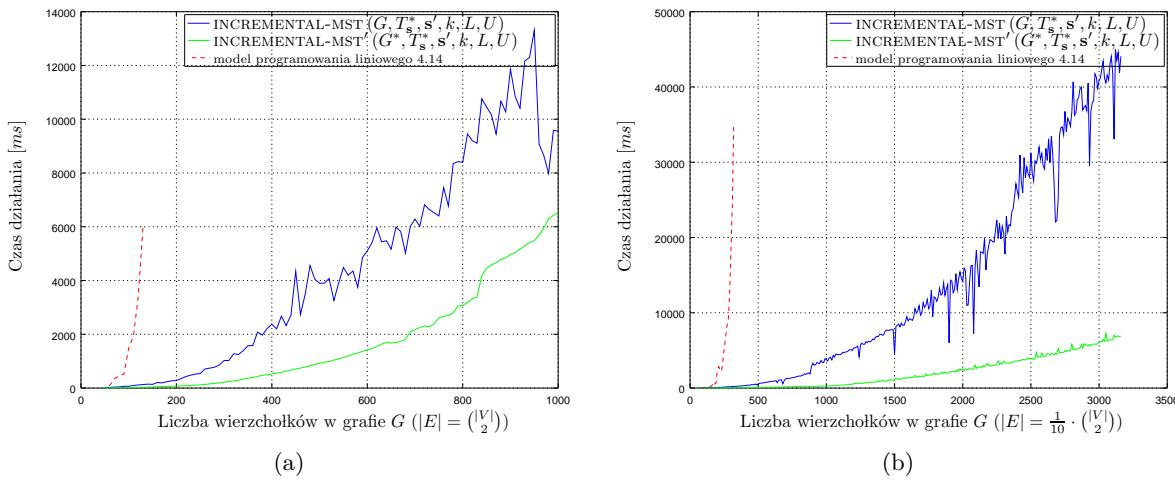
Testy stabilności algorytmów i inne pomniejsze testy zostały zaś wykonane na lokalnej maszynie pod kontrolą systemu LINUX UBUNTU w wersji 15.10 z 4 GB pamięci dynamicznej oraz 74 GB przestrzeni dyskowej, zarządzanej przez procesor INTEL® CORE™ i5 CPU M 540 @ 2.53GHz. Ze względu na różnice systemów obu środowisk, zdecydowano się na komplikowanie rozwiązania przy pomocy starszej wersji kompilatora G++ (4.9.2) (gdzie pierwotnie rozwiązania były budowane przy pomocy tego narzędzia w wersji 5.2.1).

7.1 Problem INCREMENTAL

Wykorzystane przez nas przykłady będą głównie opierać się na jednym z parametrów grafu, jakim jest jego gęstość — w przypadku problemu e oraz wszystkich problemów pochodnych (AIMST oraz RRIMST), gdzie całość rozwiązania opiera się na wyborze takiego, które spełnia pewne określone własności względem innego rozwiązania (oryginalnego), kluczowym parametrem, decydującym w głównej mierze o zachowaniu się dowolnego z przedstawianych poniżej algorytmów, jest bez wątpienia gęstość grafu, o czym się przekonamy, analizując chociażby wykresy dotyczące algorytmu lokalnego przeszukiwania z listą ruchów zakazanych TABU



SEARCH. Na tę chwilę wystarczy, że zauważymy, iż im większa jego gęstość, tym więcej krawędzi w grafie, tym wyższe stopnie posiadają wierzchołki¹, a co za tym idzie — algorytm, którego zadaniem jest zwrócić najlepsze drzewo rozpinające T^* , które nie może się różnić od drzewa początkowego T_0 o więcej niż k krawędzi, na każdym etapie będzie musiał podejmować coraz więcej decyzji (zauważmy, że bez względu na gęstość grafu, liczba krawędzi składających się na jego dowolne drzewo rozpinające, nie ulega zmianie, co znaczy, że wraz ze wzrostem gęstości, rośnie także stosunek $\frac{|E|}{|T|}$, co oczywiście przekłada się bezpośrednio na czas działania takiego algorytmu). Szczególnie widoczne jest to w przypadku, gdy gęstość grafu jest na tyle mała, że sam ten fakt zaczyna „sugerować” poprawne rozwiązania (to znaczy liczba krawędzi jest na tyle mała, że zbiór $\{e : e \in T_0\}$, gdzie T_0 jest początkowym rozwiązaniem problemu IMST, zaczyna stanowić większość krawędzi w grafie). Na wykresach 7.1a oraz 7.1b uchwyciono tą zależność, gdzie niebieskim kolorem zaznaczono wykresy podstawowego algorytmu rozwiązujejącego problem t , zaś druga krzywa z pary przedstawia czas, w jakim z tymi samymi instancjami problemu poradził sobie algorytm, w którym zastosowaliśmy wszystkie nasze uwagi, dotyczące możliwości poprawienia jego efektywności względem pierwotnego algorytmu.



Rysunek 7.1: Wykres zależności czasu pracy algorytmów: INCREMENTAL-MST (G, T_s^*, s', k, L, U) oraz INCREMENTAL-MST' $(G^*, T_s^*, s', k, L, U)$ od liczby krawędzi w grafie.

W tym drugim przypadku możemy zauważać nie tylko większą regularność otrzymywanych rezultatów (czas, potrzebny na rozwiązywanie coraz to większych instancji problemu przez drugi z algorytmów, jest dużo bardziej proporcjonalny do rozmiaru danych wejściowych), lecz także dużo mniejszą zależność od liczby wierzchołków w grafie (wystarczy przywołać linie 2–9 pseudokodu 4, opisanego w rozdziale 5, aby zobaczyć, że czas wykonywania się wspomnianego fragmentu kodu stanowi znaczny ułamek całkowitego czasu działania algorytmu, gdzie dany kod jest zależny w głównej mierze od liczby wierzchołków w grafie²).). Jak możemy zauważać w przypadku tego algorytmu, czasy jego działania dla obu przedstawionych konfiguracji są niemal identyczne, co może świadczyć o tym, że głównym wyznacznikiem złożoności obliczeniowej dla niego jest liczba krawędzi grafu (gdzie w przypadku obu wykresów ich liczby są porównywalne: $\binom{1000}{2} \approx \binom{3610}{2} \cdot \frac{1}{10}$), nie zaś wierzchołków. Z analizy złożoności jednak wiemy, że czas działania opisywanego algorytmu to w najlepszym przypadku (gdy przyjmiemy, że potrafimy rozwiązać problem minimalnego drzewa rozpinającego w czasie $O(m \cdot \alpha(m, n))$) czas rzędu $O(m \cdot \alpha(m, n) + \log(n) \cdot n \cdot \alpha(n, n))$ — zatem przyczyną, dla której omawiany algorytm działa w podobnym czasie zarówno dla grafu z tysiącem wierzchołków jak i dla takiego z liczbą 3160 węzłów, musimy szukać gdzie indziej. Przyczyna takiego a nie innego zachowania jest z goła duża prostsza do wyjaśnienia — fakt tak niewielkiej, w porównaniu z grafem pełnym, gęstości powoduje, że na samym już początku z większym prawdopodobieństwem otrzymamy dwa drzewa T_s^* oraz $T_{s'}^*$, których część wspólna będzie większa, niż tych samych drzew odnalezionych dla grafu pełnego. Stąd, dla niewielkiej gęstości grafu, algorytm, by znaleźć optymalne rozwiązanie problemu e , musi wykonać dużo mniejszą liczbę kroków, niż w przypadku grafu pełnego (im większa początkowa część wspólna minimalnych drzew rozpinających T_s^*

¹Stopniem wierzchołka określamy liczbę krawędzi, z którymi jest bezpośrednio połączony.

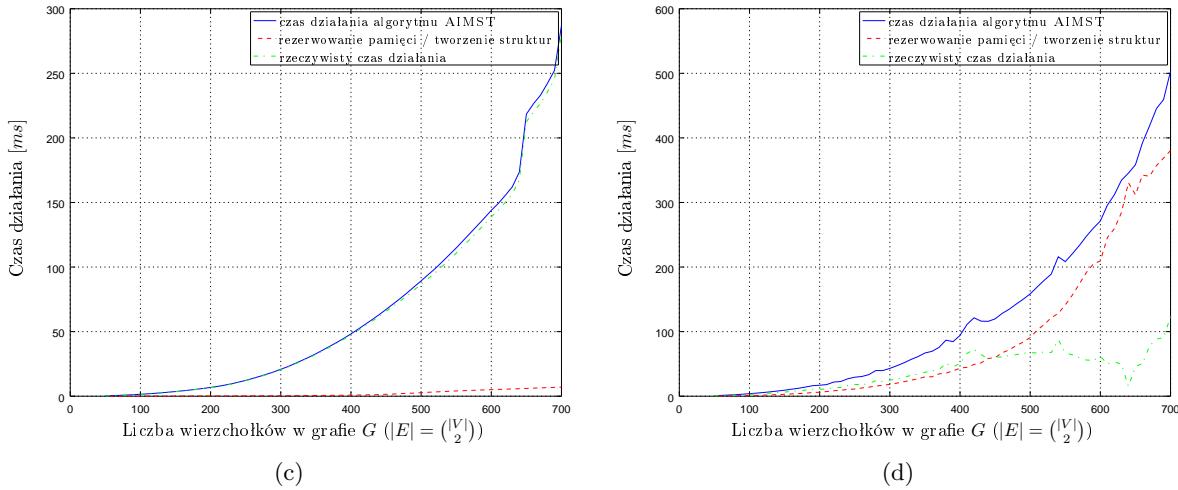
²Całość wspomnianego kodu wykonuje się w pętli $|T_s^* \setminus T_{s'}^*|$ razy, gdzie $|T_s^*| = |T_{s'}^*| = |V| - 1$.

oraz $T_{\mathbf{s}'}^*$, tym mniejszy zbiór $\Lambda = \left\{ c'_{e_i} - c'_{e_j} : e_i \in T_{\mathbf{s}}^* \setminus T_{\mathbf{s}'}^* \wedge e_j \in T_{\mathbf{s}'}^* \setminus T_{\mathbf{s}}^* \right\}$ do przeglądnienia ma algorytm). Jak widzimy, zaznaczony przerywaną, czerwoną linią, czas obliczeń, potrzebny do znalezienia rozwiązania dla modelu liniowego, prezentowanego w rozdziale 4 (patrz model 4.14), jest znacznie większy, w porównaniu do czasu działania algorytmów, które powstały w oparciu o ten model.

Wspomniane wykresy otrzymano dla grafów, których liczba krawędzi wahała się od 50 do 3160, gdzie dla każdego z nich eksperymenty zostały powtórzone 100 razy, aby wyeliminować wpływ czynników losowych na postać generowanych wykresów. Mimo tego musimy zdawać sobie sprawę, że wszystkie dane były generowane w sposób losowy — ze względu na charakterystykę rozpatrywanego problemu nie było zatem możliwe zapewnienie większej skali podobieństwa pomiędzy kolejnymiinstancjami grafu, niż przedstawiono na wykresach (zdecydowano się za parametrem k przyjąć wartość wprost proporcjonalną do liczby krawędzi, składających się na drzewo rozpinające grafu).

7.2 Problem adwersarza

Kolejnym problemem, stojącym na drodze do problemu odpornej optymalizacji z możliwością poprawy, jest problem adwersarza, który omawialiśmy w rozdziale 3 (patrz podrozdział 3.4.2) — wspomnieliśmy wtedy tylko o tym, że algorytm, rozwiązujący dane zagadnienie, ze względu na charakterystykę matematycznego sformułowania, opisującego problem ($\max_{\mathbf{s}' \in S} \min_{\mathbf{y} \in X_{\mathbf{x}}^k} v(\mathbf{y}, \mathbf{s}')$), może zostać zaimplementowany z wykorzystaniem mechanizmu wątków, które umożliwiają wykonywanie się obliczeń dla zawartych w nim podproblemów ($\min_{\mathbf{y} \in X_{\mathbf{x}}^k} v(\mathbf{y}, \mathbf{s}')$) w tym samym czasie, niezależnie od pozostałych (jeżeli liczba dostępnych wątków w odniesieniu do liczby scenariuszy jest wystarczająco duża i co najmniej równa liczbie tych drugich).



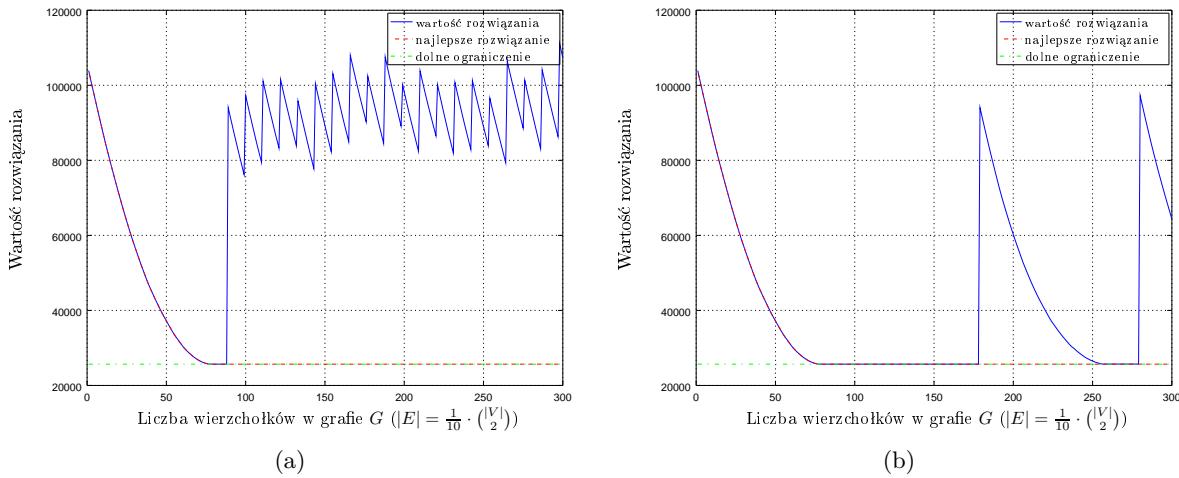
Rysunek 7.1: Czas działania algorytmu rozwiązujecego problem adwersarza dla 70 scenariuszy, wygenerowanych dla grafu pełnego. Na każdym z wykresów przedstawiono czas działania algorytmu (niebieska linia), na który składają się takie procesy jak: alokacja pamięci oraz tworzenie wymaganych struktur danych (czarna, przerywana linia) oraz sam proces obliczeń, gdzie przebieg tego ostatniego został zaznaczony przerywaną, zieloną linią. (a) Czas działania algorytmu w przypadku uruchomienia go bez wsparcia dla wielu wątków — algorytm sekwencyjnie wykonuje obliczenia dla każdego z 70 scenariuszy, (b) w odróżnieniu od sytuacji na drugim wykresie, gdzie zezwolono algorytmowi na wykorzystanie pełnej równoległości, gdzie każdy z 70 scenariuszy może być rozpatrywany niezależnie.

Na wykresach 7.1c oraz 7.1d przedstawiono dwa sposoby podejścia do rozpatrywanego problemu — na drugim z nich widzimy algorytm z, zaproponowany przez nas, wykorzystaniem wielu wątków procesorów, zaś pierwszy ukazuje czas działania tego samego algorytmu, tyle że pracującego sekwencyjnie. Co zaskakujące, czas działania tego ostatniego jest dużo lepszy niż dla algorytmu wykorzystującego wielowątkowość — wyjaśnienie tego zjawiska jest jednak oczywiste, jeżeli weźmiemy pod uwagę fakt, iż aby możliwe były

równoległe obliczenia na grafie, gdzie zmieniają one jego właściwości (koszty krawędzi), muszą one być wykonywane na niezależnych od siebie obiektach, co już na samym początku wydłuża czas działania całego algorytmu o czas potrzebny na K -krotne skopiowanie całej struktury grafu, gdzie K to liczba scenariuszy adwersarza. W rozpatrywanym przypadku celowo wykorzystaliśmy graf pełny, aby spotęgować ten negatywny efekt rozbicia pracy algorytmu na wiele wątków. W grafie pełnym liczbę krawędzi szacujemy na $|V|^2$, co stanowi niemal najbardziej znaczący składnik złożoności algorytmu IMST, jaką otrzymaliśmy pod koniec rozdziału 5 — $O(m \cdot \alpha(m, n) + \log(n) \cdot n \cdot \alpha(n, n))$, gdzie $m \cdot \alpha(m, n) \approx m$. W przypadku gdy nie braliśmy pod uwagę czasu potrzebnego na te operacje (czas ten zaznaczyliśmy czerwonym kolorem na wykresach 7.1c oraz 7.1d), bądź rozwiązały problem w inny sposób (np. zbudowali nasze struktury od podstaw z myślą o wielowątkowości, co pozwoliłoby nam przenieść znaczną część obliczeń, związanych z powielaniem struktur danych, na równoległe wątki³), w oczywisty sposób drugi z przedstawionych wariantów okazałby się bezkonkurencyjny (rzeczywisty czas działania, dla którego nie braliśmy pod uwagę czasu, poświęconego na organizowanie danych dla algorytmu, zaznaczyliśmy na wykresach kolorem zielonym).

7.3 Tabu Search

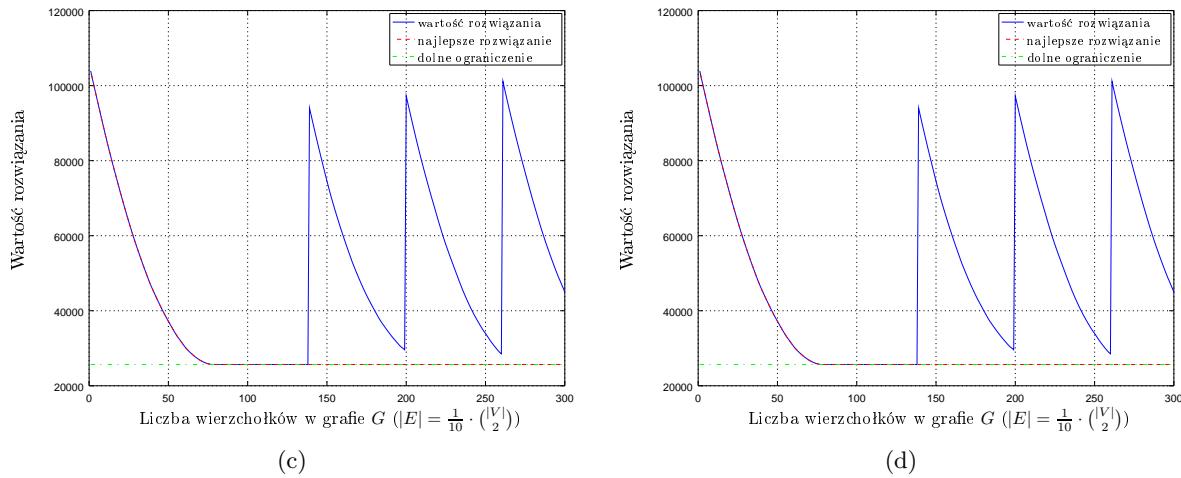
Tak jak to miało miejsce wyżej, także dla problemu odpornej optymalizacji dyskretnej z możliwością poprawy dla minimalnego drzewa rozpinającego przy wykorzystaniu algorytmu lokalnego przeszukiwania, gęstość grafu będzie miała niebagatelne znaczenie dla przebiegu całego procesu obliczeniowego. Jak mogliśmy zaobserwować w poprzedniej części, w przypadku grafów pełnych pomysł zrównoleglania okazał się bardzo niekorzystny, ze względu na konieczność wielokrotnego przekopiowywania dużej ilości danych (zwłaszcza, kwadratowej w stosunku do V , liczby krawędzi), w celu oddzielenia od siebie obliczeń.



Rysunek 7.2: Wartości rozwiązań zwarcane przez algorytm TABU SEARCH dla grafu o 100 wierzchołkach, 495 krawędziach (gęstość grafu $q = \frac{1}{10}$), dla parametru $k = 10$, współczynników funkcji oceny ruchu (patrz 6.3) $\alpha_1 = \alpha_2 = \alpha_3 = \alpha_4 = \alpha_5 = \frac{1}{2}$. Długość kadencji każdego z elementów na liście ruchów zabronionych wynosiła 20 iteracji, gdzie po (a) 10, (b) 60 iteracjach, w przypadku braku polepszenia wartości dla najlepszego do tej pory znalezionego rozwiązania, algorytm porzucał obecnie badaną ścieżkę, rozpoczynając poszukiwanie rozwiązań w innym obszarze ich przestrzeni.

Dla algorytmu TABU SEARCH możemy zaobserwować zgoła odwrotną własność — zle jego zachowanie jest efektem wprowadzenia, w charakterze danych wejściowych, grafu o zbyt małej gęstości. Wykresy 7.2a, 7.2b, 7.2c oraz 7.2d ukazują sposób zachowania się algorytmu dla tak wprowadzonych danych — ze względu na niewielki stosunek liczby krawędzi do wierzchołków, algorytm TABU SEARCH jednostajnie podąża w kierunku

³Musimy rozwiązać problem jednoczesnego dostępu do danych oryginalnego grafu, gdyż prezentowane przykłady wykorzystują struktury, które w takich przypadkach wymuszają sekwencyjne odczytywanie danych przez każdy wątek po kolei, co oczywiście znacznie osłabia potęgę, drzemiącą w podejściu równoległym do problemu.



Rysunek 7.2: Wartości rozwiązań zwracane przez algorytm TABU SEARCH dla grafu o 100 wierzchołkach, 495 krawędziach (gęstość grafu $q = \frac{1}{10}$), dla parametru $k = 10$, współczynników funkcji oceny ruchu (patrz 6.3): $\alpha_1 = \alpha_4 = \alpha_5 = \frac{1}{10}$ oraz $\alpha_2 = \alpha_3 = \frac{6}{5}$. Długość kadencji każdego z elementów na liście ruchów zabronionych wynosiła (c) 0 (nie obowiązywała w ogóle), (d) 180 iteracji, gdzie po 60 iteracjach, w przypadku braku polepszenia wartości dla najlepszego do tej pory znalezionego rozwiązania, algorytm rozpoczynał poszukiwanie innych rozwiązań od początku (w możliwie najbardziej oddalonym od obecnego rozwiązania punkcie ich przestrzeni).

coraz to lepszych rozwiązań, zamiast mieć możliwość przejścia do innych, bardziej oddalonych rozwiązań. Takie zachowanie się algorytmu w obliczu tak zadanych danych jest bardzo ciekawe, gdyż sugeruje, że w takich przypadkach TABU SEARCH zaczyna się zachowywać jak algorytm zuchlanny, gdzie pierwsze części obu wykresów 7.2a oraz 7.2b wizualizują właśnie takie zachowanie — algorytm w pierwszej iteracji za każdym razem decyduje się na, wybrany z sąsiedztwa aktualnego rozwiązania, ruch, który skutkuje poprawieniem wartości funkcji celu. Jak możemy zauważyć, wydłużenie czasu trwania pojedynczej iteracji algorytmu nie wpływa na sposób jego zachowania — w pewnym momencie algorytm dociera do rozwiązania, które wydaje się być minimum lokalnym, z którego nie ma już ucieczki. Następstw takiego zachowania się algorytmu TABU SEARCH jest kilka: na wykresach 7.2c oraz 7.2d przedstawiliśmy jego przebieg dla różnych konfiguracji, między innymi zmieniliśmy współczynniki funkcji oceny ruchu (porównaj opisy 7.2a i 7.2c) a także okresy kadencji elementów na liście ruchów zakazanych (porównaj 7.2c i 7.2d), gdzie żadna z tych zmian (szczególnie ta ostatnia) zdaje się nie mieć większego wpływu na, znajdywane w kolejnych iteracjach, rozwiązania. Dzieje się tak dlatego, iż (tak jak to już zauważaliśmy) algorytm z każdym nastepnym krokiem znajduje rozwiązanie, którego wartość jest lepsza od poprzedniej — za każdym razem wykonuje inny ruch, który skutkuje odnalezieniem rozwiązania, które nie pojawiło się do tej pory. W związku z tym lista ruchów zakazanych przez znaczną ilość czasu nie ma na przebieg algorytmu najmniejszego wpływu, jako że algorytm przez bardzo długi czas nie sięga po ruchy, które już kiedyś wykonał. Z tego samego powodu nasza funkcja oceny ruchu, zmiana jej współczynników, nie wywiera większego wpływu na działanie algorytmu TABU SEARCH dla tak zadanych danych — jeśli przywołamy jej definicję, zobaczymy, że w tak zaistniałych okolicznościach, bez względu na wybrany ruch, parametr K przy współczynnikach od α_2 do α_5 będzie w większości przypadków wynosił $K = 1$, co w żaden sposób nie będzie faworyzowało jednego wyboru ruchu ponad inne — dopiero w następnych iteracjach, gdy algorytm zaczyna powtarzać pewne ruchy, możemy dostrzec drobne zmiany względem przedstawionych wykresów.

7.3.1 Heurystyka a rozwiązanie optymalne

Na wykresach od 7.2a do 7.2d przedstawiliśmy koncept **dolnego ograniczenia**, który akurat we wspomnianych przypadkach zrównywał się z wartością, znajdowaną cyklicznie przez algorytm TABU SEARCH. Wielokrotnie podkreślaliśmy, że rozważany problem RECOVERABLE ROBUST INCREMENTAL MINIMUM SPANNING TREE jest problemem klasy NP i dlatego też zdecydowaliśmy się na wykorzystanie podejścia heurystycznego



— dlaczego zatem deklarujemy istnienie wartości, która dla przedstawionego przypadku jest równa rozwiązańiu, zwracanemu przez taki algorytm, co więcej, deklarując, że jest to wartość optymalna? Nasze dolne ograniczenie jest efektem pracy modelu programowania liniowego, zaprezentowanego w [10], rozwiązującego następujący problem:

$$\min \left(\sum_{e \in E_X} C_e \cdot x_e + \sum_{e \in E_Y} (c_e + d_e) \cdot y_e \right), \quad (7.1)$$

gdzie zbiory E_X oraz E_Y są rozłączne i odpowiadają kolejno: części oryginalnej rozwiązania, która nie jest jednocześnie elementem nowego, którego krawędzie reprezentuje zbiór E_Y . Wykorzystaliśmy tak skonstruowany model do własnych celów, obliczając za jego pomocą wartość poniższego wyrażenia:

$$\min_{\mathbf{x} \in X} v(\mathbf{x}, \mathbf{s}) + \max_{\mathbf{s}' \in S} \min_{\mathbf{y} \in X_{\mathbf{x}}^k} v(\mathbf{y}, \mathbf{s}') \quad (7.2)$$

dla jednego scenariusza \mathbf{s}' , którym dysponuje adwersarz. To pozwala nam na znaczne uproszczenie powyższego wzoru:

$$|S| = 1 \rightarrow \min_{\mathbf{x} \in X} v(\mathbf{x}, \mathbf{s}) + \max_{\mathbf{s}' \in S} \min_{\mathbf{y} \in X_{\mathbf{x}}^k} v(\mathbf{y}, \mathbf{s}') \equiv \min_{\mathbf{x} \in X} v(\mathbf{x}, \mathbf{s}) + \min_{\mathbf{y} \in X_{\mathbf{x}}^k} v(\mathbf{y}, \mathbf{s}'), \quad (7.3)$$

co jest dokładnie tym, co liczy wspomniany model programowania liniowego. Zwróćmy uwagę, że z tak otrzymanego wzoru nie wynika już dłużej to, że adwersarz będzie wybierał swój scenariusz na podstawie rozwiązania \mathbf{x} — uruchamiając model liniowy dla tak zdefiniowanego problemu, osobno dla każdego ze scenariuszy adwersarza, otrzymanymi wynikami będą wartości rozwiązań przypadków, w których dla każdego scenariusza adwersarza wybieraliśmy najlepsze dla nas rozwiązanie. Tak zdefiniowany problem jest równoznaczny z problemem, stawianym przez RRIMST, tylko wtedy, gdy zbiór scenariuszy, spośród których może wybierać adwersarz, zawiera tylko jeden scenariusz, bądź każdy z nich jest taki sam (lub wybrane przez nas rozwiązanie nie zależy od wyboru adwersarza, gdyż koszty krawędzi, które należą do rozwiązania, w nich wszystkich są identyczne) — taki też przypadek zachodzi dla wykresów od 7.2a do 7.2d, gdzie wykorzystaliśmy tylko dwa scenariusze, spośród których mógł wybierać adwersarz, a dodatkowo okazało się, że model, uruchomiony dla obydwu scenariuszy, zwracał identyczną wartość rozwiązania.

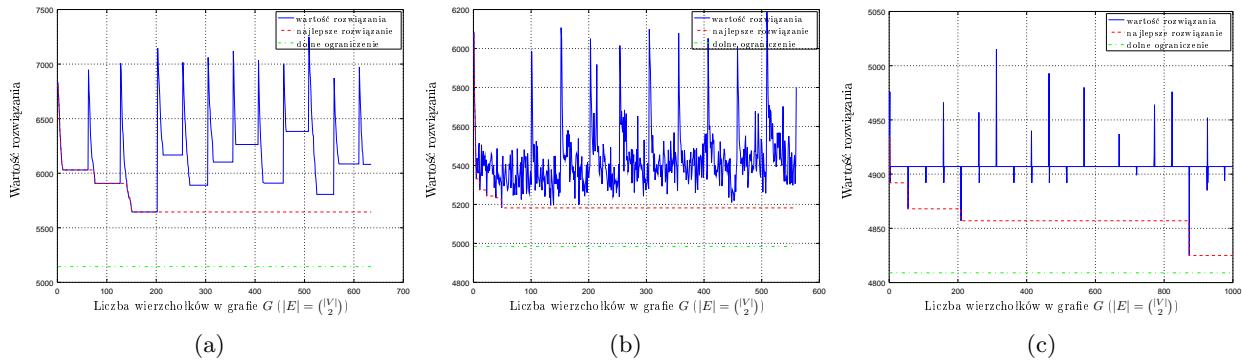
W ogólnym przypadku nie możemy jednak na to liczyć, zaś wartość, którą zwraca tak zdefiniowany model liniowy, nazywać będziemy **dolnym ograniczeniem** na wartość rozwiązania oryginalnego problemu RRIMST. Ne możemy bowiem zaprzeczyć, że znalezione przez model rozwiązanie nie jest rozwiązaniem dopuszczalnym — odpowiada ono sytuacji, w której adwersarzowi „nie zależy” na odgrywaniu swojej roli i zamiast wybierania rozwiązań dla nas najlepszych, pozwala je wybrać nam (bądź sam wybiera takie, które dla nas jest najlepsze). Tak jak opisywaliśmy to przy omawianiu relaksacji Lagrange'a (patrz dowód lematu 4.2.1), tak samo w tym przypadku chcemy uzyskać najlepsze z możliwych dolnych ograniczeń na wartość optymalną naszego problemu poprzez wybór największej wartości spośród otrzymanych, to jest:

$$LB \leq LB^* = \max_{LB \in \mathcal{LB}} LB \leq v_{RRIMST}^*, \quad (7.4)$$

gdzie LB reprezentuje dowolne z otrzymanych dolnych ograniczeń, zaś v_{RRIMST}^* — optymalną wartość rozwiązania problemu RECOVERABLE ROBUST INCREMENTAL SPANNING TREE.

7.3.2 Próg tolerancji

Progiem tolerancji dla nowego rozwiązania problemu RRIMST nazwiemy parametr k dla jednego z jego podproblemów (INCREMENTAL MINIMUM SPANNING TREE), gdzie k w oczywisty sposób pełni rolę ograniczenia, które mówi nam jak wiele może się zmienić względem pierwotnego rozwiązania w obliczu konieczności konstrukcji nowego. Innymi słowy, jeżeli wartość parametru k jest duża względem liczby wierzchołków w grafie (krawędzi w minimalnym drzewie rozpinającym), jesteśmy w stanie tolerować nawet rozwiązania takie, których część wspólna ze starym rozwiązaniem jest niewielka. Prowadzi to do skrajnie różnych zachowań algorytmu TABU SEARCH: począwszy od sytuacji, w których wartość parametru jest bardzo mała (dla $k = 1$ patrz wykres 7.3a), przechodząc przez pośrednie stany (dla $k = 7$ patrz wykres 7.3b), aż po sytuacje, w których jego wartość bliska jest $|V|$ (dla $k = 25$ patrz wykres 7.3c).



Rysunek 7.3: Wartości rozwiązań zwracane przez algorytm TABU SEARCH dla grafu o 30 wierzchołkach, 435 krawędziach (gęstość grafu $q = 1$), dla współczynników funkcji oceny ruchu (patrz 6.3): $\alpha_1 = \frac{1}{10}$, $\alpha_2 = \alpha_3 = \alpha_4 = \alpha_5 = \frac{1}{5}$, dla różnych wartości parametru k . Długość każdej iteracji dla wszystkich trzech wykresów wynosi 50. Wykresy przedstawiają przebieg algorytmu dla (a) $k = 1$, (b), $k = 7$ (c) $k = 25$, gdzie krawędzi w minimalnym drzewie rozpinającym jest 29.

Widzimy wyraźnie zależności, które występują między poszczególnymi rozwiązaniami. W przypadku gdy $k = 1$, otrzymujemy sytuację bardzo podobną do tej, którą przedstawiały wykresy od 7.2a do 7.2d — tym razem jednak takie zachowanie się algorytmu TABU SEARCH jest podyktowane bardzo niewielkimi możliwościami wpływania na rozwiązanie pomiędzy kolejnymi jego krokami, gdzie w każdej iteracji możliwa jest tylko zamiana jednej krawędzi. Z uwzględnieniem listy ruchów zakazanych, prowadzi to do tego, że algorytm może „poruszać się” w bardzo ograniczonym zakresie i jedyne, co nie jest mu zabronione, to wybieranie z iteracji na iterację coraz to lepszych rozwiązań, czego przejawem jest właśnie wykres 7.3a. Z drugiej zaś strony widzimy, że poprzez wybranie zbyt dużej wartości k w stosunku do liczby wierzchołków w grafie, powodujemy, że algorytm TABU SEARCH natychmiast zaczyna produkować te same rozwiązania, jako że praktycznie nie są one ograniczone parametrem k . Na tym przykładzie doskonale widać sytuacje, w której to losowe rozwiązanie (wygenerowane w momencie przekroczenia dopuszczalnej liczby iteracji) okazuje się być lepsze od najlepszego dotąd znalezionego, gdzie zarówno na wykresie 7.3a jak i 7.3b mamy do czynienia z taką sytuacją w bardziej naturalny sposób, gdzie polepszenia wartości rozwiązania występują w trakcie trwania obliczeń, w ramach dopuszczalnej liczby iteracji (choć i tu możemy dostrzec specyficzne zachowania, gdzie przez dość długi okres, na wykresie 7.3a, widzimy, że odnajdywane co iteracje rozwiązania są lepsze od swoich poprzedników — można wręcz powiedzieć, że wykres 7.3a w bardzo wyraźny sposób przedstawia główną ideę algorytmu TABU SEARCH, pomimo że, zastosowany dla $k = 1$, zachowuje się pomiędzy „restartami” jak algorytm zachłanny). Na samym końcu mamy wykres 7.3b, którego sposób zachowania świadczy o poprawnym dobraniu stosunków wszystkich współczynników algorytmu do tak zadanych danych.

Choć wszystkie trzy wykresy mają tylko charakter poglądowy, mają przedstawiać różnice, jakie można uzyskać w działaniu prezentowanego algorytmu, możemy dla wszystkich trzech podać miarę ich efektywności, za jaką będziemy przyjmować procent, o jaki zbliżyliśmy się do wartości dolnego ograniczenia w stosunku do początkowego rozwiązania (zobacz tabele od 7.1a do 7.1c), w czasie działania algorytmu. Jak możemy zaobserwować, dla wielu różnych konfiguracji algorytm potrafi sobie radzić z zadanym problemem bardzo różnie, zazwyczaj nie zwracając gorszej odpowiedzi niż 10% od największego dolnego ograniczenia⁴. Na koniec przyglądnijmy się jeszcze czterem wykresom, gdzie każdy z nich przedstawia przebieg algorytmu TABU SEARCH dla problemu RECOVERALBE ROBUST INCREMENTAL SPANNING TREE, dla pełnego grafu o 30 wierzchołkach. Dla każdej z instancji problemu zastosowaliśmy różne, opisane pod odpowiednimi wykresami, parametry algorytmu TABU SEARCH, otrzymując dla niektórych z nich wartości rozwiązań, których odległość od dolnego ograniczenia na wartość optymalną nie przekracza 1%. Szczególnie warty zauważenia jest sytuacja, którą uchwyciliśmy na wykresie 7.5a, gdzie po ponad 500 iteracjach zostało znalezione takie rozwiązanie, którego wartość okazała się przybliżyć nas do optymalnego rozwiązania z 0,78% do 0,19%. Pokazuje

⁴Największe z dolnych ograniczeń nie jest wyznacznikiem, mówiącym jak daleko dane rozwiązanie jest oddalone od rozwiązania optymalnego w rzeczywistości — pokazuje jedynie największą możliwą różnicę między tymi dwoma rozwiązaniami, jako że to optymalne znajduje się gdzieś pomiędzy nimi.



to jak bardzo nieprzewidywalną może okazać się stosowana przez nas heurystyka i jak trudnym zadaniem jest eksperymentalne dobieranie jej wszystkich współczynników tak, aby otrzymywane rozwiązania były jak najlepsze.

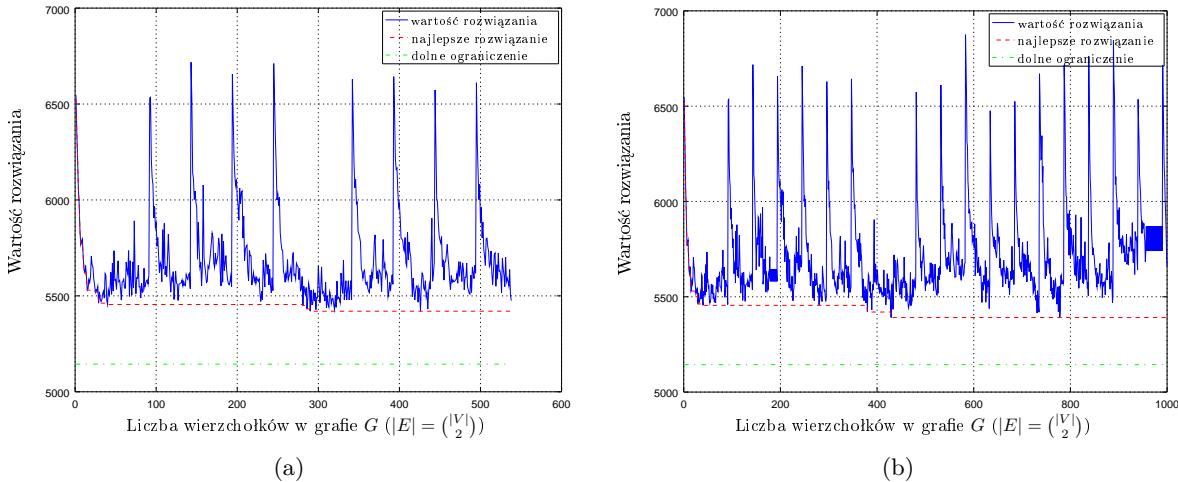
Tablica 7.1: Tabele przedstawiające odległości znajdowanych rozwiązań od dolnych ograniczeń, wyliczonych dla konkretnych instancji problemu. Kolejno w tabeli są przedstawione: numer iteracji algorytmu TABU SEARCH, dla której prezentowana jest aktualnie najlepsza, znalezione do tej pory, wartość rozwiązania zadanego problemu oraz wyrażenie, pokazujące procentowo odległość tej wartości od dolnego ograniczenia. (a) Miejsca, w których na wykresie 7.3a możemy obserwować poprawę najlepszego znalezionego rozwiązania. W ciągu 151 iteracji algorytmu zwracane rozwiązanie zostało poprawione o nieco ponad 23% w stosunku do rozwiązania początkowego i znajduje się co najwyżej 9,74% od rozwiązania optymalnego. (b) Te same dane przedstawione zostały dla wykresu 7.3b (c) oraz 7.3c.

#	$v_{TS}(T, S)$	$\frac{v_{TS}(T, S) - v_{TS}^{LB*}}{v_{TS}^{LB*}}$	#	$v_{TS}(T, S)$	$\frac{v_{TS}(T, S) - v_{TS}^{LB*}}{v_{TS}^{LB*}}$	#	$v_{TS}(T, S)$	$\frac{v_{TS}(T, S) - v_{TS}^{LB*}}{v_{TS}^{LB*}}$
1	6837	32,91%	1	6086	22,11%	1	4936	2,64%
5	6389	24,20%	2	5731	14,99%	2	4936	2,64%
9	6082	18,23%	4	5452	9,39%	3	4892	1,73%
73	6030	17,22%	5	5331	6,96%	54	4892	1,73%
76	5907	14,83%	11	5274	5,82%	55	4868	1,23%
141	5869	14,09%	23	5243	5,20%	209	4857	1,00%
145	5776	12,29%	39	5234	5,02%	210	4857	1,00%
151	5645	9,74%	49	5182	3,97%	873	4825	0,33%

(a)

(b)

(c)



(a)

(b)

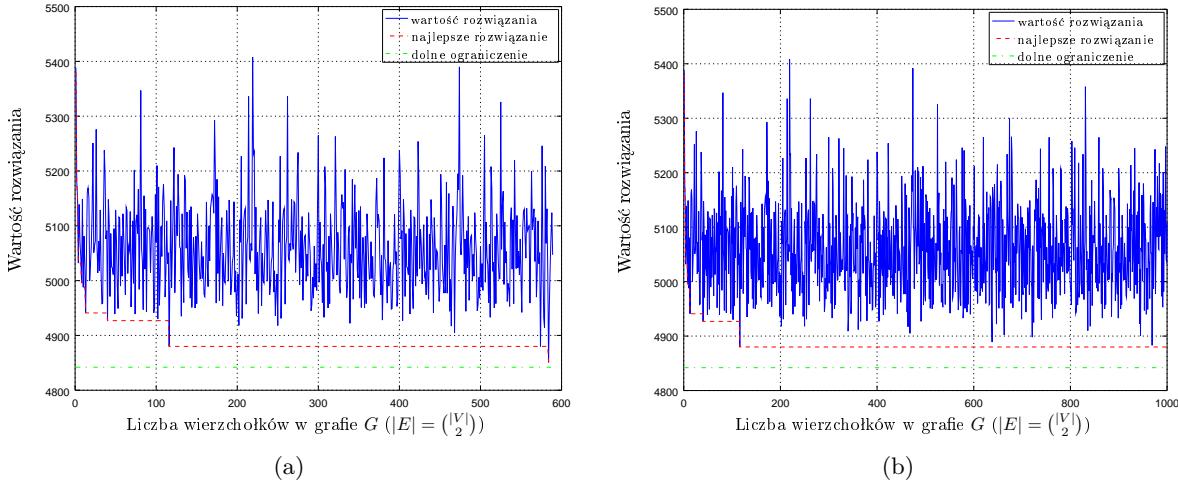
Rysunek 7.4: Wartości rozwiązań zwracane przez algorytm TABU SEARCH dla grafu o 30 wierzchołkach, 435 krawędziach (gęstość grafu $q = 1$), dla parametru $k = 3$ oraz maksymalną liczbą iteracji wynoszącą 50. (a) Przebieg algorytmu TABU SEARCH dla współczynników funkcji oceny ruchu $\alpha_1 = \frac{1}{10}$, $\alpha_2 = \alpha_3 = \alpha_4 = \alpha_5 = \frac{1}{5}$. Na przestrzeni 290 iteracji, w czasie których algorytm 4 razy rozpoczynał poszukiwanie rozwiązania w innym punkcie przestrzeni rozwiązań, początkowe rozwiązanie zostało poprawione o nieco ponad 20% (zobacz tabelę 7.2a). (b) Przebieg algorytmu dla tej samej instancji problemu ze zmienionymi współczynnikami funkcji oceny ruchu: $\alpha_1 = \frac{1}{5}$, $\alpha_2 = \alpha_3 = \alpha_4 = \alpha_5 = \frac{4}{5}$. W tym przypadku, tak jak to widać w tabelach 7.2a oraz 7.2b, zmiana sposobu podejmowania decyzji odnośnie następnych ruchów pozwoliła jeszcze bardziej zbliżyć się do optymalnego rozwiązania dla problemu.

Tablica 7.2: Tabele przedstawiające odległości znajdowanych rozwiązań od dolnych ograniczeń, wyliczonych dla instancji problemów, które zostały przedstawione na wykresach: (a) 7.4a oraz (b) 7.4b.

#	$v_{TS}(T, S)$	$\frac{v_{TS}(T, S) - v_{TS}^{LB*}}{v_{TS}^{LB*}}$	#	$v_{TS}(T, S)$	$\frac{v_{TS}(T, S) - v_{TS}^{LB*}}{v_{TS}^{LB*}}$
1	6546	27,26%	1	6546	27,26%
3	6227	21,05%	4	6059	17,79%
9	5768	12,13%	9	5768	12,13%
13	5622	9,29%	15	5530	7,50%
25	5519	7,29%	25	5519	7,29%
40	5455	6,05%	40	5455	6,05%
281	5450	5,95%	380	5420	5,37%
290	5420	5,37%	429	5391	4,80%

(a)

(b)



(a)

(b)

Rysunek 7.5: Wartości rozwiązań zwracane przez algorytm TABU SEARCH dla grafu o 30 wierzchołkach, 435 krawędziach (gęstość grafu $q = 1$), dla parametru $k = 15$ oraz maksymalną liczbą iteracji wynoszącą 50. (a) Przebieg algorytmu TABU SEARCH dla współczynników funkcji oceny ruchu identycznych jak dla wykresu 7.4a: $\alpha_1 = \frac{1}{10}$, $\alpha_2 = \alpha_3 = \alpha_4 = \alpha_5 = \frac{1}{5}$. Na przestrzeni prawie 600 iteracji początkowe rozwiązanie zdolano poprawić o nieco ponad 11%, osiągając tym samym niemal wartość dolnego ograniczenia. (b) Przebieg algorytmu dla tej samej instancji problemu ze zmienionymi współczynnikami funkcji oceny ruchu: $\alpha_1 = \frac{1}{5}$, $\alpha_2 = \alpha_3 = \alpha_4 = \alpha_5 = \frac{4}{5}$. Odwrotnie niż to miało miejsce na wykresach 7.4a oraz 7.4b, w tym przypadku zmiana parametrów funkcji oceny ruchu, pomimo zwiększenia liczby ogólnej iteracji, nie przyniosła oczekiwanych rezultatów w postaci polepszenia rozwiązania (zobacz tabele 7.3a oraz 7.3b).

Tablica 7.3: Tabele przedstawiające odległości znajdowanych rozwiązań od dolnych ograniczeń, wyliczonych dla instancji problemów, które zostały przedstawione na wykresach: (a) 7.5a oraz (b) 7.5b.

#	$v_{TS}(T, S)$	$\frac{v_{TS}(T, S) - v_{TS}^{LB*}}{v_{TS}^{LB*}}$	#	$v_{TS}(T, S)$	$\frac{v_{TS}(T, S) - v_{TS}^{LB*}}{v_{TS}^{LB*}}$
1	5388	11,28%	1	5388	11,28%
3	5151	6,38%	3	5151	6,38%
8	4996	3,18%	4	5032	3,92%
10	4988	3,02%	8	4996	3,18%
13	4941	2,04%	10	4988	3,02%
40	4927	1,76%	13	4941	2,04%
116	4880	0,78%	40	4927	1,76%
584	4851	0,19%	116	4880	0,78%

(a)

(b)



Zakończenie

Celem powyższej pracy była dogłębna analiza oraz implementacja heurystycznego algorytmu TABU SEARCH, rozwiązującego problem RECOVERABLE ROBUST INCREMENTAL MINIMUM SPANNING TREE dla dyskretnego zbioru scenariuszy, oraz zbadanie, na drodze wykonywanych eksperymentów, sposobu jego zachowania się w obliczu różnych instancji zadanego problemu, dla eksperimentalnie dobieranych parametrów, wymaganych do jego działania. Ze względu na klasę problemu, do której należy rozwiązywane zagadnienie (klasa problemów NP-trudnych, nieaproksymowalnych), zostaliśmy zmuszeni do oszacowywania jakości otrzymywanych wyników, w sensie stopnia ich oddalenia od optymalnych rozwiązań, przez przyjęcie za optymalne rozwiązanie największych dolnych ograniczeń, jakie udało nam się otrzymać dla zadawanych danych, przy wykorzystaniu modelu liniowego, prezentowanego w pracy [10], której autorzy skupili się na bardzo podobnym problemie, jednak dotyczącym kosztów ciągłych dla krawędzi (w odróżnieniu od nas, gdzie my badaliśmy możliwości odnajdywania rozwiązań dla przypadku kosztów, definiowanych przez dyskretne zbiory scenariuszy). Otrzymane wyniki okazują się jednak bardzo dobre, wskazując na słuszność decyzji o wykorzystaniu algorytmu TABU SEARCH, celem rozwiązywania instancji problemu RECOVERABLE ROBUST INCREMENTAL MINIMUM SPANNING TREE, do którego powstania doprowadziła nas szczegółowa analiza problemu, wykonana we wszystkich rozdziałach od 2 do 6. Otrzymywane dzięki zastosowanemu algorytmowi rozwiązania, oddalone od optimum o mniej niż 10 procent (gdzie otrzymywaliśmy i rozwiązania, których ten sam wskaźnik wynosił zaledwie dziesiąte części procenta), pozwalają stwierdzić, że wykonana przez nas praca dała oczekiwane rezultaty¹.

¹Na podstawie eksperymentów, możemy dodatkowo stwierdzić, że otrzymywane wyniki są tym bardziej zbliżone do wartości optymalnych, im większy parametr k (dla problemu INCREMENTAL MINIMUM SPANNING TREE) zadamy.



Bibliografia

- [1] Ahuja, Ravindra K., Magnanti, Thomas L., Orlin, and James B. *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.
- [2] Hassene Aissi, Cristina Bazgan, and Daniel Vanderpoorten. Min–max and min–max regret versions of combinatorial optimization problems: A survey. *European Journal of Operational Research*, 197(2):427–438, wrzesień 2009.
- [3] I. Averbakh and V. Lebedev. Interval data min–max regret network optimization problems. *Discrete Applied Mathematics*, 138(3):289–301, kwiecień 2004.
- [4] Xiaohui Bei, Ning Chen, and Shengyu Zhang. Solving linear programming with constraints unknown. In *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part I*, pages 129–142, 2015.
- [5] Chazelle and Bernard. A minimum spanning tree algorithm with inverse-ackermann type complexity. *J. ACM*, 47(6):1028–1047, listopad 2000.
- [6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Wprowadzenie do algorytmów*. Wydawnictwo Naukowe PWN, 7 edition, 2014.
- [7] Apostolos Dimitromanolakis. Analysis of the golomb ruler and the sidon set problems and determination of large near-optimal golomb rulers. Master’s thesis, University of Toronto, 2002.
- [8] Saul I. Gass. *George B. Dantzig*, pages 217–240. Springer US, Boston, MA, 2011.
- [9] Fred W. Glover and Manuel Laguna. *Tabu Search*. Springer, 1998.
- [10] Mikita Hradovich, Adam Kasperski, and Paweł Zieliński. The robust recoverable spanning tree problem with interval costs is polynomially solvable. *CoRR*, abs/1602.07422, 2016.
- [11] Filip Jany. Równoległe implementacje algorytmów lokalnego przeszukiwania. Master’s thesis, Politechnika Wrocławskiego, 1 2015.
- [12] Donald B. Johnson. A priority queue in which initialization and queue operations take $O(\log \log d)$ time. *Mathematical systems theory*, 15(1):295–309, 1981.
- [13] Adam Kasperski, Adam Kurpisz, and Paweł Zieliński. *Recoverable Robust Combinatorial Optimization Problems*, pages 147–153. Springer International Publishing, Cham, 2014.
- [14] Adam Kasperski, Mariusz Makuchowski, and Paweł Zieliński. A tabu search algorithm for the minmax regret minimum spanning tree problem with interval data. *Journal of Heuristics*, 18(4):593–625, 2012.
- [15] Adam Kasperski and Paweł Zieliński. An approximation algorithm for interval data minmax regret combinatorial optimization problems. *Information Processing Letters*, 97(5):177–180, marzec 2006.
- [16] Adam Kasperski and Paweł Zieliński. On the approximability of minmax (regret) network optimization problems. *Inf. Process. Lett.*, 109(5):262–266, 2009.
- [17] Adam Kasperski and Paweł Zieliński. Complexity of the robust weighted independent set problems on interval graphs. *Optimization Letters*, 9(3):427–436, 2015.

- [18] P. Kouvelis and G. Yu. *Robust discrete optimization and its applications*. Kluwer Academic Publishers., Boston, 1997.
- [19] Thomas L. Magnanti and Laurence A. Wolsey. Optimal trees. In *Network Models*, volume 7 of *Handbooks in Operations Research and Management Science*, pages 503–615. Elsevier, 1995.
- [20] Ebrahim Nasrabadi and James B. Orlin. Robust optimization with incremental recourse. *CoRR*, abs/1312.4075, 2013.
- [21] Nikulin and Yury. Simulated annealing algorithm for the robust spanning tree problem. *Journal of Heuristics*, 14(4):391–402, 2008.
- [22] Papadimitriou, Christos H., Steiglitz, and Kenneth. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1982.
- [23] Tarjan and R. E. Finding optimum branchings. *Networks*, 7(1):25–35, 1977.
- [24] Wilson and David Bruce. Generating random spanning trees more quickly than the cover time. In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, STOC ’96, pages 296–303, New York, NY, USA, 1996. ACM.
- [25] Hande Yaman, Oya Ekin Karasan, and Mustafa C. Pinar. The robust spanning tree problem with interval data. *Operations Research Letters*, 29(1):31–40, sierpień 2001.
- [26] Onur Şeref, Ravindra K. Ahuja, and James B. Orlin. Incremental network optimization: Theory and algorithms. *Operations Research*, 53(3):586–594, styczeń 2008.



Biblioteka: RIT

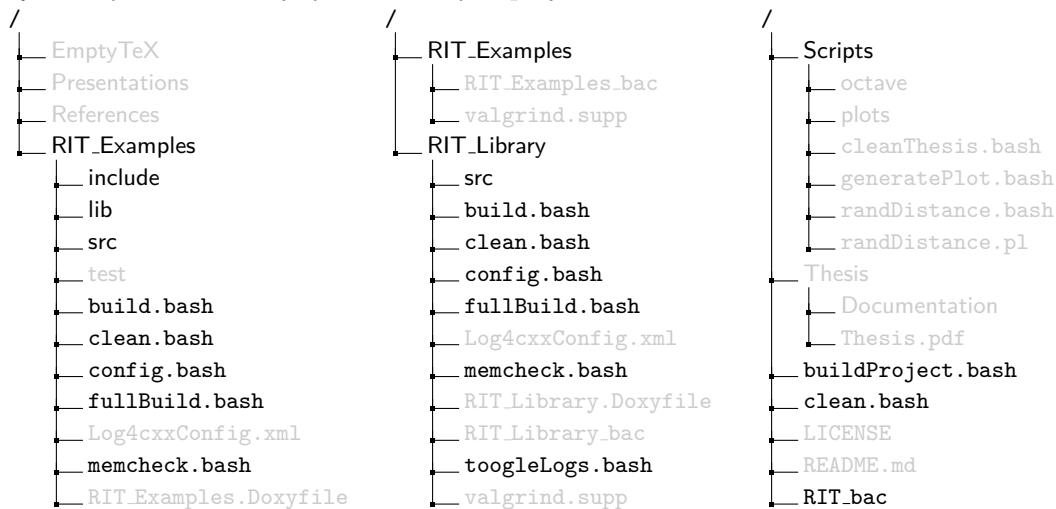
Poniżej opisana biblioteka RIT (ang. *Recoverable Incremental Tree*) stanowi zwieńczenie naszej pracy — zawiera implementacje wszystkich algorytmów, których pseudokody pojawiły się w powyższym tekście, zarówno tych klasycznych, imperatywnych, jak i dotyczących problemów programowania liniowego, bądź całkowitoliczbowego. Całość biblioteki została napisana w języku C++ w jego najnowszym, dostępnym na chwilę powstawania tego dokumentu, standardzie C++14, z wykorzystaniem elementów bibliotek stron trzecich, takich jak: IBM® CPLEX® OPTIMIZER, GRAPHVIZ, GTEST oraz LOG4CXX. Poniższy rozdział zawiera skrócony opis jej funkcjonalności¹, sposób instalacji oraz korzystania z niej w podstawowym zakresie.

A.1 Wymagania i instalacja

Całość omawianej biblioteki została zaimplementowana w środowisku systemu UBUNTU 15.10 dla architektury 64-bitowej, w związku z tym wszelkie przykłady oraz wykorzystywane skrypty zostały napisane z myślą o użytkownikach systemu z tej rodziny — proces komplikacji oraz instalacji biblioteki może się różnić w zależności od wykorzystywanego systemu operacyjnego. Dodatkowo jeden z prezentowanych skryptów korzysta z konsolowego API programu ECLIPSE MARS.1 (4.5.1), przystosowanego do pisania w językach C oraz C++ (ECLIPSE CDT), który posłużył do napisania całości biblioteki, co tym samym czyni go preferowanym narzędziem w przypadku chęci budowy biblioteki RIT na podstawie jej, udostępnionych na zdalnym repozytorium, źródeł. Poniżej prezentowany proces ma w tym pomóc.

A.1.1 Struktura projektu

Prezentowane drzewo katalogów ilustruje strukturę projektu oraz precyzuje położenie wszystkich, istotnych z punktu widzenia instalacji, skryptów. Nim przejdziemy do budowy biblioteki RIT ze źródeł, zapoznamy się ze wszystkimi ważniejszymi składowymi projektu: RIT, RIT_LIBRARY oraz RIT_EXAMPLES.



¹Pełen projekt wraz z jego dokumentacją został umieszczony na zdalnym repozytorium pod adresem: <https://github.com/PWrGitHub194238/RIT>.



Zaprezentowana powyżej struktura przedstawia wszystkie główne komponenty projektu, łącznie ze skryptami czy folderami, które nie będą na razie dla nas istotne (zostały one zaznaczone szarym kolorem w celu lepszej przejrzystości przedstawianej struktury katalogów). Etap budowania oraz uruchamiania aplikacji jest mocno zautomatyzowany, dlatego też, podczas wykonywania wszelkich czynności, będziemy głównie korzystać z plików, napisanych dla jednej z najbardziej popularnych powłok systemowych — skryptów o rozszerzeniu `*.bash`. Omówimy teraz przeznaczenie każdego z nich w folderach: `RIT_Library` oraz `RIT_Examples`, jako że w znacznym stopniu pokrywają się one ze sobą (są przeznaczone dla dwóch różnych podprojektów, z czego sposób budowy każdego z nich jest bardzo podobny).

```
/  
└── RIT_Library / RIT_Examples  
    └── config.bash
```

Główny skrypt dla projektów `RIT_LIBRARY` oraz `RIT_EXAMPLES`. Po wywołaniu wiersza poleceń (`CRTL + ALT + T`) i wykonaniu polecenia `./config.bash` (lub `bash config.bash` w przypadku, gdy plik skryptu nie posiada atrybutu, określającego go jako plik wykonywalny), ukażą nam się następujące opcje do wyboru:

```
1 # :~/git/RIT/RIT_Library$ ./config.bash  
2 You have to specify at least one action from these:  
3     autocompletion  
4     help  
5     install  
6     initBuild
```

Wybierając pierwszą z nich (wpisując w terminalu `./config.bash autocompletion` i zatwierdzając swój wybór), zostaniemy poproszeni o zezwolenie na podwyższenie uprawnień uruchomionego procesu — w przypadku wyrażenia zgody, do katalogu systemowego `~/etc/bash_completion.d/` zostanie przeniesiony plik `RIT_Library_bac` (bądź `RIT_Examples_bac`), który umożliwia systemowi operacyjnemu podpowiadanie nam składni wszystkich przyszłych komend (zasugeruje nam także możliwe ich parametry). Aby zmiany wprowadzone przez skrypt były widoczne, wymagane jest ponowne uruchomienie wiersza poleceń (`CRTL + ALT + T`). Od tej chwili możemy, poprzez podwójne naciśnięcie klawisza `TAB`, wypisać wszystkie, możliwe do zastosowania w danej chwili (na podstawie do tej pory wpisanego ciągu znaków w wierszu poleceń), parametry. Dodatkowo, jeżeli argument określonego parametru powinien spełniać określone właściwości (np. być plikiem wykonywalnym), po naciśnięciu klawisza `TAB` automatycznie zostanie podpowiadana jego wartość (jeśli jest tylko jedna możliwa):

```
1 # :~/git/RIT/RIT_Library$ ./config.bash autocompletion  
2 In order to successfully execute this command, extra privileges will be needed:  
3     [sudo] [...]  
4 You have granted this script root privileges.  
5 Please, restart shell in order to changes take effect.
```

Oczywiście uruchomienie skryptu z, następnym w kolejności, parametrem `help`, spowoduje wypisanie opisu wszystkich, możliwych do zastosowania w nim, parametrów wraz z pełnym wyrażeniem, opisującym sposób jego wywołania. Następna z możliwych opcji dla omawianego skryptu (`install`) pozwala użytkownikowi na przeprowadzenie instalacji komponentów, wymaganych do poprawnego działania biblioteki. Poniżej zaprezentowano rozszerzony opis parametrów, które użytkownik może podać, w celu sprecyzowania zachowania uruchamianego polecenia, a które zostaną podpowiadane:

```
1 # :~/git/RIT/RIT_Library$ ./config.bash install  
2 -d--defaults - wszystkie parametry, o które pyta skrypt w trakcie wykonywania,  
3             są uzupełniane wartościami domylnymi, a sam proces  
4             nie jest przerywany zapytaniem o ich podanie,  
5 -f--force-yes - skrypt podczas instalacji odwołuje się do swoich,  
6                 podwyższonych na czas jego działania, uprawnień,  
7                 gdzie niepodanie tej flagi skutkowałoby pojawiением się prośb  
8                 o podjęcie decyzji. W przypadku zastosowania tej flagi,  
9                 domylnymi odpowiedziami są odpowiedzi twierdzące/przyzwalające.  
10 --rapidjson-path <arg> - ścieżka katalogu, do którego zostaną pobrane pliki  
11             dla wymaganego w aplikacji komponentu rapidJSON.  
12 --rapidjson-include-path <arg> - ścieżka katalogu, do którego zostaną przeniesione  
13             pliki nagłówkowe wymaganego komponentu rapidJSON,
```

gdzie argumentami ostatnich dwóch parametrów mogą być zarówno ścieżki względne jak i bezwzględne. W czasie całego procesu zostaniemy kilkakrotnie zapytani o przyzwolenie na pobranie oprogramowania, w oparciu o które skrypt przeprowadzi automatyczną instalację następujących elementów: `LIBLOG4CXX10V5`, `LIBLOG4CXX10-DEV`, `GRAPHVIZ`, `GRAPHVIZ-DEV`, `RAPIDJSON`. W przypadku braku większej liczby komponentów, skrypt poprosi o przyzwolenie na instalację następujących elementów: `GIT` oraz `LIBPTHREAD-STUBS0-DEV`. Jeżeli dane oprogramowanie jest już przez nas posiadane, zostanie pominięte jego pobieranie oraz instalacja:



```
1 # :~/git/RIT/RIT_Library$ ./config.bash install
2 In order to successfully install all dependencies,
3 apt-get will be called, extra privileges will be needed:
4 [sudo] [...]
5 You have granted this script root privileges.
6 Following dependencies will be installed:
7     liblog4cxx10v5
8     liblog4cxx10-dev
9     graphviz
10    graphviz-dev
11    RapidJSON.
12 Confirm installation of this packages? (y/n): [y]
13 Installing 'Log4cxx' library...
14 [...]
15 Confirm installation of package "liblog4cxx10v5"? [y]
16 [...]
17 Confirm installation of package "liblog4cxx10-dev"? [y]
18 [...]
19 Installing 'GraphViz' library...
20 [...]
21 Installing 'RapidJSON' library...
22 Enter where to checkout RapidJSON project ("~/git/RIT/RIT_Library/RapidJSON/" if left blank): []
23 Enter where to move RapidJSON header files ("~/usr/local/include/rapidjson/" if left blank): []
24 "git" is required to checkout GTest project's sources.
25 Confirm installation of package "git"? [y]
26 [...]
27 "pthread" library will be needed to build GTest form sources.
28 Confirm installation of package "libpthread-stubs0-dev"? [y]
29 [...]
30 Checking out the RapidJSON project...
31 [...]
32 RapidJSON project has been saved into "~/git/RIT/RIT_Library/RapidJSON/".
33 [...]
34 RapidJSON header files have been moved to "~/usr/local/include/rapidjson/".
35 Delete RapidJSON project sources? (y/n): [y]
36 [...]
37 Package 'liblog4cxx10v5' is already installed. Skipping...
```

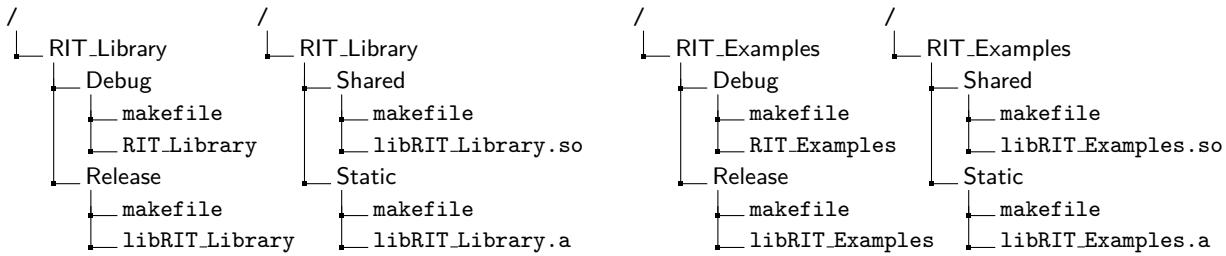
Po zakończeniu się powyższego procesu, projekt, dla którego uruchamialiśmy powyższy skrypt (RIT_LIBRARY — przebieg instalacji dla RIT_EXAMPLES jest bliźniaczo podobny i nie będziemy go osobno prezentować²), jest już gotowy do bycia zbudowanym ze źródeł, zawartych w katalogach `./src/src/` oraz `./src/include/`. Po pobraniu i instalacji wszelkich wymaganych przez bibliotekę zależności następnym krokiem jest wywołanie skryptu `config.bash` z parametrem `initBuild`, w celu budowy aplikacji — w tym przypadku skrypt zostanie poproszony o wygenerowanie w odpowiednich katalogach plików, zawierających zbiory reguł, dotyczących procesu budowania aplikacji (biblioteki bądź przykładowego programu, który ją wykorzystuje):

```
1 # :~/git/RIT/RIT_Library$ ./config.bash initBuild --path ~/workspace
2 WARNING: In order to Eclipse successfully generate project's makefiles,
3 You have to run this script as Eclipse's owner.
4 Java HotSpot(TM) 64-Bit Server VM warning: ignoring option MaxPermSize=256m; support was removed in 8.0
5 [...]
6 20:51:35 **** Clean-only build of configuration Debug for project RIT_Library ****
7 [...],
```

gdzie argumentem parametru `--path` jest ścieżka do katalogu roboczego programu ECLIPSE³. W przypadku poprawnego zakończenia się uruchomionego skryptu, następujące katalogi zostaną wygenerowane w jednej z możliwych lokalizacji (wraz ze skompilowanymi do pliku wykonywalnego/biblioteki współdzielonej/statycznej projektami):

²Dla projektu RIT_EXAMPLES dodatkową zależnością jest biblioteka GTEST, służąca do przeprowadzania automatycznych testów biblioteki RIT_LIBRARY, zawartych w tym projekcie. Zależności projektu możemy pobrać chociażby wykonując polecenie `./config.bash install -d -f` (z poziomu katalogu RIT_EXAMPLES).

³Celem skryptu jest uruchomienie wspomnianej aplikacji w trybie cichym (bez pokazywania jej graficznego interfejsu) i rozpoczęcie procesu komplikacji, wskazanego w skrypcie, projektu dla wszystkich profili dostępnych w jego ustawieniach (domyślnie: Debug, Release, Shared, Static). Operacja ta ma głównie na celu wymuszenie automatycznego wygenerowania plików `makefile`, zawierających reguły budowania projektu, z których następnie będą korzystać pozostałe skrypty — już bez odwoływania się do zewnętrznego programu. Aby zatem z powodzeniem wywołać opisywany skrypt, zakładamy, że przedstawiane w tym rozdziale projekty (RIT_LIBRARY, RIT_EXAMPLES) są bezpośrednio klonowane z repozytorium za pośrednictwem wspomnianej aplikacji (w takim wypadku program ECLIPSE sam zainicjalizuje w swoim katalogu roboczym oba projekty).



Tak wygenerowane pliki (w szczególności `./RIT_Library/Static/libRIT_Library.a`) następnie będziemy chcieli przenieść, przy pomocy innego skryptu, do odpowiednich katalogów testowego projektu `./RIT_EXAMPLES` — główną jego częścią są testy jednostkowe, mające na celu pokryć jak największy obszar funkcjonalności biblioteki. Program w znacznej mierze jest oparty o funkcjonalność, którą udostępniają mu pliki nagłówkowe przed chwilą zbudowanej biblioteki — procesem przeniesienia wymaganych plików z jednego projektu do drugiego zajmie się następujący skrypt:

```
/  
└── buildProject.bash,
```

który, uruchomiony z parametrem `moveFiles`, dokonuje przesunięcia statycznej biblioteki `libRIT_Library.a` oraz odpowiadających jej plików nagłówkowych z katalogu `./RIT_Library/src/include/` odpowiednio do `./RIT_Examples/lib/` i `./RIT_Examples/include/RIT/`:

```
1 # :~/git/RIT$ ./buildProject.bash moveFiles
2 Moving static library 'libRIT_Library.a'
3 from '~/git/RIT/RIT_Library/libRIT_Library.a' to 'RIT_Examples/lib'.
4 Coping includes from 'RIT_Library/src/include' to 'RIT_Examples/include/RIT'.
5 Done.
```

Po tej operacji możemy przejść do projektu `./RIT_Examples`, a następnie skompilować go za pomocą polecenia `./build.bash`, które umożliwia nam wybór pomiędzy kilkoma rodzajami komplikacji:

```
1 # :~/git/RIT/RIT_Examples$ ./build.bash
2 all      - buduj projekt dla wszystkich możliwych konfiguracji (wymienionych poniżej),
3 debug    - buduj projekt wraz z dołączonymi informacjami,
4           pozwalającymi na szczegółową analizę plików źródłowych projektu,
5 release  - buduj projekt w finalnej jego formie,
6 shared   - buduj projekt jako bibliotekę współdzieloną,
7 static   - buduj projekt jako bibliotekę statyczną,
8 doc      - wygeneruj tylko dostępną dla projektu dokumentację,
9 -f --force,
```

gdzie ostatnia flaga wymusza ponowną komplikację całego projektu⁴.

A.1.2 Pomocnicze skrypty

Przedstawimy teraz alternatywną ścieżkę budowy obydwu projektów wraz z pozostałymi rozwiązaniami, usprawniającymi poruszanie się pomiędzy nimi. Na początku zaznaczmy, że kroki, które opisaliśmy dla wywoływanych procedur: `./config.bash install` oraz `./config.bash initBuild`, są nadal konieczne, gdyż generują pliki, które gwarantują poprawne wykonywanie się następnych procedur. Przyjrzyjmy się jeszcze raz poleceniu `./buildProject.bash`, po którego wywołaniu otrzymamy pełną listę poleceń dla niego dostępnych, których poszerzony opis umieszczony został poniżej:

```
1 # :~/git/RIT$ ./buildProject.bash
2 autocompletion - podobnie jak to miało miejsce w opisanych wcześniej przykładach,
3                   wywołanie skryptu z tym parametrem spowoduje dodanie informacji
4                   o dostępnych dla tego skryptu parametrach do systemu,
5 buildAll        - buduje oba projekty, troszcząc się jednocześnie
6                   o wszystkie procesy, które zachodzą po drodze
7 buildStaticLib - buduje tylko statyczną wersję biblioteki RITLIBRARY,
8                   potrzebną do poprawnego uruchamiania drugiego z projektów,
9 buildExample   - komplkuje tylko aplikację testową,
10 clean          - porząduje projekty, usuwając z nich
11                   wygenerowane rozwiązania jak i pliki dzienników,
12                   przechowujących zapisy, wygenerowane podczas działania aplikacji,
13 -l --without-log - buduje projekt z usuniętymi z niego zależnościami
```

⁴W przypadku gdy chcielibyśmy kilkukrotnie dokonać komplikacji jego źródeł, bez nanoszenia w międzyczasie żadnych zmian, wszystkie procesy, poza pierwszym, zostaną zaniechane, chyba, że wywołamy je razem z flagą `--force`.



```
14      względem zewnętrznej biblioteki LOG4CXX  
15 -f --force      - wymusza przebudowanie projektów w przypadku,  
16          gdy od czasu jego ostatniej komplikacji  
17          nie zmieniły się ich źródła.
```

Pomineliśmy tutaj jeden z parametrów, `moveFiles`, gdyż ten opisaliśmy już wcześniej. Jak widzimy, powyższy skrypt, poza możliwością przeniesienia plików, wymaganych przez projekt RIT_EXAMPLES, oferuje także możliwość bezproblemowego zbudowania każdego z projektów osobno, bądź skompilowania całości poprzez wykonanie jednego polecenia (`./buildProject.bash buildAll`). Dodatkowo na samym początku tego rozdziału wspomnieliśmy o tym, że oba projekty wykorzystują elementy bibliotek zewnętrznych — jedną z nich jest biblioteka służąca do prowadzenia dziennika zdarzeń aplikacji, LOG4CXX. Dzięki wykorzystaniu flagi `--without-log` jesteśmy w stanie skompilować dany projekt bez tej jednej zależności (wspomnianą bibliotekę oraz sposób jej wykorzystania omówimy później).

Kolejność opisywania pozostałą części skryptów jest przypadkowa i nie ma dla nas znaczenia, gdyż mają one tylko charakter pomocniczy i ich uruchamianie nie jest w żaden sposób wymagane w celu przeprowadzenia, powyżej opisanego, procesu komplikacji.

```
/  
└ clean.bash
```

Tak samo jak `./buildProject.bash`, skrypt ten bazuje na funkcjach, będących częścią plików `./RIT_Library/clean.bash` oraz `./RIT_Examples/clean.bash` — jest dla nich alternatywą. W odróżnieniu od wymienionych, uruchomienie tego skryptu spowoduje próbę przeglądu wszystkich katalogów (nie tylko `./RIT_Library` i `./RIT_Examples`) w poszukiwaniu plików o rozszerzeniach, wskazujących na to, że powstały w wyniku innych operacji i można je usunąć (np. pliki powstałe przy komplikacji projektów, dzienniki aplikacji, inne pliki tymczasowe). Jeżeli w danym katalogu znajduje się już skrypt o nazwie `clean.bash`, zostanie on uruchomiony automatycznie z parametrem `all`, w celu usunięcia wszystkich zbędnych plików z danego katalogu (możliwe jest także ręczne uruchomienie poszczególnych skryptów z jednym z parametrów: `all`, `debug`, `shared`, `static`, `doc`).

```
/  
└ RIT_Library / RIT_Examples  
    └ memcheck.bash
```

Skrypt został napisany w celu walidacji komplikowanego rozwiązania — testuje podany program pod kątem wycieków pamięci i, w przypadku istnienia takowych, zwraca ich liczbę oraz całkowity obszar pamięci, jaki jest tracony w wyniku uruchomienia takiego programu. Skrypt przyjmuje szereg parametrów, które opisujemy poniżej.

```
1 # :~/git/RIT/RIT_Library$ ./memcheck.bash  
2 -f --force-yes - działanie skryptu opiera się o funkcjonalność zewnętrznego programu  
3           do dynamicznej analizy kodu VALGRIND. W przypadku braku takowego oprogramowania  
4           zostaniemy poproszeni o zgodę na jego instalację. Jeżeli w parametrach programu  
5           podamy omawianą flagę, takie powiadomienie nie wystąpi,  
6           gdyż skrypt jest zmuszony do cichego (bez wchodzenia  
7           w interakcję z użytkownikiem) wykonania wszystkich poleceń.  
8 -r --runnable <arg> - ścieżka do pliku wykonywalnego, będącego wynikiem komplikacji projektu  
9           napisanego w jednym z języków, które VALGRIND obsługuje: C, C++.  
10          W przypadku podania tej opcji i wymuszenia na systemie operacyjnym  
11          podpowiedź argumentów, zostaną wypisane tylko ścieżki do tych plików,  
12          które posiadają status plików wykonywalnych (w przypadku istnienia  
13          pojedynczego pliku, jego nazwa zostanie automatycznie w całości uzupełniona).  
14 -o --output <arg> - ścieżka pliku, do którego zostaną przekierowane wszystkie komunikaty  
15           wygenerowane przez program VALGRIND w trakcie jego działania. Informacje  
16           wypisywane przez sam skrypt nie są do tego pliku przekierowywane.  
17 -s --suppressions <arg> - ścieżka do pliku zawierającego listę opisów błędów,  
18           które podczas dynamicznej analizy kodu mają być ignorowane,
```

gdyż część błędów, będąca wynikiem takiej analizy, może nie powstawać bezpośrednio z winy testowanego oprogramowania (RIT_LIBRARY lub RIT_EXAMPLES), a być efektem wykorzystywania bibliotek zewnętrznych⁵, zaś zastosowanie tej flagi pozwala na ich pominięcie. Przykładowe wywołanie skryptu przedstawiono poniżej:

```
1 # :~/git/RIT/RIT_Library$ ./memcheck.bash -r ./BinaryIMST_Library -o out.log -s ./valgrind.supp  
2 'Valgrind' is required in order to memcheck this application.  
3 Package 'valgrind' is already installed. Skipping...  
4 Running valgrind with additional configuration from file  
5 './git/BinaryIMST/BinaryIMST_Library/valgrind.sup'  
6 (log file will be generated as 'out.log')...  
7 [...]  
8 No errors.
```

⁵Jednym z takich błędów jest wyciek pamięci powstały w wyniku wykorzystywania niektórych struktur standardowej biblioteki języka (C++ STL), których sposób zarządzania pamięcią powoduje zgłaszanie takich informacji.



Następnym skryptem wartym omówienia jest `toggleLogs.bash` — jego uruchomienie skutkuje automatycznym usunięciem z wybranego projektu zależności w postaci biblioteki LOG4CXX. Należy zwrócić uwagę na fakt możliwego spowolnienia działania aplikacji ze względu na konieczność wypisywania bardzo dużej liczby informacji do wydzielonego pliku, bądź bezpośrednio do wiersza poleceń — z uwagi na to, że w niektórych przypadkach może nam zależeć na zmaksymalizowaniu efektywności stosowanych algorytmów (bardziej niż na dokładnym informowaniu nas przez aplikację o podjętych przez nią działaniach), mamy możliwość tymczasowego wydzielenia wszystkich elementów wspomnianej biblioteki (jeśli nie chcemy wyłączać lecz usunąć całkowicie daną funkcjonalność):

```

1 # :~/git/RIT$ ./RIT.Library/toggleLogs.bash
2 Create directory '~/git/RIT/RIT_Library/extracted_log/src/include'.
3 Create directory '~/git/RIT/RIT_Library/extracted_log/src/src'.
4 -----
5 Removing logs from project...
6 Move logging-related sources out from project...
7 Move '~/git/RIT/RIT_Library/src/src/log' > '~/git/RIT/RIT_Library/extracted_log/src/src'.
8
9 Comment logging-related code in source files...
10
11 Parsing file: ~/git/RIT/RIT_Library/src/src/heap/VertexHeapItem.cpp
12 [...]
13 Parsing file: ~/git/RIT/RIT_Library/src/src/bundle/EN_US_Bundle.cpp
14
15 All logging-related code in source files has been commended.
16
17 Move logging-related headers out from project...
18 Move '~/git/RIT/RIT_Library/src/include/log' > '~/git/RIT/RIT_Library/extracted_log/src/include'.
19
20 Comment logging-related code in header files...
21
22 Parsing file: ~/git/RIT/RIT_Library/src/include/heap/FibonacciHeap.hpp
23 [...]
24 Parsing file: ~/git/RIT/RIT_Library/src/include/typedefs/struct.hpp
25
26 All logging-related code in headers files has been commended.

```

Uruchomienie skryptu ponownie spowoduje umieszczenie wszystkich usuniętych informacji na powrót w projekcie, którego wywoływany skrypt dotyczy. Bezpośrednio z już omówionych skryptów (`toggleLogs.bash`, `build.bash`, `mem-check.bash`) korzysta następny z nich: `fullBuild.bash`, który obsługuje cały proces komplikacji rozwiązania, który jest podzielony na następujące etapy:

- usunięcia z projektu zależności od biblioteki LOG4XX w celu zmaksymalizowania efektywności budowanego rozwiązania,
- komplikacji całego projektu,
- przeanalizowania wygenerowanej aplikacji pod kątem wycieków pamięci i zapisania wyników do pliku (domyślnie `valgrind.log`),
- przywrócenia do projektu usuniętej wcześniej funkcjonalności.

A.1.3 Biblioteka: log4cxx

Proponowane przez nas rozwiązanie w postaci biblioteki RIT zapewnia możliwość zapisywania operacji, wykonywanych przez aplikacje ją wykorzystującą. Dla każdego z projektów, w odpowiadających im plikach Log4cxxConfig.xml, znajduje się szczegółowa konfiguracja, determinująca sposób działania opisywanej funkcjonalności, gdzie przykładową konfigurację przedstawiamy poniżej:

```

1 <category name="utils.TabuSearchUtils" additivity="false">
2   <priority value="warn" />
3   <appender-ref ref="appxFileAppender" />
4   <appender-ref ref="appxConsoleAppender" />
5   <!-- <appender-ref ref="appxXMLAppender" /> -->
6   <!-- <appender-ref ref="appxChainsawXMLAppender" /> -->
7 </category>

```

W powyżej prezentowanym przykładzie przedstawiono konfigurację dla pliku `TabuSearchUtils.cpp`, znajdującego się w katalogu `./RIT_Library/src/src/utils/`. Ma ona bezpośredni wpływ na sposób zachowania się wszystkich instrukcji w danym pliku, które powodują wygenerowanie zapisów w dzienniku zdarzeń aplikacji np. `TRACE(logger, klucz, ...)`.

Dla powyżej zaprezentowanego przykładu, wymieniony fragment kodu nie zostałby wykonany — jego poziom, jaki sobą reprezentuje, jest za niski, w odniesieniu do jego konfiguracji (WARN). W tym przypadku wszystkie akcje z pliku, którego tyczy się przykładowa konfiguracja, przypisane do niższego poziomu niż zadeklarowany w konfiguracji, nie zostaną wykonane (OFF < FATAL < ERROR < WARN < INFO < DEBUG < TRACE < ALL). Pozostała część prezentowanej konfiguracji determinuje sposób w jaki dane wyrażenia (wszystkie zapisy dziennika zdarzeń występujące w pliku `./RIT_Library/src/src/utils/TabuSearchUtils`) będą przetwarzane. Odpowiednio:

- `appxFileAppender` wskazuje, że będą one zapisywane do zewnętrznego pliku (domyślnie `appxLogFile.log`),
- `appxConsoleAppender` dodatkowo przekierowuje wszystkie informacje do wiersza poleceń, tego samego, w którym została uruchomiona aplikacja,
- oznaczone jako nieaktywne, wiersze z wartościami `appxXMLAppender` oraz `appxChainsawXMLAppender` zabraniają tym samym przesyłania tych samych danych w jeszcze inne miejsca, gdzie pierwsza z opcji generowałaby na ich podstawie odpowiednio sformatowany plik XML, druga zaś pozwalałaby na przechwytywanie generowanych informacji zewnętrznemu programowi, APACHE CHAINSAW.

Przykłady wygenerowanych dzienników zdarzeń (ich fragmenty) zostaną przytoczone niżej (patrz podrozdział A.3), przy okazji omawiania właściwej funkcjonalności biblioteki RIT.

Dziennik zdarzeń w Apache Chainsaw

Korzystanie z zewnętrznej aplikacji, w celu polepszenia kultury pracy z zapisami zdarzeń, generowanych przez aplikację, jest bardzo intuicyjne. Aby rozpocząć przechwytywanie wszystkich komunikatów z aplikacji (w celu np. ich swobodnego filtrowania), należy w głównym pliku konfiguracyjnym zadbać o to, by odpowiednie części konfiguracji, te tyczące się interesujących nas elementów, zezwalały na przekierowywanie komunikatów do wspomnianej aplikacji (`<appender-ref ref="appxChainsawXMLAppender" />`) jak i miały odpowiednio ustawiony poziom, który będzie wyższy niż ten, który podany w APACHE CHAINSAW. Samo korzystanie z owego programu jest bardzo proste i sprowadza się do utworzenia obiektu, który będzie odbierał przekazywane do niego zapisy dziennika zdarzeń (w tym wypadku powinien być to obiekt `XMLSocketReceiver`), ustawienie odpowiedniego portu nasłuchiwanego (`<param name="Port" value="4448" />`) oraz poziomu szczegółów, jaki nas interesuje. Po tej konfiguracji program powinien być gotowy do pracy.

A.1.4 Pozostałe biblioteki i skrypty grafowe

Poza biblioteką LOG4CXX, w projekcie znalazły zastosowanie jeszcze dwie: GTEST oraz GRAPHVIZ, gdzie pierwsza z nich pozwala na pisanie automatycznych testów jednostkowych dla zaimplementowanej biblioteki (testy te są częścią aplikacji RIT_EXAMPLES i to ich wynik jest domyślnie zwracany w przypadku poprawnego zakończenia się wykonywanego programu), druga pełni jedynie funkcje pomocnicze — pozwala konwertować wejściowe formaty grafowe dla programu na inne (np. do formatu DOT, który później może posłużyć do utworzenia schematu grafu zewnętrznymi programami, które potrafią odczytać dany format). Do omówienia pozostały nam jeszcze dwa skrypty, które pomimo tego, że nie mają bezpośredniego zastosowania w procesie budowy biblioteki, są wielce przydatne w trakcie jej użytkowania np. do generowania danych.

```
/  
└── Scripts  
    ├── randDistance.bash  
    └── randDistance.pl
```

Skrypt służy do szybkiego, automatycznego generowania danych, mających głównie charakter scenariuszy adwersarza dla zadanego już grafu G . Podając na jego wejściu odpowiednie parametry, w ciągu bardzo krótkiego czasu otrzymamy w zamian plik z definicjami kosztów tego samego grafu, które w określonym przez nas stopniu różnią się od tych, którymi charakteryzował się oryginalny graf. Poniżej opisujemy wszystkie parametry omawianego skryptu wraz z przykładowym jego wywołaniem⁶:

```
1 # :~/git/RIT/Scripts$ ./randDistance.bash  
2 -l --lowerBound <arg> - dolna wartość przedziału zaburzeń kosztów w grafie  
3             - do każdej krawędzi (do jej kosztu) zostanie dodana wartość  
4             z przedziału, którego dolny kres reprezentuje dany parametr,  
5 -u --upperBound <arg> - górna wartość przedziału, gdzie każda waga krawędzi  
6             zostanie zastąpiona nowym kosztem, określonym za pomocą tej  
7             oraz poprzedniej wartości. Do kosztów krawędzi zostanie dodana  
8             losowo wybrana (z rozkładem jednostajnym) wartość z przedziału [l; u],
```

⁶Do poprawnego działania skrypt wymaga obecności interpretera języka skryptowego PERL.



```

9 -i --input <arg>      - ścieżka do pliku z definicją grafu, zapisana w formacie
10                                zdefiniowanym przez 9th DIMACS Implementation Challenge,
11 -o --output <arg>      - ścieżka pliku, do którego będą zapisane nowe koszty dla grafu wejściowego,
12 -s --seed <arg>       - prawdopodobieństwo zmiany kosztu krawędzi.
13                                Im wyżej, tym więcej wag krawędzi ulegnie zmianie.

```

Aby otrzymać zupełnie losowy rozkład dla wejściowych kosztów grafu, wszystkie w nim koszty powinny mieć wartość równą C , zaś parametrami, które określają odpowiednio najmniejszą i największą możliwą wartość dodaną do kosztu krawędzi, powinny być odpowiednio $-C$ oraz C , gdzie $seed = 1$. Przy takiej konfiguracji prawdopodobieństwo zmiany kosztów krawędzi w grafie wynosi 1 (zmianie ulegną wszystkie koszty), zaś każdy z nich zostanie wybrany z przedziału $[C - C, C + C]$ (zdecydowaliśmy się ograniczyć koszty do nieujemnych wartości całkowitych).

```

/
└─ Scripts
    └─ generatePlot.bash

```

W celu jak największego ułatwienia generowania wykresów na podstawie danych, które zwracają algorytmy biblioteki RIT, możemy wykorzystać powyższy skrypt. Jego parametry, które przyjmuję, są o tyle nietypowe, że dzielą się na dwie części: te, dotyczące sposobu zachowania się skryptu w sensie globalnym, oraz te, które bezpośrednio odpowiadają za wygląd linii, kreślonych przez program na podstawie wprowadzonych danych, gdzie te drugie mogą być powtarzane cyklicznie (to znaczy, że ostatnie trzy parametry wywołania stanowią nierozerwalną całość, zaś powielenie ich występowania skutkować będzie naniesieniem na ten sam rysunek wykresów większej liczby funkcji — w tym wypadku kolejność parametrów obu grup względem siebie oraz drugiej grupy, opisującej wygląd generowanych wykresów, jest istotna).

```

1 # :~/git/RIT/Scripts$ ./generatePlot.bash
2 bash generatePlot.bash <bash flags> [<file name> <octave line style> <function label> [...]]
3
4 <bash flags> - podzbiór flag, które przyjmuje skrypt, a które definiują jego sposób zachowania:
5   -d --defaults           - jeżeli ta flaga jest obecna w linii wywołania polecenia,
6                                skrypt pominie wszelkie pytania odnośnie tych parametrów,
7                                dla których została przewidziana wartość domyślna
8                                (wszystkie poniżej opisane, poza następną),
9   -f --force-yes          - skrypt do działania wymaga zainstalowania środowiska GNU Octave,
10                             toteż w przypadku jego braku zostaniemy poproszeni o zgodę
11                             na jego instalację, chyba że zastosowana będzie opisywana flaga,
12   -p --octave-path <arg> - ścieżka do skryptu GNU Octave, odpowiedzialnego
13                                za rysowanie wykresów w tymże środowisku (GENERATEPLOT.M),
14   -i --input-data-path <arg> - ścieżka do katalogu, w którym umieszczone są pliki z danymi,
15   -o --output-data-path <arg> - ścieżka katalogu, do którego będą zapisywane wygenerowane pliki,
16   -t --plot-title <arg>   - tytuł wygenerowanego przez skrypt wykresu,
17   -x --x-axis-title <arg> - nazwa osi odciętych na wykresie,
18   -y --y-axis-title <arg> - nazwa osi rzędnych,
19   -n --file-output-name <arg> - nazwa samego pliku z wygenerowanym przez skrypt wykresem,
20   -e --file-extension <arg> - rozszerzenie przybierane przez generowany plik (np. EPS, PDF),
21   -h --help                - drukuje skróconą instrukcję sposobu korzystania ze skryptu.
22
23 <file name> - nazwa pliku z danymi dla skryptu w określonym formacie. Dane w pliku:
24     1             1
25     2             4
26     3             9
27     ...
28 są przykładem funkcji kwadratowej  $f(n) = n^2$ .
29
30 <octave line style> - styl rysowania przez interpreter GNU Octave w formacie FMT (np. ,,b-'')
31
32 <function label> - nazwa konkretnej funkcji na wykresie, odpowiadająca podanym wyżej danym.

```

Poniżej przedstawiono strukturę, z której korzysta skrypt GENERATEPLOT.BASH w przypadku pozostawienia wartości domyślnych jako parametrów (gdy flaga `-d` jest ustaliona, bądź nie zostały wprowadzone inne dane).

```

/
└─ Scripts
    └─ OCTAVE
        └─ PLOTS
            └─ INDATA ... Domyślny katalog, gdzie skrypt będzie szukał danych wejściowych.
            └─ OUTDATA ... Domyślny katalog, gdzie skrypt będzie zapisywał wygenerowane pliki.
                generatePlot.bash

```

```
└── randDistance.bash
    └── randDistance.perl
```

Przykładowe wywołanie takiego skryptu może wyglądać na przykład tak:

```
1 # :~/git/RIT/Scripts$ ./generatePlot.bash data1 b- label1 data2 g-- label2
2 -----
3
4 Add to dataArray: data1
5 Add to styleArray: b-
6 Add to legendArray: label1
7 Add to dataArray: data2
8 Add to styleArray: g--
9 Add to legendArray: label2
10 Enter path to generatePlot.m Octave function ("./octave" if left blank): []
11 Enter path to folder with Data Input Files ("./plots/inData" if left blank): []
12 Enter path to output folder for generated plot ("./plots/outData" if left blank): []
13 Enter title of generated plot ("{EMPTY TITLE}" if left blank): []
14 Enter title of label for axis X ("{X LABEL}" if left blank): []
15 Enter title of label for axis Y ("{Y LABEL}" if left blank): []
16 Enter name of output file ("out" if left blank): []
17 Enter extension of output file ("epsc" if left blank): []
18 Plot configuration summary:
19 Octave function path :          ~/git/RIT/Scripts/octave/
20 Input data path   :          ~/git/RIT/Scripts/plots/inData/
21 Output plot path :          ~/git/RIT/Scripts/plots/outData/
22 Plot's title     :          {EMPTY TITLE}
23 Plot X label    :          {X LABEL}
24 Plot Y label    :          {Y LABEL}
25 Output file      :          out.epsc
26
27 Do you want to create this plot? (y/n): [y]
28     [...]
29 Plot has been saved to: "~/git/RIT/Scripts/plots/outData/" as "out.epsc",
```

gdzie powyższe wywołanie skryptu spowoduje wygenerowanie pliku z dwoma wykresami funkcji, gdzie pierwszy z nich jest narysowany niebieską linią ciągłą (dla danych z pliku `~/git/RIT/Scripts/plots/inData/data1`), drugi zaś, opisany jako „label2”, przerywaną linią zieloną (patrz Rysunek A.1). Aby uzyskać więcej informacji na temat dostępnych sposobów formatowania linii, warto zapoznać się z dokumentacją GNU OCTAVE.

A.2 Możliwości biblioteki

Tak jak wspomnieliśmy na samym początku tego rozdziału, biblioteka RIT implementuje wiele z, dotąd przez nas poznanych, algorytmów. Aby zaprezentować możliwość ich wykorzystania, przedstawimy kilka przypadków działania aplikacji, gdzie każdym z nich będziemy starali odpowiedzieć się na jedno z podstawowych pytań, jakie mogą się nasunąć podczas zapoznawania się ze strukturą wspomnianej biblioteki. Należy mieć na uwadze, że prezentowane w tej części wycinki kodu źródłowego, w charakterze odpowiedzi na każde z zadanych przez nas pytań, nie są odpowiedziami jedynymi słusznymi — większość obiektów, jakie będziemy prezentować, biblioteka RIT pozwala stworzyć na więcej niż jeden sposób. My zaś będziemy starali się prezentować tylko te przypadki użycia, które naszym zdaniem pojawiać się mogą najczęściej przy korzystaniu z niej.

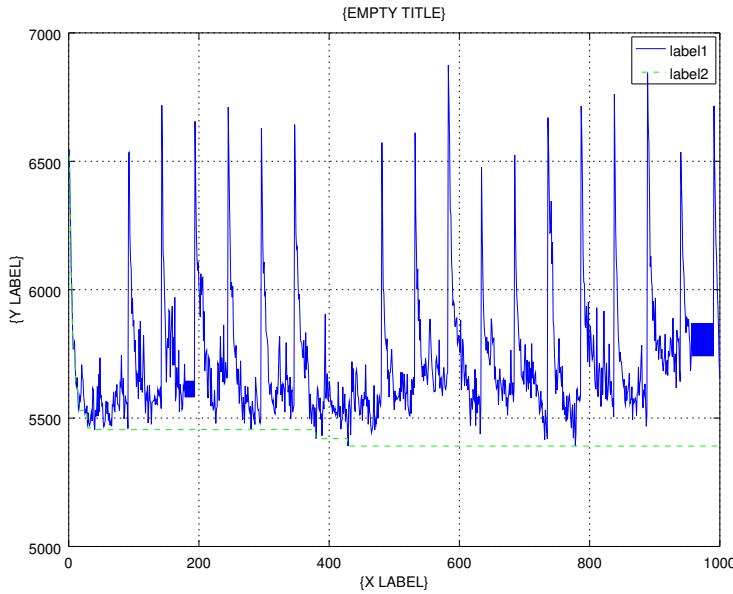
A.2.1 Budowanie grafu

Podstawową czynnością jaką musimy mieć możliwość wykonać, jest budowa grafu, na którym będziemy mogli operować.

W jaki sposób zbudować graf przy wykorzystaniu udostępnionych funkcji?

Biblioteka RIT zapewnia kilka sposobów na generowanie grafów, dla których będziemy później chcieli wykonywać obliczenia. Główne z nich to:

- odczytanie definicji całego grafu z pliku — główne z obsługiwanych formatów opiszemy poniżej, na tę chwilę



Rysunek A.1: Wygenerowany wykres dwóch funkcji dla przykładowego wywołania skryptu GENERATE-PLOT.BASH, obrazujący sposób zachowania się algorytmu TABU SEARCH dla przykładowej instancji grafu $G = (V, E)$, gdzie $|V| = 30$ (G jest grafem pełnym), dla okresu szukania rozwiązania równego 50 (po którym następuje „restart” algorytmu).

jesteśmy zainteresowani tylko sposobem wywołania danej funkcjonalności⁷:

```

1 #include <RIT/exp/IOExceptions.hpp>
2 #include <RIT/utils/enums/InputFormat.hpp>
3 #include <RIT/utils/enums/InputMode.hpp>
4 #include <RIT/utils/IODutils.hpp>
5 #include <RIT/utils/MemoryUtils.hpp>
6
7 int main() {
8     GraphIF* g { };
9     try {
10         g = InputUtils::readGraph("ściezka do pliku *.gr", InputFormat::GR, InputMode::HDD);
11         MemoryUtils::removeGraph(g, true, true);
12     } catch (IOExceptions::FileNotFoundException& e) {
13         return 0;
14     }
15     return 0;
16 }
```

gdzie wartość enumerowana **InputFormat** odpowiada za format, który zdecydowaliśmy się wczytać do programu, **InputMode** — za sposób jego załadowania, gdzie **HDD** odpowiada standardowej metodzie odczytu pliku, której będziemy używać. Należy oczywiście pamiętać o usunięciu przed chwilą stworzonego grafu najwcześniej jak to tylko możliwe, jeżeli chcemy ustrzec się przed późniejszymi problemami związanymi z zarządzaniem pamięcią przez program (każdy prezentowany przykład będzie zawierać również kod usuwający stworzone przez siebie obiekty).

- dynamiczna konstrukcja grafu — umożliwia stworzenie dowolnej struktury grafowej przy wykorzystaniu metod, udostępnianych przez bibliotekę. Poniżej widoczny przykład przedstawia sposób stworzenia grafu z trzema wierzchołkami, gdzie każdy z nich jest połączony z pozostałymi (graf pełny). Dla danej klasy **GraphIF**, podobnie jak dla zdecydowanej większości obiektów, które będziemy wykorzystywać w prezentowanych kodach, jest dostępna większa liczba konstruktorów — sposób generowania grafu dla każdego z nich różni się od poniższego.

⁷W prezentowanych przypadkach ujawniamy obsługę wyjątków, zgłoszonych w czasie wykonywania programu, przykładowo prezentując sposób ich przechwytywania tylko w pierwszym z nich.



W tym konkretnym przypadku graf jest budowany na podstawie dwóch zbiorów danych: jego wierzchołków V oraz, łączących je, krawędzi E . Poniżej prezentowany sposób konstrukcji grafu $G = (V, E)$ jest zatem naturalny:

```
1 VertexSetIF* vSet = new VertexSetImpl { 3 };
2 EdgeSetIF* eSet = new EdgeSetImpl { 3 };
3 GraphIF* g = new GraphImpl { vSet, eSet };
4
5 for (unsigned int idx = 0; idx < 3; idx += 1) {
6     vSet->push_back(new VertexImpl { idx });
7 }
8
9 for (unsigned int idx = 0; idx < 3; idx += 1) {
10    eSet->push_back(
11        new EdgeImpl { idx, VertexPair(vSet->getElementAt(idx),
12                                     vSet->getElementAt((idx + 1) % 3)), (EdgeCost) idx });
13 }
14 MemoryUtils::removeGraph(g, true, true);
```

gdzie kolejno stworzyliśmy trzy wierzchołki grafu, później zaś, z wykorzystaniem operacji wyznaczania reszty z dzielenia, jego krawędzie: e_{01} , e_{12} oraz e_{20} , gdzie koszt każdej z nich wynosi idx .

- losowe generowanie struktury — udostępnione przez klasę pomocniczą `GraphUtils` funkcje, umożliwiają między innymi wygenerowanie losowego grafu na podstawie takich danych jak:
 - liczba wierzchołków docelowego grafu,
 - liczba krawędzi / gęstość grafu (jedna z dwóch informacji),
 - najniższy możliwy koszt łuku,
 - największa waga, jaką może przyjąć krawędź.

```
1 GraphIF* g = GraphUtils::getRandomGraph(10,0.5,10,15);
2
3 MemoryUtils::removeGraph(g, true, true);
```

gdzie pierwszymi dwoma parametrami w tym przypadku są: liczba krawędzi oraz gęstość grafu. Powyższy fragment kodu został wykorzystany do wygenerowania wszystkich instancji grafów, dla których zostały przeprowadzone eksperymenty w rozdziale 7 (oczywiście dla różnych wartości argumentów).

Jakie formaty wejściowe obsługuje biblioteka?

Biblioteka RIT posługuje się istniejącym standardem zapisu informacji grafowych, jaki zastosowano podczas dziewiątej edycji programu DIMACS IMPLEMENTATION CHALLENGE, która skupiała się na problemie wyszukiwania najkrótszych ścieżek w grafach. Wykorzystanie tej struktury dla problemu minimalnego drzewa rozpinającego nie stanowi żadnej trudności, gdyż oba te problemy są oczywiście problemami grafowymi. Przykładowy plik, zawierający opis grafu, jego wierzchołków oraz krawędzi, wygląda tak, jak pokazano poniżej:

```
1 c 9th DIMACS Implementation Challenge: Shortest Paths
2 c http://www.dis.uniroma1.it/~challenge9
3 c Sample graph file
4 c
5 p mst 6 8
6 c graph contains 6 nodes and 8 arcs
7 c node ids are numbers in 0..5
8 c
9 a 0 1 17
10 c arc from node 0 to node 1 of weight 17
11 c
12 a 0 2 10
13 a 1 3 2
14 a 2 4 0
15 a 3 2 0
16 a 3 5 3
17 a 4 1 0
18 a 4 5 20
```

gdzie pierwsza litera rozpoczynająca nową linię, niesie za sobą informację co w danej linii można znaleźć. Opisując każdą z nich po kolei mamy:



- c — linie zawierające dowolny komentarz, nie dłuższy jednak niż `IUtils::impl::MAX_CHARS_IN_LINE` (stała biblioteki RIT_LIBRARY, definiowana ze względów optymalizacji procesu samego odczytywania pliku),
- p — linia, rozpoczynająca się takim symbolem, zawiera definicję problemu, który dany plik opisuje. W naszym przypadku jest to problem minimalnego drzewa rozpinającego (**mst**),
- a — trzy liczby, które znajdują się w każdej linii, rozpoczynającej się od tego znaku, znaczą kolejno:
 - numer wierzchołka, z którego definiowana krawędź wychodzi (gdzie wierzchołki numerowane są od zera),
 - identyfikator drugiego węzła z pary dla krawędzi (krawędzie są nieskierowane, więc kolejność zapisu wierzchołków jest umowna),
 - koszt definiowanej krawędzi.

Jak zapisać wygenerowany graf?

W celu zapisania do pliku grafu, wygenerowanego jedną z dwóch ostatnich opisanych przez nas metod, należy posłużyć się poniżej prezentowanym fragmentem kodu:

```

1 GraphIF* g = GraphUtils::getRandomGraph(10, 0.5, 10, 15);
2
3 OutputUtils::exportGraph(g, "ścieżka do pliku", OutputFormat::GR);
4
5 MemoryUtils::removeGraph(g, true, true);

```

A.2.2 Rozwiązywanie problemów grafowych

Wszystkie prezentowane przypadki użycia, pozwalające na rozwiązywanie problemów grafowych, mają bardzo podobną do siebie budowę. Uruchomienie procesu poszukiwania minimalnego drzewa rozpinającego dla grafu G wymaga od nas stworzenia instancji obiektu, który implementuje odpowiednie dla każdego problemu metody. Wszystkie obiekty, którymi będziemy się posługiwać, mają budowę, pozwalającą na bardzo łatwą manipulację nimi. Przykładem takiego obiektu jest `MSTSolverImpl` (razem z podobnymi sobie dalej nazywany krótko *solverem*),

```

1 GraphIF* g = GraphUtils::getRandomGraph(10, 0.5, 10, 15);
2 MSTSolverIF* mstSolver = new MSTSolverImpl { g };
3
4 delete mstSolver;
5 MemoryUtils::removeGraph(g, true, true);

```

który swoją definicję opiera na zawartości pliku nagłówkowego `<RIT/mstsolver/MSTSolverInclude.hpp>`, który wygląda następująco:

```

1 #ifndef SRC_INCLUDE_MSTSOLVER_MSTSOLVERINCLUDE_HPP_
2 #define SRC_INCLUDE_MSTSOLVER_MSTSOLVERINCLUDE_HPP_
3
4 #define MSTSolverIF_PrimeHeap
5
6 #include "MSTSolverIF.hpp"
7
8 #ifdef MSTSolverIF_PrimeHeap
9 #include "PrimeHeap.hpp"
10 #elif defined(MSTSolverIF_Kruskal)
11 #include "Kruskal.hpp"
12 #endif
13
14 #typedef PrimeHeap MSTSolverImpl;
15
16 #endif /* SRC_INCLUDE_MSTSOLVER_MSTSOLVERINCLUDE_HPP_ */

```

Ten oraz znaczna część obiektów, które implementuje biblioteka RIT, jest przedstawiona właśnie w taki sposób. Taka modularna budowa biblioteki RIT umożliwia szybką zmianę implementacji praktycznie dowolnej struktury, która jest wykorzystywana w aplikacji, bez potrzeby jej przebudowywania (w takiej sytuacji od użytkownika wymaga się jedynie podmiany dwóch linii odpowiedniego pliku nagłówkowego by przedefiniować dany typ obiektu na inny).

Inną możliwością wyboru odpowiedniego narzędzia jest skorzystanie z klasy pomocniczej `SolverFactory`, która bazuje na identycznym rozwiążaniu, co przedstawione powyżej, jednak pozwala na dużo wygodniejszą formę pracy z aplikacją.



Jak rozwiązać problem minimalnego drzewa rozpinającego?

Aby rozwiązać dowolną instancję zadanego problemu, możemy posłużyć się poniższym przykładem, który generuje przykładowy graf, tworzy domyślną (`MSTSolverEnum::DEFAULT`) klasę `solvera`, dostarcza nam rozwiązanie problemu, wypisuje je a na końcu usuwa wszystkie obiekty z pamięci:

```
1 GraphIF* g = GraphUtils::getRandomGraph(10, 0.5, 10, 15);
2 MSTSolverIF* mstSolver = SolverFactory::getMSTSolver(MSTSolverEnum::DEFAULT, g);
3 EdgeSetIF* solution = mstSolver->getMST();
4
5 std::cout << solution->toString() << std::endl;
6 std::cout << solution->getTotalEdgeCost() << std::endl;
7
8 MemoryUtils::removeCollection(solution, false);
9 delete mstSolver;
10 MemoryUtils::removeGraph(g, true, true);
```

Oprócz takich klasycznych implementacji jak algorytm Prima czy Kruskala, biblioteka RIT umożliwia nam rozwiązanie tych samych problemów za pomocą programowania liniowego bądź całkowitoliczbowego. Do poprawnego funkcjonowania tej części funkcjonalności niezbędny jednak jest pakiet optymalizacyjny IBM® CPLEX® OPTIMIZER. Z uwagi na powyższy fakt, klasy, które umożliwiają dostęp do tego typu `solverów`, nie mają wspólnego interfejsu z pozostałymi obiektami, tak aby można było łatwo usunąć z projektu biblioteki RIT kłopotliwe zależności (np. w przypadku braku owego pakietu).

```
1 GraphIF* g = GraphUtils::getRandomGraph(10, 0.5, 10, 15);
2 CPLEX_LP_MSTSolverIF* lpMstSolver = new CPLEX_LP_MSTSolverImpl { g };
3 //MSTSolverIF* mstSolver = SolverFactory::getMSTSolver(MSTSolverEnum::DEFAULT, g);
4 //EdgeSetIF* solution = mstSolver->getMST();
5 EdgeSetIF* solution = lpMstSolver->getMST();
6
7 std::cout << solution->toString() << std::endl;
8 std::cout << solution->getTotalEdgeCost() << std::endl;
9
10 MemoryUtils::removeCollection(solution, false);
11 delete lpMstSolver;
12 //delete mstSolver;
13 MemoryUtils::removeGraph(g, true, true);
```

W powyższym kodzie celowo pozostawiono komentarze, aby uwypuklić fakt prostoty zamiany jednego algorytmu na inny. Choć powiedzieliśmy, że obiekty, które opierają się na pakiecie IBM® CPLEX® OPTIMIZER, nie mają części wspólnej z pozostałymi klasami, implementują one niemal te same interfejsy, dzięki którym możemy zminimalizować liczbą potrzebnych do wykonania zmian, w przypadku chęci wykorzystania innego typu `solvera`.

Jak rozwiązać problem minimalnego drzewa rozpinającego w wersji INCREMENTAL?

Poniżej przedstawiono przykładowy kod, dzięki któremu szybko można zacząć pracę z problemami typu INCREMENTAL MINIMUM SPANNING TREE. Jak możemy się przekonać, nic nie stoi na przeszkodzie by łączyć wiele różnych instancji `solverów` w jednym ciągu obliczeń, mając dodatkowo wpływ na rodzaj każdego z nich (poprzez edycję plików `<RIT/mstsolver/MSTSolverInclude.hpp>` oraz `<RIT/imstsolver/IMSTSolverInclude.hpp>` lub posługiwanie się klasą `SolverFactory`):

```
1 IncrementalParam k = 5;
2 GraphIF* g = GraphUtils::getRandomGraph(10, 0.5, 10, 15);
3 MSTSolverIF* mstSolver = SolverFactory::getMSTSolver(MSTSolverEnum::DEFAULT, g);
4 IMSTSolverIF* imstSolver = SolverFactory::getIMSTSolver(IMSTSolverEnum::DEFAULT, g);
5 EdgeSetIF* baseSolution = mstSolver->getMST();
6 delete mstSolver;
7
8 EdgeSetIF* solution = imstSolver->getIMST(k, baseSolution);
9 MemoryUtils::removeCollection(baseSolution, false);
10
11 std::cout << solution->toString() << std::endl;
12 std::cout << solution->getTotalEdgeCost() << std::endl;
13
14 MemoryUtils::removeCollection(solution, false);
15 delete imstSolver;
16 MemoryUtils::removeGraph(g, true, true);
```



Analogicznie do problemu minimalnego drzewa rozpinającego możemy dla powyższego kodu skonstruować jego odpowiednik, który w całości będzie oparty o pakiet optymalizacyjny IBM® CPLEX® OPTIMIZER. Należy mieć jednak świadomość tego, że takie rozwiązanie będzie dużo mniej wydajne niż prezentowane i zostało zaimplementowane jedynie w celu potwierdzenia poprawności działania pozostałych solverów.

Jak rozwiązać problem adwersarza?

W celu rozwiązania problemu adwersarza potrzebujemy dodatkowo wczytać do programu sam zestaw scenariuszy, z których to algorytm będzie korzystał. Przykład, jak to zrobić, jest prezentowany poniżej:

```

1 IncrementalParam k = 5;
2 GraphIF* g = InputUtils::readGraph("ściezka do pliku *.gr", InputFormat::GR, InputMode::HDD);
3 GraphEdgeCostsSet adversarialScenarios { };
4 MSTSolverIF* mstSolver = SolverFactory::getMSTSolver(g);
5 AIMSTSsolution aimstsSolution { };
6 AIMSTSsolverIF* aimstsSolver = SolverFactory::getAIMSTSsolver(g,
7 adversarialScenarios, k);
8
9 EdgeSetIF* baseSolution = mstSolver->getMST();
10 delete mstSolver;
11
12 EdgeSetIF* solution { };
13
14 for (int i = 1; i < 10; i += 1) {
15     adversarialScenarios.insert(
16         InputUtils::readCosts("ściezka do pliku *.gr", InputFormat::GR, InputMode::HDD)
17     );
18 }
19
20 aimstsSolution = aimstsSolver->getMST(baseSolution);
21 solution = AIMSTSutils::getEdgeSet(aimstsSolution);
22
23 MemoryUtils::removeCollection(baseSolution, false);
24 delete aimstsSolver;
25
26 std::cout << solution->toString() << std::endl;
27 std::cout << solution->getTotalEdgeCost() << std::endl;
28
29 MemoryUtils::removeCollection(solution, false);
30 MemoryUtils::removeScenarioSet(adversarialScenarios);
31 MemoryUtils::removeGraph(g, true, true);

```

Jak rozwiązać problem odpornej optymalizacji z możliwością poprawy?

Tak jak w przypadku odpowiedzi na wszystkie poprzednie pytania, tak i tutaj zaprezentujemy przykładowy kod, dzięki któremu możliwe jest szybkie wczytanie danych dla problemu RECOVERABLE ROBUST INCREMENTAL SPANNING TREE oraz uruchomienie algorytmu TABU SEARCH, który jest domyślną implementacją obiektu, rozwiązującego dane zagadnienie (zwracającego rozwiązanie jak najbliższe optymalnego).

```

1 IncrementalParam k { 5 };
2 TabuIterationCount tabuPeriod { 100 };
3 TabuIterationCount numberofPathIterations { 50 };
4 TabuIterationCount numberofIterations { 2000 };
5
6 GraphIF* g = InputUtils::readGraph("ściezka do pliku *.gr", InputFormat::GR,
7 InputMode::HDD);
8 GraphEdgeCostsSet adversarialScenarios { };
9 RIMSTSsolverIF* rimstsSolver = new RIMSTSsolverImpl { AIMSTSsolverEnum::DEFAULT,
10     IMSTSsolverEnum::DEFAULT, MSTsolverEnum::DEFAULT, g, adversarialScenarios, k,
11     tabuPeriod, numberofPathIterations, numberofIterations
12 };
13
14 for (int i = 1; i < 10; i += 1) {
15     adversarialScenarios.insert(
16         InputUtils::readCosts("ściezka do pliku *.gr", InputFormat::GR, InputMode::HDD)
17     );
18 }

```

```
19
20 EdgeSetIF* solution = rimstSolver->getMST();
21 delete rimstSolver;
22
23 std::cout << solution->toString() << std::endl;
24 std::cout << solution->getTotalEdgeCost() << std::endl;
25
26 MemoryUtils::removeCollection(solution, false);
27 MemoryUtils::removeScenarioSet(adversarialScenarios);
28 MemoryUtils::removeGraph(g, true, true);
```

A.3 Przykładowa sesja

Poniżej zaprezentowano zapis zdarzeń, powstały w wyniku uruchomienia algorytmu dla problemu INCREMENTAL MINIMUM SPANNING TREE, dla grafu o 5 wierzchołkach i 6 krawędziach, gdzie zapisywanyymi informacjami są tylko te zdarzenia, których źródło znajduje się w jednym z plików: IMSTSOLVERIF.CPP lub BINARYSEARCH_V2.CPP (wszystkie inne zostały pominięte).

```
1 [...]
2 2016-06-19 18:45:29,123 INFO IMSTSolverIF (./src/src/imstsolver/IMSTSolverIF.cpp:59) - No new base edge set was used. Running Incremental MST Solver on previous solution:
3   (0) <----- [ 3 ] -----> (1)
4   (1) <----- [ 3 ] -----> (2)
5   (2) <----- [ 5 ] -----> (3)
6   (4) <----- [ 2 ] -----> (3)
7 with total edge costs: 13.00000000.
8 2016-06-19 18:45:29,124 INFO IMSTSolverIF (./src/src/imstsolver/IMSTSolverIF.cpp:68) - New set of edges' costs has been given. Changing costs of edges as shown below:
9   (0) <----- [ 3 ==> 4 ] -----> (1)
10  (3) <----- [ 8 ==> 6 ] -----> (0)
11  (1) <----- [ 6 ==> 1 ] -----> (4)
12  (4) <----- [ 2 ==> 9 ] -----> (3)
13  (2) <----- [ 5 ==> 7 ] -----> (3).
14 2016-06-19 18:45:29,125 INFO BinarySearch_v2 (./src/src/imstsolver/BinarySearch_v2.cpp:345) - There are some changes it costs of graph's edges.
15 New unbounded solution will be calculated for k parameter value: 1.
16 2016-06-19 18:45:29,126 INFO IMSTSolverIF (./src/src/imstsolver/IMSTSolverIF.cpp:101) - New MST solution has given set of edges that is not a part of original solution:
17   (1) <----- [ 1 ] -----> (4)
18   (3) <----- [ 6 ] -----> (0).
19 Total number of edges in set: 2.
20 2016-06-19 18:45:29,126 INFO BinarySearch_v2 (./src/src/imstsolver/BinarySearch_v2.cpp:360) - Unbounded solution that was found has got unacceptable number of edges
21 that are not in base solution. Looking for 1-bounded optimal solution for Incremental MST problem...
22 2016-06-19 18:45:29,126 INFO BinarySearch_v2 (./src/src/imstsolver/BinarySearch_v2.cpp:51) - Generating graph that contains smaller set of edges
23 based on base edge set (with 4 edges):
24   (0) <----- [ 4 ] -----> (1)
25   (1) <----- [ 3 ] -----> (2)
26   (2) <----- [ 7 ] -----> (3)
27   (4) <----- [ 9 ] -----> (3),
28 and k-unbounded edge set that forms MST solution (4 edges):
29   (0) <----- [ 4 ] -----> (1)
30   (1) <----- [ 1 ] -----> (4)
31   (1) <----- [ 3 ] -----> (2)
32   (3) <----- [ 6 ] -----> (0).
33 2016-06-19 18:45:29,128 INFO BinarySearch_v2 (./src/src/imstsolver/BinarySearch_v2.cpp:84) - Graph with 5 vertices and the following set of edges
34 was generated (with 6 edges) and will be used to solve MST in Incremental version:
35   (0) <----- [ 4 ] -----> (1)
36   (3) <----- [ 6 ] -----> (0)
37   (1) <----- [ 1 ] -----> (4)
38   (4) <----- [ 9 ] -----> (3)
39   (1) <----- [ 3 ] -----> (2)
40   (2) <----- [ 7 ] -----> (3).
41 2016-06-19 18:45:29,130 INFO BinarySearch_v2 (./src/src/imstsolver/BinarySearch_v2.cpp:93) - Edge preprocessing. Cost of every edge
42 in shrunken graph (6 edges) will be temporary changed as follows:
43   c'_{e_{i,j}} = c_{e_{i,j}} + (mi^2 + i)/ (m + 1)^3.
44 2016-06-19 18:45:29,130 INFO BinarySearch_v2 (./src/src/imstsolver/BinarySearch_v2.cpp:101) - Temporary changing edge cost as follows:
45   (0) <----- [ 4 ==> 4.02041 ] -----> (1).
46 2016-06-19 18:45:29,130 INFO BinarySearch_v2 (./src/src/imstsolver/BinarySearch_v2.cpp:101) - Temporary changing edge cost as follows:
47   (3) <----- [ 6 ==> 6.0758 ] -----> (0).
48 2016-06-19 18:45:29,131 INFO BinarySearch_v2 (./src/src/imstsolver/BinarySearch_v2.cpp:101) - Temporary changing edge cost as follows:
49   (1) <----- [ 1 ==> 1.16618 ] -----> (4).
50 2016-06-19 18:45:29,131 INFO BinarySearch_v2 (./src/src/imstsolver/BinarySearch_v2.cpp:101) - Temporary changing edge cost as follows:
51   (4) <----- [ 9 ==> 9.29155 ] -----> (3).
52 2016-06-19 18:45:29,131 INFO BinarySearch_v2 (./src/src/imstsolver/BinarySearch_v2.cpp:101) - Temporary changing edge cost as follows:
53   (1) <----- [ 3 ==> 3.4519 ] -----> (2).
54 2016-06-19 18:45:29,132 INFO BinarySearch_v2 (./src/src/imstsolver/BinarySearch_v2.cpp:101) - Temporary changing edge cost as follows:
55   (2) <----- [ 7 ==> 7.64723 ] -----> (3).
56 2016-06-19 18:45:29,132 TRACE BinarySearch_v2 (./src/src/imstsolver/BinarySearch_v2.cpp:125) - Generating data to calculate  $\lambda$  parameters.
57 2016-06-19 18:45:29,132 INFO BinarySearch_v2 (./src/src/imstsolver/BinarySearch_v2.cpp:141) - Based on original MST solution's edges:
58   (0) <----- [ 4.02041 ] -----> (1)
59   (1) <----- [ 3.4519 ] -----> (2)
60   (2) <----- [ 7.64723 ] -----> (3)
61   (4) <----- [ 9.29155 ] -----> (3)
62 with 4 edges, MST for new costs:
63   (0) <----- [ 4.02041 ] -----> (1)
64   (1) <----- [ 1.16618 ] -----> (4)
65   (1) <----- [ 3.4519 ] -----> (2)
66   (3) <----- [ 6.0758 ] -----> (0)
67 with 4 edges, following sets was generated:
68 T^{\{0\}} \ T^{\{*}} (edges that are only in old solution - 2 edges):
69   (4) <----- [ 9.29155 ] -----> (3)
70   (2) <----- [ 7.64723 ] -----> (3),
71 T^{\{*}} \ T^{\{0\}} (edges that are only in new solution - 2 edges):
72   (3) <----- [ 6.0758 ] -----> (0)
73   (1) <----- [ 1.16618 ] -----> (4).
74 2016-06-19 18:45:29,135 TRACE BinarySearch_v2 (./src/src/imstsolver/BinarySearch_v2.cpp:145) - Generating set of edges
75 that are in complementary set of original and new minimum spanning tree sets.
76 2016-06-19 18:45:29,136 TRACE BinarySearch_v2 (./src/src/imstsolver/BinarySearch_v2.cpp:154) - Sorting generated set in ascending order.
```



```

77 2016-06-19 18:45:29,136 TRACE BinarySearch_v2 (.../src/src/imstsolver/BinarySearch_v2.cpp:165) - Generating set of edges
78  that are in complementary set of new and original minimum spanning tree sets.
79 2016-06-19 18:45:29,136 TRACE BinarySearch_v2 (.../src/src/imstsolver/BinarySearch_v2.cpp:174) - Sorting generated set in descending order.
80 2016-06-19 18:45:29,136 INFO BinarySearch_v2 (.../src/src/imstsolver/BinarySearch_v2.cpp:187) - Checking correctness lower and upper bounds of  $\lambda$  parameter...
81 2016-06-19 18:45:29,136 INFO BinarySearch_v2 (.../src/src/imstsolver/BinarySearch_v2.cpp:210) - Using default  $\lambda$  bounds:
82 [1.571428571
83 : 8.125364431].
84 Generating indexes of  $\lambda$  parameter feasible set...
85 2016-06-19 18:45:29,136 INFO BinarySearch_v2 (.../src/src/imstsolver/BinarySearch_v2.cpp:421) - Performing Incremental MST binary search.
86 will start new iteration with initial  $\lambda$  bounds:
87 [1.571428571: 8.125364431].
88 2016-06-19 18:45:29,136 TRACE BinarySearch_v2 (.../src/src/imstsolver/BinarySearch_v2.cpp:433) - Generating lower indexes for feasible  $\lambda$  parameter set,
89 starting with  $\lambda(1, 0)$  (while first parameter will be decreasing, second - increasing).
90 2016-06-19 18:45:29,137 TRACE BinarySearch_v2 (.../src/src/imstsolver/BinarySearch_v2.cpp:437) - Performing search for minimal index 'j' such that:
91  $\lambda_{LB}$  <=  $\lambda(1, j)$ ,
92  $\lambda_{LB}$  = 1.571428571
93 2016-06-19 18:45:29,137 TRACE BinarySearch_v2 (.../src/src/imstsolver/BinarySearch_v2.cpp:448) - Minimal index 'j' such that:
94  $\lambda_{LB}$  <=  $\lambda(1, j)$ ,
95  $\lambda_{LB}$  = 1.571428571
96 has been found and it equals 0 (3.215743440).
97 2016-06-19 18:45:29,137 TRACE BinarySearch_v2 (.../src/src/imstsolver/BinarySearch_v2.cpp:437) - Performing search for minimal index 'j' such that:
98  $\lambda_{LB}$  <=  $\lambda(0, j)$ ,
99  $\lambda_{LB}$  = 1.571428571
100 2016-06-19 18:45:29,138 TRACE BinarySearch_v2 (.../src/src/imstsolver/BinarySearch_v2.cpp:448) - Minimal index 'j' such that:
101  $\lambda_{LB}$  <=  $\lambda(0, j)$ ,
102  $\lambda_{LB}$  = 1.571428571
103 has been found and it equals 0 (1.571428571).
104 2016-06-19 18:45:29,138 TRACE BinarySearch_v2 (.../src/src/imstsolver/BinarySearch_v2.cpp:462) - Generating upper indexes for feasible  $\lambda$  parameter set,
105 starting with  $\lambda(0, 1)$  (while first parameter will be increasing, second - decreasing).
106 2016-06-19 18:45:29,138 TRACE BinarySearch_v2 (.../src/src/imstsolver/BinarySearch_v2.cpp:466) - Performing search for maximal index 'j' such that:
107  $\lambda(0, j)$  <=  $\lambda_{UB}$ ,
108  $\lambda_{UB}$  = 8.125364431.
109 2016-06-19 18:45:29,139 TRACE BinarySearch_v2 (.../src/src/imstsolver/BinarySearch_v2.cpp:474) - Maximal index 'j' such that:
110  $\lambda(0, j)$  <=  $\lambda_{UB}$ ,
111  $\lambda_{UB}$  = 8.125364431
112 has been found and it equals 1 (6.481049563).
113 2016-06-19 18:45:29,139 TRACE BinarySearch_v2 (.../src/src/imstsolver/BinarySearch_v2.cpp:466) - Performing search for maximal index 'j' such that:
114  $\lambda(1, j)$  <=  $\lambda_{UB}$ ,
115  $\lambda_{UB}$  = 8.125364431.
116 2016-06-19 18:45:29,139 TRACE BinarySearch_v2 (.../src/src/imstsolver/BinarySearch_v2.cpp:474) - Maximal index 'j' such that:
117  $\lambda(1, j)$  <=  $\lambda_{UB}$ ,
118  $\lambda_{UB}$  = 8.125364431
119 has been found and it equals 1 (8.125364431).
120 2016-06-19 18:45:29,140 TRACE BinarySearch_v2 (.../src/src/imstsolver/BinarySearch_v2.cpp:487) - Counting feasible  $\lambda$  parameters...
121 2016-06-19 18:45:29,140 TRACE BinarySearch_v2 (.../src/src/imstsolver/BinarySearch_v2.cpp:510) - For parameter 'i' (0), 2 expressions of type  $\lambda(i, j)$ 
122 was found (from  $j = 0$  to 1) and will be added to feasible solution which size is now 2.
123 2016-06-19 18:45:29,140 TRACE BinarySearch_v2 (.../src/src/imstsolver/BinarySearch_v2.cpp:510) - For parameter 'i' (1), 2 expressions of type  $\lambda(i, j)$ 
124 was found (from  $j = 0$  to 1) and will be added to feasible solution which size is now 4.
125 2016-06-19 18:45:29,140 INFO BinarySearch_v2 (.../src/src/imstsolver/BinarySearch_v2.cpp:525) - Size of generated  $\lambda$  parameter feasible set
126 is small enough (4) to execute  $\lambda$ -based MST binary search (threshold was 4).
127 2016-06-19 18:45:29,140 TRACE BinarySearch_v2 (.../src/src/imstsolver/BinarySearch_v2.cpp:534) - Adding feasible  $\lambda$  parameter which satisfy:
128 1.571428571 <=  $\lambda(0, 0)$  = 1.571428571 <= 8.125364431.
129 2016-06-19 18:45:29,141 TRACE BinarySearch_v2 (.../src/src/imstsolver/BinarySearch_v2.cpp:534) - Adding feasible  $\lambda$  parameter which satisfy:
130 1.571428571 <=  $\lambda(0, 1)$  = 6.481049563 <= 8.125364431.
131 2016-06-19 18:45:29,141 TRACE BinarySearch_v2 (.../src/src/imstsolver/BinarySearch_v2.cpp:534) - Adding feasible  $\lambda$  parameter which satisfy:
132 1.571428571 <=  $\lambda(1, 0)$  = 3.215743440 <= 8.125364431.
133 2016-06-19 18:45:29,141 TRACE BinarySearch_v2 (.../src/src/imstsolver/BinarySearch_v2.cpp:534) - Adding feasible  $\lambda$  parameter which satisfy:
134 1.571428571 <=  $\lambda(1, 1)$  = 8.125364431 <= 8.125364431.
135 2016-06-19 18:45:29,142 INFO BinarySearch_v2 (.../src/src/imstsolver/BinarySearch_v2.cpp:246) - Executing binary search for 4  $\lambda$  parameters, sorted in ascending order...
136 2016-06-19 18:45:29,142 INFO BinarySearch_v2 (.../src/src/imstsolver/BinarySearch_v2.cpp:251) - Searching for a solution in  $\lambda$  parameter space bounded by:
137 lower index : 0,
138 lower  $\lambda$  value : 1.571428571
139 upper index : 3,
140 upper  $\lambda$  value : 8.125364431.
141 2016-06-19 18:45:29,142 INFO BinarySearch_v2 (.../src/src/imstsolver/BinarySearch_v2.cpp:256) - Calculate MST solution for selected  $\lambda$  parameter:
142 index : 1,
143  $\lambda$  value : 3.215743440.
144 2016-06-19 18:45:29,143 INFO BinarySearch_v2 (.../src/src/imstsolver/BinarySearch_v2.cpp:217) - Setting new graph's edges cost based on  $\lambda$  parameter value: 3.215743440...
145 2016-06-19 18:45:29,143 INFO BinarySearch_v2 (.../src/src/imstsolver/BinarySearch_v2.cpp:224) - Temporary changing edge cost as follows:
146 (0) <----- [ 4.02041 => 0.804665 ] -----> (1).
147 2016-06-19 18:45:29,143 INFO BinarySearch_v2 (.../src/src/imstsolver/BinarySearch_v2.cpp:224) - Temporary changing edge cost as follows:
148 (1) <----- [ 3.4519 => 0.236152 ] -----> (2).
149 2016-06-19 18:45:29,144 INFO BinarySearch_v2 (.../src/src/imstsolver/BinarySearch_v2.cpp:224) - Temporary changing edge cost as follows:
150 (2) <----- [ 7.64723 => 4.43149 ] -----> (3).
151 2016-06-19 18:45:29,144 INFO BinarySearch_v2 (.../src/src/imstsolver/BinarySearch_v2.cpp:224) - Temporary changing edge cost as follows:
152 (4) <----- [ 9.29155 => 6.0758 ] -----> (3).
153 2016-06-19 18:45:29,144 INFO IMSTSolverIF (.../src/src/imstsolver/IMSTSolverIF.cpp:101) - New MST solution has given set of edges that is not a part of original solution:
154 (1) <----- [ 1.16618 ] -----> (4).
155 Total number of edges in set: 1.
156 2016-06-19 18:45:29,145 INFO BinarySearch_v2 (.../src/src/imstsolver/BinarySearch_v2.cpp:303) - For  $\lambda$  parameter value 3.215743440 an optimal solution
157 has been found (with total cost of edges: 6.638483965). Solution includes following edges:
158 (0) <----- [ 0.804665 ] -----> (1)
159 (1) <----- [ 0.236152 ] -----> (2)
160 (1) <----- [ 1.16618 ] -----> (4)
161 (2) <----- [ 4.43149 ] -----> (3)
162 in which there are 1 edges that are not in base edge set:
163 (1) <----- [ 1.16618 ] -----> (4).

```