#project_tcs [2021-04-26]

[P. Lu et al. SOSA 2021]

Generalized Sorting with Predictions

Presenter: Jongseo Lee (leejseo)

Generalized Sorting Problem

Sorting as a problem over graph

Consider sorting n elements $A_1, A_2, \cdots, A_n$. Define followings:

- DAG $\vec{G}$ with n vertices. ($\vec{G}$ is a tournament.)
  - $(i, j) \in \vec{E}$ if and only if $A_i < A_j$
- A graph G with n vertices. (G is a complete graph.)
  - $\{i, j\} \in E$ if and only if $(i, j) \in \vec{E} \vee (j, i) \in \vec{E}$
- $\vec{E}' := \varnothing$
- Action: probe $\{i, j\} \in E$, reveal its direction, add one of (i, j) or (j, i) to $\vec{E}'$.
- Goal: find a Hamiltonian path of $\vec{G}$.

Then, the performance (number of comparisons) of the algorithm is $\left|\vec{E}'\right|$ after finding a Hamiltonian path.

Generalized Sorting Problem

Consider sorting n elements $A_1, A_2, \cdots, A_n$. Define followings:

- DAG $\vec{G}$ with n vertices.
    - $(i, j) \in \vec{E}$ if and only if $A_i < A_j$
- A graph G with n vertices.
    - $\{i, j\} \in E$ if and only if $(i, j) \in \vec{E} \vee (j, i) \in \vec{E}$
- $\vec{E}' := \varnothing$
- Action: probe $\{i, j\} \in E$, reveal its direction, add one of (i, j) or (j, i) to $\vec{E}'$.
- Goal: find a Hamiltonian path of $\vec{G}$.

Previous work introduced a randomized algorithm requiring $\tilde{O}(n^{3/2})$ probes.

Examples

1. Ordinary sorting - G is a complete graph and G' is a tournament & DAG. $|\vec{E}'| \geq \Omega(n \log n)$.
2. Bolt-nut matching - Given a set of n bolts and n nuts of different sizes. There's 1-1 mapping between them. A bolt's size can only be compared with a nut and vice versa. We want to match them (and arrange them in a line).

   In this case, G is a complete bipartite graph. Well-known quick sort solution gives us $|\vec{E}'| = O(n \log n)$ in average. An algorithm with $|\vec{E}'| = \Theta(n \log n)$ exists.

Generalized Sorting with Predictions

Predictions

This paradigm tries to use "prediction" from machine learning, which may allow us to create more efficient algorithms. The algorithm should satisfy two things:

1. If prediction is good, we aim for the algorithm with near theoretical optimal performance.
2. Performance of the algorithm shouldn't be worse than prediction-less algorithm even if prediction is bad.

Generalized Sorting with Predictions

Instance of the problems:

- $\vec{G} = (V, \vec{E})$, $G = (V, E)$: same as in the previous problem
- $\vec{P}$: it represents the predicted direction of the edge
  - For each $\{u, v\} \in E$, exactly one of (u, v) and (v, u) belongs to $\vec{P}$
- n will denote $|V|$ and w will denote $|\vec{P} - \vec{E}|$, that is "how much the prediction is wrong"
- Action: probe $\{i, j\} \in E$, reveal its direction, add one of (i, j) or (j, i) to $\vec{E}'$.
- Goal: find a Hamiltonian path of $\vec{G}$.

Performance is evaluated by $|\vec{E}|$ in terms of n and w.

Some Notations

- For $\vec{E}' \subset \vec{E}$, $<_{\vec{E}'}$ is defined as $u <_{\vec{E}'} v \iff$ path from u to v in $\vec{E}'$ exists
- For $V' \subset V$, $<_{V'}$ is defined as $u <_{V'} v \iff$ path from u to v in $\vec{E}'$ only passing vertices in V' and edges in $\vec{E}'$ exists
- $\mathcal{N}_{in}(\vec{G}_P, u)$: the set of in-neighbors of u in $\vec{P}$
- S_u: the "real" in-neighbors of u among $\mathcal{N}_{in}(\vec{G}_P, u)$.
- At the fixed moment, T_u is the set of elements of $\mathcal{N}_{in}(G_{\vec{P}}, u)$ that the prediction is not known to be wrong.

Results

This paper introduces two results:

- $O(n \log n + w)$ randomized algorithm
- $O(nw)$ deterministic algorithm

$O(n \log n + w)$ **randomized algorithm**

Define:

- A: $\{u : \text{direction of all edges between } \mathcal{N}_{in}(\vec{G}_P, u) \text{ is known}\}$.

Starting from $A = \varnothing$, the algorithm iteratively expends A until $A = V$.

Ideal Vertex

A vertex u is an ideal vertex if $T_u \subseteq A$ and $<_A$ restricted to T_u forms a total order.

We will try to increase the size of A by repeatedly adding an **ideal vertex** to A.

Ideal Vertex

A vertex u is an ideal vertex if $T_u \subseteq A$ and $<_A$ restricted to T_u forms a total order.

We will try to increase the size of A by repeatedly adding an **ideal vertex** to A.

Question. When can we add an ideal vertex u to A?

Ideal Vertex

Recall definition of A:

- $A = \{u : \text{direction of all edges between } \mathcal{N}_{in}(\vec{G}_P, u) \text{ is known}\}$.

To add a vertex u to A, direction of edges between T_u and u should be determined.

(Because, we already know that direction of edges between $\mathcal{N}_{in}(\vec{G}_P, u) - T_u$ and u are mis-predicted.)

Ideal Vertex

To add a vertex u to A, direction of edges between T_u and u should be determined.

For a largest vertex x of T_u, probe (x, u).

- Case 1 - our prediction is right: direction of edges between T_u and u are correct from transitivity
- Case 2 - our prediction is wrong: $T_u := T_u - x$ and repeat this

Ideal Vertex

To add a vertex u to A, direction of edges between T_u and u should be determined.

For a largest vertex x of T_u, probe (x, u).

- Case 1 - our prediction is right: direction of edges between T_u and u are correct from transitivity
- Case 2 - our prediction is wrong: $T_u := T_u - x$ and repeat this

So, if we can find out new ideal vertex repeatedly, we need $O(n + w)$ more probes to add them.

However, we can't guarantee that an ideal vertex always exists.

Active Vertex

A vertex u is an ~~ideal~~ active vertex if

- $\cancel{T_u \subseteq A}$ $S_u \subseteq A$
- $<_A$ restricted to $\cancel{T_u}$ S_u is a total order

Note that since $S_u \subseteq T_u$, every ideal vertex is an active vertex.

Active Vertex

Proposition. If $V - A \neq \varnothing$, an active vertex exists.

Proof. Let $v_1 \to v_2 \to \cdots v_n$ be a Hamiltonian path of $\vec{G}$. Pick the smallest index k that has not been added to A yet. Then, $S_{v_k} \subseteq \{v_1 \cdots, v_{k-1}\} \subseteq A$ and $<_A$ is clearly a total order in S_{v_k}. $\square$

Active Vertex

Suppose we know an active vertex.

- **Question.** How can we make it into an ideal vertex? (Ideal vertices are what we are adding to A.)

Active Vertex

Suppose we know an active vertex.

- **Question.** How can we make it into an ideal vertex? (Ideal vertices are what we are adding to A.)

To do that, we need to identify vertices in $T_u - S_u$ but S_u is invisible to us.

Certificate

Let u be a non-ideal vertex. What in-neighbors of u makes it non-ideal?

1. $v \in T_u$ s.t. $v \notin A$
   - ($T_u \not\subseteq A$)
2. $v_1, v_2 \in T_u \cap A$ s.t. there's no path between them (i.e. $v_1 \not<_A v_2$ and $v_2 \not<_A v_1$)
   - $<_A$ restricted to T_u is not a total order

Certificate

Now, we will try to make an active vertex u into an ideal vertex. Consider three cases:

1. $\exists v \in T_u$ s.t. $v \notin A$
   - Probe (v, u), which is a mis-predicted edge
2. $\exists v_1, v_2 \in T_u \cap A$ s.t. there's no path between them
   - Probe (v_1, u) and (v_2, u), at least one of them is mis-predicted
3. After repeating 1 or 2, there's no element of T_u satisfying the condition of 1 or 2.
   - In this case, we are done!

Certificate

Let u be an inactive vertex. In this case direction of (v, u) or $(v_1, u), (v_2, u)$ may be correct or incorrect.
But, if they are correct, it tells us that u is inactive. Now, we define **certificate** as below:

1. Type-1 certificate: $v \in S_u - A$
2. Type-2 certificate: $v_1, v_2 \in S_u \cap A$ s.t. there's no path between v_1 and v_2 in A

Certificate

If a certificate exists for a vertex u, u is inactive, so we don't need to probe other edges coming in to u until A is updated and a certificate becomes invalid.

The exact moment when certificate becomes invalid is:

1. Type-1 certificate v: when v is added to A
2. Type-2 certificate (v_1, v_2): when A is get updated and a path between v_1 and v_2 appears in $G[A]$

The Algorithm

Starting from $A = \varnothing$, while $A \neq V$ do following:

Pick a vertex $u \in V - A$ without valid certificates (tie-break by indices)

1. If u is an ideal vertex:

   We will use strategy from Ideal Vertex section

   Pick maximal element x w.r.t. $<_A$; probe (x, u)

   - If direction of (x, u) is correct: $A := A \cup \{u\}$

2. Otherwise

   i. If $\exists v \in T_u - A$, randomly pick v and probe (v, u)

   ii. Randomly pick v_1, v_2 with $v_1 \not<_A v_2$ and vice versa

   Probe (v_1, u) and (v_2, u)

We use randomness to prevent probing too many edges in 2.

Performance

Proposition. Although the algorithm uses random, all vertices are added to A with fixed order.

Proof. "Activeness" of a vertex only relys on A and a vertex added to A is always the active vertex with smallest index. $\square$

Performance

We will use some proposition from very simple probability theory without proof.

Proposition. For $i = 1, 2, \cdots, n$, randomly pick Z_i from $[i]$. Take the unique permutation Y s.t. Y_i is Z_i-th largest element of $\{Y_1, \cdots, Y_i\}$. Then, Y is uniformly distributed over the set of permutations of $[n]$.

Proposition. For a random permuation Y of $[n]$, the number of Y_i s.t. $Y_i = \max_{j \le i} Y_j$ does not exceed $6 \ln n + 6$ with probability $\geq 1 - \frac{1}{2n^2}$.

Performance

Theorem. With high probability, the algorithm probes at most $O(n \log n + w)$ edges.

Proof.

While the algorithm repeats its routine, exactly one of following case appears:

1. Add a vertex u to A.
2. Find a certificate of u where u does not have a valid certificate now.
3. FInd a mis-predicted edge.

Performance

Theorem. With high probability, the algorithm probes at most $O(n \log n + w)$ edges.

Proof.

1. Add a vertex u to A.
2. Find a certificate of u where u does not have a valid certificate now.
3. FInd a mis-predicted edge.

At each case, we probe ≤ 2 edges, so it suffices to estimate the number of appearance of each cases.

Clearly, 1 appears at most n times and 3 appears at most $w(= |\vec{P} - \vec{E}|)$ times.

Performance

Theorem. With high probability, the algorithm probes at most $O(n \log n + w)$ edges.

Proof.

1. Add a vertex u to A.
2. Find a certificate of u where u does not have a valid certificate now.
3. FInd a mis-predicted edge.

We will estimate how much time 2 appears.

- Key idea: certificates of a vertex becomes invalid in some fixed order.

Performance

Theorem. With high probability, the algorithm probes at most $O(n \log n + w)$ edges.

Proof.

2. Find a certificate of u where u does not have a valid certificate now.

Fix a vertex u, number type-1 certificates $1, 2, \cdots, |S_u|$ in the order "found" by the algorithm.

- Note: at the initial state, every vertices of S_u are type-1 certificates of u.

Performance

Theorem. With high probability, the algorithm probes at most $O(n \log n + w)$ edges.

Proof.

2. Find a certificate of u where u does not have a valid certificate now.

Define the permutation $Y_{u,1}, \cdots, Y_{u,|S_u|}$ in the order they became invalid. (Certificate i is $Y_{u,i}$-th earliest to become invalid.) We know that the permutation Y is uniformly distibuted.

If $Y_{u,i} < \max_{j \le i} Y_{u,j}$, i has became invalid before the algorithm found it.
i.e. The algorithm found at most $\left|\{i : Y_{u,i} = \max_{j \le i} Y_{u,i}\}\right|$ valid type-1 certificates of u.

Therefore, the algorithm found at most $6 \ln n + 6$ valid type-1 certificates of u with probability $\ge 1 - \frac{1}{2n^2}$.

Performance

Theorem. With high probability, the algorithm probes at most $O(n \log n + w)$ edges.

Proof.

Therefore, the algorithm found at most $6 \ln n + 6$ valid type-1 certificates of u with probability $\geq 1 - \frac{1}{2n^2}$.

Applying union bound, the algorithm found at most $6n \ln n + 6n$ type-1 certificates with probability $\geq 1 - \frac{1}{2n}$.

In the similar way, we can deduce that the algorithm found at most $12n \ln n + 6n$ type-2 certificates with probability $\geq 1 - \frac{1}{2n}$.

Therefore, the algorithm probes $O(n \log n + w)$ edges with probability $\geq 1 - \frac{1}{n}$. $\square$

$O(nw)$ **algorithm**

Definitions

We will try to "correct" mis-predicted edges, by probing at most $O(n)$ edges to fix one edges.

At the specific moment:

- $\vec{E}'$: the set of edges that we already probed
- $\vec{G}_C = (V, \vec{P}_C)$: the graph that we already corrected:

$$\vec{P}_C = \{(v, u) \in \vec{P} : (u, v) \notin \vec{E}'\} \cup \vec{E}'$$

The Simple Strategy

We will maintain $\vec{G}_C$ and consider following cases while there's no Hamiltonian path in $\vec{E}'$.

1. If $\vec{G}_C$ has a cycle
2. Otherwise: $\vec{G}_C$ is a DAG

The Simple Strategy

We will maintain $\vec{G}_C$ and consider following cases while there's no Hamiltonian path in $\vec{E}'$.

1. If $\vec{G}_C$ has a cycle
   - Since $\vec{G}$ is a DAG, we may probe $\le O(n)$ edges of a cycle.
2. Otherwise: $\vec{G}_C$ is a DAG
   i. If $\exists$ two vertices v_1, v_2 with in-degree 0
      Since $\vec{G}$ has the unique vertex with in-degree 0, we may probe $\le O(n)$ edges incident to v_1 or v_2.
   ii. Otherwise
      Topological sort $\vec{G}_C$ and get $a_1, \cdots, a_n$, and we may probe (a_i, a_{i+1}) for every i.

Clearly, it works using only $O(nw)$ probes.

Conclusion

Conclusion

We have discovered two algorithms for generalized sorting with predictions:

1. A randomized algorithm using $O(n \log n + w)$ probes
2. A deterministic algorithm using $O(nw)$ probes