Modelowanie matematyczne

ZIMPL – kontynuacja

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Struktura pliku Zimpl (.zpl)

- Po analizie i zrozumieniu istoty rzeczywistego problemu, warto jest podzielić proces transformacji problemu na model matematyczny na następujące etapy:
  1. **Zbiory**
     - Ile obiektów występuje w problemie?
     - Co je rozróżnia?
     - Jak mogą być pogrupowane?
     - Czy potrzebne będą zbiory pomocnicze (np. zbiór indeksów, zbiór potęgowy, iloczyn kartezjański)?
  2. **Parametry**
     - Jakie dane (liczbowe) są niezbędne do opisu konkretnej instancji problemu?
  3. **Zmienne**
     - Co należy zaplanować, co podlega optymalizacji?
     - W jaki sposób możemy wpływać na końcowy wynik?
     - Jakiego typu są zmienne (binarne, całkowitoliczbowe, zmiennoprzecinkowe)?
  4. **Funkcja celu**
     - Jaki jest najważniejszy cel optymalizacji?
  5. **Ograniczenia**
     - Jakie są wewnętrzne relacje między obiektami w modelu?
     - Jakie są ograniczenia na wartości zmiennych, dlaczego zmienne nie mogą mieć wartości nieskończonych?
- Ważniejszym poleceniem Zimpla jest #
- Oznacza ono początek komentarza (reszta linii jest ignorowana)
  - `# to jest komentarz`

Wyrażenia

- Zimpl ma dwa podstawowe typy danych: tekstowe (napisy, string) i liczbowe.
- **Tekst** (string) jest ograniczony przez podwójne cudzysłowy, np. `"Hello World!"`.
- Teksty można dodawać (konkatenacja): `"Hello " + "World!"="Hello World!"`.
- Funkcja `substr(s,b,l)` zwraca podciąg *s*, zaczynający się od pozycji *b* i długości *l*.
    - `substr("Hello World",5,3) = "o W"`
- Długość tekstu *s* można sprawdzić funkcją `lenght(s)`.
    - `length("Hello World") = 12`
- **Liczby** zapisuje się standardowo, np: `2, -6.5, 5.2e-6`.
- Zimpl rozpoznaje podstawowe wyrażenia matematyczne:
    - Działania arytmetyczne: `a+b, a-b, a*b, a/b, a mod b, a^ b (a**b), a!`
    - Wartość bezwzględna: `abs(a)`
    - Znak liczby: `sgn(a)`
    - Zaokrąglanie: `round(a), floor(a), ceil(a)`
    - Liczby pseudolosowe w przedziale $[m, n]$: `random(m,n)`
- Do porównywania zmiennych (zarówno tekstowych i numerycznych) można używać operatorów `<,>,<=,>=,==,!=`
- Wyrażenia Boolowskie mogą być składane za pomocą operacji `and, or, not, xor`
- Dzięki konstrukcji `if ... then ... else` obliczenia mogą zależeć od spełnienia pewnych warunków:
    - `if (s=="a") then "Hello" else "World" end;`
    - `if (a > 3) then 3*a else a-3 end;`

Zbiory

- **Zbiór** jest kolekcją rozróżnialnych elementów.
- W Zimplu zbiory możemy definiować na kilka sposobów:
  - `set A := { 1, 2, 3 };`
  - `set B := { <1>, <2>, <3> }; # równy A`
  - `set C := { 1 to 3 }; # równy A, B`
  - `set D := { 1 .. 3 }; # równy A, B, C`
  - `set E := { 1 to 10 by 3 }; # daje { 1, 4, 7, 10 }`
  - `set Colors := { ”green”, ”red”, ”blue” };`
  - `set X := { <1, 2, ”x”>, <6, 5, ”y”>, <1.1, 2.2, ”abc” > };`
- Zbiory definiowane są słowem kluczowym `set`, operatorem `:=` i wyrażeniem ograniczonym klamrami `{,}`.
- Zbiory składają się z krotek. Krotka jest ograniczona przez `<,>`.
- Każda krotka występuje w zbiorze co najwyżej raz.
- Krotki składają się z elementów, które mogą być liczbami lub tekstami.
- Typy elementów muszą być identyczne we wszystkich krotkach w zbiorze.
- W szczególności, wszystkie krotki w zbiorze muszą być równej długości.
- Kilka błędnych przykładów:
  - `set F := { 1, 2, <3> };`
  - `set G := { <1, 2>, <3, 4>, <1, 2> }; # ostrzeżenie, nie błąd`
  - `set H := { <3, 4>, <2.2, 5>, <4, ”x”> };`
  - `set I := { <1>, <1, 2>, <1, 2, 3> };`

- Istnieje szereg poleceń do tworzenia nowych zbiorów ze zbiorów już istniejących.
  - Iloczyn kartezjański: `A*B` lub `A cross B` definiuje zbiór $\{(x, y) \colon x \in A, y \in B\}$
    - `set A := { 1, 2}; set B := { "a", "b"};`
    - `set C := A*B; # C = {<1,"a">,<1,"b">,<2,"a">,<2,"b">}`
  - Suma zbiorów: `A+B` lub `A union B` definiuje zbiór $\{x \colon x \in A \vee x \in B\}$
  - Przecięcie zbiorów: `A inter B` definiuje zbiór $\{x \colon x \in A \wedge x \in B\}$
  - Różnica zbiorów: `A \ B` lub `A-B` lub `A without B` definiuje zbiór $\{x \colon x \in A \wedge x \notin B\}$
  - Różnica symetryczna: `A symdiff B` definiuje zbiór $\{x \colon (x \in A \wedge x \notin B) \vee (x \notin A \wedge x \in B)\}$
    - `set A := { 1 to 4}; set B := { 3 to 6};`
    - `set C := A + B; # C = {1 to 6}`
    - `set D := A inter B; # D = {3,4}`
    - `set E := A - B; # E = {1,2}`
    - `set F := A symdiff B; # E = {1,2,5,6}`
  - Rzutowanie: `proj(A, <współrzędna>)` definiuje zbiór zawierający rzut A na podaną współrzędną.
    - `set A := {<1,2>, <1,3>, <2,3>, <5,3>, <5,5>};`
    - `set B := proj(A,<1>); # B = {1,2,5}`
    - `set C := proj(A,<2>); # C = {2,3,5}`
  - Najmniejszy/największy argument: Jeśli f jest funkcją określoną na zbiorze I, wtedy `argmin <i> in I : f(i)` (odpowiednio `argmax <i> in I : f(i)`) definiuje zbiór zawierający element I minimalizujący (maksymalizujący) wartość funcki f.
    - `set A := {1,2,3};`
    - `set B := argmin<i,j> in A*A : i+j; # B = {<1,1>}`

Zbiory indeksowane i warunkowe

- Za pomocą słowa `with` możemy definiować nowe (pod-)zbiory z istniejących przez wybór elementów spełniających wyrażenie Boolowskie.
  - `set A := { 1 to 5};`
  - `set B := { <i,j> in A*A with i >= 3 and j < 2 }; # C = {<3,1>,<4,1>,<5,1>}`
- Polecenie `indexset(A)` zwraca zbiór indeksów A.
  - `set A := { "a", "xyz", "hello world"}; set B := {10,17,24};`
  - `set I := indexset(A); # C = {1,2,3}`
  - `set J := indexset(B); # C = {1,2,3}`

Używając klamerek `[ ]` możemy indeksować jeden zbiór za pomocą elementów drugiego. Jak definiować takie zbiory?

- Za pomocą operacji przypisania: listy par indeks-wartość.
  - `set I := {1,2,3};`
  - `set A[I] := <1> "a", <2> 5, 7.7, <3> 1, "c";`
- Jeśli zbiór indeksowany jest funkcją krotki indeksującej, możemy użyć jej w definicji.
  - `set K[<i> in I] := {1 to i }; # K[1] = {1}, K[2] = {1,2}, K[3] = {1,2,3}`
- Można też użyć funkcji zwracającej zbiór indeksujący.
  - Zbiory potęgowe: `powerset(I)` tworzy zbiór wszystkich podzbiorów I.
    - `set P[] := powerset(I); # P[1] = { }, P[2] = {1}, P[3] = {2}, P[4] = {3},`
      `# P[5] = {1,2}, P[6] = {1,3}, P[7] = {2,3}, P[8] = {1,2,3}`
  - Wszystkie podzbiory o ustalonej liczbie elementów: `subset(I,n)` i `subset(I,n,m)`
    - `set S[] := subsets(I,2); # S[1] = {1,2}, S[2] = {1,3}, S[3] = {2,3}`
    - `set T[] := subsets(I,2,3); # T[1] = {1,2}, T[2] = {1,3}, T[3] = {2,3}, T[4] = {1,2,3}`

- Parametry mogą być deklarowane za pomocą słowa kluczowego `param`.
  - `param c = 299792458; # prędkość światła`
- Parametry mogą być deklarowane wraz ze zbiorem indeksującym.
- W takim przypadku definicją parametru jest ciąg par. Pierwszym elementem pary jest krotka indeksująca.
  - `set I := {1,2,3}; set A := {"a","xyz"};`
  - `param a[I] := <1> "x", <2> "y"; param value[A] := <"a"> 22, "xyz" -2/3;`
- Parametry mogą być przypisane indeksom w dowolnej kolejności, nie każdy indeks musi mieć przypisany parametr.
- Zbiór indeksujący może być zdefiniowany podczas definicji parametru.
  - `param a[<i> in {1 to 4} with i mod 2 ==0] := 3*i; # a[2]=6, a[4]=12`
- Parametr może też mieć przypisaną wartość domyślną.
  - `param a[{1 to 100}] := <20> 1, <80> 0 default -1;`

Tablice parametrów

- Wielowymiarowe parametry mogą być deklarowane za pomocą słowa kluczowego `param`.
- Dane są uporządkowane w tabeli z symbolami | po bokach.
- Nagłówek określa indeksy kolumn.
- Po lewej stronie dodany jest indeks wiersza.
  - `set I := { 1 to 10 }; set J := {"a","b","c","d","e", "f" };`
  - ```
    param A[I*J] := | "a", "c", "f"|
                    |2 |  12,  -1,  10|
                    |8 |  17,   0, 3*2|;
    ```
- Indeks kolumn musi być jednowymiarowy, indeks wierszy może być wielowymiarowy.
  - ```
    param B[I*I*I] := |  1, 2,  3|
                    |2, 7| 10, 2, 17|
                    |8, 9| 11, 0, -1|;
    ```
- Za tabelą można dodać kolejne wartości oraz wartość domyślną.
  - ```
    param C[I*I] := | 3, 4|
                    | 2| 1, 3|
                    | 7| 8, 0|, <1,2> 16, <8,9> -1 default -99;
    ```
    ```

Odczyt danych i parametrów z pliku

- Dużych ilości danych nie powinniśmy umieszczać bezpośrednio do modelu .zpl.
- Znacznie wygodniej jest przechowywać je w osobnych plikach i wczytywać je stamtąd.
- Służy do tego polecenie `read` (zarówno dla zbiorów, jak i parametrów).
- Ogólna składnia:
  - `read "nazwa_pliku" as "wzorzec" [skip n] [fs s] [match s] [comment s];`
- Parametry w klamrach [ ] są opcjonalne, *n* oznacza liczbę, *s* oznacza tekst.
- Przypuśćmy że mamy plik `miasta.dat` zawierający następujące 5 linii:

```
#Nazwa;Nr;X;Y;Num
Warszawa;12;x;y;7; następna linia jest pusta

Gdynia;4;x;y;5
      "Ustrzyki Górne" ; 2 :  x y , 8
```

- Kilka przykładów:

```
set P := { read "miasta.dat" as "<1s>" };
# P = { "# Nazwa", "Warszawa", "Gdynia", "Ustrzyki Górne" }
set Q := { read "miasta.dat" as "<1s,5n,2n>" skip 1 use 2 };
# Q = { <"Warszawa",7,12>, <"Gdynia",5,4> }
param c[P] := read "miasta.dat" as "<1s> 5n" comment "#";
# c["Warszawa"] = 7, c["Gdynia"] = 5, c["Ustrzyki Górne"] = 8
```

- Każda linia jest dzielona przez: spację, tabulator, przecinek, średnik, dwukropek lub zdefiniowany separator *fs*.

Odczyt danych i parametrów z pliku (c.d.)

- Jeśli linia jest dzielona przez spację, średnik lub dwukropek, wszystkie spacje są usuwane. Tekst w podwójnych cudzysłowach nie jest dzielony, a same cudzysłowy są usuwane.
- Podobnie jak w przypadku tabel można dodać kolejne wartości lub wartość domyślną.
- Opcjonalny argument `match` może być użyty do parsowania wyrażeń regularnych.

Zmienne

- Słowo kluczowe `var` definiuje nową zmienną.
- Istnieją trzy typy zmiennych:
  - Ciągłe (o wartościach rzeczywistych): `var a real;`
  - Dyskretne (o wartościach całkowitych): `var n integer;`
  - Decyzyjne (o wartościach binarnych): `var x binary;`
- Zmienne rzeczywiste lub całkowite mogą mieć dolne i górne ograniczenia.
- Domyślnymi ograniczeniami są 0 i $+\infty$.
- Poniższe polecenia są równoważne:
  - `var a;`
  - `var a real;`
  - `var a real >= 0 <= +infinity;`
- Inne przykłady:
  - `var z real >= -2 <= 25;`
  - `var a real >= -infinity <= +infinity;`
- `infinity` nie oznacza "matematycznej" nieskończoności, tylko nieskończoność "komputerową".
- Zmienne binarne zawsze są ograniczone przez 0 i 1.
- Zmienne mogą być indeksowane zbiorami.
  - `set A := { 1 to 100 };`
  - `var x[A] binary; # tworzy zmienne binarne x[1], x[2], ..., x[100]`
  - `var y[<a> in A] integer >= 1 <= 3*a;`

Przykład: Problem plecakowy

- Dysponujemy plecakiem o ograniczonej pojemności 12 kg.
- Dana jest lista przedmiotów, które możemy zapakować do plecaka, wraz z ich wagami (w kg).

Przedmiot	1	2	3	4	5	6
Waga	4	3	6	3	3	2

- Całkowita waga wszystkich przedmiotów przekracza pojemność plecaka.
- Musimy zatem dokonać wyboru, które przedmioty zabierzemy.
- Każdy przedmiot ma określoną wartość (wyrażoną w skali liczbowej)

Przedmiot	1	2	3	4	5	6
Wartość	7	5	9	4	2	1

- Naszym celem jest taki wybór przedmiotów do plecaka, by zmaksymalizować ich łączną wartość.

Przykład: Problem plecakowy

- Wprowadzamy zmienne binarne $x_1, \ldots, x_6$.
- Jeśli x_i, przedmiot i zostaje zabrany do plecaka. W przeciwnym przypadku nie zabieramy przedmiotu i.
- Jedynym ograniczeniem jest pojemność plecaka.

$$4x_1 + 3x_2 + 6x_3 + 3x_4 + 3x_5 + 2x_6 \leq 12$$

- Naszym celem jest maksymalizacja całkowitej wartości.

$$\max 7x_1 + 5x_2 + 9x_3 + 4x_4 + 2x_5 + x_6$$

- Jest to zagadnienie programowania całkowitoliczbowego, gdyż nie możemy zabrać ułamkowej części przedmiotu.
- Przyjmując, że zmienne x_i mogą mieć dowolne (ciągłe) wartości ze zbioru $[0, 1]$, otrzymalibyśmy pewne przybliżenie naszego problemu.
- Nazywamy je **relaksacją** problemu wyjściowego.
- Każde dopuszczalne rozwiązanie modelu wyjściowego jest dopuszczalnym rozwiązaniem modelu po relaksacji.
- Zatem optymalne rozwiązanie modelu po relaksacji jest górnym ograniczeniem na optymalne rozwiązanie problemu wyjściowego.

- Ogólna postać ograniczenia to: `subto nazwa : wyr. porównanie wyr.`
- Nazwą może być dowolny tekst zaczynający się od litery.
- Wyrażenie może być takiego samego typu jak funkcja celu.
- Porównaniem może być: `<=`, `==`, `>=`.
  - `subto plecak: sum <i> in I do i*x[i] <= 10;`
- Za pomocą słowa kluczowego `forall` można definiować wiele ograniczeń:
  - `set V: {1,2,3}; set A:= {<1,2>, <1,3>, <2,3>};`
  - `subto przeplyw : forall <j> in V do`
    `sum <i,j> in A do x[i,j] == sum <j,k> in A do x[j,k];`
  - `# flow_1 : 0 = x[1,2] + x[1,3]`
    `# flow_2 : x[1,2] = x[2,3]`
    `# flow_3 : x[1,3] + x[2,3] = 0`
- Za pomocą polecenia `if .. then .. else .. end` można definiować kilka wariantów ograniczenia.
  - `subto c1: forall <i> in I do`
    `if (i mod 2 == 0) then 3*x[i] >= 4`
    `else -2 * y[i] <= 3 end;`
  - `subto c2: forall <i> in I do`
    `if (i mod 2 == 0) then 3*x[i] else -2 * y[i] <= 3 end;`

Definicje funkcji

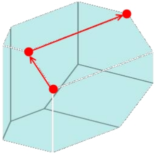
- Zimpl pozwala na definiowanie własnych funkcji.
- Definicja funkcji określa typ wartości zwracanej.
- Definicja funkcji musi rozpoczynać się od: `defnumb`, `defstrg`, `defbool` albo `defset`.
- Po tym następuje nazwa funkcji i jej argumenty w nawiasach ( ).
- Po operatorze przypisania := opisujemy samą funkcję.
  - `defnumb dist(a,b) := sqrt(a**2 + b**2);`
  - `defstrg kolor(a) := if a<=0 then "czarny" else "czerwony" end;`
  - `defbool poprawne(a,b) := a < b and a >= 10 or b < 5;`
  - `defset wieksze(i) := { <j> in A with j > i};`

Debugowanie

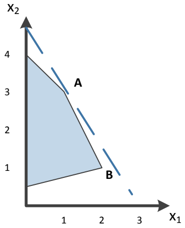
- Polecenie `do print` wypisuje na ekran wartości numeryczne, teksty i zbiory.
- Przydaje się to do sprawdzania, czy jakaś część obliczeń dała oczekiwany wynik.
- Polecenie `print` może być połączone z `forall`.
  - `set I := { 1 to 10 };`
  - `do print I;`
  - `do print ”Liczba elementów w I:”, card(I);`
  - `do forall <i> in I with i > 5 do print sqrt(i);`
- Polecenie `do check` sprawdza wyrażenie logiczne.
- Jeśli wynikiem sprawdzenia jest fałsz, obliczenia zostają przerwane.
  - `do forall <i> in I do check i < 10;`

Programowanie liniowe

- Przy pomocy zagadnienia programowania liniowego można zamodelować bardzo problemów i zagadnień występujących w przemyśle.



- Zaletą modelów liniowych jest to, że szybko można znaleźć dla nich rozwiązanie optymalne.
- Służy do tego np. metoda sympleks.



- Idea metody sympleks opiera się na obserwacji, że wartość optymalna jest osiągana na wierzchołku obszaru dopuszczalnych rozwiązań.

Funkcja celu

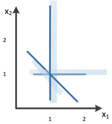
- Pewna firma może być zainteresowana
  - maksymalizacją zysku
  - minimalizacją kosztów
  - maksymalizacją użyteczności
  - maksymalizacją/minimlizacją liczby pracowników
  - maksymalizacją satysfakcji klientów
  - maksymalizacją szansy przetrwania na rynku
- Może się również zdarzyć, że nie ma funkcji celu. W takiej sytuacji chcemy tylko sprawdzić, czy dane ograniczenia są w ogóle możliwe do spełnienia.
- Typową sytuacją zastosowania programowania liniowego jest optymalizacja jednej funkcji celu.
- Jednak programowanie liniowe może być wykorzystane także w sytuacji, gdy celów jest kilka.

Jedna funkcja celu

- W modelu liniowym zakładamy, że koszt produkcji towaru w sposób liniowy zależy od jego ilości.
- To znaczy, że kosz produkcji jednostki towaru jest taki sam dla każdej jednostki.
- Nie bierzemy zatem pod uwagę kosztów rozpoczęcia produkcji ani administracji.
- Tak rozumiany koszt będziemy nazywać kosztem jednostkowym.
- Nie należy mylić go z kosztem średnim.
- W sytuacji, gdy koszty rozpoczęcia produkcji i administracji są stałe, a wyprodukowanie każdej następnej jednostki ma taki sam koszt, możemy użyć **modelowania całkowitoliczbowego**.

- Różne funkcje celu **zazwyczaj** prowadzą do różnych rozwiązań, nawet przy tych samych ograniczeniach.
- Czasami jednak niezależnie od wyboru funkcji celu rozwiązanie jest takie samo.

$$\begin{aligned} x_1 + x_2 &\leq 2 \\ x_1 &\geq 1 \\ x_2 &\geq 1 \end{aligned}$$



- Te ograniczenia definiują dokładnie jedno rozwiązanie: $x_1 = x_2 = 1$, niezależnie od doboru funkcji celu.
- Czasami w praktycznych problemach zdarza się, że istnieje tylko jedno dopuszczalne rozwiązanie.
- Powinniśmy jednak być podejrzliwi, jeśli model wskazuje to samo rozwiązanie niezależnie od doboru funkcji celu.
- O ile nie popełniliśmy błędu w formułowaniu modelu oznacza to, że do znalezienia rozwiązania wystarczy rozwiązać układ równań liniowych (a nie zagadnienie PL).
- W dalszej części wykładu będziemy zakładać, że mamy do czynienia z "prawdziwym" zagadnieniem programowania matematycznego, tzn. dobór funkcji celu ma istotny wpływ na rozwiązanie.

- W sytuacji, gdy w modelu jest wiele funkcji celu, mamy kilka możliwości poradzenia sobie z tym problemem.
  - Rozwiązać zagadnienie dla każdej funkcji celu osobno.
  - Jako funkcję celu przyjąć kombinację liniową wszystkich funkcji celu. Kluczowe jest wtedy odpowiednie dobranie współczynników tej kombinacji.
  - Wybrać jedną z pośród danych funkcji celu, a dla pozostałe dodać jako ograniczenia (z pewną satysfakcjonującą nas wartością jako prawą stroną).
  - Jeśli mamy funkcje celu $f_1, \ldots, f_m$, które chcemy maksymalizować, możemy zamiast nich maksymalizować $\min\{f_1(x), \ldots f_k(x)\}$.
    - Dodajemy nową zmienną y.
    - Do ograniczeń dodajemy $f_i(x) \geq y$ dla każdego $i \in \{1, \ldots, m\}$.
    - Funkcją celu jest $\max y$.