Report

I. Introduction

Enumerating the solutions of some optimization problem can be quite useful when the best solution is irrelevant due to some reason. In these situation, we might want to use the second-best solution, and if that's not possible either, to use the third-best solution and so on. For example, assume that we want to get from a point A to another point B, but that the shortest path from A to B is blocked, or jammed. We would like to find an alternative path to avoid those problems, which is not much longer than the optimal path.

Another use of enumeration is to find a group of high quality solutions. For example, when querying a search engine or a data base, or even when creating a list of recommendations.

There are algorithms who can efficiently enumerate the top-k solutions of a problem, however, these solutions tend to be quite similar to each other, making them sometimes useless. Assume for instance, that we use one of those algorithms to find a replacement for the blocked path from the previous example. If all the alternative paths that were found agree on many edges, it's likely that they would not help replacing the blocked path, since they might contain the same problematic section that made the first path unattractive.

To handle this issue, we need to consider diversity when enumerating the top-k solutions. A diverse set of solutions, even if a little more expensive according to the original cost metric of the problem, can have more "value" and be more useful than a set of cheap, but almost identical solutions.

In this work, we'll describe a variation of a traditional enumeration algorithm (Lawler's Algorithm [1]) that takes diversity into consideration. The modified algorithm penalizes solutions according to their similarity to what was already enumerated, making it "harder" for almost-identical solutions to appear in the output. As for it's complexity, the new version of the algorithm guarantee incremental polynomial time.

In practice, as can be seen in section VIII, the modified algorithm achieves a considerable improvement in diversity over the original algorithm, with a relatively small affect on the average solution cost.

II. Preliminaries

In this section we'll present some important concepts and terms used to formalize the discussion over the Diversified Enumeration Problem.

Definition 1. (Optimization Problem $\Pi(M,S,C)$) Let M be a collection of objects, and let S be the group of all legal solutions (subsets of objects from M), defined by some constraints over the elements of M. Let $C : S \to \mathbb{R}^+$ be the cost function used to evaluate the quality of each possible solution in S. Solving an optimization problem $\Pi(M,S,C)$ is equivalent to finding a solution $s_{opt} \in S$ such that $s_{opt} = argmin(\min_{s\in S}\{C(s)\})$.

Remark 2. For the purpose of this work, we make the assumption that the cost function is "additive", that is, for every solution $s = \{m_1, m_2, \ldots, m_l\} \in S$:

$$C(s) = \sum_{i=1}^{l} c(m_i)$$

Where $c : M \to \mathbb{R}^+$ evaluates the individual cost of an element $m \in M$, and holds that $c(m) = C(\{m\})$.

We will also make the assumption that the optimization problem can be solved in polynomial time: $O(q(|M|))$.

Many optimization problems can be modeled using this notation. For example, see section VII-B.

Definition 3. (Diversity Function, diversity score) A diversity function $D : S^K \to [0,1]$ takes a series of K solutions $\sigma = [s_1, \ldots, s_n]$ as an argument, and outputs the diversity score of σ, which is a number between $[0,1]$. The higher is the diversity score of σ, the more diversified σ is.

-

Definition 4. (The Diversified Top-K Enumeration Problem $DEP(\Pi(M,S,C),K,D)$) Let $\Pi(M,S,C)$ be an optimization problem, with M, S and C as defined in Definition 1. Let $K \in \mathbb{N}$ be the number of required solutions, and let D be a diversity function. The Diversified Top-K Enumeration Problem aims to find $\sigma = [s_1, \ldots, s_n]$, a series of $n \leq K$ elements in S, where:

1) $\forall i,j \in [1,n], j \neq i, C(s_i) \leq C(s_j)$
2) $D(\sigma)$ is maximized.
3) $\sum_{i=1}^{n} C(s_i)$ is minimized.

Definition 5. (Penalty Function) A penalty function is a function $p : M, \mathbb{N} \cup \{0\} \to \mathbb{R}^+$ that takes an element $m \in M$ and the number of times it appeared in past solutions o, and returns an updated cost for this element. It must hold that for $o = 0$ $p(m,0) = c(m)$, and for $o > 0$ $p(m,o) \geq c(m)$, where $c(m)$ is the original cost of m.

A. The Complexity of Enumeration Algorithms

An enumeration algorithm is considered tractable if it can guarantee one of the following:

1) Polynomial Total Time - The time it takes to output all of the solutions is polynomial in the input size + the output size.
2) Incremental Polynomial Time - For all i, the time it takes to produce the i-th solution is polynomial in the input size and i.
3) Polynomial Delay - The delay between successive solutions is polynomial in the input size.

Polynomial delay is considered better than incremental polynomial time, and incremental polynomial time is considered better than polynomial total time.

III. Related Work

There is a short summery of the articles that were used as a background for this project:

A. Enumerating The Top-K solutions

The methods for solving the diversified top-K problem described in this report relies heavily on Lawler's enumeration algorithm[1]. In short, Lawler suggests a general procedure to enumerate all the solutions of a problem, in a ranked order, meaning that the best solution is produced first, the second-best is produces next and so on. For a more detailed explanation about the algorithm, see III-C.

B. Enumerating The Diversified Top-K solutions

1) Enumerating Diversified Solutions of a Specific Problem: Some articles tries to find methods to enumerate the solutions of a specific problem, while taking diversity into account: in [2] the authors suggests a method to enumerate solutions of the shortest path problem. They used Lawler's algorithm to enumerate paths, while preventing farther investigation of partial solutions that seemed too similar to the optimal path by using logical constraints.

2) A General Framework for Diverse Enumeration: In [3] the authors suggests a general framework for finding a set of K diversified solutions: they convert the original problem into the problem of finding a maximal-weight k independent set, than they use an A^* and divide and conquer approach to solve it.

C. Lawler's Algorithm

Lawler's algorithm [1] is used to produce solutions of some optimization problem $\Pi(M, S, C)$ in ranked order. Intuitively, the algorithm divides S into parts using logical constraints, finds the minimum-cost solution in each part, and outputs the best solution among the local minima. It repeats this process iteratively, and in each iteration produces the next-best solution.

A pseudo-code for the algorithm appears in Algorithm 1. Lawler's algorithm gets an optimization problem $\Pi(M, S, C)$, and $K > 0$, which is the number of solutions to produce. In lines $2-5$ the algorithm initializes Q, a priority queue that holds the solutions that were found during the execution ordered by their cost (the top element holds the minimal-cost solution). Each element in Q represents the best solution in some subsection of S defined by logical constraints. It contains the constraints I, E and the optimal solution to $\Pi(M, S, C)$ under those constraints. I is the inclusion constraint, and defines what elements $m \in M$ the solution must contain, similarly, E is the exclusion constraint, and defines what elements $m \in M$ must not appear in the solution. We'll denote the subsection of S defined by I and E as $S(I, E)$. When initializing Q, I and E are set to be $\emptyset$, so the solution pushed into Q in line 5 is the optimal solution to the problem $\Pi(M, S, C)$.

In lines $6-14$ the algorithm iteratively outputs new solutions:

Algorithm 1 Lawler's Algorithm

```
    Input problem, K>0
 1: procedure LAWLER'S ALGORITHM
 2:     k ← 0
 3:     Q ← []
 4:     I, E ← ∅
 5:     top ← null
 6:     Q.push([getSolution(problem,I,E),I,E])
 7:     while Q do
 8:         top ← Q.pop()
 9:         print(top[0])
10:         k ← k + 1
11:         if k == K then
12:             break
13:         end if
14:         insertNewSolutions(Q, top)
15:     end while
16: end procedure
```

in each iteration i, the algorithm pops Q to get the $k'th$-best solution of the problem, and if $k < K$ it inserts Q new elements derived from the $k'th$ solution by updating I and E. The process is repeated until $k = K$ or there are no more distinct solutions to be found.

If $\Pi(M, S, C)$ can be solved in polynomial time, Lawler's algorithm guarantees polynomial delay between the produced solutions.

IV. Diversified Top-K Solutions Search

To enumerate K solutions with regard to diversity, we tried to modify Lawler's algorithm so that on one hand the resulted solutions would be different enough, and on the other hand, would not cost significantly more than the original top-K solutions. The original algorithm doesn't take diversity into consideration, since solutions are enumerated according to the function cost only, and the later measures the cost of each solution individually, without taking into account it's similarity to what was already produced.

Therefore, we tried to manipulate the solutions cost so it would contain penalty for solutions similar to what the algorithm had already chosen. That way, if we have several solutions in Q with similar costs (original costs, without penalty), solutions different from the ones already produced would have more chance of being enumerated.

To put this idea into practice, we kept a record of how many times each element $m \in M$ had appeared in past solutions:

$$Occur_k = \{(m, o_k) | o_k = \text{\#occurrences of } m \text{ until iteration } k\}$$

and used it to change the solutions cost: in each iteration k, the cost function is updated in the next manner:

$$C_k(s) = \sum_{j=1}^{l} p(m_j, Occur_k[m_j])$$

where $s = (m_1, m_2, \ldots, m_l)$ is a solution in S, p is a penalty function (see 5) and $Occure_k[m_j]$ stands for the number of

occurrences o_k corresponding to m in $Occur_k$.

In the modified algorithm, instead of requiring that the solution enumerated in the $k'th$ iteration would the $k'th$ best solution according to C, we require that it would be the minimal-cost solution according to C_k that wasn't enumerated yet. For convenience, from now on we'll regard this requirement as the queue invariant:

Definition 6. (The queue invariant) in the beginning of each iteration $k \leq K$ of the modified Lawler's algorithm, the priority queue Q holds the minimal-cost solution according to C_k that wasn't enumerated yet.

In order to keep the queue invariant, the priority queue Q needs to be updated after changing the cost function, since it doesn't necessarily holds the solutions in the right order anymore, and some of the elements (s, I, E) might no longer contain the optimal solution for $S(I, E)$ (with regard to the new cost function). After updating Q while considering those two problems, the queue invariant should hold again in the beginning of the next iteration.
A pseudo-code of the modified version appears in Algorithm 2

Algorithm 2 Lawler's Algorithm Modification

Input *problem, K>0*
procedure DIVERSIFIED LAWLER
  $k \leftarrow 0$
  $Q \leftarrow$ *[]*
  $occur \leftarrow \{\}$
  $I, E \leftarrow \emptyset$
  $top \leftarrow null$
  Q.push([getSolution(problem,I,E),I,E])
  while Q **do**
    top $\leftarrow$ *Q.pop()*
    $print(top[0])$
    $k \leftarrow k + 1$
    if $k == K$ **then**
      $break$
    end if
    $occur.update(top)$
    $problem.updateCosts(occur)$
    $update(\&Q, problem)$
    $Q.insertNewSolutions(top)$
  end while
end procedure

V. UPDATING THE QUEUE

As mentioned, after changing the cost function, Q needs to be updated in order to make the queue invariant hold again in the next iteration.
We will describe two methods for updating Q: the first method (the Naive Approach) updates Q in a naive fashion, and illustrates how inefficient this process can be at the worst case scenario. The second method (the Lazy Approach) tries to filter unnecessary calculations using the fact that the cost of a solution can only rise as the algorithm proceeds. Both approaches has the same time complexity, however, the lazy method performs significantly better in practice.

A. The Naive Approach

In the naive approach, we build a new updated queue Q' using Q. To build Q' all that's required is to do is to iteratively pop the top element of Q: (s, I, E), re-solve the updated optimization problem $\Pi'(M, S, C_k)$ under the constraints I, E, and then push the result (s', I, E) into Q'. A pseudo-code for this approach appears in Algorithm 3 . It is

Algorithm 3 The Naive Approach

Input *QPtr, updatedProb*
procedure NAIVEUPDATE
  $Q' \leftarrow$ *[]*
  while $!QPtr.empty()$ **do**
    top $\leftarrow$ *QPtr.pop()*
    $Q'.push([getSolution(updatedProb, I, E), I, E])$
  end while
  $QPtr^* \leftarrow$ *Q'*
end procedure

easy to see that when using this updating approach, the queue invariant holds, since in the beginning of each iteration all of the elements in Q contain the optimal solutions for the parts of S they represent, and they ordered according to the updated cost function.

B. The Lazy Approach

The lazy approach is based on the observation that in some cases, in order to satisfy the queue invariant, there's no need to recalculate and reorder all of the elements in Q, but just a limited number of them. Remember, that in order to satisfy the invariant, it's enough that in the end of each iteration the **head** of Q contains a suitable solution. So, as long as the head has the required property, the order or the content of the other elements in Q doesn't matter.

In the lazy approach, we'll make only the necessary updates to ensure that the head of Q satisfies the invariant. To do so, after changing the cost function, we'll check if Q's head already contains a solution to satisfy the invariant, and only if it doesn't, we'll update it and repeat the test with the next head
To test if Q's head (s, I, E) satisfies the invariant, the lazy method can check it's cost, and see whether it hadn't change since the last time this element was updated. The reason behind this, is that solutions cost can only ascent as the iterations go, implying that:

1) Even with respect to the new cost function, s is still the cheapest solution in $S(I, E)$.
2) s is still the least cheapest solution in Q.

Hence, s is the best solution according to the updated cost function, and it wasn't enumerated yet, so the queue invariant holds. If the head doesn't pass the test, we can pop it, re-solve

the optimization problem for $S(I,E)$ with respect to the new cost function, and push the result back into Q. Eventually, the head will be either one of the elements we have updated, or an element that passes the test. In both cases, the queue invariant holds.

Sometimes, it isn't even necessary to re-solve the optimization problem when the head (s,I,E) doesn't pass the test. If the costs of all solutions in $S(I,E)$ are effected at least as badly as the cost of s, s is still a local minimum. It's enough to pop (s,I,E), recalculate s's cost (which is cheaper then finding a new solution), and insert the updated element back into Q. These kind of situations happen when the only records that were changed in $Occur$ since the last update of (s,I,E) are not in $s \backslash I$.

A pseudo-code for the lazy approach appears in Algorithm 4

Algorithm 4 The lazy approach

```
    Input QPtr, updatedProb
 1: procedure LAZYUPDATE
 2:     if !QPtr.isEmpty() then
 3:         top ← Qptr.peek()
 4:         while top.currentScore != top.updatedScore do
 5:             QPtr.pop()
 6:             if top.updatedEelemnts ⊆ top.I then
 7:                 top.updateCost()
 8:                 QPtr.push(top)
 9:             else
10:                 newSol ← getSol(updatedProb,I,E)
11:                 QPtr.push([newSol, I, E])
12:             end if
13:             top ← Qptr.peek()
14:         end while
15:     end if
16: end procedure
```

VI. Complexity Guarantee

Both algorithms (the one using the naive approach and the one using the lazy approach) guarantee incremental-polynomial time, as can be shown by proving the following theorems:

Lemma 7. *Denote $U(Q_i)$ as the cost of updating Q in iteration i. The time complexity of updating Q using the naive approach is $U(Q_i) = O(i \cdot |M| (q(|M|) + \log(i) + \log(|M|)))$*

Proof: In the naive updating approach there is no meaning to the order of the elements in Q_i, therefore, in practice, it is possible to treat Q_i as a list of elements, and iterating over it's elements instead of popping them one by one as the pseudo code in Algorithm 3 suggests. For each element in Q_i, the optimization problem is re-solved, and an element with the updated solution is inserted into Q'. Since the cost of solving the optimization problem for each element is $O(q(|M|))$, and since the cost of inserting an element into Q' is $O(\log(|Q_i|))$, the cost of updating the queue naively is:

$$O(|Q_i| (q(|M|) + \log(|Q_i|)))$$

In Lawler's algorithm, in each iteration there are at most $O(|M|)$ new elements that are inserted into the queue, so the the queue's size in iteration i is at most $O(i \cdot |M|)$. Using this information, we can deduce that:

$$\begin{aligned} U(Q_i) &= O(i \cdot |M| (q(|M|) + \log(i \cdot |M|))) \\ &= O(i \cdot |M| (q(|M|) + \log(i) + \log(|M|))) \end{aligned}$$

as required.

Theorem 8. *The modified Lawler's algorithm achieves an incremental polynomial time when using the naive updating method.*

Proof: To show that the algorithm is polynomial-incremental, it is required to prove that the time needed to enumerate the first k solutions is polynomial in k and the size of the input $|M|$. To do so, we'll first analyze the time interval between every subsequent solutions:
Assume that the algorithm had just returned the k-th solution, the time it takes to return the next solution is consisted of:

1) The time it takes to update $Occure$ (line 15 in Algorithm 2)
2) The time it takes to update the cost function (line 16 in Algorithm 2)
3) The time it takes to update the queue (line 17 in Algorithm 2)
4) The time it takes to insert the queue new solutions to the queue (line 17 in Algorithm 2)
5) The time it takes to pop the $k+1$-th solution from the queue (line 9 in Algorithm 2)

Note: In this proof, we'll assume that Q is implemented as a heap, meaning that peeking costs $O(1)$, and that popping the top element or pushing a new element into Q costs $O(\log(|Q|))$.

Bounding 1. and 2.:
Updating $Occur$ can be accomplished in $O(|M|)$ time, since there are $O(|M|)$ elements at most in every solution $s \in S$, and if $Occur$ is implemented as a hash table, updating $O(|M|)$ records takes $O(|M|)$ time. From the same reason, it is possible to update the score function in $O(|M|)$ time, since there are at most $|M|$ element costs that need to be updated.

Bounding 3.
According to Lemma 7 the time complexity of updating Q in iteration k using the naive approach is:

$$U(Q_k) == O(k \cdot |M| (q(|M|) + \log(k) + \log(|M|)))$$

Bounding 4.
In each iteration of the algorithm at most $|M|$ solutions could be generated. Therefore, the time it takes to accomplish 4. is $O(|M| (q(|M|) + \log|Q_k|))$ where $|Q_k|$ is the size of Q in

iteration k. If in each iteration there are $O(|M|)$ elements that are inserted into Q, $|Q_k|$ is at most $k \cdot |M|$, and therefore the cost of inserting new solutions to the queue can be expressed as:

$$O(|M|(q(|M|) + \log(k \cdot |M|))) = O(|M|(q(|M|) + \log(k) + \log|M|))$$

Bounding 5.
The time it takes to pop the $k+1$-th solution is:

$$O(\log|Q_k|) = O(\log(k) + \log(|M|))$$

according to our assumption that Q is implemented as a heap.

We can sum it all up and express the waiting time between two consecutive solutions as follows:

$$TimeInterval(k, k+1) = O(U(Q_k))$$

and use it to express the waiting time it takes to return k solutions:

$$\begin{aligned}
T(k) =& O\left(\sum_{i=1}^{k} TimeInterval(i, i+1)\right) \\
=& O\left(\sum_{i=1}^{k} U(Q_i)\right) \\
=& O\left(\sum_{i=1}^{k} i \cdot |M|(q(|M|) + \log(i) + \log(|M|))\right) \\
=& O\left(\sum_{i=1}^{k} i \cdot |M|(q(|M|) + i + |M|)\right) \\
=& O\left(\sum_{i=1}^{k} i \cdot |M| q(|M|) + i^2|M| + i|M|^2\right) \\
=& O\left(|M| \cdot \left(q(|M|)\sum_{i=1}^{k} i + \sum_{i=1}^{k} i^2 + |M|\sum_{i=1}^{k} i\right)\right) \\
=& O\left(|M| \cdot \left(q(|M|)k^2 + k^3 + |M|k^2\right)\right) \\
=& O\left(|M|k^3 + |M| q(|M|) k^2 + |M|^2 k^2\right)
\end{aligned}$$

which is polynomial in $|M|$ and k, as required.

Lemma 9. *Denote $U(Q_i)$ as the cost of updating Q in iteration i. The time complexity of updating Q using the lazy approach is $U(Q_i) = O(d \cdot (q(|M|) + \log(|M|) + \log(i)))$ where d is the number of elements it was necessary to update.*

Proof: d is equal to the number of while iterations performed in Algorithm 4. The cost of a single iteration, at the worst case, is $O(q(|M|) + \log(|M| \cdot i))$ (following that the queue size is at most $i \cdot |M|$ as explained in Lemma 7). We can deduce that the total updating cost is:

$$\begin{aligned}
U(Q_i) &= O(d \cdot (q(|M|) + \log(|M| \cdot i))) \\
&= O(d \cdot (q(|M|) + \log(|M|) + \log(i)))
\end{aligned}$$

as required.

Note: at the worst case, $d = |Q_i|$, and the complexity of updating the queue using the lazy method is no better that updating it naively. However, as presented in section VIII, the lazy updating approach achieves better results in practice.

Theorem 10. *The modified Lawler's algorithm achieves an incremental polynomial time when using the lazy updating method.*

Proof: The proof immediately follows using the same logic as in the proof of Theorem 8, and replacing d with $|Q_i|$.

VII. Implementation

The original Lawler's algorithm and the two variations described in section IV are implemented using python 3.7 and NetworkX [4]. NetworkX is a python package that allows to load, manipulate and study the structure of complex networks. The suggested implementation is available at [5]. In this section, we'll present the prototype, describe how we applied it on a specific problem, and suggest further optimizations that can be used to shorten the algorithms run-time.

A. Enumerators Implementation

The different algorithms (Lawler's, The lazy version, the naive version) are implemented as generators, that get a general problem object, and process it without any previous assumptions. All of the enumerators follow the general framework described Algorithm 1, but each one of them updates it's queue (if at all) differently. Since the algorithms don't assume anything on the nature of the problem, they can be applied to any optimization problem that can be modeled after the notation suggested in definition 1. However, the general problem class is abstract, and to evaluate the algorithms, at least one specific problem had to be implemented. The problem we chose to implement is the shortest path problem in directed graphs.

B. Example - The Shortest Path Problem

We can use the terms defined in Definition 1 to represent, for example, the shortest path problem. Let $G(E, V)$ be a directed graph, where E represents the edges of G and V it's nodes. Let $x \in V$ be the source node and $y \in V$ be the destination. Also, let $w : E \to \mathbb{R}^+$ be a weight function.
To formalize this problem in the same terms we used in Definition 1, we can set $M = E$, S to be:

$$S = \{[e_1, \ldots, e_n] \,|\, e_1[0] = x, e_n[1] = y, \forall i \in [1, n]\, e_i \in E\}$$

and of course, the weight function can be used to construct the cost function, that is, for all $s = (e_1, e_2, \ldots, e_l) \in S$:

$$C(s) = \sum_{i=1}^{l} w(e_i)$$

In this problem, I defines a sequence of edges any $s \in S(I, E)$ should start with, and E defines the edges s mustn't contain. In the suggested implementation, the constraints are applied by temporarily removing the excluded edges from G (defined by E), then temporarily removing the edges defined

by I, and finding the shortest path from the midpoint (the last node I enforces) to y using Dijkstra's algorithm. After a solution from $S(I, E)$ was found, G returns to it's previous state.
Penalty is applied iteratively, meaning that in each iteration of the algorithm, after producing the k-th solution, the coast of each edge that appears in the k-th solution is increased. For example, if $p(e, o) = c(e) \cdot 2^o$, then in each iteration e appears in the produced solution the cost of e is doubled by 2.

C. Further Optimizations

It is likely that solutions that are far more expansive then the optimal solutions are mostly irrelevant. To shorten the modified algorithm running time, we decided to limit the produced solutions cost up to $\times 1.5$ the optimal cost. To do so, we pruned all of the nodes $v \in G$ that hold that the shortest paths from x to v costs more then $\times 1.5$ the optimal cost from x to y.

VIII. Evaluation

In this section, we describe our experimental results. The goal of the experiments is to demonstrate the improvement in diversity the modified algorithms achieves, and the affect on the solutions cost.
To perform the experiments, we used the shortest path optimization problem (described in VII-B) as a test case. We used Pennsylvania road network graph [6] as an input for the problem, using two fixed points as a source and a destination.

A. Running Times

In this section, we'll compare the running times of the Lawler's algorithm to that of the lazy variant and to that of the naive variant. The running times of all algorithms appears in table I . As expected, Lawler's algorithm has the best results.

TABLE I
Algorithms Running Times

	Original	Naive	Lazy
$K = 5$	$0.021s$	$0.171s$	$0.057s$
$K = 10$	$0.031s$	$0.609s$	$0.265s$
$K = 50$	$0.194s$	$40.05s$	$11.631s$

Regarding the other two, the lazy variant performs significantly better than the naive variant, especially when K is large.
Figure 1 follows the time it takes each algorithm to generate the k-th solution during one execution with $K = 50$. From the figure, the bigger k is, the more time it takes the two variants to generate the $k'th$ solution. It makes sense, since the lazy and naive variants only achieve incremental-polynomial time complexity. Like before, the lazy variant performs better than the naive variant.
Both experiments were performed using $penalty(m, o) = c(m) \cdot 1.2^o$ as the penalty function.

Fig. 1. Time to Generate The k-th Solution

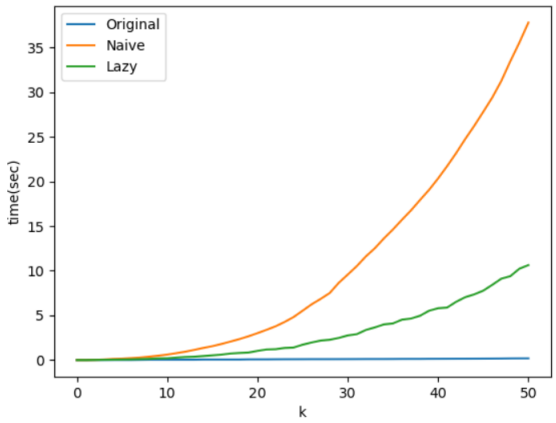

TABLE II
Average Path Cost

	Original	Lazy
$K = 5$	12.8	13.4
$K = 10$	13.1	14.1
$K = 50$	14.32	22.26

B. Average Path Cost Comparison

In table II we present the average path cost the modified algorithm and the original Lawler's algorithm achieved. For this experiment, we used $penalty(m, o) = c(m) \cdot 1.2^o$ as a penalty function.
As the results show, with $K = 5, 10$ there isn't a significant difference in average costs between the lazy variant and the original algorithm. With $K = 50$ the difference is already larger. The average cost of the solutions produced by the lazy variant can be controlled by the optimization described in section VII-C. Pruning the graph more massively will result in lower average cost of the solutions produced by the lazy algorithm, but would (adversely) affect the diversity as well.

C. Diversity Evaluation

This section is dedicated to evaluate the quality of the solution diversity our approach provides.

1) Comparison of Different Diversity Functions: In table III we compare the diversity our algorithm achieved relatively to that of the original algorithm, while using different diversity functions. The diversity functions we used are:

TABLE III
Different Diversity Functions Comparison

	Original	Lazy
D_1	$2.59 \cdot 10^{-5}$	$11.02 \cdot 10^{-5}$
$D_2, t = 0.25$	0.62	0.92
$D_3, t = 0.25$	0.23	0.66
$D_2, t = 0.5$	0.24	0.64
$D_3, t = 0.5$	0.08	0.42
$D_2, t = 0.75$	0.08	0.36
$D_3, t = 0.75$	0.06	0.2

1) D_1(percentage of edges seen) - let $\sigma = [p_1, \ldots, p_k]$ be a series of k paths in the input graph $G = (V, E)$, then:

$$D_1(\sigma) = \frac{|\{e|e \in E \wedge \exists i \in [1,k]\, e \in edges(p_i)\}|}{|E|}$$

where $edges(p_i)$ is a group consisted of the edges that appear in p_i.

2) D_2(Jaccard distance) - let $\sigma = [p_1, \ldots, p_k]$ be a series of k paths in the input graph $G = (V, E)$, and let $d = [p'_1, p'_2, \ldots, p'_l]$ be a series of $l \leq k$ paths constructed as follows: for each $i \in [1,k]$ p_i can be added to d if d is empty or if $\forall j \in [1,l]$ it holds that

$$1 - \frac{|edges(p_i) \cap edges(p'_j)|}{|edges(p_i) \cup edges(p'_j)|} > t$$

where $t \in [0,1]$.
Then, D_2 is defined as:

$$D_2(\sigma) = \frac{|d|}{k}$$

3) D_3(path-replacement distance) - let $\sigma = [p_1, \ldots, p_k]$ be a series of k paths in the input graph $G = (V, E)$, and let $d = [p'_1, p'_2, \ldots, p'_l]$ be a series of $l \leq k$ paths constructed as follows: for each $i \in [1,k]$ p_i can be added to d if d is empty or if $\forall j \in [1,l]$ it holds that

$$\frac{|edges(p_i) \setminus edges(p'_j)|}{|edges(p_i)|} > t$$

where $t \in [0,1]$.
Then, D_2 is defined as:

$$D_3(\sigma) = \frac{|d|}{k}$$

For this experiment, we used $K = 50$ and $penalty(m, o) = c(m) \cdot 1.2^o$.
From the results, it is clear that the lazy variation achieves improvement in diversity over the original algorithm according to all diversity functions. In most cases, the modified algorithm gets about $\times 3$ diversified solutions comparing to the original algorithm.

2) Comparison of Different Penalty Functions: In figure IV we compare the effect of different penalty functions on the diversity. The experiment was performed using $K = 50$ and $D_3$with $t = 0.75$ as the diversity function.
As described in the results, p_2 achieves better diversity then p_1. This is expected, since p_2 gives "heavier" punishments then p_1, making solutions with edges that have already seen in past iterations more expensive and unattractive. The same can be said about p_2 and p_3.

TABLE IV
PENALTY FUNCTIONS COMPARISON

	Lazy
$p_1(m, o) = c(m) + o$	0.18
$p_2(m, o) = c(m) \cdot 1.2^o$	0.2
$p_3(m, o) = (c(m) + 1)^o$	0.22

IX. CONCLUSIONS

Diversity is often a desirable attribute for a solution group, even at the cost of a small reduction in other quality parameters. The modified enumeration algorithm presented can achieve significantly better diversity, with reasonable affect on the original solutions cost. Yet, both versions of the modified algorithm are significantly slower then the original. In addition, it's also hard to control the output's cost-diversity trade off.
Some interesting directions for future work might be to investigate one of the problems mentioned above. Other ideas might be to find an optimal penalty function, or to design an enumeration algorithm that can emphasize other quality parameters, such as solutions simplicity, length, popularity etc.

REFERENCES

[1] E. L. Lawler, "A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem," *Management Science*, 1972.
[2] S. F. C. N.J. van der Zijpp, "Path enumeration by finding the constrained k-shortest paths," *Elsevier*, 2004.
[3] L. C. Lu Qin, Jeffrey Xu Yu, "Diversifying top-k results," *PVLDB*, 2012.
[4] *NetworkX*. [Online]. Available: https://networkx.github.io/
[5] *Project code at github*. [Online]. Available: https://github.com/noystl/GuidedWork
[6] *SNAP*. [Online]. Available: https://snap.stanford.edu/data/index.html