CSC2420 Spring 2015: Lecture 4

Allan Borodin

January 29, 2015

Announcements and todays agenda

- Thank you for submitting assignments. Please note what I have now appended to start of the assignment. Will post some new questions for next assignment over the weekend.
- Tomorrow Friday, Jan 30, 11 AM, GB221. Talk by Aaron Potechin on "hidden clique problem".
- We will have a number of "theoretical CS" talks this term, some at this time slot and some at Tuesdays and Thursdays at 11.
- Todays agenda
  1. We begin with a natural DP for the knapsack problem.
  2. We then consider an alternative DP which with the use of scaling allows us to derive a FPTAS for the knapsack problem.
  3. DP based approach for deriving a PTAS (poly in n and m, and exponential in $\frac{1}{\epsilon}$) for makespan on identical machines.
  4. DP algorithms that make more than one recursive call per decision choice.
  5. Mention some attempts to formally model DP algorithms.
  6. Introducing local search

A pseudo polynomial time “natural DP” for knapsack

Consider an instance of the (NP=hard) knapsack problem; that is, we are given item $\{(v_k, s_k) | 1 \leq k \leq n\}$ and a knapsack capacity C. Following along the lines of the WISP DP, the following is a reasonably natural approach to obtain a “pseudo polynomal space and time” DP:

- For $1 \leq i \leq n$ and $0 \leq c \leq C$, define $V[i, c]$ to be the value of an optimum solution using items $\mathcal{I}_i \subseteq \{I_1, \ldots, I_i\}$ and satisfying the size constraint that $\sum_{I_j \in \mathcal{I}_i} s_j \leq c$.
- A corresponding recursive DP is as follows:
    1. $V[0, c] = 0$ for all c
    2. For $i > 0$, $V[i, c] = \max\{A, B\}$ where
        - ⋆ $A = V[i-1, c]$
        - ⋆ $B = v_i + V[i-1, c-s_i]$ if $s_i \leq c$ and $V[i-1, c]$ otherwise.

    Note: easy to make mistakes so again have to verify that this recursive definition is correct.
- The space and time complexity is $O(nC)$ which is pseudo polynomial in the sense that C can be exponential in the encoding of the input.

Dynamic programming and scaling

We have previously seen that with some use of brute force and greediness, we can achieve PTAS algorithms for the identical machines makespan (polynomial in the number n of jobs but exponential in the number m of machines) and knapsack problems. We now consider how to achieve an FPTAS for the knapsack problem and then how dynamic programming (DP) can be used to acheive a PTAS for the makespan problem which is polynomial in m and n.
To achieve these improved bounds we will combine dynamic programming with the idea of scaling inputs.

An FPTAS for the knapsack problem

Let the input items be $I_1, \ldots, I_n$ (in any order) with $I_k = (v_k, s_k)$. The idea for the knapsack FPTAS begins with a different "pseudo polynomial" time DP for the problem, namely an algorithm that is polynomial in the numeric values v_j (or $V = \sum v_j$) (but not polynomial in the encoded length $|v_j|$ of the input values as in what I called the natural DP for the knapsack problem).

Define $S[j, v] =$ the minimum size s needed to achieve a value of at least v using only inputs $I_1, \ldots I_j$; this is defined to ∞ if there is no way to achieve this profit using only these inputs.

This again is the essense of DP algorithms; namely, defining an approriate generalization of the problem such that

1. the desired result can be easily obtained from ths array $S[\ ,\ ]$
2. each entry of the array can be easily computed given "previous entries"

How to compute the array $S[j, v]$ and why is this sufficient

- The value of an optimal solution is $max\{v : S[n, v] \leq C\}$.
- We have the following equivalent recursive definition that shows how to compute the entries of $S[j, v]$ for $0 \leq j \leq n$ and $v \leq \sum_{j=1}^{n} v_j$.
  1. Basis: $S[0, v] = \infty$ for all $v > 0$ and $S[j, 0] = 0$ for all $j \geq 0$.
  2. Induction: $S[j, v] = \min\{A, B\}$ where $A = S[j-1, v]$ and $B = S[j-1, \max\{v - v_j, 0\}] + s_j$
- The running time is $O(nV)$ where $V = \sum_{j=1}^{n} v_j$.
- Finally, to obtain the FPTAS the idea (due to Ibarra and Kim [1975]) is simply that the high order bits/digits of the item values give a good approximation to the true value of any solution and scaling these values down (or up) to the high order bits does not change feasibility.

The better PTAS for makespan

- We can think of m as being a parameter of the input instance and now we want an algorithm whose run time is poly in m, n for any fixed $\epsilon = 1/s$.
- The algorithm's run time is exponential in $\frac{1}{\epsilon^2}$.
- We will need a combination of paradigms and techniques to achieve this PTAS; namely, DP and scaling (but less obvious than for the knapsack scaling) and binary search.

The high level idea of the makespan PTAS

- Let T be a candidate for an achievable makespan value. We csan assume $T \geq max_j p + j$.
- Our goal is to either show that the makespan is greater than T or provide a solution with makespan at most $(1 + \epsilon)T$.
- Depending on T and the ϵ required, we will scale down "large" (i.e. if $p_i \geq T/s = T \cdot \epsilon$) to the largest multiple of T/s^2 so that there are only $d = s^2$ values for scaled sizes of the large jobs.
- When there are only a fixed number d of job sizes, we can use DP to test (and find) in time $O(n^{2d})$ if there is a solution that achieves makespan T. Accounting for the scaling down will only increase the makespan for the large jobs to be at most $(1 + \epsilon)T$.
- If there is a solution for the scaled large jobs then small jobs can either be greedily scheduled on some machine with current makespan T or makespan T is not possible.
- We use binary search to find a good T. That is, if we were not successful for a given T, then we double our estimate for T; if successful we halve the estimate.

The optimal DP for a fixed number of job values

- Let $z_1, \ldots, z_d$ be the d different job sizes and let $n = \sum n_i$ be the total number of jobs with n_i being the number of jobs of szie z_i.
- $M[x_1, \ldots, x_d] =$ the minimum number of machines needed to schedule x_i jobs having size z_i within makespan T.
- Then n jobs can be scheduled within makespan T iff $M[n_1, \ldots, n_d]$ is at most m.

Computing $M[x_1, \ldots, x_d]$

- Clearly $M[0, \ldots, 0] = 0$ for the base case.
- Let $V = \{(v_1, , v_d) | \sum_i v_i z_i \leq T\}$ be the set of configurations that can complete on one machine within makespan T; that is, scheduling v_i jobs with size z_i on one machine does not exceed the target makespan T.
- $M[x_1, \ldots, x_d] = 1 + \min_{(v_1,\ldots,v_d) \in V : v_i \leq x_i} M[x_1 - v_1, \ldots, x_d - v_d]$
- There are at most n^d array elements and each entry uses approximately n^d time to compute (given previous entries) so that the total time is $O(n^{2d})$.
- Must any (say DP) algorithm be exponential in d?

Large jobs and scaling (not worrying about any integrality issues)

- A job is large if $p_i \geq T/s = T \cdot \epsilon$
- Scale down large jobs to have size $\tilde{p}_i =$ largest multiple of $T/(s^2)$
- $p_i - \tilde{p}_i \leq T/(s^2)$
- There are at most $d = s^2$ job sizes $\tilde{p}$
- There can be at most s large jobs on any machine not exceeding target makespan T.

Accounting for the scaling down and taking care of the small jobs

- If we cannot schedule the large jobs with makepsan T, then clearly T is too small. Otherwise to account for the scaling down, each large job was scaled down by at most $\frac{1}{s^2}T$ and there are at most s jobs on any machine so that the large jobs will not exceed makespan $(1+1/s)T = (1+\epsilon)$.
- We now wish to add in the small jobs with sizes less than T/s. We try to add small jobs to some machine with current makespan T (before adding the small job). If this is not possible, then makespan T is not possible since then every machine will be exceeding makespan T and thus the total load $\sum_{j=1}^{n} p_j > mT$ proving that makespan T is not possible. The addition of any small job can only increase the makespan by at most $T\epsilon$.
- If we can add in all the small jobs then to account for the scaling we note that each of the at most s large jobs were scaled down by at at most $T/(s^2)$ so this only increases the makespan to $(1+1/s)T$.

DP using many simultaneous recursive calls

- The previous DP examples made a choice (with respect to a possible opimal solution) and then made one recursive call for each of the possible choices.
- Many DP algorithms will do a number of simultaneous recursive calls for a given choice.
- A simple example is a dynamic program for computing the all pairs least cost paths in a directed graph $G = (V, E)$ based on the following semantic array. Let $C[k, u, v]$, for $0 \leq k \leq \lceil \log_2 V \rceil$ and $u \neq v$, be the least cost path π from u to v with π having path length at most 2^k.
- Another example is a simple DP for the WMIS problem on trees (and then of course for forests). . Namely, root the tree $T = (V, E)$ at an arbitrary node r. Then for every node $u \in V$, we want to compute the optimal WMIS on the subtree rooted at u.
  The choice to be made is whether or not to include u in the optimal subtree rooted at u.

Graphs of bounded treewidth

- Given the simplicity of trees, it is interesting to consider what graphs are "close to being trees" and then would share (to some extent) the nice properties of trees.
- This has led to the concept of the treewidth of a graph. Like degree k graphs, k-claw free graphs, inductive k independent graphs, etc. every graph $G = (V, E)$ has treewdith at most $|V| - 1$.
- The definition will insure that trees (and forests) have treewidth 1 and that the complete graph will have treewidth $|V| - 1$.
- The definition will also insure that graphs of small treewidth can be decomposed (similar to trees) into independent components so that dynamic programming can be applied.
- Claim: "Large networks in the real world often have very small treewidth". See Chapter 10 or Kleinberg and Tardos as well as many other sources concerning bounded treewidth and related concepts such as clique-width and path-width.

Graphs of bounded treewidth continued

- A tree decomposition of $G = (V, E)$ is a tree $T = (I, F)$ where each node $i \in I$ represents a bag X_i such that
  1. $\bigcup_{i \in I} X_i = V$
  2. $\forall (u, v) \in E$, there is some $i \in I$ such that $\{u, v\} \in X_i$
  3. $\forall v \in V$, the set $I_v = \{i \in I | v \in X_i\}$ is a connected subtree T' of T; that is, there is a unique path in T between any two nodes in I_v.
- The treewidth of a tree decomposition is defined as $\max_{i \in I} |X_i| - 1$. And the treewidth of G is the minimum treewidth over all tree decompositions of G

Complexity of WMIS on graphs of bounded treewidth

Theorem: There is an $O(k4^{k+1}|V|)$ time optimal DP algorithm for WMIS applied to graphs of treewidth k

Very brief sketch of WMIS treewidth result

- Given a graph $G = (V, E)$ and k, it is possible in time exponential in k^3 and linear in $|E|$ to determine if G has treewidth k and if so to construct a tree decomposition $T = (I, F)$ of width k with $|I| \leq |V|$.
- Having rooted the tree decomposition, the idea of the recursion at any tree node i (with say bag X_i) is to consider each of the at most 2^{k+1} possible independent subsets of X_i.
- This is an example of what is called paramterized complexity; that is, finding an appropriate parameter k of (say) an NP-hard problem and showing that it ican be computed (usually one means optimally computed) in time $f(k) \cdot poly(|w|)$ where w is an encoding of the input.
- Graphs of treewidth k are a subset of graphs having inductive degree k and hence a subset of inductive k independent graphs.

The pBT model: An attempt to model some simple DP (and backtracking) algorithms

- In an extension of the priority framework, Alekhnovich et al [2011] consider the pBT model (for *prioritized branching tree* or *prioritized backtrack*) where upon considering an input item, the algorithm can branch on different possible decisions. The algorithm can also terminate branches whenever it wishes.
- For search problems, the goal is to have a branch that produces a feasible solution if one exists, and for optimization problems the solution having the best approximation ratio is chosen. (Aside: it would have been better to just have non deterministic branching instead of branching on decisions.)
- The complexity of such an algorithm is size (or maximum "width") or the time in say a depth first search of the pBT tree.
- The pBT model can capture DPs where the implicit induction is on the number of items as in the interval scheduling and knapsack DPs.

Some pBT results

- In the pBT model, we can optimally solve one machine interval scheduling with fixed order width n (the number of intervals) using the standard DP, and $\Omega(n)$ width is required for any adaptive order pBT that optimally solves the problem. Furthermore for any fixed m, the width required for optimally solving the m machine problem is $\Omega(n^m)$ which can be achieved again using DP.
- In the pBT model, we have the following result for the knapsack problem: We can obtain a $(1+\epsilon)$-approximation with width $O(\frac{1}{\epsilon^2})$ (based on the Lawler adaption of the Ibarra and Kim FPTAS) and any adaptive order pBT algorithm that achieves a $(1+\epsilon)$-approximation requires width $\Omega(\frac{1}{\epsilon^{3.17}})$ and width $\binom{n/2}{n/4} = \Omega(2^{n/2}/\sqrt{n})$ for optimality. The lower bounds hold even for the Subset-Sum problem.
- Chvátal [1980] established an exponential time bound for the knapsack problem with respect to a model that captures a style of branch and bound algorithms. Similar attempts to formalize some branch and bound methods were obtained by Chvátal [1977] for the MIS problem and by McDiarmid [1979] for the graph colouring.

The pBP model: a more ambitious DP model

- The pBP (for *prioritized branching program*) model extends the pBT model by combining merging with branching so that the underlying structure of a pBP algorithm is a rooted DAG and not a rooted tree.
- The semantics are a little involved but the idea is meant to better capture memoization which is central to DP algorithms (in the sense of distinguishing them from divide and conquer algorithms).
- In the pBP model, there is an optimal $O(n^3)$ width algorithm for solving the shortest path problem when there are negative weights but not negative cycles. If the input graph has negative cycles the algorithm will output an arbitrary set of edges. Here the input items are .... In contrast, any pBT algorithm would require exponential width to solve the promise version of the shortest path problem on some instance (which could be a graph with negative cycles).
- For the bipartite matching problem where the input items are edges, any pBP algorithm requires exponential width. A challenge is to prove such a result when the input items are vertices.
- There is an optimal max flow algorithm for bipartite matching.

Combinatorial DP Programs

Finally we mention another model by Bompadre [2012] that also captures a limited class of DP algorithms. This model is incomparable with the pBT and pBP models. There are a number of positive and negative results derived by Bompadre.

Local Search: the other conceptually simplest approach

We now begin a discussion of the other (than greedy) conceptually simplest search/optimization algorithm, namely local search.

The vanilla local search paradigm

```
"Initialize" $S$
While there is a "better" solution $S'$ in "$Nbhd(S)$"
$S := S'$
End While
```

If and when the algorithm terminates, the algorithm has computed a *local optimum*. To make this a precise algorithmic model, we have to say:

1. How are we allowed to choose an initial solution?
2. What consititutes a reasonable definition of a local neighbourhood $Nbhd(S)$?
3. What do we mean by "better"?

Answering these questions (especially as to defining local neighbourhood) will often be quite problem specific.

Towards a precise definition for local search

- We clearly want the initial solution to be efficiently computed and to be precise we can (for example) say that the initial solution is a random solution, or a greedy solution or adversarially chosen. Of course, in practice we can use any efficiently computed solution.
- We want the local neighbourhood $Nbhd(S)$ to be such that we can efficiently search for a "better" solution (if one exists).
  1. In many problems, a solution S is a subset of the input items or equivalently a $\{0,1\}$ vector, and in this case we often define the $Nbhd(S) = \{S' | d_H(S, S') \leq k\}$ for some small k where $d_H(S, S')$ is the Hamming distance.
  2. More generally whenever a solution is a vector over a small domain D, we can use Hamming distance to define a local neighbourhood. Hamming distance k implies that $Nbhd(S)$ can be searched in at most time $|D|^k$.
  3. We can view Ford Fulkerson flow algorithms (to be discussed) as local search algorithms where the (possibly exponential size but efficiently searchable) neighbourhood of a flow solution S are flows obtained by adding an augmenting path flow.

What does "better" solution mean? Oblivious and non-oblivious local search

- For a search problem, we would generally have a non-feasible initial solution and "better" can then mean "closer" to being feasible.
- For an optimization problem it usually means being an improved solution which respect to the given objective. For reasons I cannot understand, this has been termed *oblivious* local search.
- For some applications, it turns out that rather than searching to improve the given objective function, we search for a solution in the local neighbourhood that improves a related potential function and this has been termed non-oblivious local search.
- In searching for an improved solution, we may want an arbitrary improved solution, a random improved solution, or the best improved solution in the local neighbourhood.
- For efficiency we may insist that there is a "sufficiently better" improvement rather than just better.

The weighted max cut problem

- Our first local search algorithm will be for the (weighted) max cut problem defined as follows:

The (weighted) max-cut problem

- Given a (undirrected) graph $G = (V, E)$ and in the weighted case the edges have non negative weights.
- **Goal:** Find a partition (A, B) of V so as to maximize the size (or weight) of the cut $E' = \{(u, v) | u \in A, v \in B, (u, v) \in E\}$.

- We can think of the partition as a characteristic vector χ in $\{0, 1\}^n$ where $n = |V|$. Namely, say $\chi_i = 1$ iff $v_i \in A$.
- Let $N_d(A, B) = \{(A', B') |$ the characteristic vector of (A') is Hamming distance at most d from $(A)\}$
- So what is a natural local search algorithm for (weighted) max cut?

A natural oblivious local search for weighted max cut

Single move local search for weighted max cut

```
Initialize $(A, B)$ arbitrarily
WHILE there is a better partition $(A', B') \in N_1(A, B)$
    $(A, B) := (A', B')$
END WHILE
```

- This single move local search algorithm is a $\frac{1}{2}$ approximation; that is, when the algorithm terminates, the value of the computed local optimum will be at least half of the (global) optimum value.
- In fact, if W is the sum of all edge weights, then $w(A, B) \geq \frac{1}{2}W$.
- This kind of ratio is sometimes called the absolute ratio or totality ratio and the approximation ratio must be at least this good.
- The worst case (over all instances and all local optima) of a local optimum to a global optimum is called the locality gap.
- It may be possible to obtain a better approximation ratio than the locality gap (e.g. by a judicious choice of the initial solution) but the approximation ratio is at least as good as the locaity gap.

Proof of totality gap for the max cut single move local search

- The proof is based on the following property of any local optimum:

$$\sum_{v \in A} w(u,v) \leq \sum_{v \in B} w(u,v) \text{ for every } u \in A$$

- Summing over all $u \in A$, we have:

$$2 \sum_{u,v \in A} w(u,v) \leq \sum_{u \in A, v \in B} w(u,v) = w(A,B)$$

- Repeating the argument for B we have:

$$2 \sum_{u,v \in B} w(u,v) \leq \sum_{u \in A, v \in B} w(u,v) = w(A,B)$$

- Adding these two inequalites and dividing by 2, we get:

$$\sum_{u,v \in A} w(u,v) + \sum_{u,v \in B} w(u,v) \leq w(A,B)$$

- Adding $w(A,B)$ to both sides we get the desired $W \leq 2w(A,B)$.

The complexity of the single move local search

- **Claim:** The local search algorithm terminates on every input instance.
  - Why?

- Although it terminates, the algorithm could run for exponentially many steps.

- It seems to be an open problem if one can find a local optimum in polynomial time.

- However, we can achieve a ratio as close to the state $\frac{1}{2}$ totality ratio by only continuing when we find a solution (A', B') in the local neighborhood which is "sufficiently better". Namely, we want

$$w(A', B') \geq (1 + \epsilon) w(A, B) \text{ for any } \epsilon > 0$$

- This results in a totality ratio $\frac{1}{2(1+\epsilon)}$ with the number of iterations bounded by $\frac{n}{\epsilon} \log W$.

Final comment on this local search algorithm

- It is not hard to find an instance where the single move local search approximation ratio is $\frac{1}{2}$.

- Furthermore, for any constant d, using the local Hamming neighbourhood $N_d(A, B)$
  still results in an approximation ratio that is essentially $\frac{1}{2}$.
  And this remains the case even for $d = o(n)$.

- It is an open problem as to what is the best "combinatorial algorithm" that one can achieve for max cut.

- There is
  a vector program relaxation of a quadratic program that leads to a .878 approximation ratio.

Exact Max-k-Sat

- **Given:** An exact k-CNF formula
$$F = C_1 \wedge C_2 \wedge \ldots \wedge C_m,$$
where $C_i = (\ell_i^1 \vee \ell_i^2 \ldots \vee \ell_i^k)$ and $\ell_i^j \in \{x_k, \bar{x}_k \,|\, 1 \leq k \leq n\}$.

- In the weighted version, each C_i has a weight w_i.

- **Goal:** Find a truth assignment τ so as to maximize
$$W(\tau) = w(F \,|\, \tau),$$
the weighted sum of satisfied clauses w.r.t the truth assignment τ.

- It is NP hard to achieve an approximation better than $\frac{7}{8}$.

The natural oblivious local search

- A natural oblivious local search algorithm uses a Hamming distance d neighbourhood:
  $N_d(\tau) = \{\tau' : \tau$ and τ' differ on at most d variables $\}$

Oblivious local search for Exact Max-k-Sat

```
Choose any initial truth assignment $\tau$
WHILE there exists $\hat{\tau} \in N_d(\tau)$ such that $W(\hat{\tau}) > W(\tau)$
    $\tau := \hat{\tau}$
END WHILE
```

How good is this algorithm?

- Note: Following the standard convention for Max-Sat, I am using approximation ratios < 1.
- It can be shown that for $d = 1$, the approximation ratio for Exact-Max-2-Sat is $\frac{2}{3}$.
- In fact, for every exact 2-Sat formula, the algorithm finds an assignment τ such that $W(\tau) \geq \frac{2}{3} \sum_{i=1}^{m} w_i$, the weight of all clauses, and we say that the "totality ratio" is at least $\frac{2}{3}$.
- (More generally for Exact Max-k-Sat the ratio is $\frac{k}{k+1}$).
- This ratio is essentially a tight ratio for any $d = o(n)$.
- This is in contrast to a naive greedy algorithm derived from a randomized algorithm that achieves totality ratio $(2^k - 1)/2^k$.
- "In practice", the local search algorithm often performs better than the naive greedy and one could always start with the greedy algorithm and then apply local search.

Analysis of the oblivious local search for Exact Max-2-Sat

- Let τ be a local optimum and let
  - S_0 be those clauses that are not satisfied by τ
  - S_1 be those clauses that are satisfied by exactly one literal by τ
  - S_2 be those clauses that are satisfied by two literals by τ

  Let $W(S_i)$ be the corresponding weight.

- We will say that a clause involves a variable x_j if either $x_j$or $\bar{x}_j$ occurs in the clause. Then for each j, let
  - A_j be those clauses in S_0 involving the variable x_j.
  - B_j be those clauses C in S_1 involving the variable x_j such that it is the literal x_j or $\bar{x}_j$ that is satisfied in C by τ.
  - C_j be those clauses in S_2 involving the variable x_j.

  Let $W(A_j), W(B_j)$ be the corresponding weights.

Analysis of the oblivious local search (continued)

- Summing over all variables x_j, we get
  - $2W(S_0) = \sum_j W(A_j)$ noting that each clause in S_0 gets counted twice.
  - $W(S_1) = \sum_j W(B_j)$
- Given that τ is a local optimum, for every j, we have

$$W(A_j) \leq W(B_j)$$

  or else flipping the truth value of x_j would improve the weight of the clauses being satisfied.

- Hence (by summing over all j),

$$2W_0 \leq W_1.$$

Finishing the analysis

- It follows then that the ratio of clause weights not satisfied to the sum of all clause weights is

$$\frac{W(S_0)}{W(S_0)+W(S_1)+W(S_2)} \leq \frac{W(S_0)}{3W(S_0)+W(S_2)} \leq \frac{W(S_0)}{3W(S_0)}$$

- It is not easy to verify but there are examples showing that this $\frac{2}{3}$ bound is essentially tight for any N_d neighbourhood for $d = o(n)$.

- It is also claimed that the bound is at best $\frac{4}{5}$ whenever $d < n/2$. For $d = n/2$, the algorithm would be optimal.

- In the weighted case, as in the max-cut problem, we
have to worry about the number of iterations. And here again
we can speed up the termination by insisting that any improvement
has
to be sufficiently better.

Using the proof to improve the algorithm

- We can learn something from this proof to improve the performance.
- Note that we are not using anything about $W(S_2)$.
- If we could guarantee that $W(S_0)$ was at most $W(S_2)$ then the ratio of clause weights not satisfied to all clause weights would be $\frac{1}{4}$.
- Claim: We can do this by enlarging the neighbourhood to include $\tau' =$ the complement of τ.

The non-oblivious local search

- We consider the idea that satisfied clauses in S_2 are more valuable than satisfied clauses in S_1 (because they are able to withstand any single variable change).
- The idea then is to weight S_2 clauses more heavily.
- Specifically, in each iteration we attempt to find a $\tau' \in N_1(\tau)$ that improves the potential function
$$\frac{3}{2}W(S_1) + 2W(S_2)$$
instead of the oblivious $W(S_1) + W(S_2)$.
- More generally, for all k, there is a setting of scaling coefficients $c_1, \ldots, c_k$, such that the non-oblivious local search using the potential function $c_1 W(S_1) + c_2 W(S_2 + \ldots + c_k W(S_k)$ results in approximation ratio $\frac{2^k-1}{2^k}$ for exact Max-k-Sat.