CSC2420 Spring 2015: Lecture 5

Allan Borodin

February 5, 2015

Announcements and todays agenda

- Announcements
  The first three questions for the second problem set are now posted.
- Todays agenda
  1. Finish up discusion of non-oblvious local search for Exact Max-2-Sat
  2. Some experiments regarding various Max-Sat algorithms
  3. Max Flow and the Ford Fulkerson template.
  4. Some applications of max flow

The non-oblivious local search for Exact Max-2-Sat

- 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). Recall that S_i are those closes satisfied by exactly i literals.

- 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 experimental results concerning Max-Sat

- Of course, one wonders whether or not a worse case approximation will actually have a benefit in "practice".
- "In practice", local search becomes more of a "heuristic" where one uses various approaches to escape (in a principled way) local optima and then continuing the local search procedure. Perhaps the two most commonly used versions are Tabu Search and Simulated Annealing.
- Later, we will also discuss methods based on "random walks".
- This takes us beyond the scope of this course as we view these algorithmic idea as starting points.
- But for what it is worth, here are some experimental results both for artifically constructed instances and well as for one of the many benchmark test sets for Max-Sat. These experimetns suggest that non-oblivios local search can have "practical" as well as theoretical application.

Experiment for unweighted Max-3-Sat

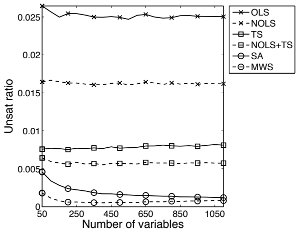

Fig. 1. Average performance when executing on random instances of exact MAX-3-SAT.

[From Pankratov and Borodin]

Experiment 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]

The jump local search algorithm for makespan on identical machines

- Start with any initial solution
- It doesn't matter how jobs are arranged on a machine so the algorithm can move any job (on a "critical machine" defining the current makespan value) if that move will "improve things".
    - That is, a (successful) jump move is one that moves any job to another machine so that either the makespan is decreased or the number of machines determining the current makespan is decreased.
- **Note:** Strictly speaking, this is a non-oblivious local search as we may not be decreasing the current makepsan.
- Finn and Horowitz [1979] prove:
  that the "locality gap" for this local search algorithm is $2 - \frac{2}{m+1}$.
  That is, this is the worst case ratio for some local optimum compared to the global optimum.
- To bound the number of iterations, in moving J_k,
  it should be moved to a machine having the current minimum load.

A more complicated local search for makespan

- The jump local search does not provide as good an approximation as the LPT greedy algorithm and doesn't provide a constant (independent of m) approximation for the makespan problem in the uniformly related machines model.

- There is a more involved neighbourhood called the **push neighbourhood**, inspired by the Kernighan and Lin variable depth local search algorithms for graph partitioning and TSP.

Push operation

A push operation is a sequence of jumps defined as follows:

- A push is initiated by a jump of a job J_k on a critical machine to a machine M_i on which it "fits in the sense that $p_k + \sum_{J_j \text{ on } M_i \text{ and } p_j \geq p_k} p_j$ is less than the current makespan.
- If smaller (i.e. with $p_j < p_k$) jobs on M_i cause the makespan on M_i to equal or exceed the current makespan then in order of smallest jobs first, we keep moving small jobs to a priority queue.
- We then try to move jobs (in order of the largest job first) on the queue to a machine on which it fits (in which case the operation was unsuccessful) and continue the process until either there is no machine on which it fits or the priority queue is empty.

Locality gaps for push local search

- Since a push optimal solution is also a jump optimal solution, it follows that the push local search has locality gap at most $2 - \frac{2}{m+1}$.
- The current lower bound on the locality gap is $\frac{4m}{3m+1}$
- The bound $\frac{8}{7}$ is tight for $m = 2$ and hence beats LPT for $m = 2$ machines.
- For uniformly related machines, the jump locality gap is at most $2 - \frac{2}{m+1}$ and the lower bound is arbitrarily close to $3/2$.
- Push does not give a constant (independant of m) approximation for the restricted or unrelated machines models.

The (metric) facility location and k-median problems

- Two of the most studied problems in operations research and CS algorithm design are the related uncapacitated facility location problem (UFL) and the k-median problem. In what follows we restrict attention to the (usual) metric case of these problems defined as follows:

Definition of UFL

Input: (F, C, d, f) where F is a set of faciltites, D is a set of clients or cities, d is a metric distance function over $F \cup D$, and f is an opening cost function for facilities.
Output: A subset of facilities F' minimizing $\sum_{i \in F'} f_i + \sum_{j \in C} d(j, F')$ where f_i is the opening cost of facility i and $d(j, F') = min_{i \in F'} d(j, i)$.

- In the capacitated version, facilities have capacities and cities can have demands (rather than unit demand). The constraint is that a facility can not have more assigned demand than its capacity so it is not possible to always assign a city to its closest facility.

UFL and k-median problems continued

Deifnition of k-median problem

Input: (F, C, d, k) where F, C, d are as in UFL and k is the number of facilities that can be opened.
Output: A subset of facilities F' with $|F'| \leq k$ minimizing $\sum_{i \in F'} f_i + \sum_{j \in C} d(j, F')$

- These problems are clearly well motivated. Moreover, they have been the impetus for the development of many new algorithmic ideas which we will hopefully at least touch upon throughout the course.
- There are many variants of these problems and in many papers the problems are defined so that $F = C$; that is, any city can be a facility. If a solution can be found when F and C are disjoint then there is a solution for the case of $F = C$.

UFL and k-median problems continued

- It is known (Guha and Khuller) that UFL is hard to approximate to within a factor better than 1.463 assuming NP is not a subset of $DTIME(n^{\log \log n})$ and the k-median problem is hard to approximate to within a factor better than $1 + 1/e \approx 1.736$ (Jain, Mahdian, Saberi).
- The UFL problem is better understood than k-median. After a long sequence of improved approximation results the current best polynomial time approximation is 1.488 (Li, 2011).
- For k-median, 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 Svennsson (2013) have obtained a $(1 + \sqrt{3} + \epsilon)$ 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.

Some concluding comments (for now) on local search

- We will hopefully return later to local search and in particular non-oblivious local search.
- But suffice it to say now 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).
- Although local search with all its variants is viewed as a great "practical" approach for many problems, local search is not often analyzed. It is not surprising then that there hasn't been much interest in formalizing the method and establishing limits.
- LP is itself often solved by some variant of the simplex method, which can also be thought of as a local search algorithm, moving fron one vertex of the LP polytope to an adjacent vertex.
  - No such method is known to run in polynomial time in the worst case.

Ford Fulkerson max flow based algorithms

A number of problems can be reduced to the max flow problem.

Flow Networks

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

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 $\pi$
    $f := f + f_{pi}$   % 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)

The max flow-min cut theorem

Ford Fulkerson Max Flow-Min Cut Theorem

The following are equivalent:

1. f is a max flow
2. There are no augmenting paths wrt flow f; that is, no $s - t$ path in G_f
3. $val(f) = c(S, T)$ for some cut (S, T) ; hence this cut (S, T) must be a min (capacity) cut since $val(f) \leq c(S, T)$ for all cuts.

Hence the name max flow $(=)$ min cut

Comments on max flow - min cut theorem

- As previously mentioned, the Ford Fulkerson algorithms can be viewed as local search algorithms.
- This is a rather unusual local search algorithm in that any local optimum is a global optimum.
- Suppose we have a flow network in which all capacities are integral. Then :
    1. Any Ford Fulkerson implementation must terminate.
    2. If the sum of the capacities for edges leaving the source s is C, then the algorithm terminates in at most C iterations and hence with complexity at most $O(mC)$.
    3. Ford Fulkerson implies that there is an optimal integral flow. (There can be other non integral optimal flows.)

Good and bad ways to implement Ford Fulkerson

- There are bad ways to implement the networks such that
  1. There are networks with non rational capacities where the algorithm does not terminate.
  2. There are networks with integer capacities where the algorithm uses exponential (in representation of the capacities) time to terminate.
- There are various ways to implement Ford-Fulkerson so as to achieve polynomial time. Edmonds and Karp provided the first polynomial time algorithm by showing that a shortest length augmenting path yields the time bound $O(|V| \cdot |E|^2)$. For me, the conceptually simplest polynomial time analysis is the Dinitz algorithm which has time complexity $O(|V|^2|E|)$ and also has the advantage of leading to the best known time bound for unweighted bipartite matching. I think the best known worst case time for max flow is the preflow-push-relabel algorithm of Goldberg and Tarjan with time $O(|V| \cdot |E| \ polylog(|E|)$. or maybe $O(|V| \cdot |E|)$.

The Dinitz (sometimes written Dinic) algorithm

- Gven a flow f, define the leveled graph $L_f = (V', E')$ where $V' = \{v | v \text{ reachable from } s \text{ in } G_f\}$ and $(u, v) \in E'$ iff $level(v) = level(u) + 1$. Here $level(u)$ = length of shortest path from s to u.
- A blocking flow $\tilde{f}$ is a flow such that every s to t path in L_f has a saturated edge.

The Dinitz Algorithm

```
Initialize $f(e) = 0$ for all edges $e$
While $t$ is reachable from $s$ in $G_f$ (else no augmenting path)
    Construct $L_f$ corresponding to $G_f$
    Find a blocking flow $\hat{f}$ wrt $L_f$ and set $f := f + \hat{f}$
End While
```

The run time of Dinitz' algorithm

Let $m = |E|$ and $n = |V|$

- The algorithm halts in at most $n - 1$ iterations (i.e. blocking steps).
- The residual graph and the levelled graph can be computed in time $O(m)$ with breadth first search and using depth first search we can compute a blocking path in time $O(mn)$. Hence the total time for the Dinitz blocking flow algorithm is $O(mn^2)$
- A unit network is one in which all capaities are in $\{0,1\}$ a nd for each node $v \neq s, t$, either v has at most one incoming edge (i.e. of capacity 1) or at most one outgoing edge. In a unit network, the Dinitz algorithm terminates within $2\sqrt{n}$ iterations and hence on such a network, a max flow can be computed in time $O(m\sqrt{n})$ (Hopcroft and Karp [1973].

Application to unweighted bipartite matching

- We can transform the maximum bipartite matching problem to a max flow problem.
  Namely, given a bipartite graph $G = (V, E)$, with $V = X \cup Y$, we create the flow network $\mathcal{F}_G = (G', s, t, c)$ where
  - $G' = (V', E')$ with $V' = V \cup \{s, t\}$ for nodes $s, t \notin V$
  - $E' = E \cup \{(s, x) | x \in X\} \cup \{(y, t) | y \in Y\}$
  - $c(e) = 1$ for all $e \in E'$.

  .

Claim: Every matching M in G gives rise to an integral flow f_M in $\mathcal{F}_G$ with $val(f_M) = |M|$; conversely every integral flow f in $\mathcal{F}_G$ gives rise to a matching M_f in G with $|M| = val(f)$.

- Hence a maximum size bipartite matching can be computed in time $O(m\sqrt{n})$ using the Hopcroft and Karp adatpion of the blocking path algorithm.
- Similar ideas allow us to compute the maximum number of edge (or node) disjoint paths in directed and undirected graphs.

Additional comments on maximum bipartite matching

- There is a nice terminology for augmenting paths in the context of matching. Let M be a matching in a graph $G = (V, E)$. A vertex v is *matched* if it is the end point of some edge in M and otherwise if is *free*. A path π is an alternating path if the edges in π alternate between M and $E - M$.
- Abusing terminology briefly, an augmenting path (relative to a matching M) is an alternating path that starts and ends in a free vertex. An augmenting path in a graph shows that the matching is not a maximum and can be immediately improved.
- Clearly the existence of an augmenting path in a bipartite graph G corresponds to an augmenting path in the flow graph $\mathcal{F}_G$ used to show that bipartite matching reduce to flows.

The Konig-Egevary Theorem

- In any graph, the size of every vertex cover must be at least as large as the size of any matching.

Theorem: Konig [1931], Egervary [1931]

In a bipartite graph, the minimum size of a vertex cover equals the size of a maximum matching.

- In a bipartite graph, a minimum vertex cover and maximum size matching can be efficiently computed at the same time.
- The Konig-Egervary theorem and the efficient computation of the min vertex cover/maximum matching is a key ingrediant of the Hungarian method for computing an optimal matching in a weighted bipartite matching, which is sometimes called the assignment problem.

The weighted bipartite matching problem

- Can the flow algorithm for unweighted bipartite matching be modified for weighted bipartite matching?
- The obvious modification would set the capacity of $< x, y > \in E$ to be its weight $w(x, y)$ and the capacity of any edge $< s, x >$ could be set to $\max_y\{w(x, y)\}$ and similarly for the weight of edges $< y, t >$.
- Why doesnt this work?
- It is true that if G has a matching of total weight W then the resulting flow network has a flow of value W.
- But the converse fails! Why?

Setting up the Hungarian Algorithm

- We will see that this method is intimately tied to the linear programming (LPs) and duality. I am not sure about the history of the Hungarian method; it was formalized in 1955 by Kuhn who attributed the method to Hungarians Konig and Egerva'ry.
- Let $G = (X \cup Y), E)$ be a weighted bipartite graph with w_{ij} denoting the weight of edge (x_i, y_j). Without loss of generality we can assume that G is a complete bipartite graph and that $|X| = |Y| = n$.
- A weighted cover is a labelling $(\mathbf{u}, \mathbf{v})$ of the vertices $(u_i = \ell(x_i)$ and $v_j = \ell(y_j))$ such that $u_i + v_j \geq w_{ij}$ for all i, j. (As we will see later, the labels are the dual variables in a natural LP representation of the weighted matching problem.)
- Given a cover, the equality graph $G_{\mathbf{u},\mathbf{v}}$ is the graph whose edges correspond to those (x_i, y_j) such that $u_i + v_j = w_{ij}$.

The Hungarian Algorithm: Kuhn after Konig and Egervary

We will explain the algorithm when we get to LP duality but for now here is a statement taken from West's text :

The Hungarian Algorithm

```
Let $(\mathbf{u}, \mathbf{v})$ be any initital cover
Let $M$ be a maximum matching in the equality graph
While  $M$ is not a perfect matching
    Let $Q$ be a vertex cover of size $|M|$.
    Let $R = X \cap Q$ and $T = Y \cap Q$
    Let $\epsilon = \min\{u_i + v_j - w_{ij} : x_i \in X - R, y_j \in Y - T\}$
    $u_i := u_i - \epsilon$ for $x_i \in X - R$
    $v_j := v_j + \epsilon$ for $y_j \in T$
    Form the new equality graph and maximum matching
End While
Return $M$ as a maximum weighted matching.
```

The metric labelling problem

We consider a problem well motivated by applications in, for example, information retrieval. (See Kleinberg and Tardos text)

The metric labelling problem

Given: graph $G = (V, E)$, a set of labels $L = \{a_1, \ldots, a_r\}$ in a metric space M with distance δ, and a cost function $\kappa : V \times L \rightarrow \Re^{\geq 0}$. The goal is to construct an assignment α of labels to the nodes V so as to minimize $\sum_{i \in V} \kappa(i, \alpha(i)) + \sum_{(i,j) \in E} p_{i,j} \cdot \delta(\alpha(i), \alpha(j))$

The idea is that κ represents a cost for labelling the node (e.g. a penalty for a bad classification of a web page), p represents the importance of that edge (e.g. where in a web page a particular link occurs) and δ represents the (basic or unweighted) cost of giving different labels to nodes that are related (e.g. the penalty for different labellings of web pages that are linking to each other or otherwise seem to be discussing similar topics.

The binary label case

- A case of special interest and the easiest to deal with is when the metric is the binary $\{0, 1\}$ metric; that is, $\delta(a, b) = 1$ if $a \neq b$ and 0 otherwise. (When there are only two labels, the binary $\{0, 1\}$ metric is the only metric.)
- The case of two labels suggests that the problem might be formulated as a min cut problem. Indeed this can be done to achieve an optimal algorithm when there are only two labels. For more than two labels, the binary metric case remains NP hard but a there is a 2-approximation via a local search algorithm that uses min cuts to search a local neighbourhood.

The case of two labels

- The problem for two labels can be restated as follows: find a partition $V = A \cup B$ of the nodes so as to minimize $\sum_{i \in A} b_i + \sum_{j \in B} a_j + \sum_{(i,j) \in A \times B} p_{i,j}$
- We transform this problem to a min cut problem as follows: construct the flow network $\mathcal{F} = (G', s, t, c)$ such that
  - $G' = (V', E')$
  - $V' = V \cup \{s, t\}$
  - $E' = \{(u, v) | u \neq v \in V\} \cup \{(s, u) | u \in V\} \cup \{(u, t) | u \in V\}$
  - $c(i, j) = c(j, i) = p_{i,j}; c(s, i) = a_i; c(i, t) = b_i$

Claim:

For any partition $V = A \cup B$, the capacity of the cut $c(A, B) = \sum_{i \in A} b_i + \sum_{j \in B} a_j + \sum_{(i,j) \in A \times B} p_{i,j}$.

-

Flow networks with costs

We now augment the definition of a flow network $\mathcal{F} = (G, s, t, c, \kappa)$ where $\kappa(e)$ is the non negative cost of edge e. Given a flow f, the cost of a path or cycle π is $\sum_{e \in \pi} \kappa(e) f(e)$.

MIn cost flow problem

Given a network $\mathcal{F}$ with costs, and given flow f in $\mathcal{F}$, the goal is to find a flow f of minimum cost. Sometimes we are only interested in a min cost max flow.

- Given a flow f, we can extend the definition of an augmenting path in $\mathcal{F}$ to an augmenting cycle which is just a simple cycle (not necessarily including the source) in the residual graph G_f.
- If there is a negative cost augmenting cycle, then the flow can be increased on each edge of this cycle which will not change the flow (by flow conservation) but will reduce the cost of the flow.
- A negative cost cycle in a directed graph can be detected by the Bellman Ford DP for the single source shortest path problem.

Bellman-Ford DP for shortest path

- Dijkstra's single source least cost path algorithm is very efficient but is restricted to directed graphs where all edges are non-negative.
- The definition of a least cost path is well defined as long as there are no negative directed cycles. If all cycles have positive cost, then a least cost path must be a simple path.
- The Bellman-Ford single source least cost path algorithm DP algorithm is less efficient itan Dijkstra but works correctly as long as there are no negative cycles.
- Bellman-Ford is based on the following semantic array:
  $C[i, v]$ is the least cost of a simple path π from source s to v having path length at most i.
- Moreover, Belmman-Ford allows us to detect negative cycles. How?

Weighted interval scheduling on m machines

We have seen that :

1. For the unweighted interval scheduling problem on m machines, the "best fit" EFT greedy algorithm is optimal.
2. For $m = 1$, there is an optimal DP or priority stack (i.e. local ratio) algorithm (again using the EFT ordering) that optimally solves the weighted problem.
3. For arbitrary m, the local ratio algorithm has approximation ratio $2 - \frac{1}{m}$ and no fixed order priority stack algorithm can be an optimum. An optimal priority pBT algorith requires time $O(n^m)$.
4. Arkin and Silverberg [1987] achieve time $O(n^2 \log n)$ by reducing the m machine weighted interval problem to a min cost max flow problem.
5. It can also be seen that the weighted version of m machine interval scheduling problem is polynomial time since the problem can be expressed as an integer program (IP) which is totally unimodular.
6. Yannakakis and Gavril [1987] show that the Maximum m colourable subgraph problem is NP hard for split graphs (which are chordal).