CSC2420 Spring 2016: Lecture 3

Allan Borodin

January 28, 2016

Announcements and todays agenda

- I plan to add one or two DP questions this weekend to complete Assignment 1.
- Most likely I will make the assignment due next Thursday or it could be the week after.
- Todays agenda
  1. Provide another example (for the weighted vertex cover problem) of a non obvious greedy algorithm.
  2. We then move on to dynamic programming.

Another example where the "natural greedy" is not best

- Before moving (for now) beyond greedy-like algorithms, we consider another example (as we saw for set packing) where the "natural greedy algorithm" does not yield a good approximation.
- The vertex cover problem: Given node weighted graph $G = (V, E)$, with node weights $w(v), v \in V$.
  Goal: Find a subset $V' \subset V$ that covers the edges (i.e. $\forall e = (u, v) \in E$, either u or v is in V') so as to mininize $\sum_{v \in V'} w(v)$.
- Even for unweighted graphs, the problem is known to be NP-hard to obtain a 1.3606 approximation and under another (not so universally believed) conjecture (UGC) one cannot obtain a $2 - \epsilon$ approximation.
- For the unweighted problem, there are simple 2-approximation greedy algorithms such as just taking V' to be any maximal matching.
- The set cover problem is as follows: Given a weighted collection of sets $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$ over a universe U with set weights $w(S_i)$.
  Goal: Find a subcollection $\mathcal{S}'$ that covers the universe so as to minimize $\sum_{S_i \in \mathcal{S}'} w(S_i)$.

The natural greedy algorithm for weighted vertex cover (WVC)

If we consider vertex cover as a special case of set cover (how?), then the natural greedy (which is essentially optimal for set cover) becomes the following:

$d'(v) := d(v)$ for all $v \in V$

% $d'(v)$ will be the residual degree of a node

While there are uncovered edges

Let v be the node minimizing $w(v)/d'(v)$

Add v to the vertex cover;

remove all edges in $Nbhd(v)$;

recalculate the residual degree of all nodes in $Nbhd)v)$

End While

Figure : Natural greedy algorithm for weighted vertex cover. Approximation ratio $H_n \approx \ln n$ where $n = |V|$.

Clarkson's [1983] modified greedy for WVC

$d'(v) := d(v)$ for all $v \in V$

% $d'(v)$ will be the residual degree of a node

$w'(v) := w(v)$ for all $v \in V$

% $w'(v)$ will be the residual weight of a node

While there are uncovered edges

Let v be the node minimizing $w'(v)/d'(v)$

$w := w'(v)/d'(v)$

$w'(u) := w'(u) - w$ for all $u \in Nbhd(v)$

% For analysis only, set $we(u, v) = w$

Add v to the vertex cover;

remove all edges in $Nbhd(v)$;

recalculate the residual degree of all nodes in $Nbhd(v)$

End While

Figure : Clarkson's greedy algorithm for weighted vertex cover. Approximation ratio 2. Invariant: $w(v) = w'(v) + sum_{e \in E} we(e)$

Dynamic Programming (DP)

- The application and importance of dynamic programming goes well beyond search and optimzation problems.
- We will consider a few more or less "natural" DP algorithms and at least one not so obvious DP algorithm.
- In greedy like algorithms (and also local search, our next major paradigm) it is often easy to come up with reasonably natural algorithms (although we have seen some not so obvious examples) whereas sometimes the analysis can be relatively involved.
- In contrast, once we come up with an appropriate DP algorithm, it is often the case that the analysis is relatively easy.
- Here informally is the essense of DP algorithms: define an approriate generalization of the problem (which we usually give in the form of a multi-dimensional array) such that
  1. the desired result can be easily obtained from the array $S[\ ,\ ,\ \ldots\ ]$
  2. each entry of the array can be easily computed given "previous entries"

What more precisely is dynamic programming?

- So far, there are only a few attempts to formalize precise mdoels for (types) of dynamic programming algorithms.
- There are some who say this is not a useful question.
- I would disagree with the following comment: Whatever can be done in polynomial time, can be done by a polynomial time DP algorithm. What is the reasoning behind such a comment?
  Open problem: Can there be an optimal polynomal time DP (in any "reasonable" meaning of what is DP) for the maximum size or weight bipartite matching problem? Note: There are polynomial time optimal algorithms for these problem.
- And there may be more fundamdental or philosophical reasons for arguing against such attempts to formalize concepts.

What more precisely is dynamic programming?

- So far, there are only a few attempts to formalize precise mdoels for (types) of dynamic programming algorithms.
- There are some who say this is not a useful question.
- I would disagree with the following comment: Whatever can be done in polynomial time, can be done by a polynomial time DP algorithm. What is the reasoning behind such a comment?
  Open problem: Can there be an optimal polynomal time DP (in any "reasonable" meaning of what is DP) for the maximum size or weight bipartite matching problem? Note: There are polynomial time optimal algorithms for these problem.
- And there may be more fundamdental or philosophical reasons for arguing against such attempts to formalize concepts.
- Samuel Johnson (1709-1784): All theory is against freedom of the will; all experience for it.

Bellman 1957 (in the spirit of Samuel Johnson)

Bellman (who introduced dynamic programming) argued against attempts to formalize DP.

We have purposely left the description a little vague, since it is the spirit of the approach to these processes that is significant, rather than a letter of some rigid formulation. It is extremely important to realize that one can neither axiomatize mathematical formulation nor legislate away ingenuity. In some problems, the state variables and the transformations are forced upon us; in others, there is a choice in these matters and the analytic solution stands or falls upon this choice; in still others, the state variables and sometimes the transformations must be artificially constructed. Experience alone, combined with often laborious trial and error, will yield suitable formulations of involved processes.

Some simple DP algorithms

- Let's begin with an example used in many texts, namely a DP for the weighted interval scheduling problem WISP.
- We have already claimed that no priority algorithm can yield a constant approximation ratio but that we can obtain a 4-approximation using a revocable accaptance priority algorithm and an optimal algorithm using a priority stack algorithm.
- The optimal DP algorithm for WISP is based on the following "semantic array":
  - Sort the intervals $I_j = [s_j, f_j)$ so that $f_1 \leq f_2 \ldots \leq f_n$ (i.e. the PEO).
  - Define $\pi(i) = \max j : f_j \leq s_i$ (Note; if we do not want intervals to touch then use $f_j < s_i$.)
  - The definition of $\pi()$ is specific to this problem and I do not know a generalization for chordal graphs and hence the DP approach does not naturally extend.
  - For $1 \leq i \leq n$, Define $V[i]$ = optimal value obtainable from intervals $\{I_1, \ldots I_i\}$.

The DP for WISP continued

- We defined the array $V[\,]$ just in terms of the optimal value but the same array element can also contain a solution associated with this optimal value.
- So how would we efficiently compute the entries of $V[\,]$.
- The computation or recursive array (let's temporarily call it $\tilde{V}[\,]$) associated with $V[\,]$ is defined as follows:
  1. $\tilde{V}[1] = v_1$
  2. For $i > 1$, $\tilde{V}[i] = \max\{A, B\}$ where
     - $A = V[i-1]$
     - $B = v_i + \tilde{V}[\pi(i)]$

     That is, either we use the i^{th} interval or we don't.
- So why am I being so pedantic about this distinction between $V[\,]$ and $\tilde{V}[\,]$?

The DP for WISP continued

- We defined the array $V[\,]$ just in terms of the optimal value but the same array element can also contain a solution associated with this optimal value.
- So how would we efficiently compute the entries of $V[\,]$.
- The computation or recursive array (let's temporarily call it $\tilde{V}[\,]$) associated with $V[\,]$ is defined as follows:
  1. $\tilde{V}[1] = v_1$
  2. For $i > 1$, $\tilde{V}[i] = \max\{A, B\}$ where
     - $A = V[i-1]$
     - $B = v_i + \tilde{V}[\pi(i)]$

     That is, either we use the i^{th} interval or we don't.
- So why am I being so pedantic about this distinction between $V[\,]$ and $\tilde{V}[\,]$?
- I am doing this here just to point out that a proof of correctness would require showing that these two arrays are indeed equal! I will hereafter not make this distinction with the understanding that one does have to show that the computational or recursive array does indeed compute the entries correctly.

Some comments on DP and the WISP DP

- We can sort the intervals and compute $\pi()$ in time $O(n \log n)$ and then sequentially compute the entries of V in time $O(1)$ per iteration.
- We can also recursivley compute V, BUT must use memoization to avoid recomputing entries.
- To some extent, the need to use memoization distinguishes dynamic programming from divide and conquer.
- We can extend this DP to optimally solve the weighted interval scheduling problem when there are m machines; that is, we want to schedule intervals so that there is no intersection on any machine.
- This extension would directly lead to time (and space) complexity $O(n^{m+1})$; $O(n^m)$ with some more care.
- As we will soon discuss, we can model this simple type of DP by a priority branching tree (pBT) algorithm as formulated by Alekhnovich et al. Within this model, we can prove that for any fixed m, the width (and hence the space and thus time) of the algorithm for optimally scheduling intervals on m machines is $\Omega(n^m)$. The curse of dimensionality is necessary within this model.

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 $i > 0$, $V[i] = \max\{A, B\}$ where
     - $A = V[i-1, c]$
     - $B = v_i + V[c - s_i]$ 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 encoing of the input.

Dynamic programming and scaling

We have previously mentioned (slide 12 of lecture 1) that with some use of brute force and greediness, we can achieve a PTAS algorithm (achieving $(1+\epsilon)$-approximation) for the identical machines makespan (polynomial time in the number n of jobs but exponential in the number m of machines and exponential in $\frac{1}{\epsilon}$) and a PTAS for the knapsack problem (polynomial time in n and exponential in $\frac{1}{\epsilon}$). We now consider how dynamic programming (DP) can be used to achieve a PTAS for the makespan problem which is polynomial in m and n, and how to achieve an FPTAS for the knapsack problem.
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 $V = \sum v_j$) (rather than the encoded length $|v_j|$) of the input values.

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 "pevious 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
  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 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 also knapsack DPs as we will see).

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 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.

The better PTAS for makespan on identical machines

- In lecture 1 we stated that Graham showed a PTAS for the maksepan problem on identical machines. Namely, he showed a ratio of at most $(1+\epsilon)$ so that *for any fixed* m, this is a PTAS (polynomial time approximation scheme) with time $O(m^{m/\epsilon} + n \log n)$.
- We now think of m as being a parameter of the input instance and 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. 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 values 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 soluton that achieves makespan T.
- If there is such a solution then small jobs can be greedily scheduled without increasing the makespan too much.
- We use binary search to find a good T.

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 size 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.
- The n jobs can be scheduled within makespan T iff $M[n_1, , n_d]$ 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.

Taking care of the small jobs and accounting for the scaling down

- We now wish to add in the small jobs with sizes less than T/s. We continue to try to add small jobs as long as some machine does not exceed the target makespan T. If this is not possible, then makespan T is not possible.
- 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 $(1 + 1/s)T$.

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 sometimes 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 (undirected) 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 $\ell_i^j \in \{x_k, \bar{x}_k \mid 1 \leq k \leq n\}$.

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

- **Goal:** Find a truth assignment τ so as to maximize

$$W(\tau) = w(F \mid \tau),$$

the weighted sum of satisfied clauses w.r.t the truth assignment τ.

- It is NP hard to achieve an approximation better than $\frac{7}{8}$ for (exact) Max-3-Sat and hence for the non exact versions of Max-k-Sat for $)k > 3$.

The natural oblivious local search

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

Oblivious local search for Exact Max-k-Sat

Choose any initial truth assignment τ
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 (for example) a greedy algorithm and then apply local search.