Wrocław, 29 marca 2021

Zaawansowane **T**echniki **O**ptymalizacji

Laboratorium 3

Rozwiązywanie wybranych zadań optymalizacji w środowisku CPLEX `python`

prowadzący: *dr inż. Jarosław Rudy*

1 Cel laboratorium

Celem laboratorium jest zapoznanie się z metodami modelowania i rozwiązywania problemów z wykorzystaniem biblioteki CPLEX z poziomu wybranego języku programowania. Temat obejmuje przygotowanie środowiska pracy, włącznie z pobraniem i ustawieniem biblioteki CPLEX, modelowanie problemu wraz z jego ograniczeniami, zapis instancji testowej oraz wybór metody rozwiązania, uruchomienie optymalizatora oraz analizę otrzymanych wyników.

2 Przebieg zajęć

Laboratorium obejmuje zajęcia nr 5 i 6 (4 godziny zajęć). Praca odbywa się w ramach grup dwuosobowych. Każda grupa otrzymuje do zrealizowania trzy problemy optymalizacji dyskretnej lub ciągłej z poniższych list.

Zagadnienia dyskretne:

1. Problem sumy podzbioru (problem 2.1).
2. Zagadnienie transportowe (problem 2.2).
3. Zagadnienie przydziału (problem 2.3).
4. Kwadratowe zagadnienie przydziału (problem 2.4).
5. Dyskretny problem plecakowy (problem 2.5).

Zagadnienia ciągłe (jako modyfikacje problemów dyskretnych):

1. Problem inspekcji (problem 5.1).
2. Kwadratowe zagadnienie przydziału (problem 5.2).
3. Problem dwuplecakowy (problem 5.3).
4. Szeregowanie zadań na równoległych maszynach (problem 5.4).

Modele zadanych problemów należy zapisać używając wybranego języka programowania oraz biblioteki CPLEX. Następnie należy uruchomić optymalizator, dokonać analizy wyniku (w tym logów) pracy optymalizatora. Należy też oszacować zakres stosowalności i/lub złożoność optymalizatora dla danego problemu. Uwaga: dane wejściowe należy wygenerować korzystając z dostępnych generatorów liczb pseudolosowych (na stronie kursu).

Naliczanie oceny zaczynamy od 2. Za poprawne przedstawienie (model, generacja danych wejściowych, analiza wyników) każdego z trzech problemów grupa otrzymuje +1 do oceny. Na ocenę wpływa również czas oddania zadania oraz wiedza studenta.

3 Praca z pakietem optymalizacyjnym CPLEX w wybranym jęzkyu programowania

3.1 Dostęp i instalacja

`CPLEX Optimizer` jest matematycznym narzędziem do rozwiązywania problemów optymalizacyjnych przy użyciu technik programowania liniowego, mieszanego programowania całkowitoliczbowego lub programowania kwadratowego. Dostęp do pakietu w postaci biblioteki możliwy jest z poziomu m.in. następujących języków programowania:

- `python`,
- `C/C++`,
- `C#`,
- `Java`.

Szczegółowy sposób przygotowania środowiska pracy dostępny jest na stronie IBM:

- do pracy z biblioteką CPLEX konieczne jest najpierw zainstalowanie aplikacji ILOG,
- ustawienie CPLEX dla języka `python` oraz tutorial dla języka `python` oraz dokumentacja klasy Model,
- ustawienie CPLEX w języku Java z użyciem Eclipse'a oraz tutorial dla języka `Java`,
- ustawienie CPLEX dla języka `C/C++` oraz tutorial dla języka `C` oraz tutorial dla języka `C++`,
- ustawienie CPLEX w systemie Windows,
- tutorial dla języka `C# .NET`,

Poniżej przedstawiono sposób instalacji niezbędnych narzędzi do przeprowadzenia laboratorium z poziomu języka `python`.

Na początku należy pobrać instalator `python` wyłącznie w wersji 3.7 lub 3.8) z oficjalnej strony python.org. W przypadku systemu MS Windows, należy pamiętać o zaznaczeniu opcji `Add Python 3.8 to PATH`. W przypadku systemu wielu dystrybucji GNU/Linux do instalacji środowiska `python` można użyć polecenia `sudo apt install python3` w wierszu poleceń.

Ze względu na mały rozmiar rozważanych na laboratorium projektów, do pisania i uruchamiania biblioteki CPLEX w języku `python` można z powodzeniem wykorzystać zwykłą konsolę. Możliwe jest też skorzystanie dedykowane środowisko programistyczne (IDE). Dla `pythona` możliwymi IDE są m.in. `JetBrains PyCharm` (dostępne za darmo dla MS Windows, Linux i MacOS do pobrania ze pobrać z oficjalnej strony producenta.) lub `Visual Studio` (dla MS Windows).

Ostatnim krokiem jest konfiguracja projektu. Najpierw należy odnaleźć katalog w którym zainstalowano IBM ILOG CPLEX Optimization Studio. Dla systemu MS Windows domyślna ścieżka ma postać:

```
C:\Program Files\IBM\ILOG\CPLEX_Studio_Community201
```

Wewnątrz katalogu ILOG-a należy przejść do katalogu `python`, a następnie `docplex`. W katalogu tym jest skrypt `setup.py`, który należy uruchomić z poziomu konsoli. Alternatywnie można ręcznie ustawić odpowiednią zmienną środowiskową. W środowisku IDE `PyCharm` należy utworzyć nowy projekt i przejść do jego ustawień (menu `File`). W opcjach (`Settings`) należy przejść do ustawień interpretera i poprzez symbol plusa (+) dodać pakiet `docplex`.

3.2 Modelowanie problemu

Poniżej zostanie przedstawiony sposób modelowania problemu optymalizacji na przykładzie języka `Python` pod systemem Linux. Więcej informacji można. Rozpatrzmy optymalizację dyskretną na przykładzie omawianego wcześniej problemu maksymalizacji wyników testu jednokrotnego wyboru. Dla przypomnienia, dany jest test n pytań, gdzie dla każdego pytania podane jest k wariantów odpowiedzi, wraz z wartością punktową. Zadanie polega na wyborze dla każdego pytania wariantu tak, aby zmaksymalizować sumę punktów uzyskanych za testu. Zauważmy, że w każdym pytaniu można zaznaczyć co najwyżej jeden wariant, co oznacza że można nie zaznaczać żadnego wariantu.

Zaczniemy od utworzenia pustego pliku o rozszerzeniu `.py` np. `test.py`. W celu umożliwiania na bieżąco testowania tworzonego kodu krótko omówimy sposób uruchamiania skryptu z poziomu konsoli. Generalnie dostępne są dwa sposoby (oba zakładają że znajduje się w katalogu z plikiem `test.py`:

1. `python3 test.py` – ta wersja wywołuje jawnie interpreter i nie wymaga dodatkowych zmian w pliku,
2. `./test.py` – ta wersja wymaga umieszczenia w *pierwszej* linii pliku `test.py` odpowiedniej informacji o interpreterze dla pliku (jest to tzw. shebang). Zakładając że ścieżka do pliku interpretera ma postać `/usr/bin/python3`, to linia ta powinna wyglądać jak poniżej:

   ```
   #!/usr/bin/python3
   ```

   Ponadto, w tym przypadku przed pierwszym uruchomieniem należy dodać uprawnienia wykonywania dla pliku poprzez wywołanie komendy `chmod u+x test.py`.

Przejdźmy teraz do opisu modelu. Pierwszym krokiem jest dołączanie odpowiedniego modułu. Do rozwiązania naszego problemu będziemy używać klasy `Model` z modułu `docplex.mp.model`, które dołączamy poprzez

```
from docplex.mp.model import Model
```

Zacznijmy od utworzenia zmiennych n i k, na razie niech mają stałe wartości:

```
n = 6
k = 4
```

Teraz możemy utworzyć nasz model `m`, aby to zrobić należy utworzyć obiekt wspomnianej klasy: `m = Model(name='test')`. Warto dodać też nazwę dla naszego modelu.

Teraz przejdziemy do przedstawienia zmiennych decyzyjnych. Do ich utworzenia należy używać odpowiednich metod modelu (`binary_var()`, `integer_var()` lub `continuous_var()`). Metody można wywołać z trzema różnymi parametrami opcjonalnymi: (1) `lb` – dolne ograniczenie zmiennej, domyślnie 0, (2) `ub` – górne ograniczenie zmiennej, domyślne nieskończoność oraz (3) `name` – nazwa zmiennej, pomocna podczas wyświetlania rozwiązania przy większej licznie zmiennych decyzyjnych. Przykładowo pojedynczą zmienną decyzyjną możemy utworzyć następująco:

```
x = m.continuous_var(lb=-99, ub=199, name='x')
```

będzie to zmienna ciągła (rzeczywista `float`), która może przyjąć wartości z przedziału `[-99,199]`.

W przypadku większej liczby zmiennych decyzyjnych warto posłużyć się listami lub słownikami. Dla dwóch zmiennych całkowitych można utworzyć dwuelementową listę:

```
x = [ m.integer_var(name='x1'),  m.integer_var(name='x2')]
```

lub w wersji z pętlą:

```
x = [ m.integer_var(name='x{0}.format(i)') for i in range(0,2) ]
```

W przypadku naszego testu będziemy potrzebować listy zawierającej listy zmiennych decyzyjnych:

```
x =[]
for i in range(0,n):
    x.append([m.binary_var(name='x{0}{1}'.format(i,j)) for j in range(0,k)])
```

lub jednowymiarowego słownika:

```
x = {(i,j): m.binary_var(name='x{0}{1}'.format(i,j))
    for i in range(1, n + 1 )
        for j in range(1, k+1)}
```

W tym przypadku kluczem (w definicji przed dwukropkiem) słownika jest krotka `(i,j)`, zaś wartością (po dwukropku) jest zmienna decyzyjna. Dla uzyskania wszystkich potrzebnych par klucz-wartość, całość jest „powtarzana" w obu wymiarach dzięki podanym konstrukcjom `for` z zakresem.

W ramach przetestowania modelu możemy utworzyć macierz `values` zawierającą wartości punktowe poszczególnych wariantów pytań. W języku `python` macierz będzie w postaci listy list:

```
values = [
    [7,2,9,3],
    [0,6,7,1],
    [4,1,6,3],
    [1,4,8,5],
    [3,3,9,8],
    [3,2,2,2]
    ];
```

Mając zdefiniowane dane wejściowe i zmienne decyzyjne, należy określić żądany sposób optymalizacji poprzez wywołanie metody `maximize` lub `minimize` zmiennej modelu `m`, jako argument podając funkcję celu. W przypadku naszego testu funkcja celu jest sumą, wobec czego konieczne jest wywołanie metody `sum` obiektu `m` oraz podanie zakresów sumowania. Ostatecznie funkcja celu (dla zmiennych decyzyjnych w postaci listy) przyjmie postać:

```
m.maximize(m.sum(x[i][j]*values[i][j] for i in range(0,n) for j in range(0,k)))
```

lub (dla zmiennych decyzyjnych w postaci słownika):

```
m.maximize(m.sum(x[i,j]*values[i-1][j-1]
    for i in range(1, n + 1 ) for j in range(1, k+1)))
```

Kolejnym krokiem jest sformułowanie ograniczeń problemu. Ograniczenia wprowadzamy przy pomocy metody `{add_constraint()` obiekt `m`. Podobnie jak w przypadku aplikacji języka OPL aplikacji ILOG, dostępne są trzy typy ograniczeń: (1) `==`, (2) `<=` lub (3) `>=`. Dla problemu testu jednokrotnego wyboru ograniczenia mogą przyjąć formę (zmienne decyzyjne jako lista)

```
m.add_constraint(x[0][0] + x[0][1] + x[0][2] + x[0][3] <= 1)
m.add_constraint(x[1][0] + x[1][1] + x[1][2] + x[1][3] <= 1)
m.add_constraint(x[2][0] + x[2][1] + x[2][2] + x[2][3] <= 1)
m.add_constraint(x[3][0] + x[3][1] + x[3][2] + x[3][3] <= 1)
m.add_constraint(x[4][0] + x[4][1] + x[4][2] + x[4][3] <= 1)
m.add_constraint(x[5][0] + x[5][1] + x[5][2] + x[5][3] <= 1)
```

lub (zmienne decyzyjne jako słownik)

```
m.add_constraint(x[1,1] + x[1,2] + x[1,3] + x[1,4] <= 1)
m.add_constraint(x[2,1] + x[2,2] + x[2,3] + x[2,4] <= 1)
m.add_constraint(x[3,1] + x[3,2] + x[3,3] + x[3,4] <= 1)
m.add_constraint(x[4,1] + x[4,2] + x[4,3] + x[4,4] <= 1)
m.add_constraint(x[5,1] + x[5,2] + x[5,3] + x[5,4] <= 1)
m.add_constraint(x[6,1] + x[6,2] + x[6,3] + x[6,4] <= 1)
```

Oczywiście powyższy zapis zajmuje dużo miejsca i jest niewygodny. Dlatego należy wykorzystywać zakresy i pętle dla skrócenia i uogólnienia zapisu (wersja listowa)

```
for i in range(0, n):
    m.add_constraint(m.sum(x[i][j] for j in range(0, k)) <= 1)
```

lub (wersja słownikowa):

```
for i in range(1, n + 1):
    m.add_constraint(m.sum(x[i,j] for j in range(1, k+1)) <= 1)
```

Oprócz metody `sum` dostępne są również metody `min`, `max`, `abs`.

Na koniec należy wywołać metodę `solve()` obiektu `m` w celu rozwiązania problemu optymalizacyjnego oraz za pomocą metody `print_solution()` wyświetlić rozwiązanie problemu:

```
objective: 42
  x_1_3=1
  x_2_3=1
  x_3_3=1
  x_4_3=1
  x_5_3=1
  x_6_1=1
```

w przypadku zmiennych decyzyjnych zostaną wypisane tylko zmienne decyzyjne o wartości różnej od 0. Dodatkowe informacje o modelu (między innymi liczba zmiennych decyzyjnych oraz ograniczeń) można wypisać przy użyciu metody `print_information()`. Dodatkowo można wymusić na metodzie `solve()` wypisanie szczegółowych logów (w tym czas znalezienia rozwiązania) wywołując ją z parametrem `log_output=True`.