OptSeq II によるスケジューリング入門

OptSeq II

Python 言語からの呼び出し方法

LOG OPT Co., Ltd.

ご注意

D

目 次

OptSeq（オプトシーク）は，スケジューリング問題に特化した最適化ソルバーである．スケジューリング問題は，通常の混合整数最適化ソルバーが苦手とするタイプの問題であり，実務における複雑な条件が付加されたスケジューリング問題に対しては，専用の解法が必要となる．OptSeq は，スケジューリング問題に特化した**メタヒューリスティクス** (metaheuristics) を用いることによって，大規模な問題に対しても短時間で良好な解を探索することができるように設計されている．

OptSeq のダウンロードならびに詳細については，`http://www.logopt.com/OptSeq/OptSeq.htm` を参照されたい．

ここでは，スケジューリング最適化ソルバー OptSeq を，Python 言語から直接よび出して求解するためのモジュール `optseq` の使用法について解説する．このモジュールは，すべて Python で書かれたクラスで構成されており，ソースコードも `http://www.logopt.com/optseq.htm` からダウンロードでき，ユーザーが書き換え可能である．

以下の構成は次の通り．

1 節では，OptSeq で対象とする資源制約付きスケジューリング問題について述べる．

2 節では，OptSeq に内在する諸クラスについて解説する．

3 節では，最適化の動作をコントロールするためのパラメータについて述べる．

4 節では，種々の例を用いて OptSeq の使用法を具体的に解説する．

1　資源制約付きスケジューリング問題

ここでは，OptSeq で対象とする資源制約付きスケジューリング問題について簡単に解説する．

行うべき仕事（ジョブ，作業，タスク）を**活動** (activity ; 作業) とよぶ． 専門用語としては「活動」が好みであるが，以下では，実際問題を意識して「作業」とよぶことにする．スケジューリング問題の目的は作業をどのようにして時間軸上に並べて遂行するかを決めることであるが，ここで対象とする問題では作業を処理するための方法が何通りかあって，そのうち 1 つを選択することによって処理するものとする．このような作業の処理方法を**モード** (mode) とよぶ．納期や納期遅れのペナルティ（重み）は作業ごとに定めるが，作業時間や資源の使用量はモードごとに決めることができる（4.3 節）．

作業を遂行するためには**資源** (resource) を必要とする．資源の使用可能量は時刻ごとに変化しても良いものとする．また，モードごとに定める資源の使用量も作業開始からの経過時間によって変化しても良いものとする．通常，資源は作業完了後には再び使用可能になるものと仮定するが，お金や原材料のように一度使用するとなくなってしまうものも考えられる．そのような資源を**再生不能資源** (nonrenewable resource) とよぶ（4.7 節）．

作業間に定義される**時間制約** (time constraint) は，ある作業（先行作業）の処理が終了するまで，別の作業（後続作業）の処理が開始できないことを表す先行制約を一般化したものであり，先行作業の開始（完了）時刻と後続作業の開始（完了）時刻の間に以下の制約があることを規定する（4.8 節）．

$$\text{先行作業の開始（完了）時刻} + \text{時間ずれ} \leq \text{後続作業の開始（完了）時刻}$$

ここで，時間ずれは任意の整数値であり負の値も許すものとする．この制約によって，作業の同時開始，最早開始時刻，時間枠などの様々な条件を記述することができる．

OptSeq では，モードを作業時間分の小作業の列と考え，処理の途中中断や並列実行も可能であるとする．その際，中断中の資源使用量や並列作業中の資源使用量も別途定義できるものとする（4.9 節，4.10 節）．また，時刻によって変化させることができる**状態** (state) が準備され，モード開始の状態の制限やモードによる状態の推移を定義できる（4.11 節）．

2 OptSeq のクラス

OptSeq の Python モジュール (`optseq`) における基本クラスには，以下のものがある．

- モデル `Model`
- 作業 `Activity`（作業はしばしば「仕事」,「活動」,「ジョブ」,「タスク」ともよばれる.）
- モード `Mode`
- 資源 `Resource`
- 時間制約 `Temporal`
- 状態 `State`

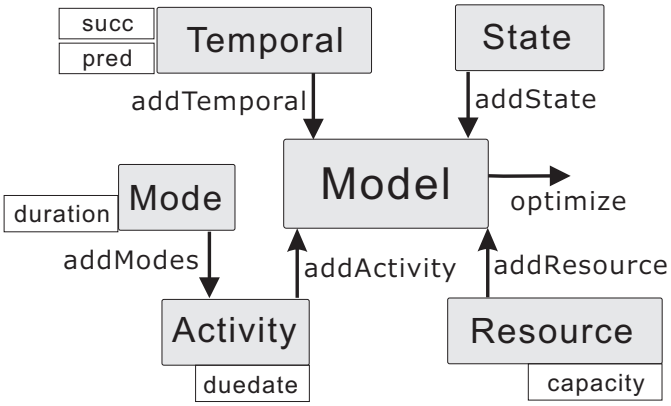


図 1: OptSeq におけるクラス間の関係と主要なメソッド・属性

図 1 にクラス間の関係と主要なメソッド・属性を示す．本節では上の諸クラスについて解説を行う．

注意：
OptSeq では作業，モード，資源名を文字列で区別するため重複した名前を付けることはできない．なお，使用できる文字列は, 英文字 (`a`–`z`, `A`–`Z`), 数字 (`0`–`9`), 大括弧 (`[ ]`), アンダーバー (_), および `@` に限定される．また，作業名は `source`,`sink` 以外，モードは `dummy` 以外の文字に限定される．

2.1 モデル

Python から OptSeq をよび出して使うときに，最初にすべきことは

```
from optseq import *
```

と optseq モジュールを読み込むことであるが，続いてすべきことはモデルクラスのインスタンス model を生成することである．OptSeq では引数なしで（もしくは名前を引数として），以下のように記述する．

```
model = Model()
model = Model('名前')
```

モデルインスタンス model は，以下のメソッドをもつ．

addActivity は，モデルに 1 つの作業を追加する．返値は作業インスタンスである．

```
作業インスタンス = model.addActivity(name, duedate='inf', weight=1, autoselect=False)
```

引数の名前と意味は以下の通り．

name は作業の名前を文字列で与える．ただし作業の名前に'source', 'sink' を用いることはできない．

duedate は作業の納期を 0 以上の整数もしくは，無限大を表す文字列'inf' で与える．この引数は省略可で，既定値は'inf' である．

weight は作業の完了時刻が納期を遅れたときの単位時間あたりのペナルティである．省略可で，既定値は 1.

autoselect は作業に含まれるモードを自動選択するか否かを表すフラグである．モードを自動選択するとき True，それ以外のとき False を設定する．既定値は False．状態によってモードの開始が制限されている場合には，autoselect を True に設定しておくことが望ましい．

addResource はモデルに資源を 1 つ追加する．返値は資源インスタンスである．

```
資源インスタンス = model.addResource(name, capacity, rhs=0, direction='<=', weight='inf')
```

引数の名前と意味は以下の通り．

name は資源の名前を文字列で与える．

capacity は資源の容量（使用可能量の上限）を辞書もしくは正数値で与える．正数値で与えた場合には，開始時刻は 0，終了時刻は無限大と設定される．辞書のキーはタプル (開始時刻 , 終了時刻) であり，値は容量を表す正数値である．開始時刻と終了時刻の組を**区間** (interval) とよぶ．離散的な時間を考えた場合には，時刻 $t-1$ から時刻 t の区間を**期** (period) t と定義する．時刻の初期値を 0 と仮定すると，期は 1 から始まる整数値をとる．区間 (開始時刻 , 終了時刻) に対応する期は,「開始時刻 +1，開始時刻 +2，...，終了時刻」となる（図 2）．

rhs は再生不能資源制約の右辺定数を与える．省略可で，既定値は 0.

direction は再生不能資源制約の種類（制約が等式か不等式か，不等式の場合には方向）を示す文字列を与える．文字列は'<=', '>=', '=' のいずれかとする．省略可であり，既定値は '<=' である．

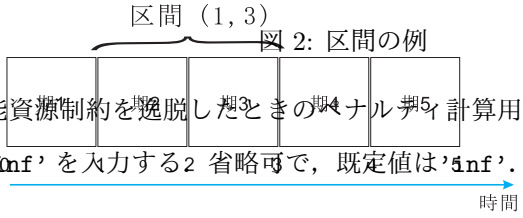


図 2: 区間の例

`weight` は 再生不能資源制約を逸脱したときのペナルティ計算用の重みを与える．正数値もしくは無限大を表す文字列`'inf'` を入力する．省略可で，既定値は`'inf'`.

`addTemporal` はモデルに時間制約を 1 つ追加する．返値は時間制約インスタンスである．

```
時間制約インスタンス = model.addTemporal(pred, succ, tempType='CS', delay=0)
```

時間制約は，先行作業と後続作業の開始（もしくは完了）時刻間の関係を表し，以下のように記述される．

$$\text{先行作業の開始（完了）時刻} + \text{時間ずれ} \leq \text{後続作業の開始（完了）時刻}$$

ここで**時間ずれ** (delay) は時間の差を表す整数値である．先行（後続）作業の開始時刻か完了時刻のいずれを対象とするかは，時間制約のタイプで指定する．タイプは，**開始時刻** (start time) のとき文字列`'S'`，**完了時刻** (completion time) のとき文字列`'C'` で表し，先行作業と後続作業のタイプを 2 つつなげて`'SS'`, `'SC'`, `'CS'`, `'CC'` のいずれかから選択する．引数の名前と意味は以下の通り．

`pred` は**先行作業** (predecessor) のインスタンスもしくは文字列`'source'` を与える．文字列`'source'` は，すべての作業に先行する開始時刻 0 のダミー作業を定義するときに用いる．

`succ` は**後続作業** (successor) のインスタンスもしくは文字列`'sink'` を与える．文字列`'sink'` は，すべての作業に後続するダミー作業を定義するときに用いる．

`tempType` は時間制約のタイプを与える．`'SS'`, `'SC'`, `'CS'`, `'CC'` のいずれかから選択し，省略した場合の既定値は`'CS'`（先行作業の完了時刻と後続作業の開始時刻）である．

`delay` は先行作業と後続作業の間の時間ずれであり，整数値（負の値も許すことに注意）で与える．既定値は 0 である．

`addState` はモデルに状態を追加する．引数は状態の名称を表す文字列 `name` であり，返値は状態インスタンスである．

```
状態インスタンス = model.addState(name)
```

`optimize` はモデルの最適化を行う．返値はなし．最適化を行った結果は，作業，モード，資源，時間制約インスタンスの属性に保管される．

`write` は最適化されたスケジュールを簡易 **Gantt チャート** (Gantt chart) [1] としてテキストファイルに出力する．引数はファイル名 (`filename`) であり，その既定値は `optseq.txt` である．ここで出力される Gantt チャー

[1] Henry Gantt によって 100 年くらい前に提案されたスケジューリングの表記図式なので，Gantt の図式という名前がついている．実は，最初の発案者はポーランド人の Karol Adamiecki で 1896 年まで遡る．

トは，作業別に選択されたモードや開始・終了時刻を示したものであり，資源に対しては使用量と容量が示される．

writeExcel は最適化されたスケジュールを簡易 Gantt チャートとしてカンマ区切りのテキスト (csv) ファイルに出力する．引数はファイル名 (filename) とスケールを表す正整数 (scale) である．ファイル名の既定値は optseq.csv である．スケールは，時間軸を scale 分の 1 に縮めて出力するためのパラメータであり，Excel の列数が上限値をもつために導入された．その既定値は 1 である．なお，Excel 用の Gantt チャートでは，資源の残り容量のみを表示する．

モデルインスタンスは，モデルの情報を文字列として返すことができる．たとえば，モデルインスタンス model の情報は，

```
print(model)
```

で得ることができる．作業，モード，資源，時間制約，状態のインスタンスについても同様であり，print 関数で情報を出力することができる．

モデル（作業，モード，資源，時間制約，状態）の情報は，インスタンスの属性に保管されている．インスタンスの属性は「インスタンス. 属性名」でアクセスできる．

モデルインスタンスは以下の属性をもつ．

activities はモデルに含まれる作業名をキー，作業インスタンスを値とした辞書である．

modes はモデルに含まれるモード名をキー，モードインスタンスを値とした辞書である．

resources はモデルに含まれる資源名をキー，資源インスタンスを値とした辞書である．

temporals はモデルに含まれる時間制約の先行作業名と後続作業名のタプルをキー，時間制約インスタンスを値とした辞書である．

Params は最適化をコントロールするためのパラメータインスタンスである．パラメータについては，3 節で詳述する．

2.2 作業クラス

成すべき仕事（ジョブ，活動，タスク）を総称して**作業** (activity) とよぶ．作業クラスのインスタンスは，モデルインスタンス model の作業追加メソッド (addActivity) の返値として生成される．

```
作業インスタンス=model.addActivity(name, duedate='inf', weight=1)
```

上のメソッドの引数については，2.1 節を参照．

作業には任意の数のモード（作業の実行方法）を追加することができる．モードの追加は，以下のメソッドで行う．

addModes は作業にモードを追加する．引数は（任意の数の）モードインスタンス．

```
作業インスタンス.addModes(モードインスタンス1, モードインスタンス2, ... )
```

作業の情報は，作業インスタンスの属性に保管されている．作業インスタンスは以下の属性をもつ．

name は作業名である．

duedate は作業の納期であり，0 以上の整数もしく無限大'inf' を入力する．

weight は作業の完了時刻が納期を遅れたときの単位時間あたりのペナルティである．

modes は作業に付随するモードインスタンスのリストを保持する．

selected は探索によって発見された解において選択されたモードインスタンスを保持する．

start は探索によって発見された解における作業の開始時刻である．

completion は探索によって発見された解における作業の終了時刻である．

execute は探索によって発見された解における作業の実行を表す辞書である．キーは作業の開始時刻と終了時刻のタプル，値は並列実行数を表す正数値である．

たとえば，作業インスタンス act の納期を 10，重みを 2 に変更したい場合には，納期を表す属性が duedate，重みを表す属性が weight であるので，以下のようにすれば良い．

```
act.duedate=10
act.weight=2
```

2.3 モードクラス

作業の処理方法を**モード** (mode) とよぶ． 作業は少なくとも 1 つのモードをもち，そのうちのいずれかを選択して処理される．

モードのインスタンスは，モードクラス Mode から生成される．

```
モードインスタンス = Mode(name, duration=0)
```

引数の名前と意味は以下の通り．

name はモードの名前を文字列で与える．ただしモードの名前に'dummy' を用いることはできない．

duration はモードの作業時間を非負の整数で与える．既定値は 0.

モードインスタンスは，以下のメソッドをもつ．

addResource はモードを実行するときに必要な資源とその量を指定する．

```
モードインスタンス.addResource(resource,requirement={},rtype=None)
```

引数と意味は以下の通り．

resource は追加する資源インスタンスを与える．

requirement は資源の必要量を辞書もしくは正数値で与える．辞書のキーはタプル (開始時刻 , 終了時刻) であり，値は資源の使用量を表す正数値である．正数値で与えた場合には，開始時刻は 0，終了時刻は無限大と設定される．

注： 作業時間が 0 のモードに資源を追加することはできない．その場合には実行不可能解と判断される．

rtype は資源のタイプを表す文字列．'break', 'max' のいずれかから選択する（既定値は None）．'break' を与えた場合には，中断中に使用する資源量を指定する．'max' を与えた場合には，並列処理中に使用する資源の「最大量」を指定する．省略可で，その場合には通常の資源使用量を規定し，並列処理中には資源使用量の「総和」を使用することになる．

addBreak は中断追加メソッドである．モードは単位時間ごとに分解された作業時間分の小作業の列と考えられる．小作業を途中で中断してしばらく時間をおいてから次の小作業を開始することを**中断** (break) とよぶ．中断追加メソッド (addBreak) は，モード実行時における中断の情報を指定する．

```
モードインスタンス.addBreak(start=0, finish=0, maxtime='inf')
```

引数と意味は以下の通り．

start は中断可能な最早時刻を与える．省略可で，既定値は 0.

finish は中断可能時刻の最遅時刻を与える．省略可で，既定値は 0.

maxtime は最大中断可能時間を与える．省略可で，既定値は無限大（'inf'）.

addParallel は並列追加メソッドである．モードは単位時間ごとに分解された作業時間分の小作業の列と考えられる．資源量に余裕があるなら，同じ時刻に複数の小作業を実行することを**並列実行** (parallel execution) とよぶ． 並列追加メソッド (addParallel) は，モード実行時における並列実行に関する情報を指定する．

```
モードインスタンス.addParallel(start=1, finish=1, maxparallel='inf')
```

引数と意味は以下の通り．

start は並列実行可能な最小の小作業番号を与える．省略可で，既定値は 1.

finish は並列実行可能な最大の小作業番号を与える．省略可で，既定値は 1.

maxparallel は同時に並列実行可能な最大数を与える．省略可で，既定値は無限大 ('inf').

addState は状態追加メソッドである．状態追加メソッド (addState) は，モード実行時における状態の値と実行直後（実行開始が時刻 t のときには，時刻 $t+1$）の状態の値を定義する．

```
モードインスタンス.addState(state , fromValue=0, toValue=0)
```

引数と意味は以下の通り

state はモードに付随する状態インスタンスを与える．省略不可.

fromValue はモード実行時における状態の値を与える．省略可で，既定値は 0.

toValue はモード実行直後における状態の値を与える．省略可で，既定値は 0.

モードインスタンスは以下の属性をもつ．

name はモード名である．

duration はモードの作業時間である．

requirement ははモードの実行の資源･資源タイプと必要量を表す辞書である．キーは資源名と資源タイプ（None: 通常，'break':中断中，'max':並列作業中の最大資源量）のタプルであり，値は資源必要量を表す辞書である．この辞書のキーはタプル（開始時刻，終了時刻）であり，値は資源の使用量を表す正数値である．

breakable は中断の情報を表す辞書である．辞書のキーはタプル（開始時刻，終了時刻）であり，値は中断可能時間を表す正数値である．

parallel は並列作業の情報を表す辞書である．辞書のキーはタプル (開始小作業番号 , 終了小作業番号) であり，値は最大並列可能数を表す正数値である．

autoselect はモードを自動選択するとき True，それ以外のとき False を設定する．既定値は False である．

2.4 資源クラス

資源インスタンスは，モデルの資源追加メソッド (addResource) の返値として生成される．

```
資源インスタンス = model.addResource(name, capacity, rhs=0, direction='<=', weight='inf')
```

上のメソッドの引数については，2.1 節を参照されたい．

資源インスタンスは，以下のメソッドをもつ．

addCapacity は資源に容量を追加するためのメソッドであり，資源の容量を追加する．引数と意味は以下の通り．

start は資源容量追加の開始時刻（区間の始まり）を与える．

finish は資源容量追加の終了時刻（区間の終わり）を与える．

amount は追加する資源量を与える．

setRhs(rhs) は再生不能資源を表す線形制約の右辺定数を rhs に設定する．引数は整数値（負の値も許すことに注意）とする．

setDirection(dir) は再生不能資源を表す制約の種類を dir に設定する．引数の dir は'<=', '>=', '=' のいずれかとする．

addTerms(coeffs,vars,values) は，再生不能資源制約の左辺に 1 つ，もしくは複数の項を追加するメソッドである．作業がモードで実行されるときに 1，それ以外のとき 0 となる変数（値変数）を x[作業, モード] とすると，追加される項は，

$$\text{係数} \times x[\text{作業}, \text{モード}]$$

と記述される．addTerms メソッドの引数は以下の通り．

coeffs は追加する項の係数もしくは係数リスト．係数もしくは係数リストの要素は整数（負の値も許す）．

vars は追加する項の作業インスタンスもしくは作業インスタンスのリスト．リストの場合には，リスト coeffs と同じ長さをもつ必要がある．

values は追加する項のモードインスタンスもしくはモードインスタンスのリスト．リストの場合には，リスト coeffs と同じ長さをもつ必要がある．

資源インスタンスは以下の属性をもつ．

name は資源名である．

capacity は資源の容量（使用可能量の上限）を表す辞書である．辞書のキーはタプル (開始時刻 , 終了時刻) であり，値は容量を表す正数値である．

rhs は再生不能資源制約の右辺定数である．

direction は再生不能資源制約の方向を表す．

terms は再生不能資源制約の左辺を表す項のリストである．各項は (係数 , 作業インスタンス , モードインスタンス) のタプルである．

2.5 時間制約クラス

時間制約インスタンスは，モデルに含まれる形で生成される．時間制約インスタンスは，上述したモデルの時間制約追加メソッド (addTemporal) の返値として生成される．

```
時間制約インスタンス = model.addTemporal(pred, succ, tempType='CS', delay=0)
```

上のメソッドの引数については，2.1 節を参照されたい．

時間制約インスタンスは以下の属性をもつ．

pred は先行作業のインスタンスである．

succ は後続作業のインスタンスである．

type は時間制約のタイプを表す文字列であり，'SS'（開始，開始）,'SC'（開始，完了）,'CS'（完了，開始）,'CC'（完了，完了）のいずれかを指定する．

delay は時間制約の時間ずれを表す整数値である．

2.6 状態クラス

状態インスタンスは，モデルに含まれる形で生成される．状態インスタンスは，上述したモデルの状態追加メソッド (addState) の返値として生成される．

```
状態インスタンス = model.addState(name)
```

状態インスタンスは，指定時に状態の値を変化させるためのメソッド addValue をもつ．

addValue(time, value) は，状態を時刻 time（非負整数値）に値 value（非負整数値）に変化させることを指定する．

状態インスタンスは以下の属性をもつ．

name は状態の名称を表す文字列である．

value は，時刻をキーとし，その時刻に変化する値を値とした辞書である．

3 パラメータ

OptSeq に内在されている最適化ソルバーの動作は，**パラメータ** (parameter) を変更することによってコントロールできる．モデルインスタンス model のパラメータを変更するときは，以下の書式で行う．

```
model.Params.パラメータ名 = 値
```

たとえば，計算時間の上限 (TimeLimit) を 1 秒に変更したい場合には，

```
model.Params.TimeLimit=1
```

とする．

以下に代表的なパラメータとその意味を記す．

RandomSeed は乱数系列の種を設定する．既定値は 1.

Makespan は最大完了時刻（一番遅く終わる作業の完了時刻）を最小にするとき True，それ以外のとき（各作業に定義された納期遅れの重み付き和を最小にするとき）False を設定する．既定値は False.

TimeLimit は最大計算時間 (秒) を設定する．既定値は 600 秒.

Initial は，前回最適化の探索を行った際の最良解を初期値とした探索を行うとき True，それ以外のとき False を表すパラメータである．既定値は False である．最良解の情報は作業の順序と選択されたモードとしてファイル名 optseq_best_act_data.txt に保管されている．このファイルを書き換えることによって，異なる初期解から探索を行うことも可能である．

OutputFlag は計算の途中結果を出力させるためのフラグである．True のとき出力 On，False のとき出力 Off. 既定値は False.

4 例題

ここでは，幾つかの例題を通して OptSeq の使用法を解説する．本節の構成は以下の通り．

4.1 節では，プロジェクトスケジューリングの古典モデルである PERT(Program Evaluation and Review Technique) を例として，基本的な時間制約の使い方を学ぶ．

4.2 節では，資源制約付きの PERT の例題を通して，資源のモデル化方法を説明する．

4.3 節では，並列ショップスケジューリングとよばれる問題のモデル化を解説する．

4.4 節では，作業のモードの概念を説明する．

4.5 節では，一般の資源制約付きスケジューリング問題のモデル化について解説する．

4.6 節では，納期遅れの最小化について解説する．

4.7 節では，再生可能資源（通常の機械や人のような資源）と再生不能資源（お金や原料のような資源）の違いを解説し，再生不能資源を表現する方法を説明する．

4.8 節では，時間制約と，それを用いた種々の作業間のタイミングの設定法について解説する．

4.9 節では，作業を途中で中断することを許す場合の記述法について説明する．

4.10 節では，作業の並列処理を簡単に記述する方法について解説する．

4.11 節では，状態変数の使用法について述べる．

4.12 節では，複雑なジョブショップスケジューリングのモデル化について解説する．

4.1 PERT

あなたは航空機会社のコンサルタントだ．あなたの仕事は，着陸した航空機をなるべく早く離陸させるためのスケジュールをたてることだ．航空機は，再び離陸する前に幾つかの作業をこなさなければならない．まず，乗客と荷物を降ろし，次に機内の掃除をし，最後に新しい乗客を搭乗させ，新しい荷物を積み込む．当然のことであるが，乗客を降ろす前に掃除はできず，掃除をした後でないと新しい乗客を入れることはできず，荷物をすべて降ろし終わった後でないと，新しい荷物は積み込むことができない．また，この航空機会社では，乗客用のゲートの都合で，荷物を降ろし終わった後でないと新しい乗客を搭乗させることができないのだ．作業時間は，乗客降ろし 13 分，荷物降ろし 25 分，機内清掃 15 分，新しい乗客の搭乗 27 分，新しい荷物の積み込み 22 分とする．さて，最短で何分で離陸できるだろうか？

これは，PERT(Program Evaluation and Review Technique) とよばれる，スケジューリング理論の始祖とも言える古典的なモデルである．ちなみに，PERT は，第 2 次世界大戦中における米国海軍のポラリス潜水艦に搭載するミサイルの設計・開発時間の短縮に貢献したことで有名になり，その後オペレーションズ・リサーチの技法の代表格となった．

PERT は OptSeq を用いて容易に解くことができる．まず，OptSeq のモジュールを読み込み，モデルインスタンス `model` を作成する．

```
from optseq import *
model=Model()
```

次に，モデルに作業を追加する．そのために，作業時間を表すデータを，作業をキー，作業時間を値とする辞書 duration として準備する．次に，作業を保管するための空の辞書 act とモードを保管するための空の辞書 mode を作成しておく．

```
duration ={1:13, 2:25, 3:15, 4:27, 5:22 }
act={}
mode={}
```

次に，モデルインスタンス model の addActivity メソッドを用いてモデルに作業を追加する．また，各作業にはモードを定義する必要があるので，モードクラス Mode を用いてモードに必要な作業時間を定義した後で，addModes メソッドを用いて作業にモードを追加する．

```
for i in duration:
    act[i]=model.addActivity('Act[{0}]'.format(i))
    mode[i]=Mode('Mode[{0}]'.format(i), duration[i])
    act[i].addModes(mode[i])
```

次に，モデルに addTemporal メソッドを用いて時間制約を追加する．ここでは，ある作業が完了した後でないと，他の作業が開始できないことを表す通常の先行制約なので，時間制約のタイプは CS(Completion-Start) となる．これは既定値なので，以下のように省略して入力できる．

```
model.addTemporal(act[1],act[3])
model.addTemporal(act[2],act[4])
model.addTemporal(act[2],act[5])
model.addTemporal(act[3],act[4])
```

最後に，optimize メソッドを用いて求解する．求解の前にモデルのパラメータを変更しておく．パラメータはモデルの Params 属性で変更する．TimeLimit は求解するときの制限時間なので，それを 3 秒に設定する．Makespan は最大完了時刻最小化のときは True，それ以外のときは False を表すパラメータなので，それを True に設定する．また，結果の詳細を見たい場合には，OutputFlag を True に設定しておく．

```
model.Params.TimeLimit  = 3
model.Params.Makespan   = True
model.Params.OutputFlag = True
model.optimize()
```

上のコードの実行結果は以下のようになる（出力には説明を加えてある）．

```
# reading data ... done: 0.00(s) データを読むためにかかった時間は 0 秒
# random seed: 1                 乱数の初期値は 1 と設定
# tabu tenure: 1                 タブーサーチのパラメータであるタブー期間の初期値は 1
# cpu time limit: 3.00(s)        計算時間上限は 3 秒
# iteration limit: 1073741823    反復回数の上限
# computing all-pairs longest paths and strongly connected components ... done
#scc 7
objective value = 55 (cpu time = 0.00(s), iteration = 0) 目的関数値 55 が CPU 時間 0 秒反復 0 回で求まった
0: 0.00(s): 55/55
--- best activity list ---
source activity[2] activity[5] activity[1] activity[3] activity[4] sink 最適な作業の投入順序

--- best solution ---          最良解
source ---: 0 0                ダミーの始点の開始時刻が 0
sink ---: 55 55                ダミーの終点の開始時刻が 55（完了時刻）
activity[1] ---: 0 0--13 13    作業 1 は 0 に開始されて 13 に終了
activity[2] ---: 0 0--25 25    作業 2 は 0 に開始されて 25 に終了
activity[3] ---: 13 13--28 28 作業 3 は 13 に開始されて 28 に終了
activity[4] ---: 28 28--55 55 作業 4 は 28 に開始されて 55 に終了
activity[5] ---: 25 25--47 47 作業 5 は 25 に開始されて 47 に終了
```

```
objective value = 55     目的関数値は 55
cpu time = 0.00/3.00(s) 計算時間
iteration = 1/62605      反復回数
```

これから，最後の作業が完了する時刻（離陸の時間）が 55 分後であり，そのための各作業の開始時間が，それぞれ $0, 0, 13, 28, 25$ 分後であることが分かる．

4.2 資源制約付き PERT

> あなたは航空機会社のコンサルタントだ．リストラのため作業員の大幅な削減を迫られたあなたは，前節の例題と同じ問題を 1 人の作業員で行うためのスケジュールを作成しなければならなくなった．作業時間や時間制約は，前節と同じであるとするが，各々の作業は作業員を 1 人占有する（すなわち，2 つの作業を同時にできない）ものとする．どのような順序で作業を行えば，最短で離陸できるだろうか？

この問題は資源制約付きプロジェクトスケジューリング問題になるので，$\mathcal{NP}$-困難であるが，OptSeq を用いればやはり容易に解くことができる．

上で作成したプログラムに資源制約を追加する．資源（この例題の場合は作業員）`res` の使用可能量の上限（容量）は，モデルの `addResource` メソッドを用いて追加する．引数は，資源名を表す文字列と使用可能量上限を表す `capacity` である．

```
res=model.addResource('worker',capacity=1)
```

使用可能量上限は，開始時刻と終了時刻を指定して{(開始時刻, 終了時刻):供給量}と入力することもできる．

```
res=model.addResource('worker',capacity={(0,'inf'):1})
```

ここで `inf` は無限大 (infinity) を表し，OptSeq では非常に大きな数を表すキーワードである．

モードに資源の使用量を追加するときには，モードの `addResource` メソッドを用いて追加する．第 1 引数は資源を表すインスタンスであり，この場合は上で定義した `res` である．第 2 引数 `requirement` には，使用量を表す正数，もしくは{(開始時刻, 終了時刻):使用量}の形式の辞書を入力する．ここでは,(開始時刻, 終了時刻）を表す区間を，各作業の作業時間（この例題の場合，作業 1 は 0 から 13 ，作業 2 は 0 から 25 など）と書かずに，`0,'inf'` と書くことにする．これはモードに作業時間を入力してあるため，この作業時間外では資源が使われないからである．資源使用量が時間によらず一定の場合には，単に資源使用量 1 を指定して，`addResource(res, 1)` と書いても良い．

```
act={}
mode={}
for i in duration:
    act[i]=model.addActivity('Act[{0}]'.format(i))
    mode[i]=Mode('Mode[{0}]'.format(i),duration[i])
    mode[i].addResource(res,{(0,'inf'):1})
    act[i].addModes(mode[i])
```

上のように変更したプログラムを実行すると以下の結果を得る．

```
--- best activity list ---
source activity[2] activity[5] activity[1] activity[3] activity[4] sink
```

```
--- best solution ---
source ---: 0 0
sink ---: 102 102
activity[1] ---: 47 47--60 60
activity[2] ---: 0 0--25 25
activity[3] ---: 60 60--75 75
activity[4] ---: 75 75--102 102
activity[5] ---: 25 25--47 47

objective value = 102
cpu time = 0.00/3.00(s)
iteration = 0/64983
```

この結果は，最後の作業が完了する時刻（離陸の時間）が 102 分後であり，そのための各作業の開始時間が，それぞれ $47, 0, 60, 575, 25$ 分後であることを表している．

4.3 並列ショップスケジューリング

あなたは F1 のピットクルーだ．F1 レースにとってピットインの時間は貴重であり，ピットインしたレーシングカーに適切な作業を迅速に行い，なるべく早くレースに戻してやることが，あなたの使命である．

作業 1： 給油準備 (3 秒)

作業 2： 飲料水の取り替え　(2 秒)

作業 3： フロントガラス拭き (2 秒)

作業 4： ジャッキで車を持ち上げ (2 秒)

作業 5： タイヤ (前輪左側) 交換 (4 秒)

作業 6： タイヤ (前輪右側) 交換 (4 秒)

作業 7： タイヤ (後輪左側) 交換 (4 秒)

作業 8： タイヤ (後輪右側) 交換 (4 秒)

作業 9： 給油 (11 秒)

作業 10： ジャッキ降ろし (2 秒)

各作業には，作業時間のほかに，この作業が終わらないと次の作業ができないといったような時間制約がある．作業時間と時間制約は，図 3 のようになっている．
いま，あなたを含めて 3 人のピットクルーがいて，これらの作業を手分けして行うものとする．作業は途中で中断できないものとすると，なるべく早く最後の作業を完了させるには，誰がどの作業をどういう順番で行えばよいのだろうか？

この問題は**並列ショップ** (parallel shop) スケジューリングとよばれる問題であり，$\mathcal{NP}$-困難である．

まず，モデルインスタンスを作成し，データを辞書として準備しておく．

```
from optseq import *
model=Model()
duration ={1:3, 2:2, 3:2, 4:2, 5:4, 6:4, 7:4, 8:4, 9:11, 10:2 }
```

資源は 3 単位使えるものと定義し，作業時間と資源の使用量を以下のように定義する．

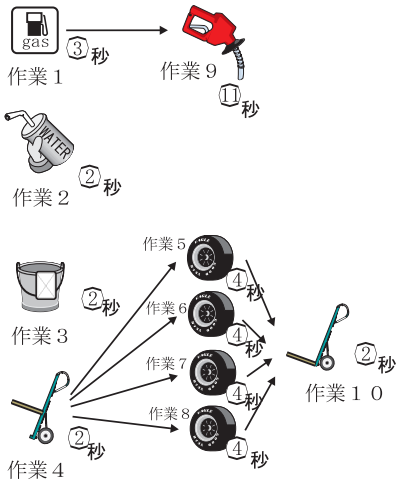


図 3: ピットクルーの作業の作業時間と時間制約

```
res=model.addResource('worker',capacity={(0,'inf'):3})
act={}
mode={}
for i in duration:
    act[i]=model.addActivity('Act[{0}]'.format(i))
    mode[i]=Mode('Mode[{0}]'.format(i),duration[i])
    mode[i].addResource(res,{(0,'inf'):1})
    act[i].addModes(mode[i])
```

次に，作業間の時間制約を定義する．

```
model.addTemporal(act[1],act[9])
for i in range(5,9):
    model.addTemporal(act[4],act[i])
    model.addTemporal(act[i],act[10])
```

最後に，パラメータを設定して求解する．また，ここでは Gantt チャートをテキストファイルに書き出す．

```
model.Params.TimeLimit=1
model.Params.Makespan=True
model.optimize()
model.write('chart.txt')
```

プログラムを実行すると以下の結果を得る（以下では簡易出力を行い，作業開始・終了時刻だけを出力する）．

```
source --- 0 0
sink --- 14 14
Act[1] --- 0 3
Act[2] --- 0 2
Act[3] --- 0 2
Act[4] --- 2 4
Act[5] --- 8 12
Act[6] --- 4 8
Act[7] --- 8 12
Act[8] --- 4 8
Act[9] --- 3 14
Act[10] --- 12 14
```

ファイルに出力された Gantt チャートを，図 4 に示す．

```
  activity    mode   duration | 1| 2| 3| 4| 5| 6| 7| 8| 9|10|11|12|13|14|
-------------------------------------------------------------------------|
  Act[1]   Mode[1]      3     |==|==|==|  |  |  |  |  |  |  |  |  |  |  |
  Act[2]   Mode[2]      2     |==|==|  |  |  |  |  |  |  |  |  |  |  |  |
  Act[3]   Mode[3]      2     |==|==|  |  |  |  |  |  |  |  |  |  |  |  |
  Act[4]   Mode[4]      2     |  |  |==|==|  |  |  |  |  |  |  |  |  |  |
  Act[5]   Mode[5]      4     |  |  |  |  |  |  |  |  |==|==|==|==|  |  |
  Act[6]   Mode[6]      4     |  |  |  |  |==|==|==|==|  |  |  |  |  |  |
  Act[7]   Mode[7]      4     |  |  |  |  |  |  |  |  |==|==|==|==|  |  |
  Act[8]   Mode[8]      4     |  |  |  |  |==|==|==|==|  |  |  |  |  |  |
  Act[9]   Mode[9]      11    |  |  |  |==|==|==|==|==|==|==|==|==|==|==|
 Act[10]   Mode[10]     2     |  |  |  |  |  |  |  |  |  |  |  |  |==|==|
-------------------------------------------------------------------------
   resource usage/capacity    |
-------------------------------------------------------------------------
            worker            | 3| 3| 2| 2| 3| 3| 3| 3| 3| 3| 3| 2| 2|
                              | 3| 3| 3| 3| 3| 3| 3| 3| 3| 3| 3| 3| 3|
-------------------------------------------------------------------------
```

図 4: 並列ショップスケジューリングの Gantt チャート（==は作業を処理中を表す.）

4.4 並列ショップスケジューリング 2–モードの概念と使用法–

ここでは，前節の例題の拡張を「モード」の概念を用いて解いてみる．

いま，前節で扱った 3 人の作業員が,「給油準備作業」を協力して作業を行い，時間短縮ができる場合を考える．1 人でやれば 3 秒かかる作業が，2 人でやれば 2 秒，3 人がかりなら 1 秒で終わるものとする．これは，作業に 3 つのモードをもたせ，それぞれ作業時間と使用資源量を以下のように設定することによって表現できる．

モード 1： 作業時間 3 秒，人資源 1 人

モード 2： 作業時間 2 秒，人資源 2 人

モード 3： 作業時間 1 秒，人資源 3 人

作業とモードに関するコードを以下のように変更することによって，拡張された並列ショップスケジューリング問題を解くことができる．

```
for i in duration:
    act[i]=model.addActivity('Act[{0}]'.format(i))
    if i==1:
        mode[1,1]=Mode('Mode[1_1]',3)
        mode[1,1].addResource(res,{(0,'inf'):1})
        mode[1,2]=Mode('Mode[1_2]',2)
        mode[1,2].addResource(res,{(0,'inf'):2})
        mode[1,3]=Mode('Mode[1_3]',1)
        mode[1,3].addResource(res,{(0,'inf'):3})
        act[i].addModes(mode[1,1],mode[1,2],mode[1,3])
    else:
        mode[i]=Mode('Mode[{0}]'.format(i),duration[i])
        mode[i].addResource(res,{(0,'inf'):1})
        act[i].addModes(mode[i])
```

結果は以下のようになり，作業 1 はモード 3 で実行され，全体として 1 秒短縮されることが分かる（図 5）．

```
source --- 0 0
sink --- 13 13
Act[1] Mode[1_3] 0 1
Act[2] --- 1 3
```

```
  activity    mode   duration | 1| 2| 3| 4| 5| 6| 7| 8| 9|10|11|12|13|
-------------------------------------------------------------------------|
  Act[1]  Mode[1_3]     1     |==|  |  |  |  |  |  |  |  |  |  |  |  |
  Act[2]   Mode[2]      2     |  |==|==|  |  |  |  |  |  |  |  |  |  |
  Act[3]   Mode[3]      2     |  |  |  |  |  |  |  |  |  |  |  |==|==|
  Act[4]   Mode[4]      2     |  |==|==|  |  |  |  |  |  |  |  |  |  |
  Act[5]   Mode[5]      4     |  |  |  |  |  |  |  |==|==|==|==|  |  |
  Act[6]   Mode[6]      4     |  |  |  |==|==|==|==|  |  |  |  |  |  |
  Act[7]   Mode[7]      4     |  |  |  |  |  |  |  |==|==|==|==|  |  |
  Act[8]   Mode[8]      4     |  |  |  |==|==|==|==|  |  |  |  |  |  |
  Act[9]   Mode[9]      11    |  |==|==|==|==|==|==|==|==|==|==|==|  |
 Act[10]   Mode[10]     2     |  |  |  |  |  |  |  |  |  |  |  |==|==|
-------------------------------------------------------------------------
   resource usage/capacity    |
-------------------------------------------------------------------------
            worker            | 3| 3| 3| 3| 3| 3| 3| 3| 3| 3| 3| 3| 2|
                              | 3| 3| 3| 3| 3| 3| 3| 3| 3| 3| 3| 3| 3|
-------------------------------------------------------------------------
```

図 5: 拡張された並列ショップスケジューリングの Gantt チャート

```
Act[3] --- 11 13
Act[4] --- 1 3
Act[5] --- 7 11
Act[6] --- 3 7
Act[7] --- 7 11
Act[8] --- 3 7
Act[9] --- 1 12
Act[10] --- 11 13
```

4.5 資源制約付きスケジューリング

あなたは 1 階建てのお家を造ろうとしている大工さんだ．あなたの仕事は，なるべく早くお家を完成させることだ．お家を造るためには，幾つかの作業をこなさなければならない．まず，土台を造り，1 階の壁を組み立て，屋根を取り付け，さらに 1 階の内装をしなければならない．ただし，土台を造る終える前に 1 階の建設は開始できず，内装工事も開始できない．また，1 階の壁を作り終える前に屋根の取り付けは開始できない．
各作業とそれを行うのに必要な時間（単位は日）は，以下のようになっている．

土台： 2 人の作業員で 1 日

1 階の壁： 最初の 1 日目は 2 人，その後の 2 日間は 1 人で，合計 3 日

内装： 1 人の作業員で 2 日

屋根： 最初の 1 日は 1 人，次の 1 日は 2 人の作業員が必要で，合計 2 日

いま，作業をする人は，あなたをあわせて 2 人いるが，相棒の 1 人は作業開始 3 日目に休暇をとっている．さて，最短で何日でお家を造ることができるだろうか？

この問題を解くためにはデータをどのように保持するかが鍵となる．ここでは辞書を用いた方法を示す．資源使用量は，空の辞書 `req` を作り（ 1 行目），各作業の作業時間と資源の使用量を辞書 `req` に保管しておく．例えば， 2 行目の `req[1]={(0,1):2}`は，作業 1 が，時刻 0 から時刻 1 （ 1 日目）に資源を（作業員） 2 単位使うことを表す．

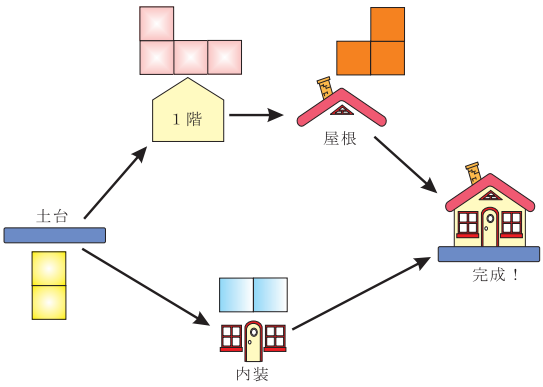


図 6: お家を造るための作業，制約，必要な人数

```
req={}
req[1]={(0,1):2 }
req[2]={(0,1):2 ,(1,3):1}
req[3]={(0,2):1 }
req[4]={(0,1):1,(1,2):2 }
```

また，使用できる資源量も日によって変わるので，以下のように入力する．まず，モデルの `addResource` メソッドを用いて，資源名を`'worker'` とする資源 `res` を作る．次に，資源の `addCapacity` メソッドを用いて資源制約を入力する．例えば， 2 行目の `res.addCapacity(0,2,2)` は，時刻 0 から時刻 2 （ 1 日目と 2 日目）に資源が 2 単位使えることを表す．

```
res=model.addResource('worker')
res.addCapacity(0,2,2)
res.addCapacity(2,3,1)
res.addCapacity(3,'inf',2)
```

作業，モードならびに時間制約は，今までと同じように設定すれば良い．

```
act={}
mode={}
for i in duration:
    act[i]=model.addActivity('Act[{0}]'.format(i))
    mode[i]=Mode('Mode[{0}]'.format(i),duration[i])
    mode[i].addResource(res,req[i])
    act[i].addModes(mode[i])
model.addTemporal(act[1],act[2])
model.addTemporal(act[1],act[3])
model.addTemporal(act[2],act[4])
```

上のプログラムを実行すると，以下の結果を得る（図 7）．

```
source --- 0 0
sink --- 6 6
Act[1] --- 0 1
Act[2] --- 1 4
Act[3] --- 3 5
Act[4] --- 4 6
```

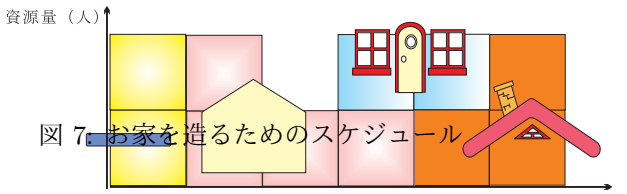


図 7: お家を造るためのスケジュール

4.6 納期遅れ最小化スケジューリング

あなたは売れっ子連載作家だ．あなたは，A, B, C, D の 4 社から原稿を依頼されており，それぞれ，どんなに急いで書いても 1 日，2 日，3 日，4 日かかるものと思われる．各社に約束した納期は，それぞれ 5 日後，9 日後，6 日後，4 日後であり，納期から 1 日遅れるごとに 1 万円の遅延ペナルティを払わなければならない．

会社名	A	B	C	D
作業時間（日）	1	2	3	4
納期（日後）	5	9	6	4

どのような順番で原稿を書けば，支払うペナルティ料の合計を最小にできるだろうか？

まず，各作業の納期（作業時間）を，作業番号をキー，納期（作業時間）を値とした辞書 due(duration) として準備しておく．

```
due={1:5,2:9,3:6,4:4}
duration={1:1, 2:2, 3:3, 4:4 }
```

また，作業を表す資源 res を準備しておく．

```
res=model.addResource('writer')
res.addCapacity(0,'inf',1)
```

次に，納期と納期遅れのペナルティを作業に追加する．納期と納期遅れのペナルティは，モデルに作業を追加するとき，addActivity(’作業名’,duedate=納期，weight=ペナルティ) のように入力する．また，ペナルティの既定値は 1 なので，この例題の場合，ペナルティの入力を省略して記述している．

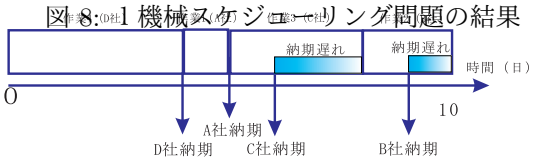
```
1 act={}
2 mode={}
3 for i in duration:
4     act[i]=model.addActivity('Act[{0}]'.format(i),duedate=due[i])
5     mode[i]=Mode('Mode[{0}]'.format(i),duration[i])
6     mode[i].addResource(res,{(0,'inf'):1})
7     act[i].addModes(mode[i])
```

この例題の場合，納期遅れ最小化を目的とするため，モデルのパラメータ Makespan を False に設定する必要があるが，パラメータ Makespan の既定値は False であるので省略してもかまわない．

プログラムを実行すると，以下の結果を得る（図 8）．

```
source --- 0 0
sink --- 10 10
Act[1] --- 4 5
Act[2] --- 8 10
Act[3] --- 5 8
Act[4] --- 0 4
```

図 8:1 機械スケジューリング問題の結果



4.7 CPM

あなたは，航空機会社のコンサルタントだ．今度は，作業時間の短縮を要求されている．(ただし，資源制約（人の制限）はないものとする．）いま，航空機の離陸の前にする作業の時間が，費用をかけることによって短縮でき，各作業の標準時間，新設備導入によって短縮したときの時間，ならびにそのときに必要な費用は，以下のように推定されているものとする．

作業 1： 乗客降ろし 13 分．10 分に短縮可能で，1 万円必要．

作業 2： 荷物降ろし 25 分．20 分に短縮可能で，1 万円必要．

作業 3： 機内清掃 15 分．10 分に短縮可能で， 1 万円必要．

作業 4： 新しい乗客の搭乗 27 分．25 分に短縮可能で， 1 万円必要．

作業 5： 新しい荷物積み込み 22 分．20 分に短縮可能で， 1 万円必要．

さて，いくら費用をかけると，どのくらい離陸時刻を短縮することができるだろうか？

これは，**クリティカルパス法** (Critical Path Method; CPM) とよばれる古典的な問題である．CPM は，作業時間を費用（お金）をかけることによって短縮できるという仮定のもとで，費用と作業完了時刻のトレードオフ曲線を求めることを目的とした PERT の変形で，資源制約がないときには効率的な解法が古くから知られているが，資源制約がつくと困難な問題になる．ここでは，この問題が「モード」と「再生不能資源」を用いて，OptSeq でモデル化できることを示す．

作業は通常の作業時間と短縮時の作業時間をもつが，これは作業に付随するモードで表現することができる．問題となるのは，作業時間を短縮したときには，費用がかかるという部分である．費用は資源の一種と考えられるが，いままで考えていた資源とは異なる種類の資源である．いままで考えていた資源は，機械や人のように，作業時間中は使用されるが，作業が終了すると，再び別の作業で使うことができるようになる．このような，作業完了後に再び使用可能になる資源を，**再生可能資源** (renewable resource) とよぶ．一方，費用（お金）や原材料のように，一度使うとなくなってしまう資源を，**再生不能資源** (nonrenewable resource) とよぶ．

CPM の例題に対して，再生不能資源（お金）の上限を色々変えて最短時間を求める．まず，各々の作業に対して，通常の作業時間をもつ場合と，短縮された作業時間をもつ場合の 2 つのモードを追加し，さらに短縮モードを用いた場合には，再生不能資源を 1 単位使用するという条件を付加する．

まず，2 種類の作業時間を種類別に辞書に保管しておく．`durationA` が通常の作業時間，`durationB` が再生不能資源を使う場合の作業時間である．

```
durationA = {1:13, 2:25, 3:15, 4:27, 5:22 }
durationB = {1:10, 2:20, 3:10, 4:25, 5:20 }
```

再生不能資源も，モデルの addResource メソッドを用いて追加する．まず，(’資源名’，rhs=右辺，direction=制約の向き，weight=ペナルティ) を引数として再生不能資源インスタンス res を生成する．

```
res=model.addResource('money',rhs=4,direction='<=', weight='inf')
```

ちなみに，再生不能資源に対しては，制約を逸脱したときのペナルティ(重み）を無限大’inf’ 以外にも設定可能である．制約の逸脱を許す場合には，重みを正数値として入力する．この場合では，制約の逸脱は絶対に許さないことを指定しているので，逸脱をしない解がない場合には，実行不可能 (infeasible) という結果を返す．

作業とモードの追加方法は，前節までの例題と同じである（1 から 5 行目）．次に，addTerms(資源使用量，作業，モード) を用いて，再生不能資源制約の左辺に関する入力を行う（6 行目）．

```
1 for i in durationA:
2     act[i]=model.addActivity('Act[{0}]'.format(i))
3     mode[i,1]=Mode('Mode[{0}][1]'.format(i),durationA[i])
4     mode[i,2]=Mode('Mode[{0}][2]'.format(i),durationB[i])
5     act[i].addModes(mode[i,1],mode[i,2])
6     res.addTerms(1,act[i],mode[i,2])
```

再生不能資源の上限を 4, 1, 0 と変えてプログラムを実行すると，図 9 の結果を得る．

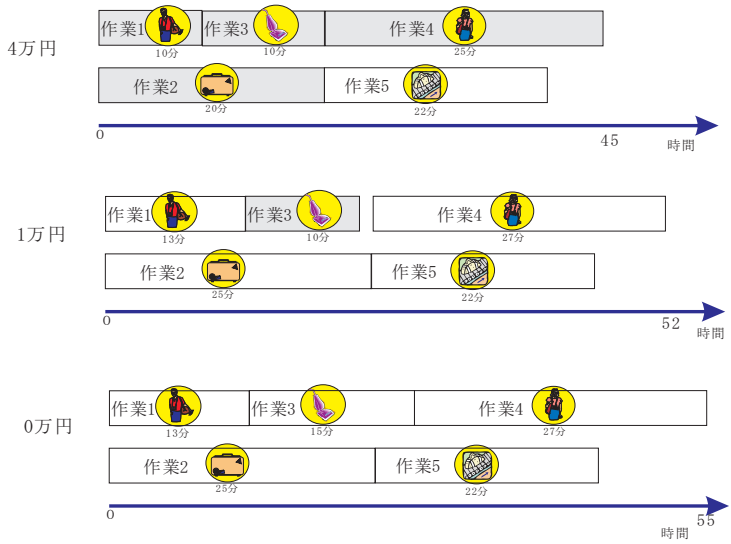


図 9: 再生不能資源の上限が 4, 1, 0 のときのスケジュール（色のついた矩形で表された作業は，短縮モードで実施されている．）

4.8 時間制約

OptSeq では，通常の先行制約の他に，様々な時間に関する制約が準備されている．時間制約を用いることによって，実際問題で発生する様々な付加条件をモデル化することができる．

時間制約の適用例として，4.1 節 の PERT の例題において，作業 3 と作業 5 の開始時刻が一致しなければならないという制約を考えてみる．

開始時刻を一致させるためには，制約タイプは SS（Start-Start の関係）とし，時間ずれは 0 と設定する．また，制約は「作業 3 の開始時刻 $\leq$ 作業 5 の開始時刻」と「作業 5 の開始時刻 $\leq$ 作業 3 の開始時刻」の 2 本を追加する．

```
model.addTemporal(act[3],act[5],'SS',0)
model.addTemporal(act[5],act[3],'SS',0)
```

このような制約を付加して求解すると，以下のような結果が得られる．

```
source --- 0 0
sink --- 67 67
Act[1] --- 0 13
Act[2] --- 0 25
Act[3] --- 25 40
Act[4] --- 40 67
Act[5] --- 25 47
```

確かに，作業 3 と作業 5 の作業開始時刻が一致していることが確認できる．

また，作業の開始時間の固定も，この時間制約を用いると簡単にできる．OptSeq では，すべての作業に先行するダミーの作業'source' が準備されている．この作業は必ず時刻 0 に処理されるので，開始時刻に相当する時間ずれをもつ時間制約を 2 本追加することによって，開始時刻を固定することができる．

たとえば，作業 5（名前は act[5]）の開始時刻を 50 分に固定したい場合には，

```
model.addTemporal('source', act[5],'SS',delay=50)
model.addTemporal(act[5], 'source','SS',delay=-50)
```

と時間制約を追加する．これは，作業 5 の開始時刻が'source' の開始時刻（0）の 50 分後以降であることと，'source' の開始時刻が作業 5 の開始時刻の −50 分後以降（言い換えれば，作業 5 の開始時刻が 50 分後以前）であることを規定する．

4.9 作業の途中中断

多くの実際問題では，緊急の作業などが入ってくると，いま行っている作業を途中で中断して，別の（緊急で行わなければならない）作業を行った後に，再び中断していた作業を途中から行うことがある．このように，途中で作業を中断しても，再び（一から作業をやり直すのではなく）途中から作業を続行することを「作業の途中中断」とよぶ．

OptSeq では，これを作業を分割して処理することによって表現する．たとえば，作業時間が 3 時間の作業があったとする．時間の基本単位を 1 時間としたとき，この作業は，1 時間の作業時間をもつ 3 つの小作業に分割して処理される．

しかし，実際問題では，中断可能なタイミングが限られている場合もある．たとえば，料理をするときに，材料を切ったり，混ぜたりするときには，途中で中断することも可能だが，いったんオーブンレンジに入れたら，途中でとめたりすることはできない．OptSeq では，作業（モード）の時間の区間に対して，最大中断可能時間を入力することによって，様々な作業の**中断** (break) を表現する．

例として，4.6 節の納期遅れ最小化問題において，すべての作業がいつでも最大 1 日だけ中断できる場合を考える．作業の途中中断は，addBreak(区間の開始時刻, 区間の終了時刻, 最大中断時間) を用いて追加する（4 行目）．

```
1 for i in duration:
2     act[i]=model.addActivity('Act[{0}]'.format(i),duedate=due[i])
3     mode[i]=Mode('Mode[{0}]'.format(i),duration[i])
4     mode[i].addBreak(0,'inf',1)
5     act[i].addModes(mode[i])
```

```
  activity    mode   duration | 1| 2| 3| 4| 5| 6| 7| 8| 9|10|11|12|13|
-------------------------------------------------------------------------|
  Act[1]   Mode[1]      1     |  |  |  |  |..|==|  |  |  |  |  |  |  |
  Act[2]   Mode[2]      2     |  |  |  |  |  |  |..|==|==|  |  |  |  |
  Act[3]   Mode[3]      3     |  |  |  |  |  |  |  |  |..|==|..|==|==|
  Act[4]   Mode[4]      4     |==|==|==|..|==|  |  |  |  |  |  |  |  |
-------------------------------------------------------------------------
   resource usage/capacity    |
-------------------------------------------------------------------------
            writer            | 1| 1| 1| 0| 1| 1| 0| 1| 1| 1| 0| 1| 1|
                              | 1| 1| 1| 0| 1| 1| 0| 1| 1| 1| 0| 1| 1|
-------------------------------------------------------------------------
```

図 10: 中断を許した納期遅れ最小化問題の Gantt チャート（==は作業を処理中を，. . は中断中を表す.）

また，段取りを伴う生産現場においては，中断の途中で他の作業を行うことが禁止されている場合がある．これは，休日の間に異なる作業を行うと，再び段取りなどの処理を行う必要があるため，作業を一からやり直さなければならないからである．これは，作業の中断中でも資源を使い続けていると表現することによって回避することができる．

中断中に資源を使用する場合も，通常の資源を追加するのと同様に addResource メソッドを用いて追加する．この場合，引数として (資源,(区間):資源使用量,'break') を入力する．たとえば，すべての作業が中断中も資源を 1 単位使用することを表すには，mode[i].addResource(res,(0,'inf'):1,'break') と入力する．

例題の資源を表す「作家」が 4 日目と 7 日目と 11 日目に休暇を入れたときの納期遅れ最小化問題を解くための資源データは，以下のように定義される．

```
res=model.addResource('writer')
res.addCapacity(0,3,1)
res.addCapacity(4,6,1)
res.addCapacity(7,10,1)
res.addCapacity(11,'inf',1)
```

このデータを用いて中断可能な問題を解いたときの Gantt チャートの出力結果は，図 10 のようになり，作業 3 と作業 4 が中断されて処理されていることが確認できる．

4.10 作業の並列処理

4.4 節で解説した並列ショップスケジューリング問題の拡張では，複数の機械（作業員）によって作業時間が短縮されることを，複数のモードを用いることによって表現していた．ここでは，複数資源による作業の並列処理を，より簡単に表現するための方法を紹介する．

前節の作業の途中中断と同じように，作業を，単位時間の作業時間をもつ小作業に分解して考える．いま，資源使用量の上限が 1 より大きいとき，分解された小作業は，並列して処理できるものとする．ただし，無制限に並列処理ができない場合も多々あるので，OptSeq では，最大並列数とよばれるパラメータを用いて表現する．

並列処理は，作業モードに対する addParallel メソッドを用いて定義される．書式は，addParallel(開始小作業番号, 終了小作業番号, 最大並列数) である．

たとえば，

```
mode.addParallel(1,1,3)
```

```
mode.addParallel(2,3,2)
```

は, 最初の小作業は最大 3 個，2 番目, 3 番目の小作業は最大 2 個の小作業を並列処理可能であることを意味する. 並列処理は小作業を表す矩形の上に，点線の矩形を入れて表す（図 11）. 点線の矩形の数だけ，通常の作業の上に重ねて処理できることを表し，図 11 の例では，図右に示してある作業の組み合わせの中で，最も良いものが選択される.

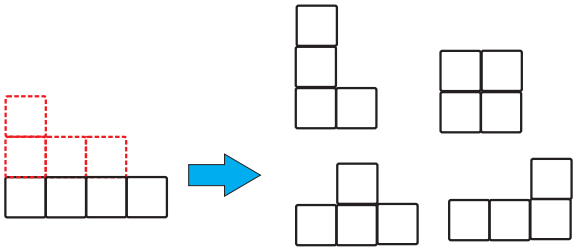
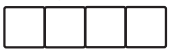


図 11: 並列作業の設定例（作業時間は 4）



4.4 節の並列ショップスケジューリング問題において，給油作業（作業時間 3 秒）を，最初の（1 番目の）小作業を最大 3 個並列可能とした場合の作業モードの定義は以下のようになる.

```
act={}
mode={}
for i in duration:
    act[i]=model.addActivity('Act[{0}]'.format(i))
    mode[i]=Mode('Mode[{0}]'.format(i),duration[i])
    mode[i].addResource(res,{(0,'inf'):1})
    if i==1:
        mode[i].addParallel(1,1,3)
    act[i].addModes(mode[i])
```

計算結果は以下のようになり，1 秒短縮して 13 秒で作業が終了することが確認できる.

```
--- best activity list ---
source jackup tire1 prepare water oil tire2 tire4 tire3 front jackdown sink

--- best solution ---
source ---: 0 0
sink ---: 13 13
prepare ---: 0 0--1[2] 1--2 2
water ---: 1 1--3 3
front ---: 10 10--12 12
jackup ---: 0 0--2 2
tire1 ---: 2 2--6 6
tire2 ---: 3 3--7 7
tire3 ---: 7 7--11 11
tire4 ---: 6 6--10 10
oil ---: 2 2--13 13
jackdown ---: 11 11--13 13

objective value = 13
cpu time = 0.00/3.00(s)
iteration = 25/23174
```

ここで，並列で行われた作業（名称は prepare）においては，並列数が [ ] で表示される. 0 0--1[2] 1--2 2

は、「時刻 0 に処理を開始し，時刻 0〜1 は並列数 2 で処理し，その後，時刻 1〜2 に並列なしで処理し，時刻 2 で完了」を表す．

4.11 状態変数

状態変数とは，時間の経過とともに状態とよばれる非負整数の値が変化する変数である．作業のモードが，特定の状態でないと開始できないという制約を付加することによって，通常のスケジューリングモデルでは表現しきれない，様々な付加条件をモデル化することが可能になる．

まず，状態を定義するには，モデルクラスの addState メソッドを用いる．addState メソッドの引数は，状態の名称を表す文字列であり，返値は状態インスタンスである．たとえば，無記名の状態 state をモデルインスタンス model に追加するには，以下のようにする．

```
state = model.addState()
```

状態は，基本的には 1 つ前の時刻の値を引き継ぎ，ユーザーの指定によって特定の時刻にその値を変化させることができる変数である．状態が，ある時間においてある値に変化することを定義するためには，addValue メソッドを用いる．たとえば，状態インスタンス state が時刻 0 に値 1 になることを定義するには，次のように記述する．

```
state.addValue( time=0, value=1 )
```

状態は，モードで開始された直後に変化させることができる．そのためには，モードインスタンスの addState メソッドを用いて定義する．addState メソッドの引数は，状態インスタンス (state)，開始時の状態（fromValue; 非負整数），開始直後に変化する状態（toValue; 非負変数）である．

```
モードインスタンス.addState(state, fromValue, toValue)
```

これによって，モード開始時の状態 state が値 fromValue でなくてはならず，開始直後（開始時刻が t であれば $t+1$ に）状態が値 toValue に変化することになる．これによって，処理モードに「ある状態のときしか開始できない」といった制約を加えることが可能になる．この記述は，状態のとる値を変えて，任意の回数行うことができる．たとえば，状態変数 state の取り得る値が $1,2,3$ の 3 種類としたとき，

```
mode.addState( state, 1, 2)
mode.addState( state, 2, 3)
mode.addState( state, 3, 1)
```

とすれば，開始時点での状態に関係なく処理は可能で，状態は巡回するように 1 つずつ変化することになる．

また，これを利用して「あるタイプのモード（以下の mode）は 1 週間に高々3 回しか処理できない」ことは，以下のように表すことができる．

```
state = model.addState()
state.addValue( time=0,  value=0 )
state.addValue( time=7,  value=0 )
state.addValue( time=14, value=0 )
    ...
mode = Mode('mode')
mode.addState( state, 0, 1)
mode.addState( state, 1, 2)
mode.addState( state, 2, 3)
```

状態は 7 日ごとに 0 にリセットされ，モード mode は状態 state が 0,1,2 のときだけ開始でき，状態が 3 のときには開始できない．したがって 7 日の間に 3 回だけ開始できることになる．

上述のような「開始時状態が指定された処理モード」を与える場合，通常，そのモードを含む作業の定義において，モードを自動選択 (autoselect) にしておくことが推奨される．OptSeq では,「まず各作業の処理モードをそれぞれ選び，その後，ある順序にしたがって作業を前づめにスケジュールしていく」ということの繰り返しを行う．したがって,「開始時状態が指定された処理モード」が存在すると，処理モードの選び方によっては,「スケジュールを生成していくとき，割り付け不可能」ということが頻繁に起こりえる．これを防ぐため,「あらかじめ処理モードを指定せず，前づめスケジュールしながら適切な処理モードを選択する」ことが必要になり，これを実現するのがモードの自動選択なのである．

作業に対してモードを自動選択に設定するには，作業クラスのコンストラクタの引数 autoselect を用いるか，作業インスタンスの属性 autoselect を用いる．具体的には，以下の何れの書式でも，自動選択に設定できる．

```
act1=model.addActivity('Act1', autoselect=True)
act1.autoselect = True
```

注意: 作業の定義に autoselect を指定した場合には，その作業に制約を逸脱したときの重みを無限大とした（すなわち絶対制約とした）再生不能資源を定義することはできない．かならず重みを既定値の無限大'inf' ではない正数値と設定し直す必要がある．

状態の実務的な利用例として，順序依存の段取り作業を考えよう．例題の仮定は以下の通り．

- 資源は機械 1 台
- 作業は A, B の 2 つのタイプがあり，それぞれ 3 個ずつの計 6 個．処理時間はすべて 3 時間
- タイプの異なる作業間の段取り時間は 5 時間，同タイプであれば段取り時間 1 時間
- メイクスパン（最大完了時刻）最小化が目的

これを実現するためには,

- 段取り用の状態変数を定義．値 1 は A タイプ作業の処理直後に，値 2 は B タイプ作業の処理直後に対応
- 各作業に対して，段取り作業を定義
- A タイプの作業の段取り作業には，AtoA，BtoA の 2 つのモードを設定
- B タイプの作業の段取り作業には，AtoB，BtoB の 2 つのモードを設定
- 段取り作業と本作業の間に他の作業が割り込まないように,「段取り作業の完了時刻 = 本作業の開始時刻」となるよう時間制約を追加し，さらに，本作業は開始直後に中断可能（中断中も資源を消費）となるよう記述

とすれば良い．

まず，モデルインスタンス model を準備し，作業時間と段取り時間を辞書に保管しておく．duration[i,j] は作業 i（1 は A，2 は B を表す）の j 番目の作業時間，setup は作業タイプ間の段取り時間を表す．

```
model=Model()
duration={(1,1):3,(1,2):3,(1,3):3,(2,1):3,(2,2):3,(2,3):3}
setup    ={(1,1):1,(1,2):5,(2,1):5,(2,2):1}
```

次に，作業，段取り作業，モード，段取りモードを表す辞書を準備し（1 行目)，機械を表す資源インスタンス machine の使用可能量上限を 1 に設定し（2 行目），状態変数 state を定義し，その初期状態を 1（タイプ A の作業の段取り状態）に設定する（3,4 行目）.

```
1 act, act_setup, mode, mode_setup={},{},{},{}
2 machine=model.addResource('machine',1)
3 state=model.addState('Setup_State')
4 state.addValue(time=0,value=1)
```

段取り替えを表すモードは，状態を i から j へ変化させ，機械を 1 単位占有することを定義しておく.

```
for (i,j) in setup:
    mode_setup[i,j]=Mode('Mode_setup[{0}_{1}]'.format(i,j),setup[i,j])
    mode_setup[i,j].addState(state,i,j)
    mode_setup[i,j].addResource(machine,{(0,'inf'):1})
```

続いて，段取り作業と本作業を定義する．2,3 行目では，段取り作業を定義する．これは，タイプ A の作業からの段取り mode_setup[1,i] か，タイプ B の作業からの段取り mode_setup[2,i] のいずれかであるので，その両者を段取り作業 act_setup[i,j] に追加する．また，この作業は「開始時状態が指定された処理モード」であるので，autoselect 引数を True に設定しておく．4 行以降では本作業を定義する．本作業 act[i,j] の作業モード mode[i,j] は，開始直後に中断可能であり（7 行目），その間も機械を占有する（8 行目）.

```
1 for (i,j) in duration:
2     act_setup[i,j]  = model.addActivity('Setup[{0}_{1}]'.format(i,j), autoselect=True)
3     act_setup[i,j].addModes( mode_setup[1,i], mode_setup[2,i] )
4     act[i,j]  = model.addActivity('Act[{0}_{1}]'.format(i,j))
5     mode[i,j] = Mode('Mode[{0}_{1}]'.format(i,j),duration[i,j])
6     mode[i,j].addResource(machine,{(0,'inf'):1})
7     mode[i,j].addBreak(0,0)
8     mode[i,j].addResource(machine,{(0,'inf'):1},'break')
9     act[i,j].addModes(mode[i,j])
```

最後に時間制約を追加する．段取り作業が完了した瞬間に本作業が始まるように,「段取り作業の終了時刻 $\leq$ 本作業の開始時刻」と「本作業の開始時刻 $\leq$ 段取り作業の終了時刻」の 2 本の制約を定義する.

```
for (i,j) in duration:
    model.addTemporal(act_setup[i,j], act[i,j],'CS')
    model.addTemporal(act[i,j], act_setup[i,j],'SC')
```

上のモデルを完了時刻最小化で最適化すると以下の結果を得る.

```
source --- 0 0
sink --- 28 28
Setup[1_2] Mode_setup[1_1] 4 5
Act[1_2] --- 5 8
Setup[1_3] Mode_setup[1_1] 0 1
Act[1_3] --- 1 4
Setup[2_1] Mode_setup[2_2] 20 21
Act[2_1] --- 21 24
Setup[2_3] Mode_setup[2_2] 24 25
Act[2_3] --- 25 28
Setup[2_2] Mode_setup[1_2] 12 17
Act[2_2] --- 17 20
Setup[1_1] Mode_setup[1_1] 8 9
Act[1_1] --- 9 12
```

表 1: 4 作業 3 機械スケジューリング問題のデータ

	作業 1	作業 2	作業 3
作業 1	機械 1 / 7 日	機械 2 / 10 日	機械 3 / 4 日
作業 2	機械 3 / 9 日	機械 1 / 5 日	機械 2 / 11 日
作業 3	機械 1 / 3 日	機械 3 / 9 日	機械 2 / 12 日
作業 4	機械 2 / 6 日	機械 3 / 13 日	機械 1 / 9 日

タイプ A の作業を 3 個連続で行い，その後にタイプ B の作業を行っていることが確認できる．

4.12 ジョブショップスケジューリング

例として，4 作業 3 機械のスケジューリング問題を考える．各作業はそれぞれ 3 つの子作業（これを以下では作業とよぶ）1, 2, 3 から成り, この順序で処理しなくてはならない．各作業を処理する機械, および処理日数は，表 1 の通りである．

このように，作業によって作業を行う機械の順番が異なる問題は，**ジョブショップ** (job shop) とよばれ，スケジューリングモデルの中でも難しい問題と考えられている．

目的は最大完了時刻最小化とする．ここでは，さらに以下のような複雑な条件がついているものと仮定する．

1. 各作業の初めの 2 日間は作業員資源を必要とする操作がある．この操作は平日のみ, かつ 1 日あたり高々 2 個しか行うことができない．
2. 各作業は，1 日経過した後だけ，中断が可能．
3. 機械 1 での作業は，最初の 1 日は 2 個まで並列処理が可能．
4. 機械 2 に限り, 特急処理が可能．特急処理を行うと処理日数は 4 日で済むが, 全体で 1 度しか行うことはできない．
5. 機械 1 において, 作業 1 を処理した後は作業 2 を処理しなくてはならない.

この問題は，機械および作業員資源を再生可能資源とした 12 作業のスケジューリングモデルとして OptSeq で記述できる．

まず，モデルインスタンス `model` を生成し，機械を表す資源を追加する．このとき，機械資源の容量（使用可能量の上限）を 1 と設定しておく．

```
from optseq import *
model=Model()
machine={}
for j in range(1,4):
    machine[j]=model.addResource('machine[{0}]'.format(j),capacity={(0,'inf'):1})
```

作業員も資源であり，この場合には，1 日あたり高々2 個しか行うことができないので，資源の容量は，平日は 2，休日は 0 名と設定する（ただし最初の日は月曜日と仮定する）．

```
manpower=model.addResource('manpower')
for t in range(9):
    manpower.addCapacity(t*7,t*7+5,2)
```

最後に，特急処理が高々1 回しか行うことができないことを表すために，予算 budget と名付けた再生不能資源を追加し，制約の右辺 rhs を 1 に設定しておく．

```
budget=model.addResource('budget_constraint',rhs=1)
```

次に，作業とモードに関する記述を行う．

まず，表 1 のデータを保管するために，作業の番号と作業の番号のタプルをキー，機械番号と作業時間のタプルを値とした辞書 JobInfo を以下のように準備しておく．

```
JobInfo={ (1,1):(1,7), (1,2):(2,10), (1,3):(3,4),
          (2,1):(3,9), (2,2):(1,5), (2,3):(2,11),
          (3,1):(1,3), (3,2):(3,9), (3,3):(2,12),
          (4,1):(2,6), (4,2):(3,13), (4,3):(1,9)
    }
```

特急処理を行うモード express を準備しておく（これは機械 2 に限定した処理で作業時間は 4 である）．

```
1 express=Mode('Express',duration=4)
2 express.addResource(machine[2],{(0,'inf'):1},'max')
3 express.addResource(manpower,{(0,2):1})
4 express.addBreak(1,1)
```

作業とモードは辞書 act,mode に保管する（1,2 行目）．6 行目は，並列作業中でも 1 単位の機械資源を使用することを表し，7 行目は，作業員が最初の 2 日間だけ必要なことを表し，8 行目は，1 日経過後に 1 日だけ中断が可能なことを表す．また，機械 1 上では並列処理が可能であり（9,10 行目），機械 2 に対しては通常モードと特急モード express を追加し，さらに特急モードで処理した場合には予算資源 budget を 1 単位使用するものとする（11 から 13 行目）．

```
1  act={}
2  mode={}
3  for (i,j) in JobInfo:
4      act[i,j]=model.addActivity('Act[{0}][{1}]'.format(i,j))
5      mode[i,j]=Mode('Mode[{0}][{1}]'.format(i,j),duration=JobInfo[i,j][1])
6      mode[i,j].addResource(machine[JobInfo[i,j][0]],{(0,'inf'):1},'max')
7      mode[i,j].addResource(manpower,{(0,2):1})
8      mode[i,j].addBreak(1,1)
9      if JobInfo[i,j][0]==1:
10         mode[i,j].addParallel(1,1,2)
11     if JobInfo[i,j][0]==2:
12         act[i,j].addModes(mode[i,j],express)
13         budget.addTerms(1,act[i,j],express)
14     else:
15         act[i,j].addModes(mode[i,j])
```

先行制約（作業の前後関係）を表す時間制約を同じ作業に含まれる作業間に設定しておく．

```
for i in range(1,5):
    for j in range(1,3):
        model.addTemporal(act[i,j],act[i,j+1])
```

条件「機械 1 において, 作業 1 を処理した後は作業 2 を処理しなくてはならない」を記述するためには，多少のモデル化のための工夫が必要となる．この制約は，直前先行制約とよばれ，以下のようにしてモデル化を行うことができる．

1. 処理時間 0 のダミーの（仮想の）作業 dummy （以下のモデルファイルでは d_act）を導入し, 時刻 0 で中断可能と設定する. そして, 中断中, 資源「機械 1」を消費し続けるものと定義する.

2. 時間制約を用いて, (作業 1 の完了時刻) = (dummy の開始時刻) および (dummy の完了時刻) = (作業 2 の開始時刻) の 2 つの制約を追加する.

この結果, 作業 1act[1,1] の完了後, 作業 2act[2,2] が開始されるまで機械 1 の資源は消費され続けることになり, 他の作業を行うことはできないことになる.

```
d_act=model.addActivity('dummy_activity')
d_mode=Mode('dummy_mode')
d_mode.addBreak(0,0)
d_mode.addResource(machine[1],{(0,0):1},'break')
d_act.addModes(d_mode)
model.addTemporal(act[1,1],d_act,tempType='CS')
model.addTemporal(d_act,act[1,1],tempType='SC')
model.addTemporal(d_act,act[2,2],tempType='CS')
model.addTemporal(act[2,2],d_act,tempType='SC')
```

最後に，目的である最大完了時刻最小化をパラメータ Makespan で設定し，計算時間上限 1 秒で求解した後で Gantt チャートをファイル chart.txt に出力する.

```
model.Params.TimeLimit=1
model.Params.OutputFlag=True
model.Params.Makespan=True
model.optimize()
model.write('chart.txt')
```

実行したときの出力例を以下に示す. プログラム終了時に, 探索で得られた最良スケジュール（完了時刻は 38）が表示される. たとえば, Act[2][2] の開始時刻は 9 で, まず時刻 9〜10 の間に 2 つの小作業が並列処理された後 (9--10[2]), 残りの小作業が時刻 13 まで行われる.

```
source,---, 0 0
sink,---, 38 38
Act[3][2],---, 23 23--32 32
Act[1][3],---, 32 32--33 35--38 38
Act[2][1],---, 0 0--9 9
Act[2][3],Mode[2][3], 21 21--32 32
Act[4][2],---, 10 10--23 23
Act[1][2],Mode[1][2], 8 8--9 10--19 19
Act[3][3],Express, 32 32--33 35--38 38
Act[3][1],---, 14 14--15[2] 15--16 16
Act[4][3],---, 23 23--32 32
Act[2][2],---, 9 9--10[2] 10--13 13
Act[4][1],Mode[4][1], 2 2--8 8
Act[1][1],---, 0 0--7 7
dummy_activity,---, 7 9
```

最適解を簡易 Gantt チャートで示したもの (chart.txt) は，次ページの図 12 のようになる.

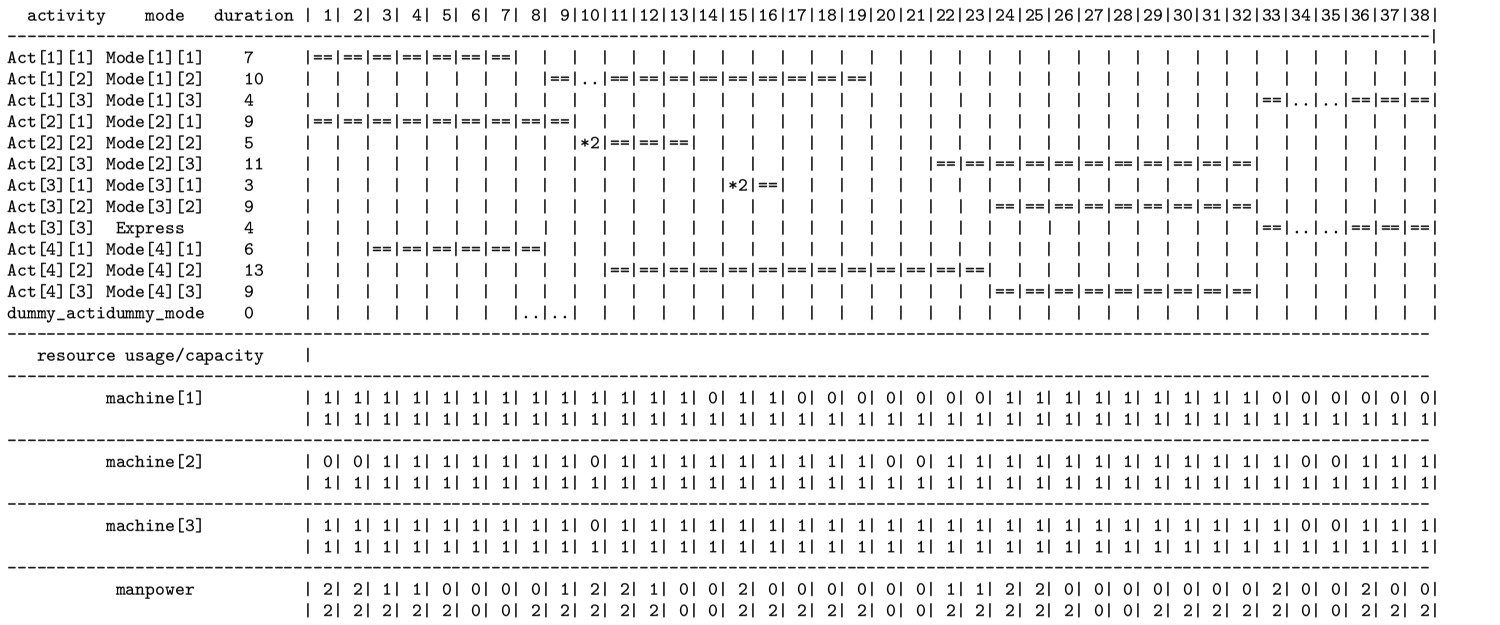


図 12: 例題の Gantt チャート表示（==は作業を処理中，. . は中断中，*2 は並列で処理中を表す.）

問題 1

表 2 のような作業に対して，以下の問いに答えよ．

1. 作業時間の楽観値に対して作業 H が完了する最早時刻を求めよ．
2. 作業時間の平均値に対して作業 H が完了する最早時刻を求めよ．
3. 作業時間の悲観値に対して作業 H が完了する最早時刻を求めよ．
4. 作業時間が楽観値と悲観値の間の一様分布と仮定したときの，作業 H が完了する最早時刻の分布を matplotlib を用いて描け．

表 2: 先行作業と作業時間（楽観値，平均値，悲観値）

作業名	先行作業	楽観値	平均値	悲観値
A	なし	1	2	3
B	なし	2	3	4
C	A	1	2	3
D	B	2	4	6
E	C	1	4	7
F	C	1	2	3
G	D,E	3	4	5
H	F,G	1	2	3

問題 2

4.3 節の並列ショップスケジューリングの例題に対して，以下のような変更を行ったときのスケジュールを求めよ．

1. 作業員 4 人で作業を行うとした場合
2. 作業間の時間制約をなくしたと仮定した場合
3. 作業時間をすべて 1 秒短縮したと仮定した場合

問題 3

4.5 節の資源制約付きスケジューリングの例題に対して，2 階を建てる作業（作業時間は 2 人で 2 日）と，2 階の内装を行う作業（作業時間は 1 人で 2 日）追加した場合のスケジュールを求めよ．ただし，2 階を建てる作業は，1 階の壁を取り付けた後でないと開始できず，屋根の取り付けと 2 階の内装は，2 階を建てた後でないと開始できないものと仮定する．

問題 4

4.7 節での例題で，作業時間と短縮したときの費用が，以下のように設定されている場合を考え，予算が 0 から 10 万円に変化したときの最早完了時刻を求めよ．

作業 1： 乗客降ろし 13 分．12 分に短縮可能で，1 万円必要．11 分に短縮するには，さらに 1 万円必要．

作業 2： 荷物降ろし 25 分．23 分に短縮可能で，1 万円必要．21 分に短縮するには，さらに 1 万円必要．

作業 3： 機内清掃 15 分．13 分に短縮可能で， 1 万円必要．11 分に短縮するには，さらに 1 万円必要．

作業 4： 新しい乗客の搭乗 27 分．26 分に短縮可能で， 1 万円必要．25 分に短縮するには，さらに 1 万円必要．

作業 5： 新しい荷物の積み込み 22 分．21 分に短縮可能で，1 万円必要．20 分に短縮するには，さらに 1 万円必要．

問題 5

あなたは 6 つの異なる得意先から製品の製造を依頼された製造部長だ．製品の製造は特注であり，それぞれ $1, 4, 2, 3, 1, 4$ 日の製造日数がかかる．ただし，製品の製造に必要な材料の到着する日は，それぞれ $, 0, 2, 4, 1, 5$ 日後と決まっている．得意先には上得意とそうでもないものが混在しており，それぞれの重要度は $3, 1, 2, 3, 1, 2$ と推定されている．製品が完成する日数に重みを乗じたものの和をなるべく小さくするように社長に命令されているが，さてどのような順序で製品を製造したら良いのだろう？

問題 6

4.3 節の並列ショップスケジューリングの例題に対して，以下のような変更を行ったときのスケジュールを求めよ．

1. すべての作業が途中中断可能と仮定した場合
2. すべての作業が並列処理可能と仮定した場合

索 引