AMPL Hjälp

1 Att komma igång

Ni kan hämta filer under /users/mdstud/ouo02/data. De filer som är av intresse är farm.mod, farm.dat farm2.mod samt farm2.dat. farm innehåller det deterministiska problemet, farm2 innehåller ett problem med tre scenarier.

Exemplet i detta papper finns även det, och filerna heter nisse.mod och nisse.dat. I detta papper är problemen lösta med CPLEX. Om man inte gör något kommer minos att användas vilket fungerar bra så länge inga heltalskrav finns i modellen. För modeller med heltaliga variabler måste man använda cplex. Man byter till cplex genom att ge kommandot `options solver ampl_cplex`.

2 Farm-exemplet

Allt i detta papper relaterar till farm-exemplet. Exemplet består av två filer, farm.mod vilken innehåller den matematiska modellen och farm.dat vilken innehåller data till modellen.

För att prova farm-exemplet skriver ni
```
ampl: model farm.mod;
```

Kommandot laddar in modellfilen farm.mod i AMPL. Skriv nu kommandot
```
ampl: show;
```

AMPL listar nu alla saker som finns i minnet. Utskriften borde vara

```
parameters:
max_land        planting_cost   quota           selling_price
penalty         purch_price     req_amount      yield

sets:   CROPS   QUOTA_CROPS   REQ_CROPS

variables:   buying   farming   selling   selling_at_penalty

constraints:   Area   Mass_balance   Quotas   Required

objective:   profit
```

Detta är alla komponenter som finns i modellen. Skriver vi nu
```
ampl: show planting_cost;
```

så får vi utskriften

```
param planting_cost{CROPS};
```

vilket är definitionen av vår parameter planting_cost. Mer om detta senare.

Om vi vill se värdet av planting_cost skriver vi istället
```
ampl: display planting_cost;
```

Nu borde vi få svaret

```
Error executing "display" command:

        No values for planting_cost.
```

Detta beror på att vi än så länge bara har specificerat att det existerar en parameter som heter planting_cost, men ej fyllt den med innehåll. Det gör vi genom att ladda in parametrarna till problemet med kommandot
`ampl: data farm.dat;`

Ett nytt försök med `display platning_cost` ger oss nu

```
planting_cost [*] :=
Beets  260
 Corn  230
Wheat  150
;
```

vilket är en tabell över kostnaderna för att plantera en hektar mark.

De variabler vi har i modellen är buying, farming, selling och selling_at_penalty. Skriver vi kommandot
`ampl: display selling;`

så får vi

```
selling [*] :=
Beets  0
 Corn  0
Wheat  0
;
```

vilket är hur mycket vi säljer av varje gröda. Alla variabler är noll då vi ännu inte har löst problemet. Vi löser genom kommandot
`ampl: solve;`
vilket ger utskriften

```
CPLEX 5.0.1: optimal solution; objective 118600
5 iterations (2 in phase I)
```

Ni kan själva skriva kommandofiler för att automatisera vissa uppgifter (se nedan).

Nu kan vi titta på variabeln selling varvid vi får

```
ampl: display selling ;
Beets  6000
 Corn     0
Wheat   100
;
```

Om vi försöker ladda in problemet en gång till så får vi följande problem

```
farm.mod, line 2 (offset 34):
```

```
        CROPS is already defined
context:  set  >>> CROPS; <<<                    # all crops

farm.mod, line 3 (offset 75):
        REQ_CROPS is already defined
context:  set  >>> REQ_CROPS  <<< within CROPS;  # crops with minimum requirements, subset o

.
.
```

Detta beror på att AMPL redan har komponenter i minnet. När vi försöker ladda in problemet igen tror AMPL att vi försöker specificera nya komponenter med samma namn vilket ger krockar. Om man vill ladda in en ny version av problemet måste man först skriva
`ampl: reset;`
Vill man bara radera parametrarnas värde så skriver man
`ampl: reset data;`

Vill man bara ändra en enskild parameter kan man göra detta med kommandot let.
`ampl: let planting_cost["Corn"]:=100;;`
ändrar kostnaden för att plantera majs till 100 $/ha. Efter ändringar så måste naturligtvis problemet lösas om för att ändringarna skall ta effekt.

Löser vi problemet igen får vi

```
CPLEX 5.0.1: optimal solution; objective 132000
1 iterations (0 in phase I)
```

Vinsten har ökat då det är billigare att plantera. Tittar vi på det nya värdet av selling `display` så får vi

```
ampl: display innehåll;
selling [*] :=
Beets  6000
 Corn   120
Wheat     0
;

Kommandot {\tt let} kan även användas på variabler.
\\{\tt
ampl: let selling["Beets"]:=10000;
}\\
sätter tvingar försäljningen av betor till 10000;
Om vi vill att det nya värdet på variabeln skall bevaras även när vi löser om
problemet så måste vi fixera värdet.
Detta görs med kommandot fix.
\\{\tt
ampl: fix selling["Beets"];
}\\
fixerar det nya värdet av försäljning av betor;
```

3 AMPL-språkets viktigaste komponenter.

När man specificerar en modell i AMPL-språket så skriver man två olika filer. En modellfil, som innehåller själva modellen, och en datafil vilken innehåller data för ett specifikt problem. Om vi tittar på farming-exemplet så säger vi i modellfilen att vi vill odla grödor för att tillgodose eget behov samt sälja för pengar. I data-filen skriver vi vilka grödor vi skall odla, och hur mycket det kostar att köpa och sälja olika grödor. I datafilen kan vi inte införa några nya mängder eller definiera parametrar som inte redan finns beskrivna i modellfilen. Datafilen fyller alltså modellens skal med innehåll.

De kommandon som är nödvändiga då man specificerar sin modell är:

set	:	deklarerar en mängd
param	:	deklarerar en parameter
var	:	deklarerar en variabel
minimize	:	deklarerar en målfunktion
subject to	:	deklarerar ett bivillkor.

3.1 set

I farming-exemplet står det i modellfilen
`set CROPS;`

Det betyder att vi deklarerar en mängd grödor vi vill odla. I datafilen skriver vi
`set CROPS := Beets Corn Wheat;`

Då fyller vi mängden med innehåll, vi säger att vi odlar tre olika grödor och att dessa är majs, vete och betor.

Om vi hade försökt att deklarera detta set för första gången endast i datafilen så hade AMPL vägrat. I datafilen får vi endast fylla de set som tidigare deklarerats i modellfilen.

På mängder (set) kan man använda olika operatorer. Den mest användbara är union. Om bonden t.ex hade haft två olika produktgrupper

```
set CROPS;
set CATTLE;
```

så hade vi kunnat bilda mängden allt vi kan producera genom att i modellfilen skriva
`set PRODUCTS := CROPS union CATTLE;`
Nu kan vi definiera bivillkor (se nedan) vilka skall gälla på setet PRODUCTS men vi kan även ha bivillkor vilka bara gäller setet CROPS.

På samma sätt kan man bilda skärningen av set genom kommandot `inter`.

3.2 param

Parametrar deklareras genom ordet param. Om vi till exempel ville att bonden skulle räkna med moms (vat) så skulle vi kunna definiera en parameter för detta genom att i modellfilen skriva
`param vat;`
vilket gör att begreppet vat finns i modellen.

I data-filen skriver vi
`param vat := 0.25;`

för att tala om den faktiska moms-satsen.

Vi kan även indexera våra parametrar över mängder. I farming-exemplet har vi till exempel deklarationen
`param yield CROPS;`
Detta betyder att vi för varje element i mängden CROPS anger hur mycket avkastning vi får per hektar.

Parametern fylls sedan med innehåll i data-filen genom kommandot

```
param  yield  :=
Beets    20
Corn     3
Wheat    2.5;
```

Det går även att ha två eller flera dimensioner på sina parametrar Vi kan t.ex definiera
`param kalle{SET1,SET2}; .`

3.3 var

Variabler definieras även de i modellfilen. Variabler används aldrig i datafilen. I farm har vi deklarationen
`var farming{CROPS} >= 0 ;`

Detta deklarerar en variabel för hur mycket vi odlar av varje gröda, vidare ser vi till at vi aldrig odlar negativa mängder av en gröda.

Om vi deklarerar variabeln x som
`var x <= 5, >= 3;`
så kan vi ange både en övre och nedre gräns.

Då får vi skriva som följer (i modellfilen). Först definierar vi två set

```
set SET1;
set SET2;
```

Sedan definierar vi en övre gräns för variabeln. (Parametern måste fortfarande fyllas med innehåll i data-filen).

```
param max_för_y{SET1,SET2};
```

Sedan måste vi tilldela gränsen för varje enskild medlem av vektorn x vilket görs genom

```
var y{i in SET1,j in SET2} >= 0, <= max_för_y[i,j];
```

Vill vi ange att en variablel skall vara heltalig görs detta med hjälp av nyckelordet `integer` och binära variabler markeras med `binary`. Vill vi till exempel ha en heltalig variabel mella 3 och 5 så anges detta med

```
var x integer <= 5, >= 3;
```
och en binär variabel deklareras som

```
var z binary;
```

3.4 minimize

Vi måste ha en målfunktion för att kunna ha något att optimera. En målfunktion deklareras med orden `minimize` resp `maximize`. I farming-exemplet har vi

```
maximize profit:sum{i in CROPS} selling[i]*selling_price[i]
         - sum{i in REQ_CROPS} buying[i]*purch_price[i]
         - sum{i in QUOTA_CROPS} penalty[i]*selling_at_penalty[i]
         - sum{i in CROPS} planting_cost[i]*farming[i];
```

Notera att man kan göra radbrytningar nästan över allt (dock ej inne i ord). AMPL fortsätter att läsa och tolkar allt fram till ";som en sats.

Man kan summera över fler index genom att bara addera dem med ","emellan.

3.5 subject to

Till sist så har vi bivillkor. I farming-exemplet har vi bland annat bilvillkoren .

```
subject to Mass_balance {i in CROPS diff REQ_CROPS}:
      farming[i]*yield[i]>= selling[i];
```

Om vi bryter ned detta så står det på första raden
`subject to Mass_balance {i in CROPS diff REQ_CROPS}:`

Detta betyder att vi skapar ett villkor för varje i tillhörande mängden CROPS som ej finns i REQ_CROPS. Sedan (efter :) kommer själva villkoret. Detta säger att den yta vi odlar gånger avkastningen på denna yta måste överstiga den mängd vi säljer. På samma sätt som med summor så kan man indexera över flera set separerade med ",".

Vill man ha ett enstaka villkor så struntar man bara i `{.....}` i definitionen (se t.ex. Area i farm.mod).

4 Analys av lösningen

Genom att ange suffix på variabler och bivillkor kan man få ut extra information. Genom att skriva
`display Required.slack;`
så får vi ut värdet på slackvariabeln tillhörande bivillkoren Required.samtliga Lagerbivillkor.

```
Required.slack [*] :=
 Corn  0
Wheat  0
;
```

Genom kommandot
`display Required.dual;`
får vi värdet på dualvariabeln tillhörande required.

Genom att skriva
`ampl: display Required.dual, Mass_balance.dual;`
så kan man få många olika uppgifter i en tabell. Man kan naturligtvis ange fler parametrar separerade med ",".

5 Mest sannolika missarna

Q: Jag skrev ett kommando men inget hände, när jag sedan skriver in nästa så får jag märkliga fel av typen

```
syntax error
context: >>..............
```

A: Du har nog glömt ";"efter förra kommandot. Om inget händer när ett kommando skrivs, titta på prompten. Står det
`ampl?`
så väntar ampl fortfarande på slutet på förra kommandot.

Q: Jag får ett felmeddelande med hänvisning till en viss rad när jag försöker ladda in min modell. När jag tittar på raden ser den jätterätt ut. Vad göra?

A: Åter igen ";". Titta på raden innan och se om den avslutas med ";". AMPL varnar på närmast efterföljande rad då det är där satsen avslutas på ett felaktigt sätt.

Q: När jag försöker få in min modell så måste jag skriva kommandona
`reset; model minmodell.mod; data minmodell.dat;` väldigt många gånger i följd. Kan man underlätta detta på något sätt?

A: Ja, skriv en kommandofil. Skapa helt enkelt en fil som tex heter minfil. Skriv in kommandona du vill upprepa många gånger i minfil. Kommandona körs genom att skriva `include minfil`.

Q: Jag få fel av typen
`invalid subscript innehåll[j,i]` trots att jag borde indexerat rätt.

A: Kontrollera ordningen på dina index.

6 Mer information

En uppsjö av färdiga exempel finns på
http://www.ampl.com/cm/cs/what/ampl//BOOK/EXAMPLES/index_files.html .

Mer info om ampl finns på http://www.ampl.com/cm/cs/what/ampl/.
Denna sida är närmast att betrakta som överkurs och innehåller mest info om scriptspråk och generiska sätt att komma åt samtliga bivillkorl etc.

En rapport om en äldre version av AMPL med div. exemplel och definition av satser finns på
http://slack.iems.nwu.edu/ampl/papers.html
där man även kan hitta ett kapitel ur boken om AMPL.