OR-Tools

Laurent Perron, Google

Operations Research @ Google

- Operations Research team based in Paris
- Started ~6 years ago
- Currently, 10 people
- Mission:
  - Internal consulting: build and help build optimization applications
  - Tools: develop core optimization algorithms
- A few other software engineers with OR background distributed in the company

OR-Tools Overview

- https://code.google.com/p/or-tools/
- Open sourced under the Apache License 2.0
- C++, java, Python, and .NET interface
- Known to compile on Linux, Windows, Mac OS X
- Constraint programming + Local Search
- Wrappers around GLPK, CLP, CBC, SCIP, Sulum, Gurobi, CPLEX
- OR algorithms
- ~200 examples in Python and C++, 120 in C#, 40 in Java
- Interface to Minizinc/Flatzinc

OR-Tools: Constraint Programming

- Google Constraint programming:
  - Integer variables and constraints
  - Basic Scheduling support
  - Strong Routing Support.
  - No floats, no sets
- Design choices
  - Geared towards Local Search
  - No strong propagations (JC's AllDifferent)
  - Very powerful callback mechanism on search.
  - Custom propagation queue (AC5 like)

OR-Tools: Local Search

- Local search: iterative improvement method
  - Implemented on top of the constraint programming engine
  - Easy modeling
  - Easy feasibility checking for each move
- Large neighborhoods can be explored with constraint programming
- Local search
- Large neighborhood search
- Default randomized neighborhood
- Metaheuristics: simulated annealing, tabu search, guided local search

OR-Tools: Algorithms

- Bron-Kerbosch to find cliques in an undirected graph.
- Naive Dijkstra and Bellman-Ford algorithm to find shortest paths
- Find graph symmetry
- Min Cost Flow
- Max Flow
- Linear Sum Assignment
- And more to be implemented as needed

OR-Tools: Linear Solver Wrappers

- Unified API on top of CLP, CBC, GLPK, SCIP, Sulum, Gurobi, and CPLEX.
- Implemented in C++ with Python, java, and C# wrapping.
- Expose basic functionalities:
  - variables, constraints, reduced costs, dual values, activities...
  - Few parameters: tolerances, choice of algorithms, gaps

What is a CP Solver

A CP Solver has many aspects:

- Declaration of the model (variable, constraint, objectives
- Constraint propagation.
- Search declaration and execution

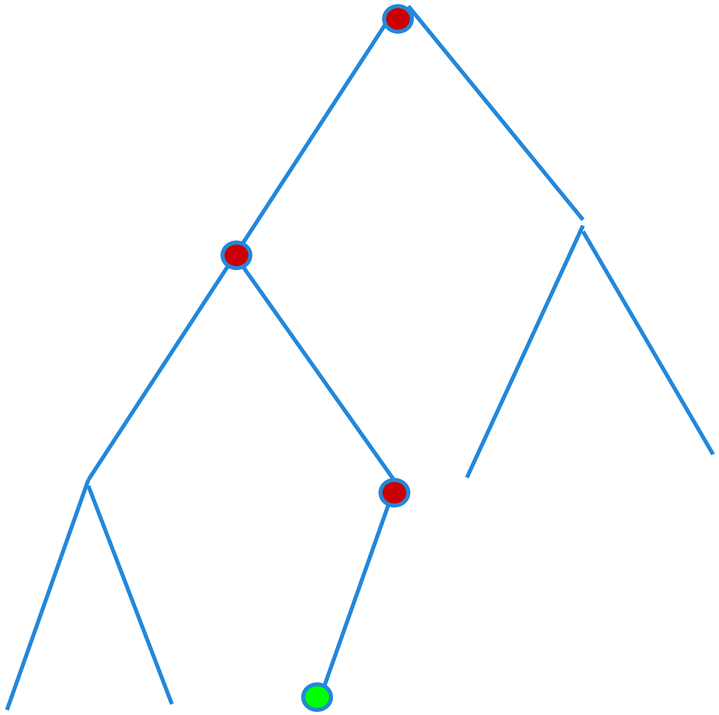
Search

CP is usually based on tree search (Branch and Bound in case of optimization).

Tree search need to be able to store/restore states to survive failure and be complete.

Tree can be described as a sequence of choice points.

CP Tree Search



Depth First Search

In case of depth first search, there is only a linear number of open nodes.

2 strategies to implement backtrack:

- Trail (save old value, old address) and restore upon backtrack
- Copy: copy on write and delete upon backtrack

Extending Tree Search

Using events on search (enter node, exit node, reach solution, failure), you can implement:

- Limits
- Objective
- Local Search
- Meta heuristics

Propagation

Theory says that domain reduction converges towards a fixed point (unique).

Domain reductions are performed in sequence.

Naive propagation queue

The domain of a variable is a set of possible values.

```
class Var {
  int[] Values();
  bool IntersectWith(int[] values);
}

class Constraint {
  bool Propagate();  // True if a modification occured.
}
```

Naive propagation queue - 2

You can write a simple queue

```
modified = False
do
  for (Constraint ct in constraints)
    modified |= ct.Propagate()
while (modified)
```

Implementing a queue a variable

The domain of a variable is a set of possible values.

```
class Var {
  int[] Values();
  bool IntersectWith(int[] values);
  void Register(Constraint ct);
  void Process();
}

class Constraint {
  void Post();  // Call register on all variables.
  bool Propagate();  // True if a modification occured.
}
```

Implementing a queue a variable - 2

Main idea:

have a queue of modified variables.

pop var

var.Process()

  call ct.Propagate() on all registered constraints

    Enqueue modified variables

repeat until queue is empty

Improving the queue of variable

Implement different events

- Variable domain is a singleton
- Variable bounds have changed
- Variable domain has changed

Fire relevant events in the process method.

Support propagation algorithms

Example:

x == y (equality constraint)

```
for v in x.domain()
  if not y.Contains(v)  // New API
    x.Remove(v)
for v in y.domains()
  if not x.Contains(v)
    y.Remove(v)
```

Improving equality

Pathological case 1: x [0, .., 10000], y[2, 4]

- Look at bounds first
- Requires Min(), Max(), SetMin(), SetMax()

Pathological case 1: x[0, .., 10000] -> x[0, 2, .., 10000]

- Knows that only 1 was removed
- Remove it from y

Weaker propagation:

- Only propagate bounds.

Dealing with value removal

Value removal is an important event

but:

- pathological cases: x[0..100000] -> x[1]
  - generate 99999 events?
- Processing one value at a time has a high overhead
  - Motivation to group
  - Complexify writing constraints

Extension: Reified equality

boolvar = (x == y) # boolvar is true if x = y, false otherwise.

Range based algorithm

if boolvar = true

Propagate like equality

else if boolvar = false

Propagate like inequality

else

Compare x and y bounds and propagate boolvar

Extension: Reified equality - 2

Support based version

Search one value that is common to x and y

Only search for a new one when this value is no longer valid

The Quest for the Perfect Sum

The standard algorithm

Sum(xi) = z

- Sum(min(xi)) <= min(z)
- Sum(max(xi)) >= max(z)

What can we deduce from the bounds of Z?

[0..1] + [0..1] + [0..1] = [0..1]   Nothing

[0..1] + [0..1] + [0..10] = [4..7] -> [2..7] for 3rd term

Back-Propagation of the Sum

- [0..1] + [0..1] + [0..10] = [4..7]
  - Sum of mins: 0 + 0 + 0 = 0
  - Sum of max: 1 + 1 + 10 = 12
- [0..10] = [4..7] - [0..1] - [0..1]
  - min(0..10) >= min(4..7) - max(0..1) - max(0..1)
  - min(0..10) >= min(4..7) - (sum of max) + max(0..10)
  - min(0..10) >= 4 - 12 + 10 = 2
  - thus [0..10] -> [2..10]
  - and [2..10] -> [2..7]
- By default, all 3 terms will be checked.

Complexity of the Sum

Linear between the xi and z, in both directions

How to improve it:

- propagate delta:
  - xi[a + da, b - db] -> sum [zmin+da, zmax-db]
- Divide and conquer on the array
  - tree based split, in nothing to deduce -> complexity based on the size of the block
  - but, propagation of delta in log(n) instead of constant time
  - Use diameter optimization, if xi greater than the slack between the sum of xi and the bound of the z, it can absorb any reduction

More work on the sum

If sum is a scalar product sum(ai.xi) = b.z

- We can add a gcd constraint
  - gcd(ai) divides b
- if z is constant,
  - gcd(ai|xi non bound) divides
    - b*z - ai.xi (xi bound)
- If z is not constant, move to left part and move constants to the right hand side.

Even More Work on the Sum

If sum(ai * bi) = z, ai > 0, bi boolean variables

Then sort ai increasingly,

Start from the end, if bi is unbound:

- if ai > zmax - sum min(xi), then bi = 0,
  - continue
- if ai > sum max(xi) - zmin, then bi = 1
- else stop

This is the perfect propagation

Complexity is linear down a branch

The Next Level

Can we achieve arc consistency in the sum?

i.e. :

{1, 5, 6} + [0..2] = {1, 2, 3, 5, 6, 7, 8}?

There are three options:

- Count the number of supports for each value of each variable.
- Use a table constraint (explicit representation of the graph of the constraint).
- Use bitset manipulation.

Looking at Search

Decision

```
class Decision:
    Apply(Solver)
    Refute(Solver)

  solver.AssignVariableValue(var, value)
  solver.VariableLessOrEqualValue(var, value)
  solver.VariableGreaterOrEqualValue(var,
value)
```

Decision Builder

class DecisionBuilder:

  Next(Solver) # Returns a decision, or None

solver.Phase() // Assign variables

solver.Compose() // Sequential composition

solver.Try() // Parallel (or) composition

solver.SolveOnce(db) // Find 1 solution in a sub-tree

Large Neighborhood Search

```
class PyLns:
    InitFragments(self)
   # returns a list of indices of the variables to relax
    NextFragment(self)
```

Look at pyls_api.py

LNS Example

```
class OneVarLns(pywrapcp.PyLns):
  def __init__(self, vars):
    pywrapcp.PyLns.__init__(self, vars)
    self.__index = 0

  def InitFragments(self):
    self.__index = 0

  def NextFragment(self):
    if self.__index < self.Size():
      self.__index += 1
      return [self.__index - 1]
    else:
      return []
```

LNS Example - 2

```
class SteelRandomLns(pywrapcp.PyLns):
  def __init__(self, x, rand, lns_size):
    pywrapcp.PyLns.__init__(self, x)
    self.__random = rand
    self.__lns_size = lns_size
  def InitFragments(self):
    pass
  def NextFragment(self):
    fragment = []
    while len(fragment) < self.__lns_size:
      pos = self.__random.randint(0, self.Size() - 1)
      fragment.append(pos)
    return fragment
```

Local Search

```
class MoveOneVarUp(pywrapcp.IntVarLocalSearchOperator):
  def __init__(self, vars):
    pywrapcp.IntVarLocalSearchOperator.__init__(self, vars)
    self.__index = 0
  def OneNeighbor(self):
    current_value = self.OldValue(self.__index)
    self.SetValue(self.__index, current_value + 1)
    self.__index = (self.__index + 1) % self.Size()
    return True
  def OnStart(self):
    pass
  def IsIncremental(self):
    return False
```

Filters

```
class pywrapcp.IntVarLocalSearchFilter:
  def OnSynchronize(self, delta):
  def Accept(self, delta, _):
  def IsIncremental(self):
    return False
```

Logical steps

Write a model with a naive search heuristics

then test model on small instances

then improve model/search heuristics

then implement randomized heuristics and combine with fast restarts

Logical steps

then use Large Neighborhood Search with random neighborhoods

then use Large Neighborhood Search with structured neighborhoods

then use Local Search

then add filters

Vertical extensions to CP

Scheduling

class IntervalVar

StartMin(), StartMax(), DurationMin(), DurationMax(), EndMin(), EndMax(), MayBePerformed(), MustBePerformed()

Constraints:

DisjunctiveConstraint (with optional transition times)

CumulativeConstraint

Convex Hull

Binary dependencies

Scheduling examples

Jobshop FT06

Simple meeting with alternative resources

Routing

Routing model

- visits (optional, disjunctive, penalties)
- vehicles (can have different cost structures)
- First Solution heuristics, tuned Local Search

Dimensions (values accumulated along the path)

- Transit, cumul, slack, total span
- Can use a TransitValue(from, to, vehicle) callback
- Can be used to add cost components
- Can be used to constrain the path

Routing examples

[TSP (in python)]

[Capacitated Vehicle Routing Problem With Time Windows (C#)]

[Pickup and Delivery Problem (C++)]