Configuration Checking: A Local Search Technique

Shaowei Cai

University of Chinese Academy of Sciences (UCAS)

2016

The Cycling Problem of Local Search

Local search algorithms suffer from the cycling problem, i.e., revisit to a candidate solution that has been visited recently.

- This cycling problem wastes a local search much time and prevents it from getting out of local minima.
- The cycling problem is an inherent problem of local search as the method does not allow to memorize all previously visited parts of the search space.
- Impractical to incorporate local search with an additional mechanism to remember all previously visited candidate solutions, which requires exponential space.

Previous Methods

- Naive methods
  - Random walk
  - Non-improving search
  - Restart
- The tabu mechanism forbids reversing the recent changes, where the strength of forbidding is controlled by a parameter called tabu tenure [proposed by Glover, 1989].

Configuration Checking

A new strategy to deal with the cycling problem in local search: Configuration Checking (CC).

- The idea
  - A local search procedure maintains a current candidate solution C.
  - For a solution component, when considering changing its state (or value), check whether its configuration has changed. If not, it is forbidden to change.
- New
  - Previous heuristics usually refer to the information of a solution component but neglect its circumstance.
  - CC takes the circumstance of solution components into account.

Configuration Checking

Configuration Checking (CC) is generic for combinatorial search problems. In particular, CC is suitable for the following types of problems:

- Subset Problems: to find a subset from a universe set such that satisfies the constraints (and optimized).
- Assignment Problems: to find an assignment to all variables such that satisfies the constraints (and optimized).

Configuration Checking

Configuration Checking (CC) is generic for combinatorial search problems. In particular, CC is suitable for the following types of problems:

- Subset Problems: to find a subset from a universe set such that satisfies the constraints (and optimized).
- Assignment Problems: to find an assignment to all variables such that satisfies the constraints (and optimized).

CC has been successfully applied to many problems:

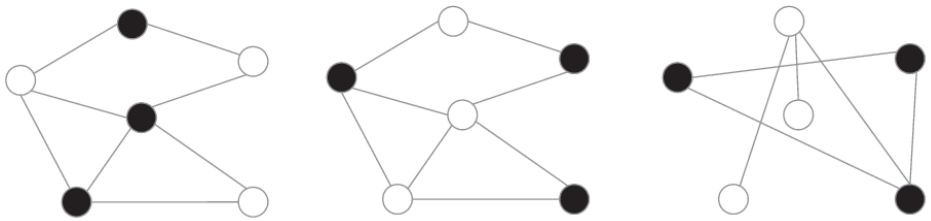
- Vertex Cover (AIJ 2011, AAAI 2012, JAIR 2013)
- Clique (AAAI 2016)
- SAT (AAAI 2012, AIJ 2013, ISAIM 2014, JAIR 2014, IEEE T. Cybernetics 2015)
- MaxSAT (IEEE T. Computers 2015, J. Heuristics 2015)
- Set Cover (Science China 2015, EJOR 2015)
- Golomb Rulers (CPAIOR 2015)

Vertex Cover

Definition

Given an undirected graph $G = (V, E)$, a vertex cover is a subset $S \subseteq V$ such that every edge in G has at least one endpoint in S.

Vertex Cover is closely related to Independent Set and Clique.



The MVC Problem

Definition

The Minimum Vertex Cover (MVC) problem is to find the minimum sized vertex cover in a graph.

The decision version: given k, to find a k-sized vertex cover in a graph.

Remark

Minimum Vertex Cover (MVC) is equivalent to Maximum Clique (MC) and Maximum Independent Set (MIS) problems.

Local Search for MVC

A framework of local search algorithms for MVC solve the problem by solving its decision problem iteratively

- whenever finding a k-sized vertex cover, the algorithm goes on to search for a $(k-1)$-sized vertex cover.
- the core of this framework is a local search for k-sized vertex cover, where k is fixed.

Local Search for MVC

A framework of local search algorithms for MVC solve the problem by solving its decision problem iteratively

- whenever finding a k-sized vertex cover, the algorithm goes on to search for a $(k-1)$-sized vertex cover.
- the core of this framework is a local search for k-sized vertex cover, where k is fixed.

Local search for a fixed size vertex cover

- View the solution space as a set of points connected to each other.
- There is a cost function which needs to be minimized that can be computed for each point, i.e. number of uncovered edges.
- Local search involves starting at some point in the solution space, and moving to adjacent points in an attempt to lower the cost function.

Local Search for MVC

A step to a neighboring candidate solution consists of exchanging two vertices: a vertex $u \in C$ is removed from C, a vertex $v \notin C$ is put into C, where C is the current candidate solution.

The red vertices are selected for cover, and the red edges indicate that they are not covered by the current solution.

CC for MVC

- CC for MVC
  - A local search procedure maintains a current candidate solution C.
  - When selecting a vertex for adding into C, check whether its configuration has changed since its last removal from C. If not, it is forbidden to add back to C.

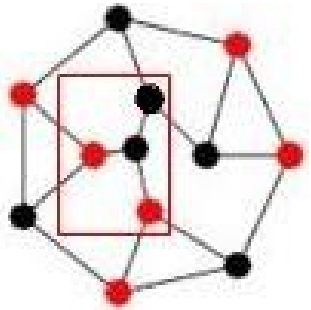
Configuration

Definition

Given an undirected graph $G = (V, E)$ and C the current candidate solution (for covering), the **state** of a vertex v is $s_v \in \{1, 0\}$, where $s_v = 1$ means $v \in C$, and $s_v = 0$ means $v \notin C$.

Definition

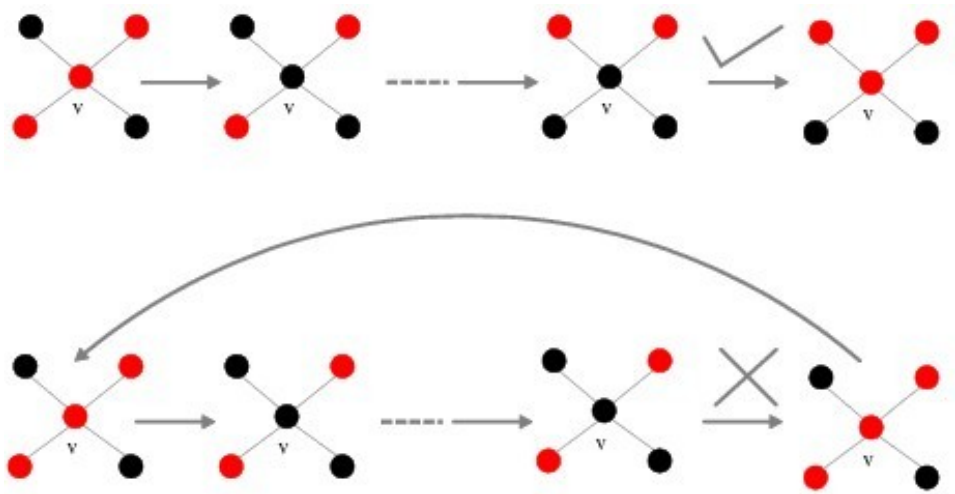
Given an undirected graph $G = (V, E)$ and C the current candidate solution, **configuration** for v is a vector C_v, which consists of state of all its neighboring vertices.



Configuration Checking

Definition

The Configuration Checking strategy: When selecting a vertex to add into the current candidate solution C, for a vertex $v \notin C$, if the configuration for v never changes since v's last removing from C, then it is forbidden to be added back to C.



reduce local structure cycles!

An Implementation of Configuration Checking

In order to implement the configuration checking strategy in MVC local search algorithms, we employ an array *confChange*, whose element is an indicator.

- $confChange[v] = 1$ means the configuration for v has changed since v's last leaving C;
- $confChange[v] = 0$ on the contrary.

During the search procedure, the vertex with $confChange[v] = 0$ *are forbidden to be added to the current candidate solution.*

An Implementation of Configuration Checking

In order to implement the configuration checking strategy in MVC local search algorithms, we employ an array *confChange*, whose element is an indicator.

- $confChange[v] = 1$ means the configuration for v has changed since v's last leaving C;
- $confChange[v] = 0$ on the contrary.

During the search procedure, the vertex with $confChange[v] = 0$ *are forbidden to be added to the current candidate solution.*

We maintain the *confChange* array as follows:

- In the beginning, all $confChange[v]$ are initialized to 1.
- When removing v from C, $confChange[v]$ is reset to 0.
- When u changes its *state*, for each $v \in N(u)$, $confChange[v]$ is set to 1.

Apply CC to Local Search for VC

In each exchanging step

- choose a vertex $u \in C$ and a vertex $v \notin C$ with confChange[v]=1;
- $C := [C \setminus u] \cup v$;
- update confChange values of u, v and their neighboring vertices;

The SAT Problem

- A set of boolean variables: $X = \{x_1, x_2, ..., x_n\}$.
- Literals: $x_1, \neg x_1, x_2, \neg x_2, ..., x_n, \neg x_n$
- A set of Clauses: $x_1 \vee \neg x_2, x_2 \vee x_3, x_2 \vee \neg x_4, \neg x_1 \vee \neg x_3 \vee x_4, ...$
- A Conjunctive Normal Form (CNF) formula:

$$\varphi = (x_1 \vee \neg x_2) \wedge (x_2 \vee x_3) \wedge (x_2 \vee \neg x_4) \wedge (\neg x_1 \vee \neg x_3 \vee x_4)$$

- The satisfiability problem (SAT): test whether there exists an assignment of truth values to the variables in F under which F evaluates to true.
- The maximum satisfiability problem (MAXSAT): find an assignment that satisfies the most clauses in F.

Local Search for SAT

- Local search starts at some point in the solution space, and moves to adjacent points in an attempt to lower the cost function, i.e. number of unsatisfied clauses.

Algorithm 1: Local Search Framework for SAT

```
begin
    s ← a randomly generated truth assignment;
    while not reach terminal condition do
        if s satisfies F then return s ;
        pick a variable x;
        s := s with x flipped;
    return "Solution not found";
end
```

Notation in Local Search for SAT

- Given an assignment α, a clause is *satisfied* if it has at least one true literal, and *unsatisfied* if all the literals in the clause are false literals.
- We use $cost(F, \alpha)$ to denote the number (or total weight) of all unsatisfied clauses under an assignment s.
- For a variable x, let $score(x) = cost(F, \alpha) - cost(F, \alpha')$, measuring the benefit of flipping x, where α' is obtained from α by flipping the value of x.
- Two variables are neighbors iff they occur in at least one clause.
- Let $N(x) = \{y | y \in Var(F)$ and y occurs in at least one clause with $x\}$, which is the set of all *neighboring variables* of variable x.

Configuration Checking for SAT

Definition

The *configuration* of a variable x is a vector C_x consisting of truth value of all x's neighboring variables.

Configuration Checking for SAT: For a variable x, if its configuration has not changed since the last time x was flipped, then x is forbidden to be flipped.

An Accurate Implementation of CC

An accurate implementation of CC

- Store the configuration (i.e., truth values of all its neighbors) for a variable x when it is flipped
- Check the configuration when considering flipping a variable

An Accurate Implementation of CC

An accurate implementation of CC

- Store the configuration (i.e., truth values of all its neighbors) for a variable x when it is flipped
- Check the configuration when considering flipping a variable

For a formula F, let $\Delta(F) = max\{\#N(x) : x \in V(F)\}$.
It needs $O(\Delta(F))$ for both storing and checking the configuration for a variable.
Thus, the worst case complexity of CC in each step is

$$O(\Delta(F)) + O(\Delta(F)n) = O(\Delta(F)n)$$

An Approximate Implementation of CC

We employ an array *confChange*. During the search procedure, the variables with $confChange[x] = 0$ are forbidden to be flipped.

- $confChange[x] = 1$ means the configuration of x has been changed since x's last flip;
- $confChange[x] = 0$ on the contrary.

An Approximate Implementation of CC

We employ an array *confChange*. During the search procedure, the variables with $confChange[x] = 0$ are forbidden to be flipped.

- $confChange[x] = 1$ means the configuration of x has been changed since x's last flip;
- $confChange[x] = 0$ on the contrary.

Maintain the *confChange* array

- Rule 1: In the beginning, for each variable x, $confChange[x]$ is initialized as 1.
- Rule 2: When flipping x, $confChange[x]$ is reset to 0, and for each $y \in N(x)$, $confChange[y]$ is set to 1.

An Approximate Implementation of CC

Complexity of the approximate implementation

- $O(1)$ for checking whether a variable is configuration changed (check whether confChange[x]=1).
- update confChange values for $N[x]$.

Thus, the worst case complexity of CC in each step is

$$O(1 \cdot n) + O(\Delta(F)) = O(n)$$

Indeed, the number of candidate variables for flipping is much less than n.

Size of A Variable's Neighborhood

The effectiveness of CC is closely related to the neighborhood of variables. For a random k-SAT formula $F_k(n, m)$, we calculated the size of each variable's neighborhood in F as follows.

We fix an arbitrary variable x. For any other variable y, $y \notin N(x)$ happens if and only if there does not exist such a clause that x and y both appear in.
For any clause, the probability that x and y both appear in the clause is $p = \binom{n-2}{k-2} / \binom{n}{k} = \frac{k(k-1)}{n(n-1)}$
The probability that $y \in N(x)$ is
$Pr(y \in N(x)) = 1 - Pr(y \notin N(x)) = 1 - (1-p)^m$
Let I_y be the indicator variable for the event $\{y \in N(x)\}$, then $\mathbb{E}(I_y) = Pr(y \in N(x)) = 1 - (1-p)^m$.

Size of A Variable's Neighborhood

The expectation of the size of $N(x)$ can be obtained as following

$$\mathbb{E}(\#N(x)) = \mathbb{E}(\sum_{y \in V(F) \setminus \{x\}} I_y) \quad (1)$$

$$= \sum_{y \in V(F) \setminus \{x\}} \mathbb{E}(I_y) \quad (2)$$

$$= (n-1)(1-(1-p)^m) \quad (3)$$

$$\approx (n-1)(1-(\frac{1}{e})^{pm}) \quad (4)$$

$$= (n-1)\left(1-(\frac{1}{e})^{\frac{k(k-1)}{n(n-1)}m}\right) \quad (5)$$

$$= (n-1)\left(1-(\frac{1}{e})^{\frac{k(k-1)r}{(n-1)}}\right) (let \quad r = m/n) \quad (6)$$

Size of A Variable's Neighborhood

Using the limit $\lim_{N\to+\infty}(1-e^{-\frac{c}{N}}) = \frac{c}{N}$ (where c is a constant), and let $N = n-1$, $c = k(k-1)r$, we have

$$\lim_{n\to\infty} \mathbb{E}(\#N(x)) = (n-1)\left(1-(\frac{1}{e})^{\frac{k(k-1)r}{(n-1)}}\right) \quad (7)$$

$$= (n-1)\frac{k(k-1)r}{n-1} \quad (8)$$

$$= k(k-1)r \quad (9)$$

When CC Becomes Ineffective

Now, by using the inequality $(1-\frac{1}{n})^n < \frac{1}{e}$ $(n>1)$, we have

$$\mathbb{E}(\#N(x)) = (n-1)(1-(1-p)^m) \quad (10)$$

$$> (n-1)(1-(\frac{1}{e})^{pm}) \quad (11)$$

$$= (n-1)\left(1-(\frac{1}{e})^{\frac{k(k-1)m}{n(n-1)}}\right) \quad (12)$$

$$= (n-1) - \frac{n-1}{e^{\frac{k(k-1)r}{(n-1)}}} \quad (13)$$

When $n-1 < e^{\frac{k(k-1)r}{(n-1)}}$, or equivalently, $ln(n-1) < \frac{k(k-1)r}{(n-1)}$, we have $\mathbb{E}(\#N(x)) > (n-1)-1 = n-2$. In this case, the expected size of the neighborhood of a variable is greater than $n-2$. This indicates that most variables have all other variables as their neighbourhood. In this case, the CC strategy almost degrades to the tabu method with tabu tenure 1.

When CC Becomes Ineffective

Let $f(n) = ln(n-1) - \frac{k(k-1)r}{n-1}$, it is clear that $f(n)$ is a monotonic increasing function of n $(n > 1)$. Thus, $f(n) < 0$ occurs if and only if $n \leq \lfloor n^* \rfloor$, where n^* is a real number such that $f(n^*) = 0$. We have calculated the value of such n^* for random k-SAT formulas $(k = 3, 4, 5, 6, 7)$ near the phase transition.

Formulas	3-SAT ($r = 4.2$)	4-SAT ($r = 9.0$)	5-SAT ($r = 20$)	6-SAT ($r = 40$)	7-SAT ($r = 85$)
n^*	11.652	32.348	90.093	223.095	564.595

Table: The n^* value such that when $n < n^*$, the size of any variable's neighborhood is bigger than $n - 2$, and the CC strategy degrades to the tabu method with tabu tenure 1.

More analyses?

More questions to be answered

- Why CC can be useful?
- More general subclasses where CC is ineffective?
- Subclasses and algorithms where CC is effective?

Variants of CC for SAT

The typical CC strategy for SAT is based on the Configuration which is defined as the vector of the truth value of all x's neighboring variables. This CC strategy is called NVCC (Neighboring Variables based CC).

We can have variants of CC strategy by defining configuration in different ways.

Variants of CC for SAT

The typical CC strategy for SAT is based on the Configuration which is defined as the vector of the truth value of all x's neighboring variables. This CC strategy is called NVCC (Neighboring Variables based CC).

We can have variants of CC strategy by defining configuration in different ways.

Here is another CC strategy called Clause States based CC (CSCC), based on a new definition of configuration.

Definition

The configuration of a variable x is a vector that consists of the states of all the clauses in which x appears.

Combination of CC strategies

We can also combine different CC strategy in one algorithm.
A successful example is the Double Configuration Checking (DCC) for SAT, which is based on the following observation:

Lemma

If a variable is configuration changed w.r.t. CSCC criterion, then it is is configuration changed w.r.t. NVCC, while the reverse is not true.

In DCC heuristic, the algorithm first selects a configuration changed variable is w.r.t. CSCC criterion; if no such variable, it then selects a configuration changed variable is w.r.t. NVCC criterion.

CC with Aspiration (CCA)

Aspiration mechanism: if a variable has a large score, then it can override the CC criterion.

A local search for SAT based on CCA heuristic:

Algorithm 2: a variable selection heuristic based on CCA

begin
  if $\exists$ *CC variables with positive score* **then**
    return a CC variable with the greatest *score*;
  if $\exists$ *significant variables i.e.* $score(x) > sigscore$ **then**
    return a significant variable with the greatest *score*;
  return a variable in a random unsatisfied clause;
end

Success of CC for SAT

- 40% of the solvers participating in the random track of SAT Competition 2013 use CC strategies.
- 6 medal-awarded solvers use CC strategy: 2 Gold medals, 2 Silver medals and 2 Bronze medals.
- winners in the incomplete track of recent MaxSAT Evaluations use CC strategies.
- AAAI 2013 Tutorial Forum: "It (CCASat) is a brand new solver. It is outstanding. It is likely changing the game."