Implementations and empirical comparison of K shortest loopless path algorithms

Marta M. B. Pascoal

Departamento de Matemática da Universidade de Coimbra,
Apartado 3008, 3001-454 Coimbra, Portugal
Phone: +351 239 791150, Fax: +351 239 832568

Instituto de Engenharia de Sistemas e Computadores – Coimbra
Rua Antero de Quental, 199, 3000-033 Coimbra, Portugal

E-mail: marta@mat.uc.pt

November 2006

Abstract: The first work on the network optimisation problem of ranking K shortest paths, $K \in \mathbb{N}$, appeared in 1959, more or less simultaneously with the first papers on the shortest path problem. Since then the particular problem (when $K = 1$) has merited much more attention than its generalisation. Nevertheless, several titles are included on the very complete bibliography about the K shortest paths problem, online at `http://www.ics.uci.edu/~eppstein/bibs/kpath.bib`, many of them concerning problems related with the K shortest paths problem and real word applications. With the development of computers and data structures there has been an increasing interest from the researchers on this problem, as larger problems resulting from real world applications, demanding more space of memory as well as quick responses, can now be solved.

In this paper we focus on the K shortest loopless paths problem, the variant where paths are not allowed to have repeated nodes. We survey algorithms for this problem, introduce a new method to solve it, and compare empirically their implementations.

Keywords: network optimisation, shortest path, K shortest loopless paths, algorithms.

1 Introduction

The ranking of shortest paths is a classical network optimisation problem. The first title on this topic appeared in 1959 [6], more or less at the same time the first papers on the shortest path problem, the resulting problem when $K = 1$, have been published. Since then the particular problem (for $K = 1$) has merited much more attention from the researchers than its generalisation. In fact, while hundreds of references on the shortest path problem can be found in the specialised literature (see, for instance, [2]) only around fifty titles are included on the very complete bibliography on the K shortest paths problem, available at `http://www.ics.uci.edu/~eppstein/bibs/kpath.bib`. Moreover, most of the titles in this bibliography concern problems related with the ranking of shortest paths problem and real word applications.

Some reasons for the less interest that the K shortest paths problem has merited from the researchers can be pointed out, one of them being the great amount of data manipulated in the general problem. In fact, computers random access memory is becoming cheaper and data structure more efficient but not many years ago large problems, as those resulting from real world applications, could only be solved on super computers. Nevertheless, ranking loopless paths can be used to obtain alternative solutions to the optimal, when it is intended to look for paths subject to additional constraints, or as a subproblem of other combinatorial problems, and, despite the less interest on the problem, some works on this subject have been published.

It is common to consider two types of problems: an unconstrained one in which all paths are allowed, the K shortest paths problem, and another one in which only loopless paths, that is paths with no repeated nodes, are admitted, the K shortest loopless paths problem. These two variants of the problem are closely related. In general the later one is more interesting from a practical point of view, however its resolution is harder, as a consequence of the constraint on repeating nodes. In this work we focus on approaches for solving the K shortest loopless paths problem.

Section 2 first provides some notation and defines the problem of ranking the K shortest loopless paths. Then the algorithms for this problem are briefly introduced and implementations details are discussed. Finally, section 3 is devoted to the presentation of test results comparing those implementations.

2 The K shortest loopless paths problem

Let $(\mathcal{N}, \mathcal{A})$ be a network with n nodes and m arcs, where any $(i, j) \in \mathcal{A}$ is assigned with the value $c_{ij} \in \mathbb{R}$, that denotes the cost, or distance, of (i, j).

A path p from $i \in \mathcal{N}$ to $j \in \mathcal{N}$ in $(\mathcal{N}, \mathcal{A})$ is a sequence of the form $p = \langle i = v_1, v_2, \ldots, j = v_{\ell(p)} \rangle$, where $(v_k, v_{k+1}) \in \mathcal{A}$, for any $k \in \{1, \ldots, \ell(p) - 1\}$. Here $\ell(p)$ is called the length of p, that is, its number of nodes, while i and j are called the initial and terminal nodes of path p, respectively. The total cost, or distance, of p is defined by $c(p) = \sum_{(u,v) \in p} c_{uv}$. Given x and y two nodes of p, $\mathrm{sub}_p(x, y)$ represents its subpath from x to y. A path is said to be loopless when it has no repeated nodes.

The set of (loopless) paths from i to j in $(\mathcal{N}, \mathcal{A})$ will be denoted by $\mathcal{P}_{ij}$ $(\bar{\mathcal{P}}_{ij})$, and given an initial, s, and a terminal, t, nodes $(s \neq t)$, then $\mathcal{P}$ $(\bar{\mathcal{P}})$ will be used for $\mathcal{P}_{st}$ $(\bar{\mathcal{P}}_{st})$. The concatenation of $p \in \mathcal{P}_{ij}, q \in \mathcal{P}_{j\ell}$, denoted by $p \diamond q$, is the path from i to ℓ formed by path p followed by q.

Given $K \in \mathbb{N}$, in the K shortest loopless paths problem is intended to compute loopless paths $p_1, \ldots, p_K$ from s to t in $(\mathcal{N}, \mathcal{A})$, by non-decreasing order of the cost, that is, such that $c(p_1) \leq \ldots \leq c(p_K)$ and $c(p_K) \leq c(p)$, for any $p \in \bar{\mathcal{P}} - \{p_1, \ldots, p_K\}$.

2.1 Deviation algorithms for the K shortest loopless paths problem

The algorithms for this problem use a set X to store the loopless paths candidates to p_k, $k = 1, \ldots, K$. This set is initialised with the shortest path, p_1, and when $p_1, \ldots, p_{k-1}$ have been determined p_k is the shortest candidate in X. Once p_k is selected, and deleted, in X its nodes are analysed, in order to generate new candidates with a low cost. As the generated candidates are deviations of p_k (for they have an initial subpath common with p_k after what they split at some node) these algorithms are also known as deviation algorithms, and their variants differ on the deviations computed or on the method used to obtain them. The easiest way to find such a deviation is to delete the arc of p_k that starts at the scanned node, and then take the shortest path from that node to t. However, this procedure can generate paths with loops, therefore it demands the loopless condition to be checked and the selection of an *a priori* unknown number of candidates, until K loopless paths have been determined. Algorithms based on this idea can be found in [9, 10]. In this following we will focus on deviation algorithms that generate only loopless paths, and therefore that scan exactly K loopless paths.

Yen's algorithm The first algorithm developed to rank loopless paths was presented by Yen in 1971 [11]. Assuming the k shortest loopless path has the form $p_k = \langle v_1 = s, \ldots, v_{\ell(p_k)} = t \rangle$, for $k = 1, \ldots, K$, Yen's proposal to obtain new candidates is to partition the set of loopless paths in the following way:

$$\begin{array}{rcl} \bar{\mathcal{P}} - \{p_1\} & = & \bigcup_{i=1}^{\ell(p_1)} \bar{\mathcal{P}}^1(v_i)\,, \\ \bar{\mathcal{P}}^j(v_{d(p_k)}) - \{p_k\} & = & \bigcup_{i=d(p_k)}^{\ell(p_k)} \bar{\mathcal{P}}^k(v_i), \quad k > 1\,, \end{array} \tag{1}$$

where $\bar{\mathcal{P}}^j(v_i)$ denotes the set of the loopless paths, different from $p_1, \ldots, p_j$, that have $\mathrm{sub}_{p_j}(s, v_i)$ as the initial subpath, common with path p_j, for some $1 \leq j < k$. When a p_k is picked up in X the set $\bar{\mathcal{P}}^j(v_{d(p_k)})$ where p_k was determined is considered, which means that it is partitioned by computing the shortest loopless path in each of the subsets in (1). Yen noted that the best deviation from p_k at node v_i is $\mathrm{sub}_{p_k}(s, v_i) \diamond q_i$, where q_i is the shortest path from v_i to t, when the nodes $v_1, \ldots, v_{i-1}$ and the arcs $(v_i, x) \in \{p_1, \ldots, p_k\}$ are removed from the network. The loopless path p_k is called the father of the new candidates determined (known as p_k sons or p_k deviations) and $v_{d(p_k)}$ the deviation node of p_k. Thus, analysing a given p_k consists of modifying $(\mathcal{N},\mathcal{A})$, by deleting some arcs and some nodes, and solving a shortest path problem between a pair of nodes.

Perko's algorithm In 1986 Perko [10] presented implementations of deviation algorithms for ranking loopless paths, in particular of Yen's algorithm. From his work we remark the utilisation of upperbounds on the cost of the candidates to generate, in order to decrease the number of stored deviations, as well as to reduce the number of solved subproblems, and the introduction of a special representation of the list of nodes to be labeled when solving the sequence of shortest path problems resulting from the analysis of some p_k, in order to avoid the initialisations. We will go into further details later on.

Martins & Pascoal's algorithm Noting the similarity of the subproblems that have to be solved in Yen's algorithm when scanning some p_k, in 2003 Martins & Pascoal [8] introduced a variant where the nodes are analysed by a particular order. Instead of deleting arcs and nodes in the network, analysing the p_k nodes from $v_{\ell(p_k)} = t$ to $v_{d(p_k)}$ allows to reoptimise each shortest path by the insertion of new arcs and nodes, and to replace the resolution of shortest path problems by these reoptimisations

Katoh, Ibaraki & Mine's algorithm Another deviation algorithm that uses only loopless paths was introduced in 1982 by Katoh, Ibaraki & Mine [7]. This algorithm is only valid for undirected networks and it uses their characteristics to generate, at most, three deviations for each analysed p_k. The difference between Yen's algorithm and this one is in the partition used to generate new candidates, and therefore in the candidates generated when some p_k is scanned. Let p_j be the father of some loopless path p_k, and:

- v_δ be the deviation node of a son of p_j, previous to $v_{d(p_k)}$ and farther from s,
- v_γ be the deviation node of another son of p_j, closer to s but after $v_{d(p_k)}$.

If $\bar{\mathcal{P}}^j_k(v_\delta, v_\gamma)$ denotes the set of loopless paths of the form $q' = \text{sub}_{p_j}(s, v_\delta) \diamond q \notin \{p_1, \ldots, p_k\}$, where q is a path from v_δ to t that deviates from p_j before v_γ, then,

$$\begin{aligned} \bar{\mathcal{P}} - \{p_1\} &= \bar{\mathcal{P}}^1_2(s,t)\,, \\ \bar{\mathcal{P}}^j_k(v_\delta, v_\gamma) - \{p_k\} &= \bar{\mathcal{P}}^j_{k+1}(v_\delta, v_{d(p_k)}) \cup \bar{\mathcal{P}}^j_{k+1}(v_{d(p_k)}, v_\gamma) \cup \bar{\mathcal{P}}^k_{k+1}(v_{d(p_k)+1}, t), \quad k > 1\,. \end{aligned} \tag{2}$$

The analysis of a p_k consists of determining the shortest loopless paths in the subsets above, namely:

- the shortest path in $\bar{\mathcal{P}}^j_{k+1}(v_\delta, v_{d(p_k)})$, i.e., which deviates from p_j between v_δ and $v_{d(p_k)}$,
- the shortest path in $\bar{\mathcal{P}}^j_{k+1}(v_{d(p_k)}, v_\gamma)$, i.e., which deviates from p_j between $v_{d(p_k)}$ and v_γ,
- and the shortest path in $\bar{\mathcal{P}}^k_{k+1}(v_{d(p_k)+1}, t)$, i.e., which deviates from p_k between $v_{d(p_k)+1}$ and t.

Furthermore, Katoh *et al.* showed that the solution of each of these subproblems can be found by solving two single source shortest path problems, after proper modifications of $(\mathcal{N},\mathcal{A})$. Let $\mathcal{T}_s$ be the tree of shortest paths from s to any node, $\mathcal{T}_t$ be the tree of shortest paths from any node to t, and let $\mathcal{T}_s(i)$ be the path from s to $i \in \mathcal{N}$ in $\mathcal{T}_s$ and $\xi_s(i)$ be the index of the node where $\mathcal{T}_s(i)$ and $\mathcal{T}_s(t)$ split (analogously for $\mathcal{T}_t(i)$ and $\xi_t(i)$). Given $p^* = \langle s = v_1, \ldots, v_{\ell(p^*)} = t\rangle = \mathcal{T}_s(t) = \mathcal{T}_t(s)$ the shortest path that deviates from p^* before a node v_α is of type 1 or 2:

1. $\mathcal{T}_s(i) \diamond \mathcal{T}_t(i)$, with $i \in \mathcal{N}$ such that $\xi_s(i) < \alpha$,
2. $\mathcal{T}_s(i) \diamond \langle i,j\rangle \diamond \mathcal{T}_t(j)$, with $(i,j) \in \mathcal{A} - (\mathcal{T}_s \cup \mathcal{T}_t)$ and $\xi_s(i) < \alpha$.

In [5] Hadjiconstantinou & Christofides presented details about an implementation of Katoh *et al.*'s algorithm.

Hybrid algorithm Deviation algorithms similar to Yen's algorithm can be applied to the unconstrained ranking paths problem, although in that case it's easier to compute a deviation. In fact, the best path from s to t that deviates from p_k at v_i has the form: $\text{sub}_{p_k}(s, v_i) \diamond (v_i, j) \diamond \mathcal{T}_t(j)$, such that (v_i, j) doesn't belong to any of the candidates computed so far. The generation of new candidates by this method is more efficient, as each shortest path problem is replaced by the selection of an arc (which, as described in subsection 2.3, can be made in constant time), however it may produce paths with loops when $\text{sub}_{p_k}(s, v_i)$ and $\mathcal{T}_t(j)$ share some nodes. The hybrid algorithm uses this candidate generation procedure whenever it returns a loopless path, and changes into the Yen's algorithm routine otherwise.

2.2 Implementation considerations

The main operations, both in Yen's and Katoh *et al.*'s algorithms, are changing the original network, by deleting nodes and arcs, and solving subproblems, consisting of finding shortest paths between a pair of nodes. When implementing these algorithms other issues have to be taken into account, concerning the data structures used for representing the network and for storing the generated loopless path candidates.

Storage of paths The nodes of each generated candidate depend on its father, and on the conditions of the network where it was obtained, namely the deleted nodes and arcs. Once this information is known, also the complete loopless path can be found by means of solving a shortest path problem. Nevertheless, to avoid the replication of these problems, as well as to get direct access to the nodes of a given candidate, it is useful to store the shortest path that is the solution of each subproblem solved whenever some p_k node is scanned. For this purpose a trie structure can be used. A trie works like a tree but takes advantage of the fact that many of the stored loopless paths start with the same sequence of nodes. Thus, a loopless path p is represented by: $\text{sub}_p(v_{d(p)}, t)$, $v_{d(p)}$, and a link to its father loopless path. Its nodes can be retrieved by scanning the stored final part of every antecessor of p, from its father until p_1. See [3] for details on this data structure. On the other hand, when implementing the hybrid algorithm the data structure has to be hybrid as well, taking into account that the paths generated by Yen's procedure have to be represented with the trie structure, and that it's enough to represent the paths resulting from the choice of one arc by the arc deleted when obtaining it.

Still related with this point, the structure used to access the candidate loopless paths has to be established. Unlike in Katoh *et al.*'s algorithm, where at most $3K$ candidates are computed, in Yen's algorithm the number of generated candidates depends on the number of nodes that have to be scanned for every path p_k, which yields to a bound of Kn candidates. Assuming K to be known in advance we can opt, in either case, for:

- storing all the computed candidates,
- maintaining at most the K shortest ones in every step of the algorithm (this implies to replace the stored candidate with the worst cost whenever a better one is found, after K loopless paths are known),
- or, an intermediate approach, storing every candidate until there are K and after that keeping only those with a cost lower than the maximum cost of those K candidates.

In each iteration of those algorithms the shortest candidate has to be selected in the set X. Thus, it is recommended to represent this set as an ordered data structure, for instance as a heap or with the Dial variant for priority queues, with an array of buckets with a cyclically moving array index (see [1]).

Network representation The structure used to represent the network depends on the method used to find shortest paths. Two options are usually considered: one is to analyse the arcs emerging from any node in the network, and therefore representing it in the forward star form, while the other is to analyse the arcs that end in any node in the network, therefore using the backward star form. See [4] for details on these data representations.

Network modifications As mentioned above the subproblems to be solved appear in subnetworks of $(\mathcal{N}, \mathcal{A})$, and the necessary modifications are nodes and arcs deletions. The easiest, and the most efficient, way to implement these changes is to mark the non-available nodes and arcs, for instance using negative values whenever a node/arc is deleted, or else using two extra arrays.

Shortest path algorithms As already mentioned, the key for the algorithms by Yen and Katoh *et al.* is the resolution of several shortest path problems, between a given pair of nodes in the first case, and with a single source node in the second. In straightforward implementations of these algorithms any method for these problems can be used.

Still related with this point, but apart from selecting the routine for the subproblems, we drive our attention to two proposals. On the one hand the data structure used by Perko [10] for solving the successive shortest path problems when scanning some p_k, that aims to decrease the number of initialisations in such a sequence of resolutions, and on the other the variant of Martins & Pascoal [8], that takes advantage of the similarity of the subproblems to be solved when scanning each p_k, and replace them by the reoptimisation of shortest paths for each node of p_k that is analysed.

Finally, concerning both the number of candidates that are stored and the number of subproblems that are solved, we recall that a deviation of p_k at v_i is $\text{sub}_{p_k}(s, v_i) \diamond (v_i, j) \diamond \mathcal{T}_t(j)$, for a certain arc (v_i, j), and the minimum cost of a path of this form is a lowerbound on the cost of the deviation to be determined when scanning v_i. Then, the resolution of the subproblem can be skipped whenever that cost exceeds the maximum cost of a stored candidate (assuming at least K loopless paths are known). It should be noted that this last bound is related with the algorithms for the K shortest loopless paths problem that are allowed to generate paths containing loops.

2.3 Theoretical complexity bounds

As expected, in terms of the number of operations performed and the space of memory used, the implementations described fit into two groups, the Yen's like algorithms and the algorithm by Katoh *et al.*. We shall also see that the hybrid method can be considered as an "outsider".

Let us first focus on the number of operations performed by the original version of Yen's algorithm. Among those operations one should remark the shortest path computation, of $\mathcal{O}(c(n))$, and later the selection and analysis of K loopless paths. Considering that at most K loopless paths are maintained in the candidates list, the insertion of a new candidate can be done in $\mathcal{O}(\log K)$, while the selection of the best one can be done in $\mathcal{O}(1)$. For each p_k that is listed at most n nodes have to be scanned, and therefore, after deleting some nodes and arcs of the original network, n shortest path problems have to be solved and their solutions have to be stored. Thus, the worst-case complexity order for this algorithm is $\mathcal{O}(c(n) + K(nc(n) + n + \log K))$, or simply $\mathcal{O}(Kc(n))$ if we omit the term $\mathcal{O}(\log K)$ (which is, in general, dominated by the others). Two factors are expected to strongly influence the effective number of operations: the method that is used to solve the shortest path problem, as well as the length of the loopless paths generated, that is, their number of nodes.

When implementing Perko's version of this algorithm only some initializations are avoided and then Yen's algorithm complexity order remains valid. The situation is slightly different when concerning Martins and Pascoal implementation. In fact its worst-case occurs when inserting a new node into a path is as hard as solving a new point-to-point shortest path problem, and this leads to the worst-case bound for Yen's algorithm. In an "optimistic" case labeling the nodes corresponds to computing a single shortest path, and then the number of operations is improved to $o(Kc(n))$.

As mentioned earlier, the hybrid algorithm demands the choice of an arc when every node of a path p_k is scanned. In order to make immediate the choice of the arc (v_i, j) when scanning node v_i, each arc cost c_{xy} can be replaced by the reduced cost $\bar{c}_{xy} = c_{xy} - c(\mathcal{T}_t(x)) + c(\mathcal{T}_t(y))$, for any $(x, y) \in \mathcal{A}$. As $c_{xy} \geq 0$ for any $(x, y) \in \mathcal{A}$, and $c_{xy} = 0$ for $(x, y) \in \mathcal{T}_t$, the arc to select is the one with the minimum reduced cost that starts at v_i. For applying this procedure a pre-processing phase is necessary, which should consist of: determining $\mathcal{T}_t$, replacing the costs by the reduced costs, and sorting $\mathcal{A}$, in order to represent $(\mathcal{N}, \mathcal{A})$ in the sorted forward star form – see [4].

The worst-case number of operations performed with the hybrid method is identical to Yen's, $\mathcal{O}(Knc(n))$, while in an optimistic case all deviations are loopless, so $o(Kn \log K + c(n) + m \log n)$, or simply $o(Kn + c(n) + m \log n)$.

The method by Katoh *et al.* has the advantage of limiting to 3 the number of candidates generated with the scan of each p_k, therefore the worst-case number of operations is $\mathcal{O}(c(n) + K(d(n) + \log K))$, where $d(n)$ stands for the number of operations for solving the single source shortest path problem, or simply $\mathcal{O}(Kd(n))$.

In terms of the necessary memory space it should be noted that in Yen's algorithm the network, as well as the trie structure containing the candidate loopless paths, have to be stored. As each loopless path has, at most, n nodes the worst-case space bound for this method is $\mathcal{O}(m+n+Kn^2)$, that is, $\mathcal{O}(Kn^2)$. Once again the length of the candidates generated determines, not only, the size of the trie, but also the number of candidates generated itself. The same bound is valid for the variants by Perko and by Martins & Pascoal, as their implementations compute exactly the same candidates, and use the same structure to store them.

The situation is different for algorithms by Katoh *et al.* and the hybrid. In the first case up to 3 new candidates are generated for any p_k, therefore the worst-case memory space complexity is $\mathcal{O}(m+n+Kn)$, or $\mathcal{O}(m+Kn)$. As for the number of operations, the hybrid algorithm shares the Yen's algorithm space complexity order in the worst-case, $\mathcal{O}(Kn^2)$. However, as a candidate is obtained without solving a shortest path problem can be identified only by the deleted arc, in an optimistic case the complexity coincides with the Katoh *et al.*'s algorithm one, $o(m+Kn)$.

3 Implementations and computational tests

In this section test results on the core benchmarks provided are reported comparing the following implementations:

- `Y`: A straightforward implementation of Yen's algorithm, where nodes in p_k are analysed by the usual order (from $v_{d(p_k)}$ to t). A label correcting algorithm is used to solve the single source shortest path problems. The list of temporary labels is manipulated as a FIFO.
- `YD`: Similar to `Y` but using a label setting algorithm to solve the single source shortest path problems. The list of temporary labels is implemented using the Dial priority queue.
- `YDI`: Similar to `YD` but now point to point shortest path problems are solved. With this goal Dijkstra's algorithm is interrupted as soon as the terminal node has a permanent label.
- `P`: An implementation of the variant proposed by Perko, using its list of labeled nodes.
- `MP`: An implementation of the variant by Martins & Pascoal. Labeling of nodes uses a FIFO list.
- `HY`: A hybrid version of Yen's algorithm that first looks for a deviation with the form $\text{sub}_{p_k}(s, v_i) \diamond (v_i, j) \diamond \mathcal{T}_t(j)$, and uses it if it contains no loops. Otherwise a shortest path problem is solved, as in the common Yen's algorithm with a label setting technique halted as soon as the terminal node has a permanent label.
- `KIM`: An implementation of Katoh, Ibaraki & Mine's algorithm. A label correcting algorithm using a FIFO list is applied to solve the single source shortest path problems.

All the codes were written in C language and compiled with the GNU compiler and optimisation option `-O3`. As the experiments made with the `YD` version showed a performance worse than the `Y` code, the results for that variant will not be presented.

In these implementations a trie structure was used to represent the candidates. In the hybrid algorithm this structure is maintained together with the representation of loopless path deviations with a pointer to its father and the deleted arc. The set of candidates was represented following the Dial's variant of priority queues. After K candidates have been stored only candidates better than those ones were stored. Codes `Y`, `YD` and `YDI` demanded the network to be represented in the forward star form, while for the remaining ones the backward star form was used as well.

3.1 Random graphs

The tests over the `Random-n` family provided for the workshop were carried out on a Pentium 4 with a 2.4 GHz processor, 1 MB of cache and 1 Gb of RAM, running over SUSE Linux 9.3, With this first set of tests we intended to compare the implementations behaviour, namely in terms of the shortest path computation, and of the $K = 100$ shortest loopless path ranking. The results presented in the following are average values obtained for each graph, when ranking 100 loopless paths between 1000 source-destination pairs of nodes.

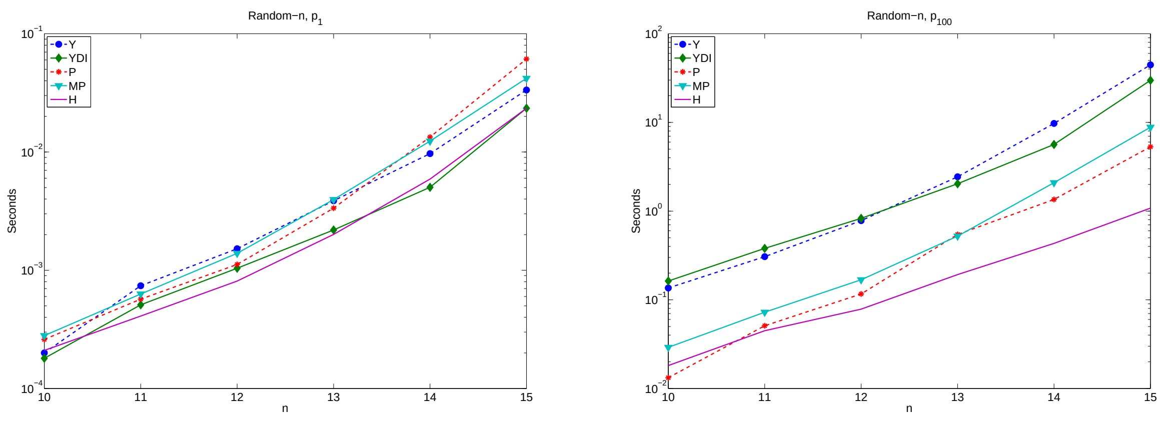

Figure 1: Average CPU times (in log scale) on the `Random-n` class

Figure 1 presents the average CPU times of each implementation to compute the shortest path, p_1, and then to find paths $p_1, \dots, p_{100}$, while table 1 presents the average total CPU times. When the total execution time of an implementation was too high (approximately 1 minute) that implementation was excluded from the following tests.

Code\n	10	11	12	13	14	15	16
`Y`	0.13606	0.30722	0.78312	2.44271	9.75297	44.65218	—
`YDI`	0.16301	0.38020	0.82049	2.03801	9.74852	37.09130	78.42155
`P`	0.01449	0.05254	0.11957	0.55174	1.37516	5.41105	21.21274
`MP`	0.03040	0.07325	0.16993	0.52642	2.11223	8.90611	30.11306
`HY`	0.01936	0.04626	0.08135	0.19749	0.44646	1.14174	3.70977

Code\n	16	17	18	19	20	21
`HY`	3.70977	7.16976	15.25910	22.11902	58.28594	166.71764

Table 1: Average total CPU times (in seconds) for $K = 100$ on the `Random-n` class

The times to determine the shortest path were very close for all the codes, although those that use a label setting algorithm have ran, in general, slightly faster than the ones using the label correcting version.

In what concerns the loopless path ranking we can distinguish two sets of methods: the straightforward implementations of Yen's algorithm, `Y` and `YDI`, and the codes `P`, `MP`, `H`. It is clear the increase of the running times with the value of n that defines the number of nodes in the network (2^n). Interrupting the shortest path routine when the destination node is reached seems to be advantageous, except for the smallest problems. `P` outperformed `MP`, although its running times appeared to be more sensitive to the increase of n than those obtained by `MP`. Nevertheless, `HY` was the most efficient code in almost any problem.

These conclusions were reinforced by the second set of tests, aimed to evaluate the efficiency of the codes with the growth of K, the number of ranked loopless paths. This time $K = 1000$ loopless paths were computed and only the graphs with $n = 10, 11, 12, 13, 14, 15$ were used. The partial running times obtained are shown in figure 2 and appear to increase linearly with K for every code. For these higher values of K the implementation `Y` ran faster than `YDI` in small networks ($n = 10$), and for $n \geq 13$, the inverse situation was observed. Unlike what happened in the first set of tests, now `P` was slower than `MP`. Still, the hybrid version was the code with the most efficient and the most stable performance, being able to rank 1000 paths in 4 seconds.

Recalling that the codes `Y`, `YDI`, `P` and `MP` have identical space requirements, figure 3 and table 2 present average results on the number of loopless path nodes stored when ranking $K = 1000$ loopless paths in the classes `Random-n`, `Long-n` and `Square-n` involving the implementations `MP` and `HY`. The partial results (in

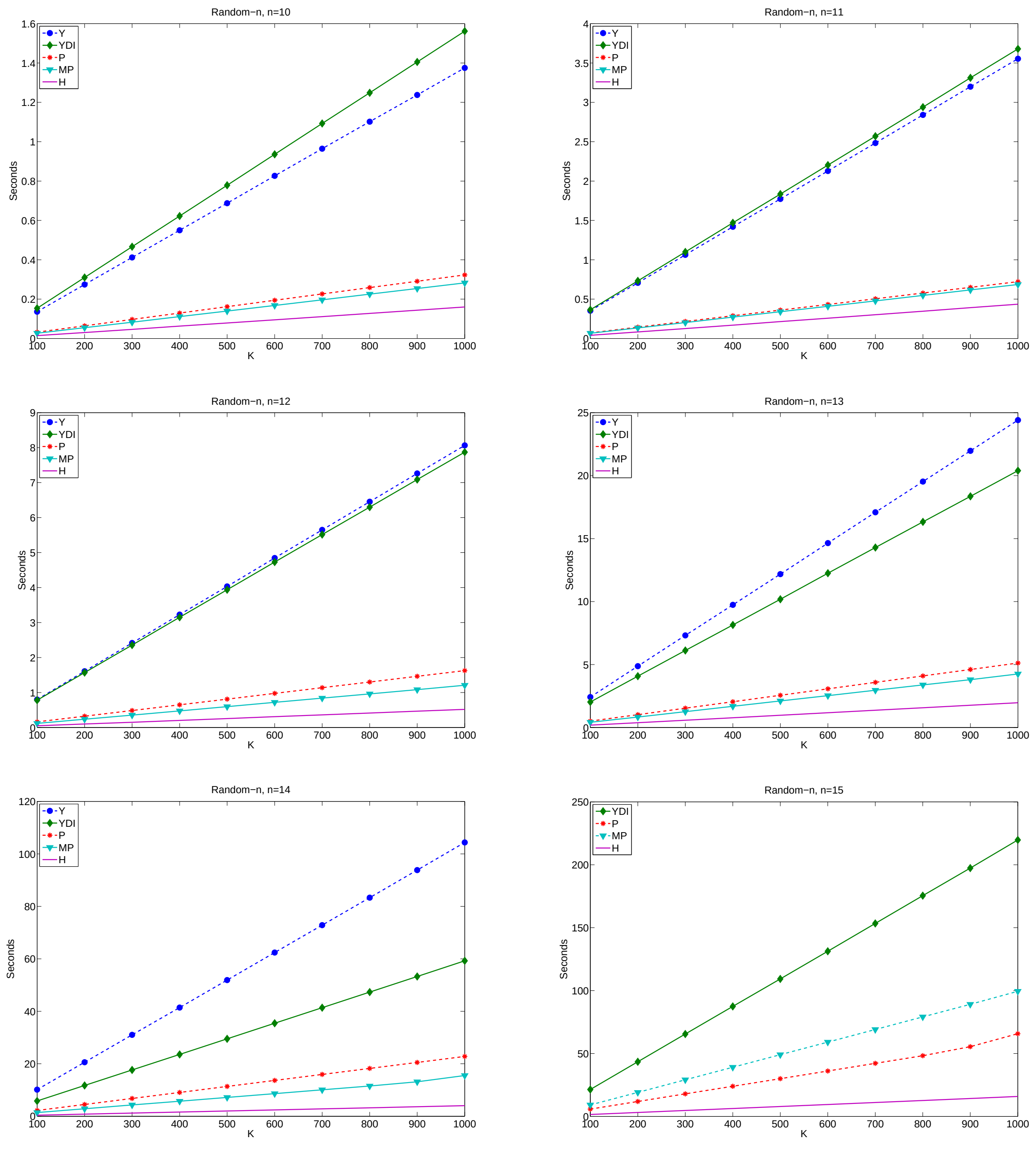

Figure 2: Average partial CPU times on the `Random-n` class

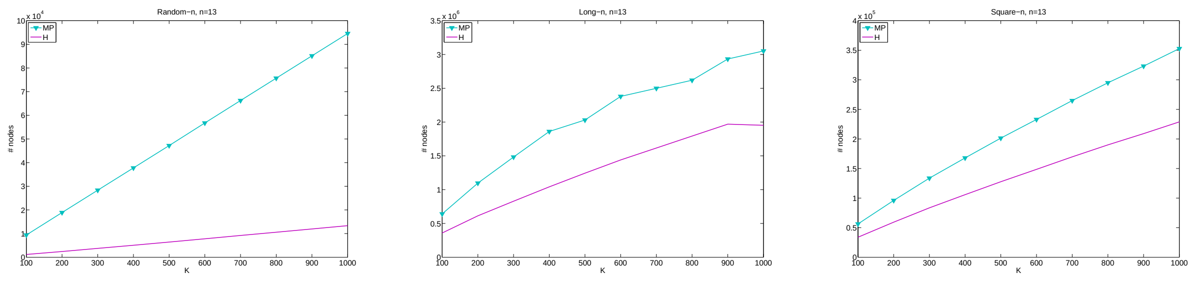

Figure 3: Average partial number of loopless path nodes stored

n	10	11	12	13	14	15	16
`MP` on `Random-n`	57912	71239	80287	94586	111906	123934	142163
`HY` on `Random-n`	12799	17595	11667	13350	15810	12125	14100
`MP` on `Long-n`	120308	312380	996225	3052028	—	—	—
`HY` on `Long-n`	91787	219387	625630	1952148	—	—	—
`MP` on `Square-n`	74634	134770	234473	352835	791557	1154152	—
`HY` on `Square-n`	57703	99179	156961	228919	484212	667407	—

Table 2: Average total number of loopless path nodes stored

figure 3) for different values of n were similar, in relative terms, therefore only the $n = 13$ case is depicted. The size of the trie structure used, that is, the number of candidate nodes stored, represented in these graphics shown a linear dependence with K. The difference of that size for the two codes was evident and increased with K, specially for the `Random-n` graphs.

n	10	11	12	13	14	15	16
# of nodes scanned	6944	7682	8237	9002	9875	10365	10939
# of shortest path computations	1606	1980	1282	1450	1553	1207	1632
# of shortest paths stored	1408	1748	1110	1174	1320	907	1284

Table 3: Average total number of subproblems solved by `HY` for $K = 1000$ on the `Random-n` class

Table 3 concerns only the `Random-n` class and allows to get more complete information about the subproblems that `HY` solved, and how it improved the other Yen's like variants. It should be remarked that the first line, concerning the number of nodes that were scanned during the ranking, is also valid for the remaining Yen's like versions. The main difference is that in `HY` some of the scans consist only is selecting one arc (instead of implying a shortest path computation). The number of times that a shortest path problem had to be solved is given in the second line, while the last one simply shows the cases when that problem had a solution.

From the number of nodes scanned in table 3 and the total number of nodes stored in table 2 one can conclude that the average length of the listed loopless paths on these graphs was between 8 and 13 (increasing with the size of the graph). As we shall see later, the average length is bigger for the other classes, between 10 and 25 nodes in the `Square-n` class, and from 12 to 56 in the `Long-n` class.

Finally, one can conclude that in the instances tested the majority of the candidates have been generated only by the arc selection procedure, thus decreasing the number of shortest path computations, which explains the best performance of code `HY`.

Intending to study the behaviour of `HY` for bigger problems, a few more tests were made considering higher values of K. The average results when finding $K = 10000$ loopless paths in `Square-n.15.0.gr` for 50 queries are summarised in figure 4.

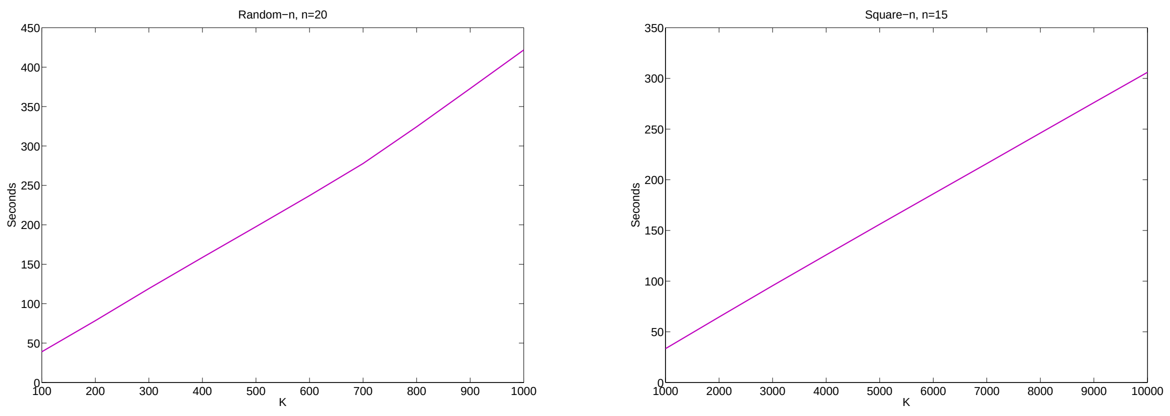

Figure 4: Average partial CPU times of `HY` on big dimension problems

3.2 Grid graphs

The tests over the `Long-n` and `Square-n` families were carried out on the same machine and the sets of experiments were also similar to the ones mentioned in the last section for the `Random-n` graphs. We begin by showing average running times to find the shortest path and to rank $K = 100$ loopless paths on the `Long-n` class of graphs, in figure 5 and table 4.

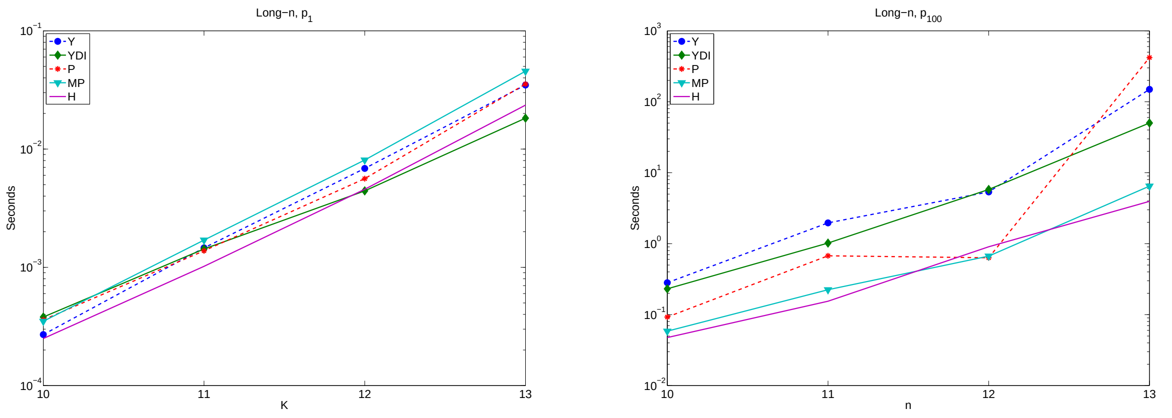

Figure 5: Average CPU times (in log scale) on the `Long-n` class

It was harder to solve the same problem on these benchmarks, for any of the codes, therefore the results presented concern only smaller values of n. This may be due to the fact that the loopless paths in these type of networks are longer, which, as mentioned, makes the codes behaviour closer to the worst-case complexity order. In fact, as mentioned in the previous section, the average length of the listed loopless paths varied between 12 and 56 nodes – see table 5 – which is greater than the length observed for the `Random-n` class.

Code `Y` continued presenting worse results than `YDI`, but the relative behaviour of the other versions depended on the classes of graphs. The average results now obtained with `P` were clearly worse than the ones obtained with `MP`. This was particularly noted for the `Long-n` class of benchmarks, where `P` had a highly unstable behaviour. In fact, some of the problems were solved very quickly (even faster than with the `HY` code), but many others took much longer (even longer than with the `Y` code). For instance, in the 1000 queries ran for the graph `Long-n.13.0.gr` the range of the total CPU times of `P` for finding $p_1, \ldots, p_{100}$

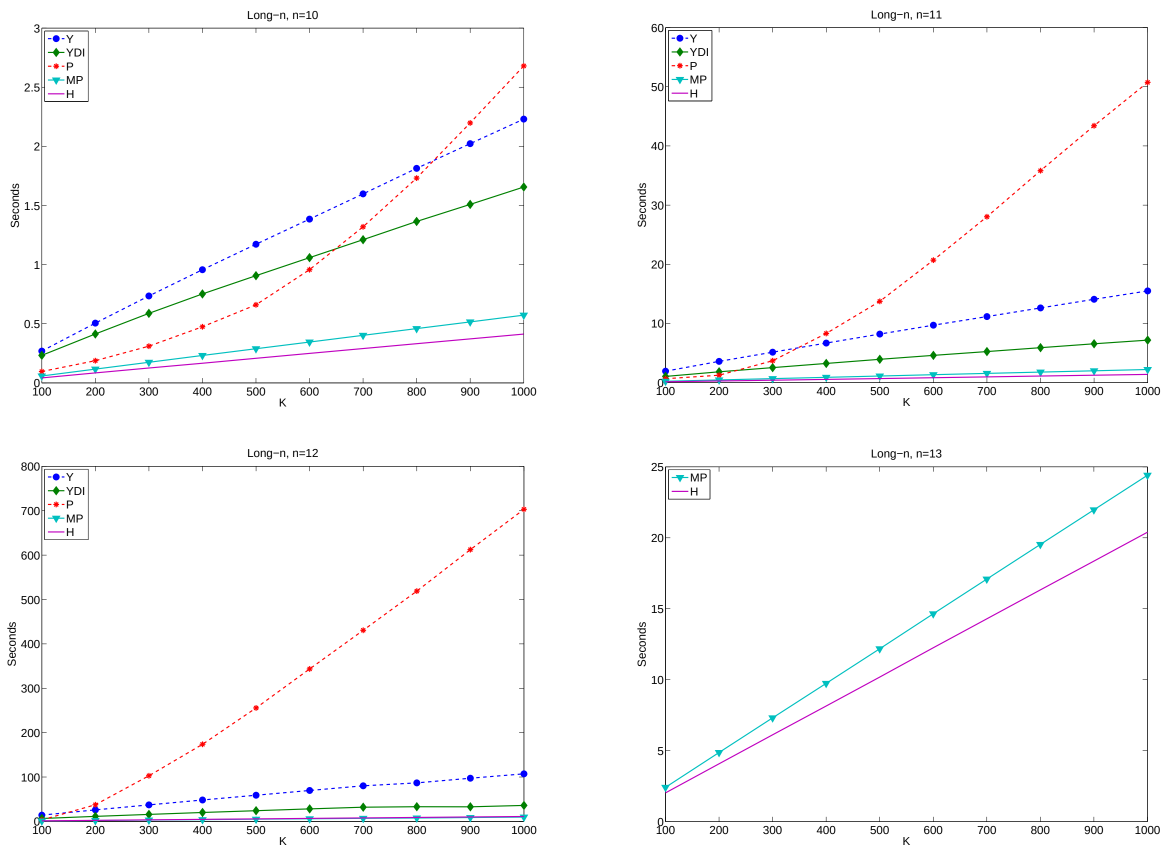

Figure 6: Average partial CPU times on the `Long-n` class

Code\n	10	11	12	13	14	15
`Y`	0.28219	1.96509	5.37230	149.74375	—	—
`YDI`	0.23153	1.02151	5.81163	41.93847	—	—
`P`	0.09284	0.67735	0.63993	419.60120	—	—
`MP`	0.05874	0.22719	0.67731	6.51990	158.75268	—
`HY`	0.04779	0.15646	0.91010	3.97553	21.95746	50.19182

Table 4: Average total CPU times (in seconds) for $K = 100$ on the `Long-n` class

n	10	11	12	13	14
# of nodes scanned	9974	15769	27592	54784	183400
# of shortest path computations	8028	11370	17756	32219	106777
# of shortest paths stored	7804	11175	17399	31700	106082

Table 5: Average total number of subproblems solved by `HY` for $K = 1000$ on the `Long-n` class

varied between 0.025 and 10088.840 seconds.

Table 5 also shows that, unlike what happened in the `Random-n` class, now most of the candidates generated with code `HY` were obtained by means of a shortest path computation, which might explain the worst performance of this code.

Code\n	10	11	12	13	14	15	16
`Y`	0.18252	0.70128	2.28784	10.31545	72.59006	—	—
`YDI`	0.14302	0.43074	1.23832	4.44411	18.56675	81.36826	—
`P`	0.05430	0.24050	0.67023	2.82218	15.46259	229.43365	—
`MP`	0.04108	0.12028	0.32425	1.10768	5.60908	31.75998	119.05751
`HY`	0.03415	0.07844	0.17907	0.46606	1.18535	3.58459	11.76380

Code\n	17	18	19
`HY`	33.88140	104.98517	251.74028

Table 6: Average total CPU times (in seconds) for $K = 100$ on the `Square-n` class

Instances in the `Square-n` class were easier to solve than for class `Long-n`, but still analogous remarks are due concerning the implementations performance on these benchmarks. The average running times to compute p_1 and $p_1, \dots, p_{100}$ on those graphs are presented in figure 7 and table 6, while figure 8 shows the partial times when ranking 1000 loopless paths.

As observed for the `Long-n` graphs, the behaviour of `P` was very unstable on `Square-n` graphs, although with less dramatic results. For instance, in graph `Square-n.14.0.gr` the CPU times varied between 2.348 and 16.565 for code `MP`, and between 0.167 and 8.832 for code `HY`, while for `P` the minimum was 0.143 and the maximum 79.995 seconds. Now the average length of the loopless paths ranked varied between 10 and 25 nodes – see table 7 – and still in about 70% of the cases code `HY` had to apply a shortest path routine in order to obtain a new candidate.

3.3 Road graphs

The tests on real-world instances were carried out on a Pentium 4 with a 3 GHz processor, 2 MB of cache and 1 Gb of RAM, running over SUSE Linux 9.3.

This was the only class where `KIM` code was tested. One of the reasons is that the graphs in this family are undirected, and the other one is that the performance of this code, in terms of running times was clearly worse than the best straightforward implementation of Yen's algorithm, `YDI`. Table 8 and figure 9 show average results of the total and partial running times, respectively, over 50 queries for the smallest of these networks, `USA-road-d.NY.gr` and `USA-road-t.NY.gr`.

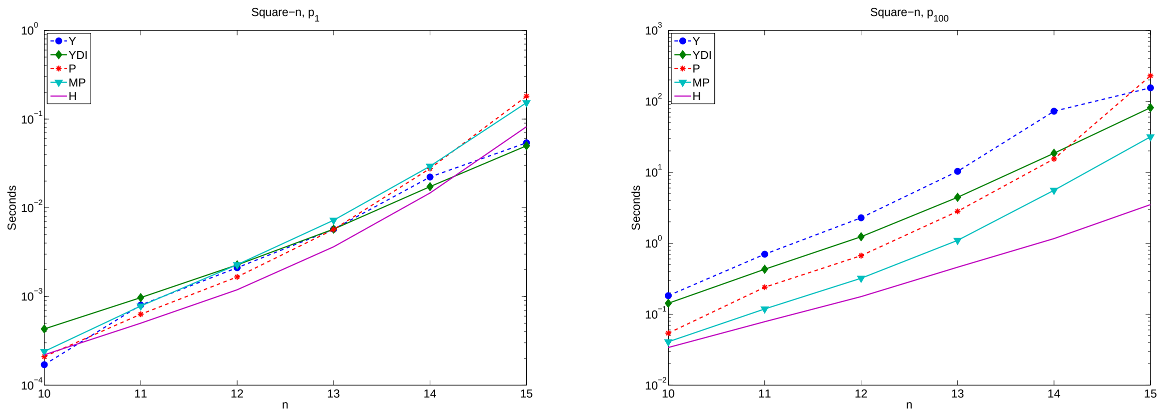

Figure 7: Average CPU times (in log scale) on the `Square-n` class

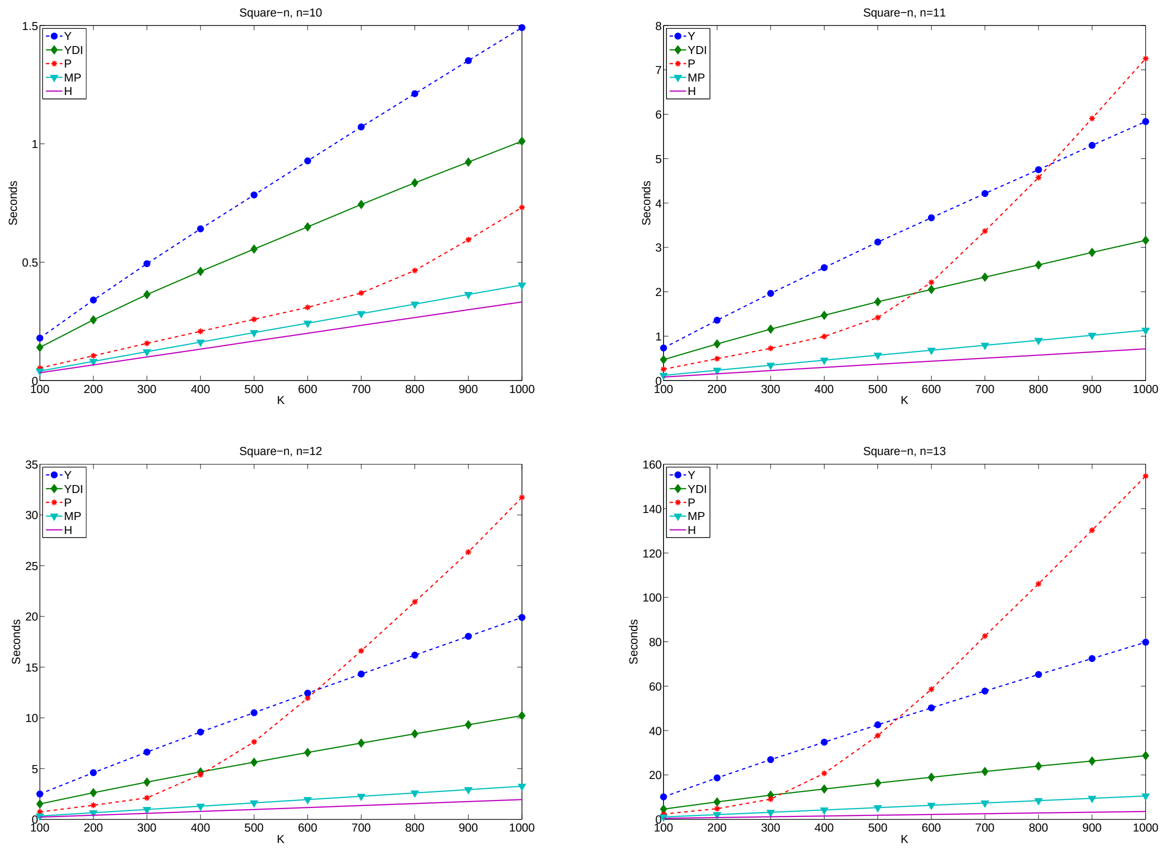

Figure 8: Average partial CPU times on the `Square-n` class

n	10	11	12	13	14	15
# of nodes scanned	7732	11147	14367	18883	27574	45842
# of shortest path computations	6221	8419	9909	12641	17552	28974
# of shortest paths stored	6021	8270	9735	12549	17388	28873

Table 7: Average total number of subproblems solved by `HY` for $K = 1000$ on the `Square-n` class

	`NY-d`	`NY-t`
`YDI`	0.01510	0.06438
`KIM`	4.09345	2.26890

(a) p_1 time

	`NY-d`	`NY-t`
`YDI`	4.83838	22.84906
`KIM`	183.52124	110.13991

(b) Total time

	`NY-d`	`NY-t`
`YDI`	11162	109413
`KIM`	4426	3643

(c) Total # of paths computed

Table 8: Average results with $K = 10$ on the `USA-road` classes

Besides the fact that `KIM` had a clear advantage over `YDI` in terms of memory requirements, expressed by the number of candidate loopless paths it generates – table 8.(c) –, that performance could only be achieved by means of solving problems harder than the point-to-point shortest path problem, the single-source shortest path problem, and using a demanding structure to keep the network conditions information. As table 8 and figure 9 show, this resulted in running times much worse than those presented by `YDI`.

As the networks in this family have a high number of nodes only the code `HY` and some of the smallest networks were considered for this set of tests. Table 9 and figure 10 summarise the average results attained when ranking $K = 100$ loopless paths on those networks for 50 source-destination pairs of nodes.

	`NY`	`BAY`	`COL`	`FLA`
d	2.75922	4.47386	8.96694	10.09979
t	1.04658	3.78218	12.27720	32.33331

(a) p_1

	`NY`	`BAY`	`COL`	`FLA`
d	28.29541	93.10358	472.97441	301.72003
t	38.52300	151.06403	372.77004	675.13664

(b) Total

Table 9: Average CPU times (in seconds) for code `HY` with $K = 100$ on the `USA-road` classes

The instances that consider the arc distance as the cost were significantly harder to solve than those with travel time costs, except for the benchmark `USA-road-{d,t}.COL.gr`, even when this was not the case for computing only the shortest path. Besides the problems dimension `HY` was able to list 100 loopless paths in approximately 2,5 minutes on a network with 321270 nodes.

4 Conclusions

The ranking of loopless paths by non-decreasing order of cost is being studied since 1971. The methods proposed in the literature to solve it can be seen as deviation algorithms and those that strictly compute loopless paths can be grouped into two classes: Yen's algorithms (for general networks) and Katoh, Ibaraki & Mine's algorithm (for undirected networks).

Although Katoh *et al.*'s approach saves memory space, it was too slow in the tests that were performed, when comparing to any of the Yen's methods.

Many implementations of Yen's algorithm can be made. We have focused our attention over two straightforward implementations, one using a label correcting algorithm to compute shortest paths and another using a label setting algorithm, interrupted when the destination node is has a permanent label, instead, as well as over a proposal by Perko, that intends to reduce the number of initializations along the algorithm, and a proposal by Martins & Pascoal, that intends to reduce the number of point-to-point shortest path problems that have to be solved. In the experiments ran the label setting version was faster than the label correcting version, except for the smallest instances, yet they were both outperformed by the variants of Perko and Martins & Pascoal. Perko's method has shown excellent results for some of the instances, but very bad results for others, while Martins & Pascoal's seemed to be more robust.

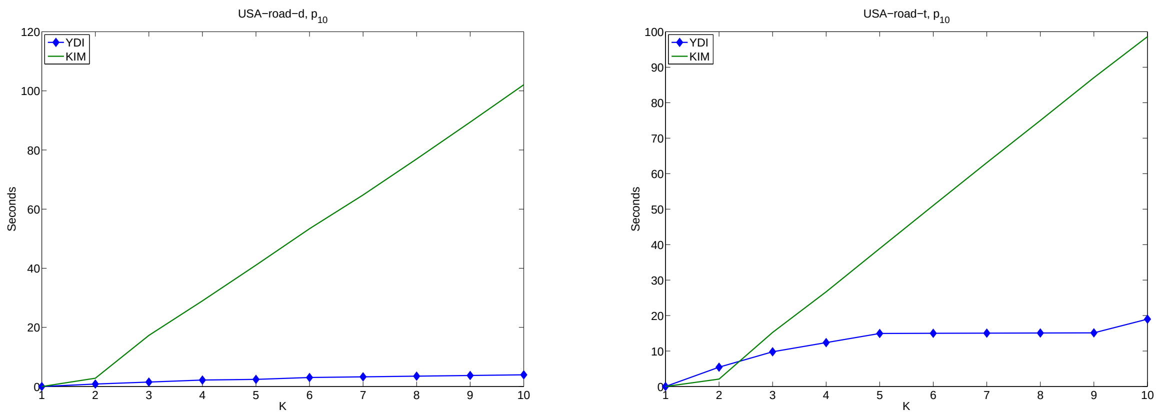

Figure 9: Partial CPU times on the `USA-road` classes

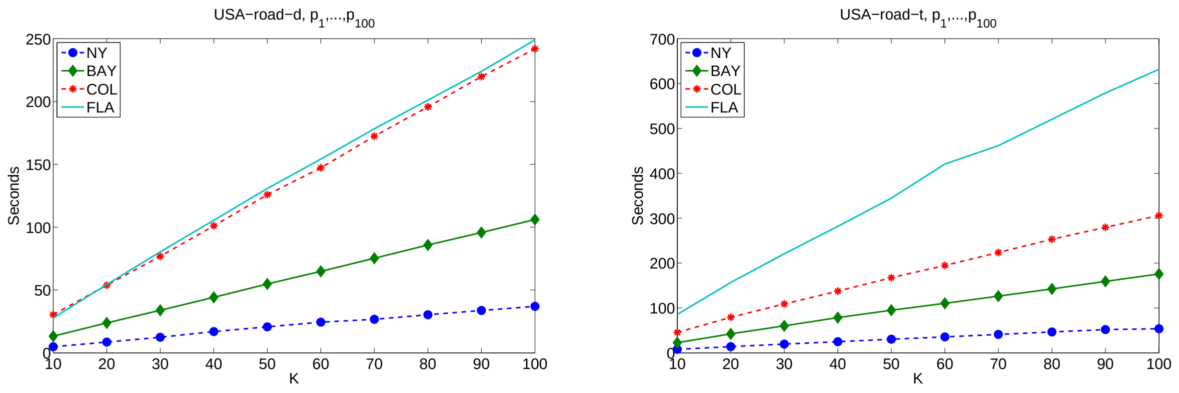

Figure 10: Partial CPU times of code `HY` on the `USA-road` classes

A new method to solve this problem was introduced, that only finds loopless paths but combines Yen's algorithm and deviation algorithms for unconstrained paths ranking. For this set of test this new approach has shown to be more efficient than any of the others. In the random instances provides with 2^15 nodes it was able to rank 1000 loopless paths in approximately 5 seconds, and in the grid instances the same time was about 3 seconds (in square grids) and 20 seconds (in long grids).

References

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows : Theory, Algorithms and Applications.* Prentice Hall, Englewood Cliffs, NJ, 1993.

[2] B. V. Cherkassky, A. V. Goldberg, and T. Radzik. Shorthest paths algorithms: Theory and experimental evaluation. *Mathematical Programming*, 73:129–196, 1996.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* The MIT Press, Cambridge, MA, 2001.

[4] R. Dial, G. Glover, D. Karney, and D. Klingman. A computational analysis of alternative algorithms and labelling techniques for finding shortest path trees. *Networks*, 9:215–348, 1979.

[5] E. Hadjiconstantinou and N. Christofides. An efficient implementation of an algorithm for finding K shortest simple paths. *Networks*, 34(2):88–101, 1999.

[6] W. Hoffman and R. Pavley. A method for the solution of the N th best path problem. *Journal of the Association for Computing Machinery*, 6(4):506–514, 1959.

[7] N. Katoh, T. Ibaraki, and H. Mine. An efficient algorithm for K shortest simple paths. *Networks*, 12:411–427, 1982.

[8] E. Q. V. Martins and M. M. B. Pascoal. A new implementation of Yen's ranking loopless paths algorithm. *4OR – Quarterly Journal of the Belgian, French and Italian Operations Research Societies*, 1(2):121–134, 2003.

[9] E. Q. V. Martins, M. M. B. Pascoal, and J. L. E. Santos. Deviation algorithms for ranking shortest paths. *The International Journal of Foundations of Computer Science*, 10(3):247–263, 1999. `(http://www.mat.uc.pt/~marta/Publicacoes/deviation.ps.gz).`

[10] A. Perko. Implementation of algorithms for K shortest loopless paths. *Networks*, 16:149–160, 1986.

[11] J. Y. Yen. Finding the K shortest loopless paths in a network. *Management Science*, 17:712–716, 1971.