MATHEMATICAL COMMUNICATIONS
Math. Commun., Vol. **14**, No. 1, pp. 119-134 (2009)

A fast implementation of the optimal off-line algorithm for solving the k-server problem

Tomislav Rudec[1], Alfonzo Baumgartner[1] and Robert Manger[2,*]

[1] *Faculty of Electrical Engineering, University of Osijek, Kneza Trpimira 2b, 31 000 Osijek, Croatia*
[2] *Department of Mathematics, University of Zagreb, Bijenička cesta 30, 10 000 Zagreb, Croatia*

Received December 12, 2008; accepted March 2, 2009

Abstract. The optimal off-line algorithm for solving the k-server problem is usually implemented by network flows. In this paper, we first propose certain modifications to each step of the original network-flow implementation. Next, by experiments we demonstrate that the proposed modifications improve the speed of the algorithm. Finally, we investigate how similar ideas for improvement can also be applied to some related on-line algorithms.
AMS subject classifications: 05C85, 68Q25, 90B10, 90B35, 90C27, 90C35

Key words: k-server problem, off-line algorithms, on-line algorithms, optimality, implementation, network flows, execution time, experiments

1. Introduction

This paper deals with the *k-server problem* [10], where one has to decide how k mobile servers should serve a sequence of requests. The goal of an algorithm for solving such problem is not only to serve the given requests, but also to minimize the total cost of serving.

It is usually required that the solution to the k-server problem is produced in on-line fashion [7]. Indeed, each request ought to be served before the next request arrives, and each decision may be based only on already seen requests. However, in order to measure performance of on-line algorithms, it is also useful to consider off-line algorithms. Contrary to any on-line procedure, whose serving is never quite satisfactory due to lack of information on future requests, an off-line procedure knows the whole input in advance and can deal with requests as they arrive at truly minimum total cost. Thus we can speak about the optimal off-line algorithm.

On the first sight, finding an optimal solution to the k-server problem seems to be a fairly complex task even if the whole sequence of requests is known in advance. However, it has been shown that the optimal off-line algorithm can be implemented quite efficiently in polynomial time. The well known implementation from [5] is based on network flows, and it reduces the original k-server problem to a minimal-cost maximal flow problem on a suitably constructed network.

[*]Corresponding author. *Email addresses:* `Tomislav.Rudec@etfos.hr` (T. Rudec), `Alfonzo.Baumgartner@etfos.hr` (A. Baumgartner), `Robert.Manger@math.hr` (R. Manger)

The aim of this paper is to propose certain modifications to the mentioned network flow implementation of the optimal off-line algorithm. Our modifications are expected to improve the original version of the algorithm in terms of speed. The aim of the paper is also to evaluate experimentally the obtained improvements, and to consider applicability of similar ideas in some slightly different contexts.

The paper is organized as follows. Section 2 lists preliminaries about the k-server problem and the corresponding algorithms. Section 3 specifies the original network-flow implementation of the optimal off-line algorithm, which runs in k steps. Section 4 proposes modifications of the first step of the original algorithm. Section 5 does the same for the second step. Section 6 considers modifications of the remaining steps. Section 7 presents experimental measurements of speed-up that is produced by all modifications together. Section 8 briefly describes how the same or similar modifications can also be applied to network-flow implementations of some related on-line algorithms. The final Section 9 gives conclusions.

2. Preliminaries

In the *k-server problem* [10] we have k servers each of which occupies a location in a fixed metric space M. Repeatedly, a request r_i at some location x in M appears. To serve a request at x, a corresponding algorithm must move a server to x unless it already has a server at that location. Whenever the algorithm moves a server from a location x to a location y, there incurs a cost equal to the distance between x and y in M. The goal of good serving is not only to serve requests, but also to minimize the total distance moved by all servers.

In the on-line version of the k-server problem the sequence of requests in not known in advance. Each request must be served immediately before the next request arrives. In order to decide how to serve the current request r_i, an on-line algorithm may take into account only the already seen requests r_1, r_2, $\ldots$, r_{i-1}, r_i, and it cannot use any information about the future requests r_{i+1}, r_{i+2}, $\ldots$. On the other hand, in the off-line version of the problem the sequence of requests is fixed and given at the beginning. Consequently, an off-line algorithm can take together the whole sequence as input and produce the complete serving plan at once.

As a concrete instance of the k-server problem, let us consider the metric space M consisting of locations A, B, C, D, E, with distances given as shown in Figure 1. Suppose that $k = 3$ servers are initially located at A, B and E. Let the first request appear in C. Assume also that all forthcoming requests are going to appear in A, B and C and none in E. Our serving algorithm has to decide first which of the three servers should be moved to C. If the algorithm works on-line, it would possibly move the nearest server from A. But an off-line algorithm would know that such a choice is not optimal in the long run. Namely, moving the distant server from E is more profitable since it would enable that all forthcoming requests are served at no additional costs.

In this paper we are mostly concerned with the optimal off-line algorithm - OPT. As input this algorithm takes the initial configuration of servers $S^{(0)}$ and the whole sequence of requests $r_1, r_2, \ldots, r_n$. It considers all possibilities and produces a complete list of server moves that enables serving of all requests at truly minimum

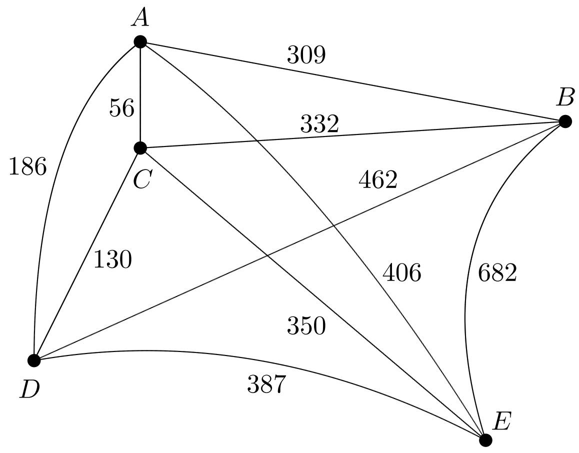

Figure 1. *A k-server problem instance*

total cost.

Although the k-server problem must usually be solved in on-line fashion, OPT is still useful as a tool for evaluation of on-line algorithms. For instance, there is an important measure of goodness of on-line algorithms involving OPT, which is called *competitiveness* [11]. An on-line algorithm is said to be competitive if its performance is only a bounded number of times worse than that of OPT on any input.

In this paper we are also concerned with the on-line *work function algorithm* (WFA) [2, 9, 10]. Although being rather complex, this algorithm is still very important since it exhibits the best characteristics regarding competitiveness [1, 2, 9]. To serve the request r_i, the WFA switches from the current server configuration $S^{(i-1)}$ to a new configuration $S^{(i)}$, obtained from $S^{(i-1)}$ by moving one server into the requested location (if necessary). Among k possibilities (any of k servers could be moved) $S^{(i)}$ is chosen so that

$$F(S^{(i)}) = C_{\mathrm{OPT}}(S^{(0)}, r_1, r_2, \ldots, r_i, S^{(i)}) + d(S^{(i-1)}, S^{(i)}) \tag{1}$$

becomes minimal. Thus the objective function $F(S^{(i)})$ is defined as a sum of two parts.

- The first part, usually called the *work function*, is the minimal total cost of starting from $S^{(0)}$, serving in turn $r_1, r_2, \ldots, r_i$, and ending up in $S^{(i)}$.
- The second part is the distance traveled by a server to switch from $S^{(i-1)}$ to $S^{(i)}$.

As we can see from (1), the WFA is closely related to OPT. Indeed, each step of the WFA consists of k off-line optimization problem instances plus some simple arithmetics. But note that the optimization problems within the WFA are not quite equivalent to those within OPT; namely there is an additional constraint regarding the final configuration of servers. Still, we expect that any implementation of OPT can be incorporated into the WFA after a slight adjustment.

Together with the original WFA, we will also consider its "lightweight" version denoted as the w-WFA [3], which is based on the idea that the sequence of previous requests and configurations should be examined through a moving window of size w. In its i-th step the w-WFA acts as if r_{i-w+1}, r_{i-w+2}, $\ldots$, r_{i-1}, r_i was the whole sequence of previous requests, and as if $S^{(i-w)}$ was the initial configuration of servers. In other words, the objective function $F()$, originally defined by (1), is redefined in the following way:

$$F(S^{(i)}) = C_{\mathrm{OPT}}(S^{(i-w)}, r_{i-w+1}, r_{i-w+2}, \ldots, r_{i-1}, r_i, S^{(i)}) + d(S^{(i-1)}, S^{(i)}). \quad (2)$$

According to [3], the w-WFA assures similar quality of serving as the original WFA but runs dramatically faster. As visible from (2), the w-WFA is again closely related to OPT; namely it again reduces to a series of slightly altered off-line optimization problems.

3. The optimal off-line algorithm

The optimal off-line algorithm OPT can be realized relatively easily by network flow techniques [4]. We now describe the original implementation from [5]. According to [5], finding the optimal strategy to serve a sequence of requests r_1, r_2, $\ldots$, r_n by k servers reduces to computing the minimal-cost maximal flow on a suitably constructed network with $2n + k + 2$ nodes. The details of this construction are shown in Figure 2.

As we can see from Figure 2, the network corresponding to the off-line problem consists of a source node $\bar{s}$, a sink node $\bar{t}$, and three additional layers of nodes. The first layer represents the initial server configuration $S^{(0)}$, i.e. the node $s_j^{(0)}$ corresponds to the initial location of the j-th server. The remaining two layers represent the whole sequence of requests, i.e. the nodes r_p and r'_p both correspond to the location of the p-th request.

Only some pairs of nodes in the network shown in Figure 2 are connected by arcs. Note that an r_p is connected only to the associated r'_p. Also, a link between an r'_p and an r_q exists only if $q > p$. All arcs are assumed to have unit capacity. The costs of arcs leaving $\bar{s}$ or entering $\bar{t}$ are 0. An arc connecting r_p with r'_p has the cost $-L$, where L is a suitably chosen very large positive number. All other arc costs are equal to *distances* between the corresponding locations.

It is obvious that the maximal flow through the network shown in Figure 2 must have the value k. Moreover, the maximal flow can be decomposed into k disjunct unit flows from $\bar{s}$ to $\bar{t}$. Each unit flow determines the trajectory of the corresponding server and the requests that are accomplished by that server. If the chosen constant L is large enough, then the minimal-cost maximal flow will be forced to use all arcs between r_p and r'_p, thus assuring that all requests will be served at minimum cost.

Actual computation of the optimal flow can be accomplished by the *flow augmentation* method [4]. We start with a flow that is not of maximum value but has the minimal cost among those with that value. Then in each step we augment the value of the current flow in such a way that it still has the minimal cost among those with the same value. After a sufficient number of steps we obtain the desired minimal-cost maximal flow.

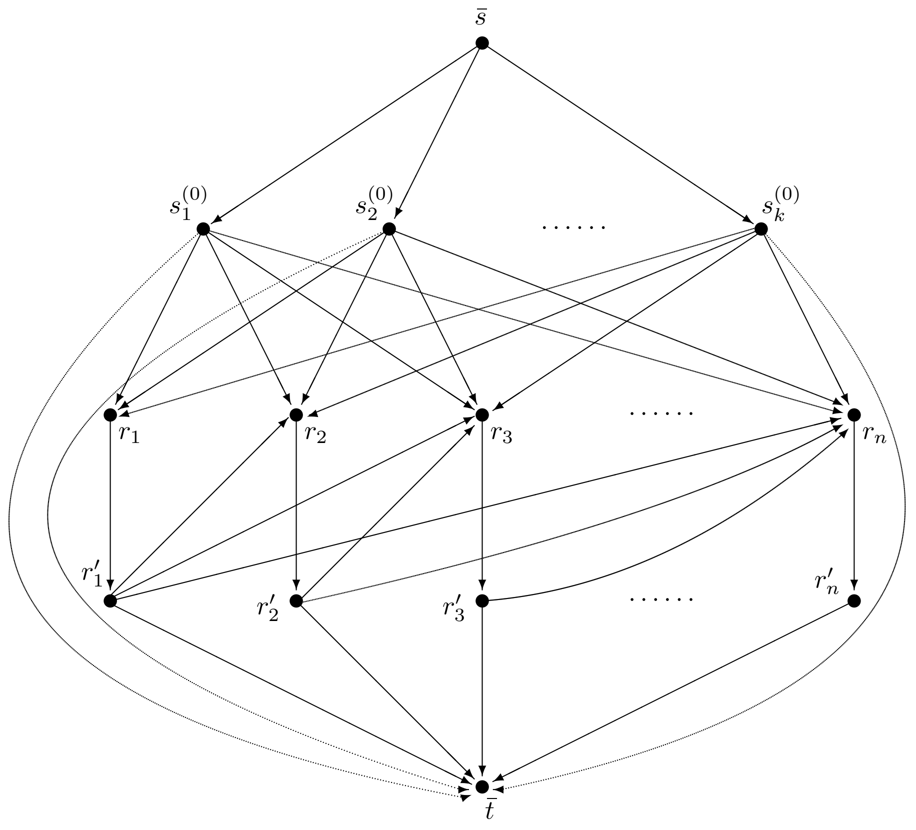

Figure 2. *The network describing the optimal solution to the off-line k-server problem*

In our particular case the computation can be started with the null flow. Namely, since the involved network is acyclic, the null flow obviously has the minimal cost among those with value 0. In each step, augmentation is achieved by solving a single-source shortest path problem in the corresponding *displacement network* [4]. The obtained shortest path determines the unit flow that has to be superimposed onto the current flow in order to obtain augmentation. Since the maximal flow has value k and each augmentation increments the flow value by one unit, computing the minimal-cost maximal flow reduces to exactly k steps.

The mentioned shortest path problems can be solved by various sub-algorithms. Relatively efficient computation is assured by Dijkstra's procedure [8]. It is well known that Dijkstra's procedure can be applied only to networks whose arc costs are nonnegative. On the first sight, our networks do not qualify since they contain costs $-L$. Still, it turns out that Dijkstra can be used after a suitable transformation of arc costs in each step. The details of that transformation can be found in [6].

Putting it all together, the network flow realization of OPT for solving the k-server problem on a sequence of n requests reduces to k steps. Each of those steps reduces to a single-source shortest path problem in a displacement network consisting of $2n + k + 2$ nodes. With some preprocessing, any shortest path problem can be solved by Dijkstra's procedure. Since Dijkstra's procedure has a quadratic computational complexity and the complexity of the mentioned preprocessing is of the same order, we can estimate that one step of the algorithm takes $\mathcal{O}((n + k)^2)$ time. Consequently, the computing time for the whole original implementation of the algorithm is $\mathcal{O}(k \cdot (n + k)^2)$.

4. Modifying the first step

For a given network, the associated displacement network can be constructed exactly as defined in [4]. However, in our case the standard construction may be simplified thanks to the fact that only unit flows and unit capacities are used. Indeed, the displacement network can quickly be obtained from the original network by reversing directions of all arcs saturated by a flow, and by changing signs of their costs. The previously mentioned shortest path problem within the displacement network consists of finding a path from the source $\bar{s}$ to the sink $\bar{t}$ with the minimal sum of arc costs.

Since our algorithm starts with the null flow, the displacement network in the first step is identical to the original network. Thus the first step reduces to finding the shortest (cheapest) path from $\bar{s}$ to $\bar{t}$ in the network shown in Figure 2. We claim that such path must include all arcs of the form $r_p \to r'_p$. Indeed, each of those arcs has a negative cost $-L$, while all other arcs have positive costs. By assumption, L is chosen so large that inclusion of $-L$ compensates for any additional positive costs. So inclusion of an additional $r_p \to r'_p$ surely makes a path shorter.

Now let us see how our shortest path looks like. Thanks to the special structure of our network, any path from $\bar{s}$ to $\bar{t}$ that includes all arcs $r_p \to r'_p$ must have the form:

$$\bar{s} \to s_x^{(0)} \to r_1 \to r'_1 \to r_2 \to r'_2 \to \cdots \to r_{n-1} \to r'_{n-1} \to r_n \to r'_n \to \bar{t}.$$

Here, $s_x^{(0)}$ is one of the nodes from the first layer in Figure 2. To fully identify the shortest path, we must choose the right x. Since all other costs are fixed, x must obviously be chosen so that the cost of the arc $s_x^{(0)} \to r_1$ is minimal. Or differently speaking, we must pick up the node $s_x^{(0)}$ representing the server that is nearest to the location of the first request. Thus determining the shortest path reduces to finding the minimum among k values.

Note that after the first step of the algorithm the arcs of the form $r_p \to r'_p$ are never used again. Indeed, because of saturation during the first step, these arcs become reversed in further steps, and they obtain very large positive costs $+L$. Inclusion of such reversed arcs surely produces paths with positive lengths. A path with a positive length can never be the shortest since there always exists at least one "shortcut" of the form $\bar{s} \to s_j^0 \to \bar{t}$ with the length 0. Consequently, arcs of the form $r_p \to r'_p$ can be discarded or ignored after the first step.

To summarize, our modification of the first step of the algorithm consists of the following.

- No general path-finding procedure is used. Instead, the shortest path is directly determined according to the specification shown above.
- Arcs of the form $r_p \to r'_p$ are only implicitly assumed to exist in the shortest path, but in fact they are removed from the network.

With this modification, the first step is accomplished in time needed to find the minimum among k values, thus in time $\mathcal{O}(k)$, which is much faster than in the original implementation.

Note also that the modified version of the first step does not use the constant L explicitly. Moreover, thanks to deletion of arcs $r_p \to r'_p$ the same constant also becomes irrelevant for the remaining steps. Thus it is in fact not necessary to specify L at all! From the practical point of view, this is a very convenient side-effect of our modification. Namely, large constants are common in network flow models, but in actual computation they can easily cause errors or numerical instabilities if they are chosen too small or too large. Our modified algorithm avoids such difficulties.

5. Modifying the second step

The displacement network in the second step of the algorithm corresponds to the flow which has been obtained in the first step. Additionally, all reversed arcs of the form $r'_p \to r_p$ have been deleted, as explained in the previous section.

Let $s_x^{(0)}$ be the node chosen in the first step as a part of the shortest path from the source $\bar{s}$ to the sink $\bar{t}$. Note that the node r'_n is also included in the same path. Then the following claims are true.

- The reversed arc $s_x^{(0)} \to \bar{s}$ cannot be used in the second or any other shortest path and may therefore be removed from the present and all further networks.
- The reversed arc $\bar{t} \to r'_n$ also cannot be used in any shortest path and can again be removed permanently from the network.
- Without $\bar{t} \to r'_n$, the node r'_n becomes isolated, and can as well be removed from the network.
- After the above mentioned removals, the remaining displacement network used in the second step becomes acyclic.

To prove that the arc $s_x^{(0)} \to \bar{s}$ really cannot take part in any shortest path, we will assume the opposite and show that such assumption leads to contradiction. Indeed, with the arc $s_x^{(0)} \to \bar{s}$ included, the shortest path obviously contains a cycle from $\bar{s}$ back to $\bar{s}$. Such cycle must have a negative length since otherwise it could be removed from the shortest path in order to make it simpler or even shorter. A cycle with a negative length can be used to modify the network flow obtained in the previous step of the algorithm, so that the flow value remains the same and the cost decreases. This is clearly a contradiction with the basic property of the flow-augmenting method: namely the flow in each step is chosen so that it always has the minimal cost among those with the same value.

To prove that the arc $\bar{t} \to r'_n$ also cannot be a part of a shortest path we can use analogous arguments as for the arc $s_x^{(0)} \to \bar{s}$. Namely, the inclusion of $\bar{t} \to r'_n$ in a shortest path implies existence of a cycle from $\bar{t}$ back to $\bar{t}$ with a negative length, which is again not possible.

Now we will prove that after the mentioned removals of obsolete arcs and nodes the displacement network in the second step really becomes acyclic. We will simply show for each particular node that it cannot be a part of any cycle.

- The source $\bar{s}$ cannot be incorporated into a cycle since its only incoming arc $s_x^{(0)} \to \bar{s}$ has been deleted.
- A similar argument is valid for the sink $\bar{t}$, thanks to removal of the arc $\bar{t} \to r'_n$.
- A node $s_j^{(0)}$ $(j \neq x)$ cannot be in a cycle since it can be entered only from $\bar{s}$, and we already know that $\bar{s}$ is not in a cycle.
- The node $s_x^{(0)}$ cannot be in a cycle since it can be entered only from r_1, r_1 can be entered only from some $s_j^{(0)}$ $(j \neq x)$, and $s_j^{(0)}$ in not in a cycle.
- The node r_1 cannot be within a cycle since its only outgoing arc leads to $s_x^{(0)}$, and $s_x^{(0)}$ is not in a cycle.
- A node r_p $(p = 2, 3, \ldots, n)$ cannot be a part of a cycle since its only outgoing arc leads to r'_{p-1}, while at the same time all outgoing arcs from r'_{p-1} lead towards some r_q with $q > p$. Thus there is no way to return back to r_p.
- The only remaining nodes are r'_p $(p = 1, 2, \ldots, n-1)$. But they alone cannot form a cycle since they are not directly connected.

Acyclicity is an important property of a network since it allows finding shortest paths very efficiently by simple one-way scanning of nodes. Namely, thanks to acyclicity, it is possible to find a "topological" ordering of nodes [8], i.e. such sequence where for each arc $u \to v$ the starting node u is put in the sequence before the ending node v. The scanning procedure processes nodes in topological order and computes for each node its distance from the source $\bar{s}$, by taking into account only the costs of its incoming arcs and the already computed distances for its direct predecessors.

It is easy to check that for our particular network the following sequence of nodes determines a topological ordering:

$$\bar{s}, s_1^{(0)}, s_2^{(0)}, \ldots, s_{x-1}^{(0)}, s_{x+1}^{(0)}, \ldots, s_k^{(0)}, r_1, s_x^{(0)}, r_2, r'_1, r_3, r'_2, \ldots, r_n, r'_{n-1}, \bar{t}.$$

Thus to compute the shortest path efficiently, the nodes should be scanned in the order shown above.

Putting it all together, our modification of the second step of the algorithm consists of the following.

- The arcs $\bar{s} \to s_x^{(0)}$ and $r'_n \to \bar{t}$ used in the previous shortest path are permanently removed from the network.
- The isolated node r'_n is also permanently removed from the network.
- The shortest path problem in the remaining displacement network is solved by scanning the nodes in the topological order shown above.

Since a suitable topological ordering of nodes is already known in advance, sorting the nodes takes no time. The proposed scanning procedure takes only the time needed to examine each arc in the network exactly once. On the other hand, the Dijkstra procedure combined with preprocessing of arc costs would process each arc at least twice. Moreover, Dijkstra would also require a lot of additional operations for finding certain minima. Consequently, our modified implementation of the second step should run considerably faster than the standard combination of Dijkstra with preprocessing. A more accurate analysis of operations shows that the speedup within the second step should be at least 4.

6. Further modifications

Now we consider the third, fourth, or any of the remaining steps of the algorithm. The displacement network now corresponds to the flow that has been obtained in the step preceding the current step. Many arcs and nodes have already been deleted, according to the rules stated in Sections 4 and 5 and the rules that will shortly be introduced in this section.

Let $\bar{s} \to s_x^{(0)}$ be the first arc of the shortest path obtained in the previous step. Let $r'_y \to \bar{t}$ be the last arc of the same path. Then the following claims are true.

- The reversed arc $s_x^{(0)} \to \bar{s}$ cannot be used in any shortest path and may therefore be removed from the current and all further networks.
- The reversed arc $\bar{t} \to r'_y$ also cannot be used in any shortest path and can again be removed permanently from the network.
- Without $\bar{t} \to r'_y$, the node r'_y becomes inaccessible since it does not have any other incoming arc. Thus r'_y can as well be removed from the network together with its outgoing arcs.

The first two claims above are in fact generalizations of similar claims from Section 5, and they can be proved in the same way as in Section 5. The third claim is a consequence of the following two facts.

- Initially, the node r'_y has only one incoming arc - see Figure 2.
- For any chosen node, the number of incoming (or outgoing) arcs remains constant through the whole algorithm.

To see why the number of incoming (or outgoing) arcs for a given node cannot change, let us remember that in our networks only unit flows and unit capacities are used. A new unit flow through the chosen node will saturate exactly one incoming and one outgoing arc, thus reversing their directions while keeping the total number of incoming (or outgoing) arcs the same.

The three claims above assure that the current displacement network can be simplified. However, contrary to Section 5, now we cannot guarantee that such simplified network is acyclic. Thus the associated shortest path problem cannot be solved by straightforward one-way scanning of nodes, as it has been done in Section 5. Instead, a more general path-finding procedure must be used, such as Dijkstra with preprocessing of arc costs. Still, thanks to special properties of our network, it is possible to use a slightly modified version of Dijkstra. Namely, by similar reasoning as in the previous paragraph, it can be shown that each node r_p has only one outgoing arc. Moreover, according to [6], the preprocessed cost of that outgoing arc must be 0. A consequence is that within the Dijkstra procedure it is necessary to maintain a list of only n nodes r_p, instead of all $k + 2n$ nodes. This modification produces a version of Dijkstra whose computing time is slightly better than for the standard version, although still within the same order of magnitude.

It can easily happen that the shortest path obtained in the current step turns out to have the length 0. We claim that in such case the whole algorithm can be stopped since the flow from the previous step already describes the optimal solution. To prove this claim, let us note that the zero-length shortest path can always be chosen as a shortcut of the form

$$\bar{s} \to s_j^{(0)} \to \bar{t},$$

where $s_j^{(0)}$ is a suitable node not affected by the present flow. For reasons similar as before, the arcs constituting the shortcut can be removed from further networks. Thus the displacement network in the next step will look almost the same as in the current step, except that the above two arcs will be missing. Consequently, the next step will again produce a zero-length shortest path in a form of another shortcut, ..., and so on until the last step. Emergence of a zero-length shortest path in fact means that our particular instance of the off-line k-server problem is solved optimally by less than k servers, so that the remaining servers should stay idle.

In accordance with the observed properties, we propose the following modification of the third, fourth, or any further step of the algorithm.

- The arcs $\bar{s} \to s_x^{(0)}$ and $r'_y \to \bar{t}$ used in the previous shortest path are permanently removed from the network.
- The node r'_y is also permanently removed from the network together with its outgoing arcs.
- The shortest path problem in the remaining displacement network is solved by a customized version of Dijkstra with appropriate preprocessing of arc costs.
- If the newly obtained shortest path has the length 0, then the whole algorithm is stopped.

Thanks to removal of arcs and nodes, the displacement network in each step becomes simpler, thus enabling faster execution.

7. Experimental evaluation of speedup

In order to produce experimental results, we have developed two C++ programs. The first of them implements the optimal off-line algorithm OPT exactly as it has been described in Section 3. The second program follows our improved implementation of the same algorithm, thus incorporating all modifications described in Sections 4-6. Both programs have been tested on the same k-server problem instances, by using a computer with a 2.8 GHz CPU and 2GB of memory. The obtained solutions as well as the corresponding computing times have been recorded.

As expected, for each particular problem instance both versions of OPT have produced exactly the same solutions, i.e. the same server trajectories and the same serving costs. However, the computing times needed to reach those solutions have been quite different. Thus experimenting has enabled measuring of the overall speedup produced cumulatively by all modifications of OPT described in Sections 4-6.

The results of our experiments regarding speedup are presented in Tables 1-3. The first table contains the absolute computing times (in milliseconds) for the original OPT. Similarly, Table 2 lists the absolute computing times (again in milliseconds) for the modified OPT. Table 3 gives relative speedups of the modified OPT compared to the original OPT, i.e. it contains entries from the first table divided by the corresponding entries from the second table. In each table, one entry corresponds to one particular problem instance. Thereby, the problem instances are organized in columns and rows, according to their parameters n (request sequence length) and k (number of servers), respectively.

	$n = 1000$	$n = 1500$	$n = 2000$	$n = 2500$	$n = 3000$
$k = 2$	218	468	828	1188	1704
$k = 3$	313	704	1250	1812	2594
$k = 5$	531	1203	2125	3078	4391
$k = 10$	1110	2422	4281	6140	8859
$k = 20$	2218	4891	8610	12375	17703

Table 1. *Computing times of the original OPT (in milliseconds)*

For very short request sequences both versions of OPT run very quickly, so that accurate measurement of time becomes impossible. Therefore only larger problem instances are shown in our tables. For such instances, the modified OPT is always at least 5 times faster than the original OPT. The best result is obtained for $k = 2$ and $n = 1000$, where the speedup reaches 13.6. In general, we can observe that for a fixed n the speedup becomes better as k becomes smaller.

The observed general behavior of the speedup regarding k is easy to explain. Namely, in the modified OPT the first two steps have been substantially improved compared to the original OPT. On the other hand, the improvement of the remaining steps is not so dramatic, i.e. the respective computing times are still within the same order of magnitude as in the original implementation. If k is small, then the relative

	$n = 1000$	$n = 1500$	$n = 2000$	$n = 2500$	$n = 3000$
$k = 2$	16	47	78	125	187
$k = 3$	31	78	141	234	343
$k = 5$	62	156	296	453	640
$k = 10$	172	360	640	1000	1390
$k = 20$	343	750	1343	2078	2937

Table 2. *Computing times of the modified OPT (in milliseconds)*

	$n = 1000$	$n = 1500$	$n = 2000$	$n = 2500$	$n = 3000$
$k = 2$	13.625	9.957	10.615	9.504	9.112
$k = 3$	10.097	9.026	8.865	7.744	7.563
$k = 5$	8.565	7.712	7.179	6.795	6.861
$k = 10$	6.453	6.728	6.689	6.140	6.373
$k = 20$	6.466	6.521	6.411	5.955	6.028

Table 3. *Speedup of the modified OPT vs. the original OPT*

proportion of the well improved first two steps within the whole algorithm becomes larger, so that the overall speedup rises.

There are however some entries in Table 3 that do not conform to the above rule. For instance, the speedup for $k = 20$ and $n = 1000$ happens to be better than for $k = 10$ and $n = 1000$. Such exceptions can be explained by peculiarities of particular concrete problem instances. Namely, both versions of OPT can take advantage of some special data values and occasionally produce some additional savings.

8. Adjustments for on-line algorithms

Now we are concerned with the question of how the presented improvements of OPT can be applied to speed up the work function algorithm - WFA. As it has already been observed, the optimization problem instances within the WFA have similar but slightly different form than those solved by OPT. Therefore the WFA cannot be improved simply by calling the already improved versions of OPT. Instead, we must first find a suitable network flow formulation for the WFA itself, which should be similar to the one used for OPT. Then we should try to apply similar modifications to the WFA as we have done for OPT.

A network flow implementation of the WFA can be obtained by following definition (1). According to (1), the i-th step of the WFA can be reduced to k minimal-cost maximal flow problems. Each of those flow problems solves one optimization problem instance from (1), and uses a network that is slightly different than the one shown in Figure 2. Network redesign is needed to implement the constraint within (1) dealing with the final configuration of servers $S^{(i)}$. There are more possibilities how such adjusted network should look like. One version is shown in Figure 3.

The network in Figure 3 consists of $2i + 2k$ nodes. Arc costs and capacities are determined similarly as it has been explained for the network in Figure 2. The main difference compared to Figure 2 is that the fourth layer of nodes has been added, which is analogous to the first layer, and which specifies the currently chosen version of the final server configuration $S^{(i)}$. Note that the second and third layer now correspond only to the requests $r_1, r_2, \ldots, r_{i-1}$. Still, since the final configuration $S^{(i)}$ always covers the location of the last request r_i, we are sure that r_i will also be served at no additional cost.

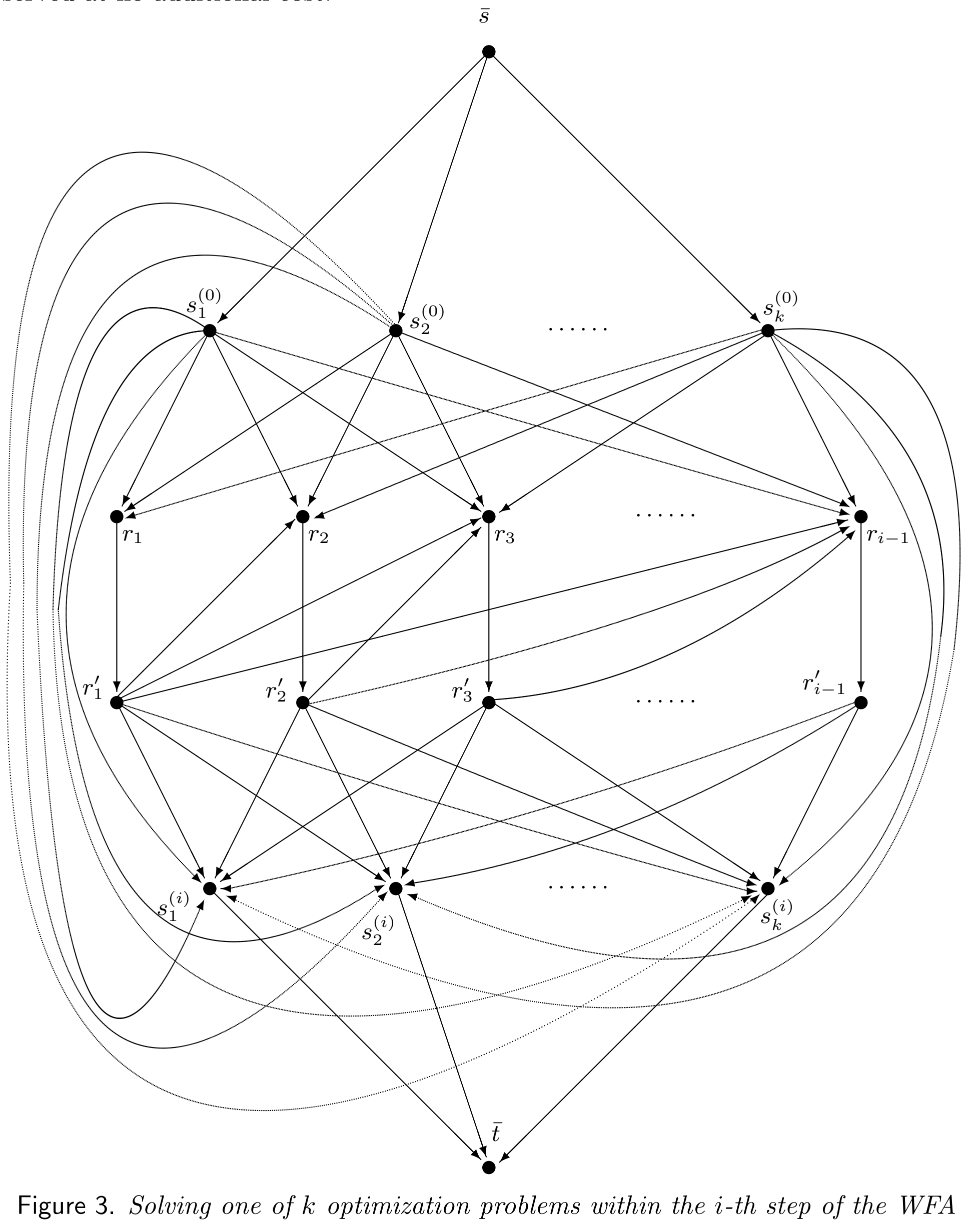

Figure 3. *Solving one of k optimization problems within the i-th step of the WFA*

When we switch from one particular version of $S^{(i)}$ to another, the structure of the whole network remains the same, only the costs of arcs entering the fourth level must be adjusted in order to reflect different final setting of servers.

Actual computation of the optimal flow in the network shown in Figure 3 can again be accomplished by the flow augmentation method, which now reduces to k steps each involving one single-source shortest path problem in a displacement network having $2i + 2k$ nodes. All path problems can again be solved by Dijkstra's procedure after appropriate preprocessing. Since the i-th step of the WFA reduces to k network flow problem instances, it follows that the computing time of the i-th step amounts to $\mathcal{O}(k^2 \cdot (i+k)^2)$.

The described implementation of the WFA can be improved by speeding up the flow augmentation algorithm that solves each particular network flow problem instance given by Figure 3. After careful analysis we have found out that most modifications of particular steps of OPT, which have been described in Sections 4-6, can as well be applied within the context of Figure 3. More precisely, the following claims are true.

- In the first step of the flow augmenting algorithm: the shortest path has the form
$$\bar{s} \to s_x^{(0)} \to r_1 \to r_1' \to r_2 \to r_2' \to \cdots \to r_{i-1} \to r_{i-1}' \to s_y^{(i)} \to \bar{t}.$$
Here x is chosen so that the cost of the arc $s_x^{(0)} \to r_1$ is minimal, and y is chosen so that the cost of the arc $r_{i-1}' \to s_y^{(i)}$ is minimal. Thus the first step can be accomplished in time $\mathcal{O}(k)$, since it reduces to finding two independent minima, each among k values.

- After the first step of the flow augmenting algorithm: all arcs of the form $r_p \to r_p'$ can permanently be removed from the network. Their costs do not need to be specified explicitly.

- At the beginning of the second, third, or any further step of the flow augmenting algorithm: the arcs of the form $\bar{s} \to s_x^{(0)}$ and $s_y^{(i)} \to \bar{t}$, which have been used within the shortest path in the previous step, can permanently be removed from the network.

- In the second step of the flow augmenting algorithm: the displacement network is acyclic. Thus the shortest path problem in the second step can be solved by scanning the nodes in topological order. One appropriate ordering of nodes is:
$$\begin{gathered}\bar{s}, s_1^{(0)}, s_2^{(0)}, \ldots, s_{x-1}^{(0)}, s_{x+1}^{(0)}, \ldots, s_k^{(0)}, r_1, s_x^{(0)}, r_2, r_1', r_3, r_2', \ldots \\ \ldots, r_{i-1}, r_{i-2}', s_y^{(i)}, r_{i-1}', s_1^{(i)}, s_2^{(i)}, \ldots, s_{y-1}^{(i)}, s_{y+1}^{(i)}, \ldots, s_k^{(i)}, \bar{t}.\end{gathered}$$

The proofs of the above claims are similar to those given in Sections 4-6. By applying the listed modifications of the flow augmenting algorithm, we can obtain a similar speedup of the WFA as achieved for OPT.

Note that the network flow techniques described in this section can also be applied to the lightweight version of the work function algorithm - the w-WFA. More precisely, the i-th step of the w-WFA can be reduced to k minimal-cost maximal flow problems with networks built according to Figure 3. But now each network consists of only $2w + 2k$ nodes. Actual computation can again be accomplished by the flow augmentation method and Dijkstra's procedure. The resulting computing time of the w-WFA is $\mathcal{O}(k^2 \cdot (w + k)^2)$ per step. We see that the w-WFA is more suitable for practical purposes than the WFA since its computing time does not rise from step to step. After applying the improvements from this section, the network implementation of the w-WFA becomes faster and therefore even more suitable for practical use.

9. Conclusions

Although the k-server problem is in essence an on-line problem, it is still interesting to consider the corresponding optimal off-line algorithm. Namely, a fast implementation of the optimal off-line algorithm can provide benchmarks for assessing performance of on-line algorithms. Also, such implementation can be used as a building block within certain complex on-line algorithms that rely on solving some auxiliary off-line subproblems.

In this paper we have described how the conventional network flow implementation of the optimal off-line algorithm can be improved in terms of speed. According to the presented experimental measurements, our improved version of the algorithm is indeed considerably faster than the original version, especially if the number of servers k is relatively small. Most of the proposed modifications of the off-line algorithm can as well be applied within the mentioned complex on-line algorithms, but after some adjustments.

The presented ideas turn out to be quite successful in speeding up the first two steps of the optimal off-line algorithm, while their effect on the remaining steps is not so significant. Our future plan is to consider and evaluate some additional ideas for improvement of that last part of the algorithm.

References

[1] Y. Bartal, E. Grove, *The harmonic k-server algorithm is competitive*, Journal of the ACM **47**(2000), 1-15.

[2] Y. Bartal, E. Koutsoupias, *On the competitive ratio of the work function algorithm for the k-server problem*, Theoretical Computer Science **324**(2004), 337-345.

[3] A. Baumgartner, R. Manger, Ž. Hocenski, *Work function algorithm with a moving window for solving the on-line k-server problem*, Journal of Computing and Information Technology **15**(2007), 325-330.

[4] M. S. Bazaraa, J. J. Jarvis, H. D. Sherali, *Linear Programming and Network Flows*, Wiley-Interscience, New York, 2004.

[5] M. Chrobak, H. Karloff, T. H. Payne, S. Vishwanathan, *New results on server problems*, SIAM Journal on Discrete Mathematics **4**(1991), 172-181.

[6] J. Edmonds, R. M. Karp, *Theoretical improvements in algorithmic efficiency for network flow problems*, Journal of the ACM **19**(1972), 248-264.

[7] S. Irani, A. R. Karlin, *Online computation*, in: *Approximation Algorithms for NP-Hard Problems*, (D. Hochbaum, Ed.), PWS Publishing Company, 1997, 521-564.

[8] D. Jungnickel, *Graphs, Networks and Algorithms*, Springer, Berlin, 2005.

[9] E. Koutsoupias, C. Papadimitrou, *On the k-server conjecture*, in: *Proceedings of the 26-th Annual ACM Symposium on Theory of Computing, Montreal, Quebec, Canada, May 23-25, 1994*, (F.T. Leighton, M. Goodrich, Eds.), ACM Press, 1994, 507-511.

[10] M. Manasse, L. A. McGeoch, D. Sleator, *Competitive algorithms for server problems*, Journal of Algorithms **11**(1990), 208-230.

[11] D. Sleator, R. E. Tarjan, *Amortized efficiency of list update and paging rules*, Communications of the ACM **28**(1985), 202-208.