AMPL：数理計画のためのモデリング言語

⇄

第 15 章

ネットワーク線形計画問題

節点と枝でモデルを描き、*node* と *arc* で *AMPL* に語らせる

転送　最大フロー　最短路　割当　node / arc

AGENDA

本章の内容

15.1	**最小費用転送モデル**	節点・枝の集合、供給/需要、費用/容量、収支制約 Balance
15.2	**その他のネットワークモデル**	最大フロー・最短路・輸送/割当モデルへの応用
15.3	**node と arc によるネットワークモデルの宣言**	net_in / net_out、from・to・obj 句で図に沿って記述
15.4	**node と arc 宣言の規則**	収支条件の書式、側面制約・側面変数との組み合わせ
15.5	**ネットワーク線形計画問題を解く**	純粋ネットワーク構造と専用アルゴリズムによる高速化

ネットワークフローの基本構造

- 節点（node）＝円　… 都市・交差点などの地点
- 枝（arc）＝矢印　… ある節点から別の節点へ走る有向リンク
- フロー（flow）＝矢印の向きに沿って枝を移動する量
- 枝には「費用, 容量」の2つの数字が付く（図の例）
- **決定変数＝枝上のフロー量、制約＝上下限と各節点でのフロー保存の2種類だけ**

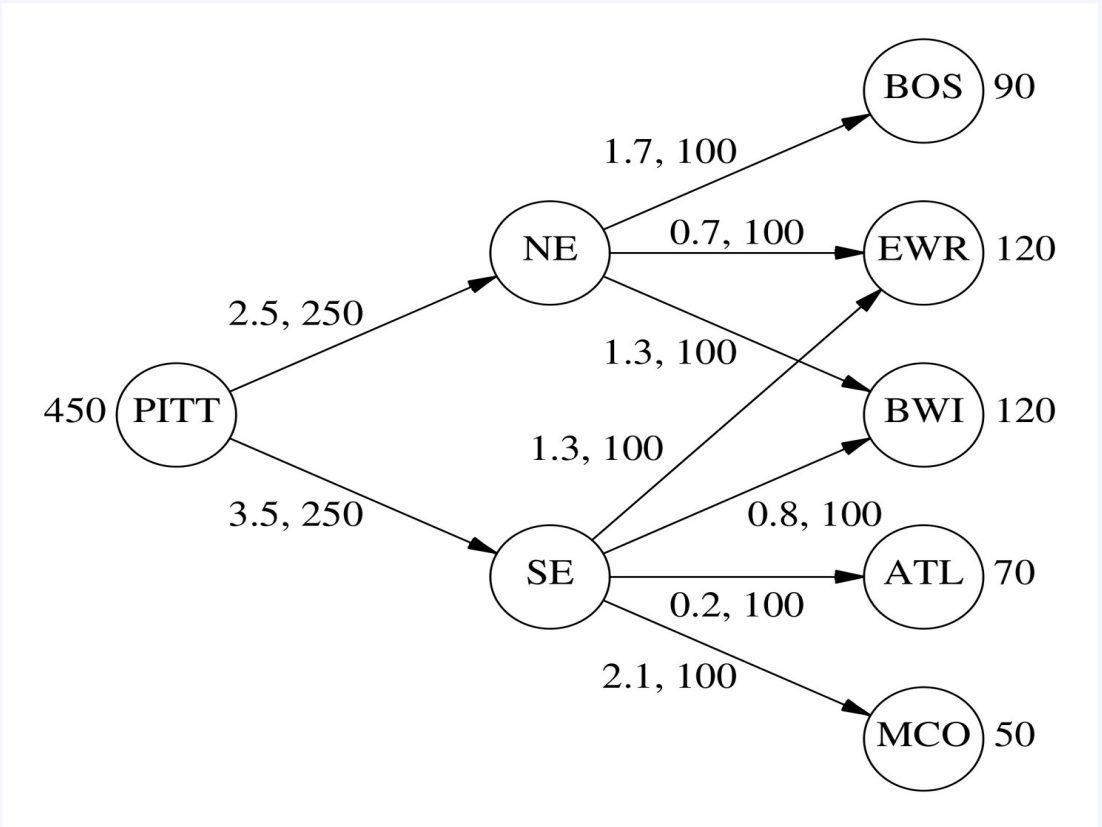

図15-1　有向ネットワーク（各枝の数字は「費用, 容量」）

例題：都市間パッケージ輸送

- PITT工場が来週450千パッケージを製造
- NE・SEの2配送センターが工場から受け取り、各倉庫へ転送
- BOS・EWR・BWI・ATL・MCOの5倉庫がそれぞれ必要量をもつ
- 各リンクには「1000パッケージ当たりの費用」と「輸送上限（容量）」がある
- **目的：容量を守り需要を満たす最小費用の輸送計画を求める**

倉庫ごとの需要（千パッケージ）

倉庫	需要
BOS	90
EWR	120
BWI	120
ATL	70
MCO	50

- 供給合計 450
- ＝
- 需要合計 450
- （90+120+120
- +70+50）

450
PITT の供給（千パッケージ）

5
倉庫（需要地点）の数

9
リンク（枝）の本数

一般的な転送モデル（1）— 宣言と目的関数

- 都市の集合 CITIES と、都市の順序対の部分集合 LINKS でリンクを表す
- 各都市に供給 supply と需要 demand（ともに ≥ 0）を定義
- 各リンクに費用 cost と容量 capacity を定義
- 変数 Ship[i,j] は非負、かつ容量以下に制限
- **目的：全リンクの cost × Ship の総和を最小化**

net1.mod（抜粋）

```
set CITIES;
set LINKS within (CITIES cross CITIES);

param supply {CITIES} >= 0;
param demand {CITIES} >= 0;

param cost {LINKS} >= 0;
param capacity {LINKS} >= 0;

var Ship {(i,j) in LINKS} >= 0,
   <= capacity[i,j];

minimize Total_Cost:
   sum {(i,j) in LINKS} cost[i,j] * Ship[i,j];
```

一般的な転送モデル（2）— 収支制約 Balance

```
subject to Balance {k in CITIES}:
   supply[k]
    + sum {(i,k) in LINKS} Ship[i,k]
   = demand[k]
    + sum {(k,j) in LINKS} Ship[k,j];
```

- 各都市 k で「供給＋入るフロー」＝「需要＋出るフロー」
- sum{(i,k) in LINKS} Ship[i,k] … k に入るすべてのリンクの和
- sum{(k,j) in LINKS} Ship[k,j] … k から出るすべてのリンクの和
- **この添字づけの約束が収支制約を簡潔に書く鍵**

「正味需要」形式

```
# 変数を左辺、定数を右辺に移すと
subject to Balance {k in CITIES}:
     sum {(i,k) in LINKS} Ship[i,k]
   - sum {(k,j) in LINKS} Ship[k,j]
   = demand[k] - supply[k];
```

入る量 − 出る量 ＝ 正味需要

正の値：倉庫都市　／　負の値：工場都市　／　零：転送都市

特化した転送モデル

- 工場・配送センター・倉庫の3種類を別々の集合で宣言
- 工場は1つだけ → 記号パラメータ p_city で表す
- DW_LINKS は「配送センター→倉庫」の対だけに限定
- **不適切な配置（倉庫間リンクなど）を構造的に排除できる**
- 一般モデルの CITIES・LINKS・supply・demand はこれらから導出

net2.mod（抜粋）

```
param p_city symbolic;
set D_CITY;
set W_CITY;
set DW_LINKS within
   (D_CITY cross W_CITY);

param p_supply >= 0;
param w_demand {W_CITY} >= 0;

set CITIES =
   {p_city} union D_CITY union W_CITY;
set LINKS =
   ({p_city} cross D_CITY) union DW_LINKS;
```

転送モデルの変形

1 不等式の収支

供給が需要を上回りうるなら P_Bal の = を <= に。

```
subject to P_Bal:
  sum{i in D_CITY} PD_Ship[i] <=
p_supply;
```

2 輸送中の損失

損失率 pd_loss を導入し、届く量を (1-pd_loss)×Ship に。

```
param pd_loss {D_CITY} >= 0, < 1;
subject to D_Bal {i in D_CITY}:
  (1-pd_loss[i])*PD_Ship[i] =
    sum{(i,j) in DW_LINKS} DW_Ship[i,j];
```

3 単位の換算

需要が箱、輸送が千パッケージなら換算係数 ppc を使う。

```
param ppc integer > 0;
subject to W_Bal {j in W_CITY}:
  sum{(i,j) in DW_LINKS}
    (1000/ppc)*DW_Ship[i,j] =
w_demand[j];
```

最大フローモデル

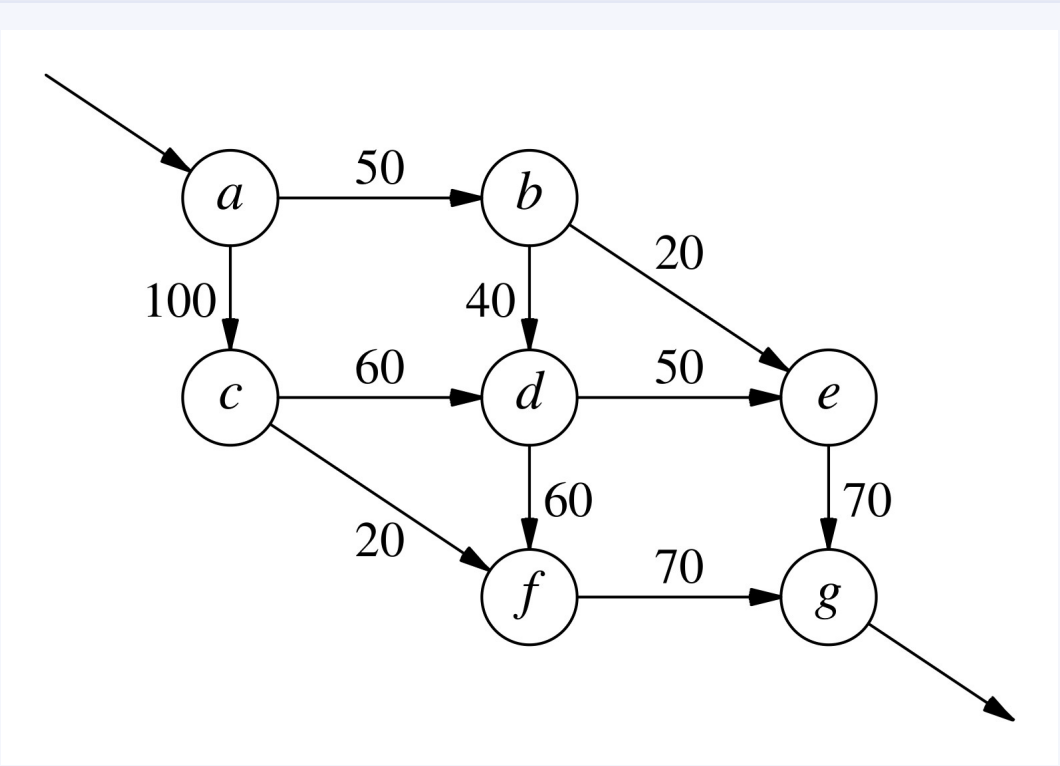

図15-5　交通流ネットワーク（数字は毎時の容量）

- a から入り g から出る最大交通流を求める

- 収支制約 Balance は入口・出口を除く各交差点だけに課す
- 目的：入口から出る総フロー Entering_Traff を最大化
- **最適値：毎時 130 台**

netmax.mod（抜粋）

```
set ROADS within
   (INTER diff {exit}) cross (INTER diff {entr});
param cap {ROADS} >= 0;
var Traff {(i,j) in ROADS} >= 0,
   <= cap[i,j];

maximize Entering_Traff:
   sum {(entr,j) in ROADS} Traff[entr,j];

subject to Balance {k in INTER diff {entr,exit}}:
  sum{(i,k) in ROADS} Traff[i,k]
   = sum{(k,j) in ROADS} Traff[k,j];
```

最短路モデル

- 同じネットワーク図（図15-5）を使い、容量を移動時間 time に読み替える
- 変数 Use[i,j] は「最短路に含まれるなら1、含まれなければ0」
- 目的：time × Use の総和（総移動時間）を最小化
- 追加の制約 Start：入口から流れ出る量をちょうど1に固定

netshort.mod（抜粋）

```
param time {ROADS} >= 0;
var Use {(i,j) in ROADS} >= 0;
minimize Total_Time:
   sum {(i,j) in ROADS} time[i,j]*Use[i,j];
subject to Start:
   sum {(entr,j) in ROADS} Use[entr,j] = 1;
```

AMPL 対話セッション

```
ampl: model netshort.mod;
ampl: solve;
MINOS 5.5: optimal solution found.
1 iterations, objective 140
ampl: option omit_zero_rows 1;
ampl: display Use;
Use :=
a b    1
b e    1
e g    1
;
```

最短路：a → b → e → g　（所要 140 分）

輸送モデルと割当モデル

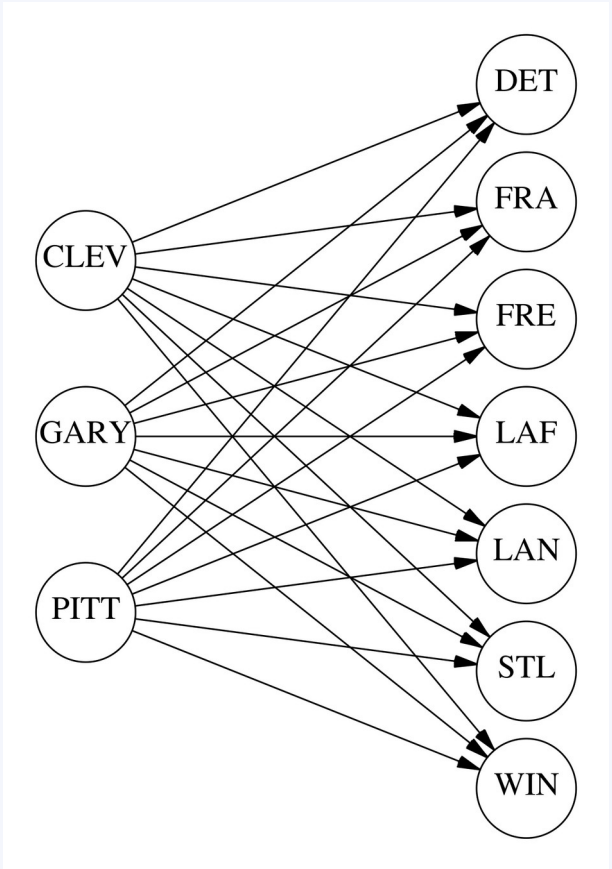

図15-8　二部ネットワーク

- 節点は発地群と着地群の2つに分かれる二部構造
- 枝はすべて発地→着地を結ぶ（発地間・着地間はなし）
- **このネットワーク上の最小費用転送＝輸送モデル／割当モデル**

netasgn.mod（抜粋）

```
set PEOPLE;  set PROJECTS;
set ABILITIES within
   (PEOPLE cross PROJECTS);

var Assign {(i,j) in ABILITIES}
   >= 0, <= limit[i,j];

subject to Supply {i in PEOPLE}:
  sum{(i,j) in ABILITIES}
     Assign[i,j] = supply[i];
```

- 係数は時間・人数・順位のいずれでもよい
- 順位を係数にすると「希望順位の合計最小化」割当になる

なぜ node と arc が必要か

- 代数的な var / subject to は変数と制約から出発し、節点・枝は式の中に暗黙に埋め込まれる
- ネットワークが複雑になるほど収支制約の式は読み取りにくくなる
- **node と arc は逆から出発：節点と枝を直接宣言し、フロー変数と収支制約を暗黙に生成**
- 図の見え方（節点が先、枝が後）にそのまま対応する

代数的な表現（読み取りにくい）

```
sum {(i,k) in LINKS} Ship[i,k]
  - sum {(k,j) in LINKS} Ship[k,j]
```

node / arc の表現（節点収支が一目瞭然）

```
net_in
```

「入るフロー − 出るフロー」を1語で表す組込みキーワード
（net_out はその符号反転）

node と arc で書き直す — 一般的な転送モデル

net1node.mod（抜粋）

```
# set / param 宣言は前と同じ

minimize Total_Cost;

node Balance {k in CITIES}:
   net_in = demand[k] - supply[k];

arc Ship {(i,j) in LINKS} >= 0,
   <= capacity[i,j],
   from Balance[i], to Balance[j],
   obj Total_Cost cost[i,j];
```

- **node Balance**
- 各 CITIES の要素に1つの節点。net_in = demand − supply で正味需要を指定
- **arc Ship**
- 上下限は var と同じ書式。from・to で結ぶ節点、obj で目的関数への係数を指定
- **minimize は名前だけでよい（項は arc 側の obj 句が供給）**

node 宣言 — 収支条件の書式

- 書式：node 名前 {添字式}: 収支条件;
- 収支条件は net_in（または net_out）と算術式を =, <=, >= で結ぶ
- 上下限つきの範囲条件（arith <= net <= arith）も書ける
- **収支条件を省略すると net_in = 0（＝ net_out = 0）が既定**

3種類の節点の例（net3node.mod）

```
node Plant: net_out = p_supply;
node Dist {i in D_CITY};
node Whse {j in W_CITY}:
   net_in = w_demand[j];
```

許される収支条件の形

```
net-expr = arith-expr
net-expr <= arith-expr
net-expr >= arith-expr
arith-expr = net-expr
arith-expr <= net-expr
arith-expr >= net-expr
arith <= net-expr <= arith
arith >= net-expr >= arith
```

net-expr は ± net_in（または net_out）に定数項 arith-expr を足したものに限られる。

arc 宣言 — from / to / obj 句

- 書式：arc 名前 {添字}, 上下限, from 節点, to 節点, obj 目的 係数;
- from・to はどちらか片方を省略してもよい（枝の片端がぶら下がる）
- 係数を from/to 句の末尾に付けると輸送損失・単位換算を表せる
- **if を使った添字式で条件つきに from / to を切り替えられる**

損失係数の例

```
arc PD_Ship {i in D_CITY} >= 0, <= pd_cap[i],
   from Plant, to Dist[i] 1-pd_loss[i],
   obj Total_Cost pd_cost[i];
```

条件つき from / to（最大フロー）

```
arc Traff {(i,j) in ROADS} >= 0,
   <= cap[i,j],
   from {if i = entr} Entr_Int,
   from {if i <> entr} Intersection[i],
   to   {if j = exit} Exit_Int,
   to   {if j <> exit} Intersection[j],
   obj Entering_Traff
      (if i = entr then 1);
```

付加制約と付加変数

付加制約 —— 多品種フロー

arc の名前は subject to の中で通常の変数として使える。製品ごとの枝流量に共同の容量制限をかけられる。

```
subject to Multi {(i,j) in LINKS}:
  sum {p in PRODS} Ship[p,i,j]
     <= cap_joint[i,j];
```

→ ネットワーク構造は壊れるが、費用・容量部分とは分離して書ける（*netmulti.mod*）。

付加変数 —— 原料からの生産

先に var で定義した通常変数を node の収支条件に含められる。

```
var Feed {f in FEEDS, k in CITIES}
   >= 0, <= limit[f,k];
node Balance {p in PRODS, k in CITIES}:
  net_out = supply[p,k] - demand[p,k]
   + sum{f in FEEDS} yield[p,f]*Feed[f,k];
```

→ *Feed[f,k] が製品ごとのネットワークを1つの線形計画問題に結びつける（netfeeds.mod）。*

純粋ネットワーク構造と解の速さ

- 各変数が高々2本の制約に、係数 +1 と −1 で1回ずつ現れる ＝「純粋な」ネットワークLP
- **供給・需要・上下限が整数なら、必ず全整数の最適解が存在**
- node / arc で書けば AMPL が自動でネットワーク構造をソルバーに伝達
- 専用のネットワークシンプレックス法で大幅に高速化されうる

代数的定式化（net3.mod）

```
CPLEX 8.0.0: optimal solution; objective 1819
1 dual simplex iterations (0 in phase I)
```

```
ampl: model net3node.mod; data net3.dat; solve;
CPLEX 8.0.0: optimal solution; objective 1819
Network extractor found 7 nodes and 7 arcs.
7 network simplex iterations.
```

同じ目的関数値 1819 だが…

node / arc 定式化ではネットワーク抽出器が7節点・7枝を検出し、専用のネットワークシンプレックス法が適用される。

まとめ

1	**ネットワーク線形計画問題**	決定変数＝枝のフロー、制約＝上下限とフロー保存だけの特別なLP
2	**転送・最大フロー・最短路・割当**	同じ変数/制約の型を、目的や収支条件の違いで使い分ける
3	**node と arc 宣言**	net_in / net_out と from・to・obj で図に沿って直接記述できる
4	**側面制約・側面変数**	多品種フローなど、ネットワークを超える拡張も自然に組み込める
5	**純粋ネットワーク構造**	整数最適解の保証と専用アルゴリズムによる高速な求解につながる