RC23457 (W0412-001) December 1, 2004
Computer Science

IBM Research Report

A Faster Exponential-Time Algorithms for Max 2-Sat, Max Cut, and Max *k*-Cut

Alexander D. Scott, Gregory B. Sorkin
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

A FASTER EXPONENTIAL-TIME ALGORITHM FOR MAX 2-SAT, MAX CUT AND MAX k-CUT

ALEXANDER D. SCOTT AND GREGORY B. SORKIN

Abstract. A recent line of research has been to speed up exponential-time algorithms (deterministic or randomized) for maximization problems such as Max 2-Sat and Max Cut. The fastest previous algorithms run in time $\widetilde{O}(2^{m/5})$ for a Max 2-Sat instance with m clauses, and $\widetilde{O}(2^{m/4})$ for a Max Cut instance with n vertices and m edges (by applying the Max 2-Sat algorithm to a transformation of the graph).

In this paper, we work with a larger class of problems, "Max $(r, 2)$-CSP". We give a deterministic algorithm which solves any m-constraint Max $(r, 2)$-CSP instance in time $\widetilde{O}(r^{19m/100})$ (i.e., $\widetilde{O}(r^{m/5.26\cdots})$) and space $O(mn)$. In particular, in time $\widetilde{O}(2^{19m/100})$ we can solve Max Cut, Max Dicut, Max 2-Lin, Max 2-Sat, Max-Ones-2-Sat, maximum independent set, and minimum dominating set, and in time $\widetilde{O}(k^{19m/100})$ we can solve Max k-Cut; weighted versions of any of these problems can be accommodated at no extra cost. The algorithm is relatively simple, using just two transformation rules and one splitting rule, and the analysis hinges on a small linear program.

1. Introduction

A recent line of research has been to speed up exponential-time algorithms (deterministic or randomized) for maximization problems such as Max 2-Sat and Max Cut. The fastest previous algorithms (see Section 2 for further discussion) run in time $\widetilde{O}(2^{m/5})$ for a Max 2-Sat instance with m clauses, and $\widetilde{O}(2^{m/4})$ for a Max Cut instance with n vertices and m edges (by applying the Max 2-Sat algorithm to a transformation of the graph).

In this paper, we consider a more general class of problems rather than specific problems such as Max Cut or Max 2-SAT. Like previous approaches to problems of this sort, our method is to repeatedly reduce an instance to a smaller one until it becomes trivial; the reductions are then reversed to recover the solution to the original instance. However, with our reductions, an instance of Max Cut, for example, may be transformed to a problem which is *not* an instance of Max Cut; this impels us towards the more general problem class Max $(r, 2)$-CSP which *is* closed under our reductions. We believe that the acceptance of this greater generality is largely responsible for the simplicity of our algorithm and its improved running time: reductions do not need to be tailored to stay within a smaller problem class, so they can be fewer, simpler, and more powerful. Before sketching the recent history of Max $(r, 2)$-CSP and related problems let us be concrete and define (and discuss) the class Max $(r, 2)$-CSP. Readers willing to live with a lacuna may prefer to skip to Section 2, which places our results in the context of previous work.

1.1. Defining Max $(r, 2)$-CSP.

The problem Max Cut is to partition the vertices of a given graph into two classes so as to maximize the number of edges "cut" by the partition. Think of each edge as being a function on the classes (or "colors") of its endpoints, with value 1 if the endpoints are of different colors, 0 if they are the same: Max Cut is equivalent to finding a 2-coloring of the vertices which maximizes the sum of these edge functions. This view naturally suggests a generalization.

An *instance* (G, S) of Max $(r, 2)$-CSP is given by an "underlying" graph $G = (V, E)$ and a set S of "score" functions. Writing $[r] = \{1, \ldots, r\}$ for the set of available colors, we have a "dyadic" score function $s_e : [r]^2 \to \mathbb{R}$ for each edge $e \in E$, a "monadic" score function $s_v : [r] \to \mathbb{R}$ for each vertex $v \in V$, and finally a single "niladic" score function $s_0 : [r]^0 \to \mathbb{R}$ which takes no arguments and is just a constant convenient for bookkeeping.

A *candidate solution* is a function $\phi : V \to [r]$ assigning "colors" to the vertices, and its *score* is

$$s(\phi) \doteq s_0 + \sum_{v \in V} s_v(\phi(v)) + \sum_{uv \in E} s_{uv}(\phi(u), \phi(v)). \tag{1}$$

An *optimum solution* ϕ is one which maximizes $s(\phi)$.

We don't want to belabor the notation for edges, but we wish to take each edge just once, and (since s_{uv} need not be a symmetric function) with a fixed notion of which endpoint is "u" and which is "v". We will typically assume that $V = [n]$ and any edge uv is really an ordered pair (u, v) with $1 \le u < v \le n$. We remark that the "2" in the name Max $(r, 2)$-CSP refers to the fact that the score functions take 2 or fewer arguments (3-Sat, for example, is out of scope). Replacing 2 by a larger value would also mean replacing the underlying graph with a hypergraph, and changes the picture significantly.

An obvious complexity issue is raised by our allowing scores to be arbitrary *real* values. Our algorithm will add, subtract, and compare these values (never introducing a value larger than the sum of those in the input) and we assume that each such operation can be done in time $O(1)$ and represented in space $O(1)$. If desired, scores may be limited to integers, and the length of the integers factored in to the algorithm's complexity, but this seems uninteresting and we will not remark on it further.

1.2. **Notation.** For brevity, we will often write "d-vertex" in lieu of "vertex of degree d". Otherwise, our notation is fairly standard, and we reserve the symbols introduced above for these roles: G for the underlying graph of a Max $(r, 2)$-CSP instance, n and m for its number of vertices and edges, and $[r] = \{1, \dots, r\}$ for its allowed colors. The notation $\widetilde{O}(\cdot)$ suppresses polynomial factors in any parameters, so for example $\widetilde{O}(r^{cn})$ may mean $O(r^3 n\, r^{cn})$.

1.3. **Remarks on Max $(r, 2)$-CSP.** Our assumption of an undirected underlying graph is sound even for a problem such as Max Dicut (maximum directed cut): the edge (u, v) has score 1 if $(\phi(u), \phi(v)) = (0, 1)$, and score 0 otherwise. There is also no loss of generality in assuming that an input instance has a simple underlying graph (no loops or multiple edges), or by considering only maximization and not minimization problems.

Readers familiar with the class $\mathcal{F}$-Sat (see for example Marx [Mar04], Creignou [Cre95], or Khanna [KSTW01]) will realize that when the arity of $\mathcal{F}$ is limited to 2, Max $(r, 2)$-CSP contains $\mathcal{F}$-Sat, $\mathcal{F}$-Max-Sat and $\mathcal{F}$-Min-Sat, and similar problems where we maximize the weight rather than merely the number of satisfied clauses. This includes Max 2-Sat, Max 2-Lin (satisfying as many as possible of m 2-variable linear equalities and/or inequalities). Max $(r, 2)$-CSP also contains $\mathcal{F}$-Max-Ones; for example Max-Ones-2-Sat.

Max $(r, 2)$-CSP also includes problems that are not obviously structured around pairwise constraints, such as Max Cut, maximum independent set ("Max IS") and minimum dominating set. For example, to model Max IS as a Max $(r, 2)$-CSP, let $\phi(v) = 1$ if vertex is to be included in the set and 0 otherwise, define vertex scores $s_v(\phi(v)) = \phi(v)$ and define edge scores $s_{uv}(\phi(u), \phi(v)) = -2$ if $\phi(u) = \phi(v) = 1$, and 0 otherwise

2. Literature survey

We are not sure where the class (a, b)-CSP was first introduced, but this model, where each variable has at most a possible colors and there are general constraints each involving at most b variables, is extensively exploited by Beigel and Eppstein's $\widetilde{O}(1.3829^n)$-time 3-coloring algorithm [BE00]. The optimization class Max (a, b)-CSP may first have been introduced in our own [SS03], to which we shall return in a moment.

Finding relatively fast exponential-time algorithms for NP-hard problems is a fairly broad field of endeavor. It includes Schöning's randomized algorithm for 3-Sat [Sch99], taking time $\widetilde{O}((4/3)^n)$ for an instance on n variables. This result is fairly distant from ours because it considers 3-variable clauses, it considers satisfaction rather than maximization, the algorithm is randomized, and it is parametrized by the number of variables n rather than constraints m.

Narrowing the scope to Max $(r,2)$-CSPs with time parametrized in m, we begin our history with an algorithm of Niedermeier and Rossmanith [NR00]: it solves Max Sat generally, and solves Max 2-Sat instances in time $\widetilde{O}(2^{0.347\cdots\, m})$. The Max 2-Sat result was improved by Hirsch to $\widetilde{O}(2^{m/4})$ [Hir00]. Gramm, Hirsch, Niedermeier and Rossmanith showed how to solve Max 2-Sat in time $\widetilde{O}(2^{m/5})$, and used a transformation from Max Cut into Max 2-Sat to allow Max Cut's solution in time $\widetilde{O}(2^{m/3})$ [GHNR]. Kulikov and Fedin show how to solve Max Cut in time $\widetilde{O}(2^{m/4})$ [KF02].

The present paper has its roots in a conference paper [SS03], where the present authors showed a polynomial-expected-time algorithm for solving Max Cut (and other Max $(2,2)$-CSPs) on *random* graphs up to the giant-component threshold. In a 2-page aside, that paper showed how to solve *arbitrary* Max $(2,2)$-CSP instances in time $\widetilde{O}(2^{m/5})$. (Our subsequent journal submission, [SS04], improves the random-graph result to *linear* expected time, with a uniform time bound extending through the so-called scaling window; since this was complicated enough, consideration of arbitrary instances was dropped.) That paper introduced the Max $(2,2)$-CSP model and the general approach we take here. The present paper expands on the telegraphic aside of [SS03] to give a more complete and correct proof of the same running-time bound, while extending it straightforwardly to $\widetilde{O}(r^{m/5})$ for Max $(r,2)$-CSP instances.

More profoundly, this paper improves the running time from $\widetilde{O}(r^{m/5})$ to $\widetilde{O}(r^{19m/100})$. This is somewhat remarkable because in the $\widetilde{O}(r^{m/5})$ result, the "m/5" is a bound on the number of "III-reductions", and the bound is tight: achieved by the complete graph K_5. In fact, on a collection of disjoint K_5's with m edges in all, the previous algorithm really would require time $\widetilde{O}(r^{m/5})$; the improvement comes from a new focus on connected components of the graph underlying a CSP instance. The improved result requires a somewhat more complicated algorithm (albeit still using just the same three simple reductions), along with more sophisticated analysis (we analyze the depth of a certain reduction tree rather than just the number of reductions). An apparent contradiction comes from that fact that on a single K_5 the two algorithms perform identically; since the $m/5$ bound is tight, how is $19m/100$ possible? The answer is that the $m/5$ is really replaced by $2+19m/100$, as per Lemma 6.

3. Reductions

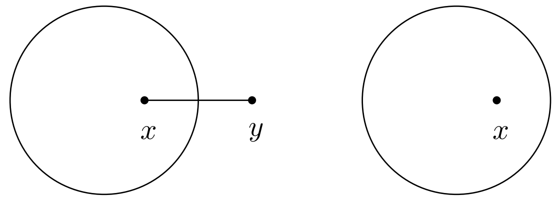
As with most of the works surveyed above, our algorithm is based on progressively reducing the instance to one with fewer vertices and edges until the instance becomes trivial.

Probably because we work in the general class Max $(r,2)$-CSP, rather than trying to stay within a smaller class such as Max 2-Sat or Max Cut, our reductions are simpler and fewer than is typical. For example, [GHNR] uses seven reduction rules; we have just three. The first two reductions each produce equivalent problems with fewer vertices, while the third produces a pair of problems, each with fewer vertices, one of which is equivalent to the original problem. We expand the previous notation (G,S) for an instance to (V,E,S), where $G=(V,E)$.

Reduction I: Let y be a vertex of degree 1, with neighbor x. Reducing (V,E,S) on y results in a new problem (V',E',S') with $V'=V\setminus y$ and $E'=E\setminus xy$. S' is the restriction of S to V' and E', except that for all colors $C\in[r]$ (and in total time $O(r^2)$) we set

$$s'_x(C)=s_x(C)+\max_{D\in[r]}\{s_{xy}(CD)+s_y(D)\}.$$

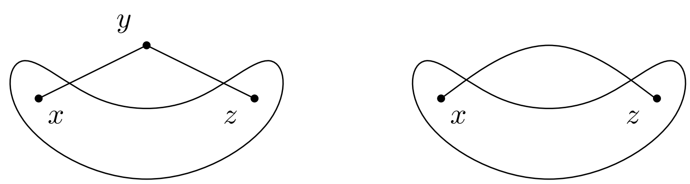
Note that any coloring ϕ' of V' can be extended to a coloring ϕ of V in r ways, depending on the color assigned to y; writing (ϕ', D) for the extension in which $\phi(x) = D$, the defining property of the reduction is that $S'(\phi') = \max_D S(\phi', D)$. In particular, $\max_{\phi'} S'(\phi') = \max_\phi S(\phi)$, and an optimal coloring ϕ' for the instance $\textsc{max}(V', E', S')$ can be extended to an optimal coloring ϕ for $\textsc{max}(V, E, S)$ in time $O(1)$.



Reduction II: Let y be a vertex of degree 2, with neighbors x and z. Reducing (V, E, S) on y results in a new problem (V', E', S') with $V' = V \setminus y$ and $E' = (E \setminus \{xy, yz\}) \cup \{xz\}$. S' is the restriction of S to V' and E', except that for $C, D \in [r]$ (and in total time $O(r^3)$) we set

$$s'_{xz}(CD) = s_{xz}(CD) + \max_{E \in [r]} \{s_{xy}(CE) + s_{yz}(ED) + s_y(E)\}$$

if there was already an edge xz, discarding the first term $s_{xz}(CD)$ if there was not. As in Reduction I, any coloring ϕ' of V' can be extended to V in r ways, according to the color D assigned to y, and the defining property of the reduction is that $S'(\phi') = \max_D S(\phi', D)$. In particular, $\max_{\phi'} S'(\phi') = \max_\phi S(\phi)$, and an optimal coloring ϕ' for $\textsc{max}(V', E', S')$ can be extended to an optimal coloring ϕ for $\textsc{max}(V, E, S)$ in time $O(1)$.



Reduction III: Let y be a vertex of degree 3 or higher. Where reductions I and II each had a single reduction of (V, E, S) to (V', E', S'), here we define r different reductions: for each color C there is a reduction (V, E, R) to (V^C, E^C, S^C) corresponding to assigning the color C to y. We define $V' = V \setminus y$, and E' as the restriction of E to $V \setminus y$. For $C, D, E \in \{R, B\}$, S^C is the restriction of S to $V \setminus y$, except that we set

$$(s^C)_0 = s_0 + s_y(C),$$

and, for every neighbor x of y,

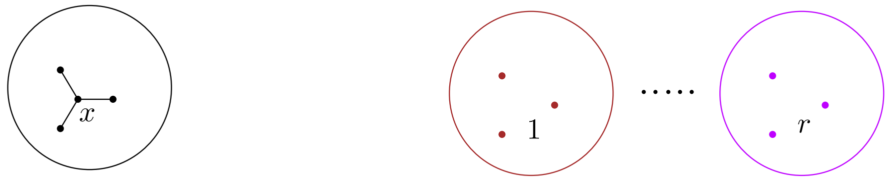
$$(s^C)_x(D) = s_x(C) + s_{xy}(CD).$$

Each reduction can be generated in time $O(rn)$.

As in the previous reductions, any coloring ϕ' of $V \setminus y$ can be extended to V in r ways, (ϕ', C) where color C is given to y, and now (this is different!) $S_C(\phi') = S(\phi', C)$. Furthermore,

$$\max_C \max_{\phi'} S_C(\phi') = \max_\phi S(\phi),$$

and an optimal coloring on the left *is* an optimal coloring on the right.



Reduction 0: We define one more "pseudo-reduction". If a vertex y has degree 0 (so it has no dyadic constraints), set (in time $O(r)$) $s_0 = s_0 + \max_{C \in [r]} s_y(C)$ and delete y from the instance entirely.

4. Algorithm A

Note that while a III-reduction produces r different sub-instances, all have the same underlying graph — the original graph with one vertex deleted. For both efficiency of implementation and ease of analysis, we define Algorithm A as running in two phases.

4.1. **First phase.** In the first phase we merely perform *graph* reductions, recording the sequence of reduction vertices. We reduce on a vertices in the following decreasing order of preference: a 0-reduction on a vertex of degree 0; a I-reduction on a vertex of degree 1; a II-reduction on a vertex of degree 2; or (still in order of preference) a III-reduction on a vertex of degree 5 or more, 4, or 3. The output of this phase is simply the sequence of vertices on which we reduced.

An efficient implementation of the first phase calls for maintaining "stacks" of vertices according to their degrees. Initially, in time m, each vertex is put into a stack depending on whether it has degree 0, 1, 2, 3, 5 or more, 4, or 3. We will maintain a dirty version of the stacks, with the property that a vertex may appear in several stacks, but is guaranteed at least to appear in the "correct" stack.

A complication comes from the fact that there may be many II-reductions, a II-reduction on vertex y adjacent to x and z introduces a pair of parallel edges between x and z if they were already adjacent, and unfortunately checking for this takes time $O(\min(\deg(x), \deg(y)))$, not $O(1)$. We get around this by maintaining a "dirty" version of the graph, which may have multiple edges, but has the property that collapsing multiple edges would not affect the "degree class" of any vertex, i.e., whether it has degree 0, 1, 2, 3, 4, or 5 or more. To "check" the degree of a vertex v, we iterate through its neighbors u; either add u to a list of v's distinct neighbors or, if u has been seen before, merge the new edge with the old one; and terminate when we run out of neighbors *or* when we have seen at least 5 distinct neighbors. Note that k such checks can be performed in time $O(k+m)$: the number of "distincts" is at most $5k$ and the number of "merges" is at most m.

In the first phase, then, we pop a vertex v off the first non-empty stack. We check v's degree, and if it does not match its stack, we discard v and pop the next vertex. If v's degree is correct, we reduce on v. This changes the degrees of v's neighbors, so we check the degree of each neighbor u; if the reduction changes u's degree class then we put u into the appropriate new stack (without removing it from its old stack, which would be too time-consuming).

Lemma 1. *Applying the first phase to a graph with n vertices and m edges produces a sequence of reductions following the stated preference order, and takes time and space $O(m+n)$.*

Proof. Correctness of the first phase is immediate from its definition.

Ignoring the degree checking, a graph reduction on vertex v takes time $O(\deg(v))$, so all the reductions together take time $O(m)$. Since the degree of a vertex can only decrease with time, each vertex is pushed onto each stack at most once. Correspondingly, there are at most $6n$ vertex pops; an immediate consequence is that the total space is $O(m+n)$. Each pop requires one degree check, and a reduction on v requires $\deg(v)$ degree checks, for a total of $O(n+m)$ degree checks, altogether taking time $O(m+n)$. □

4.2. **Second phase.** The second phase is recursive. In one "step" we follow the sequence of reductions defined by the first phase, stopping with a III-reduction (or when we run out of 0, I and II-reductions). Each 0, I and II-reduction transforms the instance, while the III-reduction defines a set of r sub-instances. Recursively, we solve these sub-instances one at a time, select the one with the best score, and "reverse" the 0, I and II-reductions to construct the solution to the original instance.

Lemma 2. *Applying the second phase to a Max $(r,2)$-CSP instance on n variables, with m constraints, and a first-phase reduction sequence with ℓ III-reductions, solves the instance in time $O(r^3 n r^\ell)$ and space $O(n\ell)$.*

Proof. Correctness of the algorithm is assured by the defining properties of the reductions. Normalizing to get rid of the $O(\cdot)$ notation, a time bound of $f(n,\ell) = 2r^3 n(1 + \cdots + r^d)$ is proved by induction on ℓ, the base case being straightforward and the inductive step following from showing that the time needed to break the problem down into r sub-problems, solve them, and return the best solution, $r^3 n + r f(n, \ell-1) + rn$, is $\leq f(n,\ell)$. The space bound comes from storing the best solution seen so far at each of the ℓ levels, and may also be proved inductively. □

4.3. **Recursion depth.** We now show that the value of ℓ in the previous two lemmas is at most $m/5$. This is the crux of the analysis.

Lemma 3. *A graph G with n vertices and m edges is reduced to an empty graph after no more than $m/5$ III-reductions.*

Proof. While the graph has maximum degree 5 or more, Algorithm A III-reduces only on such vertices, destroying at least 5 edges; any I- or II-reductions included in the same step only increase the number of edges destroyed. Thus, it suffices to prove the lemma for graphs with maximum degree 4 or less. Since the reductions never increase the degree of any vertex, the maximum degree will always remain at most 4.

In this paragraph, we give some intuition for the rest of the argument. Algorithm A III-reduces on vertices of degree 4 as long as possible, before III-reducing on vertices of degree 3, whose neighbors must then all be of degree 3 (vertices of degree 0, 1 or 2 would trigger a 0-, I- or II-reduction in preference to the III-reduction). Referring to Figure 1, we note that each III-reduction on a vertex of degree 3 destroys 6 edges if we were immediately to follow up with II-reductions on its neighbors; similarly reduction on a 4-vertex destroys at least 5 edges unless the 4-vertex has no degree-3 neighbor. The only problem comes from reductions on vertices of degree 4 all of whose neighbors are also of degree 4. The fact that the algorithm terminates with 0 degree-3 vertices is sufficient to limit the number of such reductions.

We proceed by considering the various types of reductions and their effect on the number of edges and the number of 3-vertices. The reductions are catalogued in Table 1. The first row, for example,

deg	#nbrs of deg				destroys					steps
	4	3	2	1	e	4	3	2	1	
4	4	0	0	0	4	5	-4	0	0	1
4	3	1	0	0	4	4	-2	-1	0	1
4	2	2	0	0	4	3	0	-2	0	1
4	1	3	0	0	4	2	2	-3	0	1
4	0	4	0	0	4	1	4	-4	0	1
3	0	3	0	0	3	0	4	-3	0	1
2					1	0	0	1	0	0
$\frac{1}{2}e$	1	0	0	0	$\frac{1}{2}$	1	-1	0	0	0
$\frac{1}{2}e$	0	1	0	0	$\frac{1}{2}$	0	1	-1	0	0
$\frac{1}{2}e$	0	0	1	0	$\frac{1}{2}$	0	0	1	-1	0
$\frac{1}{2}e$	0	0	0	1	$\frac{1}{2}$	0	0	0	1	0

TABLE 1. Tabulation of the effects of various reductions in Algorithm A.

shows that III-reducing on a vertex of degree 4 with no neighbors of degree 4 and (thus) 4 neighbors

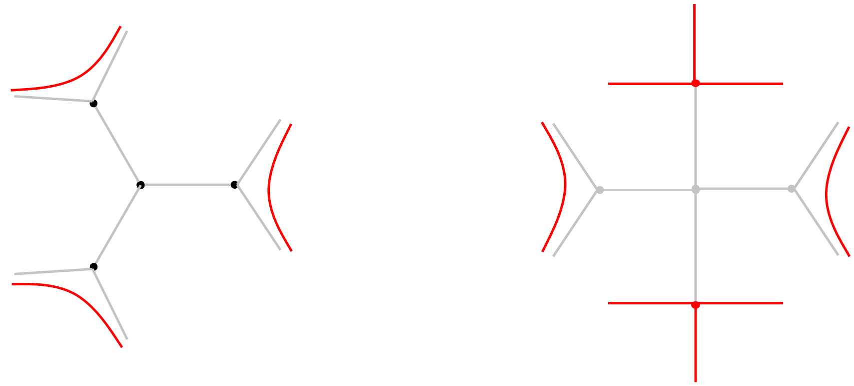


FIGURE 1. Left, a reduction on a 3-vertex with 3 3-neighbors, followed by II-reductions on those neighbors, destroys 6 edges and 4 3-vertices. (We will not actually force any particular II-reductions after a III-reduction.) Right, similarly, a reduction on a 4-vertex with k 3-neighbors ($k = 2$ here) destroys $4 + k$ edges and a net of $2k - 4$ 3-vertices (k 3-vertices are destroyed, but $4 - k$ 4-vertices become 3-vertices).

of degree 3, destroys 4 edges, and (changing the neighbors from degree 4 to 3) destroys 5 vertices of degree 4 and creates 4 vertices of degree 3. The remaining rows till the line similarly illustrate the other III-reductions. Below the line, II-reductions and I-reductions are decomposed into parts. As shown just below the line, a II-reduction, regardless of the degrees of the neighbors, first destroys 1 edge and 1 2-vertex, and counts as 0 steps (steps count only III-reductions). In the process, the II-reduction may create a parallel edge, which may (or may not) be deleted by Algorithm A. Since the effect of an edge deletion depends on the degrees of its neighbors, to minimize the number of cases we treat the edge deletion as two half-edge deletions, each destroying $\frac{1}{2}$ an edge, and whose effect depends on the degree of the half-edge's incident vertex. For example the next line shows deletion of a half-edge incident to a 4-vertex changing it a 3-vertex and destroying half an edge. I-reductions are also captured by the last four rows of the table.

The sequence of reductions reducing a graph to the empty graph can be parametrized by an 11-vector $\vec{n}$ giving the number of reductions (and partial reductions) described by the rows of the table, so for example its first element is the number of III-reductions on 4-vertices whose neighbors are also all 4-vertices. Since the reductions destroy all m edges, the dot product $\vec{n}$ with the table's column "destroys e" (call it $\vec{e}$) must be precisely m. Since all vertices of degree 4 are destroyed, the dot product of $\vec{n}$ with the column "destroys 4" (call it $\vec{d_4}$) must be ≥ 0, and the same goes for the "destroy" columns 3, 2 and 1. The number of III-reductions is the dot product of $\vec{n}$ with the "steps" column, $\vec{n} \cdot \vec{s}$. How large can it possibly be?

To find out, let us maximize $\vec{n} \cdot \vec{s}$ subject to the constraints that $\vec{n} \cdot \vec{e} = m$ and that $\vec{n} \cdot \vec{d_4}$, $\vec{n} \cdot \vec{d_3}$, $\vec{n} \cdot \vec{d_2}$ and $\vec{n} \cdot \vec{d_1}$ are all ≥ 0. Instead of maximizing over proper reduction collections $\vec{n}$ (which appears hard to characterize) we maximize over the larger class of non-negative real vectors $\vec{n}$, giving an upper bound on the proper maximum. Maximizing the linear function $\vec{n} \cdot \vec{s}$ of $\vec{n}$ subject to a set of linear constraints is simply solving a linear program (LP). (The LP's constraint matrix and objective function are the part of Table 1 right of the double line.) To avoid dealing with "m" in the LP, we set $\vec{n}' = \vec{n}/m$, and solve the LP with constraints $\vec{n}' \cdot \vec{e} = 1$ (and as before $\vec{n}' \cdot \vec{d_4} \geq 0$, etc) to maximize $\vec{n}' \cdot \mathit{steps}$; the quantity of interest is $\vec{n} \cdot \vec{s} = m\vec{n}' \cdot \vec{s}$. The "$\vec{n}'$" LP is a small linear program with all constant entries, and its maximum is precisely $1/5$, showing that the number of III-reduction steps is at most $m/5$.

That the LP has a maximum of 5 at most can be verified from the LP's dual solution of $\vec{y} = (0.20, 0, -0.05, -0.2, -0.1)$. It is easy to check that in each row, the "steps" value is less than or equal to the dot product of this dual vector with the "destroys" values. Writing D for the whole "destroys" constraint matrix, we thus have $\vec{n} \cdot \vec{s} \leq \vec{n} \cdot (D\vec{y}) = (\vec{n}D) \cdot \vec{y}$. But $\vec{n}D$ is the LP's constraints: its first element must be 1 and the remaining elements non-negative; the first element of $\vec{y}$ is 0.2 and its remaining elements are non-positive, so $\vec{n} \cdot \vec{s} \leq (\vec{n}D) \cdot \vec{y} \leq 0.2$. This establishes that the number of type-III reductions can be at most 1/5th the number of edges m, concluding the proof. □

Theorem 4. *A Max $(r, 2)$-CSP instance on n variables with m dyadic constraints can be solved in time $O(m + r^3 n r^{m/5})$ and space $O(mn)$.*

The proof is an immediate consequence of the three lemmas. We remark without proof that the space bound can be improved to $O(m + n)$ by replacing the second phase's depth-ℓ recursive implementation by a depth-first search on a tree with r^ℓ leaves.

The primal solution of the LP, which describes the worst case, uses 1 III-reduction of a 4-vertex with all 4-neighbors, 1 III-reduction on a 3-vertex, and 3 II-reductions. (We have renormalized for convenience; the actual values are 1/10th of these.) As it happens, this LP worst-case bound is achieved by the graph K_5, whose 10 edges are destroyed by two III-reductions and then some I- and II-reductions.

5. Algorithm B

Despite the fact that the "search depth" bound of $m/5$ is achievable, a slightly different perspective yields a better result. We will show that in any sequence of reductions, all but at most two "bad" reductions can be paired up with other redutions, yielding a depth bound of $2 + 19m/100$. (Or to put it another way around, $2 + m/(100/19)$, where $100/19 \approx 5.26$.)

The result will be achieved by Algorithm B. It is similar to Algorithm A but, first of all, is more particular about its splitting pivots. When performing a type-III reduction, Algorithm B selects a vertex in the following decreasing order of preference: a vertex of degree ≥ 6; a vertex of degree 5 with at least 1 neighbor of degree 3 or 4; a vertex of degree 5 whose neighbors all have degree 5; a vertex of degree 4 with at least 1 neighbor of degree 3; a vertex of degree 4 whose neighbors all have degree 4; a vertex of degree 3. When Algorithm B makes any such reduction with any degree-3 neighbor, it immediately follows up with II-reductions on all those neighbors, as was shown in Figure 1. (If k of these neighbors are linked to one another in a cycle, or a cycle interrupted by just one other vertex, we cannot actually perform II-reductions till the bitter end, but the effect is the same: we still destroy k edges and k 2-vertices).

As with Algorithm A, in Algorithm B a first phase performs just the graph reductions, starting a "step" with any number of I- and II-reductions and ending it with a III-reduction (or when the graph becomes empty). Note that (unlike I- and II-reductions), a III-reduction on a vertex v can split the component containing v into sub-components. We say that a III-reduction on v splits its component into $k(v)$ components, where we allow the cases $k(v) = 1$ (no split) and $k(v) = 0$ (the component becomes empty). This defines a "reduction forest": a node v in the forest corresponds to a graph reduction on v, and the $k(v)$ children of node v are the first vertices reduced upon in each component formed by the reduction.

This reduction forest plays the role in analyzing Algorithm B that the simple sequence of reductions did in analyzing Algorithm A. For simplicity of expression, we will henceforth assume that the original graph is connected, and speak of the reduction tree. Contracting out of the reduction tree the nodes corresponding to 0, I and II-reductions, leaving only those for III-reductions, defines a "splitting tree"; it is only used for the analysis and need not be constructed. The key parameter is the depth h of the splitting tree; this is the same as the maximum number of III-reduction nodes on any root-to-leaf path of the reduction tree. (A root-to-leaf path in the splitting tree defines a sequence of III-reductions on the graph, and for a path of depth h thus defines r^h possible *CSP* reductions.)

5.1. **Building the splitting tree.** We divide the first phase of the algorithm, which works with the underlying graph, into two stages. In the first stage, we build an ordered sequence of reductions; this works just as in Algorithm A and we do not discuss it further. Starting with this sequence of reductions, in the second stage we assemble them into the reduction tree.

Lemma 5. *A reduction tree on n vertices, whose splitting tree has depth h, can be constructed in time $O(hn)$.*

Proof. The proof is by analysis of an algorithm which reads backwards through the sequence of reductions, gluing trees together to make larger ones. The details are fairly straightforward and are omitted due to space constraints.

□

5.2. **Splitting tree depth.** Analogous to Lemma 3 characterizing algorithm A, the next lemma is the heart of the analysis of algorithm B.

Lemma 6. *For a graph G with m edges, the depth of the splitting tree is $h \leq 2 + (19/100)m$.*

Proof. As with Lemma 3, it suffices to prove this lemma for graphs with maximum degree at most 5.

Define a "bad" reduction to be one on a 5-vertex all of whose neighbors are also of degree 5 or on a 4-vertex all of whose neighbors are of degree 4. (These two reductions destroy 5 and 4 edges respectively, while, except for reducing on a 4-vertex with three 4-neighbors and one 3-neighbor, every other reduction, coupled with the II-reductions it enables, destroys at least 6 edges.) The analysis is aimed at controlling the number of these reductions.

For shorthand, we write reductions in terms of the degree of the vertex on which we are reducing followed by the numbers of neighbors of degrees 5, 4, and 3, so for example the "bad" reduction on a 5-vertex is written (5|500).

Within a component, a (5|500) reduction is performed only if there is no 5-vertex adjacent to a 3- or 4-vertex; this means the component *has* no 3- or 4-vertices, since otherwise a path from such a vertex to the 5-vertex includes an edge incident on a 5-vertex and a 3- or 4-vertex. We track the component containing one vertex, say vertex 1, as it is reduced. If the component necessitates a bad 5-reduction, one of four things must be true:

(1) This is the first degree-5 reduction in this branch of the splitting tree.
(2) The previous reduction was on a (5|005) vertex, and left no vertices of degree 3 or 4.
(3) The previous reduction was on a 5-vertex and produced vertices of degree 3 or 4 in this component, but they were destroyed by I- and II-reductions.
(4) The previous reduction was on a 5-vertex and produced vertices of degree 3 or 4, but split them all off into other components.

As in the proof of Lemma 3, for each type of reduction we will count: its contribution to the depth (normally 1 or 0, but we also introduce pseudo-reductions counting as 2); the number of edges it destroys; and the number of vertices of degree 4, 3, 2, and 1 it destroys. Table 2 shows this tabulation. Recall that in Algorithm B we immediately follow each III-reduction with II-reductions on each 2-vertex it produces, so for example in row 1 a (5|005) reduction destroys a total of 10 edges and 5 3-vertices; it also creates 5 2-vertices but immediately reduces them away.

The table's boldfaced rows and the new column "forces" require explanation. We eliminate the normal (5|500) reduction from the table, replacing it with versions corresponding to the cases above.

Case (1) above can occur only once. Weakening this constraint, we will allow it to occur any number of times, but we will count its depth contribution as 0, and add 1 to the depth at the end. For this reason, the first bold row in Table 2 has depth 0 not 1.

In case (2) we may pair the bad (5|500) reduction with its preceding (5|005) reduction. This defines a new "pair" pseudo-reduction shown as the second bold row of the table: it counts for 2 steps, destroys 15 edges, etc. (Other, non-paired (5|005) reductions are still allowed as before.)

In case (3) we wish to similarly pair the (5|500) reduction with a I- or II-reduction, but we cannot say specifically with which sort. The "forces" column of Table 2, constrained to be non-negative (like the "destroys" columns it follows), constrains each (5|500) reduction for this case to be accompanied by at least one I- or II-reduction (and so two half-edge reductions) of any sort.

In case (4), the (5|500) reduction produces a non-empty side component destroyed with the usual reductions but adding depth 0 to the component of interest. These reductions are a nonnegative combination of half-edge reductions, so we can pair the (5|500) reduction with any two of these, as in case (3). Thus case (4) does not require any additions to the table.

Together, the four cases mean that we were able to eliminate (5|500) reductions as a possibility, replacing them with less harmful possibilities represented by the first three bold rows in the table.

We may reason identically for bad reductions on 4-vertices, contributing the other three bold rows to the table.

Note also that type-0 and I reductions, as well as the combination of parallel edges, can be written as a nonnegative combination of half-edge reductions.

deg	#nbrs of deg					destroys					forces	depth
	5	4	3	2	1	e	4	3	2	1		
5	0	0	5	0	0	10	0	5	0	0	0	1
5	0	1	4	0	0	9	1	3	0	0	0	1
5	0	2	3	0	0	8	2	1	0	0	0	1
5	0	3	2	0	0	7	3	−1	0	0	0	1
5	0	4	1	0	0	6	4	−3	0	0	0	1
5	0	5	0	0	0	5	5	−5	0	0	0	1
5	1	0	4	0	0	9	−1	4	0	0	0	1
5	1	1	3	0	0	8	0	2	0	0	0	1
5	1	2	2	0	0	7	1	0	0	0	0	1
5	1	3	1	0	0	6	2	−2	0	0	0	1
5	1	4	0	0	0	5	3	−4	0	0	0	1
5	2	0	3	0	0	8	−2	3	0	0	0	1
5	2	1	2	0	0	7	−1	1	0	0	0	1
5	2	2	1	0	0	6	0	−1	0	0	0	1
5	2	3	0	0	0	5	1	−3	0	0	0	1
5	3	0	2	0	0	7	−3	2	0	0	0	1
5	3	1	1	0	0	6	−2	0	0	0	0	1
5	3	2	0	0	0	5	−1	−2	0	0	0	1
5	4	0	1	0	0	6	−4	1	0	0	0	1
5	4	1	0	0	0	5	−3	−1	0	0	0	1
5	**5**	**0**	**0**	**0**	**0**	**5**	**−5**	**0**	**0**	**0**	**0**	**0**
5 + 5	**5**	**0**	**5**	**0**	**0**	**15**	**−5**	**5**	**0**	**0**	**0**	**2**
5	**5**	**0**	**0**	**0**	**0**	**5**	**−5**	**0**	**0**	**0**	**−1**	**1**
4	0	0	4	0	0	8	1	4	0	0	0	1
4	0	1	3	0	0	7	2	2	0	0	0	1
4	0	2	2	0	0	6	3	0	0	0	0	1
4	0	3	1	0	0	5	4	−2	0	0	0	1
4	**0**	**4**	**0**	**0**	**0**	**4**	**5**	**−4**	**0**	**0**	**0**	**0**
4 + 4	**0**	**4**	**4**	**0**	**0**	**12**	**6**	**0**	**0**	**0**	**0**	**2**
4	**0**	**4**	**0**	**0**	**0**	**4**	**5**	**−4**	**0**	**0**	**−1**	**1**
3	0	0	3	0	0	6	0	4	0	0	0	1
2	0	0	0	0	0	1	0	0	1	0	1	0
$\frac{1}{2}e$	1	0	0	0	0	$\frac{1}{2}$	−1	0	0	0	$\frac{1}{2}$	0
$\frac{1}{2}e$	0	1	0	0	0	$\frac{1}{2}$	1	−1	0	0	$\frac{1}{2}$	0
$\frac{1}{2}e$	0	0	1	0	0	$\frac{1}{2}$	0	1	−1	0	$\frac{1}{2}$	0
$\frac{1}{2}e$	0	0	0	1	0	$\frac{1}{2}$	0	0	1	−1	$\frac{1}{2}$	0
$\frac{1}{2}e$	0	0	0	0	1	$\frac{1}{2}$	0	0	0	1	$\frac{1}{2}$	0

TABLE 2. Tabulation of the effects of various reductions in Algorithm B.

In analyzing a branch of the splitting tree, let vector $\vec{n}$ count the number of reductions of each type, as in the proof of Lemma 3. As before, the dot product of $\vec{n}$ with the "destroys e" column is constrained to be 1 (we will skip the version where it is m and go straight to the normalized form), its dot products with the other "destroys" columns must be non-negative, and the question is how large its dot product x with the "depth" column can possibly be. Unnormalizing, the splitting-tree depth of vertex 1 as we counted it is at most xm, and the true depth (accounting for the possible case (1) occurrences for 4- and 5-vertices) is at most $2 + xm$.

As before, x is found by solving the LP: it is 19/100. The dual solution (0.190, −0.005, −0.035, −0.190, −0.095) confirms this as the maximum possible, again just as in the proof of Lemma 3.

This concludes the proof. □

We remark that the maximum is achieved by a weight vector with just four nonzero elements, putting relative weights of 8, 6, 5 and 21 on the reductions (5|410), (4|031), (3|003) and (2|000). That is, the proof worked by essentially eliminating bad reductions of types (5|500) and (4|040) (which destroy only 5 and 4 edges respectively, in conjunction with the II-reductions they enable), and the bound produced uses the second-worst reductions of types (5|410) and (4|031) (each destroying 5 edges, with the II-reductions), which it is forced to balance out with favorable III-reductions of type (3|003) and II-reductions.

6. Algorithm

We begin with the first phase described above, and use the splitting tree to solve the problem recursively. At each step, to solve an instance, we do a series of I- and II-reductions specified by the tree (time $O(r^3 n)$), concluding with all r versions of the single specified III-reduction (time $O(r^3 n)$ for each). For each version we recursively solve the subinstance formed, solving each of its components separately. If the version's cost is better than that of previous versions, we record its solution. Finally, we extend the solution back to the original instance (time $O(r^3 n)$).

Theorem 7. *Algorithm B solves a Max $(r, 2)$-CSP instance (G, S), where G has n vertices, m edges, and reduction depth h, in time $O\left(m + nr^{h+3}\right)$ and space $O(mn)$.*

Proof. The proof, by iteration on h, is omitted due to space constraints. The only embellishment on the corresponding proof for Algorithm A is the observation that the time $\sum f(n_i, h_i)$ for a collection of components with $\sum n_i = n$ has its worst case when there is just a single component. □

From Lemma 6, we immediately get the following corollary.

Corollary 8. *Algorithm B solves a Max $(r, 2)$-CSP instance (G, S), where G has n vertices and m edges, in time $O\left(nr^{5+19m/100}\right)$ and space $O(mn)$.*

References

[BE00] Richard Beigel and David Eppstein, *3-coloring in time $O(1.3289^n)$*, arXiv:cs.DS/0006046v1, 2000.

[Cre95] N. Creignou, *A dichotomy theorem for maximum generalized satisfiability problems*, J. Comput. System Sci. **51** (1995), no. 3, 511–522.

[GHNR] Jens Gramm, Edward A. Hirsch, Rolf Niedermeier, and Peter Rossmanith, *New worst-case upper bounds for MAX-2-SAT with an application to MAX-CUT*, Discrete Applied Mathematics, In Press.

[Hir00] Edward A. Hirsch, *A new algorithm for MAX-2-SAT*, STACS 2000 (Lille), Lecture Notes in Comput. Sci., vol. 1770, Springer, Berlin, 2000, pp. 65–73.

[KF02] A. S. Kulikov and S. S. Fedin, *Solution of the maximum cut problem in time $2^{|E|/4}$*, Zap. Nauchn. Sem. S.-Peterburg. Otdel. Mat. Inst. Steklov. (POMI) **293** (2002), no. Teor. Slozhn. Vychisl. 7, 129–138, 183.

[KSTW01] S. Khanna, M. Sudan, L. Trevisan, and D.P. Williamson, *The approximability of constraint satisfaction problems*, SIAM J. Comput. **30** (2001), no. 6, 1863–1920.

[Mar04] Dániel Marx, *Parameterized complexity of constraint satisfaction problems*, Proceedings of the 19th IEEE Annual Conference on Computational Complexity (CCC'04), 2004, pp. 139–149.

[NR00] Rolf Niedermeier and Peter Rossmanith, *New upper bounds for maximum satisfiability*, J. Algorithms **36** (2000), no. 1, 63–88.

[Sch99] U. Schöning, *A probabilistic algorithm for k-SAT and constraint satisfaction problems*, Proceedings of the 40th IEEE Symposium on Foundations of Computer Science (FOCS), October 1999, pp. 410–414.

[SS03] Alexander D. Scott and Gregory B. Sorkin, *Faster algorithms for MAX CUT and MAX CSP, with polynomial expected time for sparse instances*, Proceedings of Random 2003 (Baltimore, MD, 2003), August 2003.

[SS04] ———, *Solving sparse semi-random instances of MAX CUT and MAX CSP in linear expected time*, submitted for publication, 29pp., 2004.

(Alexander D. Scott) Department of Mathematics, University College London, London WC1E 6BT, UK

E-mail address: scott@math.ucl.ac.uk

(Gregory B. Sorkin) Department of Mathematical Sciences, IBM T.J. Watson Research Center, Yorktown Heights NY 10598, USA

E-mail address: sorkin@watson.ibm.com