Branch and Bound Algorithms to solve Semiring Constraint Satisfaction Problems

Louise Leenen[1,2] and Aditya Ghose[1]

[1] Decision Systems Laboratory, SCSSE, University of Wollongong, Australia
`ll916,aditya@uow.edu.au`
[2] CSIR, South Africa
`lleenen@csir.co.za`

Abstract. The Semiring Constraint Satisfaction Problem (SCSP) framework is a popular approach for the representation of partial constraint satisfaction problems. Considerable research has been done in solving SCSPs, but limited work has been done in building general SCSP solvers. This paper is part of a series in which incremental changes are made to a branch and bound (BnB) algorithm for solving SCSPs. We present two variants of a BnB algorithm: a backjumping algorithm and a forward checking algorithm. These algorithms are based on the maximal constraints algorithms of Freuder and Wallace [1], and we show they perform better than the BnB algorithm on some problem instances.

1 Introduction and Background

Semiring Constraint Satisfaction is a general framework for constraint satisfaction where classical CSPs, Fuzzy CSPs, Partial CSPs, and others (over finite domains) can be cast [2]. Existing work on SCSP methods include [3–7]. Previously we presented a BnB algorithm to solve SCSPs [8] which is described in Section 2. Section 3 contains the main contribution of this paper: two new algorithms to solve SCSPs: a backjumping and a forward checking algorithm. In Section 4 we test the algorithms. An overview of the SCSP framework follows.

Definition 1. *A c-semiring is a tuple* $S = \langle A,+,\times,\mathbf{0},\mathbf{1}\rangle$ *such that*

- *A is a set with $\mathbf{0},\mathbf{1} \in A$;*
- *$+$ is defined over (possibly infinite) sets of elements of A as follows* [3]*:*
  - *for all $a \in A$, $\sum(\{a\}) = a$;*
  - *$\sum(\emptyset) = \mathbf{0}$ and $\sum(A) = \mathbf{1}$;*
  - *$\sum(\bigcup A_i, i \in I) = \sum(\{\sum(A_i), i \in I\})$ for all sets of indices I;*
- *$\times$ is a commutative, associative, and binary operation such that $\mathbf{1}$ is its unit element and $\mathbf{0}$ is its absorbing element;*
- *for any $a \in A$ and $B \subseteq A$, $a \times \sum(B) = \sum(\{a \times b, b \in B\})$.*

The set A contains the preference values to be assigned to the constraint tuples. We derive a partial ordering $\leqslant_S$ over the set A: $\alpha \leqslant_S \beta$ iff $\alpha + \beta = \beta$.[4] The minimum element is $\mathbf{0}$, and $\mathbf{1}$ is the maximum element.

[3] When $+$ is applied to sets of elements, we will use the symbol $\sum$ in prefix notation.
[4] Singleton subsets of the set A are represented without braces.

Definition 2. *A constraint system is a 3-tuple* $CS = \langle S_p, D, V\rangle$, *where* $S_p = \langle A_p, +_p, \times_p, \mathbf{0}, \mathbf{1}\rangle$ *is a c-semiring,* V *is an ordered finite set of variables, and* D *is a finite set containing the allowed values for the variables in* V.

Definition 3. *Given a constraint system* $CS = \langle S_p, D, V\rangle$, *where* $S_p = \langle A_p, +_p, \times_p, \mathbf{0}, \mathbf{1}\rangle$, *a constraint over* CS *is a pair* $c = \langle def_c^p, con_c\rangle$ *where* $con_c \subseteq V$ *(the type), and* $def_c^p : D^k \rightarrow A_p$ *(k is the cardinality of* con_c*). A Semiring Constraint Satisfaction Problem (SCSP) over* CS *is a pair* $P = \langle C, con\rangle$ *where* C *is a finite set of constraints over* CS *and* $con = \bigcup_{c \in C} con_c$.

Definition 4. *Given a constraint system* $CS = \langle S_p,D,V\rangle$ *with* V *totally ordered via* $\preceq$, *consider any k-tuple* $t = \langle t_1,t_2,\ldots,t_k\rangle$ *of values of* D *and two sets* $W = \{w_1,\ldots,w_k\}$ *and* $W' = \{w'_1,\ldots,w'_m\}$ *such that* $W' \subseteq W \subseteq V$ *and* $w_i \preceq w_j$ *if* $i \leq j$ *and* $w'_i \preceq w'_j$ *if* $i \leq j$. *The projection of* t *from* W *to* W', *written* $t \downarrow^W_{W'}$, *is defined as the tuple* $t' = \langle t'_1,\ldots,t'_m\rangle$ *with* $t'_i = t_j$ *if* $w'_i = w_j$.

Definition 5. *Given a constraint system* $CS = \langle S_p,D,V\rangle$ *where* $S_p = \langle A_p,+_p,\times_p,\mathbf{0},\mathbf{1}\rangle$ *and two constraints* $c_1 = \langle def^p_{c_1},con_{c_1}\rangle$ *and* $c_2 = \langle def^p_{c_2},con_{c_2}\rangle$ *over CS, their combination, written* $c_1 \otimes c_2$, *is* $c = \langle def^p_c,con_c\rangle$ *with* $con_c = con_{c_1} \cup con_{c_2}$ *and* $def^p_c(t) = def^p_{c_1}(t \downarrow^{con_c}_{con_{c_1}}) \times_p def^p_{c_2}(t \downarrow^{con_c}_{con_{c_2}})$. *Let* $(\bigotimes C)$ *denote* $c_1 \otimes c_2 \otimes ... \otimes c_n$ *with* $C = \{c_1, ..., c_n\}$. *Given a constraint* $c = \langle def^p_c,\ con_c\rangle$ *over CS, and a set* I $(I \subseteq V)$, *the projection of* c *over* I, *written* $c \Downarrow I$, *is the constraint* $c' = \langle def^p_{c'},\ con_{c'}\rangle$ *over CS with* $con_{c'} = I \cap con_c$ *and* $def^p_{c'}(t') = \sum_{\{t | t \downarrow^{con_c}_{I \cap con_c} = t'\}} def^p_c(t)$.

Definition 6. *Given a SCSP* $P = \langle C,con\rangle$ *over CS, the solution of* P *is a constraint is* $Sol(P) = (\bigotimes C) = \langle def^p_c,\ con\rangle$. *The maximal solution of* P *is* $MSol(P) = \{\langle t,v\rangle | def^p_c(t) = v$ *and there is no* t' *such that* $v <_{S_p} def^p_c(t')\}$.

2 A Branch and Bound Algorithm for SCSPs

In this section we describe Algorithm 1 which appeared in [8]. Assume a SCSP $P = \langle C, con\rangle$ over $CS = \langle S_p, D, V\rangle$, where S_p has best/worst semiring values of $\mathbf{1}$/ $\mathbf{0}$. The upper bound, *UB*, contains the semiring value of the best solution found so far and is initialised with $\mathbf{0}$. For each node in the search tree, its lower bound, *LB* is a semiring value associated with a search path from the root node up to a particular node, and is initialised with $\mathbf{1}$. If $LB \leq_{S_p} UB$, the subtree below the current node is pruned, and the algorithm backtracks to a node at a higher level in the tree.

Note that the starred parameters are variable parameters, and *Queue* contains the decision variables. In line 6 we calculate the semiring value *NewLB* associated with the current node by considering all the constraints which have *var* as a member of its type, and is such that all the variables in their types have been instantiated (i.e. we do a back check). Simply combine the semiring values of the tuples of all these constraints. In line 7 we combine *NewLB* with the lower bound value of the search path up to current node's ancestor, i.e. *LB*. In lines 8-11 we calculate an estimated associated (lower bound) semiring value for the search path below the current node up to a leaf. This estimated lower bound is a semiring value a such that $NewLB \otimes a$ is maximal. This value a is combined with $NewLB$. Note that the variables can be partitioned in disjoint sets, and each partition can be solved as a smaller instance of the original problem.

Algorithm 1 BnB(NoInstantiated*,Queue*,LB*,UB,BestSolution)

```
Require: $V$; $C$; $D$; $S_p$; $N$ (number of variables).
 1: if ($NoInstantiated < N$) then
 2:   $var$ = pop $Queue$; [current decision variable]
 3:   while ( $var\_domain$ not empty) do
 4:     $var\_value$ = select best value from $var\_domain$;
 5:     $var\_domain$ = $var\_domain$ - $var\_value$;
 6:     $NewLB$ = lower bound value for current node;
 7:     $NewLB = \times_p(LB, NewLB)$; [lower bound value for search path including current node]
 8:     if $NoInstantiated \neq N\text{-}1$ then
 9:       $FullLB = FindFullLB(NewLB)$; [estimated lower bound]
10:     else
11:       $FullLB = NewLB$; [complete assignment]
12:     if ($UB <_{S_p} FullLB$) then
13:       $LB = FullLB$;
14:       if ($NoInstantiated = N\text{-}1$) then
15:         $UB = LB$;
16:         $BestSolution$ = current assignment of values for decision variables;
17:         if ($UB = 1$) then
18:           return 1;
19:       if ( $BBnB(Noinstantiated{+}1, Queue, LB, UB, BestSolution) = 1$) then
20:         return 1;
21: return 0;
```

Table 1. Constraint Definitions

t	⟨0, 0⟩	⟨0, 1⟩	⟨0, 2⟩	⟨0, 3⟩	⟨0, 4⟩	⟨1, 0⟩	⟨1, 1⟩	⟨1, 2⟩	⟨1, 3⟩	⟨1, 4⟩	⟨2, 0⟩	⟨2, 1⟩	⟨2, 2⟩
$def^p_{c_1}(t)$	3	2	3	1	0	0	2	3	4	2	3	2	4
$def^p_{c_2}(t)$	3	2	3	0	2	1	1	1	3	3	1	4	0
$def^p_{c_3}(t)$	3	0	4	3	0	2	2	3	3	3	3	2	3
$def^p_{c_4}(t)$	4	0	1	2	2	2	2	1	2	2	2	0	2
$def^p_{c_5}(t)$	2	0	2	2	0	3	3	2	2	0	0	4	3

t	⟨2, 3⟩	⟨2, 4⟩	⟨3, 0⟩	⟨3, 1⟩	⟨3, 2⟩	⟨3, 3⟩	⟨3, 4⟩	⟨4, 0⟩	⟨4, 1⟩	⟨4, 2⟩	⟨4, 3⟩	⟨4, 4⟩	-
$def^p_{c_1}(t)$	1	2	2	2	0	0	0	0	2	2	1	2	-
$def^p_{c_2}(t)$	1	0	4	3	0	2	3	1	3	1	2	2	-
$def^p_{c_3}(t)$	0	2	3	3	0	2	0	0	3	1	3	4	-
$def^p_{c_4}(t)$	3	2	3	2	1	0	3	4	3	3	2	1	-
$def^p_{c_5}(t)$	0	0	0	3	0	2	2	1	3	3	4	0	-

Example 1. Consider a SCSP where $S_p = \langle\ \{0, \ldots, 5\},\ max, min, 0, 5\}\rangle$, $V = con = \{A, B, C, D, E, F, G, H\}$, $D = \{0, 1, 2, 3, 4\}$, and $C = \{c_1, c_2, c_3, c_4, c_5\}$. Table 1 contains the constraint definitions with $c_1 = \langle def^p_{c_1}, \{A, B\}\rangle$, $c_2 = \langle def^p_{c_2}, \{C, D\}\rangle$, $c_3 = \langle def^p_{c_3}, \{E, A\}\rangle$, $c_4 = \langle def^p_{c_4}, \{F, G\}\rangle$, and $c_5 = \langle def^p_{c_5}, \{F, H\}\rangle$. The order of instantiation is: A, B, E, C, D, F, G, H.
Phase 1: Select tuple $\langle 1, 3\rangle$ in row 2 to assign $A = 1$ (LB= 4). Then $B = 3$ (LB= 4). In row 10 select tuple $\langle 3, 1\rangle$ to set $E = 3$. LB= 3. Then $C = 2$, $D = 1$, $F = 0$, $G = 0$, $H = 0$. LB= 2. $UB = 2$. **Phase 2**: Backtrack. *H=2; H=3*; backtrack; *G=3; G=4; G=2*; backtrack; *F=4* (*LB=3*); *G=0*; *H=3. UB= 3.* **Phase 3:** *H=1; H=2; H=0*; backtrack; *G=1; G=2; G =3; G=4*; backtrack; *F=2; F=3;*

F=1; backtrack; *D=0; D=3*; backtrack; *C=3; C=0; C=1; C=4*; backtrack; *E=4; E=1; E=2*; backtrack; *B =2; B=1; B=4*; backtrack; *A=2* (*LB=4*); B = 2; E = 0; C = 2; D = 1; F = 0. **Note phase 4:** *G=0*; *H=0* (*LB=2*); backtrack; *H=2; H=3*; backtrack; *G=3; G=4; G=2*; backtrack. *F=4* (*LB=4*); *G=0*; *H=3.* Maximal solution with associated semiring value of 4: *A=2; B=2; C=2; D=1; E=0; F=4; G=0; H=3.*

3 Two new algorithms for solving SCSPs

Backjumping (BJ) [9] remembers the depth of failure, i.e. the deepest level, l at which any of the values for a variable fails. When all values have been tried for a variable, BJ can proceed directly to the level l. The BJ algorithm for maximal constraint satisfaction (Max-CSP) [1] does not always jump back all the way to the deepest level of failure. If any values below the level l were inconsistent when chosen, it only jumps back to the deepest such level: other choices of values at level l may result in fewer inconsistencies. In our new BJ algorithm, Algorithm 2, *FailDepth1* contains the level where a variable's value may fail first (i.e. closer to the root.) *I_D* (inconsistency depth) keeps track of the deepest level of failure for any value. *R_D* (return depth) is adjusted if a value fails: it is assigned the maximum value at which a failure has been detected. *R_D* is initialised with the value 0, and it controls the level to which recursion is rolled back. If the current node decreases *LB* (line 22), its level becomes the new *I_D* value. *FailDepth2* (line 26) controls the extent to which recursion is rolled back.

Example 2. BJ and BnB proceed in exactly the same way up to Phase 4. After finding *UB= 3*, both algorithms backtrack until A = 2, and then extend the search up to the leaf, try various values for H and then backtrack. Here BnB tries various values for G, and backtracks before assigning F = 4. However, **BJ avoids backtracking from H to G: it backjumps from H to F.**

Forward checking (FC) [10] combines backtracking with a check for local consistency ahead in the search tree. In a partial CSP context, we require an initial value for a bound on the allowed number of unsatisfied constraints. For each value, the number of domains with no supporting values is counted and this arc consistency count (ac) is a lower bound on the increment in the expected number of unsatisfied constraints that will be incurred should this value be added to the solution. In a particular search path, the *ac* for a proposed value assignment to the current variable can be added to the number of unsatisfied constraints so far, and compared to the current bound *NB*. If the sum is not less than *NB*, then the current search path will not result in a better solution.

Our FC algorithm is similar to Algorithm 1 except for the addition just after line 5, of a call to*ForwardCheck(var,var-value)* to perform forward checking. This procedure ensures that the newly instantiated decision variable has a consistent value in the domains of all (uninstantiated) variables that appear in the same constraints. For each of these constraints it ensures that every one of the uninstantiated variables of the constraint has at least one domain value such that the resulting value tuple's associated semiring value does not fail. Note that we have to restore domain values removed from domains during a forward check for a decision variable *var* with the value *var-value*, if this value fails.

Algorithm 2 BJ(NoInstantiated*,Queue*,Dom*,LB*,UB,BestSol,R_D*,I_D*)

```
Require: $V$; $C$; $S_p$; $N$.
1: if (NoInstantiated $<$ N) then
2:    var = pop Queue;
3:    if (var_domain not empty) then
4:       var_value = select best value from var_domain;
5:       var_domain = var_domain - var_value;
6:       NewLB = lower bound for current node;
7:       NewLB = $\times_p$(LB, NewLB);
8:       FailDepth1 = first node form root where var_value fails
9:       if NoInstantiated $\neq$ N-1 then
10:         FullLB = FindFullLowerBound(NewLB); [estimated lower bound]
11:      else
12:         FullLB = NewLB; [complete assignment]
13:   else
14:      return R_D;
15:   if (UB $<_{S_p}$ FullLB) then
16:      FailDepth1 = NoInstantiated;
17:      if (NoInstantiated = N-1) [complete assignment] then
18:         UB = FullLB;
19:         BestSol = current assignment of values for decision variables;
20:         if (UB = 1) then
21:            return N; /* finished */
22:      if (FullLB $<_{S_P}$ LB) then
23:         I_D = NoInstantiated;
24:      FailDepth2 = BJ(NoInstantiated+1,Queue,Dom,FullLB,UB,BestSol,0,I_D);
25:      if (FailDepth2 < NoInstantiated) or (FailDepth2 = N) then
26:         return FailDepth2;
27:      else
28:         R_D = max(FailDepth1,R_D,I_D);
29:         return BJ(NoInstantiated,Queue,Dom,LB,UB,BestSol,R_D,I_D);
30:   R_D = max(FailDepth1,R_D,I_D);
31:   return BJ(NoInstantiated,Queue,Dom,LB,UB,BestSol,R_D,I_D);
32: return NoInstantiated - 1;
```

Example 3. FC and BnB both find $UB = 3$, backtrack and try 1 and 2 for H. BnB then sets $H = 0$, but **FC eliminates this value from the domain of H as part of its forward checking** at the point were it set $F = 4$. It used constraint c_4 to find this value for F, but value 0 for H fails in c_5 during forward checking. Both algorithms then backtrack to G and try values 1 and 2. BnB also tries 3 and 4 for G, while **FC has eliminated both these values from the domain of G** because they fail in c_4. Both algorithms proceed in the same way until Phase 4. After $A = 2$, BnB extends the search down to the leaf node, where $H = 0$. FC only extends the search up to $G = 0$. **FC removed all values from the domain of H when F was instantiated.** FC then backtracks one level, but **all values for G have been removed**, so it backtracks and set $F = 4$.

4 Experiments and Conclusion

We compare the BJ and FC algorithms (without variable partitioning) to the BnB algorithm with (BnBPart) and without variable partitioning. The experiments were performed on an Intel Core 2 Duo processor at 2GHz with 2GB

RAM. We solved 3 sets of randomly generated binary SCSPs that are instances of fuzzy CSPs. All the problems have $S_p = \langle\{0, 0.3, 0.5, 0.8, 1\}\rangle$, *max*, *min*, 0, 1⟩, and 10 domain values. The problems in sets 1/ 2/ 3 have 80/100/120 variables, and 10/10/20 constraints. All sets have 50 problem instances with a tightness of 70% and 50 instances with a tightness of 90%. Set 3 also has 50 instances with a tightness of 85%. A tightness (T) of x% means that (100-x)% of all the tuples have been assigned the best semiring value. Table 2 shows the average runtimes. There is not a significant difference in the performance of the different algorithms for smaller problems, but the results of FC and BJ on Set 3 with 90% tightness are much better than BnB's results. Note that BPart's runtimes are at most 3 times faster than those of BnB, but on average at least 25 times faster than CON'FLEX [4], a fuzzy CSP solver.

We presented two new algorithms for SCSPs that are extensions of the well-known Max-CSP algorithms. Our new algorithms perform better than the BnB algorithm on some problems. In the future, we plan to develop a backmarking algorithm and to test our algorithms on non-trivial SCSPs.

Table 2. Average runtimes of the three algorithms (in seconds)

Set	1	1	2	2	3	3	3
T	70%	90%	70%	90%	70%	85%	90%
BnB	0.0124	0.0552	0.0206	0.0544	0.0688	0.1518	0.8078
BJ	0.0102	0.0774	0.0210	0.0772	0.0778	0.2142	0.6596
FC	0.0194	0.0696	0.0216	0.0698	0.0876	0.2056	0.4758
BPart	0.0164	0.0346	0.0204	0.0382	0.061	0.0916	0.3234

References

1. Freuder, E.C., Wallace, J.W.: Partial constraint satisfaction. AI **58** (1992) 21–70.
2. Bistarelli, S., Montanari, U., Rossi, F.: Semiring-based constraint solving and optimization. Journal of the ACM **44**(2) (1997) 201–236.
3. Delgado, A., Olarte, C., Perez, J., Ruede, C.: Implementing semiring-based constraints using Mozart. In: Proceedings of MOZ 2004. (2004)
4. Georget, Y., Codognet, P.: Compiling semiring-based constraint with clp(fd,s). In: Proceedings of CP-1998. (1998)
5. Bistarelli, S., Fargier, H., Montanari, U., Rossi, F., Schiex, T., Verfaillie, G.: Semiring-based CSPs and valued CSPs: Basic properties and comparison. Constraints **4** (1999) 199–240.
6. Bistarelli, S., Rossi, F., Pilan, I.: Abstracting soft constraints: Some experimental results on Fuzzy CSPs. In: Proceedings of CLCSP-2003. (2003)
7. Bistarelli, S., Fung, S., Lee, J., Leung, H.: A local search framework for semiring-based constraint satisfaction problems. In: Proceedings of Soft-2003. (2003.)
8. Leenen, L., Anbulagan, A., Meyer, T., Ghose, A.: Modelling and solving semiring constraint satisfaction problems by transformation to weighted semiring Max-SAT. In: Proceedings of (Australian) AI-2007. (2007) 202–212.
9. Gaschnig, J.: Experimental case studies of backtrack vs. Waltz-type vs. new algorithms for satisficing assignment problems. In: Proceedings of CCSCSI-78. (1978.)
10. Haralick, R., Elliot, G.: Increasing tree search efficiency for constraint satisfaction problems. Artificial Intelligence **14** (1980) 263–313.