1 AMPL – komputerowy język opisu zadań PL

1.1 Uniwersalny język modelowania i algorytm rozwiązywania

Dwa klucze do sukcesu w badaniach operacyjnych

- język pozwalający opisać szeroką klasę zadań optymalizacyjnych oraz
- algorytm pozwalający rozwiązać wszystkie zadania opisane w tym języku.

Przykładem jest programowanie liniowe

- język wyrażeń liniowych, funkcji, równań i nierówności,
- algorytm simplex (symplex) wraz dodatkowymi metodami.

Komputerowe języki modelowania zadań LP

- American Mathematical Programming Language (**AMPL**) – najpopularniejszy.
- na laboratorium korzystamy z **GNU Linear Programming Kit (GLPK)**, zawierającego język **GNU MathProg**, który jest podzbiorem AMPL,
- są i inne języki, m.in. GAMS, MPL, Lingo.

Inne przykłady języków i algorytmów

- programowanie dynamiczne: duża klasa zadań rozwiązywanych podobnym algorytmem, ale nie ma komputerowego języka opisywania takich zadań.
- programowanie ograniczeń: wyrażenia liniowe i logiczne zapewniają ogromne możliwości definiowania zadań, ale algorytmy są jeszcze mało efektywne.

1.2 Ogólna postać zadania PL w języku AMPL

```
1  # Deklaracje zbiorów
2  ...
3  # Deklaracje parametrów
4  ...
5  # Deklaracje zmiennych
6  ...
7  # Polecenie funkcji celu
8  ...
9  # Polecenia ograniczeń
10 ...
11 # Sekcja danych
12 data;
13 ...
14 end;
```

1.3 Model AMPL dla zadania z przykładu 1.

Deklaracje zmiennych

```
# Komentarze po hash'ach
set Wyroby;  # Deklaracje zbiorów
set Stadia;

# Deklaracje parametrów
param zysk {Wyroby} >= 0;
param czas_pracy {Stadia} >= 0;
param obciazenie {Stadia, Wyroby} >= 0;

# Deklaracje zmiennej
var liczba {Wyroby} >= 0;
```

Funkcja celu i ograniczenia

```
maximize calkowity_zysk:
    sum{wyrob in Wyroby} zysk[wyrob]*liczba[wyrob];
```

Maksymalizuj całkowity zysk:
sumę po wszystkich wyrobach ze zbioru Wyroby
iloczynu zysku i liczby danego wyrobu.

```
subject to max_obciazenie {stadium in Stadia}:
    sum {wyrob in Wyroby}
        obciazenie[stadium, wyrob]*liczba[wyrob]
    <= czas_pracy[stadium];
```

Dla każdego stadium ze zbioru Stadia:
suma po wszystkich wyrobach ze zbioru Wyroby
iloczynu obciążenia i liczby danego wyrobu
musi być mniejsza lub równa czasowi pracy stadium.

Dane

```
    data;
        set Wyroby := zszywacz dziurkacz;
        set Stadia := produkcja malowanie montaz;
        param zysk :=
            zszywacz    12
            dziurkacz   8;
        param czas_pracy :=
            produkcja   80
            malowanie   100
            montaz      75;
        param obciazenie:
                        zszywacz    dziurkacz   :=
            produkcja   4           2
            malowanie   2           3
            montaz      5           1;
    end;
```

1.4 Podstawy AMPL

American Mathematical Programming Language

Na zajęciach używamy solvera, t.j. programu rozwiązującego zadania, **GNU Linear Programming Kit** z interpreterem języka **MathProg** będącego podzbiorem **AMPL**.

Literatura:

1. GNU Linear Programming Kit, *Reference Manual*, Version 4.0, Draft Edition, 2003. (http://gnuwin32.sourceforge.net/downlinks/glpk–doc.php)

2. R. Fourer, D.M.Gay, B.W. Kemighan, *AMPL - AModeling Language for Mathematical Programming*, The Scientic Press, Danvers MA, 1993.

1.4.1 Zbiory nieuporządkowane

```
1  # Deklaracja zbioru nieuporządkowanego
2  # i jego elementów w modelu
3  set DniTygodnia := {"Po","Wt","Sr","Cz","Pi","So","Ni"};
4  # Deklaracja zbioru z elementami w sekcji danych,
5  # jako podzbiór innego zbioru
6  set DniWolne within DniTygodnia;
7  # Zbiór wyliczany jako różnica innych zbiorów
8  set DniRobocze := DniTygodnia diff DniWolne;
9  # Drukowanie wylicznych zbiorów i parametrów
10 display DniRobocze;
11 data;
12     set DniWolne := So Ni;
13 end;
```

```
1 Start of display output
2 Display statement at line 9
3 DniRobocze: Po Wt Sr Cz Pi
4 End of display output
```

1.4.2 Zbiory uporządkowane i rzadkie wyliczane

```
# Parametr skalarny, całkowitoliczbowy, dodatni
param LiczbaOperacji integer >0;
# Deklaracja wyliczanego zbioru uporządkowanego
# tj. zbioru liczb całkowitych
set Operacje := 1..LiczbaOperacji by 2;
# Deklaracja zbioru złożonego (pochodnego)
# jako wynik wyrażenia ''indeksowego''
set Pary := {j in Operacje, k in Operacje: j < k};
set Pary := {(j,k) in Operacje cross Operacje: j < k};
# Wyrażenie j < k możliwe tylko dla zbioru uporządkowanego

display Operacje, Pary;
data; param LiczbaOperacji := 8; end;
```

```
Start of display output
Operacje:   1   3   5   7
Pary: (1,3) (1,5) (1,7)
            (3,5) (3,7)
                  (5,7)
```

1.4.3 Zbiory rzadkie enumerowane

```
param LiczbaWezlow integer >0;
set Wezly := 1..LiczbaWezlow by 2;

# Deklaracja zbioru złożonego - rzadkiego
set Luki1 within Wezly cross Wezly;
set Luki2 within {i in Wezly, j in Wezly: i<>j};
set Luki3 within {Wezly, Wezly};

data;
    param LiczbaWezlow := 8;
    # Enumeracja (wypisanie) wszystkich elementów
    set Luki1 :=  (1,3)  (1,5)  (3,5);
    set Luki2 :=   1 3    1 5    3 5;
    set Luki3 :     1   3   5   7   :=
                1   -   +   +   -
                3   -   -   +   -
                5   -   -   -   -
                7   -   -   -   -   ;
end;
```

1.4.4 Parametry i zmienne

```
set N := {"op1", "op2", "op3", "op4"};  # Operacje
set Luki within N cross N; # Kolejność wykonywania

# Parametr - wektor zdefiniowany na zbiorze N, dodatni
param p {N} >    0; # Czas wykonywania
param a {N} >= 0; # Liczba pracowników

# Parametr o wartościach nie mniejszych od poprzedniego
param d {j in N} >= p[j]; # Termin zakończenia
param s {Luki} >= 0; # Czas przezbrojenia

# Zmienna — wektor, nieujemna
var c {N} >= 0;

display Luki, a;
```

```
data;
    # Jeżeli nie podano inaczej to wartość 0
    param default 0: a :=  op2 3    op4 4;
    # Tabelaryczne zestawienie danych
    # zdefiniowanych na tym samym zbiorze
    param:  p   d   :=
        op1     4   5
        op2     2   7
        op3     3   6
        op4     1   4;
    # Dane dla zbioru Luki oraz parametru s
    param:  Luki: s :=
            op1 op2 3
            op1 op4 1
            op2  op3  2;
end;
```

```
Luki:   (op1,op2)   (op1,op4)   (op2,op3)
a[op2] = 3  a[op4] = 4
```

1.4.5 Funkcja celu i ograniczenia

$\min \sum_{j \in \mathcal{N}} c_j x_j$ — Minimalizacja danej funkcji celu

```
1  minimize FunkcjaCelu:
2      sum {j in N} c[j] * x[j];
```

$\sum_{i \in \mathcal{M}} \sum_{j \in \mathcal{N}} a_{ij} y_{ij} \geqslant d$ — Jedno ograniczenie

```
3  subject to DotrzymanieJednegoWarunku:
4      sum {i in M, j in N}
5          a[i,j] * y[i,j] >= d;
```

$\sum_{j \in \mathcal{N}} a_{ij} y_{ij} \geqslant b_i, \quad \forall i \in \mathcal{M}$ — Grupa podobnych ograniczeń

```
6  s.t. DotrzymanieWarunkuDlaKazdego {i in M}:
7      sum {j in N}
8          a[i,j] * y[i,j] >= b[i];
```

$y_{ij} \geqslant b_i, \quad \forall i \in \mathcal{M}, \forall j \in \mathcal{N}$ — Grupa ograniczeń

```
9  s.t. DotrzymanieWarunkuDlaKazdejPary  {i in M, j in N}:
10     y[i,j] >= b[i];
```

Zamiast **subject to** wystarczy napisać s.t.

$\sum_{i \in \mathcal{M}: b_i > 0} \sum_{j \in \mathcal{N}: i > j} a_{ij} y_{ij} \geqslant d$ — Suma warunkowa

```
11 SumaWarunkowa:
12     sum {i in M, j in N:
13         b[i] > 0 and i > j}  # Warunek
14             a[i,j] * y[i,j] >= d;
```

Nie trzeba też powtarzać **subject to** przy każdym ograniczeniu, wystarczy raz, przed pierwszym ograniczeniem.

$\sum_{j \in \mathcal{N}: a_{ij} > 0} a_{ij} y_{ij} \geqslant b_i, \quad \forall i \in \mathcal{M} : b_i > 0$ — Warunkowa grupa ograniczeń

```
15 OgraniczenieWarunkowe_SumaWarunkowa
16     {i in M:
17         b[i] >0}:      # Warunek
18             sum {j in N:
19                 a[i,j] > 0}      # Warunek
20                     a[i,j] * y[i,j] >= b[i];
```