CSC2420: Algorithm Design, Analysis and Theory Fall 2017

Allan Borodin and Nisarg Shah

October 4, 2017

Lecture 4

Announcements:

- Assignment 1 is now complete. I had originally set the due date for next Wednesday, October 11. I think this is still feasible but will entertain requests for some class extension if this seems necessary.

Todays agenda:

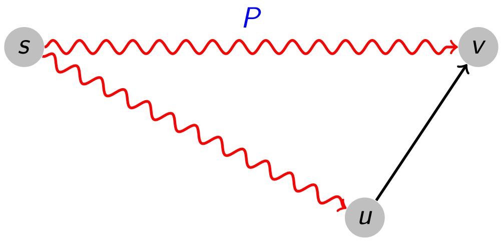
- Different styles of DP algorithms
- Some interesting applications of DP not in undergraduate texts.
- Begin local search

Different styles of dynamic programming

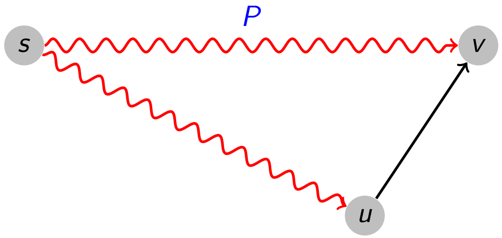
- Let's consider the single source least cost paths problem which is efficiently solved by Dijkstra's greedy algorithm for graphs in which all edge costs are non-negative.
- The least cost paths problem is still well defined as long as there are no negative cycles; that is, the least cost path is a simple path.
- The Bellman-Ford algorithm algorithm correctly computes shortest paths from a single source assuming no negative cycles.

Single source least cost paths for graphs with no negative cycles

- Following the DP paradigm, we consider the nature of an optimal solution and how it is composed of optimal solutions to "subproblems".
- Consider an optimal simple path P from source s to some node v.
  - This path could be just an edge.
  - But if the path P has length greater than 1, then there is some node u which immediately proceeds v in P. If P is an optimal path to v, then the path leading to u must also be an optimal path.



Single source least cost paths for graphs with no negative cycles



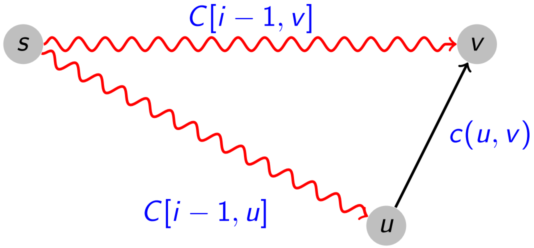
- This leads to the following semantic array:

$C[i, v] =$ the minimum cost of a simple path with path length at most i from source s to v. (If there is no such path then this cost is ∞.)

- The desired answer is then the single dimensional array derived by setting $i = n - 1$. (Any simple path has path length at most $n - 1$.)

How to compute the entries of C

- We can construct $C[i, v]$ from $C[i-1, \ldots]$ as follows:



- Let $C'[i, v]$ be the minimum value among
  - $C[i-1, v]$
  - $C[i-1, u] + c(u, v)$ for all $(u, v) \in E$.

The computation of the entries of C continued

- The computational array is defined as:

$$C[i,v] = \begin{cases} 0 & \text{if } i = 0 \text{ and } v = s \\ \infty & \text{if } i = 0 \text{ and } v \neq s \\ \min\{A, B\} & \text{otherwise} \end{cases}$$

$$A = C[i-1, v]$$

$$B = \min\Big\{ C[i-1, u] + c(u,v) : (u,v) \in E \Big\}$$

- Why is this slightly different from before?
  - Namely, showing the equivalence between the semantic and computationally defined arrays is not an induction on the indices of the input items in the solution which is intrinsic to the priority and pBT models.
  - But it is based on some other parameter (i.e. the path length) of the solution.
- Time complexity: n^2 entries $\times$ $O(n)$ per entry $=$ $O(n^3)$ in total.

Aside: Computing maximum profit path using the same DP?

- To define this problem properly we want to say "maximum cost simple path" since cycles will add to the cost of a path. Note: It is more common to refer to profits and not costs in a maximization problem but for notational consistency lets stay with costs.
- (For least cost we did not have to specify that the path is simple once we assumed no negative cycles.)
- Suppose we just replace min by max in the least cost DP. Namely,

$M[i, v]$ = the maximum cost of a simple path with path length at most i from
source s to v. (If there is no such path then this cost is $-\infty$.)

Computing the entries of M

- The recursive computation (corresponding to the min cost computational) would be

$$M[i,v] = \begin{cases} 0 & \text{if } i = 0 \text{ and } v = s \\ -\infty & \text{if } i = 0 \text{ and } v \neq s \\ \max\{A, B\} & \text{otherwise} \end{cases}$$

$$A = M[i-1, v]$$

$$B = \max\Big\{ M[i-1, u] + c(u,v) : (u,v) \in E \Big\}$$

- Is this correct?

What goes wrong?

- The problem calls for a maximum simple path but the recursion

$$B = \max\Big\{ M'[i-1,u] + c(u,v) : (u,v) \in E \Big\}$$

  does not guarantee that the path through u will be a simple path as v might occur in the path to u. Algorithm would work for a DAG.

- In fact, determining the maximum cost of a simple path is NP-hard.
  - A special case of this problem is the Hamiltonian path problem: does a graph $G = (V, E)$ have a simple path of length $|V| - 1$?
  - The Hamiltonian path problem is a variant of the "notorious" (NP-hard) traveling salesman problem (TSP).

The traveling salesman problem (TSP)

Traveling salesman problem (TSP)

Given a graph $G = (V, E)$ with a cost function $c : E \to \mathbb{R}_{\geq 0}$ determine if the cost of a simple cycle containing all the nodes (i.e. cycle length is $n = |V|$) assuming the graph has such a Hamiltonian cycle.

Without loss of generality, we can assume a complete graph (using $c(e) = \infty$ for any missing edges).

It is is roughly equivalent to consider the least cost Hamiltonian path problem. Namely, finding a least cost *simple* path of length *exactly* (and NOT at most) length $n - 1$ from some given starting node u. For the same reason as in the maximum cost path discussion, the least cost Hamiltonian path problem cannot be obtained by modifying the least cost path DP. Namely, we cannot dismiss the possibility of cycles in the path.

Not all exponentials are equal; using DP to obtain a better exponential time algorithm

A naive way to compute the least cost Hamiltonian path problem (with some given initial node u) is to consider all $(n-1)!$ simple paths of length $n-1$. A good estimate to $n!$ is Sterling's approximation:

$$n! \approx \sqrt{2\pi n}(\frac{n}{e})^n$$

.

By using an appropriate DP, we can reduce that time complexity to $O(n^2 2^n)$ which is still of course exponential but grows much slower than the factorial function ($n!$).

The more efficient exponential algorithm

Here is the idea as expressed in the following semantic array: For each subset $S \subseteq V$ with $u \in S$ and $v \notin S$
$C[S, v]$ is the least cost simple path from u to v containing each node in S exactly once.

If $|S| = 1$, then $C[S, v] = c(u, v)$ % S must be $\{u\}$
Else if $|S| > 1$, then $C[S, v] = \min_{x \notin S} C[S, x] + c(x, v)$

We note that the least cost Hamiltonian path problem is "NP-hard to approximate" to any constant whereas there are efficient approximations if the cost function satisfies the triangle inequality.

The all pairs least cost problem

- We now wish to compute the least cost path for all pairs (u, v) in an edge weighted directed graph (with no negative cycles).

- We can repeat the single source DP for each possible source node: complexity $O(n^4)$

- We can reduce the complexity to $O(n^3 logn)$ using the DP based on the semantic array

$E[j, u, v] =$ cost of shortest path of path length at most 2^j from u to v.

- Notice that again we would prove correctness by an induction on the length of a path. This DP is also different from the preceding DPs in that each entry $E[j, u, v]$ makes two recursive calls to $E[j-1, u, w]$ and $E[j-1, w, v]$ for each vertex w.

An $O(n^3)$ DP for the all pairs problem

- Let's assume (without loss of generality) that $V = \{1, 2, \ldots, n\}$.
- We now define the semantic array

$G[k, u, v]$ = the least cost of a (simple) path π from u to v such that the internal nodes in the path π are in the subset $\{1, 2, \ldots, k\}$.

- The recursive computation of G is as follows:

$$G[0, u, v] = \begin{cases} 0 & \text{if } u = v \\ c(u, v) & \text{if } (u, v) \text{ is an edge} \\ \infty & \text{otherwise.} \end{cases}$$

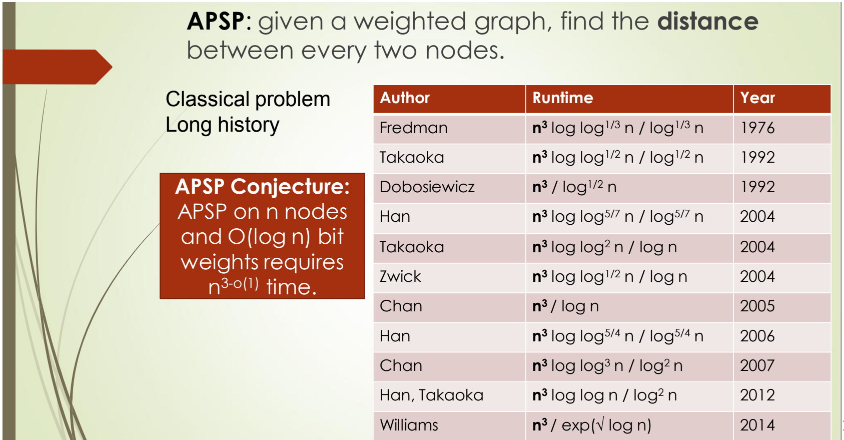
$$G[k + 1, u, v] = \min\{A, B\}$$

where $A = G[k, u, v]$ and $B = G[k, u, k + 1] + G'[k, k + 1, v]$.

- Like the recursion for the previous array $E[j, u, v]$, the recursion here uses two recursive calls for each entry.
- Time complexity: n^3 entries $\times$ $O(1)$ per entry $=$ $O(n^3)$ in total.

Is $O(n^3)$ the best we can do for the all pairs shortest paths problem (APSP)?

Currently there is no $O(n^{3-\epsilon})$ time algorithm for the APSP.



APSP: given a weighted graph, find the **distance** between every two nodes.

Classical problem
Long history

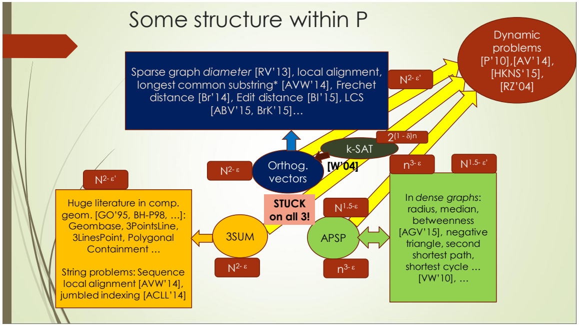
APSP Conjecture: APSP on n nodes and O(log n) bit weights requires $n^{3-o(1)}$ time.

Author	Runtime	Year
Fredman	$n^3 \log\log^{1/3} n / \log^{1/3} n$	1976
Takaoka	$n^3 \log\log^{1/2} n / \log^{1/2} n$	1992
Dobosiewicz	$n^3 / \log^{1/2} n$	1992
Han	$n^3 \log\log^{5/7} n / \log^{5/7} n$	2004
Takaoka	$n^3 \log\log^2 n / \log n$	2004
Zwick	$n^3 \log\log^{1/2} n / \log n$	2004
Chan	$n^3 / \log n$	2005
Han	$n^3 \log\log^{5/4} n / \log^{5/4} n$	2006
Chan	$n^3 \log\log^3 n / \log^2 n$	2007
Han, Takaoka	$n^3 \log\log n / \log^2 n$	2012
Williams	$n^3 / \exp(\sqrt{\log n})$	2014

What is the "evidence" for the APSP conjecture?

Analogous to the theory of NP completeness and polynomial time reductions, there is a relatively newer area of complexity theory-algorithm design call *fine grained complexity*.

The APSP is stated for a RAM model (with $O(\log n)$ bit words), for n node graphs with edge weights in $\{1, \ldots, n^c\}$ for some constant c.


Some structure within P
Sparse graph *diameter* [RV'13], local alignment, longest common substring* [AVW'14], Frechet distance [Br'14], Edit distance [BI'15], LCS [ABV'15, BrK'15]…
Dynamic problems [P'10],[AV'14], [HKNS'15], [RZ'04]
$N^{2-\varepsilon'}$
$2^{(1-\delta)n}$
k-SAT
[W'04]
Orthog. vectors
$N^{2-\varepsilon}$
$n^{3-\varepsilon}$
$N^{1.5-\varepsilon'}$
$N^{2-\varepsilon'}$
STUCK on all 3!
$N^{1.5-\varepsilon}$
Huge literature in comp. geom. [GO'95, BH-P98, …]: Geombase, 3PointsLine, 3LinesPoint, Polygonal Containment …
String problems: Sequence local alignment [AVW'14], jumbled indexing [ACLL'14]
3SUM
APSP
$N^{2-\varepsilon}$
$n^{3-\varepsilon}$
In *dense graphs*: radius, median, betweenness [AGV'15], negative triangle, second shortest path, shortest cycle … [VW'10], …

A similar DP (using 2 recursive calls)

The chain matrix product problem

- We are given n matrices (say over some field) $M_1, \ldots, M_n$ with M_i having dimension $d_{i-1} \times d_i$.
- **Goal:** compute the matrix product

$$M_1 \cdot M_2 \cdot \ldots \cdot M_n$$

using a given subroutine for computing a single matrix product $A \cdot B$.

- We recall that matrix multiplication is associative; that is,

$$(A \cdot B) \cdot C = A \cdot (B \cdot C).$$

- But the number of operations for computing $A \cdot B \cdot C$ generally depends on the order in which the pairwise multiplications are carried out.

The matrix chain product problem continued

- Let us assume that we are using classical matrix multiplication and say that the scalar complexity for a $(p \times q)$ times $(q \times r)$ matrix multiplication is pqr.

- For example say the dimensions of A, B and C are (respectively) 5×10, 10×100 and 100×50.

- Then using $(A \cdot B) \cdot C$ costs $5000 + 25000 = 30000$ scalar operations whereas $A \cdot (B \cdot C)$ costs $50000 + 2500 = 52500$ scalar ops.

- **Note:** For this problem the input is these dimensions and not the actual matrix entries.

Parse tree for the product chain

- The matrix product problem then is to determine the parse tree that describes the order of pairwise products.
- At the leaves of this parse tree are the individual matrices and each internal node represents a pairwise matrix multiplication.
- Once we think of this parse tree, the DP is reasonably suggestive:

The root of the optimal tree is the last pairwise multiplication and the subtrees are subproblems that must must be computed optimally.

The DP array for the matrix chain product problem

$C[i,j]$ = the cost of an optimal parse of $M_i \cdot \ldots \cdot M_j$ for $1 \leq i \leq j \leq n$.

- The recursive computation is :

$$C[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min\{C[i,k] + C[k+1,j] + d_{i-1}d_kd_j : i \leq k < j\} & \text{if } i < j \end{cases}$$

- Essentially in all these cases we are computing an optimal parse tree.

Some DPs not occurring in texts

Dynamic programming by itself or when combined with scaling and other techniques can yield interesting optimal and approximation algorithms. The following problems provide additional more involved DP examples.

- The Calinescu et al [2011] constant approximation algorithms for the *resource allocation problem*. A job is an interval I_j having a profit v_i and a bandwidth or resource requirement $b_i \leq B$. An optimal DP can be used for the "big" bandwidth jobs and various algorithms can be used for the small jobs. Approximation bounds follow by taking the maximum profit between big and small bandwidths. This problem generalizes the interval selection and knapsack problems.
- An optimal DP algorithm is Baptiste's [1999] algorithm for maximizing the weighted profit of scheduling equal processing time jobs with release times and deadlines. A related result is the Chuzhoy et al [2006] pseudo polynomial time algorithm for maximizing the profit of unweighted jobs with release times, deadlines and processing times such that $d_i - r_i \leq kp_i$ for all i and some fixed k. That is, each job has some fixed relative "window size" in which to be scheduled.

Scheduling jobs with integral release times, dealines and processing times

The Baptiste and Chuzhoy et al problems are sub-cases of the more general *throughput problem* where a job J_i is described by (r_i, d_i, p_i, v_i) such that $r_i + p_i \leq d_i$ and v_i is the weight of value of the i^{th} job if scheduled so as to complete before its deadline d_i.
We are assuming all parameters are integral. If we allow different processing times p_i for each job, the problem becomes strongly NP hard and weakly NP hard if $r_i = 0$ for all i . More precisely, when $r_i = 0$, the knapsack is a special case and a similar algorithm (DP + scaling) can be used to achieve a FPTAS.

The problem specializes to the weighted interval selection problem when $r_i + p_i = d_i$ for all i and as we know this can be efficiently optimally solved. It is also optimally solved by a greedy algorithm when $r_i = 0$ and $p_i = p = 1$ for all jobs J_i.

The Baptiste DP

In contrast to the above cases which can be optimally (or nearly optimally) solved by relatively intuitive algorithms , the Baptiste and Chuzhoy et al problems seem to require a very non trivial DP. The Baptiste's time bound in $O(n^7)$.
The Baptiste problem assumes all processing time $p_i = p$ for some fixed p.
We will just state the algorithm to get a sense of its complexity.
Define what is the set of allowable starting times $AST = \{t : t = r_i + \ell \cdot p$ for some $r_i, \ell \in \{0, \ldots, n\}$

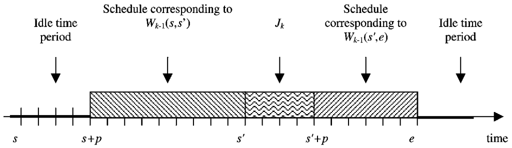
In order to define the appropriate array, we first need the following definition of jobs with release times in certain time intervals, namely:
$U_k(s, e) = \{J_i | i \leq k$ and $s \leq r_i < e\}$ for $s \leq e$

Baptiste DP continued

We define the desired array $W_k(s,e)$ of values; it is the maximum value that can be achieved in a schedule S of jobs $J_i \in U_k(s,e)$ that satisfy the following properties:

- S is idle before time $s+p$
- S is idle after time e
- The starting times of jobs in S belong to AST.

The following figure in Baptiste's paper illustrates his DP solution:


Idle time period
Schedule corresponding to $W_{k-1}(s,s')$
J_k
Schedule corresponding to $W_{k-1}(s',e)$
Idle time period
s
$s+p$
s'
$s'+p$
e
time

$W_k(s,e) = W_{k-1}(s,e)$ if $r_k \notin [s,e)$ or if J_k not used; otherwise
$W_k(s,e) =$
$\max_{s' \in AST : \max(r_k, s+p) \le s' \le \min(d_k, e) - p} (w_k + W_{k-1}(s,s') + W_{k-1}(s',e))$

Local Search: another conceptually simple approach

We now begin a discussion of *local search* which for me, along with greedy algorithms, is one of the two conceptually simplest search/optimization paradigms.

The vanilla local search paradigm

```
"Initialize" $S$
While there is a "better" solution $S'$ in the "local neighbourhood" $Nbhd(S)$
    $S := S'$
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 constitutes a reasonable definition of a local neighbourhood?
3. What do we mean by "better"?

Answering these questions (especially as to defining a 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 search-able) 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. I think it should be called greedy 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 locality 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:

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

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

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

- Repeating the argument for B we have:

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

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

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

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

The complexity of the single move local search

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

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 $\frac{7}{8}$ for (exact) Max-3-Sat and hence for the non exact versions of Max-k-Sat for $k \geq 3$.

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 $\tau$
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 (for example) a 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 $W(A_j), W(B_j), W(C_j)$ 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.

Sketch of $\frac{3}{4}$ totality bound for the non oblivious local search for Exact Max-2-Sat

- Let $P_{i,j}$ be the weight of all clauses in S_i containing x_j.
- Let $N_{i,j}$ be the weight of all clauses in S_i containing $\bar{x}_j$.
- Here is the key observation for a local optimum τ wrt the stated potential:

$$-\frac{1}{2}P_{2,j} - \frac{3}{2}P_{1,j} + \frac{1}{2}N_{1,j} + \frac{3}{2}N_{0,j} \leq 0$$

- Summing over variables $P_1 = N_1 = W(S_1)$, $P_2 = 2W(S_2)$ and $N_0 = 2W(S_0)$ and using the above inequality we obtain

$$3W(S_0) \leq W(S_1) + W(S_2)$$

Some comparative experimental results for local search based Max-Sat algorithms

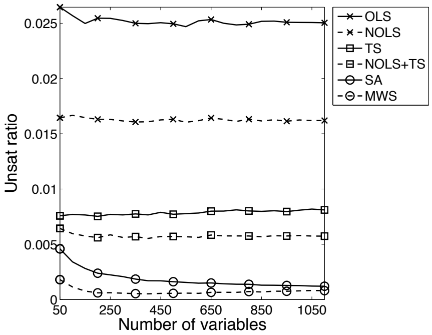


Fig. 1. Average performance when executing on random instances of exact MAX-3-SAT.

[From Pankratov and Borodin 2010]

More experiments for benchmark Max-Sat

	OLS	NOLS	TS	NOLS+TS	SA	MWS
OLS	0	457	741	744	730	567
NOLS	160	0	720	750	705	504
TS	0	21	0	246	316	205
NOLS+TS	8	0	152	0	259	179
SA	30	50	189	219	0	185
MWS	205	261	453	478	455	0

Table 2. MAX-SAT 2007 benchmark results. Total number of instances is 815. The tallies in the table show for how many instances a technique from the column improves over the corresponding technique from the row.

[From Pankratov and Borodin 2010]

More experiments for benchmark Max-Sat

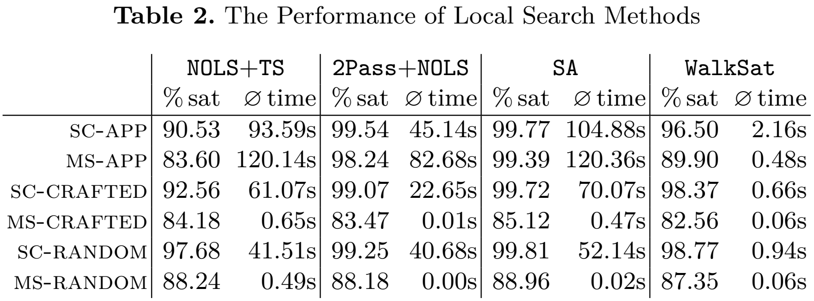


Table 2. The Performance of Local Search Methods

	NOLS+TS		2Pass+NOLS		SA		WalkSat	
	% sat	∅ time	% sat	∅ time	% sat	∅ time	% sat	∅ time
SC-APP	90.53	93.59s	99.54	45.14s	99.77	104.88s	96.50	2.16s
MS-APP	83.60	120.14s	98.24	82.68s	99.39	120.36s	89.90	0.48s
SC-CRAFTED	92.56	61.07s	99.07	22.65s	99.72	70.07s	98.37	0.66s
MS-CRAFTED	84.18	0.65s	83.47	0.01s	85.12	0.47s	82.56	0.06s
SC-RANDOM	97.68	41.51s	99.25	40.68s	99.81	52.14s	98.77	0.94s
MS-RANDOM	88.24	0.49s	88.18	0.00s	88.96	0.02s	87.35	0.06s

Figure: Table from Poloczek and Williamson 2017