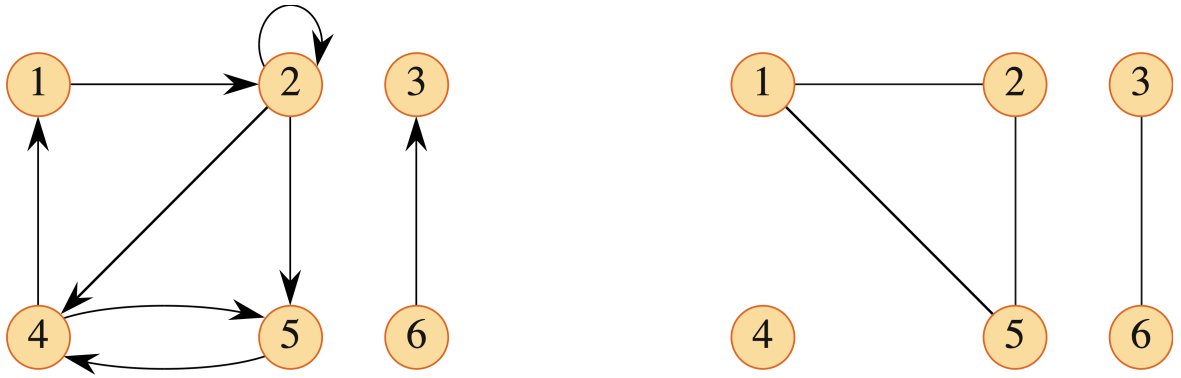
Wprowadzenie do grafów i terminologia

Graf $G = (V, E)$ jest parą składającą się ze zbioru **węzłów** (*nodes*) albo **wierzchołków** (*vertex/vertices*) V i zbioru łączących je **łuków** (*arcs*) albo **krawędzi** (*edges*) E. Rozróżniamy **grafy skierowane** (po lewej poniżej), gdzie każdy łuk ma ma początek i koniec, oraz **grafy nieskierowane** (po prawej poniżej), gdzie oba końce każdego łuku są równoprawne.



W grafach skierowanych łuk można zapisać jako parę węzłów (u, v), a więc łuk $(4, 5)$ jest różny od łuku $(5, 4)$, i w danym grafie może występować jeden z nich, lub oba. Możliwy jest również łuk z węzła do samego siebie, tzw. **pętla** (łuk $(2, 2)$ powyżej).
W grafach nieskierowanych łuki nie określają kolejności węzłów i można je zapisać za pomocą dwuelementowych zbiorów $\{u, v\}$, co wyklucza możliwość istnienia łuku z węzła do samego siebie (pętli).
W obu rodzajach grafów mogą istnieć izolowane węzły, lub grupy węzłów.

Wierzchołek może nie mieć żadnych wpadających do niego łuków (wierzchołek izolowany), lub mieć jeden lub więcej takich łuków. Liczbę łuków wchodzących do, wychodzących z wierzchołka, oraz jego pętli nazywamy **stopniem** tego wierzchołka. W grafie nieskierowanym o n wierzchołkach stopień węzła wynosi maksymalnie $n-1$. W grafie skierowanym stopień wierzchołka może wynosić maksymalnie $2n-1$.

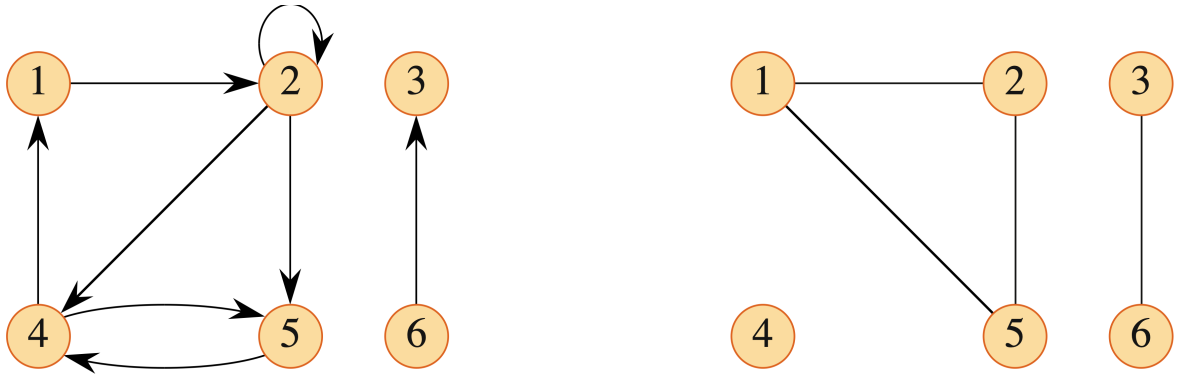
Nieskierowany graf o n wierzchołkach może mieć maksymalnie $\frac{n(n-1)}{2}$ łuków, natomiast graf skierowany maksymalnie n^2 łuków. Graf nieskierowany zawierający wszystkie możliwe połączenia między węzłami nazywa się **pełnym**, natomiast graf zawierający dużo mniej niż ta maksymalna liczba łuków nazywamy **grafem rzadkim**.

Ścieżka w grafie jest ciągiem wierzchołków połączonych łukami. W grafie skierowanym ścieżka, której wszystkie łuki podążają w jednym kierunku nazywa się **ścieżką skierowaną**. Gdy wszystkie wierzchołki ścieżki są różne to jest to **ścieżka prosta**. **Długością** ścieżki jest liczba jej łuków. Gdy w grafie istnieje ścieżka od węzła u do v to węzeł v jest **osiągalny** z u (w grafie skierowanym musi to być ścieżka skierowana).

Gdy pierwszy i ostatni węzeł ścieżki są te same, to taka ścieżka jest **cyklem**. Pętla jest przykładem cyklu długości 1. Graf bez cykli (prostych) nazywa się **acykliczny**. Skierowane grafy acykliczne stanowią specjalną klasę grafów i używany jest dla nich skrót **dag** (*directed acyclic graph*).

Graf nieskierowany jest **spójny** jeśli każdy wierzchołek jest osiągalny z każdego innego. Jeśli graf nie jest spójny, to składa się ze **spójnych składowych**, czyli podgrafów, które są spójne. Spójnymi składowymi mogą być pojedyncze węzły. Na przykład, nieskierowany graf na rysunku poniżej ma trzy spójne składowe: $\{1, 2, 5\}, \{3, 6\}, \{4\}$.

Analogicznie, graf skierowany jest **silnie spójny** jeśli dla każdej pary wierzchołków, każdy jest osiągalny z drugiego. Jeśli graf nie jest spójny, to składa się ze **silnie spójnych składowych**, czyli podgrafów, które są silnie spójne. Na przykład, skierowany graf na rysunku po lewej ma trzy silnie spójne składowe: $\{1, 2, 4, 5\}, \{3\}, \{6\}$. Można sprawdzić, że spośród wierzchołków: 1,2,4,5 każda para jest wzajemnie osiągalna. Natomiast zbiór wierzchołków $\{3, 6\}$ nie tworzy silnie spójnej składowej, ponieważ wierzchołek 6 nie jest osiągalny z wierzchołka 3.



W niektórych zastosowaniach łuki mogą mieć przypisane im **wagi**, które reprezentują jakieś atrybuty łuku, np. długość drogi, przepustowość łącza, itp.

Konwencje dla algorytmów przetwarzania grafów

Określając czasy działania algorytmów grafowych wyrażamy je względem wielkości grafu. Jednak ta wielkość obejmuje zarówno liczbę wierzchołków $|V|$ jak i liczbę węzłów $|E|$. Konsekwentnie, asymptotyczne wyrażenia O lub Θ będą jako argumenty przyjmowały jedną z tych zmiennych, lub obie. Dla celów zapisywania tych wyrażeń będziemy przyjmowali uproszczone oznaczenia $V = |V|$ i $E = |E|$. Na przykład, wyrażenie $O(VE)$ jest uproszczonym zapisem dla $O(|V| \cdot |E|)$.

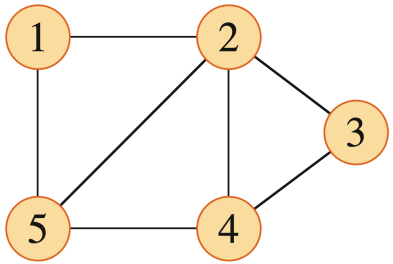
Przeszukiwanie grafu oznacza systematyczne przechodzenie wzdłuż krawędzi grafu w celu odwiedzenia wszystkich jego wierzchołków. Niektóre algorytmy mogą zaczynać pracę od przeszukania grafu w celu pozyskania informacji o jego strukturze. Niektóre inne algorytmy bazują na jakimś schemacie przeszukiwania grafu w celu wykonania swojej pracy.

Reprezentacja grafów nieskierowanych

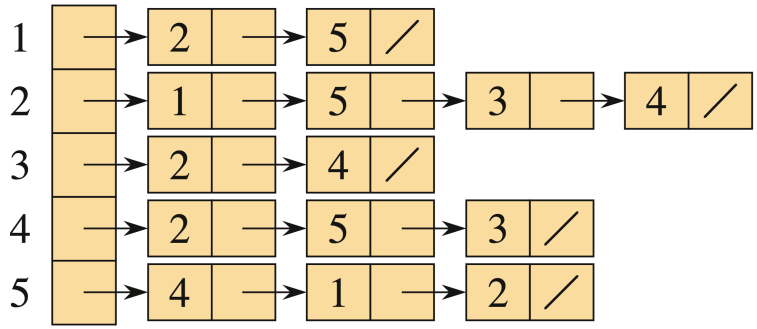
Najczęściej stosowane są poniższe dwie reprezentacje grafów.

Lista sąsiedztwa (b) dla grafu (a) poniżej, w którym wierzchołki są ponumerowane kolejno, jest tablicą o nazwie Adj o rozmiarze $|V|$, w której każda pozycja i zawiera listę numerów wierzchołków połączonych łukami z wierzchołkiem i. Zauważmy, że każdy łuk pojawia się w tych listach dwa razy, dla każdego z końcowych wierzchołków.

Macierz sąsiedztwa (c) dla tego samego grafu (a) jest tablicą dwuwymiarową o nazwie A o rozmiarze $|V| \times |V|$, w której każdy element $a_{i,j}$ jest równy 1 jeśli graf zawiera łuk (i, j), oraz równy 0 w przeciwnym wypadku. Tutaj podobnie, każdy łuk jest reprezentowany w macierzy sąsiedztwa dwukrotnie, jako (i, j) i (j, i) (z wyjątkiem łuków z węzła do samego siebie, jeśli występują). Łatwo zauważyć, że taka macierz jest zawsze symetryczna.



(a)



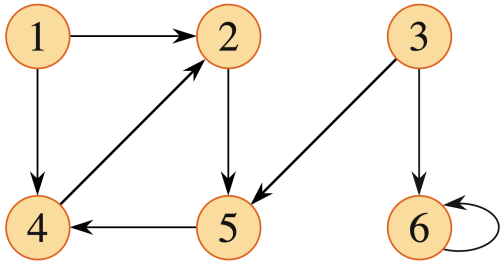
(b)

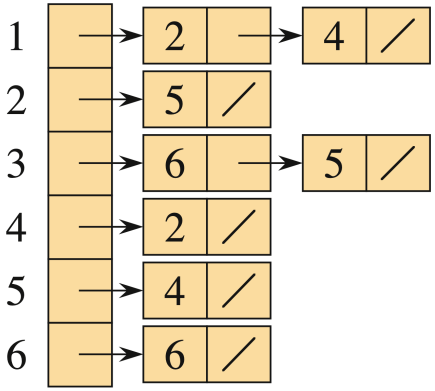
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

(c)

Reprezentacja grafów skierowanych

W przypadku grafów skierowanych stosuje się takie same reprezentacje, ale każdy łuk pojawia się tylko raz zarówno na listach sąsiedztwa, jak i w macierzy sąsiedztwa, która w tym przypadku nie jest już symetryczna.





	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Listy sąsiedztwa są ogólnie bardziej elastyczną formą reprezentacji, ponieważ pozwalają na dołączenie dowolnych informacji zarówno do poszczególnych węzłów (elementów tablicy Adj) jak i łuków (elementów list sąsiedztwa). Jednak wymagają przeszukiwania.

Macierz sąsiedztwa na pozór wykorzystuje więcej pamięci, ponieważ reprezentuje zarówno wszystkie krawędzie istniejące, jak i te nieistniejące, za pomocą wartości 0. Jednak w wielu zastosowaniach macierz jest wygodniejsza, ze względu na bezpośredni dostęp do jej elementów, niewymagający przeszukiwania list. Dodatkowo, w przypadku łuków z wagami macierz może zawierać wagi łuków, a w przeciwnym przypadku może zawierać tylko pojedyncze bity zamiast wartości 1/0.

Przeszukiwanie wszerz (BFS)

Przeszukiwanie wszerz (*Breadth-First Search*, BFS) jest jednym z najprostszych algorytmów przeszukiwania i bazą dla wielu innych algorytmów. Dla grafu $G = (V, E)$ w przeszukiwaniu BFS wyróżniony jest jeden wierzchołek s zwany **źródłem** (*source*). Celem jest dotarcie do wszystkich wierzchołków osiągalnych z s. Algorytm buduje drzewo przeszukiwania wszerz o korzeniu s, które zawiera wszystkie wierzchołki osiągalne z s, ale w odróżnieniu od oryginalnego grafu G zawiera dokładnie po jednej ścieżce do wszystkich tych wierzchołków (w grafie G tych ścieżek może być wiele).

Jednocześnie algorytm oblicza odległość od s każdego wierzchołka osiągalnego z s w postaci długości najkrótszej ścieżki od s to każdego osiągalnego wierzchołka, wyrażanej liczbą krawędzi (inaczej zwaną liczbą kroków). Algorytm systematycznie oblicza i wykorzystuje następujące atrybuty każdego wierzchołka v grafu:

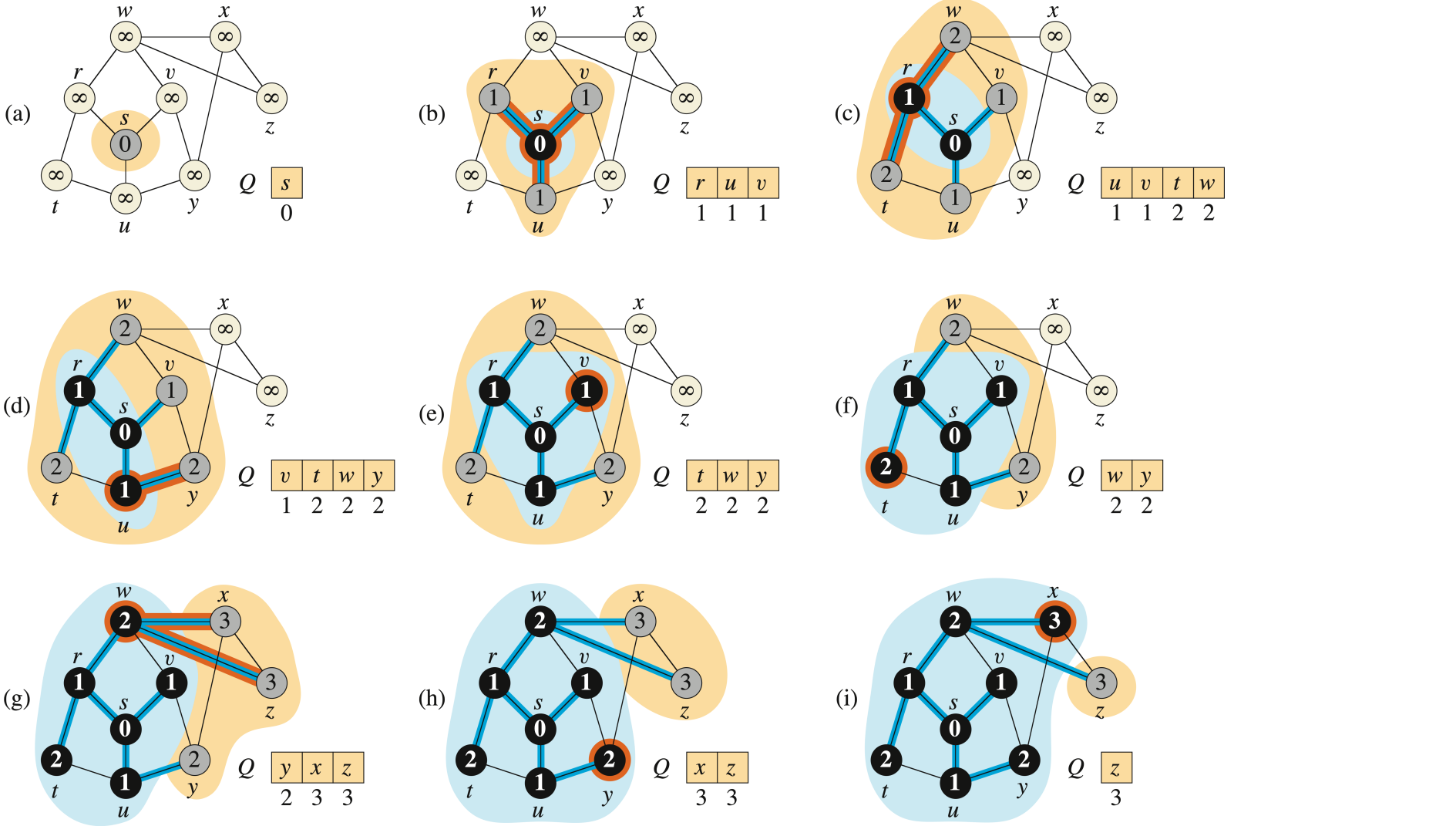
$v.color$ — jest kolorem v: BIAŁY, SZARY, lub CZARNY; początkowo wszystkie wierzchołki są białe, wierzchołki początkowo znalezione na ścieżce z s stają się szare, i ostatecznie stają się czarne gdy wszyscy ich sąsiedzi w G zostali odwiedzeni,

$v.d$ — jest odległością v od źródła s, równą $+\infty$ gdy v nie jest osiągalny z s,

$v.\pi$ — jest poprzednikiem (rodzicem) v w drzewie przeszukiwania wszerz, równym NIL gdy v nie ma poprzednika (jest nieosiągalny z s, lub jest równy s).

BFS wykorzystuje listy sąsiedztwa i działa na grafach skierowanych i nieskierowanych:

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2        u.color = WHITE
 3        u.d = ∞
 4        u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11        u = DEQUEUE(Q)
12        for each vertex v in G.Adj[u] // search the neighbors of u
13              if v.color == WHITE    // is v being discovered now?
14                    v.color = GRAY
15                    v.d = u.d + 1
16                    v.π = u
17                    ENQUEUE(Q, v)    // v is now on the frontier
18        u.color = BLACK              // u is now behind the frontier
```



Przykład przeszukiwania BFS grafu nieskierowanego. Rysunki przedstawiają stan grafu i kolejki Q na początku każdej iteracji pętli **while**. Liczby w węzłach i kolejce Q oznaczają odległości. Pomarańczowe tło wskazuje na wierzchołek właśnie usunięty z kolejki i łuki właśnie dodane do drzewa przeszukiwania. Niebieskie łuki stanowią zbudowane drzewo przeszukiwania BFS.

Nazwa algorytmu „wszerz" odnosi się do jego zachowania przypominającego równomierne rozlewanie się odwiedzonej części grafu.

Żaden węzeł w odległości d od źródła nie zostanie usunięty z kolejki Q stanowiącej granicę obszaru przeszukiwania, zanim nie będą z niej usunięte wszystkie węzły znajdujące się w odległości $< d$ od źródła.

Przeszukiwanie wszerz ma szereg interesujących własności.

Algorytm oblicza długości **najkrótszych ścieżek** od źródła do wszystkich węzłów grafu osiągalnych ze źródła (a dla nieosiągalnych określa tę długość jako ∞).

Algorytm efektywnie konstruuje drzewo przeszukiwania wszerz, za pomocą wskaźników poprzednika $v.\pi$ w każdym węźle, pozwalających odnaleźć najkrótszą ścieżkę ze źródła do każdego węzła grafu, odpowiadającą tej obliczonej najkrótszej odległości. Tę ścieżkę — w razie potrzeby — budujemy od wierzchołka docelowego v wstecz do źródła s podążając za wskaźnikami poprzednika.

Można udowodnić, że czas działania algorytmu BFS wynosi $O(V + E)$, czyli jest liniowy względem reprezentacji list sąsiedztwa.

Przeszukiwanie wgłąb (DFS)

Przeszukiwanie wgłąb (*Depth-First Search*, DFS) próbuje podążać „głębiej" w graf, jeśli jest to tylko możliwe. Zatem po odwiedzeniu węzła v algorytm podejmuje przeszukiwanie węzłów do których prowadzą krawędzie wychodzące z v, a dopiero gdy wszystkie takie węzły i wszystkie takie krawędzie zostaną w pełni zbadane, algorytm wraca do poprzednika węzła v aby badać „rodzeństwo" v.

Proces ten powtarza się tak długo, aż zostaną odwiedzone wszystkie węzły osiągalne ze źródła s. Wtedy, o ile zostały w grafie jakieś nieodwiedzone wierzchołki, to jeden z nich jest wybierany jako nowe źródło, i algorytm kontynuuje pracę.

Zatem w odróżnieniu od BFS, który buduje tylko pojedyncze drzewo przeszukiwania wszerz składające się z węzłów osiągalnych z s, DFS może zbudować **las przeszukiwania wgłąb** składający się z **drzew przeszukiwania wgłąb**, obejmujących wszystkie węzły grafu.

Ta różnica między BFS a DFS nie jest różnicą samych algorytmów, ale raczej sposobem ich ujęcia zastosowanym w podręczniku CLRS, i w konsekwencji też tutaj. Byłoby całkowicie możliwe sformułowanie algorytmów inaczej, np. na odwrót, z BFS restartującym z kolejnych źródeł, i DFS kończącym z jednym drzewem wynikowym.

Algorytm DFS w czasie pracy koloruje węzły podobnie jak BFS. Zaczyna od pokolorowania wszystkich węzłów na biało, potem zmienia na szary kolor każdego węzła odwiedzanego po raz pierwszy, i ostatecznie zmienia na czarny kolor węzła, którego przetwarzanie zostało zakończone, ponieważ algorytm zbadał wszystkie węzły z jego listy sąsiedztwa.

Ponadto, algorytm odnotowuje dla każdego węzła następujące **etykiety czasowe** (*timestamps*):

$v.d$ — jest numerem kroku obliczeń gdy węzeł v jest odkryty po raz pierwszy, i pokolorowany na szaro,

$v.f$ — jest numerem kroku obliczeń gdy przeszukiwanie kończy przetwarzanie listy sąsiedztwa węzła v, i koloruje go na czarno.

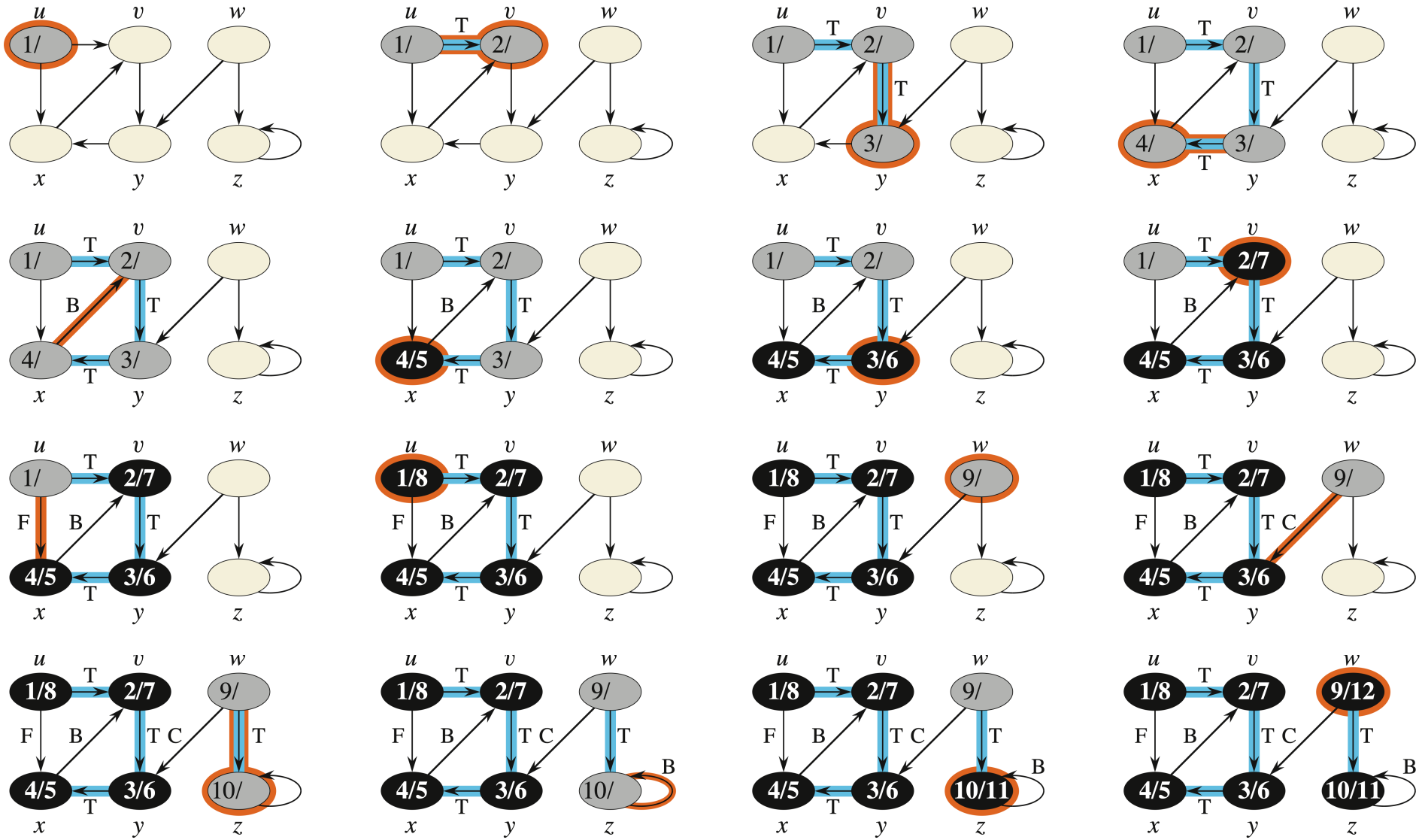
Etykiety czasowe są liczbami całkowitymi z przedziału $[1, 2|V|]$, ponieważ dla każdego węzła jest tylko jedno zdarzenie odkrycia i jedno zdarzenie zakończenia przetwarzania.

Dla każdego węzła v zachodzi $v.d < v.f$. Węzeł v jest biały przed krokiem $u.d$, szary pomiędzy $v.d$ i $v.f$, i czarny potem.

Algorytm DFS działa zarówno dla grafów skierowanych jak i nieskierowanych. Można udowodnić, że czas działania algorytmu DFS wynosi $\Theta(V + E)$, podobnie jak BFS.

```
DFS(G)
1  for each vertex u ∈ G.V
2        u.color = WHITE
3        u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6        if u.color == WHITE
7              DFS-Visit(G, u)

DFS-Visit(G, u)
 1  time = time + 1                  // white vertex u has just been discovered
 2  u.d = time
 3  u.color = GRAY
 4  for each vertex v in G.Adj[u]    // explore each edge (u, v)
 5        if v.color == WHITE
 6              v.π = u
 7              DFS-Visit(G, v)
 8  time = time + 1
 9  u.f = time
10  u.color = BLACK                  // blacken u; it is finished
```



Przykład przeszukiwania DFS grafu skierowanego: pomarańczowe podświetlenie — aktualnie analizowany łuk i węzeł, niebieskie łuki — budowany las przeszukiwania wgłąb, liczby w węzłach — etykiety czasowe. Etykiety łuków: T - łuk drzewa/lasu przeszukiwania, F/B - łuk do potomka/przodka w drzewie, C - łuk pomiędzy drzewami.

Własności przeszukiwania wgłąb

Algorytm DFS przedstawiony w powyższym pseudokodzie jest niedeterministyczny, ponieważ pętla „**for** each vertex $u \in G.V$" w wierszu 5 procedury DFS może wybierać węzły w dowolnej kolejności. Wynik działania algorytmu zależy od tej kolejności. Na przykład, dla grafu na poprzedniej stronie, gdyby przeszukiwanie DFS zaczęło się od węzła w zamiast od u to wynikiem byłyby inne dwa drzewa (drzewo $(w\ (z)\ (y\ (x\ (v))))$ oraz (u)).

Dla grafu skierowanego jest również możliwe, że jedno przeszukanie DFS danego grafu wygenerowałoby pojedyncze drzewo, a inne przeszukania las dwóch lub więcej drzew (np. rozważ różne warianty przeszukiwania DFS grafu $\textcircled{u} \longrightarrow \textcircled{v} \longrightarrow \textcircled{w} \longrightarrow \textcircled{x}$).

Algorytm przeszukiwania DFS można zapisać podobnie do przeszukiwania BFS wykorzystując stos zamiast kolejki. W ten sposób, po zakończeniu przeszukiwania danego poddrzewa algorytm wróci do ostatnio odłożonego na stos węzła, zanim wznowi przeszukiwanie węzłów zapamiętanych wcześniej. W powyższym pseudokodzie rekurencyjna implementacja procedury $\mathrm{DFS\text{-}Visit}$ również wykorzystuje stos, tylko niejawnie — jest to stos rekurencyjnych wywołań funkcji.

Literatura i materiały pomocnicze

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L Rivest, Clifford Stein: Wprowadzenie do algorytmów, PWN, 2024, dodatek B.4, i rozdział 20.