Lekcja 7: Grafy

Wstęp

Po dwóch lekcjach omawiających struktury drzewiaste dotarliśmy do momentu, gdy możemy już pokazać strukturę, która jest uogólnieniem drzew. Mamy na myśli grafy, dzieki którym umiemy modelować najrozmaitsze sieci połączeń, tak bardzo popularne w wielu dziedzinach codziennego życia. W tej lekcji dowiecie się o strukturze i algorytmach grafowych, które pozwolą wyszukać najkrótsze połączenia między węzłami sieci i połączyć wszystkie węzły w sposób optymalny.

Zaś w przedostatnim rozdziale proponujemy coś extra - algorytm, którego używają twórcy gier komputerowych.

Wprowadzenie do grafów

Grafy znamy dobrze z matematyki. Okazuje się, że bardzo często są one wykorzystywane w informatyce. Wystepuje szereg zagadnień i problemów grafowych, których efektywne rozwiązania zachęcają do użycia ich w wielu problemach obliczeniowych. W tym rozdziale dowiemy się o trzech zasadniczych grupach problemów grafowych:

- przeszukiwaniu grafu
- wyznaczaniu najkrótszych ścieżek między wierzchołkami grafu
- wyznaczaniu minimalnego drzewa rozpinającego w grafie

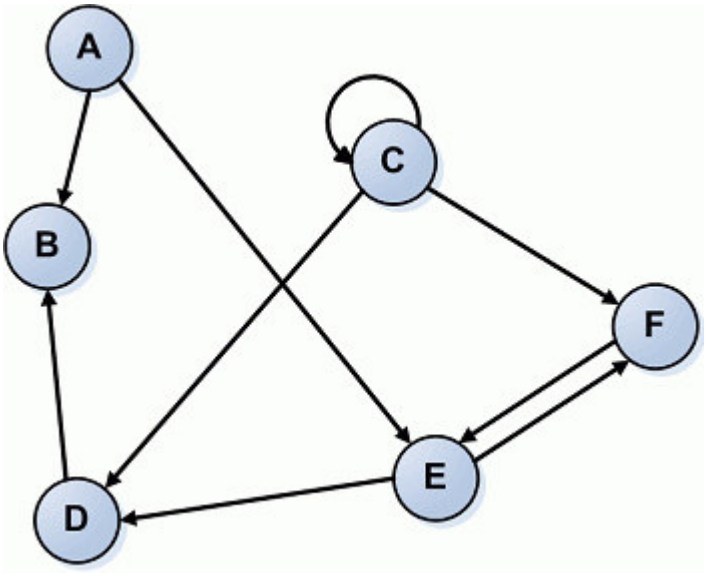
Zacznijmy od podstaw. Zazwyczaj graf przedstawia się za pomocą symboli **G = (V, E)**, gdzie **V** jest to **zbiór wierzchołków** grafu, zwanych również **węzłami** (ang. *vertex*), natomiast **E** to **zbiór krawędzi** grafu (ang. *edge*). Dlatego my również będziemy przede wszystkim używali tych oznaczeń. Krawędzie mają przy tym swoją wagę (koszt przejścia), który najczęściej wyobrażamy sobie jako wartość odległości lub czasu przejścia pomiędzy węzłami, lecz interpretacje są tu nieograniczone.

Grafy dzielimy zasadniczo na dwa rodzaje – grafy skierowane oraz nieskierowane. **Graf skierowany** posiada krawędzie, dla których jest określony jeden kierunek. Dzięki konkretnej krawędzi można przejść z jednego wierzchołka grafu do drugiego, ale nie na odwrót. Oczywiście nic nie stoi na przeszkodzie, by między dwoma węzłami grafu istniało kilka krawędzi skierowanych w przeciwne strony i z różnymi kosztami przejścia. Przykładem jest na przykład schemat sieci szlaków turystycznych w terenie górskim, gdzie czas przejścia od podnóża góry na szczyt jest znacznie dłuższy niż powrotne zejście z niej (może być również kilka różnych dróg wejściowych/ zejściowych o różnej skali trudności).

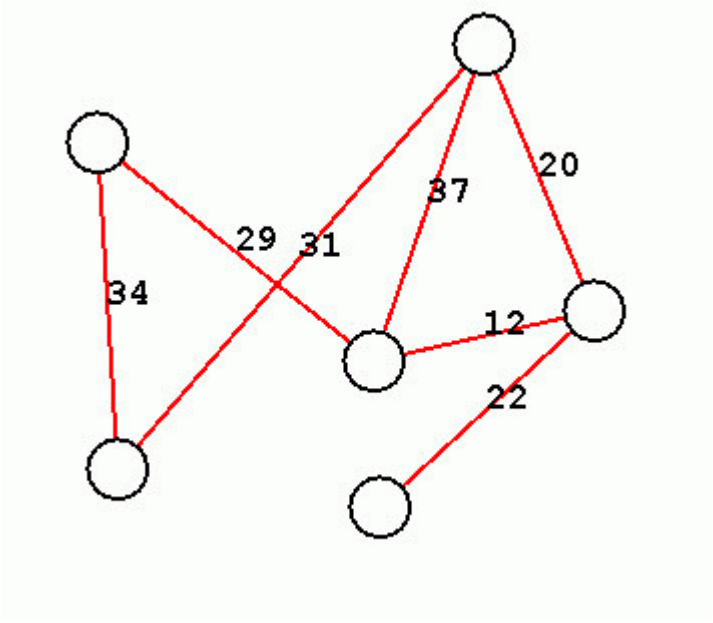
Graf nieskierowany zawiera natomiast krawędzie, które są niejako dwukierunkowe. Po krawędzi łączącej wierzchołki u i v możemy przejść zarówno z węzła u do v, jak i w przeciwną stronę, z wierzchołka v do u. Tak naprawdę zbiór takich grafów jest podzbiorem grafów skierowanych. Stosunkowo łatwo to sobie wyobrazić – każda krawędź w grafie nieskierowanym może być zastąpiona przez dwie krawędzie skierowane o tym samym koszcie łączące te same wierzchołki – wówczas otrzymamy graf skierowany o dokładnie tych samych możliwościach przejścia między węzłami, co początkowy graf nieskierowany. Zatem każdy graf nieskierowany można zamienić na skierowany, ale nie na odwrót. Oczywiście zarówno grafy nieskierowane, jak i zorientowane (skierowane) mogą być grafami **ważonymi**.

Poniżej przedstawiamy po jednym przykładzie grafu skierowanego i nieskierowanego; ten pierwszy nie ma zaznaczonych wag krawędzi, co oznacza, że wszystkie one są jednakowe, więc nie mają znaczenia; w grafie drugim wartości wag sa wpisane przy krawędziach.

- graf skierowany



- graf nieskierowany



Tak jak jest to przedstawione na rysunkach, krawędzie w grafie skierowanym najczęściej występują w postaci strzałek, natomiast w grafie nieskierowanym – w postaci linii prostych bez żadnych grotów.

Listy sąsiedztwa i macierz sąsiedztwa

Kolejną ważna rzeczą, jaką należy koniecznie omówić, jest sposób reprezentacji grafu w pamięci komputera. Może być on zapisany:

- w postaci listy sąsiedztwa
- za pomocą macierzy sąsiedztwa

Pierwsze podejście – czyli tworzenie listy sąsiedztwa – polega na utworzeniu dla każdego z wierzchołka grafu listy dynamicznej zawierającej wierzchołki sąsiadujące z tym węzłem (bezpośrednio za pomocą krawędzi). Ważne jest to, że wierzchołki umieszczone w tej liście nie muszą być w żaden sposób posortowane. Natomiast tworzenie macierzy sąsiedztwa dla grafu zakłada, że utworzymy w pamięci macierz kwadratową o rozmiarze N x N, gdzie N jest licznością zbioru V. W takiej tablicy zapisuje się wartości logiczne (np. przy użyciu liczb 0 lub 1) określające, czy istnieje krawędź z wierzchołka i-tego do j-tego. Jeśli graf zawiera taką krawędź, to T[i,j] =1, gdzie T jest macierzą (tablicą) sąsiedztwa.

Wróćmy do przykładowego grafu skierowanego zamieszczonego na rysunku. Warto zastanowić się, jak należałoby przedstawić ten graf w formie listy sąsiedztwa. Taka lista może występować w dwóch wersjach:

- z listą węzłów, do których prowadzą krawędzie wychodzące z danego wierzchołka

- z listą węzłów, z których prowadzą krawędzie przychodzące do danego wierzchołka

Obie listy sąsiedztwa wyglądałyby następująco:

Lista węzłów wychodzących

A	B, E
B	NULL
C	C, D, F
D	B
E	D, F
F	E

Lista węzłów przychodzących

A	NULL
B	A, D
C	C
D	C, E
E	A, F
F	C, E

Obie listy sąsiedztwa są równorzędne i można ich używać zamiennie, aczkolwiek częściej stosowana, bardziej naturalna jest lista zawierająca krawędzie wychodzące. Reprezentacja grafu za pomocą list sąsiedztwa jest szczególnie przydatna w przypadku, gdy graf zawiera stosunkowo małą liczbę krawędzi w stoisunku do liczby wierzchołków. Dlatego im graf jest rzadszy, tym bardziej opłacalne jest użycie listy sąsiedztwa. Wówczas ilość pamięci potrzebna do zapisania grafu jest zminimalizowana. Niestety odbywa się to kosztem czasu wyszukiwania krawędzi – aby stwierdzić, czy istnieje krawędź z u do v, musimy przejrzeć listę sąsiedztwa danego węzła, co jest znacznie bardziej czasochłonne niż natychmiastowe sprawdzenie wartości znajdującej się w komórce tablicy T[i, j]. Warto nadmienić jeszcze o sposobie implementacji list sąsiedztwa. Otóż najczęściej używa się do tego tablicy o rozmiarze równym liczbie wierzchołków, zawierającej wskaźniki do początku list z węzłami wychodzącymi z danego wierzchołka (lub do niego przychodzącymi). Listy sąsiedztwa mają też swoje uzasadnienie również dla grafów o strukturze zmieniającej się w trakcie działania programu

A zatem w przypadku grafów z duża liczbą krawędzi najbardziej efektywne jest zastosowanie macierzy sąsiedztwa do zapisu struktury grafu w pamięci komputera. Im mniej jest zer w tablicy sąsiedztwa, tym mniejszy jest narzut pamięci na zapis grafu, przez co jest ona w mniejszym stopniu „marnowana". Zyskujemy za to wygodny dostęp do elementów macierzy i prostotę sprawdzania (w czasie O(1)) istnienia konkretnej krawędzi w grafie. Dla naszego przykładowego grafu macierz sąsiedztwa przedstawia się w sposób następujący:

	A	B	C	D	E	F
A		1			1	
B						
C			1	1		1
D		1				
E				1		1
F					1	

Porównajmy zajętość pamięci przez omówione dwa sposoby reprezentowania grafu:

- lista sąsiedztwa posiada złożoność pamięciową O(V+E), wynika to z faktu, że tworzy się listę dla każdego wierzchołka, a każda krawędź występuje łącznie dokładnie raz
- macierz sąsiedztwa wymaga zarezerwowania pamięci w komputerze o wielkości $O(V^2)$, gdyż potrzebna jest tablica o rozmiarze n x n, gdzie n jest liczbą wierzchołków grafu. W przypadku grafu nieskierowanego macierz sąsiedztwa jest macierzą symetryczną (gdyż jedna krawędź prowadzi zarówno z u do v, jak i z v do u). Dzięki temu do zapisania pełnej informacji o grafie wystarczy nam pamiętanie wartości z głównej przekątnej oraz „górnego" trójkąta macierzy.

Przeprowadzone porównanie potwierdza fakt, że na ogół efektywniejszym sposobem zapisu grafu jest zastosowanie listy sąsiedztwa – za wyjątkiem sytuacji, gdy graf jest wyjątkowo gęsty (liczba krawędzi jest zbliżona do n^2) oraz gdy szczególnie cenne jest szybkie określenie istnienia krawędzi.

W przypadku grafów ważonych razem z wierzchołkiem na liście przechowywana jest waga krawędzi prowadząca do niego z węzła, którego dana lista dotyczy.

W przypadku macierzy sąsiedztwa, sposób zapisu wag wydaje się bardzo intuicyjny. W tablicy zamiast wartości logicznej 0/1 zapisywana jest właśnie waga krawędzi. Jeśli dana krawędź nie występuje, to zapisuje się w odpowiedniej komórce tablicy wartość NULL/**nil** (która może być reprezentowana przez pewną ustaloną wartość liczbową).

Przeszukiwanie grafów

Najbardziej podstawową i niezbędną do poznania grupą algorytmów grafowych jest przeszukiwane grafu – przechodzenie między jego wierzchołkami. Czynność ta służy najczęściej do wykonywania innych, dodatkowych operacji na węzłach grafu. Zatem metody przeszukiwania grafu są niejako bazą dla innych algorytmów grafowych. Tak naprawdę istnieją dwa odmienne podejścia do „zwiedzania" węzłów w grafie:

- przeszukiwanie w głąb
- przeszukiwanie wszerz

Przeszukiwanie grafów w głąb

Przeszukiwanie w głąb (ang. *Depth-First Search*, **DFS**) jest algorytmem o prostej do zrozumienia idei. Na początku wszystkie wierzchołki grafu oznacza się jako nieodwiedzone, a w momencie dojścia do danego węzła grafu staje się on odwiedzony. Otóż rozpoczynając przechodzenie od wybranego, startowego wierzchołka grafu, poruszamy się „w głąb" grafu poprzez wybieranie kolejnych krawędzi składających się na możliwie najprostszą ścieżkę aż do momentu, gdy nie będziemy mogli przejść do kolejnego, nieodwiedzonego jeszcze wierzchołka. Wówczas należy wrócić do węzła ostatnio odwiedzonego i sprawdzić kolejną drogę (za pomocą krawędzi wychodzących z niego do nieodwiedzonych węzłów). Czynność tę powtarzamy rekurencyjnie dopóty, dopóki wszystkie wierzchołki, do których istnieje ścieżka z węzła startowego, nie zostaną odwiedzone. W przypadku, gdy w grafie będą jeszcze nieodwiedzone wierzchołki, wówczas obieramy jeden z nich jako nowy węzeł startowy i ponownie rozpoczynamy przeszukiwanie. Metoda jest kontynuowana do momentu odwiedzenia wszystkich wierzchołków grafu. Metoda DFS jest zatem dobrym przykładem algorytmu z powrotami (którego idea została omówiona w lekcji 1. podręcznika).

W przypadku algorytmu przeszukiwania w głąb najczęściej jest używana reprezentacja grafu w postaci list sąsiedztwa. Oprócz tego używa się dodatkowej tablicy zapamiętującej fakt odwiedzenia węzła (nazwijmy ją na przykład odwiedzony). Tak jak wcześniej wspomnieliśmy, algorytm oznacza najpierw wszystkie wierzchołki grafu jako nieodwiedzone, a następnie jest wywoływana procedura przeszukiwania w głąb dla każdego nieodwiedzonego jeszcze węzła:

Procedura **DFS** (graf **G**)

```
1. for (każdy węzeł v zbioru V wierzchołków grafu)
2. odwiedzony[v]  = 0      // 0 w przypadku węzła nieodwiedzonego
3.                         // 1 - gdy węzeł został odwiedzony
4. for (każdy węzeł v zbioru V wierzchołków grafu)
5.    if (odwiedzony[v] = 0)
6.     odwiedzDFS(v)
```

Procedura *odwiedzDFS* zawiera w sobie rekurencyjne przechodzenie „w głąb", rozpoczynając od wybranego wierzchołka, natomiast procedura DFS zapewnia wykonanie przeszukania całego grafu, rozpoczynając przeglądanie go w razie potrzeby z kilku węzłów początkowych, gdyby pozostawały w nim jeszcze nieodwiedzone wierzchołki.

Procedura **odwiedzDFS**(węzeł **v**)

```
1. odwiedzony[v] = 1 // zaznaczenie faktu odwiedzenia
2. for (każdy węzeł u będący sąsiadem v, znajdujący się na jego liście sąsiedztwa)
```

```
3.      if (odwiedzony[u] = 0) {
4.      //operacje związane z odwiedzaniem
5.          odwiedzDFS(u)  // rekurencyjne wywołanie
6.      }
```

Podczas odwiedzania wierzchołka mogą być wykonywane różne operacje związane z przetwarzaniem danych zawartych w węzłach. Jest to miejsce, gdzie następuje połączenie algorytmu przeszukiwania grafu z innym algorytmami grafowymi, które wykorzystują przechodzenie przez wierzchołki grafu do innych obliczeń.

Oprócz tego w momencie odwiedzenia wierzchołka (po raz pierwszy), często stosuje się zapamiętanie jego poprzednika (czyli węzła, z którego przyszliśmy). Dzięki temu możliwe jest utworzenie drzewa przeszukiwania w głąb, które obrazuje, w jaki sposób (za pomocą których krawędzi) algorytm dotarł do poszczególnych węzłów. Ponieważ w algorytmie DFS może być konieczne kilkukrotne rozpoczynanie przeszukiwania – wówczas zostanie utworzonych **kilka drzew przeszukiwań** (czyli **las**). Takie drzewo jest tworzone na przykład w algorytmie A*, który będzie omówiony w tym podręczniku (choć wykorzystuje on inny rodzaj przeszukiwania – wszerz).

Czasem konieczne jest zapamiętanie, w jakiej kolejności wierzchołki zostały odwiedzone. Wówczas, najczęściej w dodatkowej tablicy, zapisuje się tę kolejność (w odpowiedniej komórce tablicy przeznaczonej dla odwiedzanego węzła), z wykorzystaniem zmiennej, która zwiększa się wraz z odwiedzaniem kolejnych wierzchołków.

Algorytm przeszukiwania w głąb może być stosowany zarówno dla grafów nieskierowanych, jak i dla grafów skierowanych. Działa również poprawnie dla obu rodzajów grafów w przypadku innego podziału – dla grafów **spójnych** i **niespójnych**. Teraz jest więc już najwyższy czas na wprowadzenie ich definicji.

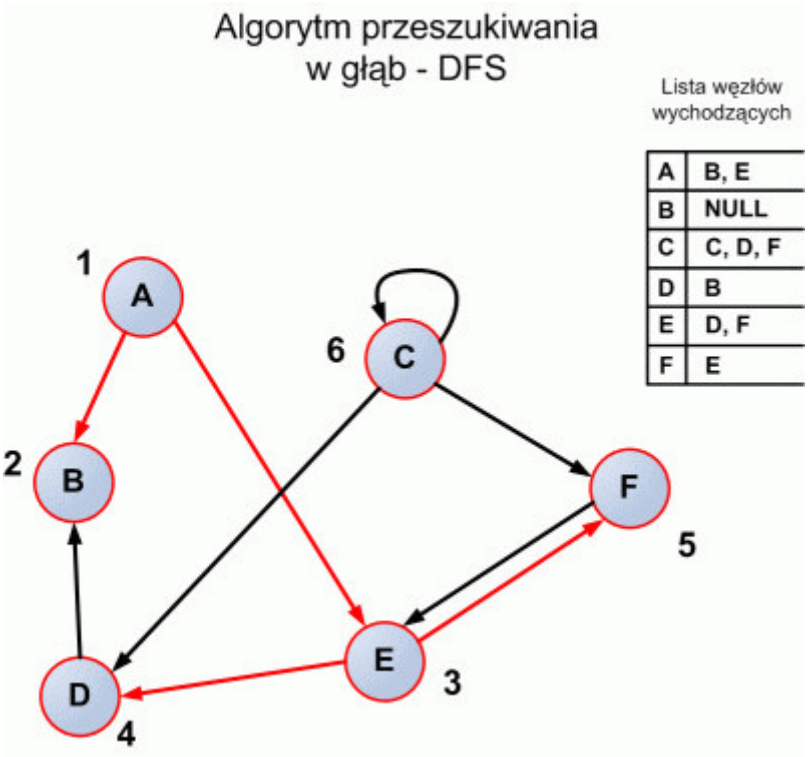
Graf nazywamy spójnym, gdy między dwoma jego dowolnymi wierzchołkami istnieje łącząca je ścieżka (czyli z dowolnego miejsca w grafie można dotrzeć do każdego innego). Jeśli graf nie posiada takiej cechy, to jest on grafem niespójnym.

W przypadku grafu spójnego procedura *odwiedzDFS* będzie wywołana tylko jeden raz wewnątrz procedury *DFS* (gdyż „za jednym zamachem" zostaną odwiedzone wszystkie wierzchołki grafu – z definicji są one osiągalne z węzła początkowego). Natomiast dla grafu niespójnego procedura ta będzie uruchamiana kilkakrotnie.

W przypadku grafów skierowanych spójność grafów można podzielić na dwie grupy: słabą spójność oraz silną spójność. Graf skierowany jest **słabo spójny**, jeśli odpowiadający mu graf skierowany (czyli po odrzuceniu kierunków krawędzi) jest spójny. Natomiast graf jest **silnie spójny**, gdy dla dowolnej pary węzłów u i v można znaleźć ścieżkę prowadząca zarówno z u do v, jak i z v do u.

Dla grafów silnie spójnych procedura *odwiedzDFS* rzeczywiście będzie wywołana jednokrotnie, natomiast dla grafów słabo spójnych – to zależy. Decyduje o tym wybór węzła startowego – jeśli zostanie nim wierzchołek, z którego są osiągalne wszystkie pozostałe węzły grafu, to *odwiedzDFS* będzie wywołana tylko jeden raz. Wynika to z faktu, że w grafie słabo spójnym **istnieją** wierzchołki, z których można dotrzeć do wszystkich pozostałych. Natomiast w grafie silnie spójnym ze **wszystkich** węzłów grafu możemy dostać się do wszystkich pozostałych jego węzłów.

Poniżej prezentujemy przeszukiwanie w głąb na przykładzie grafu już pokazanego w tym rozdziale (czerwone strzałki wraz z węzłami oznaczają drzewo przeszukiwania):



Widzimy na rysunku, że algorytm DFS został wywołany najpierw dla węzła A, z którego dostępne są wszystkie wierzchołki poza węzłem C, który nie jest osiągalny (graf ten jest słabo spójny). Dlatego po odwiedzeniu węzłów A, B, E, D, F (w takiej kolejności) procedura DFS musiała zostać ponownie wywołana, tym razem z węzłem C ustalonym jako początkowy.

Na koniec rozważań o metodzie przeszukiwania grafu w głąb, warto wspomnieć o jej złożoności obliczeniowej. Ponieważ każdy wierzchołek jest odwiedzany dokładnie raz, a każda krawędź wychodząca z każdego węzła jest sprawdzana (w celu potencjalnego odwiedzenia sąsiada) również jeden raz, więc algorytm DFS wykonuje się w czasie **O(V+E)**, co jest dobrym wynikiem, gdyż liniowo zależy od rozmiaru grafu. Metoda DFS ma wiele zastosowań: jako baza do algorytmów wyznaczających najkrótsze ścieżki w grafie, do algorytmów przeglądających drzewa gier i wielu innych metod działających na strukturach grafowych.

Przeszukiwanie grafów wszerz

Przeszukiwanie wszerz (ang. *Breadth-First Search*, w skrócie **BFS**) jest drugim z najbardziej znanych sposobów przeszukiwania grafu. Zasada jest odmienna niż w przypadku metody BFS – w pierwszej kolejności przeglądany jest węzeł początkowy (źródłowy), następnie odwiedzane są po kolei węzły sąsiadujące ze źródłem, później wierzchołki sąsiednie (te jeszcze nieodwiedzone) sąsiadów źródła itd. A zatem, jeśli wykorzystujemy listy sąsiedztwa do reprezentacji grafu, to najpierw przeglądamy wierzchołki z listy sąsiedztwa węzła początkowego, a potem wierzchołki z list sąsiedztwa węzłów, które znajdują się na liście przynależnej do źródła. W skrócie można powiedzieć, że w pierwszej kolejności algorytm dociera do węzłów oddalonych od wierzchołka początkowego o jedną krawędź, potem o dwie krawędzie itd. – stąd właśnie nazwa algorytmu: poruszamy się „wszerz" na jednym poziomie, następnie na kolejnym i na kolejnym...

Poniżej przedstawiamy zwarty pseudokod algorytmu:

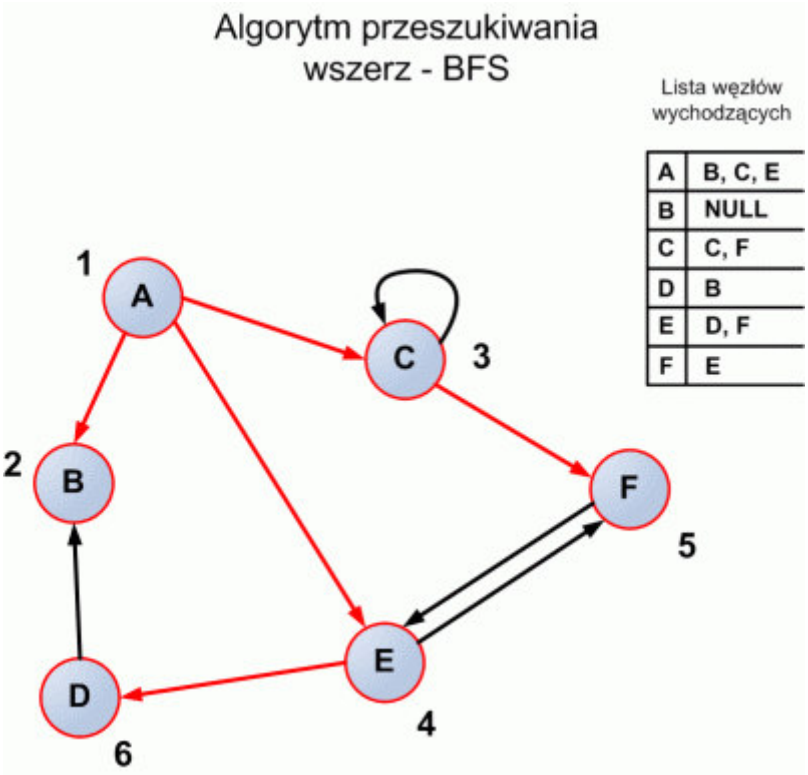
Procedura **BFS** (graf **G**)

```
1.  for (każdy węzeł v zbioru V wierzchołków grafu)
2.      odwiedzony[v]  = 0  // 0 w przypadku węzła nieodwiedzonego
3.                  // 1 - gdy węzeł został odwiedzony
4.
5.  utwórz  pustą kolejkę Q
6.  for (każdy węzeł v zbioru V wierzchołków grafu)
7.      if (odwiedzony[v] = 0) {
8.          odwiedzony[v] = 1
9.          wstaw węzeł v do Q
10.         while (kolejka Q nie jest pusta) {
11.             pobierz węzeł u z kolejki Q
12.             //operacje związane z odwiedzaniem u
13.             for (każdy węzeł w będący sąsiadem u,
14.             znajdujący się na jego liście sąsiedztwa)
```

```
15.                 if (odwiedzony[w] = 0) {
16.                     odwiedzony[w] = 1
17.                     wstaw węzeł w do Q
18.                 }
19.             }
20.         }
```

Jak widzimy, do przeszukiwania wszerz stosowana jest kolejka Q (w przeciwieństwie do DFS, gdzie w sposób niejawny wykorzystywany był stos do celów rekurencyjnych wywołań). W pierwszej kolejności do Q trafia pierwszy wierzchołek z listy V (początkowy), następnie jest on wyjmowany z kolejki, po czym trafiają do niej wszystkie węzły sąsiadujące z węzłem początkowym. W dalszej kolejności ponownie pobieramy węzły z kolejki Q i umieszczamy wszystkie wierzchołki sąsiednie (jeszcze nieodwiedzone) w kolejce. Proces ten jest kontynuowany do momentu opróżnienia kolejki. Jeśli okaże się, że w grafie pozostały jeszcze nieprzejrzane węzły, to jeden z nich trafia do kolejki Q i proces przeszukiwania postępuje dalej – aż cały graf zostanie przeszukany.

Warto poprzeć teorię przydatnym rysunkiem. Ponieważ dla naszego przykładowego grafu skierowanego przeszukiwanie wszerz daje dokładnie takie same rezultaty, jak przeszukiwanie w głąb (tzn. węzły są odwiedzane w identycznej kolejności, możecie to sprawdzić - przy pewnych strukturach grafu czasem może wystąpić takie zjawisko), zdecydowaliśmy się lekko zmodyfikować układ krawędzi w grafie. W ten sposób zaprezentowana metoda będzie bardziej widoczna:



Na rysunku czerwonymi strzałkami zaznaczono drzewo przeszukiwania wszerz (do którego należą oczywiście również wierzchołki grafu). W tym drzewie każdy węzeł jest oddalony od wierzchołka początkowego o minimalną liczbę krawędzi (fakt ten wynika z przechodzenia przez graf kolejnymi poziomami).

Metoda BFS działa poprawnie (podobnie jak DFS) dla grafów nieskierowanych i skierowanych, spójnych oraz niespójnych.

Nie wypada nie wspomnieć o złożoności obliczeniowej metody BFS. Jak wynika z przedstawionego opisu oraz pseudokodu, każdy węzeł v ze zbioru wierzchołków V jest dokładnie raz umieszczany w kolejce Q, a także raz jest z niej wyjmowany. Zatem operacje te potrzebują czasu O(V). Prócz tego wiemy, że podczas usuwania wierzchołka z kolejki sprawdzana jest lista jego sąsiadów (dokładnie raz dla każdego węzła), zatem każda krawędź jest również jednokrotnie analizowana. Wynika z tego, że potrzebujemy na to czasu O(E). W związku z tym całkowity czas działania algorytmu przeszukiwania wszerz jest liniowo proporcjonalny do rozmiaru grafu i wynosi **O (V+E)**. Podsumowując, oba algorytmy, DFS i BFS posiadają identyczną złożoność obliczeniową.

Algorytm przeszukiwania wszerz posiada szereg zastosowań. Ponieważ znajduje on optymalną liczbę krawędzi na ścieżce od źródła do określonego węzła, to przy użyciu pewnych modyfikacji jest on stosowany do obliczania najkrótszych ścieżek między wierzchołkami grafu (np. w algorytmie Dijkstry, co zobaczycie w rozdziale 3). Znany algorytm Prima znajdujący minimalne drzewo rozpinające grafu (powrócimy do niego w rozdziale 5 tej lekcji) również bazuje na przeszukiwaniu wszerz. Tym samym metoda BFS umożliwiła rozwój wielu innych, bardziej złożonych algorytmów, obecnie bardzo często wykorzystywanych w praktyce.

Grafy a drzewa

Wprowadzone definicje grafów pozwalają nam w odmienny sposób spojrzeć na struktury drzewiaste poznane w poprzednim rozdziale. Okazuje się bowiem, że drzewa można traktować jako szczególne przypadki grafów. Należy jednak wcześniej zdefiniować pojęcie cyklu w grafie: **cykl** jest to ścieżka prosta zamknięta, czyli taka, której koniec pokrywa się z początkiem i która nie zawiera oprócz tego innych powtarzających się węzłów. Teraz już możemy podsumować:

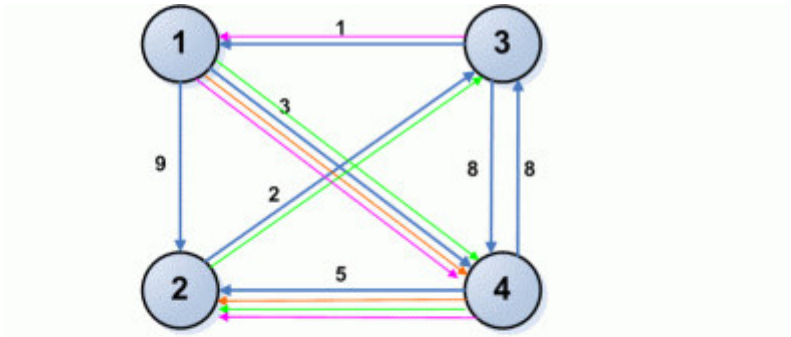
Graf nieskierowany, który jest spójny i nie ma cykli, nazywamy drzewem.

Sprawdzenie, że tak jest w istocie, pozostawiamy Wam jako ćwiczenie...

Algorytm Floyda-Warshalla

We wprowadzeniu do grafów wspomnieliśmy, że jedną z najważniejszych grup problemów występujących w strukturach grafowych jest wyznaczanie najkrótszych ścieżek między dowolnymi węzłami grafu. Rozwiązanie takiego problemu ma bardzo dużo zastosowań, jak na przykład wyliczanie minimalnych odległości (lub czasu podróży) między zbiorem miast (między którymi można się przemieszczać za pomocą połączeń lotniczych lub sieci autostrad).

Algorytm Floyda-Warshalla jest przykładem algorytmu służącego do rozwiązywania właśnie takiego typu problemów. Wyznacza on koszt najkrótszej ścieżki między każdą parą wierzchołków w grafie ważonym. Do wyznaczania najkrótszych ścieżek stosuje się najczęściej reprezentację grafu w postaci macierzy sąsiedztwa. Rozwiązanie problemu jest również zapisywane w postaci tablicy dwuwymiarowej D o wymiarach N x N, gdzie N jest liczbą wierzchołków grafu. W rezultacie wykonania algorytmu w każdej komórce D[i,j] znajduje się wartość odpowiadająca kosztowi najkrótszej ścieżki prowadzącej z wierzchołka i do wierzchołka j, co widać poniżej.



Tablica najkrótszych ścieżek D

Węzeł początkowy \ Węzeł końcowy	1	2	3	4
1	0	8	10	3
2	3	0	2	6
3	1	9	0	4
4	8	5	7	0

Tablica węzłów pośrednich P

Węzeł początkowy \ Węzeł końcowy	1	2	3	4
1	0	4	4	1
2	3	0	2	3
3	3	4	0	1
4	3	4	2	0

Oprócz samego kosztu najkrótszej drogi prowadzącej między dwoma wierzchołkami, równie cenna jest informacja o przebiegu takiej ścieżki (czyli przez jakie pośrednie wierzchołki przechodzi ta droga). W tym celu równolegle do wyznaczania kosztów ścieżek w tablicy D, wyznaczane są indeksy poprzedników (węzłów poprzedzających inne węzły) znajdujących się na najkrótszych ścieżkach prowadzących do konkretnych węzłów. Jest to zapisywane, i w trakcie działania algorytmu sukcesywnie aktualizowana jest tablica

zawierająca indeksy poprzedników (również o rozmiarze N x N), która oznaczymy jako P i szczegółowo omówimy później na przykładzie. .

Wartościową zaletą algorytmu Floyda-Warshalla jest fakt, że może być on stosowany dla grafów o ujemnych wagach (w przeciwieństwie do algorytmu Dijkstry wyznaczającego najkrótsze ścieżki z jednego węzła źródłowego – dowiecie się o nim w następnym podrozdziale). Omawiany algorytm jest przykładem metody stosującej zasady programowania dynamicznego. Niebawem w skrócony sposób wyjaśnimy, dlaczego tak właśnie jest.

Na razie spróbujemy Wam przedstawić w sposób możliwie prosty i zwięzły sposób główną ideę algorytmu. Należy przyjąć, że zgodnie z macierzą sąsiedztwa węzły grafu są oznaczone indeksami od 1 do n. Początkowo elementy tablicy D są równe odpowiadającym im elementom macierzy sąsiedztwa T (wystarczy proste przepisanie wartości). Należy pamiętać, że macierz sąsiedztwa T jest konstruowana w taki sposób, że:

- T[i,j] jest równa wadze krawędzi łączącej węzły i, j, jeśli w grafie jest taka krawędź
- T[i, j] = 0, jeśli i=j
- T[i,j]= +Inf, jeżeli nie istnieje krawędź łącząca wierzchołki i, j

A zatem we wstępnej fazie algorytmu najkrótsze ścieżki między dowolną parą węzłów posiadają koszt równy wadze krawędzi łączącej te dwa wierzchołki (jeśli takowa istnieje - najkrótsza ścieżka nie może zawierać w tym momencie żadnych węzłów pośrednich). W kolejnych krokach algorytm rozpatruje kolejno węzły oznaczone indeksem 1,2,....,k,....,n. Najpierw sprawdzane jest (dla każdej pary węzłów i, j), czy ścieżka z i do j prowadząca przez pośredni węzeł o indeksie 1 jest krótsza niż aktualnie znana najkrótsza droga między wierzchołkami i oraz j (bez żadnych węzłów pośrednich). Jeśli tak właśnie jest, to długość najkrótszej ścieżki jest aktualizowana (wiedzie ona teraz przez węzeł 1). W kolejnym kroku algorytm sprawdza, czy włączenie węzła 2 do najkrótszej ścieżki z i do j nie poprawi wartości jej długości. Wówczas wartość D[i,j] będzie zawierać długość najkrótszej ścieżki prowadzącej bezpośrednio z węzła i do j lub też zawierającej co najwyżej węzły 1 oraz 2 jako wierzchołki pośrednie. W ogólności, po k-tym kroku, zostaną obliczone najkrótsze ścieżki między parami wierzchołków, zawierające węzły pośrednie ze zbioru wierzchołków o indeksach (1,..,k). Tym samym po zakończeniu działania algorytmu w tablicy D będą wyliczone koszty najkrótszych ścieżek, które mogą przebiegać przez wszystkie węzły grafu G – w tym momencie otrzymywane jest ostateczne rozwiązanie problemu znalezienia najkrótszych ścieżek miedzy parami wierzchołków grafu.

Algorytm Floyda-Warshalla jest przykładem programowania dynamicznego, gdyż w danym kroku (iteracji) algorytm wykorzystuje wartości D[i,j] obliczone w poprzednim kroku metody (tak, jak to pokazaliśmy przed momentem, algorytm w kroku k-tym sprawdza, czy można „poprawić", poprzez włączenie do niej węzła k-tego, ścieżkę mogącą do tej pory zawierać węzły pośrednie ze zbioru (1,...,k-1)).

Rozwiązanie problemu charakteryzuje się wobec tego poniższym równaniem rekurencyjnym (wartość k oznacza krok iteracji):

$$D[i,j]^{(k)} = \begin{cases} T[i,j], & \textbf{gdy } \mathbf{k = 0} \\ \min(D[i,j]^{(k-1)}, D[i,k]^{(k-1)} + D[k,j]^{(k-1)}), & \textbf{gdy } \mathbf{k \geq 1} \end{cases}$$

Dla przypomnienia, T[i, j] oznacza wagę krawędzi łączącej wierzchołki i, j.

Sama implementacja algorytmu jest prosta, a kod programu jest naprawdę krótki. Algorytm w n-krokowej pętli przegląda całą tablicę D i sprawdza możliwość poprawy kosztu najkrótszej ścieżki między każdą parą wierzchołków grafu. Wraz z uaktualnianiem tablicy D, modyfikowana jest również tablica P, która zawiera największy indeks węzła pośredniego znajdującego się na najkrótszej ścieżce między parami wierzchołków.

Procedura **FloydWarshall** (graf **G**)

```
1.  skopiuj wartości macierzy sąsiedztwa T do tablicy D
2.  inicjalizuj tablicę P zerami
3.
4.  for (każdy węzeł k grafu spośród węzłów (1,..,n) )
5.      for (każdy węzeł i grafu spośród węzłów (1,..,n) )
6.          for (każdy węzeł j grafu spośród węzłów (1,..,n) )
7.              if (D[i,k]+D[k,j] < D[i,j]) {
8.                  D[i,j] = D[i,k] + D[k,j]     //
    najkrótsza ścieżka prowadzi teraz przez węzeł k
9.                  P[i,j] = k
10.             }
```

Warto nadmienić, że w przypadku, gdy najkrótsza ścieżka między parą wierzchołków nie zawiera węzłów pośrednich, wówczas wartość w tablicy P dla tej pary węzłów wynosi 0. Dzięki tablicy P jesteśmy w stanie „odtworzyć" przebieg najkrótszej ścieżki między parą wierzchołków. Ponieważ wartość r= P[i,j] wskazuje nam na jeden z węzłów pośrednich ścieżki z i do j, to musimy następnie

sprawdzić, czy ścieżki z i do r oraz z r do j również posiadają węzły pośrednie (naturalnie odczytując wartości P[i,r] oraz P[r,j]). Proces „wyłuskiwania" wszystkich węzłów pośrednich kontynuujemy do momentu, gdy wszystkie cząstkowe ścieżki będą składały się z pojedynczych krawędzi.

Poniżej przedstawiamy zwięzły pseudokod algorytmu wypisywania najkrótszej ścieżki (stosując rekurencyjne wywołania):

Procedura **wypiszSciezke** (węzeł **i**, węzeł **j**)

```
1. if (P[i,j] jest różne od 0)  {
2.     wypiszSciezke (węzeł i , węzeł P[i,j])
3.     drukuj P[i,j]
4.     wypiszSciezke (węzeł P[i,j] , węzeł j)
5. }
```

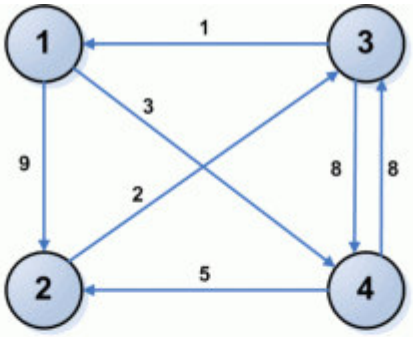
Jak zaznaczyliśmy na początku omawiania algorytmu Floyda-Warshalla, istnieje inny sposób zapisu wartości indeksów wierzchołków w P. Wartością P[i,j] może być indeks poprzednika (czyli dokładnie ostatniego węzła) znajdującego się bezpośrednio przed węzłem docelowym na najkrótszej ścieżce. Przy takim zapisie odtworzenie najkrótszej drogi jest nieco łatwiejsze, gdyż w kodzie funkcji *wypiszSciezke* wywołanie rekurencyjne występuje raz, a nie dwukrotnie. Sposób modyfikacji funkcji *FloydWarshall*, który prowadzi do zastosowania drugiego wariantu zapisywania wartości tablicy P, zostawiamy dla Waszych przemyśleń.

Tradycyjnie warto wspomnieć o złożoności obliczeniowej algorytmu. Pseudokod jest bardzo czytelny, dzięki czemu z łatwością możemy wskazać rząd wielkości czasu potrzebnego do wykonania algorytmu. W kodzie funkcji istnieje potrójnie zagnieżdżona pętla for, każda z nich zawiera liczbę iteracji równą liczbie wierzchołków grafu. Żadne inne czasochłonne operacje nie są wykonywane (wyłącznie proste przypisania i porównania), tak więc złożoność czasowa algorytmu Floyda-Warshalla wynosi $O(V^3)$. Dużą zaletą tej metody jest jej prostota i brak stosowania złożonych struktur danych (np. kopców binarnych czy Fibonacciego). Niemniej jednak złożoność obliczeniowa wskazuje, że algorytm działa w rozsądnym czasie jedynie dla grafów o umiarkowanych rozmiarach.

Co ciekawe, dla grafów rzadkich o dużych rozmiarach. efektywniejszym algorytmem do obliczania najkrótszych ścieżek między parami wierzchołków jest algorytm Johnsona. Wykorzystuje on dwa inne algorytmy grafowe (Bellmana-Forda oraz Dijkstry) oraz stosuje złożone struktury danych (m.in. kopce binarne lub Fibonacciego), co sprawia, że jest on zdecydowanie trudniejszy w implementacji niż algorytm Floyda-Warshalla. Niemniej jednak w przypadku konieczności znalezienia ścieżek w naprawdę dużym grafie (i do tego w miarę rzadkim), warto zastanowić się nad zastosowaniem metody opracowanej przez Johnsona. Jeszcze jedną ciekawostkę stanowi fakt, że używa on reprezentacji grafu w postaci listy sąsiedztwa.

Przykład

Aby uzmysłowić w sposób klarowny działanie algorytmu Floyda-Warshalla, przedstawiamy Wam poniższy przykład graficzny. Oto załóżmy, że chcemy wyznaczyć wszystkie najkrótsze ścieżki dla grafu o poniższej strukturze (tego samego, który pokazaliśmy na wstępie tego rozdziału):



Tablica najkrótszych ścieżek D

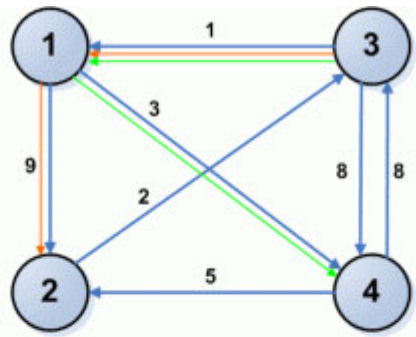
Węzeł początkowy \ Węzeł końcowy	1	2	3	4
1	0	9	999	3
2	999	0	2	999
3	1	999	0	8
4	999	5	8	0

Tablica węzłów pośrednich P

Węzeł początkowy \ Węzeł końcowy	1	2	3	4
1	0	1	0	1
2	0	0	2	0
3	3	0	0	3
4	0	4	4	0

Jak widzimy, od początku „towarzyszą" nam tablice D i P. Wartość 999 w komórkach tablicy D jest odpowiednikiem +Inf, natomiast wartość 0 w tablicy P oznacza, że na ścieżce między wierzchołkiem początkowym a końcowym nie ma ani jednego wierzchołka pośredniego (0 jest praktycznym zapisem wartości NULL/nil).

Rozpoczynamy właściwy, pierwszy krok nadrzędnej pętli. Sprawdzamy, czy wstawienie wierzchołka k=1 poprawi wartości najkrótszych ścieżek między wierzchołkami grafu:



Tablica najkrótszych ścieżek D

Węzeł początkowy \ Węzeł końcowy	1	2	3	4
1	0	9	999	3
2	999	0	2	999
3	1	10	0	4
4	999	5	8	0

Tablica węzłów pośrednich P

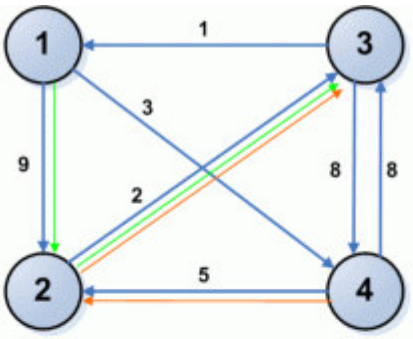
Węzeł początkowy \ Węzeł końcowy	1	2	3	4
1	0	1	0	1
2	0	0	2	0
3	3	1	0	1
4	0	4	4	0

Okazuje się, że zastosowanie węzła 1 jako wierzchołka pośredniego poprawia długość najkrótszej ścieżki między węzłami 3->2 oraz 3->4. Zmiany wartości najkrótszych ścieżek zaznaczono pomalowaniem odpowiednich komórek tablicy T, a przebieg uaktualnionych

ścieżek nakreślono na grafie kolorowymi strzałkami. W związku z dodaniem nowego węzła pośredniego do ścieżek, w odpowiednie miejsca w tablicy P wstawiana jest wartość indeksu węzła 1.

W tej i w kolejnych iteracjach przyjrzyjmy się, że na rysunku grafu wszystkie zaktualizowane najkrótsze ścieżki (zaznaczone kolorowymi strzałkami) przechodzą przez analizowany w danym kroku węzeł pośredni.

Przechodzimy do drugiego kroku iteracji. W tym momencie próbujemy poprawić najkrótsze ścieżki przy użyciu węzła o indeksie k=2:



Tablica najkrótszych ścieżek D

Węzeł początkowy \ Węzeł końcowy	1	2	3	4
1	0	9	11	3
2	999	0	2	999
3	1	10	0	4
4	999	5	7	0

Tablica węzłów pośrednich P

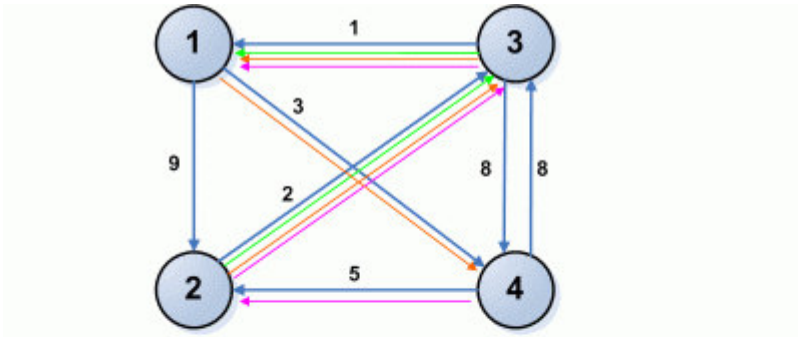
Węzeł początkowy \ Węzeł końcowy	1	2	3	4
1	0	1	2	1
2	0	0	2	0
3	3	1	0	1
4	0	4	2	0

W tym kroku algorytm poprawia dwie ścieżki poprzez umieszczenie w nich węzła 2. Mianowicie dotyczy to:

- drogi 1->3, gdyż T[1,2]+T[2,3] = 9 +2 = 11 i jest mniejsze niż T[1,3]= 999 **(oczywiście mamy na myśli dotychczasową wartość T[1,3], która właśnie jest modyfikowana)**
- drogi 4->3, gdyż T[4,2]+T[2,3] = 5 + 2 = 7 i jest mniejsze niż T[4,3] = 8

Tak jak w poprzedniej iteracji, zmienione wartości najkrótszych ścieżek, jak i ich przebieg zaznaczono kolorami, a w tablicy P wstawiono dla uaktualnionych ścieżek indeks nowo umieszczonego węzła pośredniego.

Przejdźmy do iteracji o numerze k=3:



Tablica najkrótszych ścieżek D

Węzeł początkowy \ Węzeł końcowy	1	2	3	4
1	0	9	11	3
2	3	0	2	6
3	1	10	0	4
4	8	5	7	0

Tablica węzłów pośrednich P

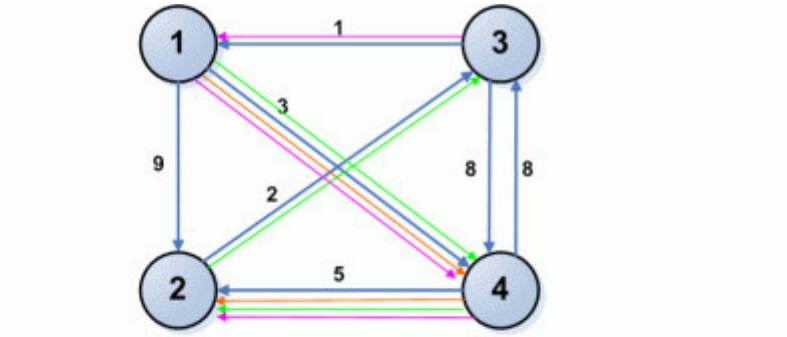
Węzeł początkowy \ Węzeł końcowy	1	2	3	4
1	0	1	2	1
2	3	0	2	3
3	3	1	0	1
4	3	4	2	0

Tym razem aż trzy ścieżki wymagały poprawienia. Są nimi:

- ścieżka 2->1, gdyż T[2,3]+T[3,1] = 2 +1 = 3 i jest mniejsze niż T[2,1]= 999
- ścieżka 2->4, gdyż T[2,3]+T[3,4] = 2 +4 = 6 i jest mniejsze niż T[2,4]= 999
- ścieżka 4->1, gdyż T[4,3]+T[3,1] = 7 +1 = 8 i jest mniejsze niż T[4,1]= 999

W tablicy P w odpowiednich miejscach wstawiamy indeks węzła 3.

Pozostał nam już tylko ostatni, czwarty krok iteracji:



Tablica najkrótszych ścieżek D

Węzeł początkowy \ Węzeł końcowy	1	2	3	4
1	0	8	10	3
2	3	0	2	6
3	1	9	0	4
4	8	5	7	0

Tablica węzłów pośrednich P

Węzeł początkowy \ Węzeł końcowy	1	2	3	4
1	0	4	4	1
2	3	0	2	3
3	3	4	0	1
4	3	4	2	0

Jak się okazało, ponownie trzy drogi wymagały uaktualnienia – w tym przypadku za pomocą węzła o indeksie 4. Odnosi się to do ścieżek:

- 1->2, gdyż T[1,4]+T[4,2] = 3 +5 = 8 i jest mniejsze niż T[1,2]= 9
- 1->3, gdyż T[1,4]+T[4,3] = 3 +7 = 10 i jest mniejsze niż T[1,3]= 11
- 3->2, gdyż T[3,4]+T[4,2] = 4 +5 = 9 i jest mniejsze niż T[3,2]= 10

Ponieważ sprawdziliśmy już wszystkie cztery węzły w roli wierzchołków pośrednich należących do najkrótszych ścieżek, algorytm Floyda-Warshalla kończy w tym miejscu swój proces. Wszystkie najkrótsze ścieżki między każdą parą węzłów grafu zostały wyznaczone.

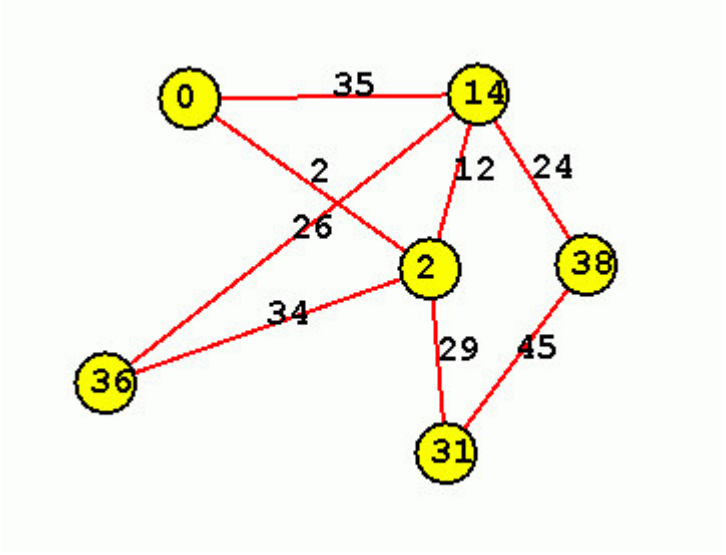
Możemy również na podstawie tablicy P odtworzyć przebieg najkrótszej ścieżki: przykładowo chcemy wiedzieć, którędy biegnie najkrótsza droga z węzła 1 do węzła 3. W tym celu celu odczytujemy P[1,3] = 4, następnie sprawdzamy P[1,4] = 1 (czyli ścieżka ta jest pojedynczą krawędzią) oraz P[4,3]= 2. Następnie odczytujemy wartość P[4,2] = 4 oraz P[2,3] = 2, a zatem doszliśmy już do elementarnych części tej ścieżki. Przebiega ona w sposób następujący: **1->4->2->3**.

Algorytm Dijkstry

Najbardziej znanym i rozpowszechnionym algorytmem wyszukiwania najkrótszych ścieżek w grafie z jednym źródłem jest algorytm Dijkstry. Został on opracowany w roku 1959r. przez jednego z najsłynniejszych pionierów informatyki - zajrzyjcie od razu do Wikipedii. Algorytm Dijkstry wyznacza najkrótsze ścieżki z jednego wybranego węzła (źródła) do pozostałych węzłów grafu. Bardzo ważnym warunkiem koniecznym do poprawnego działania tego algorytmu jest to, by wszystkie krawędzie grafu posiadały nieujemne wagi. W naszych rozważaniach będziemy zajmowali się grafami spójnym i nieskierowanymi, jednakże metoda Dijkstry działa również dla skierowanych grafów (które są tak naprawdę nadzbiorem grafów nieskierowanych).

Algorytm wyszukuje **najkrótsze ścieżki z jednego źródła** do wszystkich pozostałych węzłów grafu, jednakże może być on w prosty sposób wykorzystany do znalezienia najkrótszej ścieżki między dwoma węzłami – wystarczy zatrzymać algorytm w momencie, gdy poszukiwana najkrótsza ścieżka zostanie znaleziona, a nie potrzebujemy informacji o reszcie pozostałych jeszcze nieokreślonych ścieżek między źródłem a innymi węzłami. Metoda Dijsktry ma zastosowanie szczególnie we wszelkich zagadnieniach transportowych, które możemy zamodelować postaci grafu – dotyczy to m.in. zarówno transportu towarów między miastami, jak i przesyłania pakietów danych między węzłami sieci internetowej.

Poniżej przedstawiamy przykładowy graf z wyznaczonymi najkrótszymi ścieżkami z węzła znajdującego się w lewym-górnym rogu (o wartości etykiety równej 0):



Metoda Dijkstry polega na wprowadzeniu podziału zbioru wierzchołków grafu na trzy zbiory:

- zbiór **Q**, w którym znajdują się wierzchołki, do których „dotarł" już algorytm, ale nie możemy jeszcze określić, czy znaleziono do nich najkrótszą ścieżkę ze źródła
- zbiór **S**, w którym znajdują się wierzchołki (węzły) grafu o już ustalonej najkrótszej ścieżce ze źródła, dalej nie będą już rozpatrywane w algorytmie
- pozostałe węzły, czyli zbiór **V- Q – S**, gdzie V jest zbiorem wszystkich wierzchołków grafu G = (V, E)

Warto nadmienić, że częściej spotykana wersja algorytmu Dijkstry zakłada, że do zbioru Q trafiają „od razu" wszystkie węzły grafu G, a następnie są po kolei pobierane pojedynczo węzły, które trafiają do zbioru S.

Idea omawianego algorytmu jest następująca. Rozpoczynamy od dodania węzła źródłowego r do zbioru Q. Następnie rozpoczynamy cykliczne wykonywanie ciągu operacji, na który składa się:

- wybranie ze zbioru Q wierzchołka o znanej najmniejszej odległości od źródła (nazwijmy go Vi)
- dodanie powyższego węzła do zbioru S
- przejrzenie wszystkich sąsiednich węzłów wierzchołka Vi, które nie należą do zbioru S – następnie jest sprawdzane, czy droga ze źródła do sąsiada (oznaczmy ten węzeł jako Vj) prowadząca przez węzeł Vi jest krótsza niż najkrótsza ścieżka znana do tej pory, jeśli tak jest, to następuje uaktualnienie najkrótszej ścieżki oraz dodanie tego węzła do zbioru Q

Jeżeli w pewnym momencie okaże się, że zbiór Q jest pusty, to należy zakończyć działanie algorytmu – najkrótsze ścieżki do wszystkich węzłów osiągalnych ze źródła zostały już wyznaczone.

Czynność sprawdzania, czy ścieżka przechodząca przez Vi i prowadząca do Vj jest krótsza od dotychczas znanej najkrótszej ścieżki do Vj, jest nazywana **relaksacją krawędzi** (Vi, Vj). W przypadku gdy powyższy warunek jest spełniony, długość najkrótszej ścieżki oraz ostatni węzeł poprzedzający Vj na tej ścieżce muszą zostać zapamiętane.

Relaksacja krawędzi jest jedynym miejscem w algorytmie, w którym długości najkrótszych ścieżek oraz określenie poprzedników węzłów są zmienianie. Podobnie i inne algorytmy wyznaczające najkrótsze ścieżki w grafie z jednym źródłem korzystają z metody relaksacji krawędzi – np. znany algorytm Bellmana-Forda.

Poniżej prezentujemy zapis algorytmu w formie pseudokodu:

Procedura **Dijkstra** (graf **G**, węzeł źródłowy **r**)

```
1. S - zbiór pusty
2. Q - zbiór pusty
3. for (każdy węzeł i grafu G(V, E) ) {
4.
5.     D[i] = nieskończoność   //tablica D przechowująca wartości aktualnie najkrótszej
6.                 //ścieżki od źródła do węzła vi
7.
```

```
    P[i] = 0    //tablica P przechowująca indeks węzła, który jest
            //poprzednikiem węzła i na najkrótszej ścieżce od źródła
            //do i
}
dodaj źródło (węzeł) r do zbioru Q
D[r] = 0

while (zbiór Q nie jest pusty) {
    wybierz ze zbioru Q wierzchołek o indeksie i, dla którego wartość D[i]
 będzie najmniejsza
    dodaj wierzchołek i do zbioru S

    for (każdy wierzchołek j sąsiadujący z wierzchołkiem i, który nie należy do S)
        if  (D[i] +w[i,j] < D[j]) { //oznacza to, że znaleziono krótszą ścieżkę
                        //do i przez węzeł j niż znaleziona do tej
                        //pory, należy
                        //ją zaktualizować i określić
                        //nowego poprzednika
            D[j] = D[i] +w[i,j]
            P[j] = i
            dodaj węzeł j do zbioru Q
        }
}
```

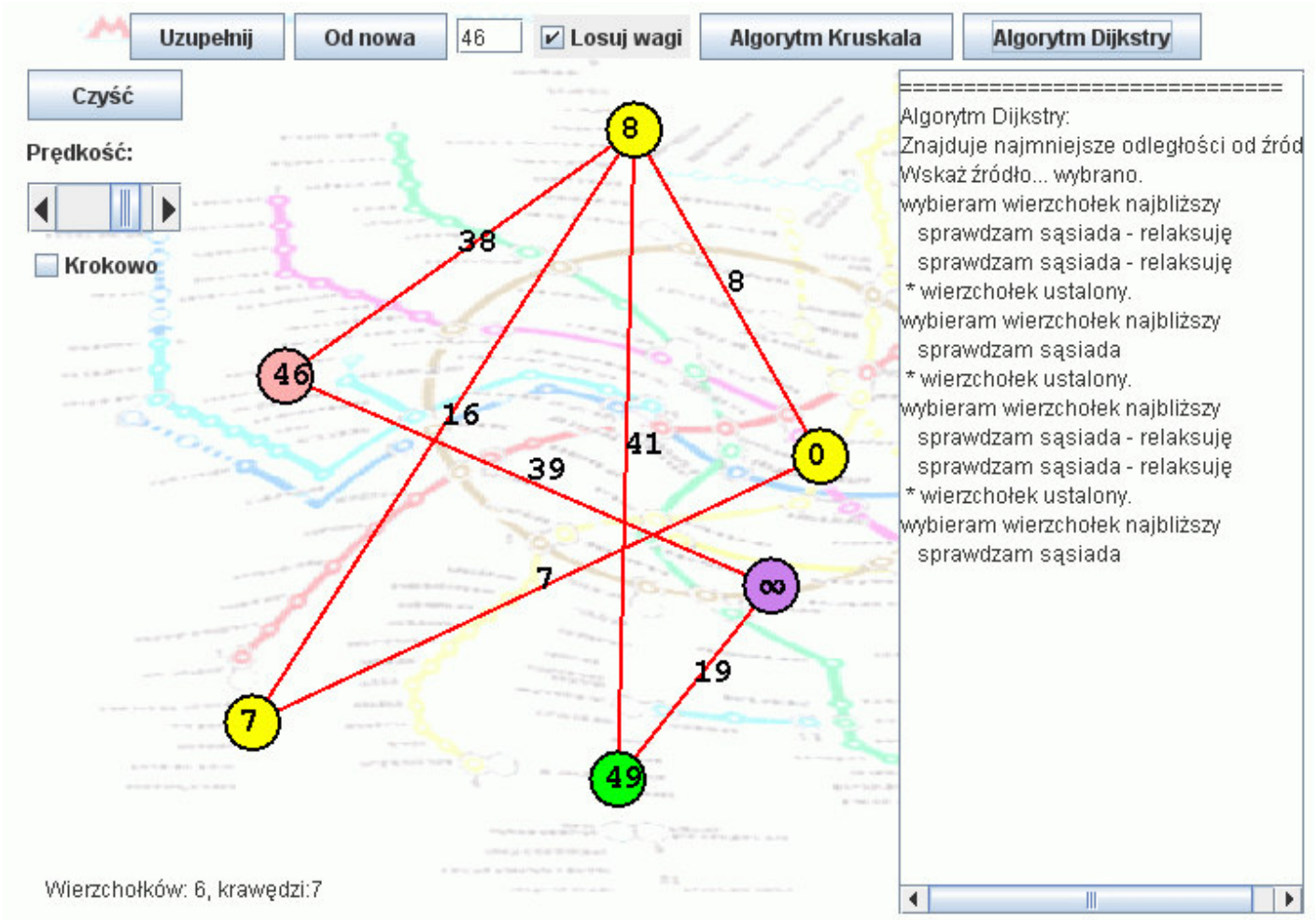
Teraz warto uzasadnić, dlaczego ten algorytm działa poprawnie i znajduje najkrótsze ścieżki od źródła do wszystkich pozostałych węzłów grafu. Otóż wiemy, że w zbiorze S znajdują się wierzchołki grafu, do których już znaleziono najkrótszą ścieżkę ze źródła. W danym kroku iteracji wybieramy jeden wierzchołek ze zbioru Q – oznaczmy go jako x. Wiemy, że odległość węzła x od źródła r jest najmniejsza spośród odległości wszystkich wierzchołków należących do Q od węzła początkowego. Z tego też powodu na ścieżce prowadzącej ze źródła do węzła x mogą znajdować się węzły ze zbioru S, natomiast nie znajduje się na niej żaden inny węzeł ze zbioru Q (powiedzmy węzeł y), do którego odległość ze źródła byłaby mniejsza, niż odległość z tego źródła do węzła x – gdyż w takim przypadku to wierzchołek y zostałby wybrany z Q i dołączony do S jako ten o najkrótszej wyznaczonej ścieżce (zamiast węzła x). A ponieważ krawędź łącząca ostatni węzeł z S leżący na najkrótszej ścieżce do x z pierwszym węzłem należącym do Q na tej ścieżce została zrelaksowana (podczas przenoszenia węzła do zbioru S), zatem wyznaczona ścieżka do x jest faktycznie najkrótszą ścieżką prowadzącą do niego od źródła r.

Na koniec rozważań warto zastanowić się nad złożonością obliczeniową algorytmu Dijkstry. W przypadku najprostszej implementacji w postaci zwykłej listy oraz liniowego wyszukiwania najmniejszego elementu (węzła z Q o najmniejszej wartości D) złożoność takiej realizacji wynosi w przybliżeniu $O(V^2)$.

Natomiast jeśli graf posiada stosunkowo mniej krawędzi niż graf pełny na tej samej liczbie wierzchołków (gdy jest odpowiednio rzadki) to do implementacji metody Dijkstry możemy posłużyć się kolejką priorytetową zrealizowaną w postaci kopca binarnego lub Fibonacciego. W pierwszym przypadku złożoność obliczeniowa jest szacowana na O((E+V)lg V) O(E lg V). Czas obliczeń możemy jeszcze poprawić właśnie przez zastosowanie struktury kopca Fibonacciego, dla której czas wykonania operacji zmiany wartości klucza (w naszym przypadku zmiany wartości D[i] węzła i) jest zredukowany do O(1). Dzięki temu całkowity czas działania algorytmu w tej wersji wynosi O(E + VlgV) i jest najkrótszy z wszystkich wspomnianych.

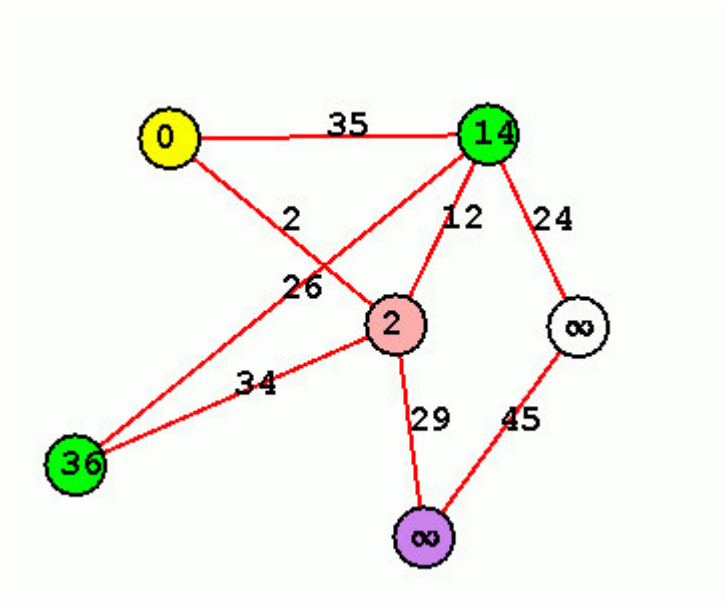
Aplet

Działanie algorytmu Dijkstry można ze szczegółami prześledzić na rozbudowanym aplecie, do którego się wchodzi przez kliknięcie w obrazek:



Węzły grafu przyjmują różne kolory – na samym początku wszystkie posiadają kolor biały oraz mają przypisaną do nich etykietę odległości równą +Inf (po wybraniu źródła wartość w tym wierzchołku wynosi 0). W kolejnych krokach wierzchołek grafu może zostać „pomalowany" w następujący kolor:

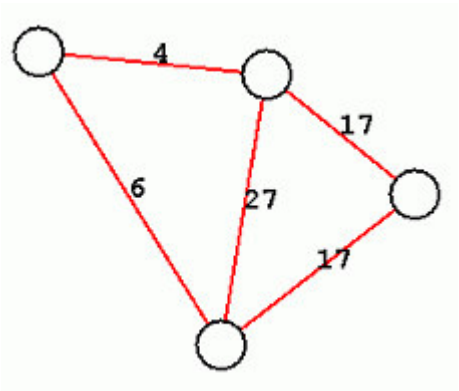
- **jasnoróżowy** – wyznacza on węzeł, który jest wybierany spośród wierzchołków zbioru Q jako ten o najmniejszej wartości D (znanej odległości od źródła)
- **zielony** – określa on węzeł, który znajduje się w zbiorze Q
- **żółty** – określa on węzeł, który znajduje się w zbiorze S (a więc znana jest już najkrótsza ścieżka ze źródła do tego wierzchołka)
- **fioletowy** – określa on węzeł, który jest w tym momencie „przeglądany" jako jeden z sąsiadów aktualnego węzła jasnoróżowego



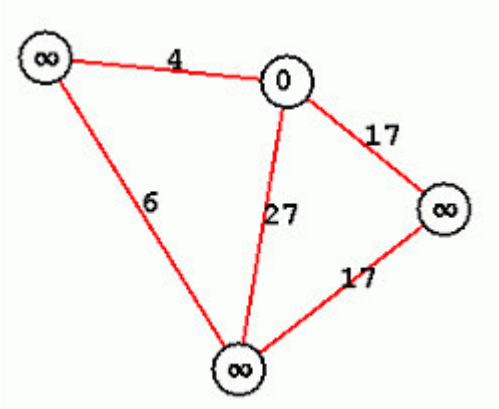
W momencie wyboru jednego z wierzchołków ze zbioru Q, węzeł zostaje pomalowany na kolor jasnoróżowy, następnie wszyscy jego sąsiedzi są sprawdzani (malowani na fioletowo), a następnie przyjmują oni kolor zielony (trafiają oni do zbioru Q, jeśli wcześniej w nim nie byli). W momencie zmiany koloru z fioletowego na zielony może (ale nie musi) nastąpić relaksacja tego wierzchołka – poprawienie najkrótszej znanej ścieżki ze źródła to sprawdzanego wierzchołka. Gdy już wszystkie wierzchołki będą posiadały kolor żółty, oznaczać to będzie, że algorytm Dijkstry został zakończony – znaleziono najkrótsze ścieżki z węzła źródłowego do wszystkich pozostałych wierzchołków grafu.

Przykład

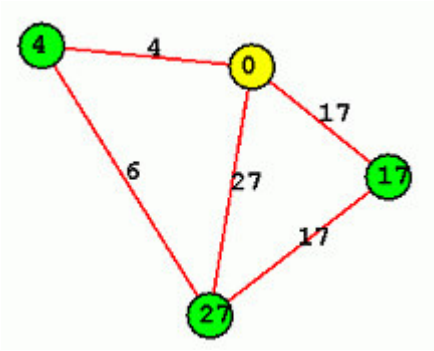
Poniżej zostanie przedstawiony krótki przykład obrazujący działanie algorytmy Dijkstry na podstawie apletu umieszczonego w naszym podręczniku. Przyjmijmy, że mamy graf, które składa się z czterech wierzchołków i pięciu krawędzi – taki, jak na poniższym rysunku:



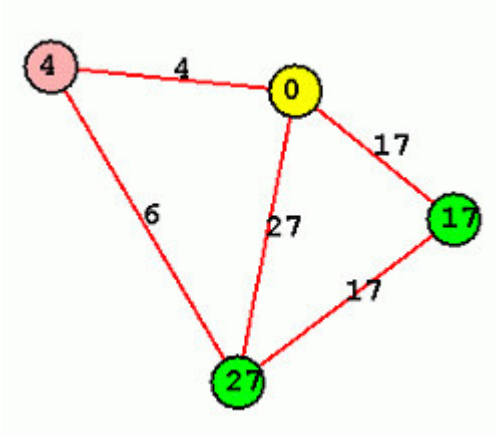
Następnie ustalamy jeden z wierzchołków jako węzeł źródłowy (ten z wartością 0 w środku):



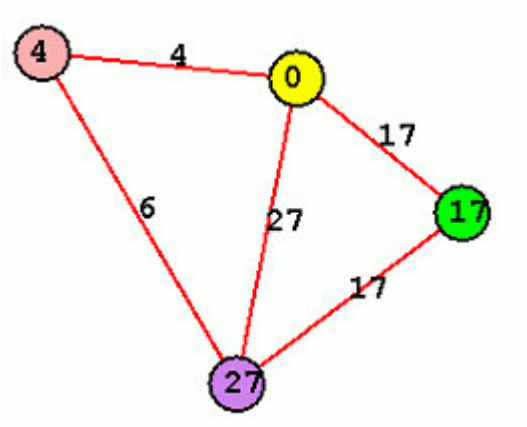
Zgodnie z algorytmem węzeł startowy trafia do zbioru Q i ponieważ jest on jedynym wierzchołkiem w tym zbiorze, wybieramy go i przenosimy do zbioru S. Następnie przeglądamy wszystkich jego trzech sąsiadów, dla których zostają wyznaczone aktualnie znane najkrótsze ścieżki. Węzły te przyjmują kolor zielony. Natomiast węzeł źródłowy został pomalowany na kolor żółty.



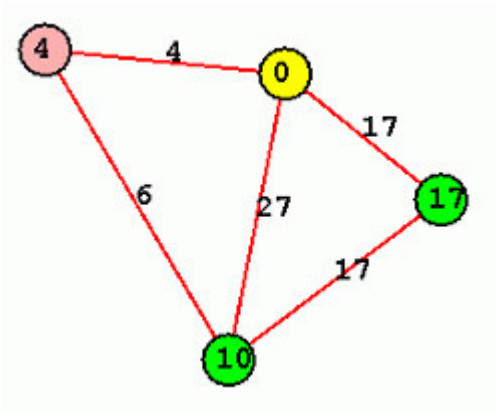
W tym momencie w zbiorze Q znajdują się 3 węzły. Wybieramy ten węzeł, który ma najmniejszą wartość wpisaną w kółko obrazujące dany wierzchołek. Algorytm wybiera zatem węzeł o wartości D=4 (znajdujący się skrajnie po lewej stroni) – krok ten jest zobrazowany pomalowaniem węzła na kolor jasnoróżowy:



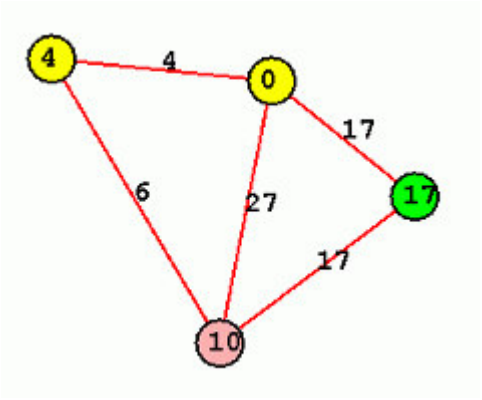
Następnie należy przejrzeć wszystkie sąsiednie węzły wierzchołka jasnoróżowego, które nie należą do zbioru S (a zatem nie są koloru żółtego). Okazuje się, że jest jeden taki węzeł – o wartości D=27. Przeglądanie sąsiada i sprawdzenie, czy można dokonać relaksacji, „odnotowywane" jest pomalowaniem go na kolor fioletowy:



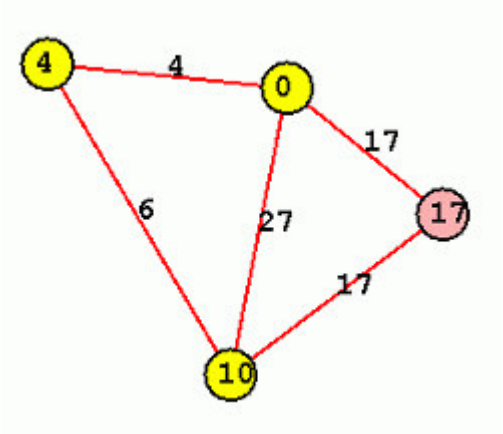
Jak widzimy, ścieżka prowadząca przez węzeł „4" do węzła o etykiecie „27" jest krótsza niż znana do tej pory. Zachodzi więc relaksacja krawędzi łączącej te dwa wierzchołki i uaktualnienie (zmniejszenie) wartości D.



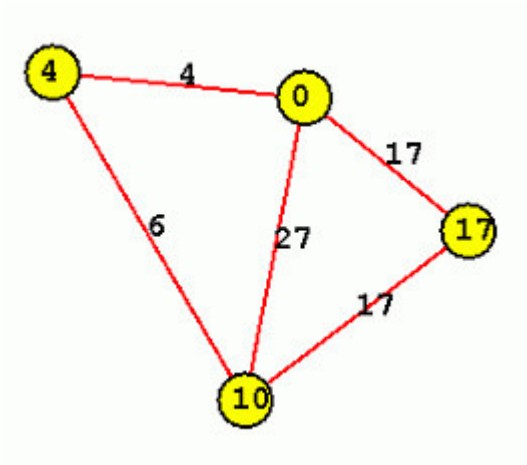
Ponieważ wszyscy sąsiedzi zostali już sprawdzeni, wierzchołek o wartości D=4 staje się węzłem żółtym. W kolejnym kroku pętli algorytmu wybieramy węzeł o etykiecie „10” (ponieważ 10<17).



Aktualny węzeł jasnoróżowy posiada tylko jednego sąsiada w kolorze innym niż żółty (jest nim węzeł o wartości D=17). Ponieważ ścieżka ze źródła do węzła „17” prowadząca przez węzeł „10” nie jest krótsza niż najkrótsza ścieżka znana do tej pory (gdyż 10+17=27 nie jest mniejsze niż 17), więc nie następuje relaksacja i uaktualnienie wartości D dla wierzchołka o etykiecie „17”. Węzeł „10” jest przenoszony do zbioru S, a następnie zostaje wybrany spośród zbioru Q węzeł o etykiecie „17”.

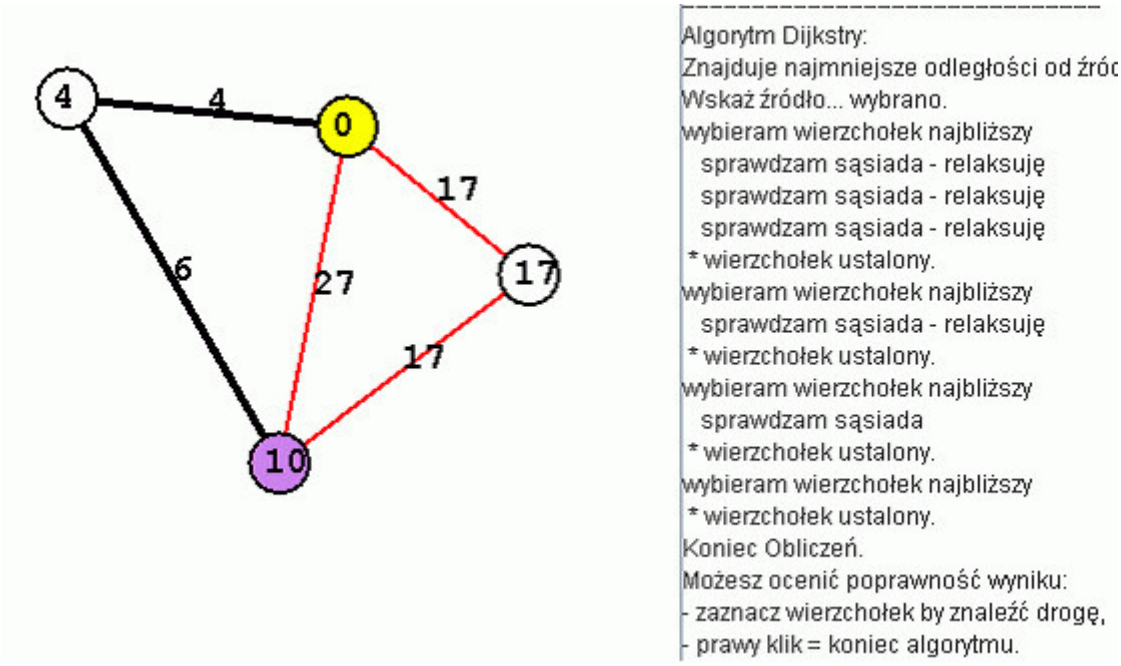


Ponieważ wszyscy jego sąsiedzi mają znalezione (ustalone) najkrótsze ścieżki ze źródła, nie dokonujemy relaksacji żadnej z krawędzi. Węzeł o wartości D=17 również trafia do zbioru S – najkrótsza ścieżka prowadząca do niego ze źródła ma wartość (koszt) równy 17.



Jak łatwo zauważyć, nie ma już węzłów w kolorze zielonym – zbiór Q stał się pusty. Tym samym wszystkie najkrótsze ścieżki ze źródła do pozostałych wierzchołków grafu zostały wyznaczone. Potwierdza do graficznie schemat grafu, gdyż wszystkie węzły są w kolorze żółtym.

Na koniec możemy prześledzić przebieg najkrótszych ścieżek ze źródła do pozostałych wierzchołków grafu. W tym celu należy kliknąć LPM na konkretny węzeł. Poniższy rysunek zawiera zaznaczoną najkrótszą ścieżkę z węzła początkowego do wierzchołka o etykiecie „10”.



Algorytm A*

Algorytm A* (*A-star*, *A-gwiazdka*) należy do grupy algorytmów przeszukujących graf, przed którymi postawiono zadanie znalezienia najkrótszej ścieżki między dwoma węzłami grafu. Został on opracowany przez P. Harta, N. Nilssona oraz B. Raphaela w 1968 r. I choć algorytm ten ma już swoje lata, to jest nadal podstawową metodą używaną w grach komputerowych (nawet tych technologicznie zaawansowanych) do wyznaczania tras przejścia obiektu między dwoma punktami w przestrzeni gry. Natomiast próżno go szukać w klasycznych podręcznikach z dziedziny algorytmów i struktur danych. Postanowiliśmy go Wam więc tu pokazać.

Przedstawiony w poprzednim rozdziale algorytm Dijkstry rozwiązuje identyczny problem, co algorytm A*. Problem jednak w tym, że podczas poszukiwania najkrótszej ścieżki algorytm Dijkstry przechodzi i sprawdza stosunkowo dużą liczbę węzłów grafu. Naukowcy zajmujący się tworzeniem nowych pomysłów w tej dziedzinie postanowili więc stworzyć „coś szybszego”. Wymyślili A* - ten algorytm również znajduje optymalną ścieżkę, ale przechodzi przez znacznie mniejszą liczbę węzłów grafu, co znacząco wpływa na jego efektywność.

Zanim szczegółowo przedstawimy zasadę działania algorytmu A*, pozwolimy Wam pobawić się apletem, który służy do jego ilustracji. Z racji tego, że algorytm ten jest często wykorzystywany w grach komputerowych i zagadnieniach sztucznej inteligencji, przyjmiemy założenie, że będziemy omawiać go na przykładzie planszy dwuwymiarowej składającej się z kwadratowych pól, będących odpowiednikami węzłów grafu. Należy jednak wyraźnie podkreślić, że jest to tylko forma prezentacji tego algorytmu, który sam w sobie ma postać równie ogólną, co algorytm Dijkstry.

A* na planszy

Załóżmy więc, że mamy planszę jak poniżej, gdzie zielony kwadrat określa pole startowe, natomiast kwadrat czerwony jest polem, do którego chcemy dotrzeć. Tutaj plansza jest pusta, ale mogą być na niej najrozmaitsze przeszkody (czyli „nieczynne kwadraty”), przez które nie można poprowadzić ścieżki – przykładowo w grach mogą one reprezentować takie elementy, jak ściany, rzeki, inni gracze itp..

Algorytm A* szuka najkrótszej drogi łączącej pole startowe z docelowym, zaczynając od sprawdzenia pól (jakie są w otoczeniu punktu, w którym znajduje się algorytm oraz które jeszcze nie były rozpatrywane), przez które prowadzą potencjalnie najbardziej obiecujące drogi do celu. Jest to zachowanie właściwe dla algorytmów typu „Best-first search”, opartych na zasadzie rozpatrywania w pierwszej kolejności (potencjalnie) najwartościowszych przypadków. Jednakże algorytm A* nie posiada cech algorytmu zachłannego, gdyż przy poszukiwaniu najkrótszej drogi uwzględnia on długość (koszt) drogi od punktu startowego do punktu aktualnie rozpatrywanego.

Do dalszych rozważań niezbędne jest wprowadzenie dodatkowych parametrów charakteryzujących problem. Otóż każdemu polu w przestrzeni przypisane są 3 parametry liczbowe:

- G - długość (koszt) ścieżki prowadzącej z punktu startu do aktualnej, rozpatrywanej pozycji w przestrzeni (jest to rzeczywista długość ścieżki, którą już wygenerowaliśmy)
- H – szacunkowa długość ścieżki prowadząca z aktualnej pozycji do punktu docelowego; wartość H jest najczęściej wyznaczana metodami heurystycznymi, gdyż z oczywistego względu nie znamy tej długości (gdyby tak było, użycie takiego algorytmu byłoby niepotrzebne)
- F = G+H – wartość równa sumie długości dwóch powyższych ścieżek.

W najprostszym ujęciu przyjmuje się, że długość (koszt) przejścia z jednego kwadratu do drugiego jest liczony w jednostkach pól jako odległość euklidesowa (dla uproszczenia obliczeń zakłada się zwykle, że jeśli bok kwadratu=10, to jego przekątna ma długość równą 14). Oczywiście koszt przejścia między węzłami grafu może być różny – w grach występują w wielu miejscach trudniejsze trasy niż normalna droga, takie jak przekroczenie rzeki, ścieżka prowadząca przez góry. Dlatego też w celu zróżnicowania tras przejścia definiuje się różne koszty przejścia między polami na mapie.

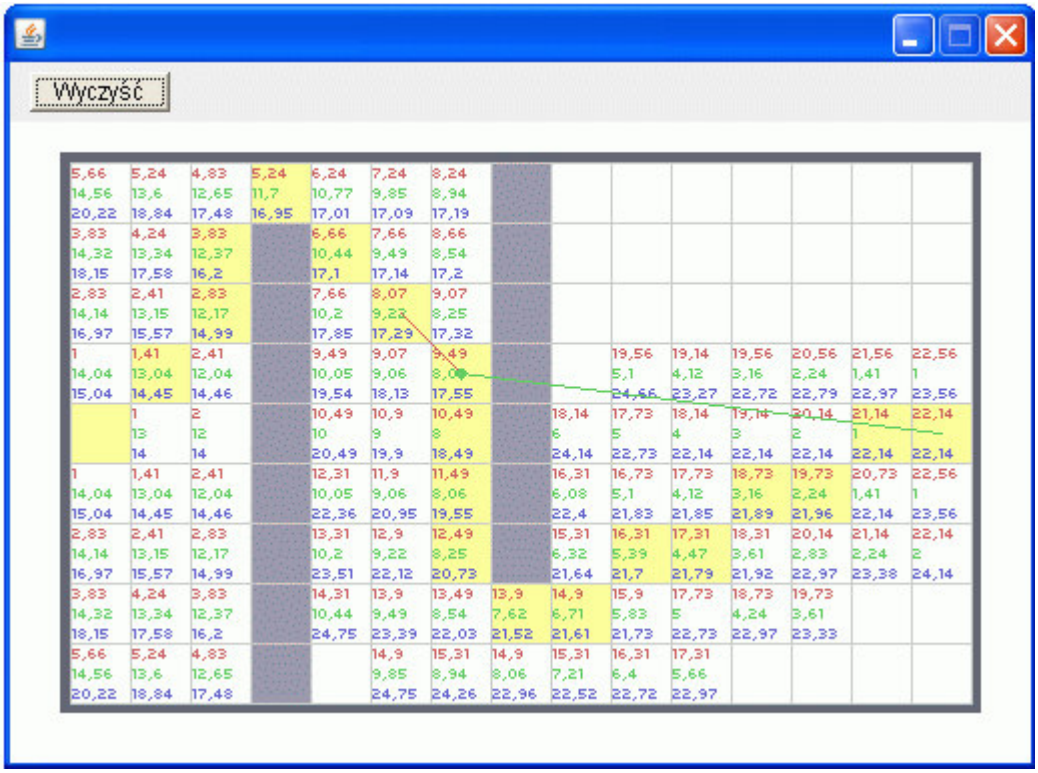
Zanim przejdziemy do omówienia szczegółow algorytmu, działanie jego warto prześledzić z wykorzystaniem apletu, do którego się wchodzi klikając w poniższy obrazek, ilustrujący stan planszy w dowolnym momencie. Przeszkody są szarymi kwadratami, a najkrótsza ścieżka między lewym punktem startowym a prawym docelowym podświetlana jest na żółto.

Oprócz tego w niektórych kwadratach – polach planszy, widnieją trzy wartości liczbowe:

- czerwona – oznaczająca wartość funkcji G dla danego węzła
- zielona – określająca wartość funkcji H dla danego węzła

- niebieska – wartość funkcji F, jest sumą dwóch poprzednich wartości

Należy podkreślić, że algorytm A* wymaga rozpatrzenia (sprawdzenia) tylko tych pól, które zawierają te trzy wartości, natomiast przez całą resztę pustych pól planszy algorytm nie przechodzi.



Funkcje heurystyczne

W algorytmie A* niezmiernie ważną rzeczą jest wybranie odpowiedniej **funkcji heurystycznej** (zwanej często w żargonie informatyków i graczy *heurystyką*, co jest oczywiście niepoprawne, bo heurystyka to metoda postępowania) czyli funkcji h, która oblicza wartości parametru H dla poszczególnych punktów przestrzeni. Zastosowanie takiej funkcji h, która dla każdego pola przestrzeni nie doszacowuje faktycznej najkrótszej odległości pola od celu (czyli posługuje się taką wartością odległości od celu, że zawsze jest ona nie większa, niż odległość faktyczna), **gwarantuje znalezienie optymalnego rozwiązania**. W miarę wzrostu jakości oszacowania szybkość działania algorytmu A* rośnie (sprawdzana jest mniejsza liczba węzłów), natomiast w momencie, gdy zaadoptujemy funkcję heurystyczną przeszacowującą rzeczywistą odległość pól przestrzeni, to algorytm jeszcze bardziej ograniczy liczbę przeszukiwań (a tym samym ograniczymy czas jego działania), natomiast może to grozić nieznalezieniem rozwiązania optymalnego (choć otrzymany rezultat będzie najczęściej suboptymalny). Dlatego przy rzeczywistym zastosowaniu tego algorytmu np. w grze komputerowej warto zastanowić się, czy ważniejsza jest dla nas szybkość obliczeń (by był krótki czas oczekiwania na poruszenie się obiektu w grze), czy dokładność w wyznaczaniu najkrótszej drogi.

W najprostszych implementacjach najczęściej używa się następujących funkcji heurystycznych:

- typu Manhattan (odległość dwóch węzłów to suma ich odległości w pionie i w poziomie – „poruszamy się po ulicach") - nie ma ona własności niedoszacowania, ale kosztem pewności wyniku przyspiesza znacznie obliczenia
- oszacowanie odległości dwóch węzłów przez obliczenie standardowej euklidesowej ich odległości przestrzeni – taka funkcja zapewnia niedoszacowanie wartości odległości od celu (nie ma drogi krótszej, niż odcinek łączący dwa punkty) i tym samym znalezienie optymalnego rozwiązania.

Zasada działania

W celu realizacji algorytmu tworzy się listę pól (węzłów) otwartych (**OL**) oraz listę pól zamkniętych (**CL**). Do listy CL w trakcie wykonywania algorytmu będą trafiały pola, którymi nie będziemy się już dalej zajmować, natomiast do listy OL wstawiane będą węzły, które w przyszłości będą jeszcze wymagały rozpatrzenia. Na początku obie listy OL i CL będą zbiorami pustymi. W pierwszym kroku dodajemy pole początkowe do listy pól otwartych. Dalsza część algorytmu (wraz ze wspomnianymi przed chwilą krokami) przebiega następująco:

Procedura **AStar** (graf **A**, węzeł **początkowy**, węzeł **końcowy**)

```
1.  inicjalizuj OL, CL
2.  dodaj węzeł początkowy do OL
3.
4.  while (lista OL nie jest pusta) {
5.      wybierz ze zbioru OL pole o najmniejszej wartości F (nazwijmy je Q)
6.      umieść pole Q na liście CL
7.      if (Q jest węzłem docelowym)
8.          znaleziono najkrótszą ścieżkę, wyjście z funkcji
9.      for (każdy z 8 sąsiadów Q) {
10.         if (sąsiad jest na CL lub sąsiad jest zabronionym polem)
11.             nic nie rób
12.         else if (sąsiad nie znajduje się na OL) {
13.             przenieś go do OL
14.             Q staje się rodzicem sąsiada
15.             oblicz wartości G, H, F sąsiada
16.         }
17.         else {
18.             oblicz nową wartość G sąsiada
19.             if ( nowaG < G) {
20.                 Q staje się rodzicem sąsiada
21.                 G = nowaG
22.                 oblicz nową wartość F sąsiada (F=nowaF)
23.             }
24.         }
25.     }
26. }
27.  /
    / jeśli znaleźliśmy się w tym miejscu - to znaczy, że przejrzeliśmy wszystkie dostępne po
    la
28. /
    / wychodzimy tutaj z procedury, ścieżka między polem startowym a docelowym nie istnieje
    (żadna)
```

Teraz jeszcze przedstawimy garść uzupełniających wyjaśnień do wyżej przedstawionego pseudokodem algorytmu. Każde pole (węzeł) ma sąsiadujące pola w pionie, w poziomie oraz po ukosie, czyli ma 8 sąsiadów, których należy sprawdzić. Węzeł Q jest wybierany jako ten spośród listy pól OL, po którym można oczekiwać najkrótszego dotarcia do celu – ma najmniejszą wartość F, więc ścieżka prowadząca przez niego wydaje się w tej chwili najwartościowsza. W rzeczywistych implementacjach algorytmu A* lista OL jest najczęściej posortowana – względem rosnącej wartości F, a sama lista przechowywana jest w strukturze sterty binarnej, co daje wysoką efektywność algorytmu.

W momencie rozpatrywania sąsiadów Q obliczamy ich parametry i ewentualnie uaktualniamy, jeśli znaleźliśmy właśnie krótszą ścieżkę do sąsiada niż do tej pory wyznaczona (teraz najkrótsza trasa będzie prowadzić przez pole Q). Po tym kroku wszyscy sąsiedzi Q będą się znajdować na liście OL i oczekiwać na ewentualną dalszą poprawę ścieżki dotarcia do nich. Proces przeglądania pól kończy się, gdy na liście OL nie będzie żadnych węzłów grafu - oznaczać to będzie, że przeanalizowaliśmy już wszystkich sąsiadów i „sąsiadów sąsiadów", a żadnego innego węzła nie pominęliśmy – być może są takie pola, do których nie można dotrzeć i tym samym wyznaczyć ścieżki. W pesymistycznym przypadku może się okazać, że jednym z takich „niedostępnych" pól jest między innymi pole docelowe – wówczas niestety nie ma do węzła końcowego ani jednej ścieżki z pola startowego.

Algorytm A* kończy się również w przypadku, gdy z listy OL wybrany zostanie (i przeniesiony do CL) węzeł Q, który jest węzłem docelowym. W tym momencie najkrótsza ścieżka ze źródła do celu została znaleziona – jest zapisana w postaci listy powiązań węzłów należących do tej ścieżki z ich ojcami – poczynając od pola końcowego, przechodząc przez wszystkich ojców dotrzemy do węzła startowego, odtwarzając tym samym najkrótszą trasę łączącą oba pola (w implementacji algorytmu każdy węzeł musi zawierać miejsce na zapisanie informacji o powiązaniu z jego ojcem).

Należy podkreślić, że zaprezentowany tu algorytm, używający zbioru węzłów zamkniętych (co teoretycznie nie zawsze jest konieczne), wymaga, by zastosowana funkcja heurystyczna była funkcją spójną (ang. *consistent*), lub inaczej monotoniczną, czyli by dla każdej pary węzłów ojciec-syn była prawdziwa nierówność **F(ojciec) <= F(syn)**. Warunek ten oznacza, że w miarę zbliżania się do celu coraz dokładniej wiemy, jaki może być rzeczywisty koszt najkrótszej ścieżki od punktu początkowego do docelowego (oszacowanie jest coraz precyzyjniejsze, choć nadal jest spełniony warunek niedoszacowania gwarantujący znalezienie optymalnej ścieżki). Własność ta, co ciekawe, jest zarazem nierównością trójkąta, zawsze spełnioną w przestrzeni euklidesowej. Można to sprawdzić korzystając z apletu i udowodnić korzystając z definicji funkcji F jako sumy G i H (co pozostawiamy Wam jako ćwiczenie).

Podsumowanie

Algorytm A* należy traktować jako uogólnienie szeroko znanego (i również stosowanego) algorytmu Dijkstry – w którym nie stosuje się funkcji heurystycznej do określenia przypuszczalnej odległości do celu (dla wszystkich węzłów wartość H=0). Dlatego też algorytm

Dijkstry średnio przeszukuje daleko więcej węzłów grafu niż algorytm A*, co czyni go istotnie wolniejszym i zarazem mniej efektywnym. Jednak w przypadku, gdy nie mamy informacji o położeniu punktu docelowego (to znaczy mamy kilka równie wartościowych pól docelowych i chcemy dotrzeć do najbliższego z nich) – wówczas algorytm Dijkstry znacznie „sprawniej" znajdzie ten najbliższy punkt docelowy niż algorytm A*.

Podsumowując, algorytm A* jest algorytmem znajdującym ścieżkę punktu startowego do punktu docelowego dla każdego problemu, jeśli przynajmniej jedna taka ścieżka istnieje. Dodatkowo, jeśli funkcja h używana do oszacowania odległości węzła od celu jest funkcją heurystyczną niedoszacowującą, to:

- algorytm A* znajduje najkrótszą (optymalną) trasę łączącą punkt startowy z punktem końcowym (docelowym)
- algorytm A* przechodzi (sprawdza) mniejszą ilość węzłów (np. pól przestrzeni) niż jakikolwiek inny algorytm używający tej samej funkcji h.

Złożoność obliczeniowa omawianego algorytmu wyszukiwania najkrótszej ścieżki jest różna – przy pesymistycznym układzie danych i źle dobranej funkcji h liczba odwiedzanych węzłów jest wartością proporcjonalną wykładniczo do długości rozwiązania (czyli najkrótszej ścieżki). Jednakże udowodniono, że jeśli odpowiednio dobierzemy funkcję heurystyczną h, czyli w taki sposób, by odchylenie wartości h od faktycznej odległości od celu było nie większe niż logarytm z liczby określającej wartość tej odległości, to w takim przypadku złożoność obliczeniowa algorytmu (liczba przeglądanych węzłów) należy do klasy wielomianowej. W skrócie można to ująć, że im dokładniejsze oszacowanie h, tym algorytm A* wykonuje się szybciej.

Przy okazji warto też wspomnieć o jeszcze jednej własności algorytmu A*. Otóż wielokrotnie zdarza się, że wyznaczona najkrótsza ścieżka ma kształt, moglibyśmy powiedzieć, nieestetyczny – z nadmiarową zmianą kierunku ścieżki. Gdybyśmy wyobrazili sobie grę z takim sposobem poruszania się obiektów, to stwierdzilibyśmy, że zachowują się one w sposób sztuczny.

Są dwa rodzaje metod „poprawiania" kształtu najkrótszych ścieżek:

- poprzez dobór współczynników kosztu ścieżek w momencie wyszukiwania najkrótszej trasy
- poprzez poprawę już wyznaczonej najkrótszej ścieżki.

W pierwszym wariancie stosuje się zabieg polegający na dodawaniu chociażby niewielkiej kary do kosztu G – w przypadku, gdy następuje zmiana kierunku trasy – w ten sposób zapewniamy preferowanie ścieżek o prostych kształtach. Natomiast drugi rodzaj metod obejmuje sposoby prostowania już wyznaczonej ścieżki – najczęściej używa się metody punktów widoczności, w której rozpatruje się kolejne punkty ścieżki i sprawdza się „patrząc" do przodu, czy nie widzimy jakiegoś innego punktu znajdującego się na ścieżce – wtedy nowa trasa prowadzi prostą linią między tymi dwoma polami (oczywiście z nowymi punktami pośrednimi, a pomijając wcześniej wyznaczone pośrednie węzły).

Niezależnie od sposobu poprawiania ścieżek, może się zdarzyć, że będą one posiadały nieraz ostre zakręty – wówczas warto zastosować ich wygładzanie. Często używanym sposobem jest wykorzystanie krzywych sklejanych.

Na zakończenie należy podkreślić, że algorytmem A* wkroczyliśmy niepostrzeżenie i całkiem nieformalnie w dziedzinę sztucznej inteligencji. Mamy nadzieję, że zachęci to Was do studiowania obszernej i fascynującej wiedzy w tym zakresie, nie będącej przedmiotem niniejszego podręcznika. Na początek zapraszamy po raz kolejny do serwisu **ważniak.mimuw.edu.pl**, tym razem do podręcznika Sztuczna inteligencja. Tam też możecie poczytać o algorytmie A*, a także odnaleźć rozdział **o grach dwuosobowych**.

Powracając zaś do samej A-gwiazdki, warto odnotować ciekawy, przejrzyście napisany artykuł A* Pathfinding for Beginners. Czytając go można się wspomagać polskim tłumaczeniem. No i jeszcze warto zajrzeć na angielską stronę Wikipedii.

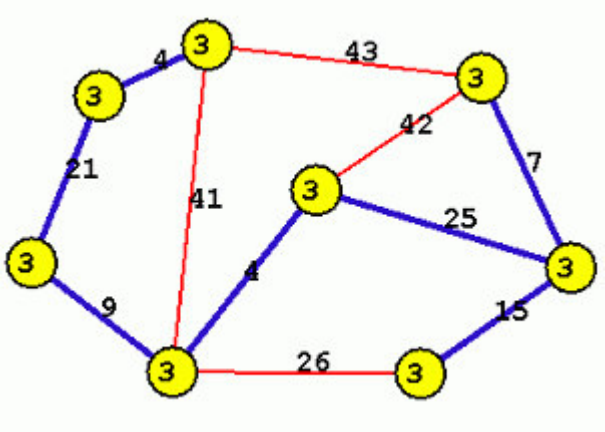
Minimalne drzewa rozpinające

W rozdziale wstępnym, wprowadzającym do zagadnień związanych z grafami, wspomnieliśmy, że jednymi z najistotniejszych problemów towarzyszących strukturom grafowym są zagadnienia znajdowania **minimalnego drzewa rozpinającego**. Najpierw należy pokrótce wyjaśnić, co się kryje pod tym hasłem drzewo rozpinające grafu. Otóż jest to zbiór V wszystkich wierzchołków grafu wraz z podzbiorem zbioru krawędzi E, które razem tworzą drzewo - a zatem brak jest **cykli**, czyli niezerowych ścieżek, w których wierzchołkiem początkowym i końcowym jest ten sam węzeł. Natomiast minimalne drzewo rozpinające (ang. *Minimum Spanning Tree*, MST) jest to takie drzewo rozpinające, które w przypadku grafu ważonego posiada najmniejszą z możliwych całkowitą sumę wag wszystkich krawędzi. Ponieważ każda krawędź łączy dokładnie dwa wierzchołki i poszukujemy drzewa rozpinającego, to do połączenia ze sobą n wierzchołków wystarczy nam dokładnie n-1 krawędzi. Dotyczy to grafu spójnego, w przeciwnym razie wystarczy mniej krawędzi, ale nie utworzymy wówczas jednego drzewa rozpinającego – potrzebnych będzie ich kilka.

Minimalne drzewo rozpinające grafu jest to spójny podgraf grafu spójnego, mający tę własność, że wszystkie węzły są połączone, a suma wag jego krawędzi jest najmniejsza.

W sposób formalny omawiany problem przedstawia się następująco: poszukujemy takiego drzewa rozpinającego o zbiorze T krawędzi (będącego podzbiorem E), których suma wag $\sum_{t \in T} w(t)$ będzie minimalna. W przypadku grafów nieważonych każde drzewo rozpinające jest minimalne (gdyż można przyjąć, że każda krawędź takiego grafu ma wagę jednostkową, a liczba krawędzi drzewa jest zawsze taka sama). Oczywiście może się również zdarzyć, że w niektórych grafach ważonych także istnieje kilka równorzędnych minimalnych drzew rozpinających.

W celu zobrazowania drzewa MST przedstawiamy poniższy rysunek grafu oraz minimalnego drzewa rozpinającego na tym grafie (krawędzie drzewa zaznaczone na niebiesko):



Metody znajdowania minimalnych drzew rozpinających mają bardzo szerokie i praktyczne zastosowanie we współczesnym świecie, przydatne są m.in. przy:

- minimalizacji długości tras lotniczych w celu stworzenia sieci połączeń między zbiorem miast, również dotyczy to podobnego zagadnienia, jakim jest budowa sieci autostrad
- minimalizacji długości połączeń w układach elektronicznych, sieciach komputerowych, sieciach TV kablowej itp.

Najbardziej znanymi algorytmami znajdowania minimalnego drzewa rozpinającego są powszechnie używane:

- algorytm Prima
- algorytm Kruskala

Obie powyższe metody zostały opracowane przez autorów o tych nazwiskach w latach 50-tych XX wieku. Choć obecnie są najczęściej wykorzystywane, nie są one jednak pierwszymi rozwiązaniami problemu znajdywania MST. Już w latach dwudziestych ubiegłego stulecia pierwszy efektywny algorytm wynalazł naukowiec o nazwisku Boruvka.

W naszym podręczniku zajmiemy się przede wszystkim algorytmem Kruskala, a także podamy najważniejsze wspólne cechy i różnice w porównaniu z metodą Prima (która jest zbliżona do skonstruowanego w latach 30-tych algorytmu Jarnika).

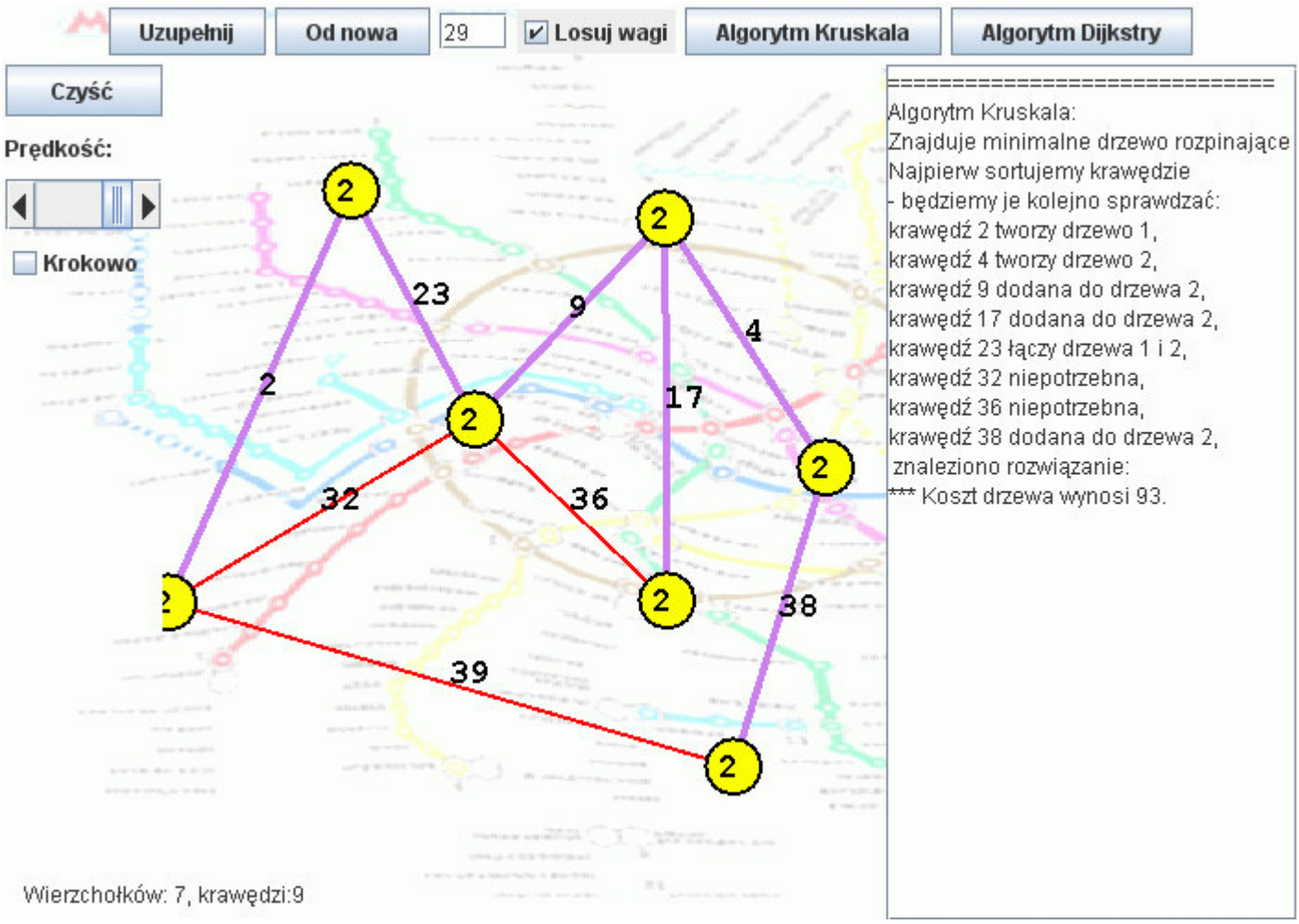
Algorytm Kruskala

Algorytm Kruskala, podobnie jak i Prima, jest metodą zachłanną. W danym kroku algorytm wybiera cząstkowe rozwiązanie możliwie najlepsze w danym momencie. Mimo że taktyka zachłanna nie zawsze gwarantuje całościowe optymalne rozwiązanie problemu, to w tym przypadku tak właśnie jest (podobna właściwość dotyczy wcześniej omawianego algorytmu Dijkstry). Ogólnie rzecz biorąc cała idea obu algorytmów opiera się na iteracyjnym doborze tych krawędzi grafu G, które są częścią końcowego minimalnego drzewa rozpinającego. W kolejnych krokach wybieramy więc po jednej krawędzi aż do momentu, gdy będziemy już posiadali zbiór krawędzi tworzący drzewo MST – możemy powiedzieć, że drzewo w kolejnych krokach „rozrasta się".

Metoda opracowana przez Kruskala jest stosunkowo łatwa do zrozumienia. Mianowicie na początku zakładamy, że posiadamy **las** (czyli **zbiór osobnych drzew**). Każdym drzewem tego lasu jest pojedynczy wierzchołek grafu G. W kolejnych krokach algorytmu odpowiednie krawędzie grafu są wybierane w celu rozrastania się lasu. W ostatnim kroku (po dodaniu n-1 krawędzi) z lasu powstaje jedno drzewo, które okazuje się być poszukiwanym minimalnym drzewem rozpinającym. A zatem na początku mamy **n drzew w lesie** (pojedynczych wierzchołków), w następnym kroku n-1 drzew, w kolejnym n-2, a po zakończeniu algorytmu **dokładnie jedno drzewo**. Podejście zachłanne odnośnie algorytmu Kruskala polega na wyborze krawędzi o najmniejszej wadze, za pomocą której połączone zostaną dwa rozłączne drzewa. Ponieważ w każdym kroku dodawana jest krawędź o najmniejsze możliwej wadze, to ostatecznie w drzewie rozpinającym suma wag tych krawędzi będzie rzeczywiście minimalna.

Działanie algorytmu możecie prześledzić przy pomocy znanego już Wam apletu. Jest to ten sam aplet, dzięki któremu mogliście wyznaczać najkrótsze ścieżki algorytmem Dijkstry. Tym samym znacie już funkcjonalność poszczególnych przycisków i suwaków, a

także sposób tworzenia, modyfikowania i usuwania grafu. Aby wyznaczyć drzewo MST, należy kliknąć na przycisk „Algorytm Kruskala". Wówczas, wraz z przebiegiem algorytmu, zostaną pokolorowane na fioletowo krawędzie dodawane do drzewa rozpinającego, a etykiety w wierzchołkach będą określały numer drzewa (dotyczy to drzew różnych od jednowierzchołkowych). W panelu komunikatów zostaną wyświetlone odpowiednie informacje, w jakiej kolejności dodawane są poszczególne krawędzie do MST.



Aplet wymaga, aby graf był spójny, w celu znalezienia właściwego rozwiązania. Jest to dosyć oczywiste, że przy niespełnieniu tego warunku nie jesteśmy w stanie utworzyć pojedynczego minimalnego drzewa rozpinającego obejmującego wszystkie wierzchołki grafu. Wówczas można jedynie utworzyć kilka drzew rozpinających – tyle, z ilu „części" składa się nasz graf.

Pseudokod algorytmu Kruskala jest bardzo prosty i daje się zapisać w poniższej postaci:

Procedura **Kruskal** (graf **G**)

```
1.  posortuj krawędzie V grafu G według niemalejącej wartości ich wag
2.  utwórz las, którego drzewami będą pojedyncze węzły grafu
3.  for (każda krawędź (u,v) z posortowanej listy)   // zaczynając od krawędzi
4.                                                   // o najmniejszej wadze
5.      if (węzły u i v należą do różnych drzew lasu)
6.          dodaj krawędź (u,v) do lasu
7.
8.       // jeśli należą do tego samego drzewa, to odrzucamy tę krawędź
9.      // gdyż wówczas powstałby cykl
10.
11.   // zbiór dodanych krawędzi wraz z wierzchołkami grafu tworzy MST
12.
```

W algorytmie wybierane są tylko te krawędzie, które łączą dwa osobne drzewa, w przeciwnym razie powstałby cykl w jednym z drzew (które przestałoby nim być) – co jest sprzeczne z założeniem znalezienia minimalnego **drzewa** rozpinającego.

Wykorzystując do implementacji algorytmu kopce binarne, można osiągnąć złożoność czasową algorytmu Kruskala rzędu **O(E lg V)**, która okazuje się wielkością zadowalającą. W podobnym czasie wykonuje się algorytm Prima, dla którego można jednak wykorzystać

kopce Fibonacciego, dzięki czemu złożoność tego algorytmu może ulec obniżeniu do wartości O(E + VlgV) – taka implementacja algorytmu Prima jest szczególnie przydatna przy grafach gęstych. Wspomnianą wartość złożoności poznaliśmy już przy omawianiu metody Dijkstry –gdyż są one do siebie podobne w budowie.

Między algorytmami Kruskala i Prima jest kilka istotnych różnic. Warto przedstawić przynajmniej najważniejsze z nich:

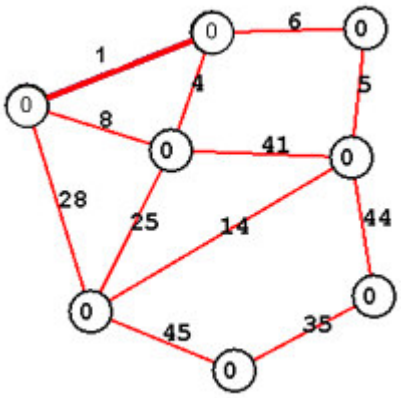
- algorytm Kruskala rozpoczyna od lasu i dopiero na samym końcu zostaje utworzone jedno drzewo rozpinające (we wcześniejszej fazie może istnieć kolekcja drzew)
- natomiast w metodzie Priama przez cały czas wykonywania algorytmu drzewo rozpinające stanowi jedną całość, co może wydawać się bardziej „estetyczne”
- w algorytmie Kruskala wybieramy „najlżejsze” krawędzie, za pomocą których możemy połączyć dwa rozdzielne drzewa
- z kolei w algorytmie Priama dołączamy do drzewa rozpinającego krawędź o minimalnej wadze, dzięki której scalimy aktualne drzewo rozpinające z jednym z wierzchołków nienależących do tego drzewa

Oczywiście istnieje również szereg podobieństw tych metod. Obie wymagają sortowania krawędzi według wag, które następnie są wybierane do drzewa w sposób zachłanny. W przypadku obu algorytmów złożoność czasowa jest taka sama, a także obie w sposób efektywny wykorzystują strukturę kopców binarnych do implementacji. Jeszcze jedną wspólną cechą jest fakt, że krawędzie dodawane do drzewa rozpinającego na pewno już w nim pozostaną (są częścią minimalnego drzewa rozpinającego) i żaden z algorytmów nie zakłada usuwania krawędzi z takiego drzewa w trakcie działania metody.

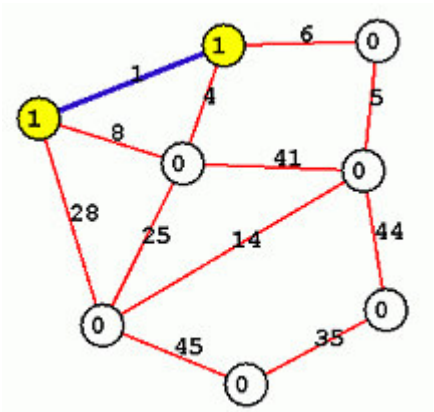
Przykład

Na koniec rozważań o sposobach wyznaczania minimalnego drzewa rozpinającego przedstawiamy krótki i zwięzły przykład, który z pewnością rozwieje wszelkie wątpliwości, w jaki sposób tworzone jest MST poprzez algorytm Kruskala.

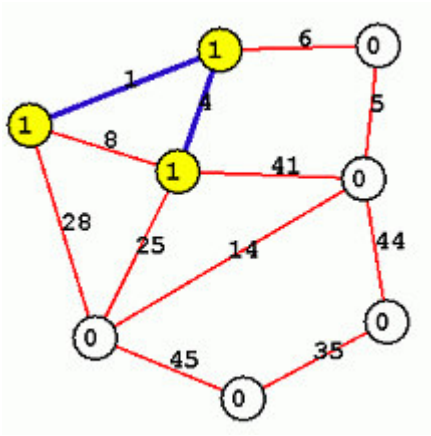
Przyjmijmy, że chcemy wyznaczyć MST dla poniższego grafu:



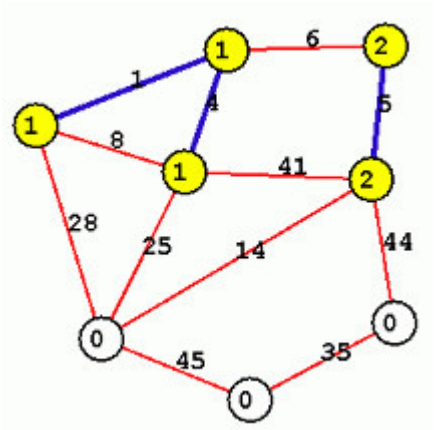
W pierwszym kroku należy posortować krawędzie grafu w porządku niemalejącym. Kolejność ta będzie przedstawiała się następująco (podajemy wagi tych krawędzi): 1, 4, 5, 6, 8, 14, 25, 28, 35, 41, 44, 45. Pamiętajmy, że każdy wierzchołek jest początkowo odrębnym drzewem. Algorytm ma za zadanie sprawdzać w kolejności sortowania poszczególne krawędzie. I tak:



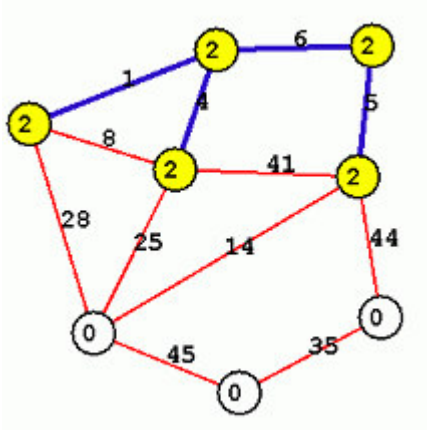
Krawędź o wadze 1 połączyła dwa rozłączne drzewa w jedno (oznaczone etykietami 1 w wierzchołkach oraz niebieską krawędzią). Następnie sprawdzamy krawędź o wadze 4:



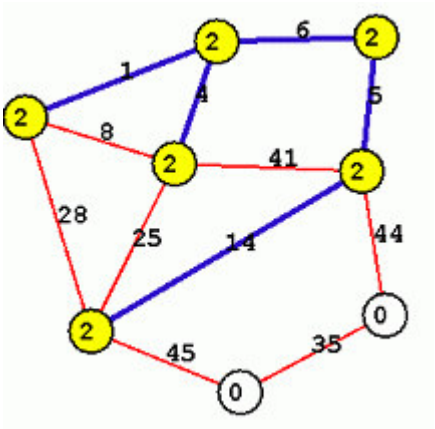
Końce tej krawędzi należą do różnych drzew, więc dodajemy ją do lasu rozpinającego. W kolejnym kroku sprawdzamy krawędź „5”:



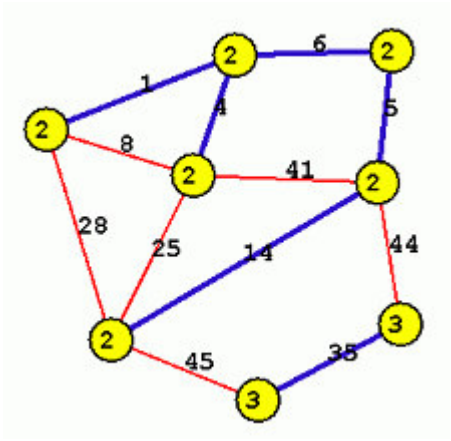
Krawędź ta łączy dwa drzewa (będące wierzchołkami) i tworzy jedno drzewo o wierzchołkach, w których widnieją etykiety o wartości 2. Dalej należy rozpatrzyć, czy krawędź o wadze 6 należy włączyć do lasu rozpinającego.



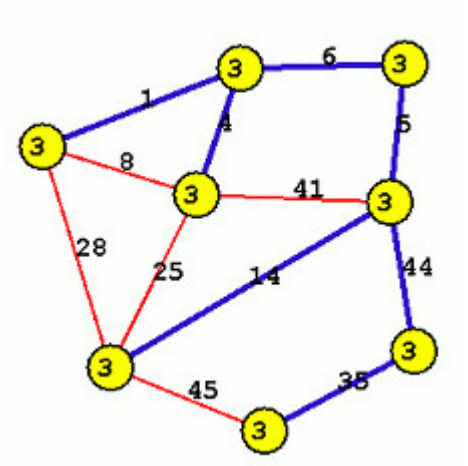
Okazuje się, że tak – krawędź „6” łączy dwa drzewa – jedno z etykietami 1, a drugie z etykietami 2. W ten sposób powstaje jedno drzewo o etykietach wierzchołków równych 2. Następna krawędź na posortowanej liście - to krawędź o wadze 8. Gdybyśmy dodali ją do drzewa rozpinającego, powstałby cykl – a zatem odrzucamy ją i sprawdzamy kolejną krawędź – „14”. Dzięki niej można połączyć drzewo „2” z innym, jednowierzchołkowym drzewem – tym samym krawędź zostaje dodana do drzewa rozpinającego.



Następne w „kolejce" do sprawdzenie są krawędzie 25 oraz 28. Dodanie każdej z nich spowoduje utworzenie cyklu w drzewie „2", a więc nie bierzemy ich pod uwagę. Algorytm przechodzi do krawędzi o wadze 35. Ponieważ łączy ona dwa wierzchołki należące do pojedynczych drzew, „akceptujemy" ją i tworzymy nowe drzewo o etykietach „3".



Przechodzimy do krawędzi „41" – utworzyłaby ona cykl w drzewie „2", więc opuszczamy ją i sprawdzamy przydatność krawędzi „44". Dzięki niej możemy połączyć drzewo „2" i „3", w związku z tym dodajemy ja do drzewa rozpinającego.



Ostatnią krawędzią na liście jest łuk o wadze 45. Nie trafia on jednak do drzewa rozpinającego, gdyż za jego sprawą powstałby niepożądany cykl.

Na tym kończy się działanie algorytmu. Powyższy rysunek potwierdza, że zostało utworzone minimalne drzewo rozpinające – wszystkie wierzchołki grafu zostały włączone do jednego wspólnego drzewa i żaden węzeł nie pozostał odosobniony. Całkowita suma wszystkich wag krawędzi należących do MST jest równa 109 i jest to minimalna wartość dla tego grafu.