Scheduling with Conflicts on Bipartite and Interval Graphs

Sandy Irani* Vitus Leung†

Abstract

In this paper, we consider the online scheduling of jobs that may be competing for mutually exclusive resources. We model the conflicts between jobs with a *conflict graph*, so that the set of all concurrently running jobs must form an independent set in the graph. This model is natural and general enough to have applications in a variety of settings; however, we are motivated by the following two specific applications: traffic intersection control and session scheduling in high speed local area networks with spatial reuse. Our results focus on two special classes of graphs motivated by our applications: bipartite graphs and interval graphs. The cost function we use is maximum response time. In all of the upper bounds, we devise algorithms which maintain a set

*Supported in part by NSF grants CCR-0105498 and CCR-9625844 and by ONR Award N00014-00-1-0617. Address: University of California, Irvine, Information and Computer Science, 444 Computer Science Building, Irvine, CA 92697-3425. Email: irani@ics.uci.edu .

†Address: Sandia National Laboratories, P.O. Box 5800, Albuquerque, NM 87185-1110. Email: vjleung@cs.sandia.gov. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000. Part of this work was done while at the University of California, Irvine and supported by Caltrans Contract 65Q090.

of invariants which bound the accumulation of jobs on cliques (in the case of bipartite graphs, edges) in the graph. The lower bounds show that the invariants maintained by the algorithms are tight to within a constant factor. For a specific graph which arises in the traffic intersection control problem, we show a simple algorithm which achieves the optimal competitive ratio.

1 Introduction

In this paper, we consider scheduling jobs which are competing for limited resources. Jobs arrive in the system through time and require a certain set of resources to be completed. Any two jobs which require the same resource can not be executed simultaneously. We model the conflicts between jobs by a *conflict graph*, where each node in the graph represents a type of job. Jobs of the same type have the same requirements. If two types of jobs demand a common resource, there is an edge between those nodes in the graph. Thus, at all times, the set of jobs currently being executed must belong to nodes which form an independent set in the graph. Note that if there are two jobs of the same type in the system, one must wait until the other is completed.

We were motivated by the following two specific applications:

Traffic Intersection Control. Today's traffic intersection controllers are based on forty year old signal phasing strategies. Signal phasings are optimized offline with historical data, downloaded into the controller and triggered by the presence of vehicles [10, 17, 28]. Golumbic [18] suggests a way to model this problem with circular arc graphs. Even the state of the art in adaptive traffic signal control only extends the optimization to a few seconds before every phase change. However, one expected consequence of an effective advanced traveler information system (ATIS) [1, 2, 21] is the rerouting of congested traffic to streets and arterials that may either temporarily be under-utilized or which normally operate below capacity. Under such conditions, signal settings which have been determined based on recurrent traffic demand will not, in general, be "tuned" to accommodate the transient demand generated by the real-time driver information. As a result, system performance (as well as the effectiveness of the ATIS) is limited by the capacity of the signal system to adapt to transient traffic demand. Better strategies will be necessary for many of the proposed Intelligent Transportation Systems.

A traffic intersection is depicted in Figure 1. As all drivers should know, the traffic on approach 1 is typically not allowed to proceed with the traffic on approaches 2, 3, 4, 7, or 8. The intersection does not include right turns because right turns are often allowed whenever traffic permits and are usually ignored by traffic engineers. Also, we do not consider other intersection configurations because they are rare and again, usually ignored by traffic engineers when they consider the traffic intersection control problem in general. In fact, our intersection is the one for which the NEMA (National Equipment Manufacturers Association) standard 8-phase dual ring controller is designed.

The complete conflict graph for the traffic intersection is also depicted in Figure 1. The intersection controller must schedule the vehicles through the intersection so as to avoid any conflicts. We consider a 'job' to be a platoon (or closely spaced line) of cars which must pass through the intersection. We will assume that all platoons are roughly the same size. Note that an approach may have multiple platoons waiting to pass through the intersection at any given time.

Scheduling in high-speed local-area networks with spatial reuse. Local area networks with spatial reuse allow the concurrent access and transmission of user data with no intermediate buffering of packets. If some node s has to send data to some other node t, a 'session' is established between the

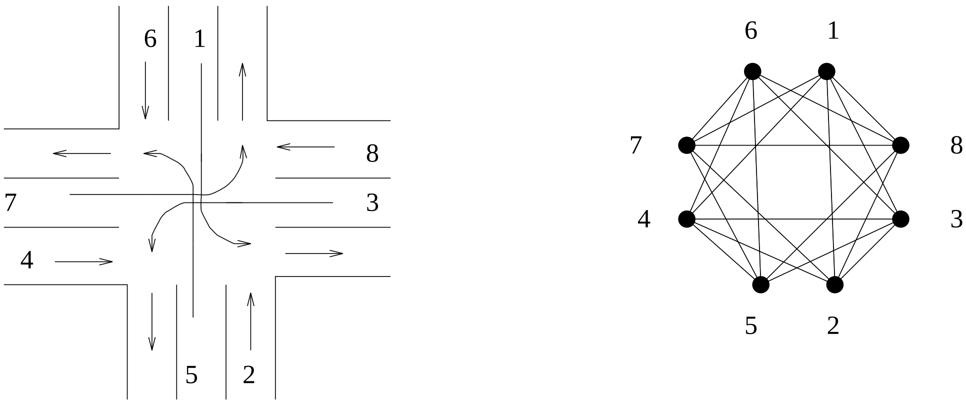

Figure 1: The graph depicting a traffic intersection.

s and t. A session typically lasts for much longer than its data transmission time and can be active only if it has exclusive use of all the links in its route from s to t. Therefore, sessions whose routes share at least one link are in conflict. Data transmissions among sessions must be scheduled so as to avoid these conflicts. We will assume that the data transmissions to be scheduled are roughly the same size and that a session may have multiple data transmissions waiting to be scheduled. This problem has been considered on ring and bus networks [12]. We examine the problem of scheduling connections on a bus where there is exactly one possible route between any two pairs of points. Thus, if connections are defined by the two nodes which must be connected, it is determined whether a given pair of connections will conflict with each other. In this situation, each possible connection between a pair of nodes in the network corresponds to a node in the conflict graph. Two nodes in the conflict graph are adjacent if the two corresponding paths in the network share an edge.

In the first application and for small networks in the second application, it is reasonable to assume that each job requires roughly the same amount of time to execute. Thus, we adopt a discrete model of time and assume that each job requires one time unit to be completed once it is started. At the beginning of a time unit, jobs may arrive on any subset of the nodes in the conflict graph; several jobs can arrive on the same node. The algorithm then chooses any independent set of nodes and schedules a job from each node in the independent set. At the end of the time unit, the scheduled jobs are gone from the graph, and all other jobs remain on their respective nodes. Then at the beginning of the next time unit, another set of jobs may arrive.

There are two natural optimization problems that arise from this model. The first is to minimize the *total response time* of all jobs in the system. The second is to minimize the *maximum response time* of any job which enters the system. We focus on bounding the maximum response time of any job which is the maximum, over all jobs j, of $c_j - a_j$, where c_j is the time when j completes and a_j is the time when j arrives. In both applications we consider, it is important to guarantee the best turnaround time to any job entering the system. In the proofs, it will be convenient to refer to the *latency* of a job, which is the time that a job waits in the system before it is started. Since each job requires only one time unit to complete once it is started, the final latency of a job is one less than its response time. Therefore, the results for maximum response time and maximum latency are asymptotically the same.

1.1 Our Results

Typical scheduling algorithms are faced with the problem of making decisions about which jobs to perform at a given time without any knowledge of the demands that will be made on the system in the future. Such algorithms are said to be *online*. We compare the performance of the online algorithm to the performance of the optimal offline algorithm. An algorithm is said to be *c-competitive* if for any sequence of jobs, its cost on the sequence is at most

c times the cost of the optimal offline algorithm on that sequence plus an additive term which is independent of the sequence. The *competitive ratio* of the algorithm is the infimum over all c such that the algorithm is c-competitive [9]. In some of our results, we measure the performance of our algorithm by a generalized version of the competitive ratio in which we bound the cost of our algorithms by a (not necessarily linear) function of the optimal offline cost.

Minimizing the maximum response time for general conflict graphs is NP-hard, even when all jobs arrive in a single time unit: the problem is equivalent to graph coloring [20]. Even approximating the minimum maximum response time to within a fixed polynomial factor is NP-hard [23]. Our results focus on two special classes of conflict graphs motivated by our applications: interval graphs and bipartite graphs. Such graphs can be optimally colored in polynomial time.

In Section 2 we consider the conflict graph from the traffic intersection control problem depicted in Figure 1. We argue that the problem of scheduling with the traffic intersection conflict graph is equivalent to scheduling on a $K_{2,2}$. In Section 2.1, we describe a simple algorithm and prove that it obtains a competitive ratio of 2 on a $K_{2,2}$. We show in Section 2.2 that this bound is optimal.

We then consider more general classes of conflict graphs: bipartite graphs, which include $K_{2,2}$, and interval graphs. Interval graphs model the problem of scheduling connections on a bus as follows: an interval graph consists of a graph where each node can be identified with a closed interval on the real line. Two nodes are adjacent if and only if their corresponding intervals intersect. Consider a bus with processors $v_1, v_2, v_3, \ldots, v_n$ connected in order along the bus. Associate each processor with a point in the real line. The points are ordered from left to right as the processors are arranged on the bus. A connection between two nodes v_i and v_j for $i < j$ can be represented by an interval from the point just to the right of v_i to the point which is just to the left of v_j. Thus, two intervals intersect if and only if the two corresponding connections share a link.

In Section 3 we prove lower bounds for these classes of graphs. For bipartite and interval graphs, we show that for any L and any algorithm, there is a graph and a sequence of job arrivals which forces the algorithm to accumulate $\Omega(nL)$ jobs on a clique but can be serviced offline with a maximum latency of L for any job. Throughout this paper n will denote the number of nodes in the conflict graph and L will denote the maximum latency of a job under the optimal offline schedule. These bounds give a lower bound of $\Omega(n)$ for the competitiveness of any deterministic algorithm on interval and bipartite graphs where the cost is the maximum response time of a job.

In sections 4.1 and 4.2, we present scheduling algorithms for bipartite and interval graphs, respectively. We prove that for any sequence of jobs, the algorithm for bipartite graphs can complete every job in time $O(n^2L^2)$. For any sequence of jobs, the algorithm for interval graphs can complete every job in time $O(n^3L^2)$. In all of the upper bounds, we devise algorithms which maintain a set of invariants which bound the accumulation of jobs on cliques (in the case of bipartite graphs, edges) in the graph.

1.2 Previous Work

Over the past twenty-seven years, most of the work done on scheduling with conflicts between certain pairs of jobs has been based on the well-known *Dining Philosophers* paradigm [5, 11, 13, 16, 24] which is inherently a problem in decentralized control. More recently, Motwani, Phillips, and Torng [25] and Bar-Noy, Mayer, Schieber, and Sudan [6] considered problems with centralized control. In [25], each vertex in the conflict graph represents a single job and two vertices are adjacent when their corresponding jobs are in conflict. When jobs arrive at integral times and have unit execution time, Motwani *et al.* showed that the competitive ratio for the makespan is 2. When jobs arrive at arbitrary times and have arbitrary execution times, they gave an upper bound of 3 on the competitive ratio for the makespan using a model which allows for preemption. In [6], each vertex in the conflict graph represents a task that is to be scheduled as often as possible and again, two vertices are adjacent when their corresponding tasks cannot be scheduled concurrently. Bar-Noy *et al.* were interested in maximizing a measure of fairness among competing types.

A number of authors have also considered offline problems in multiprocessor scheduling where conflicts among jobs are represented by a graph [4, 7, 8, 19]. Schaffter [27] considers a closely related offline problem which seeks to find a schedule for a set of jobs such that the makespan of the schedule is minimized subject to the condition that certain sets of jobs can not be computed in parallel. More recently, Coffman, Feldman, Kahale and Poonen [14] and Coffman, Robert and Stolyer [15] have examined problems that are simlar to the ones addessed in this paper. They look at scheduling non-overlapping intervals on the real line, where two intervals can not be scheduled simulataneously if they overlap. Thus, their 'conflict graph' is an interval graph. They examine a greedy algorithm under probabilistic assumptions over the input sequence.

1.3 Definitions

A node in a conflict graph G represents a class of jobs to be scheduled and two adjacent nodes in G represent two classes of jobs that cannot be scheduled together. Time is divided into discrete time units. At the beginning of a time unit, jobs may arrive on any subset of the nodes in G; several jobs can arrive on the same node. All jobs arrive with latency 0. The algorithm then chooses any independent set of nodes and schedules the oldest job waiting on each node in the independent set. At the end of the time unit, the scheduled jobs are gone from the graph and the latency of all the remaining jobs has increased by one. Then at the beginning of the next time unit, another set of jobs may arrive. Unless otherwise specified, the beginning of a time unit shall refer to the point just after the arrival of new jobs.

The *weight* of a node is defined to be the number of jobs waiting on the node, including the jobs that have just arrived in the system (i.e. the jobs with latency 0). A node is said to be *empty* if it has weight 0. The *weight* of an edge is the sum of the weights of its two endpoints.

2 The Traffic Intersection Graph

Consider the traffic intersection graph shown in Figure 1. Since approaches 1 and 2 are adjacent and have exactly the same neighborhood, they can be merged into one node. The same for approaches 3 and 4, 5 and 6, 7 and 8. The resulting conflict graph is a $K_{2,2}$. Name the nodes of the $K_{2,2}$ r_1, r_2 on the right side and l_1, l_2 on the left side.

2.1 An algorithm for $K_{2,2}$

During each time unit, the online greedy algorithm chooses the side of the graph with the oldest job. Then for each node on that side of the graph which has jobs remaining, it schedules the oldest job from that node. Note that although the following bound is expressed in terms of an upper bound for the maximum latency incurred by the optimal offline algorithm, the algorithm does not need to know this value in choosing which jobs to schedule.

Lemma 1 *Let L be an upper bound for the maximum latency for the optimal* *offline algorithm for the sequence of jobs. The online algorithm will maintain the following invariants at all times:*

1. *If the online algorithm has an edge of weight w, then the optimal schedule has at least* $\max\{w - L, 0\}$ *jobs on the edge.*

2. *If the online algorithm has a node of weight w, then the optimal schedule has at least* $\max\{w - L, 0\}$ *jobs on the node.*

Proof. Let us assume that the online algorithm has maintained the invariants through time $t-1$ and we will prove that the invariants are still maintained after time t. When new jobs arrive at the beginning of time unit t, the invariants are still maintained. We must now prove that they are still maintained at the end of time unit t after both online algorithm and offline algorithm have each scheduled jobs from an independent set of nodes in the conflict graph. Assume without loss of generality that l_1 is the node with the algorithm's oldest job at the beginning of time unit t.

If the online algorithm has any jobs on l_2 at the beginning of time t, then it schedules a job from every edge in the graph, and invariant 1 will certainly be maintained at the end of time t. Note that the optimal can only schedule one job from each edge in a single time unit.

If the online algorithm has no jobs on l_2 at the beginning of time t, then as long as invariant 2 will be satisfied at the end of time t, invariant 1 will also be satisfied. The only problem with guaranteeing invariant 2 is if the online algorithm has a node on the right side with at least $L + 1$ jobs. Suppose r_1 is a node with at least $L + 1$ jobs. (The same argument will hold for r_2 if it has at least $L + 1$ jobs.) Let's suppose the online algorithm has x jobs on l_1 and y jobs on r_1 at the beginning of time t. This means that the optimal schedule has at least $x + y - L$ jobs on (l_1, r_1) at least $y - L$ of which must be on r_1.

We must be certain the optimal schedule will continue to have at least $y - L$ jobs on r_1 at the end of time t in order to maintain invariant 2. We address two cases:

Case 1: The online algorithm has a job on l_1 with latency at least L at the beginning of time t. If the optimal schedule still has the latency-at-least-L job on l_1, it must schedule that one, which means that $y - L$ jobs still remain on r_1 at the end of time t. Now suppose that the optimal schedule does not still have the latency-at-least-L job on l_1. We first observe that no more than $x - 1$ jobs have arrived on l_1 in the last $L - 1$ time units. This is because the online algorithm has at most $x - 1$ jobs that are more recent than the latency-at-least-L job and would not have scheduled a more recent job over the latency-at-least-L job. Thus, the optimal schedule can have at most $x - 1$ jobs on l_1, which means that it has at least $y - L + 1$ jobs on r_1 at the beginning of time unit t. Thus, at least $y - L$ remain after time t.

Case 2: The online algorithm does not have a job on l_1 with latency at least L at the beginning of time t. Since l_1 has the oldest job, then the online algorithm does not have a job on r_1 with latency at least L. This means that y jobs have arrived on r_1 in the last $L - 1$ time units. Any algorithm (even the optimal) could only have scheduled at most $L - 1$ of these jobs by the beginning of time unit t. This means that the optimal schedule has at least $y - L + 1$ jobs on r_1 before time t and at least $y - L$ jobs at the end of time unit t. ■

Theorem 2 *The online greedy algorithm is 2-competitive*

Proof. We will show that for any sequence of job arrivals which can be scheduled offline with a maximum latency of L, the algorithm will never have a job with latency more than $2L$. This gives an upper bound of $(2L+1)/(L+1)$ on the competitive ratio of the algorithm where the cost is the maximum response time.

As long as the invariants are maintained, we will never have an edge of weight $2L+2$ or more. Otherwise, the optimal schedule would have an edge of weight $L+2$ which would mean it eventually incurs a latency of at least $L+1$ on some job. Since the algorithm never has more than $2L+1$ jobs on an edge, when a job j arrives on a node, say l_1, there are never more than $2L$ other jobs on any edge incident to l_1. Let x be the number of jobs already on l_1 when the job j arrives. We know that $0 \leq x \leq 2L$. There are at most $2L - x$ jobs on either node on the right side.

After the left side has been chosen x times, j will be the oldest job on l_1 and will be scheduled the next time the left side is chosen. Thus, if we can prove that the right side is not chosen more than $2L - x$ times before j is scheduled, then we know that j will wait in the system at most $2L$ time units before it is scheduled. After the right side has been chosen $2L - x$ times, if j has not yet been chosen, then the left side has the oldest job in the system. ∎

2.2 A lower bound for $K_{2,2}$

In this subsection, we show that the greedy algorithm described in the previous subsection achieves the optimal competitive ratio. All of the lower bounds in this paper are proven using an adversary technique. The adversary concocts a sequence of job arrivals based on the behavior of the online algorithm. At the beginning of each time unit, the adversary has the power to have jobs arrive on any subset of the nodes in the conflict graph based on the algorithms previous choices. After the job sequence has been determined, the adversary then can determine a schedule for the jobs in an offline manner. We then compare the cost of the online algorithm to the cost of the adversary (or offline algorithm). Sometimes, it is convenient to describe the scheduling choices made by the adversary as the sequence is described instead of after the entire sequence has been determined. Thus, when we refer to the number of jobs the adversary has on a node in the graph at some point in time, we are referring to the offline schedule determined by the adversary.

Lemma 3 *Let A be an arbitrary scheduling algorithm. If at the end of some time unit (say t), the adversary has no jobs and A has i jobs on a single edge (l_x, r_y), then the adversary can force a configuration in which A has an edge of weight $i+1$ while the adversary has an empty graph, i.e. has no jobs left. Furthermore, the adversary never has a job with latency more than $i+1$.*

Proof. The adversary has a job arrive on each of l_x and r_y until the algorithm has $i+1$ jobs on l_x or r_y. This will happen by the beginning of some time unit $t' \leq t+i+1$. Assume without loss of generality that l_x is the node on which the algorithm has $i+1$ jobs. The adversary can schedule the jobs incurring a latency of at most $i+1$ so that after time t', l_x is empty.

This allows the adversary to adopt the following strategy to empty out its own graph while forcing the algorithm to keep $i+1$ jobs on an edge. Let r_z be the node on the right side of the graph which is not r_y. For each of the next $i+1$ time units, the adversary has a job arrive on r_z. The adversary schedules the oldest job on r_y and the new job on r_z. After the $i+1$ time units, the adversary's graph is empty. Since a job was requested on r_z in each time unit, the algorithm still has $i+1$ jobs on (l_x, r_z). ■

Theorem 4 *For $K_{2,2}$, no algorithm can have a competitive ratio for the maximum response time better than 2.*

Proof. We will show that for any positive integer L, there is an adversary who can force the algorithm to have a job of latency at least $2L$ while the adversary only has a latency of at most L. This gives a lower bound of $(2L+1)/(L+1)$ for the competitive ratio of any deterministic algorithm. Since L can be chosen to be arbitrarily large, the theorem follows.

There will be L stages. As j goes from 0 to $L-1$, the algorithm will start each stage with j jobs on an edge. We then invoke Lemma 3 to obtain an edge with $j+1$ jobs. At the end of the whole process, the algorithm has an edge (l_x, r_y) with L jobs while the adversary's graph is empty. For the next L time units, the adversary has a job arrive on l_x and r_y. By always scheduling the oldest job, the adversary never incurs a latency of more than L. The algorithm will have $2L+1$ jobs on edge (l_x, r_y) and must incur a latency of at least $2L$. ■

3 Lower Bounds for Bipartite and Interval Graphs

In order to prove the lower bound for bipartite and interval graphs, we prove several lemmas for a conflict graph consisting of a simple path and one additional node u adjacent to some of the nodes in the path. The number of nodes in the path will be $n = 4(k+1)$ for any $k \geq 1$. Number every other edge in the path from $-k$ to k, starting with the second edge. The rest of the edges in the path are unnumbered. The nodes in the path are numbered consecutively from $-2k-2$ to $2k+1$. The path is pictured in Figure 2. For bipartite graphs, u is adjacent to all the odd nodes in the path. For interval graphs, u is adjacent to all the nodes in the path.

Fix an arbitrary positive integer L. Let A be an arbitrary scheduling

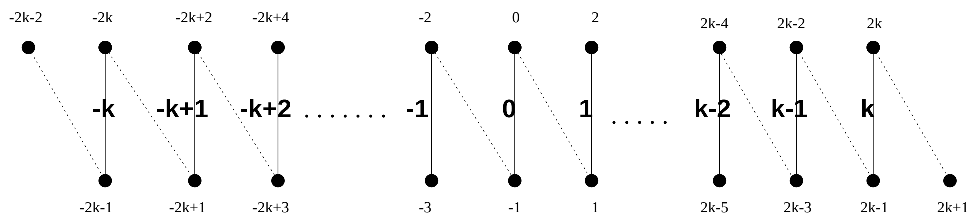

Figure 2: The path

algorithm. Suppose at the end of some time unit (say t), for some $m \leq k$ and some $i < L$, the adversary has no jobs and A has at least $Lm + i$ jobs on each odd node in the path from node $-2m-1$ to node $2m-1$. We will eventually prove that the adversary can force A to accumulate at least $Lm + i + 1$ jobs on each odd node while the adversary has an empty graph.

The adversary has jobs arrive in time units $t+1$ through $t + Lm + i + 1$ or until the algorithm has $Lm + i + 1$ jobs on an unnumbered edge. The jobs offered in time unit $t' \in \{t+1, t+2, \ldots, t+Lm+i+1\}$ are described below. Let $j = \lceil (t'-t)/L \rceil - 1$.

- Have a job arrive on each of the odd numbered nodes incident to edges $-m, -m+1, \ldots, -(j+1)$.
- Have a job arrive on each of the two endpoints in edges $-j, -j+2, \ldots, j-2, j$.
- Have a job arrive on each of the even numbered nodes incident to edges $j+1, j+2, \ldots, m$.

Note that jobs are never placed on node $-2m-2$ or node $2m+1$. Figure 3 shows how the jobs arrive for $j = 2$ and $j = 3$ on a path with $k = m = 5$.

Lemma 5 *For some $t' \in \{t+1, t+2, \ldots, t+Lm+i+1\}$, at the end of time unit t', the algorithm has at least $Lm+i+1$ jobs on one of the unnumbered edges in the path from node $-2m-2$ to $2m+1$.*

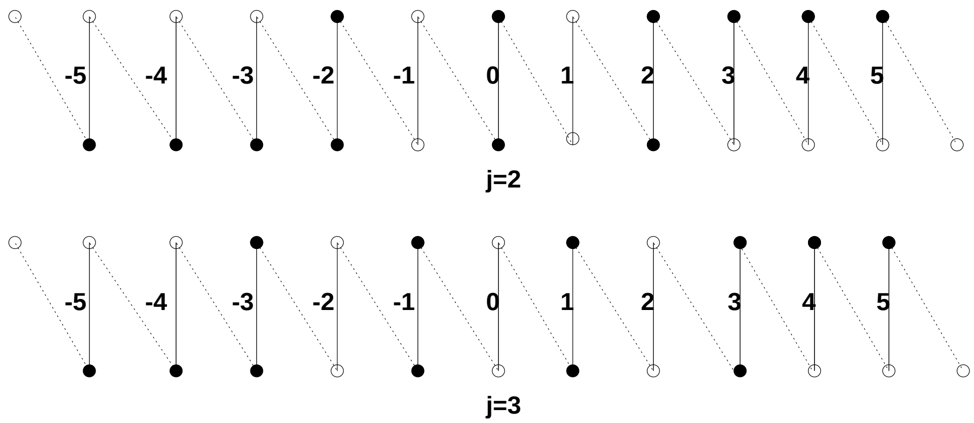

Figure 3: The job arrival pattern for a path in which $k = 5$. Jobs arrive on the shaded nodes.

Proof. Notice that in each time step, a new job is placed on all the unnumbered edges in the path from node $-2m-2$ to $2m+1$. Thus, in order for the algorithm not to accumulate an extra job on these edges, it must remove a job from each such edge every time unit. Since node $-2m-2$ never has any jobs on it, the only way for the algorithm to schedule a job from every unnumbered edge is to schedule a job from all the odd nodes in the range $-2m-1$ to $2m-1$. By time $t + Lm + i + 1$, the node $2m$ (and hence an unnumbered edge) has weight at least $Lm + i + 1$. ■

Lemma 6 *Let e be any unnumbered edge in the path. For any t' in $\{t+1, t+2, \ldots, t+Lm+i+1\}$, the adversary can schedule the jobs incurring a latency of at most L so that after time t', e is empty.*

Proof. For each $t+1 \le t' \le t + Lm + i + 1$, let

$$j = \left\lceil \frac{t'-t}{L} \right\rceil - 1.$$

During time t', the adversary will schedule the following.

1. The job that just arrived on each of the odd numbered nodes in edges $-m, -m+1, \ldots, -(j+1)$.

2. The oldest job on each of the even or each of the odd numbered nodes in edges $-j, -j+1, \ldots, j$.

   Every time j changes, the adversary switches the parity of the set of nodes from which it schedules a job in part 2. The initial parity is chosen so that a job is always requested from edge e. This ensures that e remains empty. Suppose that e is the edge in between edges a and $a+1$. If a is even, the adversary will pick the odd nodes in part 2 when j is odd and the even nodes when j is even. If a is odd, the adversary will pick the even nodes in part 2 when j is odd and the odd nodes when j is even.

3. The job that just arrived on each of the even numbered nodes in edges $j+1, j+2, \ldots, m$.

Parts 1 and 3 schedule the jobs that arrive on each of the odd numbered nodes incident to edges $-m, -m+1, \ldots, -(j+1)$ and even numbered nodes incident to edges $j+1, j+2, \ldots, m$ as soon as they arrive. Therefore, we only have to worry about the jobs that arrive on each of the two endpoints in edges $-j, -j+2, \ldots, j-2, j$.

During a time unit, a job arrives on each of the two endpoints in every other numbered edge between edges $-j$ and j, and those edges shift every time j changes. Therefore, part 2 schedules the jobs that arrive on half of the unnumbered edges as soon as they arrive and jobs that arrive on the other half of the unnumbered edges L time units after they arrive. ■

Lemma 7 *If at the end of some time unit (say t), for some $m \leq k$ and some $i < L$, the adversary has no jobs and A has $Lm + i$ jobs on each odd node in* *the path from node* $-2m-1$ *to node* $2m-1$*, then the adversary can force a configuration in which* A *has* $Lm+i+1$ *jobs on each odd node in the path from node* $-2m-1$ *to node* $2m-1$ *(node* $-2m-3$ *to node* $2m+1$ *if* $i = L-1$*), while the adversary has an empty graph. Furthermore, the adversary never has a job with latency more than* L*.*

Proof. The adversary's strategy proceeds in two steps. The adversary will first force a situation where A has an edge of weight $Lm+i+1$ while the adversary has an empty graph. Furthermore, the adversary never has a job with latency more than L. Then the adversary will force the algorithm to accumulate $Lm+i+1$ jobs on each odd node while he has an empty graph and never has a job which incurs a latency greater than L.

The adversary strategy is the following. Keep offering jobs as described above until the algorithm has an unnumbered edge e with $Lm+i+1$ jobs at some time t'. The adversary will serve the sequence up to that point so that its edge e is empty at time t'. Furthermore, if an odd node has x jobs, then all even nodes have at most $L-x$ jobs.

This allows the adversary to adopt the following strategy to empty out its own graph while forcing the algorithm to keep $Lm+i+1$ jobs on edge e. If the odd nodes in the adversary's graph have the oldest job, then the adversary has a job arrive on the odd node incident to e. The adversary schedules the oldest job on each odd node. Alternatively, if the even nodes in the adversary's graph have the oldest job, then the adversary has a job arrive on the even node incident to e. The adversary schedules the oldest job on each even node. After L time units, the adversary's graph is empty. Since a job was requested on e in each time unit, the algorithm still has $Lm+i+1$ jobs on e. Thus, we have proven that the adversary can force a situation where A has an edge of weight $Lm+i+1$ while the adversary has an empty graph.

Now the adversary will force there to be $Lm+i+1$ jobs on every odd node. The adversary will use the extra node u described at the beginning of this section. Recall that u is not part of the path. Rather, it is an additional node which is adjacent to at least all the odd nodes in the path. The adversary does the following:

- For nL consecutive time steps, have a job arrive on v, the odd node incident to edge e. The adversary can schedule each job as it arrives. The algorithm must eventually have all $Lm+i+1$ jobs on v if it avoids having a job of latency nL.

- For nL consecutive time steps have a job arrive on u. Eventually the algorithm must schedule all jobs on v and have $Lm+i+1$ jobs on u if it avoids having a job of latency nL. The adversary schedules each job as it arrives.

- If $i \leq L-2$, let the set K be the nodes $-2m-1, -2m+1, \ldots, 2m-1$. Otherwise (we are setting up for a new stage), let K be the nodes $-2(m+1)-1, -2(m+1)+1, \ldots, 2(m+1)-1$. For the next nL time units, have a job arrive on each node in K. The adversary schedules all jobs as they arrive. The algorithm must eventually have $Lm+i+1$ jobs on each node in K.

■

Theorem 8 *For bipartite graphs and interval graphs, no algorithm can have a competitive ratio for the maximum response time better than $n/4$.*

Proof. Let the number of nodes n be $4(k+1)+1$ for some $k \geq 1$. Let L be an arbitrary positive integer. The adversary will concoct a job arrival sequence whose latency is at most L for any job under the optimal offline schedule while the algorithm will have a job whose latency is at least $nL/4 - 1$. Since L can be chosen to be arbitrarily large, the theorem holds.

The graph will consist of the bipartite or interval graph described at the beginning of this section. There will be $k+1$ stages and L steps within a stage. As m goes from 0 to k and i goes from 0 to $L-1$, we start each step with $Lm+i$ jobs on each of the odd nodes from $-2m-1$ to $2m-1$. We then invoke Lemma 7 to obtain $Lm+i+1$ jobs on each odd node in the path from node $-2m-1$ to node $2m-1$ (node $-2m-3$ to node $2m+1$ if $i = L-1$). At the end of the whole process, the algorithm has an edge (v,w) with $L(k+1)$ jobs while the adversary's graph is empty. For the next L time units, the adversary has a job arrive on w and v. By always scheduling the oldest job, the adversary never incurs a latency of more than L. The algorithm will have $L(k+2)$ jobs on edge (v,w) and must incur a latency of at least $L(k+2)-1$. By plugging in

$$k = \frac{n-1}{4} - 1,$$

the algorithm's latency is at least

$$\frac{Ln}{4} - 1.$$

■

4 Upper Bounds

For both the upper bounds presented in this section, we assume that we know an upper bound L on the maximum latency of the optimal algorithm for the sequence seen so far. We start with a guess that $L = 1$. We prove that if the maximum latency of the optimal algorithm is at most L, then our algorithm will pick an independent set in every time period and will schedule every job within a certain bound. If the maximum latency of the optimal schedule exceeds L, then it may be necessary to increase L. In this case, we use a standard trick and double L [3]. In this way, we know that the optimal maximum latency always lies in the interval from $L/2$ to L. The details of this technique and how it applies to each algorithm are discussed at the end of each subsection for bipartite and interval graphs separately.

4.1 Bipartite Graphs

We will call any job with latency at least L an *old* job. Let $X = (\frac{n}{2}+4)L+2$. At any time instant, let H be the subgraph consisting of all edges whose weight is more than X and any node incident to such an edge. Suppose we have a simple path in the graph G and we number the edges according to their sequence in the path, starting with 1. We say that the path is an *H-path* if all the odd numbered edges are in H. Call a node *immediately forbidden* if it is adjacent to a node of weight at least $X+1$, is non-empty and has no old jobs.

As soon as a node becomes immediately forbidden, it is *marked* and will remain marked until it is *unmarked.* The set of *forbidden* nodes consists of all marked nodes which have at most one old job. The concepts immediately forbidden and forbidden are transient and depend on the state of the node at a given moment in time. This differs from the process of marking a node since a node remains marked until it is unmarked. Note that an immediately forbidden node is also forbidden, but a forbidden node may no longer be immediately forbidden.

A time unit in which there are no immediately forbidden nodes is said to be *unconstrained.* All the nodes become unmarked in an unconstrained time unit. During an unconstrained time unit, there are no forbidden nodes. This follows from the fact that all nodes are unmarked.

The algorithm schedules the oldest job from each node in an independent set which is chosen in a series of three rounds as follows:

Round 1: Pick all nodes that are adjacent only to empty nodes.

Round 2: Pick any node v such that there is an odd length H-path from a forbidden node to v. The length of a path is the number of edges in the path.

Round 3:

Case 1: The current time unit is unconstrained. Suppose it is the t^{th} unconstrained time unit. If t is odd, pick the left side of the graph. If t is even pick the right side of the graph. Pick all nodes on the chosen side of the graph which have not already been picked and are not adjacent to a node which has already been picked.

Case 2: The current time unit is not unconstrained. Pick a side of the graph arbitrarily. Pick all nodes from H on the chosen side of the graph which have not already been picked and are not adjacent to a node which has already been picked.

The algorithm will maintain the following invariants:

1. For $a \geq 1$, if the algorithm has weight $X+a$ on an edge, then the optimal schedule has weight at least a on that edge.

2. For $a \geq 1$, if the algorithm has weight $X+a$ on a node, then the optimal schedule has weight at least a on that node.

3. There is no odd length H-path between two forbidden nodes.

Note that the invariants imply that there is never a node or edge with more than $X + L + 1$ jobs (i.e. $a \leq L + 1$), otherwise the optimal schedule would have more than $L + 1$ jobs on a node or edge and would incur a latency of more than L.

Lemma 9 *If the invariants hold, the algorithm will pick an independent set.*

Proof. The only way for two adjacent nodes, w_1 and w_2, to be picked is if there are two forbidden nodes, v_1 and v_2, such that there is an odd length H-path from v_1 to w_1 and from v_2 to w_2. This means that there is an odd length H-path from v_1 to v_2 which violates the last invariant. ∎

Lemma 10 *If the maximum latency of the optimal algorithm is at most L, then the invariants are maintained at all times.*

Proof. We first prove that invariant 3 holds as a result of invariant 1. Consider a forbidden node v; v has at most one job with latency at least L. Any node can have at most $2L$ jobs with latency less than L because if more than $2L$ jobs arrive at the same node during L consecutive time units, then the optimal latency is more than L. Thus, v has weight at most $2L + 1$.

Now consider a simple path with an odd number of edges and an even number of nodes, $p_0, p_1, \ldots, p_{2k+1}$. Suppose that invariant 3 is false, i.e. that p_0 and p_{2k+1} are both forbidden and that for each $0 \leq i \leq k$, the edge (p_{2i}, p_{2i+1}) is in H. Consider three consecutive nodes in the path $p_{2i-2}, p_{2i-1}, p_{2i}$. If the weight of p_{2i-2} is x, then the weight of p_{2i-1} is at least $X + 1 - x$, since the edge (p_{2i-2}, p_{2i-1}) is in H. But then the weight of node p_{2i} is at most $x + L$ since the edge (p_{2i-1}, p_{2i}) has weight at most $X + L + 1$. Thus, the weight of consecutive even nodes in the path increases by at most L along the path. Since p_{2k+1} has weight at most $2L + 1$, p_{2k} has weight at least $X - 2L$. Furthermore, since p_0 has weight at most $2L + 1$, there must be at least

$$\left\lceil \frac{(X - 2L) - (2L + 1)}{L} \right\rceil > \frac{n}{2}$$

even nodes in the path. Since there are the same number of even nodes and odd nodes in the path, there are not enough nodes in the graph.

Next we argue that if invariant 1 holds for all instances, then invariant 2 must hold as well. Suppose there is a node v for which invariant 2 does not hold. If we add a node w adjacent only to v which never receives a job, the behavior of the algorithm (and optimal schedule) would be identical. So if invariant 2 does not hold for v, then invariant 1 would not hold for the edge (v, w).

We now prove that invariant 1 holds for each edge (v, w) in H. Suppose that invariant 1 holds just after time unit $t-1$. At the beginning of time unit t, new jobs arrive and the invariant is still maintained. During time unit t the online algorithm and the optimal schedule both execute a set of jobs from an independent set in the graph. We must prove that invariant 1 is still maintained after this occurs. We address two cases:

Case 1: The algorithm has jobs waiting on both w and v at the beginning of time t. Let u be a neighbor of v. If u is picked in Round 2, then w is also picked since that path u, v, w would be a continuation of an odd length H-path from some forbidden node. The same holds for a neighbor of w. Thus, if neither v or w is picked in Round 2, then neither is adjacent to a node which has been picked in Round 2. Since the edge (w, v) is in H, one of them will be picked in Round 3. Thus, if invariant 1 holds at the beginning of a time t, it holds at the end of time step t.

Case 2: The algorithm has no jobs on w. (The same argument holds if there are no jobs on v). Thus, v must have weight at least $X+1$. If v has a non-empty neighbor all of whose jobs are not old, then v is adjacent to a forbidden node and would have been picked in Round 2. Furthermore, if it is adjacent to only empty nodes, it will be chosen in Round 1.

It remains to address the case where v has weight at least $X+1$ and is adjacent to some node u with an old job. Let's say that v has weight $X+a$ and u has weight b. This means that the optimal schedule has at least $a+b$ jobs on (v,u). Since u has an old job and weight b, at least a jobs must be on v.

Let b' be the number of jobs that have arrived on node u in the last $L-1$ time units. We know that the algorithm has all b' of these jobs still on u since the old job is still remaining. Thus, the b jobs which the algorithm has on u include the old job and the b' jobs which arrived since, and we can conclude that $b' \leq b-1$.

If the optimal schedule has the old job on u, it must pick it in the next time step in which case a of its jobs will remain on v. If the optimal schedule does not have the old job, then it has at most $b' \leq b-1$ jobs on u. In this case, the optimal schedule has at least $a+1$ jobs on v and at least a will remain after the current step. ∎

Note that if the optimal algorithm never has a job with latency more than L, then it can never be the case that more than $2L$ jobs arrive on a single edge during a period of L consecutive time units. This fact, combined with Lemma 10, implies that as long as the optimal latency is at most L, the premise of the lemma below will be true.

Lemma 11 *Suppose that the invariants are maintained and there have never been more than $2L$ jobs arriving on a single edge during a period of L consecutive time units. Then a node can be immediately forbidden for at most L consecutive time units after which it will remain marked and can not be immediately forbidden again until it is unmarked or more than nL consecutive time units have passed.*

Proof. Consider a node v which becomes immediately forbidden at time t. We must establish that if v remains immediately forbidden for the next L time units, then it will not be chosen for the next L time units. This, in turn, implies that one of its jobs must reach a latency of L by time $t + L$ at which point, v is no longer immediately forbidden. Consider a time t' in the interval from t to $t + L$. v must have a neighbor with weight at least $X + 1$ at time t' by the definition of being immediately forbidden. Furthermore, v is in H (also by definition of being immediately forbidden) which means that any neighbor of v with jobs will be chosen in Round 2 at time t'. This implies that v can not be chosen at time t'.

Now we must establish that v can not become immediately forbidden until it is unmarked or at least nL consecutive time units have passed. Suppose that v does become immediately forbidden before it becomes unmarked. In order for v to become immediately forbidden again, it must have all of its old jobs scheduled and must also have a neighbor with weight at least $X + 1$. Consider the last time which v has any old jobs before it becomes immediately forbidden again. At this moment, it has exactly one old job. Since v is still marked, it must be forbidden. The only way for v to be chosen is if all of its neighbors are empty. Otherwise, if v is part of the subgraph H, a neighbor of v with jobs would have been chosen in Round 2, and hence v would not be chosen in Rounds 2 or 3. If v is not part of the subgraph H, it can not be chosen in Rounds 2 or 3. (Recall that since v is marked, the time unit is not unconstrained which means we would enter Case 2 of Round 3 in which only nodes in H can be chosen). Thus, at the moment that v has its last old job scheduled, all of its neighbors are empty.

At time t, node v becomes immediately forbidden which means that it had a neighbor u which had weight at least $X + 1$. It stops being immediately forbidden by time $t + L$ at which time u still has at least $X - L + 1$ jobs. In order for v to become immediately forbidden again without an unconstrained time period, u must clear all it's jobs which will take at least $X - L$ time units. Then one of its neighbors, say w, must accumulate $X+1$ jobs. Suppose that node w accumulates $X + 1$ jobs in r time units. This means that $r \geq X + 1 - L$. Otherwise, the optimal maximum latency would be more than L. Thus, the total amount of time which must pass after v is no longer immediately forbidden and before it becomes immediately forbidden again is at least $2(X - L) > nL$. ■

Theorem 12 *The maximum response time of a job is* $O(n^2L^2)$.

Proof. When a job j arrives on a node v, there are at most $(X+L+1)$ other jobs on v. We will argue that every $Ln+1$ time units, there is an unconstrained time unit. This implies that every $2(Ln + 1)$ time units, at least one job on v gets eliminated. Thus after $2(X + L + 1)(Ln + 1) = O(n^2L^2)$ time units, j will be scheduled.

Let t_1 be an unconstrained time period and let t_2 be the next unconstrained time period. We must establish that $t_2 - t_1 \leq Ln$. Suppose this is not the case. Since a single node can remain immediately forbidden for only L time units, some node u must become immediately forbidden twice in the interval from t_1 through $t_1 + nL$. Since there are no unconstrained time periods in this interval, u can not become unmarked during the interval. However, Lemma 11 says that u must either become unmarked or the interval must be greater than nL time units which is a contradiction. ■

Now we must address what to do if the optimal latency exceeds L. The upper bound on the maximum latency of the algorithm follows from Lemma 11 which requires that the invariants are maintained and that at most L jobs can arrive on an edge in any period of L consecutive time units. If either of these conditions fails, then we know that the maximum latency of the optimal algorithm has exceeded L. In this case, we double the value of L, unmark all nodes and continue. We will call a phase the period of time between the points at which L doubles. Let l_i be the value for L in the i^{th} phase. We are guaranteed that the latency of any job within the i^{th} phase is $O(n^2(l_i)^2)$. Suppose that there are m phases and L is the true upper bound on the optimal algorithm's maximum latency. Since $\sum_{i=1}^{m} n^2(l_i)^2 = O(n^2L^2)$, the bound follows.

4.2 Interval Graphs

We will call a node in the graph and the closed interval on the real line which defines the node by the same name. For an interval v, let $l(v)$ be its left endpoint and $r(v)$ be its right endpoint. Without loss of generality, we will assume that all endpoints of all intervals are distinct. This can be achieved by shifting coincident endpoints by a very small amount so that the conflict graph remains the same. This is possible because it is the interval overlaps and not the actual intervals themselves that create the conflict graph.

Consider a point p on the real line. We call the *weight* of point p to be the sum of the weights of the nodes whose intervals contain p. We will initially assume that the algorithm knows that the maximum latency in the optimal schedule is L. Let $X = (2L + 1)n$. The algorithm will maintain the following invariant at all times:

- For any $a \geq 1$, if the algorithm has weight $X + a$ on p, then the optimal schedule has weight at least a on p.

Note that since the optimal schedule never has a job with latency more than L, it never has a point with weight more than $L + 1$. If the invariant is maintained, this means that the algorithm never has a point with weight more than $X + L + 1$.

The set of all points of weight $X + j$ forms a finite set of disjoint intervals (some closed, some open, some half-open). We will call this set of intervals $\mathcal{I}_j$. Note that for any $I \in \mathcal{I}_j$, a non-empty node either has an empty intersection with I or completely contains I. The set of all points of weight at least $X + 1$ form a finite set of disjoint closed intervals which we will call $\mathcal{I}$. Now we will form a set of intervals $\mathcal{M}$ which include for each $\mathcal{I}_j$, each interval in $\mathcal{I}_j$ which is immediately to the right or immediately to the left of a node with a job of latency at least L. That is, consider an interval $I \in \mathcal{I}_j$. If the left side of I is open and its left 'endpoint' is the right endpoint of a node with latency at least L, then $I \in \mathcal{M}$. Similarly for the right endpoint of I.

Finally consider the set of points which are in some interval in $\mathcal{I}$ but not some interval in $\mathcal{M}$. When these points are grouped in maximal contiguous intervals, they form a finite set of disjoint intervals. We will call this set of intervals $\mathcal{J}$ and will name them from left to right: $J_1, J_2, \ldots, J_m$. Note that some of the intervals in $\mathcal{J}$ may be open or half-open.

During each time step, the algorithm must pick an independent set of nodes and schedule the oldest job from each of those nodes. The algorithm will pick an independent set which covers $\mathcal{J}$. We will then prove that this is sufficient to maintain the invariant. To do this, we will define a set of *forbidden* nodes. Once this set has been determined, we can greedily pick an independent set from the set of non-forbidden nodes. In order to determine the set of forbidden nodes, we require some definitions. A node is *immediately forbidden* if it is non-empty and only has jobs of latency less than L. We *mark* every node when it becomes immediately forbidden. At the beginning of a given time unit, if there are no immediately forbidden nodes, then we unmark all nodes. As in the case of bipartite graphs, the state of being immediately forbidden is a transitory state: it just depends on the condition of the node at a given instant of time. However, when a node is marked, it remains marked until it is explicitly unmarked.

The set of forbidden nodes can be determined in a given time period as follows: if there are no immediately forbidden nodes, then the set of forbidden nodes in this time unit is then empty. If there are immediately forbidden nodes, then we initialize the set of forbidden nodes to be all the marked nodes which have at most one job with latency at least L. We then make a pass through the graph from right to left considering each interval J_i in reverse order. Let J be the current interval under consideration. If there are any non-empty unforbidden nodes which contain $l(J)$, then let w be the one with the right-most left endpoint. Add to the forbidden set all non-empty nodes whose right endpoint is in the interval $[l(w), l(J)]$. (Refer to Figure 4).

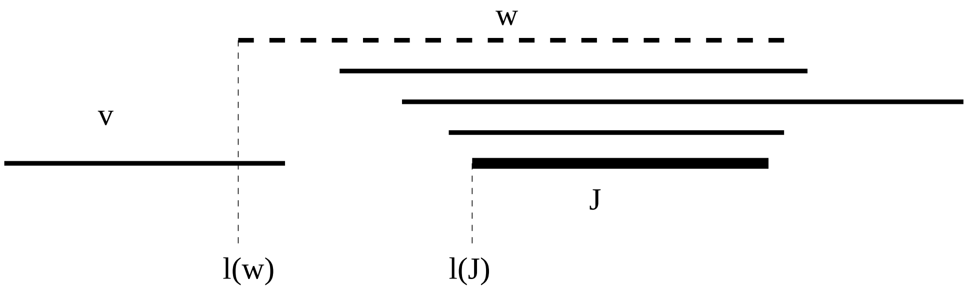

Figure 4: The forbidden nodes are shown in solid. The unforbidden node is shown in a dashed line. The interval J must be covered by some node; v becomes forbidden because its right endpoint lies in $[l(w), l(J)]$

Once the set of forbidden nodes has been determined, we can pick the independent set as follows. If there are no forbidden nodes in the graph, start with the node v with the oldest job. Scan right from v and then left from v, greedily picking the independent set (i.e., whenever the end of the current node is reached, pick the next non-empty node reached). If there are forbidden nodes then scan from left to right. Pick the first non-empty unforbidden node that is reached. When the right endpoint of that node is reached, pick the non-empty unforbidden node with the next left endpoint. Continue until the end of the intervals have been reached.

During each time step, the complete algorithm is as follows:

- Determine the sets of intervals $\mathcal{I}_j$ for $1 \leq j \leq L+1$.
- Determine the set of intervals $\mathcal{M}$.
- Determine the set of intervals $\mathcal{J}$.
- Determine the set of forbidden nodes.
- Pick an independent set of nodes and execute the oldest job from each of those nodes.

Lemma 13 *Whenever the invariant is maintained, for any point p on the real line, the total weight of forbidden nodes which intersect p is at most $X = (2L+1)n$.*

Proof. Let $f(p)$ denote the total weight of all forbidden nodes which contain the point p. We will scan the real line from right to left keeping track of the maximum $f(p)$ seen so far. We will show that $f(p)$ can jump by at most $2L+1$ at any point. Since there are n nodes and $f(p)$ can only increase at the right endpoint of a node, the maximum total weight of forbidden nodes on any point is at most $X = (2L+1)n$.

Now suppose that the maximum $f(p)$ we have seen so far increases when the right endpoint of a forbidden node v has been reached. Let the previous maximum be x. If v was added to the forbidden set when it was initialized, then v has at most one job with latency at least L. Any node can have at most $2L$ jobs with latency less than L because if more than $2L$ jobs arrive during

L consecutive time units, then the optimal latency is more than L. Thus, the weight of v is at most $2L+1$.

Now suppose that v was added because for some interval $J \in \mathcal{J}$, there is an unforbidden node w which covers $l(J)$. Furthermore, w is the node with the left-most left endpoint of all unforbidden nodes which cover $l(J)$ and $r(v)$ falls in the interval $(l(w), l(J)]$. Consider the point just to the right of $l(J)$. We know that at this point, the weight is at least $X+1$ (since it is in $\mathcal{I}$) and the weight of the forbidden nodes is at most x. Thus, the weight of the non-forbidden nodes at $l(J)$ is at least $X+1-x$. Since none of these unforbidden nodes have left endpoints in $(r(v), l(J)]$, the weight of the unforbidden nodes at $r(v)$ must be at least $X+1-x$. Since the weight of $r(v)$ is at most $X+L+1$, the weight of forbidden nodes at $r(v)$ is at most $x+L$. ∎

Lemma 14 *Suppose that the invariant is true just before the online algorithm and the optimal schedule choose the set of jobs they will schedule in a given time unit. Let p be any point contained in an interval in $\mathcal{M}$. If the online algorithm has weight at least $X+a$ on p at the beginning of the time unit, then the optimal schedule will have weight at least a on p at the end of the time unit.*

Proof. Consider some interval $I \in \mathcal{M}$. I is an interval in $\mathcal{I}_j$ for some $j \geq 1$ which was placed in $\mathcal{M}$ because I is immediately next to a node w which has a job of latency at least L. Without loss of generality, assume that I is to the right of w. Suppose that the algorithm has b jobs on w and has a weight of c on I. The algorithm has weight $b+c$ on $r(w)$. Since the invariant is maintained at the beginning of the time unit, the optimal schedule has at least weight $b+c-X$ on $r(w)$ and at least weight $c-X$ on I at the beginning of the time unit.

If the optimal schedule has a job with latency at least L on w, it must schedule that job and will continue to have weight $c-X$ on I at the end of the time unit. If the optimal schedule does not have any jobs of latency at least L on w, then it must have weight at most $b-1$ on w. This is due to the fact that since the algorithm has not scheduled any jobs of latency less than L, if the optimal schedule has any such jobs, then the algorithm also has them. Thus, in this case, the optimal schedule has weight at most $b-1$ on w which means it must have weight at least $c-X+1$ on I at the beginning of time unit t. At least $c-X$ of these jobs will remain at the end of the time unit. ∎

Lemma 15 *Suppose the invariant is true just before the online algorithm chooses its independent set, then the independent set chosen by the online algorithm covers* $\mathcal{J}$.

Proof. We first consider the case where there are no forbidden nodes. This means that every non-empty node has a job of latency at least L. Consider some interval $I \in \mathcal{I}_j$ for some j. For I not to be in $\mathcal{M}$ (i.e. for it to be contained in an interval in $\mathcal{J}$), it can not be to the immediate left or immediate right of any node with jobs. This implies that $l(I)$ is the left endpoint of a non-empty node, say v_l, and $r(I)$ is the right endpoint of a non-empty node which we will call v_r. Since $I \in \mathcal{J}$, there are no endpoints of non-empty nodes in the interior of I. Now suppose we are scanning from left to right in building the independent set and I has not been completely covered by a node in the independent set. Then $l(I)$ is not covered and v_l will be chosen and I will be covered. Similarly, if we are scanning left and I has not been covered, then v_r will be chosen.

Now suppose that there are forbidden nodes. The independent set is determined by scanning from left to right. Let J_i be the leftmost (i.e. first) interval in $\mathcal{J}$ encountered which is not completely covered by the independent set. Let J_i' be the leftmost subinterval of J_i which is not covered.

Suppose that J_i' is to the immediate right of some node v in the independent set. This node v is in the independent set and therefore must have some jobs. If v has a job of latency at least L, then the point just to the right of v is in $\mathcal{M}$ and not $\mathcal{J}$. Thus, v must only have jobs of latency less than L. This means that v is immediately forbidden and would not have been chosen to be in the independent set.

Therefore the interval to the immediate left of J_i' must not be covered by the independent set. This implies that $l(J_i')$ is $l(J_i)$. We know that there must be some unforbidden node which contains $l(J_i)$ and is non-empty. This follows from Lemma 13 and the fact that the weight of $l(J_i)$ is at least $X+1$. Let w be the node with the rightmost left endpoint which is non-empty and covers $l(J_i)$. Since w was not chosen, it must intersect with a node v in the independent set whose right endpoint is in $[l(w), l(J_i)]$. However, this would imply that v is forbidden and would not have been chosen in the independent set. $\blacksquare$

Lemma 16 *The invariant is maintained by the online algorithm at all times.*

Proof. Suppose that the invariant is maintained at the end of some time unit $t-1$. When new jobs arrive at the beginning of time unit t, the invariant is still maintained. At this point, the online algorithm chooses its independent set. By Lemma 15, this independent set covers $\mathcal{J}$. Lemma 14, implies that the invariant is still maintained for every point in $\mathcal{M}$, even after both online and offline algorithms choose their independent set. The fact that $\mathcal{I} = \mathcal{M} \cup \mathcal{J}$ implies that the invariant is maintained for all points at the end of time unit t. $\blacksquare$

Lemma 17 *A node can be immediately forbidden for at most L time units after which it will not be immediately forbidden again until there is a time unit in which all nodes are not forbidden.*

Proof. When a node becomes immediately forbidden it is non-empty. Furthermore, none of its jobs will ever be scheduled as long as it is immediately forbidden. This means that within L time units, one of its jobs will have latency at least L and it will no longer be immediately forbidden.

As soon as a node becomes immediately forbidden, it is marked. It will remain marked until there is a time unit in which there are no immediately forbidden nodes at which point all nodes are unmarked and hence there are no forbidden nodes as well. It remains to prove that once a node is marked, after it becomes not immediately forbidden, it can not become immediately forbidden again until it is unmarked.

We prove this remaining claim by contradiction. Suppose that a node v which is immediately forbidden becomes not immediately forbidden and then becomes immediately forbidden again before it is unmarked. In order for it to become immediately forbidden again, all its jobs must have latency less than L. Consider the point in time just before this happens. At this point, v has exactly one job which has latency at least L. Since v is marked, this implies it would have been forbidden. Since no jobs are scheduled from forbidden nodes, the job with latency more than L would never have been scheduled and v would not have become immediately forbidden. ∎

Theorem 18 *The maximum response time of a job is* $O(n^3L^2)$.

Proof. Consider the set of jobs in the system when a job j arrives. There are at most $n(X+L+1)$ jobs. By Lemma 17, at least every $nL+1$ time units, there are no forbidden nodes. This means that at least every $nL+1$ time units, the oldest job in the system will get scheduled. Thus, we are guaranteed that within $n(X+L+1)(nL+1)$ time units, j will be scheduled. ∎

If the invariant is ever violated (which implies that the optimal latency has exceeded L), then the value of L is doubled and all nodes are unmarked. Note that a node may be immediately forbidden with the new value of L in which case it is marked again. The proof of Theorem 18 uses Lemma 17 and the fact that there are never more than $X + L + 1$ jobs on a node which follows from the invariant. Lemma 17 still holds except that a node may become immediately forbidden at one of these doubling times without having a point when there are no unforbidden nodes. We will call a phase the period of time between the points at which L doubles. Let l_i be the value for L in the i^{th} phase. We are guaranteed that the latency of any job within the i^{th} phase is $O(n^3(l_i)^2)$. Suppose that there are m phases and L is the true upper bound on the optimal algorithm's maximum latency. Since $\sum_{i=1}^{m} n^3(l_i)^2 = O(n^3L^2)$, the bound follows.

5 Open Problems

There is still quite a gap in the bounds for maximum response time. In particular, is it possible to obtain a competitive ratio for either bipartite or interval graphs which is strictly a function of n?

Modeling the problem of scheduling connections on other local area network architectures would require the use of more general classes of intersection graphs (e.g. tree graphs, the intersection graphs of paths in a tree [26]). Is it possible to obtain bounds on more general classes of intersection graphs? In particular, is there a bound for circular arc graphs which would model ring networks which are addressed in [12] and traffic intersection control as modeled in [18]? Alternatively, is there a graph for which there is no competitive algorithm?

Finally, the assumption that the input sequence of job arrivals can be arbitrarily bad may be somewhat pessimistic. An interesting direction for future research would be to restrict the power of the adversary by allowing only certain input distributions as proposed by Koutsoupias and Papadimitriou [22].

References

[1] H.M. Al-Deek, S.S. Ishak and A.E. Radwan, Impact of traffic diversion with ATIS on travelers' safety. *Computers & Industrial Engineering*, **34** (1998), 547–558.

[2] H.M. Al-Deek and A. Kanafani, Modeling the benefits of advanced traveler information systems in corridors with incidents. *Transportation Res. Part C*, **1** (1993), 303–324.

[3] J. Aspnes, Y. Azar, A. Fiat, S. Plotkin and O. Waarts, On-line routing of virtual circuits with applications to load balancing and machine scheduling. *JACM*, vol. 44, (no. 3), (May 1997), 486–504.

[4] B.S. Baker and E.G. Coffman, Jr., Mutual exclusion scheduling, *Theoret. Comput. Sci.*, **162** (1996), 225–243.

[5] J. Bar-Ilan and D. Peleg, Distributed resource allocation algorithms, in *Proceedings for the 6th International Workshop on Distributed Algorithms* (A. Segall and S. Zaks, eds.). Springer-Verlag. New York, 1992, pp. 277–291.

[6] A. Bar-Noy, A. Mayer, B. Schieber and M. Sudan, Guaranteeing fair service to persistent dependent tasks. in *SIAM Journal on Computing*, vol.27, (no.4), SIAM, Aug. 1998. p.1168-89.

[7] H.L. Bodlaender and K. Jansen, On the complexity of scheduling incompatible jobs with unit-times, in *Mathematical Foundations of Computer*

Science (A.M. Borzyszkowski and S. Sokolowski, eds.). Springer-Verlag. Berlin, 1993, pp. 291–300.

[8] H.L. Bodlaender, K. Jansen and G.J. Woeginger, Scheduling with incompatible jobs, *Discrete Appl. Math.*, **55** (1994), 219–232.

[9] A. Borodin and R. El-Yaniv, *Online Computation and Competitive Analysis*, Cambridge University Press, New York, 1998.

[10] D. Bullock and C. Hendrickson, Roadway traffic control software. *IEEE Trans. Control Systems Technology*, **2** (1994), 255–264.

[11] K. Chandy and J. Misra, The drinking philosophers problem, *ACM Trans. Programming Lang. and Systems*, **6** (1984), 632–646.

[12] J. Chen, I. Cidon and Y. Ofek, A local fairness algorithm for gigabit LANs/MANs with spatial reuse, *IEEE J. Selected Areas in Comm.*, **11** (1993), 1183–1192.

[13] M. Choy and A.K. Singh, Efficient fault-tolerant algorithms for distributed resource allocation. In *ACM Transactions on Programming Languages and Systems*, vol.17, (no.3), May 1995. p.535-59.

[14] E.G. Coffman, Jr., A. Feldmann, N. Kahale and B. Poonen, Computing Call Admission Capacities in Linear Networks, *Prob. Eng. Inf. Sci.*, **13** (1999), 387–406.

[15] E.G. Coffman, Jr., P. Robert and A.L. Stolyar, The interval packing process of linear networks, *Performance Evaluation Review*, vol.27, (no.3), (December 1999), 3–4.

[16] E.W. Dijkstra, Hierarchical ordering of sequential processes, *Acta Inform.*, **1** (1971), 115–138.

[17] R.E. Fenton, IVHS/AHS: driving into the future, *IEEE Control Systems Mag.*, vol.14, (no.6), (December 1994), 13–20.

[18] M.C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, New York, 1980.

[19] K. Jansen, Scheduling of incompatible jobs on unrelated machines, *Internat. J. of Found. of Comput. Sci.*, **4** (1993), 275–291.

[20] R.M. Karp, Reducibility among combinatorial problems, in *Complexity of Computer Computations* (R.E. Miller and J.W. Thatcher, eds.). Plenum. New York, 1972, pp. 85–103.

[21] A.J. Khattak, J.L. Schofer and F.S. Koppelman, Commuters' enroute diversion and return decisions: analysis and implications for advanced traveler information systems. *Transportation Res. Part A*, **27** (1993), 101–111.

[22] E. Koutsoupias and C.H. Papadimitriou, Beyond competitive analysis, in *Proceedings for the 35th Symposium on the Foundations of Computer Science*. IEEE Computer Society. Washington, DC, 1994, pp. 394–400.

[23] C. Lund, M. Yannakakis, On the hardness of approximating minimization problem, in *Journal of the Association for Computing Machinery*, vol.41, (no.5), Sept. 1994. p.960-81.

[24] N. Lynch, Upper bounds for static resource allocation in a distributed system, *J. Comput. System Sci.*, **23** (1981), 254–278.

[25] R. Motwani, S. Phillips and E. Torng, Non-clairvoyant scheduling, in *Theoretical Computer Science*, vol.130, (no.1), 1 Aug. 1994. p.17-47.

[26] P.L. Renz, Intersection representations of graphs by arcs, *Pacific J. Math.*, **34** (1970), 501–510.

[27] M.W. Schaffter, Scheduling with forbidden sets, *Discrete Applied Mathematics*, **72** (1997), 155–166.

[28] F.V. Webster, *Traffic Signal Settings*, Her Majesty's Stationery Office, London, 1958.