Chapter 2: First steps with or-tools: cryptarithmetic puzzles

or-tools library

Nikolaj van Omme   Laurent Perron

April 19, 2012

- **Prerequisites:**

- **Prerequisites:**
  - Some basic knowledge of C++.

- **Prerequisites:**
  - Some basic knowledge of C++.
  - Some basic knowledge of Constraint Programming.

- **Prerequisites:**
  - Some basic knowledge of C++.
  - Some basic knowledge of Constraint Programming.

- **Code:**

- **Prerequisites:**
  - Some basic knowledge of C++.
  - Some basic knowledge of Constraint Programming.

- **Code:** (`documentation/tutorials/C++/chap2`)

- **Prerequisites:**
  - Some basic knowledge of C++.
  - Some basic knowledge of Constraint Programming.

- **Code:** (`documentation/tutorials/C++/chap2`)
  - `cp_is_fun1.cc`: A simple cryptarithmetic puzzle.

- **Prerequisites:**
  - Some basic knowledge of C++.
  - Some basic knowledge of Constraint Programming.

- **Code:** (`documentation/tutorials/C++/chap2`)
  - `cp_is_fun1.cc`: A simple cryptarithmetic puzzle.
  - `cp_is_fun2.cc`: Use of `SolutionCollectors` to collect some or all solutions.

- **Prerequisites:**
  - Some basic knowledge of C++.
  - Some basic knowledge of Constraint Programming.

- **Code:** (`documentation/tutorials/C++/chap2`)
  - `cp_is_fun1.cc`: A simple cryptarithmetic puzzle.
  - `cp_is_fun2.cc`: Use of `SolutionCollectors` to collect some or all solutions.
  - `cp_is_fun3.cc`: Use of the Google gflags library to parse command line parameters.

- **Prerequisites:**
  - Some basic knowledge of C++.
  - Some basic knowledge of Constraint Programming.

- **Code:** (`documentation/tutorials/C++/chap2`)
  - `cp_is_fun1.cc`: A simple cryptarithmetic puzzle.
  - `cp_is_fun2.cc`: Use of `SolutionCollectors` to collect some or all solutions.
  - `cp_is_fun3.cc`: Use of the Google gflags library to parse command line parameters.
  - `cp_is_fun4.cc`: Use of parameters.

- A simple cryptarithmetic puzzle.

- A simple cryptarithmetic puzzle.

```
      C P
+     I S
+   F U N
---------
= T R U E
```

```
      C P
+     I S
+   F U N
---------
= T R U E
```

- A simple cryptarithmetic puzzle.
- A feasible solution:
  `C=2 P=3 I=7 S=4 F=9 U=6 N=8 T=1 R=0 E=5`

```
      C P
+     I S
+   F U N
---------
= T R U E
```

- A simple cryptarithmetic puzzle.
- A feasible solution:
  `C=2 P=3 I=7 S=4 F=9 U=6 N=8 T=1 R=0 E=5`
- Indeed: `23+74+968 = 1065`

```
      2 3
+     7 4
+   9 6 8
---------
= 1 0 6 5
```

```
      C P
+     I S
+   F U N
---------
= T R U E
```

- A simple cryptarithmetic puzzle.
- A feasible solution:
  `C=2 P=3 I=7 S=4 F=9 U=6 N=8 T=1 R=0 E=5`
- Indeed: `23+74+968 = 1065`
- 72 solutions!

```
      2 3
+     7 4
+   9 6 8
---------
= 1 0 6 5
```

```
      C P
+     I S
+   F U N
---------
= T R U E
```

- A simple cryptarithmetic puzzle.
- A feasible solution:
  `C=2 P=3 I=7 S=4 F=9 U=6 N=8 T=1 R=0 E=5`
- Indeed: `23+74+968 = 1065`
- 72 solutions! In base 10!

```
      2 3
+     7 4
+   9 6 8
---------
= 1 0 6 5
```

- **Describe:**

- **Describe:**

- **Model:**

- **Describe:**

- **Model:**

- **Solve:**

- **Describe:**
  - Goal of the puzzle?
- **Model:**
- **Solve:**

- **Describe:**
  - Goal of the puzzle?
    replace letters by digits such that `CP+IS+FUN=TRUE`
- **Model:**
- **Solve:**

- **Describe:**
  - Goal of the puzzle?
    replace letters by digits such that `CP+IS+FUN=TRUE`
  - What are the unknowns?
- **Model:**
- **Solve:**

- **Describe:**
  - Goal of the puzzle?
    replace letters by digits such that `CP+IS+FUN=TRUE`
  - What are the unknowns?
    one letter = one variable
- **Model:**
- **Solve:**

- **Describe:**
  - Goal of the puzzle?
    replace letters by digits such that `CP+IS+FUN=TRUE`
  - What are the unknowns?
    one letter = one variable
  - What are the constraints?

- **Model:**

- **Solve:**

- **Describe:**
  - Goal of the puzzle?
    replace letters by digits such that `CP+IS+FUN=TRUE`
  - What are the unknowns?
    one letter = one variable
  - What are the constraints?
    - sum has to be verified

- **Model:**

- **Solve:**

- **Describe:**
  - Goal of the puzzle?
    replace letters by digits such that `CP+IS+FUN=TRUE`
  - What are the unknowns?
    one letter $=$ one variable
  - What are the constraints?
    - sum has to be verified
    - two different letters represent two different digits

- **Model:**

- **Solve:**

- **Describe:**
  - Goal of the puzzle?
    replace letters by digits such that `CP+IS+FUN=TRUE`
  - What are the unknowns?
    one letter = one variable
  - What are the constraints?
    - sum has to be verified
    - two different letters represent two different digits
    - first digit of a number can not be 0
- **Model:**
- **Solve:**

- **Describe:**
  - Goal of the puzzle?
    replace letters by digits such that `CP+IS+FUN=TRUE`
  - What are the unknowns?
    one letter = one variable
  - What are the constraints?
    - sum has to be verified
    - two different letters represent two different digits
    - first digit of a number can not be 0
    - 10 letters: we need at least 10 different digits
- **Model:**
- **Solve:**

- **Describe:**
  - Goal of the puzzle?
    replace letters by digits such that `CP+IS+FUN=TRUE`
  - What are the unknowns?
    one letter = one variable
  - What are the constraints?
    - sum has to be verified
    - two different letters represent two different digits
    - first digit of a number can not be 0
    - ~~10 letters: we need at least 10 different digits~~
- **Model:**
- **Solve:**

- **Describe:**
  - Goal of the puzzle?
    replace letters by digits such that `CP+IS+FUN=TRUE`
  - What are the unknowns?
    one letter = one variable
  - What are the constraints?
    - sum has to be verified
    - two different letters represent two different digits
    - first digit of a number can not be 0
    - ~~10 letters: we need at least 10 different digits~~
- **Model:**
  - Given a base `b`, digits range from 0 to `b` − 1
- **Solve:**

- **Describe:**
  - Goal of the puzzle?
    replace letters by digits such that `CP+IS+FUN=TRUE`
  - What are the unknowns?
    one letter = one variable
  - What are the constraints?
    - sum has to be verified
    - two different letters represent two different digits
    - first digit of a number can not be 0
    - ~~10 letters: we need at least 10 different digits~~
- **Model:**
  - Given a base $\mathtt{b}$, digits range from 0 to $\mathtt{b}-1$
  - $\mathtt{C}\cdot\mathtt{b}+\mathtt{P}+\mathtt{I}\cdot\mathtt{b}+\mathtt{S}+\mathtt{F}\cdot\mathtt{b}^2+\mathtt{U}\cdot\mathtt{b}+\mathtt{N}=\mathtt{T}\cdot\mathtt{b}^3+\mathtt{R}\cdot\mathtt{b}^2+\mathtt{U}\cdot\mathtt{b}+\mathtt{E}$.
- **Solve:**

- **Describe:**
  - Goal of the puzzle?
    replace letters by digits such that `CP+IS+FUN=TRUE`
  - What are the unknowns?
    one letter = one variable
  - What are the constraints?
    - sum has to be verified
    - two different letters represent two different digits
    - first digit of a number can not be 0
    - ~~10 letters: we need at least 10 different digits~~
- **Model:**
  - Given a base b, digits range from 0 to $b - 1$
  - $C \cdot b + P + I \cdot b + S + F \cdot b^2 + U \cdot b + N = T \cdot b^3 + R \cdot b^2 + U \cdot b + E$.
  - `AllDifferent(C,P,I,S,F,U,N,T,R,E)`
- **Solve:**

- **Describe:**
  - Goal of the puzzle?
    replace letters by digits such that `CP+IS+FUN=TRUE`
  - What are the unknowns?
    one letter = one variable
  - What are the constraints?
    - sum has to be verified
    - two different letters represent two different digits
    - first digit of a number can not be 0
    - ~~10 letters: we need at least 10 different digits~~
- **Model:**
  - Given a base $\mathtt{b}$, digits range from 0 to $\mathtt{b}-1$
  - $\mathtt{C}\cdot\mathtt{b}+\mathtt{P}+\mathtt{I}\cdot\mathtt{b}+\mathtt{S}+\mathtt{F}\cdot\mathtt{b}^2+\mathtt{U}\cdot\mathtt{b}+\mathtt{N}=\mathtt{T}\cdot\mathtt{b}^3+\mathtt{R}\cdot\mathtt{b}^2+\mathtt{U}\cdot\mathtt{b}+\mathtt{E}$.
  - `AllDifferent(C,P,I,S,F,U,N,T,R,E)`
  - $\mathtt{C},\mathtt{I},\mathtt{F},\mathtt{T}\in[1,\mathtt{b}-1]$ and $\mathtt{P},\mathtt{S},\mathtt{U},\mathtt{N},\mathtt{R},\mathtt{E}\in[0,\mathtt{b}-1]$
- **Solve:**

- **Describe:**
  - Goal of the puzzle?
    replace letters by digits such that `CP+IS+FUN=TRUE`
  - What are the unknowns?
    one letter = one variable
  - What are the constraints?
    - sum has to be verified
    - two different letters represent two different digits
    - first digit of a number can not be 0
    - ~~10 letters: we need at least 10 different digits~~
- **Model:**
  - Given a base $\mathtt{b}$, digits range from 0 to $\mathtt{b}-1$
  - $\mathtt{C}\cdot\mathtt{b}+\mathtt{P}+\mathtt{I}\cdot\mathtt{b}+\mathtt{S}+\mathtt{F}\cdot\mathtt{b}^2+\mathtt{U}\cdot\mathtt{b}+\mathtt{N}=\mathtt{T}\cdot\mathtt{b}^3+\mathtt{R}\cdot\mathtt{b}^2+\mathtt{U}\cdot\mathtt{b}+\mathtt{E}.$
  - `AllDifferent(C,P,I,S,F,U,N,T,R,E)`
  - $\mathtt{C},\mathtt{I},\mathtt{F},\mathtt{T}\in[1,\mathtt{b}-1]$ and $\mathtt{P},\mathtt{S},\mathtt{U},\mathtt{N},\mathtt{R},\mathtt{E}\in[0,\mathtt{b}-1]$
- **Solve:** Later ...

Licensed under the Apache License, Version 2.0

Headers

Namespace

Helper functions

Is it?

CP solver

Variables

Assert-like macro

Constraints

```
// Copyright 2012 Google
...
#include <vector>

#include "base/logging.h"
#include "constraint_solver/constraint_solver.h"

namespace operations_research {

// helper functions
IntVar* const MakeBaseLine2(Solver*  s,
...
}

IntVar* const MakeBaseLine3(Solver* s,
...
}

IntVar* const MakeBaseLine4(Solver* s,
...
}

void CPIsFun() {
  // Constraint programming engine
  Solver solver("CP is fun!");

  const int64 kBase = 10;

  // Decision variables
  IntVar* const c = solver.MakeIntVar(1, kBase - 1, "C");
...
  IntVar* const e = solver.MakeIntVar(0, kBase - 1, "E");

  // We need to group variables in a vector to be able to use
  // the global constraint AllDifferent
  std::vector<IntVar*> letters;
  letters.push_back(c);
...
   letters.push_back(e);

   // Check if we have enough digits
   CHECK_GE(kBase, letters.size());

   // Constraints
   solver.AddConstraint(solver.MakeAllDifferent(letters       ));

   // CP + IS + FUN = TRUE
   IntVar* const term1 = MakeBaseLine2(&solver, c, p, kBase);
   IntVar* const term2 = MakeBaseLine2(&solver, i, s, kBase);
   IntVar* const term3 = MakeBaseLine3(&solver, f, u, n, kBase);
   IntVar* const sum_terms = solver.MakeSum(solver.MakeSum(term1,
                                                           term2),
                                            term3)->Var();

   IntVar* const sum    = MakeBaseLine4(&solver, t, r, u, e, kBase);

   solver.AddConstraint(solver.MakeEquality(sum_terms, sum));
```

Decision builder

Search

Main function

```

  DecisionBuilder* const db = solver.MakePhase(letters,
                                               Solver::CHOOSE_FIRST_UNBOUND,
                                               Solver::ASSIGN_MIN_VALUE);

  solver.NewSearch(db);

  if (solver.NextSolution()) {
    LOG(INFO) << "Solution found:";
    LOG(INFO) << "C=" << c->Value() << " " << "P=" << p->Value() << " "
...
  } else {
    LOG(INFO) << "Cannot solve problem.";
  }  // if (solver.NextSolution())

  solver.EndSearch();
}

}  // namespace operations_research

// ----- MAIN -----
int main(int argc, char **argv) {
  operations_research::CPIsFun();
  return 0;
}
```

To use the library, we need to include a few headers:

To use the library, we need to include a few headers:

```
#include "base/logging.h"
#include "constraint_solver/constraint_solver.h"
```

To use the library, we need to include a few headers:

```
#include "base/logging.h"
#include "constraint_solver/constraint_solver.h"
```

- `logging.h`: logging facilities and some assert-like macros.

To use the library, we need to include a few headers:

```
#include "base/logging.h"
#include "constraint_solver/constraint_solver.h"
```

- `logging.h`: logging facilities and some assert-like macros.
- `constraint_solver.h`: main entry point to the CP solver.

The whole library is nested in the namespace `operations_research`:

The whole library is nested in the namespace `operations_research`:

```
namespace operations_research {
  IntVar* const MakeBaseLine2(...) {
    ...
  }
  ...
  void CPIsFun() {
    // Magic happens here!
  }
}  // namespace operations_research
```

The whole library is nested in the namespace `operations_research`:

```
namespace operations_research {
  IntVar* const MakeBaseLine2(...) {
    ...
  }
  ...
  void CPIsFun() {
    // Magic happens here!
  }
}  // namespace operations_research
```

`CPIsFun()` is where all the magic happens:

The whole library is nested in the namespace `operations_research`:

```
namespace operations_research {
  IntVar* const MakeBaseLine2(...) {
    ...
  }
  ...
  void CPIsFun() {
    // Magic happens here!
  }
}  // namespace operations_research
```

`CPIsFun()` is where all the magic happens:

```
int main(int argc, char **argv) {
  operations_research::CPIsFun();
  return 0;
}
```

The CP solver:

The CP solver:

- is the main engine to solve an instance;

The CP solver:

- is the main engine to solve an instance;
- is responsible for the creation of the model;

The CP solver:

- is the main engine to solve an instance;
- is responsible for the creation of the model;
- has a very rich Application Programming Interface (API) and

The CP solver:

- is the main engine to solve an instance;
- is responsible for the creation of the model;
- has a very rich Application Programming Interface (API) and
- provides a lots of functionalities.

The CP solver:

- is the main engine to solve an instance;
- is responsible for the creation of the model;
- has a very rich Application Programming Interface (API) and
- provides a lots of functionalities.

The CP solver is created as follows:

The CP solver:

- is the main engine to solve an instance;
- is responsible for the creation of the model;
- has a very rich Application Programming Interface (API) and
- provides a lots of functionalities.

The CP solver is created as follows:

```
Solver solver("CP is fun!");
```

To create the decision variables:

To create the decision variables:

```
const int64 kBase = 10;
IntVar* const c = solver.MakeIntVar(1, kBase - 1, "C");
IntVar* const p = solver.MakeIntVar(0, kBase - 1, "P");
...
IntVar* const e = solver.MakeIntVar(0, kBase - 1, "E");
```

To create the decision variables:

```
const int64 kBase = 10;
IntVar* const c = solver.MakeIntVar(1, kBase - 1, "C");
IntVar* const p = solver.MakeIntVar(0, kBase - 1, "P");
...
IntVar* const e = solver.MakeIntVar(0, kBase - 1, "E");
```

Factory methods in or-tools

The solver API provides numerous factory methods to create different objects. These methods start with `Make` and return a pointer to the newly created object.
The solver automatically takes ownership of these objects and deletes them appropriately.

To create the decision variables:

```
const int64 kBase = 10;
IntVar* const c = solver.MakeIntVar(1, kBase - 1, "C");
IntVar* const p = solver.MakeIntVar(0, kBase - 1, "P");
...
IntVar* const e = solver.MakeIntVar(0, kBase - 1, "E");
```

Factory methods in or-tools

The solver API provides numerous factory methods to create different objects. These methods start with `Make` and return a pointer to the newly created object.
The solver automatically takes ownership of these objects and deletes them appropriately.

Warning:

Never delete explicitly an object created by a factory method!

- Assert-like macros are defined in the header `logging.h`.

- Assert-like macros are defined in the header `logging.h`.
- Base has to be at greater than or equal to 10:

- Assert-like macros are defined in the header `logging.h`.
- Base has to be at greater than or equal to 10:

```
CHECK_GE(kBase, letters.size());
```

- Assert-like macros are defined in the header `logging.h`.
- Base has to be at greater than or equal to 10:

```
CHECK_GE(kBase, letters.size());
```

- `CHECK_GE(x,y)` checks if condition (x) >= (y) is true.

- Assert-like macros are defined in the header `logging.h`.
- Base has to be at greater than or equal to 10:

```
CHECK_GE(kBase, letters.size());
```

- `CHECK_GE(x,y)` checks if condition (x) >= (y) is true.
- If not, the program is aborted and the cause is printed:

```
[23:51:34] examples/cp_is_fun1.cc:108: Check failed:
                                                    (kBase) >= (letters.size())
Aborted
```

To multiply an integer variable with an integer constant:

To multiply an integer variable with an integer constant:

```
IntVar* const var1 = solver.MakeIntVar(0, 1, "Var1");
IntVar* const var2 = solver.MakeProd(var1,36)->Var();
```

To multiply an integer variable with an integer constant:

```
IntVar* const var1 = solver.MakeIntVar(0, 1, "Var1");
IntVar* const var2 = solver.MakeProd(var1,36)->Var();
```

To add two IntVar given by their respective pointers:

To multiply an integer variable with an integer constant:

```
IntVar* const var1 = solver.MakeIntVar(0, 1, "Var1");
IntVar* const var2 = solver.MakeProd(var1,36)->Var();
```

To add two IntVar given by their respective pointers:

```
IntVar* const var3 = solver.MakeSum(var1,var2)->Var();
```

To multiply an integer variable with an integer constant:

```
IntVar* const var1 = solver.MakeIntVar(0, 1, "Var1");
IntVar* const var2 = solver.MakeProd(var1,36)->Var();
```

To add two IntVar given by their respective pointers:

```
IntVar* const var3 = solver.MakeSum(var1,var2)->Var();
```

Warning:

Never store a pointer to an `IntExpr` nor a `BaseIntExpr` in the code. The safe code should always call `Var()` on an expression built by the solver, and store the object as an `IntVar*`.

To construct a sum, we use a combination of `MakeSum()` and `MakeProd()` factory methods:

To construct a sum, we use a combination of `MakeSum()` and `MakeProd()` factory methods:

```
IntVar* const term1 = solver.MakeSum(solver.MakeProd(c,kBase),p)->Var();
```

To construct a sum, we use a combination of `MakeSum()` and `MakeProd()` factory methods:

```
IntVar* const term1 = solver.MakeSum(solver.MakeProd(c,kBase),p)->Var();
```

If the number of terms in the sum to construct is large, you can use `MakeScalProd()`:

To construct a sum, we use a combination of `MakeSum()` and `MakeProd()` factory methods:

```
IntVar* const term1 = solver.MakeSum(solver.MakeProd(c,kBase),p)->Var();
```

If the number of terms in the sum to construct is large, you can use `MakeScalProd()`:

```
IntVar* const var1 = solver.MakeInt(...);
...
IntVar* const varN = solver.MakeInt(...);
std::vector<IntVar*> variables;
variables.push_back(var1);
...
variables.push_back(varN);

std::vector<int64> coefficients(N);
// fill vector with coefficients
...
IntVar* const sum = solver.MakeScalProd(variables, coefficients)->Var();
```

To create the sum constraint, we use the factory method `MakeEquality()` that returns a pointer to a `Constraint` object:

To create the sum constraint, we use the factory method `MakeEquality()` that returns a pointer to a `Constraint` object:

```
IntVar* const term1 = ...
IntVar* const term2 = ...
IntVar* const term3 = ...

IntVar* const sum_terms = solver.MakeSum(solver.MakeSum(term1,
                                                          term2),
                                         term3)->Var();
IntVar* const sum = ...

Constraint* const sum_constraint = solver.MakeEquality(sum_terms, sum);
```

To create the sum constraint, we use the factory method `MakeEquality()` that returns a pointer to a `Constraint` object:

```
IntVar* const term1 = ...
IntVar* const term2 = ...
IntVar* const term3 = ...

IntVar* const sum_terms = solver.MakeSum(solver.MakeSum(term1,
                                                        term2),
                                          term3)->Var();
IntVar* const sum = ...

Constraint* const sum_constraint = solver.MakeEquality(sum_terms, sum);
```

To add a constraint:

To create the sum constraint, we use the factory method `MakeEquality()` that returns a pointer to a `Constraint` object:

```
IntVar* const term1 = ...
IntVar* const term2 = ...
IntVar* const term3 = ...

IntVar* const sum_terms = solver.MakeSum(solver.MakeSum(term1,
                                                        term2),
                                         term3)->Var();
IntVar* const sum = ...

Constraint* const sum_constraint = solver.MakeEquality(sum_terms, sum);
```

To add a constraint:

```
solver.AddConstraint(sum_constraint);
```

To create the sum constraint, we use the factory method `MakeEquality()` that returns a pointer to a `Constraint` object:

```
IntVar* const term1 = ...
IntVar* const term2 = ...
IntVar* const term3 = ...

IntVar* const sum_terms = solver.MakeSum(solver.MakeSum(term1,
                                                        term2),
                                         term3)->Var();
IntVar* const sum = ...

Constraint* const sum_constraint = solver.MakeEquality(sum_terms, sum);
```

To add a constraint:

```
solver.AddConstraint(sum_constraint);
```

or:

To create the sum constraint, we use the factory method `MakeEquality()` that returns a pointer to a `Constraint` object:

```
IntVar* const term1 = ...
IntVar* const term2 = ...
IntVar* const term3 = ...

IntVar* const sum_terms = solver.MakeSum(solver.MakeSum(term1,
                                                         term2),
                                         term3)->Var();
IntVar* const sum = ...

Constraint* const sum_constraint = solver.MakeEquality(sum_terms, sum);
```

To add a constraint:

```
solver.AddConstraint(sum_constraint);
```

or:

```
solver.AddConstraint(solver.MakeEquality(sum_terms, sum));
```

The global `AllDifferent` constraint:

The global `AllDifferent` constraint:

```
std::vector<IntVar*> letters;
letters.push_back(c);
letters.push_back(p);
...
letters.push_back(e);

solver.AddConstraint(solver.MakeAllDifferent(letters));
```

A `DecisionBuilder` is responsible for creating the actual search tree.

A `DecisionBuilder` is responsible for creating the actual search tree.

```
DecisionBuilder* const db = solver.MakePhase(letters,
                                             Solver::CHOOSE_FIRST_UNBOUND,
                                             Solver::ASSIGN_MIN_VALUE);
```

A `DecisionBuilder` is responsible for creating the actual search tree.

```
DecisionBuilder* const db = solver.MakePhase(letters,
                                             Solver::CHOOSE_FIRST_UNBOUND,
                                             Solver::ASSIGN_MIN_VALUE);
```

Parameters of the method `MakePhase`:

A `DecisionBuilder` is responsible for creating the actual search tree.

```
DecisionBuilder* const db = solver.MakePhase(letters,
                                             Solver::CHOOSE_FIRST_UNBOUND,
                                             Solver::ASSIGN_MIN_VALUE);
```

Parameters of the method `MakePhase`:

- `letters`: `std::vector` with pointers to the `IntVar` decision variables.

A DecisionBuilder is responsible for creating the actual search tree.

```
DecisionBuilder* const db = solver.MakePhase(letters,
                                             Solver::CHOOSE_FIRST_UNBOUND,
                                             Solver::ASSIGN_MIN_VALUE);
```

Parameters of the method MakePhase:

- letters: std::vector with pointers to the IntVar decision variables.
- Solver::CHOOSE_FIRST_UNBOUND: next selected IntVar variable in the search is the first unbounded variable.

A `DecisionBuilder` is responsible for creating the actual search tree.

```
DecisionBuilder* const db = solver.MakePhase(letters,
                                              Solver::CHOOSE_FIRST_UNBOUND,
                                              Solver::ASSIGN_MIN_VALUE);
```

Parameters of the method `MakePhase`:

- `letters`: `std::vector` with pointers to the `IntVar` decision variables.
- `Solver::CHOOSE_FIRST_UNBOUND`: next selected `IntVar` variable in the search is the first unbounded variable.
- `Solver::ASSIGN_MIN_VALUE`: assign the smallest available value to the selected `IntVar`.

To prepare for a new search:

To prepare for a new search:

```
DecisionBuilder* const db = ...
solver.NewSearch(db);
```

To prepare for a new search:

```
DecisionBuilder* const db = ...
solver.NewSearch(db);
```

To actually search for the next solution in the search tree:

To prepare for a new search:

```
DecisionBuilder* const db = ...
solver.NewSearch(db);
```

To actually search for the next solution in the search tree:

```
if (solver.NextSolution()) {
  // Do something with the current solution
} else {
  // The search is finished
}
```

We print out the found solution and check if it is valid:

We print out the found solution and check if it is valid:

```
if (solver.NextSolution()) {
  LOG(INFO) << "Solution found:";
  LOG(INFO) << "C=" << c->Value() << " " << "P=" << p->Value() << " "
            << "I=" << i->Value() << " " << "S=" << s->Value() << " "
            << "F=" << f->Value() << " " << "U=" << u->Value() << " "
            << "N=" << n->Value() << " " << "T=" << t->Value() << " "
            << "R=" << r->Value() << " " << "E=" << e->Value();

  // Is CP + IS + FUN = TRUE?
  CHECK_EQ(p->Value() + ... , ... +
    kBase * kBase * kBase * t->Value());
} else {
  LOG(INFO) << "Cannot solve problem.";
}  // if (solver.NextSolution())
```

We print out the found solution and check if it is valid:

```
if (solver.NextSolution()) {
  LOG(INFO) << "Solution found:";
  LOG(INFO) << "C=" << c->Value() << " " << "P=" << p->Value() << " "
            << "I=" << i->Value() << " " << "S=" << s->Value() << " "
            << "F=" << f->Value() << " " << "U=" << u->Value() << " "
            << "N=" << n->Value() << " " << "T=" << t->Value() << " "
            << "R=" << r->Value() << " " << "E=" << e->Value();

  // Is CP + IS + FUN = TRUE?
  CHECK_EQ(p->Value() + ... , ... +
    kBase * kBase * kBase * t->Value());
} else {
  LOG(INFO) << "Cannot solve problem.";
}  // if (solver.NextSolution())
```

The output is:

We print out the found solution and check if it is valid:

```
if (solver.NextSolution()) {
  LOG(INFO) << "Solution found:";
  LOG(INFO) << "C=" << c->Value() << " " << "P=" << p->Value() << " "
            << "I=" << i->Value() << " " << "S=" << s->Value() << " "
            << "F=" << f->Value() << " " << "U=" << u->Value() << " "
            << "N=" << n->Value() << " " << "T=" << t->Value() << " "
            << "R=" << r->Value() << " " << "E=" << e->Value();

  // Is CP + IS + FUN = TRUE?
  CHECK_EQ(p->Value() + ... , ... +
    kBase * kBase * kBase * t->Value());
} else {
  LOG(INFO) << "Cannot solve problem.";
}  // if (solver.NextSolution())
```

The output is:

```
$[23:51:34] examples/cp_is_fun1.cc:132: Solution found:
$[23:51:34] examples/cp_is_fun1.cc:133: C=2 P=3 I=7 S=4 F=9 U=6 N=8 T=1 R=0 E=5
```

To obtain all the solutions:

To obtain all the solutions:

```
while (solver.NextSolution()) {
  // Do something with the current solution
} else {
  // The search is finished
}
```

To obtain all the solutions:

```
while (solver.NextSolution()) {
  // Do something with the current solution
} else {
  // The search is finished
}
```

To finish the search:

To obtain all the solutions:

```
while (solver.NextSolution()) {
  // Do something with the current solution
} else {
  // The search is finished
}
```

To finish the search:

```
solver.EndSearch();
```

`SolutionCollector`:

`SolutionCollector`:

- Subclass of SearchMonitors.

`SolutionCollector`:

- Subclass of SearchMonitors.
- Different flavors:

`SolutionCollector`:

- Subclass of SearchMonitors.
- Different flavors:
  - `FirstSolutionCollector`.

`SolutionCollector`:

- Subclass of SearchMonitors.
- Different flavors:
  - `FirstSolutionCollector`.
  - `LastSolutionCollector`.

`SolutionCollector`:

- Subclass of SearchMonitors.
- Different flavors:
  - `FirstSolutionCollector`.
  - `LastSolutionCollector`.
  - `BestValueSolutionCollector`.

`SolutionCollector`:

- Subclass of SearchMonitors.
- Different flavors:
  - `FirstSolutionCollector`.
  - `LastSolutionCollector`.
  - `BestValueSolutionCollector`.
  - `AllSolutionCollector`.

`SolutionCollector`:

- Subclass of SearchMonitors.
- Different flavors:
  - `FirstSolutionCollector`.
  - `LastSolutionCollector`.
  - `BestValueSolutionCollector`.
  - `AllSolutionCollector`.
- Corresponding factory methods:

`SolutionCollector`:

- Subclass of SearchMonitors.
- Different flavors:
  - `FirstSolutionCollector`.
  - `LastSolutionCollector`.
  - `BestValueSolutionCollector`.
  - `AllSolutionCollector`.
- Corresponding factory methods:
  - `MakeFirstSolutionCollector()`.
  - `MakeLastSolutionCollector()`.
  - `MakeBestValueSolutionCollector()`.
  - `MakeAllSolutionCollector()`.

If you are only interested in global results:

If you are only interested in global results:

```
SolutionCollector* const all_solutions =
                                      solver.MakeAllSolutionCollector();
...
DecisionBuilder* const db = ...
...
solver.NewSearch(db, all_solutions);
while (solver.NextSolution()) {};
solver.EndSearch();

LOG(INFO)<< "Number of solutions:" << all_solutions->solution_count();
```

If you are only interested in global results:

```
SolutionCollector* const all_solutions =
                                        solver.MakeAllSolutionCollector();
...
DecisionBuilder* const db = ...
...
solver.NewSearch(db, all_solutions);
while (solver.NextSolution()) {};
solver.EndSearch();

LOG(INFO)<< "Number of solutions:" << all_solutions->solution_count();
```

Instead of using `NewSearch()`, `NextSolution()` repeatedly and `EndSearch()`, you can use the `Solve()` method:

If you are only interested in global results:

```
SolutionCollector* const all_solutions =
                                          solver.MakeAllSolutionCollector();
...
DecisionBuilder* const db = ...
...
solver.NewSearch(db, all_solutions);
while (solver.NextSolution()) {};
solver.EndSearch();

LOG(INFO)<< "Number of solutions:" << all_solutions->solution_count();
```

Instead of using `NewSearch()`, `NextSolution()` repeatedly and `EndSearch()`, you can use the `Solve()` method:

```
solver.Solve(db,all_solutions);
```

To effectively store some solutions in a `SolutionCollector`, you have to *declare* the variables you are interested in.

To effectively store some solutions in a `SolutionCollector`, you have to *declare* the variables you are interested in.

First, you create a `SolutionCollector`:

To effectively store some solutions in a `SolutionCollector`, you have to *declare* the variables you are interested in.

First, you create a `SolutionCollector`:

```
FirstSolutionCollector* const first_solution =
                               solver.MakeFirstSolutionCollector();
```

To effectively store some solutions in a `SolutionCollector`, you have to *declare* the variables you are interested in.

First, you create a `SolutionCollector`:

```
FirstSolutionCollector* const first_solution =
                              solver.MakeFirstSolutionCollector();
```

Then you *declare* the variable you are interested in:

To effectively store some solutions in a `SolutionCollector`, you have to *declare* the variables you are interested in.

First, you create a `SolutionCollector`:

```
FirstSolutionCollector* const first_solution =
                              solver.MakeFirstSolutionCollector();
```

Then you *declare* the variable you are interested in:

```
first_solution->Add(c);
```

To effectively store some solutions in a `SolutionCollector`, you have to *declare* the variables you are interested in.

First, you create a `SolutionCollector`:

```
FirstSolutionCollector* const first_solution =
                        solver.MakeFirstSolutionCollector();
```

Then you *declare* the variable you are interested in:

```
first_solution->Add(c);
```

Warning:

The method `Add()` simply adds the variable c to the `SolutionCollector`. The variable c is not tied to the solver, i.e. you will not be able to retrieve its value by `c->Value()` after a search with the method `Solve()`.

Launch the search:

Launch the search:

```
solver.Solve(db,first_solution);
```

Launch the search:

```
solver.Solve(db,first_solution);
```

After the search, you can retrieve the value of `c` as follows:

Launch the search:

```
solver.Solve(db,first_solution);
```

After the search, you can retrieve the value of `c` as follows:

```
first_solution->solution(0)->Value(c)
```

Launch the search:

```
solver.Solve(db,first_solution);
```

After the search, you can retrieve the value of `c` as follows:

```
first_solution->solution(0)->Value(c)
```

or as follows:

Launch the search:

```
solver.Solve(db,first_solution);
```

After the search, you can retrieve the value of `c` as follows:

```
first_solution->solution(0)->Value(c)
```

or as follows:

```
first_solution->Value(0,c)
```

Let's use the `AllSolutionCollector` to store and retrieve the values of the 72 solutions:

Let's use the `AllSolutionCollector` to store and retrieve the values of the 72 solutions:

```
SolutionCollector* const all_solutions =
                                      solver.MakeAllSolutionCollector();
//  Add the variables to the SolutionCollector
all_solutions->Add(letters);
...
DecisionBuilder* const db = ...
...
solver.Solve(db, all_solutions);

//  Retrieve the solutions
const int number_solutions = all_solutions->solution_count();
LOG(INFO) << "Number of solutions: " << number_solutions << std::endl;

for (int index = 0; index < number_solutions; ++index) {
  LOG(INFO) << "Solution found:";
  LOG(INFO) << "C=" << all_solutions->Value(index,c) << " "
            << "P=" << all_solutions->Value(index,p) << " "
            ...
            << "E=" << all_solutions->Value(index,e);
}
```

We are not limited to the variables of the model:

We are not limited to the variables of the model:

```
SolutionCollector* const all_solutions =
                                      solver.MakeAllSolutionCollector();
//  Add the interesting variables to the SolutionCollector
all_solutions->Add(c);
all_solutions->Add(p);
//  Create the variable kBase * c + p
IntVar* v1 = solver.MakeSum(solver.MakeProd(c,kBase), p)->Var();
//  Add it to the SolutionCollector
all_solutions->Add(v1);
...
DecisionBuilder* const db = ...
...
solver.Solve(db, all_solutions);

//  Retrieve the solutions
const int number_solutions = all_solutions->solution_count();
LOG(INFO) << "Number of solutions: " << number_solutions << std::endl;

for (int index = 0; index < number_solutions; ++index) {
  LOG(INFO) << "Solution found:";
  LOG(INFO) << "v1=" << all_solutions->Value(index,v1);
}
```

`Assignment` stores the solution (in parts or as a whole):

Assignment stores the solution (in parts or as a whole):

```
SolutionCollector* const all_solutions =
                                     solver.MakeAllSolutionCollector();
// Add the interesting variables to the SolutionCollector
IntVar* v1 = solver.MakeSum(solver.MakeProd(c,kBase), p)->Var();
// Add it to the SolutionCollector
all_solutions->Add(v1);
...
DecisionBuilder* const db = ...
...
solver.Solve(db, all_solutions);

// Retrieve the solutions
const int number_solutions = all_solutions->solution_count();
LOG(INFO) << "Number of solutions: " << number_solutions << std::endl;

for (int index = 0; index < number_solutions; ++index) {
  Assignment* const solution = all_solutions->solution(index);
  LOG(INFO) << "Solution found:";
  LOG(INFO) << "v1=" << solution->Value(v1);
}
```

Depending on the search,

Depending on the search, `Solve()` is equivalent to either:

or

Depending on the search, `Solve()` is equivalent to either:

```
solver.NewSearch();
solver.NextSolution();
solver.EndSearch();
```

or

Depending on the search, `Solve()` is equivalent to either:

```
solver.NewSearch();
solver.NextSolution();
solver.EndSearch();
```

or

```
solver.NewSearch();
while (solver.NextSolution()) {...};
solver.EndSearch();
```

The Google's flags library:

The Google's flags library:

- quite similar to other command flags libraries.

The Google's flags library:

- quite similar to other command flags libraries.
- flag definitions may be scattered in different files.

The Google's flags library:

- quite similar to other command flags libraries.
- flag definitions may be scattered in different files.

To define a flag, use the corresponding macro:

The Google's flags library:

- quite similar to other command flags libraries.
- flag definitions may be scattered in different files.

To define a flag, use the corresponding macro:

- `DEFINE_bool`.

The Google's flags library:

- quite similar to other command flags libraries.
- flag definitions may be scattered in different files.

To define a flag, use the corresponding macro:

- `DEFINE_bool`.
- `DEFINE_int32`.

The Google's flags library:

- quite similar to other command flags libraries.
- flag definitions may be scattered in different files.

To define a flag, use the corresponding macro:

- `DEFINE_bool`.
- `DEFINE_int32`.
- `DEFINE_int64`.

The Google's flags library:

- quite similar to other command flags libraries.
- flag definitions may be scattered in different files.

To define a flag, use the corresponding macro:

- `DEFINE_bool`.
- `DEFINE_int32`.
- `DEFINE_int64`.
- `DEFINE_uint64`.

The Google's flags library:

- quite similar to other command flags libraries.
- flag definitions may be scattered in different files.

To define a flag, use the corresponding macro:

- `DEFINE_bool`.
- `DEFINE_int32`.
- `DEFINE_int64`.
- `DEFINE_uint64`.
- `DEFINE_double`.

The Google's flags library:

- quite similar to other command flags libraries.
- flag definitions may be scattered in different files.

To define a flag, use the corresponding macro:

- `DEFINE_bool`.
- `DEFINE_int32`.
- `DEFINE_int64`.
- `DEFINE_uint64`.
- `DEFINE_double`.
- `DEFINE_string`.

To define a flag:

To define a flag:

```
...
#include "base/commandlineflags.h"
...
DEFINE_int64(base, 10, "Base used to solve the problem.");
...
namespace operations_research {
...
```

To define a flag:

```
...
#include "base/commandlineflags.h"
...
DEFINE_int64(base, 10, "Base used to solve the problem.");
...
namespace operations_research {
...
```

To parse the command line:

To define a flag:

```
...
#include "base/commandlineflags.h"
...
DEFINE_int64(base, 10, "Base used to solve the problem.");
...
namespace operations_research {
...
```

To parse the command line:

```
int main(int argc, char **argv) {
  google::ParseCommandLineFlags(&argc, &argv, true);
  operations_research::CPIsFun();
  return 0;
}
```

All defined flags are accessible as normal variables with the prefix `FLAGS_` prepended:

All defined flags are accessible as normal variables with the prefix `FLAGS_` prepended:

```
const int64 kBase = FLAGS_base;
```

All defined flags are accessible as normal variables with the prefix FLAGS_ prepended:

```
const int64 kBase = FLAGS_base;
```

To change the base with a command line argument:

All defined flags are accessible as normal variables with the prefix FLAGS_ prepended:

```
const int64 kBase = FLAGS_base;
```

To change the base with a command line argument:

```
./cp_is_fun4 --base=12
```

All defined flags are accessible as normal variables with the prefix FLAGS_ prepended:

```
const int64 kBase = FLAGS_base;
```

To change the base with a command line argument:

```
./cp_is_fun4 --base=12
```

If you want to know what the purpose of a flag is:

All defined flags are accessible as normal variables with the prefix `FLAGS_` prepended:

```
const int64 kBase = FLAGS_base;
```

To change the base with a command line argument:

```
./cp_is_fun4 --base=12
```

If you want to know what the purpose of a flag is:

- `--help`: prints all the flags.

All defined flags are accessible as normal variables with the prefix `FLAGS_` prepended:

```
const int64 kBase = FLAGS_base;
```

To change the base with a command line argument:

```
./cp_is_fun4 --base=12
```

If you want to know what the purpose of a flag is:

- `--help`: prints all the flags.
- `--helpshort`: prints all the flags defined in the same file as `main()`.

All defined flags are accessible as normal variables with the prefix `FLAGS_` prepended:

```
const int64 kBase = FLAGS_base;
```

To change the base with a command line argument:

```
./cp_is_fun4 --base=12
```

If you want to know what the purpose of a flag is:

- `--help`: prints all the flags.
- `--helpshort`: prints all the flags defined in the same file as `main()`.
- `--helpon=FILE`: prints all the flags defined in file `FILE`.

All defined flags are accessible as normal variables with the prefix `FLAGS_` prepended:

```
const int64 kBase = FLAGS_base;
```

To change the base with a command line argument:

```
./cp_is_fun4 --base=12
```

If you want to know what the purpose of a flag is:

- `--help`: prints all the flags.
- `--helpshort`: prints all the flags defined in the same file as `main()`.
- `--helpon=FILE`: prints all the flags defined in file `FILE`.
- `--helpmatch=S`: prints all the flags defined in the files `*S*.*`.

You can invoke the constructor of the `Solver` that takes a `SolverParameters` struct:

You can invoke the constructor of the Solver that takes a SolverParameters struct:

```
// Use some profiling and change the default parameters of the solver
SolverParameters solver_params = SolverParameters();
// Change the profile level
solver_params.profile_level = SolverParameters::NORMAL_PROFILING;

// Constraint programming engine
Solver solver("CP is fun!", solver_params);
```

You can invoke the constructor of the `Solver` that takes a `SolverParameters` struct:

```
// Use some profiling and change the default parameters of the solver
SolverParameters solver_params = SolverParameters();
// Change the profile level
solver_params.profile_level = SolverParameters::NORMAL_PROFILING;

// Constraint programming engine
Solver solver("CP is fun!", solver_params);
```

We can now ask for a detailed report after the search is done:

You can invoke the constructor of the `Solver` that takes a `SolverParameters` struct:

```
// Use some profiling and change the default parameters of the solver
SolverParameters solver_params = SolverParameters();
// Change the profile level
solver_params.profile_level = SolverParameters::NORMAL_PROFILING;

// Constraint programming engine
Solver solver("CP is fun!", solver_params);
```

We can now ask for a detailed report after the search is done:

```
// Save profile in file
solver.ExportProfilingOverview("profile.txt");
```

You can invoke the constructor of the `Solver` that takes a `SolverParameters` struct:

```
// Use some profiling and change the default parameters of the solver
SolverParameters solver_params = SolverParameters();
// Change the profile level
solver_params.profile_level = SolverParameters::NORMAL_PROFILING;

// Constraint programming engine
Solver solver("CP is fun!", solver_params);
```

We can now ask for a detailed report after the search is done:

```
// Save profile in file
solver.ExportProfilingOverview("profile.txt");
```

The `SolverParameters` struct mainly deals with the internal usage of memory and is for advanced users.

Suppose we want to limit the available time to solve a problem:

Suppose we want to limit the available time to solve a problem:

```
DEFINE_int64(time_limit, 10000, "Time limit in milliseconds");
```

Suppose we want to limit the available time to solve a problem:

```
DEFINE_int64(time_limit, 10000, "Time limit in milliseconds");
```

Register the `SearchMonitor`:

Suppose we want to limit the available time to solve a problem:

```
DEFINE_int64(time_limit, 10000, "Time limit in milliseconds");
```

Register the `SearchMonitor`:

```
SolutionCollector* const all_solutions =
                                    solver.MakeAllSolutionCollector();
...
// Add time limit
SearchLimit* const time_limit = solver.MakeTimeLimit(FLAGS_time_limit);
solver.Solve(db, all_solutions, time_limit);
```

Everything is coded in C++.

Everything is coded in C++. Thanks to SWIG, the or-tools library is also available in:

Everything is coded in C++. Thanks to SWIG, the or-tools library is also available in:

- Python.

Everything is coded in C++. Thanks to SWIG, the or-tools library is also available in:

- Python.
- Java.

Everything is coded in C++. Thanks to SWIG, the or-tools library is also available in:

- Python.
- Java.
- .NET (using Mono on non Windows platforms).

Everything is coded in C++. Thanks to SWIG, the or-tools library is also available in:

- Python.
- Java.
- .NET (using Mono on non Windows platforms).
- ...