Optymalizacja 1 **Natalia Kwiecień**
Projekt 1 - **Pivot Rules** **Jakub Dudziński**

Wprowadzenie

Projekt ma na celu zaimplementowanie kilku przykładowych funkcji dokonujących wyboru zmiennych (wierzchołków) w algorytmie sympleks oraz zbadanie jak wybór danej funkcji wpływa na liczbę wykonywanych przez algorytm kroków. W tym celu przetestowaliśmy każdą z zaimplementowanych metod na problemach liniowych w formacie lp. W pierwszej części opiszemy każdą z funkcji. W drugiej problemy, na których testowaliśmy zadane metody.

Przegląd używanych funkcji

Porządek leksykograficzny, minimum

Funkcja ta (zaimplementowana w linijkach *#12 - #16* pod nazwą *lexicographical_min_entering(self)* oraz *lexicographical_min_leaving(self)*) wybiera minimalny indeks z możliwych do wejścia/wyjścia wierzchołków. Tu zbyt wiele tłumaczyć nie trzeba.

Porządek leksykograficzny, maksimum

Analogicznie do poprzedniej funkcji, metoda ta wybiera maksymalny indeks z możliwych do wejścia i wyjścia wierzchołków (linijki *#20 - #24*, *lexicographical_max_entering(self)* oraz *lexicographical_max_leaving(self)*).

Największy wzrost

Metoda zaimplementowana w linijkach *#28 - #53* pod nazwą *largest_increase(self)*. Zwraca ona parę wierzchołków (wierzchołek wejściowy i wyjściowy) w postaci listy, która prowadzi do największego wzrostu funkcji celu. Na początku przechowujemy wartość funkcji celu dla wierzchołka, z którego startujemy, w zmiennej *obj_now*. Przypisujemy zmiennej *obj_max* (zmienna przechowująca dotychczasową wartość maksymalną funkcji celu) wartość *obj_curr*, czyli wartość funkcji celu w wierzchołku, do którego teraz wchodzimy. Iterujemy poprzez wszystkie możliwe pary wejścia-wyjścia i porównujemy przyrosty funkcji dla poszczególnych par (linijka *#47*). Jeśli dany zestaw wierzchołków poprawia nam wzrost funkcji to przypisujemy zmiennej *obj_max* wartość *obj_curr*. Na sam koniec zwracana nam jest szukana para wierzchołków.

Najmniejszy wzrost

Metoda analogiczna do powyższej. Działanie kodu również analogiczne, więc nie będziemy powtarzać rozumowania. Funkcja zwraca nam listę dwóch wierzchołków, dla których wzrost funkcji celu jest najmniejszy (linijki *#57 - #83*, pod nazwą *smallest_increase(self)*).

Największy współczynnik

Metoda ta (linijki *#88 - #91*, *max_coefficient_entering(self)*)wybiera zmienną wejściową, przy której stoi największy współczynnik w funkcji celu. Zmienna *max_value* odpowiada za tenże współczynnik. Zmienna *max_index* mówi nam, dla jakiego indeksu na liście znajdziemy ten współczynnik. A cała funkcja zwraca nam zmienną występującą pod tym indeksem. Przy wyciąganiu wierzchołków korzystamy w naszym programie z metody *lexicographical_min_leaving(self)*.

Losowy wybór wierzchołka

Jest to metoda wybierająca losowy wierzchołek wejściowy/wyjściowy ze wszystkich możliwych wejść/wyjść (*possible_entering()/possible_leaving()*) z prawdopodobieństwem jednostajnym. Zaimplementowana w linijkach *#95 - #99* pod nazwą *uni_random_entering(self)* oraz *uni_random_leaving(self)*.

Średnia ważona

Metoda, która determinuje wybór zmiennych wejściowych i wyjściowych w oparciu o przypisane każdej z możliwości prawdopodobieństwo. Prawdopodobieństwa dla wszystkich wejść/wyjść zadane są w następujący sposób. Będziemy chcieli utworzyć listę indeksów B, w której niektóre indeksy będą pojawiały się częściej od innych (ten zabieg wyznaczy nam różne prawdopodobieństwa dla różnych wierzchołków). Niech A będzie listą wszystkich możliwych wejść/wyjść. Wtedy iterując od środka listy A w lewą stronę dodajemy czytane indeksy elementów z listy A na listę B i zwiększamy ilość dodawanych indeksów dwa razy przy każdym kroku iteracyjnym. To samo robi kolejna pętla czytająca indeksy elementów z A od $\lfloor \frac{len(A)}{2} + 1 \rfloor$ miejsca do ostatniego. Uruchamiamy funkcję *random.choice(B)*, znaną nam z poprzedniej metody, która wybierze element z B z prawdopodobieństwem jednostajnym. Ostatecznie niektóre z indeksów będą bardziej prawdopodobne do wylosowania, gdyż występują częściej. Funkcja zwraca nam wierzchołek o danym indeksie z listy A. Metoda zakodowana w linijkach *#103 - #133* pod nazwami *weighted_random_entering(self)* i *weighted_random_leaving(self)*.

Najbardziej stroma krawędź

Metoda wybierająca wierzchołek minimalizujący kąt między gradientem funkcji celu a krawędzią łączącą stary wierzchołek z nowym. W liniach kodu *#138 - #139* tworzymy bazę pod wektor funkcji celu oraz pod wierzchołek w którym obecnie jesteśmy (*v_old*). W linijkach *#146 - #153* uzupełniamy je o odpowiednie wartości. Dążymy do zmaksymalizowania takiego wyrażenia:

$$\frac{c^T(v_{new} - v_{old})}{||v_{new} - v_{old}||}$$

W pętli z linijki *#155* szukamy niezdegenerowanego wierzchołka, czyli takiego, dla którego powyższy wzór ma sens. Jeśli takiego nie znajdziemy, to wybieramy jakikolwiek. W pętli *#181* szukamy wierzchołka *v_new* maksymalizującego wyrażenie. Funkcja zwraca nam parę wierzchołków (wejścia i wyjścia).

Pierwszy od lewej

Metoda wybierająca ze wszystkich możliwych wejść/wyjść wierzchołek z początku listy (linijki *#204 - #208*, *farthest_left_entering(self)* oraz *farthest_left_leaving(self)*.

Pierwszy od prawej

Metoda wybierająca ze wszystkich możliwych wejść/wyjść wierzchołek z końca listy (linijki *#212 - #216*, *farthest_right_entering(self)* oraz *farthest_right_leaving(self)*.

Użytkowanie kodu

Poszczególne funkcje dla programu wybieramy w linijkach *#222 - #244*, poprzez odkomentowanie odpowiedniej metody wybierania wierzchołka wejścia/wyjścia wewnątrz funkcji *my_entering(self)* oraz *my_leaving(self)*.

Wyniki testów i ich analiza

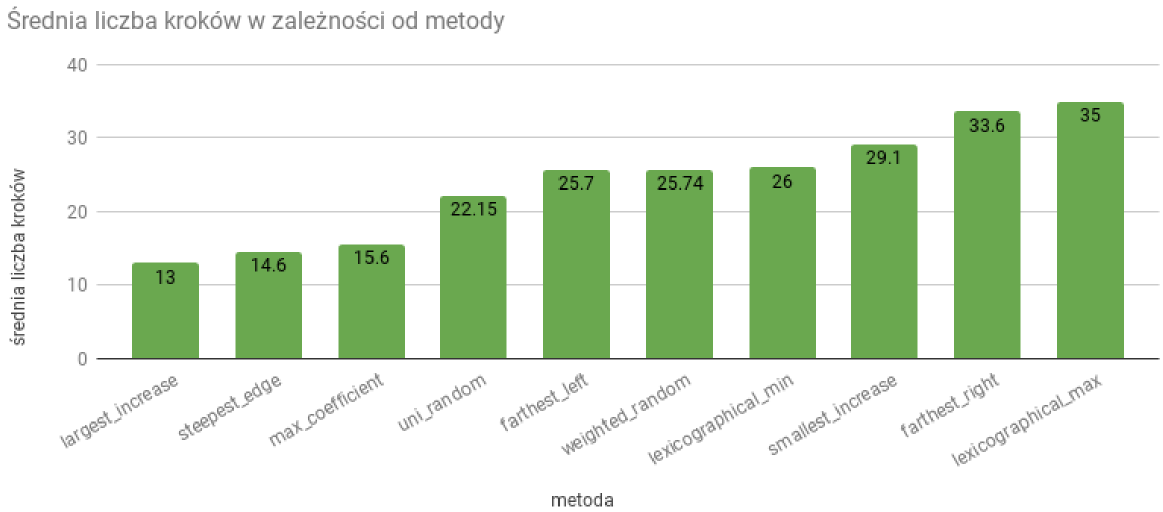
W tym podrozdziale przedstawimy wyniki (liczbę kroków) uzyskane przy użyciu metod opisanych w poprzednim podrozdziale. Wśród problemów testowych znajdują się proponowane problemy (https://github.com/henrykmichalewski/optymalizacja_2017/tree/master/projekt_1/testy), problemy z laboratorium oraz 3 problemy (Problem 8, 9, 10) wygenerowane za pomocą pliku *LosujProblem.py* znajdującego się w podkatalogu *Testy*. Wyniki testów prezentuje poniższa tabela. W przypadku problemów całkowitoliczbowych rozważyliśmy wariant zrelaksowany problemu. Dla metod *uni_random* i *weighted_random* liczba kroków w tabeli jest średnią z pięciu testów. Rozważyliśmy także problemy *Furniture* i *Whiskas Model*, ale w obu przypadkach wszystkie metody uzyskały taką samą liczbę kroków (równą 2), więc postanowiliśmy nie uwzględniać tych wyników w tabelii. Przeprowadziliśmy także testy na dwóch, dużych problemach generowanych losowo (liczba kroków oscylowała w okolicach 150 kroków), jednak metoda *smallest_increase* ich nie ukończyła, dlatego również zostały pominięte.

Tablica 1: Liczba kroków dla poszczególnych metod i problemów.

Problem / Metoda	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	avg
lexicographical_min	2	29	37	8	7	11	5	53	47	61	26
lexicographical_max	3	30	71	6	8	2	3	71	71	85	35
largest_increase	2	15	28	8	7	3	3	19	24	21	13
smallest_increase	3	36	50	8	7	6	3	61	48	69	29.1
max_coefficient	2	20	24	6	5	9	3	26	21	40	15.6
uni_random	2,8	22.5	27.2	7.8	7.2	4.4	3.8	37.6	56.2	52	22.2
weighted_random	2.2	20.4	38.8	5.6	7.4	5	4.2	37.8	61	75	25.7
steepest_edge	2	15	19	4	7	9	3	26	19	42	14.6
farthest_left	3	23	56	4	9	6	5	61	21	69	25.7
farthest_right	2	45	68	8	8	7	3	81	19	95	33.6

(1) Beer Distribution Problem, (2) 5 Queens, (3) 6 Queens, (4) Maximum Vertex Cover, (5) Computer Plant Problem (6) Whiskas Model 2, (7) American Steel Problem, (8) Problem 8, (9) Problem 9 , (10) Problem 10

Jak widać z tabeli powyżej wyróżniają się (pozytywnie) trzy metody: *largest_increase*, *steepest_edge* i *max_coefficient*. W prawie wszystkich testach które wykonaliśmy *largest_increase* zanotowało najlepszy lub drugi najlepszy wynik. Uzyskało także średnio najniższą liczbę kroków we wszystkich testach. Szczególnie dobrze wypadło w teście (10) i (8). Metoda *steepest_edge* wypadła niewiele gorzej. Wykonała średnio 1,8 kroków więcej. Uzyskała bardzo dobre wyniki w testach (3) i (9). W teście (10) uzyskała niezadowalający wynik. Metoda *max_coefficient* uzyskała niezłe wyniki we wszystkich testach. Trzy pierwsze metody uzyskały dobre (średnio mniej niż 16 kroków) wyniki. (Por. wykres poniżej).



Ich zaletą jest na pewno szybkie rozwiązywanie standardowych problemów liniowych. Mogłyby mieć one jednak problem z przypadkami zdegenerowanymi - brak wzrostu funkcji celu (*largest_increase*), wiele wierzchołków zdegenerowanych (*steepest_edge*) czy pętle. Każda z tych metod wymagała zaimplementowania dodatkowej metody wyboru wierzchołka w przypadku zdegenerowanym. Te trzy metody są także najmniej wydajne pod względem czasu obliczeń. Każda z tych metod wymaga zbadania wszystkich możliwych par wierzchołków wejścia-wyjścia, co jest bardzo czasochłonne.

Nad wyraz dobrze wypadły metody losowe, sa one jednak niedetermistyczne i ciężko wyobrazić sobie ich użycie w praktyce, do rozwiązywania dużych problemów liniowych.

Ciekawie zaprezentowały się metody *farthest_left* i *farthest_right*. Pierwsza z nich uzyskała znaczącą przewagę nad drugą. Prawdopodobnie kolejność występowania zmiennych na liście *possible_entering* i *possible_leaving* nie jest przypadkowa i te metody (mogą!) odpowiadać odpowiednio regule wkładania i wyciągania z bazy wierzchołków, które wkładano/wyciągano do bazy najrzadziej i najczęściej. Oczywistą zaletą tych metod jest praktycznie zerowa złożoność obliczeniowa.

Reguła *smallest_increase* jest jedyną zaimplementowaną przez nas 'złośiwą' regułą (tzn. rozumiemy przez to taką regułę, która z założenia nie dąży do jak najlepszego zwiększenia funkcji celu). Jak widać na histogramie wypadła ona niekorzystnie w porównaniu z regułą *largest_increase*. Wykonała ona (średnio) dwukrotnie więcej kroków od swojej rywalki. Taka reguła jest zarówno bardzo nie wydajna pod względem złożoności obliczeniowej, jak i wysoce niewydajna pod względem liczby kroków (dla większych testów ta reguła zanotowała trzy *DNF*).

W sposób podobny do powyższego możnaby tworzyć 'złośliwe' reguły analogiczne do *stepest_edge* i *max_coefficient*, jednak nie zostały one zaimplementowane (wynik byłby chyba zbyt łatwy do przewidzenia, jak widać na przykładzie *smallest_increase*).

Wśród reguł leksykograficznych możemy zauważyć znaczącą różnicę. *Lexicographical_min* wydaje się być znacznie lepsza od swojej rywalki *lexicographical_max*. Obie cechuje praktycznie zerowa złożoność obliczeniowa i dość duża liczba kroków potrzebna do zakończenia działania metody sympleks. Oczywistą zaletą metody *lexicographical_min*, zwaną także *Regułą Blanda* jest to, że nie pozwala na wpadnięcie w cykl.

Wnioski

Jak widać z powyższej analizy reguły można podzielić na kilka, zasadniczych grup:

- **Reguły o niskiej liczbie kroków**. Są one jednak często niewydajne obliczeniowo. Do tej grupy należą: *largest_increase*, *steepest_edge* i *max_coefficient*.
- **Reguły wydajne obliczeniowo**. Robią one jednak (średnio) więcej kroków od reguł z pierwszej grupy. Do tej grupy należą: *farthest_left*, *farthest_right*, *lexicographical_min*, *lexicographical_max*.
- **Reguły zapobiegające pętlom**. Do tej grupy należy: *lexicographical_min*
- **Reguły niedeterministyczne**.Metody niepraktyczne. Do tej grupy należą: *uni_random*, *weighted_random*.
- **Reguły 'złośliwe'**. Bez praktycznego zastosowania. Do tej grupy należy: *smallest_increase*.

Do każdego problemu możemy dobrać pewną, odpowiednią regułę w zależności od jego specyfiki (np. pętle) i tego czy zależy nam na wydajności obliczeniowej czy też zminimalizowaniu liczby kroków algorytmu sympleks.