Programowanie Współbieżne algorytm: b-skojarzenia b-adoratorów
Autor: Aleksandra Grzyb, indeks: 385552

1.Implementacja

współczynnik:UPDATE, MULTIPLY

Wczytywanie z pliku

(przenumerowanie)
W celu ułatwienia dalszych obliczeń wierzchołki podane w pliku tekstowym są przenumerowane na liczby z zakresu [0, ilość wierzchołków -1]. W wektorze staryNumer, przechowuję wczytywane indeksy, potrzebne do porównywania krawędzi grafu.

Przygotowania do wyszukiwania adoratorów (wykonywane jednokrotnie)

(sortowanie Sąsiadów, inicjalizowanie)

Po wczytywaniu następuje częściowe sortowanie sąsiedztwa, uwzględniając podany w zadaniu porządek na krawędziach grafu:

(1) W(v,w) :<: W(v,x) <=> (W(v,w) < W(v,x)) || (W(v,w)==W(v,x) && w < x)

patrząc oczywiście na starą numerację.
Sortujemy MULTIPLY*max z możliwych bvalue dla danego wierzchołka. Używamy dostępnej w algorithm partial_sort. Zapamiętujemy ilość posortowanych sąsiadów.

Przygotowania do wyszukiwania adoratora (wykonywane przed każdym szukaniem adoratorów)

(tworzenie atrap adoratorów, tworzenie kolejki z wartościami bvalue)

Adoratorzy są umieszczani w kontenerze/ multiset, dzięki napisaniu własnej klasy komparator, adoratorzy będą uporządkowani według (1) reguły. W celu ułatwienia pisania kodu umieszczam atrapy, czyli pary (0, 0) w ilości zwracanej przez bvalue dla danego wierzchołka. Robimy to współbieżnie, każdy wątek dostaje kolejkę, w której są aktualnie nie przetworzone wierzchołki, zabiera wierzchołek uprzednio ją blokując i tworzy dla niego atrapy adoratorów.

Szukanie adoratorów (wykonywane limitb + 1 razy)

(przetwarzanie wierzchołków, szukanie adoratora,sortowanie, najmniejszy adorator, wynik, aktualizowanie odrzuconych adoratorów)

Wierzchołki są przetwarzane współbieżnie. Każdy wątek odpytuje kolejkę wierzchołków, czy są jakieś do obliczenia. Następnie jeśli jest, szuka dla niego adoratorów (blokuje kolejkę i usuwa z niej ostatni), wpp. kończy działanie.
Szukanie adoratorów odbywa się tak, jak w opisanym algorytmie. Dla każdego wierzchołka trzymamy najmniejszego (według porządku(1)) adoratora w tablicy typu atomic. Dzięki temu nie musimy niczego blokować, aby dowiedzieć się, czy znaleźliśmy potencjalnie lepszego adoratora. Sąsiadów przechodzimy iteratorem, który jest zapamiętywany w globalnym

wektorze dla każdego wierzchołka. W przypadku odrzucenia danego sąsiada, zwiększamy iterator, zmniejszamy liczbę posortowanych sąsiadów oraz sprawdzamy, czy nie należy sortować.
Kiedy znajdziemy kandydata na adoratora, blokujemy sąsiedni wierzchołek. Upewniamy się, że kandydat jest lepszy. Jeśli tak, umieszczamy odrzuconego adoratora w unordered_map, usuwamy go z listy adoratorów. Wstawiamy nowego adoratora. Aktualizujemy najmniejszych adoratorów. Co jest warte uwagi, najpierw aktualizujemy numer, potem wagę.
Suma wag(2 * wynik) jest typu atomic<unsigned> za każdym znalezionym/usuniętym adoratorem zmieniamy ją o wagę danej krawędzi.
Kiedy ilość odrzuconych wierzchołków wynosi UPDATE są one dodawane do ogólnej mapy odrzuconych, po uprzednim jej zablokaniu.
Po zakończeniu pracy wątków, tworzymy nową kolejkę z mapy odrzuconych.
Powtarzamy powyższe operacje do momentu, kiedy mapa odrzuconych adoratorów będzie pusta.

Przywracanie informacji (wykonywane limitb razy)
(przywracanie sąsiadów, usuwanie adoratorów)
Sąsiadów przywracamy współbieżnie ustawiając ich iterator na początek. I-ty wątek przetwarza i + liczba wątków * k wierzchołki.
Usuwanie adoratorów wykonuje również współbieżnie, jednak dzielimy pracę przez kolejkę i odpytywanie jej o nieprzetworzone wierzchołki (niektóre krawędzie mogły mieć wielu adoratorów).

Blokowanie
Do blokowania używam struktury opartej na atomic.

2.Speed up

Ilość wątków	1	2	3	4	5	6	7	8
czas	29.7978	20.4423	15.5542	12.4749	11.9792	10.4477	8.97703	8.00229
speed up		1.45765	1.91573	2.38862	2.48746	2.85209	3.31933	3.72365

3.Optymalizacje

Kontenery
Wybieram najszybsze: unordered_map zamiast map.

W większości vector zamiast przykładowo list. Są to najszybsze kontenery, o ile używa się push_back(), back() i reserve(). Staram się wywoływać wspomniany reserve dla zmiennych, aby nie wywoływać zbędnego kopiowania.

Typ danych

Używam samych zmiennych typu unsigned, co nie powoduje zbędnego rzutowania.

Częściowe sortowanie

Nie porządkuję wszystkich sąsiadów. Jednak wspomniany współczynnik MULTIPLY w implementacji trudno jest wybrać w oparciu o testy. W testach z materiałów każdy wierzchołek ma średnio mniej niż 20 sąsiadów, co sądzę, że utrudnia wybór. Czasy dla (skitter.txt)

MULTIPLY

brak 34.2667 s
5 29.0768 s
9 29.4195 s
12 29.596 s

Intuicyjnie (inspirując się zamieszczoną w materiałach pracą) wybieram 9.

Wielokrotne aktualizowanie odrzuconych wierzchołków

Nie aktualizuję odrzuconych sąsiadów na koniec szukania adoratorów. Robię to, kiedy liczba odrzuconych wynosi wspomniany UPDATE. Ustawiłam go ze względu na czas wykonania w skitter.txt (plik z linku podanego w materiałach)

UPDATE

brak 45.2786 s
8 38.68 s
64 37.2788 s
128 36.9863 s
256 36.9425 s

Na podstawie pomiarów wybrałam 128.

Spin-lock

Tworzę własne zatrzaski, używając atomic. Są szybsze od mutex.

Przechowywanie najmniejszego adoratora

Zmieniając listę adoratorów, nie można z tej listy odczytywać. Zapisanie najmniejszych do atomic (tworzę dwie tablice, jedną na numery wierzchołków, drugą na wagi) umożliwia mi jednoczesny odczyt i zapis.

Uwagi:

Zdaję sobie sprawę, że wybór współczynników oparty na jednym teście jest mało miarodajny. Wierzę jednak autorom pracy “EFFICIENT APPROXIMATION ALGORITHMS FOR WEIGHTED B-MATCHING“, którzy mają podobne do moich.