SCIENCE CHINA
Information Sciences

• **Supplementary File** •

Revisiting the Efficacy of Weak Consistencies: a Study of Forward Checking

Zhe LI[1,2], Zhezhou YU[1,2], Hongbo LI[3], Jinsong GUO[4] & Zhanshan LI[1,2*]

[1]*Key Laboratory for Symbol Computation and Knowledge Engineering of National Education Ministry, Changchun* 130012*, China;*
[2]*College of Computer Science and Technology, Jilin University, Changchun* 130012*, China;*
[3]*College of Information Science and Technology, Northeast Normal University, Changchun* 130024*, China;*
[4]*Department of Computer Science, University of Oxford, Oxford* OX1 2JD*, UK*

Appendix A Background

A constraint satisfaction problem (CSP) $\mathcal{P}$ is a triple $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ where $\mathcal{X}$ is a set of n variables $\mathcal{X} = \{x_1, x_2 \ ... \ x_n\}$, $\mathcal{D}$ is a set of domains $\mathcal{D} = \{D(x_1), D(x_2) \ ... \ D(x_n)\}$ where $D(x_i)$ is a finite set of possible values for variable x_i, $\mathcal{C}$ is a set of e constraints $\mathcal{C} = \{c_1, c_2 \ ... \ c_e\}$. A constraint c consists of two parts, an ordered set of variables $scp(c) = \{x_{i_1}, x_{i_2} \ ... \ x_{i_r}\}$ and a subset of the Cartesian product $D(x_{i_1}) \times D(x_{i_2}) \times ... \times D(x_{i_r})$ that specifies the allowed combinations of values for the variables $\{x_{i_1}, x_{i_2} \ ... \ x_{i_r}\}$. A constraint involving variables x_i and x_j is denoted by c_{ij} and a value $a \in D(x_i)$ is denoted by (x_i, a). We focus on binary CSPs where every constraint $c \in \mathcal{C}$ involves only two variables. x_i and x_j are neighboring variables of each other if there exists a constraint c_{ij} in $\mathcal{C}$. We call binary CSPs constraint networks in this paper.

Local consistency (LC) conditions are properties of CSPs related to the consistency of subsets of variables or constraints. They can be used to reduce the search space and make the problem easier to solve. Given a constraint network $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ and a value a in $D(x_i)$ for some $x_i \in \mathcal{X}$, we use $\mathcal{P}|_{x_i=a}$ (resp. $\mathcal{P}|_{x_i \neq a}$) to denote the procedure of restricting $D(x_i)$ to $\{a\}$ (resp. deleting $\{a\}$ from $D(x_i)$). In the rest of this paper, we use $LC(\mathcal{P})$ to enforce some kind of local consistency on network $\mathcal{P}$. And furthermore, $LC(\mathcal{P}|_{x_i=a})$ denotes the procedure of applying some kind of local consistency to $\mathcal{P}|_{x_i=a}$. Nevertheless, it is barely solving a CSP instances only by performing local consistency processing. The following algorithm A1 gives the pseudo-code of the generalized version of the binary (non-binary) backtracking search algorithm. Backtracking search is a complete approach in which systematic exploration of the search space of a CSP instance is guaranteed. In this process, the search tree grows to find a solution or prove that no solution exists. More specically, with binary backtracking search, at each search step, a pair (x_i, a) is selected, where x_i is an unfixed variable (i.e. a variable whose domain is not singleton) , and a is a value in $D(x_i)$, and two cases are considered: the assignment $x_i = a$ and the refutation $x_i \neq a$. With non-binary scheme, at each search step, an unfixed variable x is selected, and then for each value a in $D(x_i)$, the assignment $x_i = a$ is considered.

Definition 1 (Arc Consistency, AC)**.** Given a constraint network $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$:

- A variable x_i is arc consistent with a constraint c_{ij}, where $scp(x_{ij}) = \{x_i, x_j\}$, iff $D(x_i) \neq \emptyset$ and $\forall\ a \in D(x_i)$, there exists a value $b \in D(x_j)$ s.t. the tuple (a, b) satisfies c_{ij}. (x_j, b) is called a support of (x_i, a) in constraint c_{ij}.
- A constraint $c_{ij} \in \mathcal{C}$, where $scp(x_{ij}) = \{x_i, x_j\}$, is arc consistent iff both x_i and x_j are consistent with c_{ij}.
- $\mathcal{P}$ is arc consistent iff all constraints in $\mathcal{C}$ are arc consistent.

We say a value (x_i, a) is arc consistent iff it has at least one support in each constraint involving x_i. Backtracking search is a complete method to solve CSPs. It builds up a search tree from level 0 to level n, where n is the number of variables. At each node of the search tree, a variable x_i and a value $a \in D(x_i)$ are selected and a local consistency is established to propagate the assignment. A dead-end is reached if the propagation fails, then a backtracking occurs.

Forward Checking (FC) is a classic method maintaining a partial form of AC during backtracking search. The consistency maintained in FC is called Forward Checking Consistency, as has been defined in [1]. Different from AC which propagates thorough the whole network, only the neighboring variables of the variables already been instantiated are revised by FCC.

Definition 2 (Forward Checking Consistency, FCC)**.** Given a constraint network $\mathcal{P}=\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ with a subset $\mathcal{Y} \subseteq \mathcal{X}$ where $\mathcal{Y}$ contains all the variables that have been instantiated, $\mathcal{P}$ is forward checking consistent according to the instantiation I on $\mathcal{Y}$ iff I is local consistent and for all $x_i \in \mathcal{Y}$, each constraint involving x_i is arc consistent.

* Corresponding author (email: zslizsli@163.com)

Algorithm A1 Binary (non-binary) Backtracking search

```
1: P ⇐ LC(P)
2: if P = ⊥ then
3:    return fail;
4: end if
5: if ∀x_i ∈ X, |D(x_i)| = 1 then
6:    return success;
7: end if
   /* If the algorithm is Binary Backtracking search */
8: select a value (x_i, a) of P such that |D(x_i)| > 1
9: return Binary Backtracking search(P|_{x_i=a}) or Binary Backtracking search(P|_{x_i≠a})
   /* If the algorithm is non-Binary Backtracking search */
10: select a variable x_i of P such that |D(x_i)| > 1
11: for each value a ∈ D(x_i) do
12:    if non-Binary Backtracking search(P|_{x_i=a}) then
13:       return success;
14:    end if
15: end for
16: return fail
```

Algorithm A2 AC3(FCC)

```
1: initialize Q;
2: while Q is not empty do
3:    select and remove an arc (x_i, x_j) from Q;
4:    if revise(x_i, x_j) then
5:       if D(x_i) = ∅ then
6:          return fail;
7:       else
8:          for each constraint c_ik such that k ≠ j do
9:             Q ⇐ Q ⋃ {(x_k, x_i)};
10:         end for
11:      end if
12:   end if
13: end while
14: return success;
```

FCC is a partial form of AC, so we recall the classic AC3 algorithm in Algorithm A2 and discuss the difference between AC and FCC later. The pair of a variable x_i and a constraint c_{ij} involving it is called an arc. For simplicity, the arc is denoted by the two variables (x_i, x_j). Note that (x_i, x_j) and (x_j, x_i) are different arcs. To establish Arc Consistency in a CSP, AC algorithms revise all the arcs in the problem. Before search starts, the initialization step adds all arcs in the network into Q. During the searching phase, the initialization of Q after each assignment $x_i = a$ adds only the arcs (x_j, x_i) into Q, where x_j is constrained with x_i. The revise(x_i, x_j) procedure in Algorithm A3 seeks supports for every value (x_i, a) in $D(x_i)$ and removes the values without a support. If the domain of a variable wipes out, AC fails. If we remove the lines 8-10, the AC3 algorithm degenerates into a FCC algorithm. Note that FCC is not applicable before search.

Algorithm A3 revise(x_i, x_j)

```
1: change ⇐ false;
2: for each value (x_i, a) ∈ D(x_i) do
3:    if ¬ seekSupport(x_i, a, x_j) then
4:       remove a from D(x_i);
5:       change ⇐ true;
6:    end if
7: end for
8: return change;
```

The $AC3^{bit}$ algorithm is built on the binary representation of domains and constraints [2]. Each domain $D(x_i)$ is represented by a bit vector $bitDom[x_i]$ and each value a in the domain is associated with a single bit $bitDom[x_i][a]$. When a bit is set to 1 (resp. 0), it means that the corresponding value is still in the domain (resp. removed from the domain). For each (x_i, a) and a constraint c_{ij}, a bit vector $bitSup[x_i][a][x_j]$ represents the initial supports of (x_i, a) in $D(x_j)$. If (x_j,

Algorithm A4 SEEKSUPPORT(x_i, a, x_j)

```
1: if (bitSup[x_i][a][x_j] & bitDom[x_j]) ≠ 0 then
2:     return true;
3: end if
4: return false;
```

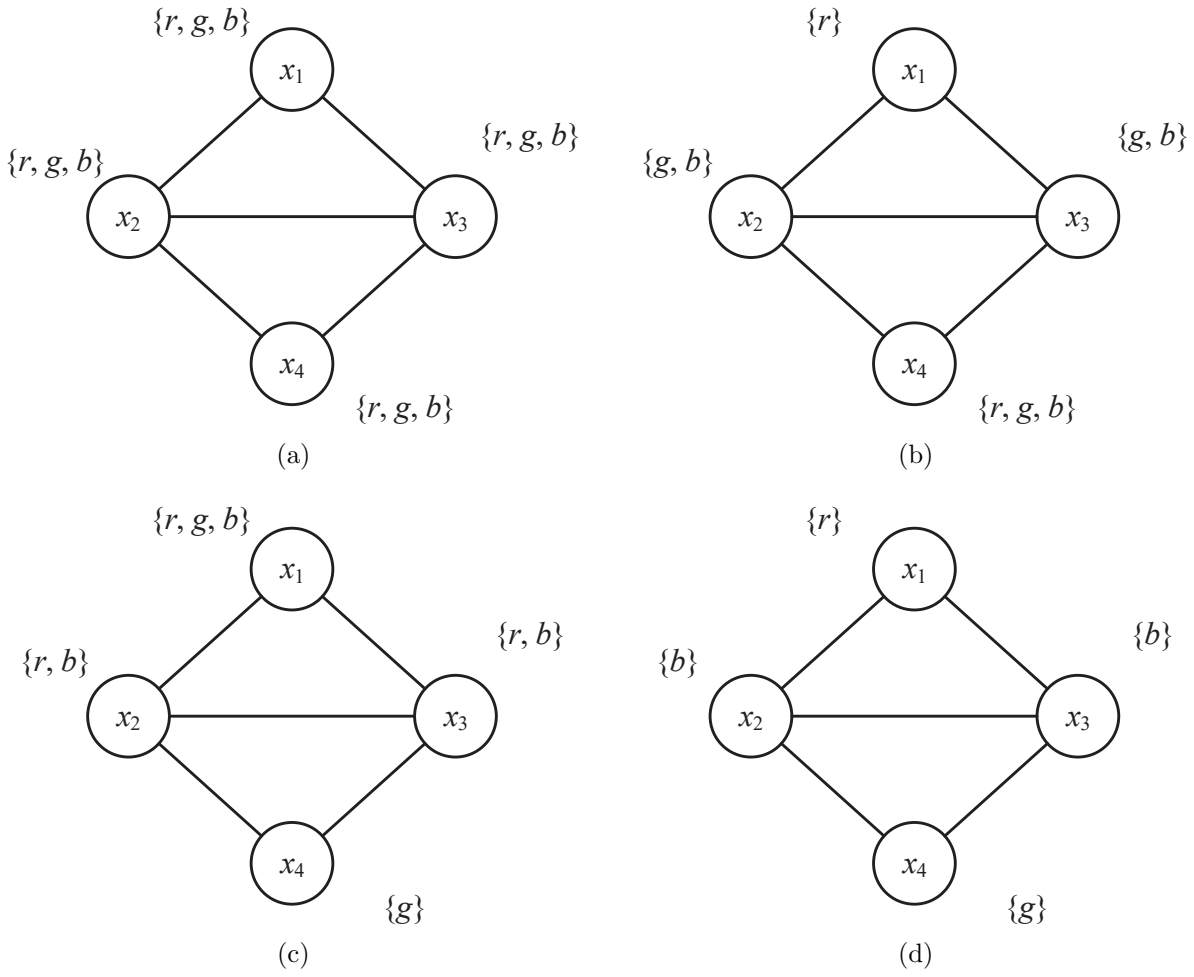

Figure B1 An example illustrating AC is strictly stronger than EFCC. (a) Original network $\mathcal{P}$; (b) $AC(\mathcal{P}|_{x_1=r})$; (c) $AC(\mathcal{P}|_{x_4=g})$; (d) $\mathcal{P}$' derived from $\mathcal{P}$ by the instantiation : $x_1 = r, x_4 = g$.

b) is a support of (x_i, a), then the b-th bit of $bitSup[x_i][a][x_j]$ is set to 1, otherwise 0.

If the computer is equipped with a L-bit processor and the domain sizes are larger than L, then each $bitDom[x_i]$ or $bitSup[x_i][a][x_j]$ will be implemented by an array. In this paper, to present the algorithm more clearly, we assume that the domain sizes are smaller than L and the i-th value in each domain equals to i. Then the SEEKSUPPORT procedure of $AC3^{bit}$ is one logical AND operation (the operator &), as is shown in Algorithm A4. If there exists a support of (x_i, a), at least one bit in the results of the AND operations is 1.

Appendix B Proof of propositions

Proposition 1. On a network with at least one variable being instantiated, AC is strictly stronger than EFCC.

Proof. We first prove that for any network $\mathcal{P}|_{\mathcal{I}}$ derived from $\mathcal{P}$ with a subset $\mathcal{Y} \subseteq \mathcal{X}$ being instantiated, if AC holds in $\mathcal{P}|_{\mathcal{I}}$, EFCC must hold too. Suppose that there is a $\mathcal{P}|_{\mathcal{I}}$, AC holds but EFCC does not hold, i.e. $\mathcal{P}|_{\mathcal{I}}$ contains some value (x_j, b) that belongs to $\mathcal{E}_{AC(x_i,\mathcal{I}(x_i))}$ where $x_i \in \mathcal{Y}$. Thus $\mathcal{P}|_{\mathcal{I}}$ is not arc consistent since (x_j, b) is arc inconsistent (otherwise it should not have been removed by AC($\mathcal{P}|_{x_i=a}$)), which contradicts with the assumption. The assumption does not hold so the stronger relationship is proved.

Strictness is proved by giving a network which is EFCC but not AC. Figure B1 gives an example of the coloring problem.

Each node is a variable and every two connected variables should have different colors. The $\mathcal{P}$' depicted in Figure B1(d) is a network derived from Figure B1(a) by the instantiation of $x_1 = r$ and $x_4 = g$. $\mathcal{P}$' is enforced forward checking consistent because there is no domain wipeout in either AC($\mathcal{P}|_{x_1=r}$) (Figure B1(b)) or AC($\mathcal{P}|_{x_4=g}$) (Figure B1(c)), and all the values of x_2 and x_3 exist in both AC($\mathcal{P}|_{x_1=r}$) and AC($\mathcal{P}|_{x_4=g}$). However, $\mathcal{P}$' is clearly not arc consistent as the c_{23} is not satisfied.

Hence the theorem is proved.

Proposition 2. EFCC is strictly stronger than FC.

The proof of this proposition boils down to proving AC($\mathcal{P}|_{x_i=a}$) can eliminate more values than eliminating values being arc inconsistent with (x_i, a) from the domains of variables constrained with x_i. It is quite obvious since the latter operation is the first step of the former one. Thereby we don't give a formal proof due to the space limitation.

Proposition 3. The time cost of the propagation process of FC^{bit} is $O(nd)$.

We don't bother to prove this complexity, which is simple. Instead, we point out that although this time complexity is the same as the normal FC algorithm as described in [3], the actual required operations are L-times fewer when running with a L-bit processor.

Proposition 4. The space complexity of FC^{bit} is $O(n^2d + ed^2)$, where n is the number of variables, e is the number of constraints and d is the maximum domain size.

Proof. For each constraint, there are $O(d)$ *bitSup* arrays, and each array $bitSup[x_i][a][x_j]$ costs d/L space, so the space cost of recording all the *bitSup*s is $O(ed^2)$. The data structure *bitDomAtLevel* needs $O(n^2d)$ space, because each level needs $O(nd/L)$ space and a complete search tree has $n + 1$ levels. Thus, the space cost of FC-bit is $O(n^2d + ed^2)$.

Proposition 5. EFC^{bit} is correct.

Proof. EFC^{bit} eliminates values at two steps. In the preprocessing step, any removed value (x_i, a) cannot lead to a network $\mathcal{P}|_{x_i=a}$ satisfying AC, so it will not appear in a solution. Thereby to prove the correctness of EFC^{bit}, we only need to further prove that after each assignment $x_i = a$, all values removed by EFCC will not appear in a solution containing (x_i, a). All the values removed after the instantiation of (x_i, a) belong to $\mathcal{E}_{AC(x_i,a)}$, i.e., they are removed because they do not satisfy AC in the network $\mathcal{P}|_{x_i=a}$. Since enforcing AC on $\mathcal{P}|_{x_i=a}$ never removes any value involved in a solution containing (x_i, a), the correctness of EFC^{bit} is proved.

As EFC^{bit} uses the same propagation algorithm as FC^{bit}'s, the propagation time complexity of EFC^{bit} is the same as FC^{bit}'s which is $O(nd)$. As for the space complexity, EFC^{bit} requires $O(n^2d^2)$ space for the *bitsup* arrays (as it maintains a *bitSup* for each pair of variables), and requires the same space for *bitDomAtLevel* arrays as FC^{bit} does, so the space complexity of EFC^{bit} is $O(n^2d^2)$.

Appendix C Experimental details

The presented running time of EFC^{bit} includes its preprocessing time. We have dropped those easy instances for which all the tested algorithms used less than 1 second to find a solution. The results of the remaining instances are present in Table C1 and C2. The best of each row is in bold. For each group of instances, we present the average cpu time (cpu) in seconds and the search nodes (#n(M)) during search, and timeout (cpu > 1800 seconds) times (#to). The timeout instances are assigned a 1800s cpu time. The #nodes is not present for the problems containing timeout instances.

References

1 Bessiere C. Constraint Propagation. Handbook of Constraint Programming, 2006, 3: 29-83.

2 Lecoutre C, Vion J. Enforcing arc consistency using bitwise operations. Constraint Programming Letters (CPL), 2008, 2: 21-35.

3 Haralick R M, Elliott G L. Increasing tree search efficiency for constraint satisfaction problems. Artificial intelligence, 1980, 14(3): 263-313.

Table C1 Results of comparing algorithms on group instances

		dom					*dom/wdeg*				
Instances		EFC^{bit}	FC^{bit}	$AC3^{bit}$	$lmaxRPC$	$rRPC3$	EFC^{bit}	FC^{bit}	$AC3^{bit}$	$lmaxRPC$	$rRPC3$
rand-2-23	cpu	**0.7365**	0.7481	12.0019	49.9357	36.2498	1.0291	0.8487	16.7316	49.3018	42.062
#=10	#n(M)	2.0874	2.0874	0.3162	**0.1205**	0.1428	2.1605	2.1605	0.3731	0.1316	0.1648
rand-2-24	cpu	1.6157	**1.5373**	25.683	111.22	76.787	2.2014	1.8606	31.533	112.11	94.714
#=10	#n(M)	4.2846	4.2846	0.65	**0.2495**	0.2936	4.4454	4.4454	0.8036	0.3257	0.3776
rand-2-25	cpu	3.082	**3.0272**	49.334	224.27	248.80	4.2401	3.7788	64.3228	194.87	177.33
#=10	#n(M)	8.1601	8.1601	1.2367	**0.4749**	0.56	8.5503	8.5503	1.6939	0.5724	0.7088
rand-2-26	cpu	**7.962**	8.2541	127.23	625.17	623.46	11.381	11.093	131.28	478.86	407.20
#=10	#n(M)	22.6891	22.6891	3.4415	**1.3317**	1.5652	23.727	23.727	3.3374	1.3937	1.6384
rand-2-27	cpu	**17.10**	17.26	288.8	1012.9	1133.9	28.45	23.88	338.78	1355.9	1071.3
	#n(M)	49.36	49.36	7.4931	**2.9123**	3.3354	51.2064	51.2064	8.4352	3.7601	4.1711
#=10	#to	**0**	**0**	**0**	**0**	1	**0**	**0**	**0**	1	**0**
2-30-15(fcd)	cpu	0.0797	0.0738	0.5116	0.9855	1.1608	0.1044	**0.0354**	0.1518	0.4481	0.6451
#=100	#n(M)	0.1488	0.1647	0.0145	0.0054	0.0063	0.05	0.0527	0.0058	**0.0027**	0.0029
2-40-19(fcd)	cpu	10.338	12.703	74.730	148.73	233.49	5.4348	**4.8345**	18.578	41.388	77.573
#=100	#n(M)	22.7294	24.3531	1.8398	0.7134	0.8147	5.099	5.1761	0.5035	**0.2518**	0.254
2-50-23(fcd)	cpu	993.75	1011.84	1593.3	1708.2	1761.2	**443.89**	448.75	1116.1	1365.1	1502.8
#=100	#to	28	28	77	87	94	**2**	3	40	56	69
40-8-753-0.1	cpu	0.2148	**0.1791**	2.4758	12.6302	7.5507	0.4514	0.3309	2.399	11.5618	7.7041
#=100	#n(M)	0.351	0.3514	0.0532	0.0232	0.0244	0.2897	0.2898	0.0465	**0.0219**	0.0242
40-11-414-0.2	cpu	0.6891	0.7427	5.5968	14.137	13.9745	0.8007	**0.6578**	2.9927	7.0673	7.7247
#=100	#n(M)	1.4309	1.4354	0.1618	0.0696	0.0749	0.5996	0.601	0.0791	**0.0356**	0.0368
40-16-250-0.35	cpu	2.0933	2.4141	9.801	18.8151	27.2095	1.0949	**0.9193**	2.6044	4.8006	8.1408
#=100	#n(M)	4.4032	4.4628	0.3461	0.1388	0.1547	0.8701	0.8819	0.0764	**0.0368**	0.0396
40-25-180-0.5	cpu	8.1589	9.0487	28.4064	42.408	87.292	1.5696	**1.27**	3.2175	5.2069	11.2108
#=100	#n(M)	17.3332	17.9892	0.9685	0.3269	0.3868	1.3494	1.4038	0.0921	**0.0426**	0.0446
40-40-135-0.65	cpu	29.513	35.535	83.846	81.679	203.67	4.1424	**1.5022**	2.079	3.3852	14.183
#=100	#n(M)	64.6532	75.0859	2.1028	0.5481	0.5988	3.8895	1.8086	0.05	**0.0255**	0.0271
40-80-103-0.8	cpu	204.95	312.60	376.97	263.3	573.87	**5.5796**	23.690	5.73835	6.3757	31.829
	#n(M)	394.70	627.48	6.0079	1.4152	0.7179	3.8272	23.7334	0.1348	0.0469	**0.0466**
#=100	#to	7	10	14	7	21	**0**	1	**0**	**0**	**0**
40-180-84-0.9	cpu	180.62	1054.4	655.62	445.06	861.02	**5.9687**	54.934	8.7941	6.7887	54.841
	#n(M)	193.61	1569.1	5.5771	1.1273	0.4163	1.7029	49.753	0.0817	**0.0286**	0.0295
#=100	#to	5	54	26	15	40	**0**	1	**0**	**0**	**0**
Geometric	cpu	35.20	32.97	87.89	101.1	109.2	0.426	**0.207**	1.901	4.079	5.378
	#n(M)	66.1	63.93	1.615	0.345	0.356	0.147	0.147	0.023	**0.009**	0.0104
#=100	#to	1	1	4	4	4	**0**	**0**	**0**	**0**	**0**
Graph Coloring	cpu	100.8	101.06	129.6	175.4	198.6	**1.088**	6.33	4.737	4.65	11.55
	#n(M)	304.05	309.05	85.94	47.42	22.03	1.249	8.825	1.098	1.098	**1.013**
#=22	#to	1	1	1	1	1	**0**	**0**	**0**	**0**	**0**
Ehi	cpu	0.0019	0.0199	3.5117	0.0153	1.1349	**0.0015**	0.2729	0.1002	0.016	0.2710
#=200	#n(M)	**0**	0.0001	0.1323	**0**	0.0012	**0**	0.0098	0.0015	**0**	0.0002
BQWH-18	cpu	5.1706	6.6758	9.2092	2.0669	4.1637	0.6411	0.9908	0.4544	**0.2522**	0.387
#=100	#n(M)	3.7145	4.8527	1.2629	0.1064	0.0824	0.1602	0.194	0.0314	0.0074	**0.0059**
BH-4-4	cpu	25.716	99.401	415.89	1225.0	1034.9	**0.0458**	215.03	1.1997	7.6485	3.8529
#=10	#n(M)	49.084	226.03	22.641	14.502	10.889	**0.0307**	228.69	0.0991	0.0454	0.0372
QWH-20	cpu	336.84	382.81	432.52	243.07	403.75	117.74	94.353	36.336	**8.8326**	30.418
	#n(M)	107.97	127.97	25.309	4.9144	1.5876	7.0676	5.1794	0.8634	0.1222	**0.0936**
#=10	#to	1	1	1	1	1	**0**	**0**	**0**	**0**	**0**
QCP-15	cpu	283.41	305.18	418.10	139.93	435.74	11.818	10.5167	12.153	**4.7966**	12.579
	#n(M)	152.11	155.75	42.706	5.4031	4.2593	1.4244	1.6439	0.3887	0.0836	**0.0702**
#=15	#to	**0**	**0**	1	**0**	1	**0**	**0**	**0**	**0**	**0**
composed	cpu	0.1731	623.94	598.64	187.81	382.93	0.1427	0.0137	**0.0081**	0.0096	0.0151
25-10-20	#n(M)	**0.0001**	748.89	93.17	18.151	13.778	0.0001	0.0038	0.0005	0.0004	0.0003
#=10	#to	**0**	3	3	1	2	**0**	**0**	**0**	**0**	**0**
driver	cpu	59.999	423.37	26.562	17.211	74.045	2.2171	1.6257	0.9831	**0.721**	2.4927
	#n(M)	7.0128	113.77	0.5631	0.1792	0.2475	0.0118	0.0396	0.0062	**0.0032**	0.0038
#=7	#to	**0**	1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
pigeons	cpu	803.33	**803.19**	1047.8	1140.0	868.70	803.38	803.32	846.16	1147.4	870.66
#=25	#to	**11**	**11**	14	15	**11**	**11**	**11**	**11**	15	12
QueensKnights	cpu	0.2498	915.68	913.70	933.46	952.57	**0.186**	10.793	0.7365	10.249	2.0213
	#n(M)	**0**[1)]	1590.6	11.7954	5.2235	3.1088	**0**	8.4776	0.0074	0.0225	0.0042
#=12	#to	**0**	6	6	6	6	**0**	**0**	**0**	**0**	**0**

Table C2 Results of comparing algorithms on single instances

Instances		*dom*					*dom/wdeg*				
		EFC^{bit}	FC^{bit}	$AC3^{bit}$	$lmaxRPC$	$rRPC3$	EFC^{bit}	FC^{bit}	$AC3^{bit}$	$lmaxRPC$	$rRPC3$
enddr2-10-by-5-3	cpu	30.122	TO	**0.6**	1.452	1.409	TO	TO	TO	901.578	TO
	#n(M)	44.4564	-	0.0147	**0.0052**	0.0094	-	-	-	14.081	-
ewddr2-10-by-5-1	cpu	TO	0.309	2.967	0.374	1.888	5.799	0.44	**0.007**	0.053	0.043
	#n(M)	-	0.0065	0.1066	0.0005	0.0097	0.0001	0.0154	0.0001	0.0001	0.0001
graph10	cpu	84.013	TO	**0.043**	0.376	TO	TO	29.066	0.132	0.269	0.337
	#n(M)	**0.0007**	-	0.0007	0.0007	-	-	0.3162	0.0012	0.0007	0.0007
graph14-f27	cpu	TO	TO	TO	TO	TO	7.855	166.156	0.156	**0.111**	0.853
	#n(M)	-	-	-	-	-	0.0141	0.911	0.0016	**0.0009**	0.0018
graph14-f28	cpu	TO	TO	TO	TO	TO	8.116	15.728	0.507	**0.369**	1.038
	#n(M)	-	-	-	-	-	0.0121	0.0402	0.005	0.0029	**0.0013**
graph8-f10	cpu	TO	TO	TO	TO	TO	11.99	617.134	**0.468**	0.633	1.258
	#n(M)	-	-	-	-	-	0.0316	7.0885	0.004	0.0034	**0.0024**
scen11	cpu	317.368	TO	1465.146	TO	TO	21.603	1.767	**0.077**	0.311	0.409
	#n(M)	85.7877	-	35.2207	-	-	0.0045	0.0061	0.0009	**0.0007**	0.0008
scen3-f10	cpu	3.716	0.318	**0.012**	0.09	0.151	4.989	2.8	0.061	0.154	0.118
	#to	0.0004	0.0015	0.0004	**0.0004**	0.0004	0.0025	0.0894	0.0008	0.0006	0.0005
scen6-w2	cpu	**0.002**	TO	TO	**0.002**	TO	**0.002**	0.092	0.022	**0.002**	0.017
	#n(M)	**0**	-	-	**0**	-	**0**	0.0005	0.0008	**0**	0.000017
scen7-w1-f4	cpu	1.583	29.59	2.453	**0.008**	25.513	1.504	1.037	0.009	0.01	0.042
	#n(M)	**0.0004**	22.6187	0.1934	0.0004	0.1934	0.0004	0.0274	0.0005	0.0004	0.0005
scen7-w1-f5	cpu	0.035	TO	TO	**0.002**	TO	0.05	270.983	0.01	**0.002**	0.023
	#n(M)	**0**	-	-	**0**	-	**0**	7.4437	0.0003	**0**	0.0001
qa-5	cpu	0.45	0.378	3.136	5.272	5.167	0.477	0.361	**0.139**	0.225	0.179
	#n(M)	1.0709	1.7019	0.3186	0.0475	0.0382	0.3755	1.0114	0.0096	**0.0012**	0.0015
qa-6	cpu	8.547	23.926	199.144	557.584	334.88	**3.024**	44.223	3.426	8.003	21.305
	#n(M)	22.957	88.0987	7.7034	1.2717	0.8759	0.5724	59.6247	0.0955	**0.0149**	0.0538