DM841

Discrete Optimization

Lecture 6

Iterative Improvement Runners

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

Review

1. Combinatorial Optimization and Terminology
2. Solution Methods
3. SAT Example: enumeration, MIP, local search, backtracking
4. Local Search: Modelling and components
5. N-Queens example
6. C++: object passing, Encapsulation, Constructors, Inheritance, Templates, STL, virtual functions, headers, namespaces, copy constructors, destructors
7. EasyLocal framework. Examples

Standard Template Library

- Static arrays `array<type>`
- Dynamic arrays `vector<type>`
- lists (no random access) `list<type>`
- sets (no repetition of elements allowed) `set<type>` (implemented as red-black trees)
- maps `map<keyttype, type>` associative containers that contain key-value pairs with unique keys. Keys are sorted. (similar to dictionaries in python) (implemented as red-black trees)
- unordered versions of sets and maps
- They require to include the std library:

```
#include<cstdlib>
#include<vector>
#include<list>
#include<map>
#include<set>
#include<algorithm>
#include<stdexcept>
using namespace std;
```

Iterators

- iterators are pointers to elements of STL containers

```
vector<int> A = {1,2,3,4};
vector<int>::iterator pt; // or vector<int>::const_iterator
for (pt=A.begin(); pt!=A.end(); pt++)
   cout<<*pt;
```

- Type inference:

```
vector<int> A = {1,2,3,4};
vector<int>::iterator pt1 = A.begin();
aut pt2 = A.begin();
```

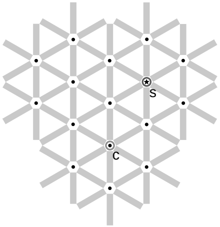
- for syntax:

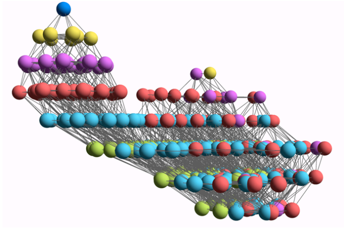
```
for (auto &x : my_array) {
    x *= 2;
}
```

Outline

1. Local Search

Local search — global view





- vertices: candidate solutions (search positions)
- vertex labels: evaluation function
- edges: connect "neighboring" positions
- s: (optimal) solution
- c: current search position

Summary: Local Search Algorithms
(as in [Hoos, Stützle, 2005])

For given problem instance π:

1. search space S_π
2. evaluation function $f_\pi : S \rightarrow \mathbf{R}$
3. neighborhood relation $\mathcal{N}_\pi \subseteq S_\pi \times S_\pi$
4. set of memory states M_π
5. initialization function $\texttt{init} : \emptyset \rightarrow S_\pi \times M_\pi)$
6. step function $\texttt{step} : S_\pi \times M_\pi \rightarrow S_\pi \times M_\pi$
7. termination predicate $\texttt{terminate} : S_\pi \times M_\pi \rightarrow \{\top, \bot\}$

Iterative Improvement

Iterative Improvement (II):
determine initial candidate solution s
while s has better neighbors **do**
  choose a neighbor s' of s such that $f(s') < f(s)$
  $s := s'$

- If more than one neighbor have better cost then need to choose one (heuristic pivot rule)
- The procedure ends in a local optimum $\hat{s}$:
  Def.: Local optimum $\hat{s}$ w.r.t. N if $f(\hat{s}) \leq f(s)\ \forall s \in N(\hat{s})$
- Issue: how to avoid getting trapped in bad local optima?
  - use more complex neighborhood functions
  - restart
  - allow non-improving moves

Decision vs Minimization

LS-Decision(π)
input: problem instance $\pi \in \Pi$
output: solution $s \in S'(\pi)$ **or** $\emptyset$

```
(s, m) := init(π)

while not terminate(π, s, m) do
    (s, m) := step(π, s, m)

if s ∈ S'(π) then
    return s
else
    return ∅
```

LS-Minimization(π')
input: problem instance $\pi' \in \Pi'$
output: solution $s \in S'(\pi')$ **or** $\emptyset$

```
(s, m) := init(π');
s_b := s;
while not terminate(π', s, m) do
    (s, m) := step(π', s, m);
    if f(π', s) < f(π', ŝ) then
        s_b := s;

if s_b ∈ S'(π') then
    return s_b
else
    return ∅
```

Examples

Iterative Improvement for SAT

- **search space** S**:** set of all truth assignments to variables in given formula F (**solution set** S'**:** set of all models of F)
- **neighborhood relation** $\mathcal{N}$**:** 1-flip neighborhood
- **memory:** not used, *i.e.*, $M := \{0\}$
- **initialization:** uniform random choice from S, *i.e.*, $\texttt{init}(\emptyset, \{a\}) := 1/|S|$ for all assignments a
- **evaluation function:** $f(a) :=$ number of clauses in F that are *unsatisfied* under assignment a (*Note:* $f(a) = 0$ iff a is a model of F.)
- **step function**: uniform random choice from improving neighbors, *i.e.*, $\texttt{step}(a, a') := 1/|I(a)|$ if $a' \in I(a)$, and 0 otherwise, where $I(a) := \{a' \mid \mathcal{N}(a, a') \wedge f(a') < f(a)\}$
- **termination**: when no improving neighbor is available *i.e.*, $\texttt{terminate}(a, \top) := 1$ if $I(a) = \emptyset$, and 0 otherwise.

Examples

Random order first improvement for SAT

URW-for-SAT(F,*maxSteps*)
input: *propositional formula* F, *integer maxSteps*
output: *a model for* F **or** $\emptyset$

choose assignment φ of truth values to all variables in F
  uniformly at random;
steps := 0;
while $\neg$(φ satisfies F) and (*steps* $<$ *maxSteps*) **do**
  select x uniformly at random from $\{x' | x'$ is a variable in F and
  changing value of x' in φ decreases the number of unsatisfied clauses$\}$
  steps := *steps*$+1$;
if φ satisfies F **then**
  return φ
else
  return $\emptyset$

Local Search Modelling
Iterative Improvement

queensLS00.co

```
import cotls;
int n = 16;
range Size = 1..n;
UniformDistribution distr(Size);

Solver<LS> m();
var{int} queen[Size](m,Size) := distr.get();
ConstraintSystem<LS> S(m);

S.post(alldifferent(queen));
S.post(alldifferent(all(i in Size) queen[i] + i));
S.post(alldifferent(all(i in Size) queen[i] - i));
m.close();

int it = 0;
while (S.violations() > 0 && it < 50 * n) {
  select(q in Size, v in Size : S.getAssignDelta(queen[q],v) < 0) {
    queen[q] := v;
    cout<<"chng @ "<<it<<": queen["<<q<<"]:="<<v<<" viol: "<<S.violations() <<endl;
  }
  it = it + 1;
}
cout << queen << endl;
```

Local Search Modelling

Best Improvement

queensLS0.co

```
import cotls;
int n = 16;
range Size = 1..n;
UniformDistribution distr(Size);

Solver<LS> m();
var{int} queen[Size](m,Size) := distr.get();
ConstraintSystem<LS> S(m);

S.post(alldifferent(queen));
S.post(alldifferent(all(i in Size) queen[i] + i));
S.post(alldifferent(all(i in Size) queen[i] - i));
m.close();

int it = 0;
while (S.violations() > 0 && it < 50 * n) {
  selectMin(q in Size,v in Size)(S.getAssignDelta(queen[q],v)) {
    queen[q] := v;
    cout<<"chng @ "<<it<<": queen["<<q<<"] := "<<v<<" viol: "<<S.violations() <<
          endl;
  }
  it = it + 1;
}
cout << queen << endl;
```

Local Search Modelling

First Improvement

queensLS2.co

```
import cotls;
int n = 16;
range Size = 1..n;
UniformDistribution distr(Size);

Solver<LS> m();
var{int} queen[Size](m,Size) := distr.get();
ConstraintSystem<LS> S(m);

S.post(alldifferent(queen));
S.post(alldifferent(all(i in Size) queen[i] + i));
S.post(alldifferent(all(i in Size) queen[i] - i));
m.close();

int it = 0;
while (S.violations() > 0 && it < 50 * n) {
  selectFirst(q in Size, v in Size: S.getAssignDelta(queen[q],v) < 0) {
    queen[q] := v;
    cout<<"chng @ "<<it<<": queen["<<q<<"] := "<<v<<" viol: "<<S.violations() <<
          endl;
  }
  it = it + 1;
}
cout << queen << endl;
```

Local Search Modelling

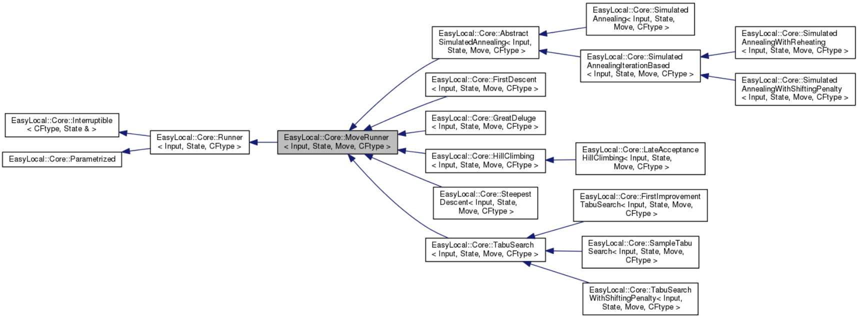
Min Conflict Heuristic

queensLS0b.co

```
import cotls;
int n = 16;
range Size = 1..n;
UniformDistribution distr(Size);

Solver<LS> m();
var{int} queen[Size](m,Size) := distr.get();
ConstraintSystem<LS> S(m);

S.post(alldifferent(queen));
S.post(alldifferent(all(i in Size) queen[i] + i));
S.post(alldifferent(all(i in Size) queen[i] − i));
m.close();

int it = 0;
while (S.violations() > 0 && it < 50 * n) {
  select(q in Size : S.violations(queen[q])>0) {
    selectMin(v in Size)(S.getAssignDelta(queen[q],v)) {
      queen[q] := v;
      cout<<"chng @ "<<it<<": queen["<<q<<"] := "<<v<<" viol: "<<S.violations() <<
            endl;
    }
    it = it + 1;
  }
}
cout << queen << endl;
```

EasyLocal: Runners

- Runner classes are the algorithmic core of the framework.
- They are responsible for performing a run of a local search technique, starting from an initial state and leading to a final one.
- `Runner` has only `Input` and `State` templates, and is connected to the solvers
- `MoveRunner` has also `Move`, and the pointers to the necessary helpers. It also stores the basic data common to all derived classes: the current state, the best state, the current move, and the number of iterations.


EasyLocal::Core::Interruptible < CFtype, State & >
EasyLocal::Core::Parametrized
EasyLocal::Core::Runner < Input, State, CFtype >
EasyLocal::Core::MoveRunner < Input, State, Move, CFtype >
EasyLocal::Core::Abstract SimulatedAnnealing< Input, State, Move, CFtype >
EasyLocal::Core::Simulated Annealing< Input, State, Move, CFtype >
EasyLocal::Core::Simulated AnnealingIterationBased < Input, State, Move, CFtype >
EasyLocal::Core::Simulated AnnealingWithReheating < Input, State, Move, CFtype >
EasyLocal::Core::Simulated AnnealingWithShiftingPenalty < Input, State, Move, CFtype >
EasyLocal::Core::FirstDescent < Input, State, Move, CFtype >
EasyLocal::Core::GreatDeluge < Input, State, Move, CFtype >
EasyLocal::Core::HillClimbing < Input, State, Move, CFtype >
EasyLocal::Core::LateAcceptance HillClimbing< Input, State, Move, CFtype >
EasyLocal::Core::Steepest Descent< Input, State, Move, CFtype >
EasyLocal::Core::FirstImprovement TabuSearch< Input, State, Move, CFtype >
EasyLocal::Core::TabuSearch < Input, State, Move, CFtype >
EasyLocal::Core::SampleTabu Search< Input, State, Move, CFtype >
EasyLocal::Core::TabuSearch WithShiftingPenalty< Input, State, Move, CFtype >

Runners::Go

```
template <class Input, class State, typename CFtype>
CFtype Runner<Input,State,CFtype>::Go(State& s) throw (ParameterNotSet, IncorrectParameterValue)
{
  InitializeRun(s);
  while (!MaxIterationExpired() && !StopCriterion() && !LowerBoundReached() && !this->TimeoutExpired())
  {
    PrepareIteration();
    try
    {
      SelectMove();
    }
    catch (EmptyNeighborhood e)
    {
      break;
    }
    if (AcceptableMove())
    {
      PrepareMove();
      MakeMove();
      CompleteMove();
      UpdateBestState();
    }
    CompleteIteration();
  }

  return TerminateRun(s);
}
```

Runners.hh

Hill Climbing in EasyLocal

A move is accepted if it is non worsening (i.e., it improves the cost or leaves it unchanged).

```
template <class Input, class State, class Move, typename CFtype>
HillClimbing<Input,State,Move,CFtype>::HillClimbing(const Input& in,
StateManager<Input,State,CFtype>& e_sm,
NeighborhoodExplorer<Input,State,Move,CFtype>& e_ne,
std::string name)
  : MoveRunner<Input,State,Move,CFtype>(in, e_sm, e_ne, name, "Hill Climbing Runner"),
// parameters
max_idle_iterations("max_idle_iterations", "Total number of allowed idle iterations", this->parameters)
{
}

template <class Input, class State, class Move, typename CFtype>
void HillClimbing<Input,State,Move,CFtype>::SelectMove()
{
  this->SelectRandomMove();
}


template <class Input, class State, class Move, typename CFtype>
bool HillClimbing<Input,State,Move,CFtype>::MaxIdleIterationExpired() const
{
  return this->iteration - this->iteration_of_best >= this->max_idle_iterations;
}

template <class Input, class State, class Move, typename CFtype>
bool HillClimbing<Input,State,Move,CFtype>::MaxIterationExpired() const
{
  return this->iteration >= this->max_iterations;
}

template <class Input, class State, class Move, typename CFtype>
bool HillClimbing<Input,State,Move,CFtype>::StopCriterion()
{
  return MaxIdleIterationExpired() || this->MaxIterationExpired();
}

template <class Input, class State, class Move, typename CFtype>
bool HillClimbing<Input,State,Move,CFtype>::AcceptableMove()
{
  return LessOrEqualThan(this->current_move_cost,(CFtype)0);
```

- The `FirstDescent` (aka First Improvement) runner performs a simple local search. At each step of the search, the first improving move in the neighborhood of current solution is selected and performed.
- The `SteepestDescent` (aka Best Improvement) runner performs a simple local search. At each step of the search, the best move in the neighborhood of current solution is selected and performed.
- The `HillClimbing` runner considers random move selection. A move is then performed only if it does improve or it leaves unchanged the value of the cost function.
- The `LateAcceptanceHillClimbing` maintains a list of previous moves and defers acceptance to k steps further.

Interruptible

An inheritable class to add timeouts (in milliseconds) to anything.

`MakeFunction` produces a function object to be launched in a separate thread by SyncRun, AsyncRun or Tester

Public Member Functions

	Interruptible ()
Rtype	**SyncRun** (std::chrono::milliseconds timeout, Args...args)
std::shared_future< Rtype >	**AsyncRun** (std::chrono::milliseconds timeout, Args...args)
void	**Interrupt** ()

Protected Member Functions

const std::atomic< bool > &	**TimeoutExpired** ()
virtual std::function< Rtype(Args &...)>	**MakeFunction** ()
virtual void	**AtTimeoutExpired** ()

Parametrized

An inheritable class representing a parametrized component.

Public Member Functions

	Parametrized (const std::string &prefix, const std::string &description)
virtual void	**ReadParameters** (std::istream &is=std::cin, std::ostream &os=std::cout)
virtual void	**Print** (std::ostream &os=std::cout) const
template<typename T > T	**GetParameter** (std::string flag)
template<typename T > void	**SetParameter** (std::string flag, const T &value)

Protected Attributes

ParameterBox	**parameters**

In constructors, eg, `AbstractLocalSearch`

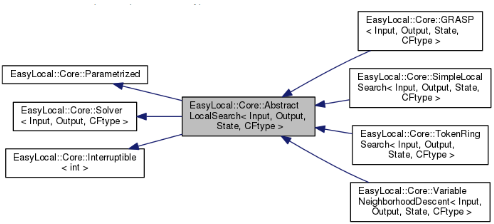
Observers

Infrastructure for printing debugging information on the runner
The command line parameter decides how much verbose the output must be:

- `--main::observer 1` for all runners with the observer attached, it writes some info on the costs everytime the runner finds a new best state.
- `--main::observer 2` it writes also all times that the runners makes a worsening move
- `--main::observer 3`, it write all moves executed by the runner.

EasyLocal: Solvers

Solver classes control the search by generating the initial solutions, and deciding how, and in which sequence, Runners and Kickers have to be activated



Solve


```
    template <class Input, class Output, class State, typename CFtype>
    typename AbstractLocalSearch<Input,Output,State,CFtype>::SolverResult AbstractLocalSearch<Input,Output,State,CFtype>::Solve() throw
(ParameterNotSet, IncorrectParameterValue)
    {
      auto start = std::chrono::high_resolution_clock::now();
      InitializeSolve();
      FindInitialState();
      if (timeout.IsSet())
      {
        SyncRun(std::chrono::milliseconds(static_cast<long long int>(timeout * 1000.0)));
      }
      else
        Go();
      p_out = std::make_shared<Output>(this->in);
      om.OutputState(*p_best_state, *p_out);
      TerminateSolve();

      double run_time = std::chrono::duration_cast<std::chrono::duration<double, std::ratio<1>>>(std::chrono::high_resolution_clock::now() -
start).count();

      return std::make_tuple(*p_out, sm.Violations(*p_best_state), sm.Objective(*p_best_state), run_time);
    }
```

```
template <class Input, class Output, class State, typename CFtype>
void SimpleLocalSearch<Input,Output,State,CFtype>::SetRunner(Runner<Input,State,CFtype>& r)
  { this->p_runner = &r; }


template <class Input, class Output, class State, typename CFtype>
void SimpleLocalSearch<Input,Output,State,CFtype>::Go()
{
  if (!p_runner)
    // FIXME: add a more specific exception behavior
    throw std::logic_error("Runner not set in object " + this->name);
  this->current_state_cost = p_runner->Go(*this->p_current_state);

  *this->p_best_state = *this->p_current_state;
  this->best_state_cost = this->current_state_cost;
}
```