Zaawansowane algorytmy

Projekt 2

Analiza algorytmów wyszukiwania

mostów w grafie.

Adrian Świderek

367867

Grupa 23

Treść zadania:

Zaimplementuj algorytmy znajdowania mostów w grafie:  
1) Naiwny. Dla każdej krawędzi wykonaj działanie: usuń - sprawdź spójność - dodaj  
2) Ulepszony naiwny. Wywołaj algorytm DFS i spisz krawędzie drzewowe (odpowiadające wywołaniom rekurencyjnym).  
Dla każdej krawędzi drzewowej wykonaj działania takie jak w punkcie 1).  
3) Algorytm Tarjana.  
  
Porównaj działanie w.w. algorytmów na reprezentacji listowej i macierzowej grafu.  
W tym celu wygeneruj losowe grafy o zadanej gęstości (liczba krawędzi przez liczbę wierzchołków).  
Jaki wpływ ma gęstość na czas działania tych algorytmów? Które operacje na grafach są najistotniejsze   
dla wydajności?

Opis działania programu:

W programie zostały zaimplementowane dwie reprezentacje grafów: reprezentacja macierzowa oraz reprezentacja listowa. Informacje o wierzchołkach i ich połączeniach w przypadku reprezentacji listowej przechowywane są w liście tablic a w przypadku reprezentacji macierzowej w tablicy dwuwymiarowej jako wartość boolean. Do sprawdzenia spójności grafu wykorzystywana jest klasa Components, która zlicza liczbę składowych i dzięki temu wiemy, że jeżeli po usunięciu krawędzi liczba składowych zwiększy się to krawędź ta była mostem. Algorytm naiwny zaimplementowany jest w funkcji deleteAllEdges() znajdującej się w klasach: MatrixGraph i ListGraph. Algorytm przechodzi przez wszystkie krawędzie usuwając po kolei każdą z nich, sprawdza spójność poprzez porównanie liczby składowych i z powrotem dodaje krawędź. Aby uniknąć powtórnego sprawdzania krawędzi takich jak np. 2-0, 0-2 wcześniej utworzyłem odpowiedni zbiór pojedynczych krawędzi. Ulepszony algorytm naiwny to funkcja deleteTreeEdges() znajdująca się w tych samych klasach i działa ona na dokładnie tej samej zasadzie co wcześniej opisany algorytm tyle, że w tym przypadku za pomocą klasy TreeEdgesDFS wywołujemy algorytm dfs i zapisujemy jedynie krawędzie drzewowe (odpowiadające wywołaniom rekurencyjnym dfs) a następnie sprawdzamy istnienie mostu tylko dla tych krawędzi. Generowanie losowych grafów odbywa się za pomocą klasy RandomGraphGenerator. W tej klasie tworzę grafy o podanej liczbie wierzchołków i o podanym wspólczynniku prawdopodobieństwa utworzenia krawędzi który decyduje o ostatecznej gęstości grafu (Erdos-Renyi model). Poza losowym decydowaniem o wygenerowaniu krawędzi dodatkowo sztucznie tworzę dwa mosty poprzez wygenerowanie 3 oddzielnych grafów (dzielę na 3 podaną liczbę wierzchołków) i połączenie ich krawędzią – oczywiście krawędzi mostowych może być znacznie więcej, zależy to przede wszystkim od gęstości grafu. Do mierzenia czasów poszczególnych operacji jak i działania całych algorytmów wykorzystałem funkcję System.nanoTime().

Wyniki przebrowadzonych badań:

(Pełne wyniki znajdują się w dwóch plikach excela w folderze z projektem.)

Operacje na grafach i ich wpływ na wydajność:

Liczba wierzchołków i gęstość w tym przypadku ma pomniejsze znaczenie dlatego pominę rozpatrywanie tego zagadnienia dla różnych wartości ponieważ za każdym razem widać tą samą zależność. W przypadku operacji wykonywanych na reprezentacji listowej grafu oczywiście największy czas wykonywania ma sprawdzanie spójności gdyż jest to najbradziej złożona operacja m.in wywoływany jest wtedy wielokrotnie algorytm dfs więc nie ma tutaj żadnego zaskoczenia. Operacja sprawdzania trwała około 12000 nanosekund, wartość ta potrafiła drastycznie wzrosnąć przy znalezieniu mostu jednak było to spowodowane tylko i wyłącznie wypisaniem informacji na ten temat, usunięcie tej funkcji obniżało czas do poziomu pozostałych wywołań. Ciekawszą rzeczą jest stosunkowo długi czas usuwania krawędzi. Było to około 4000 nanosekund i zazwyczaj czas ten był ok. 3 krotnie większy niż czas dodawania krawędzi. Spowodowane jest to faktem iż aby usunąć daną krawędź musimy wyszukać ją w naszej strukturze co zabiera stosunkowo dużo czasu. Dodanie krawędzi trwa krótko ponieważ wskazujemy konkretne miejsce do jakiego ma zostać dodana. W tym przypadku trwało to około 1400 nanosekund.

W przypadku reprezentacji macierzowej analogicznie sprawdzanie spójności trwało najdłużej z dokładnie tych samych względów (około 14000 nanosekund). Jednak usuwanie oraz dodawanie krawędzi ze względu na prostotę tego działania było zdecydowanie krótsze niż w przypadku reprezentacji listowej i trwało zazwyczaj dokładnie tyle samo dla obu tych operacji (około 400 nanosekund). Spowodowane jest to tym iż obie te funkcje wyglądają analogicznie i ograniczają się do zmiany dwóch wartości w tablicy dwuwymiarowej na true/false.

Wpływ gęstości na czas działania algorytmów:

Na poniższych wykresach widzimy, że zdecydowanie naiwne wyszukiwanie mostów trwa najdłużej. Im większa liczba wierzchołków tym ta różnica staje się bardziej widoczna. Algorytm bazujący na krawędziach drzewowych spisuje się znacznie lepiej (z wyjątkiem pierwszego przypadku gdzie najwyraźniej przez niską ilość wierzchołków musiała zostać wykonana podbna liczba operacji co dla algorytmu nawinego) jednak w przypadku większej ilości wierzchołków różnica na korzyść ulepszonego algorytmu jest zdecydowana. Za każdym razem nakrócej wykonywał się algorytm Tarjana i w przypadku tego algorytmu czas prawie zawsze był stały niezależnie od gęstości. W przypadku algorytmu naiwnego wzrost czasu wykonywania dla większej gęstości znacznie wzrastał co wynika oczywiście ze wzrostu ilości operacji do wykonania. W przypadku ulepszonego algorytmu naiwnego również można było zaobserować wahania jednak były one raczej niewielkie dla większej ilości wierzchołków i w tym przypadku gęstość wpływała na czas w małym stopniu.