Zadanie 2 - Wersja, gdzie uwzględniana jest chronologia zdarzeń.

Chronologia rozumiana jest następująco: dla danych (2, 3), (4, 2) zwiadowca 2 najpierw przekazuje swoją informację do zwiadowcy 3, potem dopiero otrzymuje informację od zwiadowcy 4 więc zwiadowca 3 nie posiada informacji od zwiadowcy 4.

Rozwiązanie problemu: Zaprezentujmy problem jako graf, gdzie wierzchołkami będą kolejni zwiadowcy, a krawędziami relacje między nimi(kto komu przekazał informację). Będzie to graf skierowany – jeden zwiadowca przekazuje informację drugiemu, w drugą stronę tak być nie musi. Na wejściu otrzymujemy ilość zwiadowców oraz pary (nr zwiadowcy przekazującego informację, nr zwiadowcy otrzymującego informację), krawędź skierowana jest od przekazującego do otrzymującego.

Algorytm:

n – ilość zwiadowców(wierzchołki)
m – ile par zwiadowców wymieniło się informacją(krawędzie)

Do rozwiązania problemu przydatna jest pomocnicza, jednowymiarowa tablica wielkości **n**(oznaczona dalej jako **P**), w której zaznaczamy jakie wierzchołki zostały odwiedzone, a które jeszcze nie, a także Set(oznaczony jako **S**), w którym przechowywana będzie część zwiadowców, których król musi wezwać. Ci zwiadowcy oprócz własnej informacji mają dostęp do danych innych zwiadowców. Początkowo ustawiamy dla każdego zwiadowcy wartość tabeli **P** na -1. Działanie jest następujące:

1. Weź nieużywaną krawędź (**i**, **j**).
2. Sprawdź czy **i** znajduje się w **S**.
3. Jeśli tak to usuń **i** oraz dodaj **j**. W przeciwnym przypadku spróbuj dodać **j**(może już tam się znajdować).
4. Ustaw **P[i]** na **1** oraz **P[j]** na **1**;
5. Wróć do 1.
6. Gdy skończą się krawędzie zlicz ile elementów w **P** jest równych -1, następnie dodaj tę wartość do wielkości zbioru **S**.

Złożoność obliczeniowa – wzięcie wszystkich krawędzi to koszt O(m). Sprawdzanie, czy element znajduje się w zbiorze **S** to koszt O(log n). Iterowanie po tablicy **P** – koszt O(n). Odwołanie do **i**-tego elementu w **P** O(1). Sumarycznie O(m * log n + n) co daje złożoność pesymistyczną (dla m = n) O(n * log n).

Złożoność pamięciowa – tablica n-elementowa O(n), set – O(n)

Poniżej znajduje się rozwiązanie dla następujących danych wejściowych:

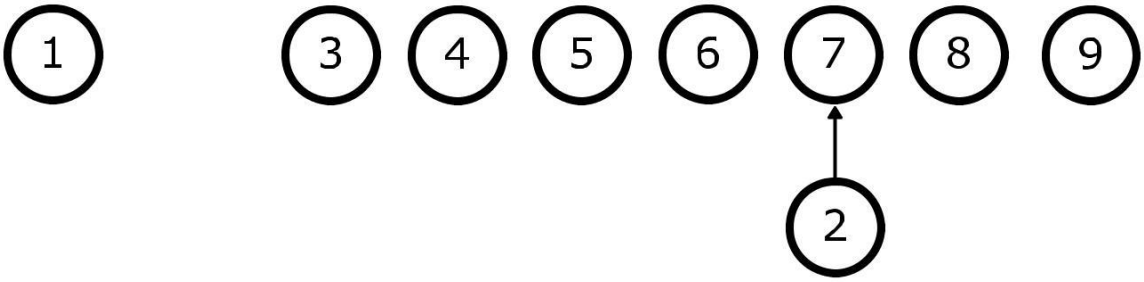
9
2 7
7 3
4 7
9 6
6 8
8 9
5 7

Przykładowe rozwiązanie:

0. Brak krawędzi

1	2	3	4	5	6	7	8	9	S = { }
-1	-1	-1	-1	-1	-1	-1	-1	-1	

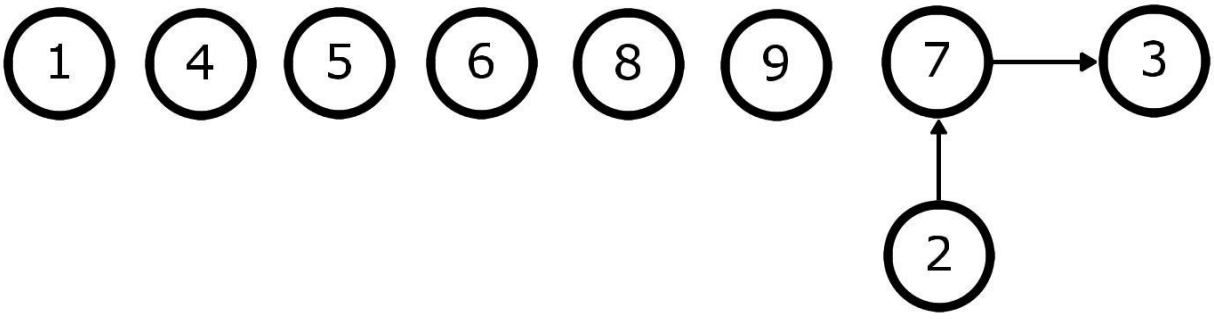
1. Wstawiam krawędź (2, 7)



Sprawdzam, czy 2 jest w zbiorze S, nie ma, więc jedynie dodaję 7. Ten zwiadowca posiada więcej niż jedną informację. Aktualizuję też tabelę P – oznaczam zwiadowców jako odwiedzonych poprzez ustawienie wartości P[2], P[7] na 1.

1	2	3	4	5	6	7	8	9	S = { **7** }
-1	**1**	-1	-1	-1	-1	**1**	-1	-1	

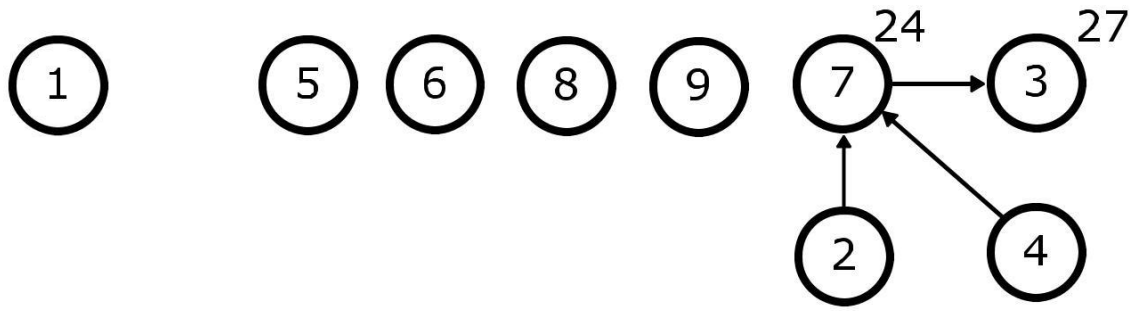
2. Krawędź (7, 3)



Sprawdzam, czy 7 jest w zbiorze, jest, zatem usuwam tego zwiadowcę ze zbioru, po czym dodaję 3 do zbioru – ten zwiadowca przejął wszelkie informacje, jakie w danym momencie posiadał 7. Aktualizuję tabelę P.

1	2	3	4	5	6	7	8	9	S = { **3** }
-1	1	**1**	-1	-1	-1	1	-1	-1	

3. Krawędź (4, 7)

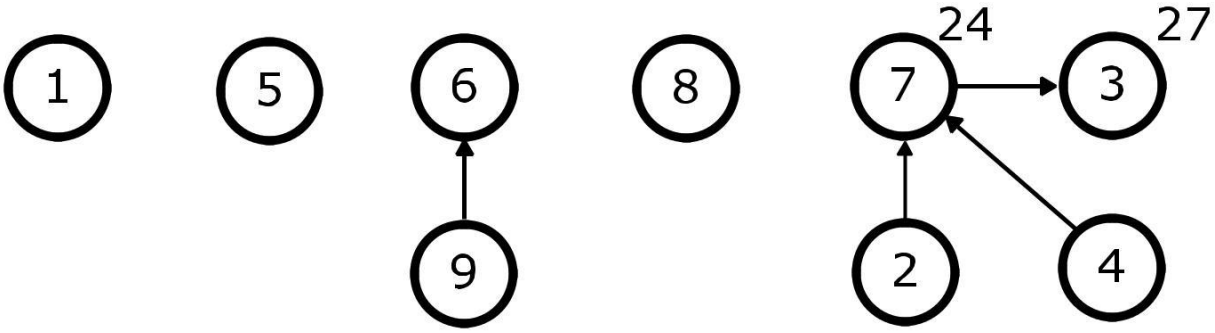


Sprawdzam, czy 4 jest w zbiorze, nie ma, zatem jedynie dodaję 7. Widać teraz, że w danej spójnej może być potrzebny więcej niż jeden zwiadowca, aby król uzyskał kompletne informacje. Aktualizuję tabelę P.

1	2	3	4	5	6	7	8	9
-1	1	1	**1**	-1	-1	1	-1	-1

S = { 3, **7** }

4. Krawędź (9, 6)

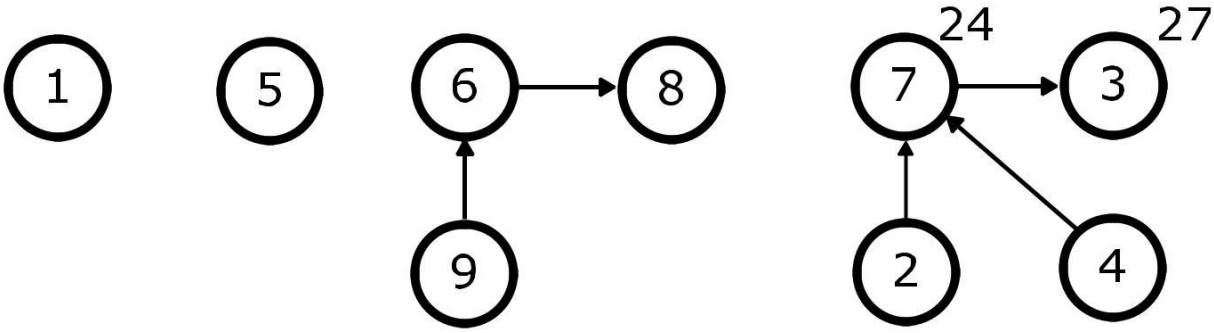


Sprawdzam, czy 9 jest w zbiorze, nie ma, zatem jedynie dodaję 6 i aktualizuję tablicę P.

1	2	3	4	5	6	7	8	9
-1	1	1	1	-1	**1**	1	-1	**1**

S = { 3, **6**, 7 }

5. Krawędź (6, 8)

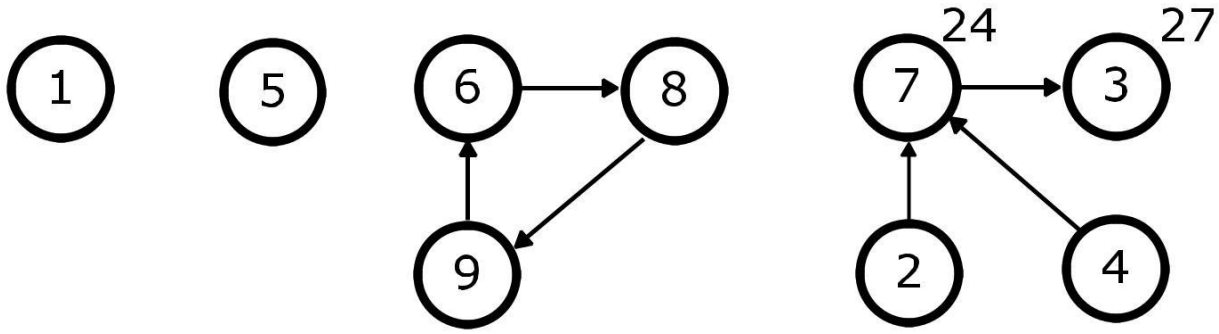


Sprawdzam, czy 6 jest w zbiorze, jest, zatem usuwam tego zwiadowcę i wstawiam 8. Aktualizuję P.

.1	2	3	4	5	6	7	8	9
-1	1	1	1	-1	1	1	**1**	1

S = { 3, 7, **8** }

6. Krawędź (8, 9)

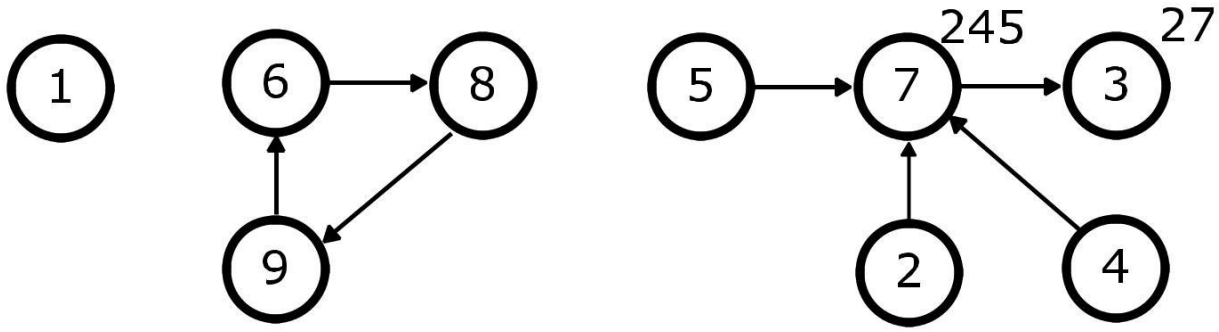


Sprawdzam, czy 8 jest w zbiorze, jest, zatem usuwam i wstawiam 9. Aktualizuję P, lecz nic to nie zmienia, gdyż 9 było już wcześniej odwiedzone.

1	2	3	4	5	6	7	8	9
-1	1	1	1	-1	1	1	1	1

S = { 3, 7, **9** }

7. Krawędź (5, 7)



Sprawdzam, czy 5 jest w zbiorze, nie ma, zatem jedynie dostawiam 7. Okazuje się, że 7 już jest, ale w secie każdy element może znajdować się tylko raz zatem nic się nie stanie. Aktualizuję P.

1	2	3	4	5	6	7	8	9
-1	1	1	1	**1**	1	1	1	1

S = { 3, 7, 9 }

Po dodaniu wszystkich krawędzi pozostała ostatnia czynność: zliczam ile wierzchołków było nieodwiedzonych(oznaczone w tabeli P przez wartość -1) i dodaję do nich wielkość zbioru S. W wyniku otrzymuję, iż król musi wezwać 4 zwiadowców, aby poznać wszystkie informacje. Są to zwiadowcy: 1, 3, 7, 9.

Wnioski po implementacji:

Wymyślając tę wersję problemu nie sądziłem, że aż tak uprości to rozwiązanie. Fakt faktem, że w drugiej opcji algorytm przeszukiwania w głąb stanowił rozwiązanie problemu także nie jest to aż tak drastyczne uproszczenie.

Największym problemem implementacyjnym było zamieszczenie wszystkich wymyślonych przeze mnie funkcjonalności w jednym projekcie. Biblioteka do testów wydajnościowych JMH tworzy nową wirtualną maszynę, w której przeprowadza testy. Utrudnia to współdzielenie danych między programami. Problem udało mi się obejść wykorzystując metodę biblioteczną, w której można zadać parametry wiersza poleceń. Przekazywałem w ten sposób nazwę pliku, na którym miały zostać przeprowadzone testy.

Wybierając Javę jako język służący do implementacji rozwiązania dodatkowo utrudniłem sobie zadanie, gdyż nie jest rzeczą łatwą stworzenie testów wydajnościowych wiernie oddających czas potrzebny na wykonanie programu. Ma na to wpływ sposób działania wirtualnej maszyny w Javie. Skonstruowana jest tak, aby w sposób wydajny obsługiwać problemy o podobnej strukturze. Podczas testów widziałem jak czas po kilku uruchomieniach programu spadał nawet o połowę. Dlatego też skonstruowałem tak testy, aby najpierw w ramach ‘rozgrzewki’ 30 razy uruchomił program. Następnie, aby zebrać względnie wiarygodne statystyki wykonuję 20 razy prawdziwe testy. JMH na końcu podsumowuje wykonane testy. Znajdują się tam informacje takie jak np. minimalny, średni, maksymalny czas wykonania oraz można ocenić rząd wielkości błędu podczas obliczeń. Ta ostatnia informacja okazała się niezwykle pomocna. W momencie gdy odnotowałem błąd około 10% zacząłem się zastanawiać co mogło go spowodować. Po wyłączeniu aplikacji działających równolegle na komputerze błąd spadł do 0.001%!

Przygotowałem też kilka testów dla danych o różnym rozmiarze, aby sprawdzić zachowanie się programu jeśli chodzi o złożoność. Zastosowanie w ramach seta struktury opartej o haszowanie umożliwiło uzyskanie złożoności liniowej. W momencie podmiany na strukturę drzewiastą zauważyłem znacznie dłuższe działanie programu, gdyż testy zostały przygotowane tak, aby do drzewa było dodawane wiele elementów przy jednoczesnym braku usuwania. Dla testów posiadających 10 milionów wierzchołków czasy różniły się o rząd wielkości. Pokazuje to jak kluczowy jest odpowiedni dobór struktury do problemu. W przypadku drzewa czas, w którym uzyskujemy dostęp do elementu jest logarytmiczny, w przypadku hasza stały.