A Near Linear-time Approximation Framework for Makespan Scheduling of Unit Jobs [⋆]

Ananth Goyal[1]

UC Berkeley CA , USA
ananthgoyal@berkeley.edu

Abstract. We present an $O((m+n)\log n)$ combinatorial approximation framework for Unit Job Scheduling with precedence constraints as an alternative to standard PTAS approaches. We are tasked with optimally scheduling n jobs with precedence constraints given $G(V,E)$ and k machines/processors so as to minimize the completion time (makespan) of the final job; this problem is known to be NP-Hard. The primary idea of our algorithm is to model the problem graphically and use price functions $\{\phi : V \to \mathbb{R}_0^+\}$ that map vertices to real numbers to break decision-ties effectively. The algorithm yields a solution that is $O\left(\mathsf{OPT}\alpha_\phi^{-1}\right)$, where α_ϕ is the percent accuracy of the price function ϕ being used. Unlike normal PTAS approaches the runtime does not depend on the error bound; but rather how precise ϕ is for G. Additionally, we show that in the word RAM model we can achieve a bound of $O(m + n\log\log n)$ when we are guaranteed all prices of vertices be integers.

Keywords: Graph Algorithms · Approximation Algorithms · Scheduling · Makespan

1 Introduction

The following paper considers the unit task scheduling problem (UTS). We are given k processors/machines and a graph $G(V,E)$ composed of 1 (or several) Directed Acyclic Graphs (DAG) where vertices represent tasks/jobs and edges represent a precedence constraint (i.e edge (u,v) implies task v requires task u to be scheduled prior to it.) At most k jobs can be completed at any time-step and every job requires only a single time-unit to complete. The goal is to compose a schedule that minimizes the makespan of all jobs. The UTS problem is a subset of the more general NP-Complete scheduling problem that can involve tasks with variable completion times, profit maximization, deadlines, etc. However, even in this simplified case, the decision version of the problem remains NP-Complete[Ull75]. Finding the minimum time possible for any graph (the optimization version) is NP-Hard, and is the focus of this paper. For any G and k, no efficient polynomial time algorithm exists that can solve this problem nor do we expect one to.

1.1 Related Work

Despite the intractability of the general problem there have been advancements in both brute-force approaches and approximation schemes. Recently, Nederlof et al. developed an enhanced brute-force approach with a bound of $O(1.995^n)$ [NSW22]. In terms of approximation algorithms, Levey and Rothvoβ showed a $(1+\epsilon)$ approximation algorithm with cost $\exp\left(\exp\left(O(\frac{k^2}{\epsilon^2}\log^2\log n)\right)\right)$ when $k = O(1)$ [LR16]; which was later improved by Garg [Gar17]. The current best bound is $n^{O\left(\frac{k^4}{\epsilon^3}\log^3\log n\right)}$ by Li [Li21]. Unlike our approach, these are all traditional PTAS approaches where the runtime depends on the desired ϵ (error) bound. It can also be shown that the classic Graham list-approximation scheme can be used for this problem and achieve the $2-\frac{1}{k}$ bound on OPT [Gra66].

1.2 Result

Theorem 1.1 *There exists an $O((m+n)\log(n))$ approximation framework for the (UTS) problem with precedence constraints that achieves $O(\mathsf{OPT}(\alpha^{-1}))$, where OPT indicates the optimal/minimal possible time for a given $G(V,E)$ and k.*

2 Preliminaries

2.1 Notation

Since our algorithm alters G we refer to $G^i(V^i,E^i)$ as the state of G on iteration i. We will refer to $n = |V|$, $m = |E|$, and k as the number of machines/processors given. It is standard in scheduling theory to indicate m as the number of machines; however, since we are modeling the problem graphically we chose to preserve the standard graph notation of n vertices and m edges.

Assumption 2.1 *We assume $n \geq k$ since for this problem having more machines than jobs is equivalent to the instance where $n = k$.*

Throughout this paper we iterate on out edges of any arbitrary vertex u with notation $\forall(u,v) \in E$; since u is already specified this is not equal to $\forall e \in E$ which indicate all edges in E. We refer to the set of root nodes (those that are available to be scheduled) on $t = i$ as λ_i. We refer to OPT as the minimal time possible for the given graph and OPT^S as the corresponding schedule/configuration where $\mathsf{OPT}^S[i] \subseteq V$ (i.e the optimal

[⋆] CS 270 Final Project Paper

jobs to be scheduled at time $t = i$). Since it is possible for their to be distinct optimal configurations but with identical makespan we identify $\mathcal{OPT}$ as the set of all optimal configurations for G. The algorithm returns S and T. S is the set which contains the outputted configuration such that $S[i] \subseteq V$ to be scheduled at time $t = i$. T is the makespan/completion time of the configuration; $T = |S|$. We use $\mathbb{R}_0^+$ to compactly identify the set of non-negative real numbers.

The representation of this problem in 3-field notation is $P|prec, p_j = 1|C_{\max}$. This stands for the scheduling problem with any variable machines, tasked with single time-unit jobs with precedence constraints, with the objective function to minimize the completion time of the final task. $Pk|prec, p_j = 1|C_{\max}$ indicates the specific case where the number of machines is fixed at any k. It was shown that this problem when $k = 2$, or formally $P2|prec, p_j = 1|C_{\max}$ is polynomial time solvable [FKN69].

Additionally, the following bounds will be needed throughout this paper particular for analysis:

Lemma 2.1 $k! \leq \sqrt{2\pi k}(\frac{k}{e})^k$

We use lemma 2.1, known as the Stirling Approximation, without proof.

Lemma 2.2 $\binom{n}{k} \leq (\frac{en}{k})^k$ *for any $k < n$.*

Proof.

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \leq \frac{n^k}{k!} \leq \frac{n^k}{\sqrt{2\pi k}(\frac{k}{e})^k} \leq \frac{n^k}{(\frac{k}{e})^k} = (\frac{en}{k})^k$$

2.2 Redundant Edges and Transitive Reduction

Consider edges $\{(u, v), (v, w), (u, w)\} \subseteq E$. Task w requires the completion of both u and v; and task v requires the completion of u. We can say edge (u, w) is redundant, since it's existence is implied by other edge(s)$\{(u, v), (v, w)\}$. Since there are no edge weights in this problem, removing (u, v) has no impact on the optimal schedule. However, in general, redundant edges like these can can over complicate the problem.

Definition 2.1 *(Transitive Reduction) The process of removing redundant edges in G, resulting in a graph G' with the minimum edges required to preserve all reachability queries in G. This is solvable in polynomial time.*

The problem of transitively reducing general graphs has been around since the 1950's. It was shown to be computationally equivalent to matrix multiplication [AGU72] which is known to take $O(n^{2.376})$ with Coppersmith-Winnograd [CW87] (which is roughly around the fastest known bound); in either case, using a matrix multiplication method is relatively slow and can take more than quadratic time. There are algorithms designed specifically for DAG's that transitively reduce graphs quicker. Examples are the buTR algorithm which costs $O(\Delta m)$ [ZZY+17] and Tang et al.'s approach costing $O(\theta d^S_{\max} n)$ [TZQ+20]. Here Δ indicates the average number of visited nodes for each processed node in the resultant graph G'. The maximum out degree of any vertex in G is indicated by d_{max} and θ is the cost of answering reachability queries.

Assumption 2.2 *We assume that the given graph $G(V, E)$ is transitively reduced.*

Claim 2.1 *The problem, (UTS with precedence constraints), remains NP-Complete even when G is guaranteed to be transitively reduced. (Unless P = NP).*

Proof. Assume for contradiction that this isn't true and that there exists a polynomial-time algorithm to solve this reduced case. We know there exists a polynomial time DAG transitive reduction algorithm (from Definition 2.1). If the problem wasn't NP-Complete after reduction, then given any G we could simply apply any existing reduction algorithm and use a polynomial time algorithm to return the minimal time for all possible schedules. And thus, the general problem before reduction would still be polynomial-time solvable–which we know not to be true (unless P = NP). We have arrived at a contradiction.

2.3 Naive Greedy Approach

Our algorithm is relatively simple and follows standard combinatorial techniques. The baseline is implemented as a greedy approach, but is enhanced using price functions (4). By definition, a standard greedy approach would choose the locally best option at each iteration. However, on any iteration i, there are at most $\binom{|\lambda_i|}{k}$ distinct choices to choose from (where λ_i is the set of vertices available) all of which satisfy local optimality (Assuming $k \leq |\lambda_i|$) since the maximal number of tasks that could be scheduled will be scheduled given the state. It then follows that a trivial greedy approach would choose any one of these with uniform probability. Implying that the probability that the optimal (or potentially one of the optimal) selections is made is:

$$Pr\left(S[i] \in \bigcup_{\mathsf{OPT}^s \in \mathcal{OPT}} \mathsf{OPT}^s[i]\right) \leq \frac{|\mathsf{OPT}^s \in \mathcal{OPT}|}{\binom{|\lambda_i|}{k}} \geq \binom{|\lambda_i|}{k}^{-1} \geq \binom{n}{k}^{-1} = \Omega\left(\frac{n}{k}\right)^{-k} = \Omega\left(\frac{k}{n}\right)^{k}$$

Note that the reason the first term is upper bounded by the second and not equal to it because it's possible and in some cases, even likely, that the optimal jobs that could be scheduled at time $t = i$ are not available for the greedy approach.

Definition 2.2 *(Decision-tie) When the algorithm has to decide which k vertices of the λ_i to schedule at time $t = i$. There are $\binom{|\lambda_i|}{k}$ possible options.*

2.4 Naively Finding OPT

Conversely, naively finding the optimal solution would require an exhaustive search of checking each option in λ_i, and subsequently $\binom{|\lambda_{i+1}|}{k}$, and so forth. To bound this we use $\lambda_{i,j_1,\cdots,j_n}$ to represent the unique selection of vertices for iteration/depth i given the previous set of selections $j_1, j_2, \cdots, j_{i-1}$ and goes until the maximal amount of depth (which is always at most n) to view all possibilities P. Putting this together we get:

$$P \leq \binom{|\lambda_1|}{k} + \sum_{j_1=1}^{\binom{|\lambda_1|}{k}} \left(\binom{|\lambda_{2,j_1}|}{k} + \cdots + \sum_{j_{n-1}=1}^{\binom{|\lambda_{n,j_1,\cdots,j_{n-1}}|}{k}} \binom{|\lambda_{n,j_1,\cdots,j_n}|}{k} \right)$$

$$\leq \left(\frac{e|\lambda_i|}{k}\right)^k + \sum_{j=1}^{\binom{|\lambda_i|}{k}} \left(\left(\frac{e|\lambda_{2,j_1}|}{k}\right)^k + \cdots + \sum_{z=1}^{\binom{|\lambda_{n,j_1,\cdots,j_{n-1}}|}{k}} \left(\frac{e|\lambda_{n,j_1,\cdots,j_n}|}{k}\right) \right)$$

$$\leq \left(\frac{en}{k}\right)^k + n\left(\left(\frac{en}{k}\right)^k + \cdots + \left(n\left(\frac{en}{k}\right)^k\right)\right)$$

$$\leq O\left(\frac{n}{k}\right)^{nk} = O\left(\frac{n}{k}\right)^n \text{ (since } n \geq k).$$

This is by no means the tightest bound for any brute force search approach (as seen earlier there exists an $O(1.995^n)$ algorithm; however, it provides some insight on the naive approach's runtime bound.

2.5 Data Structures and Operation Costs

In addition to the data structural representation of G, we need additional data structures. The first two, Q_{min} and Q_{max}, are priority queues and implemented as binary heaps (min and max respectively). Q_{min} consists of tuples $(v, w(v))$ (vertex v and its weight/in-degree $w(v)$); the second entry $w(v)$ is what the queue performs comparisons on.

Q_{max} consists of tuples $(v, \phi(v))$ (vertex v and its price $\phi(v)$); the second entry is also what the queue performs comparisons on. Q_{max} supports the max-equivalent operations as Q_{min} with the same corresponding cost. For Q_{max} a decrease or increase key operation is not needed for this algorithm since prices are all set prior to the scheduling process. Follow up work can, however, look into adjusting prices during the scheduling process using some online algorithm.

Q^i_{min} and Q^i_{max} represent the state of the queues on iteration i. At the start of any iteration i, Q^i_{min} is extracted repeatedly via $\min\{G^i_{min}\}$ until the weight of the minimum value is not 0; implying all root nodes (those that can be scheduled at time $t = i$) in state G_i are no longer in Q^i_{min}. After each removal, they get inserted into Q^i_{max}. Since $\phi(v)$ can be retrieved in constant time (from an array discussed later) the total combined cost of each removal and insertion is $O(\log n + \log k) = O(\log nk)$. And per the assumption where $n \geq k$, we can just simplify the combined cost bound to just be $O(\log n)$.

In the word RAM model we can achieve an even faster bound for our queue operations if all $\phi(v) \in \mathbb{Z}$. Using Thorup's integer priority queue [Tho03] we find that we can get $O(\log \log n)$ for each removal and $O(1)$ for each insertion and decrease key.

The last set of data structures we need are standard arrays $A_\phi : \forall \phi \in \Phi$. Since $|\Phi| \in O(1)$, the number of these arrays are also $\in O(1)$. We store $\forall v \in V : A_\phi[v] \leftarrow \phi(v)$; insertion and query costs are both $O(1)$.

Lemma 2.3 *While the state of $G^i(V^i, E^i)$ is not empty (i.e. $V^i \neq \emptyset \wedge E^i \neq \emptyset$), $\min\{Q^i_{min}\}$ will always be queried ≥ 1 times.*

Proof. We proceed via induction on the iteration count. For $i = 1$: $G^1(V^1, E^1) = G(V, E)$, since in the first iteration, no vertices and edges from the original graph have been deleted. Since we are guaranteed that $G(V, E)$ be composed of only DAGs, at least one root/source vertex must be present guaranteeing a call to $\min\{Q^i_{min}\}$. We assume for any $i = j$ the property hold. On iteration $i = j + 1$ if the graph is still not empty, at least one vertex must be present without a predecessor since on $i = j$ we were guaranteed the removal of at least one vertex (and edge) which resulted in the new root node(s) or $k < |\lambda_j|$ and thus for $i = j + 1$ there are still left over root nodes from the previous iteration.

3 Price Functions

Our algorithm utilizes price functions, introduced by Johnson [Joh77] and more recently used by Bernstein et al. [BNWN22] as a method to solve the SSSP problem with negative weights in near-linear time. In all cases, a price function is used to map vertices to a real (or integer) value $\{\phi : V \to \mathbb{R}^+_0\}$; this can be interpreted as the importance or criticality of that vertex. Use cases of price functions vary from problem and algorithm, in our case we will use them to break decision ties that cause intractability.

Definition 3.1 *(Price Function) Given graph $G(V, E)$, let $\phi : V \to \mathbb{R}^+_0$, such that $\phi(v)$ is the price of v.*

Definition 3.2 *(Simple Price Function) Any ϕ which follows the structure of $\phi(u) = c + \sum_{(u,v)\in E} (f(v) + \phi(v))$ where c is constant and $f(v)$ is a function of vertex v that costs $O(1)$ time to compute will be referred to as a simple price function.*

Definition 3.3 *(Simple Price Set) Let $\Phi = \{\phi_a, \phi_b, ...\}$ denoting the set of simple price functions; as mentioned previously, it is intended that $|\Phi| = O(1)$.*

We have the following objective function for every iteration i.

$$\max_{S \subseteq \lambda_i, |S| \leq k} \sum_{v \in S} \phi(v) \tag{1}$$

The primary idea here is to find a subset of root vertices (for every state G^i) of size at most k that has the highest net price. We will now present a series of different examples of price functions and discuss their performance.

Claim 3.1 *A perfect price function for a particular instance of $G(V, E)$ will always yield OPT*

Proof. Since all the price function does is map the vertices to real values, its conceivable that a price function exists that maps the highest values to the most critical of nodes so as to maximize the amount of jobs scheduled across all iterations. Since ϕ need not be simple, it could just be a perfect piecewise mapping where the most critical vertices (according to OPT^S) have the highest value.

We will now present some examples of simple price functions that can be used.

3.1 ϕ_a: Pricing by Number of Successors

The following price function is defined as the total number of successors and is calculated using the number of out-edges that are descendants of itself; since G is transitively reduced, this calculation is equal to the number of successive jobs.

$$\phi_a(u) = \sum_{(u,v) \in E} (1 + \phi_a(v)) \tag{2}$$

3.2 ϕ_b : Pricing by Path Lengths to Sink

The following price function is defined to as the total number of unique paths to sink vertices starting at vertex v.

$$\phi_b(u) = 1 + \sum_{(u,v) \in E} \left(\frac{-1}{|(u, v) \in E|} + \phi_b(v) \right) \tag{3}$$

3.3 ϕ_c: Pricing by Dependence

Consider a job v that has constrained by $u_1, u_2, ..., u_k$, regardless on the remaining structure of the graph, accessing what comes after v is not possible unless all u_i from 1 to k are satisfied. Hence, it is imperative to schedule tasks that are highly dependent on other vertices. To accomplish this we can set the price of a vertex to be the sum of the weight (in-degree) of all descendants plus that of it's children. This has an $O(1)$ look up time as required.

$$\phi_c(u) = \sum_{(u,v) \in E} (w(v) + \phi_c(v)) \tag{4}$$

3.4 Composite Pricing

In practice pricing jobs based on a single criterion would have limited efficacy; hence, we introduce the notion of creating a composite function.

Definition 3.4 *(Composite Price Function) Any price function that cannot be represented simply (i.e in $\phi(u) = \sum_{(u,v) \in E} (f(v) + \phi(v))$), and is some combination of a subset of Φ.*

Consider a price function that uses ϕ_a (total successors) and ϕ_b (path lengths to sink) in the form:

$$\phi(u) = \frac{\phi_a(u)}{\phi_b(u)} + \phi_a(u)$$

This can be interpreted as the the average path lengths to sink vertices plus the number of unique successors, which in practice would perform better than a single simple price function. If we substitute our recursive function definitions we get:

$$\phi(u) = \frac{\sum_{(u,v) \in E} (1 + \phi_a(v))}{1 + \sum_{(u,v) \in E} \left(\frac{-1}{|(u,v) \in E|} + \phi_b(v) \right)} + \sum_{(u,v) \in E} (1 + \phi_a(v))$$

It is impossible to reduce this expression into the simple form $\sum_{(u,v) \in E} (f(v) + \phi(v))$ and is thus considered a composite price function. If, however, we didn't use average path lengths ($\frac{\phi_a(u)}{\phi_b(u)}$) but rather just path lengths (ϕ_b) and adding that with the number of successors (ϕ_a) i.e: $\phi(v) = \phi_a(v) + \phi_b(v)$ we can show that this is actually a simple price function (in practice this would not perform as well as the previous example). Substituting the recursive definition we have:

$$\phi(u) = 1 + \sum_{(u,v) \in E} \left(\frac{-1}{|(u, v) \in E|} + \phi_b(v) \right) + \sum_{(u,v) \in E} (1 + \phi_a(v))$$

We can combine terms and plug in our definition for $\phi(u)$:

$$\phi(u) = 1 + \sum_{(u,v) \in E} \left(\frac{|(u, v) \in E| - 1}{|(u, v) \in E|} + \phi(u) \right)$$

This shows that combining simple price functions does not necessarily imply the creation of a true composite price function. It would then suffice to recursively use this definition as opposed to combining two separate definitions as it reduces the hidden constant in the runtime bound.

3.5 Primary Function and Weights for Composite Pricing

We introduce composite pricing with the intention to create a single, final function composed of simple price functions to be the source for breaking decision-ties.

Definition 3.5 *(Weighting Function) Let ω_ϕ be the corresponding weight function for any $\phi \in \Phi$ on any $v \in V$ such that $\omega_\phi(v) \in \mathbb{R}_0^+$. Note that this is different from our definition of the "weight" of a vertex which is just the in-degree of the vertex.*

Definition 3.6 *(Weighting Set) Let Ω be the set of all corresponding weight functions ω_ϕ $\forall \phi \in \Phi$.*

Definition 3.7 *(Primary Function) The composite function $C_\phi(v)$ that is used for breaking ties in the algorithm and is of the following form:*

$$C_\phi(v) = \sum_{\phi \in \Phi} (\omega_\phi(v)\phi(v)) \tag{5}$$

Claim 3.2 *For any $\phi, \phi^* \in \Phi$, it is possible for $\omega_\phi(v) = F(\phi^*(v))$, for any function F as long as $\forall v : F(\phi^*(v)) \geq 0$.*

Seeing why this is valid from a definition standpoint is clear – since $\phi(v) \in \mathbb{R}_0^+$, we would just require that $\forall v : F(\phi^*(v)) \geq 0$. The motivation for why we would want to weight certain price functions using others ones is to reduce the number of simple price functions and thus reduce the required computation time for pricing from a practical standpoint. For example, when we defined $\phi(u) = \frac{\phi_a(u)}{\phi_b(u)} + \phi_a(u)$, we can now say that $\omega_{\phi_a}(u) = \phi_b(u)^{-1}$ and $\omega_{\phi_b}(u) = 1$ and simply use equation (5) as intended.

3.6 Calculating Simple Prices Using Dynamic Programming

Lemma 3.1 *For any simple price function $\phi \in \Phi$ being used, the prices for all vertices can be calculated in $O(m + n)$ time via top down dynamic programming.*

Proof. Any $\phi \in \Phi$ follows the following structure: $\phi(u) = \sum_{(u,v)\in E}(f(v) + \phi(v))$, where $f(v)$ takes $O(1)$ time to calculate. The cost of all $f(v)$ incurs over every outgoing edge of which there are m, yielding a total cost of $O(m)$ for all $f(v)$. Since calculating prices is recursively called per vertex until a sink vertex is reached, without any optimization, the total time would be exponential. If $\phi(v)$ for all v is stored in $A_\phi[v]$ once calculated and used to calculate preceding vertices, then every vertex's price needs to be recursively called only once and on any further situation be queried (Note: this is a very standard top down dynamic programming approach). The total cost of this calculation takes an expected $O(m + n)$ because the $O(1)$ query time on the price arrays. See 4.2 for pseudocode.

Corollary 3.1 *It follows that the total time to calculate all simple prices and the primary function is $O(m + n)$.*

Proof. Since $|\Phi| = O(1)$, calculating simple prices must cost only $O(m+n)$. Then for the composite primary function for all vertices we apply $2|\Phi|$ constant time look ups (for both $\phi(v)$ and $\omega_\phi(v)$); again since $|\Phi| = O(1)$, this also comes out to $O(m + n)$. And hence, the entirety of the pre-processing stage for price calculation and data structure set up takes graphically linear ($O(m + n)$) time.

4 Algorithm Framework and Analysis

4.1 Approx

The primary algorithm (Approx) first initializes all prices, computes the primary composite function with the corresponding weights, and inserts them into the queue. Then for each time unit by choosing the (at most) k root nodes with the highest prices, removes them (and their edges) from the graph and repeats. The following pseudocode is for the primary algorithm *Approx*$(G(V, E), k, \Phi, \Omega)$.

4.2 Price

The following pseudocode is for the recursive subroutine *Price*$(G(V, E), A_\phi, f, v)$ implemented using top-down dynamic programming.

4.3 InDeg

The subroutine *inDeg*$(G(V, E))$ can be implemented in many ways with worst-case linear time cost. If $G(V, E)$ is given in adjacency list representation, we can compute the inverse graph $G^{-1}(V, E)$ (all edges are reversed) in $O(m + n)$ time. Then for all $v : w(v) = |(v, u) \in E|$ which would just be the size of the sub-list at index v. We can then insert $(v, w(v))$ into Q^i_{min}. There are other approaches that could be practically faster but would still be asymptotically insignificant due to other parts of the algorithm.

Algorithm 1: $Approx(G(V,E), k, \Phi, \Omega)$

```
Initialize $i = 1$, $S$, $Q^i_{min}$, $Q^i_{max}$, and $\forall\phi \in \Phi$, $A_\phi$    // $|A_\phi| = n$ but is empty
For all $v \in V$ : insert $(v, w(v))$ to $Q^i_{min}$ via $inDeg(G(V,E))$
foreach $\phi \in \Phi$ do
    $\forall v \in V : A_\phi[v] \leftarrow \phi(v) = \sum_{(v,u)\in E}(f(v) + \phi(u))$ via $Price(G(V,E), A_\phi, v)$
$\forall v \in V : C_\phi[v] \leftarrow \sum_{\phi\in\Phi}(A_\phi[v] \cdot \omega_\phi(v))$    // $\omega_\phi \in \Omega$ matching with $\phi \in \Phi$
while $G^i(V^i, E^i) \neq \emptyset$ do
    while $Q^i_{\min} \neq \emptyset$: $(v, w(v)) \leftarrow \min\{Q^i_{min}\}$ and insert $(v, C_\phi[v])$ to $Q^i_{max}$    // (Lemma 4.1) $\geq 1$ iter
    foreach $j \in \{1, ..., k\}$ do
        If $Q^i_{max} = \emptyset$ then break
        $(v, \phi(v)) \leftarrow \max\{Q^i_{\max}\}$ and insert $v$ to $S[i]$
    foreach $v \in S[i]$ do
        For each $(v,u) \in E$ decrease key of $(u, w(u)) \in Q^i_{min}$ to $(u, w(u) - 1)$
        Remove $v \in V$ and all $(v,u) \in E$
    $i \leftarrow i + 1$
return $S, i$
```

Algorithm 2: $Price(G(V,E), A_\phi, \phi)$

```
Unpack $f, c$ from $\phi$
If $A_\phi[v]$ contains $\phi(v)$ return $\phi(v)$ and terminate    // (Lemma 5.1) Top down memoization
if $|(v,u) \in E| > 0$ then
    $A_\phi[v] \leftarrow \phi(v) = \sum_{(v,u)\in E}(f(v) + Price(G(V,E), A_\phi, f, u))$ and return $\phi(v)$
return $c$    // Default $\phi(v)$ for sink vertices
```

4.4 Runtime Analysis

We start by adding the cost of each component of the algorithm independently. Since we have a constant number of price functions (including the composite function), the time required is still proportional to $O(m+n)$. Each edge contributes to a $Q_{\min}$ decrement and each vertex contributes to an insertion and removal for $Q_{\max}$. Which when combined, gives $O(m + n + m\log n + n(\log n + \log k))$. Removing non-dominant terms and recognizing that $n \geq k$ always, then we have $O(m\log n + n\log n) = O((m+n)\log n)$. As mentioned earlier, in the word RAM, and when all prices are integers, we can reduce this further. Each $Q_{\min}$ decrement and insertion only costs $O(1)$, and each removal costs $O(\log\log n)$. Applying our analysis before but adjusting our bounds we get $O(m + n\log\log n)$

4.5 Flaws and Approximation Metric

One of the biggest flaws with our algorithm is that it isn't a standard PTAS where it's performance would depend on ϵ. And hence, we analyze the performance from an alternative, more abstract perspective. We define some α_ϕ to be the accuracy of the composite ϕ used for the instance $G(V,E)$; recall OPT is the minimal time possible for G. We can say $\alpha_\phi = \frac{T}{\mathsf{OPT}}$, where T is the returned time of the algorithm. The next key steps for future work would be to show for a particular price function $\phi(v)$ that α_ϕ is not arbitrary but is always some constant $\alpha_\phi \in \mathbb{R}, [0,1]$ which is potentially dependent on k.

5 Next Steps

Based on this work, some future work could consist of finding a thorough, high-performing composite price function definition (and naturally, simple price functions). And furthermore, potentially show how it guarantees some bound on OPT that is fixed and known. The purpose of the paper wasn't necessarily to provide the best general composite function; rather, serve as a framework for an alternative approach to standard PTAS for approximating the optimal solution quickly. As mentioned earlier, future work can also consist of finding ways to adjust prices during the scheduling process using an online algorithm that makes price updates based off some newly found metric.

5.1 Relevant Open Problem

Is $P3|p_j = 1, prec|C_{\max}$ polynomial time solvable?

This remains one of the biggest open problems in scheduling theory and more specifically in makespan scheduling of unit jobs.

6 Acknowledgements

I would like to thank Professor Jefferey Ullman and Professor Jelani Nelson for their support and feedback throughout this work. This was submitted as my final project for CS 270 taught by Prof. Jelani Nelson in Spring 2023.

References

AGU72. Alfred V. Aho, Michael R Garey, and Jeffrey D. Ullman. The transitive reduction of a directed graph. *SIAM Journal on Computing*, 1(2):131–137, 1972.

BNWN22. Aaron Bernstein, Danupon Nanongkai, and Christian Wulff-Nilsen. Negative-weight single-source shortest paths in near-linear time. In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 600–611. IEEE, 2022.

CW87. Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 1–6, 1987.

FKN69. M Fujii, T Kasami, and K Ninomiya. Optimal sequencing of two equivalent processors. *SIAM Journal on Applied Mathematics*, 17(4):784–789, 1969.

Gar17. Shashwat Garg. Quasi-ptas for scheduling with precedences using lp hierarchies. *arXiv preprint arXiv:1708.04369*, 2017.

Gra66. Ronald L Graham. Bounds for certain multiprocessing anomalies. *Bell system technical journal*, 45(9):1563–1581, 1966.

Joh77. Donald B Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM (JACM)*, 24(1):1–13, 1977.

Li21. Shi Li. Towards ptas for precedence constrained scheduling via combinatorial algorithms. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2991–3010. SIAM, 2021.

LR16. Elaine Levey and Thomas Rothvoss. A (1+ epsilon)-approximation for makespan scheduling with precedence constraints using lp hierarchies. In *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*, pages 168–177, 2016.

NSW22. Jesper Nederlof, Céline MF Swennenhuis, and Karol Wegrzycki. Makespan scheduling of unit jobs with precedence constraints in $o(1.995\tilde{n})$ time. *arXiv preprint arXiv:2208.02664*, 2022.

Tho03. Mikkel Thorup. Integer priority queues with decrease key in constant time and the single source shortest paths problem. In *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, pages 149–158, 2003.

TZQ$^+$20. Xian Tang, Junfeng Zhou, Yaxian Qiu, Xiang Liu, Yunyu Shi, and Jingwen Zhao. One edge at a time: A novel approach towards efficient transitive reduction computation on dags. *IEEE Access*, 8:38010–38022, 2020.

Ull75. Jeffrey D. Ullman. Np-complete scheduling problems. *Journal of Computer and System sciences*, 10(3):384–393, 1975.

ZZY$^+$17. Junfeng Zhou, Shijie Zhou, Jeffrey Xu Yu, Hao Wei, Ziyang Chen, and Xian Tang. Dag reduction: Fast answering reachability queries. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 375–390, 2017.