CHIP Example Code

Limited Warranty:

This document is provided for information purposes only and subject to change without any prior notice. COSYTEC does not provide any warranties covering the information provided and specifically disclaims any liability in connection with this document.

Trademarks:

The CHIP V5 C++ Library is the property of COSYTEC.
The CHIP V5 C Library is the property of COSYTEC.
The CHIP V5 Prolog is the property of COSYTEC.

Authors:

Philip Kay

Reference Number:

Reference: COSY/WHITE/003
Version: 1.0
Revision : A
Date : April 1997

COPYRIGHT:

COSYTEC SA
Parc Club Orsay Université
4, Rue Jean Rostand
F-91893 Orsay Cedex, France
Tel. +33 1 60 19 37 38
Fax. +33 1 60 19 36 20
Email. help@cosytec.fr

1. The Ship Loading Example

1.1 The Problem

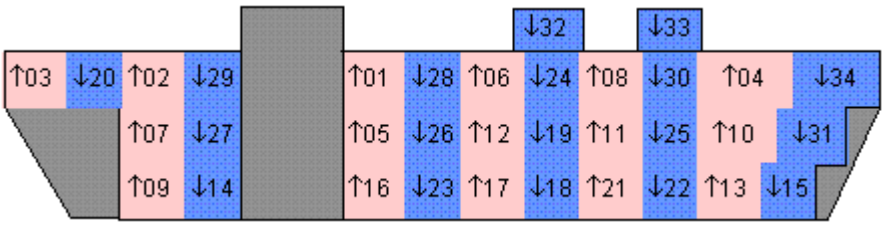
This example shows how to solve a scheduling problem where cumulative and precedence constraints occur using either the CHIP V5 Prolog, C Library or C++ Library versions. The problem is to find a schedule that minimizes the time to unload and to load a ship.

The work consists of 34 unloading/loading tasks. Each task requires a number of men and each has a given duration i.e. task1 requires 3 men and will take 4 hours to complete. We know that we have a maximum of 8 men for work and the total work should not exceed 100 hours.

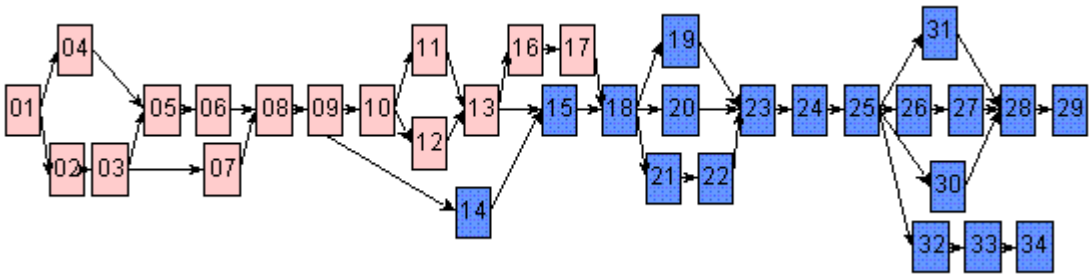
The data for the problem is shown below and is used as input for the example:

Task	Hours	Men	Successor	Task	Hours	Men	Successor
1	3	4	2, 4	18	2	7	19, 20, 21
2	4	4	3	19	1	4	23
3	4	3	5, 7	20	1	4	23
4	6	4	5	21	1	4	22
5	5	5	6	22	2	4	23
6	2	5	8	23	4	7	24
7	3	4	8	24	5	8	25
8	4	3	9	25	2	8	26, 30, 31, 32
9	3	4	10, 14	26	1	3	27
10	2	8	11, 12	27	1	3	28
11	3	4	13	28	2	6	29
12	2	5	13	29	1	8	-
13	1	4	15, 16	30	3	3	28
14	5	3	15	31	2	3	28
15	2	3	18	32	1	3	33
16	3	3	17	33	2	3	34
17	2	6	18	34	2	3	-

We also know the sequence of the activities in terms of successor activities i.e. task1 is succeeded by task2 and task4. This sequence is important to respect as the cargo is stacked on top of each other and must be unloaded in the correct order so the ship does not capsize. Loading tasks have a light shade and generally precede the unloading tasks which have a dark shade.



From this data we are able to draw the sequence graphically as:



The above drawing was produced using the **diffn** constraint for 2D placement, this is not discused in this example but more information can be found in the CHIP V5 demo set.

1.2 Solving the Problem

1.2.1 The Modelisation

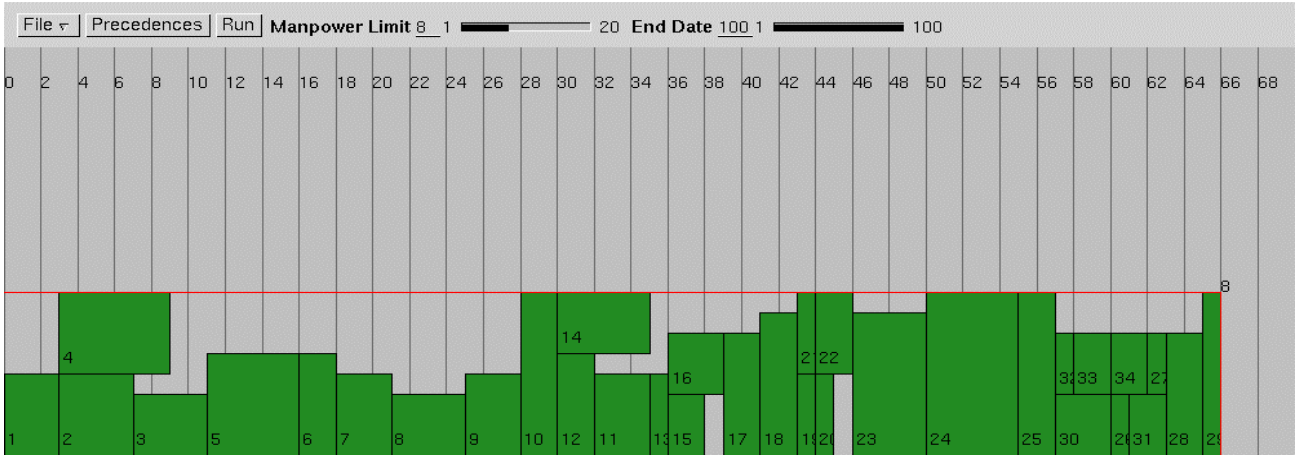
Each task is defined by 3 decision variables (*Si*, *Di*, *Mi*): starting date, duration and manpower. As data we have the number of tasks (*N*=34), their corresponding duration *Di* as a list of integers and manpower *Mi* also as a list of integers. All these tasks share the same manpower which is limited to 8 persons (*Upper*=8). The time horizon to schedule all these tasks is 100 hours (*Last*=100). In order to compute when each task starts in time we create a list of decision variables *Si*'s which range from 0 to 100.

Since the manpower is limited and shared by all tasks we use the **cumulative** constraint to solve the conflict and which accepts as arguments *Si*'s, *Di*'s and *Mi*'s. Also, the cumulative constraint requires the upper limit of resources which is a domain variable (*Limit*) ranging from 0 to 8. The aim is to minimize the total schedule and for that we create a domain variable End which is passed as an additional argument to cumulative constraint. To respect the sequence of tasks we use the inequality greater than arithmetic constraints of CHIP.

The optimization is achieved by using the built-in CHIP functionality, **min_max**, which controls the search tree. The search uses the **most_constrained** heuristic which selects the variables which have the most constraints placed on them.

1.2.2 The Solution

Below is a graphical representation of (one of) the optimum solutions of 66 hours, respecting the precedence of tasks and overall manpower maximum of 8 men:



When executed the CHIP V5 examples produce the following output. The solution given assigns a start date of 0 for task1, 3 for task2, etc. task29 which has a duration of 1 hour should start in hour 65 (hence an optimum solution of 66 hours).

```
[0, 3, 7, 3, 14, 19, 11, 21, 25, 28, 30, 33, 35, 30, 36, 36, 39, 41, 43, 44, 43, 44,
46, 50, 55, 60, 62, 63, 65, 57, 61, 57, 58, 60]
```

1.3 Example Code

1.3.1 CHIP V5 C++ Library

To solve the problem with CHIP V5 C++ Library we express the durations and manpower as integer arrays and assign these to domain variables *Dis* and *Mis*. We create an array of domain variables *Sis*, *Limit* and *End* to be used by the **ChipCumulative** constraint.

The sequence of the tasks is represented in the data as an array of structures containing the indices of the tasks and we use overloading of the operator **>=** to state the inequality constraint, we create an instance of **ChipCumulative** to handle the manpower conflict and an instance of **ChipMinmax** with the **METHOD_MOST_CONSTRAINED** heuristic to search for solutions to the problem.

```
#include "chipcc.h"

int durs[]={3,4,4,6,5,2,3,4,3,2,3,2,1,5,2,3,2,2,1,1,1,2,4,5,2,1,1,2,1,3,2,1,2,2};
int mans[]={4,4,3,4,5,5,4,3,4,8,4,5,4,3,3,3,6,7,4,4,4,4,7,8,8,3,3,6,8,3,3,3,3,3};

typedef struct _AFTER_ {
  int I; int J;
} I_after_J, * I_after_JPtr;

I_after_J prec[] = {
  {2,1},{3,2},{4,1},{5,3},{5,4},{6,5},{7,3},{8,6},{8,7},{9,8},{10,9},{11,10},
  {12,10},{13,11},{13,12},{14,9},{15,13},{15,14},{16,13},{17,16},{18,15},{18,17},
  {19,18},{20,18},{21,18},{22,21},{23,19},{23,20},{23,22},{24,23},{25,24},{26,25},
  {27,26},{28,27},{28,30},{28,31},{29,28},{30,25},{31,25},{32,25},{33,32},{34,33}
};
int ctrs = sizeof(prec)/sizeof(I_after_J);

void shipCC(int n, int Upper, int Last) {
  int i, setup = TRUE;
  ChipDvar Limit(1, Upper); // Resource Limit
  ChipDvar End(1, Last); // End of the schedule
  ChipDvars Sis(n, 0, Last); // All start variables
  ChipCumEntries entries(n); // Entries for cumulative

  for(i = 0; (i < n); i++){
    ChipDvar Di(durs[i], durs[i]); // Create one cumulative entry
    ChipDvar Mi(mans[i], mans[i]);
    entries[i+1] = ChipCumEntry(Sis[i+1], Di, Mi);
  }
  for(i = 0; (i < ctrs) && setup; i++){
    ChipDvar s1 = Sis[prec[i].I];
    ChipDvar s2 = Sis[prec[i].J];
    if (ChipPost(s1 >= s2 + durs[prec[i].J - 1]) == ChipFail) {
      cout << "Precedence failed" << endl;
      setup = FALSE;
```

```
    }
  }
  ChipCumulative cum(entries);
  cum.setHigh(Limit);
  cum.setEnd(End);
  if(setup && (cum.ChipPost() == ChipFail)){ // Post the cumulative constraint
    cout << "Cumulative failed" << endl;
    setup = FALSE;
  }
  ChipLabeling labeling(Sis);
  labeling->setSelectionMethod(METHOD_MOST_CONSTRAINED);
  ChipMinmax opt(labeling, End);
  if(setup && (opt->ChipPost() == ChipFail)){
    cout << "Optimisation failed" << endl;
  }
  cout << Sis << endl;
}

void main(){
  ChipInit();
  shipCC(34, 8, 100);
  ChipEnd();
}
```

1.3.2 CHIP V5 C Library

To solve the problem with CHIP V5 C Library we express the durations and manpower as integer arrays and assign these to domain variables *Dis* and *Mis*. We create an array of domain variables *Sis*, *Limit* and *End* to be used by the **c_cumulative** constraint.

The sequence of the tasks is represented in the data as an array of structures containing the indices of the tasks and we use **c_dom_domcst** to state the inequality constraint, we call **c_cumulative** to handle the manpower conflict and **c_min_max** with the **METHOD_MOST_CONSTRAINED** heuristic to search for solutions to the problem.

```
#include<chipc.h>

int durs[]={3,4,4,6,5,2,3,4,3,2,3,2,1,5,2,3,2,2,1,1,1,2,4,5,2,1,1,2,1,3,2,1,2,2};
int mans[]={4,4,3,4,5,5,4,3,4,8,4,5,4,3,3,3,6,7,4,4,4,4,7,8,8,3,3,6,8,3,3,3,3,3};

typedef struct _AFTER_ {
  int I; int J;
} I_after_J, *I_after_JPtr;

I_after_J prec[] = {
  {2,1},{3,2},{4,1},{5,3},{5,4},{6,5},{7,3},{8,6},{8,7},{9,8},{10,9},{11,10},
  {12,10},{13,11},{13,12},{14,9},{15,13},{15,14},{16,13},{17,16},{18,15},{18,17},
  {19,18},{20,18},{21,18},{22,21},{23,19},{23,20},{23,22},{24,23},{25,24},{26,25},
  {27,26},{28,27},{28,30},{28,31},{29,28},{30,25},{31,25},{32,25},{33,32},{34,33}
};
int ctrs = sizeof(prec)/sizeof(I_after_J);

void ship(int n, int Upper, int Last){
  int i;
  DvarPtr *Sis = (DvarPtr *) c_chip_alloc(n);
  DvarPtr *Dis = (DvarPtr *) c_chip_alloc(n);
  DvarPtr *Mis = (DvarPtr *) c_chip_alloc(n);
  DvarPtr Limit, End;

  c_create_domain_array(Sis, n, 0, Last, CONSEC); /* Create domains */
  for(i = 0; i < n; i++) {
    c_create_domain_array(&Dis[i], 1, durs[i], durs[i], INTERVAL);
    c_create_domain_array(&Mis[i], 1, mans[i], mans[i], INTERVAL);
  }
  c_create_domain_array(&Limit, 1, 1, Upper, CONSEC); /* Resource Limit */
  c_create_domain_array(&End, 1, 1, Last, CONSEC); /* End of the schedule*/
```

```
  /* Setup all precedence constraints */
  for(i = 0; i < ctrs; i++) {
    if(c_dom_domcst(Sis[prec[i].I - 1], GREATER_EQUAL,
      Sis[prec[i].J - 1], durs[prec[i].J - 1]) == FAIL){
      (void) fprintf(stdout, "precedence failed [%d, %d]\n", prec[i].I, prec[i].J);
      (void) exit(0);
    }
  }
  /* Setup the cumulative constraint */
  if(c_cumulative(n, Sis, Dis, Mis, NULL, NULL, Limit, End, NULL, NULL) == FAIL){
    (void) fprintf(stdout, "cumulative failed\n");
    (void) exit(0);
  }
  /* Generate */
  if(c_min_max(Sis, n, &End, 1, METHOD_MOST_CONSTRAINED, NULL, NULL) == FAIL){
    (void) fprintf(stdout, "optimsation failed\n");
    (void) exit(0);
  }
}

void main(){
  c_chip_init();
  ship(34, 8, 100);
  c_chip_end();
}
```

1.3.3 CHIP V5 Prolog

To solve the problem in CHIP V5 Prolog we express the data for the problem in the predicate data/3. This is used to create a list of domain variables *Sis*, and domain variables *Limit* and *End* to be used by the **cumulative** constraint.

The sequence of the tasks is represented in the data as a list of constraints and we use a feature of Prolog, *meta-programming*, to set up these constraints. In order to have direct access to the *Si* variables and *Di* variables we use the *univ* predicate to transform elements of the list into terms. The data for the inequality constraint is of the form *After* #>= *Before* which means: task at position *After* should succeed the task at position *Before*.

After stating the inequality we call **cumulative** to handle the manpower conflict and **min_max** with the **most_constrained** heuristic to search for solutions to the problem.

```
top:-
  run(8, 100).

run(Upper, Last):-
  data(N, Dis, Mis),
  length(Sis, N),
  Sis :: 0..Last,
  Limit :: 0..Upper,
  End :: 0..Last,
  precedences(L),
  set_precedences(L, Sis, Dis),
  cumulative(Sis, Dis, Mis, unused, unused, Limit, End, unused),
  min_max(labeling(Sis), End),
  writeln(Sis).

labeling(Sis):-
  labeling(Sis, 0, most_constrained, indomain).

set_precedences(L, Sis, Dis):-
  Array_starts=..[starts|Sis], % starts(S1, S2, S3, ..)
  Array_durations=..[durations|Dis], % durations(D1, D2, D3, ..)
  set_pre_lp(L, Array_starts, Array_durations).
```

```
set_pre_lp([],_,_).
set_pre_lp([After #>= Before|R], Array_starts, Array_durations):-
  arg(After, Array_starts, S2),
  arg(Before, Array_starts, S1),
  arg(Before, Array_durations, D1),
  S2 #>= S1 + D1,
  set_pre_lp(R, Array_starts, Array_durations).

% nr of tasks, duration of tasks, resource use of tasks
data(34,[3,4,4,6,5,2,3,4,3,2,3,2,1,5,2,3,2,2,1,1,1,2,4,5,2,1,1,2,1,3,2,1,2,2],
  [4,4,3,4,5,5,4,3,4,8,4,5,4,3,3,3,6,7,4,4,4,4,7,8,8,3,3,6,8,3,3,3,3,3]).

% after #>= before, task indices
precedences([2#>=1, 3#>=2, 4#>=1, 5#>=3, 5#>=4, 6#>=5, 7#>=3, 8#>=6, 8#>=7, 9#>=8,
  10#>=9, 11#>=10, 12#>=10, 13#>=11, 13#>=12, 14#>=9, 15#>=13, 15#>=14, 16#>=13,
  17#>=16, 18#>=15, 18#>=17, 19#>=18, 20#>=18, 21#>=18, 22#>=21, 23#>=19, 23#>=20,
  23#>=22, 24#>=23, 25#>=24, 26#>=25, 27#>=26, 28#>=27, 28#>=30, 28#>=31, 29#>=28,
  30#>=25, 31#>=25, 32#>=25, 33#>=32, 34#>=33]).
```

2. The Progressive Party Example

2.1 The Problem

This example shows how to solve a form of bin packing problem using the cumulative constraints with the CHIP V5 Prolog, C Library or C++ Library versions. We only give code for the solver part and do not dwell on graphical representation of the solution; this can be found in the CHIP V5 demo set.

This problem has become known as the *progressive party problem* and was originally taken from an Operational Research confernce (Jerusalem EUROXIV) and was recently revisited at the International Conference on Constraint Programming (Cassis CP95).</P>

The problem is to organise a party for 42 yatch teams at a regatta. Some of the teams act as hosts and have an available capacity ranging from 4 to 10 guests on their boat. The remaining teams act as guests and proceed to visit a number of the host boats, the size of the guest teams is known and ranges from 2 to 7 persons. In all there are 13 host boats and 29 guest teams.

Host capacities (13)	10,10,9,8,8,8,8,8,8,7,6,4,4
Size of Guest teams (29)	7,6,5,5,5,4,4,4,4,4,4,4,4,3,3,2,2,2,2,2,2,2,2,2,2,2,2,2,2

The guest teams cannot be split and party must take place over 7 consecutive hours, each guest team changing boat at each hour. The constraints to be respected are the boat capacity, and that any two groups meet only once. The goal is to minimise the number of boats needed to organise the meetings.

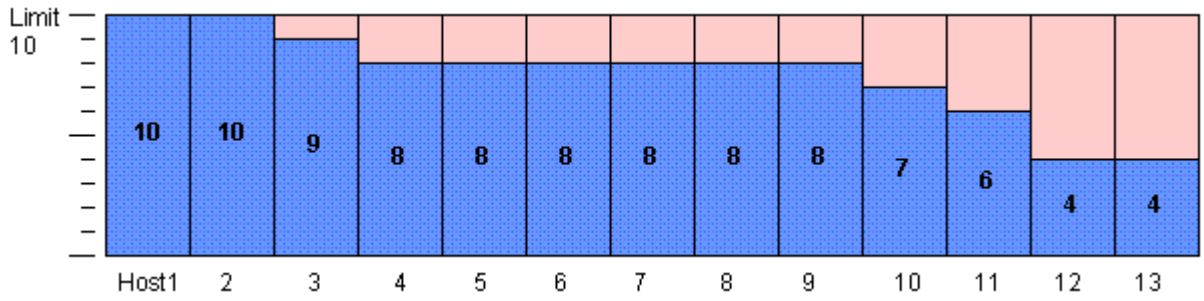
2.2 Solving the Problem

2.2.1 The Modelisation

To solve this example we model it in the form of a **bin packing** problem. We are able to express the available capacity of a host boat by using the **cumulative** constraint, we ensure that a guest team does not return to the same boat with the **alldifferent** constraint and use **conditional propagation** to express that the guest teams should only meet once at a party.

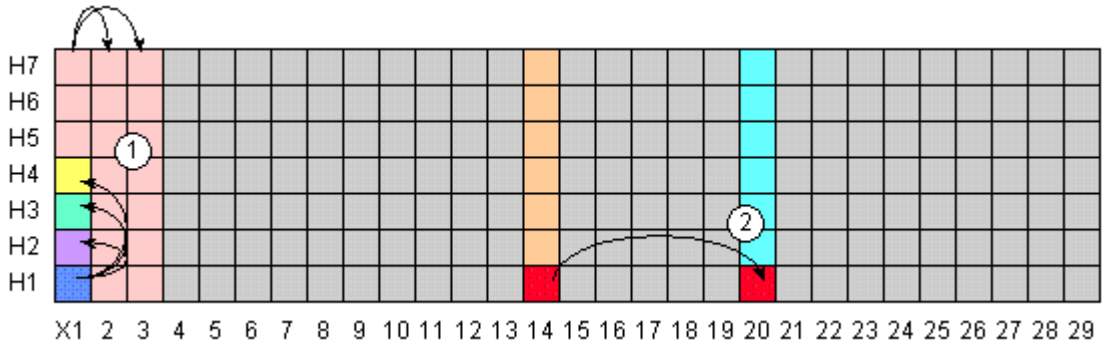
The first part of each solution defines the domain varaibles. Here we create 7 lists, one for each hour of the party, containing 29 variables, one for each guest team. In the solution these represent the host boat for each team for each hour of the party.

We next prepare the data for the **bin packing** and build a profile for the **cumulative** contraint. The general form for the constraint is to express each task in the form of its start *Si*, duration *Di* and resource *Ri*. For the **bin packing** we are not concerned with the duration so these are set to 1 for all tasks. We use the maximum available capacity of the largest boat as the *Limit* for the constraint and build a profile of dummy tasks for those boats which do not have this capacity.



This is shown above graphically; the dark area is a host's capacity and the light area the dummy tasks created. This profile is then appended to the lists of guest variables *Si*, *Di*, and *Ri* to state the **cumulative** constraint. For each hour we create the *Limit* domain variable ranging from 0 to 10.

To ensure that a team does not return to the same boat we use the **alldifferent** constraint which states that for each guest team *X1, X2, .., X29* for each hour *H1, H2, .., H7* the host boat must be different, this is done for all apirs of *Xn* and *Hn*. The diagram below **1** illustrates this.



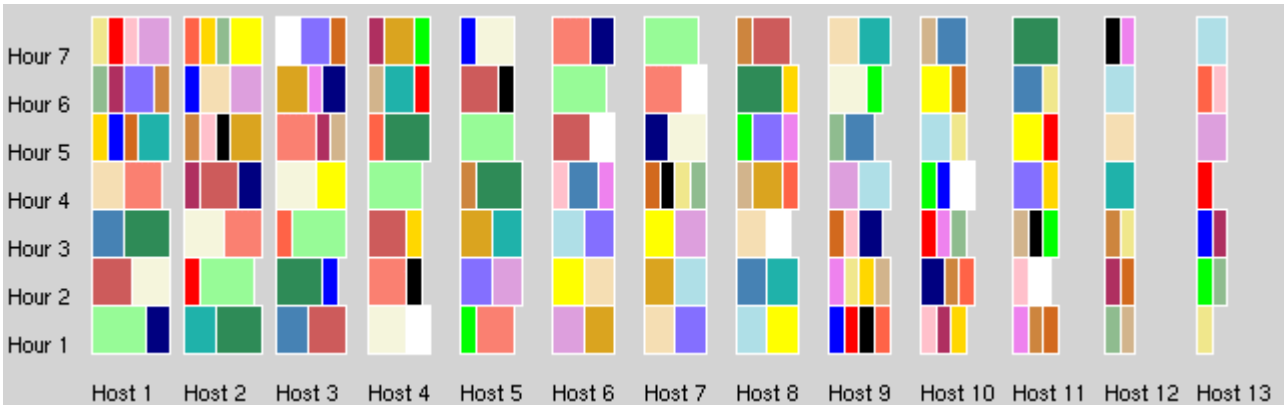
To ensure that two teams do not meet again we use **conditional propagation** - this expresses a condition which must be satisfied when the variables are assigned values. We state that if the host boat is the same for two guest teams then they cannot be the same for the other hours. i.e. as in **2** above, if *X14* is the same as *X20* during the first hour *H1* then *H2* to *H7* cannot be the same, this is done for all pairs of *Xn* and *Hn*.

To avoid redundant symmetry we further state that for one of the teams its assignment should be sequential, i.e. for guest team *X1* the first party on *H1* should be before *H2* and *H2* before *H3*, etc.

For the search heuristics we use **indomain** to assign values to the domain variables. As a strategy we chose to alternate between assigning first a large team and then a small team. This avoids trying to assign two large teams together early in the search and also trying to assign several small teams together at the end of the search.

2.2.2 The Solution

The display of the solution below shows each host boat as an column, each hour of the party as a row in the chart and each team as a coloured rectangle.



The capacity for each hosts is displayed as the width of each column i.e. host1 has a capacity of 10 guests, host13 has 4 guests, etc. The size of the guest team is displayed as the width of each rectangle i.e. team1 is made up of 7 persons, team29 has 2 persons, etc. You can quite clearly see the progression of each team from boat to boat, and if you check, can see that no two teams meet each other again.

```
[1,2,3,4,5,2,3,6,6,7,7,8,8,1,4,5,9,9,9,9,10,10,10,11,11,11,12,12,13]
[2,3,1,1,4,8,8,5,7,5,6,6,7,10,11,13,2,3,4,10,9,11,12,9,10,12,9,13,9]
[3,1,4,2,2,5,1,7,5,6,8,7,6,9,8,11,10,13,11,3,4,9,13,10,12,9,11,10,12]
[4,5,2,3,1,12,6,9,8,11,1,3,9,2,10,10,13,10,7,8,11,6,2,6,5,7,8,7,7]
[5,4,6,7,3,1,9,13,2,8,12,11,10,7,6,8,11,1,2,4,1,2,3,8,2,1,3,9,10]
[6,8,5,9,7,4,11,2,3,1,2,10,12,3,7,9,4,2,5,13,8,13,1,3,1,10,4,1,11]
[7,11,8,5,6,9,10,1,4,3,9,2,13,6,3,4,1,5,12,2,2,1,4,12,8,3,10,2,1]
```

The output of running the CHIP V5 programs produces the above results, this gives a teams corresponding host for each hour of the problem i.e. team 7 is assigned to hosts 3, 8, 1, 6, 9, 11, 10. At the first party it meets team 3, at next team 6, and so on.

2.3 ExampleCode

2.3.1 CHIP V5 C++ Library

```
#include "chipcc.h"

const int NB_HOURS = 7;
const int NB_HOSTS = 13;
int HostsCapacity[] = {10,10,9,8,8,8,8,8,8,7,6,4,4};
const int NB_GUESTS = 29;

int Guests[] = {7,6,5,5,5,4,4,4,4,4,4,4,4,3,3,2,2,2,2,2,2,2,2,2,2,2,2,2,2};

int DisplaySolutionVerbosely = 0;

class GuestFamily;
class HostBoat;

GuestFamily** TheGuestFamilies;
HostBoat** TheHostBoats;
int MaxHostCapacity = 0;
```

```
int TotalHostCapacity = 0;
int TotalGuestNumbers = 0;

class GuestFamily
{
private:
  int _no;
  int _numberOfMembers;
  ChipDvars _hostVariables;
public:
  GuestFamily(){};
  GuestFamily(int no, int number);
  ~GuestFamily(){};
  void setNo(int no) { _no = no;}
  void setNumberOfMembers(int nb) { _numberOfMembers = nb;}
  int getNumberOfMembers() { return _numberOfMembers;}
  void setHostVariables(ChipDvars vars) {g_hostVariables = vars; }
  ChipDvars& getHostVariables() { return _hostVariables;}
  ChipDvar& getHostVariable(int hour) { return _hostVariables[hour];}
  ChipResult constraintAllHostsDifferent();
  ChipResult orderHostVariables();
  void display();
};

GuestFamily::GuestFamily(int no, int number) : _no(no), _numberOfMembers(number)
{}

ChipResult GuestFamily::constraintAllHostsDifferent()
{
  // just a ChipAlldifferent
  return ChipPost(ChipAlldifferent(_hostVariables));
}

ChipResult GuestFamily::orderHostVariables()
{
  int hour =0;
  for (hour =2; hour <= NB_HOURS; hour++)
    if (ChipPost(_hostVariables[hour] > _hostVariables[hour-1]) != ChipSucceed)
      return ChipFail;
  return ChipSucceed;
}

void GuestFamily::display()
{
  if (DisplaySolutionVerbosely) {
    cout << "The guest family number " <<g_no << " with " << _numberOfMembers <<
    " members has the following program: " << endl;
    for (int hour =1; hour <= NB_HOURS; hour++) {
      cout << "H" << hour << ":boat " << _hostVariables[hour];
      if (hour < NB_HOURS)
        cout << ", ";
      else
        cout << "." << endl;
    }
  }
  else {
    cout << "guest_host( " <<g_no << " , [";
    for (int hour =1; hour <= NB_HOURS; hour++) {
      cout << _hostVariables[hour];
      if (hour < NB_HOURS)
        cout << ", ";
    }
    cout << "])." << endl;
  }
}

class HostBoat
{
private:
  int _no;
  int _capacity;
  ChipCumEntry _complementaryCumEntry;
public:
  HostBoat() {};
  HostBoat(int no, int capacity);
  ~HostBoat() {};
```

```
  int getCapacity() { return _capacity;}
  int buildComplementaryCumEntry(int maxCapacity);
  void addComplementaryCumEntry(ChipCumEntries& entries);
};

HostBoat::HostBoat(int no, int capacity) : _no(no), _capacity(capacity)
{
}

int HostBoat::buildComplementaryCumEntry(int maxCapacity)
{
  if (_capacity < maxCapacity) {
    _complementaryCumEntry = ChipCumEntry(_no, 1, maxCapacity-_capacity);
    return 1;
  }
  else {
    return 0;
  }
}

void HostBoat::addComplementaryCumEntry(ChipCumEntries& entries)
{
  if (_capacity < MaxHostCapacity)g// (!null(_complementaryCumEntry))
    entries += _complementaryCumEntry;
}

void BuildData()
{
  int no;
  int capacity;
  MaxHostCapacity = 0;

  // Allocating NB_HOSTS objets
  TheHostBoats = new HostBoat* [NB_HOSTS+1];
  for (no = 1; no <= NB_HOSTS; no++) {
    capacity = HostsCapacity[no-1];
    TheHostBoats[no -1] = new HostBoat(no, capacity);
    if (capacity > MaxHostCapacity )
      MaxHostCapacity = capacity;
    TotalHostCapacity += capacity;
  }

  TheGuestFamilies = new GuestFamily* [NB_GUESTS];
  for (no = 1; no <= NB_GUESTS; no++) {
    TheGuestFamilies[no -1] = new GuestFamily(no, Guests[no-1]);
    TotalGuestNumbers += Guests[no-1];
  }
}

void DisplaySolution()
{
  int no;

  for (no = 1; no <= NB_GUESTS; no++) {
    TheGuestFamilies[no-1]->display();
  }
}

ChipResult PostCapacityConstraints()
{
  int no;
  int hour = 0;
  int nbExtraEntries = 0;

  // checking global capacity is OK
  if (TotalHostCapacity < TotalGuestNumbers)
    return ChipFail;

  // building entries shared by different hours
  for (no = 1; no <= NB_HOSTS; no++) {
    TheHostBoats[no -1]->buildComplementaryCumEntry(MaxHostCapacity);
  }

  for (hour =1; hour <= NB_HOURS; hour++) {
    ChipCumEntries entries1;
    // adding entries for guests to be placed
```

```
    for (no = 1; no <= NB_GUESTS; no++) {
      entries1 += ChipCumEntry(TheGuestFamilies[no -1]->getHostVariable(hour), 1,
      TheGuestFamilies[no -1]->getNumberOfMembers());
    }
    for (no = 1; no <= NB_HOSTS; no++) {
      TheHostBoats[no -1]->addComplementaryCumEntry(entries1);
    }
    ChipCumulative cumulative1(entries1);
    cumulative1.setHigh(ChipDvar(0, MaxHostCapacity));
    cumulative1.setEnd(NB_HOSTS + 1);
    if (!entries1.isEmpty() && (ChipPost(cumulative1) != ChipSucceed))
      return ChipFail;
  }
  return ChipSucceed;
}

ChipResult PostFamiliesMeetOnlyOnce()
{
  int no1, no2;
  int hour;

  for (no1 = 1; no1 <= NB_GUESTS; no1++) {
    for (no2 = no1+1; no2 <= NB_GUESTS; no2++) {
      // These two families meet only once
      ChipDlinTerm linearTerm;
      for (hour =1; hour <= NB_HOURS; hour++) {
        // for each hour create a boolean var for occurence of a meeting
        ChipDvar equalityVar(0, 1);
        if (ChipPost(ChipEntailment((TheGuestFamilies[no1 -1]->getHostVariable(hour)
          == TheGuestFamilies[no2 -1]->getHostVariable(hour) + 0),
          equalityVar)) == ChipSucceed)
          linearTerm += equalityVar;
      }
      // posting the constraint for these two families:
      // number of meetings<=1

      if (ChipPost(linearTerm <= 1) != ChipSucceed)
        return ChipFail;
    }
  }
  return ChipSucceed;
}

ChipResult BuildProblemConstraints()
{
  int no;

  // creating the variables for each GuestFamily
  for (no = 1; no <= NB_GUESTS; no++) {
    TheGuestFamilies[no -1]->setHostVariables(ChipDvars(NB_HOURS, 1, NB_HOSTS));
    // and post the difference constraint
    TheGuestFamilies[no -1]->constraintAllHostsDifferent();
  }

  // contraint the first family to order variables -> visit of bigger boats before
  if (TheGuestFamilies[0]->orderHostVariables() != ChipSucceed) {
    cout << "Ordering of variables for first family failed" << endl;
    return ChipFail;
  }
  else
    cout << "Ordering of variables for first family succeeded" << endl;

  if (PostFamiliesMeetOnlyOnce() != ChipSucceed) {
    cout << "Posting of constraint for families meeting only once failed" << endl;
    return ChipFail;
  }
  else
    cout << "Posting of constraint for families meeting only once succeeded" << endl;

  // post capacity constraint
  if (PostCapacityConstraints() != ChipSucceed) {
    cout << "Posting of capacity constraints failed" << endl;
    return ChipFail;
  }
  else
    cout << "Posting of capacity constraintsgsucceeded" << endl;
```

```
  return ChipSucceed;
}

ChipResult SolveProblem()
{
  int no, noMax, hour;
  int nbHours = NB_HOURS;
  int nbGuests = NB_GUESTS;

  // Choice of solution
  // Take the heuristic of putting variables hour by hour,
  // putting guest families with smaller indexes before (they are the bigger one)
  // Here an optimisation consist in putting the smaller families in between
  // to improve the liability of big and small guest families to meet
  // Labeling in the input order

  ChipDvars AllVariables;

  for (hour = 1; hour <= nbHours; hour++) {
    for (no = 1, noMax = nbGuests; (no <= nbGuests) && (no<=noMax) ; no++, noMax--) {
      AllVariables += TheGuestFamilies[no -1]->getHostVariable(hour);
      if (noMax>no)
        AllVariables += TheGuestFamilies[noMax -1]->getHostVariable(hour);
      else
        break;
    }
  }

  // creating labeling object
  ChipLabeling labeling(AllVariables);
  labeling.setSelectionMethod(METHOD_INPUT_ORDER);
  labeling.indomainMin();
  cout << "Starting for " << nbHours << " hours " << nbGuests << " families" << endl;
  if (ChipPost(labeling) == ChipSucceed) {
    cout << "Solution for " << nbHours << " hours " << nbGuests << " families" << endl;
    return ChipSucceed;
  }
  else {
    cout << "No solution for " << nbHours <<" hours " << nbGuests << " families" <<
endl;
    return ChipFail;
  }
}

main()
{
  ChipInit();
  BuildData();
  if ((BuildProblemConstraints() == ChipSucceed) && (SolveProblem() == ChipSucceed))
    DisplaySolution();
  ChipEnd();
}
```

2.3.2 CHIP V5 C Library

```
#include "chipc.h"

#define NB_HOURS 7
#define NB_HOSTS 13
#define NB_GUESTS 29
int host[] = {10,10,9,8,8,8,8,8,8,7,6,4,4};
int guest[] = {7,6,5,5,5,4,4,4,4,4,4,4,4,3,3,2,2,2,2,2,2,2,2,2,2,2,2,2,2};
int max_host;

void solve_problem()
{
  int i, j, no, no1, no2;
  DvarPtr HostVars[NB_GUESTS][NB_HOURS];
  int zeros[NB_HOURS];
```

```
  int ones[NB_HOURS];
  int comps = 0;
  int level;
  Labeling enumerate;
  DvarPtr * AllVariables = (DvarPtr *) c_chip_alloc(NB_GUESTS*NB_HOURS);

  for (i = 0; i < NB_HOURS; i++) {
    zeros[i] = 0;
    ones[i] = 1;
  }
  for (no = 0; no < NB_GUESTS; no++) {
    c_create_domain_array(&(HostVars[no][0]), NB_HOURS, 1, NB_HOSTS, INTERVAL);
    if (c_alldifferent(&(HostVars[no][0]), zeros, NB_HOURS) == FAIL) {
      (void) printf("Fails: alldifferent\n");
      return;
    }
  }
  for (i=0; i<NB_HOURS-1; i++) {
    if (c_dom_domcst(HostVars[0][i], LESS, HostVars[0][i+1], 0) == FAIL) {
      (void) printf("Fails: ordering of first line\n");
      return;
    }
  }
  for(no1 = 0; no1 < NB_GUESTS; no1++) {
    for(no2 = no1+1; no2 < NB_GUESTS; no2++) {
      DvarPtr* equalityDemons= (DvarPtr *) c_chip_alloc(NB_HOURS);
      c_create_domain_array(equalityDemons, NB_HOURS, 0, 1, CONSEC);

      for(i = 0; i < NB_HOURS; i++) {
        c_entailment(HostVars[no1][i], EQUAL, HostVars[no2][i], 0, equalityDemons[i]);
      }
      if(c_dom_linear_constrain(equalityDemons, ones, NB_HOURS, LESS_EQUAL, 1)==FAIL) {
        (void) printf("Fails: limitation of meeting occurences\n");
        return;
      }
    }
  }
  for(no = 0; no < NB_HOSTS; no++) {
    if(host[no] < max_host )
    comps++;
  }

  for(i = 0; i < NB_HOURS; i++) {
    DvarPtr * Start = (DvarPtr *) c_chip_alloc(NB_GUESTS+comps);
    DvarPtr * Dur   = (DvarPtr *) c_chip_alloc(NB_GUESTS+comps);
    DvarPtr * Res   = (DvarPtr *) c_chip_alloc(NB_GUESTS+comps);
    DvarPtr   High;
    DvarPtr   End;

    for (no = 0; no < NB_GUESTS; no++) {
      Start[no] = HostVars[no][i];
      c_create_domain_array(&Dur[no], 1, 1, 1, INTERVAL);
      c_create_domain_array(&Res[no], 1, guest[no], guest[no], INTERVAL);
    }

    for (j = 0, no = 0; no < NB_HOSTS; no++) {
      if (host[no] < max_host ) {
        level = max_host - host[no];
        c_create_domain_array(&Start[NB_GUESTS+j], 1, no+1, no+1, INTERVAL);
        c_create_domain_array(&Dur[NB_GUESTS+j], 1, 1, 1, INTERVAL);
        c_create_domain_array(&Res[NB_GUESTS+j], 1, level, level, INTERVAL);
        j++;
      }
    }
    c_create_domain_array(&High, 1, 1, max_host, INTERVAL);
    c_create_domain_array(&End, 1, 1, NB_HOSTS+1, INTERVAL);
    if (c_cumulative(NB_GUESTS+comps,
      Start , Dur, Res, NULL, NULL, High, End, NULL, NULL) == FAIL) {
      (void) printf("Fails: max capacity\n");
      return;
    }
  }
  enumerate.type = LABELING_INDOMAIN;
  enumerate.indomain = LABELING_INDOMAIN_MIN;
  enumerate.solution = LABELING_FIRST_SOL;
  enumerate.continue_var = NULL;
```

```
  enumerate.continue_all = NULL;

  j = 0;
  for (i = 0; i < NB_HOURS; i++) {
    for (no = 0, no2 = NB_GUESTS-1; (no < NB_GUESTS); no++, no2--) {
      AllVariables[j] = HostVars[no][i];
      j++;
      if (no2>no)
        AllVariables[j++] = HostVars[no2][i];
      else
        break;
    }
  }
  if (c_labeling(AllVariables, NB_GUESTS*NB_HOURS, METHOD_INPUT_ORDER,
    &enumerate)==TRUE) {
    for (no = 0; no < NB_GUESTS; no++) {
      (void) printf("\nGuest %d: ", no+1);
      c_domain_array_print(stdout, &(HostVars[no][0]), NB_HOURS);
    }
  }
  else {
    (void) printf("no solution\n");
  }
}

void main() {
  c_chip_init();
  max_host = host[0];
  solve_problem();
  c_chip_end();
}
```

2.3.3 CHIP V5 Prolog

```
top:-
  hosts(H),
  length(H, Hn),
  guests(G),
  length(G, N),
  length(One, N),
  One :: 1..1,
  host_profile(H, Profile),
  length(X, 7),
  create_vars(X, N, Hn, One, G, Profile),
  different(X, [First|Guests]),
  pairing([First|Guests]),
  ordered(First),
  label(X).

host_profile(L, profile(S, D, R, Limit)):-
  Limit :: 0..10,
  maximum(Limit, L),
  profile(L, S, D, R, 1, Limit).

profile([], [], [], [], _, _).
profile([Limit|X1], S1, D1, R1, N, Limit):-
  !,
  N1 is N + 1,
  profile(X1, S1, D1, R1, N1, Limit).
profile([X|X1], [N|S1], [1|D1], [R|R1], N, Limit):-
  R is Limit - X,
  N1 is N + 1,
  profile(X1, S1, D1, R1, N1, Limit).

create_vars([], _, _, _, _, _).
create_vars([X|List], N, Hn, One, G, Profile):-
  length(X, N),
  X :: 1..Hn,
  bin(X, One, G, Profile),
  create_vars(List, N, Hn, One, G, Profile).
```

```
bin(Start, Dur, Res, profile(Start1, Dur1, Res1, Limit)):-
  append(Start, Start1, S),
  append(Dur, Dur1, D),
  append(Res, Res1, R),
  L :: 0..Limit,
  cumulative(S, D, R, unused, unused, L, 14, unused).

different([[]|OtherLists], []).
different(LofLists, [ListForFirst| Rest]):-
  extract_first_from_lol(LofLists, ListForFirst, RL),
  alldifferent(ListForFirst),
  different(RL, Rest).

extract_first_from_lol([], [], []).
extract_first_from_lol([[First|R1]| R2], [First|RF], [R1|RR]):-
  extract_first_from_lol(R2, RF, RR).

label([]).
label([L|List]):-
  reverse(L, R),
  label(L, R),
  writeln(L),
  label(List).

label([], _).
label([X|List], RList):-
  indomain(X, min),
  label(RList, List).

pairing([]).
pairing([H|T]):-
  pairing(H, T),
  pairing(T).

pairing(_, []).
pairing(X, [G|G1]):-
  pair(X, G),
  pairing(X, G1).

pair([], []).
pair([X|X1], [Y|Y1]):-
  if X #= Y then
    pair_diff(X1, Y1)
  else
    true,
  pair(X1, Y1).

pair_diff([], []).
pair_diff([X1|XX], [Y1|YY]):-
  X1 #\= Y1,
  pair_diff(XX, YY).

ordered([]).
ordered([_]).
ordered([X, Y|R]):-
  X #< Y,
  ordered([Y|R]).

hosts([10,10,9,8,8,8,8,8,8,7,6,4,4]).

guests([7,6,5,5,5,4,4,4,4,4,4,4,4,3,3,2,2,2,2,2,2,2,2,2,2,2,2,2,2]).
```

3. Bibliography

R. Faure
Le déchargement et le Chargement d'un Navire
Exercises et Problèmes Résolus de Recherche Opérationnelle T3 Vol T3, pages 279-282, Masson 1991

Aggoun, N. Beldiceanu
Extending CHIP in Order to Solve Complex Scheduling and Placement Problems
Journal of Mathematical and Computer Modelling, Vol. 17, No. 7, pages 57-73Pergamon Press, 1993

B. Smith, S. Brailsford, P. M. Hubbard, H.P. Williams
The Progressive Party Problem: Integer Linear Programming and Constraint Propagation compared
Constraints: An International Journal, Vol1 1&2, September 1996