Załącznik nr 1/1
do Zarządzenia Rektora PG nr 5/2015 z 10 lutego 2015 r.

POLITECHNIKA GDAŃSKA

WYDZIAŁ ELEKTRONIKI, TELEKOMUNIKACJI I INFORMATYKI

Imię i nazwisko autora rozprawy: Michał Pańczyk
Dyscyplina naukowa: informatyka

ROZPRAWA DOKTORSKA

Tytuł rozprawy w języku polskim: Algorytmy samostabilizujące w sieciach o wybranych topologiach

Tytuł rozprawy w języku angielskim: Self-stabilizing algorithms in certain topology networks

Promotor *podpis*	Drugi promotor *podpis*
dr hab. Halina Bielak, prof. nadzw. UMCS	
Promotor pomocniczy *podpis*	Kopromotor *podpis*

Gdańsk, rok 2016

Algorytmy samostabilizujące w sieciach o wybranych topologiach

Michał Pańczyk

praca doktorska wykonana pod kierunkiem
dr hab. Haliny Bielak, prof. nadzw. UMCS

Wydział Elektroniki, Telekomunikacji i Informatyki
Politechniki Gdańskiej

Gdańsk * 2016

Serdecznie dziękuję
Pani dr hab. Halinie Bielak,
prof. nadzw. UMCS
za zainteresowanie mnie tematem
algorytmów samostabilizujących,
a także za współpracę i pomoc
w czasie badań nad nimi.

Spis treści

Wykaz oznaczeń **1**

1 Wprowadzenie **3**
1.1 Wstęp i tezy rozprawy 3
1.2 Algorytmy samostabilizujące 4
1.3 Zarys teorii grafów 10

2 Przegląd dotychczasowych wyników **14**
2.1 Algorytmy dla dowolnych grafów 14
2.2 Algorytmy dla drzew 16

3 Centra grafów maksymalnych zewnętrznie planarnych i ich iloczynów kartezjańskich **19**
3.1 Wprowadzenie 19
3.2 Klasyczny algorytm wyszukiwania centrum 20
3.3 Samostabilizujący algorytm wyszukiwania centrum 22
3.4 Rozszerzenie na iloczyn kartezjański MOP z K_2 29
3.5 Rozszerzenie na iloczyn kartezjański MOP z P_m 31

4 Wybrane algorytmy samostabilizujące dla drzew **35**
4.1 Algorytm znajdowania ważonego centroidu 35
4.2 Algorytm wyznaczania centrum artykulacyjnego 44

5 Zbiór cykli fundamentalnych w dowolnym grafie **49**
5.1 Wprowadzenie 49
5.2 Notacja 50
5.3 Algorytm 51
5.4 Zbieżność i złożoność 56

6 Program komputerowy **59**

7 Podsumowanie **61**

Bibliografia **63**

Dodatek A Kod programu **70**

Wykaz oznaczeń

Przedstawiamy poniżej listę oznaczeń, używanych w niniejszej pracy. Pomijamy oznaczenia specyficzne dla poszczególnych algorytmów, w szczególności znaczenie zmiennych, gdyż są one wyjaśnione bezpośrednio przed pierwszym ich użyciem.

$V(G)$	zbiór wierzchołków grafu G
$E(G)$	zbiór krawędzi grafu G
$\|A\|$	moc zbioru A
n	liczba wierzchołków grafu (w dyskusji na temat złożoności)
$Diam$	średnica grafu
$N(v)$	zbiór sąsiadów wierzchołka v
$G-v$	graf powstały po usunięciu wierzchołka v z grafu G.
$\deg(v)$	stopień wierzchołka v, $\deg(v) = \|N(v)\|$
Δ	stopień grafu
$d(u, v)$	odległość między wierzchołkami u oraz v
$e(v)$	acentryczność wierzchołka v
$e(v, u, s)$	acentryczność krawędziowa wierzchołka v względem krawędzi $\{u, v\}$ po stronie s
$r(G)$	promień grafu G
$C(G)$	centrum grafu G
$G[A]$	graf indukowany przez zbiór wierzchołków A w grafie G
$A \times B$	iloczyn kartezjański zbiorów lub grafów A i B

$A \oplus B$	różnica symetryczna zbiorów lub grafów A i B
G^*	graf dualny grafu G
K_n	graf pełny o n wierzchołkach
P_n	ścieżka o n wierzchołkach
C_n	cykl o n wierzchołkach
MOP	graf maksymalny zewnętrznie planarny
$\emptyset$	pusta strona w grafie maksymalnym zewnętrznie planarnym
w_v	waga wierzchołka v
$\mathcal{W}_G$	waga grafu G
$z\uparrow$	zmienna z ma lokalnie poprawną wartość
$z\downarrow$	zmienna z ma lokalnie niepoprawną wartość

Rozdział 1

Wprowadzenie

1.1 Wstęp i tezy rozprawy

Praca niniejsza jest poświęcona samostabilizującym algorytmom znajdującym newralgiczne węzły w sieciach. Rozpatrujemy różne definicje tego, co jest newralgicznym elementem systemu rozproszonego; jednak zawsze będzie to kryterium oparte na strukturze sieci. W każdym przypadku wynikiem działania algorytmu będzie wskazanie węzła sieci, którego znaczenie jest największe zgodnie z przyjętymi kryteriami. Raz będzie to węzeł, w którym najbardziej opłaca się umiejscowić centrum dowodzenia sieci, innym razem będzie to element sieci, którego awaria lub wyłączenie przyniosłoby największe straty w jakości funkcjonowania systemu. O ważności może również przesądzać potencjalna liczba połączeń między węzłami sieci, które odbywają się za pośrednictwem danego węzła.

Niezależnie od przyjętego kryterium, w każdym z tych przypadków będziemy starali się wykorzystać narzuconą topologię sieci do skonstruowania nowego lub zmniejszenia złożoności obliczeniowej już istniejącego samostabilizującego algorytmu działającego na danym systemie. W przypadku sieci drzewiastych korzyścią w stosunku do sieci o dowolnej topologii, jest zapewnienie — na mocy definicji drzewa — braku istnienia cykli. Ma to niebagatelne znaczenie przy konstruowaniu algorytmów samostabilizujących, ponieważ likwiduje problem np. cyklicznego wędrowania po sieci fałszywych wartości obliczanych przez algorytm.

Podobnie w przypadku grafów maksymalnych zewnętrznie planarnych — mimo występowania w nich cykli — unikamy zapętlenia dzięki wykorzystaniu faktu, że ich graf słaby dualny jest drzewem.

Dodatkowo przedstawimy model systemów quasi-samostabilizujących, który wprowadza możliwość chronienia przed przejściowym błędem pewnego

zbioru zmiennych w węźle obliczeniowym. Na przykładzie algorytmu znajdującego zbiór cykli fundamentalnych pokażemy, że model ten umożliwia stworzenie algorytmów działających szybciej w stosunku do podstawowego modelu samostabilizacji.

Powyższe rozważania pozwalają nam sformułować następujące tezy niniejszej rozprawy:

- Istnieją topologie sieci rozproszonych, które umożliwiają konstrukcję algorytmów samostabilizujących działających efektywniej, niż algorytmy dla sieci o dowolnej topologii.
- Zastosowanie modelu quasi-samostabilizacji umożliwia zwiększenie efektywności działania niektórych algorytmów samostabilizujących.

Dodatkowo w rozdziale 6 prezentujemy napisaną przez nas aplikację komputerową, która znacząco ułatwia testowanie nowo tworzonych algorytmów na przykładowych grafach. W dodatku A umieściliśmy jej kod źródłowy. Program ten jest symulatorem działania systemu samostabilizującego uruchomionego pod kontrolą wybranego algorytmu.

1.2 Algorytmy samostabilizujące

Od początku historii obliczeń jednymi z najbardziej pożądanych cech były duża szybkość wykonywania oraz niezawodność. Obliczenia rozproszone pomagają osiągnąć te cele. Przykładem jeszcze sprzed ery komputerów (jakie znamy obecnie) może być podział zadania wyznaczenia tablic logarytmicznych pomiędzy wielu rachmistrzów, z których każdy miał za zadanie obliczyć logarytmy stosunkowo niewielkiej części wszystkich liczb mających znaleźć się w tablicach. Jeśli obliczenie każdej wartości było wykonane przez dwóch rachmistrzów niezależnie od siebie, przy łączeniu cząstkowych wyników w całość można je było ze sobą porównać i w ten sposób znacząco zwiększyć szanse na wykrycie pomyłek. Oczywiście gdy do pracy zatrudnieni byli trzej (lub więcej) rachmistrze, cała praca wykonywana była w krótszym czasie, niż gdyby całość miał obliczyć jeden lub dwaj z nich.

W ostatnich kilkunastu latach motywacja do dzielenia obliczeń na wiele komputerów wzrosła z powodu spowolnienia przyrostu wydajności nowo produkowanych elektronicznych jednostek obliczeniowych [Wal16]. Pisząc „jednostek", mamy na myśli procesory jednordzeniowe lub też pojedyncze rdzenie procesorów. To spowolnienie świadczy o załamaniu się wąsko rozumianego prawa Moore'a [Moo65].

Jednym z rozwiązań tego problemu jest właśnie podział zadań pomiędzy wiele procesorów. Jeśli algorytm jest podzielony na więcej, niż jeden wątek

obliczeniowy, mówimy o programie *współbieżnym*. Współbieżność możemy rozumieć jako potencjalną równoległość wykonywania wątków programu [Var13]. Jeśli program współbieżny wykonuje się na jednym procesorze, wykonanie poszczególnych wątków rozłożone jest w czasie i mówimy wtedy o *przeplocie*. *Równoległe* wykonanie programu ma miejsce, gdy algorytm wykonuje się jednocześnie na wielu procesorach. Jeśli te procesory nie są częścią jednego komputera (mogą być nawet od siebie oddalone o tysiące kilometrów), mówimy o programie i systemie *rozproszonym*.

Programowanie współbieżne, a w szczególności rozproszone, stwarza trudności, jakie nie występują w programach sekwencyjnych. Programy współbieżne mogą ulec blokadzie, zakleszczeniu, zagłodzeniu; wymagają też synchronizacji i komunikacji między wątkami [Ben06].

Z drugiej strony systemy rozproszone jako całość są bardziej odporne na awarie. Gdy jakiś węzeł systemu rozproszonego ulegnie awarii, może go zastąpić inny węzeł.

Idea algorytmów samostabilizujących została zaproponowana przez Dijkstrę [Dij74] w roku 1974. Jego artykuł przez dekadę pozostawał niezauważany aż do publikacji Lamporta [Lam85], w której zwrócił on uwagę na doniosłość zaproponowanego przez Dijkstrę podejścia do konstruowania algorytmów rozproszonych. Od tego czasu można zauważyć znaczące zainteresowanie tematem i wielki postęp w tej dziedzinie.

Pracami, które mogą służyć za wprowadzenie do algorytmów samostabilizujących są: przeglądowy artykuł Schneidera [Sch93] oraz książka Doleva [Dol00].

Rozproszony system samostabilizujący będziemy modelować za pomocą spójnych grafów prostych. Węzły obliczeniowe systemu rozproszonego będą modelowane wierzchołkami grafu, zaś ich połączeniom, umożliwiającym bezpośrednią komunikację, będą odpowiadały krawędzie grafu. W niniejszej rozprawie przyjmujemy, że system jest *jednolity* (ang. *uniform*), tzn. każdy węzeł wykonuje ten sam program, co oznacza, że zawiera również jednakowy zestaw przyporządkowanych do niego zmiennych. W literaturze występują również systemy *półjednolite* (ang. *semi-uniform*) [DIM93, GP03], w których jeden z węzłów obliczeniowych wykonuje algorytm inny, niż pozostałe.

W systemach samostabilizujących każdy z wierzchołków zazwyczaj ma swój unikalny w obrębie systemu identyfikator. W definicjach reguł będziemy przyjmować, że wartość i jest identyfikatorem bieżącego wierzchołka, na rzecz którego działa dana reguła. Rozważa się również systemy, w których wierzchołki nie mają identyfikatorów, czyli są *anonimowe*.

Samostabilizujący program (algorytm) działający na opisanym systemie to zbiór reguł manipulujących wartościami zmiennych. Każda reguła składa się z dwóch części:

- wartownika — predykatu zależnego tylko od zmiennych w danym wierzchołku oraz w (bezpośrednio) sąsiadujących wierzchołkach,
- zbioru instrukcji przypisania, które ustawiają wartości zmiennych, zmieniając stan lokalny wierzchołka.

Przykład reguły algorytmu samostabilizującego

---

etykieta:
if $z_i \neq nazwana_wartość$ **then**
$z_i := nazwana_wartość$
where

$$nazwana_wartość = \begin{cases} \max_{j \in N(i)} z_j & \text{jeśli } \max_{j \in N(i)} z_j > 0 \\ 0 & \text{w pozostałych przypadkach} \end{cases}$$

W powyższym przykładzie wartownik to wyrażenie pomiędzy słowami kluczowymi **if** i **then**; instrukcja przypisania znajduje się w kolejnej linii. Blok **where** jest opcjonalny — służy tylko uproszczeniu zapisu rozbudowanych wyrażeń matematycznych poprzez nadanie im krótkich nazw. Zadaniem przedstawionej reguły jest utrzymywanie wartości zmiennej z_i zgodnie z definicją *nazwanej_ wartości*.

Wartownik należy do logiki pierwszego rzędu [Ben12]. Interesować nas będą tylko takie postacie wartowników i instrukcji przypisania, które są obliczalne a ich złożoność obliczeniowa jest niewielka lub też rozmiar danych wejściowych — w kontekście sąsiedztwa pojedynczego wierzchołka np. liczba sąsiadów — jest ograniczony od góry. W praktycznych zastosowaniach powyższe założenia są naturalnie spełnione dzięki charakterowi problemu, który jest rozwiązywany przez dany algorytm, tzn. nie ma potrzeby stosowania wyrażeń, które nie spełniałyby wspomnianych warunków.

Wartości wszystkich zmiennych przyporządkowanych do danego wierzchołka definiują jego *stan lokalny*, zaś suma stanów lokalnych we wszystkich wierzchołkach grafu tworzy *stan globalny* systemu.

Problem grafowy będziemy określać poprzez zdefiniowanie oczekiwanego stanu globalnego systemu, który nazwiemy *stanem dozwolonym* (ang. *legitimate state*). Na przykład dozwolonym stanem dla problemu wyznaczenia ukorzenionego drzewa rozpinającego graf będzie sytuacja, gdzie pewna zmienna w każdym wierzchołku (poza jednym wyróżnionym — korzeniem) wskazuje jednego sąsiada (w korzeniu: zmienna wskazuje własny wierzchołek lub nic nie wskazuje) w ten sposób, że wędrując od dowolnego wierzchołka zgodnie ze wskazaniami wspomnianej zmiennej zawsze dojdziemy do korzenia. Każdy

stan różny od dozwolonego będziemy nazywać *niedozwolonym* (ang. *illegitimate*).

Jeśli w danym wierzchołku wartownik którejś z reguł okaże się prawdziwy, mówimy, że reguła jest *aktywna*. Podobnie będziemy mówili, że wierzchołek ją zawierający również jest *aktywny*. W aktywnym wierzchołku mogą wykonać się instrukcje przypisania jednej z aktywnych reguł na rzecz zmiennych lokalnych dla danego wierzchołka. Mówimy wtedy, że wykonał się *ruch*. Czasem zamiennie będziemy też mówić o wykonaniu reguły.

Wartości zmiennych, które powodują aktywność reguł algorytmu w danym węźle obliczeniowym określamy jako *lokalnie niepoprawne*. Jeśli reguła, której wartownik zależy od takiej zmiennej, wykona się, zmieniając wartość zmiennej w ten sposób, że reguła ta przestaje być aktywna a ponadto nie istnieje żadna inna reguła, która może w kolejnym ruchu zmienić wartość zmiennej, to mówimy, że zmienna osiągnęła *lokalnie poprawną* wartość.

Celem działania algorytmu samostabilizującego jest sprowadzenie systemu ze stanu niedozwolonego do stanu dozwolonego w skończonej liczbie ruchów, co nazywamy *ustabilizowaniem* systemu według danego algorytmu. Jeśli algorytm spełnia to założenie, mówimy, że ma własność *zbieżności* (ang. *convergence*). Dodatkowo algorytm samostabilizujący musi wykazywać się jeszcze *domkniętością* (ang. *closure*), czyli jeśli system znajduje się w stanie dozwolonym, to algorytm nie może go doprowadzić do stanu niedozwolonego.

W tym miejscu możemy jeszcze dokonać podziału na algorytmy *ciche* (ang. *silent*), które zatrzymują się osiągnąwszy stan dozwolony. To oznacza, że w stanie dozwolonym żadna ich reguła nie może być aktywna. Dopełnieniem powyższej klasy (wśród algorytmów domkniętych) są algorytmy *nieciche* (ang. *nonsilent*), które mogą wykonywać ruchy nawet w stanie dozwolonym.

Warunkiem koniecznym dla zbieżności algorytmu jest, aby w stanie niedozwolonym co najmniej jeden wierzchołek w systemie był aktywny.

System może się znaleźć w stanie niedozwolonym w wyniku *przejściowego błędu* (ang. *transient failure*), czyli zmiany topologii grafu będącego modelem systemu lub wartości zmiennej w wierzchołku (wtedy taki wierzchołek zawiera *defekt*), a na poziomie implementacji systemu: zmiany licznika instrukcji algorytmu w jednostce obliczeniowej wierzchołka. Zmiana taka ma charakter zewnętrzny, tzn. nie jest wynikiem działania algorytmu. Ponadto przejściowy błąd może dotyczyć jedynie stanu a nie algorytmu — nasz model zakłada, że przejściowy błąd nie może być skutkiem modyfikacji reguł algorytmu (np. pamięci rozkazów realizujących algorytm w jednostce obliczeniowej). Inną przyczyną niedozwolonego stanu systemu może być to, że w takim stanie został uruchomiony. Zakładamy, że system osiągnie stan dozwolony

pod warunkiem, że przejściowe błędy nie wystąpią przez wystarczająco długi czas.

O wyborze aktywnego wierzchołka i reguły, która ma się w nim wykonać, decyduje abstrakcyjny algorytm zwany *dyspozytorem* (ang. *scheduler, daemon*). Projektując i analizując algorytm możemy patrzeć na dyspozytora jak na przeciwnika w grze, który stara się zmaksymalizować liczbę ruchów, wykonanych przez algorytm; oczywiście z drugiej strony — im mniej ruchów wykona algorytm, tym lepiej. Rozważa się dyspozytorów z różnymi ograniczeniami i na ich podstawie dokonuje kolejnych klasyfikacji systemów samostabilizujących.

Jeżeli dyspozytor w pojedynczym kroku może wybierać więcej, niż jeden aktywny wierzchołek do wykonania ruchu, mówimy, że jest *rozproszony* lub *asynchroniczny* [BGM89], w przeciwnym razie — *centralny* lub *sekwencyjny* [Dij74]. Projektowanie i analiza algorytmów z rozproszonym dyspozytorem jest co najmniej tak samo trudna, jak dla dyspozytora centralnego. Można to uzasadnić tym, że dyspozytor centralny jest podklasą dyspozytorów rozproszonych. Ponadto intuicyjnie możemy to uzasadnić tym, że w przypadku dyspozytora centralnego nie występuje przeplot w obliczaniu wartowników i wykonywaniu przypisań, w szczególności dla dwóch sąsiednich wierzchołków. Zachodzi tu pełna analogia do porównywania programowania sekwencyjnego z wielowątkowym lub równoległym.

Inną cechą, która dzieli dyspozytorów na rozłączne klasy, jest *sprawiedliwość* (ang. *fairness*) [Dol00, DTY15]. Dyspozytor jest *silnie sprawiedliwy* (ang. *strongly fair*), jeśli każdy wierzchołek, który jest aktywny nieskończenie często, zostanie wybrany aby wykonać ruch. *Słaba sprawiedliwość* (ang. *weakly fairness*) polega na tym, że aby zagwarantować sobie wybranie przez dyspozytora i wykonanie ruchu, wierzchołek musi być aktywny nieprzerwanie. Dyspozytor *niesprawiedliwy* (ang. *unfair, proper, adversarial*) musi wybrać dany aktywny wierzchołek jedynie wtedy, gdy w systemie nie ma już innego aktywnego wierzchołka — jednak dopóki w systemie są inne aktywne wierzchołki, dyspozytor może pomijać dany wierzchołek dowolnie długo.

W rozdziale 5 przedstawiamy algorytm działający na podklasie systemów samostabilizujących, którą nazwaliśmy systemami *quasi-samostabilizującymi* [BP13]. Różnicą w stosunku do podstawowych założeń jest objęcie części zmiennych przyporządkowanych do węzła ochroną przed przejściowymi błędami oraz zapewnienie, że system uruchamia się z określonymi wartościami tych zmiennych. To ostatnie założenie może być zapewnione w realizacjach sprzętowych poprzez zainicjowanie wspomnianych zmiennych pożądanymi wartościami na końcu działania programu uruchomieniowego węzła obliczeniowego. Dzięki zastosowaniu quasi-samostabilizacji osiągnęliśmy algorytm o mniejszej złożoności w porównaniu z jego zwykłym samostabilizującym

pierwowzorem. Wstępne rozważania teoretyczne na temat takich systemów można znaleźć w pracach Arory i Goudy [AG93] oraz u Schneidera [Sch93].

W literaturze rozważane są jeszcze inne podklasy systemów samostabilizujących z rozluźnionymi w różny sposób założeniami, co — z jednej strony — ułatwia (a czasem w ogóle umożliwia) zaprojektowanie algorytmu rozwiązującego dany problem — z drugiej strony — narzuca na system dodatkowe wymagania sprzętowe lub, jak wspomniano, rozluźnia wymagania stawiane wobec algorytmu. Między innymi z tych powodów, a także dla zwartości pracy, nie będziemy się zajmować systemami probabilistycznymi [IJ90], systemami pseudosamostabilizującymi [BGM93], k-stabilizacją [BGK98, GT02] ani systemami słabo samostabilizującymi [Gou01].

Oceniając złożoność obliczeniową algorytmu samostabilizującego będziemy brali pod uwagę liczbę ruchów, które musiałyby się wykonać, aby system osiągnął stan dozwolony (ustabilizował się). Tak, jak w przypadku algorytmów klasycznych, będzie nas interesować pesymistyczna liczba ruchów.

W literaturze miarą szybkości działania algorytmów samostabilizujących bywa również liczba *rund* [DIM97], które się wykonają zanim algorytm ustabilizuje system. Mając dany ciąg ruchów E wykonywanych przez algorytm na systemie, pierwszą rundą nazwiemy taki początkowy najkrótszy podciąg ruchów, w którym każdy węzeł, który był aktywny w początkowej konfiguracji albo wykonał ruch, albo przestał być aktywny w wyniku ruchów wykonanych przez sąsiadujące węzły. Druga runda jest definiowana analogicznie dla pozostałego ciągu ruchów itd. Dla ustalonego grafu będącego modelem systemu, w przypadku niesprawiedliwego dyspozytora liczba ruchów przypadających na jedną rundę może być potencjalnie nieskończona. Dlatego w naszych rozważaniach skupimy się na szacowaniu liczby ruchów, a nie rund.

Wymiana informacji pomiędzy węzłami obliczeniowymi może przebiegać za pomocą przekazywania wiadomości bezpośrednio między węzłami, wykorzystując kolejki komunikatów lub za pomocą współdzielonej pamięci. W obu przypadkach niskopoziomowa realizacja algorytmu samostabilizującego wymaga stałego, systematycznego przepływu informacji pomiędzy węzłami. Gdyby tego zabrakło i informacja o zmianie wartości zmiennych w każdym z węzłów była przesyłana tylko w momencie zmiany dokonanej przez algorytm, niemożliwe byłoby stwierdzenie zmiany spowodowanej przejściowym błędem.

Systemy, które zakładają, że przez cały czas działania topologia sieci nie zmieni się, nazywamy *statycznymi*. Natomiast tam, gdzie możliwe jest dodanie lub usunięcie wierzchołków i połączeń pomiędzy nimi, mówimy o systemach *dynamicznych*. Taka zmiana sieci musi jednak zachowywać założoną klasę topologii, np. jeśli dynamiczny algorytm zaprojektowany jest dla drzew, to jakakolwiek zmiana topologii musi zachować własności drzewa. Zmianę to-

pologii sieci w systemie dynamicznym bez utraty ogólności traktujemy jako jeden z rodzajów przejściowego błędu.

W niniejszej rozprawie we wszystkich prezentowanych przez nas algorytmach samostabilizujących przyjmujemy, że system jest cichy, jednolity i dynamiczny, natomiast dyspozytor jest niesprawiedliwy i sekwencyjny.

1.3 Zarys teorii grafów

Będziemy używać terminów i definicji zgodnych z klasycznym podręcznikiem Hararego [Har69].

Definicja 1. *Oznaczmy przez* $G = (V(G), E(G))$ graf*, gdzie* $V(G)$ *i* $E(G)$ *to skończone zbiory odpowiednio* wierzchołków *i* krawędzi*. Krawędź* $e \in E(G)$ *to dwuelementowy zbiór wierzchołków z* $V(G)$*.*

Z tak sformułowanej definicji grafu wynika, że krawędzie łączą różne wierzchołki (graf nie ma *pętli*) i w grafie nie ma dwóch takich samych (*wielokrotnych*) krawędzi — mówimy wtedy, że graf jest *prosty*. Zauważmy ponadto, że nie wyróżniamy żadnego z elementów (*końców*) krawędzi, więc graf jest *nieskierowany*.

Jeśli wierzchołki $u, v \in V(G)$ połączone są krawędzią $e = \{u, v\} \in E(G)$, mówimy, że u i v *sąsiadują* ze sobą lub są *sąsiadami*, natomiast wierzchołek u i krawędź e są wtedy *incydentne*, podobnie jak v i e. Liczbę krawędzi incydentnych z wierzchołkiem v nazywamy jego stopniem i oznaczamy $\deg(v)$. Stopniem Δ grafu nazywamy największy stopień jego wierzchołków.

Definicja 2. Podgrafem *grafu* H *nazywamy graf* $G \subset H$*, jeśli zachodzi* $V(G) \subset V(H)$ *oraz* $E(G) \subset E(H)$*.*

Mówimy, że graf jest *maksymalny* ze względu na pewną swoją właściwość, jeśli nie jest możliwe dołożenie do niego krawędzi (bez dodatkowych wierzchołków), tak aby nie utracił rzeczonej właściwości.

Graf $G \subset H$ *rozpina* H, jeśli $V(G) = V(H)$. Graf *indukowany* $H[S]$ w grafie H przez zbiór $S \subset V(H)$ to maksymalny podgraf grafu H, którego zbiorem wierzchołków jest S.

Definicja 3. *Grafy* G *oraz* H *są* izomorficzne*, jeśli istnieje bijekcja między ich wierzchołkami, która zachowuje relację sąsiedztwa.*

Graf o n wierzchołkach, z których każdy jest połączony krawędzią z pozostałymi, nazywamy grafem *pełnym* i oznaczamy K_n. Podgraf, który jest pełny nazywamy *kliką*. Graf K_1 nazywamy *trywialnym*.

Definicja 4. *Ciąg* $v_1, v_2, \ldots, v_k$ *wierzchołków, z których każde dwa kolejne sąsiadują ze sobą* ($\{v_i, v_{i+1}\} \in E(G)$, $i = 1, \ldots, k-1$)*, oraz żaden z nich (ewentualnie poza pierwszym i ostatnim, jeśli* $k > 3$*) się nie powtarza* ($v_i \neq v_j$, $i \neq j$, $\{i, j\} \neq \{1, k\}$)*, nazywamy* ścieżką[1]. *Jeśli pierwszy wierzchołek ścieżki jest również jej ostatnim* ($v_1 = v_k$)*, mówimy wtedy o* cyklu.

Graf o n wierzchołkach, którego wierzchołki stanowią ścieżkę a jedyne krawędzie łączą kolejne wierzchołki ścieżki, będziemy oznaczać jako P_n, zaś analogiczny cykl o n wierzchołkach jako C_n. W szczególności cykl C_3 będziemy nazywać *trójkątem*.

Jeśli każdą parę wierzchołków łączy ścieżka, mówimy, że graf jest *spójny*. Spójny graf, który nie zawiera cykli, nazywamy *drzewem*. Graf nazywamy *n-spójnym*, jeśli minimalna liczba wierzchołków, które należy usunąć, aby rozspójnić graf lub otrzymać graf trywialny, jest nie mniejsza, niż n. W przypadku, gdy wystarczy usunąć jeden wierzchołek, aby rozspójnić graf, nazywamy go *punktem artykulacji*.

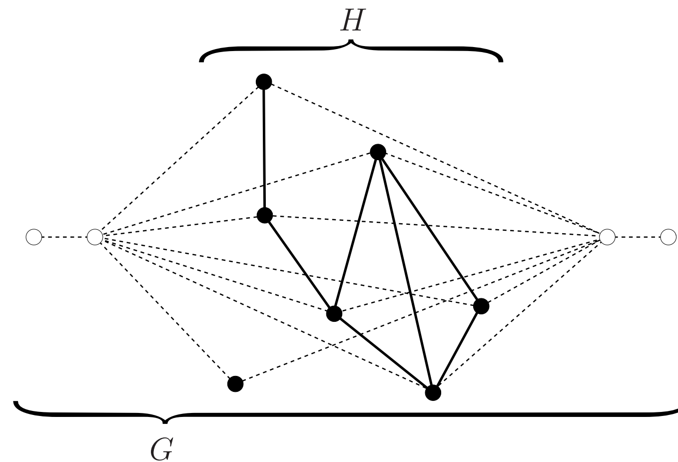
Definicja 5 ([HO71])**.** Liczbą artykulacyjną *wierzchołka* v *nazywamy liczbę par wierzchołków* $u, w \neq v$ *takich, że każda ścieżka z* u *do* w *zawiera* v*. Liczba artykulacyjna grafu to największa liczba artykulacyjna spośród jego wierzchołków.* Centrum artykulacyjnym *nazywamy taki zbiór wierzchołków, dla których liczba artykulacyjna jest równa liczbie artykulacyjnej grafu.*

Liczba artykulacyjna punktu artykulacji jest zawsze dodatnia. Jeśli graf jest dwuspójny, jego liczba artykulacyjna jest równa 0.

Odległością $d(u, v)$ między wierzchołkami u i v w G będziemy nazywać liczbę krawędzi wchodzących w skład najkrótszej ścieżki łączącej u z v. *Acentrycznością* $e_G(v)$ (wierzchołkową) wierzchołka v w grafie G jest odległość v do najdalszego wierzchołka w tymże grafie, tj. $e_G(v) = \max_{u \in V(G)} d_G(v, u)$. *Promieniem* $r(G)$ nazywamy minimalną acentryczność wśród wszystkich wierzchołków grafu G: $r(G) = \min_{v \in V(G)} e_G(v)$. *Centrum* $C(G)$ grafu G nazwiemy zbiór wierzchołków o minimalnej acentryczności, tj. $C(G) = \{v \in V(G) : e_G(v) = r(G)\}$.

Копылов i Тимофеев [KT77] podali twierdzenie (bez dowodu), według którego dla dowolnego grafu H istnieje graf G, którego centrum indukuje H. S. T. Hedetniemi (według [BMS81]) podał sposób na skonstruowanie grafu G dla danego H (rysunek 1.1), używając jedynie 4 dodatkowych wierzchołków, tj. $|V(G)| = |V(H)| + 4$.

[1] Polscy autorzy nie są tutaj zgodni i czasem stosują termin *droga*, podczas gdy dopuszczają, by w ścieżce dowolne wierzchołki mogły się powtarzać. My stosujemy konsekwentnie terminologię zgodną z książkami Hararego [Har69] i w tym przypadku również Diestela [Die05].



Rysunek 1.1: Konstrukcja grafu G o własności $C(G) = V(H)$.

Graf *planarny* to taki, który można przedstawić na płaszczyźnie w ten sposób, aby żadne krawędzie się nie przecinały. Przedstawienie grafu planarnego w powyższy sposób na płaszczyźnie nazywamy grafem *płaskim*. Grafy płaskie dzielą płaszczyznę na *regiony*, z których jeden jest nieograniczony i nazywamy go regionem *zewnętrznym*. Mówimy, że krawędź lub wierzchołek są *styczne* do regionu, jeśli należą do jego brzegu. Jeśli region ograniczony jest cyklem prostym, będziemy czasem utożsamiać tenże cykl z ograniczonym przez niego regionem.

Grafem *maksymalnym planarnym* nazywamy graf planarny, do którego nie można dodać krawędzi tak, by nie przestał być planarnym. W grafie zewnętrznie planarnym istnieje region, do którego wszystkie wierzchołki są styczne.

Problem znajdowania centrum grafu, szczególnie w kontekście systemów rozproszonych, ma duże znaczenie praktyczne. Jest również obiektem szerokich badań. W oczywisty sposób pozwala zminimalizować koszt komunikacji w sieciach z wyróżnionym jednym węzłem-centralą.

Jordan [Jor69] udowodnił, że centrum dowolnego drzewa składa się z jednego lub dwóch sąsiadujących ze sobą wierzchołków.

Definicja 6 ([Bol98])**.** *Graf* dualny G^* *grafu* G *o regionach* $F_1, F_2, \ldots, F_q$ *i krawędziach* $e_1, e_2, \ldots, e_m$ *ma wierzchołki* $v_1, v_2, \ldots, v_q$ *i krawędzie* $f_1, f_2, \ldots,$ f_m *takie, że* f_i *łączy* v_j *z* v_k *wtedy i tylko wtedy, gdy regiony* F_j *i* F_k *sąsiadują ze sobą poprzez krawędź* e_i*. Zauważmy, że* G *jest grafem dualnym grafu* G^**. Graf* słaby dualny *to graf dualny pozbawiony wierzchołka odpowiadającego zewnętrznemu (nieograniczonemu) obszarowi.*

Na koniec zdefiniujemy konstrukcję, której użyjemy w rozdziale 3.

Definicja 7. Iloczyn kartezjański $G_1 \times G_2$ *dwóch grafów* G_1 *i* G_2 *jest grafem, którego zbiór wierzchołków definiujemy jako iloczyn kartezjański zbiorów wierzchołków poszczególnych grafów:*

$$V(G_1 \times G_2) = V(G_1) \times V(G_2)$$

natomiast zbiór krawędzi definiujemy następująco:

$$\begin{gathered} E(G_1 \times G_2) = \{\{(u_1, u_2), (v_1, v_2)\} \mid \\ u_1, v_1 \in V(G_1) \wedge u_2, v_2 \in V(G_2) \wedge \\ ((u_1 = v_1 \wedge \{u_2, v_2\} \in E(G_2)) \vee \\ (u_2 = v_2 \wedge \{u_1, v_1\} \in E(G_1)))\}. \end{gathered}$$

Rozdział 2

Przegląd dotychczasowych wyników

W niniejszym rozdziale skupimy się na przeglądzie znanych w literaturze algorytmów samostabilizujących, które rozwiązują problemy wyznaczania drzewa rozpinającego oraz wyboru wiodącego wierzchołka zarówno w sieciach o dowolnej topologii, jak i ograniczonej do drzew. Wspomniane zagadnienia często są ze sobą ściśle związane. W szczególności, jeśli wyznaczone drzewo rozpinające ma być ukorzenione, jego korzeń jest naturalnym kandydatem do roli wiodącego wierzchołka.

2.1 Algorytmy dla dowolnych grafów

Problem wyznaczania drzewa rozpinającego w systemach samostabilizujących został rozwiązany przez Doleva *et al.* [DIM93], przy czym ich algorytm, stabilizujący system po $\mathcal{O}(Diam)$ rundach, wymaga, by w systemie jeden wierzchołek (uznawany w rezultacie za korzeń) wykonywał inny algorytm. Jest to zatem algorytm półjednolity.

Afek *et al.* [AKY90] pokazali algorytm jednolity, który stabilizuje system po $\mathcal{O}(n^2)$ rundach. Huang i Chen [HC92] oraz Sur i Srimani [SS92] również pokazali (bez określenia złożoności) algorytmy wyznaczające drzewo rozpinające, które wymagają, by każdy węzeł obliczeniowy znał rozmiar n sieci lub przynajmniej jego górne ograniczenie N.

Zagadnienie wyznaczania jednocześnie wiodącego wierzchołka i drzewa rozpinającego w statycznych sieciach samostabilizujących było podejmowane przez wielu autorów. Arora i Gouda [AG90] podali algorytm wymagający pamięci rzędu $\mathcal{O}(\log N)$ na każdy proces i stabilizujący sieć po $\mathcal{O}(N)$ rundach, gdzie N to górne ograniczenie liczby wierzchołków grafu, które musi

być z góry znane wszystkim wierzchołkom. Dolev i Herman [DH95] zaprezentowali niecichy algorytm stabilizujący w czasie $\mathcal{O}(Diam)$ rund, ale wymagający $\mathcal{O}(n \log n)$ pamięci na każdy proces oraz sprawiedliwego dyspozytora. Algorytmy Awerbucha *et al.* [AKM+93] oraz Burman i Kuttena [BK07] potrzebują $\mathcal{O}(Diam)$ rund do ustabilizowania systemu i $\mathcal{O}(\log D \log n)$ pamięci na proces, gdzie D jest górnym ograniczeniem średnicy sieci. Ponadto oba algorytmy wymagają, by każdy węzeł obliczeniowy z góry znał ograniczenie D. Algorytm podany przez Afeka i Bremlera [AB97] stabilizuje sieć po $\mathcal{O}(n)$ rundach i wymaga $\mathcal{O}(\log n)$ pamięci w każdym procesie, ale niestety wymaga również uczciwego dyspozytora. Datta, Larmore i Vemula [DLV08] zaprezentowali jednolity, działający pod kontrolą niesprawiedliwego dyspozytora, samostabilizujący algorytm, który wymaga $\mathcal{O}(\log n)$ pamięci na każdy proces. Algorytm ten stabilizuje sieć po $\mathcal{O}(n)$ rundach, a po dodatkowych $\mathcal{O}(Diam)$ wycisza się.

Dla sieci dynamicznych Dolev *et al.* [DIM97] zaprezentowali randomizowane rozwiązanie problemu stabilizujące system po $\mathcal{O}(\Delta Diam \log n)$ rundach. Derhab i Badache [DB08] oraz Ingram *et al.* [ISWW09] pokazali algorytmy wyznaczające wiodący wierzchołek, które jednak wymagają zgodnie działających zegarów w każdym węźle obliczeniowym lub istnienia globalnego zegara.

Datta, Larmore i Piniganti [DLP10] zaproponowali cichy, asynchroniczny algorytm samostabilizujący, pozbawiony powyżej opisanej wady, wyznaczający wiodący wierzchołek w dowolnym grafie po $\mathcal{O}(n)$ rundach. Wadą rozwiązania jest istnienie w każdym węźle zmiennej, której wartość może rosnąć bez ograniczeń, więc teoretyczne zapotrzebowanie na pamięć w każdym węźle jest nieograniczone.

Rozwiązanie Kravchika i Kuttena [KK13], wymagające synchronicznego dyspozytora, stabilizuje system w $\mathcal{O}(Diam)$ rundach. Każdy węzeł obliczeniowy wykorzystuje $\mathcal{O}(\log n)$ bitów, tak jak rozwiązanie zaproponowane przez Dattę, Larmore'a i Vemulę [DLV11a, DLV11b]. Przy czym rozwiązanie [DLV11a] działa na systemach quasi-samostabilizujących.

Kosowski i Kuszner [KK05] pokazali półjednolity algorytm stabilizujący system po $\mathcal{O}(nDiam)$ ruchach. Altisen *et al.* [ACD+15] zaprezentowali cichy algorytm działający pod kontrolą rozproszonego i niesprawiedliwego dyspozytora, który wykorzystuje $\mathcal{O}(\log n)$ bitów na proces i zatrzymuje sieć po $\mathcal{O}(n)$ rundach a po $\mathcal{O}(n^3)$ ruchach.

2.2 Algorytmy dla drzew

Blair i Manne [BM03] skonstruowali samostabilizujący algorytm znajdowania centroidu w drzewach, o złożoności $\mathcal{O}(n^2)$ ruchów. Dla zupełności treści pracy przedstawiamy pełną treść tego algorytmu wraz z twierdzeniami opisującymi własności algorytmu (Algorytm 1), ponieważ będziemy się do niego odwoływać w dalszych częściach niniejszej pracy. Na potrzeby algorytmu dla wzajemnie sąsiednich węzłów i oraz j w drzewie T zdefiniowana jest zmienna $size_i(j)$, której wartość po ustabilizowaniu systemu oznaczać będzie liczbę węzłów w spójnej składowej grafu $T-j$, która zawiera i, gdzie $i, j \in V(T)$. Dodatkowo zdefiniowano predykat

$$sizeCorrect_i \Leftrightarrow \left(\forall_{j \in N(i)} \; size_i(j) = 1 + \sum_{k \in N(i) \setminus \{j\}} size_k(i) \right).$$

Algorytm 1: Algorytm Blaira i Mannego [BM03]

R1:
  if $\exists_{j \in N(i)} \; size_i(j) \neq 1 + \sum_{k \in N(i)-\{j\}} size_k(i)$ **then**
    $size_i(j) := 1 + \sum_{k \in N(i)-\{j\}} size_k(i)$
R2:
  if $sizeCorrect_i \wedge (\forall_{j \in N(i)} \; size_j(i) < n_i/2) \wedge p_i \neq i$ **then**
    $p_i := i$
R3:
  if $sizeCorrect_i \wedge (\exists_{j \in N(i)} \; size_j(i) > n_i/2) \wedge p_i \neq j$ **then**
    $p_i := j$
R4:
  if $sizeCorrect_i \wedge (\exists_{j \in N(i)} \; size_j(i) = n_i/2) \wedge ID_i > ID_j \wedge p_i \neq i$ **then**
    $p_i := i$
R5:
  if $sizeCorrect_i \wedge (\exists_{j \in N(i)} \; size_j(i) = n_i/2) \wedge ID_i < ID_j \wedge p_i \neq j$ **then**
    $p_i := j$

Twierdzenie 3.10 z pracy [BM03] można sformułować w następujący sposób.

Twierdzenie 1 ([BM03])**.** *Dla każdej sieci o topologii drzewa algorytm R1–R5 stabilizuje system w co najwyżej $2n^2 - n$ ruchach, z wartościami p_i dla wszystkich węzłów i definiującymi ukorzenienie drzewa w jednym z jego centroidów.*

Harary i Ostrand [HO71] wprowadzili pojęcie *centrum artykulacyjnego* (ang. *cutting center*) w drzewach, wykorzystując *liczbę artykulacyjną* (ang. *cutting number*) (definicja 5, s. 11). Ponadto Harary i Slater [HS86] przedstawili liniowy algorytm znajdujący centrum artykulacyjne w drzewie i podali wzór służący obliczaniu liczby artykulacyjnej w danym wierzchołku:

$$c(v) = \sum_{1 \leqslant i < j \leqslant k+1} |C_i| \cdot |C_j| = \frac{1}{2} \left(\sum_{1 \leqslant i \leqslant k+1} |C_i| \cdot (p - 1 - |C_i|) \right), \qquad (2.1)$$

gdzie C_x oznacza jedno z drzew powstałych po usunięciu wierzchołka v, zaś p to liczba wierzchołków w całym drzewie. Pierwsza część wzoru jest zapisem definicji liczby artykulacyjnej, zaś druga wynika z faktu, że $|C_i|$ wierzchołków ze spójnej składowej łączyło się za pośrednictwem v z $(p-1-|C_i|)$ wierzchołkami z reszty drzewa. Jednak w ten sposób każde połączenie byłoby liczone dwukrotnie, stąd współczynnik $\frac{1}{2}$.

Chaudhuri i Thompson [CT04] przedstawiają samostabilizujący algorytm znajdujący centrum artykulacyjne w drzewach ukorzenionych. Autorzy [CT04] popełnili kilka znaczących błędów, które poniżej przedstawiamy.

- Założono, że każdy węzeł z góry zna rozmiar systemu (liczbę wszystkich węzłów). Systemy samostabilizujące mogą ulegać przejściowym błędom, w tym zmianie wartości zmiennych na niewłaściwe. Rozmiar systemu n również miałby być taką zmienną w każdym węźle, której wartość mogłaby być zaburzona i powodowałoby to błędne działanie algorytmu. Aby przywrócić poprawną wartość zmiennej n w każdym węźle, należałoby dodać fragment odpowiedzialny za wykrywanie takiej usterki i jej usuwanie, o czym autorzy nie wspominają.

- Podobnie zostało założone, że drzewo jest ukorzenione i każdy węzeł zna zbiór swoich dzieci oraz zna swojego ojca (poza korzeniem). Analogicznie, jak w punkcie powyżej, zmienne określające te własności mogą zostać zaburzone w trakcie działania systemu, co wymuszałoby odtworzenie ich prawidłowego stanu. Autorzy również nie odnoszą się do tej kwestii.

  Połowicznym rozwiązaniem mogłoby być zapisanie tych wartości w każdym węźle na stałe tak aby nie mogły ulec zmianie w wyniku przejściowych błędów. Jednak to rozwiązanie stawia pod znakiem zapytania sens uruchamiania całego samostabilizującego algorytmu (zamiast klasycznego o mniejszej złożoności) — skoro znana jest z góry topologia drzewa i wiadomo, że nie zmieni się w czasie działania systemu oraz że wiedza ta jest wykorzystana w celu zapisania każdemu węzłowi jego dzieci

i rodzica, to równie dobrze można w ten sam sposób wyznaczyć, które węzły stanowią centrum artykulacyjne drzewa a następnie zapisać tę binarną informację w każdym węźle.

- Algorytm podzielono na 3 fazy, przy założeniu, że wszystkie wykonują się sekwencyjnie [CT04, s. 188], tzn. faza druga nie zacznie się wykonywać, zanim nie zakończy się działanie fazy pierwszej itd. Nie wskazano, jaki mechanizm miałby być za to odpowiedzialny — przeciwnie — w opisie modelu obliczeń określono, że wybór, który z uprzywilejowanych węzłów wykona ruch, jest niedeterministyczny [CT04, s. 185], co przeczy sekwencyjności wykonywania się faz.

- W dowodzie złożoności algorytmu ($\mathcal{O}(n^2)$ ruchów) [CT04, s. 188], wykorzystano założenie z poprzedniego punktu, tym samym wprowadzając do niego błąd. Łatwo dać kontrprzykład pokazujący, że złożoność algorytmu jest co najmniej $\mathcal{O}(n^3)$ ruchów: weźmy drzewo w postaci ścieżki prostej o n węzłach. Załóżmy ponadto, że wszystkie wartości $D(i)$ w takim systemie są nieprawidłowe, więc wszystkie węzły są uprzywilejowane (faza I). Pesymistyczny przypadek polega na tym, że pierwszy ruch wykona korzeń, obliczając swoją wartość D na podstawie błędnej wartości D w swoim jedynym dziecku. Następnie dziecko korzenia wyznaczy swoją wartość D na podstawie wartości ze swojego dziecka i znowu korzeń wykona ruch. Każda kolejna fala obliczeń będzie w ten sposób zaczynała się o jeden węzeł dalej od korzenia i podążała w jego stronę, więc liczba ruchów będzie równa $1 + 2 + 3 + \ldots + n$, co w sumie da $\frac{n(n+1)}{2}$ ruchów.

  Zauważmy teraz, że fazy mogą się przeplatać, więc po każdym obliczeniu wartości D w którymś z węzłów, może się aktywować (pośrednio przez fazę II) reguła 3 z fazy III. Wykonanie tej reguły w węźle powoduje jej aktywację w węźle-rodzicu i tak aż do korzenia. Zatem po każdym z $\mathcal{O}(n^2)$ ruchów z fazy I może się wykonać $\mathcal{O}(n)$ ruchów fazy III. Daje to ostatecznie $\mathcal{O}(n^3)$ ruchów.

W rozdziale 4.2 prezentujemy samostabilizujący algorytm o złożoności $\mathcal{O}(n^3)$ ruchów, oparty na algorytmie Blaira i Mannego [BM03] oraz Chaudhuriego i Thompsona [CT04], który jest pozbawiony wymienionych wyżej defektów, wyznaczający centrum artykulacyjne w drzewie nieukorzenionym.

Wybór wiodącego węzła (ang. *leader election*) jest zagadnieniem teorii grafów, które ma bardzo dużo zastosowań [APR15, FJ01, FZAM08, Pat14, XS06]. Antonoiu i Srimani [AS96] udowodnili, że nie istnieje algorytm wyboru liścia jako lidera w dowolnym drzewie, w którym węzły nie mają unikalnych identyfikatorów.

Rozdział 3

Centra grafów maksymalnych zewnętrznie planarnych i ich iloczynów kartezjańskich

3.1 Wprowadzenie

Grafy zewnętrznie planarne były i nadal są intensywnie badane. Sysło [Sys79] podał wiele własności spełnianych przez grafy zewnętrznie planarne. Proskurowski [Pro80] podał wszystkie siedem postaci centrów grafów maksymalnych zewnętrznie planarnych. Farley i Proskurowski [FP80] skonstruowali klasyczny (sekwencyjny) algorytm wyznaczający centrum grafów zewnętrznie planarnych — prezentujemy go w rozdziale 3.2.

Użyjemy definicji przytoczonej przez Farleya i Proskurowskiego w [FP80].

Definicja 8. *Graf* maksymalny zewnętrznie planarny *(w skrócie MOP) otrzymujemy poprzez triangulację wielokąta (przykład na rysunku 3.1).*

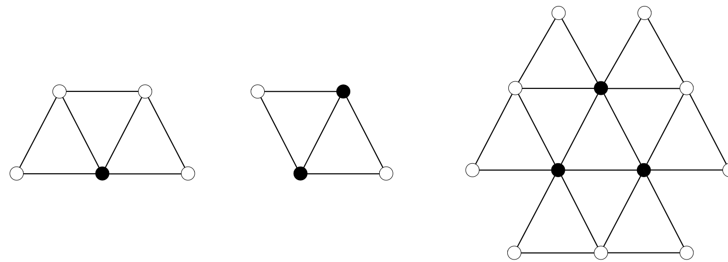
Często w literaturze obiekty definiowane przy pomocy definicji 8 bywają nazywane grafami zewnętrznie planarnymi *dwuspójnymi*, podczas gdy w zwykłych grafach maksymalnych zewnętrznie planarnych może występować punkt artykulacji. W dalszej części będziemy zajmować się tylko obiektami zgodnymi z definicją 8 czyli grafami dwuspójnymi. Najmniejszym takim grafem jest trójkąt K_3. Wszystkie wierzchołki mogą być styczne do jednego regionu; najczęściej uznajemy, że jest to region zewnętrzny i przyjmiemy tak w dalszych rozważaniach.

Definicja 9. *W grafie maksymalnym zewnętrznie planarnym G każda krawędź $p = \{i, j\} \in E(G)$ dzieli zbiór wszystkich wierzchołków z wyłączeniem*

wierzchołków i oraz j ($V \setminus \{i, j\}$) na dwa rozłączne zbiory wierzchołków, zwane stronami *krawędzi, indukujących spójne podgrafy.*

Jedna ze stron może być pusta. Ma to miejsce w przypadku, gdy dzieląca krawędź jest styczna z regionem zewnętrznym. Będziemy nazywać takie krawędzie *zewnętrznymi.* Zauważmy, że wszystkie krawędzie zewnętrzne tworzą jedyny cykl Hamiltona [Har69] w takim grafie. Pozostałe krawędzie, czyli te, które rozdzielają regiony wewnętrzne, będziemy nazywać *wewnętrznymi.*

W grafie maksymalnym zewnętrznie planarnym każde dwa sąsiednie wierzchołki i, j mają co najwyżej dwóch wspólnych sąsiadów k, l, z których każdy należy do różnych stron S_k (dla k) oraz S_l (dla l) podzielonych przez krawędź $\{i, j\}$. Dlatego, aby jednoznacznie wyróżnić stronę krawędzi $\{i, j\}$, wystarczy wskazać ten z wierzchołków k lub l (jako *wyróżnik strony*), który należy do odpowiedniej strony. Przyjmujemy, że takim wyróżnikiem dla pustej strony jest symbol $\emptyset$.



Rysunek 3.1: Przykłady grafów maksymalnych zewnętrznie planarnych z centrami: z jednym, dwoma i trzema wierzchołkami.

3.2 Klasyczny algorytm wyszukiwania centrum

Farley i Proskurowski [FP80] wprowadzili pojęcie acentryczności krawędziowej.

Definicja 10. *Niech będą dane dwa wierzchołki i, j połączone krawędzią oraz*

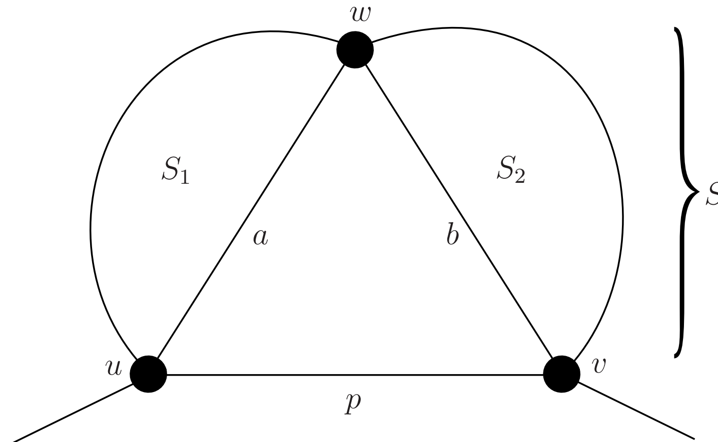
- *jeden z ich dwóch wspólnych sąsiadów — w przypadku, gdy krawędź $\{i, j\}$ jest wewnętrzna lub*
- *jedyny ich wspólny sąsiad albo $\emptyset$ — w przypadku, gdy krawędź $\{i, j\}$ jest zewnętrzna*

oznaczony jako k. Acentryczność krawędziową *$e(i, j, k)$ dla danego grafu G definiujemy jako funkcję $e_G: V(G)^2 \times (V(G) \cup \{\emptyset\}) \rightarrow \mathbb{Z}$ określoną następująco:*

- *bezwzględna wartość $e(i, j, k)$ jest równa acentryczności wierzchołka i w podgrafie indukowanym przez zbiór wierzchołków $S_k \cup \{i, j\}$,*
- *$e(i, j, k)$ jest ujemna wtedy i tylko wtedy, gdy odległość od j do wszystkich wierzchołków ze zbioru $S_k \cup \{i, j\}$ odległych od i o $d = |e(i, j, k)|$ jest równa $d-1$.*

Zwróćmy uwagę, że pierwszy punkt powyższej definicji określa moduł acentryczności krawędziowej, natomiast drugi punkt określa jej znak. Ponadto trzeci argument funkcji e_G jest równy $\emptyset$ w przypadku, kiedy strona jest pusta.

Przedstawimy poniżej oryginalny sekwencyjny algorytm Farleya i Proskurowskiego [FP80], obliczający acentryczności wierzchołków; wraz z kluczowymi lematami i ich ilustracją. Algorytm oblicza acentryczność krawędziową rekurencyjnie. Proces zaczyna się od acentryczności krawędzi zewnętrznych, dla których $e(\cdot, \cdot, \emptyset) = -1$.



Rysunek 3.2: [FP80] Ilustracja lematów 1, 2 i 3.

Lemat 1 ([FP80]). *Niech będzie dana krawędź $p = \{u, v\}$, należąca do maksymalnego grafu zewnętrznie płaskiego G, oraz jej niepusta strona S. Oznaczmy dodatkowo $e_1 = e(v, w, S_2)$ oraz $e_2 = e(w, v, S_2)$. Ponadto niech r oznacza acentryczność krawędziową wierzchołka u w podgrafie indukowanym $G[S_2 \cup \{u, v, w\}]$. Wtedy*

$$r = \begin{cases} -(1 + e_2) & \text{dla } e_2 > 0, \\ |e_2| & \text{w przeciwnym razie.} \end{cases}$$

Lemat 2 ([FP80]). *Niech będzie dana krawędź* $p = \{u, v\}$ *oraz jej niepusta strona* S *w grafie maksymalnym zewnętrznie płaskim* G*. Oznaczmy* $e_3 = e(w, u, S_1)$ *oraz* $d_1 = e(v, u, S)$*. Dodatkowo niech* r *oznacza acentryczność krawędziową wierzchołka* u *w podgrafie indukowanym* $G[S_2 \cup \{u, v, w\}]$ *jak w lemacie 1. Wtedy*

$$d_1 = \begin{cases} |e_3| & \text{dla } |e_3| \geqslant |r|, \\ r & \text{w przeciwnym razie.} \end{cases}$$

Lemat 3 ([FP80]). *Niech będzie dana krawędź* $p = \{u, v\}$ *oraz jej niepusta strona* S *w grafie maksymalnym zewnętrznie płaskim* G*. Dodatkowo niech*

$$q = \begin{cases} -(1 + e(w, u, S_1)) & \text{dla } e(w, u, S_1) > 0, \\ |e(w, u, S_1)| & \text{w przeciwnym razie.} \end{cases}$$

Wtedy

$$e(u, v, S) = \begin{cases} |e(w, v, S_2)| & \text{dla } e(w, v, S_2)| \geqslant |q|, \\ q & \text{w przeciwnym razie.} \end{cases}$$

Farley i Proskurowski udowodnili powyższe lematy i zastosowali je w swoim sekwencyjnym algorytmie, który prezentujemy poniżej.

Algorytm 2: [FP80]

Dany jest maksymalny graf zewnętrznie planarny M. Acentryczność krawędziową każdej krawędzi obliczamy w następujący sposób:

1. Dla wszystkich krawędzi $\{u, v\}$ na cyklu Hamiltona w M, przypisz wartość -1 do $e(u, v, \emptyset)$ oraz do $e(v, u, \emptyset)$

2. W każdym trójkącie (u, v, w), w którym wartości $e(w, u, S_1)$, $e(u, w, S_1)$, $e(w, v, S_2)$, oraz $e(v, w, S_2)$ są obliczone, przypisz wartości $e(v, u, S)$ i $e(u, v, S)$ zgodnie z zasadami opisanymi przez lematy 1, 2 oraz 3.

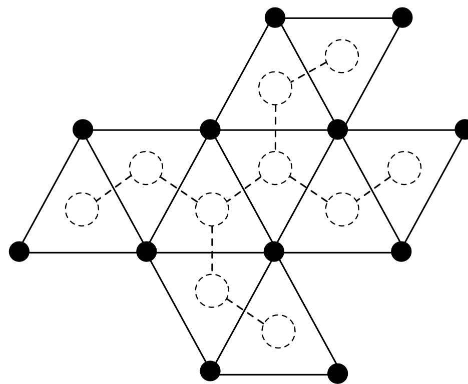
3.3 Samostabilizujący algorytm wyszukiwania centrum

Przedstawimy teraz samostabilizujący algorytm wyznaczający centrum grafu maksymalnego zewnętrznie planarnego, który jest oparty na sekwencyjnym

algorytmie Farleya i Proskurowskiego [FP80]. Poniższy algorytm został opublikowany w [PB14].

Będziemy używać grafu słabego dualnego (definicja 6, s. 12) do danego grafu maksymalnego zewnętrznie planarnego [FGH74]. Taki graf słaby dualny jest zawsze drzewem [Sys79] — przykład na rysunku 3.3. Od tej pory pisząc o grafie dualnym będziemy mieli zawsze na myśli graf słaby dualny.

Informacje na temat wierzchołka $v_i^* \in G^*$ grafu dualnego będą przechowywane w trzech najbliższych wierzchołkach podgrafu $K_3 \subset G$ (otaczającego wierzchołek v_i^*) pierwotnego grafu maksymalnego zewnętrznie planarnego G.



Rysunek 3.3: Przykład grafu maksymalnego zewnętrznie planarnego oraz jego grafu (słabego) dualnego.

Poniżej przedstawiamy notację, której będziemy używać w dalszej części niniejszego rozdziału. Następnie podamy reguły algorytmu oraz intuicyjne wyjaśnienie zasady jego działania. Na koniec udowodnimy poprawność naszego algorytmu.

$N(i)$ — zbiór sąsiadów wierzchołka i, tj. $N(i) = \{j : \{i, j\} \in E(G)\}$.

$n(i)$ — zmienna przechowująca zbiór $N(i)$. Zauważmy, że wartość $n(i)$ może nie być poprawna w początkowej fazie działania algorytmu lub po wystąpieniu przejściowego błędu. Natomiast $N(i)$ jest zbiorem, którego wartość jest określana na bieżąco na podstawie połączeń wierzchołka i z innymi wierzchołkami i nie może być wyznaczony przez wierzchołki inne niż i. Dlatego stosujemy zmienną $n(i)$, która umożliwia poinformowanie sąsiadów wierzchołka i, jakich on ma sąsiadów.

$c(i, j)$ — zmienna znajdująca się w wierzchołku i, której wartością jest zbiór wspólnych sąsiadów wierzchołków i oraz j.

$e(i, j, k)$ — zmienna znajdująca się w wierzchołku i, której wartością jest acentryczność krawędziowa wierzchołka i względem krawędzi $\{i, j\}$, po stronie zawierającej wspólnego sąsiada k wierzchołków i i j; $k = \emptyset$ dla pustej strony (zewnętrznego regionu).

$opp(i, j, k)$ — zmienna znajdująca się w wierzchołku i, przechowująca identyfikator strony przeciwnej do k względem krawędzi $\{i, j\}$.

$v(i)$ — zmienna znajdująca się w wierzchołku i, przechowująca acentryczność wierzchołka i. Nie jest to acentryczność krawędziowa; zauważmy, że w stanie dozwolonym jest ona równa $v(i) = \max_k |e(i, j, k)|$ dla dowolnego sąsiada $j \in N(i)$.

$m(i, j, k)$ — zmienna znajdująca się w wierzchołku i, przyporządkowana wierzchołkowi drzewa dualnego leżącemu wewnątrz regionu $\{i, j, k\}$. Jej wartością jest para, z której pierwszy element to promień grafu maksymalnego zewnętrznie planarnego; zaś drugim elementem pary jest wskazanie strony (regionu), z której pochodzi informacja o promieniu grafu. Drugi element tym samym wskazuje również, w którą stronę należy pójść, aby przybliżyć się do centrum grafu. Jeśli co najmniej jeden z wierzchołków i, j lub k wchodzi w skład centrum grafu, wartość drugiego elementu pary to $\emptyset$.

Przedstawiliśmy notację, zatem możemy teraz zaprezentować reguły naszego algorytmu (algorytm 3). W dalszej części przedstawiamy znaczenie i intuicję związaną z poszczególnymi częściami algorytmu.

W regule numer *1* przypisywana jest poprawna wartość zmiennej $n(i)$, aby każdy z sąsiadów wierzchołka i miał informację o zbiorze sąsiadów $N(i)$. Dzięki temu dwa sąsiednie wierzchołki i oraz j mogą określić zbiór wspólnych sąsiadów i przechować w zmiennych odpowiednio: $c(i, j)$ oraz $c(j, i)$, co jest wykonywane w regule *2*.

Reguły *3a*, *3b* i *4* — będące adaptacją sekwencyjnego algorytmu Farleya i Proskurowskiego — obliczają acentryczności krawędziowe i wierzchołkowe.

Reguła *5* zapewnia rozpropagowanie informacji o minimalnej acentryczności (promieniu) po całym grafie przy użyciu drzewa dualnego. W regule tej używamy pomocniczej funkcji $MinEcc$ (Funkcja 4, s. 27), która oblicza wartość $m(i, j, k)$. Używamy w niej dwóch funkcji rzutujących parę na poszczególne jej elementy: *fst* i *snd* wybierają odpowiednio pierwszy i drugi element pary.

Funkcja $MinEcc$ oblicza minimalną acentryczność w grafie i stronę, w którą należy się przemieścić, aby dotrzeć do wierzchołka o tejże najmniejszej

Algorytm 3: Wyznaczanie centrum MOP — pierwsza część

1:
if $n(i) \neq N(i)$ **then**
$n(i) := N(i)$

2:
if $\exists_{k \in n(i)} : \ c(i,k) \neq n(i) \cap n(k)$ **then**
$c(i,k) := n(i) \cap n(k)$

3a:
if $\exists_{j \in n(i)} : \ (|c(i,j)| = 1 \wedge (e(i,j,\emptyset) \neq -1 \vee opp(i,j,\emptyset) \neq k \vee opp(i,j,k) \neq \emptyset))$ **then**
$e(i,j,\emptyset) := -1$
$opp(i,j,\emptyset) := k$
$opp(i,j,k) := \emptyset$
where
$\{k\} = c(i,j)$

3b:
if $\exists_{j \in n(i)} : \ (|c(i,j)| = 1 \wedge (e(i,j,k) \neq d \vee v(i) \neq \max(|e(i,j,k)|, 1))$ **then**
$e(i,j,k) := d$
$v(i) := \max(|e(i,j,k)|, 1)$
where

$$q = \begin{cases} -(1 + e(j,k,opp(j,k,i))) & \text{dla } e(j,k,opp(j,k,i)) > 0, \\ e(j,k,opp(j,k,i)) & \text{w pozostałych przypadkach} \end{cases}$$

$$d = \begin{cases} |e(i,k,opp(i,k,j))| & \text{if } |e(i,k,opp(i,k,j))| \geqslant |q|, \\ q & \text{w pozostałych przypadkach} \end{cases}$$

$\{k\} = c(i,j)$

acentryczności. Funkcja obliczana jest dla wierzchołków i, j, k wyznaczających trójkąt K_3, wewnątrz którego mieści się wierzchołek grafu dualnego. Ilustracja działania funkcji $MinEcc$ znajduje się na rysunku 3.4 (s. 28).

Pierwszym krokiem jest przyjęcie, jako potencjalnie najmniejszej, wartości z wierzchołka i, w którym funkcja jest obliczana. W tym przypadku jako stronę przyjmujemy $\emptyset$, w celu oznaczenia faktu, że minimalna wartość została znaleziona w lokalnym wierzchołku drzewa dualnego. W drugim i trzecim kroku porównywane są wartości zmiennych $m(k,i,j)$ i $m(j,i,k)$ zapisane w wierzchołkach odpowiednio: k oraz j. Jeśli wartości w wierzchołkach k lub j pochodzą z regionów stycznych poprzez krawędzie $\{i,k\}$ lub $\{i,j\}$, to nie są zaufane. Gdybyśmy ich użyli, nieprawidłowe wartości mogłyby oscy-

Algorytm 3: Wyznaczanie centrum MOP — druga część

4:

if $\exists_{j \in n(i)} : (|c(i,j)| = 2 \wedge \exists_{k \in c(i,j)} : (e(i,j,k) \neq d \vee opp(i,j,k) \neq l \vee opp(i,j,l) \neq k \vee v(i) \neq \max(|e(i,j,k)|, |e(i,j,l)|)))$
then

$e(i,j,k) := d$

$opp(i,j,k) := l$

$opp(i,j,l) := k$

$v(i) := \max(|e(i,j,k)|, |e(i,j,l)|)$

where

$$q = \begin{cases} -(1 + e(j,k,opp(j,k,i))) & \text{dla } e(j,k,opp(j,k,i)) > 0, \\ e(j,k,opp(j,k,i)) & \text{w pozostałych przypadkach} \end{cases}$$

$$d = \begin{cases} |e(i,k,opp(i,k,j))| & \text{dla } |e(i,k,opp(i,k,j))| \geqslant |q|, \\ q & \text{w pozostałych przypadkach} \end{cases}$$

$\{k, l\} = c(i,j)$

5:

if $\exists_{j,k \in n(i)} : (k \in c(i,j) \wedge m(i,j,k) \neq MinEcc(i,j,k))$ **then**

$m(i,j,k) := MinEcc(i,j,k)$

lować po krawędzi drzewa dualnego przez dowolnie długi czas. W ostatnich dwóch krokach sprawdzane są (i ewentualnie użyte) wartości z regionów sąsiednich, z którymi graniczy wierzchołek i.

Lemat 4. *Algorytm składający się z reguł 1–4 zatrzymuje się po* $\mathcal{O}(n^2)$ *ruchach.*

Dowód. Stabilizacja reguły *1* jest oczywista i zajmuje $\mathcal{O}(n)$ ruchów, ponieważ jej wartownik nie zależy od zmiennych w wierzchołkach sąsiednich.

Reguła *2* zależy od wartości obliczonych przez regułę *1*, dlatego pojedynczy wierzchołek wykona co najwyżej stałą liczbę ruchów i sumaryczna złożoność w tym przypadku również jest $\mathcal{O}(n)$.

Podobnie reguła *3a* wykona się $\mathcal{O}(n)$ razy, ponieważ jej wartownik zależy jedynie od zmiennych lokalnych oraz tych obliczonych w sąsiadach przez dwie wcześniejsze reguły — reguła ta oblicza acentryczności dla przypadku bazowego (jak w rekurencyjnym algorytmie [FP80]): zewnętrznej strony grafu zewnętrznie planarnego.

Gdy zmienna $c(i,j)$ zostanie poprawnie obliczona w węźle i, jej wartość nigdy się już nie zmieni. Od tego momentu poprawnie obliczone, na podstawie $c(i,j)$, wartości zmiennych $e(i,j,\emptyset)$, $opp(i,j,\emptyset)$ i $opp(i,j,k)$ również nie mogą ulec zmianie w wyniku działania algorytmu.

Funkcja 4: $MinEcc(i, j, k)$

$v := v(i)$
$dir := \emptyset$
if $fst(m(k, i, j)) < v \wedge snd(m(k, i, j)) \in \{opp(k, j, i), \emptyset\}$ **then**
 $(v, dir) := m(k, i, j)$
if $fst(m(j, i, k)) < v \wedge snd(m(j, i, k)) \in \{opp(j, k, i), \emptyset\}$ **then**
 $(v, dir) := m(j, i, k)$
if $fst(m(i, j, opp(i, j, k))) < v \wedge snd(m(i, j, opp(i, j, k))) \neq k$ **then**
 $v := fst(m(i, j, opp(i, j, k)))$
 $dir := opp(i, j, k)$
if $fst(m(i, k, opp(i, k, j)) < v \wedge snd(m(i, k, opp(i, k, j))) \neq j$ **then**
 $v := fst(m(i, k, opp(i, k, j))$
 $dir := opp(i, k, j)$
return (v, dir)

W tym momencie we wszystkich wierzchołkach reguła *3a* przestała być aktywna. Reguła *4* dotyczy tylko grafów większych niż K_3, a w szczególności krawędzi oddzielających regiony wewnętrzne grafu. Weźmy dowolną taką krawędź, która dodatkowo ma po (przynajmniej) jednej ze stron $S = \{k\}$ tylko jeden wierzchołek k. Na tejże krawędzi reguła *4* obliczy acentryczność krawędziową dla obu końców krawędzi $\{i, j\}$ i strony przeciwnej do S na podstawie wartości acentryczności obliczonych przez regułę *3a* w wierzchołkach i oraz j — ilustracja na rysunku 3.5. W tym przypadku, gdy $|S| = 1$, acentryczności te są równe -1.

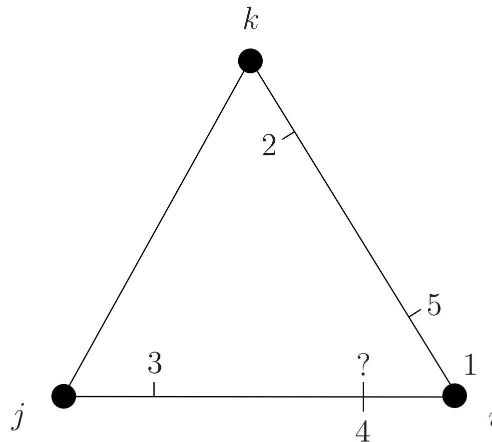
Wykonanie reguły *4* w przykładzie jak powyżej stanowi początek ostatniej „fali” wykonań reguły *4*, która wyznaczy poprawne wartości acentryczności krawędziowych. Takich fal może być $\mathcal{O}(n)$, a każda z nich składa się z $\mathcal{O}(n)$ ruchów. W rezultacie daje to $\mathcal{O}(n^2)$ ruchów wykonanych według reguły *4*.

Na koniec zajmiemy się zliczeniem ruchów wykonanych według reguły *3b*. Uruchomi się ona za każdym razem, gdy fala obliczeń reguły *4* dotrze do krawędzi zewnętrznej. Reguła *3b* wyznacza acentryczność na takiej krawędzi po stronie, z której nadeszła fala. Mnożąc liczbę fal przez liczbę takich krawędzi, otrzymujemy $\mathcal{O}(n^2)$.

Ostatecznie wszystkie etapy sumują się do $\mathcal{O}(n^2)$ ruchów dla całego algorytmu 1–4. □

Lemat 5. *Po zatrzymaniu się reguł 1–4 algorytmu 3 faza 5 zatrzyma się po $\mathcal{O}(n^2)$ ruchach.*

Dowód. Najbardziej pesymistycznym przypadkiem jest sytuacja, gdzie każdy



Rysunek 3.4: Ilustracja obliczenia funkcji $MinEcc(i, j, k)$. Kolejność obliczeń jest następująca:

1. $(v(i), \emptyset)$ (linie 1–2 w funkcji $MinEcc$),
2. $m(k, i, j)$ (linie 3–5),
3. $m(j, i, k)$ (linie 6–8),
4. $m(i, j, opp(i, j, k))$ (linie 9–12),
5. $m(i, k, opp(i, k, j))$ (linie 13–16).

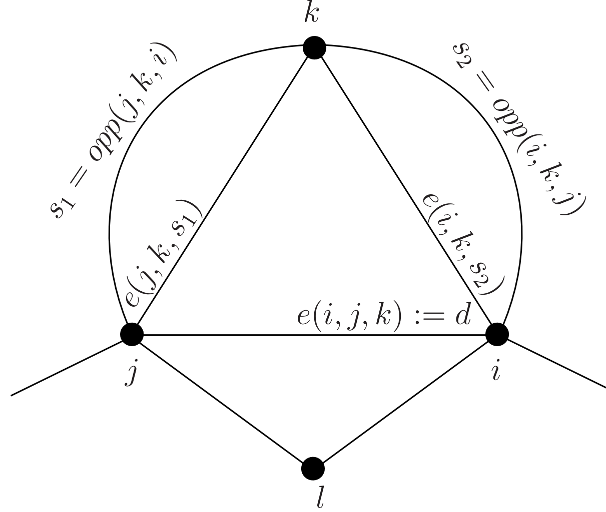
Znak zapytania symbolizuje zmienną $m(i, j, k)$, która jest właśnie obliczana.

wierzchołek zawiera lokalnie niepoprawną wartość zmiennej v i $m(\cdot, \cdot, \cdot)$.

Na początek przyjmijmy pesymistyczny przypadek, gdy zmienna $v(i)$ w każdym z wierzchołków i ma wartość mniejszą niż rzeczywista poprawna acentryczność wierzchołkowa, a ponadto nie ma dwóch wierzchołków i oraz j, które miałyby jednakowe wartości zmiennych $v(i)$ i $v(j)$.

Podobnie najbardziej pesymistyczna kolejność rozprzestrzeniania się poprawnej wartości zmiennej $m(\cdot, \cdot, \cdot)$ byłaby wtedy, gdy wartość $v(i)$ dla pewnego i, która jest mniejsza od rzeczywiście poprawnej wartości, rozprzestrzenia się pośród węzłów k, w których zmienne $v(k)$ mają wartość mniejszą od $v(i)$.

Ta faza zajmuje $\mathcal{O}(n)$ ruchów. Po jej zakończeniu drzewo dualne zawiera jako wartość promienia grafu zewnętrznie planarnego niepoprawną wartość $v(i)$. Pozostało jeszcze $n-1$ kandydatów z niepoprawną wartością minimalnej acentryczności do rozprzestrzenienia po drzewie dualnym. Ponownie pesymistyczny przypadek polega na rozprzestrzenieniu się największej spo-



Rysunek 3.5: Ilustracja reguły *4*.

śród tych wartości. Każda z tych faz składa się z $\mathcal{O}(n)$ ruchów a faz jest $\mathcal{O}(n)$, więc faza 5 zajmuje w sumie $\mathcal{O}(n^2)$ ruchów. □

Z lematów 4 i 5 oraz z faktu, że po każdym ruchu wykonanym przez reguły *1–4* może się wykonać cała faza *5*, wynika następujące twierdzenie.

Twierdzenie 2. *Algorytm 3 wykonuje się w $\mathcal{O}(n^4)$ ruchach.*

Na koniec przytaczamy twierdzenie dotyczące poprawności algorytmu.

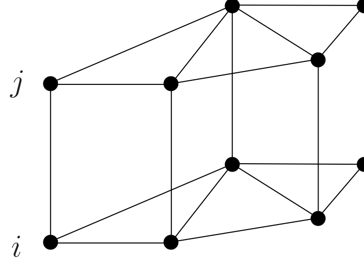
Twierdzenie 3. *Po zatrzymaniu się algorytmu 3 system znajduje się w stanie dozwolonym.*

Dowód. Przypuśćmy, że po zakończeniu algorytmu system jest w stanie niedozwolonym. Oznacza to, że wartownik reguły *5* jest spełniony co najmniej w jednym z wierzchołków. Przeczy to założeniu, że algorytm się zatrzymał. □

3.4 Rozszerzenie na iloczyn kartezjański MOP z K_2

W tej części zaprezentujemy samostabilizujący algorytm wyznaczający centrum iloczynu kartezjańskiego grafu maksymalnego zewnętrznie planarnego oraz grafu K_2. Algorytm ten został opublikowany w [BP14].

W iloczynie kartezjańskim dowolnego grafu (w szczególności planarnego) G i K_2 możemy wyróżnić dwie *warstwy*, z których każda jest izomorficzna z grafem G (rysunek 3.6).



Rysunek 3.6: Przykład iloczynu kartezjańskiego grafu maksymalnego zewnętrznie planarnego oraz K_2.

Twierdzenie 4. *W iloczynie kartezjańskim $G = M \times K_2$, gdzie M jest grafem maksymalnym zewnętrznie planarnym, graf indukowany przez $C(G)$ jest iloczynem kartezjańskim K_2 z grafem indukowanym przez $C(M_i)$ dla każdej z warstw M_i grafu G.*

Dowód. Niech $x \in V(G)$, wtedy $ecc_G(x) = ecc_{M_i}(x) + 1$, $i \in \{1, 2\}$. Stąd $C(G) = C(M_1) \cup C(M_2)$. □

Aby dostosować algorytm wyznaczania centrum dla grafu maksymalnego zewnętrznie planarnego do iloczynu takiego grafu z K_2, należy znaleźć sposób na określenie, czy dwa sąsiednie wierzchołki leżą w jednej warstwie, czy nie. W tym celu zdefiniujemy predykat

$$\text{pairing}(i, j) \Leftrightarrow (j \in N(i)) \wedge ((N(i) \cap N(j)) = \emptyset), \tag{3.1}$$

który stwierdza, czy wierzchołki i oraz j leżą w różnych warstwach na odpowiadających sobie pozycjach.

Użyjemy predykatu pairing do zdefiniowania *sąsiedztwa warstwowego*

$$L(i) = \{j \in N(i) : \neg\text{pairing}(i, j)\}, \tag{3.2}$$

czyli zbioru sąsiadów wierzchołka i, które leżą w tej samej warstwie.

Teraz możemy już nanieść drobną zmianę (algorytm 5) do reguły *1*, gdzie zastępujemy wyrażenie $N(i)$ wyrażeniem $L(i)$. Jest to jedyna wymagana zmiana w algorytmie. Z tego powodu zachowuje on wszystkie pierwotne właściwości dotyczące poprawności i czasu działania.

Algorytm 5: Reguła *1* algorytmu dla $MOP \times K_2$

1:
if $n(i) \neq L(i)$ **then**
$n(i) := L(i)$

3.5 Rozszerzenie na iloczyn kartezjański MOP z P_m

Dla grafów $MOP \times P_2$ udowodniliśmy, że centrum zawiera się w obu warstwach grafu (twierdzenie 4). Uogólnimy to twierdzenie na iloczyny kartezjańskie $MOP \times P_m$.

Twierdzenie 5. *Podgraf indukowany przez centrum iloczynu kartezjańskiego $MOP \times P_m$ jest iloczynem kartezjańskim podgrafów indukowanych przez centrum MOP oraz centrum P_m.*

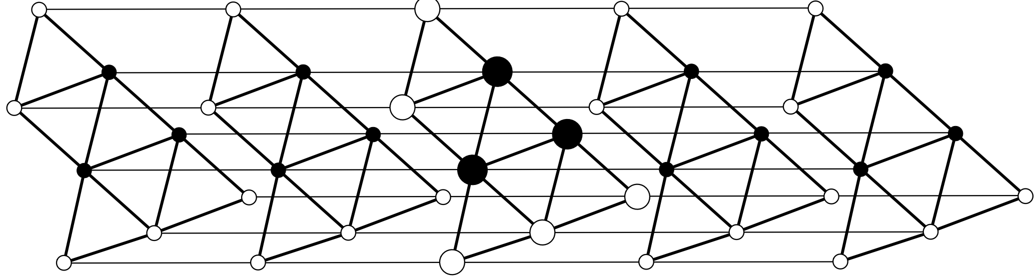
Dowód. Niech będzie dany maksymalny zewnętrznie planarny graf M oraz ścieżka P_m. Oznaczmy przez $r(M)$ acentryczność dowolnie wybranego wierzchołka $u \in C(M)$ i przez $r(P_m)$ acentryczność dowolnie wybranego wierzchołka $v \in C(P_m)$. Weźmy teraz graf $G = M \times P_m$ oraz wierzchołek $w = (u, v) \in V(G)$. Z definicji iloczynu kartezjańskiego mamy, że krawędzie w grafie G mają postać $\{(u_i, v_k), (u_i, v_l)\}$ lub $\{(u_i, v_k), (u_j, v_k)\}$, gdzie $\{u_i, u_j\} \in E(M)$ oraz $\{v_k, v_l\} \in E(P_m)$. Stąd wynika, że $e_G(w) = r(M) + r(P_m)$.

Aby zakończyć dowód, wystarczy wykazać, że w grafie G nie istnieje wierzchołek $x \in V(G)$ o acentryczności mniejszej, niż $e_G(w)$. Gdyby tak było, to w skład ścieżki łączącej x z najdalszym wierzchołkiem z $V(G)$ musiałaby wchodzić chociaż jedna krawędź postaci $\{(u_i, v_k), (u_j, v_l)\}$ ($u_i, u_j \in M$, $u_i \neq u_j$, $v_k, v_l \in P_m$, $v_k \neq v_l$), co, z definicji iloczynu kartezjańskiego grafów (definicja 7, s. 13), jest niemożliwe. □

Aby rozszerzyć nasz algorytm na iloczyn kartezjański ze ścieżką P_m (rysunek 3.7), powtórnie wykorzystamy predykat pairing (3.1) do rozpoznawania, czy dane dwa sąsiadujące wierzchołki należą do jednej warstwy. Dla wierzchołka i definiujemy zbiór sąsiadów należących do warstwy innej niż warstwa zawierająca wierzchołek i następująco

$$P(i) = \{j \in N(i) : \text{ pairing}(i, j)\}. \tag{3.3}$$

Ze wzorów (3.2) oraz (3.3) wynika, że dla każdego wierzchołka i zachodzi równość $N(i) = L(i) \cup P(i)$ i jednocześnie $L(i) \cap P(i) = \emptyset$. W iloczynie kartezjańskim $MOP \times P_m$ dla każdego wierzchołka i zbiór $P(i)$ zawiera 0 (gdy $m = 1$),



Rysunek 3.7: Przykład iloczynu kartezjańskiego grafu maksymalnego zewnętrznie planarnego oraz ścieżki P_5. Wierzchołki należące do centrum poszczególnych warstw zostały zaczernione. Wierzchołki należące do centralnej warstwy zostały powiększone. Wierzchołki należące do centrum całego grafu $MOP \times P_5$ są zaznaczone dużymi, zaczernionymi kółkami.

1 lub 2 elementy. Zatem jesteśmy w stanie rozróżnić warstwy, do których należą wierzchołki i, na skrajne (wtedy $|P(i)| \leqslant 1$; $|P(i)| = 0$, gdy ścieżka P_m jest zdegenerowana do pojedynczego wierzchołka P_1) oraz wewnętrzne (wtedy $|P(i)| = 2$). Do każdego węzła i wprowadzamy nową zmienną lx_i, której wartość zależy od warstwy, w której leży wierzchołek i

$$lx_i = \begin{cases} 0 & \text{dla } |P(i)| \leqslant 1, \\ 1 + \min_{j \in P(i)} lx_j & \text{dla } |P(i)| = 2. \end{cases}$$

Poprawnie obliczone wartości zmiennych lx_i w każdym z węzłów i pozwalają określić, czy dana warstwa jest jedyną lub jedną z dwóch centralnych warstw grafu. Jeśli dla danego wierzchołka i nie istnieje sąsiad j leżący w innej warstwie, dla którego wartość lx_j byłaby większa, niż lx_i, wtedy wierzchołek i należy do warstwy centralnej. Dla uproszczenia dalszego zapisu zdefiniujemy predykat $isInCentralLayer()$, który stwierdza, czy dany wierzchołek i należy do centralnej warstwy

$$isInCentralLayer(i) \Leftrightarrow \forall_{j \in P(i)} (lx_i \geqslant lx_j).$$

Poniżej przedstawiamy reguły algorytmu S-MOPxP-Center — algorytm 6. Wyznacza on centrum iloczynu kartezjańskiego $MOP \times P_m$ jako iloczyn kartezjański centrum ścieżki i centrum grafu maksymalnego zewnętrznie planarnego.

Po ustabilizowaniu się całego systemu, znajdując się w dowolnym wierzchołku grafu, możemy dojść do centrum wędrując w pierwszej kolejności do wierzchołków zawierających coraz większą wartość zmiennej lx (czyli przemieszczając się do warstw coraz bliższych centralnej). Gdy już dalsza wędrówka między warstwami jest niemożliwa z powodu braku węzłów o większej

Algorytm 6: S-MOPxP-CENTER — pierwsza część

1a:
  if $n(i) \neq N(i)$ **then**
    $n(i) := N(i)$
1b:
  if $p(i) \neq P(i)$ **then**
    $p(i) := P(i)$
1c:
  if $l(i) \neq L(i)$ **then**
    $l(i) := L(i)$
1d:
  if $|p(i)| \leqslant 1 \wedge lx_i \neq 0$ **then**
    $lx_i := 0$
1e:
  if $|p(i)| = 2 \wedge lx_i \neq 1 + \min_{j \in P(i)} lx_j$ **then**
    $lx_i := 1 + \min_{j \in P(i)} lx_j$
2:
  if $isInCentralLayer(i) \wedge \exists_{k \in L(i)} :\ c(i,k) \neq L(i) \cap n(k)$ **then**
    $c(i,k) := L(i) \cap n(k)$

wartości lx, należy wędrować tak, jak w przypadku pojedynczego grafu MOP — zgodnie z wartościami zmiennych $m(\cdot,\cdot,\cdot)$.

Na podstawie twierdzenia 5 centrum $MOP \times P_m$ znajduje się w warstwie centralnej czyli generowanej przez centrum ścieżki P_m. Odpowiadają za to reguły *1–1e*.

Za wyznaczenie centrum w centralnej warstwie odpowiadają reguły *2–5*, których poprawność została już udowodniona w rozdziale 3.3.

Złożoność algorytmu jest tego samego rzędu, jak wcześniejszego ($\mathcal{O}(n^4)$), z zastrzeżeniem, że faza opisana przez reguły *2–5* może wykonywać ruchy niezależnie od siebie w każdej warstwie, dopóki reguły *1a–d* nie wskażą tylko jednej lub dwóch centralnych warstw, dla których wymagane jest wykonanie reguły *2–5*.

Algorytm 6: S-MOPxP-Center — druga część

3a:

if $\exists_{j \in L(i)} : (|c(i,j)| = 1 \wedge (e(i,j,\emptyset) \neq -1 \vee opp(i,j,\emptyset) \neq k \vee opp(i,j,k) \neq \emptyset))$ **then**

$e(i,j,\emptyset) := -1$
$opp(i,j,\emptyset) := k$
$opp(i,j,k) := \emptyset$

where

$\{k\} = c(i,j)$

3b:

if $\exists_{j \in L(i)} : (|c(i,j)| = 1 \wedge (e(i,j,k) \neq d \vee v(i) \neq \max(|e(i,j,k)|, 1)))$ **then**

$e(i,j,k) := d$
$v(i) := \max(|e(i,j,k)|, 1)$

where

$$q = \begin{cases} -(1 + e(j,k,opp(j,k,i))) & \text{dla } e(j,k,opp(j,k,i)) > 0, \\ e(j,k,opp(j,k,i)) & \text{w pozostałych przypadkach} \end{cases}$$

$$d = \begin{cases} |e(i,k,opp(i,k,j))| & \text{dla } |e(i,k,opp(i,k,j))| \geqslant |q|, \\ q & \text{w pozostałych przypadkach} \end{cases}$$

$\{k\} = c(i,j)$

4:

if $\exists_{j \in L(i)} : (|c(i,j)| = 2 \wedge \exists_{k \in c(i,j)} : (e(i,j,k) \neq d \vee opp(i,j,k) \neq l \vee opp(i,j,l) \neq k \vee v(i) \neq \max(|e(i,j,k)|, |e(i,j,l)|)))$ **then**

$e(i,j,k) := d$
$opp(i,j,k) := l$
$opp(i,j,l) := k$
$v(i) := \max(|e(i,j,k)|, |e(i,j,l)|)$

where

$$q = \begin{cases} -(1 + e(j,k,opp(j,k,i))) & \text{dla } e(j,k,opp(j,k,i)) > 0, \\ e(j,k,opp(j,k,i)) & \text{w pozostałych przypadkach} \end{cases}$$

$$d = \begin{cases} |e(i,k,opp(i,k,j))| & \text{dla } |e(i,k,opp(i,k,j))| \geqslant |q|, \\ q & \text{w pozostałych przypadkach} \end{cases}$$

$\{k, l\} = c(i,j)$

5:

if $\exists_{j,k \in L(i)} : (k \in c(i,j) \wedge m(i,j,k) \neq MinEcc(i,j,k))$ **then**

$m(i,j,k) := MinEcc(i,j,k)$

Rozdział 4

Wybrane algorytmy samostabilizujące dla drzew

4.1 Algorytm znajdowania ważonego centroidu

Blair i Manne [BM03] pokazali szybki algorytm wyboru lidera w drzewach bez wag. My zaprezentujemy zmodyfikowany algorytm dla drzew z dodatnimi wagami na wierzchołkach, opublikowany w [BP12].

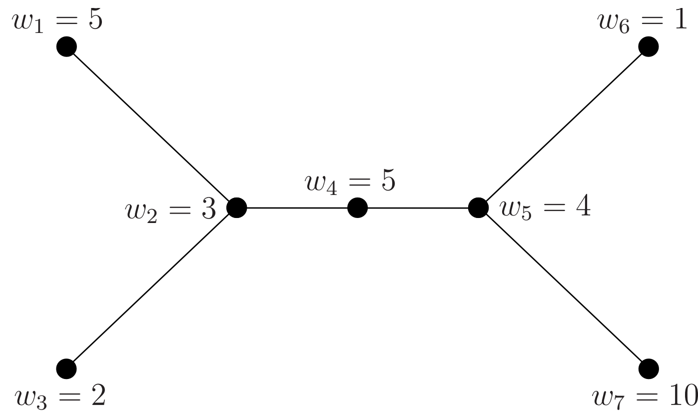
Niech będzie dane niezakorzenione drzewo T. Każdemu węzłowi i drzewa przyporządkowana jest waga w_i będąca dodatnią liczbą całkowitą (przykład na rysunku 4.1). Sumę wag wszystkich węzłów oznaczamy przez $\mathcal{W}_T$, zaś przez $T_i(j)$ będziemy oznaczać poddrzewo zawierające węzeł i, które powstanie po usunięciu krawędzi $\{i, j\}$ z drzewa T.

Definicja 11. Ważonym centroidem *nazywamy węzeł drzewa, którego usunięcie rozdzieli drzewo na dwa poddrzewa, w których suma wag wierzchołków jest nie większa od $\mathcal{W}_T/2$.*

Lemat 6. *Istnieje co najmniej jeden centroid w drzewie z wagami.*

Dowód. Załóżmy, że nie istnieje żaden centroid w drzewie. To oznacza, że jeśli usuniemy dowolny węzeł, zawsze jedno z powstałych poddrzew będzie miało sumę wag większą od $\mathcal{W}_T/2$.

Wybierzmy na początku dowolny węzeł w drzewie. Będziemy wędrować do sąsiada, który znalazłby się w poddrzewie o wadze większej od $\mathcal{W}_T/2$, gdybyśmy usuwali dany węzeł. Powtarzajmy tę czynność do momentu, gdy cofniemy się do węzła, w którym już byliśmy. Od tego momentu będziemy wędrować pomiędzy dwoma wciąż tymi samymi węzłami. Jeśli usuniemy krawędź je łączącą, powstaną dwa poddrzewa, z których każde ma wagę większą



Rysunek 4.1: Przykład drzewa z wagami.

od $\mathcal{W}_T/2$. Prowadzi to do sprzeczności, ponieważ suma ich wag musi być równa $\mathcal{W}_T$. □

Udowodniliśmy istnienie centroidów w każdym drzewie, więc teraz ograniczymy ich liczbę od góry.

Twierdzenie 6. *Liczba centroidów w drzewie z wagami jest równa 1 lub 2. Ponadto, jeśli istnieją 2 centroidy, to są one sąsiadami.*

Dowód. Załóżmy, że w danym drzewie istnieją dwa centroidy i, j (rysunek 4.2), które nie sąsiadują ze sobą. Wtedy istnieje co najmniej jeden węzeł pomiędzy nimi, który należy do poddrzewa indukowanego przez zbiór węzłów B, do którego nie należy ani i, ani j. Ponadto oznaczmy przez A i C zbiory wierzchołków należące do poddrzew zakorzenionych w sąsiadach (spoza B) wierzchołków odpowiednio i oraz j.

Możemy zatem zapisać równanie

$$w_A + w_i + w_B + w_j + w_C = \mathcal{W}_T, \tag{4.1}$$

gdzie w_A, w_B, w_C, w_i, w_j oznaczają wagi poddrzew indukowanych przez zbiory odpowiednio $A, B, C, \{i\}, \{j\}$. Węzeł i jest centroidem, więc zachodzi nierówność:

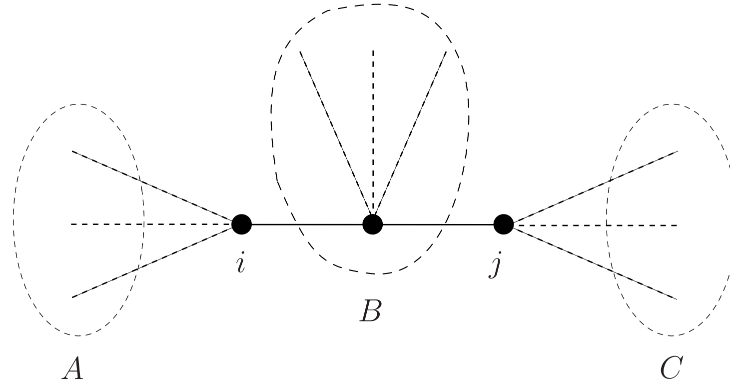
$$w_B + w_j + w_C \leqslant \mathcal{W}_T/2.$$

Podobnie dla j zachodzi:

$$w_A + w_i + w_B \leqslant \mathcal{W}_T/2.$$

Jeśli dodamy do siebie stronami dwie powyższe nierówności, otrzymamy:

$$w_A + w_i + 2w_B + w_j + w_C \leqslant \mathcal{W}_T,$$



Rysunek 4.2: Zakładana topologia drzewa w dowodzie Twierdzenia 6: i oraz j to centroidy.

co, porównując z (4.1), implikuje $w_B \leqslant 0$. Stoi to w sprzeczności z założeniem, że wagi wszystkich wierzchołków są dodatnie. To dowodzi, że centroidy zawsze ze sobą sąsiadują.

Wszystkie centroidy tworzą klikę, ponieważ każdy musi sąsiadować z każdym innym. W przypadku 3 lub więcej centroidów taka klika nie może wystąpić w drzewie, więc maksymalna liczba centroidów w drzewie to 2. □

Poniżej prezentujemy samostabilizujący algorytm W-Centroid znajdujący ważony centroid w drzewie z wagami. Algorytm wykonuje się w dwóch fazach. W czasie wykonania pierwszej z nich dla każdego węzła i, dla każdego jego sąsiada j obliczana jest waga poddrzewa $T_j(i)$.

Po ustabilizowaniu się pierwszej fazy każdy węzeł może określić wagę całego drzewa. Druga faza algorytmu wybiera centroid na podstawie wartości obliczonych w czasie pierwszej fazy.

Wykonanie obu faz się przeplata; nazywamy je fazami tylko dlatego, że poprawność wyników obliczeń drugiej zależy od zakończenia się pierwszej.

Na potrzeby naszego algorytmu każdy węzeł i zawiera tablicę wag W_i. Po zakończeniu pierwszej fazy, wartością $W_i[j]$ będzie waga poddrzewa $T_i(j)$. Poniżej podajemy algorytm pierwszej fazy.

Algorytm 7: Pierwsza faza algorytmu W-Centroid

R1:
if $\exists_{j \in N(i)}\, W_i[j] \neq w_i + \sum_{k \in N(i)-\{j\}} W_k[i]$ **then**
$W_i[j] := w_i + \sum_{k \in N(i)-\{j\}} W_k[i]$

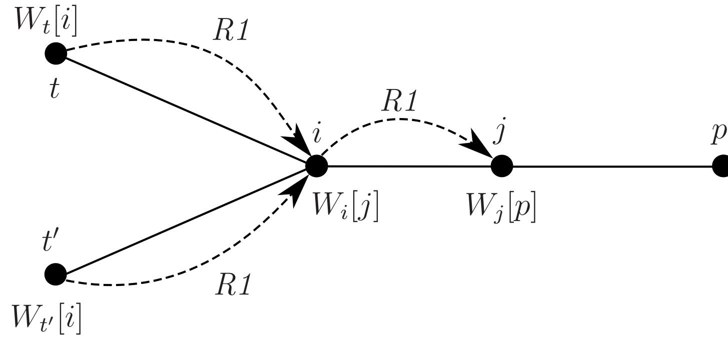
Poniższy lemat, dotyczący stabilizacji algorytmu R1 (algorytm 7), jest

analogiczny do lematu 3.1 z pracy [BM03], przy czym dotyczy drzew z wagami.

Lemat 7. *Dla drzewa z wagami algorytm R1 zatrzymuje się.*

Dowód. Niech będzie dany ciąg $E_1, E_2, \ldots, E_k$ wykonań reguły *R1* na ciągu kolejnych węzłów $s_1, s_2, \ldots, s_k$ tak, że ruch $E_i, 1 < i \leqslant k$ powoduje pierwszą zmianę wartości $W_{s_i}[s_{i+1}]$ w węźle s_i, wykorzystującą wartość $W_{s_{i-1}}[s_i]$ z węzła s_{i-1}. Na przykład ruch E_2 jako pierwszy spowoduje aktualizację wartości $W_{s_2}[s_3]$ wykorzystując wartość $W_{s_1}[s_2]$. Ostatni ruch dotyczy wartości $W_{s_k}[s_{k+1}]$.

Pokażemy teraz, że $s_i \neq s_j$ dla $i \neq j$. Zgodnie z regułą *R1*, zachodzą dwie nierówności: $s_i \neq s_{i+1}$ oraz $s_i \neq s_{i+2}$. Pierwsza wynika wprost z definicji reguły *R1*. Aby uzasadnić drugą nierówność, zauważmy, że na wartość $W_i[j]$ mają wpływ (poza w_i) tylko wartości $W_t[i], t \neq j$ (przykład na rysunku 4.3). Z kolei $W_i[j]$ ma wpływ tylko na $W_j[p], p \neq i$.



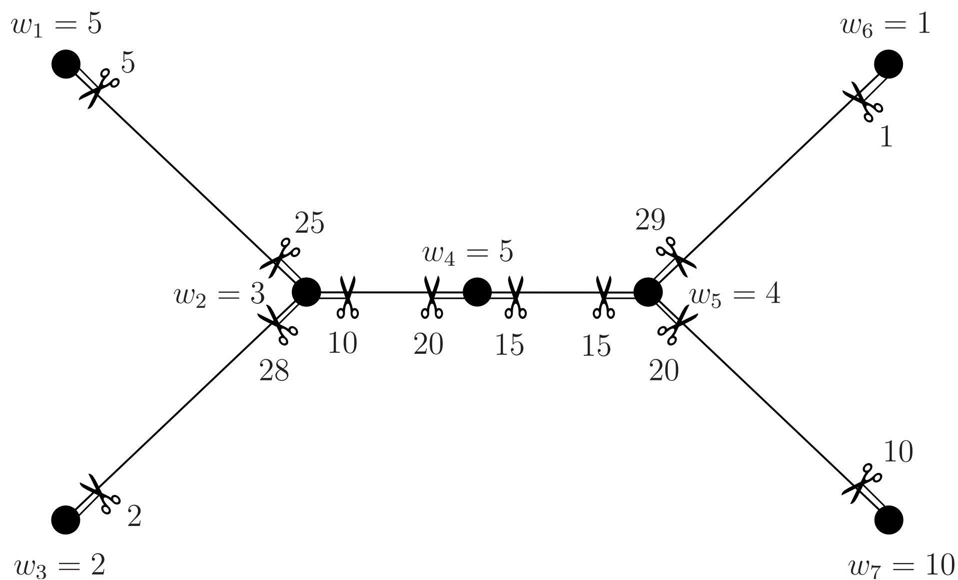
Rysunek 4.3: [BM03] Ciągi wykonania reguły *R1* wzdłuż ścieżek w drzewie. Wzajemną zależność poprawnych wartości zmiennych $W_\cdot[\cdot]$ pokazano strzałkami.

Weźmy teraz $t = s_i$ a $j = s_{i+c}$ dla $c \geqslant 2$. Równość $s_i = s_{i+c}$ implikuje istnienie cyklu w rozpatrywanym grafie. Mamy wtedy sprzeczność, gdyż rozpatrujemy drzewa. Skoro żadne dwa węzły w ciągu $s_1, s_2, \ldots, s_k$ nie mogą być sobie równe, to musi zachodzić $k \leqslant n$ z zasady szufladkowej Dirichleta.

W drzewie istnieje dokładnie jedna ścieżka pomiędzy dwoma wybranymi węzłami. Stąd liczba wszystkich opisanych powyżej maksymalnych ścieżek wykonania reguły *R1* jest równa co najwyżej n^2, jeśli rozróżniany jest zwrot ścieżki.

Przypuśćmy, że algorytm *R1* się nie zatrzymuje. To oznacza, że przynajmniej jedna z maksymalnych ścieżek S wykonania reguły *R1* musi wystąpić wielokrotnie w ciągu wszystkich wykonań reguły. Oznaczmy przez S^α i S^β dwie maksymalne różne ścieżki wykonań reguły *R1*, gdzie ciągi wykonania

są identyczne, czyli $E_i^\alpha = E_i^\beta$. Zauważmy, że początkowy ruch w każdym maksymalnym ciągu zależny jest od początkowego stanu systemu. Zatem ruchy E_1^α oraz E_1^β muszą być identyczne i pozostawiają po sobie identyczny stan w pierwszym (tym samym) elemencie ścieżki. Podobnie kolejny ruch w węźle s_2 — zależny tylko od stanu początkowego, wagi w_{s_2} oraz stanu $W_{s_1}[s_2]$ w węźle s_1 (obliczonego przez analogiczny ruch w s_1) — daje ten sam stan węzła s_2. Powtarzając to rozumowanie indukcyjnie dla każdego węzła w ścieżce, wnioskujemy, że S^α i S^β składają się z tych samych ruchów, więc są tożsame. Przeczy to założeniu, że algorytm się nie zatrzymuje. □



Rysunek 4.4: Drzewo z rysunku 4.1 z wyznaczonymi poprawnie (po zakończonej stabilizacji fazy pierwszej) wartościami $W_i[j]$ dla każdego węzła i oraz jego sąsiada j.

Lemat 8. *Po zatrzymaniu się algorytmu R1 dla drzewa z tablicą wag wartości $W_i[j]$ są obliczone poprawnie dla wszystkich węzłów i oraz ich sąsiadów j (rysunek 4.4).*

Dowód. Zastosujemy dowód indukcyjny względem rozmiaru drzewa.

Przypadkiem bazowym jest pojedynczy liść. Występują wtedy dwie możliwości: albo wartość $W_i[j]$ w liściu i jest poprawna, albo nie jest poprawna przed rozpoczęciem działania algorytmu. Jeśli wystąpiła druga możliwość, reguła *R1* w węźle i jest aktywna i po skończonym czasie zostanie wykonana, więc przestanie być aktywna. Jeśli zaś od początku $W_i[j]$ było poprawne,

reguła *R1* również nie jest aktywna. W obu przypadkach, z powodu nieaktywności reguły *R1*, wartość $W_i[j]$ nigdy się nie zmieni i do końca działania algorytmu pozostanie poprawna.

Założenie indukcyjne polega na stwierdzeniu, że dla danego węzła i każdy jego sąsiad k poza węzłem j ma obliczoną poprawnie wartość $W_k[i]$. Wtedy, jeśli w węźle i jest niepoprawna wartość zmiennej $W_i[j]$, może ona być poprawiona uwzględniając mieszczące się w sąsiadach wartości $W_k[i]$. Obliczona w ten sposób wartość zmiennej $W_i[j]$ jest poprawna i nie ulegnie zmianie do końca działania algorytmu, co dowodzi tezę indukcyjną i tym samym kończy dowód. □

Zajmiemy się teraz maksymalną liczbą ruchów, które musi wykonać algorytm, aby się zatrzymać. Niech $|T_i(j)|$ będzie liczbą węzłów w poddrzewie $T_i(j)$, zaś $c_i(j)$ liczbą zmian zmiennej $W_i[j]$ w trakcie działania algorytmu *R1*. Poniżej przytaczamy (przystosowany do naszego algorytmu) lemat z pracy [BM03], który pozwoli nam określić złożoność naszego algorytmu.

Lemat 9. *Po zakończeniu działania algorytmu R1 dla drzewa z tablicą wag, dla każdego węzła i oraz jego sąsiada j zachodzi nierówność* $c_i(j) \leqslant |T_i(j)|$.

Dowód. Dowód jak dla lematu 3.2 w [BM03]. □

Liczba ruchów wykonywanych przez algorytm *R1* jest taka, jak w drzewach bez wag, zatem następujący lemat również ma zastosowanie w naszym algorytmie.

Lemat 10. *Algorytm R1 dla drzewa z wagami wykonuje co najwyżej* $n\cdot(n-1)$ *ruchów.*

Dowód. Dla każdej pary węzłów i oraz j połączonych krawędzią zachodzi równość $|T_i(j)| + |T_j(i)| = n$. Stąd z lematu 9 łączna liczba zmian zmiennych $W_i[j]$ i $W_j[i]$ wynosi co najwyżej n. Zauważmy, że w drzewie jest $n-1$ par połączonych węzłów, co kończy dowód. □

Po ustabilizowaniu się pierwszej fazy czyli algorytmu *R1* każdy węzeł może określić wagę całego drzewa. Aby to udowodnić, wprowadzimy predykat, analogicznie do tego wprowadzonego przez Blaira i Mannego [BM03] dla rozmiaru drzewa, który orzeka, czy z punktu widzenia węzła i można określić wagę całego drzewa:

$$wCorrect_i \Leftrightarrow \left(\forall_{j \in N(i)} \;\; (\; W_i[j] = w_i + \sum_{k \in N(i)-\{j\}} W_k[i] \;) \right).$$

Porównując *wCorrect* z wartownikiem reguły *R1*, można stwierdzić, że dla danego węzła predykat jest prawdziwy wtedy i tylko wtedy, gdy reguła *R1* jest nieaktywna.

Po ustabilizowaniu się algorytmu *R1*, zachodzi poniższy lemat, który wskazuje, jak węzeł i może obliczyć wagę całego drzewa.

Lemat 11. *Jeśli zachodzi predykat* $wCorrect_i$*, wtedy* $W_i[j]+W_j[i] = W_i[k]+W_k[i]$ *dla dowolnych sąsiadów* j,k *węzła* i*.*

Dowód. Po ustabilizowaniu algorytmu *R1*, z definicji predykatu $wCorrect_i$ wynika:

$$\begin{aligned} W_i[j] + W_j[i] &= w_i + \textstyle\sum_{q\in N(i)-\{j\}} W_q[i] + W_j[i] \\ &= w_i + \textstyle\sum_{q\in N(i)} W_q[i] \\ &= w_i + \textstyle\sum_{q\in N(i)-\{k\}} W_q[i] + W_k[i] \\ &= W_i[k] + W_k[i] \end{aligned}$$

□

W ten sposób każdy węzeł i może stwierdzić, czy waga $\mathcal{W}_{T_j(i)}$ zapisana w którymś z jego sąsiadów j jest większa od połowy wagi całego drzewa. Jeśli ta własność zachodzi, węzeł i nie może być ważonym centroidem, odległość od centroidu jest mniejsza dla j, niż dla i. Jest nawet możliwe, że j jest wtedy centroidem. Z drugiej strony, jeśli dla żadnego sąsiada j węzła i nie zachodzi $\mathcal{W}_{T_j(i)} > \mathcal{W}_T/2$, węzeł i jest centroidem.

Zgodnie z twierdzeniem 6 (s. 36) w drzewie mogą wystąpić dwa centroidy. Nasz algorytm kończy działanie wskazując centroid o większym identyfikatorze.

Za każdym razem, gdy predykat $wCorrect_i$ jest spełniony, węzeł i może obliczyć wagę całego drzewa. Wtedy to obliczenie będzie poprawne lokalnie. Globalna poprawność jest zapewniona, gdy algorytm *R1* zakończył swoje działanie we wszystkich wierzchołkach. Przez $\mathcal{W}_i$ będziemy oznaczać wagę całego drzewa obliczoną przez węzeł i. Zauważmy, że póki algorytm *R1* się nie skończył, może zachodzić sytuacja, że $\mathcal{W}_i \neq \mathcal{W}_T$ dla pewnego i.

Poniżej (s. 42) prezentujemy drugą fazę składającą się z czterech kolejnych (ostatnich) reguł naszego algorytmu. Ich celem jest wskazanie centroidu — po ustabilizowaniu, jeśli węzeł jest centroidem, wskazuje na siebie; w przeciwnym razie wskazuje na sąsiada, którego odległość od centroidu jest mniejsza. Tak więc każdy węzeł i zawiera zmienną p_i, której wartością jest identyfikator sąsiada lub samego siebie w przypadku centroidu.

W wartownikach wszystkich reguł od *R2* do *R5* predykat $wCorrect_i$ musi być spełniony. W wyniku tego reguły te są nieaktywne, póki, z punktu widzenia węzła i, wagi sąsiadujących z nim poddrzew nie zostały jeszcze poprawnie

Algorytm 8: Druga faza algorytmu W-Centroid

R2:
if $(wCorrect_i) \wedge (\forall_{j \in N(i)} W_j[i] < \mathcal{W}_i/2) \wedge (p_i \neq i)$ **then**
$p_i := i$

R3:
if $(wCorrect_i) \wedge (\exists_{j \in N(i)} W_j[i] > \mathcal{W}_i/2) \wedge (p_i \neq j)$ **then**
$p_i := j$

R4:
if $(wCorrect_i) \wedge (\exists_{j \in N(i)} W_j[i] = \mathcal{W}_i/2) \wedge (ID_i > ID_j) \wedge (p_i \neq i)$ **then**
$p_i := i$

R5:
if $(wCorrect_i) \wedge (\exists_{j \in N(i)} W_j[i] = \mathcal{W}_i/2) \wedge (ID_i < ID_j) \wedge (p_i \neq j)$ **then**
$p_i := j$

obliczone za pomocą reguły *R1*. Stąd jakakolwiek reguła spośród *R2* aż do *R5* może być aktywna jedynie, gdy *R1* nie jest aktywna.

Przeznaczeniem reguły *R2* jest oznaczenie w węźle i, że i jest jedynym centroidem. Reguła *R3* ustawia wartość p_i na identyfikator sąsiada, który jest bliżej centroidu, niż i (jeśli oczywiście i nie jest centroidem). Reguły *R4* oraz *R5* są aktywne jedynie w sytuacji występowania dwóch centroidów — wtedy obie wybierają centroid o większym identyfikatorze: *R4* jest aktywowana w wybieranym centroidzie i wskazuje na tenże węzeł, natomiast *R5* jest aktywowana w centroidzie, który nie będzie wskazywany — wskazuje wybierany centroid.

Lemat 12. *Algorytm R1–R5 zatrzymuje się po co najwyżej* $2n^2 - n$ *ruchach.*

Dowód. Zgodnie z lematem 10 pierwsza faza (stabilizacja *R1*) potrzebuje co najwyżej $n^2 - n$ ruchów.

Druga faza składająca się z reguł *R2–R5* potrzebuje następującej liczby ruchów: każdy węzeł może wykonać co najwyżej jeden ruch spośród *R2–R5* przed jakimkolwiek wykonaniem reguły *R1*, co daje n ruchów. Ponadto po każdym ruchu *R1* w węźle może się wykonać jeden ruch spośród *R2–R5*, co daje kolejne $n^2 - n$ ruchów. W sumie dostajemy $n^2 - n + n + n^2 - n = 2n^2 - n$ ruchów. □

Lemat 13. *Po zatrzymaniu się algorytmu R1–R5, wartości* p_i *w każdym węźle i wyznaczają drzewo zakorzenione w centroidzie drzewa.*

Dowód. Wykonanie reguł *R2–R5* nie ma żadnego wpływu na aktywność reguły *R1*. Dlatego od momentu zakończenia pierwszej fazy algorytmu wartości $W_i[j]$ będą poprawne i nie zmienią się do końca działania algorytmu.

Weźmy stan systemu po zakończeniu działania drugiej fazy. Każdy węzeł i, który nie jest centroidem, ma dokładnie jednego sąsiada j, dla którego $W_j[i] > \mathcal{W}_T/2$. Tuż po zakończeniu pierwszej fazy, w takim węźle i mogła być aktywna reguła *R3*, ale nie mogła być aktywna żadna z reguł *R2*, *R4* ani *R5*. Po wykonaniu reguły *R3* zmienna p_i będzie wskazywać na węzeł j, dla którego odległość od centroidu jest mniejsza, niż dla i.

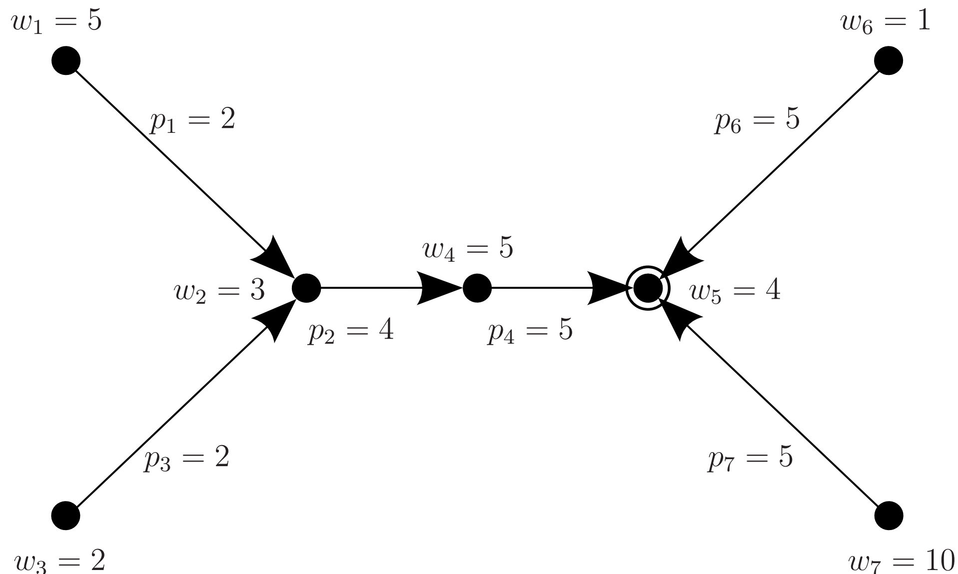
Załóżmy teraz, że istnieje jeden ważony centroid, czyli nie ma węzła dla którego zachodziłaby równość $W_j[i] = \mathcal{W}_T/2$. Każdy sąsiad j centroidu i spełnia nierówność $W_j[i] < \mathcal{W}_T/2$. Wtedy węzeł i może zastosować regułę *R2*, wskazując siebie jako centroid: $p_i = i$. Dodatkowo, i nie może wykonać żadnej z reguł *R3–R5*, dlatego wszystkie inne węzły mogą zastosować tylko regułę *R3*, co omówiliśmy w poprzednim akapicie.

Załóżmy teraz drugi przypadek — gdy w drzewie występują dwa centroidy. Oznacza to, że istnieją dwa sąsiadujące ze sobą węzły p oraz q, spełniające równość $W_p[q] = W_q[p] = \mathcal{W}_T/2$. Wtedy żaden z węzłów i w drzewie nie spełnia nierówności $W_j[i] < \mathcal{W}_T/2$ dla wszystkich sąsiadów $j \in N(i)$. Stąd oba wierzchołki będące centroidami nie mogą mieć aktywnych reguł *R2* ani *R3*, zaś wszystkie pozostałe mogą mieć aktywną tylko regułę *R3*. Węzły niebędące centroidem ustawią wartości p_i zgodnie z regułą *R3*, jak opisaliśmy powyżej. Węzły będące centroidami są sąsiadami, z których ten o większym identyfikatorze zostanie korzeniem (wybranym przez algorytm centroidem), zaś drugi będzie na niego wskazywał. □

Z naszych rozważań wynika ostatecznie następujący wniosek.

Wniosek 1. *Jeśli algorytm R1–R5 rozpocznie swoje działanie w systemie o dowolnym stanie początkowym, wykona co najwyżej $2n^2 - n$ ruchów. Po zatrzymaniu wartości zmiennych p_i w każdym węźle i będą wskazywać najkrótszą drogę do wybranego centroidu.*

Końcowy stan przykładowego systemu pokazanego na rysunku 4.1 jest pokazany na rysunku 4.5.



Rysunek 4.5: Stan ustabilizowanego systemu pokazanego uprzednio na rysunku 4.1. Pokazano stan zmiennej p_i w każdym węźle i tak, że strzałka ma zwrot $i \to j$ jeśli $p_i = j$. System zawiera dwa centroidy (węzły 4 i 5), z których algorytm wybrał ten o większym identyfikatorze (na rysunku zaznaczony dodatkowym okręgiem).

4.2 Algorytm wyznaczania centrum artykulacyjnego

Poniżej prezentujemy algorytm S-CuttingCenter (Algorytm 9, s. 45) — poprawioną wersję algorytmu Chaudhuriego i Thompsona [CT04], o którym pisaliśmy w rozdziale 2.2 (s. 16). W pierwotnym algorytmie zastępujemy dwie pierwsze fazy regułami *R1*–*R5* z algorytmu Blaira i Mannego [BM03] oraz dodajemy regułę *R6* ze zmienną c_i dla każdego węzła i, oznaczającą liczbę artykulacyjną (definicja 5, s. 11). Celem działania tych sześciu reguł jest ukorzenienie drzewa oraz wyznaczenie liczby artykulacyjnej. W przeciwieństwie do S-CuttingCenter, oryginalny algorytm [CT04] wymagał, by drzewo było ukorzenione. Dla poprawności i złożoności algorytmu nie ma znaczenia, który z węzłów zostanie korzeniem. Po ustabilizowaniu tej części algorytmu każdy węzeł poza korzeniem będzie wskazywał swojego rodzica (zmienna p), dodatkowo będzie miał również obliczoną liczbę artykulacyjną (zmienna c).

Reguła *R6* może być aktywna tylko wtedy, gdy żadna z wcześniejszych reguł nie jest aktywna — zapewnia to rozbudowany warunek, który składa się z koniunkcji zaprzeczonych fragmentów warunków reguł *R2*–*R5* oraz za-

Algorytm 9: S-CuttingCenter — pierwsza część (reguły *R1–R6*)

R1:
if $\exists_{j\in N(i)} : size_i(j) \neq 1 + \sum_{k\in N(i)-\{j\}} size_k(i)$ **then**
$size_i(j) := 1 + \sum_{k\in N(i)-\{j\}} size_k(i)$

R2:
if $sizeCorrect_i \wedge (\forall_{j\in N(i)}\ size_j(i) < n_i/2) \wedge p_i \neq i$ **then**
$p_i := i$

R3:
if $sizeCorrect_i \wedge (\exists_{j\in N(i)}\ size_j(i) > n_i/2) \wedge p_i \neq j$ **then**
$p_i := j$

R4:
if $sizeCorrect_i \wedge (\exists_{j\in N(i)}\ size_j(i) = n_i/2) \wedge ID_i > ID_j \wedge p_i \neq i$ **then**
$p_i := i$

R5:
if $sizeCorrect_i \wedge (\exists_{j\in N(i)}\ size_j(i) = n_i/2) \wedge ID_i < ID_j \wedge p_i \neq j$ **then**
$p_i := j$

R6:
if $sizeCorrect_i \wedge \neg\Big((\forall_{j\in N(i)}\ size_j(i) < n_i/2) \wedge p_i \neq i\Big) \wedge$
$\neg\Big((\exists_{j\in N(i)}\ size_j(i) > n_i/2) \wedge p_i \neq j\Big) \wedge$
$\neg\Big((\exists_{j\in N(i)}\ size_j(i) = n_i/2) \wedge ID_i > ID_j \wedge p_i \neq i\Big) \wedge$
$\neg\Big((\exists_{j\in N(i)}\ size_j(i) = n_i/2) \wedge ID_i < ID_j \wedge p_i \neq j\Big) \wedge c_i \neq$
$\frac{1}{2}\sum_{j\in N(i)}\Big(size_j(i) \cdot \sum_{k\in N(i)\setminus\{j\}} size_k(i)\Big)$ **then**
$c_i := \frac{1}{2}\sum_{j\in N(i)}\Big(size_j(i) \cdot \sum_{k\in N(i)\setminus\{j\}} size_k(i)\Big)$

przeczonego całego wartownika reguły *R1*.

Druga część algorytmu S-CuttingCenter (algorytm 10) to faza III algorytmu Chaudhuriego i Thompsona [CT04].

Blair i Manne udowodnili, że algorytm składający się z reguł *R1–R5* stabilizuje system w stanie ukorzenionym. Poprawność algorytmu *R1–R5* wynika z twierdzenia 1 (s. 16).

Ponadto zmienna $size_i$ w każdym węźle i, po zatrzymaniu się algorytmu ma poprawną wartość, w przeciwnym razie reguła *R1* byłaby nadal aktywna.

Zmienna c_i w każdym węźle i zależy od wartości $size_j(i)$ w każdym sąsiedzie j węzła i. Z wartownika reguły *R6* oraz tego, że c_i jest modyfikowane tylko w *R6*, wynika, że jeśli system się ustabilizował, to

$$c_i = \frac{1}{2}\sum_{j\in N(i)}\left(size_j(i) \cdot \sum_{k\in N(i)\setminus\{j\}} size_k(i)\right),$$

Algorytm 10: S-CuttingCenter — druga część (reguły *R7–R11*)

R7:
  if $DMAX_i \neq \max(\{DMAX_j : p_j = i\} \cup \{c_i\})$ **then**
    $DMAX_i := \max(\{DMAX_j : p_j = i\} \cup \{c_i\})$
R8:
  if $p_i = i \wedge AMAX_i \neq DMAX_i$ **then**
    $AMAX_i := DMAX_i$
R9:
  if $AMAX_i \neq AMAX_{p_i}$ **then**
    $AMAX_i := AMAX_{p_i}$
R10:
  if $c_i = AMAX_i \wedge CCENTER_i \neq 1$ **then**
    $CCENTER_i := 1$
R11:
  if $c_i \neq AMAX_i \wedge CCENTER_i \neq 0$ **then**
    $CCENTER_i := 0$

zatem każdy węzeł ma obliczoną liczbę artykulacyjną w postaci c_i. Odpowiada to formule (2.1) na s. 17.

Wprost z definicji reguł *R7–R11* wynika, że wykonanie każdej reguły powoduje jej dezaktywację. Jeśli algorytm się zakończył, to wszystkie zmienne p mają poprawne wartości — wynika to z twierdzenia 1. Wtedy z fałszywości wartowników reguł *R7–R11* wynika, że zmienne $DMAX$, $AMAX$ i $CCENTER$ przyjęły oczekiwane wartości, tj. $DMAX_i$ to największa liczba artykulacyjna w poddrzewie zakorzenionym w węźle i; $AMAX_i$ to największa liczba artykulacyjna w całym drzewie; zaś $CCENTER_i$ jest prawdziwa wtedy i tylko wtedy, gdy i należy do centrum artykulacyjnego.

Złożoność części *R1–R5* będącej algorytmem Blaira i Mannego [BM03] wynosi $\mathcal{O}(n^2)$ — twierdzenie 1. Reguła *R6* nie zwiększa złożoności algorytmu *R1–R6* w stosunku do *R1–R5*.

Lemat 14. *Algorytm R1–R6 zatrzymuje się po wykonaniu $\mathcal{O}(n^2)$ ruchów.*

Dowód. Rozpatrzmy, jako pierwszy, przypadek gdy w momencie uruchomienia systemu lub po wystąpieniu przejściowego błędu reguła *R6* jest aktywna w pewnym podzbiorze węzłów systemu. W każdym z tych węzłów reguła *R6* wykona się co najwyżej raz i stan systemu będzie taki, jak w drugim rozpatrywanym przypadku.

W drugim przypadku warto zauważyć, że wartownik reguły *R6* wyklucza jej aktywność równocześnie z którąkolwiek z reguł *R1–R5*. Ponadto wykonanie reguły *R6* nie wpływa na aktywacje reguł *R1–R5*, ponieważ *R6* mody-

fikuje tylko zmienną c, która nie występuje w wartownikach wcześniejszych reguł.

Z drugiej strony po każdym wykonaniu którejkolwiek z reguł *R1–R5* w węźle i lub którymś z jego sąsiadów, reguła *R6* może stać się aktywna i wykonać ruch w węźle i, po czym powraca do stanu nieaktywnego. Zatem po każdym ruchu według reguł *R1–R5*, reguła *R6* może się wykonać co najwyżej 2 razy, co dodaje do ogólnej złożoności liczbę ruchów tego samego rzędu jak złożoność części *R1–R5*, pozostawiając w sumie $\mathcal{O}(n^2)$ ruchów. □

Lemat 15. *W wyniku ruchu wykonanego przez jedną z reguł R1–R6, może się wykonać co najwyżej $\mathcal{O}(n)$ ruchów według reguł R7–R11.*

Dowód. Jeden ruch według reguł *R1–R6* może zmienić wartość zmiennej c w co najwyżej jednym węźle drzewa. Zmiana wartości c w jednym z węzłów może powodować aktywację i wykonanie w nim reguły *R7* i dalszy ciąg wykonań tej reguły w węzłach na ścieżce od danego węzła aż do korzenia — co najwyżej n ruchów. Następnie dla wszystkich węzłów wykona się maksymalnie jeden ruch reguły z pary *R8*, *R9* i jeden ruch z pary *R10*, *R11* — co najwyżej $2n$ ruchów. Łącznie daje to nie więcej, niż $3n$ ruchów. □

Lemat 16. *Od momentu uruchomienia systemu lub wystąpienia przejściowego błędu reguły R7–R11 mogą się wykonać co najwyżej $\mathcal{O}(n^2)$ razy zanim reguły R1–R6 wykonają nie więcej, niż $\mathcal{O}(n)$ ruchów.*

Dowód. Pesymisty przypadek ma miejsce, gdy wszystkie zmienne $DMAX$ i $AMAX$ mają niepoprawne wartości. Pierwszy ruch wykonuje korzeń, poprawiając wartość swojej zmiennej $DMAX$, co powoduje następnie poprawienie wartości zmiennej $AMAX$ w całym drzewie od korzenia, w dół aż do wszystkich liści. Kolejna fala ruchów zaczyna się w dziecku korzenia od aktualizacji zmiennej $DMAX$, wędruje do korzenia i ponownie do wszystkich liści (następuje poprawienie $AMAX$). Takich fal może być $\mathcal{O}(n)$, a każda zajmuje $\mathcal{O}(n)$ ruchów, co daje $\mathcal{O}(n^2)$ ruchów.

W przypadku wystąpienia przeplotu z regułami *R1–R6*, po każdym z $\mathcal{O}(n)$ ruchów *R1–R6*, zgodnie z lematem 15, wykona się maksymalnie $\mathcal{O}(n)$ ruchów, co w sumie daje również $\mathcal{O}(n^2)$ ruchów. □

Ostatecznie otrzymujemy złożoność całego algorytmu S-CuttingCenter.

Twierdzenie 7. *Algorytm S-CuttingCenter zatrzymuje się po $\mathcal{O}(n^3)$ ruchach.*

Dowód. Bez utraty ogólności podzielmy wykonanie algorytmu na dwa etapy: pierwszy, w którym reguły *R1–R6* wykonały $\mathcal{O}(n)$ ruchów oraz reszta ruchów aż do zakończenia.

Zgodnie z lematem 16, w pierwszym etapie algorytm wykona co najwyżej $\mathcal{O}(n^2)$ ruchów. Z lematu 14 wynika, że w drugim etapie reguły *R1–R6* wykonają się $\mathcal{O}(n^2)$ razy, zaś na mocy lematu 15 po każdym z tych wykonań może się wykonać $\mathcal{O}(n)$ ruchów według reguł *R7–R11*. Zatem drugi etap może zająć maksymalnie $\mathcal{O}(n^3)$ ruchów.

W sumie pierwszy i drugi etap wymagają $\mathcal{O}(n^3)$ ruchów do zakończenia algorytmu. □

Rozdział 5

Zbiór cykli fundamentalnych w dowolnym grafie

5.1 Wprowadzenie

W tym rozdziale przedstawimy modyfikację algorytmu Chaudhuriego [Cha99] znajdującego zbiór cykli fundamentalnych w dowolnym grafie. Oryginalny algorytm będziemy nazywać SASFC I. Nasz zmodyfikowany algorytm SASFC II ma pesymistyczną złożoność czasową $\mathcal{O}(n)$, podczas gdy SASFC I ma złożoność $\mathcal{O}(n^2)$.

Niech będzie dana para cykli C_1 oraz C_2 w grafie G. Różnicą symetryczną cykli C_1 i C_2 nazywamy taki cykl $C = C_1 \oplus C_2$, że krawędź $e \in E(C)$ należy albo do $E(C_1)$, albo do $E(C_2)$. Zbiór cykli fundamentalnych (ang. *set of fundamental cycles*) jest minimalnym (w sensie liczności) zbiorem cykli, którego elementy umożliwiają zbudowanie każdego cyklu w grafie przy pomocy różnicy symetrycznej.

Oryginalny algorytm Chaudhuriego SASFC I potrzebuje $\mathcal{O}(n^2)$ ruchów, aby się zatrzymać, pod warunkiem, że dane jest rozpinające drzewo przeszukiwania wgłąb grafu. Złożoność $\mathcal{O}(n^2)$ algorytmu SASFC I jest możliwa do osiągnięcia jedynie wtedy, gdy informacja na temat drzewa rozpinającego jest nienaruszalna w trakcie działania algorytmu, czyli w modelu quasi-samostabilizacji.

Nasz algorytm SASFC II wymaga dwubitowej zmiennej q w każdym wierzchołku grafu. Stan początkowy zmiennej q powinien być ustawiony w trakcie lub tuż po wyznaczeniu drzewa rozpinającego wgłąb. Zmienna q musi być umieszczona w pamięci chronionej, tak aby nie była możliwa jej modyfikacja z zewnątrz w wyniku przejściowego błędu. Gdy wyznaczone jest drzewo rozpinające wgłąb (DFS) oraz w każdym wierzchołku zainicjowana jest zmienna q,

może się rozpocząć właściwa część algorytmu SASFC II.

Tak, jak w przypadku SASFC I, wynik działania SASFC II zapisany jest w sposób rozproszony w całym grafie jako stan zmiennych lokalnych poszczególnych wierzchołków. Gdy algorytm się zatrzymuje, każdy wierzchołek przechowuje informację, ile cykli fundamentalnych przez niego przechodzi i jakie są ich identyfikatory.

5.2 Notacja

Przedstawimy teraz notację, której będziemy używać w dalszej części rozdziału — większość jest zgodna z tą używaną w [Cha99]. Przez $T_G(r) = (V, E_T)$ będziemy oznaczać drzewo rozpinające wgłąb grafu G, gdzie r jest korzeniem. Za pomocą $NT = E(G) \setminus E_T$ będziemy oznaczać zbiór cięciw grafu G względem drzewa T, czyli krawędzi z G nienależących do T.

Będziemy ponadto używać następującej notacji:

$n(i)$ zbiór właściwych sąsiadów wierzchołka i,

$p(i)$ rodzic wierzchołka i w drzewie $T_G(r)$ (przyjmujemy, że $p(r) = null$),

$c(i)$ zbiór dzieci wierzchołka i w drzewie $T_G(r)$

$nt(i)$ zbiór cięciw incydentnych z wierzchołkiem i,

$C(i, j)$ cykl fundamentalny zawierający cięciwę $\{i, j\}$ oraz ścieżkę pomiędzy wierzchołkami i i j, zawartą w drzewie $T_G(r)$,

$a(i)$ zbiór przodków wierzchołka i w drzewie $T_G(r)$ (przyjmujemy, że $i \in a(i)$),

$d(i)$ zbiór potomków wierzchołka i w drzewie $T_G(r)$ (przyjmujemy, że $i \in d(i)$),

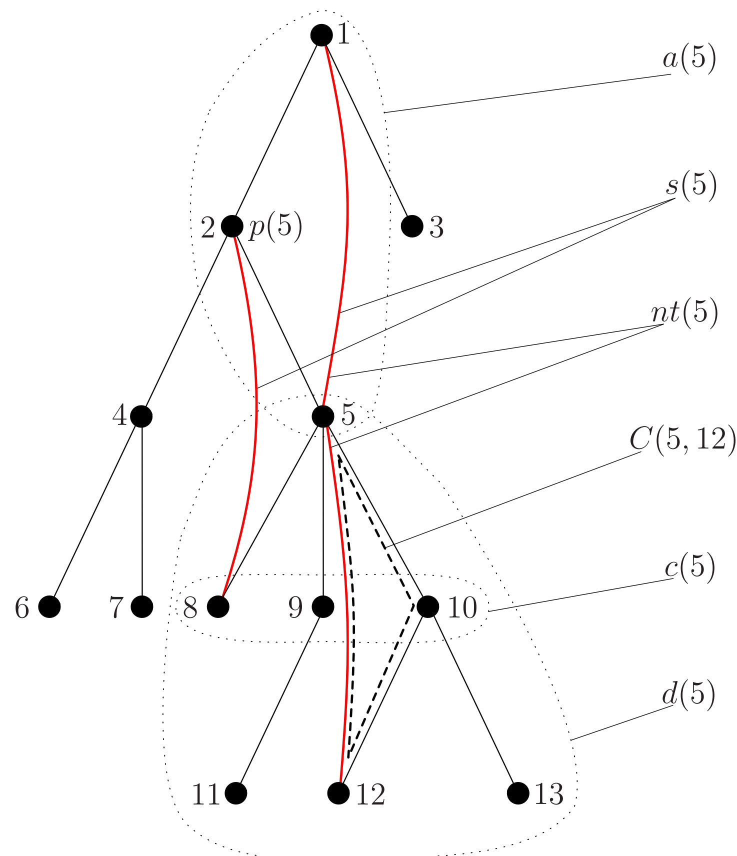
$s(i)$ zbiór cięciw łączących potomków wierzchołka i z właściwymi przodkami i, czyli $s(i) = \{\{x, y\} : \{x, y\} \in NT, x \in d(i), y \in a(i) \setminus \{i\}\}$,

$su(i)$ suma zbiorów $s(j)$ dla każdego dziecka j węzła i, czyli $su(i) = \bigcup_{j \in c(i)} s(j)$,

$fc(i)$ zbiór cięciw należących do cykli zawierających wierzchołek i, czyli $fc(i) = \{\{x, y\} : \{x, y\} \in NT, x \in a(i), y \in d(i)\}$,

$A \oplus B$ — różnica symetryczna zbiorów A i B; w przypadku grafów: graf, którego zbiór krawędzi jest różnicą symetryczną zbiorów krawędzi grafów A i B.

Na rysunku 5.1 przedstawiono przykłady ilustrujące powyżej przedstawioną notację.



Rysunek 5.1: Przykład użycia notacji względem wierzchołka $i = 5$: $n(5) = \{1,2,8,9,10,12\}$, $su(5) = \{\{2,8\},\{5,12\}\}$, $fc(5) = \{\{1,5\},\{2,8\},\{5,12\}\}$. Czerwone linie oznaczają cięciwy. Linie ciągłe oznaczają krawędzie należące do drzewa, natomiast jeden z cykli ($C(5,12)$) zaznaczono linią przerywaną.

5.3 Algorytm

Jak już było wspomniane, nasz algorytm SASFC II jest modyfikacją algorytmu Chaudhuriego [Cha99], który oznaczamy na potrzeby niniejszej pracy

jako SASFC I. Nasza zmiana wprowadza jedynie inną prekondycję dla systemu w momencie startu oraz kolejność, w jakiej węzły obliczeniowe (wierzchołki grafu) są uaktywniane w celu wykonania ruchu. Z tego powodu zmienia się złożoność obliczeniowa w stosunku do oryginalnego algorytmu: z $\mathcal{O}(n^2)$ dla SASFC I maleje do $\mathcal{O}(n)$ dla SASFC II.

W obu przypadkach, jeśli nie jest znane rozpinające drzewo przeszukiwania wgłąb, musi być ono wyznaczone np. za pomocą algorytmu Collina i Doleva [CD94]. Konieczność wyznaczenia drzewa rozpinającego w przypadku obu algorytmów zwiększy złożoność obliczeniową do $\mathcal{O}(n^3)$.

Algorytm SASFC II do swojego działania wymaga istnienia zmiennej $q(i)$ dla każdego wierzchołka i. Na początku działania algorytmu jej wartość powinna być ustawiona na stałą α.

Jeśli dane jest poprawne drzewo rozpinające, ale wartości $q(i)$ są zainicjowane nieprawidłowo, złożoność wraca do poziomu $\mathcal{O}(n^2)$. W takim pesymistycznym przypadku algorytm SASFC II działa jak SASFC I.

Znaczenie wszystkich zmiennych (poza $q(i)$) oraz nadawane im wartości w algorytmie SASFC II są takie same, jak w oryginalnym algorytmie Chaudhuriego SASFC I. Poniżej przedstawiamy algorytm SASFC I; jego pierwotną formę możemy znaleźć w pracy [Cha99].

Załóżmy, że drzewo rozpinające DFS jest dane. Zmienne $p(i)$, $c(i)$, $nt(i)$ zawarte w każdym wierzchołku określają strukturę drzewa $T_G(r)$. Głównym zadaniem algorytmu jest wyznaczenie wartości $fc(i)$ dla każdego wierzchołka i w systemie. Poprawne wartości zmiennej $fc(i)$ rozprzestrzeniają się wraz ze zmienną $s(i)$ od liści aż do korzenia. Algorytm Chaudhuriego przedstawiony jest poniżej jako algorytm 11.

Algorytm 11: SASFC I: oryginalny algorytm Chaudhuriego [Cha99].

liść:
  if $c(i) = \emptyset \wedge (s(i) \neq nt(i) \vee fc(i) \neq nt(i))$ **then**
    $s(i) := nt(i)$
    $fc(i) := nt(i)$
wewnętrzny:
  if $c(i) \neq \emptyset \wedge (s(i) \neq nt(i) \oplus su(i) \vee fc(i) \neq su(i))$ **then**
    $s(i) := nt(i) \oplus su(i)$
    $fc(i) := su(i)$

Poniżej przytaczamy twierdzenie, którego dowód można znaleźć w artykule [Cha99].

Twierdzenie 8. *Algorytm SASFC I stabilizuje system po co najwyżej $\mathcal{O}(n^2)$ ruchach, pod warunkiem, że dane jest drzewo rozpinające DFS.*

Idea różnicy symetrycznej w regule dotyczącej węzłów wewnętrznych (wiersze 5–6 algorytmu 11) jest następująca: wraz z rozprzestrzenianiem się poprawnie obliczonej wartości s (oraz su) po ścieżce od liścia do korzenia, pierwsze napotkanie danej cięciwy (dolny koniec) powoduje jej dodanie do zbioru s. Drugie (ostatnie) napotkanie tejże cięciwy (górny koniec) spowoduje usunięcie jej ze zbioru s.

Prostym przykładem wykonania algorytmu SASFC I, który pokazuje kwadratowy czas działania, jest graf, którego drzewo rozpinające DFS jest prostą ścieżką, a węzły wykonują ruchy w falach przesuwających się w górę drzewa, zaś każda fala zaczyna się o jeden poziom niżej od poprzedniej. Pierwszy ruch wykona korzeń; kolejny ruch — syn korzenia a następny — znowu korzeń, ponieważ wykonanie drugiego ruchu uaktywniło korzeń. W ten sposób ostatnia fala zaczęłaby się od jedynego liścia a skończyła w korzeniu. W sumie, n fal, z których i-ta miałaby i ruchów, dałoby $(n^2 + n)/2$ ruchów.

Usprawnienie w naszym algorytmie SASFC II polega na tym, że zmiana wartości zmiennej fc lub s nie następuje natychmiast po wykryciu jej niepoprawnego stanu (co może doprowadzić do sytuacji, jak opisana powyżej), lecz jest odkładana do momentu, w którym wszyscy potomkowie mają już poprawnie obliczone wartości tych zmiennych. Jest to możliwe dzięki zaznaczeniu danego węzła jako wymagającego przeliczenia wartości fc i s (stan β). Jednocześnie węzeł nie może powrócić bezpośrednio po obliczeniu wartości zmiennych do stanu α, ponieważ musi poczekać, aż wszystkie węzły w drzewie przeliczą poprawne wartości fc oraz s. Gdy już wszystkie węzły przeliczą wartości fc i s, każdy węzeł przyjmuje stan δ — w pierwszej kolejności korzeń a po nim jego potomkowie rekurencyjnie. Następnie, począwszy od liści aż do korzenia, wszystkie węzły przyjmują stan α.

Stan węzła reprezentuje zmienna q, której zbiór dopuszczalnych wartości jest określony następująco:

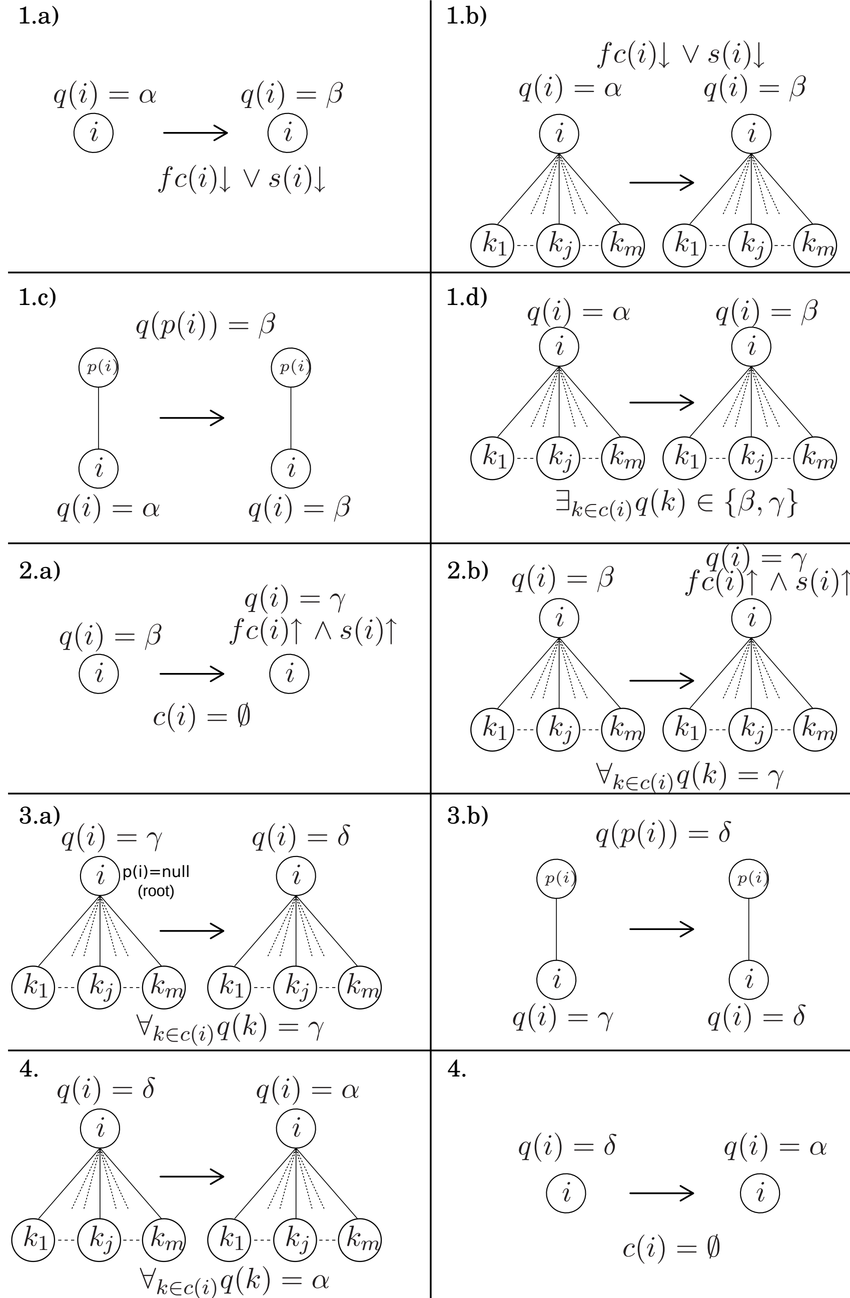
$$q(i) \in \{\alpha, \beta, \gamma, \delta\}.$$

Jak już wspomnieliśmy, zakładamy, że drzewo rozpinające DFS jest już wyznaczone i zapisane w sposób rozproszony w węzłach systemu. Według Arory i Goudy [AG93] oraz Schneidera [Sch93] stosuje się algorytmy samostabilizujące, w których stan początkowy systemu jest zawężony przez pewną określoną w specyfikacji algorytmu prekondycję. W opisywanym algorytmie będziemy przyjmować, że początkowo $q(i) = \alpha$ dla każdego węzła i systemu.

Możemy spełnić ten warunek, umieszczając zmienną q w pamięci chronionej przed modyfikacją z zewnątrz — takiej, która może być zapisywana jedynie jako wynik działania akcji umieszczonej w regule algorytmu.

Na stronie 55 przedstawiamy algorytm SASFC II (algorytm 12), którego reguły ilustrujemy rysunkiem 5.2 (s. 54). Dla uproszczenia umieszczono

na nim jedynie krawędzie należące do drzewa rozpinającego grafu (bez cięciw). Rysunek 5.3 na stronie 58 przedstawia przykład wykonania algorytmu.



Rysunek 5.2: Ilustracja reguł algorytmu SASFC II; $k \in c(i)$. Poprzez $z\downarrow$ oznaczamy lokalną niepoprawność zmiennej z.

Algorytm 12: SASFC II

1.a:
if $q(i) = \alpha \wedge c(i) = \emptyset \wedge (s(i) \neq nt(i) \vee fc(i) \neq nt(i))$ **then**
$q(i) := \beta$

1.b:
if $q(i) = \alpha \wedge c(i) \neq \emptyset \wedge (s(i) \neq nt(i) \oplus su(i) \vee fc(i) \neq su(i))$ **then**
$q(i) := \beta$

1.c:
if $q(i) = \alpha \wedge q(p(i)) = \beta$ **then**
$q(i) := \beta$

1.d:
if $q(i) = \alpha \wedge c(i) \neq \emptyset \wedge \exists_{k \in c(i)}(q(k) \in \{\beta, \gamma\})$ **then**
$q(i) := \beta$

2.a:
if $q(i) = \beta \wedge c(i) = \emptyset$ **then**
$q(i) := \gamma$
$s(i) := nt(i)$
$fc(i) := nt(i)$

2.b:
if $q(i) = \beta \wedge c(i) \neq \emptyset \wedge \forall_{k \in c(i)}(q(k) = \gamma)$ **then**
$q(i) := \gamma$
$s(i) := nt(i) \oplus su(i)$
$fc(i) := su(i)$

3.a:
if $q(i) = \gamma \wedge p(i) = null \wedge \forall_{k \in c(i)}(q(k) = \gamma)$ **then**
$q(i) := \delta$

3.b:
if $q(i) = \gamma \wedge q(p(i)) = \delta$ **then**
$q(i) := \delta$

4:
if $q(i) = \delta \wedge \forall_{k \in c(i)}(q(k) = \alpha)$ **then**
$q(i) := \alpha$

5.4 Zbieżność i złożoność

Pokażemy teraz, że system stabilizuje się po dokładnie $4n$ ruchach, jeśli wystąpiły jakiekolwiek defekty (niezależnie od ich liczby). Przez cały czas zakładamy, że po wystąpieniu defektu spełniony jest warunek $\forall_{i \in V(G)} q(i) = \alpha$ oraz wyznaczone jest drzewo rozpinające DFS.

Lemat 17. *Każdy węzeł i zmienia stan zmiennej $q(i)$ cyklicznie od stanu α, poprzez β, γ, δ, aż do początkowego α.*

Dowód. Dowód wynika z reguł algorytmu SASFC II. Wykażemy, że graf przejść między stanami zmiennej q jest prostym cyklem. Tylko reguły *1.a–1.d* zmieniają stan z α, w wyniku czego ustawiają stan β. Następnie, tylko reguły *2.a* oraz *2.b* zmieniają stan z β i ustawiają go na γ. Analogicznie reguły *3.a* i *3.b* zmieniają stan γ na δ, natomiast reguła *4.* zmienia δ na stan α. □

Lemat 18. *Jeśli węzeł i zmienia stan $q(i)$ z β na γ, wówczas wszyscy jego potomkowie mają poprawnie wyznaczone wartości zmiennych s, fc i znajdują się w stanie γ.*

Dowód. Zgodnie z regułą *2.b* węzeł wewnętrzny może zmienić swój stan z β na γ tylko wtedy, gdy wszystkie jego dzieci mają stan γ. Dla każdego jego dziecka możemy przeprowadzić podobne wnioskowanie; rekurencyjnie doprowadzi nas to do wniosku, że każdy potomek węzła i w drzewie musiał być chociaż przez chwilę w stanie γ, w momencie, gdy węzeł i zmieniał stan z α na γ.

Aby zakończyć dowód, wystarczy pokazać, że węzeł i nie może zmienić swojego stanu z γ, póki jego rodzic $p(i)$ również ma stan γ. Innymi słowy: węzeł w stanie γ jest nieaktywny, póki jego rodzic nie jest w stanie δ. Wynika to z faktu, że jedynie reguła *3.b* może zmienić stan i; jednak może być ona aktywna tylko wtedy, gdy stan $p(i)$ jest równy γ.

Z faktu, że wartości s oraz fc są wyznaczane podczas zmiany stanu na γ oraz z tego, że stan ten propaguje się od liści w górę do korzenia, wynika, że wartości s oraz fc są obliczane prawidłowo. □

Z lematu 18 wynika, że jeśli korzeń zmienia swój stan na γ, każdy inny węzeł w drzewie wyznaczył już poprawne wartości zmiennych s i fc oraz znajduje się w stanie γ.

Lemat 19. *Jeśli węzeł zmienia swój stan na δ, wtedy każdy jego przodek jest już w stanie δ.*

Dowód. Rodzic $p(i)$ węzła i musi mieć stan δ, aby węzeł i mógł zmienić stan na δ (reguła *3.b*). Powtarzając to rozumowanie indukcyjnie, dochodzimy do wniosku, że każdy przodek miał stan δ przynajmniej w przeszłości.

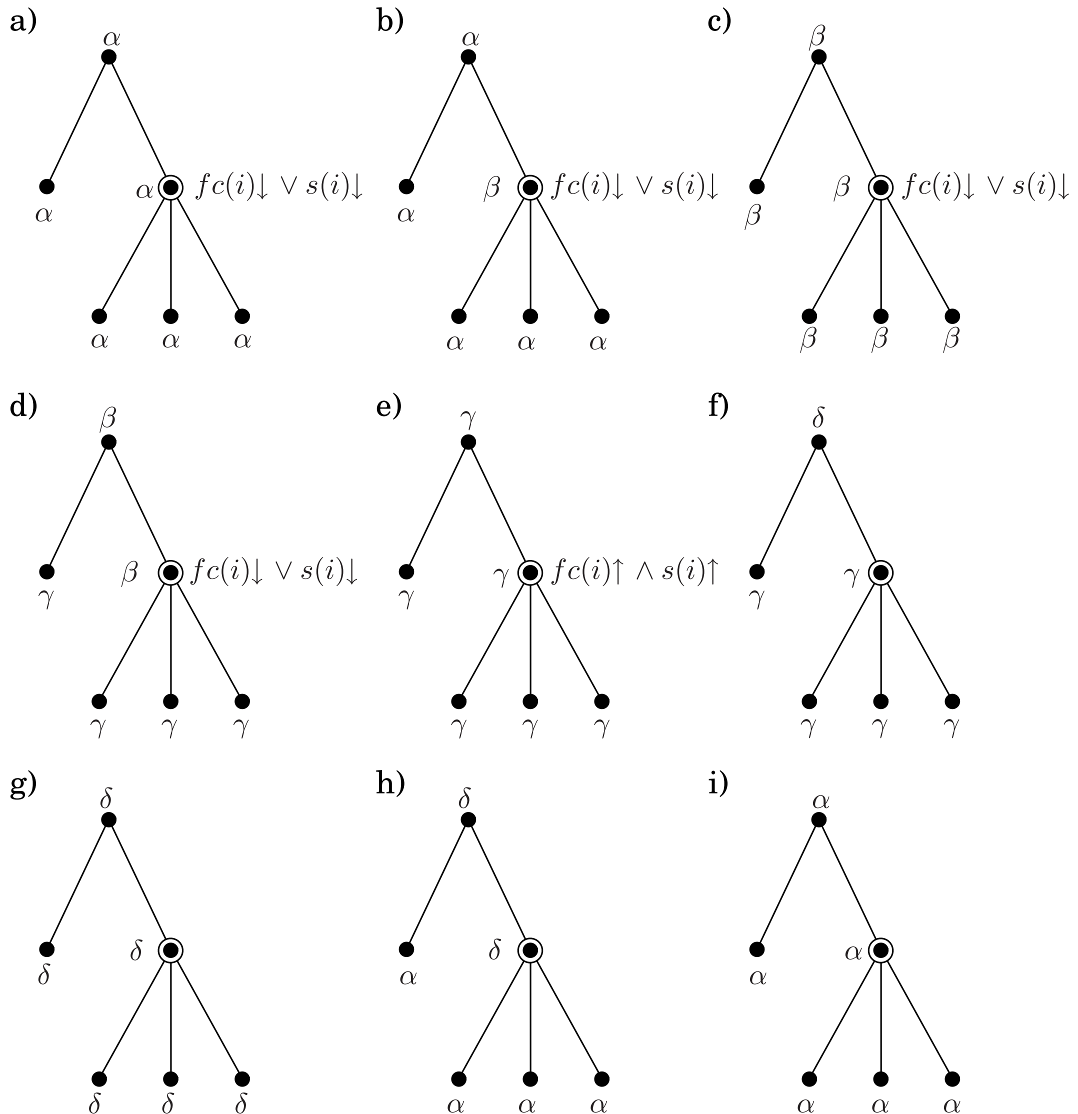
Teraz wystarczy tylko pokazać, że żaden przodek nie mógł zmienić swojego stanu (aż do momentu zmiany stanu i na δ). Aby dowolny z przodków mógł zmienić stan z δ na α, każdy z jego dzieci musi być w stanie α (reguła *4*). Znowu, powtarzając to rozumowanie indukcyjnie w dół do węzła i i biorąc pod uwagę lemat 17, dochodzimy do wniosku, że póki węzeł i nie osiągnął stanu α, wszyscy jego przodkowie mają stan δ. □

W szczególności na mocy lematu 19 liść może zmienić stan na δ tylko wtedy, gdy każdy jego przodek również ma stan δ.

Reguła *4* stanowi blokadę — węzeł musi poczekać, aż wszystkie jego dzieci zmienią stan z δ na α, aby samemu móc zmienić stan na α. Ponownie na zasadzie indukcji, własność ta zachodzi nie tylko dla dzieci, ale też wszystkich potomków danego węzła.

Węzeł — po przejściu całego cyklu zmian stanu zmiennej q — zawiera poprawne wartości zmiennych fc oraz s. Z lematów 17–19 wynika, że po wystąpieniu defektu przynajmniej w jednym węźle, każdy węzeł w systemie musi przejść cały cykl zmian stanu, przywracając poprawne wartości wszystkich swoich zmiennych. Mamy stąd następujące twierdzenie.

Twierdzenie 9. *Jeśli dane jest drzewo rozpinające DFS, każdy węzeł ma stan $q = \alpha$ i w systemie występuje dowolna liczba defektów, to algorytm ustabilizuje system po $4n$ ruchach.*



Rysunek 5.3: Przykład wykonania algorytmu SASFC II:
a) węzeł i (otoczony okręgiem) ma defekt,
b) stan i zmienia się na β (reguła *1.b*),
c) przodkowie (reguła *1.d*) i potomkowie (reguła *1.c*) (rekurencyjnie) przyjmują stan β,
d) liście przyjmują stan γ (reguła *2.a*),
e) stan γ propaguje się w górę aż do korzenia (reguła *2.b*),
f) korzeń przyjmuje stan δ (reguła *3.a*),
g) stan δ wędruje w dół aż do liści (reguła *3.b*),
h) liście przyjmują stan α (reguła *4*),
i) stan α propaguje się w górę do korzenia (reguła *4*); po tej fazie system jest ustabilizowany.

Rozdział 6

Program komputerowy

Jako wsparcie w opracowywaniu algorytmów stworzyliśmy program komputerowy [Pań16], którego zadaniem jest symulowanie działania systemów rozproszonych pod kontrolą dowolnego algorytmu samostabilizującego. Program ten można uruchamiać w środowisku tekstowym, jak również w postaci graficznego interfejsu użytkownika. Na rysunku 6.1 pokazany został widok programu w czasie symulowania algorytmu znajdującego centroid w grafie z wagami, który przedstawiliśmy w podrozdziale 4.1 (przykład systemu jak na rysunku 4.1, s. 36).

Program został napisany w języku Python przy użyciu bibliotek PyQt. W aplikacji intensywnie wykorzystywane są obiektowe właściwości języka Python. W ten sposób implementacja algorytmu samostabilizującego w celu jego przetestowania w naszym programie polega jedynie na stworzeniu klasy dziedziczącej po bazowej klasie węzła i utworzeniu w niej jednej metody na każdą regułę algorytmu. Zadaniem takiej metody jest obliczenie wartownika i ewentualnych nowych wartości zmiennych, jeśli wartownik okaże się prawdziwy.

Program wykonuje jeden ruch po naciśnięciu przycisku. Podgląd stanu systemu jest nieprzerwanie widoczny w postaci drzewka z wartościami zmiennych dla każdego węzła. Ponadto dostępna jest również historia wykonanych ruchów, gdzie widoczne jest: który węzeł wykonał ruch, zgodnie z którą regułą oraz jaka zmienna i jak zmieniła wartości. Aktualna wersja programu wykorzystuje dyspozytora, który jest niesprawiedliwy i sekwencyjny. Standardowo, jeśli reguły są jednakowe dla wszystkich węzłów, system jest jednolity, choć w łatwy sposób można je zróżnicować np. ze względu na identyfikator lub jakąkolwiek inną obliczalną własność węzła.

Kod źródłowy umieściliśmy w dodatku A. Program udostępniliśmy na wolnej licencji, zatem każdy chętny ma dostęp do jego kodu źródłowego i może go modyfikować na własne potrzeby. Jest to tym łatwiejsze, że źródła podstawowej aplikacji (bez przykładowych implementacji algorytmów) zawierają

mniej niż 600 linii kodu. Ponadto program jest napisany w zwarty sposób, zawiera komentarze dokumentujące działanie poszczególnych partii kodu.

	move	node	rule	variable	pre-value	post-value
1	1	6	1	W.node(5)	67.1596174...	1
2	2	4	1	W.node(5)	44.3958494...	135.216105...
3	3	7	1	W.node(5)	17.6232640...	10
4	4	2	1	W.node(3)	150.842317...	177.538694...
5	5	1	1	W.node(2)	168.345477...	5
6	6	2	1	W.node(3)	177.538694...	14.1932167...
7	7	2	1	W.node(1)	52.7899549...	297.692365...
8	8	3	1	W.node(2)	288.499148...	2
9	9	1	3	p	node(1)	node(2)
10	10	5	1	W.node(6)	50.7141451...	149.216105...

Rysunek 6.1: Okno programu symulatora algorytmów samostabilizujących. Zielone węzły są aktywne; duże wskazują na siebie jako na centroid. Pod rysunkiem grafu znajduje się historia wykonanych ruchów. Po prawej stronie umieszczony jest podgląd aktualnego stanu systemu (wartości zmiennych w każdym z węzłów).

Rozdział 7

Podsumowanie

W niniejszej pracy pokazaliśmy, że możliwe jest wykorzystanie specyficznych cech topologicznych sieci do konstrukcji szybkich algorytmów samostabilizujących.

W rozdziale 3 wykorzystaliśmy topologię grafów maksymalnych zewnętrznie planarnych do skonstruowania samostabilizującego algorytmu znajdującego centrum grafu o złożoności $\mathcal{O}(n^4)$. Jest to adaptacja klasycznego algorytmu Farleya i Proskurowskiego [FP80]. Idąc krok dalej, rozszerzyliśmy powyższy algorytm na sieci będące iloczynami kartezjańskimi pierwotnej klasy grafów (MOP) z kliką K_2 i ogólnie ze ścieżkami P_m.

W rozdziale 4 zajęliśmy się drzewami. W sekcji 4.1 pokazaliśmy algorytm wyznaczający ważony centroid drzewa ($\mathcal{O}(n^2)$), bazujący na algorytmie Blaira i Mannego [BM03], zaś w sekcji 4.2 zaprezentowaliśmy poprawiony algorytm (o złożoności $\mathcal{O}(n^3)$) Chaudhuriego i Thompsona [CT04].

Rozdział 5 poświęciliśmy algorytmowi wyznaczającemu zbiór cykli fundamentalnych w dowolnym grafie, przy czym algorytm ten znowu wykorzystuje specyficzną strukturę wyznaczoną jako drzewo rozpinające wgłąb danego grafu. Zaprezentowany algorytm obniża złożoność obliczeniową (w stosunku do oryginalnego algorytmu Chaudhuriego [Cha99]) do $\mathcal{O}(n)$ ruchów dzięki wykorzystaniu zaproponowanego przez nas modelu quasi-samostabilizacji.

W świetle przedstawionych przykładów możemy stwierdzić, że tezy rozprawy zostały potwierdzone. Mamy świadomość, że niektóre z zaprezentowanych algorytmów wydają się bardzo rozbudowane. Powstaje pytanie, czy możliwe byłoby zredukowanie liczby reguł zaprezentowanych algorytmów bez pogorszenia ich złożoności.

W naturalny sposób pojawiają się dalsze szczegółowe kierunki badań w postaci poszukiwania kolejnych topologii i zagadnień z teorii grafów, które można rozwiązywać w sposób efektywniejszy, niż przypadki ogólne. W szerszej perspektywie można pokusić się o badania, które odpowiedziałyby na py-

tania:

- Czy poza drzewami i ich pochodnymi istnieją inne klasy grafów, na których algorytmy samostabilizujące szybciej stabilizują system?
- Czy istnieje ogólna zależność, a jeśli tak, to jaka, między złożonością algorytmów klasycznych a samostabilizujących dla grafów o różnych topologiach?
- Dla jakich innych kryteriów wyróżniania wierzchołków w grafie (pełniących specjalną funkcję z racji położenia) można łatwo zbudować (lub też otrzymać szybki) algorytm samostabilizujący znajdujący takie wierzchołki?
- Czy model quasi-samostabilizacji ma praktyczne zastosowania w realizacjach sprzętowych?

Bibliografia

[AB97] Yehuda Afek, Anat Bremler. Self-stabilizing unidirectional network algorithms by power-supply. *SODA*, wolumen 97, strony 111–120. Citeseer, 1997.

[ACD+15] Karine Altisen, Alain Cournier, Stéphane Devismes, Anaïs Durand, Franck Petit. Self-stabilizing leader election in polynomial steps. *Stabilization, Safety, and Security of Distributed Systems*, strony 106–119. Springer, 2015.

[AG90] Anish Arora, Mohamed G. Gouda. Distributed reset. *Foundations of Software Technology and Theoretical Computer Science*, strony 316–331. Springer, 1990.

[AG93] Anish Arora, Mohamed G. Gouda. Closure and convergence: A foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19(11):1015–1027, 1993.

[AKM+93] Baruch Awerbuch, Shay Kutten, Yishay Mansour, Boaz Patt-Shamir, George Varghese. Time optimal self-stabilizing synchronization. *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, strony 652–661. ACM, 1993.

[AKY90] Yehuda Afek, Shay Kutten, Moti Yung. Memory-efficient self stabilizing protocols for general networks. *Distributed Algorithms*, strony 15–28. Springer, 1990.

[APR15] John Augustine, Gopal Pandurangan, Peter Robinson. Fast Byzantine Leader Election in Dynamic Networks. Yoram Moses, redaktor, *DISC*, wolumen 9363 serii *Lecture Notes in Computer Science*, strony 276–291. Springer, 2015.

[AS96] Gheorghe Antonoiu, Pradip K. Srimani. A Self-Stabilizing Leader Election Algorithm for Tree Graphs. *Journal of Parallel and Distributed Computing*, 34(2):227–232, 1996.

[Ben06] Mordechai Ben-Ari. *Principles of Concurrent and Distributed Programming.* Prentice Hall international series in computer science. Addison-Wesley, 2006.

[Ben12] Mordechai Ben-Ari. *Mathematical Logic for Computer Science, 3rd Edition.* Springer, 2012.

[BGK98] Joffroy Beauquier, Christophe Genolini, Shay Kutten. k-stabilization of reactive tasks. *Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*, strona 318. ACM, 1998.

[BGM89] James E. Burns, Mohamed G. Gouda, Raymond E. Miller. On relaxing interleaving assumptions. *Proceedings of the MCC Workshop on Self-Stabilizing Systems, MCC Technical Report No. STP-379-89*, 1989.

[BGM93] James E. Burns, Mohamed G. Gouda, Raymond E. Miller. Stabilization and pseudo-stabilization. *Distributed Computing*, 7(1):35–42, 1993.

[BK07] Janna Burman, Shay Kutten. Time Optimal Asynchronous Self-stabilizing Spanning Tree. Andrzej Pelc, redaktor, *DISC*, wolumen 4731 serii *Lecture Notes in Computer Science*, strony 92–107. Springer, 2007.

[BM03] Jean R. S. Blair, Fredrik Manne. Efficient Self-stabilizing Algorithms for Tree Network. *23rd International Conference on Distributed Computing Systems (ICDCS 2003), 19-22 May 2003, Providence, RI, USA*. IEEE Computer Society, 2003.

[BMS81] Fred Buckley, Zevi Miller, Peter J. Slater. On graphs containing a given graph as center. *Journal of Graph Theory*, 5(4):427–434, 1981.

[Bol98] Béla Bollobás. *Modern Graph Theory.* Graduate texts in mathematics. Springer, Heidelberg, wydanie corrected, 1998.

[BP12] Halina Bielak, Michał Pańczyk. A self-stabilizing algorithm for finding weighted centroid in trees. *Annales UMCS, Informatica*, 12(2):27–37, 2012.

[BP13] Halina Bielak, Michał Pańczyk. A quasi self-stabilizing algorithm for detecting fundamental cycles in a graph with DFS spanning

tree given. *2013 Federated Conference on Computer Science and Information Systems (FedCSIS)*, strony 293–297. IEEE, 2013.

[BP14] Halina Bielak, Michał Pańczyk. A self-stabilizing algorithm for locating the center of cartesian product of K_2 and maximal outerplanar graphs. *Position Papers of the 2014 Federated Conference on Computer Science and Information Systems*, strony 93–99, 2014.

[CD94] Zeev Collin, Shlomi Dolev. Self-stabilizing depth-first search. *Information Processing Letters*, 49(6):297–301, 1994.

[Cha99] Pranay Chaudhuri. A self-stabilizing algorithm for detecting fundamental cycles in a graph. *Journal of Computer and System Sciences*, 59(1):84–93, 1999.

[CT04] Pranay Chaudhuri, Hussein Thompson. A self-stabilizing graph algorithm: finding the cutting center of a tree. *International Journal of Computer Mathematics*, 81(2):183–190, 2004.

[DB08] Abdelouahid Derhab, Nadjib Badache. A self-stabilizing leader election algorithm in highly dynamic ad hoc mobile networks. *Parallel and Distributed Systems, IEEE Transactions on*, 19(7):926–939, 2008.

[DH95] Shlomi Dolev, Ted Herman. Superstabilizing protocols for dynamic distributed systems. *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, strona 255. ACM, 1995.

[Die05] Reinhard Diestel. *Graph Theory (Graduate Texts in Mathematics)*. Springer, August 2005.

[Dij74] Edsger W. Dijkstra. Self-stabilizing Systems in Spite of Distributed Control. *Communications of the ACM*, 17(11):643–644, 1974.

[DIM93] Shlomi Dolev, Amos Israeli, Shlomo Moran. Self-Stabilization of Dynamic Systems Assuming Only Read/Write Atomicity. *Distributed Computing*, 7(1):3–16, 1993.

[DIM97] Shlomi Dolev, Amos Israeli, Shlomo Moran. Uniform dynamic self-stabilizing leader election. *IEEE Transactions on Parallel and Distributed Systems*, 8(4):424–440, 1997.

[DLP10] Ajoy Kumar Datta, Lawrence L. Larmore, Hema Piniganti. Self-stabilizing Leader Election in Dynamic Networks. Shlomi Dolev, Jorge Arturo Cobb, Michael J. Fischer, Moti Yung, redaktorzy, *SSS*, wolumen 6366 serii *Lecture Notes in Computer Science*, strony 35–49. Springer, 2010.

[DLV08] Ajoy Kumar Datta, Lawrence L. Larmore, Priyanka Vemula. Self-Stabilizing Leader Election in Optimal Space. Sandeep S. Kulkarni, André Schiper, redaktorzy, *SSS*, wolumen 5340 serii *Lecture Notes in Computer Science*, strony 109–123. Springer, 2008.

[DLV11a] Ajoy Kumar Datta, Lawrence L. Larmore, Priyanka Vemula. An $\mathcal{O}(n)$-time self-stabilizing leader election algorithm. *J. Parallel Distrib. Comput.*, 71(11):1532–1544, 2011.

[DLV11b] Ajoy Kumar Datta, Lawrence L. Larmore, Priyanka Vemula. Self-stabilizing leader election in optimal space under an arbitrary scheduler. *Theoretical Computer Science*, 412(40):5541–5561, 2011.

[Dol00] Shlomi Dolev. *Self-stabilization*. MIT press, 2000.

[DTY15] Stéphane Devismes, Sébastien Tixeuil, Masafumi Yamashita. Weak vs. Self vs. Probabilistic Stabilization. *International Journal of Foundations of Computer Science*, 26(3):293–320, 2015.

[FGH74] Herbert J. Fleischner, Dennis P. Geller, Frank Harary. Outerplanar graphs and weak duals. *Journal of the Indian Mathematical Society*, 38:215–219, 1974.

[FJ01] Faith E. Fich, Colette Johnen. A Space Optimal, Deterministic, Self-Stabilizing, Leader Election Algorithm for Unidirectional Rings. Jennifer L. Welch, redaktor, *DISC*, wolumen 2180 serii *Lecture Notes in Computer Science*, strony 224–239. Springer, 2001.

[FP80] Arthur M. Farley, Andrzej Proskurowski. Computation of the center and diameter of outerplanar graphs. *Discrete Applied Mathematics*, 2(3):185–191, 1980.

[FZAM08] Jose Alberto Fernandez-Zepeda, Juan Paulo Alvarado-Magana. Analysis of the average execution time for a self-stabilizing leader election algorithm. *International Journal of Foundations of Computer Science*, 19(06):1387–1402, 2008.

[Gou01] Mohamed G. Gouda. The theory of weak stabilization. *Self-Stabilizing Systems*, strony 114–123. Springer, 2001.

[GP03] Felix C. Gärtner, Henning Pagnia. Time-efficient self-stabilizing algorithms through hierarchical structures. *Self-Stabilizing Systems*, strony 154–168. Springer, 2003.

[GT02] Christophe Genolini, Sébastien Tixeuil. A lower bound on dynamic k-stabilization in asynchronous systems. *Reliable Distributed Systems, 2002. Proceedings. 21st IEEE Symposium on*, strony 212–221. IEEE, 2002.

[Har69] Frank Harary. *Graph theory.* Addison Wesley, 1969.

[HC92] Shing-Tsaan Huang, Nian-Shing Chen. A Self-Stabilizing Algorithm for Constructing Breadth-First Trees. *Information Processing Letters*, 41(2):109–117, 1992.

[HO71] Frank Harary, Phillip A. Ostrand. The cutting center theorem for trees. *Discrete Mathematics*, 1(1):7–18, 1971.

[HS86] Frank Harary, Peter J. Slater. A Linear Algorithm for the Cutting Center of a Tree. *Information Processing Letters*, 23(5):317–319, 1986.

[IJ90] Amos Israeli, Marc Jalfon. Token management schemes and random walks yield self-stabilizing mutual exclusion. *Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, strony 119–131. ACM, 1990.

[ISWW09] Rebecca Ingram, Patrick Shields, Jennifer E. Walter, Jennifer L. Welch. An asynchronous leader election algorithm for dynamic networks. *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, strony 1–12. IEEE, 2009.

[Jor69] Camille Jordan. Sur les assemblages de lignes. *Journal für die reine und angewandte Mathematik*, 70(185):81, 1869.

[KK05] Adrian Kosowski, Łukasz Kuszner. A Self-stabilizing Algorithm for Finding a Spanning Tree in a Polynomial Number of Moves. Roman Wyrzykowski, Jack Dongarra, Norbert Meyer, Jerzy Wasniewski, redaktorzy, *PPAM*, wolumen 3911 serii *Lecture Notes in Computer Science*, strony 75–82. Springer, 2005.

[KK13] Alex Kravchik, Shay Kutten. Time Optimal Synchronous Self Stabilizing Spanning Tree. Yehuda Afek, redaktor, *DISC*, wolumen 8205 serii *Lecture Notes in Computer Science*, strony 91–105. Springer, 2013.

[KT77] Г. Н. Копылов, Е. А. Тимофеев. О центрах и радиусах графов. *Успехи математических наук*, 32(6 (198)):226–226, 1977.

[Lam85] Leslie Lamport. Solved Problems, Unsolved Problems and Non-Problems in Concurrency. *Operating Systems Review*, 19(4):34–44, 1985.

[Moo65] Gordon E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8):114–117, 1965.

[Pań16] Michał Pańczyk. Selfstabilising Systems Simulator. `https://github.com/mpanczyk/SelfStabSimulator`, 2015–2016.

[Pat14] Stacy Patterson. In-Network Leader Selection for Acyclic Graphs. *Computer Research Laboratory*, abs/1410.6533, 2014.

[PB14] Michał Pańczyk, Halina Bielak. A self-stabilizing algorithm for locating the center of maximal outerplanar graphs. *Journal of Universal Computer Science*, 20(14):1951–1963, 2014.

[Pro80] Andrzej Proskurowski. Centers of maximal outerplanar graphs. *Journal of Graph Theory*, 4(1):75–79, 1980.

[Sch93] Marco Schneider. Self-stabilization. *ACM Computing Surveys (CSUR)*, 25(1):45–67, 1993.

[SS92] Sumit Sur, Pradip K. Srimani. A Self-Stabilizing Distributed Algorithm to Construct BFS Spanning Trees of a Symmetric Graph. *Parallel Processing Letters*, 2:171–179, 1992.

[Sys79] Maciej M. Sysło. Characterizations of outerplanar graphs. *Discrete Mathematics*, 26(1):47–53, 1979.

[Var13] Carlos A. Varela. *Programming Distributed Computing Systems: A Foundational Approach*. MIT Press, 2013.

[Wal16] M. Mitchell Waldrop. The chips are down for Moore's law. *Nature*, 530(7589):144–147, 2016.

[XS06] Zhenyu Xu, Pradip K. Srimani. Self-stabilizing Anonymous Leader Election in a Tree. *International Journal of Foundations of Computer Science*, 17(2):323–336, 2006.

Dodatek A

Kod programu

Plik basenode.py:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import uuid
from functools import (
  total_ordering,
)
from PyQt5.QtCore import (
  QPoint,
  Qt,
)
from PyQt5.QtGui import (
  QBrush,
  QColor,
)

from randomtypes import (
  RandomInt,
  RandomBool,
)
import utils

@total_ordering
class BaseNode(object):
  '''Abstract class representing a node in a self-stabilising system.

```

```
27          Its subclasses ought to implement a self-stabilising algorithm
28          by implementing methods 'rule_*', for example 'rule_1'.
29
30          A 'rule_x' should return a pair:
31           - bool - True if the guard of the rule evaluates to True,
32           - dict - dictionary with variable names as keys
33                    and their new values as the dict values.
34        '''
35
36        def __init__(self, **kwargs):
37
38          self.id = uuid.uuid4()
39          self.net = None
40          self._r = 10
41
42          self.neighbours = set()
43          for neighbour in kwargs.pop('neighbours', ()):
44            self.connect(neighbour)
45
46          self.variables = set(('id',))
47          for var, val in kwargs.items():
48            self.__setattr__(var, val)
49            self.variables.add(var)
50          for var, type_ in self.__class__.variables.items():
51            if var not in self.__dict__:
52              self.__setattr__(var, type_())
53              self.variables.add(var)
54
55        def __repr__(self):
56          return '{}({})'.format(self.__class__.__name__, self.id)
57
58        def __str__(self):
59          return 'node({})'.format(self.id)
60
61        def __le__(self, other):
62          return self.id <= other.id
63
64        def connect(self, other):
65          self.neighbours.add(other)
66          other.neighbours.add(self)
67          if self.net is None:
```

```
68        self.net = other.net
69      if other.net is None:
70        other.net = self.net
71      assert self.net is other.net
72      if self.net is not None:
73        self.net.nodes.add(self)
74        self.net.nodes.add(other)
75
76    def disconnect(self, other):
77      if other in self.neighbours:
78        self.neighbours.remove(other)
79      if self in other.neighbours:
80        other.neighbours.remove(self)
81
82    def is_active(self):
83      '''Check if there exists any active rule in the node.
84      '''
85      return bool(self.get_active_rules())
86
87    @classmethod
88    def get_rule_names(klass):
89      names = [
90        method_name[5:]
91        for method_name
92        in dir(klass)
93        if method_name.startswith('rule_')
94      ]
95      if not names:
96        raise NotImplementedError(
97          'You must implement at least one rule in the {} class.'.format(
98            klass.__name__,
99          )
100       )
101     return names
102
103   def get_rule(self, name):
104     return self.__getattribute__('rule_' + name)
105
106   def get_active_rules(self):
107     return [
108       name
```

```
109         for name in self.get_rule_names()
110         if self.get_rule(name)()[0]
111       ]
112
113   def assign(self, **kwargs):
114     '''Assign new values to the node's variables.
115        This way a node may change its state.
116     '''
117     self.__dict__.update(kwargs)
118
119   def move(self):
120     '''Pick up a random active rule in the node
121        and make a move according to its assignment.
122     '''
123     active_rule_names = self.get_active_rules()
124     assert active_rule_names, 'There is no active rule in the node.'
125     rule_name = utils.random_pick(active_rule_names)
126     is_active, variables = self.get_rule(rule_name)()
127     assert is_active
128     self.assign(**variables)
129     return rule_name
130
131   def get_state(self):
132     return {
133       var: self.__getattribute__(var)
134       for var in self.variables
135     }
136
137   def get_random_neighbour(self):
138     return utils.random_pick(list(self.neighbours))
139
140   def centre(self):
141     return QPoint(self._x, self._y)
142
143   def get_color(self):
144     if self.is_active():
145       return QColor(0x1f, 0xff, 0x1f)
146     return QColor(0x8f, 0x8f, 0x8f)
147
148   def get_radius(self):
149     return self._r
```

```
150
151   def set_radius(self, r):
152     self._r = r
153
154   radius = property(get_radius, set_radius)
155
156   def draw(self, painter):
157     painter.setBrush(QBrush(self.get_color()))
158     painter.drawEllipse(
159       self.centre(),
160       self.radius,
161       self.radius,
162     )
163     painter.drawText(
164       self._x-self.radius,
165       self._y-self.radius,
166       2*self.radius,
167       2*self.radius,
168       Qt.AlignCenter | Qt.AlignVCenter,
169       str(self.id)
170     )
```

Plik basenetwork.py:

```
1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3
4  import random
5  import copy
6  from PyQt5.QtCore import (
7    QRect,
8  )
9
10 import utils
11
12 class BaseNetwork(object):
13
14   def __init__(self):
15     self.nodes = set()
16
17   def get_active_nodes(self):
```

```
18        return [node for node in self.nodes if node.is_active()]
19
20      def is_stabilised(self):
21        return not self.get_active_nodes()
22
23      def __iadd__(self, node):
24        self.nodes.add(node)
25        node.net = self
26        return self
27
28      def connect(self, node1, node2):
29        for node in node1, node2:
30          self.nodes.add(node)
31        node1.connect(node2)
32
33      def move(self):
34        node = utils.random_pick( self.get_active_nodes() )
35        info = {'prestate': copy.copy(node)}
36        rule = node.move()
37        info['rule'] = rule
38        info['poststate'] = copy.copy(node)
39        return info
40
41      def run(self):
42        while not self.is_stabilised():
43          yield self.move()
44
45      def draw_edge(self, painter, edge):
46        node1, node2 = edge
47        painter.drawLine(node1.centre(), node2.centre())
48
49      def draw(self, painter):
50
51        # Edges drawing
52        edges = {
53          frozenset((node, neighbour))
54          for node in self.nodes
55          for neighbour in node.neighbours
56        }
57        for edge in edges:
58          self.draw_edge(painter, edge)
```

```
59
60        # Nodes drawing
61        for node in self.nodes:
62          node.draw(painter)
63
64      def boundingBox(self):
65        margin = 25
66        mini_x = min(node._x for node in self.nodes)
67        maxi_x = max(node._x for node in self.nodes)
68        mini_y = min(node._y for node in self.nodes)
69        maxi_y = max(node._y for node in self.nodes)
70        width = maxi_x - mini_x
71        height = maxi_y - mini_y
72        return QRect(
73          mini_x-margin,
74          mini_y-margin,
75          2*margin+width,
76          2*margin+height
77        )
78
79      def adjustedBoundingBox(self, painter):
80        boundingBox = self.boundingBox()
81        bbRatio = boundingBox.height() / boundingBox.width()
82        viewPort = painter.viewport()
83        vpRatio = viewPort.height() / viewPort.width()
84        if vpRatio > bbRatio:
85          diff = boundingBox.width() * (vpRatio-bbRatio)
86          boundingBox.adjust( 0, -diff/2, 0, diff/2)
87        elif vpRatio < bbRatio:
88          diff = boundingBox.height() * (1/vpRatio-1/bbRatio)
89          boundingBox.adjust( -diff/2, 0, diff/2, 0 )
90        return boundingBox
```

Plik utils.py:

```
1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3
4  import random
5  import copy
```

```

def random_pick(sequence):
  '''Choose one rule amongst the active ones.
  '''
  return random.choice(sequence)

def updated_dict(d, *args):
  '''Return copy of a nested dictionary d
  with keys args[:-1] updated with value args[-1].
  For example: updated_dict(d, 'key1', 'key2', 'value')
  will return copy of d with d['key1']['key2']=='value'.
  '''
  if len(args) == 1:
    return args[0]
  d = copy.copy(d)
  d[args[0]] = updated_dict(d[args[0]], *args[1:])
  return d

def diff_dict(d0, d1):
  '''Return diff between two dictionaries
  in form of a dict with keys only for different values.
  The differences are inspected recursively
  (for values being another dictionaries)
  and respectively returned.
  '''
  diff = {}
  for key, value0 in d0.items():
    value1 = d1[key]
    if isinstance(value0, dict):
      diff_rek = diff_dict(value0, value1)
      if diff_rek:
        diff[key] = diff_rek
    elif value0 != value1:
      diff[key] = (value0, value1)
  return diff

def formated_diff(diff):
  '''Iterates over the diff_dict items
  and yields pairs:
    - array of keys with different value,
    - pair with former and actual value.
```

```
47    '''
48    for key, value in diff.items():
49      if isinstance(value, dict):
50        for rek_key, rek_value in formated_diff(value):
51          yield ((key, rek_key), rek_value)
52      else:
53        yield key, value
```

Plik randomtypes.py:

```
1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3
4  import random
5
6  MAX_RAND_INT = 2**31
7  MAX_RAND_FLOAT = 300.0
8
9  class RandomType(object):
10   pass
11
12 def RandomInt(upper_bound=MAX_RAND_INT):
13   class ClassRandomInt(int, RandomType):
14     def __new__(T):
15       value = random.randint(0, upper_bound)
16       return int.__new__(T, value)
17   ClassRandomInt.__name__ = 'RandomInt'
18   return ClassRandomInt
19
20 def RandomBool():
21   class ClassRandomBool(int, RandomType):
22     def __new__(T):
23       return int.__new__(T, random.randint(0, 1))
24     def __repr__(self):
25       return 'True' if self else 'False'
26   ClassRandomBool.__name__ = 'RandomBool'
27   return ClassRandomBool
28
29 def RandomFloat(upper_bound=MAX_RAND_FLOAT):
30   class ClassRandomFloat(float, RandomType):
```

```
31      def __new__(T):
32        value = random.random()*upper_bound
33        return float.__new__(T, value)
34    ClassRandomFloat.__name__ = 'RandomFloat'
35    return ClassRandomFloat
```

Plik app.py:

```
1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3
4  import sys
5  from PyQt5.QtWidgets import (
6    QApplication,
7    QWidget,
8    QPushButton,
9    QVBoxLayout,
10   QHBoxLayout,
11 )
12 from PyQt5.QtGui import (
13   QPainter,
14   QBrush,
15   QColor,
16   QStandardItemModel,
17   QStandardItem,
18   QFont,
19 )
20 from PyQt5.QtCore import (
21   Qt,
22 )
23 from PyQt5.QtWidgets import (
24   QTreeView,
25   QTableView,
26 )
27
28 from utils import (
29   diff_dict,
30   formated_diff,
31 )
32 from basenode import (
33   BaseNode,
```

```
34  )
35
36  def typeValueItem(value):
37    typeFont = QFont()
38    typeFont.setItalic(True)
39    textList = [value.__class__.__name__]
40    if hasattr(value, '__iter__'):
41      if value:
42        textList.append('({} items)'.format(str(len(value))))
43      else:
44        textList.append('(empty)')
45    valueItem = QStandardItem(
46      ' '.join(textList)
47    )
48    valueItem.setFont(typeFont)
49    return valueItem
50
51  def scalarValueItem(value):
52    valueFont = QFont()
53    valueFont.setBold(True)
54    valueItem = QStandardItem(
55      '{} (type: {})'.format(
56        str(value),
57        value.__class__.__name__,
58      )
59    )
60    valueItem.setFont(valueFont)
61    return valueItem
62
63  def getItems(obj, level=0):
64    exclude_keys = ('net', '_r', 'variables')
65    retVal = []
66    if level > 1 and isinstance(obj, BaseNode):
67      return retVal
68    if isinstance(obj, dict):
69      for key, value in sorted(obj.items()):
70        if key not in exclude_keys\
71          and not (
72            isinstance(key, str)\
73            and key.startswith('_')
74          ):
```

```
75          keyItem = QStandardItem(str(key))
76          children = []
77          if hasattr(value, '__iter__'):
78            valueItem = typeValueItem(value)
79            for child in getItems(value, level+1):
80              keyItem.appendRow(child)
81          else:
82            valueItem = scalarValueItem(value)
83          retVal.append([keyItem, valueItem])
84    elif hasattr(obj, '__dict__'):
85      retVal = getItems(obj.__dict__, level+1)
86    elif isinstance(obj, (set, list, tuple)):
87      for item in sorted(obj):
88        keyItem = QStandardItem(str(item))
89        children = getItems(item, level+1)
90        valueItem = typeValueItem(item)
91        for child in children:
92          keyItem.appendRow(child)
93        retVal.append([keyItem, valueItem])
94    return retVal
95
96  class MovesView(QTableView, object):
97    def __init__(self, parent=None):
98      super(MovesView, self).__init__(parent)
99      self.setModel(QStandardItemModel(self))
100     self.model().setHorizontalHeaderLabels([
101       'move',
102       'node',
103       'rule',
104       'variable',
105       'pre-value',
106       'post-value',
107     ])
108     self.moveNo = 1
109
110   def addRow(self, status):
111     backgrounds = (
112       QBrush(QColor(0xef, 0xef, 0xef)),
113       QBrush(QColor(0xff, 0xff, 0xff)),
114     )
115     for keys, values in formated_diff(
```

```
116         diff_dict(
117           status['prestate'].get_state(),
118           status['poststate'].get_state(),
119         )
120       ):
121         items = [QStandardItem(str(x)) for x in (
122           self.moveNo,
123           status['prestate'].id,
124           status['rule'],
125           '.'.join(str(key) for key in keys),
126           values[0],
127           values[1],
128         )]
129         for item in items:
130           item.setBackground(
131             backgrounds[self.moveNo % 2]
132           )
133         self.model().appendRow(items)
134       self.moveNo += 1
135       self.resizeColumnsToContents()
136       self.scrollToBottom()
137 
138 class VariablesView(QTreeView, object):
139   def __init__(self, network, parent=None):
140     super(VariablesView, self).__init__(parent)
141     self.network = network
142     self.redraw()
143 
144   def redraw(self):
145     self.setModel(QStandardItemModel(self))
146     for row in getItems(self.network):
147       self.model().appendRow(row)
148     self.expandAll()
149     self.resizeColumnToContents(0)
150     self.resizeColumnToContents(1)
151 
152 class DrawArea(QWidget, object):
153 
154   def __init__(self, network, parent=None):
155     super(DrawArea, self).__init__(parent)
156     self.network = network
```

```
157
158   def redraw(self):
159     self.update()
160
161   def paintEvent(self, event):
162     painter = QPainter(self)
163     painter.fillRect(
164       painter.window(),
165       QBrush(Qt.white),
166     )
167     painter.setRenderHint(QPainter.Antialiasing)
168     painter.setWindow(
169       self.network.adjustedBoundingBox(painter)
170     )
171     self.network.draw( painter )
172
173
174 class MainWindow(QWidget, object):
175
176   def __init__(self, network, parent=None):
177
178     super(MainWindow, self).__init__(parent)
179
180     self.setWindowTitle('Selfstabilising Systems Simulator')
181
182     self.move_button = QPushButton("Move", self)
183     self.move_button.clicked.connect(self.make_move)
184
185     self.network = network
186     self.drawArea = DrawArea(self.network, self)
187     self.variablesView = VariablesView(self.network.nodes, self)
188     self.movesView = MovesView(self)
189
190     self.mainLayout = QHBoxLayout(self)
191     self.leftLayout = QVBoxLayout()
192     self.leftLayout.addWidget(self.drawArea, 1)
193     self.leftLayout.addWidget(self.movesView, 1)
194     self.rightLayout = QVBoxLayout()
195     self.rightLayout.addWidget(self.move_button)
196     self.rightLayout.addWidget(self.variablesView)
197
```

```
198      self.mainLayout.addLayout(self.leftLayout, 3)
199      self.mainLayout.addLayout(self.rightLayout, 3)
200
201    def make_move(self):
202      status = self.network.move()
203      self.movesView.addRow(status)
204      if self.network.is_stabilised():
205        self.move_button.setEnabled( False )
206      self.drawArea.redraw()
207      self.variablesView.redraw()
208
209
210  def main():
211    app = QApplication(sys.argv)
212    import centroid
213    network = centroid.test_network()
214    w = MainWindow(network)
215    w.show()
216    return app.exec()
217
218  if __name__ == '__main__':
219    main()
```