Moja implementacja algorytmu b-adoratorów jest bardzo prosta. Wszystkim poza wczytywaniem danych zajmują się dwie klasy - Graph i Node reprezentujące graf i poszczególne wierzchołki tego grafu. Wszystkie informacje o tym grafie takie jak zbiór wierzchołków, listy sąsiedztwa, inne zmienne pomocnicze są trzymane w tych dwóch klasach. To one zajmują się wykonaniem algorytmu, który moim zdaniem jest trochę prostszą wersją algorytmu z treści zadania i pracy *KHAN, Arif, et al. Efficient approximation algorithms for weighted b-matching.*

Z faktu, że rozpatrujemy grafy proste wynika, że żadna para wierzchołków nie będzie połączona więcej niż jedna krawędzią, zatem żaden wierzchołek nie może wystąpić więcej niż raz na pewnej liście sąsiedztwa(*). Sortując tą listę (:<:) otrzymamy listę malejących pod względem atrakcyjności kandydatów dla danego wierzchołka (v). Zauważmy, że nigdy nie chcemy się po tej liście cofać. Jeżeli jesteśmy na i-tej pozycji tej listy, to dla wierzchołków (x) z tej listy na pozycjach [0, i) zachodzi jedna z trzech możliwości:

i) v jest połączony z x
ii) v był połączony z x, ale pojawił się lepszy kandydat u który zastąpił v
iii) v nie połączył się z x, ponieważ istniał już wtedy lepszy kandydat u

Dla pierwszego przypadku jesteśmy już połączeni z x, więc oczywiście nie chcemy łączyć się z nim jeszcze raz (prowadziłoby to do błędów), a dla drugiego i trzeciego zostalibyśmy odrzuceni (:c), ponieważ znaleziono już b(x) lepszych kandydatów od nas(**). Połączenie tych dwóch obserwacji ((*) i (**)) pozwala nam na pozbycie się struktury T opisanej w treści zadania. Oszczędzam w ten sposób czas i pamięć potrzebną na utrzymywanie i aktualizacje tej struktury, więc chyba mogę uznać to za moją pierwszą optymalizację. Reszta algorytmu działa tak samo jak algorytm opisany w treści zadania i pracy KHAN, Arif, et al.

Kolejną zastosowaną przeze mnie optymalizacją jest częściowe sortowanie list sąsiedztwa opisane w pracy naukowej KHAN, Arif, et al. Zdecydowałem się na p = 7, które jest po prostu średnią wartością najbardziej optymalnych p dla różnych grafów. Samo sortowanie wykonuję funkcją std::partial_sort z biblioteki <algorithm>.

Ostatnią optymalizacją jest wyposażenie każdego wątku we własną kolejkę wierzchołków, które trzeba będzie znowu rozpatrzeć. Ta zmiana również została opisana w danej pracy naukowej. Wątki zamiast dodawać każdy element na kolejkę osobno, najpierw budują własną, a po rozpatrzeniu wszystkich danych im wierzchołków przepisują ją do wspólnej.

Testowanie mojego rozwiązania na maszynie students z wykorzystaniem funkcji time dla dwóch identycznych wywołań potrafiło zwrócić czasy różniące się nawet o 30 sekund dla średniego czasu równego minutę. Dlatego zmuszony byłem obliczać czas działania funkcji na laptopie z jednym, dwurdzeniowym procesorem. Program wywołuję na przerobionym grafie road_PA pobranym ze strony SNAP, dla blimit = 20. Czas wczytania wejścia wynosi około 2,5 sekundy niezależnie od liczby wątków (został on odjęty).

| Number of threads | Execution time | Acceleration |
|---|---|---|
| **1** | 16.584s | 1.00 |
| **2** | 13.634s | 1.22 |
| **3** | 21.426s | 0.77 |
| **4** | 24.213s | 0.69 |
| **5** | 26.363s | 0.63 |
| **6** | 27.770s | 0.60 |
| **7** | 28.142s | 0.58 |
| **8** | 29.464s | 0.56 |

Widzimy wyraźne przyśpieszenie dla 2 wątków. Biorąc pod uwagę, że cały test trwa tylko 16 sekund dla jednego wątku, przyśpieszenie o 3 sekundy dla dwóch wątków uważam za dobry wynik. Niestety dla 3 i więcej wątków nie byłem w stanie zaobserwować przyśpieszenia, ponieważ sprzęt na którym obliczałem czas wykonywania programu posiada tylko dwa procesory. Z tego powodu wywołanie programu dla więcej niż dwóch wątków spowalnia program zamiast go przyśpieszać.