最適化・シミュレーション演習

第4回 分枝限定法

- 授業サポートページ
  http://www.shiina.mgmt.waseda.ac.jp/optsim/
- 数理計画による最適化と(離散事象型)シミュレーションに関する演習を行う．使用するソフトウェアは，AMPL(＋GurobiまたはCPLEX)，および，Simul8を想定している．
- 演習では，数理計画による最適化やシミュレーションの実践的能力を身につけることを目指す。履修者は，Cを使用できる環境を有するPCを持参すること。受講者は，実験室にて，演習で使用する C，AMPL，Simul8をダウンロードできる。

数理計画問題を解く解法(アルゴリズム)

- 「定式化」された問題は、適当な解法で解き、解を求める
- ソフトウェアを利用して解くか、自分でプログラムを作成
- 解法の種類(数理計画法)
  - 厳密解法(最適性を保証する解を求める解法)
  - 近似解法(最適性は保証しないが、良い解を早く求める)
- 列挙法:いかに「すべての」解を『列挙』するか(直接／間接)
  - 陽に(explicit)列挙／陰に(implicit)列挙
- バックトラック法(backtrack algorithm)
  - プログラムは比較的容易(列挙順序は固定)
- 分枝限定法(branch and bound method)
  - 解の列挙順序は解法の詳細に大きく依存

ナップサック問題の緩和問題
連続緩和:整数条件を緩め、$0 \leqq x_j \leqq 1$ とする

(KP-LP) max $\Sigma_{j=1}^{n} c_j x_j$
s.t. $\Sigma_{j=1}^{n} a_j x_j \leqq b, 0 \leqq x_j \leqq 1, \ \forall j$

$c_j/a_j \geqq c_{j+1}/a_{j+1}, j=1,\ldots,n\text{-}1$が成り立つと仮定してよい(効率順)

例: (KP-LP) 最大化 $z = 7x_1 + 8x_2 + 3x_3$
制約 $3x_1 + 4x_2 + 2x_3 \leqq 6$
$0 \leqq x_j \leqq 1, \ \forall j$

- 2.33(万円/kg)≧2(万円/kg)≧1.5(万円/kg)
- 緩和問題の最適解は効率順に品物を詰め込み、入りきらない品物を一部分だけ入れる $x_1 =$, $x_2 =$, $x_3 =$
- 緩和問題の最適解は0-1条件を満たしていない。
- よって、$x_1 =$, $x_2 =$, $x_3 =$ へと小数値を切り捨てると(KP)の実行可能解が得られるが、最適解ではない。
- 緩和問題の最適解を、四捨五入、切上げ、切下げなどにより整数値に丸めても、対応する整数計画問題の最適解が得られるとは限らない
- 緩和問題の最適目的関数値 ≧ 元問題の最適目的関数値 .

ナップサック問題の連続緩和問題(貪欲解)
効率の良い品物から(小数解を許し)詰め込む

(KP1) 最大化 $z = 10x_1 + 6x_2 + 2x_3$
制約 $4x_1 + 3x_2 + 2x_3 \leqq 5, \quad 0 \leqq x_j \leqq 1, \ \forall j$
緩和問題の最適解: $x_1 =$, $x_2 =$, $x_3 =$
ナップサック問題の最適解: $x_1 =$, $x_2 =$, $x_3 =$
緩和問題の最適目的関数値 ≧ 元問題の最適目的関数値 .

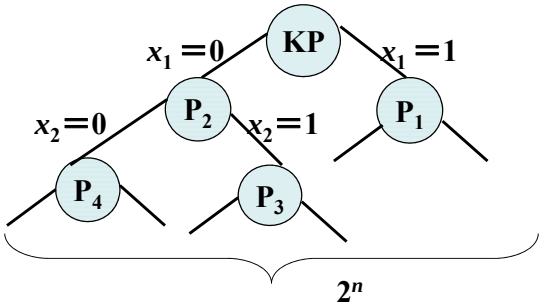
(KP2) 最大化 $z = 18x_1 + 15x_2 + 6x_3 + 2x_4$
制約 $5x_1 + 4x_2 + 2x_3 + x_4 \leqq 6, \quad 0 \leqq x_j \leqq 1, \ \forall j$
効率 ≧ ≧ ≧ なので、 の順に入れる
緩和問題の最適解: $x_1 =$, $x_2 =$, $x_3 =$, $x_4 =$
ナップサック問題の最適解: $x_1 =$, $x_2 =$, $x_3 =$, $x_4 =$
緩和問題の最適目的関数値 ≧ 元問題の最適目的関数値 .

緩和問題は、条件を緩めているので良い値が出る

分枝限定法 Branch and Bound Method

- 分枝(branching) 変数値を枝分かれしながら固定
  ===> 子問題、部分問題へ分枝
- 限定(bounding) 条件を考慮して解の範囲を限定
  ===>全列挙を行う必要がない場合もある
- 基本的用語(z^*:元問題の最適目的関数値)
- 下界値≦z^*≦上界値
- 最大化問題では、下界値は実行可能解から、上界値は緩和問題から得られる
  - 暫定(目的関数)値: これまでに見つかっている最良の解の目的関数値
  - 暫定解: これまでに見つかっている最良解
  - 分枝変数: 分枝の対象となる変数
  - 分枝頂点: 分枝の対象となる頂点(子問題)
  - 未分枝頂点: まだ探索が終わっていない頂点(子問題)

分枝操作(branching operation)



- ある問題(「親問題」)の実行可能領域をいくつかに分割して, 複数の「子問題(部分問題)」を生成する操作.
- 分枝変数: 分割に用いた変数
- 列挙木
- 子問題＝列挙木の頂点

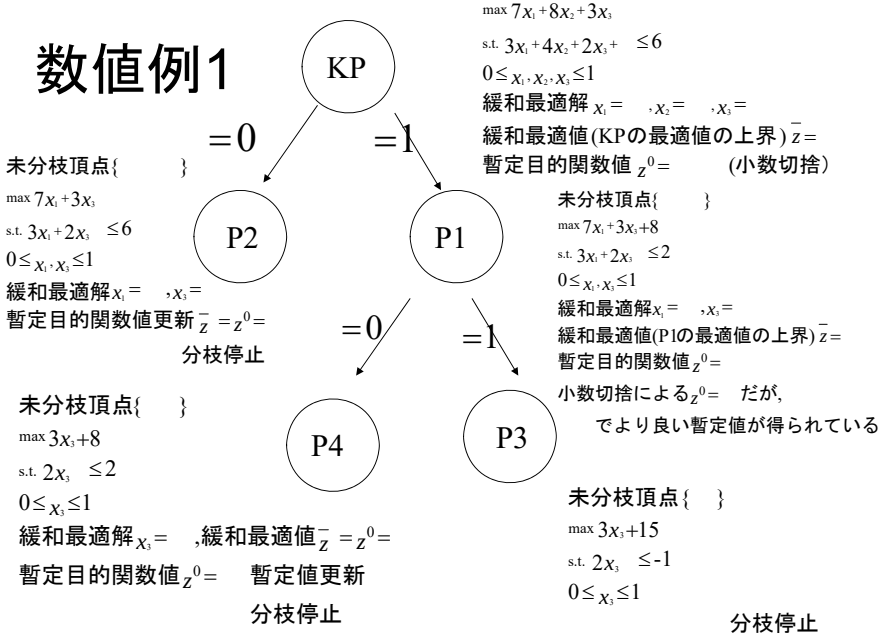
限定操作(bounding operation)　<最大化の場合>

- ある子問題に分枝操作を施す必要があるか否かを判定
- 具体的には, 緩和問題から得られる上界値が, 手元にある暫定値より小さい場合、分枝を停止
- 緩和してもこれまでに得られた解より悪い値しか得られない

0-1ナップサック問題に対する分枝限定法の設計例

1)上界値の算出方法
　線形緩和(LP)
2)下界値(実行可能解)の算出方法(KPは容易)
　LP最適解の切り下げ
3)分枝変数の選択(どの変数で枝分かれするか？)
　LP最適解で小数値をとる変数=0 問題番号を大きく(左子問題)
　LP最適解で小数値をとる変数=1 問題番号を小さく(右子問題)
4)分枝頂点(分枝子問題)の選択(どの頂点から分枝するか？)
　深さ優先規則(未分枝頂点リストから最も番号の大きいもの)
5)分枝が停止する場合
　0-1条件を満たす
　実行不可能な場合
　限定操作

数値例1



分枝限定法の実装

- 分枝限定法(BB)は、混合整数計画問題に対しては、一般的なソルバーでも使用可能
- 分枝限定法のカスタマイズ
  - 分枝切除法(分枝限定法＋切除平面法)
  - 分枝価格法(分枝限定法＋列生成法)
- 他にも、特定の細かい分枝ルールを制御したい、あるいはLPベースでない場合など
- これまではLPに基づくBBですら、実装は困難
  - 各子問題のデータ、未分枝頂点の管理が難しい
- AMPL(second edition以降)によると、反復計算(データ置換(定数、係数)、制約式、変数追加など)容易に

ナップサック問題(緩和問題)のモデルファイル

- 次のような 3 つのファイルを作業フォルダAMPLWORK (C:¥amplcml¥amplwork)に作成する。
- knapsack-relax.mod ファイル:ナップサック問題の連続緩和問題を記述したファイル
- knapsack-relax.dat ファイル:ナップサック問題のデータを記述したファイル
- knapsack-relax.run ファイル: AMPL のコマンドを記述したファイル
- ファイル knapsack-relax.mod の内容 (＃以降はコメント)

```
param Nitem;                    #品物数をNitem と定義
param weight{i in 1..Nitem};    #各品物に重さ(weight)を定義(各定数、変数の次元はNitem でよい)
param value{i in 1..Nitem};     #各品物に価値(value)を定義
param l{i in 1..Nitem};         #0-1 制約の緩和制約における下限
param u{i in 1..Nitem};         #0-1 制約の緩和制約における上限
param capacity;                 #ナップサックの容量
var x{i in 1..Nitem} >=0 ;      #各品物(入れる/入れない)に対応する変数、binary にすればMIP
maximize Profit: sum{i in 1..Nitem}value[i]*x[i];                                  #目的関数Profit
subject to Capacity_Constraint: sum{i in 1..Nitem}weight[i]*x[i] <= capacity;  #ナップサックの容量制約
subject to Upper_Lower_Bound {i in 1..Nitem}: l[i] <= x[i] <=u[i];              #0-1 制約を緩和した上下限
```

ナップサック問題(緩和問題)のコマンドスクリプト

- knapsack-relax.run ファイル: AMPL のコマンドを記述したファイル

```
model knapsack-relax.mod ;      #モデルファイル指定
data knapsack-relax.dat ;       #データファイル指定
option display_round 6;         #小数点以下6 桁表示
option solver cplex;            #ソルバにCPLEX を指定、これがないとMINOS が動く
solve;                          #問題の求解(貪欲解だが、ソルバで解く)
expand > knapsack-relax.sol;          #ファイル(knapsack-relax.sol)に定式化表示
display Profit > knapsack-relax.sol;  #ファイル(knapsack-relax.sol)に目的関数値
display x > knapsack-relax.sol;       #ファイル(knapsack-relax.sol)に最適解
display Capacity_Constraint.dual > knapsack-relax.sol;      #ナップサック制約の双対変数
display Upper_Lower_Bound.dual > knapsack-relax.sol;      #0-1 緩和制約の双対変数
                        #制約名.dual で双対変数
                        # >ファイル名 を省略すると、表示は画面に
```

- コマンドプロンプトにおいて、「ampl knapsack-relax.run 」と入力
- 最適解情報ファイル(knapsack-relax.sol)が作成される
- 演習問題の解を確認しよう

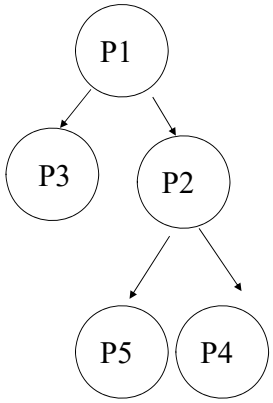
結果ファイル(knapsack-relax.sol)

```
Profit = 13.000000      #最適目的関数値

x [*] :=                #最適な x の値;
1  1.000000
2  0.750000
3  0.000000
;

#双対変数 y の値
Capacity_Constraint.dual = 2.000000

#双対変数 z の値
Upper_Lower_Bound.dual [*] :=
1  1.000000
2  0.000000
3  0.000000
;
```

未分枝頂点リスト ACTIVE_NODELIST

- 未分枝頂点を配列ACTIVE_NODELISTに保存
- 子問題を解く場合、その頂点番号が削除
- 分枝する場合、新たな頂点番号が追加
- ACTIVE_NODELIST={1}
- ACTIVE_NODELIST={2,3}
- ACTIVE_NODELIST={2}
- ACTIVE_NODELIST={5,4}
- ACTIVE_NODELIST={4}
- ACTIVE_NODELIST=φ
- 未分枝頂点リストが空集合なら終了

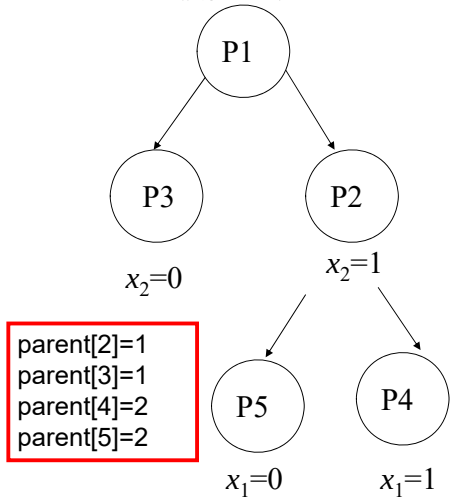
- 配列のサイズ size_activenode
- 子問題を解くと、サイズは1減少
- 分枝すると、サイズは2つ増加
- size_activenode=1
- size_activenode= 2
- size_activenode= 1
- size_activenode= 2
- size_activenode= 1
- size_activenode= 0
- 配列のサイズが0ならば終了



各頂点でのデータ保存

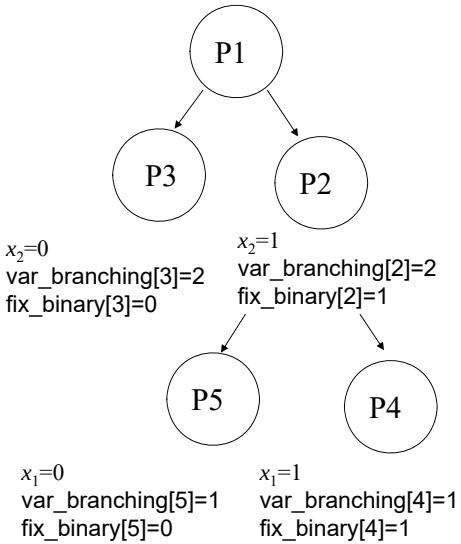
根ノードを問題1と定義
演習とは異なる

- 各子問題のデータ(定式化、knapsack-relax.datに対応するデータ)をすべての子問題について保存すると膨大な記憶領域必要
- 各子問題に加えられる0-1限定制約のみを保存
- 各頂点の親頂点を保存



子問題のデータ作成:データ構造

- 同時に解ける問題は1つ
- 子問題のデータをその都度作成
- 分枝制約を等式制約として加えるのでなく、
  l[i] <= x[i] <=u[i]
  のl[i],u[i]を書き換える
- var_branching[i]：子問題 i で変数 $x_{\text{var\_branching[i]}}$ が分枝変数
- fix_binary[i]：子問題 i で分枝変数 $x_{\text{var\_branching[i]}}$ が固定される値



子問題のデータ作成:子問題ー根への道

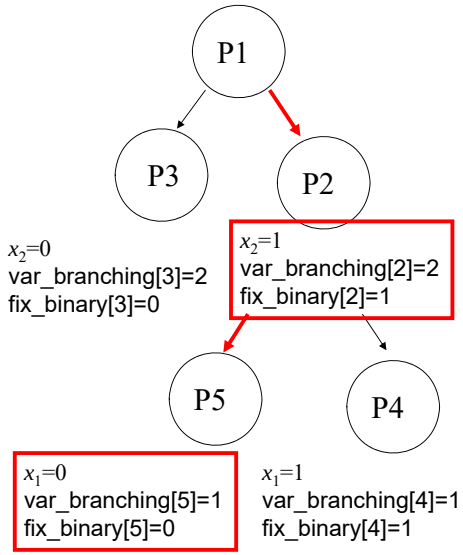
- 子問題選択は、未分枝頂点リストの末尾(奥行優先)
  ACTIVE_NODELIST[size_activenode]
- 当該子問題より、親ノードを根ノードまで探索
- 子問題5で、0=l[1]<=x_1<=u[1]=0

l[var_branching[5]]:= fix_binary[5];
u[var_branching[5]]:= fix_binary[5];

- 子問題2=parent[5]を探索
- 子問題2で、1=l[2]<=x_2<=u[2]=1

l[var_branching[2]]:= fix_binary[2];
u[var_branching[2]]:= fix_binary[2];

- 子問題2の親問題は根(頂点1)なので終了



分枝するための条件

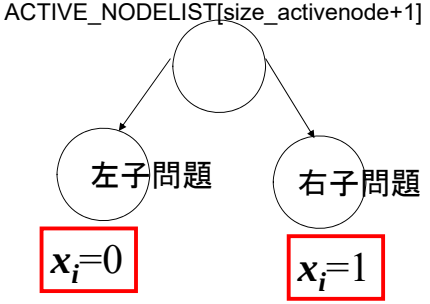
- 次の3つが同時に成り立つときのみ分枝（and条件）
- 整数解が得られない
  exists {i in 1..Nitem}
  x[i]*(1-x[i]) > epsilon
  $x_i(1-x_i)$, $i=1,...,n$ のうち、判定係数εより大きいものが存在
- 子問題が実行可能
  (solve_result="solved")
- 子問題の目的関数値が暫定値よりよい(Profit > tmpprofit)

- 左の条件より次の3つのいずれかが成り立つとき分枝停止（or条件）
- 整数解が得られた
  exists {i in 1..Nitem}
  x[i]*(1-x[i]) <= epsilon
- 子問題が実行不可能
  (solve_result="infeasible")
- 子問題の目的関数値が暫定値以下になる(これ以上分枝操作を続けても、暫定値よりよい解は得られない：限定操作) Profit <= tmpprofit

子問題への分枝

- 解く問題は常に未分枝頂点リストの末尾
- 現在までに生成された問題総番号total_node_number
- 子問題解いた時点で
  size_activenode＝
  size_activenode－1

問題番号
ACTIVE_NODELIST[size_activenode+1]



- 左子問題
  total_node_number+2
- 分枝変数
  var_branching[total_node_number+2]=i
  fix_binary[total_node_number+2]=0

- 右子問題
  total_node_number+1
- 分枝変数
  var_branching[total_node_number+1]=i
  fix_binary[total_node_number+2]=1

演習問題

- 分枝停止となる場合、分枝停止理由をファイルに表示させる
  - 整数解が得られた
  - 実行不可能
  - 暫定値よりよい解がない（限定操作）
- 施設配置問題への適用
- （発展的課題：奥行優先以外の子問題選択規則（幅優先）は難しい…）

結果ファイル（.sol）：出力はカスタマイズできる

```
反復1
**********
Total number of subproblems 1
Current Problem 1
ACTIVE_NODELIST[1]=1
size_activenode = 1.000000

solve_result = solved

Profit = 13.000000

:    l        u        x       :=
1   0.000000   1.000000   1.000000
2   0.000000   1.000000   0.750000
3   0.000000   1.000000   0.000000
;

INTEGER CONSTRAINT VIOLATED
Subproblem 2 and 3 generated
TEMPORALLY objective value       -
     10000000000.000000
```

```
反復2
**********
Total number of subproblems 3
Current Problem 3
ACTIVE_NODELIST[1]=2
ACTIVE_NODELIST[2]=3
size_activenode = 2.000000

solve_result = solved

Profit = 10.000000

:    l        u        x       :=
1   0.000000   1.000000   1.000000
2   0.000000   0.000000   0.000000
3   0.000000   1.000000   1.000000
;

BRANCHING stopped: integer solution obtained
TEMPORALLY objective value       -10000000000.000000
TEMPORALLY objective value refined 10.000000
```

分枝停止理由をファイルに表示させる

- 実行不可能

```
if (solve_result="infeasible") then
 {
    printf: "BRANCHING stopped: infeasibility ¥n" >bb-knapsack.sol;
 }
```

- 整数解が得られた

```
if ((forall {i in 1..Nitem} x[i]*(1-x[i]) < epsilon)
   and (solve_result="solved")) then
{
  printf: "BRANCHING stopped: integer solution obtained ¥n" >bb-knapsack.sol;
 }
```

- 暫定値よりよい解がない（限定操作）

```
 if ( (Profit <= tmpprofit)
    and (solve_result="solved")) then
{
    printf: "BRANCHING stopped: bounding procedure ¥n" >bb-knapsack.sol;
}
```