Artelys
solutions en optimisation

dash optimization

Xpress-Kalis

User guide

Release 2007.2

Last update 26 July, 2007

[これは、皆様のご便宜のための参考翻訳です。
誤訳、不適切な訳があるかもしれませんので、
原文と併読なさるようにしてください。
宜しくお願い致します。
なお、CP グロッサリーは未翻訳です。
どうぞ、英文マニュアルをお読みください。

Published by Dash Optimization Ltd
(c) Copyright Dash Associates and Artelys SA 2007. All rights reserved.

All trademarks referenced in this manual that are not the property of Dash Associates or Artelys SA are acknowledged.

All companies, products, names and data contained within this document are completely fictitious and are used solely to illustrate the use of Xpress-MP and Kalis. Any similarity between these names or data and reality is purely coincidental.

How to Contact Dash

USA, Canada and all Americas

Dash Optimization Inc

Information and Sales: info@dashoptimization.com

Licensing: license-usa@dashoptimization.com

Product Support: support-usa@dashoptimization.com

Tel: +1 (201) 567 9445

Fax: +1 (201) 567 9443

Dash Optimization Inc.

560 Sylvan Avenue

Englewood Cliffs

NJ 07632

USA

Japan

Dash Optimization Japan

Information and Sales:

info@jp.dashoptimization.com

Licensing:

license@jp.dashoptimization.com

Product Support:

support@jp.dashoptimization.com

Tel: +81 43 297 8836

Fax: +81 43 297 8827

WBG Marive-East 21F FASuC B2124

2-6 Nakase Mihama-ku

261-7121 Chiba

Japan

Worldwide

Dash Optimization Ltd

Information and Sales: info@dashoptimization.com

Licensing: license@dashoptimization.com

Product Support: support@dashoptimization.com

Tel: +44 1926 315862

Fax: +44 1926 315854
Leam House, 64 Trinity Street
Leamington Spa
Warwickshire CV32 5YN
UK

For the latest news and Xpress-MP software and documentation updates, please visit the Xpress-MP website at http://www.dashoptimization.com or subscribe to our mailing list.

How to Contact Artelys

Artelys SA
Information and Sales: info-kalis@artelys.com
Licensing and Product Support: support-kalis@artelys.com
Tel: +33 1 44 77 89 00
Fax: +33 1 42 96 22 61
12, rue du Quatre Septembre
75002 Paris Cedex
France

For the latest news about Kalis, training course programs, and examples, please visit the Artelys website at http://www.artelys.com.

第１章 イントロダクション 8
第１章 イントロダクション 8
1.1 Xpress-Kalis 8
1.1.1 製品バージョンについてのノート 9
1.2 ソフトウェアのインストレーション 9
1.3 制約計画法（Constraint Programming）の基本概念 9
1.4 この文書の内容 10
第２章 モデル作成の基礎 12
2.1 最初のモデル 12
2.1.1 Mosel を使って行うインプリメンテーション 12
2.1.2 モデルをランする 15
2.2 ファイルからのデータ入力 19
2.2.1 Model formulation 20
2.2.2 Implementation インプリメンテーション 20
2.2.3 結果 21
2.2.4 データ・ドリブン・モデル 21
2.3 最適化と列挙（enumeration） 24
2.3.1 最適化 24
2.3.2 列挙（Enumeration） 26
2.4 連続変数 28
第３章 制約式 30
3.1 制約式のハンドリング 30
3.1.1 モデルの定式化 31
3.1.2 Implementation 31
3.1.3 結果 32
3.1.4 制約式に名前を付ける 32
3.1.5 制約式を明示的に渡す 34
3.1.6 明示的な制約式のプロパゲーション 34
3.2 算数制約式（Arithmetic constraints） 35
3.3 all_different: 数独 35
3.3.1 モデルの定式化 36
3.3.2 インプリメンテーション 37
3.3.3 結果 39
3.4.1 Model の定式化 41
3.4.2 インプリメンテーション 42

3.4.3 結果 45
3.5 element: 一つの機械でのジョブを順序付ける 45
3.5.1 モデルの定式化１ 46
3.5.2 モデル１のインプリメンテーション 47
3.5.3 結果 50
3.5.4 disjunction を使う代替的な定式化 51
3.5.5 モデル２のインプリメンテーション 52
3.6 occurrence: 砂糖の生産 gar production 55
3.6.1 モデルの定式化 56
3.6.2 インプリメンテーション 56
3.6.3 結果 58
3.7 distribute: 人員配置 59
3.7.1 Model formulation 59
3.7.2 インプリメンテーション 59
3.7.3 結果 62
3.8 implies : ペイントの生産 63
3.8.1 モデル１の定式化 63
3.8.2 モデル１のインプリメンテーション 64
3.8.3 モデル２の定式化 66
3.8.4 2 のインプリメンテーション 66
3.8.5 結果 67
3.9 equiv:所得税の税務署の配置場所 68
3.9.1 モデルの定式化 68
3.9.2 インプリメンテーション 70
3.9.3 結果 72
3.10 cycle : ペイントの生産 72
3.10.1 モデルの定式化 73
3.10.2 インプリメンテーション 73
3.10.3 結果 74
3.11 Generic binary constraints: オイラーのナイトツアー 75
3.11.1 モデルの定式化 75
3.11.2 インプリメンテーション 76
3.11.3 結果 78
3.11.4 代替的なインプリメンテーション 78
3.11.5 代替的なインプリメンテーション　その二 82
第４章 列挙（Enumeration） 84

4.1 事前に定義されているサーチ戦略 84
4.2 列挙を中断し、再開する方法 86
4.3 サーチの callback 87
4.4 ユーザサーチ戦略の定義 87
4.4.1 モデルの定式化 88
4.4.2 インプリメンテーション 89
4.4.3 ユーザによるサーチ 91
4.4.4 結果 94
第 5 章 スケジュール作成 95
5.1 タスクと資源 95
5.2 先行制約条件（Precedence constraints） 96
5.2.1 モデルの定式化 97
5.2.2 インプリメンテーション 97
5.2.3 結果 99
5.2.4 オブジェクトをスケジュールしない代替的な定式化 100
5.3 離接的なスケジュール作成（Disjunctive scheduling）： unary resources 101
5.3.1 モデルの定式化 101
5.3.2 インプリメンテーション 102
5.3.3 結果 105
5.4 Cumulative scheduling: discrete resources 105
5.4.1 モデルの定式化 106
5.4.2 インプリメンテーション 106
5.4.3 結果 108
5.4.4 オブジェクトをスケジュールしない代替的な定式化 108
5.4.5 インプリメンテーション 109
5.5 再生可能（renewable）な資源、再生可能ではない（non-renewable）資源 111
5.5.1 モデルの定式化 111
5.5.2 インプリメンテーション 113
5.5.3 結果 115
5.5.4 オブジェクトをスケジュールしない代替的な定式化 116
5.5.5 インプリメンテーション 117
5.6 拡張: 段取り時間 118
5.6.1 モデルの定式化 119
5.6.2 インプリメンテーション 119
5.6.3 結果 121
5.7 列挙（Enumeration） 121

5.7.1 変数ベースの列挙（Variable-based enumeration） 122
5.7.2 タスクベースの列挙（Task-based enumeration） 124
5.7.3 propagation algorithm の選択 129
Appendix A.. 131
トラブル・シューティング.. 131
Appendix B 132
Glossary of CP terms(未翻訳です。どうぞ、英文マニュアルをお読みください。) 132
Bibliography 133

第 1 章 イントロダクション

制約計画法（Constraint Programming）は、離散変数の非線形制約式で表現される関係に対処するのに、特にうまく機能する問題解決のアプローチです。そして、以下では、しばしば、「有限ドメイン制約計画法（finite domain Constraint Programming）」と同じ意味で「制約計画法」、もしくは、「CP」を'使用します。

CP は、過去において、「(資源制約式のある、もしくは、ない)生産スケジュール作成問題」、「タスクの順序付け、および、割当て問題（sequencing and assignment of tasks）」、「従業員計画と予定表問題（workforce planning and timetabling）」、「周波数割当問題（frequency assignment）」、「積込、カット問題（loading and cutting）[*]」、「グラフの塗りわけ問題（graph coloring）」のような多様な問題の解法として成功裏に適用なれてきています。

CP の強みは、それを、制約関係（constraint relations） の持つオリジナルな意味を保つことのできる制約式の記述のために、ハイレベルな意味論を使うことにあります。このようなハイレベルの制約式は「グローバル制約式（*global constraints*） と呼ばれます。 制約式を、数式の形に変換することは必要ありません。よくあることですが、数式の形に変換する過程で、問題の構造の多くが失われてしまいます。制約式に内在する、固有な問題に関する知識は、ソリューションアルゴリズムで利用され、アルゴリズムの効率をよくします。

1.1 Xpress-Kalis

「Xpress-Kalis」、別称、「Kalis for Mosel」により、Artelys 社の Kalis Constraint Programming ソルバーへのアクセスができます。Xpress-Kalis を通して、Kalis の制約計画の機能が、Mosel 環境で利用可能になり、ユーザは、Mosel 言語で CP モデルを定式化して、それを解くことが出来ます。

Xpress-Kalis は、「有限ドメインソルバー」と「連続(浮動小数点)変数のソルバー」を結合します。スケジュール作成問題の定式化を支援するために、このソフトウェアは、タスクと資源を表す特定のモデル作成オブジェクトの集合体を定義します。この特定のモデル作成オブジェクトの集合体は、自動的に、ある(暗黙的な)制約関係をセットアップして、このタイプの問題に専門化された、内蔵されているサーチ戦略をトリガーします。こうして、標準のスケジュール作成問題は、単に、対応するタスク、および、資源のオブジェクトをセットアップすることだけで、定義され、解くことができます。

Mosel IO ドライバーを使うことで、メモリ内でのデータ転送、モジュール mmodbc を通してのデータベースへの ODBC アクセスを含む、「Mosel 環境のすべてのデータハンドリング・ファシリティ」は、kalis により 、 なんら、余分な努力なしで使えます。

---

[*] 参照：http://www.dashoptimization.com/home/cgi-bin/example.pl?id=mosel_book_4

Mosel 言語は、ループ、また、より複雑なアルゴリズムを実行するのに必要なサブルーチンなどのような、典型的なプログラミングのための構成要素をサポートしています。Mosel は、また、Xpress-Kalis を Xpress-Optimizer に結合し、CP と LP/MIP を使って、共同で問題を解くことが出来るようにするための、異なったソルバーを結合するプラットホームとしても使用できます。(Dash Whitepaper Multiple models and parallel solving with Mosel を参照。)このマニュアルでは、Mosel を使って行うモデル作成とプログラミングの基礎について説明しますが、必要な場合、さらに高度なフィーチャーについても触れます。その使用法についての完全なドキュメンテーション、より徹底的なイントロダクション論については、Mosel language reference manual、および、 Mosel user guide を参照してください。

1.1.1 製品バージョンについてのノート

このマニュアルに記載の例は、「Xpress-Kalis の release 2007.1.0」、「Xpress-MP Release 2007A beta version of Mosel (1.7.9)」、および、「version 1.17.50 of Xpress-IVE」を使って書かれています。これらの例が、他のバージョンでランされると、得られるアウトプットは、ことなる形式で出力されるかもしれません。特に、その後の CP ソルバーのアルゴリズムの改善、Xpress-Kalis のディフォルト設定の変更により、制約式、サーチの動きは影響されるかもしれません。また、IVE インタフェースは、新機能が加えられて行くと、今後のリリースで、少し、変わって行くかもしれません。

1.2 ソフトウェアのインストレーション

Xpress-Kalis、Xpress-Mosel、Xpress-Kalis を使うには、これらのソフトウェアをインストールして、ライセンスを得なければなりません。CP モデルを扱う場合、Windows ユーザーは、グラフィカル環境 Xpress-IVE を追加インストールすると便利ですが、これは必須条件ではありません。

Mosel (および、IVE)をコンピュータにインストールするには、Xpress-MP の配布のときに、同時に配布されるインストールガイドに従ってください。次いで、Xpress-Kalis を、Xpress-Kalis の配布のときに、同時に配布されるインストラクションに従って、インストールしてください。

1.3 制約計画法(Constraint Programming)の基本概念

制約計画法(CP)の問題は、その決定変数によって定義され、また、これらの決定変数はドメインと制約式を持っています。問題定義は、通常、分岐戦略(branching strategy)によって終わります。分岐戦略は、列挙戦略(enumeration strategy)、サーチ戦略(search strategy)とも呼ばれています。

CP は、可変ドメイン(variable domain)の概念、すなわち、「決定変数が取る値の集合」を使います。「有限ドメイン制約計画法(finite domain Constraint Programming)」では、これらは、整数の集合(set)、または、インターバル(interval)です。

CP の各制約式は、通常、例えば、、グラフ理論などのような、他の領域の計算結果(result)に基づく、

それ自身のソリューションアルゴリズムを備えています。ひとたび、制約式が確定されると、それは、その変数集合を解かれた状態、すなわち、ソリューションアルゴリズムが、変数のドメインから、実行不可能と判定した値も取り除いた状態で保持します。

CP 問題の制約式は、「制約式プロパゲーション（constraint propagation）」と呼ばれるメカニズムによってリンクされています。すなわち、ある変数のドメインが変更されると、これは、この変数についてのすべての制約式の再評価をトリガーします。これにより、今度は、他の変数の変化を惹起したり、Figure1.1 の例に示されているように、最初の変数のドメインの一層の減少をトリガーしたりします。（変数の元々のドメインは、2 つの制約式の追加で減少します。 最後のステップで、2 番目の制約式の影響は、最初の制約式に伝播され、その再評価をトリガーします。）

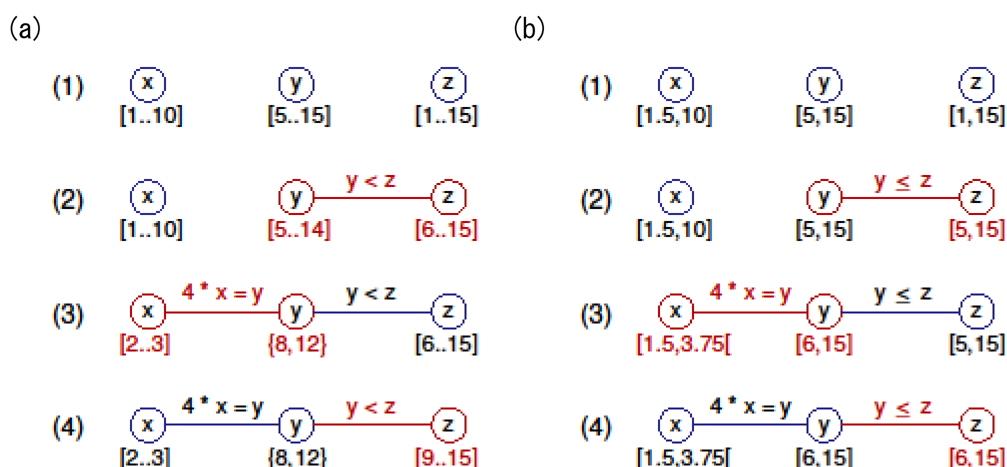


Figure 1.1: Example of constraint propagation, (a) finite domain (discrete) variables, (b) continuous variables.

CP 問題は、変数に、制約式と境界（bound）を加えることによって、徐々に、築きあげられてゆきます。CP 問題を解くことは、最初の制約式のステートメントを書くことから始まります。そして、制約式の関係を犯す値は、それに関る変数のドメインから取り除かれてゆきます。新たに加えられた制約式の結果は、すぐに全体の CP 問題に伝播されるので、一般に、問題からこの制約式を、後で変更したり、削除したりすることはできません。

場合によっては、制約式を解くこと、および、伝播のメカニズムの組み合わせだけで、ある問題インスタンス（problem instance）が実行不能であることを証明するの十分であることもあります。しかし、大抵の場合、変数のドメインを、ただ一つの値（整合性のあるインスタンス化や実行可能ソリューション）にまで減少させるための列挙（enumeration）を加えたり、そのような解が存在しないことを証明したりすることが必要です。さらに、最も良い目的関数の値(最適解)を持つ実現可能解をサーチするために、目的関数（もしくは、コスト関数）やサーチを定義することもできます。

1.4 この文書の内容

このドキュメントは、Xpress-Kalis の使い方のイントロダクションです。制約計画法の基本的な概念も

説明されますが、読者の皆さんが、CP のテクニックについての、何らかの一般的な理解があることを前提としています。また、Xpress-Kalis が Xpress-Mosel のモジュールなので、このドキュメントでは、提示される例の理解に必要であると思われる場合は、Mosel 言語のフィーチャーについても説明しています。

モデル例によって、Xpress-Kalis で定義される以下の新しい「タイプ」の使い方を例示します。

- 決定変数： 有限ドメイン（finite domain）変数と浮動小数点（floating point）変数
- 制約式： absolute value and distance、all-different、element、generic binary、linear、maximum と minimum、occurrence、logical relations
- 列挙（Enumeration）： 事前に定義された分岐スキームとユーザサーチ戦略
- スケジュール作成： タスクと資源を表す統合モデルオブジェクト

ここでは、CP モデルの一般構造、および、データハンドリングを含む、Mosel の使い方の基礎を説明します。次いで、Xpress-Kalis の異なる制約関係タイプの使い方を例示するために、一連の小さい問題が続きます。次の章は、もっと体系だった方法で、列挙戦略を定義するための、可能な、複数の方法について説明しますが、それらのいくつかは、その前で説明するモデル例で、既に、出てきているものです。スケジュール作成についての章では、スケジュール作成問題の定式化を易しくする「task」と「resource」というモデル作成のためのオブジェクトを導入します。

最初の方で出てくるいくつかの例は別として、すべての例が、以下のように示されます。

- 例についての説明
- CP モデルとしての定式化
- Implementation with Xpress-Kalis: code listing and explanations インプリメンテーションのための Xpress-Kalis コードのリストと説明
- 結果

このドキュメントに示されているモデル例は、すべて、Xpress-Kalis の配布時に、配布物の一部として提供される「例題集」の一部として含まれています。

第 2 章 モデル作成の基礎

本章では、以下のことを説明します。

- Mosel を使う
- Xpress-Kalis で、簡単な CP モデルを作成し、解く
- このソフトウェアが出力するアウトプットを理解し、分析する
- データハンドリングで、モデルを拡張する
- 目的関数を定義する
- ディフォルトの分岐戦略（branching strategy）を修正する

2.1 最初のモデル

次の問題について考えてみてください。1 から 3 という番号を付けられている「3 つの時間帯」に、4 つの会議 A、B、C、および、D をスケジュールしたいと考えています。いくつかの会議には、同じ人物の出席が必要です。このことは、これらの会議を同じ時間に開催できないことを意味しています。例えば、会議 A は、会議 B、会議 D と同時に開催することができず、会議 B は、会議 C、会議 D と同じ時間帯に開催できません。

これを、より厳密に表現すれば、以下のようになります。ここで、($m \in MEETINGS = \{A, B, C, D\}$) は、会議 m の時間帯を示し、これらは、この問題の決定変数です。

$$\forall m \in MEETINGS : plan_m \in \{1,2.3\}$$
$$plan_A \neq plan_B$$
$$plan_A \neq plan_D$$
$$plan_B \neq plan_C$$
$$plan_B \neq plan_D$$

2.1.1 Mosel を使って行うインプリメンテーション

以下のコードリストにより、上で説明した問題が、実行され、解かれます。

```
model "Meeting"
 uses "kalis"

 declarations
  MEETINGS = {'A','B','C','D'}        ! Set of meetings
  TIME = 1..3                         ! Set of time slots
  plan: array(MEETINGS) of cpvar      ! Time slot per meeting
 end-declarations
```

```
 forall(m in MEETINGS) setdomain(plan(m), TIME)

! Respect incompatibilities
 plan('A') <> plan('B')
 plan('A') <> plan('D')
 plan('B') <> plan('C')
 plan('B') <> plan('D')

! Solve the problem
 if not(cp_find_next_sol) then
  writeln("Problem is infeasible")
  exit(1)
 end-if

! Solution printing
 forall(m in MEETINGS)
  writeln("Meeting ", m, ": ", getsol(plan(m)))

end-model
```

このMoselモデルは、meeting.mosという名前で、テキストファイルとしてセーブされます。 以下で、上に書いたモデルを詳しく見てみましょう。

2.1.1.1 一般的な構造

Mosel プログラムは、すべて、model というキーワードから始まります。続いて、ユーザが選んだ model 名が続きます。そして、Mosel プログラムは、end-model というキーワードで終わります。

Mosel は、それ自体はソルバーではないので、プログラムの冒頭で、下記のステートメントで、Kalis constraint solver を使うことを指定します。

```
uses "kalis"
```

すべてのオブジェクトは、assignment で明確に定義されている場合を除き、declarations セクションでで宣言されなければなりません。例えば、i: =1 は i を整数と定義し、それに値 1 を割り当てます。モデルには、異なる場所に、このような declarations セクションがいくつかあるかもしれません。

現在、見ているこのケースでは、以下の二つの集合(set)と配列(array)を定義しています。

- MEETINGS は、strings の集合.
- TIME は、いわゆる、range set で、連続する整数の集合で、この例でえば、言えば、1 から 3 までの整数です。
- plan は、集合 MEETINGS というインデックスを付されたタイプ cpvar という決定変数の配列

です。ここに、cpvar は有限ドメイン CP 変数です（Xpress-Kalis の二つ目の決定変数のタイプは cpfloatvar で、これは、連続変数のためのものです）。

次いで、モデルは、Xpress-Kalis の手順（procedure）setdomain を使って変数のドメインを定義します。決定変数は、実際、それらの declaration により、デフォルトの大きなドメインを与えられて生成され、setdomain は、これらのドメインを、表示値に従って、デフォルトドメインの intersection まで減少させます。数学モデルと同じように、forall loop を使い、集合 MEETINGS の中で、すべてのインデックスを列挙（enumerate）します。

この後に、制約式のステートメントが続きます。今、ここで見ているモデルでは、4 つの（≠を持つ）disequality 制約式が制約式のステートメントです。

2.1.1.2 モデルを解き、アウトプットを出力する

cp_find_next_sol ファンクションで、Kalis を呼び、問題を解きます(決定変数のすべてにたいして、実行可能な値の割当てを見つける)。ここで、このファンクションの return value をテストします。もし、ソリューションが全く見つけられないと、このファンクションは、false を返してきますので、この時点で、Mosel 手順 exit を呼び、モデルの実行を止め、そうでない場合は、ソリューションを印刷します。

問題を解くために、Xpress-Kalis は、内蔵するデフォルト検索戦略を使います。後で、これらの戦略を変更する方法を説明します。

ファンクション getsol を使い、CP 変数のソリューションを得ます。一行にいくつかの項目を書くには、出力を印刷するための writeln ではなく、write を使います。

2.1.1.3 フォーマッティング

モデルを書くときに、インデント、スペース、空の行を使うと、モデルが読み易くなります。しかし、Mosel は、これらをスキップします。

Line breaks：一行に、いくつかのステートメントを書くことは可能です。それを行うには、下記のように、セミコロンでステートメントを切り離してください。

```
plan('A') <> plan('B'); plan('A') <> plan('D')
```

しかし、特別な'line end'や continuation character がないので、数行にまたがるステートメント行は、operator (+, >=, etc.)や'，'のようなキャラクタで終えなければなりません。それにより、ステートメントが終わっていないことが示されます。

例で示されているように. Mosel のコメント行は、! で始まります。複数行のコメント行は、る倍数が単線

コメントに先行する、、! で始まり、! で終わります。

2.1.2 モデルをランする

Mosel モデルをランする方法には、三つのモードがあり、その中から一つを選ぶことができます。

- Mosel command line から行う方法：この方法は、Mosel が利用可能であるすべてのプラットホームで使えます。 このモードは、例えば、異なるパラメタ設定を持つ複数のモデルの一連のランを実行したいような場合、特に便利です。このランモードでは、デバッガを含む、完全な Mosel の機能性にアクセスできます。

- Xpress-IVE のグラフィカル環境から行う方法：この方法は、Windows ユーザが使えます。IVE は、モデル作成と最適化のための開発環境で、Mosel モデルを扱うための内蔵のテキストエディタを持ち、また、多くの、ソリューション、および、サーチツリーのディスプレーも持っており、これは、開発段階で、モデルとソリューションアルゴリズムを分析するのにとても便利です。 モデルは、インタラクティブに変更しながらリランできます。

- アプリケーション・プログラムの中から行う方法： Mosel モデルは、アプリケーション・プログラム(C/C++、Java、VB、.NET)からアクセスでき、実行できます。この機能性は、通常、Mosel モデルを企業の情報システムと統合して使う場合に使用されます。

このマニュアルでは、このマニュアルで開発するモデルをランするために、最初の 2 つの方法を使います。モデルをアプリケーション・プログラムに埋め込むことについての詳細は、Mosel のユーザガイドを参照してください。

2.1.2.1 Mosel command line から行う方法

完全なモデルを、ファイル meeting.mos に入れれば、問題のソリューションに進むことができます。それには、下記の三つのステージが必要です。

- meeting.mos をコンパイルして、meeting.bim に変換する。
- コンパイルされた meeting.bim をロードする。
- ロードしたモデルをランする。

Mosel をコマンド・プロンプトから開始し、次の順序でコマンドを入力します。

```
mosel
compile meeting
load meeting
run
quit
```

これらのコマンドで、モデルはコンパイルされ、ロードされ、ランされます。そうすると、次のようなアウト

プットが出てきます。ここで、Mosel の出力はハイライトされ、太字で強調されます。

```
mosel
** Xpress-Mosel **
(c) Copyright Dash Associates 1998-2005
> compile meeting
Compiling 'meeting'...
> load meeting
> run
Meeting A: 1
Meeting B: 2
Meeting C: 1
Meeting D: 3
Returned value: 0
> quit
Exiting.
```

compile/load/run というシークエンスはよく使われるので、下記のように短く表記することもできます。

```
exec meeting
```

下記のコマンドラインから、同じステップが、即座に、行えます。

```
mosel -c "compile meeting; load meeting; run"
```

もしくは、

```
mosel -c "exec meeting"
```

-c オプションには、ダブルコーテーションで挟まれたコマンド・リストが続きます。

2.1.2.2 IVE を使う

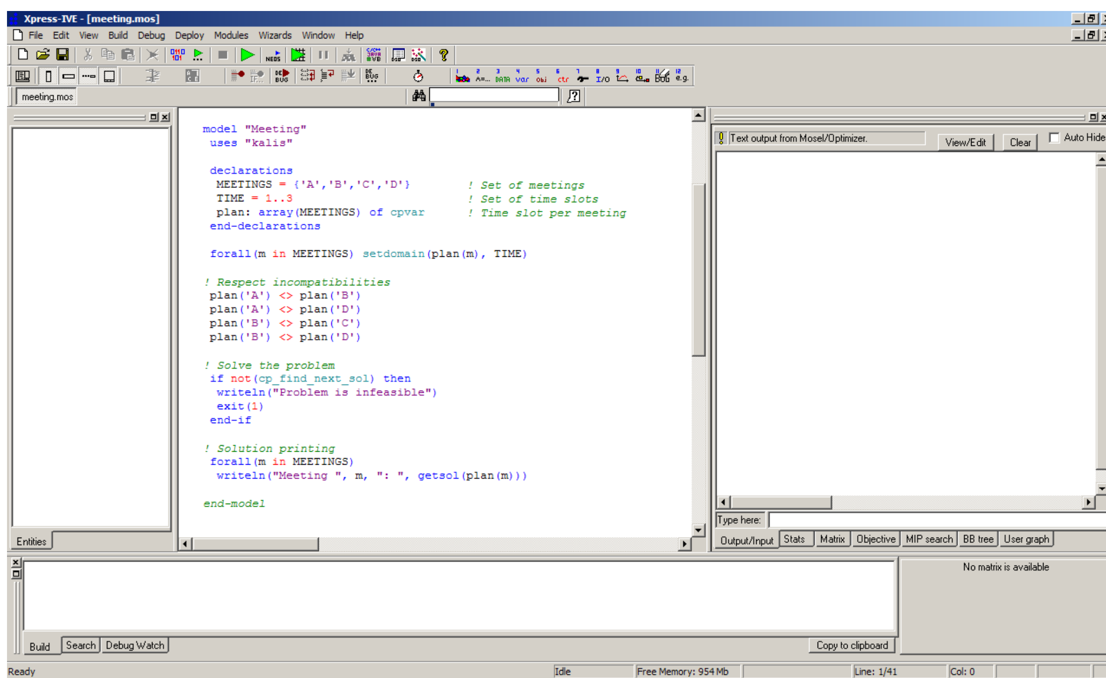
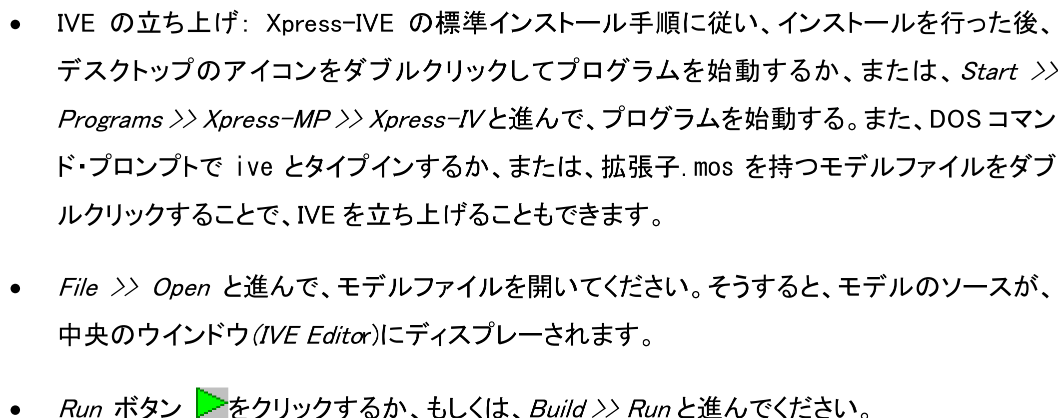


Figure 2.1: IVE after opening a model

モデルファイル meeting.mos を IVE で使うには、下記のステップを実行する必要があります。



- IVE の立ち上げ: Xpress-IVE の標準インストール手順に従い、インストールを行った後、デスクトップのアイコンをダブルクリックしてプログラムを始動するか、または、*Start >> Programs >> Xpress-MP >> Xpress-IV* と進んで、プログラムを始動する。また、DOS コマンド・プロンプトで ive とタイプインするか、または、拡張子.mos を持つモデルファイルをダブルクリックすることで、IVE を立ち上げることもできます。

- *File >> Open* と進んで、モデルファイルを開いてください。そうすると、モデルのソースが、中央のウインドウ(*IVE Editor*)にディスプレーされます。

- *Run* ボタン をクリックするか、もしくは、*Build >> Run* と進んでください。

ワークスペースの下部の *Build* 枠は、Mosel のモデル実行ステータスメッセージを表示します。モデルにシンタックスエラーが見つかると、それらは、ここに示され、エラーが検出された行とキャラクターの位置の詳細と、もしあるなら、問題の説明が表示されます。エラーをクリックすると、エラーを起こしている行にユーザを連れて行きます。

モデルのランが行われると、ワークスペース右側の *Output/Input* 枠には、プログラムによって生成されるアウトプットが表示されます。IVE は、また、CP のサーチツリーのグラフ表示(*CP search 枠*)と、問題のサマリー統計を表示(*CP 統計枠*)します。さらに、IVE では、Mosel モデルにサブルーチンを埋め

込むことによって、ユーザはグラフを描くことができます(詳細は、モジュール mmive のドキュメンテーションを見てください)。

IVE は、左手の窓の Entities 枠に、ソリューションのすべての情報を表示します。この枠で決定変数のリストを大きし、マウス･ポインタを近づけると、ソリューション値が表示されます。

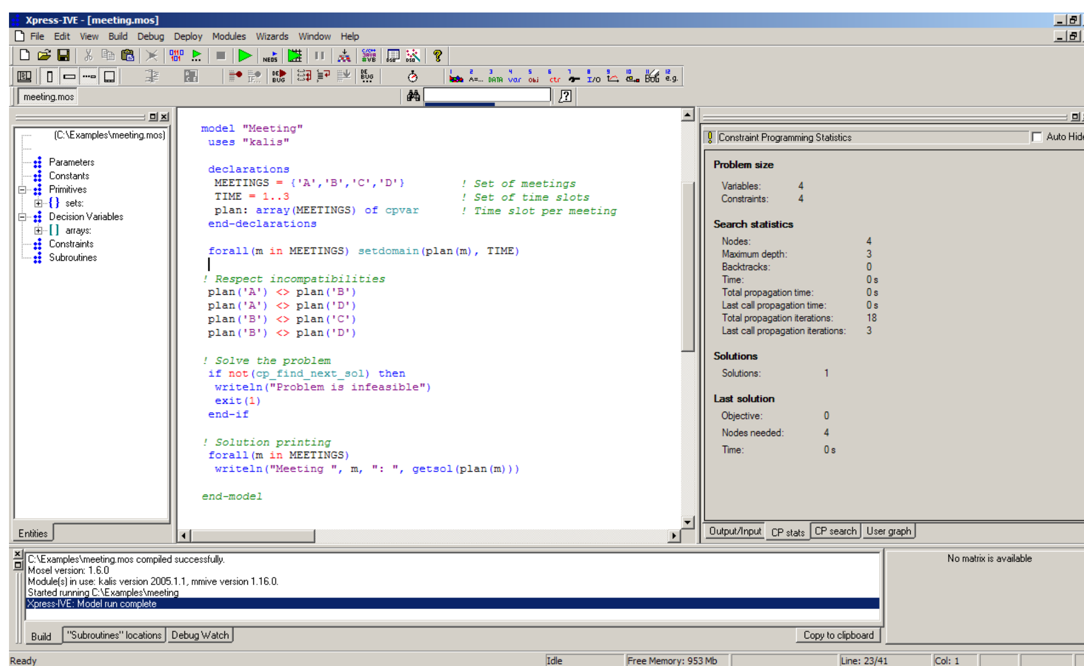


Figure 2.2: IVE after model execution

2.1.2.3 モデルをディバッグする

モデルをデバッグするための第一歩は、確かに、追加的な情報を加えることです。ここのモデルで言えば、例えば、変数のドメインの定義の後に、writeln(plan)というラインを加えることによって、決定変数の定義を印刷したり、また、手順 cp_show_prob により、問題定義を、全部、印刷したりすることもきます。デバッグを容易にするために、もっと読み易い出力を出させるには、ユーザは、決定変数に名前を与えることも出来ます。例えば、

```
forall(m in MEETINGS) setname(plan(m), "Meeting "+m)
```

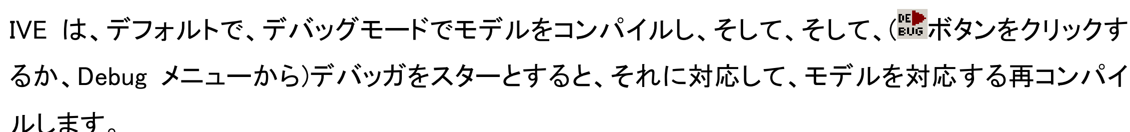
ここで、ストリングを連結する '+' サインを使用したことに注意してください。

手順 cp_show_stats を呼ぶと、CP solving のサマリー統計が表示されます。

コマンドラインバージョンで、run-time errors の詳細情報を得るためには、モデルを、 flag -g でコンパイルする必要があります。例えば、

```
mosel -c "exec -g meeting"
```

Mosel デバッガ(詳細は、Mosel 言語リファレンスマニュアルを参照)を使用するには、 compilation flag -G を使う必要があります。

IVE は、デフォルトで、デバッグモードでモデルをコンパイルし、そして、そして、(ボタンをクリックするか、Debug メニューから)デバッガをスターとすると、それに対応して、モデルを対応する再コンパイルします。

2.2 ファイルからのデータ入力

ここで、前の例を、別のデータセットと共に使用するために拡張します。 別の、異なるデータセットでモデルをランしたい場合、モデルファイルのデータセットの変更のすべてを編集するのは実用的ではなく、間違いの元となりがちです。したがって、そうする代わりに、テキストファイルからデータを読みこみますが、その方法について説明しましょう。

この問題は、Applications of optimization with Xpress-MPという文書のセクション14.4から取ってきた問題で、いま、これを解きたいと思っているとします。

ある技術大学では、いくつかのオプション・モジュールで、期末に、コース試験を計画する必要があります。試験は、すべて、2時間続きます。試験を行うために、2日間が用意され、両日とも、時間帯は、8:00－10:00、10:15－12:15、14:00－16:00、および、16:15－18:15で、合計8つの時間帯があります。Table2.1に見られるように、同じ学生が試験を受けなければならない科目は、同じ時間帯には行えず、したがって、そのような試験科目は、別の時間帯に行われなければなりません(set of incompatible exams)。

Table 2.1: Incompatibilities between different exams

	DA	NA	C++	SE	PM	J	GMA	LP	MP	S	DSE
DA	–	X	–	–	X	–	X	–	–	X	X
NA	X	–	–	–	X	–	X	–	–	X	X
C++	–	–	–	X	X	X	X	–	X	X	X
SE	–	–	X	–	X	X	X	–	–	X	X
PM	X	X	X	X	–	X	X	X	X	X	X
J	–	–	X	X	X	–	X	–	X	X	X
GMA	X	X	X	X	X	X	–	X	X	X	X
LP	–	–	–	–	X	–	X	–	–	X	X
MP	–	–	X	–	X	X	X	–	–	X	X
S	X	X	X	X	X	X	X	X	X	–	X

DSE	X	X	X	X	X	X	X	X	X	X	-

2.2.1 Model formulation

この CP モデルは、前のものと同じ構造をもっています。しかし、前のものとの違いは、データ配列 *INCOMP* で、これは、同時並行的に行えない試験科目のペアを示しており、ここでは、それらをひとつずつ書き上げることの代わりに、disequality constraints のループで定義します。

$$\forall e \in EXAM : plan_e \in \{1,\ldots,8\}$$
$$\forall d, e \in EXAM, INCOM_{de} = 1 : plan_d \neq plan_e$$

2.2.2 Implementation インプリメンテーション

この Mosel モデルは、いまや、下記にようになります。

```
model "I-4 Scheduling exams (CP)"
 uses "kalis"

 declarations
  EXAM = 1..11                        ! Set of exams
  TIME = 1..8                         ! Set of time slots
  INCOMP: array(EXAM,EXAM) of integer ! Incompatibility between exams
  EXAMNAME: array(EXAM) of string

  plan: array(EXAM) of cpvar          ! Time slot for exam
 end-declarations

 EXAMNAME:: (1..11)["DA","NA","C++","SE","PM","J","GMA","LP","MP","S","DSE"]

 initializations from 'Data/i4exam.dat'
  INCOMP
 end-initializations

 forall(e in EXAM) setdomain(plan(e), TIME)

! Respect incompatibilities
 forall(d,e in EXAM | d<e and INCOMP(d,e)=1)  plan(d) <> plan(e)

! Solve the problem
 if not(cp_find_next_sol) then
  writeln("Problem is infeasible")
  exit(1)
 end-if

! Solution printing
 forall(t in TIME) do
```

```
  write("Slot ", t, ": ")
  forall(e in EXAM)
   if (getsol(plan(e))=t) then write(EXAMNAME(e)," "); end-if
  writeln
 end-do

end-model
```

配列 *INCOMP* の値は、initializations block のファイル i4exam.dat から読み込まれます。disequality constraints の定義で、対応する array entry-conditions の値がチェックされ、loop、sum、その他の aggregate operators のインデックスは縦棒でマークされます。

このデータファイルは、下記の内容を持ちます。

```
INCOMP: [0 1 0 0 1 0 1 0 0 1 1
         1 0 0 0 1 0 1 0 0 1 1
         0 0 0 1 1 1 1 0 1 1 1
         0 0 1 0 1 1 1 0 0 1 1
         1 1 1 1 0 1 1 1 1 1 1
         0 0 1 1 1 0 1 0 1 1 1
         1 1 1 1 1 1 0 1 1 1 1
         0 0 0 0 1 0 1 0 0 1 1
         0 0 1 0 1 1 1 0 0 1 1
         1 1 1 1 1 1 1 1 1 0 1
         1 1 1 1 1 1 1 1 1 1 0]
```

2.2.3 結果

モデルは、以下の結果をプリントします。 答は、最初の 7 つの時間帯だけが試験のスケジュールに使用されることを示しています。

```
Slot 1: DA C++ LP
Slot 2: NA SE MP
Slot 3: PM
Slot 4: GMA
Slot 5: S
Slot 6: DSE
Slot 7: J
Slot 8:
```

2.2.4 データ・ドリブン・モデル

上のモデルでは、incompatibility データを、ファイルから読み込みましたが、データの一部(すなわち、index set *EXAM* と使える時間帯の数)は、モデルで、いまだ、ハードコードされています。

任意のデータセットでランできる、完全にフレキシブルなモデルを作成するには、すべてのデータ定義を、モデルからデータファイルに移動する必要があります。

新しいデータファイル i4exam2.dat は、データエントリーを定義するだけではなく、配列 *INCOMP* の index tuple も定義します。データエントリーは、すべて、その前に index tuple が付きます。set *EXAM* の内容を明確に書く必要はありません。なぜなら、Mosel は、この set に、 配列 *INCOMP* のインデックスの値を自動的に読み込むからです。さらに、このデータファイルには、時間帯 NT の数の値もあります。

```
INCOMP: [("DA" "NA") 1 ("DA" "PM") 1 ("DA" "GMA") 1 ("DA" "S") 1 ("DA" "DSE") 1
         ("NA" "DA") 1 ("NA" "PM") 1 ("NA" "GMA") 1 ("NA" "S") 1 ("NA" "DSE") 1
         ("C++" "SE") 1 ("C++" "PM") 1 ("C++" "J") 1 ("C++" "GMA") 1
 ("C++" "MP") 1 ("C++" "S") 1 ("C++" "DSE") 1
         ("SE" "C++") 1 ("SE" "PM") 1 ("SE" "J") 1 ("SE" "GMA") 1 ("SE" "S") 1
 ("SE" "DSE") 1
         ("PM" "DA") 1 ("PM" "NA") 1 ("PM" "C++") 1 ("PM" "SE") 1 ("PM" "J") 1
 ("PM" "GMA") 1 ("PM" "LP") 1 ("PM" "MP") 1 ("PM" "S") 1 ("PM" "DSE") 1
         ("J" "C++") 1 ("J" "SE") 1 ("J" "PM") 1 ("J" "GMA") 1 ("J" "MP") 1
 ("J" "S") 1 ("J" "DSE") 1
         ("GMA" "DA") 1 ("GMA" "NA") 1 ("GMA" "C++") 1 ("GMA" "SE") 1
 ("GMA" "PM") 1 ("GMA" "J") 1 ("GMA" "LP") 1 ("GMA" "MP") 1
 ("GMA" "S") 1 ("GMA" "DSE") 1
         ("LP" "PM") 1 ("LP" "GMA") 1 ("LP" "S") 1 ("LP" "DSE") 1
         ("MP" "C++") 1 ("MP" "PM") 1 ("MP" "J") 1 ("MP" "GMA") 1 ("MP" "S") 1
 ("MP" "DSE") 1
         ("S" "DA") 1 ("S" "NA") 1 ("S" "C++") 1 ("S" "SE") 1 ("S" "PM") 1
 ("S" "J") 1 ("S" "GMA") 1 ("S" "LP") 1 ("S" "MP") 1 ("S" "DSE") 1
         ("DSE" "DA") 1 ("DSE" "NA") 1 ("DSE" "C++") 1 ("DSE" "SE") 1
 ("DSE" "PM") 1 ("DSE" "J") 1 ("DSE" "GMA") 1 ("DSE" "LP") 1
 ("DSE" "MP") 1 ("DSE" "S") 1 ]

NT: 8
```

上のことを反映させるため、モデルを、いくつか変える必要があります。set *EXAM* と set *TIME* は、いまや、タイプをはっきり示すことによって、declare されています。これにより、これらの set は、それまでは、それらの値を提示することによって、constant definition であったのとは対照的に、 いまや、これらの set は dynamic set になります。その結果、決定変数 plan は、インデックスセット *EXAM* が知られる前に、そして、Mosel がこの配列を *dynamic array* として作成する前に、declare されます。これは、この declaration が空の配列を作り、インデックスリストがひとたび知られると、(Mosel 手順 create を使い)、その要素が作られる必要があることを意味しています。変数を作る前に、Xpress-Kalis のデフォルト bound を set *TIME* に対応する値に変更し、setdomain への call を置き換えます。

配列 *INCOMP* の declaration は、また、dynamic array を生成します。initializations block は、正確に、データファイルにリストされているエントリーに、値を割り当てます(前に行われている constant

declaration で、すべてのエントリーが定義されています)。これにより、disequality constraints を定義するループの condition を、定義し直すことが可能になります。こうして、いまや、すべてのデータの値を確認することに代わり、エントリーの存在をテストすればよいことになりました。大きいデータセットの場合、keyword exists を使うことで、「疎な配列(sparse array)」、すなわち、多次元のデータ配列で、0 以外のエントリーが極めて少ない配列でのループの実行時間を大きく減少できるでしょう。

```
model "I-4 Scheduling exams (CP) - 2"
 uses "kalis"

 declarations
  NT: integer                           ! Number of time slots
  EXAM: set of string                   ! Set of exams
  TIME: set of integer                  ! Set of time slots
  INCOMP: array(EXAM,EXAM) of integer ! Incompatibility between exams

  plan: array(EXAM) of cpvar            ! Time slot for exam
 end-declarations

 initializations from 'Data/i4exam2.dat'
  INCOMP NT
 end-initializations

 TIME:= 1..NT

 setparam("default_lb", 1); setparam("default_ub", NT)
 forall(e in EXAM) create(plan(e))

! Respect incompatibilities
 forall(d,e in EXAM | exists(INCOMP(d,e)) and d<e)  plan(d) <> plan(e)

! Solve the problem
 if not(cp_find_next_sol) then
  writeln("Problem is infeasible")
  exit(1)
 end-if

! Solution printing
 forall(t in TIME) do
  write("Slot ", t, ": ")
  forall(e in EXAM)
   if (getsol(plan(e))=t) then write(e," "); end-if
  writeln
 end-do

end-model
```

この、完全に「データ・ドリブン形式になったモデル」も、勿論、前のモデルと同一の解を与えます。

決定変数 *plan*を明示的に作成することの代替的な方法は、下に示すように、データの initialization の後に declaration を動かすことです。この場合、index set *EXAM* を完成させ、まとめておくこと(finalize)が重要です。そうしておくことにより、index set EXAM は、現在のコンテンツを持つ constant set になり、Mosel は、その後に declare されるすべての配列を、この set によりインデックスを付し、static array として生成します。

```
declarations
 NT: integer                          ! Number of time slots
 EXAM: set of string                  ! Set of exams
 TIME: set of integer                 ! Set of time slots
 INCOMP: array(EXAM,EXAM) of integer ! Incompatibility between exams
end-declarations

initializations from 'Data/i4exam2.dat'
 INCOMP NT
end-initializations

finalize(EXAM)
TIME:= 1..NT

setparam("default_lb", 1); setparam("default_ub", NT)
declarations
 plan: array(EXAM) of cpvar          ! Time slot for exam
end-declarations
```

2.3 最適化と列挙 (enumeration)

2.3.1 最適化

セクション 2.2.2 のモデル i4exam_ka.mos をランした結果、すべての時間帯を使用してはいない解が得られましたが、これからの連想で、この問題が必要とする最小の時間帯の数はいくつだろうか、と考えるでしょう。この質問は、一つの最適化の問題で、それを定式化しましょう。

ここで、$plan_i$変数と同じ値の範囲(the same value range)を対象に、新しい決定変数 *numslot* を導入し、この変数は、すべての $plan_i$変数に等しいか、大きいということを示す制約式を加えます。 単純化した、一つの定式化は、変数 *numslot* は、すべての $plan_i$変数の最大値と等しい、とすることです。

そうすると、目的関数は、*numslot* の値を最小化することとなり、そうすると、下記のモデルが得られます。

```
model "I-4 Scheduling exams (CP) - 3"
```

```
uses "kalis"

declarations
 NT: integer                          ! Number of time slots
 EXAM: set of string                  ! Set of exams
 TIME: set of integer                 ! Set of time slots
 INCOMP: array(EXAM,EXAM) of integer ! Incompatibility between exams

 plan: array(EXAM) of cpvar           ! Time slot for exam
 numslot: cpvar                       ! Number of time slots used
end-declarations

initializations from 'Data/i4exam2.dat'
 INCOMP NT
end-initializations

finalize(EXAM)
TIME:= 1..NT

setparam("default_lb", 1); setparam("default_ub", NT)
forall(e in EXAM) create(plan(e))

! Respect incompatibilities
forall(d,e in EXAM | exists(INCOMP(d,e)) and d<e)  plan(d) <> plan(e)

! Calculate number of time slots used
numslot = maximum(plan)

! Solve the problem
if not(cp_minimize(numslot)) then
 writeln("Problem is infeasible")
 exit(1)
end-if

! Solution printing
forall(t in TIME) do
 write("Slot ", t, ": ")
 forall(e in EXAM)
  if (getsol(plan(e))=t) then write(e," "); end-if
 writeln
end-do

end-model
```

cp_find_next_sol に代わり、ここでは、目的関数変数 numslot を持つ cp_minimize を function argument として使います。

このプログラムは、利用可能な 7 つの時間帯を前提にソリューションを与え、その答は、実行可能なスケジュールを作り出すのに必要な最少時間帯数を示します。

2.3.2 列挙 (Enumeration)

IVE の *CP stats* pane や、モデルのバージョンの終わりに追加する cp_show_stats をコールすることで得られる problem statistics を比較すると分かりますが、「実行可能なソリューションを見つける」ということから、「最適化」に切り替えると、CP ソルバーがサーチするノードの数が、かなり、増加します。

これまでのところ、Xpress-Kalis のデフォルトの列挙戦略でサーチを行ってきました。ここで、サーチするノードの数を減少させることができるかどうか、したがって、最適性を確認するのに費やされる時間を短くさせることができるかどうかを見てみましょう。

有限ドメイン変数を列挙するための Kalis のデフォルト戦略は、サーチの始まりの前に、下記のステートメントを追加することに相当します。

```
cp_set_branching(assign_var(KALIS_SMALLEST_DOMAIN, KALIS_MIN_TO_MAX))
```

assign_var は、branching scheme (`a branch is formed by assigning the next chosen value to the branching variable')を意味し、KALIS_SMALLEST_DOMAIN は、variable selection strategy (`choose the variable with the smallest number of values remaining in its domain')であり、KALIS_MIN_TO_MAX は、value selection strategy (`from smallest to largest value')です。

ここでは、時間帯の数を最小にしようとているので、最も小さい値から列挙を開始するのはよい考えのように思えます。したがって、デフォルト値を選択基準としたままでおきましょう。しかし、ここで、KALIS_SMALLEST_DOMAIN を KALIS_MAX_DEGREE に変えて、変数選択のヒューリスティクスを変えて見ましょう。そうすると、ツリーのサイズとサーチ時間が、デフォルトの半分以下になりました。

以下は、こうした改訂を行った後のモデルの全体です。

```
model "I-4 Scheduling exams (CP) - 4"
 uses "kalis"

 declarations
  NT: integer                          ! Number of time slots
  EXAM: set of string                  ! Set of exams
  TIME: set of integer                 ! Set of time slots
  INCOMP: array(EXAM,EXAM) of integer  ! Incompatibility between exams

  plan: array(EXAM) of cpvar           ! Time slot for exam
  numslot: cpvar                       ! Number of time slots used
 end-declarations

 initializations from 'Data/i4exam2.dat'
  INCOMP NT
 end-initializations
```

```
 finalize(EXAM)
 TIME:= 1..NT

 setparam("default_lb", 1); setparam("default_ub", NT)
 forall(e in EXAM) create(plan(e))

! Respect incompatibilities
 forall(d,e in EXAM | exists(INCOMP(d,e)) and d<e)  plan(d) <> plan(e)

! Calculate number of time slots used
 numslot = maximum(plan)

! Setting parameters of the enumeration
 cp_set_branching(assign_var(KALIS_MAX_DEGREE, KALIS_MIN_TO_MAX))

! Solve the problem
 if not(cp_minimize(numslot)) then
  writeln("Problem is infeasible")
  exit(1)
 end-if

! Solution printing
 forall(t in TIME) do
  write("Slot ", t, ": ")
  forall(e in EXAM)
   if (getsol(plan(e))=t) then write(e," "); end-if
  writeln
 end-do

 cp_show_stats

end-model
```

注意: 最適化のないモデルバージョンでは、ランダムに値を選ぶので、より均等に分布するスケジュールを得ようとしています。すなわち、Kalis_MIN_TO_MAX ではなく、値の選択基準 Kalis RANDOM_VALUE を使用している、ということです。

分岐戦略の定義の詳細は、第 4 章を見てください。

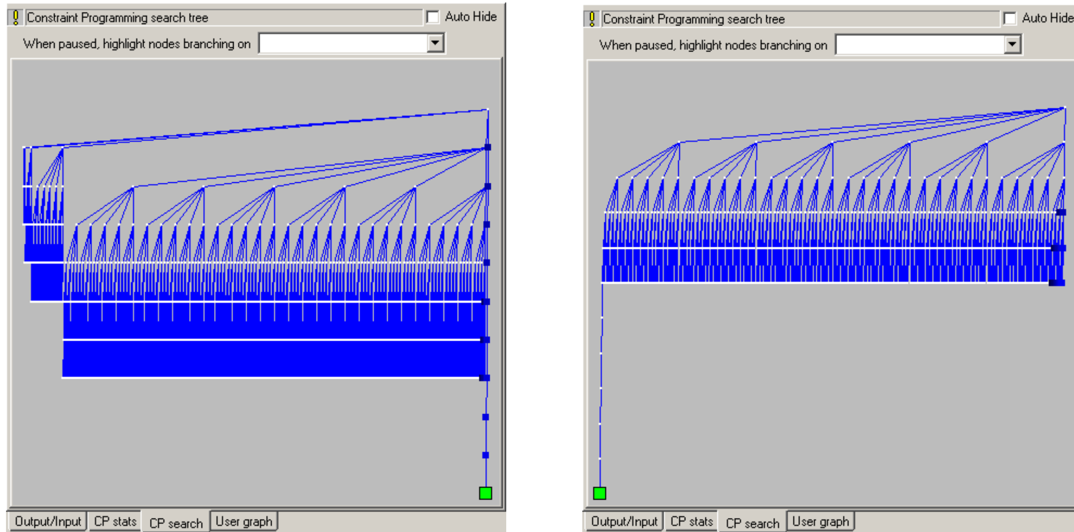


Figure 2.3: Search tree displays in IVE (left: default strategy, right: KALIS_MIN_DEGREE strategy for variable choice)

2.3.2.1 IVE search tree display

IVE では、この二つのサーチツリーの違いは、CP search pane に示されるツリーのグラフィカルな表示により、容易に目に見えるようになります(Figure2.3 を参照)。実現可能な解は緑色の正方形によって表され、最適化問題の最適解は、わずかに大きい正方形でマークされます。Figure2.3 で見るように、両方の戦略で、一つのソリューションが見つけられます。サーチツリー・ディスプレーのノードの上で、マウスのポインターを動かすと、ノード番号、分岐変数の名前を含む情報の詳細がポップアップボックスの中に現れます。また、ツリー・ディスプレーの上の selection box に変数名を示してやると、所与の変数で分岐するすべてのノードをハイライトすることもできます。IVE で、もっと意味のあるディスプレーを得たい場合は、手順 setname を使い、決定変数に名前を割り当てます(セクション 2.1.2.3 を参照)。

2.4 連続変数

本章を通して、これまで、決定変数 type cpvar(離散変数)を扱ってきました。Xpress-Kalis の 2 番目の変数 type は、連続変数(type cpfloatvar)です。このような変数は、上で、離散変数について見てきたのと同様の方法で使います。例えば、下記の例を見てください。

```
setparam("DEFAULT_CONTINUOUS_LB", 0)
setparam("DEFAULT_CONTINUOUS_UB", 10)

declarations
 x,y: cpfloatvar
end-declarations

x >= y                    ! Define a constraint
                          ! Retrieve information about continuous variables
```

```
writeln(getname(x), ":", getsol(x))
writeln(getlb(y), " ", getub(y))
```

離散変数タイプと連続変数タイプの使い方には、下記のような、いくつかの違いがあります。

- cpfloatvar を含む制約式は、厳密な不等式(strict inequalities)にはなれない(すなわち、演算子は、<=、>=、および、= だけが使用できる。
- ほとんどの global constraints(第 3 章を参照)は、cpvar のみに適用できる。
- 変数のドメインで値を列挙するサーチ戦略は、cpvar のみとしか使えない(第 4 章を参照)。
- getnext のような、ドメインの値を列挙する access functions は、cpfloatvar に適用できない。

第 3 章 制約式

本章では、異なるタイプの(最適化)問題例を使い、これらを解くために、どのように Xpress-Kalis を使う方法を示します。最初のセクションは、簡単な線形制約式の例で、問題をどのように定義し、ポストするためのいくつかの別の方法を示します。以下のいくつかのセクションでは、それぞれ、新しい制約式タイプを導入します。 ほとんどの例が、異なる制約式の組み合わせを使用するので、以下のリストは、ある、一つの制約式タイプの使用例を見つけるの役立つ TOC 思います。

- **arithmetic**: **2.1** (meeting.mos), **2.2** (exam*.mos), **3.7** (persplan.mos);
  linear equality/inequality: **3.5** (b4seq*_ka.mos), **3.6** (i1assign_ka.mos),
  3.8 (b5paint*_ka.mos), **3.9** (j5tax_ka.mos), **4.4** (i1assign_ka.mos),
  5.2 (b1stadium_ka.mos);
  Non-linear}nonlinear: **3.2**
- **all_different**: **3.3** (sudoku_ka.mos), **3.5** (b4seq_ka.mos), **3.8** (b5paint*_ka.mos),
  3.4 (freqasgn.mos),.**3.7** (persplan.mos), **3.11** (eulerkn*.mos), **4.4** (i1assign_ka.mos)
- **abs / distance**: **3.4** (freqasgn.mos)
- **cumulative**:**5.4** (d4backup2_ka.mos)
- **cycle**:**3.10** (b5paint3_ka.mos)
- **disjunctive**: **3.5** (b4seq2_ka.mos)
- **element**:**3.5** (b4seq_ka.mos), **3.6** (a4sugar_ka.mos), **3.9** (j5tax_ka.mos),
  3.8 (b5paint_ka.mos), **4.4** (i1assign_ka.mos), ; **2D**: **3.8** (b5paint2_ka.mos)
- **distribute / occurrence**: **3.6** (a4sugar_ka.mos), **3.7** (persplan.mos), **3.9** (j5tax_ka.mos)
- **maximum / minimum**: **2.3** (exam3.mos, exam4.mos), **3.5** (b4seq2_ka.mos),
  3.4 (freqasgn.mos), **4.4** (i1assign_ka.mos)
- **implies/equiv**:**3.8** (b5paint_ka.mos), **3.7** (persplan.mos), **3.9** (j5tax_ka.mos),
  3.11 (eulerkn2.mos)
- **generic binary**:3.1 (eulerkn.mos)

3.1 制約式のハンドリング

このセクションでは、セクション 2.1 の会議のスケジュールを作成するという問題を例にとります。このモデルには、離散変数についての簡単な算数制約式からなっています。しかし、ここで言えることは、すべて、他の、連続決定変数(すなわち、タイプ cpfloatvar の変数)の制約式関係を含む、すべてのタイプの Xpress-Kalis の制約式についても言えることです。

下に、この問題の制約式として、いくつか、付け加えたいと思います。

- 会議 B は、第 3 日よりも前に開催されなければならない。
- 会議 D は、第 2 日においては開催できない。
- 会議 A は、第 1 日に開催されなければならない。

3.1.1 モデルの定式化

これらの 3 つの追加制約式は、簡単な(単項)リニア制約式で表現できます。こうして、以下のように、モデルを完成します。

$$\forall m \in MEETINGS : plan_m \in \{1,2,3\}$$
$$plan_B \leq 2$$
$$plan_D \neq 2$$
$$plan_A = 1$$
$$plan_A \neq Plan_B$$
$$plan_A \neq Plan_D$$
$$plan_B \neq Plan_C$$
$$plan_B \neq Plan_D$$

3.1.2 Implementation

この新しい制約式の Mosel による実行は、極めて、簡単です。

```
model "Meeting (2)"
 uses "kalis"

 declarations
  MEETINGS = {'A','B','C','D'}        ! Set of meetings
  TIME = 1..3                         ! Set of time slots
  plan: array(MEETINGS) of cpvar      ! Time slot per meeting
 end-declarations

 forall(m in MEETINGS) do
  setdomain(plan(m), TIME)
  setname(plan(m), "plan"+m)
 end-do
 writeln("Original domains: ", plan)

 plan('B') <= 2                       ! Meeting B before day 3
 plan('D') <> 2                       ! Meeting D not on day 2
 plan('A') = 1                        ! Meeting A on day 1
 writeln("With constraints: ", plan)

! Respect incompatibilities
```

```
 plan('A') <> plan('B')
 plan('A') <> plan('D')
 plan('B') <> plan('C')
 plan('B') <> plan('D')

! Solve the problem
 if not(cp_find_next_sol) then
  writeln("Problem is infeasible")
  exit(1)
 end-if

! Solution printing
 forall(m in MEETINGS)
  writeln("Meeting ", m, ": ", getsol(plan(m)))

end-model
```

3.1.3 結果

読者の皆さんも、多分、既に、お気付きのように、モデルの中の数箇所に、変数 $plan_m$ のプリントアウトを加えました。したがって、このモデルの実行の出力は以下のようになります。

```
Original domains: [planA[1..3],planB[1..3],planC[1..3],planD[1..3]]
With constraints: [planA[1],planB[1..2],planC[1..3],planD[1,3]]
Meeting A: 1
Meeting B: 2
Meeting C: 1
Meeting D: 3
```

この出力からわかるように、制約式を記述した直後に、関係する変数のドメインは減少しています。制約式は、「即座に、かつ、自動的に」ソルバーに渡され、そして、それらの結果は問題全体に伝播されます。

3.1.4 制約式に名前を付ける

定義を行った後に、特に、制約式がロジック関係の一部なったり、(disjunctive constraints のように)ブランチされたりするような場合、後で制約式にアクセスすることが必要になることがあります。この目的に対応するため、Xpress-Kalis は、新しいタイプ、cpctr を定義します。 cpctr は、制約式を declare し、制約式が定義された後に使う「一つの名前」を、それらの制約式に与えることに使います。制約式は、制約式関係(constraint relation)を割り当てることによって、定義されます。

制約式に名前をつけることにより、制約式が自動的にソルバーに渡されないという副次的な効果があります。これは、以下のモデルで示されているように、(定義の後に)それ自身へのステートメントとして制約式の名前を書くことで行う必要があります。

```
model "Meeting (3)"
 uses "kalis"

 declarations
  MEETINGS = {'A','B','C','D'}        ! Set of meetings
  TIME = 1..3                         ! Set of time slots
  plan: array(MEETINGS) of cpvar      ! Time slot per meeting
  Ctr: array(range) of cpctr
 end-declarations

 forall(m in MEETINGS) do
  setdomain(plan(m), TIME)
  setname(plan(m), "plan"+m)
 end-do
 writeln("Original domains: ", plan)

 Ctr(1):= plan('B') <= 2             ! Meeting B before day 3
 Ctr(2):= plan('D') <> 2             ! Meeting D not on day 2
 Ctr(3):= plan('A') = 1              ! Meeting A on day 1
 writeln("After definition of constraints:¥n ", plan)

 forall(i in 1..3) Ctr(i)
 writeln("After posting of constraints:¥n ", plan)

! Respect incompatibilities
 plan('A') <> plan('B')
 plan('A') <> plan('D')
 plan('B') <> plan('C')
 plan('B') <> plan('D')

! Solve the problem
 if not(cp_find_next_sol) then
  writeln("Problem is infeasible")
  exit(1)
 end-if

! Solution printing
 forall(m in MEETINGS) writeln("Meeting ", m, ": ", getsol(plan(m)))

end-model
```

このモデルの出力から、名付けられた制約式の単なる定義によっては、変数のドメインに、なんら、影響しないことがわかります。すなわち、制約式は、Mosel で定義されますが、いまだ、Kalis ソルバーに送られていない、ということです。制約式の名前をステートした後にだけ、すなわち、それらをソルバーに送った後においてのみ、前のモデルバージョンと同じ、ドメインの減少が得られます。

```
Original domains: [planA[1..3],planB[1..3],planC[1..3],planD[1..3]]
```

```
After definition of constraints:
 [planA[1..3],planB[1..3],planC[1..3],planD[1..3]]
After posting of constraints:
 [planA[1],planB[1..2],planC[1..3],planD[1,3]]
Meeting A: 1
Meeting B: 2
Meeting C: 1
Meeting D: 3
```

3.1.5 制約式を明示的に渡す

これまでのすべての例で、制約式をソルバーに渡しても、失敗しない、すなわち、ソルバーがインフィージビリティを見つけないと、暗黙に仮定していました。

実際には、これは、必ずしも、常時、適切な仮定あるわけではありません。したがって、Xpress-Kalis は、ソルバーに明示的に制約式を渡し、それを追加した後に、制約式システムの状態を返すファンクション cp_post を定義します。このファンクションは、フィージブルには true を、そしてインフィージブルには false を返してきます。この機能は、制約を厳しくし過ぎた問題に対応するのに便利です。つまり、インフィージビリティが見つかると、それ以上の制約式の追加を止め、そして、どの制約式がインフィージビリティを惹き起しているかをユーザに知らせてくれるからです。

cp_post を使うには、前のモデルで、forall(i in 1..3) Ctr(i)という行を、下記のコードに置き換えます。

```
forall(i in 1..3)
 if not cp_post(Ctr(i)) then
  writeln("Constraint ", i, " makes problem infeasible")
  exit(1)
 end-if
```

制約式をソルバーに渡すと、そのすべてについて、返されてくる値がチェックされ、ある一つの制約式の追加がインフィージビリティを惹き起こすと、プログラムは止められます。

このモデルバージョンの出力は、前のセクションで見たものと、まったく、同一です。

3.1.6 明示的な制約式のプロパゲーション

別の方法で、制約式の行動に影響を及ぼすこともできます。自動的な制約式のプロパゲーションをオフにする(setparam("AUTO_PROPAGATE", false)と、ソルバーに渡された制約式のプロパゲーションは行われません。この場合、制約式のプロパゲーションは、cp_propagate へのコールか、列挙(subroutines cp_find_next_sol, cp_minimize, etc.)の開始によってのみ始められます。

3.2 算数制約式 (Arithmetic constraints)

前のセクションで、有限ドメイン変数の線形制約式の例をいくつか見ました。線形制約式は、算数制約式の特殊なケース、すなわち、演算子 +, -, /, *, ^, sum, prod や、abs、ln のような関数で形成されている方程式や不等式であると見なされます。ソルバーによってサポートされている算数ファンクションの全リストについては、Xpress-Kalis reference manual を参照してください。

Xpress-Kalis の算数制約式は、有限ドメイン変数(type cpvar)、連続型変数(type cpfloatvar)、もしくは、両方が混ざったもので定義されます。しかし、連続変数を含む算数制約式は、strict inequalities としては定義できないのに注意してください。これは、関係を示す演算子として、>=、<=、 および、 = だけが使用できることを意味します。

以下に、Xpress-Kalis で定義できる非線形算数制約式のいくつかの例を示します。

```
model "Nonlinear constraints"
 uses "kalis"

 setparam("DEFAULT_LB", 0)
 setparam("DEFAULT_UB", 5)
 setparam("DEFAULT_CONTINUOUS_LB", -10)
 setparam("DEFAULT_CONTINUOUS_UB", 10)

 declarations
  a,b,c: cpvar
  x,y,z: cpfloatvar
 end-declarations

 x = ln(y)
 y = abs(z)
 x*y <= z^2
 z = -a/b
 a*b*c^3 >= 150

 while (cp_find_next_sol)
  writeln("a:", getsol(a), ", b:", getsol(b), ", c:", getsol(c),
         ", x:", getsol(x), ", y:", getsol(y), ", z:", getsol(z))

end-model
```

3.3 all_different: 数独

「数独パズル」は、日本発のパズルで、最近、多くの欧米の新聞でも見られるようになりました。 このパズルは、与えられた 9x9 のボードに 1 から 9 の数字を埋めてゆくのですが、このボードの一部の枡には数字が入っています。そして、問題は、空白の升目に埋め込む数字は、その升目の所属している行、列、および、3x3 のサブボードのすべてが、1 から 9 の数字を一つだけ含むように数字を埋めて、

9x9 のボードのすべての空の枡に数字を入れ込むことです。Table 3.1 と Table 3.2 は、このパズルの 2 つの例題です。 人が解くには、扱いにくい問題ですが、これらのパズルは、CP を使って解くことが出来ます。

（訳注：「数独パズル」については、http://www.oct.zaq.ne.jp/woodside/jsudok/ を参照。

Table 3.1: Sudoku (`The Times', 26 January, 2005)

	A	B	C	D	E	F	G	H	I
1					4	3		6	
2		6	5				7		
3	8			7					3
4		5				1	3		7
5	1	2						8	4
6	9		7	5				2	
7	4					5			9
8			9				4	5	
9		3		4	6				

Table 3.2: Sudoku (`The Guardian', 29 July, 2005)

	A	B	C	D	E	F	G	H	I
1	8					3			
2		5					4		
3	2				7			6	
4				1					5
5			3				9		
6	6					4			
7		7			2				3
8			4					1	
9				9					8

3.3.1 モデルの定式化

これらの例のように、ボードの列を集合 $XS = \{A, B, \ldots, I\}$で、そして、行を集合 $YS = \{1, 2, \ldots, 9\}$で示します。XS のすべての x と、YS のすべての y に関して、決定変数 v_{xy} を定義し、その値として、位置(x, y)の数を取るとします。

この問題の制約式は、下記のものだけです。

(1) ある行の数字は、すべて、異なる数字でなければならない。

(2) ある列の数字は、すべて、異なる数字でなければならない。

(3) 3x3 のサブボードの数字は、すべて、異なる数字でなければならない。

これらの制約式は、Xpress-Kalis の all_different relation で容易に表現できます。この制約式は、この関係にあるすべての変数が、異なる値を取るようにします。

$$\forall x \in XS, y : v_{xy} \in \{1,...,9\}$$
$$\forall x \in XS : all-different(v_{x1},...,v_{x9})$$
$$\forall y \in YS : all-different(v_{Ay},...,v_{Iy})$$
$$all-different(v_{A1},...,v_{C3})$$
$$all-different(v_{A4},...,v_{C6})$$
$$all-different(v_{A7},...,v_{C9})$$
$$all-different(v_{D1},...,v_{F3})$$
$$all-different(v_{D4},...,v_{F6})$$
$$all-different(v_{D7},...,v_{F9})$$
$$all-different(v_{G1},...,v_{I3})$$
$$all-different(v_{G4},...,v_{I6})$$
$$all-different(v_{G7},...,v_{I9})$$

そして、いくつかの変数 v_{xy} を、与えられている数に固定します。

3.3.2 インプリメンテーション

Table3.2 の Sudoku パズルの Mosel インプリメンテーションは、以下のようになります。

```
model "sudoku (CP)"
 uses "kalis"

 forward procedure print_solution(numsol: integer)

 setparam("default_lb", 1)
 setparam("default_ub", 9)              ! Default variable bounds

 declarations
  XS = {'A','B','C','D','E','F','G','H','I'}   ! Columns
  YS = 1..9                             ! Rows
  v: array(XS,YS) of cpvar              ! Number assigned to cell (x,y)
 end-declarations
```

```
! Data from "The Guardian", 29 July, 2005. http://www.guardian.co.uk/sudoku
 v('A',1)=8; v('F',1)=3
 v('B',2)=5; v('G',2)=4
 v('A',3)=2; v('E',3)=7; v('H',3)=6
 v('D',4)=1; v('I',4)=5
 v('C',5)=3; v('G',5)=9
 v('A',6)=6; v('F',6)=4
 v('B',7)=7; v('E',7)=2; v('I',7)=3
 v('C',8)=4; v('H',8)=1
 v('D',9)=9; v('I',9)=8


! All-different values in rows
 forall(y in YS) all_different(union(x in XS) {v(x,y)})

! All-different values in columns
 forall(x in XS) all_different(union(y in YS) {v(x,y)})

! All-different values in 3x3 squares
 forall(s in {{'A','B','C'},{'D','E','F'},{'G','H','I'}}, i in 0..2)
  all_different(union(x in s, y in {1+3*i,2+3*i,3+3*i}) {v(x,y)})

! Solve the problem
 solct:= 0
 while (cp_find_next_sol) do
  solct+=1
  print_solution(solct)
 end-do

 writeln("Number of solutions: ", solct)
 writeln("Time spent in enumeration: ", getparam("COMPUTATION_TIME"), "sec")
 writeln("Number of nodes: ", getparam("NODES"))

!****************************************************************
! Solution printing
 procedure print_solution(numsol: integer)
  writeln(getparam("COMPUTATION_TIME"), "sec: Solution ", numsol)
  writeln("   A B C   D E F   G H I")
  forall(y in YS) do
   write(y, ": ")
   forall(x in XS)
    write(getsol(v(x,y)), if(x in {'C','F'}, " | ", " "))
   writeln
   if y mod 3 = 0 then
    writeln("   ---------------------")
   end-if
  end-do
 end-procedure
```

```
end-model

!*****************************************************************
```

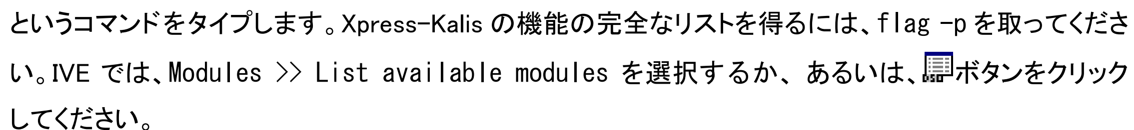
このモデルでは、cp_find_next_sol が、while ループに埋め込まれ、すべての実現可能な解をサーチします。すべてのループの実行で、見つかったソリューションを綺麗にフォーマットして印刷するために手順 print_solution がコールされます。Mosel のサブルーチンには、ローカルの宣言のために declarations blocks があるかもしれません。そして、それらは、いろいろなアーギュメントを取ります。このモデルでは、このままでは、この手順へのコールが定義の前に起こるので、キーワード forward を使い、最初のコールの前に宣言する必要があります。

サブルーチンによって印刷される情報を選択するために、Mosel の 2 つの異なるバージョンの if ステートメントを使います。すなわち、inline if と、一連のステートメントを含む if-then です。

モデルランの終わりに、ソルバーからランタイム統計(parameter COMPUTATION_TIME)と、サーチによって探索したノードの数(parameter NODES)をリトリーブします。

Xpress-Kalis タイプによって定義されたパラメタの全リストを得るには、

mosel -c "exam -p kalis"



というコマンドをタイプします。Xpress-Kalis の機能の完全なリストを得るには、flag -p を取ってください。IVE では、Modules >> List available modules を選択するか、あるいは、ボタンをクリックしてください。

3.3.3 結果

上のモデルは、以下を出力します。通常、数独パズルの解は一つしかありませんが、この問題の答も一つです。

```
0.16sec: Solution 1
   A B C   D E F   G H I
1: 8 6 9 | 2 4 3 | 1 5 7
2: 3 5 7 | 6 1 9 | 4 8 2
3: 2 4 1 | 8 7 5 | 3 6 9
   ---------------------
4: 4 9 8 | 1 3 2 | 6 7 5
5: 7 1 3 | 5 8 6 | 9 2 4
6: 6 2 5 | 7 9 4 | 8 3 1
   ---------------------
7: 1 7 6 | 4 2 8 | 5 9 3
8: 9 8 4 | 3 5 7 | 2 1 6
9: 5 3 2 | 9 6 1 | 7 4 8
   ---------------------
Number of solutions: 1
```

```
Time spent in enumeration: 0.41sec
Number of nodes: 2712
```

all_different 関係は、オプショナルな 2 番目のアーギュメントを取り、これにより、ユーザは、制約条件を評価するのに使われるプロパゲーションアルゴリズムを指定できます。ここで、例えば、2 番目のアーギュメントとして、ディフォルト設定(_Kalis FORWARD_CHECKING)を、もっと攻撃的な戦略 Kalis_GEN_ARC_CONSISTENCY に変更したとすると、

```
forall(y in YS)
 all_different(union(x in XS) {v(x,y)}, KALIS_GEN_ARC_CONSISTENCY)
```

ノードの数が、たったの一つのノードにまで減少するのが見られるでしょう。すなわち、問題が、単に制約条件をポストすることで解かれた、ということです。サーチに費やされた時間は、ゼロにまで減少しますが、制約条件をポストすることに費やさる時間は 4-5 倍になりますが、それでも、1 秒の何分の一です。この時間が永くなるのは、generalized arc consistency algorithm の計算オーバーヘッドが大きくなるからです。しかし、全体としてみると、問題の定義と、問題を解くのための時間は、前に較べ、10 分の 1 未満に短縮されます。

一般的に、generalized arc consistency algorithm は、より強い刈り込み(stronger pruning)を実現します。すなわち、このアルゴリズムは、変数のドメインから、より多くの値を取り除いてくれます。しかし、計算時間の増加のため、このアルゴリズムの使用は、いつも正当化されるというわけではありません。したがって、読者は、皆さんのモデルで、両方のアルゴリズム設定を試みるようにしてください。

3.4 abs と distance: 周波数割当

テレコミュニケーションの領域、特に、移動体通信は、周波数割当問題に関係する、興味深いさまざまな問題をもたらしました。

ここに、離散的な周波数帯要件を持つセル(ノード)のネットワークがあります。各セルには、多数の周波数(バンド)の既知の需要があります。Figure3.1 は、ネットワークの構造を示しています。「線(edge)」で結ばれたノードは、隣接する「隣人」であるとみなされます。それらのセルは、干渉を避けるために、同じ周波数を割り当てられません。その上、あるセルがいくつかの周波数を使用するなら、それらは少なくとも 2 だけ、異なっていなければなりません。ここでの目的は、ネットワークに使用される周波数の総数を最小にすることです。

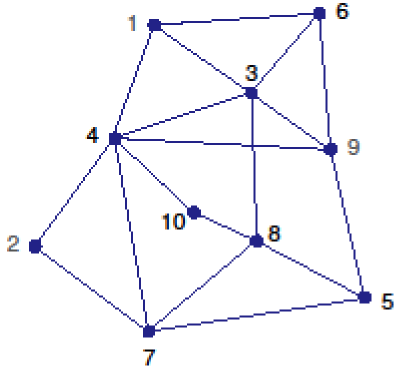


Figure 3.1: Telecommunications network

Table 3.3 は、個々のセルの周波数の数の需要を示しています。

Table 3.3: Frequency demands at nodes

Cell	1	2	3	4	5	6	7	8	9	10
Demand	4	5	2	3	2	4	3	4	3	2

3.4.1 Model の定式化

NODES がネットワークのすべてのノードの集合を、そして、DEM_n が $n \in$ *NODES* の周波数の需要を表すものとします。また、 ネットワークは、線 *LINKS* の集合として与えられます。さらに、*DEMANDS* = {1, 2, . . . , *NUMDEM*} が、周波数の集合を表すものとし、この集合では、すべてのノードは連続して付番されており、上限 *NUMDEM* が需要の総数によって与えられているとします。補助的な array $INDEX_n$ は、ノード n の *DEMANDS* の starting index を示すものとします。

あらゆる需要 $d \in DEMANDS$ に割り当てられる周波数を表すために、use_d を導入します。ここで、use_d は、set {1, 2, . . . , *NUMDEM*}.からその値を取ります。

こうして、2 つの集合の制約条件(隣接しているノードに割り当てられる異なる周波数、および、ノードの中の周波数間の最小の距離)は、以下のようにモデル化できます。

$$\forall (n,m) \in LINKS : all-different\left(\bigcup_{d=INDEZX_n}^{INDEX_n + DEM_n - 1} use_d \cup \bigcup_{d=INDEX_m}^{INDEX_m + DEM_m - 1} use_d \right)$$

$$\forall n \in NODES, \forall c < d \in INDEX_n, ..., INDEX_n + DEM_n - 1 : | use_c - use_d | \geq 2$$

目的関数は、使用される周波数の数を最小にすることです。ここでは、use_d 変数の最大の周波数の数を最小にすることで定式化します。

$$\text{minimize } maximum_{d \in DEMANDS}(use_d)$$

3.4.2 インプリメンテーション

テレコミュニケーション・ネットワークを形成する線は、リスト LINK としてモデル化されます。そこでは、「線 l (edge l)」は、(LINK(l,1),LINK(l,2))として与えられます。
同じノードに割り当てられる周波数の値に関しての制約条件のインプリメンテーションには、Kalis では、2 つの同等な選択があります。すなわち、abs 制約条件を使うか、もしくは、distance 制約条件を使うかです。

```
model "Frequency assignment"
 uses "kalis"

 forward procedure print_solution

 declarations
  NODES = 1..10                            ! Range of nodes
  LINKS = 1..18                            ! Range of links between nodes
  DEM: array(NODES) of integer             ! Demand of nodes
  LINK: array(LINKS,1..2) of integer       ! Neighboring nodes
   INDEX: array(NODES) of integer          ! Start index in 'use'
  NUMDEM: integer                          ! Upper bound on no. of freq.
 end-declarations

 DEM :: (1..10)[4, 5, 2, 3, 2, 4, 3, 4, 3, 2]
 LINK:: (1..18,1..2)[1, 3, 1, 4, 1, 6,
                     2, 4, 2, 7,
                     3, 4, 3, 6, 3, 8, 3, 9,
                     4, 7, 4, 9, 4,10,
                     5, 7, 5, 8, 5, 9,
                     6, 9, 7, 8, 8,10]
 NUMDEM:= sum(n in NODES) DEM(n)

! Correspondence of nodes and demand indices:
! use(d) d = 1, ..., DEM(1) correspond to the demands of node 1
!           d = DEM(1)+1, ..., DEM(1)+DEM(2))     - " -     node 2  etc.
 INDEX(1):= 1
 forall(n in NODES | n > 1) INDEX(n) := INDEX(n-1) + DEM(n-1)

 declarations
  DEMANDS = 1..NUMDEM                      ! Range of frequency demands
  use: array(DEMANDS) of cpvar             ! Frequency used for a demand
  numfreq: cpvar                           ! Number of frequencies used
  Strategy: array(range) of cpbranching
```

```
 end-declarations

! Setting the domain of the decision variables
 forall(d in DEMANDS) setdomain(use(d), 1, NUMDEM)

! All frequencies attached to a node must be different by at least 2
 forall(n in NODES, c,d in INDEX(n)..INDEX(n)+DEM(n)-1 | c<d)
  distance(use(c), use(d)) >= 2
!  abs(use(c) - use(d)) >= 2

! Neighboring nodes take all-different frequencies
 forall(l in LINKS)
  all_different(
   union(d in INDEX(LINK(l,1))..INDEX(LINK(l,1))+DEM(LINK(l,1))-1) {use(d)} +
   union(d in INDEX(LINK(l,2))..INDEX(LINK(l,2))+DEM(LINK(l,2))-1) {use(d)},
   KALIS_GEN_ARC_CONSISTENCY)

! Objective function: minimize the number of frequencies used, that is,
! minimize the largest value assigned to 'use'
 setname(numfreq, "NumFreq")
 numfreq = maximum(use)

! Search strategy
 Strategy(1):=assign_var(KALIS_SMALLEST_DOMAIN, KALIS_MIN_TO_MAX, use)
 Strategy(2):=assign_var(KALIS_MAX_DEGREE, KALIS_MIN_TO_MAX, use)
 cp_set_branching(Strategy(1))
 setparam("MAX_COMPUTATION_TIME", 1)
 cp_set_solution_callback("print_solution")

! Try to find solution(s) with strategy 1
 if (cp_minimize(numfreq)) then
  cp_show_stats
  sol:=getsol(numfreq)
 end-if

! Restart search with strategy 2
 cp_reset_search
 if sol>0 then                           ! If a solution was found:
  numfreq <= sol-1                       ! Add upper bound on objective
 end-if
 cp_set_branching(Strategy(2))
 setparam("MAX_COMPUTATION_TIME", 1000)

 if (cp_minimize(numfreq)) then
  cp_show_stats
 elif sol>0 then
  writeln("Optimality proven")
 else
  writeln("Problem has no solution")
 end-if
```

```
!****************************************************************
! **** Solution printout ****
 procedure print_solution
  writeln("Number of frequencies: ", getsol(numfreq))
  writeln("Frequency assignment: ")
  forall(n in NODES) do
   write("Node ", n, ": ")
   forall(d in INDEX(n)..INDEX(n)+DEM(n)-1) write(getsol(use(d)), " ")
   writeln
  end-do
 end-procedure

end-model
```

異なるサーチ戦略で実験してみると、variable selection criterion を KALIS_MAX_DEGREE に変えることで得られる戦略は、ひとたび、「良いソリューション」が見つけられると、最適性を容易に立証できるのがわかりました。したがって、この問題は以下の 2 ステップで解くことができます。まず、「良いソリューション」を見つけるために、ディフォルト戦略を使います。このサーチは、time limit を設定することで、1 秒後に止められます。そして、2 番目の戦略と、前のランで得られた目的関数の値に置いた bound を使い、再度、サーチを開始します。サーチを再開する前に、cp_reset_search を使って、ソルバーの中のサーチツリーをリセットする必要があります。

異なるサーチ戦略の実験を容易にするため、タイプ cpbranching の array Strategy を定義しました。この array は、異なるサーチ戦略定義をストアするためのものです。

このインプリメンテーションによって実証された新しいフィーチャーは、*callback* の使用、より正確には、Xpress-Kalis の *solution callback* です。solution callback は、サーチがソリューションを見つけたときは、いつでも、ソルバーがコールできるユーザサブルーチンで定義されます。その典型的な用途は、中間的なソリューションの logging や storing、もしくは、統計を取ることなどです。手順 print_solution は、単に、それまでに見つかったソリューションを印刷するに過ぎません。

問題の定式化を改善する：上の問題定式化では、あるノードのすべての demand 変数、および、これらの変数の制約条件が完全に左右対称であることに気づかれるかも知れません。他の制約条件がないので、同じセルに属す需要 d と $d+1$ に関して、下記を、変数 *use* に課すことで、これらの左右対称性を利用できるでしょう。

$$use_d + 1 \leq use_{d+1}$$

こうすることで、ディフォルト戦略を使って、問題は、40 以下のノードで最適解が得られました。これを、下記のように書くことで、さらに、もう一歩進めることも出来ます。

$$use_d + 2 \leq use_{d+1}$$

これらの制約条件の追加により、もう少し、ノード・サーチが短なります。これらの制約条件は、 abs

constraint や distance constraint の代わりに使うことさえできます。

3.4.3 結果

この問題の最適解は、11 の異なる周波数を使用します。 上記のプログラム・リストのモデルは、ノードに以下の周波数割当を印刷します。

```
Node 1: 1 3 5 7
Node 2: 1 3 5 7 10
Node 3: 2 8
Node 4: 4 6 9
Node 5: 4 6
Node 6: 4 6 9 11
Node 7: 2 8 11
Node 8: 1 3 5 7
Node 9: 1 3 5
Node 10: 2 8
```

3.5 element: 一つの機械でのジョブを順序付ける

このセクションで説明される問題は、「Applications of optimization with Xpress-MP」という本の Section 7.4、「Sequencing jobs on a bottleneck machine」から取られたものです。

この問題の目的は、単一の(ボトルネック)マシンの上で、異なる目的関数を与え、作成されるスケジュールがどのようになるかをみるためのモデルを作成することです。ここで、合計処理時間、平均処理時間、および、遅延の合計を最小にするにはどうしたらよいか、について考えましょう。

一組のタスク(または、ジョブ)が、単一の機械で処理されるとします。ここでの作業は、タスクの実行は、ひとたび開始されたら、中断されることなく、完成されるまで、継続的におこなわれるとします。Table3.4 に、すべてのタスク i に関して、そのリリース日付、タスク期間、納期が与えられています。

Table 3.4: Task time windows and durations

Cell	1	2	3	4	5	6	7
Release date	2	5	4	0	0	8	9
Duration	5	6	8	4	2	4	2
Due date	10	21	15	10	5	15	27

以下では、スケジュールの総期間(makespan)、平均処理時間、遅延時間合計を最少にすることを考え、それぞれの目的のための最適値はどのようなものになるかをみましょう。ここで、遅延時間合計とは、個々のジョブの完成が、それぞれの納期を越えてしまう時間の合計を意味します。

3.5.1 モデルの定式化 1

以下で、二つの代替的なモデルの定式化を説明します。最初の定式化は、「Applications of optimization with Xpress-MP」の中の数理計画法の定式化に近いものです。二番目の定式化は、disjunctive constraints と branching を使います。これら二つのモデル定式化では、順次、異なる目的関数を使いますが、モデル本体は同一です。

仕事のシークエンスを表すために、ここで、変数 $rank_k$ ($K \in JOBS\{1,...,NJ\}$)を導入します。あらゆるジョブ j は、ジョブのシークエンスのどかの、ただ一箇所だけに位置します。この制約条件は、変数 $rank_k$ の *all-different* で表すことができます。

$$all\text{-}different(rank_1, \ldots, rank_{NJ})$$

位置 k のジョブの処理時間 Dur_k は DUR_{rank_k} で与えられます。ここで、Dur_j は、前のセクションのテーブルで与えられたプロジェクト期間を表します。同様に、リリース時間 rel_k は、REL_{rank_k} で与えられます。ここで、REL_j は、リリース日付を表します。

$$\forall k \in JOBS : dur_k = DUR_{rank_k}$$
$$\forall k \in JOBS : rel_k = REL_{rank_k}$$

$start_k$ を位置 k のジョブの開始時刻であるとするなら、この値は、この位置に割り当てられたジョブのリリース日と少なくとも同じか、大きくなくてはなりません。このジョブの完成時刻 $comp_k$ は、その開始時刻と、そのジョブの処理時間の合計です。

$$\forall k \in JOBS : start_k \geq rel_k$$
$$\forall k \in JOBS : comp_k = start_k + dur_k$$

もう一つ、別の制約条件が必要です。それは、同時に 2 つの仕事を処理できない、という条件です。したがって、時間軸上の位置 $k+1$ のジョブは、ジョブ k が完了した位置以降に始まらなければなりません。この条件は、下のようになります。

$$\forall k \in \{1,...,NJ-1\} : start_{k+1} \geq start_k + dur_k$$

目的関数 1:目的関数 1 は、makespan(スケジュール全体の完成時間)を最小にすることです。これは、また、最後のジョブ(NJ 番目のジョブ)の完成時間を最小にすることと同等です。こうして、モデル全体は、下記のようになります。ここで、$MAXTIME$ は、例えば、すべてのリリース日とすべての処理時間の合計のような、十分に大きい値です。

$$\text{minimize} \quad comp_{NJ}$$
$$\forall k \in JOBS : rank_k \in JOBS$$
$$\forall k \in JOBS : start_k, comp_k \in \{0,...,MAXTIME\}$$

$$\forall k \in JOBS : dur_k \in \{\min_{j \in JOBS} DUR_j, ..., \max_{j \in JOBS} DUR_j\}$$
$$\forall k \in JOBS : rel_k \in \{\min_{j \in JOBS} REL_j, ..., \max_{j \in JOBS} REl_j\}$$
$$all - different(rank_1, ..., rank_{NJ})$$
$$\forall k \in JOBS : dur_k = DUR_{rank_k}$$
$$\forall k \in JOBS : rel_k = REL_{rank_k}$$
$$\forall k \in JOBS : start_k \geq rel_k$$
$$\forall k \in JOBS : comp_k = start_k + dur_k$$
$$\forall k \in \{1, ..., NJ - 1\} : start_{k+1} \geq start_k + dur_k$$

目的関数 2:平均処理時間を最小にするために、すべてのジョブの完成時間の合計を表す変数 totComp を追加します。そして、totComp を計算するために、問題に、以下の制約条件を追加します。

$$totComp = \sum_{k \in JOBS} comp_k$$

新しい目的関数は、平均処理時間を最小にすることです。これは、ジョブの処理時間を最小にすることと同等です。

minimize *totComp*

目的関数 3:

今度は、遅延の合計を最小にする、としましょう。ここでも、新しい変数を導入します。導入する変数は、納期の後にしか完了しない、ジョブの納期遅れの時間を計算する変数です。変数 $late_k$は、k番目のジョブの納期遅れの時間です。その値は、納期と完成時刻の差です。 ジョブが納期以前に終わる場合は、その値はゼロです。 こうして、以下の制約条件を得ます。

$$\forall k \in JOBS : dur_k = DUR_{rank_k}$$
$$\forall k \in JOBS : late_k \geq comp_k - due_k$$

新しい目的関数の定式化のために、すべてのジョブの総遅延を表す変数 *totLate* を導入します。 目的は、この変数の値を最小にすることです。

minimize totLate

$$totLate = \sum_{k \in JOBS} late_k$$

3.5.2 モデル 1 のインプリメンテーション

以下の Mosel インプリメンテーションは、それぞれ、異なる目的関数を使い、同じ問題を、3 回、解き、結果として得られるソリューションを、手順 print_sol および、print_sol をコールして、プリントします。

```
model "B-4 Sequencing (CP)"
 uses "kalis"

 forward procedure print_sol
 forward procedure print_sol3

 declarations
  NJ = 7                          ! Number of jobs
  JOBS=1..NJ

  REL: array(JOBS) of integer     ! Release dates of jobs
  DUR: array(JOBS) of integer     ! Durations of jobs
  DUE: array(JOBS) of integer     ! Due dates of jobs

  rank: array(JOBS) of cpvar      ! Number of job at position k
  start: array(JOBS) of cpvar     ! Start time of job at position k
  dur: array(JOBS) of cpvar       ! Duration of job at position k
  comp: array(JOBS) of cpvar      ! Completion time of job at position k
  rel: array(JOBS) of cpvar       ! Release date of job at position k
 end-declarations

 initializations from 'Data/b4seq.dat'
  DUR REL DUE
 end-initializations

 MAXTIME:= max(j in JOBS) REL(j) + sum(j in JOBS) DUR(j)
 MINDUR:= min(j in JOBS) DUR(j); MAXDUR:= max(j in JOBS) DUR(j)
 MINREL:= min(j in JOBS) REL(j); MAXREL:= max(j in JOBS) REL(j)

 forall(j in JOBS) do
  1 <= rank(j); rank(j) <= NJ
  0 <= start(j); start(j) <= MAXTIME
  MINDUR <= dur(j); dur(j) <= MAXDUR
  0 <= comp(j); comp(j) <= MAXTIME
  MINREL <= rel(j); rel(j) <= MAXREL
 end-do

! One posistion per job
 all_different(rank)

! Duration of job at position k
 forall(k in JOBS) dur(k) = element(DUR, rank(k))

! Release date of job at position k
 forall(k in JOBS) rel(k) = element(REL, rank(k))

! Sequence of jobs
 forall(k in 1..NJ-1) start(k+1) >= start(k) + dur(k)
```

```
! Start times
 forall(k in JOBS) start(k) >= rel(k)

! Completion times
 forall(k in JOBS) comp(k) = start(k) + dur(k)

! Set the branching strategy
 cp_set_branching(split_domain(KALIS_SMALLEST_DOMAIN, KALIS_MIN_TO_MAX))

!**** Objective function 1: minimize latest completion time ****
 if cp_minimize(comp(NJ)) then
  print_sol
 end-if


!**** Objective function 2: minimize average completion time ****
 declarations
  totComp: cpvar
 end-declarations

 totComp = sum(k in JOBS) comp(k)

 if cp_minimize(totComp) then
  print_sol
 end-if


!**** Objective function 3: minimize total tardiness ****
 declarations
  late: array(JOBS) of cpvar       ! Lateness of job at position k
  due: array(JOBS) of cpvar        ! Due date of job at position k
  totLate: cpvar
 end-declarations

 MINDUE:= min(k in JOBS) DUE(k); MAXDUE:= max(k in JOBS) DUE(k)

 forall(k in JOBS) do
  MINDUE <= due(k); due(k) <= MAXDUE
  0 <= late(k); late(k) <= MAXTIME
 end-do

! Due date of job at position k
 forall(k in JOBS) due(k) = element(DUE, rank(k))

! Late jobs: completion time exceeds the due date
 forall(k in JOBS) late(k) >= comp(k) - due(k)

 totLate = sum(k in JOBS) late(k)

 if cp_minimize(totLate) then
```

```
  writeln("Tardiness: ", getsol(totLate))
  print_sol
  print_sol3
 end-if

!-----------------------------------------------------------------

! Solution printing
 procedure print_sol
  writeln("Completion time: ", getsol(comp(NJ)) ,
          "  average: ", getsol(sum(k in JOBS) comp(k)))
  write("¥t")
  forall(k in JOBS) write(strfmt(getsol(rank(k)),4))
  write("¥nRel¥t")
  forall(k in JOBS) write(strfmt(getsol(rel(k)),4))
  write("¥nDur¥t")
  forall(k in JOBS) write(strfmt(getsol(dur(k)),4))
  write("¥nStart¥t")
  forall(k in JOBS) write(strfmt(getsol(start(k)),4))
  write("¥nEnd¥t")
  forall(k in JOBS) write(strfmt(getsol(comp(k)),4))
  writeln
 end-procedure

 procedure print_sol3
  write("Due¥t")
  forall(k in JOBS) write(strfmt(getsol(due(k)),4))
  write("¥nLate¥t")
  forall(k in JOBS) write(strfmt(getsol(late(k)),4))
  writeln
 end-procedure

end-model
```

注意： 読者は、なぜ、この例で、変数に名前を付けるとき、もっと、start-end が明白な名前を使わないのか、と不思議に思われるかもしれません。既にお分かりの読者の皆さんもいらっしゃるかも知れませんが、Mosel 言語では、end は、キーワードだからです。(Mosel language reference manual の reserved word のリストを参照)。これにより、end や END は、Mosel プログラムでキーワードともなされてしまう可能性があります。もっとも、End のように、大文字、小文字を組み合わせて使うバージョンも可能ですが、起こりえる混乱を回避するために、これらの変数名を使うことを推薦しません。

3.5.3 結果

スケジュールの最小 makespan は 31 で、最小の完成時間合計は 103 です(これは、平均 103/7=14.71 を与えます)。 この目的関数の値に対応するスケジュールは、ジョブ 5→ ジョブ 4→ジョブ 1→ ジョブ 7→ジョブ 6→ ジョブ 2→ ジョブ 3 です。また、完成時刻と納期を比較すると、ジョブ 1、ジョブ 2、ジョ

ブ 3、および、ジョブ 6 が納期に遅れ、遅れの合計は 21 です。

最小の遅延は 18 です。 この遅延を持つスケジュールはジョブ 5→ ジョブ 1→ ジョブ 4→ ジョブ 6→ ジョブ 2→ ジョブ 7→ ジョブ 3 で、ジョブ 4 とジョブ 7 が、1 タイム・ユニット遅くれ、そして、ジョブ 3 は、16 タイム・ユニット遅れます。このジョブ 3 は、納期である時刻 15 ではなく、時刻 31 に終わります。このスケジュールの平均完成時間は 15.71 です。

3.5.4 disjunction を使う代替的な定式化

2 番目のモデル定式化は、多分、問題を、より直接的に表現しているものでしょう。この定式化は、disjunctive constraint、および、これらの branching を使う方法での定式化です。

今度は、すべてのジョブを、開始時刻、すなわち、変数 $start_j (j \in JOBS = \{1,...,NJ\})$ によって表し、これらは、{*RELj,...,MAXTIME*}の中の値を取ります。ここで、*MAXTIME* は、例えば、すべてのリリース日とすべての処理時間の合計のような、十分に大きい値です。ここでは、下記のように、disjunction を、すべてのジョブの開始時刻と処理時間に関する単一の disjunctive relation として表現します。

$$disjunctive([start_1,...,start_{NJ}],[DUR_1,...,DUR_{NJ}])$$

この制約条件は、すべてのジョブの組合せ $i < j \in JOBS$ に関して、下記の pair-wise disjunction を置換します。

$$start_i + DUR_i \le start_j \vee start_j + DUR_j \le start_i$$

位置 k のジョブの処理時間 dur_k は、DUR_{rank_k} によって与えられます。ここで、DUR_j は、前のセクションのテーブルで与えられた処理時間を意味します。同様に、リリース日 rel_k は、RELrankk(RELj によって与えられます。ここで、REL_{rank_k} は、所与のリリース日を意味します。

$$\forall k \in JOBS : dur_k = DUR_{rank_k}$$
$$\forall k \in JOBS : rel_k = REL_{rank_k}$$

ジョブ j の完成時刻 $comp_j$ は、開始時刻と処理時間 DUR_j の合計です。

$$\forall k \in JOBS : comp_j = start_j + DUR_{j_k}$$

目的関数 1: 一番目の目的関数は、makespan(スケジュール全体の完成時刻)を最小にすること、言い換えると、最後のジョブの完成時刻を最も早くすることです。下は、このモデルの全体です。ここで、*MAXTIME* は、例えば、すべてのリリース日とすべての処理時間の合計のような、十分に大きい値です。

minimize *finish*

$$finish = \max imum_{j \in JOBS}(comp_j)$$
$$\forall k \in JOBS : comp_j \in \{0,...,MAXTIME\}$$
$$\forall k \in JOBS : start_j \in \{REL_j,...,MAXTIME\}$$
$$disjunctive([start_{1,...,}start_{NJ}],[DUR_1,...,DUR_{NJ}])$$
$$\forall j \in JOBS : comp_j = start_j + DUR_j$$

目的関数 2: 二番目の目的関数は、最初のモデルの目的関数と同じで、平均処理時間を最小にするか、または、それと同等ですが、ジョブの処理時間を最小にすることです。ここで、すべてのジョブの完成時間の合計を表す変数 totComp を導入します。

minimize *totComp*

$$totComp = \sum_{k \in JOBS} comp_k$$

目的関数 3:遅延の合計を最小にする目的関数を定式化するために、新しい変数 $late_j$ を導入します。これは、ジョブが期日までに終わらないで遅延するとき、その時間に測定するためのものです。これらの変数の値は、ジョブ j の完成日と納期 DUE_j の差を示すものです。ジョブが、納期以前終わる場合、その値はゼロです。目的は、これらの遅延変数の合計を最小にすることです。

Minimize *totLate*

$$totLate = \sum_{j \in JOBS} late_j$$
$$\forall j \in JOBS : late_j \in \{0,...,MAXTIME\}$$
$$\forall j \in JOBS : late_j \geq comp_j - DUE_j$$

3.5.5 モデル 2 のインプリメンテーション

モデル 1 と同じように、Mosel インプリメンテーションは、それぞれ、異なる目的関数を使い、同じ問題を、3 回、解き、結果として得られるソリューションを、手順 print_sol および、print_sol をコールして、プリントします。

```
model "B-4 Sequencing (CP)"
 uses "kalis"

 forward procedure print_sol
 forward procedure print_sol3

 declarations
  NJ = 7                              ! Number of jobs
  JOBS=1..NJ
```

```
  REL: array(JOBS) of integer     ! Release dates of jobs
  DUR: array(JOBS) of integer     ! Durations of jobs
  DUE: array(JOBS) of integer     ! Due dates of jobs
  DURS: array(set of cpvar) of integer  ! Dur.s indexed by start variables

  start: array(JOBS) of cpvar     ! Start time of jobs
  comp: array(JOBS) of cpvar      ! Completion time of jobs
  finish: cpvar                   ! Completion time of the entire schedule
  Disj: set of cpctr              ! Disjunction constraints
  Strategy: array(range) of cpbranching  ! Branching strategy
 end-declarations

 initializations from 'Data/b4seq.dat'
  DUR REL DUE
 end-initializations

 MAXTIME:= max(j in JOBS) REL(j) + sum(j in JOBS) DUR(j)

 forall(j in JOBS) do
  0 <= start(j); start(j) <= MAXTIME
  0 <= comp(j); comp(j) <= MAXTIME
 end-do

! Disjunctions between jobs
 forall(j in JOBS) DURS(start(j)):= DUR(j)
 disjunctive(union(j in JOBS) {start(j)}, DURS, Disj, 1)

! Start times
 forall(j in JOBS) start(j) >= REL(j)

! Completion times
 forall(j in JOBS) comp(j) = start(j) + DUR(j)

!**** Objective function 1: minimize latest completion time ****
 finish = maximum(comp)

 Strategy(1):= settle_disjunction(Disj)
 Strategy(2):= split_domain(KALIS_LARGEST_MAX, KALIS_MIN_TO_MAX)
 cp_set_branching(Strategy)

 if cp_minimize(finish) then
  print_sol
 end-if


!**** Objective function 2: minimize average completion time ****
 declarations
  totComp: cpvar
 end-declarations
```

```
 totComp = sum(k in JOBS) comp(k)

 if cp_minimize(totComp) then
  print_sol
 end-if


!**** Objective function 3: minimize total tardiness ****
 declarations
  late: array(JOBS) of cpvar      ! Lateness of jobs
  totLate: cpvar
 end-declarations

 forall(k in JOBS) do
  0 <= late(k); late(k) <= MAXTIME
 end-do

! Late jobs: completion time exceeds the due date
 forall(j in JOBS) late(j) >= comp(j) - DUE(j)

 totLate = sum(k in JOBS) late(k)
 if cp_minimize(totLate) then
  writeln("Tardiness: ", getsol(totLate))
  print_sol
  print_sol3
 end-if

!-----------------------------------------------------------------

! Solution printing
 procedure print_sol
  writeln("Completion time: ", getsol(finish) ,
          "  average: ", getsol(sum(j in JOBS) comp(j)))
  write("Rel¥t")
  forall(j in JOBS) write(strfmt(REL(j),4))
  write("¥nDur¥t")
  forall(j in JOBS) write(strfmt(DUR(j),4))
  write("¥nStart¥t")
  forall(j in JOBS) write(strfmt(getsol(start(j)),4))
  write("¥nEnd¥t")
  forall(j in JOBS) write(strfmt(getsol(comp(j)),4))
  writeln
 end-procedure

 procedure print_sol3
  write("Due¥t")
  forall(j in JOBS) write(strfmt(DUE(j),4))
  write("¥nLate¥t")
  forall(j in JOBS) write(strfmt(getsol(late(j)),4))
  writeln
```

```
end-procedure

end-model
```

このインプリメンテーションでは、settle_disjunction という新しい branching scheme を導入しました。これまで見てきた branching 戦略とは異なり、この branching scheme は、変数ではなく、制約式についての branching 戦略を定義します。この scheme により、与えられた集合から制約条件を選らんでノードを作成し、そのノードからの branch は、この disjunctive constraint を形成する、互いに排他的な制約条件の一つを、この constraint system に加えることで得られます。

注意： Xpress-Kalis の disjunctive constraint により、タスクの処理時間の間の pair-wise inequalities が確立されます。しかし、disjunctive constraint の定義は、このケースにとどまりません。disjunction は、2 つ以上のコンポーネントも持つことができ、どんなタイプの制約条件でも構いません。これには、'and' や 'or' によって制約条件を結合することで得られる他のロジック関係も含くまれています。

3.6 occurrence: 砂糖の生産 gar production

このセクションの問題は、「Applications of optimization with Xpress-MP」という本の Section 6.4「Cane sugar production」から取ったものです。

オーストラリアでの甘蔗糖の収穫は、非常に機械化されています。 サトウキビはすぐに、狭軌道のワゴンで、製糖所に輸送されます。

ワゴン 1 台分の積荷のサトウキビから取れる砂糖は、収穫された畠とサトウキビの成熟度合いに依存しています。ひとたび、収穫されると、糖分は、発酵により急速に減少してゆきます。したがって、そして、ワゴン 1 台分の積み荷は、ある時間が経つと、その価値が完全になくなります。いま、同じ量を積んだ 11 台のワゴンが製糖所に到着しました。これらのワゴンは、ワゴンごとに、1 時間経過ごとの糖分の損失と、時間数で計られる「残っている寿命」を見定めるために調べられました。以下のテーブルは、これらのデータをまとめたものです。

Table 3.5: Properties of the lots of cane sugar

Lot	1	2	3	4	5	6	7	8	9	10	11
Loss (kg/h)	43	26	37	28	13	54	62	49	19	28	30
Life span (h)	8	8	2	8	4	8	8	8	8	8	8

これらの 11 のロットは、すべて、三本ある、まったく同等の製造ラインによって処理されます。 一つのロット処理するには 2 時間かかります。そして、処理は、遅くとも、ワゴンの積み荷の寿命が終わるまでに終了していなければなりません。
製糖所のマネジャーは、現在、手許にある、処理できるロットを、どのようなスケジュールで処理したら、

砂糖のロスを最小にすることができるかを知りたいと思っています。

3.6.1 モデルの定式化

WAGONS={1, …, *NW*} でワゴンの集合を、*NL* で生産ラインの数を、*DUR* で、すべてのロットの処理時間を表すものとします。そして、各ワゴン *w* の、1 時間ごとのロスは $LOSS_w$、そして、寿命は $LIFE_w$ だとします。最適解では、製造ラインは中断なしで動くことが必要です。なぜなら、中断しなければ、ロットの始まりを遅らせなくてすむので、それだけ、砂糖のロスを抑えることができるからです。これは、あらゆるロットの処理の完了時間が *s*DUR* という形式のものであることを意味します。ここで、$s > 0$ であり、整数です。s の取る最大の値は、製糖所が稼動する、長さ *DUR* の時間帯の数、すなわち、*NS = ceil(NW/NL)* です。ここで、*ceil* は、「rounded to the next largest integer」を意味します。もし、*NW/NL* が整数であるなら、ラインは、すべて、*NS* ロットだけ処理します。そうでない場合は、いくつかのラインは、*NS-1* ロットを処理することになりますが、少なくとも 1 つのラインは *NS* ロットを処理します。 すべての場合で、最適スケジュールの長さは *NS*DUR* 時間です。ここでは、*SLOTS={1,...,NS}* を「時間帯の集合(set of time slots)」と呼びましょう。

ロットは、すべて、どこかの時間帯に割り当てられなければなりません。ここで、ワゴン *w* に割り当てられた時間帯を表す変数 $process_w$ を、このワゴンで発生するロスを表す変数 $loss_w$ を定義します。他方、NL 本の並列ラインがあるので、製糖所が稼動中の各時間帯には、すべて、最大 NL 個のロットを割り当てられます。したがって、$process_w$ 変数で、時間帯での出現回数を制限します(this constraint relation is often called cardinality constraint)。

$$s \in SLOTS : | process_w = s |_{w \in WAGONS} \leq NL$$

時間帯 *s* で処理されるロットのワゴン一台分の砂糖のロス *w* は、$COST_{ws} = s * DUR * LOSS_w$ です。ここで、変数 $loss_w$ でワゴン *w* に積まれているロットで発生するロスを示すものとします。

$$\forall w \in WAGONS : loss_w = \cos t_{w, process_w}$$

目的関数（砂糖のロス合計）は、すべてのロスの合計となります。

$$\text{minimize} \sum_{w \in WAGONS} loss_w$$

3.6.2 インプリメンテーション

以下のモデルは、この問題の Mosel インプリメンテーションです。このモデルは、時間帯の最大数の計算に function ceil を使っています。

The constraints on the processing variables are expressed by occurrence relations and the losses are obtained via element constraints. The branching strategy uses the variable selection criterion KALIS_SMALLEST_MAX, that is, choosing the variable with the smallest upper bound.

処理の変数についての制約条件は、occurrence relations で表現されています。そして、ロスは element constraint を使って表現されています。branching strategy は、変数選択基準 KALIS_SMALLEST_MAX を使っています。KALIS_SMALLEST_MAX は、最も小さい upper bound を選択します。

```
model "A-4 Cane sugar production (CP)"
 uses "kalis"

 declarations
  NW = 11                             ! Number of wagon loads of sugar
  NL = 3                              ! Number of production lines
  WAGONS = 1..NW
  NS = ceil(NW/NL)
  SLOTS = 1..NS                       ! Time slots for production

  LOSS: array(WAGONS) of integer      ! Loss in kg/hour
  LIFE: array(WAGONS) of integer      ! Remaining time per lot (in hours)
  DUR: integer                        ! Duration of the production (in hours)
  COST: array(SLOTS) of integer       ! Cost per wagon

  loss: array(WAGONS) of cpvar        ! Loss per wagon
  process: array(WAGONS) of cpvar     ! Time slots for wagon loads

  totalLoss: cpvar                    ! Objective variable
 end-declarations

 initializations from 'Data/a4sugar.dat'
  LOSS LIFE DUR
 end-initializations

 forall(w in WAGONS) setdomain(process(w), 1, NS)

! Wagon loads per time slot
 forall(s in SLOTS) occurrence(s, process) <= NL

! Limit on raw product life
 forall(w in WAGONS) process(w) <= floor(LIFE(w)/DUR)

! Objective function: total loss
 forall(w in WAGONS) do
  forall(s in SLOTS) COST(s):= s*DUR*LOSS(w)
  loss(w) = element(COST, process(w))
 end-do
```

```
 totalLoss = sum(w in WAGONS) loss(w)

 cp_set_branching(assign_var(KALIS_SMALLEST_MAX, KALIS_MIN_TO_MAX, process))

! Solve the problem
 if not (cp_minimize(totalLoss)) then
  writeln("No solution found")
  exit(0)
 end-if

! Solution printing
 writeln("Total loss: ", getsol(totalLoss))
 forall(s in SLOTS) do
  write("Slot ", s, ": ")
  forall(w in WAGONS)
   if(getsol(process(w))=s) then
    write("wagon ", strfmt(w,2), strfmt(" (" + s*DUR*LOSS(w) + ")  ", 8))
   end-if
  writeln
 end-do

end-model
```

処理変数についての制約条件の代替的な定式化は、それらを、単一の distribute relation で置き換えることです。ここでの distribute relation は、すべての時間帯で、ラインが使える最少、最大の数($MINUSE_s$=0 and $MAXUSE_s$=NL)を知らせるのに使われます。

```
forall(s in SLOTS) MAXUSE(s):= NL
distribute(process, SLOTS, MINUSE, MAXUSE)
```

Xpress-Kalis では、この問題を、さらにもう一つ、別の形で定式化することもできます。すなわち、この問題を cumulative scheduling problem(see Section 5.4)と解釈することです。そこでは、ワゴンに積まれたロットは、キャパシティが生産ラインに対応する discrete resource 上にスケジュールされる tasks of unit duration で表されると考えます。

3.6.3 結果

ロスの合計は、1620 kg の砂糖です。それに対応するスケジュールは、下の Table 3.6 に示されています。

Table 3.6: Optimal schedule for the cane sugar lots

Slot 1	Slot 2	Slot 3	Slot 4
lot 3 (74 kg)	lot 1 (172 kg)	lot 4 (168 kg)	lot 2 (208 kg)
lot 6 (108 kg)	lot 5 (52 kg)	lot 9 (114 kg)	lot 10 (224 kg)

lot 7 (124 kg)　　lot 8 (196 kg)　　lot 11 (180 kg)

3.7 distribute: 人員配置

映画館の経営者は、従業員の働く場所のプランを立てたいと思っています。劇場には、デヴィッド、アンドリュー、レスリー、ジェイソン、オリバー、マイケル、ジェーン、および、マリリンという 8 人の従業員がいます。切符売場は、3 人の配置が、劇場の入口 1 と入口 2 は、それぞれ、2 人の配置がが必要です。さらに、1 人をクロークにも配置しなければなりません。従業員のスキルや個々人の好みも考えなければならないので、以下の条件を考慮する必要があります。

1. レスリーは、劇場の 2 番目の入り口に配置しなければならない。
2. マイケルは、劇場の 1 番目の入り口に配置しなければならない。
3. デヴィッド、マイケル、および、ジェイソンは、お互いに、一緒に働けない。
4. オリバーをチケット売場に配置すると、マリリンも、そこに配置しなければならない。

3.7.1 Model formulation

PERS を人員の集合とし、LOC = {1, . . . , 4} を働く位置の集合とします。ここで、1 は切符売場、2 は入口 1、3 はは入口 3、4 はクロークを表すとします。

ここで、従業員 p に割り当てられる場所を示す決定変数 $place_p$ を導入します。こうして、以下のように、働く位置についての 4 つの制約条件を定式化できます。

$$place_{Leslie} = 3$$
$$place_{Michael} = 2$$
$$all-different(place_{David}, place_{Michael}, place_{Jason})$$
$$place_{Oliver} = 1 \Rightarrow place_{Marylin} = 1$$

また、働く位置、すべての配置人員の条件も満たさなければなりません。

$$\forall l \in: \left| place_p = l \right|_{p \in PERS} = REQ_l$$

3.7.2 インプリメンテーション

この配置問題のインプリメンテーションには、Xpress-Kalis の持つ、2 つの異なる constraint から一つを選べます。すなわち、(配置場所ごとに一つの)occurrence constraint、もしくは、(global cardinality constraint として知られている) distribute constraint です。後者は、すべての職場を対象とするものです。以下のモデルは、これらの両方のオプションを示します。セクション

3.6.2 の例とは対照的に、このモデルには、equality constraint があります。したがって、ここでは、4 つのアーギュメントを持つ distribute ではなく、3 つのアーギュメントの distribute バージョンを使います。4つのアーギュメントを持つdistributeでは、最後の2のアーギュメントは、lower bound、upper bound です。

このモデルは、attractive display のために、集合 LOC の中の配置場所の名前、および、それに対応するインデックス値についての補助的なデータ構造も導入してあります。

```
model "Personnel Planning (CP)"
 uses "kalis"

 forward procedure print_solution

 declarations
  PERS = {"David","Andrew","Leslie","Jason","Oliver","Michael",
          "Jane","Marilyn"}                ! Set of personnel
  LOC = 1..4                               ! Set of locations
  LOCNAMES = {"Ticketoffice", "Theater1", "Theater2",
              "Cloakroom"}                 ! Names of locations
  LOCNUM: array(LOCNAMES) of integer       ! Numbers assoc. with loc.s
  REQ: array(LOC) of integer               ! No. of pers. req. per loc.

  place: array(PERS) of cpvar              ! Workplace assigned to each peson
 end-declarations

! Initialize data
 LOCNUM("Ticketoffice"):= 1; LOCNUM("Theater1"):= 2
 LOCNUM("Theater2"):= 3; LOCNUM("Cloakroom"):= 4
 REQ:: (1..4)[3, 2, 2, 1]

! Each variable has a lower bound of 1 (Ticketoffice) and an upper bound
! of 4 (Cloakroom)
 forall(p in PERS) do
  setname(place(p),"workplace["+p+"]")
  setdomain(place(p), LOC)
 end-do

! "Leslie must be at the second entrance of the theater"
 place("Leslie") = LOCNUM("Theater2")

! "Michael must be at the first entrance of the theater"
 place("Michael") = LOCNUM("Theater1")

! "David, Michael and Jason cannot work with each other"
 all_different({place("David"), place("Michael"), place("Jason")})

! "If Oliver is selling tickets, Marylin must be with him"
 implies(place("Oliver")=LOCNUM("Ticketoffice"),
```

```
     place("Marilyn")=LOCNUM("Ticketoffice"))

 ! Creation of a resource constraint of for every location
 ! forall(d in LOC) occurrence(LOCNUM(d), place) = REQ(d)

 ! Formulation of resource constraints using global cardinality constraint
  distribute(place, LOC, REQ)

 ! Setting parameters of the enumeration
  cp_set_branching(assign_var(KALIS_SMALLEST_MIN, KALIS_MAX_TO_MIN, place))

 ! Solve the problem
  if not(cp_find_next_sol) then
   writeln("Problem is infeasible")
   exit(1)
  end-if

 ! Solution output
  nbSolutions:= 1
  print_solution

 ! Search for other solutions
  while (cp_find_next_sol) do
   nbSolutions += 1
   print_solution
  end-do

 ! **** Solution printout ****
  procedure print_solution
   declarations
    LOCIDX: array(LOC) of string
   end-declarations
   forall(l in LOCNAMES) LOCIDX(LOCNUM(l)):=l

   writeln("¥nSolution number ", nbSolutions)
   forall(p in PERS)
    writeln(" Working place of ", p, ": ", LOCIDX(getsol(place(p))))
  end-procedure

end-model
```

ここでは、単に、一つの実行可能なソリューションを見つけたいので、このモデルは、最初に、実行可能なソリューションの有無をテストします。実行可能なソリューションがある場合は、モデルは、見つかったソリューション以外の、他のすべての実行可能なソリューションを列挙します。 ソリューションが見つかるたびに、procedure print_solution をコールして、それをプリントします。

3.7.3 結果

この問題には、38 の実現可能なソリューションがあります。 実行可能なソリューションを、IVE を使ってグラフ表示させると、Figure3.2 のようになります。

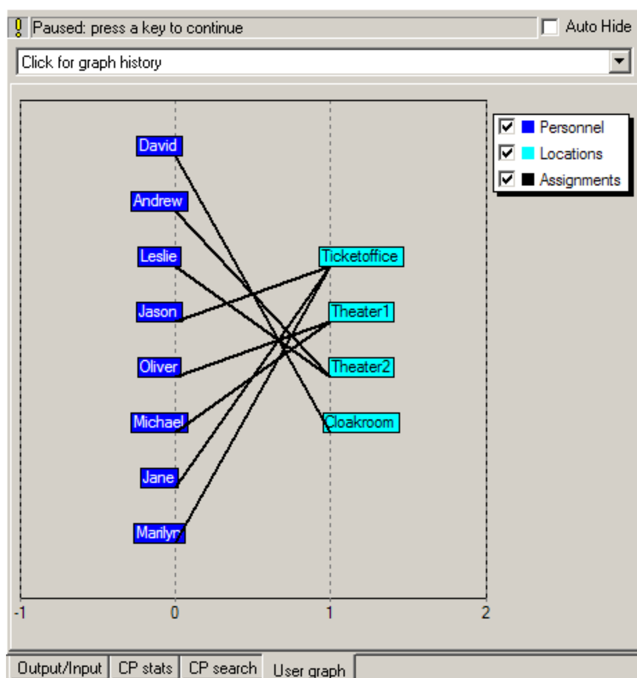


Figure 3.2: Personnel schedule representation in IVE

このグラフィック表示は、以下の Mosel コードを使って表示させたものです。詳しくは、Mosel language reference manual のモジュール mmive についての説明を参照してください。

```
procedure draw_solution
 IVEerase
 PersGraph:= IVEaddplot("Personnel", IVE_BLUE)
 LocGraph:= IVEaddplot("Locations", IVE_CYAN)
 AsgnGraph:= IVEaddplot("Assignments", IVE_BLACK)

 forall (d in LOCNAMES)
  IVEdrawlabel(LocGraph, 1.2, LOCNUM(d)-getsize(LOC)/2, d)

 idx:= 1
 forall(p in PERS) do
  IVEdrawline(AsgnGraph, 0, idx-getsize(PERS)/2,
              1, getsol(place(p))-getsize(LOC)/2)
  IVEdrawlabel(PersGraph, -0.1, idx-getsize(PERS)/2, p)
  idx:= idx + 1
 end-do
```

```
  IVEzoom(-1, getsize(PERS)/2+1, 2, -getsize(PERS)/2)
  IVEpause("press a key to continue")
end-procedure
```

3.8 implies ： ペイントの生産

このセクションの問題は、「Applications of optimization with Xpress-MP」という本の Section 7.5 にある「Paint production」から取ったものです。

毎週の生産の一部として、あるペイント会社は、安定した需要を持っている大きいクライアントに納入するため、このペイントの 5 つのバッチを生産しています。この 5 つのバッチは、いつも同じです。ペイントのバッチは、すべて、単一の、同じ生産プロセスで生産されます。生産に使う混合槽は、あるバッチから次のバッチに移るとき、必ず、洗浄されなければなりません。ペイントの 5 つの混合バッチ 1～5 の所要時間は、それぞれ 40 分、35 分、45 分、32 分、および 50 分です。 洗浄時間は、色とペイントのタイプに依存します。例えば、油ベースのペイントを水性ペイントの後に生産したり、暗い色のペイントの生産の後に、白いペイントを生産したりするような場合、洗浄時間は長くかかります。Table 3.7: Matrix of cleaning times の数字は、分単位で、$CLEAN_{ij}$ は、バッチ i とバッチ j との間の洗浄時間です。

Table 3.7: Matrix of cleaning times

	1	2	3	4	5
1	0	11	7	13	11
2	5	0	13	15	15
3	13	15	0	23	11
4	9	13	5	0	3
5	3	7	7	7	0

この会社は、他の仕事も持っているので、毎週行う、この生産活動(戦場 TOC 調合)を、できるだけ短い時間で済ませたいと考えています。そうするためには、ペイントバッチを、どのような順序で行ったらよいでしょうか。この順序は、毎週、行われるので、総処理時間を最短にするには、ある週の最後のバッチと次の週の最初のバッチの洗浄時間も考慮する必要があります。

3.8.1 モデル 1 の定式化

ここでは、Section 3.5 の問題を、2 つの代替的なモデルとして定式化しましょう。最初の定式化は、「Applications of optimization with Xpress-MP」の数理計画法による定式化に近いもので、二

番目の定式化は、two-dimensional element constraint を使用する定式化です。

$JOBS = \{1,...,NJ\}$で生産するバッチの集合を、DUR_jでバッチ j の処理時間を、$CLEAN_{ij}$で二つの連続したバッチ i とバッチ j の間の洗浄時間を示すものとします。また、各ジョブの後続ジョブを示すため、$JOBS$ の中の値を取る決定変数 $succ_j$ を導入し、各ジョブの後の洗浄に必要な時間を示す $clean_j$ を導入します。各ジョブの後の洗浄に必要な時間は、$succ_j$ の値でインデックスして得ます。 こうして、以下のように、問題の定式化を行います。

$$\text{Minimize} \quad \sum_{j \in JOBS} \left(DUR_j + clean_j \right)$$
$$\forall j \in JOBS : succ_j JOBS \setminus \{j\}$$
$$\forall j \in JOBS : clean_j = CLEAN_{j,succ_j}$$
$$all - different\left(\Upsilon_{j \in JOBS}\ succ_j \right)$$

目的関数は、すべてのバッチの処理時間と洗浄時間をまとめます。最後のall-different制約条件は、あらゆるバッチが、生産順序の中で一回しか起こらないことを保証します。

残念ながら、このモデルは、ソリューションが一つのサイクルを形成することを保証しません。実際、モデルを解くと、総作業時間239が得られますが、その中には、2つのサブサイクル1→ 3 → 2 → 1とサブサイクル4 → 5 → 4があり、ソリューションとして、適切ではありません。これを解決するための一つの方法は、モデルにdisjunctionを加え、サブサイクルをもたないソリューションを得るまで、繰り返し、モデルを解くことです。

$$\forall succ_1 \neq 3 \vee succ_3 \neq 2 \vee succ_2 \neq 1 \vee succ_1 \neq 5 \vee succ_5 \neq 4$$

しかし、この手順は、潜在的に非常に多くの disjunction を導入する可能性があるので、より大きいデータセットでは非実用的でしょう。したがって、サブサイクル除去のために、下記のように、変数 y_jと implication constraint を使い、先験的な定式化の方式を選びます。

$$\forall j \in JOBS : rank_j \in \{1,...,NJ\}$$
$$\forall i \in JOBS, \forall j = 2,...,NJ, i \neq j : succ_i = j \Rightarrow y_j = y_i + 1$$

変数 y_jは、生産サイクルの中でのジョブ j の位置に対応します。これらの制約条件で、ジョブ 1 は、いつも第 1 ポジションを取ります。

3.8.2 モデル 1 のインプリメンテーション

前のセクションで定式化されたモデルのMoselインプリメンテーションは、かなり簡単です。サブサイクル除去の制約条件は、impliesを持つロジック関係として実行されます。(equivを使い、implicationをequivalenceに置き換えることで、これらの制約条件の、より強い定式化が得られます。)

```
model "B-5 Paint production (CP)"
 uses "kalis"

 declarations
  NJ = 5                            ! Number of paint batches (=jobs)
  JOBS=1..NJ

  DUR: array(JOBS) of integer       ! Durations of jobs
  CLEAN: array(JOBS,JOBS) of integer ! Cleaning times between jobs
  CB: array(JOBS) of integer        ! Cleaning times after a batch

  succ: array(JOBS) of cpvar        ! Successor of a batch
  clean: array(JOBS) of cpvar       ! Cleaning time after batches
  y: array(JOBS) of cpvar           ! Variables for excluding subtours
  cycleTime: cpvar                  ! Objective variable
 end-declarations

 initializations from 'Data/b5paint.dat'
  DUR CLEAN
 end-initializations

 forall(j in JOBS) do
  1 <= succ(j); succ(j) <= NJ; succ(j) <> j
  1 <= y(j); y(j) <= NJ
 end-do

! Cleaning time after every batch
 forall(j in JOBS) do
  forall(i in JOBS) CB(i):= CLEAN(j,i)
  clean(j) = element(CB, succ(j))
 end-do

! Objective: minimize the duration of a production cycle
 cycleTime = sum(j in JOBS) (DUR(j)+clean(j))

! One successor and one predecessor per batch
 all_different(succ)

! Exclude subtours
 forall(i in JOBS, j in 2..NJ | i<>j)
  implies(succ(i) = j, y(j) = y(i) + 1)

! Solve the problem
 if not cp_minimize(cycleTime) then
  writeln("Problem is infeasible")
  exit(1)
 end-if
 cp_show_stats

! Solution printing
```

```
writeln("Minimum cycle time: ", getsol(cycleTime))
writeln("Sequence of batches:¥nBatch Duration Cleaning")
first:=1
repeat
 writeln("  ", first, strfmt(DUR(first),8), strfmt(getsol(clean(first)),9))
 first:=getsol(succ(first))
until (first=1)

end-model
```

3.8.3 モデル 2 の定式化

Section 3.5.1 の順序付けモデルに似た rank variable を使って、このペイント生産の問題を実行することもできます。

前と同様、$JOBS=\{1,...,NJ\}$が生産バッチの集合、DUR_jがバッチ j の処理時間、$CLEAN_{ij}$が連続したバッチ i とバッチ j の間の洗浄時間を示すとします。また、位置 k にある job の数値として、$JOBS$ の中の値を取る決定変数 $rank_k$を導入します。変数 $clean_k$ (ここで、$k\in JOBS$) は、k 番目の洗浄時間を示します。この洗浄時間は、2 の連続した変数 $rank_k$ の値で $CLEAN_{ij}$をインデクシングして得られます。こうして、以下の問題の定式化を得ます。

$$\text{Minimize} \quad \sum_{j\in JOBS} DUR_j + \sum_{k\in JOBS} clean_k$$
$$\forall k \in JOBS : rank_k \in JOBS$$
$$\forall k \in \{1,...,NJ-1\} : clean_k = CLEAN_{rank_k, rank_{k+1}}$$
$$clean_{NJ} = CLEAN_{rank_{NJ}, rank_1}$$
$$all-different\left(\Upsilon_{k\in JOBS}\ rank_k\right)$$

モデル 1 と同様、目的関数は、すべてのバッチの処理と洗浄時間の合計の最小化です。数学的観点からは厳密には必要ではありませんが、処理時間と洗浄時間をそれぞれ別個に、異なる集計インデックで集計しています。これにより、ジョブに関する集計と、ジョブの位置に関する集計との違いを示します。

このモデルでは、rank variable に関する all-different constraint を組み込み、すべてのバッチが、生産順序の中に、厳密に、ただ一度だけしか入らないようにしてあります。

3.8.4 2 のインプリメンテーション

二番目のモデルのインプリメンテーションは、Xpress-Kalis の 2-dimensional version of the element constraint を使用しています。

```
model "B-5 Paint production (CP)"
```

```
uses "kalis"

declarations
 NJ = 5                          ! Number of paint batches (=jobs)
 JOBS=1..NJ

 DUR: array(JOBS) of integer       ! Durations of jobs
 CLEAN: array(JOBS,JOBS) of integer ! Cleaning times between jobs

 rank: array(JOBS) of cpvar        ! Number of job in position k
 clean: array(JOBS) of cpvar       ! Cleaning time after batches
 cycleTime: cpvar                  ! Objective variable
end-declarations

initializations from 'Data/b5paint.dat'
 DUR CLEAN
end-initializations

forall(k in JOBS) setdomain(rank(k), JOBS)

! Cleaning time after every batch
forall(k in JOBS)
 element(CLEAN, rank(k), if(k<NJ, rank(k+1), rank(1))) = clean(k)

! Objective: minimize the duration of a production cycle
cycleTime = sum(j in JOBS) DUR(j) + sum(k in JOBS) clean(k)

! One position for every job
all_different(rank)

! Solve the problem
if not cp_minimize(cycleTime) then
 writeln("Problem is infeasible")
 exit(1)
end-if
cp_show_stats

! Solution printing
writeln("Minimum cycle time: ", getsol(cycleTime))
writeln("Sequence of batches:¥nBatch Duration Cleaning")
forall(k in JOBS)
 writeln("  ", getsol(rank(k)), strfmt(DUR(getsol(rank(k))),8),
         strfmt(getsol(clean(k)),9))

end-model
```

3.8.5 結果

この問題の最小のサイクルタイムは 243 分です。これは、1 → 4 → 3 → 5 → 2 → 1 の順序でバ

ッチを処理することで達成できる値です。また、243 分は、(圧縮できない)処理時間 202 分と 41 分の洗浄時間からなっています。

この問題の Xpress-Kalis によって作成された問題についての統計を見ると、この二番目のモデルは、(ディフォルト戦略を使用して)かなり長い列挙を行っているので、「弱い定式化」であることがわかります。

3.9 equiv:所得税の税務署の配置場所

この例は、「Applications of optimization with Xpress-MP」という本の Section 15.5 にある「Location of income tax offices」から取ったものです。

所得税担当の役所は、管轄区域の所得税徴収オフィスのネットワークを再構築を計画しようとしてます。ここで、各都市の住民の数と、すべての都市間の距離がわかっています(Table Table 3.8)。所得税担当事務所は、十分な範囲をカバーするためには、3 つの都市にオフィスを設置すべきである、という決定を行っています。これらのオフィスを、住民一人あたりの、最も近い所得税オフィスまでの平均距離を最小にするためには、オフィスをどこに設置すべきでしょうか。

Table 3.8: Distance matrix and population of cities

	1	2	3	4	5	6	7	8	9	10	11	12
1	0	15	37	55	24	60	18	33	48	40	58	67
2	15	0	22	40	38	52	33	48	42	55	61	61
3	37	22	0	18	16	30	43	28	20	58	39	39
4	55	40	18	0	34	12	61	46	24	62	43	34
5	24	38	16	34	0	36	27	12	24	49	37	43
6	60	52	30	12	36	0	57	42	12	50	31	22
7	18	33	43	61	27	57	0	15	45	22	40	61
8	33	48	28	46	12	42	15	0	30	37	25	46
9	48	42	20	24	24	12	45	30	0	38	19	19
10	40	55	58	62	49	50	22	37	38	0	19	40
11	58	61	39	43	37	31	40	25	19	19	0	21
12	67	61	39	34	43	22	61	46	19	40	21	0
Pop. (in 1000)	15	10	12	18	5	24	11	16	13	22	19	20

3.9.1 モデルの定式化

CITIES を都市の集合であるとしましょう。問題の定式化には、2 つのグループの決定変数が必要です。すなわち、変数 $build_c$ と変数 $depend_c$ です。変数 $build_c$ は、 都市 c に所得税徴収オフィスが設置さ

れた場合、そして、その場合だけ 1 という値をとります。変数 $depend_c$は、都市 cを管轄とするオフィスの番号を取ります。制約条件の定式化のために、さらに、2 つの補助変数の集合、すなわち、$depdist_c$、および、$numdep_c$を導入します。$depdist_c$は、都市 cから $depend_c$で示されるオフィスまでの距離を示し、$numdep_c$は、徴収オフィスが置かれる位置によって決まってくる管轄都市の数です。

以下の関係が、変数 $depend_c$に変数 $build_c$をリンクするのに必要です。
(1) $numdep_c$は、変数$depend_c$の中でのoffice location cの発生回数をカウントします。
(2)、都市 c にオフィスが設置される場合、そして、その場合にだけ、$numdep_c \geq 1$ となります(結果として、都市 cにオフィスが設置されない場合は、$numdep_c$は=0)。

$$\forall c \in CITIES : numdep_c = \left| depend_d = c \right|_{d \in CITIES}$$

$$\forall c \in CITIES : numdep_c \geq 1 \Leftrightarrow build_c = 1$$

設置されるオフィスの数が所与の $NUMLOC$ で制限されているので、下記の式を得ます。

$$\sum_{c \in CITIES} build_c \leq NUMLOC$$

変数 $build_c$と変数 $depend_c$ の間の二番目の関係の定式化は、実際には、implication「もし、$numdep_c \geq 1$ であるならば、そうすれば、都市 cのオフィスを設置しなければならない。そして、逆に、都市 cのオフィスが設置されないならば、$numdep_c$=0 でなければならない」とすれば十分でしょう。

最小にされるべき目的関数は、都市の住民の数によって重みづけされた総距離です。住民一人あたりの、最も近い所得税オフィスへの平均距離を得るには、結果として得られる値を、区域の総人口で割る必要があります。都市 c から最も近い所得税徴収オフィスの位置までの距離 $depdist_c$ は、discrete function、すなわち、$depend_c$の値でインデックスされた距離マトリクス $DIST_{cd}$の行 cで得られます。

$$depdist_c = DIST_{c, depend_c}$$

以上より、以下の CP モデルを得ます。

$$\text{minimize} \sum_{c \in CITIES} POP_c \cdot dist_c$$

$$\forall c \in CITIES : build_c \in \{0,1\}, depend_c \in CITIES,$$

$$numdep_c \in CITIES \cup \{0\}, depdist_c \in \{\min_{d \in CITIES} DIST_{c,d,\ldots,} \max_{d \in CITIES} DIST_{c,d}\}$$

$$\forall c \in CITIES : depdist_c = DIST_{c, depend_c}$$

$$\sum_{c \in CITIES} build_c \leq NUMLOC$$

$$\forall c \in CITIES : numdep_c = \left| depend_d = c \right|_{d \in CITIES}$$

$$\forall c \in CITIES : numdep_c \geq 1 \Leftrightarrow build_c = 1$$

3.9.2 インプリメンテーション

この問題を解くために、2 つの部分からなる branching strategy を定義します。その一つは、変数 $build_c$ のためのも、そして、もう一つは、変数 $depdist_c$ のためのものです。後者は、*split_domain* branching scheme を使って列挙されます。この branching scheme は、(値を変数に割り当てるのではなく)branching variable のドメインを数個の disjoint subsets に分割します。そして、アーギュメントとして、procedure *cp_set_branching* に、type *cpbranching* の array を渡します。このアレイの中のオーダーで、異なる戦略が適用されるます。ここでの列挙の戦略は、問題のすべての決定変数を明示的に含んではいないので、Xpress-Kalis は、サーチ戦略を適用した後に、アサインされていない変数が残っている場合は、ディフォルト戦略を使ってこれらを列挙すします。

```
model "J-5 Tax office location (CP)"
 uses "kalis"

 forward procedure calculate_dist

 setparam("DEFAULT_LB", 0)

 declarations
  NC = 12
  CITIES = 1..NC                        ! Set of cities

  DIST: array(CITIES,CITIES) of integer ! Distance matrix
  POP: array(CITIES) of integer         ! Population of cities
  LEN: dynamic array(CITIES,CITIES) of integer ! Road lengths
  NUMLOC: integer                       ! Desired number of tax offices
  D: array(CITIES) of integer          ! Auxiliary array used in constr. def.

  build: array(CITIES) of cpvar         ! 1 if office in city, 0 otherwise
  depend: array(CITIES) of cpvar        ! Office on which city depends
  depdist: array(CITIES) of cpvar       ! Distance to tax office
  numdep: array(CITIES) of cpvar        ! Number of depending cities per off.
  totDist: cpvar                        ! Objective function variable
  Strategy: array(1..2) of cpbranching  ! Branching strategy
 end-declarations

 initializations from 'Data/j5tax.dat'
  LEN POP NUMLOC
 end-initializations
```

```
! Calculate the distance matrix
 calculate_dist

 forall(c in CITIES) do
  build(c) <= 1
  1 <= depend(c); depend(c) <= NC
  min(d in CITIES) DIST(c,d) <= depdist(c)
  depdist(c) <= max(d in CITIES) DIST(c,d)
  numdep(c) <= NC
 end-do

! Distance from cities to tax offices
 forall(c in CITIES) do
  forall(d in CITIES) D(d):=DIST(c,d)
  element(D, depend(c)) = depdist(c)
 end-do

! Number of cities depending on every office
 forall(c in CITIES) occurrence(c, depend) = numdep(c)

! Relations between dependencies and offices built
 forall(c in CITIES) equiv( build(c) = 1, numdep(c) >= 1 )

! Limit total number of offices
 sum(c in CITIES) build(c) <= NUMLOC

! Branching strategy
 Strategy(1):= assign_and_forbid(KALIS_MAX_DEGREE, KALIS_MAX_TO_MIN, build)
 Strategy(2):= split_domain(KALIS_SMALLEST_DOMAIN, KALIS_MIN_TO_MAX,
                           depdist, true, 5)
 cp_set_branching(Strategy)

! Objective: weighted total distance
 totDist = sum(c in CITIES) POP(c)*depdist(c)

! Solve the problem
 if not cp_minimize(totDist) then
  writeln("Problem is infeasible")
  exit(1)
 end-if

! Solution printing
 writeln("Total weighted distance: ", getsol(totDist),
        " (average per inhabitant: ",
    getsol(totDist)/sum(c in CITIES) POP(c), ")")
 forall(c in CITIES) if(getsol(build(c))>0) then
  write("Office in ", c, ": ")
  forall(d in CITIES) write(if(getsol(depend(d))=c, " "+d, ""))
  writeln
```

```
 end-if

!-----------------------------------------------------------------

! Calculate the distance matrix using Floyd-Warshall algorithm
 procedure calculate_dist
 ! Initialize all distance labels with a sufficiently large value
  BIGM:=sum(c,d in CITIES | exists(LEN(c,d))) LEN(c,d)
  forall(c,d in CITIES) DIST(c,d):=BIGM

  forall(c in CITIES) DIST(c,c):=0    ! Set values on the diagonal to 0

! Length of existing road connections
  forall(c,d in CITIES | exists(LEN(c,d))) do
   DIST(c,d):=LEN(c,d)
   DIST(d,c):=LEN(c,d)
  end-do

! Update shortest distance for every node triple
  forall(b,c,d in CITIES | c<d )
   if DIST(c,d) > DIST(c,b)+DIST(b,d) then
    DIST(c,d):= DIST(c,b)+DIST(b,d)
    DIST(d,c):= DIST(c,b)+DIST(b,d)
   end-if
 end-procedure

end-model
```

このモデルのインプリメンテーションには、Mosel でのサブルーチンの別の使用例が含まれています。すなわち、距離データの計算は、手順 calculate_dist で行われています。その結果、モデルの構造化にサブルーチンを使用し、主モデルの定式化から、補助的なタスクを取り除くことができます。

3.9.3 結果

この問題の最適解として、人口をウエートとして、総距離 2438 という値が得られました。管轄区域には、合計 18 万 5000 人の住民がいるので、住民一人あたりの平均距離は、2438/185、すなわち、約 13.178km となります。3 つのオフィスは、ノード 1、ノード 6、ノード 11 に設立されることになります。ノード 1 のオフィスは都市 1、2、5、7 をサービス区域とし、ノード 6 のオフィスは都市 3、4、6、9 をサービス区域とし、オノード 11 のフィスは都市 8、10、11、12 をサービス区域とします。

3.10 cycle : ペイントの生産

このセクションでは、もう一度、Section 3.8 のペイント生産の問題について考えて見ましょう。この問題の目的は、すべての組の仕事の間には、ジョブの組合せで、洗浄時間が順序に依存する状態で、ジョブの集合を与えられたとき、すべてのジョブをどのような順番で行ったら、生産サイクルが最小になるかを決定することです。

3.10.1 モデルの定式化

セクション 3.8 の問題定式化は、ジョブのすべてが、一度だけ行われることを保証するするために、'all-different' constraint を使い、'element' constraint で洗浄時間を計算し、そして、生産順序でのサブサイクルを防ぐために、補助変数と制約条件を導入しました。実は、これらの制約条件が示す関係のすべてが、ただ一つの制約関係、'cycle' constraint'で表現することができます。

JOBS = {1, . . . , *NJ*} で生産するバッチの集合を、DUR_j でバッチ j の処理時間を、そして、$CLEAN_{ij}$ で二つの連続するバッチ *i* と *j* の間の洗浄時間を表すものとします。前と同じように、各ジョブに続くジョブを示すために、*JOBS* の中の値を取る決定変数 $succ_j$ を定義します。以下は、その完全なモデル定式化です。

$$\text{minimize} \quad \sum_{j \in JOBS} DUR_j + cleantime$$
$$\forall j \in JOBS : succ_j \in JOBS \setminus \{\mathrm{j}\}$$
$$cleantime = cycle\left(\left(succ_j\right)_{j \in JOBS}, \left(CLEAN_{ij}\right)_{i,j \in JOBS}\right)$$

where 'cycle' stands for the relation 'sequence into a single cycle without subcycles or repetitions'. The variable cleantime equals the total duration of the cycle.

ここで、'cycle'は、「subcycle、および、repetition なしに、単一の cycle に順序付けする」という関係を意味します。また、変数 cleantime は、$cycle\left(\left(succ_j\right)_{j \in JOBS}, \left(CLEAN_{ij}\right)_{i,j \in JOBS}\right)$に等しくなります。

3.10.2 インプリメンテーション

cycle constraint を使った Mosel モデルは、下記のようになります。

```
model "B-5 Paint production (CP)"
 uses "kalis"

 setparam("DEFAULT_LB", 0)

 declarations
  NJ = 5                              ! Number of paint batches (=jobs)
  JOBS=0..NJ-1

  DUR: array(JOBS) of integer         ! Durations of jobs
  CLEAN: array(JOBS,JOBS) of integer ! Cleaning times between jobs
  CB: array(JOBS) of integer          ! Cleaning times after a batch

  succ: array(JOBS) of cpvar          ! Successor of a batch
```

```
 cleanTime,cycleTime: cpvar         ! Durations of cleaning / complete cycle
end-declarations

initializations from 'Data/b5paint.dat'
 DUR CLEAN
end-initializations

forall(j in JOBS) do
 0 <= succ(j); succ(j) <= NJ-1; succ(j) <> j
end-do

! Assign values to 'succ' variables as to obtain a single cycle
! 'cleanTime' is the sum of the cleaning times
cycle(succ, cleanTime, CLEAN)

! Objective: minimize the duration of a production cycle
cycleTime = cleanTime +sum(j in JOBS) DUR(j)

! Solve the problem
if not cp_minimize(cycleTime) then
 writeln("Problem is infeasible")
 exit(1)
end-if
cp_show_stats

! Solution printing
writeln("Minimum cycle time: ", getsol(cycleTime))
writeln("Sequence of batches:¥nBatch Duration Cleaning")
first:=1
repeat
 writeln("  ", first, strfmt(DUR(first),8),
              strfmt(CLEAN(first,getsol(succ(first))),9) )
 first:=getsol(succ(first))
until (first=1)

end-model
```

cycle constraint で求められる入力フォーマットに従うために、タスクの番号が、0 から始まるインデックスで、付け替えられているのに注意してください。

3.10.3 結果

この問題の最適解では、ミニマム・サイクルタイムは243分となり、その内訳は、(圧縮できない)処理時間202分と洗浄時間41分です。

Xpress-Kalis によって作成されたモデル・ランについての統計によると、このモデルの'cycle'バージョンは、この問題を表現し、解くための最も効率的な方法であることが明らかになりました。これは、セクション 3.8 の 2 つの定式化よりも、より少ないノード、より短い時間で解が得られました。

3.11 Generic binary constraints: オイラーのナイトツアー

ここでの課題は、チェス盤の上で、ナイトを動かし、すべてのセルを、一回だけ、通過し、最後に、出発点へ戻ってくるツアー(経路)を見つけることです。ナイトの経路は、チェスの標準ルールに従わなければなりません。ナイトは、一回に、垂直方向に1つのセルだけ動き、併せて、水平方向には2つのセルだけ動く、もしくは、垂直方向に2つのセルだけ動き、併せて、水平方向には1つのセルだけ動かせます。つまり、Figure3.3 で示されているように、この図の中央に位置しているナイトは、茶色のセルにのみ、動ける、ということです。

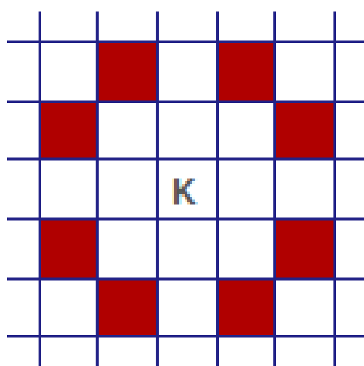


Figure 3.3: Permissible moves for a knight

3.11.1 モデルの定式化

チェス盤を表すために、ここでは、セルに、0 から N-1 までの番号を付けます。ここで、N はチェス盤のセルの数です。ナイトの経路は、ナイトが、ツアーの途中の i 番目の位置を示す N 個の変数 pos_i によって定義されます。

最初の変数を、「ツアーの開始地点」としてゼロに固定します。もう一つ、明らかに必要な制約条件は、すべての変数 pos_i は異なる値を取るということで、換言すれば、チェス盤のセルは、すべてが、まさに、一回だけ、訪問されるということです。

$$all - different(pos_1,..., pos_n)$$

ここで、「連続した位置が有効なナイトの移動を定義するかどうか」をチェックできる制約関係を確立する必要があります。この目的のために、新しい binary constraint 'valid_knight_move' を定義します。'valid_knight_move' は、与えられた二つの値の組が、ナイトの移動についてのチェスのルールに従う、許された移動を定義するかどうかチェックします。二つの値は、垂直方向、および、水平方向に、2 つ以下のセルで離れていること、そして、垂直方向、水平方向の場所の違いの合計が 3 でなければなりません。こうして、モデル全体は以下のようになります。

$$\forall i \in PATH = \{1,..., N\} : pos_i \in \{0,..., N-1\}$$

$$pos_1 = 0$$
$$all-different(pos_1,...,pos_N)$$
$$\forall i \in POS = \{1,...,N-1\}: valid_knight_move(pos_i, pos_{i+1})$$
$$valid_knight_move(pos_N, pos_1)$$

3.11.2 インプリメンテーション

ナイトの位置 *a* から位置 *b* への動きが正しいかどうかをテストすることは、function *valid_knight_move* で行えます。ここで、'div'は、余りを持たない整数の割り算(integer division without rest)で、'mod'は、整数の割り算の余り(rest of the integer division)を意味します。

```
-----------------------------------------------------------------------
function valid_knight_move(a,b)
 xa := a div E
 ya := a mod E
 xb := b div E
 yb := b mod E
 deltax} := |xa-xb|
 deltay} := |ya-yb|
 return ((deltax} <= 2) and (deltay} <= 2) and (deltax} + deltay} = 3))
end-function

-----------------------------------------------------------------------
```

以下の Mosel モデルは、movep という二つの値のペア変数についての「新しい binary constraint の実行」としての、user constraint function 'valid_knight_move' の定義です。(この constraint は、generic_binary_constraint により設定されます)。

```
model "Euler Knight Moves"
 uses "kalis"

 parameters
  S = 8                                  ! Number of rows/columns
  NBSOL = 1                              ! Number of solutions sought
 end-parameters

 forward procedure print_solution(sol: integer)

 N:= S * S                               ! Total number of cells
 setparam("DEFAULT_LB", 0)
 setparam("DEFAULT_UB", N-1)

 declarations
  PATH = 1..N                           ! Cells on the chessboard
  pos: array(PATH) of cpvar             ! Cell at position p in the tour
```

```
 end-declarations

! Fix the start position
 pos(1) = 0

! Each cell is visited once
 all_different(pos, KALIS_GEN_ARC_CONSISTENCY)

! The path of the knight obeys the chess rules for valid knight moves
 forall(i in 1..N-1)
  generic_binary_constraint(pos(i), pos(i+1), "valid_knight_move")
 generic_binary_constraint(pos(N), pos(1), "valid_knight_move")

! Setting enumeration parameters
 cp_set_branching(probe_assign_var(KALIS_SMALLEST_MIN, KALIS_MAX_TO_MIN,
                                   pos, 14))

! Search for up to NBSOL solutions
 solct:= 0
 while (solct<NBSOL and cp_find_next_sol) do
  solct+=1
  cp_show_stats
  print_solution(solct)
 end-do


! **** Test whether the move from position a to b is admissible ****
 function valid_knight_move(a:integer, b:integer): boolean
  declarations
   xa,ya,xb,yb,delta_x,delta_y: integer
  end-declarations

  xa := a div S
  ya := a mod S
  xb := b div S
  yb := b mod S
  delta_x := abs(xa-xb)
  delta_y := abs(ya-yb)
  returned := (delta_x<=2) and (delta_y<=2) and (delta_x+delta_y=3)
 end-function

!****************************************************************
! Solution printing
 procedure print_solution(sol: integer)
  writeln("Solution ", sol, ":")
  forall(i in PATH)
   write(getval(pos(i)), if(i mod 10 = 0, "¥n ", ""), " -> ")
  writeln("0")
 end-procedure
```

```
end-model
```

このモデルで使用される branching scheme は、probe_assign_var heuristic で、変数選択のための KALIS_SMALLEST_MIN(smallest lower bound を持つ変数を選択する)、および、値の選択基準である KALIS_MAX_TO_MIN(最も大きいドメイン値から最も小さいドメイン値へ)と組み合わせて使います。(上のコードリストの戦略より遅いですが)よく機能することが判明している、もう一つのサーチ戦略は、以下のものです。

```
cp_set_branching(assign_var(KALIS_SMALLEST_MIN, KALIS_MAX_TO_MIN, pos))
```

このモデルは、2 つのパラメタを定義します。 これにより、モデルを実行するとき、モデルソースを変更することなく、チェス盤(S)のサイズや、実行するときに求めるソリューションの数(NBSOL)を変えることが可能になります。

3.11.3 結果

このモデルによって印刷された最初のソリューションは、以下のツアーです。

```
0 -> 17 -> 34 -> 51 -> 36 -> 30 -> 47 -> 62 -> 45 -> 39
  -> 54 -> 60 -> 43 -> 33 -> 48 -> 58 -> 52 -> 35 -> 41 -> 56
  -> 50 -> 44 -> 38 -> 55 -> 61 -> 46 -> 63 -> 53 -> 59 -> 49
  -> 32 -> 42 -> 57 -> 40 -> 25 -> 8 -> 2 -> 19 -> 4 -> 14
  -> 31 -> 37 -> 22 -> 7 -> 13 -> 28 -> 18 -> 24 -> 9 -> 3
  -> 20 -> 26 -> 16 -> 1 -> 11 -> 5 -> 15 -> 21 -> 6 -> 23
  -> 29 -> 12 -> 27 -> 10 -> 0
```

3.11.4 代替的なインプリメンテーション

上のモデルの目的は、user constraint の定義の一例をあげることでしたが、Section 3.10 の cyclic scheduling problem 用に作成されたモデルを使って、より効率的な方法で、この問題を実行することも可能です。 set of successors (domains of variables $succ_p$)は、上で、user-defined binary constraint の実行で使用したのと同じアルゴリズムを使用することで計算できます。

Without repeating the complete model definition at this place, we simply display the model formulation, including the calculation of the sets of possible successors (procedure calculate_successors, and the modified procedure print_sol for solution printout. We use a simpler version of the 'cycle' constraints than the one we have seen in Section 3.1-0, its only argument is the set of successor variables---there are no weights to the arcs in this problem.

ここでは、モデル全体の定義を繰り返さず、sets of possible successors (procedure

calculate_successors)、および、ソリューションのプリントのための修正された procedure print_sol を含む、モデルの定式化のみを表示します。Section 3.10 で見たものよりも、より簡単な'cycle' constraint'のバージョンを使いますが、その唯一の argument は、set of successor variables です。この問題の arcs へのウエイトはまったくありません。

```
model "Euler Knight Moves"
 uses "kalis"

 parameters
  S = 8                                    ! Number of rows/columns
  NBSOL = 1                                ! Number of solutions sought
 end-parameters

 forward procedure calculate_successors(p: integer)
 forward procedure print_solution(sol: integer)

 N:= S * S                                 ! Total number of cells
 setparam("DEFAULT_LB", 0)
 setparam("DEFAULT_UB", N)

 declarations
  PATH = 0..N-1                            ! Cells on the chessboard
  succ: array(PATH) of cpvar               ! Successor of cell p
 end-declarations

! Calculate set of possible successors
 forall(p in PATH) calculate_successors(p)

! Each cell is visited once, no subtours
 cycle(succ)

! Search for up to NBSOL solutions
 solct:= 0
 while (solct<NBSOL and cp_find_next_sol) do
  solct+=1
  cp_show_stats
  print_solution(solct)
 end-do


! **** Calculate possible successors ****
 procedure calculate_successors(p: integer)
  declarations
   SuccSet: set of integer                 ! Set of successors
  end-declarations

  xp := p div S
  yp := p mod S
```

```
  forall(q in PATH) do
   xq := q div S
   yq := q mod S
   delta_x := abs(xp-xq)
   delta_y := abs(yp-yq)
   if (delta_x<=2) and (delta_y<=2) and (delta_x+delta_y=3) then
    SuccSet +={q}
   end-if
  end-do

  setdomain(succ(p), SuccSet)
 end-procedure

!**************************************************************
! **** Solution printing ****
 procedure print_solution(sol: integer)
  writeln("Solution ", sol, ":")
  thispos:=0
  nextpos:=getval(succ(0))
  ct:=1
  while (nextpos<>0) do
   write(thispos, if(ct mod 10 = 0, "¥n ", ""), " -> ")
   val:=getval(succ(thispos))
   thispos:=nextpos
   nextpos:=getval(succ(thispos))
   ct+=1
  end-do
  writeln("0")
 end-procedure

end-model
```

変数 succp のドメインの計算により、これらを、2-8 要素まで減少させます。あらゆる変数 pos_pの N= 64 と比較してください。これにより、この問題のサーチスペースは、明らかに減少します。

This second model finds the first solution to the problem after 131 nodes taking just a fraction of a second to execute on a standard PC whereas the first model requires several thousand nodes and considerably longer running times. It is possible to reduce the number of branch-and-bound nodes even further by using yet another version of the 'cycle' constraint that works with successor and predecessor variables. This version of 'cycle' performs a larger amount of propagation, at the expense of (slightly) slower execution times for our problem. The procedure calculate_successors now sets the domain of predp} to the same values as succp} for all cells p.

この二番目のモデルは、標準のパソコンで、131 ノードのサーチのあと、この問題の最初のソリューシ

ョンを、まさしく、1 秒も掛けずにみつけました。それにたいして、最初のモデルでは、数千のノードのサーチが必要で、したがって、かなり長い実行時間が掛かります。さらに、'cycle' constraint の、successor 変数、predecessor 変数を考慮するさらに別のバージョンを使うと、branch-and-bound node の数を、もっと減らすことができます。'cycle'の、このバージョンは、この問題の実行時間を、ごくわずかですが犠牲にすることと引き換えに、もっと多くのプロパゲーションを実行できます。procedure calculate_successors は、すべてのセル *p* にたいして、domain of $pred_p$ を $succ_p$ と同じ値にセットするようになります。

```
declarations
 PATH = 0..N-1                          ! Cells on the chessboard
 succ: array(PATH) of cpvar             ! Successor of cell p
 pred: array(PATH) of cpvar             ! Predecessor of cell p
end-declarations

! Calculate sets of possible successors and predecessors
 forall(p in PATH) calculate_successors(p)

! Each cell is visited once, no subtours
 cycle(succ, pred)
```

Figure3.4 は、IVE の'user graph'機能を使って作成されたナイトのツアーのグラフィカルなディスプレーです。

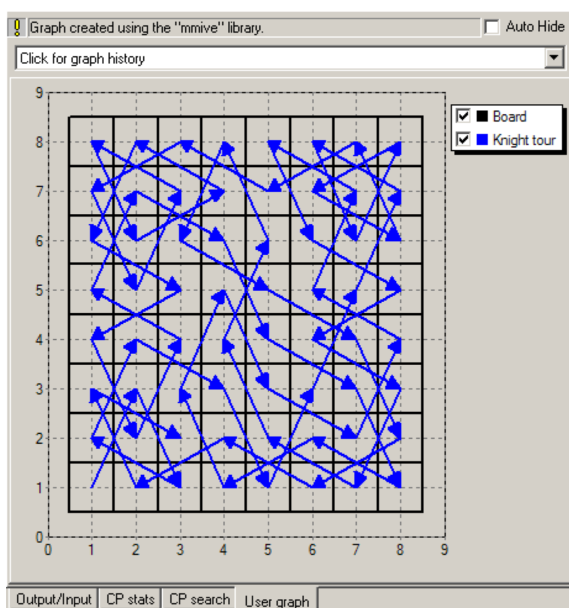


Figure 3.4: Knight's tour solution graph in IVE

3.11.5 代替的なインプリメンテーション　その二

しかし、さらに、(同じような数のノードですが、かなり短い実行時間という意味で)もっと効率的なのは、Section 3.10 で説明されている'cycle' constraint を使用する下記のモデルバージョンです。このモデルでは、1 つの位置から次の位置までの移動時間は、無意味なので、ここでは、単純に、'cycle' constraint のすべての係数を定数に固定して、cycle length を対応する値に制約します。

sets of possible successors (procedure calculate_successors)、および、ソリューションのプリントアウトのための procedure print_sol を含む、その他のすべての部分は、前のモデルからのコピーです。

```
model "Euler Knight Moves"
 uses "kalis"

 parameters
  S = 8                                  ! Number of rows/columns
  NBSOL = 1                              ! Number of solutions sought
 end-parameters

 forward procedure calculate_successors(p: integer)
 forward procedure print_solution(sol: integer)

 N:= S * S                               ! Total number of cells
 setparam("DEFAULT_LB", 0)
 setparam("DEFAULT_UB", N)

 declarations
  PATH = 0..N-1                          ! Cells on the chessboard
  succ: array(PATH) of cpvar             ! Successor of cell p
 end-declarations

! Calculate set of possible successors
 forall(p in PATH) calculate_successors(p)

! Each cell is visited once, no subtours
 cycle(succ)

! Search for up to NBSOL solutions
 solct:= 0
 while (solct<NBSOL and cp_find_next_sol) do
  solct+=1
  cp_show_stats
  print_solution(solct)
 end-do
```

```
! **** Calculate possible successors ****
 procedure calculate_successors(p: integer)
  declarations
   SuccSet: set of integer                ! Set of successors
  end-declarations

  xp := p div S
  yp := p mod S

  forall(q in PATH) do
   xq := q div S
   yq := q mod S
   delta_x := abs(xp-xq)
   delta_y := abs(yp-yq)
   if (delta_x<=2) and (delta_y<=2) and (delta_x+delta_y=3) then
    SuccSet +={q}
   end-if
  end-do

  setdomain(succ(p), SuccSet)
 end-procedure

!****************************************************************
! **** Solution printing ****
 procedure print_solution(sol: integer)
  writeln("Solution ", sol, ":")
  thispos:=0
  nextpos:=getval(succ(0))
  ct:=1
  while (nextpos<>0) do
   write(thispos, if(ct mod 10 = 0, "¥n ", ""), " -> ")
   val:=getval(succ(thispos))
   thispos:=nextpos
   nextpos:=getval(succ(thispos))
   ct+=1
  end-do
  writeln("0")
 end-procedure

end-model
```

第 4 章 列挙（Enumeration）

この章は、下記のような、Xpress-Kalis のサーチ戦略の定義に関して、その概観を説明します。

- 事前に定義されているサーチ戦略
- 列挙を中断し、再開する方法
- サーチの callback
- ユーザサーチ戦略の定義

4.1 事前に定義されているサーチ戦略

Xpress-Kalis では、ブランチ戦略は、下記の 3 つのコンポーネントから構成されます。

- サーチツリーの形を決定するのはブランチスキームです。その中には、assign_var、split_domain (enumeration of decision variables)、 settle_disjunction (enumeration over constraints)、task_serialize (enumeration of tasks in scheduling problems)、もしくは、probe_assign_var や probe_settle_disjunction のように、潜在的に不完全なサーチなども含まれています。

- 変数選択戦略は、ブランチする変数の選択を決定します。あらかじめ、事前に定義されているクライテリアには、以下の物があります。

KALIS_INPUT_ORDER:	variables in the given order
KALIS_LARGEST_MAX:	variable with largest upper bound
KALIS_LARGEST_MIN:	variable with largest lower bound
KALIS_MAX_DEGREE:	variable occurring in the largest number of constraints
KALIS_MAXREGRET_LB:	variable with largest difference between its lower bound and second-smallest domain value
KALIS_MAXREGRET_UB:	variable with largest difference between its upper bound and second-largest domain value
KALIS_RANDOM_VARIABLE:	random variable choice
KALIS_SDOMDEG_RATIO:	variable with smallest ratio domain size/degree
KALIS_SMALLEST_DOMAIN	variable with smallest number of domain values/smallest domainintervall
KALIS_SMALLEST_MAX:	variable with smallest upper bound
KALIS_SMALLEST_MIN:	variable with smallest lower bound)

KALIS_WIDEST_DOMAIN: variable with largest number of domain values/ largest domain interval

また、ユーザも、自分自身の変数選択戦略を定義することもできます(下の Section 4.4.3 を見てください)。

- ひとたび、ブランチするべき変数が決まると、value selection 戦略により、branching value の選択が決定されます。下記に列記した、あらかじめ、事前に定義された選択クライテリアが利用可能です。

  KALIS_MAX_TO_MIN: enumeration of values in decreasing order,
  KALIS_MIDDLE_VALUE: enumerate first values in the middle of the variable's domain,
  KALIS_MIN_TO_MAX: enumeration of values in increasing order,
  KALIS_NEAREST_VALUE: choose the value closest to a target value previously specified with settarget,
  KALIS_RANDOM_VALUE: choose a random value out of the variable's domain.

  あらかじめ、事前に定義された評価クライテリアは、ユーザ自身の value 選択戦略に取り替えることがもきます(Section 4.4.3 を見てください)。

branching scheme の選択によっては、列挙を構成するための、追加パラメタを指定することができます。詳細については、Xpress-Kalis reference manual を参照してください。constraint branching の場合は、明らかなことですが、変数や branching value の選択はありません。Section 5.7 で、「task-based branching strategies (branching scheme task_serialize)」の特別なケースについて議論します。連続変数(type cpfloatvar)は、いつも、branching scheme 'split_domain' を、KALIS_WIDEST_DOMAIN 変数選択と一緒に使い、これは、上でリストした変数選択のサブセットです。

ブランチイング戦略は、ある指定された変数(もしくは、適用できる場合には、constraint や task)、そして、branching scheme function のアーギュメントが省かれている場合は、モデルのすべての決定変数に適用されます(例えば、Section 3.5 のモデル b4seq_ka.mos を見てください)。

ユーザのモデルが、ブランチ戦略をなにも指定しないと、Xpress-Kalis は、ディフォルト戦略を適用して、モデルの中の、すべての変数を列挙します。よしんば、列挙のディフォルト戦略(for discrete variables: `smallest domain first' and `smallest value first')を変える必要がないと思う場合でも、決定変数の異なるグループの列挙の順序を固定するために、ブランチ戦略を定義するのは望ましいことです。前の章には、これのいくつかの例が含まれています。例えば、Section 3.6 のモデル a4sugar_ka.mos には、モデルの変数のいくつかの列挙を定義し、その結果、モデルの変数のいくつかは、残りの変数に対して優先されます。また、Section 3.9 のモデル j5tax_ka.mos では、決定変数のグループごとに、異なる列

挙で構成されるサーチ戦略の例を見ました。

モデルが離散変数と連続変数(cpvar と cpfloatvar)の両方を含んでいる場合は、ディフォルト戦略は、最初に、離散変数を列挙して、次いで、連続変数を列挙します。

4.2 列挙を中断し、再開する方法

大きいアプリケーションを解くとき、それなりの妥当な時間で、完全なサーチツリーを列挙するのは、しばしば、可能でないことがあります。したがって、Xpress-Kalis では、サーチを中断するための、下に記したような、いくつかの中断クライテリアが利用可能です。

MAX_BACKTRACKS:	maximum number of backtracks,
MAX_COMPUTATION_TIME:	limit on total time spent in search,
MAX_DEPTH:	maximum search tree depth,
MAX_NODES:	maximum number of nodes to explore,
MAX_SOLUTIONS:	maximum number of solutions,
OPT_ABS_TOLERANCE:	absolute difference between the objective function value in a solution and its best possible value (= current upper bound on objective function in maximization problems and lower bound with minimization),
OPT_REL_TOLERANCE:	relative difference between the objective function value in a solution and its best possible value (= current upper bound on objective function in maximization problems and lower bound with minimization).

これらのパラメタには、Mosel ファンクションの setparam と getparam でアクセスできます(例えば、Section 3.3 のモデル sudoku_ka.mos の問題統計の出力や、Section 3.4 のモデル freqasgn.mos の search time limit set を見てください)。

最適化問題では、パラメタ OPTIMIZE_WITH_RESTART の設定が変えられない場合、ソリューションが見つけられた後に、このポイントからサーチが続きます。サーチが、上でリストしたクライテリア 1 つによって中断されて、次に、例えば、異なるサーチ戦略で続けられても同じです。root node からサーチを再開するには、手順 cp_reset_search をコールする必要があります(例としては、Section 3.4 のモデル freqasgn.mos を見てください)。

4.3 サーチの callback

サーチの間に、「callback function」によって、ユーザ・モデルは、あらかじめ定義されたポイントで、ソルバーと対話(interact)できます。この機能は、最適化を行っている最中に、Section 3.4 のモデル freqasgn.mos で示されているように、中間的なソリューションの情報をリトリーブするのに、特に便利です。 この solution callback を除いて、ユーザは、あらゆる branch、または、あらゆるノードでコールされるファンクションを設定できます(詳細は、Xpress-Kalis reference manual を見てください)。

4.4 ユーザサーチ戦略の定義

以下の問題は、`Applications of optimization with Xpress-MP"の Section 14.1 から取ったものです。

あるワークショップには 6 つの機械があり、そのそれぞれに、オペレータを割り当てれる必要があります。あらかじめ、6 人の労働者が選択されています。6 人の労働者は、6 つの機械のそれぞれで、生産性のテストを受けています。Table 4.1 は、1 時間あたりの生産性を、個数単位で記したものです。機械は、パラレルに動かされます。これは、ワークショップ全体の生産性は、個々の機械に割り当てられた人々の生産性の合計であることを意味します。

Table: Productivity in pieces per hour

	Machines					
Workers	1	2	3	4	5	6
1	13	24	31	**19**	40	29
2	**18**	25	30	15	43	22
3	20	20	27	25	34	**33**
4	23	**26**	28	18	37	30
5	28	33	**34**	17	38	20
6	19	36	25	27	**45**	24

目的関数は、全体の生産性を最大にするには、機械への労働者の割り当てをどのように決めたらよいか、ということです。この問題を解くのに、以下のかなり自然な方法を使って、(non-optimal な)ヒューリスティック解を得ることから始めることができます。労働者 *p* を、その労働者が最も高い生産性を示した機械 *m* に割り当てます($p \rightarrow m$)。(こうして、労働者 *p* が既に機械 *m* に配置され、機械 *m* にはオペレータがいるので)行 *p*、列 *m* を削除します。このプロセスを、すべての労働者が割り当てられるまで、繰り返し行います。結果として得られた割り当ては、データテーブルの太字で示されています。 しかし、ここでの目的は、実は、この並列に稼動される 6 台の機械への労働者の割り当ての最適解を得ること、

および、直列で稼動される 6 台の機械への労働者の割り当ての最適解を得ることです。

4.4.1 モデルの定式化

このタイプの問題は、割当問題という名前でよく知られています。*PERS* で労働者の集合を、*MACH* で機械の数号を、そして、$OUTP_{pm}$で労働者 *p* の、機械 *m* での生産性を表すものとします。 *PERS* と *MACH* は、両方とも、同じサイズ *N* の集合です。ここで、*N* 個の整数変数 $assign_p$を定義します。$assign_p$は、機械の集合の中の値を取り、そこでは、$assign_p$は、労働者 *p* が割り当てられる機械の番号を示します。労働者 *p* が、単一のマシン *m* に割り当てられ、機械 *m* が、単一の労働者 *p* によって操作されるという事実は、変数 $assign_p$のすべてが異なる値を取る、という制約条件で表現されます。

$$\forall p \in PERS : assign_p \in MACH$$
$$all - different\left(\bigcup_{m \in MACH} assign_p\right)$$

さらに、$output_p$で、労働者 *p* によって生産されるアウトプットを示すことにします。これらの変数の値は、変数 $assign_p$の離散関数として得られます。

$$\forall pPERS : output_p = OUT_{p,assign_p}$$

4.4.1.1 パラレルに動かされ機械への割当て

目的関数は、下記のように定式化され、$output_p$の合計を最大にします。

$$\text{Maximize} \quad \sum_{p \in PERS} output_p$$

割当てによっては、インフィージブルかもしれません。このような場合には、対応する機械の値を、変数 $assign_p$のドメインから取り除く必要があります。

4.4.1.2 直列で稼動される機械

機械が直列で稼動される場合、それぞれの機械に割り当てられている労働者の中の、一番、生産性の低い労働者が、ライン全体の生産性を決定します。この場合も、割当は、N 個の変数 $assign_p$と、これらの変数についての *all-different* constraint によって表現できます。前のモデルの $assign_p$変数の値に、$output_p$ 変数をリンクする制約条件をともなう $output_p$ 変数もあります。これに、最小の生産性を表す変数 *pmin* を加えます。こうして、目的関数は、*pmin* を最大にすることになります。「最小を「最大にしたい」という、このようなタイプの最適化問題は、maximin 問題とか bottleneck 問題と呼ばれます。

$$\text{maximize } pmin$$

$$pmin = minimum_{p \in PERS}(output_p)$$

4.4.2 インプリメンテーション

以下のMoselプログラムは、最初に、並列マシンのケースのモデルを実行し、解きます。 その後に、機械が直列に動くケースの問題を解くのに必要な変数*pmin* を定義します。

```
model "I-1 Personnel assignment (CP)"
 uses "kalis"

 forward procedure parallel_heur
 forward procedure print_sol(text1,text2:string, objval:integer)

 declarations
  PERS = 1..6                         ! Personnel
  MACH = 1..6                         ! Machines
  OUTP: array(PERS,MACH) of integer   ! Productivity
 end-declarations

 initializations from 'Data/i1assign.dat'
  OUTP
 end-initializations

! **** Exact solution for parallel machines ****

 declarations
  assign: array(PERS) of cpvar        ! Machine assigned to a person
  output: array(PERS) of cpvar        ! Productivity of every person
  totalProd: cpvar                    ! Total productivity
  O: array(MACH) of integer           ! Auxiliary array for constraint def.
  Strategy: cpbranching               ! Branching strategy
 end-declarations

 forall(p in PERS) setdomain(assign(p), MACH)

! Calculate productivity per worker
 forall(p in PERS) do
  forall(m in MACH) O(m):= OUTP(p,m)
  element(O, assign(p)) = output(p)
 end-do

! Calculate total productivity
 totalProd = sum(p in PERS) output(p)

! One person per machine
 all_different(assign)

! Branching strategy
 Strategy:= assign_var(KALIS_LARGEST_MAX, KALIS_MAX_TO_MIN, output)
```

```
 cp_set_branching(Strategy)

! Solve the problem
 if cp_maximize(totalProd) then
  print_sol("Exact solution (parallel assignment)", "Total", getsol(totalProd))
 end-if

! **** Exact solution for serial machines ****

 declarations
  pmin: cpvar                      ! Minimum productivity
 end-declarations

! Calculate minimum productivity
 pmin = minimum(output)

! Branching strategy
 Strategy:= assign_var(KALIS_SMALLEST_MIN, KALIS_MAX_TO_MIN, output)
 cp_set_branching(Strategy)

! Solve the problem
 if cp_maximize(pmin) then
  print_sol("Exact solution (serial machines)", "Minimum", getsol(pmin))
 end-if
```

パラレルな割当問題へのソリューションが見つかると、そのソリューションをプリントアウトし、新しいブランチ戦略と新しい目的関数でサーチを再開します。(タイムリミットによる中断などがなく)最初のサーチが完全に終わったので、2 つのランの間で、ソルバーをリセットする必要は全くありません。

パラレルな割当の問題に選ばれたブランチ戦略は、Section 4.4 のイントロダクションで説明された直感的な手順にヒントを得たものです。すなわち、機械に労働者割り当てる可能な方法を列挙すること(変数 $assign_p$ に関する列挙)の代わりに、変数 $output_p$ についての列挙を定義し、largest remaining value (KALIS_LARGEST_MAX)変数を選び、大きいほうから小さいほう順(KALIS_MAX_TO_MIN)に、その値にブランチします。二番目の問題に対しては、別の途をとる必要があります。すなわち、労働者 p の生産性の値が小さい値になるのを避けるために、まず、smallest lower bound (KALIS_SMALLEST_MIN) を持つ変数 $output_p$を選びます。そして、ここでも、最も大きい値から、それらを列挙します。

上記のプログラムに、以下の手順 parallel_heur を追加しましょう。これは、Section 4.4 へのイントロダクションで説明された直感的な手順を使い、パラレルな割当問題の(non-optimal な)ソリューションをヒューリスティックに計算します。

```
procedure parallel_heur
 declarations
  ALLP, ALLM: set of integer      ! Copies of sets PERS and MACH
  HProd: integer                  ! Total productivity value
  pmax,omax,mmax: integer
```

```
 end-declarations

! Copy the sets of workers and machines
 forall(p in PERS) ALLP+={p}
 forall(m in MACH) ALLM+={m}

! Assign workers to machines as long as there are unassigned persons
 while (ALLP<>{}) do
  pmax:=0; mmax:=0; omax:=0

! Find the highest productivity among the remaining workers and machines
  forall(p in ALLP, m in ALLM)
   if OUTP(p,m) > omax then
    omax:=OUTP(p,m)
    pmax:=p; mmax:=m
   end-if

  assign(pmax) = mmax              ! Assign chosen machine to person pmax
  ALLP-={pmax}; ALLM-={mmax}       ! Remove person and machine from sets
 end-do

 writeln("Heuristic solution (parallel assignment):")
 forall(p in PERS)
  writeln("  ",p, " operates machine ", getval(assign(p)),
          " (",getval(output(p)), ")")
 writeln("  Total productivity: ", getval(totalProd))
end-procedure
```

モデルは、適切にフォーマットされた方法でソリューションを印プリントアウトするための手順を置き、完成します。

```
procedure print_sol(text1,text2:string, objval:integer)
  writeln(text1,":")
  forall(p in PERS)
   writeln("  ",p, " operates machine ", getsol(assign(p)),
           " (",getsol(output(p)), ")")
  writeln("  ", text2, " productivity: ", objval)
 end-procedure

end-model
```

4.4.3 ユーザによるサーチ

これまでのすべてのモデルのインプリメンテーションに示されている、事前に定義された変数と値の選択クライテリアを使うことの代わりに、ユーザは、ユーザ自身のサーチヒューリスティックを定義することもできます。ここでは、前のモデルのサーチ戦略を、どのように「手動」で実行するかを示します。ここでのインプリメンテーションでは、毎回、二番目のクライテリオンを加え、メインのクライテリオンが、一度に、いくつかの変数に適用されるケースで、それを断ち切ります。

変数の選択: 変数選択ファンクションは、固定フォーマットを持っています。これは、アーギュメントとして、タイプcpvarlistの変数のリストを受け取ります。そして、整数(すなわち、選ばれたブランチング変数のリスト・インデックス)を返します。このファンクションに渡される変数のリストは、既に、挙げられている変数を含むかもしれません。したがって、インプリメンテーションの最初のステップとして、いまだ、挙げられていない変数の集合Vset、もしくは、より正確に言えば、リストの中の、それらのインデックの集合を決定します。変数のリストのエントリーには、ファンクションgetvarでアクセスします。いまだ、挙げられていない変数を対象に、(ファンクションgetubを使用して)最も大きい上限値を持つ変数の集合Iset(set of variables Iset with the largest upper bound value)を計算します。最後に、これらの変数の中から、最も小さい2番目に大きい値(smallest second-largest value)を持つ変数を選びます(これは、maximum regret strategyに対応している)。決定変数のドメインの中の、predecessor (next-smallest) valuesは、ファンクションgetprevで得ます。逆に、昇順に、ドメインの値を列挙するためのファンクションgetnextも持っています。 選ばれたインデックス値は、ファンクションのリターン値として、returnedに割り当てられます。

```
function varchoice(Vars: cpvarlist): integer
  declarations
   Vset,Iset: set of integer
  end-declarations

 ! Set of uninstantiated variables
  forall(i in 1..getsize(Vars))
   if not is_fixed(getvar(Vars,i)) then Vset+= {i}; end-if

  if Vset={} then
   returned:= 0
  else
  ! Get the variable(s) with largest upper bound
   dmax:= max(i in Vset) getub(getvar(Vars,i))
   forall(i in Vset)
    if getub(getvar(Vars,i)) = dmax then Iset+= {i}; end-if
   dmin:= dmax

  ! Choose variable with smallest next-best value among those indexed by 'Iset'
   forall(i in Iset) do
    prev:= getprev(getvar(Vars,i),dmax)
    if prev < dmin then
     returned:= i
     dmin:= prev
    end-if
   end-do
  end-if
 end-function
```

2番目の最適化のラン（直列で稼動する機械）のための変数の選択戦略varchoiceminも、同じような方法で実行されます。最初に、getlbを使って、最も小さい下限値（smallest lower bound value）を持つ変数の集合Isetを確立します。これらの中から、smallest upper bound (getub)を持つ変数を選びます。

```
function varchoicemin(Vars: cpvarlist): integer
 declarations
  Vset,Iset: set of integer
 end-declarations

! Set of uninstantiated variables
 forall(i in 1..getsize(Vars))
  if not is_fixed(getvar(Vars,i)) then Vset+= {i}; end-if

 if Vset={} then
  returned:= 0
 else
 ! Get the variable(s) with smallest lower bound
  dmin:= min(i in Vset) getlb(getvar(Vars,i))
  forall(i in Vset)
   if getlb(getvar(Vars,i)) = dmin then Iset+= {i}; end-if

 ! Choose variable with smallest upper bound among those indexed by 'Iset'
  dmax:= getparam("default_ub")
  forall(i in Iset)
   if getub(getvar(Vars,i)) < dmax then
    returned:= i
    dmax:= getub(getvar(Vars,i))
   end-if
 end-if
end-function
```

値の選択: 値の選択ファンクションは、アーギュメントとして、選択されたブランチング変数を受け取って、この変数のブランチング値を返します。選んだ「値の選択クライテリオン」(Kalis_MAX_TO_MINに対応する)は、最も大きい残っている値(すなわち、変数の上限)から始め、ブランチング変数のすべての値を列挙します。

```
function valchoice(x: cpvar): integer
 returned:= getub(x)
end-function
```

assign_var、もしくは、assign_and_forbid戦略では、ファンクションcontainsを使い、ユーザの値の選択戦略が、必ず、現在、確かに、ブランチング変数のドメインにある値を返すように気をつけてください。上限、下限の間から選ばれた値は、かならずしも、ドメインにあることを保障することにはなりません。

ユーザサーチ戦略の設定: ユーザ自身の変数選択戦略や値の選択戦略を使いことを示すには、

単に、Mosel ファンクションの名前で事前に定義された定数を取り替えるだけです。

```
Strategy:= assign_var("varchoice", "valchoice", output)
```

上記は、最初の最適化のランのための戦略を定義します。

```
Strategy:= assign_var("varchoicemin", "valchoice", output)
```

上記は、直列で動く機械のケースに対して、戦略を再定義します。ファンクション valchoice は、KALIS_MAX_TO_MIN クライテリオンと、まったく、同じことを行うので、それを下記の変数選択ファンクションと組み合わせることもできます。

```
Strategy:= assign_var("varchoicemin", KALIS_MAX_TO_MIN, output)
```

4.4.4 結果

以下のテーブルは、パラレルに機械を動かす問題、および、直列で機械を動かす問題に関して得られた結果の要約です。パラレルに機械を動かす問題へのヒューリスティックな方法と厳密なソリューションの間には、著しい違いがあります。

Table 4.2: Optimal assignments for different model versions

		Person						
	Alg.	1	2	3	4	5	6	Productivity
Parallel Machines	Heur.	4 (19)	1 (18)	6 (33)	2 (26)	3 (34)	5 (45)	175
	Exact	3 (31)	5 (43)	4 (25)	6 (30)	1 (28)	2 (36)	193
Serial Machines	Exact	5 (40)	3 (30)	6 (33)	2 (26)	1 (28)	4 (27)	26

モデルにソルバー統計の出力を追加するか(cp_show_stats)、または、IVE で、CP 統計ディスプレーを使い、以下のようなことがわかります。すなわち、パラレル割当では、ユーザサーチ戦略が、事前に定義された戦略と同じサーチツリーをもたらしたこと、および、プログラム実行時間も同じであったこと、また、直列では、わずかに異なったソリューションをもたらしたことがわかりました。

第 5 章 スケジュール作成

この章では、下記のことについて説明します。

- どのように、モデル作成のオブジェクト cptask と cpresource を定義して、セットアップしたらよいか。
- これらのオブジェクトを使い、どのようにして、スケジュール作成問題を定式化し、解いたらよいうか。
- どのようにして、これらのモデル作成オブジェクトの情報にアクセスしたらよいか。

5.1 タスクと資源

問題をスケジュールし、計画することは、与えられたタスクの集合の実行のための計画を決定することに関係があります。目的は、与えられた、タスクの順序とか限られている資源についての制約条件を満たす、実行可能なスケジュールを生成したり、スケジュールの makespan などの、与えられたクライテリオンを最適化することです。

Xpress-Kalis は、標準のスケジュール作成問題の定式化を簡素化するために、いくつかの全体的なモデル作成オブジェクト(aggregate modeling objects)を定義します。タスク(処理/加工の作業、アクティビティ)は、タイプ cptask によって示され、また、資源(原材料、機械など)は、タイプ cpresource によって示されます。

これらのスケジュール作成オブジェクトを使うには、オブジェクトと、タスク期間や資源の使用のような、それらの特性を記述するだけで十分です。なぜなら、(implicit constraints と呼ばれる)Xpress-Kalis の機能により、必要な制約条件の関係は自動的にセットアップされるからです。以下のセクションでは、下記の例を使い、このメカニズムの使い方を説明します。

- このスケジュール作成問題(Section 5.2 のプロジェクト・スケジューリング問題)は、非常に簡単なケースで、タスクとタスク間の順序についての制約条件だけが関っています。
- Section 5.3 の disjunctive scheduling/sequencing problem 問題では、タスクは、同一の資源を使用するので、相互に排他的です。
- 資源は、与えられたキャパシティの範囲内で、同時に使用可能なケースです(cumulative resources, see Section 5,4)。
- 資源の種類の区分の一つは、再生可能(renewable)な資源か、そうではない(non-renewable)資源かの区分です(Section 5.5)。
- 例えば、順序依存の段取りのような、標準問題のいろいろな拡張が可能です(Section 5.6)。

cp_schedule を使って列挙を行うと、ソルバーは、対応する(スケジュール作成)問題のタイプに適した

特別なサーチ戦略を使います。これらの戦略を parameterize することもできますし、また、スケジュール作成オブジェクトのための、ユーザ自身のサーチ戦略を定義することも可能です(see Section 5.7)。あるいはまた、標準最適化ファンクション cp_minimize / cp_maximize を使用することもできます。この場合、列挙は、スケジュール作成オブジェクトによって提供される構造的な情報を利用せず、決定変数のみを使って列挙します。
例えば、制約条件の定義では、スケジュール作成オブジェクトの(タスクの開始時刻や処理時間などの)特性にアクセスし、使うことができます。こうして、ユーザは、事前に定義された標準の問題を、他のタイプの制約条件で拡張できる可能性を享受できます。Xpress-Kalisでは、type cpvarの決定変数についてのdedicated global constraintsを使って、ユーザが、aggregate modeling objectを使わずに、ユーザのスケジュール作成問題を定式化できるので、さらに大きい柔軟性が得られます。こうした理由で、本章のほとんどの例では、scheduling objectを使ったインプリメンテーション、scheduling object を使を使わないインプリメンテーションという、二つの異なるインプリメンテーションが示されています。

5.2 先行制約条件 (Precedence constraints)

おそらく、スケジュール作成問題の最も基本的なタイプは、先行制約条件によってリンクされるタスクの集合を計画することことでしょう。

このセクションで取り上げる問題は、Applications of optimization with Xpress-MP という本の Section 7.1 に記載されている「Construction of a stadium」という問題です。

ある建設会社は、スタジアムを建築する契約を取り、最も短かい時間で完成したいと考えています。Table 5.1 は、主要なタスクと、それらのタスクに掛かる時間を、週単位でリストしたものです。タスクの中には、Table 5.1 に示されている他のあるタスクの完成の後に開始できるものもあります。

Table 5.1: Data for stadium construction

	Task Description	Duration	Predecessors
1	Installing the construction site	2	none
2	Terracing	16	1
3	Constructing the foundations	9	2
4	Access roads and other networks	8	2
5	Erecting the basement	10	3
6	Main floor	6	4.5
7	Dividing up the changing rooms	2	4
8	Electrifying the terraces	2	6
9	Constructing the roof	9	4, 6

10	Lighting of the stadium	5	4
11	Installing the terraces	3	6
12	Sealing the roof	2	9
13	Finishing the changing rooms	1	7
14	Constructing the ticket office	7	2
15	Secondary access roads	4	4, 14
16	Means of signalling	3	8, 11, 14
17	Lawn and sport accessories	9	12
18	Handing over the building	1	17

5.2.1 モデルの定式化

この問題は、古典的なプロジェクト・スケジューリング問題です。 最後に、プロジェクトの完了に対応する期間 0 の架空のタスクを加えます。 こうして、ここでは、 *TASKS = {1, . . . , N}* について考えますが、N は、この架空のタスクです。

工事のタスク $j(j \in TASKS)$は、すべて、タスクオブジェクト $task_j$で示され、このタスクオブジェクトは、可変の開始時刻 $task_j.start$、および、与えられた値 DUR_jに固定された工事期間を持っています。タスク間の工事の順序は、*arcs (i, j)* を持った precedence graph with によって表され、タスク *i* がタスク *j* に優先するということを意味します。

目的関数は、プロジェクトの完成までの時間、すなわち、最後の、そして、架空のタスク *N* の開始時刻を最小にする(できるだけ早くする)ことです。こうして、下記のモデルを得ますが、ここで、開始時間についての upper bound *HORIZON* は、すべてのタスク期間の合計によって与えられます。

tasks $task_j(j \in tasks)$

minimize taskN.start

$$\forall jTASKS : task_j.start \in \{0,..., HORIZON\}$$

$$\forall jTASKS : task_j.duration = DUR_j$$

$$\forall jTASKS : task_j.predecessors = \mathrm{Y}_{j \in TASKS, s.t. \exists ARC_{ij}} \{task_i\}$$

5.2.2 インプリメンテーション

以下のモデルは、Xpress-Kalis によるこの問題のインプリメンテーションを示しています。side-constraints がないので、このスケジュールの最も早い、可能な完成時期は架空のタスク *N*を最も早く始めることになります。

task-related constraint のプロパゲーションをしどうさせるために、ファンクション cp_propagate をコールします。

ここで、架空の end task の開始を、その lower bound の制約することで、constraint propagation の効果により、すべてのタスク開始時期が、その feasible interval に減少し、そして、クリティカルパスの上にあるタスクの開始時期は、ただ一つの値に固定されます。

その後の最小化へのコール(call to minimization)は、IVE でソリューションのグラフ表示を可能にするために、すべての変数をただ一つの値に例示化する(instantiating all variables with a single value)ためのみのものです。

```
model "B-1 Stadium construction (CP)"
 uses "kalis"

 declarations
  N = 19                              ! Number of tasks in the project
                                      ! (last = fictitious end task)
  TASKS = 1..N
  ARC: array(range,range) of integer  ! Matrix of the adjacency graph
  DUR: array(TASKS) of integer        ! Duration of tasks
  HORIZON : integer                   ! Time horizon

  task: array(TASKS) of cptask        ! Tasks to be planned
  bestend: integer
 end-declarations

 initializations from 'Data/b1stadium.dat'
  DUR ARC
 end-initializations

 HORIZON:= sum(j in TASKS) DUR(j)

! Setting up the tasks
 forall(j in TASKS) do
  setdomain(getstart(task(j)), 0, HORIZON)   ! Time horizon for start times
  set_task_attributes(task(j), DUR(j))       ! Duration
  setsuccessors(task(j), union(i in TASKS | exists(ARC(j,i))) {task(i)})
 end-do                                      ! Precedences

 if not cp_propagate then
  writeln("Problem is infeasible")
  exit(1)
 end-if

! Since there are no side-constraints, the earliest possible completion
! time is the earliest start of the fictitiuous task N
 bestend:= getlb(getstart(task(N)))
 getstart(task(N)) <= bestend
 writeln("Earliest possible completion time: ", bestend)

! For tasks on the critical path the start/completion times have been fixed
```

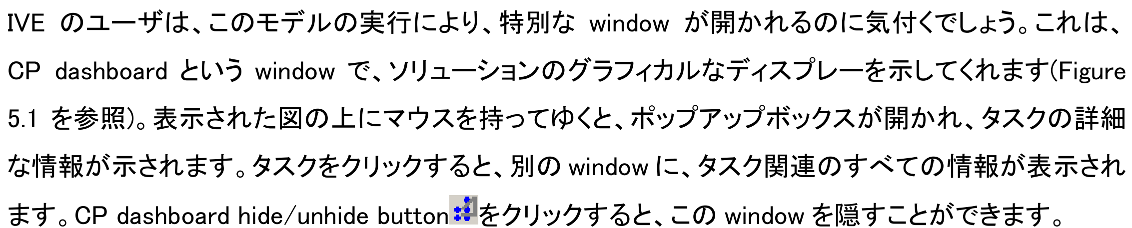
```
! by setting the bound on the last task. For all other tasks the range of
! possible start/completion times gets displayed.
 forall(j in TASKS) writeln(j, ": ", getstart(task(j)))

! Complete enumeration: schedule every task at the earliest possible date
 result:= cp_minimize(getstart(task(N)))
 forall(j in TASKS) writeln(j, ": ", getstart(task(j)))

end-model
```

あるタスクの先行タスクを示すことの代わりに、下記のように、すべてのタスクの後続タスクの集合を示すことで、同じように、precedence constraint を、はっきり規定できます。

```
setsuccessors(task(j), union(i in TASKS | exists(ARC(j,i))) {task(i)})
```

5.2.3 結果

スタジアム工事の最早な工事期間は 64 週間です。



IVE のユーザは、このモデルの実行により、特別な window が開かれるのに気付くでしょう。これは、CP dashboard という window で、ソリューションのグラフィカルなディスプレーを示してくれます(Figure 5.1 を参照)。表示された図の上にマウスを持ってゆくと、ポップアップボックスが開かれ、タスクの詳細な情報が示されます。タスクをクリックすると、別の window に、タスク関連のすべての情報が表示されます。CP dashboard hide/unhide button をクリックすると、この window を隠すことができます。

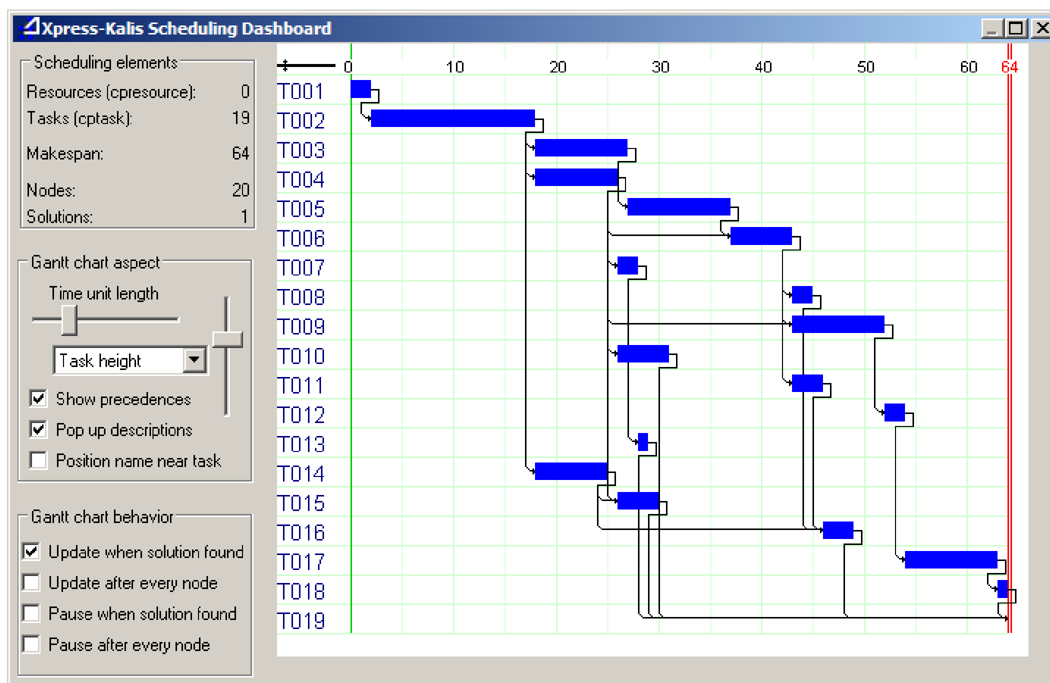


Figure 5.1: CP dashboard in IVE displaying the solution

5.2.4 オブジェクトをスケジュールしない代替的な定式化

前の定式化では、*TASKS={1、…N}*というタスクの集合を使いました。ここで、*N* は、完了を示す架空のタスクでした。ここで、あらゆるタスク *j* に関して、決定変数 $start_j$を導入し、タスクの開始時刻を示します。また、DUR_j、すなわち、タスク *j* の期間を使い、タスクの順序関係、「タスク *i* はタスク *j* に先行する」を下記の制約条件で記述します。

$$start_i + DUR_i \le start_j$$

こうして、このプロジェクト・スケジューリング問題のモデルは下記のようになります。

$$\text{minimize} \quad start_N$$
$$\forall j \in TASKS : start_j \in \{0,...,HORIZON\}$$
$$\forall i,j \in TASKS, \exists ARC_{ij} : start_i + DUR_i \le start_j$$

以下に、対応する Mosel モデルをすべて記します。precedence constraint を明示的に使っているのに注意してください。実行不可能なデータの場合、これは、インフィージビリティの原因をトレースするのに役立ちます。

```
model "B-1 Stadium construction (CP)"
 uses "kalis"

 declarations
  N = 19                            ! Number of tasks in the project
                                    ! (last = fictitious end task)
  TASKS = 1..N
  ARC: array(range,range) of integer  ! Matrix of the adjacency graph
  DUR: array(TASKS) of integer      ! Duration of tasks
  HORIZON : integer                 ! Time horizon

  start: array(TASKS) of cpvar      ! Start dates of tasks
  bestend: integer
 end-declarations

 initializations from 'Data/b1stadium.dat'
  DUR ARC
 end-initializations

 HORIZON:= sum(j in TASKS) DUR(j)

 forall(j in TASKS) do
  0 <= start(j); start(j) <= HORIZON
 end-do

! Task i precedes task j
 forall(i, j in TASKS | exists(ARC(i, j))) do
```

```
  Prec(i,j):= start(i) + DUR(i) <= start(j)
  if not cp_post(Prec(i,j)) then
   writeln("Posting precedence ", i, "-", j, " failed")
   exit(1)
  end-if
 end-do

! Since there are no side-constraints, the earliest possible completion
! time is the earliest start of the fictitiuous task N
 bestend:= getlb(start(N))
 start(N) <= bestend
 writeln("Earliest possible completion time: ", bestend)

! For tasks on the critical path the start/completion times have been fixed
! by setting the bound on the last task. For all other tasks the range of
! possible start/completion times gets displayed.
 forall(j in TASKS) writeln(j, ": ", start(j))

end-model
```

5.3 離接的なスケジュール作成 (Disjunctive scheduling) : unary resources

Section 3.5 で説明されている単一マシン上にジョブを順序付けるという問題は、モデル作成のための「task オブジェクト」と「resource オブジェクト」を使い、離接的なスケジュール作成問題(disjunctive scheduling problem) として表現できます。

読者の皆さんは、この問題が、単一マシンにおける非割込み型(non-preemptive)のタスク(ジョブ)の集合の処理の順序をスケジュールすることであると思い出してください。すべてのタスクjのリリース日、処理に必要な期間、および、完了日が与えられています。この問題は、全体の完了期間(makespan)、平均完成時間、または、遅延の合計を最小にするという、 3 つの異なる目的関数で解くことが出来ます。

5.3.1 モデルの定式化

モデル定式化の大半は、スケジュール作成のための「task オブジェクト」と「resource オブジェクト」の定義から成ります。

あらゆる job $j\left(j \in JOBS = \{1,\ldots,NJ\}\right)$ は、タスクオブジェクト $task_j$によって表され、開始日は、*taskj.start({RELj, . . . , MAXTIME})* です。ここで、*MAXTIME* は、リリース日付とすべての処理時間の合計のような、十分に大きい値で、*duration* は、与えられた処理時間 DUR_j に固定されています。すべてのジョブは、切り離せないキャパシティ(unitary capacity)を持つ同一の資源 *res* を使用します。これは、ある時点で、最大でも、1 つのジョブしか処理できないことを意味しています。こうして、暗黙的に、ジョブ間の離接的な関係を示します。

task object によって確立されるもう一つの暗黙的な制約条件は、ジョブ j の開始時間、処理時間、完成時間との間の関係です。

$$\forall j \in: task_j.end = task_j.start + task_j.duration$$

目的関数 1: 最初の目的関数は、makespan(スケジュールの完成時間)を最小にすることです。これは、最後のジョブの完成時間を最も早くすることと同等です。こうして、モデル全体は以下のようになります。ここで、*MAXTIME* は、リリース日付とすべての処理時間の合計のような、十分に大きい値です。

$$\text{resource } res$$
$$\text{tasks } task_j \left(j \in JOBS \right)$$
$$\text{minimize } finish$$
$$finish = \max imum_{j \in JOBS} \left(task_j.end \right)$$
$$res.capacity = 1$$
$$\forall j \in JOBS : task_j.end \in \{0,..., MAXTIME\}$$
$$\forall j \in JOBS : task_j.start \in \{REL_j,..., MAXTIME\}$$
$$\forall j \in JOBS : task_j.duration = DUR_j$$
$$\forall j \in JOBS : task_j.requirement_{res} = 1$$

目的関数 2: 2 番目の目的関数は、平均処理時間を最小にすることですが、これは、ジョブの完成時間(job completion times)の合計を最小にすることと同等です。その定式化は、最初のモデルの目的関数と変わりませんが、すべてのジョブの完了時間の合計を示す変数 *totComp* を追加します。

$$\text{minimize } totComp$$
$$totComp = \sum_{j \in JOBS} task_j.end$$

目的関数3:遅延の合計を最小にする目的関数を定式化するために、新しい変数 $late_j$を導入し、ジョブが納期よりも遅れる時間を測定します。これらの変数の値は、ジョブ j の完成時間と納期 DUE_jの差に対応します。ジョブが期日以前終わる場合は、この変数の値はゼロです。 目的関数は、これらの遅延変数の合計を最小にすることです。

$$\text{minimize } totLate$$
$$totLate = \sum_{j \in JOBS} late_j$$
$$\forall j \in JOBS : late_j \in \{0,..., MAXTIME\}$$
$$\forall j \in JOBS : late_j \geq task_j.end - DUE_j$$

5.3.2 インプリメンテーション

下記の Xpress-Kalis(ファイル b4seq3_ka.mos)のインプリメンテーションは、どのように必要なタスク、および、資源のモデル作成オブジェクトをセットアップするかを示しています。資源キャパシティは手順

set_resource_attributes で設定され(資源はタイプ Kalis_UNARY_RESOURCE で、これは、一度に 1 ジョブを処理することを意味する)、また、資源は set_task_attributes で設定します。後者は、アーギュメント(タスクのアトリビュート)の異なる組み合わせにたいして、いくつかの overloaded version に存在しています。詳細については、Xpress-Kalis Reference Manual を参照してください。

maximum 制約式の定式化のためには、変数の(補助)リストを使用します。Xpress-Kalis は、union(j in JOBS) getend(task(j))のような set expressions で、ユーザが access functions を使い、getstart、getduration などのオブジェクトのモデルを作成することを許していません。

```
model "B-4 Sequencing (CP)"
 uses "kalis"

 forward procedure print_sol
 forward procedure print_sol3

 declarations
  NJ = 7                          ! Number of jobs
  JOBS=1..NJ

  REL: array(JOBS) of integer     ! Release dates of jobs
  DUR: array(JOBS) of integer     ! Durations of jobs
  DUE: array(JOBS) of integer     ! Due dates of jobs

  task: array(JOBS) of cptask     ! Tasks (jobs to be scheduled)
  res: cpresource                 ! Resource (machine)

  finish: cpvar                   ! Completion time of the entire schedule
 end-declarations

 initializations from 'Data/b4seq.dat'
  DUR REL DUE
 end-initializations

! Setting up the resource (formulation of the disjunction of tasks)
 set_resource_attributes(res, KALIS_UNARY_RESOURCE, 1)

! Setting up the tasks (durations and disjunctions)
 forall(j in JOBS) set_task_attributes(task(j), DUR(j), res)

 MAXTIME:= max(j in JOBS) REL(j) + sum(j in JOBS) DUR(j)

 forall(j in JOBS) do
  0 <= getstart(task(j)); getstart(task(j)) <= MAXTIME
  0 <= getend(task(j)); getend(task(j)) <= MAXTIME
 end-do

! Start times
 forall(j in JOBS) getstart(task(j)) >= REL(j)
```

```
!**** Objective function 1: minimize latest completion time ****
 declarations
  L: cpvarlist
 end-declarations

 forall(j in JOBS) L += getend(task(j))
 finish = maximum(L)

 if cp_schedule(finish) >0 then
  print_sol
 end-if


!**** Objective function 2: minimize average completion time ****
 declarations
  totComp: cpvar
 end-declarations

 totComp = sum(j in JOBS) getend(task(j))

 if cp_schedule(totComp) > 0 then
  print_sol
 end-if


!**** Objective function 3: minimize total tardiness ****
 declarations
  late: array(JOBS) of cpvar      ! Lateness of jobs
  totLate: cpvar
 end-declarations

 forall(j in JOBS) do
  0 <= late(j); late(j) <= MAXTIME
 end-do

! Late jobs: completion time exceeds the due date
 forall(j in JOBS) late(j) >= getend(task(j)) - DUE(j)

 totLate = sum(j in JOBS) late(j)
 if cp_schedule(totLate) > 0 then
  writeln("Tardiness: ", getsol(totLate))
  print_sol
  print_sol3
 end-if


!-----------------------------------------------------------------

! Solution printing
```

```
procedure print_sol
 writeln("Completion time: ", getsol(finish) ,
         "  average: ", getsol(sum(j in JOBS) getend(task(j))))
 write("Rel¥t")
 forall(j in JOBS) write(strfmt(REL(j),4))
 write("¥nDur¥t")
 forall(j in JOBS) write(strfmt(DUR(j),4))
 write("¥nStart¥t")
 forall(j in JOBS) write(strfmt(getsol(getstart(task(j))),4))
 write("¥nEnd¥t")
 forall(j in JOBS) write(strfmt(getsol(getend(task(j))),4))
 writeln
end-procedure

procedure print_sol3
 write("Due¥t")
 forall(j in JOBS) write(strfmt(DUE(j),4))
 write("¥nLate¥t")
 forall(j in JOBS) write(strfmt(getsol(late(j)),4))
 writeln
end-procedure

end-model
```

5.3.3 結果

このモデルは、Section 3.5 のモデルバージョンにたいしてレポートされたのと似ている結果を生み出します。Figure5.2 は、IVE によって作成されたソリューションのガントチャートディスプレーを示しています。 ガントチャートの上に、資源利用ディスプレーがあります。この機械は、中断なく、タスクによって使用されます。すなわち、リリース時間と納期に与えられた制約条件を緩和しても、もっと早く終わるスケジュールを生成させるのは可能ではありません。

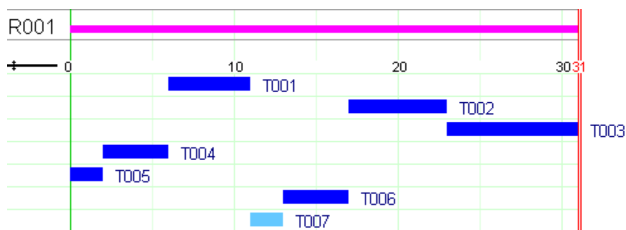


Figure 5.2: Solution display in IVE

5.4 Cumulative scheduling: discrete resources

このセクションで説明される問題は、「Applications of optimization with Xpress-MP」という本の Section 9.4 「Backing up files」からとったものです。

ここでの課題は、1.44Mb の容量の空のフロッピーディスクに、46kb、55kb、62kb、87kb、108kb、114kb、137kb、164kb、253kb、364kb、372kb、388kb、406kb、432kb、461kb、および、851kb のサイズの 16 個のファイルを保存するとき、ファイルを各ディスクにどのように配分したら、使用されるフロッピーディスクの数が最小になるか、その配分の方法を見つけることです。

5.4.1 モデルの定式化

この問題は、binpacking 問題と呼ばれている問題です。ここでは、この問題を、cumulative scheduling problemとして、どのように定式化し、といたらよいかを示します。この問題では、ディスクが資源であり、そして、ファイルがスケジュールされるタスクです。

フロッピーディスクは、それぞれ単一の、離散的な資源であると考えます。そこでは、あらゆるタイム・ユニットは 1 個のディスクを意味します（where every time unit stands for one disk)。資源キャパシティは、ディスク容量に対応しています。

ここでは、すべてのファイル f（$f \in FILES$）を、タスクオブジェクト $file_f$ で表します。ファイルは、1 ユニットの固定期間、所与のサイズ $SIZE_f$ に対応する資源リクワイアメントを持っています。こうして、タスクの`start' フィールドは、このファイルを保存するためのディスクの選択を示します。

目的関数は、使用されるディスクの数を最小にすることです。これは、タスクの`start' フィールドがとる最も大きい値(すなわち、ファイルを保存するのに使用されるディスクの数)を最小にすることに対応します。こうして、以下のモデルを得ます。

$$
\begin{gathered}
\text{resource } \mathit{disks} \\
\text{tasks } \mathit{files}_f \left(f \in FILES \right) \\
\text{minimize } \mathit{diskuse} \\
\mathit{diskuse} = \text{maximum}_{f \in FILES} \left(\mathit{file}_f.\mathit{start} \right) \\
\mathit{disks.capacity} = CAP \\
\forall f \in FILES : \mathit{file}_f.\mathit{start} \geq 1 \\
\forall f \in FILES : \mathit{file}_f.\mathit{duration} = 1 \\
\forall f \in FILES : \mathit{file}_f.\mathit{requirement}_{disks} = SIZE_f
\end{gathered}
$$

5.4.2 インプリメンテーション

Xpress-Kalis によるインプリメンテーションは、かなり簡単です。タイプ KALIS_DISCRETE_RESOURCE の資源を定義します。これは、キャパシティ総量を示します。タスクの定義は、前の例で見たものと似ています。

```
model "D-4 Bin packing (CP)"
 uses "kalis"
```

```
 declarations
  ND: integer                      ! Number of floppy disks
  FILES = 1..16                    ! Set of files
  DISKS: range                     ! Set of disks

  CAP:  integer                    ! Floppy disk size
  SIZE: array(FILES) of integer    ! Size of files to be saved

  file: array(FILES) of cptask     ! Tasks (= files to be saved)
  disks: cpresource                ! Resource representing disks
  L: cpvarlist
  diskuse: cpvar                   ! Number of disks used
 end-declarations

 initializations from 'Data/d4backup.dat'
  CAP SIZE
 end-initializations

! Provide a sufficiently large number of disks
 ND:= ceil((sum(f in FILES) SIZE(f))/CAP)
 DISKS:= 1..ND

! Setting up the resource (capacity limit of disks)
 set_resource_attributes(disks, KALIS_DISCRETE_RESOURCE, CAP)

! Setting up the tasks
 forall(f in FILES) do
  setdomain(getstart(file(f)), DISKS)          ! Start time (= choice of disk)
  set_task_attributes(file(f), disks, SIZE(f))  ! Resource (disk space) req.
  set_task_attributes(file(f), 1)    ! Duration (= number of disks used)
 end-do

! Limit the number of disks used
 forall(f in FILES) L += getstart(file(f))
 diskuse = maximum(L)

! Minimize the total number of disks used
 if cp_schedule(diskuse) = 0 then
  writeln("Problem infeasible")
 end-if

! Solution printing
 writeln("Number of disks used: ", getsol(diskuse))
 forall(d in 1..getsol(diskuse)) do
  write(d, ":")
  forall(f in FILES) write( if(getsol(getstart(file(f)))=d , " "+SIZE(f), ""))
  writeln("  space used: ",
         sum(f in FILES | getsol(getstart(file(f)))=d) SIZE(f))
 end-do
 cp_show_stats
```

```
end-model
```

5.4.3 結果

このモデルを走らせると、Table5.2 に示されている結果が得られます。すなわち、すべてのファイルをバックアップするには、3 枚のフロッピーディスクが必要です。

Table 5.2: Distribution of files to disks

Disk	File sizes (in kb)	Used space (in Mb)
1	46 87 137 164 253 364 388	1.439
2	55 62 108 372 406 432	1.435
3	114 461 851	1.426

IVE を使って結果をビジュアライズすると、Figure5.3 に示されているようなスクリーンが得られ、そこには、資源利用のプロフィールが上部に、そしてタスクのガントチャートが下部に示されます。

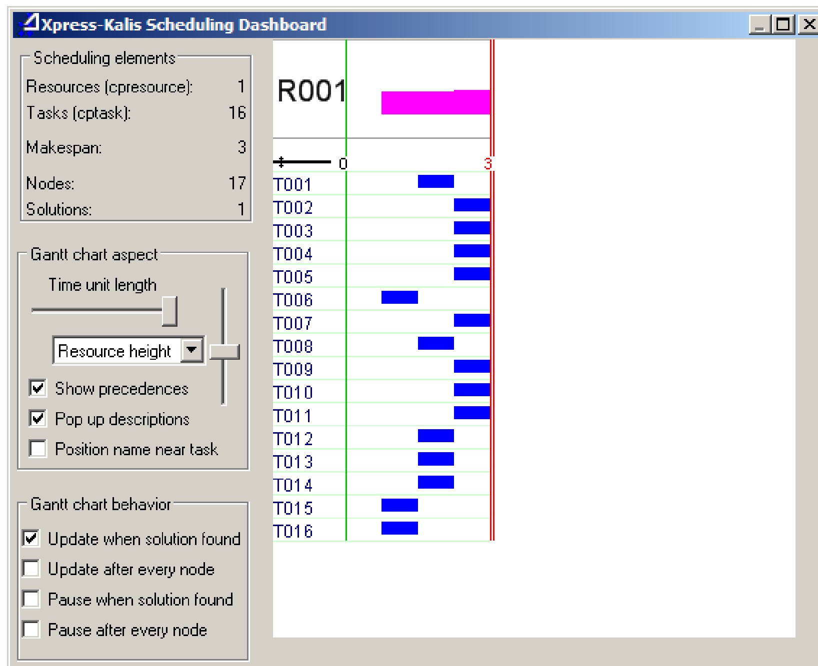


Figure 5.3: CP dashboard in IVE displaying the solution

5.4.4 オブジェクトをスケジュールしない代替的な定式化

この問題を、タスクと資源オブジェクトを定義することをする代わりに、事前に定義されたオブジェクトに分類することなく、タスクの異なった属性を表す標準の有限ドメイン変数のに関して、`cumulative'

constraintを使って定式化することもできます。単一の`cumulative' constraintは、アーギュメント間の以下の関係を確立することによって、一つの離散的な資源に関するタスクの集合のスケジューリング問題を表現します(タスクの5つの特性、すなわち、開始、処理期間、終了、資源の利用、および、サイズに関する決定変数の5つのarrayで、すべて、同一の集合R、および、コンスタント、もしくは、time-indexed resource capacityによりインデックスされている)。

$$\forall j \in R : start_j + duration_j = end_j$$
$$\forall j \in R : use_j \cdot duration_j = size_j$$
$$\forall t \in TIMES : \sum_{j \in R | t \in [UB(start_j)..LB(end_j)]} use_j \leq CAP_t$$

ここで、UB は決定変数の`upper bound'を、LB は`lower bound'意味しています。

$save_f$ はファイル f の保存に使用されるディスクを、そして、use_f はファイル f によって使われるスペースを示すものとします。scheduling object と同じように、タスクの`start'の特性は、ファイルを保存するために選ばれたディスクに対応し、タスクの resource requirement はファイルサイズです。すべてのファイルを単一のディスクに保存したいので、'duration' $dur_f$ を 1 に固定します。`cumulative' constraint の定式化に必要な残りのタスク特性、すなわち、`end' と`size' (e_f 、および、s_f)は、この問題の定式化には、必要ではありません。なぜなら、これらの値は、他の 3 つの特性で決定されるからです。

5.4.5 インプリメンテーション

以下の Mosel モデルは、cumulative constraint を使用する、二番目のモデルバージョンを実行します。

```
model "D-4 Bin packing (CP)"
 uses "kalis", "mmsystem"

 setparam("default_lb", 0)

 declarations
  ND: integer                          ! Number of floppy disks
  FILES = 1..16                        ! Set of files
  DISKS: range                         ! Set of disks

  CAP:  integer                        ! Floppy disk size
  SIZE: array(FILES) of integer        ! Size of files to be saved

  save: array(FILES) of cpvar         ! Disk a file is saved on
  use: array(FILES) of cpvar          ! Space used by file on disk
  dur,e,s: array(FILES) of cpvar      ! Auxiliary arrays for 'cumulative'
  diskuse: cpvar                      ! Number of disks used
```

```
  Strategy: array(FILES) of cpbranching ! Enumeration
  FSORT: array(FILES) of integer
 end-declarations

 initializations from 'Data/d4backup.dat'
  CAP SIZE
 end-initializations

! Provide a sufficiently large number of disks
 ND:= ceil((sum(f in FILES) SIZE(f))/CAP)
 DISKS:= 1..ND
 finalize(DISKS)

! Limit the number of disks used
 diskuse = maximum(save)

 forall(f in FILES) do
  setdomain(save(f), DISKS)        ! Every file onto a single disk
  use(f) = SIZE(f)
  dur(f) = 1
 end-do

! Capacity limit of disks
 cumulative(save, dur, e, use, s, CAP)

! Definition of search (place largest files first)
 qsort(SYS_DOWN, SIZE, FSORT)       ! Sort files in decreasing order of size
 forall(f in FILES)
  Strategy(f):=       assign_var(KALIS_SMALLEST_MIN,        KALIS_MIN_TO_MAX,
{save(FSORT(f))})
 cp_set_branching(Strategy)

! Minimize the total number of disks used
 if not cp_minimize(diskuse) then
  writeln("Problem infeasible")
 end-if

! Solution printing
 writeln("Number of disks used: ", getsol(diskuse))
 forall(d in 1..getsol(diskuse)) do
  write(d, ":")
  forall(f in FILES) write( if(getsol(save(f))=d , " "+SIZE(f), ""))
  writeln("  space used: ", sum(f in FILES | getsol(save(f))=d) SIZE(f))
 end-do

end-model
```

このモデルの実行で得られるソリューションは、同一の目的関数の値を持っていますが、ディスクへのファイルの配布は、前のそれと、「すべて、まったく同一」ではありません。この問題は、いくつかの異なった最適解を持っています。特に、ディスクの番号を変えた場合に得られる解がそうです。このよう

な場合、サーチを短くするために、symmetry breaking constraint を追加することはよいことです。symmetry breaking constraint は、実行可能解の一部を取り除き、サーチスペースのサイズを小さくしてくれます。例えば、現在の例では、最も大きいファイルを最初のディスクに割り当て、二番目に大きいファイルを最初の二つのディスクの片方に割り当て、・・・、これを、必要なディスクの数の lower bound に到達するまで行います（保存の lower bound estimate は、ファイル・サイズの合計をディスク容量で割り、端数を切り上げて、次に大きい整数にすることで得られる）。

5.5 再生可能（renewable）な資源、再生可能ではない（non-renewable）資源

これまでのセクションで見た`disjunctive--cumulative'、`unary--discrete'の区分のほかに、資源を表現したり分類したりする方法があります。もう一つの重要な特性は、再生可能（renewable）な資源、再生可能ではない（non-renewable）資源という概念です。これまでの例は、(機械や作業員などの)再生可能な資源の例でした。つまり、あるタスクによって使用されていた資源の量は、そのタスクが完成するとリリースされて、他のタスクのために利用可能になります。(例えば、お金、原料、中間製品などの)再生可能ではない（non-renewable）資源の場合は、資源を使用するタスクは、それを消費してしまいます。すなわち、利用可能な量の資源は、タスクを処理することで使われた量だけ、減少してしまいます。

また、タスクは、資源を使用するのではなく、ある量の資源を生産するかもしれません。タスクの中には、その実行の間に、ある量の資源を生み出すタスク(再生可能な資源)や、生産の結果を、資源のストックに加えるタスク(再生可能ではない資源)もあります。

ここで、以下の問題を、どのように定式化するか、その方法について考えてみましょう。ここに、P1 から P5 までの 5 つのジョブがあり、これらは、ある生産プロセスの二つの段階を表していますが、いま、これらの 5 つのジョブをスケジュールしたいと考えているとします。P1 と P2 は、(P5 から P3 までの)第二段階のジョブで必要な中間製品を生産します。それぞれのジョブの最小、最大の処理時間、費用、そして、第二段階のジョブに関しては、その利益貢献が与えられています。ケースは 2 つあります。すなわち、モデル A: 第一段階のジョブは、それらの実行の間のあらゆる時点で、所与の量の(例えば、電気、熱、蒸気などのような)中間製品を生産しますが、これらの中間製品は、即座に、最終段階のジョブで消費されます。モデル B: 中間製品は、第一段階のジョブのアウトプットとして生れ、第二段階のジョブを開始するためのインプットとして必要になります。

モデルＡの中間製品は、再生可能な資源です、そして、モデルＢでは、私たちは再生可能ではない資源のケースです。

5.5.1 モデルの定式化

FIRST = {*P*1, *P*2} で、第一段階のジョブの集合、*FINAL* = {*P*3, *P*4, *P*5} で、第二段階のジョブの集合を、集合 *JOBS* ですべてのジョブを示すものとしましょう。すべてのジョブ *j* に関して、その最小の処理時間

$MIND_j$、および、最大の処理時間 $MAXD_j$ が与えられています。$RESAMT_j$ は、ジョブにインプットとして必要な資源量、もしくは、ジョブからアウトプットとして生れる資源量です。また、第一段階のジョブ j の費用 $COST_j$、第二段階のジョブ j の利益 $PROFIT_j$ も分かっています。

モデル A（再生可能な資源）

再生可能な資源のケースは、以下のモデルのように定式化されます。資源キャパシティが、0 に設定されることに注意してください。これは、利用可能な資源の量が、第一生産段階のジョブで生産されたもののみが利用可能であることを示しています。

$$
\begin{gathered}
\texttt{resource} \quad res \\
\texttt{tasks} \quad task_j \left(j \in JOBS \right) \\
\texttt{maximize} \quad \sum_{j \in JOBS} \left(PROFIT_j - COST_j \right) \times task_j.duration \\
res.capacity = 0 \\
\forall j \in JOBS : task_j.start, task_j end \in \{0, \ldots, HORIZON\} \\
\forall j \in JOBS : task_j.duration \in \{MOND_j, \ldots, MAXD_j\} \\
\forall j \in FIRST : task_j.provision_{res} = RESAMT_j \\
\forall j \in FINAL : task_j.requirement_{res} = RESAMT_j
\end{gathered}
$$

モデル B（再生可能ではない資源）

モデル A を参考にして、以下のようにモデルBのケースを定式化します。

$$
\begin{gathered}
\texttt{resource} \quad res \\
\texttt{tasks} \quad task_j \left(j \in JOBS \right) \\
\texttt{maximize} \quad \sum_{j \in JOBS} \left(PROFIT_j - COST_j \right) \times task_j.duration \\
res.capacity = 0 \\
\forall j \in JOBS : task_j.start, task_j end \in \{0, \ldots, HORIZON\} \\
\forall j \in JOBS : task_j.duration \in \{MOND_j, \ldots, MAXD_j\} \\
\forall j \in FIRST : task_j.production_{res} = RESAMT_j \\
\forall j \in FIRST : task_j.consumption_{res} = RESAMT_j
\end{gathered}
$$

しかし、このモデルは、中間製品の生産がタスクの開始時点で始まるので、上の問題に完全には対応していません。この問題を直すために、第一段階でのすべてのジョブ j に、補助タスク End_j を導入ましょう。この補助のジョブは、処理時間 0、もともとのジョブと同一の完成時間を持っており、もともとのジョブのところで中間製品を生産します。

$$
\begin{gathered}
\forall j \in FIRST : task_{End_j}.end = task_j.end \\
\forall j \in FIRST : task_{End_j}.duration = 0
\end{gathered}
$$

$$\forall j \in FIRST : task_j.production_{res} = 0$$
$$\forall j \in FIRST : task_{End_j}.production_{res} = RESAMT_j$$

5.5.2 インプリメンテーション

以下の Mosel モデルでケース A を実行します。ここでは、ディフォルトスケジュール作成ソルバー(function cp_schedule)を使い、オプションの 2 番目のアーギュメントに値 true を入れ、目的関数を最大にしたいことを知らせます。

```
model "Renewable resource"
 uses "kalis", "mmsystem"

 forward procedure solution_found

 declarations
  FIRST = {'P1','P2'}
  FINAL = {'P3','P4','P5'}
  JOBS = FIRST+FINAL

  MIND,MAXD: array(JOBS) of integer  ! Limits on job durations
  RESAMT: array(JOBS) of integer  ! Resource use/production
  HORIZON: integer                ! Time horizon
  PROFIT: array(FINAL) of real    ! Profit from production
  COST: array(JOBS) of real       ! Cost of production
  CAP: integer                    ! Available resource quantity

  totalProfit: cpfloatvar
  task: array(JOBS) of cptask     ! Task objects for jobs
  intermProd: cpresource         ! Non-renewable resource (intermediate prod.)
 end-declarations

 initializations from 'Data/renewa.dat'
  [MIND,MAXD] as 'DUR' RESAMT HORIZON PROFIT COST CAP
 end-initializations

! Setting up resources
 set_resource_attributes(intermProd, KALIS_DISCRETE_RESOURCE, CAP)
 setname(intermProd, "IntP")

! Setting up the tasks
 forall(j in JOBS) do
  setname(task(j), j)
  setduration(task(j), MIND(j), MAXD(j))
  setdomain(getend(task(j)), 0, HORIZON)
 end-do
```

```
! Providing tasks
 forall(j in FIRST) provides(task(j), RESAMT(j), intermProd)

! Requiring tasks
 forall(j in FINAL) requires(task(j), RESAMT(j), intermProd)

! Objective function: total profit
 totalProfit = sum(j in FINAL) PROFIT(j)*getduration(task(j)) -
               sum(j in JOBS) COST(j)*getduration(task(j))
 cp_set_solution_callback("solution_found")

! Solve the problem
 starttime:= gettime
 if cp_schedule(totalProfit,true)=0 then
  exit(1)
 end-if

! Solution printing
 writeln("Total profit: ", getsol(totalProfit))
 writeln("Job¥tStart¥tEnd¥tDuration")
 forall(j in JOBS)
  writeln(j, "¥t ", getsol(getstart(task(j))), "¥t ", getsol(getend(task(j))),
          "¥t ", getsol(getduration(task(j))))

 procedure solution_found
  writeln(gettime-starttime , " sec. Solution found with total profit = ",
          getsol(totalProfit))
  forall(j in JOBS)
   write(j, ": ", getsol(getstart(task(j))), "-", getsol(getend(task(j))),
          "(", getsol(getduration(task(j))), "), ")
  writeln
 end-procedure

end-model
```

ケース B のモデルでは、二つの補助のタスクを加え、集合 ENDFIRST を作り、第一段階のジョブの完成をマークします。唯一のもう一つの違いは、資源制約条件を定義するタスク特性 produces、consumes です。ここでは、モデルの関連部分のみを繰り返します。

```
declarations
  FIRST = {'P1','P2'}
  ENDFIRST = {'EndP1', 'EndP2'}
  FINAL = {'P3','P4','P5'}
  JOBS = FIRST+ENDFIRST+FINAL

  MIND,MAXD: array(JOBS) of integer  ! Limits on job durations
  RESAMT: array(JOBS) of integer   ! Resource use/production
  HORIZON: integer                 ! Time horizon
  PROFIT: array(FINAL) of real     ! Profit from production
  COST: array(JOBS) of real        ! Cost of production
```

```
  CAP: integer                    ! Available resource quantity

  totalProfit: cpfloatvar
  task: array(JOBS) of cptask     ! Task objects for jobs
  intermProd: cpresource         ! Non-renewable resource (intermediate prod.)
 end-declarations

 initializations from 'Data/renewb.dat'
  [MIND,MAXD] as 'DUR' RESAMT HORIZON PROFIT COST CAP
 end-initializations

! Setting up resources
 set_resource_attributes(intermProd, KALIS_DISCRETE_RESOURCE, CAP)
 setname(intermProd, "IntP")

! Setting up the tasks
 forall(j in JOBS) do
  setname(task(j), j)
  setduration(task(j), MIND(j), MAXD(j))
  setdomain(getend(task(j)), 0, HORIZON)
 end-do

! Production tasks
 forall(j in ENDFIRST) produces(task(j), RESAMT(j), intermProd)
 forall(j in FIRST) getend(task(j)) = getend(task("End"+j))

! Consumer tasks
 forall(j in FINAL) consumes(task(j), RESAMT(j), intermProd)
```

5.5.3 結果

ディフォルト・サーチの行動、2 つのモデルの結果はかなり異なっています。Figure5.5 に示されている、ケース B の目的関数の値 344.9 を持つ最適解は、1 秒以下で証明されます。ケース A の良いソリューションを得るには、標準の PC で、数秒、掛かります。 最適解(Figure5.4)を見つけて、最適性を証明するには、数分が必要です。このような芳しくないサーチの行動の主な理由は、目的関数の選択です。費用ベースの目的関数は、よく propagate せず、したがって、サーチツリーをカットするのにうまく役立ちません。一般に、スケジュール作成問題での目的関数のより良い選択は、タスク決定変数(start, duration, or completion time, particularly the latter)を含む評価基準です。

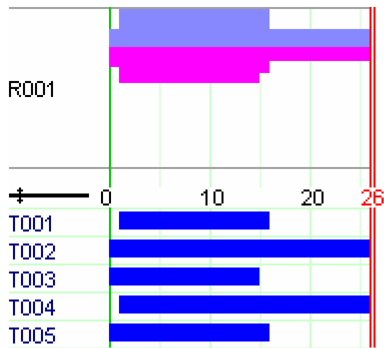


Figure 5.4: Solution for case A (resource provision and requirement)

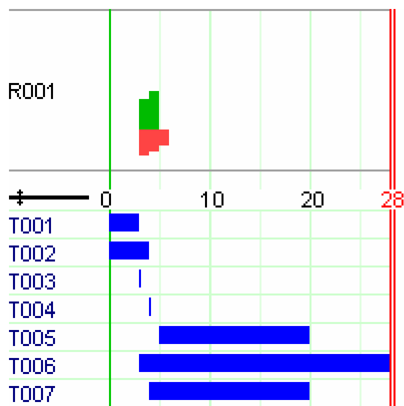


Figure 5.5: Solution for case B (resource production and consumption)

5.5.4 オブジェクトをスケジュールしない代替的な定式化

この問題の定式化は、タスクオブジェクトと資源オブジェクトを定義することの代わりに、標準の有限ドメイン変数に関する`producer–consumer' constraint を使っても定式化できます。ここで、標準の有限ドメイン変数は、事前に定義されたオブジェクトに分類されることなく、タスクの異なる、個々の属性を表します。たった一つの`producer–consumer' constraint により、再生可能ではない資源を生産、もしくは、消費するタスクの集合のスケジュール作成問題を、アーギュメント間の以下の関係を確立することによって表現します(タスクに関する 7 つの特性は、start, duration, end, per unit and cumulated resource production, per unit and cumulated resource consumption の 7 つで、すべて、同一の集合 *R* でインデックスされている)。

$$\forall j \in R : start_j + duration_j = end_j$$
$$\forall j \in R : produce_j \cdot duration_j = psize_j$$
$$\forall j \in R : consume_j \cdot duration_j = csize_j$$
$$\forall t \in TIMES : \sum_{j \in R | t \in [UB(start_j)..LB(end_j)]} (produce_j - consume_j) \geq 0$$

ここで、UB は決定変数の`upper bound'、LB は決定変数の`lower bound'を表します。

5.5.5 インプリメンテーション

以下の Mosel モデルは、producer_consumer constraint を使って、二番目のモデル・バージョンを実行します。

```
model "Non-renewable resource"
 uses "kalis"

 setparam("DEFAULT_LB", 0)

 declarations
  FIRST = {'P1','P2'}
  ENDFIRST = {'EndP1', 'EndP2'}
  FINAL = {'P3','P4','P5'}
  JOBS = FIRST+ENDFIRST+FINAL
  PCJOBS = ENDFIRST+FINAL

  MIND,MAXD: array(JOBS) of integer  ! Limits on job durations
  RESAMT: array(JOBS) of integer  ! Resource use/production
  HORIZON: integer                ! Time horizon
  PROFIT: array(FINAL) of real    ! Profit from production
  COST: array(JOBS) of real       ! Cost of production
  CAP: integer                    ! Available resource quantity

  totalProfit: cpfloatvar
  fstart,fdur,fcomp: array(FIRST) of cpvar! Start, duration & completion of jobs
  start,dur,comp: array(PCJOBS) of cpvar  ! Start, duration & completion of jobs
  produce,consume: array(PCJOBS) of cpvar ! Production/consumption per time unit
  psize,csize: array(PCJOBS) of cpvar     ! Cumulated production/consumption
 end-declarations

 initializations from 'Data/renewb.dat'
  [MIND,MAXD] as 'DUR' RESAMT HORIZON PROFIT COST CAP
 end-initializations

! Setting up the tasks
 forall(j in PCJOBS) do
  setname(start(j), j)
  setdomain(dur(j), MIND(j), MAXD(j))
  setdomain(comp(j), 0, HORIZON)
  start(j) + dur(j) = comp(j)
 end-do
 forall(j in FIRST) do
  setname(fstart(j), j)
  setdomain(fdur(j), MIND(j), MAXD(j))
  setdomain(fcomp(j), 0, HORIZON)
```

```
  fstart(j) + fdur(j) = fcomp(j)
 end-do

! Production tasks
 forall(j in ENDFIRST) do
  produce(j) = RESAMT(j)
  consume(j) = 0
 end-do
 forall(j in FIRST) fcomp(j) = comp("End"+j)

! Consumer tasks
 forall(j in FINAL) do
  consume(j) = RESAMT(j)
  produce(j) = 0
 end-do

! Resource constraint
 producer_consumer(start, comp, dur, produce, psize, consume, csize)

! Objective function: total profit
 totalProfit = sum(j in FINAL) PROFIT(j)*dur(j) -
               sum(j in FIRST) COST(j)*fdur(j)

 if not cp_maximize(totalProfit) then
  exit(1)
 end-if

 writeln("Total profit: ", getsol(totalProfit))
 writeln("Job¥tStart¥tEnd¥tDuration")
 forall(j in FIRST)
  writeln(j, "¥t ", getsol(fstart(j)), "¥t ", getsol(fcomp(j)),
          "¥t ", getsol(fdur(j)))
 forall(j in PCJOBS)
  writeln(j, "¥t ", getsol(start(j)), "¥t ", getsol(comp(j)),
          "¥t ", getsol(dur(j)))

end-model
```

このモデルは、前のモデルバージョンと同じソリューションを生成しますが、時間は、少しだけ長く掛かります(もっとも、それでも、標準の PC 上で、1 秒以下ですが・・・)。

5.6 拡張: 段取り時間

もう一度、Section 3.8 で使ったペイントのバッチの生産計画の問題について考えましょう。2 つのバッチの処理の間で、機械をクリーニングする必要があります。 クリーニング(または、セットアップ)は、順序に依存していて、非対称(asymetric)です。 目的関数は、最も短い生産サイクルを決定することです。

5.6.1 モデルの定式化

タスクオブジェクト $taskj$ で表わされたすべてのジョブ j（$j \in JOBS = \{1,...,NJ\}$)にたいして、その処理時間 DUR_j が与えられています。また、エントリー $CLEAN_{jk}$ を持ったクリーニング時間のマトリクス $CLEAN$ があり、これにより、タスク k がタスク j に続くときのクリーニング時間が与えられています。ジョブを処理する機械は、unitary capacity の資源 res としてモデル化されます。こうして、ジョブ間の disjunction が示されます。

makespan(最後のバッチの完成時間)を最小にする（最も早く完了する）という目的関数で、以下のモデルを得ます。

$$
\begin{gathered}
\texttt{resource } res \\
\texttt{tasks } tasks_j (j \in JOBS) \\
\texttt{minimize } finish \\
finish = maximum_{j \in JOBS} (task_j.end) \\
Res.capacity = 1 \\
\forall j \in JOBS : task_j.duration = DUR_j \\
\forall j \in JOBS : task_j.requirement_{res} = 1 \\
\forall j,k \in JOBS : setup(task_j.task_k) = CLEAN_{jk}
\end{gathered}
$$

元の問題の定式化でやっかいなことは、サイクルタイムを最小にしたいと思うということ自体です。すなわち、最後のジョブの完了＋最後のジョブとそれ以前のジョブの順序で影響されるセットアップ時間を最小にすることです。ここでのタスクベースのモデルには、ジョブの順序やジョブのランクについての情報が含まれていないので、生産サイクルの最初に位置するジョブと最後に位置するのジョブのインデックス値のために、補助変数 firstjob と lastjob を導入し、そして、最初のタスクと最後のタスクの間のセットアップ作業の時間のための、もう一つの変数 cleanlf を導入します。以下の制約条件は、これらの変数とタスクオブジェクトとの関係を表しています。

$$
\begin{gathered}
firstjob, lastjob \in JOBS \\
firstjob \neq lastjob \\
\forall j \in JOBS : task_j.end = finish \Leftrightarrow lastjob = j \\
\forall j \in JOBS : task_j.start = 1 \Leftrightarrow firstjob = j \\
cleanlf = CLEAN_{lastjob, firstjob}
\end{gathered}
$$

サイクルタイムを最小にすることは、$finish + cleanlf$ を最小にすることになります。

5.6.2 インプリメンテーション

以下の Mosel モデルは、上で定式化されたタスクベースのモデルを実行します。タスク間のセットアップ時間は、手順 setsetuptimes でセットされ、2 つのタスクオブジェクトと対応する処理時間の値を示

しています。

```
model "B-5 Paint production (CP)"
 uses "kalis"

 declarations
  NJ = 5                             ! Number of paint batches (=jobs)
  JOBS=1..NJ

  DUR: array(JOBS) of integer        ! Durations of jobs
  CLEAN: array(JOBS,JOBS) of integer ! Cleaning times between jobs

  task: array(JOBS) of cptask
  res: cpresource

  firstjob,lastjob,cleanlf,finish: cpvar
  L: cpvarlist
  cycleTime: cpvar                   ! Objective variable
  Strategy: array(range) of cpbranching
 end-declarations

 initializations from 'Data/b5paint.dat'
  DUR CLEAN
 end-initializations

! Setting up the resource (formulation of the disjunction of tasks)
 set_resource_attributes(res, KALIS_UNARY_RESOURCE, 1)

! Setting up the tasks
 forall(j in JOBS) getstart(task(j)) >= 1                     ! Start times
 forall(j in JOBS) set_task_attributes(task(j), DUR(j), res)  ! Dur.s + disj.
 forall(j,k in JOBS) setsetuptime(task(j), task(k), CLEAN(j,k), CLEAN(k,j))
                                     ! Cleaning times between batches

! Cleaning time at end of cycle (between last and first jobs)
 setdomain(firstjob, JOBS); setdomain(lastjob, JOBS)
 firstjob <> lastjob
 forall(j in JOBS) equiv(getend(task(j))=getmakespan, lastjob=j)
 forall(j in JOBS) equiv(getstart(task(j))=1, firstjob=j)
 cleanlf = element(CLEAN, lastjob, firstjob)

 forall(j in JOBS) L += getend(task(j))
 finish = maximum(L)

! Objective: minimize the duration of a production cycle
 cycleTime = finish - 1 + cleanlf

! Solve the problem
 if cp_schedule(cycleTime) = 0 then
  writeln("Problem is infeasible")
```

```
  exit(1)
 end-if
 cp_show_stats

! Solution printing
 declarations
  SUCC: array(JOBS) of integer
 end-declarations

 forall(j in JOBS)
  forall(k in JOBS)
   if getsol(getstart(task(k))) = getsol(getend(task(j)))+CLEAN(j,k) then
    SUCC(j):= k
    break
   end-if
 writeln("Minimum cycle time: ", getsol(cycleTime))
 writeln("Sequence of batches:¥nBatch Start Duration Cleaning")
 forall(k in JOBS)
  writeln("  ", k, strfmt(getsol(getstart(task(k))),7), strfmt(DUR((k)),8),
          strfmt(if(SUCC(k)>0, CLEAN(k,SUCC(k)), getsol(cleanlf)),9))

end-model
```

5.6.3 結果

結果は、Section 3.8 で報告されたものと似ています。 ここで、注意すべきは、このモデル定式化は、前のモデルバージョンに比較し、サーチノードと実行時間で、それほど効率的でないということです。Section 3.10 に提示されている'cycle' constraint version に比べると、特に、そうです。しかし、タスクベースの定式化は、前のモデルバージョンでは、問題の定式化が problem-specific であったのに較べ、よりジェネリックな定式化なので、フィーチャーの追加が容易だという利点もあります。

によるグラフ表示は以下の通りにです(Figure5.6)。

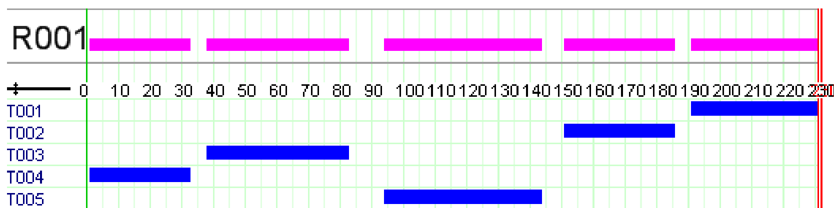


Figure 5.6: Solution graph in IVE

5.7 列挙 (Enumeration)

スケジュールの完成時間を最小にしたい場合、このソルバーによって使われる内蔵のサーチ戦略は、特に、目的に適っています。他の目的では、内蔵の戦略は良い選択でないかもしれません。特に、モ

デルがスケジュール作成オブジェクトの一部でない決定変数を含んだり(内蔵の戦略は、いつも、最初に、スケジュール作成オブジェクトを列挙します)、または、目的を最大にしたい場合などです。Xpress-Kalis では、したがって、スケジュール作成オブジェクトを含むモデルでも、標準の最適化ファンクション cp_minimize、cp_maximize が使えるようになっています。これらの最適化ファンクションが使うディフォルトサーチ戦略は、スケジュール作成を目的とした戦略とは異なっています。さらに、ユーザは、以下の例に示されているように、ユーザ自身の、問題固有の列挙を定義することもできます。

スケジュール作成問題のためのユーザによって定義された列挙な戦略は、変数ベース、タスクベースの、2 つの異なる形を取ります。前者は、第 4 章のテーマです。そして、本章では、小さい例(Section 5.7.1)を示すに止めます。後者は、job-shop scheduling example を使い、もう少し、詳しく説明します。

5.7.1 変数ベースの列挙 (Variable-based enumeration)

Section 5.4 の bin packing problem の統計を調べると、ディフォルトのスケジュール作成サーチ戦略を使うと、最適性を証明するには、数 100 のノードを必要とするのがわかります。問題固有のサーチ戦略を使うと、そんなに必要ではありません。確かに、greedy-type strategy で、十分なスペースがある最初のディスクに大きいファイルを、まず、最初に割り当てると、明らかに、より効率的です。

5.7.1.1 cp_minimize を使う

この問題で行う唯一の意思決定は、タスクの開始時間変数のための値を選んで、ディスクへファイルの配分することです。以下の Mosel コードは、ファイルサイズが大きい方から小さい方に、降順に並べ、それぞれに、可能な限り小さいディスク番号を、彼らのスタート時間割り当てる列挙戦略を定義します。ソーティングサブルーチン qsort が、モデルの最初で、uses ステートメントでロードされるモジュール mmsystem によって定義されることに注意してください。

```
declarations
 Strategy: array(range) of cpbranching
 FSORT: array(FILES) of integer
end-declarations

qsort(SYS_DOWN, SIZE, FSORT)         ! Sort files in decreasing order of size
forall(f in FILES)
 Strategy(f):=assign_var(KALIS_SMALLEST_MIN, KALIS_MIN_TO_MAX,
                         {getstart(file(FSORT(f)))})
cp_set_branching(Strategy)

if not cp_minimize(diskuse) then writeln("Problem infeasible"); end-if
```

variable selection criterion (assign_var の最初のアーギュメント)の選択は、ここでは、必ずしも、重要であるというわけではありません。なぜなら、戦略 $Strategy_f$ は、すべて、ただ一つの変数に適用するだけで、したがって、選択は、一切、行われないからです。以下のように書いても同等です。

```
declarations
 Strategy: cpbranching
 FSORT: array(FILES) of integer
 LS: cpvarlist
end-declarations

qsort(SYS_DOWN, SIZE, FSORT)        ! Sort files in decreasing order of size
forall(f in FILES)
 LS+=getstart(file(FSORT(f)))
Strategy:=assign_var(KALIS_INPUT_ORDER, KALIS_MIN_TO_MAX, LS)
cp_set_branching(Strategy)

if not cp_minimize(diskuse) then writeln("Problem infeasible"); end-if
```

このサーチで、最初に見つかったソリューションは 3 枚のフロッピーディスクを使用するソリューションでした。すなわち、すぐに、最適解を見つられたということです。全体のサーチは 17 ノードで終わり、掛かった時間は、ディフォルトのスケジュール作成戦略、最小化戦略で必要であった時間の、ほんのごく一部ですみました。

5.7.1.2 cp_schedule を使う

スケジュール作成サーチは、前処理(shaving)フェーズと、2 つの主なフェーズ(KALIS_INITIAL_SOLUTION と KALIS_OPTIMAL_SOLUTION)から成ります。ユーザは、対応するフェーズ選択で、cp_set_schedule_searchcp をコールすることで、_列挙戦略を指定できます。'initial solution' フェーズは、すばやく良いソリューションを提供することを目的とし、'optimal solution'フェーズは、最適性を証明することを目的とします。最大ノード数などのサーチ・リミットは、それぞれのフェーズに、別個に適用されますが、全体的な時間制限(parameter MAX_COMPUTATION_TIME)は、最終フェーズのみに適用されます。

scheduling search の、変数ベースの branching scheme の定義は、前に見た cp_set_strategy を cp_set_schedule_strategy で置き換え、cp_set_strategy を cp_minimize や cp_maximize による標準サーチに関して見た方法と、まったく同じ方法で行います。

```
cp_set_schedule_branching(KALIS_INITIAL_SOLUTION, Strategy)
 if cp_schedule(diskuse)=0 then writeln("Problem infeasible"); end-if
```

このサーチ戦略で、最適解は、'initial solution で、たったの 8 つのノードの後に見つけられました。そして、列挙は、そこに止まりました。なぜなら、前処理フェーズで、最適解の値である 3 枚の lower bound が証明されたからです。

NB: スケジュール作成サーチの個々のフェーズからのアウトプット・ログを得るためには、コントロール・パラメタ VERBOSE_LEVEL を 2 に設定してください。すなわち、ソリューションアルゴリズムの始まり

の前に、以下のラインをモデルに追加してください。

```
setparam("VERBOSE_LEVEL", 2)
```

5.7.2 タスクベースの列挙（Task-based enumeration）

タスクベースの列挙戦略は、列挙されるタスクを選ぶ選択戦略、タスク処理時間のための value selection heuristic、および、タスク開始時間の value selection heuristic の定義から成っています。

問題の典型的な定義を考えて見ましょう。ここに与えられたジョブの集合があり、それぞれのジョブは、固定された順序で、所与の機械の集合で処理される必要があります。機械は、一度に、1 つのジョブを処理します。生産タスクの処理時間(すなわち、機械でのジョブの処理)、および、ジョブ別の機会処理の順番（6X6 ケース）は Table5.3 に示されています。目的関数は、スケジュールの makespan(最も遅い完成時間)を最小にすることです。

Table: 6X6 job-shop instance from FT63

Job	Machines	Durations
1	3 1 2 4 6 5	1 3 6 7 3 6
2	2 3 5 6 1 4	8 5 10 10 10 4
3	3 4 6 1 2 5	5 4 8 9 1 7
4	2 1 3 4 5 6	5 5 5 3 8 9
5	3 2 5 6 1 4	9 3 5 4 3 1
6	2 4 6 1 5 3	3 3 9 10 4 1

5.7.2.1 モデルの定式化

JOBS でジョブの集合を、そして、*MACH*（*MACH* = {1, . . . , *NM*}で機械の集合を意味するとします。すべてのジョブ j は、一連のタスク $task_{jm}$ に従って生産されて、ここで、$task_{jm}$ は、$task_{j,m+1}$ がスタートする前に完了することが必要です。タスク $task_{jm}$ は、機械 RES_{jm} によって処理され、固定した処理時間 DUR_{jm} を持っています。

下のモデルは、この job-shop scheduling problem の定式化です。

$$\text{resources } res_m \left(m \in MACH = \{1,\ldots,NM\}\right)$$
$$\text{tasks } task_{jm} \left(j \in JOBS, m \in MACH\right)$$
$$\text{minimize } finish$$
$$finish = \text{maximum}_{j \in JOBS} \left(task_{j,NM}.end\right)$$
$$\forall m \in MACH : res_m.capacity = 1$$
$$\forall j \in JOBS, m \in MACH : task_{jm}.start, task_{jm}.end \in \{0,\ldots,MAXTIME\}$$

$$\forall j \in JOBS, m \in \{1,...,NM-1\} : task_{jm}.successors = \{task_{j,m+1}\}$$
$$\forall j \in JOBS, m \in MACH : task_{jm}.duration = DUR_{jm}$$
$$\forall j \in JOBS, m \in MACH : task_{jm}.requirement_{RES_{jm}} = 1$$

5.7.2.2 インプリメンテーション

以下の Mosel モデルは、job-shop scheduling 問題を実行し、問題を解くためのタスクベースのブランチ戦略を定義します。スタート変数の中から、残っているドメインが最も小さいタスクを選択し、この変数に、最も小さい可能な値から列挙を行います。タスクベースのブランチ戦略は、Xpress-Kalis の中で、ファンクション task_serialize により定義され、このファンクションは、アーギュメントとして、user task selection、処理時間変数とスタート変数のための値の選択戦略、および、適用するタスクの集合を取ります。このようなタスクベースのブランチ戦略は、まったく自由に、どんな変数ベースのブランチ戦略にも結合できます。

```
model "Job shop (CP)"
 uses "kalis", "mmsystem"

 parameters
  DATAFILE = "jobshop.dat"
  NJ = 6                                ! Number of jobs
  NM = 6                                ! Number of resources
 end-parameters

 forward function select_task(tlist: cptasklist): integer

 declarations
  JOBS = 1..NJ                          ! Set of jobs
  MACH = 1..NM                          ! Set of resources
  RES: array(JOBS,MACH) of integer      ! Resource use of tasks
  DUR: array(JOBS,MACH) of integer      ! Durations of tasks

  res: array(MACH) of cpresource        ! Resources
  task: array(JOBS,MACH) of cptask      ! Tasks
 end-declarations

 initializations from "Data/"+DATAFILE
  RES DUR
 end-initializations

 HORIZON:= sum(j in JOBS, m in MACH) DUR(j,m)
 forall(j in JOBS) getend(task(j,NM)) <= HORIZON

! Setting up the resources (capacity 1)
 forall(m in MACH)
  set_resource_attributes(res(m), KALIS_UNARY_RESOURCE, 1)
```

```
! Setting up the tasks (durations, resource used)
 forall(j in JOBS, m in MACH)
  set_task_attributes(task(j,m), DUR(j,m), res(RES(j,m)))

! Precedence constraints between the tasks of every job
 forall (j in JOBS, m in 1..NM-1)
!  getstart(task(j,m)) + DUR(j,m) <= getstart(task(j,m+1))
  setsuccessors(task(j,m), {task(j,m+1)})

! Branching strategy
 Strategy:=task_serialize("select_task", KALIS_MIN_TO_MAX,
           KALIS_MIN_TO_MAX,
           union(j in JOBS, m in MACH | exists(task(j,m))) {task(j,m)})
 cp_set_branching(Strategy)

! Solve the problem
 starttime:= gettime

 if not cp_minimize(getmakespan) then
  writeln("Problem is infeasible")
  exit(1)
 end-if

! Solution printing
 cp_show_stats
 write(gettime-starttime, "sec ")
 writeln("Total completion time: ", getsol(getmakespan))
 forall(j in JOBS) do
  write("Job ", strfmt(j,-2))
  forall(m in MACH | exists(task(j,m)))
   write(strfmt(RES(j,m),3), ":", strfmt(getsol(getstart(task(j,m))),3),
         "-", strfmt(getsol(getend(task(j,m))),2))
  writeln
 end-do

!****************************************************************
! Task selection for branching
 function select_task(tlist: cptasklist): integer
  declarations
   Tset: set of integer
  end-declarations

 ! Get the number of elements of "tlist"
  listsize:= getsize(tlist)

 ! Set of uninstantiated tasks
  forall(i in 1..listsize)
   if not is_fixed(getstart(gettask(tlist,i))) then
    Tset+= {i}
```

```
  end-if

 returned:= 0

 ! Get a task with smallest start time domain
 smin:= min(j in Tset) getsize(getstart(gettask(tlist,j)))
 forall(j in Tset)
  if getsize(getstart(gettask(tlist,j))) = smin then
   returned:=j; break
  end-if

end-function

end-model
```

5.7.2.3 結果

この問題の最適解には、55 の makespan があります。ディフォルトスケジュール作成戦略との比較では、このブランチ戦略は、300 ノードを超えていたもの 105 ノードまで減少させました。しかし、より大きいインスタンスでは、このブランチ戦略よりも、ディフォルトのスケジュール作成戦略の方が優れていつようです。ディフォルトの最小化戦略は、何分かのランタイムでは、この問題のソリューションを見つけられませんでした。

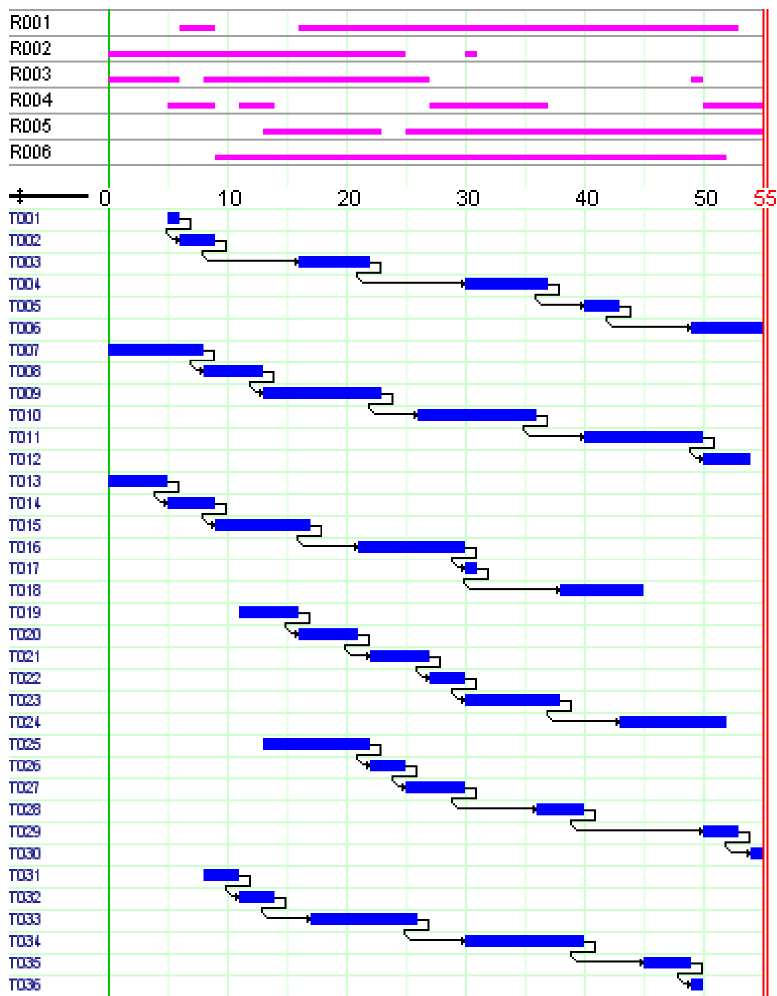


Figure 5.7: Solution graph in IVE

5.7.2.4 代替的なサーチ戦略

上で見たものに似ていますが、スケジュール作成サーチのためのユーザタスク選択戦略を定義することもできます。上のモデルに必要な唯一の変更は、cp_set_branching を cp_set_schedule_strategy で置き換え、cp_minimize を cp_schedule で置き換えることだけです。ユーザのタスク選択ファンクション select_task の定義は変わりません。

```
! Branching strategy
 Strategy:=task_serialize("select_task", KALIS_MIN_TO_MAX,
              KALIS_MIN_TO_MAX,
              union(j in JOBS, m in MACH | exists(task(j,m))) {task(j,m)})
 cp_set_schedule_strategy(KALIS_INITIAL_SOLUTION, Strategy)

! Solve the problem
 if cp_schedule(getmakespan)=0 then
  writeln("Problem is infeasible")
```

```
  exit(1)
 end-if
```

この戦略は、ユーザタスク選択による標準サーチよりも、列挙を完了するためのノードが、一層、少ないノードですみます。

ユーザ定義のタスク選択ファンクション select_task ではなく、下記の、事前に定義されたタスク選択評価基準の 1 つを使うこともできます。

KALIS_SMALLEST_EST / KALIS_LARGEST_EST:

choose task with the smallest/largest lower bound on its start time ('earliest start time'),

KALIS_SMALLEST_LST / KALIS_LARGEST_LST:

choose task with the smallest/largest upper bound on its start time ('latest start time'),

KALIS_SMALLEST_ECT / KALIS_LARGEST_ECT:

choose task with the smallest/largest lower bound on its completion time ('earliest completion time'),

KALIS_SMALLEST_LCT / KALIS_LARGEST_LCT:

choose task with the smallest/largest upper bound on its completion time ('latest completion time').

現在の例では、最も良い選択は KALIS_SMALLEST_LCT であることが判明しています(cp_schedule、および、cp_minimize の両方を使い、約 60 ノードでサーチを終了)。

```
 Strategy:=task_serialize(KALIS_SMALLEST_LCT, KALIS_MIN_TO_MAX,
              KALIS_MIN_TO_MAX,
              union(j in JOBS, m in MACH | exists(task(j,m))) {task(j,m)})
```

5.7.3 propagation algorithm の選択

スケジュール作成問題のためのサーチアルゴリズムのパーフォーマンスは、列挙戦略の定義のみに依存するのではありません。resource constraint のプロパゲーションアルゴリズムの選択も、大きく影響します。

プロパゲーション・タイプは、下記のように、手順 set_resources_attributes の最後のオプショナルナアーギュメントでセットされます。

```
set_resource_attributes(res, KALIS_DISCRETE_RESOURCE, CAP, ALG)
```

ここで、ALG は、propagation algorithm choice です。

unary resource のためには、KALIS_TASK_INTERVALS、KALIS_DISJUNCTIONS から選択します。前者は、大きいコンピュータの計算オーバーヘッドというコストを支払うことで、より強い刈り込みを行うので、小さい問題から、中程度の問題までは、KALIS_DISJUNCTIONS の選択のほうがよいでしょう。前のセクションの jobshop 問題の例を取れば、KALIS_DISJUNCTIONS を使用した場合、ディフォルトのスケジュール作成サーチは、KALIS_TASK_INTERVALS よりも約 4 倍速いですが、サーチされるノードの数は、よりわずかに大きい結果となりました。

cumulative resources のためのプロパゲーションアルゴリズムのオプションは、KALIS_TASK_INTERVALS と KALIS_TIMETABLING です。フィルタリングアルゴリズム KALIS_TASK_INTERVALS は、より強力ですが、比較的遅いので、このことは、難しい、中型の問題のためのよい選択ですが、計算オーバーヘッドが問題になる大型の問題には、KALIS_TIMETABLING が向いているでしょう。例を挙げれば、Section 5.4、5.7.1 で議論した binpacking 問題では、KALIS_TASK_INTERVALS の方が、(ディフォルトスケジュール作成サーチを使用した) KALIS_TIMETABLING よりも、3 倍速く、問題を解きました。

Appendix A

トラブル・シューティング

- No license found: Kalis for Mosel、licensing system、および、Xpress-Kalis モジュールがインストールされていなければなりません。Dash から受け取ったライセンスファイルを、Xpress-MP installation directory にコピーし、environment variable XPRESSDIR が、このディレクトリをポイントするように設定する必要があります。
- The Xpress-Kalis module is not found: Mosel のディレクトリ dso に、ファイル kalis.dso がインストールされていないときは、environment variable MOSEL_DSO を、このファイルのロケーションで、定義しなければなりません。

Appendix B

Glossary of CP terms

CP 用語のグロッサリーは、英文をお読み下さい。

Bibliography

[FM63] H. Fisher and J. F. Muth. Probalistic Learning Combinations of Local Job-Shop Scheduling Rules. In J. F. Muth and G. L. Thompson, editors, *Industrial Scheduling*, pages 225—251, Englewood Cliffs, N. J., 1963. Prentice Hall.

[Hei99] S. Heipcke. Comparing Constraint Programming and Mathematical Programming Approaches to Discrete Optimisation. The Change Problem. *Journal of the Operational Research Society,* 50(6):581—595, 1999.