# Maximizing the number of rides served for Dial-a-Ride

## Barbara M. Anthony 🄳
Southwestern University, Georgetown TX 78626, USA
anthonyb@southwestern.edu

## Sara Boyd
Southwestern University, Georgetown TX 78626, USA

## Ricky Birnbaum
Connecticut College, New London CT 06320, USA

## Ananya Christman 🄳
Middlebury College, Middlebury VT 05753, USA
achristman@middlebury.edu

## Christine Chung 🄳
Connecticut College, New London CT 06320, USA
cchung@conncoll.edu

## Patrick Davis
Connecticut College, New London CT 06320, USA

## Jigar Dhimar
Connecticut College, New London CT 06320, USA

## David Yuen
Kapolei HI 96707, USA
yuen888@hawaii.edu

—— **Abstract** ——

We study a variation of offline Dial-a-Ride, where each request has not only a source and destination, but also a revenue that is earned for serving the request. We investigate this problem for the uniform metric space with uniform revenues. While we present a study on a simplified setting of the problem that has limited practical applications, this work provides the theoretical foundation for analyzing the more general forms of the problem. Since revenues are uniform the problem is equivalent to maximizing the number of served requests. We show that the problem is NP-hard and present a 2/3 approximation algorithm. We also show that a natural generalization of this algorithm has an approximation ratio at most 7/9.

## 1 Introduction

Due to their practical applicability, Dial-a-Ride Problems (DARP) have been studied from the perspective of operations research, management science, combinatorial optimization, and theoretical computer science. There are numerous variants of the problem, but fundamentally all DARP variants require the scheduling of one or more vehicle routes and associated times to satisfy a collection of pickup and delivery requests, or rides, from specified origins to specified

destinations. Each ride can be viewed as a request between two points in an underlying metric space, with the ride originating at a *source* and terminating at a *destination*. These requests may be restricted so that they must be served within a specified time window, they may have weights associated with them, details about them may be known in advance or only when they become available, and there may be various metrics to optimize. For most variations the goal is to find a schedule that will allow the vehicle(s) to serve requests within the constraints, while meeting a specified objective. Much of the motivation for DARP arises from the numerous practical applications of the transport of both people and goods, including delivery services, ambulances, ride-sharing services, and paratransit services.

We study offline DARP on the uniform metric (i.e. where the distance between any pair of locations is the same for all pairs) with a single server where each request has a source, destination, and revenue. The server has a specified deadline after which no more requests may be served, and the goal is to find a schedule of requests to serve that maximizes the total revenue. We furthermore assume uniform revenues, and refer to this problem as *URDARP*. *URDARP* is thus equivalent to maximizing the number of rides served by the deadline. Although this form of the problem has fewer practical relevance than more general forms, it can be applied to urban settings where it is reasonable to assume that a driver would like to serve as many requests as possible and these requests take roughly the same amount of time to serve. Our motivation for analyzing this basic form of the problem is that doing so would allow us to extend the analysis to more general versions, which have more practical applications, unlike this simplest form. However, we found in the course of our work that analyzing this basic form was challenging in itself. We found that even this fundamental variant is in fact NP-hard, and its analysis elusive. We expect it will provide the theoretical foundation for analyzing the more general forms of the problem.

In particular, we have found that any *DARP* algorithm for the nonuniform revenue variant that greedily chooses one request at a time to serve can be at best a 1/2-approximation. Lemma 1 in Section 2 details what happens when, for example, the greedy strategy is based on largest revenue. We have found a similar outcome for the variant of DARP on a non-uniform metric with uniform revenues. Lemma 2 in Section 2 details what happens when the greedy strategy is based on shortest request.

We therefore consider algorithms that give preference to sequences of requests that are "chained" together, i.e. such that each request in a sequence is served immediately after the previous request of that sequence. More formally, a chain of requests is defined as a sequence of requests such that (1) for all requests except the first, the source is the destination of the previous request and (2) for all requests except the last, the destination is the source of the next request. Specifically, we consider an algorithm we call TWOCHAIN that gives preference to requests that are in chains of length at least two. We focus on this algorithm and *URDARP* because, with an understanding of this fundamental setting, we can then work to break the barrier of 1/2 in the setting with general revenues.

In sum, the focus of this work is on offline URDARP (i.e. DARP with a single vehicle, on the uniform metric, with uniform revenues). We begin by showing even this basic problem is NP-hard by a reduction from the Hamiltonian Path problem. We then show that our TWOCHAIN algorithm yields a 2/3-approximation, and exhibit an instance where TWOCHAIN serves exactly 2/3 the optimal number of requests. Since TWOCHAIN yields a tight 2/3-approximation, with a matching lowerbound instance that is quite simple and clean, we expected that the natural generalization of the algorithm, an algorithm we call $k$-CHAIN, would yield a $k/(k+1)$-approximation. Surprisingly, it does not. We exhibit an instance of *URDARP* where $k$-CHAIN earns at most 7/9 times the revenue of the optimal solution. We

conclude with a discussion of how the (non polynomial-time) algorithm that greedily always chooses the longest chain gives at most a 5/6-approximation.

## 1.1 Related Work

DARP has been extensively studied and there are numerous variations on the problem, including the number of vehicles, the objectives, the presence or absence of time windows, and how the request sequence is issued (i.e. offline or online). The 2007 survey *The dial-a-ride problem: models and algorithms* [8] provides an overview of some of the models and algorithms, including heuristics, that have been studied. A decade later, *Typology and literature review for dial-a-ride problems* [9] focuses on classifying the existing literature based upon applicability to particular real-world problems, again including both algorithms with theoretical guarantees and heuristics. To our knowledge, despite its relevance to modern-day transportation systems, the version of the problem we investigate in this paper has not been previously studied, neither for the uniform nor general metric space.

However, there are a few variants that have similarities to our version. Our DARP variant is closely related to the Prize Collecting Traveling Salesperson Problem (PCTSP) where the server earns a revenue (or prize) for every location it visits and a penalty for every location it misses, and the goal is to collect a specified amount of revenue while minimizing travel costs and penalties. PCTSP was introduced by Balas [3] and studied by many others including Awerbuch et al. [2], who gave the first approximation algorithms with polylogarithmic performance. Bienstock et al. developed a 2-approximation for a version of PCTSP where there is a cost for each edge and a penalty for each vertex, and the goal is to find a tour on a subset of the vertices that minimizes the sum of the cost of the edges in the tour and the vertices not in the tour [4]. Blum et al. gave the first constant-factor approximation algorithm for the Orienteering Problem where the input is a weighted graph with rewards on nodes and the goal is to find a path that, starting at a specified origin, maximizes the total reward collected, subject to a limit on the path length [5]. The online variant of the PCTSP, where the cities arrive over time, has also been studied by Ausiello et al. [1] who presented a 7/3-competitive algorithm.

The online revenue-maximization variant of DARP, where requests have non-uniform revenue and a release time at which they become known to the server and the goal is to serve requests so as to maximize total revenue by a specified deadline, was also studied for the uniform metric in [7] and a non-uniform metric in [6], where Christman et al. presented competitive algorithms with ratios 1/2, and 1/6, respectively.

## 2 Preliminaries

The input to URDARP is a uniform metric space, a set of requests, and a time limit $T$. Each request has a source point and a destination point in the metric space, and a revenue, where the revenues are uniform. A unit capacity *server* starts at a designated location in the metric space, the *origin*. The goal is to move the server through the metric space, serving requests one at a time so as to maximize the revenue earned in $T$ time units, which, with uniform revenues, is equivalent to maximizing the number of requests served. For an URDARP instance $I$, $\mathrm{OPT}(I)$ denotes an optimal schedule on $I$.

We refer to a move from one location to another as a *drive*. If a request is being served then we refer to it as a *service drive* (sometimes referred to in the literature as a *carrying move*). If the drive is not serving a request and solely for the purpose of moving the server from one location to another we refer to it as an *empty drive* (sometimes referred to in the

literature as an *empty move*). We refer to a sequence of one or more requests that are served without any intermediary empty drives as a *chain* and a sequence of two requests that are served without an empty drive in between as a *2-chain*.

We now provide two lemmas regarding a more generalized version of URDARP, where the revenues are nonuniform; we refer to this variant as RDARP. By an analysis similar to that of the online Greatest Revenue First (GRF) algorithm, studied by Christman and Forcier [7], it can be shown that the simple greedy algorithm that repeatedly finds and serves the highest-revenue request of those remaining is a 1/2-approximation for RDARP as well. We now give the matching bound, showing that this greedy algorithm can yield at best a 1/2-approximation.

▶ **Lemma 1.** *The approximation ratio of the greedy algorithm that repeatedly chooses a maximum-revenue request to serve for RDARP is no greater than 1/2.*

**Proof.** Consider an instance with $x$ requests chained together each with revenue $r$, and $x$ individual requests, none of which are connected to other requests, each with revenue $r + \epsilon$ for some small $\epsilon > 0$. No requests start at the origin $o$. Let $T = x + 1$. OPT will serve all of the $x$ requests that are chained together, earning $xr$ revenue. An algorithm that greedily chooses one request at a time to serve will serve only the requests with revenue $r + \epsilon$, and can serve only $\lfloor x/2 \rfloor$ of them in time $T$, earning $\lfloor x/2 \rfloor (r + \epsilon)$. ◀

We assume for the remainder of this work that revenues are uniform. We now show that if we instead consider a non-uniform metric, the approximation ratio of a similar greedy algorithm is at most 1/2.

▶ **Lemma 2.** *The approximation ratio of the algorithm that greedily chooses the shortest request to serve for DARP with uniform revenues on a non-uniform metric is no greater than 1/2.*

**Proof.** Let $a$, $b$, and $c$ denote three points on a non-uniform metric space such that the distance between $a$ and $b$ and $b$ and $c$ is $T/k$ for some positive even integer $k$ such that $T$ mod $k = 0$, and the distance between $a$ and $c$ is $T/k - \epsilon$, for some small $\epsilon > 0$. Let $a$ be the origin. Consider an instance on this space with $k/2$ requests from $a$ to $b$, $k/2$ requests from $b$ to $a$, and $k/2$ requests from $a$ to $c$. OPT will alternately serve the $k/2$ requests from $a$ to $b$ and the $k/2$ requests from $b$ to $a$, i.e. as a chain of $k$ requests. An algorithm that greedily chooses the shortest request at a time to serve will serve only the requests from $a$ to $c$ while spending $T/k - \epsilon$ time on an empty drive from $c$ to $a$ between each serve, thereby serving $k/2$ requests in total. ◀

## 2.1 Hardness

While it was already shown in [6] that the problem of offline RDARP on a general metric is NP-hard, we now show that even URDARP, where the metric is uniform and the requests have uniform revenue, is NP-hard by reduction from the Hamiltonian Path problem. The reduction proceeds as follows.

Given a directed Hamiltonian Path input $G = (V, E)$ where $n = |V|$, build a uniform metric space $G'$ with $2n + 2$ points (see Figure 1): one point will be the server origin $o$, one will be a designated "sink" point $t$, and the other $2n$ points are as follows. For each node $v \in V$, create a point $v'$ and a point $v''$ in $G'$. Create a URDARP request in $G'$ from point $v'$ to point $v''$ for each $v \in V$, which we will refer to as a *node request*. Further, for each edge $e = (u, v)$ in $E$ of $G$, create a URDARP request from point $u''$ to point $v'$ in $G'$, which we
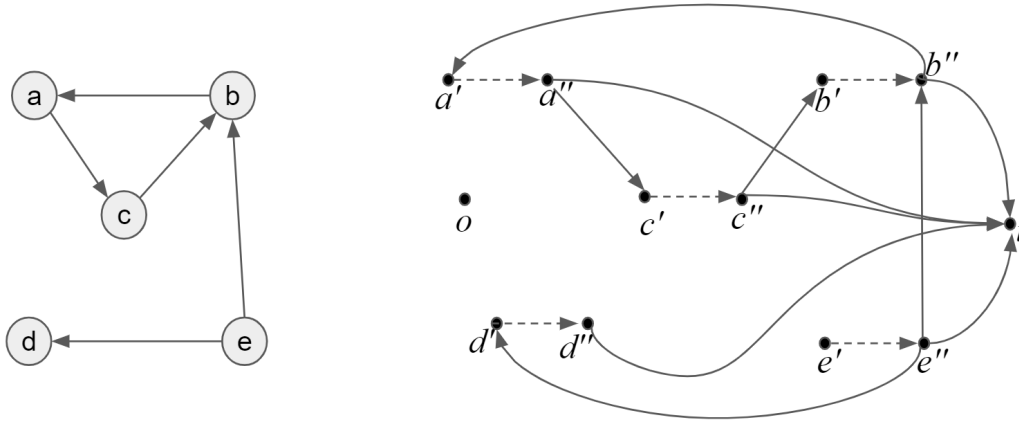
**Figure 1** An example instance $G$ of the Hamiltonian Path problem where $n = 5$ (left), and its corresponding instance for URDARP where $T = 2n + 1$ (right). Any Hamiltonian path on a graph of $n$ vertices has length $n - 1$, which would correspond to a sequence in the corresponding URDARP instance of $2n - 1$ requests (since a URDARP request is created for each vertex and each edge of $G$). But note that here there is no Hamiltonian path in $G$, yet the URDARP instance still has a sequence of requests of length $2n - 1$ which starts from $e'$. The extra edges we add from each point $v''$ to $t$ in the URDARP instance prevent such false positives by ensuring that any Hamiltonian path in $G$ will in fact correspond to a URDARP sequence of length $2n$.

will refer to as an *edge request*. Additionally, for each $v \in V$, create an *edge request* from $v''$ to the designated sink point $t$ in $G'$. Set $T = 2n + 1$. Finally, make the server origin a separate point that is one unit away from all other points.

▶ **Lemma 3.** *There is a Hamiltonian Path in $G$ if and only if $2n$ requests can be served within time $T = 2n + 1$ in the URDARP instance.*

**Proof.** Let $p = (v_1, v_2, \ldots, v_n)$ be a Hamiltonian Path in $G$. Construct the sequence of $2n$ URDARP requests in $G'$ by the node request from $v_1'$ to $v_1''$, the edge request from $v_1''$ to $v_2'$, the node request from $v_2'$ to $v_2''$, the edge request from $v_2''$ to $v_3'$, and so forth, through the edge request from $v_{n-1}''$ to $v_n'$, the node request from $v_n'$ to $v_n''$, and finally the edge request from $v_n''$ to the designated sink $t$. This sequence can be executed in time $T = 2n + 1$ since it requires one unit of time for the server to drive from the origin to $v_1'$ and $2n$ units for the remaining drives.

Conversely, consider a URDARP sequence in $G'$ of length $2n$. Note that by construction of $G'$, any sequence of URDARP requests must alternate between node requests and edge requests, where any edge to the sink is counted as an edge request (and must be a terminal request). Since destinations in $G'$ can be partitioned into the sink, single-primed points, and double-primed points, we can thus analyze the three possibilities for the destination of the final URDARP request.

If either the sink or a single-primed point is the destination for the final URDARP request, the URDARP sequence must end with an edge request. The alternating structure ensures the URDARP sequence begins with a node request, and thus contains exactly $n$ node requests and $n$ edge requests. If a double-primed point is the destination for the final URDARP request, the URDARP sequence must end with a node request. The alternating structure ensures the URDARP sequence begins with an edge request, and contains exactly $n$ edge

requests and exactly $n$ node requests. Thus, the URDARP sequence always contains $n$ node requests. This ensures that the length $n$ path in the original graph $G$ includes all $n$ vertices in the original graph $G$, and thus the existence of a Hamiltonian Path.   ◀

Due to the above reduction procedure along with Lemma 3, we have the following theorem.

▶ **Theorem 4.** *The problem URDARP is NP-hard.*

For the remainder of the work we focus strictly on URDARP (uniform metric, uniform revenues).

## 3   Algorithms

We begin by presenting our TWOCHAIN algorithm that is a 2/3-approximation for URDARP (please see Algorithm 1 for details). The idea of this polynomial-time algorithm is that it simply looks for chains of requests of length at least 2 whenever a drive is required. At each time unit if there is a request that starts at the current location of the server, the server will always serve that request (continuing the chain) rather than driving away to a different request. We note that this subtlety makes the algorithm differ from the algorithm that simply chooses *any* 2-chain to serve; the approximation ratio of this latter algorithm is an open problem. In addition, the server is never "idle" in that if there are remaining requests to serve that can be served before the deadline, the server will drive to serve one of them.

While the analysis of TWOCHAIN we provide requires many detailed cases, we were surprised to discover that simpler more elegant approaches all failed in subtle ways, indicating to us the problem is more nuanced than what one expects at first blush. We believe that the interplay between requests and the possibility for numerous criss-crosses of chains of requests prevents simpler analyses. We note that our proof actually yields a guarantee of not only 2/3 of the optimal number of requests, but instead $1/3(|OPT| + T - 1)$, where $T$ is the time limit.

### 3.1   The TWOCHAIN Algorithm

Let $S$, $T$, and $o$ denote the set of requests, time limit, and origin, respectively. Let $OPT(S, T, o)$ and $ALG(S, T, o)$ denote the schedules returned by $OPT$ and TWOCHAIN, respectively, on the instance $(S, T, o)$ and let $|OPT(S, T, o)|$ and $|ALG(S, T, o)|$ denote the number of requests served by $OPT$ and TWOCHAIN, respectively.

We begin by showing that in the special case where the deadline is more than twice the number of requests, TWOCHAIN is optimal.

▶ **Lemma 5.** *If $T \geq 2|S|$ then $|OPT(S, T, o)| = |ALG(S, T, o)| = |S|$.*

**Proof.** By induction on $|S|$. If $|S| = 1$, then clearly TWOCHAIN can serve the request if $T \geq 2$. If $|S| \geq 2$, then within the first two time units TWOCHAIN serves at least one request. So there are at most $|S| - 1$ remaining requests to serve within $T - 2$ time. Since $T \geq 2|S|$, then by the inductive hypothesis, $T - 2 \geq 2(|S| - 1)$, so TWOCHAIN can serve the remaining requests within the remaining time.   ◀

In the next lemma, we tackle the general case where the deadline $T$ is tighter. We prove a lower bound on what TWOCHAIN earns, that will suffice for later showing it yields a 2/3-approximation.

▶ **Lemma 6.** *Let $m = |OPT(S, T, o)|$. If $T < 2|S|$, then $|ALG(S, T, o)| \geq \frac{1}{3}(m + T - 1)$.*

---

**Algorithm 1** The TWOCHAIN algorithm

---

1: Input: Set $S$ of requests, time limit $T$, origin $o$
2: Set $t := T$
3: Let $S'$ denote the subset of requests $(a, b) \in S$ where $b$ is the source of another request in $S$.
4: **while** $t > 0$ **do**
5:     **if** there exists a request starting from $o$ in $S$ **then**
6:         Choose one such request $(o, b)$, with preference given to requests from $S'$.
7:         Serve $(o, b)$.
8:         $t := t - 1$
9:         Remove $(o, b)$ from $S$ (and $S'$ if $(o, b)$ was in $S'$).
10:         $o := b$, and update $S'$ based on the new $S$.
11:     **else if** $t \geq 2$ and requests remain in $S$ **then**
12:         Choose one such request $(a, b)$, with preference given to requests from $S'$.
13:         Drive from $o$ to $a$, then serve the request $(a, b)$.
14:         $t := t - 2$
15:         Remove $(a, b)$ from $S$ (and $S'$ if $(o, b)$ was in $S'$).
16:         $o := b$, and update $S'$ based on the new $S$.
17:     **else**                      ▷ no requests remain (that we have enough time to serve)
18:         $t := t - 1$

---

**Proof.** Since $T < 2|S|$, $S \neq \emptyset$. Let $k$ denote the number of requests in the first chain served by TWOCHAIN and denote this chain as $(u_0, u_1), (u_1, u_2), \ldots, (u_{k-1}, u_k)$. Let $c$ denote the number of drives TWOCHAIN makes to get to the first request, that is, either $c = 0$ if there is a request starting at $o$ and $c = 1$ if not. After TWOCHAIN serves the first chain, we are left with a smaller instance of the problem $(S_{new}, T_{new}, o_{new})$ where $S_{new} = S - \{(u_0, u_1), (u_1, u_2), \ldots, (u_{k-1}, u_k)\}$, $T_{new} = T - c - k$, and $o_{new} = u_k$.

We proceed by strong induction on $T$. If $T = 0, 1$, or $2$, then the lemma is trivially true. If $T \geq 3$, then since $|S| > T/2$, TWOCHAIN serves at least one chain. We assume inductively that $|ALG(S_{new}, T_{new}, o_{new})| \geq \frac{1}{3}(|OPT(S_{new}, T_{new}, o_{new})| + T_{new} - 1)$ and will show $|ALG(S, T, o)| \geq \frac{1}{3}(|OPT(S, T, o)| + T - 1)$.

Case 1: $k = 1$.
Case 1.1: $c = 1$. Then there is no ride starting at $o$ and the first chain has length 1, so we know that there must be no 2-chains in $S$. Then all solutions require an empty drive after each service drive, so $|ALG(S, T, o)| = m = \lfloor T/2 \rfloor \geq \frac{T}{2} - \frac{1}{2}$ and hence, $m \geq \frac{1}{3}(m + T - 1)$.
Case 1.2: $c = 0$. Then there is a ride starting at $o$ but there is no 2-chain that starts at $o$. Let $OPT(S, T, o)$ return the path $(o, v_1), (v_1, v_2), \ldots, (v_{T-1}, v_T)$. Therefore $(o, v_1)$ and $(v_1, v_2)$ cannot both be rides. Then the path $(v_2, v_3), \ldots, (v_{T-1}, v_T)$ has at least $m - 1$ rides from $S$ and therefore at least $m - 2$ rides from $S_{new} = S - \{(o, u_1)\}$. So the path $(o_{new}, v_2), (v_2, v_3), \ldots, (v_{T-1}, v_T)$ also has at least $m - 2$ rides from $S_{new}$. Thus $|OPT(S_{new}, T - 1, u_1 = o_{new})| \geq m - 2$. By induction, $|ALG(S, T, o)| \geq 1 + \frac{1}{3}(|OPT(S_{new}, T - 1, u_1)| + (T - 1) - 1) \geq 1 + \frac{1}{3}(m - 2 + T - 1 - 1) = \frac{1}{3}(m + T - 1)$.
Case 2: $k \geq 2$. There are two subcases.
Case 2.1: $T_{new} \geq 2|S_{new}|$. In this case, by Lemma 5 we have $|ALG(S_{new}, T_{new}, o_{new})| = |S_{new}| = |S| - k$. So we have:

$$|ALG(S, T, o)| = k + |ALG(S_{new}, T_{new}, o_{new})| = k + |S| - k = |S|$$

Hence, $|OPT(S,T,o)| = |S|$ as well, so recalling that $T < 2|S|$, we have, as desired:

$$|ALG(S,T,o)| = |S| = \tfrac{1}{3}(|S| + 2|S|) > \tfrac{1}{3}(|OPT(S,T,o)| + T - 1).$$

**Case 2.2:** $T_{new} < 2|S_{new}|$. Let the path $P^*$ of length $T+1-c$ be the path that traverses $OPT(S,T,o)$ starting from $o_{new}$. More formally, if $c = 0$, $P^*$ is $(o_{new}, o), (o, v_1), (v_1, v_2), \ldots (v_{T-1}, v_T)$. If $c = 1$, then since $(o, v_1)$ is not in $S$, $P^*$ is $(o_{new}, v_1), (v_1, v_2), \ldots (v_{T-1}, v_T)$. Let $r$ denote the number of requests in $(u_0, u_1), (u_1, u_2), \ldots, (u_{k-1}, u_k)$ that are also in $OPT(S,T,o)$ and note that $r \le k$. So $P^*$ has $m$ requests from $S$ and $m - r$ requests from $S_{new}$. Note that $T_{new} = T - c - k = (T + 1 - c) - (k + 1) = |P^*| - (k + 1)$. We modify $P^*$ to create a path $P$ by deleting the last $k + 1$ drives from $P^*$. Then $|P| = T_{new}$ and $P$ has at most $k + 1$ fewer requests from $S_{new}$ than $P^*$ so $P$ has at least $m - r - (k + 1)$ requests from $S_{new}$. Hence, we have:

$$|OPT(S_{new}, T_{new}, o_{new})| \ge m - r - k - 1 \tag{1}$$

There are two subcases.

**Case 2.2.1:** If $-r + k - 1 - c \ge 0$, then we have:

$$\begin{aligned}
|ALG(S,T,o)| &= k + |ALG(S_{new}, T_{new}, o_{new})| \\
&\ge k + \tfrac{1}{3}(|OPT(S_{new}, T_{new}, o_{new})| + T_{new} - 1) \text{ by ind. hyp.} \\
&\ge k + \tfrac{1}{3}(m - r - k - 1 + T_{new} - 1) \text{ by eqn.(1)} \\
&\ge \tfrac{1}{3}(m + T - 1 - r + k - 1 - c) \\
&\ge \tfrac{1}{3}(m + T - 1)
\end{aligned}$$

which is the desired equation.

**Case 2.2.2:** If $-r + k - 1 - c < 0$, then $k - r \le c$ and there are two subcases.

**Case 2.2.2.1:** $k = r$. Please see the Appendix where we show that in all subcases of Case 2.2.2.1, $P$ starts at $o_{new}$, has at least $m - r - k + 1$ requests from $S_{new}$, and has length $T_{new}$. Thus:

$$|OPT(S_{new}, T_{new}, o_{new})| \ge m - r - k + 1 \tag{2}$$

Then, since $c = 0$ or $c = 1$, we have:

$$\begin{aligned}
|ALG(S,T,o)| &\ge k + \tfrac{1}{3}(|OPT(S_{new}, T_{new}, o_{new})| + T_{new} - 1) \text{ by ind. hyp.} \\
&\ge k + \tfrac{1}{3}(m - r - k + 1 + T_{new} - 1) \text{ by eqn.(2)} \\
&\ge k + \tfrac{1}{3}(m - 2k + 1 + T - k - c - 1) \text{ since } k = r \\
&\ge \tfrac{1}{3}(m + T - c) \ge \tfrac{1}{3}(m + T - 1) \text{ since } c = 0 \text{ or } 1
\end{aligned}$$

So we are done with Case 2.2.2.1 and must now prove Case 2.2.2.2 to complete the proof.

**Case 2.2.2.2:** $k \ne r$. Recall that since $k - r \le c$, it must be that $k = r + 1$. Please see the Appendix where we show that in all subcases of Case 2.2.2.2, $P$ starts at $o_{new}$, has at least $m - r - k$ requests from $S_{new}$, and has length $T_{new}$. Thus:

$$|OPT(S_{new}, T_{new}, o_{new})| \ge m - r - k \tag{3}$$

So we have:

$$|ALG(S,T,o)| \geq k + \tfrac{1}{3}(|OPT(S_{new}, T_{new}, o_{new})| + T_{new} - 1) \text{ by ind. hyp.}$$
$$\geq k + \tfrac{1}{3}(m - r - k + T - k - c - 1) \text{ by eqn.(3)}$$
$$= \tfrac{1}{3}(m + T - 1 - r + k - c)$$
$$= \tfrac{1}{3}(m + T - 1 - r + (r + 1) - c)$$
$$\geq \tfrac{1}{3}(m + T - 1)$$

This completes the proof. ◄

▶ **Theorem 7.** TWOCHAIN *gives a 2/3 approximation for URDARP.*

**Proof.** We again proceed by considering two cases.

Case 1: $T \geq 2|S|$: Then by Lemma 5, $|ALG(S,T,o)| = |OPT(S,T,o)|$, and we are done.

Case 2: $T < 2|S|$: Then by Lemma 6, $|ALG(S,T,o)| \geq \tfrac{1}{3}(|OPT(S,T,o)| + T - 1)$.
As in Lemma 6, let $m = |OPT(S,T,o)|$. There are two subcases.

Case 2.1: If $m < T$, then $|ALG(S,T,o)| \geq \tfrac{1}{3}(m + T - 1) > \tfrac{1}{3}(m + m - 1)$. Since $|ALG(S,T,o)|$ is an integer, this implies $|ALG(S,T,o)| \geq 2m/3$.

Case 2.2: If $m = T$, then an $OPT(S,T,o)$ solution must be $(o = v_1, v_1), (v_1, v_2), \ldots, (v_{m-1}, v_m)$ where every drive must be a service drive, serving a request from $S$.
We use the same definitions of $k, r$, and $c$ as in Lemma 6 and note that $c = 0$ in this case. Denote the first chain served by TWOCHAIN as $(o = u_0, u_1), (u_1, u_2), \ldots, (u_{k-1}, u_k)$. Note that TWOCHAIN would start with a service drive right from $o$ because in this case there is a 2-chain starting at $o$. If $k = T = m$ then $|ALG(S,T,o)| = |OPT(S,T,s)|$ so we are done. If $m = 1$ or $m = 2$ then, $k = m$, so we are done. If $m = 3$ then $k = 2$ or $3$, and in both cases we have $k > 2m/3$, so we are also done.
So we consider the case where $m \geq 4$ (so $k \geq 2$) and $k < m$. After TWOCHAIN serves the first chain, the server is at $u_k$ and there is $T - k$ time remaining, so in the smaller instance of the problem, $T_{new} = T - k$, and $o_{new} = u_k$.
Since $|OPT(S,T,o)| = m$, then $|OPT(S_{new}, T + 1, u_k)| \geq m - r$, since in time $T + 1$, OPT can drive from $u_k$ to the origin, and then follow the path of $OPT(S,T,o)$ to serve $m - r$ requests (recall that $r$ is the number of requests in $OPT(S,T,o)$ that are also in the first chain of $ALG(S,T,o)$). So recalling that $T_{new} = T - k$, we have,

$$|OPT(S_{new}, T_{new}, u_k)| \geq m - r - k - 1 \tag{4}$$

And thus:

$$|ALG(S,T,o)| = |ALG(S_{new}, T_{new}, u_k)| + k$$
$$\geq \tfrac{1}{3}(|OPT(S_{new}, T_{new}, u_k)| + T_{new} - 1) + k \text{ by Lemma 6}$$
$$\geq \tfrac{1}{3}(m - r - k - 1 + T - k - 1) + k \text{ by eqn. 4}$$
$$= \tfrac{1}{3}(2m) + \tfrac{1}{3}(-r + k - 2) \geq 2m/3 \text{ unless } k = r \text{ or } k = r + 1$$

For the cases of $k = r$ and $k = r + 1$, we follow the same steps we did for these cases in the proof of Lemma 6 to modify the OPT path.

Case $k = r$: Then by Case 2.2.2.1 of the proof of Lemma 6, we have $|OPT(S_{new}, T_{new}, o_{new})| \geq m - r - k + 1$. So:

$$
\begin{aligned}
|ALG(S, T, o)| &= |ALG(S_{new}, T_{new}, u_k)| + k \\
&\geq \tfrac{1}{3}(|OPT(S_{new}, T_{new}, u_k)| + T_{new} - 1) + k \text{ by Lemma 6} \\
&\geq \tfrac{1}{3}(m - r - k + 1 + T - k - 1) + k \\
&\geq \tfrac{1}{3}(2m - 3k) + k \text{ since } T = m \text{ and } r = k \\
&\geq 2m/3
\end{aligned}
$$

Case $k = r + 1$: Then by Case 2.2.2.2 of the proof of Lemma 6, we have $|OPT(S_{new}, T_{new}, o_{new})| \geq m - r - k$. So:

$$
\begin{aligned}
|ALG(S, T, o)| &\geq \tfrac{1}{3}(m - r - k + T - k - 1) + k \\
&\geq \tfrac{1}{3}(2m - 3k) + k \text{ since } T = m \text{ and } r = k - 1 \\
&\geq 2m/3
\end{aligned}
$$

We have shown that for all cases, $|ALG(S, T, o)| \geq 2m/3$, so the proof is complete.

◀

We now show that the approximation ratio of $2/3$ for TWOCHAIN is tight.

▶ **Theorem 8.** *The approximation ratio of* TWOCHAIN *for URDARP is no greater than* $2/3$.

**Proof.** Consider an instance with three requests in a single chain with no requests starting at the origin $o$. Let $T = 4$. TWOCHAIN may select the second and third requests of the chain as its first two requests. For TWOCHAIN to drive to and then serve the two requests takes three time units. It then drives and runs out of time. On the other hand, OPT starts at the first request of the chain and completes all three requests by time $T = 4$. ◀

## 3.2    The $k$-CHAIN Algorithm

We now show that a natural generalization of TWOCHAIN, which we refer to as $k$-CHAIN (see Algorithm 2) yields at most a 7/9-approximation. This polynomial-time algorithm (which is exponential in the fixed $k$ that is selected) proceeds analogously to TWOCHAIN, but rather than prioritizing requests that are the first in a 2-chain, instead it prioritizes requests that are the first in a $k$-chain. One might expect that this algorithm yields a $k/(k+1)$-approximation but we show that, surprisingly, there exists an instance of *URDARP* where $k$-CHAIN earns at most 7/9 times the revenue of the optimal solution.

▶ **Theorem 9.** *The $k$-CHAIN algorithm yields at most a 7/9-approximation.*

**Proof.** In the input instance (see Figure 2) there is a chain of $c + k$ requests, for two positive integers $c$ and $k$, and the origin, $o$, is at the start of this chain. Denote these $c + k$ requests as $(v_0, v_1), (v_1, v_2), (v_2, v_3), \ldots, (v_{c-1}, v_c), \ldots, (v_{c+k-1}, v_{c+k})$, so $o = v_0$. In addition, for each point $v_i$, for $i = 1, 2, \ldots, c$, there is another pair of requests: one that leaves from $v_i$ to a point not on the chain, call it $v_i'$, and another that leaves from $v_i'$ and returns to $v_i$, forming a total of $c$ loops each of length 2.

Let $T = 3c$. Then $\text{OPT}(S, T, o) = \text{OPT}(S, 3c, v_0) = 3c$ since OPT can serve all the loops "on the way" as it proceeds across from $v_1$ to $v_{c+k}$. I.e., the optimal schedule is

$(v_0, v_1), (v_1, v_1'), (v_1', v_1), (v_1, v_2), (v_2, v_2'), (v_2', v_2), (v_2, v_3), \ldots, (v_{c+k-1}, v_{c+k})$.

On the other hand, Algorithm 2, which prioritizes chains of length $k$, may choose one request at a time from the "spine" of this input instance, and end up serving all the requests

---

**Algorithm 2** The $k$-CHAIN algorithm

---

1: Input: Set $S$ of requests, time limit $T$, origin $o$
2: Set $t := T$
3: For $i = 1 \ldots k$, for each request $r \in S$, add $r$ to $S_i$ if request $r$ is followed by a chain of requests of length $i - 1$. So $r \in S_i$ means $r$ is the start of a chain of length $i$. Note that if $r \in S_i$ then we also have $r \in S_j$ for all $j < i$.
4: **while** $t > 0$ **do**
5:     Let $j$ be the highest value for which $S_j$ has a request starting at $o$.
6:     **if** $j > 0$ **then**
7:         Choose one such request $(o, b)$ from $S_j$ and serve it.
8:         $t := t - 1$
9:         Remove $(o, b)$ from $S$ and update the sets $S_i$ for $i = 1 \ldots k$ as needed.
10:         $o := b$
11:     **else if** $t \geq 2$ and requests remain in $S$ **then**
12:         Let $j$ be the highest value for which $S_j$ is non-empty.
13:         Choose one request $(a, b)$ from $S_j$.
14:         Drive from $o$ to $a$, then serve the request $(a, b)$.
15:         $t := t - 2$
16:         Remove $(a, b)$ from $S$ and update the sets $S_i$ for $i = 1 \ldots k$ as needed.
17:         $o := b$
18:     **else**         ▷ no requests remain (that we have enough time to serve)
19:         $t := t - 1$

---
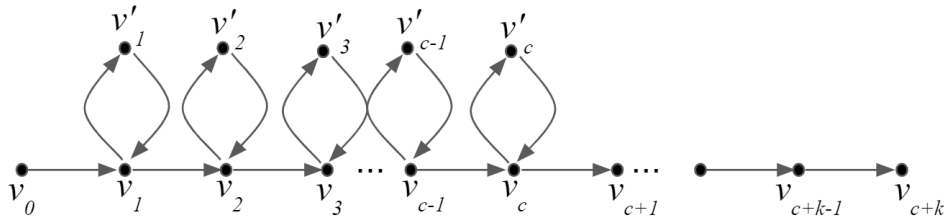


**Figure 2** An instance showing that the $k$-CHAIN algorithm has approximation ratio at most 7/9.

along the straight path first, rather than serving the loops along the way. In this event at time $c + k$ it must then go back and serve as many loops (chains of length 2) as it can in the remaining $3c - (c + k) = 2c - k$ units of time, expending one unit of time on an empty drive to the next loop after serving each loop. Hence:

$$|\text{ALG}(S, T, o)| = c + k + \left\lfloor \tfrac{2}{3}(2c - k) \right\rfloor$$

And note that

$$\lim_{c \to \infty} \frac{|\text{ALG}(S, T, o)|}{|\text{OPT}(S, T, o)|} = \lim_{c \to \infty} \frac{c + k + \left\lfloor \tfrac{2}{3}(2c - k) \right\rfloor}{3c} = \frac{7}{9}.$$

◀

## 3.3 The LONGEST-CHAIN-FIRST Algorithm

We now provide a brief discussion of the greedy algorithm that serves the longest chain of requests first, removing these requests from the instance, then serves the longest chain among the remaining requests and removes these, and continues this way until time runs out. We refer to this algorithm as the LONGEST-CHAIN-FIRST (LCF) algorithm.

Implementation of this algorithm requires a solution to the longest trail problem, where a trail is defined as a path with no repeated edges, i.e., a chain of DARP requests. Although the longest trail problem is NP-hard [10, 11], a standard poly-time algorithm that simply requires a topological sort on the vertices of the acyclic graph as a pre-processing step can be employed for finding the longest trail in acyclic graphs. We use the term *request-graph* to refer to the directed multigraph where each request is represented by an edge in the graph and each vertex in the graph is the source or destination of a request. So if we consider the space of inputs where the request-graphs are acyclic, we can employ the poly-time algorithm for finding the longest trail in an acyclic graph to implement the greedy LCF algorithm.
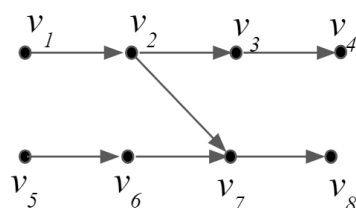


**Figure 3** An instance showing that the LCF algorithm has an approximation ratio of at most 5/6.

It turns out that even when restricting to acyclic graphs, uniform revenues and a uniform metric space, the LCF algorithm yields an approximation ratio of at most 5/6.

▶ **Theorem 10.** *The approximation ratio of the* LCF *algorithm for URDARP on acyclic request-graphs is at most* 5/6*.*

**Proof.** Please refer to Figure 3. The instance depicts a request graph for which $T = 8$ and the origin is one unit away from the source of all requests. An optimal solution is to serve the top 3-chain followed by the bottom 3-chain for a total revenue of 6. The LCF algorithm may instead start with $(v_1, v_2)$, but then take $(v_2, v_7)$, finishing with $(v_7, v_8)$. LCF would then require an empty drive to a remaining 2-chain, but after serving the 2-chain, there would be no time left to drive to and serve any more requests, so LCF earns a revenue of only 5. ◀

We expect that in future work we will be able to prove that LCF does indeed yield a 5/6 approximation for URDARP on acyclic request graphs.

─── **References** ───

**1** Giorgio Ausiello, Vincenzo Bonifaci, and Luigi Laura. The online prize-collecting traveling salesman problem. *Information Processing Letters*, 107(6):199–204, 2008.

**2** Baruch Awerbuch, Yossi Azar, Avrim Blum, and Santosh Vempala. New approximation guarantees for minimum-weight k-trees and prize-collecting salesmen. *SIAM Journal on Computing*, 28(1):254–262, 1998.

**3** Egon Balas. The prize collecting traveling salesman problem. *Networks*, 19(6):621–636, 1989.

**4** Daniel Bienstock, Michel X Goemans, David Simchi-Levi, and David Williamson. A note on the prize collecting traveling salesman problem. *Mathematical programming*, 59(1-3):413–420, 1993.

**5**    Avrim Blum, Shuchi Chawla, David R Karger, Terran Lane, Adam Meyerson, and Maria
        Minkoff. Approximation algorithms for orienteering and discounted-reward TSP. *SIAM
        Journal on Computing*, 37(2):653–670, 2007.

**6**    Ananya Christman, Christine Chung, Nicholas Jaczko, Marina Milan, Anna Vasilchenko,
        and Scott Westvold. Revenue Maximization in Online Dial-A-Ride. In *17th Workshop on
        Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS
        2017)*, volume 59, pages 1:1–1:15, Dagstuhl, Germany, 2017. `doi:10.4230/OASIcs.ATMOS.
        2017.1`.

**7**    Ananya Christman, William Forcier, and Aayam Poudel. From theory to practice: maximizing
        revenues for on-line dial-a-ride. *Journal of Combinatorial Optimization*, 35(2):512–529, 2018.

**8**    Jean-François Cordeau and Gilbert Laporte. The dial-a-ride problem: models and algorithms.
        *Annals of Operations Research*, 153(1):29–46, Sep 2007. `doi:10.1007/s10479-007-0170-8`.

**9**    Yves Molenbruch, Kris Braekers, and An Caris. Typology and literature review for dial-a-ride
        problems. *Annals of Operations Research*, 259(1):295–325, 2017.

**10**   Christos H Papadimitriou and Umesh V Vazirani. On two geometric problems related to the
        travelling salesman problem. *Journal of Algorithms*, 5(2):231–246, 1984.

**11**   stackexchange.com.   Is  the  longest  trail  problem  easier  than  the  longest
        path   problem?            `https://cstheory.stackexchange.com/questions/20682/
        is-the-longest-trail-problem-easier-than-the-longest-path-problem`.       Accessed:
        2019-02-19.

## A    Appendix

We first show that in all subcases of Case 2.2.2.1 of Lemma 6, $P$ starts at $o_{new}$, has at least
$m - r - k + 1$ requests from $S_{new}$, and has length $T_{new}$.

**Case 2.2.2.1:**  If $k = r$ then every request $(u_{i-1}, u_i)$ is in $P^*$, and in particular both $(u_{k-2}, u_{k-1})$ and
$(u_{k-1}, u_k)$ are in $P^*$. Either $(u_{k-1}, u_k)$ is the last drive of $P^*$ or it is not.

**Case 2.2.2.1.1:**  $(u_{k-1}, u_k)$ is the last drive of $P^*$. Then there is a drive $(u_{k-1}, y)$ immediately following
$(u_{k-2}, u_{k-1})$ in $P^*$. There are three subcases.

**Case 2.2.2.1.1.1:**  $(u_{k-1}, y) = (u_{k-1}, u_k)$ Then we delete the last $k + 1$ drives from $P^*$ to make $P$. Since
the $k + 1$ drives include $(u_{k-2}, u_{k-1})$ and $(u_{k-1}, u_k)$, which are not in $S_{new}$, then $P$ loses
at most $k - 1$ requests from $S_{new}$ and is a total $k + 1$ shorter than $P^*$. So $P$ has at least
$m - r - (k - 1) = m - r - k + 1$ requests from $S_{new}$ and has length $|P^*| - (k + 1) = T_{new}$.

**Case 2.2.2.1.1.2:**  $(u_{k-1}, y) \neq (u_{k-1}, u_k)$ and $(u_{k-1}, y) \notin S_{new}$. We make $P$ from $P^*$ as follows. We first
make $\hat{P}$ from $P^*$ by deleting $(u_{k-1}, u_k)$ and replacing $(u_{k-2}, u_{k-1})$ and $(u_{k-1}, y)$ by the
shortcut $(u_{k-2}, y)$. So $\hat{P}$ has at least $m - r$ requests from $S_{new}$ (since none of the deleted
requests are requests from $S_{new}$) and length two shorter than $P^*$. Then we make $P$ from
$\hat{P}$ by deleting the last $(k - 1)$ drives from $\hat{P}$. So $P$ has at least $m - r - (k - 1)$ requests
from $S_{new}$ and length $T_{new}$.

**Case 2.2.2.1.1.3:**  $(u_{k-1}, y) \neq (u_{k-1}, u_k)$ and $(u_{k-1}, y) \in S_{new}$. Note that $u_{k-1}$ is not the source of a
request in $S'_{new}$ (those requests in $S_{new}$ that start a 2-chain, as defined in line 3 of
Algorithm 1) since if it were, TWOCHAIN would have chosen that request instead of
$(u_{k-1}, u_k)$ as the next request. So $y$ cannot be the source of a request in $S_{new}$. Let $(y, z)$
be the next drive in $P^*$ after $(u_{k-1}, y)$, and we know $(y, z)$ is not in $S_{new}$. Then we make
$P$ from $P^*$ as follows. We first make $\hat{P}$ from $P^*$ by deleting $(u_{k-1}, u_k)$ and replacing
$(u_{k-2}, u_{k-1}), (u_{k-1}, y)$ and $(y, z)$ by the shortcut $(u_{k-2}, z)$. Then $\hat{P}$ has at least $m - r - 1$
requests from $S_{new}$ (since the only request from $S_{new}$ $\hat{P}$ lost was $(u_{k-1}, y)$) and length
three shorter than $P^*$. We then make $P$ from $\hat{P}$ by deleting the last $k - 2$ drives from $\hat{P}$.

So $P$ has at least $m - r - 1 - (k - 2) = m - r - k + 1$ requests from $S_{new}$ and length $|P^*| - 3 - (k - 2) = T_{new}$.

**Case 2.2.2.1.2:** $(u_{k-1}, u_k)$ is not the last drive of $P^*$ so there is a drive $(u_k, x)$ in $P^*$. Note that $(u_k, x)$ cannot be in $S_{new}$ since if it were, then TWOCHAIN would have served it after $(u_{k-1}, u_k)$ There are several subcases.

**Case 2.2.2.1.2.1:** $(u_{k-2}, u_{k-1})$ is the last drive of $P^*$ and $(u_{k-2}, u_{k-1}) = (u_k, x)$. Then we make $P$ by deleting the last $k + 1$ drives from $P^*$. Then $P$ loses at most $k - 1$ requests from $S_{new}$ since at least 2 of the $k + 1$ drives (namely, $(u_{k-2}, u_{k-1})$ and $(u_{k-1}, u_k)$) are not in $S_{new}$. So $P$ has at least $m - r - (k - 1)$ requests from $S_{new}$ and length $T_{new}$.

**Case 2.2.2.1.2.2:** $(u_{k-2}, u_{k-1})$ is the last drive of $P^*$ and $(u_{k-2}, u_{k-1}) \neq (u_k, x)$. Then we make $P$ from $P^*$ as follows. We first make $\hat{P}$ from $P^*$ by deleting $(u_{k-2}, u_{k-1})$ and replacing $(u_{k-1}, u_k)$ and $(u_k, x)$ by the shortcut $(u_{k-1}, x)$. So $\hat{P}$ has at least $m - r$ requests from $S_{new}$ (since none of $(u_{k-2}, u_{k-1})$, $(u_{k-1}, u_k)$, and $(u_k, x)$ are in $S_{new}$) and has length two shorter than $P^*$. We then make $P$ from $\hat{P}$ by deleting the last $k - 1$ drives from $\hat{P}$. So $P$ has at least $m - r - (k - 1)$ requests from $S_{new}$ and length $|\hat{P}| - (k - 1) = |P^*| - 2 - k + 1 = T_{new}$.

**Case 2.2.2.1.2.3:** $(u_{k-2}, u_{k-1})$ is not the last drive of $P^*$ so there is a drive $(u_{k-1}, y)$ in $P^*$ and $(u_{k-1}, y) = (u_{k-1}, u_k)$. Then we make $\hat{P}$ from $P^*$ by replacing $(u_{k-2}, u_{k-1})$, $(u_{k-1}, u_k)$ and $(u_k, x)$ by the shortcut $(u_{k-2}, x)$. So $\hat{P}$ has at least $m - r$ requests from $S_{new}$ and length two shorter than $P^*$. We then make $P$ from $\hat{P}$ by deleting the last $k - 1$ drives. So $P$ has at least $m - r - (k - 1)$ requests from $S_{new}$ and length $T_{new}$.

**Case 2.2.2.1.2.4:** $(u_{k-2}, u_{k-1})$ is not the last drive of $P^*$ so there is a drive $(u_{k-1}, y)$ in $P^*$ and $(u_{k-1}, u_k) \neq (u_{k-1}, y)$ and $(u_{k-1}, y) \notin S_{new}$. Then we make $\hat{P}$ from $P^*$ by replacing $(u_{k-1}, u_k)$ and $(u_k, x)$ by $(u_{k-1}, x)$ and replacing $(u_{k-2}, u_{k-1})$ and $(u_{k-1}, y)$ by $(u_{k-2}, y)$. So $\hat{P}$ has at least $m - r$ requests from $S_{new}$ (since none of $(u_{k-1}, u_k)$, $(u_k, x)$, $(u_{k-2}, u_{k-1})$ and $(u_{k-1}, y)$ are in $S_{new}$) and length two shorter than $P^*$. Then we make $P$ from $\hat{P}$ by deleting the last $k - 1$ drives. So $P$ has at least $m - r - (k - 1)$ requests from $S_{new}$ and length $T_{new}$.

**Case 2.2.2.1.2.5:** $(u_{k-2}, u_{k-1})$ is not the last drive of $P^*$ so there is a drive $(u_{k-1}, y)$ in $P^*$ and $(u_{k-1}, u_k) \neq (u_k, y)$ and $(u_{k-1}, y) \in S_{new}$. Note that $u_{k-1}$ is not the beginning of a request in $S'_{new}$, (since if it were, TWOCHAIN would have chosen that request instead of $(u_{k-1}, u_k)$ as the next request. Thus $y$ cannot be the start of a request in $S_{new}$. Either $(u_{k-1}, y)$ is at the end of $P^*$ or let $(y, z)$ denote the next drive in $P^*$ after $(u_{k-1}, y)$ and observe that $(y, z)$ is not in $S_{new}$. There are three subcases.

**Case 2.2.2.1.2.5.1:** $(u_{k-1}, y)$ is the last drive of $P^*$. Then we make $\hat{P}$ from $P^*$ by replacing $(u_{k-1}, u_k)$ and $(u_k, x)$ by $(u_{k-1}, x)$ and deleting $(u_{k-2}, u_{k-1})$ and $(u_{k-1}, y)$. So $\hat{P}$ has at least $m - r - 1$ requests from $S_{new}$ (since the only $S_{new}$ request $\hat{P}$ lost was $(u_{k-1}, y)$) and length three shorter than $P^*$. Then we make $P$ from $\hat{P}$ by deleting the last $k - 2$ drives from $\hat{P}$. So $P$ has at least $m - r - 1 - (k - 2) = m - r - k + 1$ requests from $S_{new}$ and length $T_{new} = |P^*| - 3 - (k - 2)$ shorter than $P^*$.

**Case 2.2.2.1.2.5.2:** $(u_{k-1}, y)$ is not the last drive of $P^*$ so there is a drive $(y, z)$ and $(y, z) = (u_{k-1}, u_k)$. Then to make $\hat{P}$ from $P^*$ we replace $(u_{k-2}, u_{k-1}), (u_{k-1}, y), (y, z), (u_k, x)$ by the shortcut $(u_{k-2}, x)$ . So $\hat{P}$ has at least $m - r - 1$ $S_{new}$ drives (since the only $S_{new}$ drive $\hat{P}$ lost was $(u_{k-1}, y)$) and length three shorter than $P^*$. Then we make $P$ from $\hat{P}$ by deleting the last $k - 2$ drives from $\hat{P}$. So $P$ has at least $m - r - 1 - (k - 2) = m - r - k + 1$ $S_{new}$ drives and length $T_{new} = |P^*| - 3 - (k - 2)$ shorter than $P^*$.

**Case 2.2.2.1.2.5.3:** $(u_{k-1}, y)$ is not the last drive of $P^*$ so there is a drive $(y, z)$ and $(y, z) \neq (u_{k-1}, u_k)$. Then we make $\hat{P}$ from $P^*$ by replacing $(u_{k-1}, u_k), (u_k, x)$ by $(u_{k-1}, x)$ and replacing $(u_{k-2}, u_{k-1}), (u_{k-1}, y)$ and $(y, z)$ by $(u_{k-2}, z)$. So $\hat{P}$ has at least $m - r - 1$ $S_{new}$ drives

(since the only request from $S_{new}$ $\hat{P}$ lost was $(u_{k-1}, y)$) and length three shorter than $P^*$. Then we make $P$ from $\hat{P}$ by deleting the last $k - 2$ drives from $\hat{P}$. So $P$ has at least $m - r - 1 - (k - 2) = m - r - k + 1$ requests from $S_{new}$ and length $T_{new} = |P^*| - 3 - (k - 2)$ shorter than $P^*$.

This concludes all the subcases of Case 2.2.2.1 of Lemma 6. We now show that in all subcases of Case 2.2.2.2 of Lemma 6, $P$ starts at $o_{new}$ and has at least $m - r - k$ requests from $S_{new}$.

**Case 2.2.2.2:** $k \neq r$ so it must be that $k = r + 1$. So all but one $(u_{i-1}, u_i)$ from $P$ is in $P^*$, thus at least one of $(u_{k-2}, u_{k-1})$ and $(u_{k-1}, u_k)$ is in $P^*$.

**Case 2.2.2.2.1:** $(u_{k-1}, u_k)$ is in $P^*$. There are two subcases.

**Case 2.2.2.2.1.1:** $(u_{k-1}, u_k)$ is at the end of $P^*$. Then to make $P$ from $P^*$ we delete the last $k + 1$ drives (which include $(u_{k-1}, u_k)$). So we deleted at most $k$ requests from $S_{new}$ from $P^*$ (since $(u_{k-1}, u_k)$ is not in $S_{new}$). So $P$ has at least $m - r - k$ requests from $S_{new}$ and length $T_{new}$.

**Case 2.2.2.2.1.2:** $(u_{k-1}, u_k)$ is not at the end of $P*$. Then there is a next drive $(u_k, x)$ in $P^*$. So we first make $\hat{P}$ from $P^*$ by replacing $(u_{k-1}, u_k)$ and $(u_k, x)$ by the shortcut $(u_{k-1}, x)$. So $\hat{P}$ has length one shorter than $P^*$. Note that $(u_k, x)$ cannot be in $S_{new}$ since otherwise TWOCHAIN would have continued with a request after $(u_{k-1}, u_k)$. So $\hat{P}$ has at least $m - r$ requests from $S_{new}$. Now we make $P$ by deleting the last $k$ drives from $\hat{P}$. So $P$ has at least $m - r - k$ requests from $S_{new}$ and length $T_{new} = |P^*| - 1 - k$.

**Case 2.2.2.2.2:** $(u_{k-1}, u_k)$ is not in $P^*$, and therefore $(u_{k-2}, u_{k-1})$ is in $P^*$. There are several subcases.

**Case 2.2.2.2.2.1:** $(u_{k-2}, u_{k-1})$ is at the end of $P^*$. Then to make $P$ from $P^*$ we delete the last $k + 1$ drives (which include $(u_{k-2}, u_{k-1})$). So we deleted at most $k$ requests from $S_{new}$ from $P^*$ (since $(u_{k-2}, u_{k-1})$ is not in $S_{new}$). So $P$ has at least $m - r - k$ requests from $S_{new}$ and length $T_{new}$.

**Case 2.2.2.2.2.2:** $(u_{k-2}, u_{k-1})$ is not at the end of $P^*$ and has a next drive $(u_{k-1}, y)$ that is not in $S_{new}$. We first make $\hat{P}$ from $P^*$ by replacing $(u_{k-2}, u_{k-1})$ and $(u_{k-1}, y)$ by the shortcut $(u_{k-2}, y)$. So $\hat{P}$ has at least $m - r$ requests from $S_{new}$ and is one shorter than $P^*$. We then make $P$ from $\hat{P}$ by deleting the last $k$ drives from $\hat{P}$. So $P$ has at least $m - r - k$ requests from $S_{new}$ and length $T_{new}$.

**Case 2.2.2.2.2.3:** $(u_{k-2}, u_{k-1})$ is not at the end of $P^*$ and has a next drive $(u_{k-1}, y)$ that is in $S_{new}$ and $(u_{k-1}, y)$ is at the end of $P^*$. We first make $\hat{P}$ from $P^*$ by deleting $(u_{k-2}, u_{k-1})$ and $(u_{k-1}, y)$. So $\hat{P}$ has at least $m - r - 1$ requests from $S_{new}$ (since the only request from $S_{new}$ $\hat{P}$ lost was $(u_{k-1}, y)$) and length two shorter than $P^*$. We then make $P$ from $\hat{P}$ by deleting the last $k - 1$ drives from $\hat{P}$. So $P$ has at least $m - r - 1 - (k - 1) = m - r - k$ requests from $S_{new}$ and length $T_{new} = |P^*| - 2 - (k - 1)$.

**Case 2.2.2.2.3.4:** $(u_{k-2}, u_{k-1})$ is not at the end of $P^*$ so there is a next drive $(u_{k-1}, y)$ that is in $S_{new}$ and $(u_{k-1}, y)$ is not at the end of $P^*$ so there is a next drive $(y, z)$ in $P^*$. Then by the same reasoning as in subcase 2.2.2.1.2.5, we have that $(y, z)$ is not in $S_{new}$. To make $\hat{P}$ from $P^*$ we replace $(u_{k-2}, u_{k-1}), (u_{k-1}, y)$ and $(y, z)$ by the shortcut $(u_{k-2}, z)$. So $\hat{P}$ has at least $m - r - 1$ requests from $S_{new}$ (since the only request from $S_{new}$ $\hat{P}$ lost was $(u_{k-1}, y)$) and length two shorter than $P^*$. We then make $P$ from $\hat{P}$ by deleting the last $k - 1$ drives from $\hat{P}$. So $P$ has at least $m - r - 1 - (k - 1) = m - r - k$ requests from $S_{new}$ and length $T_{new} = |P^*| - 2 - (k - 1)$.

This concludes all the subcases of Case 2.2.2.2 of Lemma 6.