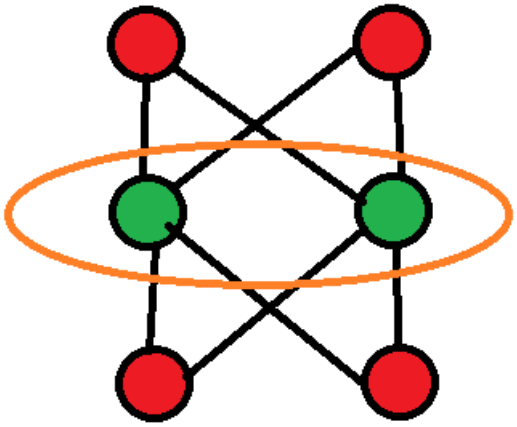
The Max Cut problem 😋

Mamy następujące zagadnienie. Dany jest graf nieskierowany. Chcemy podzielić go na dwie części w taki sposób by zmaksymalizować liczbę krawędzi, które łączą wierzchołki należące do różnych części. Chodzi po prostu o to, by część wierzchołków grafu przydzielić do podzbioru A, resztę do podzbioru B i maksymalizować liczbę krawędzi których jeden koniec leży w A a drugi koniec w B.



Okazuje się, że w ogólności rozwiązanie tego zagadnienia jest bardzo trudne. Jest to tzw. problem NP trudny, zatem (najpewniej) nie istnieje rozwiązanie tego problemu, które działałoby w czasie wielomianowym (i to z dowolnie dużym wykładnikiem).

Dla ścisłości: w pewnych szczególnych przypadkach (np. w przypadku grafów dwójdzielnych można znaleźć wydajną metodę poszukiwania ścisłego rozwiązania, my jednak skupiamy się na przypadkach ogólnych).

W przypadku wielu problemów typu NP bardzo skutecznym podejściem jest stosowanie różnego rodzaju algorytmów zrandomizowanych. Niestety nie ma pewności, że takie algorytmy dadzą nam najlepsze rozwiązanie. Jednak często są to proste do napisania i szybkie algorytmy. Można je zatem uruchamiać wielokrotnie i jako rozwiązanie brać najlepszy wynik. To zwiększa szansę sukcesu. Często też wcale nie szukamy najlepszego rozwiązania tylko rozwiązania spełniającego jakieś kryterium (np. potrzebujemy podziału grafu, w którym krawędzi przechodzących między odrębnymi zbiorami będzie więcej niż X). Wtedy od razu można sprawdzić, czy rozwiązanie oferowane przez algorytm randomizowany jest dla nas wystarczające.

Algorytmy local – search

Algorytmy local-search bazując na pewnym zadanym rozwiązaniu problemu (może być ono bardzo niedokładne) dążą do jego ulepszenia. Aby wyjaśnić działanie algorytmu local search dla problemu Max-Cut trzeba wprowadzić dwa pojęcia pomocnicze. Załóżmy, że wierzchołki grafu są podzielone na podzbiory A oraz B (natura tego podziału nie jest w tej chwili istotna, może to być nawet podział losowy). Każda z krawędzi grafu może zatem należeć do dwóch kategorii:

a) Krawędź „wewnętrzna" oba jej wierzchołki leżą w zbiorze A lub oba leżą w zbiorze B.
b) Krawędź „przecinająca" ma jeden wierzchołek w zbiorze A a drugi w B.

Algorytm działa w następujący sposób.

1) Mamy graf G o V wierzchołkach.
2) Losowo przydzielamy wierzchołki grafu G do zbiorów A albo B.
3) Algorytm Local search optymalizuje ten losowy podział w następujący sposób: jeśli z jakiegokolwiek wierzchołka grafu (dowolnego tzn. leżącego w A lub w B) liczba wychodzących z niego krawędzi „wewnętrznych" jest większa niż liczba wychodzących z niego krawędzi „przecinających" to jest on przenoszony do drugiego z podzbiorów (na skutek tego przeniesienia krawędzie „wewnętrzne" staną się „przecinającymi" i vice versa zatem zwiększymy liczbę krawędzi „przecinających"). Punkt trzeci kontynuujemy aż nie będzie już więcej wierzchołków do przeniesienia.

Implementacja punktu 3 jest następująca

```
warunek ==true

while(warunek):

    warunek ==false

    dla każdego wierzchołka X w Grafie G (pętla for):

        jeśli liczba krawędzi wewnętrznych wychodzących z X jest większa niż przecinających to:
            warunek ==true

            przenieś X do drugiego z podzbiorów
```

Wskazówki:

1) Podział grafu na podzbiory chyba najprościej zaimplementować za pomocą macierzy booleanów (tyle elementów ile wierzchołków w grafie). Jeśli wierzchołek należy do A to dla tego elementu w macierzy jest false a jeśli należy do b to jest true. Dzięki

takiemu podejściu sprawdzenie do jakiego podzbioru należy dany wierzchołek jest błyskawiczne (czas rzędu O(1)).

2) Ja sobie zdefiniowałem osobną metodę, która dla danego wierzchołka (z) sprawdza ile jest krawędzi przecinających a ile nieprzecinających. Przy implementacji przyda się metoda zwracająca wierzchołki sąsiadujące z podanym wierzchołkiem (przy implementacji grafu implementowaliśmy metodę getNeighbouringIndices). Metoda ta może być zaimplementowana tak (partition to macierz booleanów trzymająca „podział" grafu)

```
std::vector<int> Graf::innerOuterEdges(int vertNR)
 {
     std::vector<int> neighbours= getNeighbourIndices(vertNR);
     int inner=0;
     int crossing=0;
     bool vertPartRes = partition[vertNR];
     for (int i = 0; i < neighbours.size(); i++)
     {
         if (partition[neighbours[i]] == vertPartRes)
         {
             inner++;
         }
         else
         {
             crossing++;
         }
      }
       std::vector<int> res = std::vector<int>();
     res.push_back(inner);
     res.push_back(crossing);
     return res;
 }
```

3) Aby zwiększyć prawdopodobieństwo uzyskania dobrego wyniku najlepiej uruchomić algorytm (oczywiście z innym losowym podziałem wierzchołków na zbiory A i B) wiele razy np. z 500 lub więcej.

4) Trzeba uważać, aby generator liczb pseudolosowych generował różne podziały losowe. Jeśli używamy rand() to trzeba generator zainicjować przez srand(time[0]). Ale trzeba uważać, by nie powtarzać inicjalizacji srand(time[0]) wewnątrz pętli for (bo może się zdarzyć, że dla wielu sąsiednich iteracji generator dostanie to samo ziarno). Wtedy lepiej zainicjalizować generator raz, przed pętlą.

Testy.

Dla grafu z pliku testowego („GrafTest.txt") moja implementacja (500 losowań) dała najlepszy wynik 6439 (chodzi o maksymalną liczbę krawędzi przecinających).

Zadanie:

Proszę podać jaka jest maksymalna liczba krawędzi przecinających dla grafu z pliku z zadaniem („GrafZad.txt”. Proszę podać podział grafu odpowiadający tej liczbie (zawartość wektora booleanów jako 0 i 1).