Local search in CSPs

Jonas Lehtonen

April 1, 2009

CSP introduction
- What is the CSP and SAT?
- CSP to SAT encoding

Local search introduction
- What is local search?
- Simple algorithm
- Advantages and disadvantages

Advanced local search algorithms for SAT
- Random to the rescue
- Case study: WalkSAT
- Other methods

Epilogue
- Other local search problems
- Existing libraries and tools
- Conclusion

What is the CSP?

- A CSP instance is a triple P = {V, D, C} where $V = \{x_1, x_2, ..., x_n\}$ is a finite set of n variables, D is a function that maps each variable x_i to the set D_i of possible values it can take. (D_i is called the *domain* of x_i), and $C = \{C_1, C_2, ..., C_n\}$ is a finite set of *constraints*.
- Each *constraint* is a function that maps a tuple of variable domains to a value in {satisfied, unsatisfied}.
- An assignment of variables to a CSP is a mapping that assigns to each variable x_i a value $v_i$from its domain D_i. An assignment is a solution to the CSP iff each constraint evaluates to 'satisfied' when each variable x_i in its ordered set of variables is given the corresponding value v_i.

What is a SAT problem?

- The propositional SATisfiability problem (SAT) is a special case of the CSP. Each SAT instance is equivalent to a propositional formula F in conjunctive normal form, i.e. $F = \bigwedge_{i=1}^{m} c_i$ where $c_i = \bigvee_{j=1}^{k(i)} l_{ij}$. The l_{ij} are called *literals*, and their disjunctions c_i form the *clauses*.
- The domain D_i of each variable x_i is $\{\text{true}, \text{false}\}$.
- The constraints are equivalent to the clauses c_i of the above formula, that is, when the logical clause is given the corresponding assignment of variable values it is true exactly when the constraint is satisfied.

SAT example

- The formula $(\neg a \vee \neg b) \wedge (\neg a \vee c) \wedge (\neg a \vee \neg c) \wedge (d)$ is equivalent to a SAT problem where the constraints are equivalent to the clauses in parentheses. That is, there are four constraints.
- For example, the clause $(\neg a \vee \neg b)$ is equivalent to the constraint $c : \{0, 1\}^2 \rightarrow \{unsatisfied, satisfied\}$ that returns 'satisfied' when at least one of its arguments is 0.
- With the assignment {a=true, b=true, c=true, d=true} the problem's constraints are, respectively, unsatisfied, satisfied, unsatisfied, and satisfied. Hence the SAT problem is not solved.
- With the assignment {a=false, b=true, c=true, d=true} all the constraints are satisfied, and hence the SAT problem is solved.

What's SAT good for?

- SAT is NP-complete and simple to formulate, and hence of considerable interest to theoretical computing.
- The applications in the 'real world' are numerous: Hardware design and verification, robotics, scheduling, database systems, artificial intelligence...
- Local search is particularly suitable for SAT which has led to the development of good algorithms for it. These have then been translated into general CSP algorithms, so the general CSP problem has benefited significantly from research on it.
- It also plays a role in solving some more general CSPs, by encoding the CSP in SAT form.

CSP to SAT encoding

- CSP problems can sometimes be encoded as SAT problems, applying a constraint- and domain-dependent algorithm that takes the CSP as input and produces another CSP, namely a SAT problem.
- This is done in a way that is highly constraint-dependent. Encoding a+b = 8 when a and b have domains {2,3,4} is very different from encoding a*b = 8
- The reason this is done is because a lot of effort has been put into the development of SAT algorithms, so existing fast and reliable implementations are available.

CSP to SAT encoding, problems

- SAT encodings often grow in size quickly when the size of the variable domains of the CSP increase. This can cause intractably large SAT problems.
- The encoding tends to obfuscate the problem, rendering invisible useful structural information about the problem.
- It is often a better idea simply to use the existing ideas from SAT algorithms and incorporate them into CSP solvers. Because of the similarities between the problem types they are often applicable.

CSP introduction
- What is the CSP and SAT?
- CSP to SAT encoding

Local search introduction
- What is local search?
- Simple algorithm
- Advantages and disadvantages

Advanced local search algorithms for SAT
- Random to the rescue
- Case study: WalkSAT
- Other methods

Epilogue
- Other local search problems
- Existing libraries and tools
- Conclusion

What is local search for the CSP?

- Local search is a method for finding a solution to a CSP instance.
- It's based on the idea of starting with a complete assignment of variables, usually randomlly generated, and working step by step toward a solution by changing the variable assignments.
- The steps usually involve changing one variable at a time, often using some heuristic such as the number of unsatisfied constraints.

Local search definition

Given a combinatorial problem instance π, a stochastic (non-deterministic) local search algorithm is defined by the following components:

- The *search space* $S(\pi)$, a finite set of search states $s \in \pi$. In the case of CSPs, these are complete variable assignments. A set of *acceptable solutions* $S(\pi)' \subseteq S(\pi)$ is required, and the *neighbourhood relation* $N(\pi) \subseteq S(\pi) \times S(\pi)$ that gives the positions that can be reached from one search state to the other. Finally, the *memory states* $M(\pi)$ for holding information beyond the search state are needed. These can consist of a single state, or e.g. hold information on past search states.
- The initialisation function $init(\pi) : \emptyset \rightarrow D(S(\pi) \times M(\pi))$ determining the state and memory state at the beginning of the algorithm. D is a probability distribution, that is, a function that maps each of its arguments to their respective probabilities of happening.
- A step function $step(\pi) : S(\pi) \times M(\pi) \rightarrow D(S(\pi) \times M(\pi))$ determining the state and memory state to go to at each step.
- A termination predicate $terminate(\pi) : S(\pi) \times M(\pi) \rightarrow D(true, false)$ determining whether to end the search at each step.

Example local search algorithm - GSAT

- GSAT takes a SAT problem instance as an argument and returns either an assignment of variables that satisfies the problem instance or a failure notification.
- As a simple local search algorithm, it starts from a random assignment of variables and flips them (from true to false or vice versa) one at a time according to certain rules. Parameters govern both the number of times a new random assignment happens and how many steps are taken after each assignment before giving up.
- The heuristic used by GSAT is the score of each variable, that is, the number of clauses that will be unsatisfied after flipping it. Efficient implementations calculate this incrementally after the initial calculations at each try.

```
GSAT(F, maxTries, maxSteps):
  for try = 1 to maxTries
    a = randomly chosen assignment of variables in the formula F
    for step := 1 to maxSteps
      if a satisfies F then return a
      x := variable to be flipped in F in order to minimise
      the number of unsatisfied clauses. If multiple such
      variables exist, choose one at random.
      a := a with x flipped
  return "No solution found"
```

GSAT example

- Consider the SAT problem equivalent to the formula $(\neg a \vee \neg b) \wedge (\neg a \vee c) \wedge (\neg a \vee \neg c) \wedge (d)$
- Further consider a starting assignment {a=true, b=true, c=true, d=true}
- Now, flipping a would cause there to be 0 unsatisfied clauses. Similarly, for b there would be 1 unsatisfied clause, for c there would be 2 unsatisfied clauses, and for d there would be 3 unsatisfied clauses.
- a is flipped. We see that the assignment satisfies the algorithm, and GSAT ends in success, returning {a=false, b=true, c=true, d=true}

Local search problems

- Due to the incremental looking for solutions, local search algorithms can get stuck in local minima of their heuristics.
- For example, GSAT can get stuck if it reaches a local minimum of unsatisfied constraints. In that case, it could flip between this state and the one with next-fewest unsatisfied constraints until maxSteps runs out.
- They are often incomplete, i.e. not guaranteed to reach a solution even if one exists. For example, GSAT with any fixed maxTries is provably incomplete.

So why local search?

- Performance is one driving factor behind local search. Local search algorithms usually scale better with problem size than complete systematic search algorithms.
- Local search has a number of high-level concepts that have shown to be efficient across a large number of problems, so problem-specific knowledge is not so critical as it might be with other types of algorithms.

CSP introduction
- What is the CSP and SAT?
- CSP to SAT encoding

Local search introduction
- What is local search?
- Simple algorithm
- Advantages and disadvantages

Advanced local search algorithms for SAT
- Random to the rescue
- Case study: WalkSAT
- Other methods

Epilogue
- Other local search problems
- Existing libraries and tools
- Conclusion

Random selection in local search

- Local search problems often use a random set of variable values at the beginning of the search algorithm.
- To avoid the search stagnating at a local minimum of the used heuristic, some algorithms go further and make use of randomness in the local search step.
- Instead of selecting the variable which (say) will cause the lowest number of unsatisfied constraints, a random variable is selected from a set of variables.
- This can be e.g. the set of variables appearing in currently unsatisfied clauses, i.e. a conflict-directed random step.

GWSAT

- GWSAT is a simple algorithm that extends the GSAT algorithm by occasionally doing a conflict-directed random step. This happens with a parameter-set probability wp.
- The effectiveness of the algorithm depends on the parameter wp, but when suitably chosen it can be much more efficient than GSAT.
- For sufficiently high wp GWSAT does not suffer from stagnation, and indeed has been proven to be *probabilistically approximately complete* (PAC) for any wp > 0. This means that as runtime for the algorithm approaches infinity the probability of reaching a solution becomes 1.

GSAT, as seen before

```
GSAT(F, maxTries, maxSteps):
  for try = 1 to maxTries
    a = randomly chosen assignment of variables in the formula F
    for step := 1 to maxSteps
      if a satisfies F then return a
      x := variable to be flipped in F in order to minimise
      the number of unsatisfied clauses. If multiple such
      variables exist, choose one at random.
      a := a with x flipped
  return "No solution found"
```

GWSAT

```
GWSAT(F, maxTries, maxSteps, wp):
  for try = 1 to maxTries
    a = randomly chosen assignment of variables in the formula F
    for step := 1 to maxSteps
      if a satisfies F then return a
      with probability 1 - wp
        x := variable to be flipped in F in order to minimise
        the number of unsatisfied clauses. If multiple such
        variables exist, choose one at random.
      else
        x := a random variable from those
        variables currently involved in a conflict
      a := a with x flipped
  return "No solution found"
```

WalkSAT

- WalkSAT, like GWSAT, is an algorithm for solving SAT problem instances
- Like GWSAT it can also be tuned by modifying a single variable: The probability that it will take a random step in one part of the search process.
- Unlike GWSAT, WalkSAT contains a few extra optimizations. First, when it can take a step that does not cause any currently satisfied clauses to become unsatisfied, it takes it without a chance of doing a conflict-directed random step. Secondly, it targets for flipping those variables that are involved in a large number of unsatisfied clauses.

```
GSAT(F, maxTries, maxSteps):
  for try = 1 to maxTries
    a = randomly chosen assignment of variables in the formula F
    for step := 1 to maxSteps
      if a satisfies F then return a
      x := variable to be flipped in F in order to minimise
      the number of unsatisfied clauses. If multiple such
      variables exist, choose one at random.
      a := a with x flipped
  return "No solution found"
```

```
score(F, v)
   return the amount of currently satisfied
   clauses in F that will be broken by flipping v
WalkSAT(F, maxTries, maxSteps, wp):
   for try = 1 to maxTries
      a = randomly chosen assignment of variables in the formula F
      for step := 1 to maxSteps
         if a satisfies F then return a
         c = randomly selected clause in the formula F
         if there are variables v in F with score(v) = 0
            x := randomly selected variable among these
         else
            with probability 1 - wp
               x := variable in F with minimal score,
               chosen at random if multiple exist
            else
               x := random variable in c
         a := a with x flipped
   return "No solution found"
```

WalkSAT, cont.

- WalkSAT often performs significantly better than GWSAT when instance-specific optimizations to the wp parameter are applied.
- While it has the PAC property for 2-SAT (i.e. a SAT problem instance where all clauses have two literals) it's not known whether it's PAC in the general case.
- It's easier to implement even if the pseudocode is slightly more complex – due to handling only one clause at a time it doesn't have to do incremental score function updating like GWSAT.

Tabu

- Tabu search is based on the idea that being stuck at a local minimum in local search often means cycling through a small set of states.
- It solves this by marking some recent states or other aspects of the problem 'tabu', that is, making it impossible to go back to them for a set amount of steps.
- These algorithms typically take a tabu tenure parameter that says for how many steps a state is declared tabu, or to put it another way, how many states can be tabu at once.

Tabu, cont.

- WalkSAT/Tabu is a simple extension of the WalkSAT algorithm that adds a tabu tenure for flipped parameters. If a variable has been flipped recently, it can't be flipped again for tt steps. It typically performs better than vanilla WalkSAT.
- The Novelty algorithm tracks the age of **all** flipped variables, that is, the time since a variable has been flipped. After choosing the clause as in vanilla WalkSAT, it selects the variable with the lowest score in the clause. If that variable has the lowest age among the variables in the clause, then with probability wp it instead selects the variable with the next lowest score. In ties, the oldest variables are preferred.
- Novelty occasionally suffers greatly from stagnation, but usually outperforms both vanilla WalkSAT and WalkSAT/Tabu significantly.

Dynamic weighting

- By adding weights to some constraints and making the heuristic the total weight of (un)satisfied constraints the solver can be guided either toward satisfying them first or toward satisfying other constraints first.
- GSAT with clause weights is a simple algorithm that starts with all clause weights at 1. At the end of each run, it adds 1 to the clause weights of the clauses that were unsatisfied, meaning that on the next run it will treat satisfying them as more important than before. Performance is better than vanilla GSAT in some problem types, but WalkSAT derivatives like Novelty tend to outperform it.
- There are numerous algorithms using clause weights and related tricks. Guided Local Search, GENET, ESG, SAPS, Discrete Lagrangian Method are a few things to Google.

CSP introduction
- What is the CSP and SAT?
- CSP to SAT encoding

Local search introduction
- What is local search?
- Simple algorithm
- Advantages and disadvantages

Advanced local search algorithms for SAT
- Random to the rescue
- Case study: WalkSAT
- Other methods

Epilogue
- Other local search problems
- Existing libraries and tools
- Conclusion

Constraint Optimisation Problems

- Often in real-world problems you can't expect all the constraints to get satisfied.
- Constraint Optimisation Problems (CPOs) are similar to CSPs, but their objective is to maximize the number of satisfied constraints or some other heuristic that is incremental in nature.
- In real-world applications weights are often placed on the different constraints.
- For example, the problem of resource management with multiple jobs that need to be done in a constrained time.

Local search tools

- On the big business side, ILOG bears a mention. Part of IBM, it sells and supports the ILOG Solver which is a general-purpose constraint programming engine as well the Scheduler and Dispatcher for manufacturing (among other things) and transportation respectively.
- On the free software side, EasyLocal++ (whose site is down at the time of writing but the relevant paper is available) and HotFrame are object-oriented frameworks for constructing local search algorithms. No fancy GUI, requires C++ knowledge.

Summary

- Local search algorithms are useful for quickly solving both large CSPs and more specifically SATs.
- Incompleteness and stagnation are problems with the entire algorithm class, but there are a number of ways to alleviate both problems.
- A massive amount of time and effort has been put into creating good algorithms, and even the efficient ones are typically quite easy to implement.

Thank yous

- Our friendly neighbourhood professor for being understanding about scheduling and offering advice.
- Holger H. Hoos and Edward Tsang for a well-written introduction to local search in CSPs.
- Me.