CSC2420: Algorithm Design, Analysis and Theory Spring (or Winter for pessimists) 2017

Allan Borodin

January 23, 2017

Lecture 3

Announcements:

- I have posted the first 6 questions for assignment 1. When necessary, I will elaborate (and sometimes give hints) for questions during class. So please do ask for any clarifications that are needed!
- I am reserving Thursdays 3-4 for an office hour but I welcome discussing the course whnever I am free.

Todays agenda:

- Continue whirlwind discussion of basic combinatorial algorithms
- Dynamic programming
- Local search
- Ford Fulkerson Max flow and max flow based based algorithms if time permits.

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 encoding of the input.

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 "pseudo polynomial" time DP for the problem, namely an algorithm that is polynomial in the numeric value v (rather than the encoded length $|v|$) of the input values.

Define $S[j, v]$ = the minimum size s needed to achieve a profit 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 is the essense of DP algorithms; namely, defining an approriate generalization of the problem (which we give in the form of an array) such that

1. the desired result can be easily obtained from this array
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

1. The value of an optimal solution is $max\{v|S[n, v]$ is finite$\}$.
2. 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 $v \leq \sum_{j=1}^{n} v_j$.
   - Basis: $S[0, v] = \infty$ for all v
   - Induction: $S[j, v] = \min\{A, B\}$ where $A = S[j-1, v]$ and $B = S[j-1, \max\{v - v_j, 0\}] + s_j$.
3. It should be clear that while we are computing these values that we can at the same time be computing a solution corresponding to each entry in the array.
4. For efficiency one usually computes these entries iteratively but one could use a recursive program with *memoization*.
5. The running time is $O(n, V)$ where $V = \sum_{j=1}^{n} v_j$.
6. 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 combinaton 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 makespan on identical machines when there is 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.
- The array we will use to obtain the desired optimal makespan is as follows:
  $M[x_1, \ldots, x_d]$ = the minimum number of machines needed to schedule x_i jobs having size z_i within makespan T. (Here we can assume $T \geq \max p_i \geq \max z_i$ so that this minimum is finite.)
- The n jobs can be scheduled within makespan T iff $M[n_1, , n_d]$ is at most m.

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 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 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. 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:

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

- 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?

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 $k \geq 3$.

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 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.

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

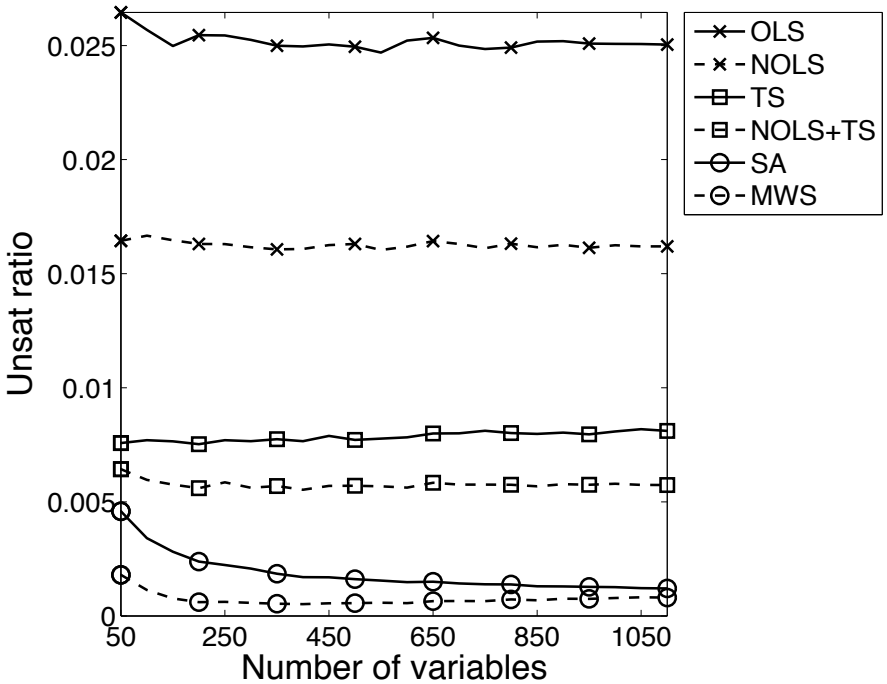

Fig. 1. Average performance when executing on random instances of exact MAX-3-SAT.

[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

Oblivious and non-oblivious local search for $k+1$ claw free graphs

- We again consider the maximum weighted independent set problem in a $k+1$ claw free graph. (Recall the argument generalizing the approximation ratio for the k set packing problem.)
- The standard greedy algorithm and the 1-swap oblivious local search both achieve a $\frac{1}{k}$ approximation for the WMIS in $k+1$ claw free graphs. Here we define an "ℓ-swap" oblivous local search by using neighbrourhoods defined by bringing in a set S of up to ℓ vertices and removing all vertices adjacent to S.
- For the unweighted MIS, Halldórsson shows that a a 2-swap oblivious local search will yield a $\frac{2}{k+1}$ approximation.

Berman's [2000] non-oblivious local search

- For the weighted MIS, the "ℓ-swap" oblivous local search results (essentially) in an $\frac{1}{k}$ locality gap for any constant ℓ.
- Chandra and Halldóssron [1999] show that by first using a standard greedy algorithm to initialize a solution and then using a "greedy" k-swap oblivious local search, the approximation ratio improves to $\frac{3}{2k}$.
- Can we use non-oblivious local search to improve the locality gap? Once again given two solutions V_1 and V_2 having the same weight, when is one better than the other?

Berman's [2000] non-oblivious local search

- For the weighted MIS, the "ℓ-swap" oblivous local search results (essentially) in an $\frac{1}{k}$ locality gap for any constant ℓ.
- Chandra and Halldóssron [1999] show that by first using a standard greedy algorithm to initialize a solution and then using a "greedy" k-swap oblivious local search, the approximation ratio improves to $\frac{3}{2k}$.
- Can we use non-oblivious local search to improve the locality gap? Once again given two solutions V_1 and V_2 having the same weight, when is one better than the other?

Berman's [2000] non-oblivious local search

- For the weighted MIS, the "ℓ-swap" oblivous local search results (essentially) in an $\frac{1}{k}$ locality gap for any constant ℓ.
- Chandra and Halldóssron [1999] show that by first using a standard greedy algorithm to initialize a solution and then using a "greedy" k-swap oblivious local search, the approximation ratio improves to $\frac{3}{2k}$.
- Can we use non-oblivious local search to improve the locality gap? Once again given two solutions V_1 and V_2 having the same weight, when is one better than the other?
- Intuitively, if one vertex set V_1 is small but vertices in V_1 have large weights that is better than a solution with many small weight vertices.

Berman's [2000] non-oblivious local search

- For the weighted MIS, the "ℓ-swap" oblivous local search results (essentially) in an $\frac{1}{k}$ locality gap for any constant ℓ.
- Chandra and Halldóssron [1999] show that by first using a standard greedy algorithm to initialize a solution and then using a "greedy" k-swap oblivious local search, the approximation ratio improves to $\frac{3}{2k}$.
- Can we use non-oblivious local search to improve the locality gap? Once again given two solutions V_1 and V_2 having the same weight, when is one better than the other?
- Intuitively, if one vertex set V_1 is small but vertices in V_1 have large weights that is better than a solution with many small weight vertices.
- Berman chooses the potential function $g(S) = \sum_{v \in S} w(v)^2$. Ignoring some small ϵ's, his k-swap non-oblivious local search achieves a locality gap of $\frac{2}{k+1}$ for WMIS on $k+1$ claw-free graphs.

Some (almost) concluding comments (for now) on local search

- For the metric k-median problem, until recently, the best approximation was by a local search algorithm. Using a p-flip (of facilities) neighbourhood, Arya et al (2001) obtain a $3 + 2/p$ approximation which yields a $3 + \epsilon$ approximation running in time $O(n^{2/\epsilon})$.
- Li and Svensson (2013) obtained a $(1 + \sqrt{3} + \epsilon) \approx 2.732 + \epsilon$ LP-based approximation running in time $O(n^{1/\epsilon^2})$. Surprisingly, they show that an α approximate "pseudo solution" using $k + c$ facilities can be converted to an $\alpha + \epsilon$ approximate solution running in $n^{O(c/\epsilon)}$ times the complexity of the pseudo solution. The latest improvement is a $2.633 + \epsilon$ approximation by Ahmadian et al (2017).
- An interesting (but probably difficult) open problem is to use non oblivious local search for the metric k-median, facility location, or k-means problems. These well motivated clustering problems play an important role in operations research, CS algorithm design and

End of current concluding remarks on local search

- Perhaps the main thing to mention now is that local search is the basis for many practical algorithms, especially when the idea is extended by allowing some well motivated ways to escape local optima (e.g. simulated annealing, tabu search) and combined with other paradigms.
- Although local search with all its variants is viewed as a great "practical" approach for many problems, local search is not often theoretically analyzed. It is not surprising then that there hasn't been much interest in formalizing the method and establishing limits.
- We will be discussing paradigms relating to Linear Programming (LP). LP is often solved by some variant of the simplex method, which can be thought of as a local search algorithm, moving fron one vertex of the LP polytope to an adjacent vertex.
- Our next "paradigm" is max flow and flow based algorithms and max flow is often solved by some variant of the Ford Fulkerson method which also can be thought of as a local search algorithm.

Ford Fulkerson max flow based algorithms

A number of problems can be reduced to the max flow problem. As suggested, max flow itself can be viewed as a local search algorithm.

Flow Networks

A flow network $\mathcal{F} = (G, s, t, c)$ consists of a "bi-directional" graph $G = (V, E)$, a source s and termnal node t, and c is a non-negative real valued (*capacity*) function on the edges.

What is a flow

A flow f is a real valued function on the edges satisfying the following properties:

1. $f(e) \leq c(e)$ for all edges e (capacity constraint)
2. $f(u, v) = -f(v, u)$ (skew symmetry)
3. For all nodes u (except for s and t), the sum of flows into (or out of) u is zero. (Flow conservation).
   Note: this is the "flow in = flow out" constraint for the convention of only having non negative flows.

The max flow problem

- The goal of the max flow problem is to find a valid flow that maximizes the flow out of the source node s. As we will see this is also equivalent to maximizing the flow in to the terminal node t. (This should not be surprising as flow conservation dictates that no flow is being stored in the other nodes.) We let $val(f) = |f|$ denote the flow out of the source s for a given flow f.
- We will study the Ford Fulkerson augmenting path scheme for computing an optimal flow. I am calling it a scheme as there are many ways to instantiate this scheme although I dont view it as a general paradigm in the way I view (say) greedy and DP algorithms.
- I am assuming that many people in the class have seen the Ford Fulkerson algorithm so I will discuss this quickly. I am following the development of the model and algorithm as in Cormen et al (CLRS). That is, we have negative flows which simplifies the analysis but may be less intuitive.

A flow f and its residual graph

- Given any flow f for a flow network $\mathcal{F} = (G, s, t, c)$, we can define the residual graph $G_f = (V, E(f))$ where $E(f)$ is the set if all edges e having *positive* residual capacity ; i.e. the residual capacity of e wrt to f is $c_f(e) = c(e) - f(e) > 0$.
- Note that $c(e) - f(e) \geq 0$ for all edges by the capacity constraint. Also note that with our convention of negative flows, even a zero capacity edge (in G) can have residual capacity.
- The basic concept underlying Ford Fulkerson is that of an augmenting path which is an $s - t$ path in G_f. Such a path can be used to augment the current flow f to derive a better flow f'.
- Given an augmenting path π in G_f, we define its residual capacity wrt f as $c_f(\pi) = \min\{c_f(e) | e \text{ in the path } \pi\}$.

The Ford Fulkerson scheme

Ford Fulkerson

$f := 0$;
$G_f := G$ %initialize
While there is an augmenting path in G_f
    Choose an augmenting path π
    $\tilde{f} := f + f_{pi}; f := \tilde{f}$ % Note this also changes G_f
End While

I call this a scheme rather than a well specified algorithm since we have not said how one chooses an augmenting path (as there can be many such paths)