Image segmentation using SCIP

Advanced practical SS 2017

Robert Schütz, Daniela Kilian

August 27, 2017

Content

1. Problem definition
2. About SCIP
3. Implementation details
4. Possible improvements
5. Demo

Problem definition

Task

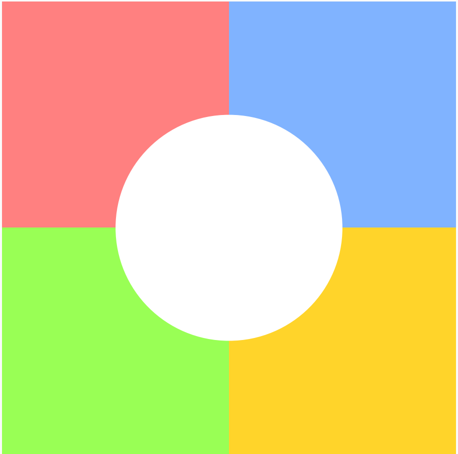


Figure 1: Input image

Task

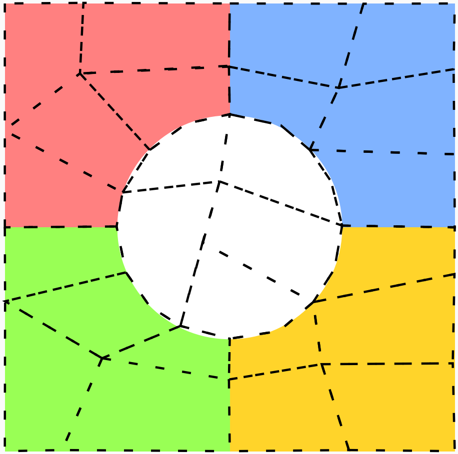


Figure 2: Generated superpixels

Task

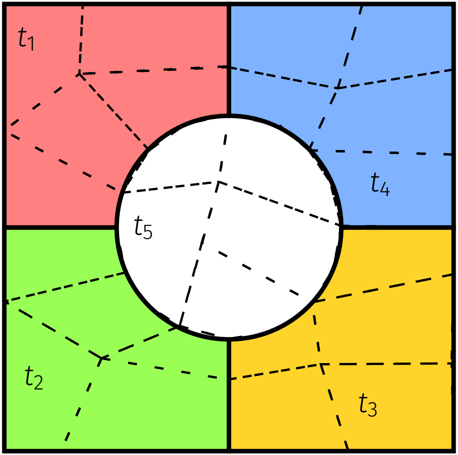


Figure 3: Master nodes and segments

Master problem

Given data in the theoretical formulation of the master problem:

- $\mathcal{S}$: set of superpixels
- $\mathcal{P} \subset 2^{\mathcal{S}}$: set of segments
- $y_s \geq 0$: color of superpixel $s \in \mathcal{S}$
- $T \subseteq \mathcal{S}$: set of master nodes
- $k = |T|$: number of segments to cover the image with
- $r_P = \sum_{s \in P} |y_t - y_s|$: error of segment $P \in \mathcal{P}$, where $t \in T$ is the single master node for which $t \in P$

Master problem

There is a binary variable x_P for every segment $P \in \mathcal{P}$.

The problem reads as follows:

$$\min \quad \sum_{P \in \mathcal{P}} r_P \cdot x_P \tag{1}$$

$$\text{s.t.} \quad \sum_{\{P \in \mathcal{P} : s \in P\}} x_P = 1 \quad \forall s \in \mathcal{S} \tag{2}$$

$$\sum_{P \in \mathcal{P}} x_P = k \tag{3}$$

$$x_P \in \{0, 1\} \quad \forall P \in \mathcal{P} \tag{4}$$

Dual problem

The dual variables are

- μ_s for all $s \in \mathcal{S}$: corresponding to (2)
- λ: corresponding to (3)

The dual problem reads as follows:

$$\max \quad \sum_{s \in \mathcal{S}} \mu_s + k \cdot \lambda \tag{5}$$

$$\text{s.t.} \quad \sum_{s \in P} \mu_s + \lambda \leq r_P \quad \forall P \in \mathcal{P} \tag{6}$$

$$\mu_s \text{ free} \quad \forall s \in \mathcal{S} \tag{7}$$

$$\lambda \text{ free} \tag{8}$$

Pricing problem

There is a pricing problem for each master node $t \in T$, which generates a segment containing t.

If $s \in \mathcal{S}$ is in the new segment, then $x_s = 1$.

$$\min \quad \underbrace{\sum_{s \in \mathcal{S}} x_s \cdot |y_t - y_s|}_{=r_{\{s \in \mathcal{S} \,:\, x_s = 1\}}} - \sum_{s \in \mathcal{S}} x_s \cdot \mu_s \tag{9}$$

$$\text{s.t.} \quad \text{connectivity constraint} \tag{10}$$

$$x_t = 1 \tag{11}$$

$$x_{t'} = 0 \quad \forall t' \in T \setminus \{t\} \tag{12}$$

$$x_s \in \{0, 1\} \quad \forall s \in \mathcal{S} \tag{13}$$

Pricing problem

Constraint (6) of the dual problem is violated by the new segment if and only if

$$\underbrace{\sum_{s \in \mathcal{S}} x_s \cdot |y_t - y_s|}_{= r_{\{s \in \mathcal{S} \,:\, x_s = 1\}}} - \sum_{s \in \mathcal{S}} x_s \cdot \mu_s < \lambda.$$

Therefore, a new segment has to satisfy this inequality in order to be added to the master problem.

Cutting planes

We look at the subgraph with vertices $\{s \in \mathcal{S} : x_s = 1\}$. If a component C is not connected to t, we add a cut for each $s \in C$:

$$\sum_{s' \in \delta(C)} x_{s'} \geq x_s$$

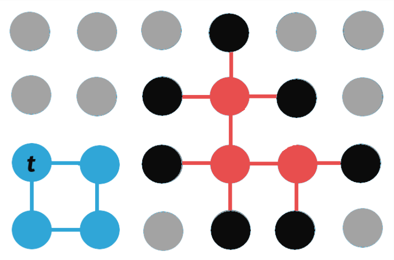


Figure 4: Violated connectivity constraint

About SCIP

What is SCIP?

From `scip.zib.de`:

> *"SCIP is currently one of the fastest non-commercial solvers for mixed integer programming (MIP) and mixed integer nonlinear programming (MINLP). It is also a framework for constraint integer programming and branch-cut-and-price."*

What is SCIP?

SCIP is developed at the Zuse Institute Berlin (ZIB). It is written in C, but can also be interfaced using

- C++
- Python
- Java

SCIP can use different LP solvers, e.g.

- SoPlex
- CPLEX

What is SCIP?

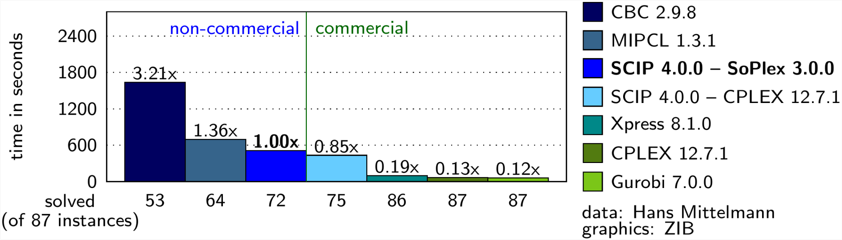


Figure 5: MIP solver benchmark

Advantages

SCIP is non-commercial and open source.

SCIP supports cutomizing

- Constraint handlers
- Separators
- Variable pricers
- Branching rules

These use callbacks, so there is no need for a custom control loop.

Also, SCIP can automatically do branching on variables declared as integral.

Implementation details

Programming Language

We are using C++. This makes it easy to interface with SCIP and add custom plugins. There are multiple SCIP examples written in C++, notably

- VRP, involving a custom pricer
- TSP, involving a custom constraint handler

We started off of these and implemented the desired functionality.

Reading the image

The image is first read from a PNG file using libpng and png++. Then, the image is segmented into superpixels using an algorithm called SLIC.

```
png::image<png::gray_pixel> pngimage(filename);
float* image = new float[pngimage.get_width() * pngimage.get_height()];
for (png::uint_32 x = 0; x < pngimage.get_width(); ++x)
{
  for (png::uint_32 y = 0; y < pngimage.get_height(); ++y)
  {
    image[x + y * pngimage.get_width()] = pngimage[y][x] / 255.0;
  }
}

segmentation = new vl_uint32[imagesize];
vl_slic_segment(segmentation, image, pngimage.get_width(), pngimage.get_height(), ...);
```

Creating the graph

To representing the graph of superpixels, we use the Boost Graph Library.

```
Graph g(superpixelcount);
for (auto p = vertices(g); p.first != p.second; ++p.first)
{
  g[*p.first].color = avgcolor[*p.first];
}
// add edges
for (png::uint_32 x = 0; x < width; ++x)
{
  for (png::uint_32 y = 0; y < height; ++y)
  {
    auto current = x + y * width;
    auto right = x + 1 + y * width;
    auto below = x + (y + 1) * width;
    if (x + 1 < width
      && segmentation[current] != segmentation[right])
    {
      // add edge to the superpixel on the right
      auto edge = add_edge(segmentation[current], segmentation[right], g);
      auto weight = boost::get(boost::edge_weight, g, edge.first);
      boost::put(boost::edge_weight, g, edge.first, weight + 1);
    }
    ...
  }
}
```

Master problem definition

First, we need to create a SCIP instance.

```
SCIP* scip;
SCIP_CALL(SCIPcreate(&scip));
SCIP_CALL(SCIPcreateProb(scip, "master_problem", ...));
SCIP_CALL(SCIPsetObjsense(scip, SCIP_OBJSENSE_MINIMIZE));
```

We start with initial segments which form a partitioning.

```
std::vector<SCIP_VAR*> vars;
for (auto segment : initial_segments)
{
  SCIP_VAR* var;
  // Set a very high objective value for the initial segments
  // so that they aren't selected in the final solution
  SCIP_CALL(SCIPcreateVar(scip, &var, "x_P", 0.0, 1.0, 10000, SCIP_VARTYPE_BINARY, ...));
  SCIP_CALL(SCIPaddVar(scip, var));
  vars.push_back(var);
}
```

Adding constraints

Finally, we add the partitioning constraints.

```
std::vector<SCIP_CONS*> partitioning_cons;
for (auto p = vertices(g); p.first != p.second; ++p.first)
{
  SCIP_CONS* cons1;
  SCIP_CALL(SCIPcreateConsLinear(scip, &cons1, "first", 0, NULL, NULL, 1.0, 1.0, ...));
  for (size_t i = 0; i != initial_segments.size(); ++i)
  {
    if (initial_segments[i].find(*p.first) != initial_segments[i].end())
    {
      SCIP_CALL(SCIPaddCoefLinear(scip, cons1, vars[i], 1.0));
    }
  }
  SCIP_CALL(SCIPaddCons(scip, cons1));
  partitioning_cons.push_back(cons1);
}

SCIP_CONS* num_segments_cons;
...
```

Pricer callback

```
SCIP_DECL_PRICERREDCOST(scip_redcost)
{
  SCIP_Real lambda = SCIPgetDualsolLinear(scip, num_segments_cons);
  for (size_t i = 0; i < master_nodes.size(); ++i)
  {
    auto p = heuristic(scip, master_nodes[i], lambda); // returns pair<redcost, superpixels>
    if (SCIPisDualfeasNegative(scip, p.first))
      SCIP_CALL(addPartitionVar(scip, master_nodes[i], p.second));
    else
    {
      for (auto s = vertices(g); s.first != s.second; ++s.first)
      {
        SCIP_Real mu_s = SCIPgetDualsolLinear(scip, partitioning_cons[*s.first]);
        SCIP_CALL(SCIPchgVarObj(scip_pricers[i], x[*s.first],
          -mu_s + std::abs(g[master_nodes[i]].color - g[*s.first].color)));
      }
      SCIP_CALL(SCIPsolve(scip_pricers[i]));
      SCIP_SOL* sol = SCIPgetBestSol(scip_pricers[i]);
      if (SCIPisDualfeasNegative(scip, SCIPgetSolOrigObj(scip_pricers[i], sol) - lambda))
      {
        SCIP_CALL(addPartitionVarFromPricerSCIP(scip, scip_pricers[i], sol, master_nodes[i]))
      }
    }
  }
  *result = SCIP_SUCCESS; // at least one improving variable was found,
                          // or it is ensured that no such variable exists
  return SCIP_OKAY;
}
```

Pricing heuristic

To make the solving process faster, a heuristic to find segments with negative reduced costs (rc) was added.

Algorithm 1 Simple pricing heuristic

```
P ← {t}
Let s ∈ δ(P) s.t. rc(P ∪ {s}) is minimal.
if rc(P) ≥ 0 then
    P ← P ∪ {s}
    goto 2
else if rc(P ∪ {s}) < rc(P) then
    P ← P ∪ {s}
    goto 2
else
    return P
end if
```

Constraint handler callback

First, we create the subgraph of all superpixels in the new component.

```
Graph& subgraph = g.create_subgraph();
std::vector<int> component(num_vertices(g));
for (auto p = vertices(g); p.first != p.second; ++p.first)
{
  if (SCIPisEQ(scip, SCIPgetSolVal(scip, sol, superpixel_vars[*p.first]), 1.0))
  {
    add_vertex(*p.first, subgraph);
  }
}
size_t num_components = connected_components(subgraph, &component[0]);
```

Constraint handler callback

```
for (int i = 0; i < num_components; i++)
{
  if (i == component[master_node])
    continue;

  // all superpixels in the component
  std::vector<Graph::vertex_descriptor> superpixels = ...;
  // all superpixels surrounding the component
  std::vector<Graph::vertex_descriptor> surrounding = ...;

  for (auto s : superpixels)
  {
    // add a constraint/row to the problem
    SCIP_ROW* row;
    SCIP_CALL(SCIPcreateEmptyRowCons(scip, &row, conshdlr, "sepa_con",
      0.0, SCIPinfinity(scip), ...));

    // sum_{all superpixels s surrounding the component} x_s >= ...
    for (auto s_ : surrounding)
      SCIP_CALL(SCIPaddVarToRow(scip, row, superpixel_vars[s_], 1.0));

    // ... >= x_s
    SCIP_CALL(SCIPaddVarToRow(scip, row, superpixel_vars[s], -1.0));

    SCIP_CALL(SCIPflushRowExtensions(scip, row));
    SCIP_CALL(SCIPaddCut(scip, sol, row, ...));
  }
}
```

Solve master problem

```
// include pricer
ObjPricerLinFit* pricer_ptr = new SegmentPricer(scip, g, ...);
SCIP_CALL(SCIPincludeObjPricer(scip, pricer_ptr, true));

// activate pricer
SCIP_CALL(SCIPactivatePricer(scip, SCIPfindPricer(scip, "pricer")));

// solve
SCIP_CALL(SCIPsolve(scip));
SCIP_SOL* sol = SCIPgetBestSol(scip);

// return selected segments
std::vector<std::vector<Graph::vertex_descriptor>> segments;
SCIP_VAR** variables = SCIPgetVars(scip);
for (int i = 0; i < SCIPgetNVars(scip); ++i)
{
  if (SCIPisEQ(scip, SCIPgetSolVal(scip, sol, variables[i]), 1.0))
  {
    auto vardata = (ObjVardataSegment*) SCIPgetObjVardata(scip, variables[i]);
    segments.push_back(vardata->getSuperpixels());
  }
}
```

Possible improvements

Branching rules

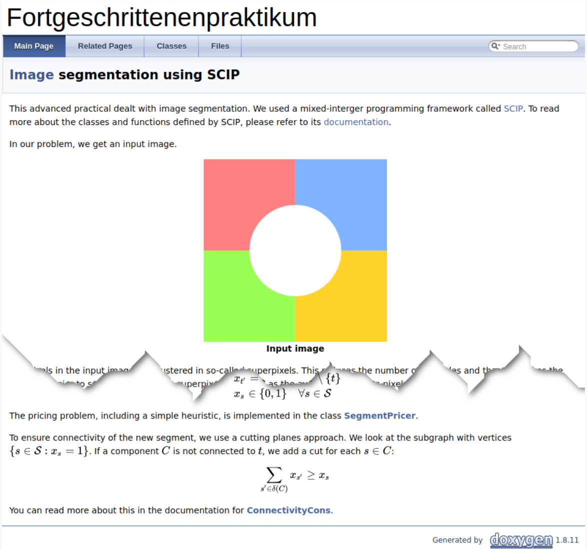
Ideally, one would develop branching rules specially for the master and pricing problem.

For example, branching on segments with minimal objective values could lead to good results.

This is out of scope for this practical.

Demo

Documentation



Fortgeschrittenenpraktikum

Main Page | Related Pages | Classes | Files | Search

Image segmentation using SCIP

This advanced practical dealt with image segmentation. We used a mixed-interger programming framework called SCIP. To read more about the classes and functions defined by SCIP, please refer to its documentation.

In our problem, we get an input image.

Input image

$$x_s \in \{0, 1\} \quad \forall s \in \mathcal{S}$$

The pricing problem, including a simple heuristic, is implemented in the class **SegmentPricer**.

To ensure connectivity of the new segment, we use a cutting planes approach. We look at the subgraph with vertices $\{s \in \mathcal{S} : x_s = 1\}$. If a component C is not connected to t, we add a cut for each $s \in C$:

$$\sum_{s' \in \delta(C)} x_{s'} \geq x_s$$

You can read more about this in the documentation for **ConnectivityCons**.

Generated by doxygen 1.8.11

Figure 6: Main Page

Documentation

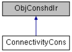
ConnectivityCons Class Reference

List of all members

Class for representing connectivity constraints This class uses cutting planes to make disconnected segments infeasible. More...

```
#include <connectivity_cons.h>
```

Inheritance diagram for ConnectivityCons:



Public Member Functions

	ConnectivityCons (SCIP *scip, Graph &g_, std::vector< Graph::vertex_descriptor > &master_nodes_, Graph::vertex_descriptor master_node_, std::vector< SCIP_VAR * > &superpixel_vars_) Constructor for the connectivity constraints class. More...
virtual	**SCIP_DECL_CONSTRANS** (scip_trans) Transforms constraint data into data belonging to the transformed problem.
	SCIP_DECL_CONSSEPALP (scip_sepalp) Separation method of constraint handler for LP solution.
	SCIP_DECL_CONSSEPASOL (scip_sepasol) Separation method of constraint handler for arbitrary primal solution.
virtual	**SCIP_DECL_CONSENFOLP** (scip_enfolp) Constraint enforcing method of constraint handler for LP solutions.
virtual	**SCIP_DECL_CONSENFOPS** (scip_enfops) Constraint enforcing method of constraint handler for pseudo solutions.
virtual	**SCIP_DECL_CONSCHECK** (scip_check) Feasibility check method of constraint handler for primal solutions.
virtual	**SCIP_DECL_CONSLOCK** (scip_lock) Variable rounding lock method of constraint handler.

Private Member Functions

size_t	**findComponents** (SCIP *scip, SCIP_SOL *sol, Graph &subgraph, std::vector< int > &component) Finds all connected components in a subgraph. More...
SCIP_RETCODE	**sepaConnectivity** (SCIP *scip, SCIP_CONSHDLR *conshdlr, SCIP_SOL *sol, SCIP_RESULT *result) Adds cutting plane, if possible. More...

Figure 7: Function overview

Documentation

Member Function Documentation

```
size_t ConnectivityCons::findComponents ( SCIP *                 scip,
                                          SCIP_SOL *             sol,
                                          Graph &                subgraph,
                                          std::vector< int > &   component
                                        )
```
private

Finds all connected components in a subgraph.

Returns

the number of connected components

Parameters

scip	pricer SCIP instance
sol	current primal solution or NULL
subgraph	subgraph with all superpixels for which $x_s = 1$
component	`component[s]` will be the index of the connected component the superpixel s belongs to

```
SCIP_RETCODE ConnectivityCons::sepaConnectivity ( SCIP *            scip,
                                                  SCIP_CONSHDLR *   conshdlr,
                                                  SCIP_SOL *        sol,
                                                  SCIP_RESULT *     result
                                                )
```
private

Adds cutting plane, if possible.

If the current solution is infeasible, a cutting plane of the following form is added for every superpixel s in a component C that is not connected to the master node t:

$$\sum_{s' \in \delta(C)} x_{s'} \geq x_s$$

Figure 8: Detailed description

Demo