Problem Solving with Finite Set Constraints in Oz. A Tutorial.

Tobias Müller

Version 1.3.2
June 15, 2006

Abstract

This document is an introduction to finite set constraint programming in Oz. Consequently, it focuses on finite set constraints but uses them in conjunction with finite domain constraints. Further, basic concepts of constraint programming will not be explained in this document. Hence, it is strongly recommended to read this tutorial after reading *"Finite Domain Constraint Programming in Oz. A Tutorial."*.

Credits

Mozart logo by Christian Lindig

License Agreement

Contents

1 Introduction

Set Values Oz 3 provides finite sets of non-negative integers as first-class values and every set value is a subset of the universal set $\mathcal{U} = \{0, \ldots, sup\}$. The value of sup is determined by the actual implementation and in Mozart Oz 3 it is $134217726 = 2^{27} - 2^{1}$.

Set Constraints A basic set constraint approximates a set value S in three different ways:

- Constraining the lower bound by set s: $s \subseteq S$. The lower bound contains all elements that are at least contained in the set value.
- Constraining the upper bound by set s: $S \subseteq s$. The upper bound contains all elements that are at most contained in the set value.
- Constraining the cardinality of a set by a finite domain interval $\{n, \ldots, m\}$: $n \leq \#S \leq m.$. The cardinality constraint determines the minimal and maximal number of elements allowed to be contained in the set.

A set constraint denotes a set value if either the lower is equal to the upper bound, the cardinality of the lower bound is equal to the upper bound of the cardinality constraints, or the cardinality of the upper bound is equal to the lower bound of the cardinality constraint.

Non-basic set constraints, as intersection $\cap$, union $\cup$, disjointness $\|$, and the like, are provided as propagators. For details on the provided set propagators see Chapter *Finite Set Constraints:* `FS`, *(System Modules)*.

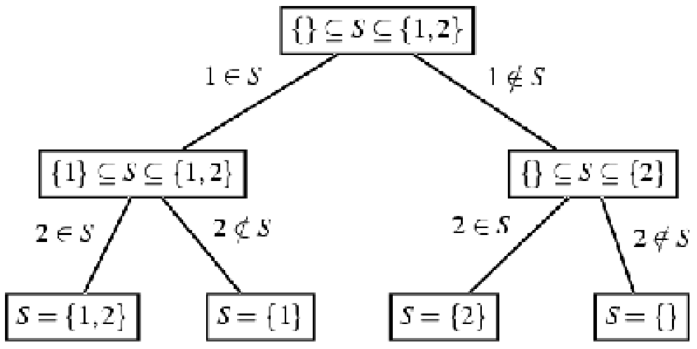
Set Constraint Propagation To explain constraint propagation, assume the basic set constraints:$\emptyset \subseteq X, Y \subseteq \{1, \ldots, 5\}$ and additionally the following non-basic constraints: $X \cup Y = \{1, \ldots, 5\}$ and $X \| Y$. Adding the constraints $1 \in X$ and $2 \notin Y$ yields the intermediate store $\{1\} \subseteq X \subseteq \{1, \ldots, 5\}$ and $\emptyset \subseteq Y \subseteq \{1, 3, 4, 5\}$. The present non-basic constraints can add even more basic constraints: the disjointness constraint removes 1 from the upper bound of Y since 1 was added to the lower bound of X.

[1]The reason for this value is as follows: efficient integers (so-called *small integers* in Mozart Oz 3) occupy 28 bits. Hence the biggest positive integer is $2^{27} - 1$. To be able to represent the cardinality of a set by a small integer, the biggest element of a set is determined to $2^{27} - 2$.

The union constraint adds 2 to the lower bound of X since 2 was removed form the upper bound of Y. After that, propagation has reached a fixed-point and leads to $\{1,2\} \subseteq X \subseteq \{1,\ldots,5\} \wedge \emptyset \subseteq Y \subseteq \{3,4,5\}$. Bringing the cardinality constraint $3 \leq \#Y \leq 5$ into play determines Y to $\{3,4,5\}$ since the upper bound has exactly 3 elements which is the minimal number required by the cardinality constraint. The disjointness constraint then removes 3, 4, 5 from X's upper bound and that way determines X to $\{1,2\}$.

Connecting Finite Sets and Finite Domains Set constraints on their own are of limited use, connecting them with finite domain constraints provides much more expressivity. The straightforward way is to connect a finite set variable via the cardinality constraint to a finite domain variable. Another technique is to provide reified versions for various set constraints as containment and the like. But there are further possiblies if the fact that the elements of a set are *integers* is exploited. For example, a finite domain can be constrained to be the minimal resp. maximal element of a set (see Chapter *Finite Set Constraints:* `FS`, *(System Modules)* for details on `FS.int.min` resp. `FS.int.max`). Another possibility is to match the elements of a set of a certain cardinality c with a tuple of c finite domains (see Chapter *Finite Set Constraints:* `FS`, *(System Modules)* for details on `FS.int.match`) that is used in Chapter 2.

Distribution Due to the fact that constraint propagation is incomplete, expectedly in case of set constraints as well, solving a problem involving set constraints requires distribution. A typical choice-point distributing a set variable is $n \in S \vee n \notin S$. The following figure illustrates that.



2

The Steiner Problem

Problem Specification The ternary Steiner problem of order n asks for $n(n-1)/6$ sets $s_i \subset \{1,\ldots,n\}$ with cardinality 3 such that every two of them share at most one element. The mathematical properties of the problem require that n mod 6 has to be either 1 or 3 [1].

Model We create a list `Ss` of $n(n-1)/6$ set variables and constrain every set to have a cardinality of 3 and to have an upper bound of $\{1,\ldots,n\}$. Further we require that the cardinality of the intersection of every two distinct sets in `Ss` must not exceed 1.

Distribution Strategy Distribution simply takes the sets as they occur in `Ss` and adds resp. removes elements from them starting from the smallest element.

Solver The solver is created by a function `Steiner` that takes the order of the Steiner problem as argument and checks if it is a valid order. In case it is valid it returns the actual solver with the list of solution sets as formal argument.

First, the list `Ss` is created and its elements' upper bounds and cardinalities are appropriately constrained. The nested loops built with `ForAllTail` and `ForAll` impose the constraint that every two sets share at most one element by stating that the cardinality of the intersection of two sets is in $\{0,1\}$. Distribution is straightforward and uses the provided library abstraction `FS.distribute` for naive distribution..

```
declare
fun {Steiner N}
   case
      N mod 6 == 1 orelse N mod 6 == 3
   then
      proc {$ Ss}
         {FS.var.list.upperBound (N*(N-1)) div 6 [1#N] Ss}
         {ForAll Ss proc {$ S} {FS.card S 3} end}

         {ForAllTail Ss
          proc {$ S1|Sr}
             {ForAll Sr
              proc {$ S2} S3 in
```

```
                              S3 = {FS.intersect S1 S2}
                              {FS.cardRange 0 1 S3}
                           end}
                       end}

                     {FS.distribute naive Ss}
                 end
              else proc {$ _} fail end
              end
           end
```

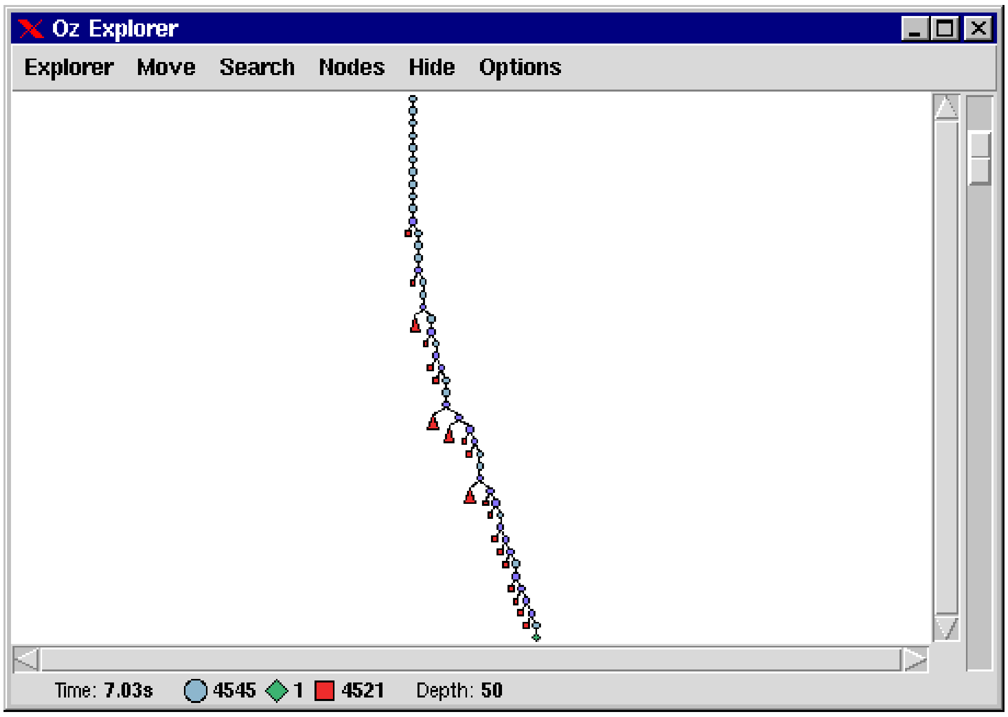
Solving the Steiner problem of order 9 by invoking the Oz Explorer

```
    {ExploreOne {Steiner 9}}
```

yields as solution

```
    [{1#3}#3 {1 4#5}#3 {1 6#7}#3 {1 8#9}#3 {2 4 6}#3 {2 5 8}#3
      {2 7 9}#3 {3#4 9}#3 {3 5 7}#3 {3 6 8}#3 {4 7#8}#3 {5#6 9}#3].
```

The search tree has depth 50, 4545 choice nodes, and 4521 failure nodes.



Time: 7.03s ● 4545 ◆ 1 🟥 4521 Depth: 50

Improving the Model A promising way to improve the efficiency of a constraint model (where the corresponding problem does not have a unique solution) is to break symmetries and thus to improve constraint propagation. Breaking symmetries can be achieved by imposing an order, in our case, an order on the set variables in `Ss`. We can simply interpret every set as a number with three digits to the base $(n+1)$. A set with three elements $\{x_1,x_2,x_3\}$ can be mapped to an integer by $(n+1)^2x_1+(n+1)x_2+x_3$.

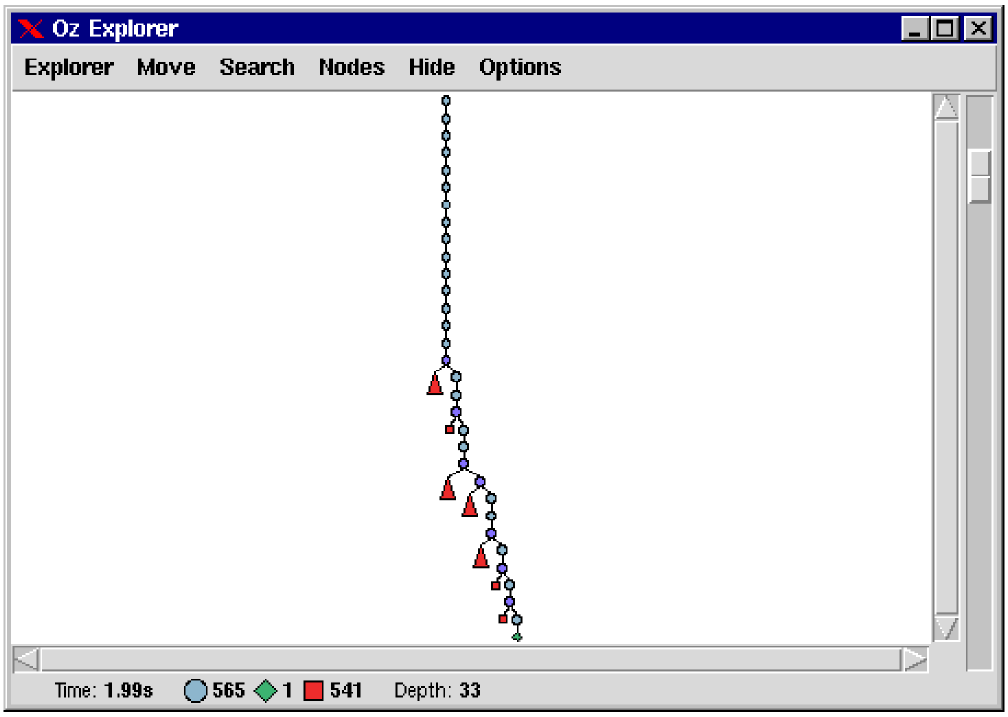
Extending the Solver The finite set library provides `FS.int.match` to match the elements of a set *s* with a fixed number of elements to a vector of size #*s* of finite domain variables. This library constraint in conjunction with `Map` is used to convert the list of sets `Ss` to a list of finite domain lists with 3 finite domains per list. Finally the order between adjacent sets is imposed by

```
N1N1*X1 + N1*X2 + X3 <: N1N1*Y1+ N1*Y2 + Y3
```

employing a `ForAllTail` loop.

```
local
   N1 = N+1  N1N1 = N1*N1
in
   {ForAllTail {Map Ss fun {$ S}
                              {FD.list 3 1#N} = {FS.int.match S}
                           end}
    proc {$ T}
       case T of [X1 X2 X3]|[Y1 Y2 Y3]|_ then
          N1N1*X1 + N1*X2 + X3 <: N1N1*Y1 + N1*Y2 + Y3
       else skip end
    end}
end
```

This code is to be inserted right before the distribution. Solving the Steiner problem of order 9 results in the following search tree.



We see that the number of choice nodes decreases from 4545 to 565 and the number of failure nodes decreases from 4521 to 54. This reduction of the search space gives us a speed-up of about 7 and reduces the memory consumption by about 5.5.

Generating Hamming Codes

Problem Specification Generate a Hamming code that fits in b-bit words to code n symbols where the Hamming distance between every two symbol codes is at least d. The Hamming distance between to words is the number of bit positions where they differ.

Model A b-bit word is modeled by a set $s \subseteq \{1,\ldots,b\}$ where $e \in s$ means that the bit at position e is set (resp. unset otherwise). The *Hamming distance* $h(a,b)$ between two words a and b represented as sets s_a and s_b can be computed by subtracting from the word size b the number of elements that is contained $(\#(s_a \cap s_b))$ resp. is not contained $(\#(\{1,\ldots,b\} \backslash (s_a \cup s_b)))$ in both sets. Thus, the Hamming distance results in

$$h(a,b) = b - \#(s_a \cap s_b) - \#(\{1,\ldots,b\} \backslash (s_a \cup s_b)).$$

Solver The function `Hamming` returns a solver to generate a Hamming code for `NumSymbols` symbols in words with `Bits` bits and a Hamming distance of `Distance`. The procedure `MinDist` implements the constraint that the Hamming distance does not exceed the value of `Distance`. The list `Xs` holds the sets representing the single codes. The nested loop (`ForAllTail` and `ForAll`) imposes `MinDist` on all pairwise distinct elements of `Xs`. The distribution employs straightforwardly a naive strategy.

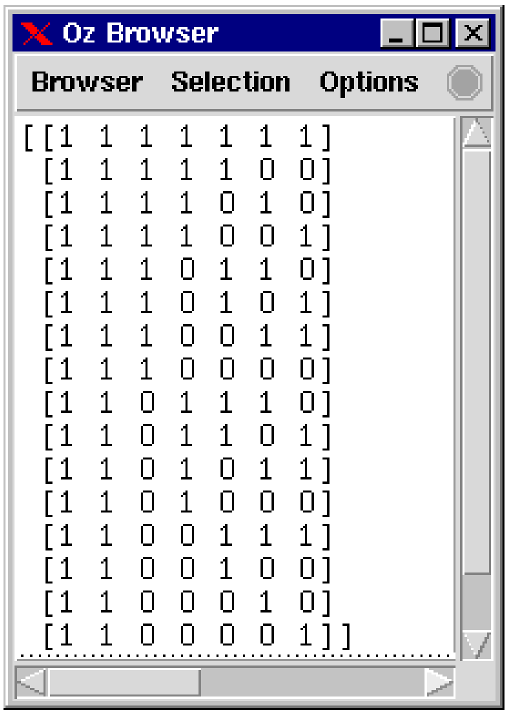
```
declare
fun {Hamming Bits Distance NumSymbols}
   proc {MinDist X Y}
      Common1s = {FS.intersect X Y}
      Common0s = {FS.complIn
                  {FS.union X Y}
                  {FS.value.make [1#Bits]}}
   in
      Bits-{FS.card Common1s}-{FS.card Common0s}>=:Distance
   end
in
   proc {$ Xs}
      Xs = {FS.var.list.upperBound NumSymbols [1#Bits]}

      {ForAllTail Xs proc {$ X|Y}
```

```
                                          {ForAll Y proc {$ Z}
                                                       {MinDist X Z}
                                                   end}
                                    end}

                        {FS.distribute naive Xs}
                     end
                  end
```

The following code generates a Hamming code for 16 symbols using 7 bit words and ensures a Hamming distance of 2.

```
{Browse
 {Map {SearchOne {Hamming 7 2 16}}.1
  fun {$ X}
     {ForThread 7 1 ~1 fun {$ Is I}
                          {FS.reified.isIn I X}|Is
                       end nil}
  end}}
```

Further, the code is nicely formatted displayed in the Oz Browser.



```
[[1 1 1 1 1 1 1]
 [1 1 1 1 1 0 0]
 [1 1 1 1 0 1 0]
 [1 1 1 1 0 0 1]
 [1 1 1 0 1 1 0]
 [1 1 1 0 1 0 1]
 [1 1 1 0 0 1 1]
 [1 1 1 0 0 0 0]
 [1 1 0 1 1 1 0]
 [1 1 0 1 1 0 1]
 [1 1 0 1 0 1 1]
 [1 1 0 1 0 0 0]
 [1 1 0 0 1 1 1]
 [1 1 0 0 1 0 0]
 [1 1 0 0 0 1 0]
 [1 1 0 0 0 0 1]]
```

Packing Files onto Disks

Problem Specification Suppose, you want to copy a set of files from your hard-disk onto as few as possible diskettes of a given size, e.g. onto common 1.44 MB diskettes. In case your files do not fit on a single diskette, it might become quite tricky to figure out the minimal number of needed diskettes and how to partition the files.

Model A diskette is modeled by a set s_i. All sets s_i form a partition of the set of all files $s_{all files}$, i.e., all s_i are pairwise disjoint and their union is $s_{all files}$. The sizes of all files contained in a set is summed up and compared with the fixed capacity of the diskette.

Distribution Strategy The distribution is two-dimensional.

- Distribute the number of diskettes starting from the minimal number possible. The minimal number is the ceiling of dividing the sum of all file sizes by the diskette size.
- Distribute the files over the sets representing the individual diskettes.

The distribution over the files could be refined by taking the size of the actual file into account. This is subject to experimentation by the reader.

Solver The function `SpreadFiles` returns a solver configured according to the actual values of the formal arguments `Files` and `DiskCap`. The returned solver's root variable `Disks` contains the set of diskettes of size `DiskCap` needed to store all files in `Files` and specifies what files have to be stored on which diskette.

The argument `Files` holds a list of individual files, where each file is represented by a record with label `file` and the features `name` and `size`. The argument `DiskCap` is an integer. The variable `FileSizes` holds a list of all files sizes and `Size` stores the sum all elements in `FileSizes`. The lower bound of the number of diskettes is held in `LB`. The finite domain `NbDisks` is used to distribute over the number of diskettes. Each file in `Files` is represented by an integer in ascending order starting from 1. These integers are stored in `AllFiles`. Finally, the sets representing the individual diskettes are held in `Ds`.

First, the number of diskettes is distributed starting from `LB`. Then, `Ds` is initialized to sets containing maximal all files. Next, the constraint that all elements of `Ds` are a partition of the set of all files is imposed. Finally, the maximum capacity of all diskettes is limited to `DiskCap` by imposing for all elements of `Ds` the constraint that the sum of the size of all their elements is less or equal to `DiskCap`. The implementation uses `FS.reified.areIn` to associate the containment of individual elements of sets to 0/1 variables. These 0/1 variable are passed to `FD.sumC` to ensure that a diskettes capacity is not exceeded. Distribution over `Ds` tries to locate file onto diskettes.

Particularities The solver represents internally individual file as integers since finite set constraints in Oz can only deal with non-negative integers. To make the produced solution readable to humans, a diskette is represented as record where the features are the files to be stored on that diskette. Such a record is constructed by imposing a feature constraint onto each element of `Disks`. Then the actual features representing the filenames are added successively by mapping the elements of the set representing the diskettes to their names. Every feature refers to the size of the file it represents. Finally, the feature constraint becomes a record by constraining its arity's width to the number of features.

```
declare
fun {SpreadFiles Files DiskCap}
   proc {$ Disks}
      FileSizes = {Map Files fun {$ F} F.size end}
      Size = {FoldL FileSizes Number.'+' 0}
      LB = Size div DiskCap +
           if Size mod DiskCap==0 then 0 else 1 end
      NbDisks    = {FD.int LB#FD.sup}
      AllFiles  = {List.number 1 {Length Files} 1}
      Ds
   in
      {FD.distribute naive [NbDisks]}

      {FS.var.list.upperBound NbDisks AllFiles Ds}

      {FS.partition Ds {FS.value.make AllFiles}}

      {ForAll Ds proc {$ D} BL in
                    {FS.reified.areIn AllFiles D BL}
                    {FD.sumC FileSizes BL '=<:' DiskCap}
                 end}

      {FS.distribute naive Ds}

      Disks = {Map Ds
               fun {$ D}
                  Disk = {RecordC.tell diskette}
               in
                  {ForAll {FS.monitorIn D}
                   proc {$ E}
                      F = {Nth Files E}
```

```
                              in
                                 Disk^(F.name) = F.size
                              end}
                             {RecordC.width Disk} = {FS.card D}
                             Disk
                          end}
              end
           end
```

Invoking the solver by

```
declare Disks =
{SearchOne {SpreadFiles [file(name:a size:360)
                         file(name:b size:850)
                         file(name:c size:630)
                         file(name:d size:70)
                         file(name:e size:700)
                         file(name:f size:210)]
            1440}}
```

produces the following result:

```
[[diskette(a:360 b:850 f:210) diskette(c:630 d:70 e:700)]]
```

The input data for this solver can be easily obtained from the respective operating system by using the module `OS` (see Chapter *Operating System Support:* `OS`, *(System Modules)* for details].

5

A Crew Allocation Problem

Problem Specification A small air-line has to assign their 20 flight attendants to 10 flights. Each flight has to be accompanied by a certain number of cabin crew (see Figure 5.1) that has to meet a couple of constraints. First, to serve the needs of international clients the cabin crew has to be able to speak German, Spanish, and French (see Figure 5.2). Further, a minimal number of stewardesses resp. stewards have to attend a flight (see Figure 5.3). Finally, every cabin crew member has two flights off after an attended flight.

Figure 5.1 Cabin crew per flight.

flight #	# of cabin staff	flight #	# of cabin staff
1	4	6	4
2	4	7	4
3	5	8	5
4	5	9	5
5	6	10	6

Figure 5.2 Cabin crew speaking foreign language per flight.

flight #	French	Spanish	German
1	1	1	1
2	1	1	1
3	1	1	1
4	2	2	1
5	2	2	1
6	1	1	1
7	1	1	1
8	1	1	1
9	1	1	1
10	1	1	1

Model The cabin crew for every flight is represented as a set. The constraints on cabin crews of individual flights are modeled in terms of constraints on the cardinality of the intersection of the cabin crew set of that flight with the sets associated with

Figure 5.3 Male resp. female cabin crew per flight.

flight #	male	female	flight #	male	female
1	1	1	6	1	1
2	1	1	7	1	1
3	1	1	8	1	1
4	2	2	9	1	1
5	3	2	10	1	1

particular restrictions. Therefore the following subsets of the cabin crew are introduced: male, female, Spanish-speaking, French-speaking, and German-speaking cabin crew. The constraint that a crew member has two flights off after an attended flight is expressed by the disjointness of the appropriate sets representing a crew per flight.

Solver The function `AssignCrew` returns a solver configured according to its arguments `FlightData` and `Crew`. As previously mentioned, the constraints on the cabin crew of a flight are expressed in terms of sets of crew members meeting these constraints. For that reason the following variables are defined:

- `Stewards` (male cabin crew members),
- `Stewardesses` (female cabin crew members),
- `FrenchSpeaking`, `GermanSpeaking`, and `SpanishSpeaking` (French-, German-, resp. Spanish-speaking cabin crew members).

Procedure `TeamConstraint` imposes the abovementioned constraints on the individual flight cabin crew sets intersecting them with appropriate sets (`FS.intersection`), and constrains the intersection's cardinality according to Figure 5.1, Figure 5.2, and Figure 5.3 (using `FS.card` and `>=:`).

The procedure `SequenceDisjoint` is responsible to ensure that every crew member may enjoy a two-flight break between two flights. It is a recursive procedure imposing `FS.disjoint` upon every 3 subsequent sets.

The actual solver declares the local variable `Flights` that contains the list of sets representing the individual crew assignments. Then, the constraints of the procedure `TeamConstraint` are imposed on `Flights` by the `Map` loop, by mapping the data provided by `FlightData` to `Flights`. The distribution is straightforward and has no particularities.

Dealing with sets of literals Often real-life applications deal with sets of names, descriptions and the like rather than integers, which can be represented by Oz literals. The functions `SetOfLiterals`, `Lits2Ints`, and `Ints2Lits` allow to model sets of literals. The function `SetOfLiterals` returns an abstract data structure that enables `Lits2Ints` and `Ints2Lits` to map literals to integers and vice versa. The last line of the solver procedure converts the internal solution to a representation corresponding to the format of `AssignCrew`'s argument `Crew` (see below).

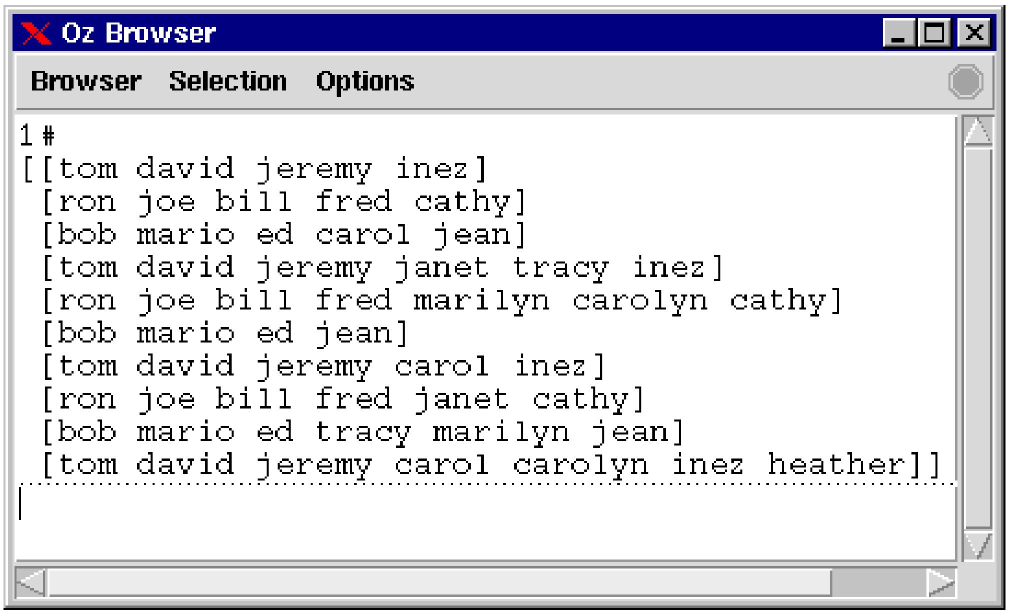
```
declare

local
   Lit2Int = {NewName}
   Int2Lit = {NewName}
in
   fun {SetOfLiterals Lits}
      sol(Lit2Int:
             {NewChunk
              {List.toRecord l2i
               {List.mapInd Lits fun {$ I L}
                                     L#I
                                   end}}}
          Int2Lit:
             {NewChunk
              {List.toRecord i2l
               {List.mapInd Lits fun {$ I L}
                                     I#L
                                   end}}})
   end

   fun {Lits2Ints SetOfLiterals Literals}
      {Map Literals fun {$ Lit}
                       SetOfLiterals.Lit2Int.Lit
                    end}
   end

   fun {Ints2Lits SetOfLiterals Ints}
      {Map Ints fun {$ Int}
                   SetOfLiterals.Int2Lit.Int
                end}
   end
end

fun {CrewProb FlightData Crew}
   CabinStaff      = {Append Crew.stewards Crew.stewardesses}
   CrewSet         = {SetOfLiterals CabinStaff}
   Stewards        = {FS.value.make
                       {Lits2Ints CrewSet Crew.stewards}}
   Stewardesses    = {FS.value.make
                       {Lits2Ints CrewSet Crew.stewardesses}}
   FrenchSpeaking  = {FS.value.make
                       {Lits2Ints CrewSet Crew.frenchspeaking}}
   GermanSpeaking  = {FS.value.make
                       {Lits2Ints CrewSet Crew.germanspeaking}}
   SpanishSpeaking = {FS.value.make
                       {Lits2Ints CrewSet Crew.spanishspeaking}}

   proc {TeamConstraint Team Flight}
```

```
            flight(no:_ crew:N stewards:NStew stewardesses:NHost
                   frenchspeaking:NFrench germanspeaking:NGerman
                   spanishspeaking:NSpanish) = Flight
         in
            {FS.card Team  N}
            {FS.card
             {FS.intersect Team Stewards}}        >=: NStew
            {FS.card
             {FS.intersect Team Stewardesses}}    >=: NHost
            {FS.card
             {FS.intersect Team FrenchSpeaking}}  >=: NFrench
            {FS.card
             {FS.intersect Team GermanSpeaking}}  >=: NGerman
            {FS.card
             {FS.intersect Team SpanishSpeaking}} >=: NSpanish
         end

         proc {SequencedDisjoint L}
            case L of A|B|C|T then
               {FS.disjoint A B}
               {FS.disjoint A C}
               {SequencedDisjoint B|C|T}
            elseof A|B|nil then
               {FS.disjoint A B}
            end
         end
      in
         proc {$ Sol}
            Flights = {FS.var.list.upperBound
                          {Length FlightData}
                          {Lits2Ints CrewSet CabinStaff}}
         in
            {Map FlightData proc {$ D F}
                                   {TeamConstraint F D}
                                end Flights}

            {SequencedDisjoint Flights}

            {FS.distribute naive Flights}

            Sol = {Map Flights
                   fun {$ F}
                      {Ints2Lits CrewSet {FS.monitorIn F}}
                   end}
         end
      end
```

The following sample data can be used to test the solver:

```
declare
Flights =
[flight(no: 1 crew:4 stewards:1 stewardesses:1
        frenchspeaking:1 spanishspeaking:1
        germanspeaking:1)
 flight(no: 2 crew:5 stewards:1 stewardesses:1
        frenchspeaking:1 spanishspeaking:1
        germanspeaking:1)
 flight(no: 3 crew:5 stewards:1 stewardesses:1
        frenchspeaking:1 spanishspeaking:1
        germanspeaking:1)
 flight(no: 4 crew:6 stewards:2 stewardesses:2
        frenchspeaking:1 spanishspeaking:1
        germanspeaking:1)
 flight(no: 5 crew:7 stewards:3 stewardesses:3
        frenchspeaking:1 spanishspeaking:1
        germanspeaking:1)
 flight(no: 6 crew:4 stewards:1 stewardesses:1
        frenchspeaking:1 spanishspeaking:1
        germanspeaking:1)
 flight(no: 7 crew:5 stewards:1 stewardesses:1
        frenchspeaking:1 spanishspeaking:1
        germanspeaking:1)
 flight(no: 8 crew:6 stewards:1 stewardesses:1
        frenchspeaking:1 spanishspeaking:1
        germanspeaking:1)
 flight(no: 9 crew:6 stewards:2 stewardesses:2
        frenchspeaking:1 spanishspeaking:1
        germanspeaking:1)
 flight(no:10 crew:7 stewards:3 stewardesses:3
        frenchspeaking:1 spanishspeaking:1
        germanspeaking:1)]

Crew =
crew(stewards:
        [tom david jeremy ron joe bill fred bob mario ed]
     stewardesses:
        [carol janet tracy marilyn carolyn cathy inez
         jean heather juliet]
     frenchspeaking:
        [inez bill jean juliet]
     germanspeaking:
        [tom jeremy mario cathy juliet]
     spanishspeaking:
        [bill fred joe mario marilyn inez heather])
```

Running the solver by `{ExploreOne {AssignCrew Flights Crew}}`. and invoking the Oz Browser on the solution node results in:



```
1 #
[[tom david jeremy inez]
 [ron joe bill fred cathy]
 [bob mario ed carol jean]
 [tom david jeremy janet tracy inez]
 [ron joe bill fred marilyn carolyn cathy]
 [bob mario ed jean]
 [tom david jeremy carol inez]
 [ron joe bill fred janet cathy]
 [bob mario ed tracy marilyn jean]
 [tom david jeremy carol carolyn inez heather]]
```

The flights are to be attended in the order they appear in the solution list. Each sublist denotes the assignment for an individual flight.

Scheduling a Golf Tournament

Problem Specification There are 32 individually playing golfers who play in groups of 4, so-called foursomes. For every week of the golf tournament new sets of foursomes are to be compiled. The task is to assign foursomes for a given number of weeks such that no player plays with another player in a foursome twice.

Maximal Number of Weeks The upper bound for the number of weeks is 10 weeks due to the following argument. There are $\binom{32}{2} = 496$ pairing of players. Each foursome takes 6 pairings and every week consists of 8 foursomes, hence, every week occupies 48 pairings. Having only 496 pairings available, at most $\lfloor 496/48 \rfloor = 10$ weeks can be assigned without duplicating foursomes. Unfortunately, only assignments for 9 weeks could be found so far. Fortunately again, this assignment could only be found by solvers using set constraints. Other approaches, using linear integer programming, failed for this problem size.

Model A foursome is modeled as a set of cardinality 4. A week is a collection of foursomes and all foursomes of a week are pairwise disjoint and their union is the set of all golfers. This leads to a partition constraint. Further, each foursome shares at most one element with any other foursome, since a golfer, of course, may occur in different foursomes but only on his own. Therefore, the cardinality of the intersection of a foursome with any other foursome of the other weeks has to be either 0 or 1.

Distribution Strategy The distribution strategy is crucial for this problem.[1] A player is taken and assigned to all possible foursomes. Then the next player is taken and assigned and so on. This player-wise distribution allows to solve instances of that problems up to 9 weeks. The approach, coming usually into mind first, to distribute a foursome completely, fails even for smaller instances of the problem.

Solver The function `Golf` returns a solver to find an assignment for `NbOfFourSomes` foursomes per week and `NbOfWeeks` weeks duration. The number of players is computed from `NbOfFourSomes` and stored in `NbOfPlayers`. The auxiliary function `Flatten` is used to flatten a list of lists of variables. Its definition is necessary since the library function of the same name works only on ground terms.

[1]The distribution strategy was proposed by Stefano Novello from IC-PARC on the newsgroup comp.constraints.

The procedure `DistrPlayers` implements the player-wise distribution strategy. It tries to create for every player on every foursome a choice point by simply enumerating all players and iterating for each player over all foursomes.

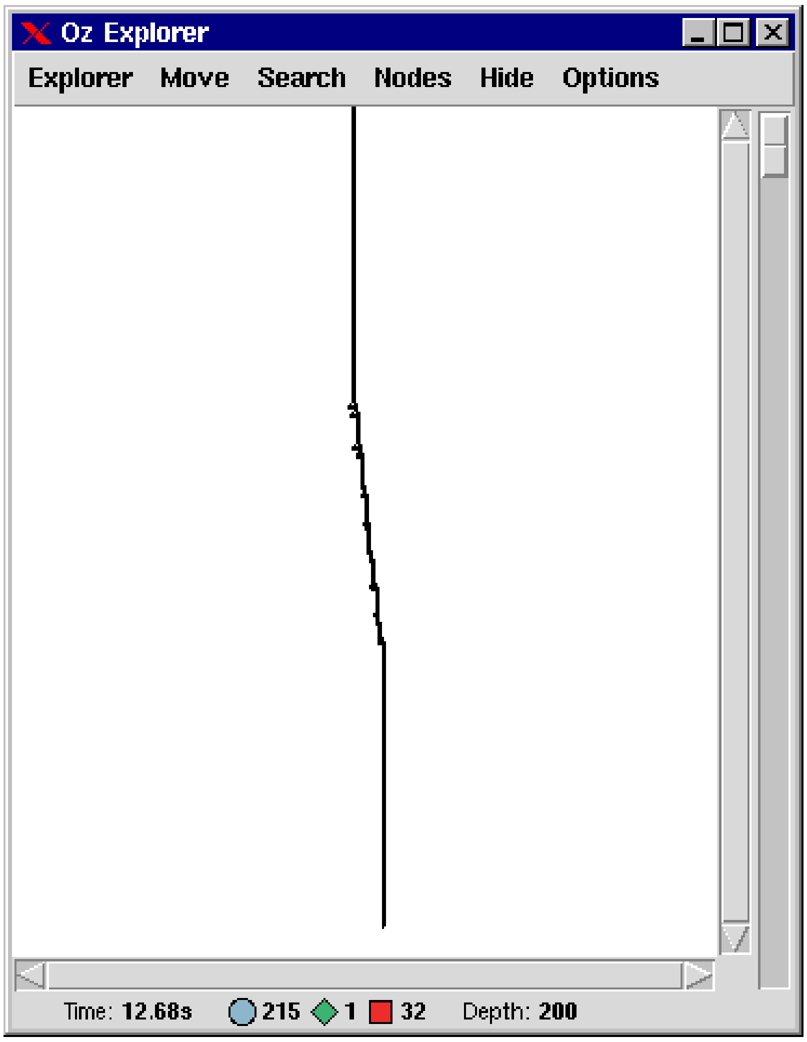
The variable `Weeks` holds `NbOfWeeks` weeks. A week is a list of foursomes. A foursome is modeled as a set. All sets are subsets of $\{1,\ldots,\text{NbOfPlayers}\}$ and have exactly 4 elements. Further, the sets modeling the foursomes of a week form a partition of the set $\{1,\ldots,\text{NbOfPlayers}\}$. These constraints are imposed by the first `ForAll` loop.

The following nested loops (`ForAllTail` and `ForAll`) impose that every foursome shares at most one element with any other foursome of other weeks. Finally, the distribution procedure is called with a flattened copy of `Weeks`, i.e., a list of all foursomes.

```
declare
fun {Golf NbOfWeeks NbOfFourSomes}
   NbOfPlayers = 4*NbOfFourSomes

   fun {Flatten Ls}
      {FoldL Ls fun {$ L R}
                   if R==nil then L
                   else {Append L R} end
                end nil}
   end

   proc {DistrPlayers AllWeeks Player Weeks}
      choice
         case Weeks
         of FourSome|Rest then
            dis {FS.include Player FourSome} then
               {DistrPlayers AllWeeks Player Rest}
            [] {FS.exclude Player FourSome} then
               {DistrPlayers AllWeeks Player Rest}
            end
         else
            if Player < NbOfPlayers then
               {DistrPlayers AllWeeks Player+1 AllWeeks}
            else skip end
         end
      end
   end
in
   proc {$ Weeks}
      FlattenedWeeks
   in
      Weeks = {MakeList NbOfWeeks}

      {ForAll Weeks
       proc {$ Week}
          Week =
          {FS.var.list.upperBound
```

```
           NbOfFourSomes [1#NbOfPlayers]}
          {ForAll Week proc {$ FourSome}
                          {FS.card FourSome 4}
                       end}
          {FS.partition Week
           {FS.value.make [1#NbOfPlayers]}}
       end}

      {ForAllTail Weeks
       proc {$ WTails}
          case WTails
          of Week|RestWeeks then
             {ForAll Week
              proc {$ FourSome}
                 {ForAll {Flatten RestWeeks}
                  proc {$ RestFourSome}
                     {FS.cardRange 0 1
                      {FS.intersect
                       FourSome RestFourSome}}
                  end}
              end}
          else skip end
       end}

      FlattenedWeeks = {Flatten Weeks}
      {DistrPlayers FlattenedWeeks 1 FlattenedWeeks}
   end
end
```

Invoking the solver by `{ExploreOne {Golf 9 8}}` produces the following search tree.



Time: 12.68s ● 215 ◆ 1 🟥 32 Depth: 200

The search tree has a depth of 200 which makes the problem a good candidate for recomputation. Invoking the search engine with a computation depth of one [2] requires 64.1 MB of heap memory. On the other hand an recomputation depth of 10 [3] decreases the required heap memory to 19.3 MB.

[2] `declare S = {Search.one.depth {Golf 9 8} 1 _}`

[3] `declare S = {Search.one.depth {Golf 9 8} 10 _}`

Bibliography

[1] C.C. Lindner and A. Rosa. Topics on steiner systems. In *Annals of Discrete Mathematics*, volume 7. North Holland, 1980.

[2] Tobias Müller. Solving set partitioning problems with constraint programming. In *Proceedings of the Sixth International Conference on the Practical Application of Prolog and the Forth International Conference on the Practical Application of Constraint Technology – PAPPACT98*, pages 313–332, London, UK, March 1998. The Practical Application Company Ltd.

[3] Tobias Müller and Martin Müller. Finite set constraints in Oz. In François Bry, Burkhard Freitag, and Dietmar Seipel, editors, *13. Workshop Logische Programmierung*, pages 104–115, Technische Universität München, 17–19 September 1997.