Distributed Inference in Sensor Networks
João Manuel Melo Silva Gueifão
joao.gueifao@tecnico.ulisboa.pt
Instituto Superior Técnico, Lisboa, Portugal
December 2015

Abstract—We consider the problem of finding the path, with given length, that accumulates the largest reward in a graph in which nodes hold prizes. This problem arises in distributed parameter estimation and corresponds to finding the most informative path, with given length, through a sensor network. We conceive a variety of algorithms that strike different trade-offs between computational complexity and solution quality. Among the variety of approaches we develop, two stand up: an integer linear program (ILP) formulation that allows the application of any general-purpose ILP solver to solve the problem at hand, and a suboptimal dynamic programming-based method that replicates the ILP performance but has predictable running time and it is far easier to implement. We also discuss the extension of our methods to the scenario in which nodes own prizes from several categories.

Index Terms—sensor networks, sensor selection, optimization, dynamic programming

I. INTRODUCTION

WIRELESS sensor networks assess the physical state of an environment; they consist of measurement units—the sensors—scattered over a region of interest and linked via wireless. Each sensor is small, cheap and battery-operated, and the measurement accuracy of each unit varies from sensor to sensor.

a) Parameter estimation: As soon as sensors take measurements, their readings must be processed to squeeze out the relevant environment’s parameter, e.g., temperature, humidity or soil pollution.

b) Rapid dynamics: When tracking physical phenomena with fast dynamics, we cannot profit from all network sensors—the time needed to collect the entire network observations and to do the lengthy number-crunching would render the estimate obsolete. In those critical scenarios we must estimate the parameter using few sensors. The obligation of activating only a small subset of sensors also holds for slow-changing phenomena that are sampled scarcely (to save network energy) and that demand quick estimators.

The limited onboard battery of each sensor restricts its wireless communication range. As a result, each sensor can reach only a small number of neighbors and this feature favors estimators that are token-based (according to [1], [2]): a data-carrying token starts at a given sensor and proceeds to amass observations by hopping from sensor to sensor; after a fixed number of jumps, the token ends its journey and the parameter is estimated based on the information available in the token. This is a distributed inference method since no fusion center is involved.

In this approach the whole network is not switched on, just the sensors that the token visits. We control the time required to form an estimate by selecting the length of the token’s path, defined as the number of visited nodes including multiplicities. For example, if the path starts at node 4, jumps to 7, jumps to 4, jumps to 9 and does a final jump to 17, then this path’s length is five; in this example, node 4 is visited twice (although it contributes only one measurement; on the second visit, node 4 acts as a jumping point for the token to reach node 9). To simplify notation, we will refer to such a path by $4 \rightarrow 7 \rightarrow 4 \rightarrow 9 \rightarrow 17$.

To design a token-based estimator we must specify the path, i.e., the sequence of visited nodes—we assume here that the path’s length is given. All paths with the same length are not equivalent as different sensors exhibit different measurement accuracies. We seek the most informative path, the one passing through the most accurate sensors.

We model the sensor network as a graph $G = (V, E)$ where $V$ is the set of nodes and $E$ is the set of edges. Each node represents a sensor and each edge represents a wireless channel between sensors. The set of nodes is denoted $V = \{v_1, v_2, \ldots, v_n\}$ and each node is endowed with a number, say, $p_i$ for node $v_i$; this number quantifies the worth of the sensor’s reading (sensors with better measurement accuracies have higher $p_i$).

It helps to think of the $p_i$’s as prizes available to be collected—only once—at the nodes.

Assume the given path’s length is $K$. In this paper, we focus on the following Prize Accumulation Problem (PAP): find a path with length $K$ that maximizes the sum of accumulated prizes.

Note the importance of the graph structure on the problem statement; if there were no graph (equivalently, the graph was fully connected, i.e., each node could reach any other node) the problem would have a trivial solution: choose the nodes with the top $K$ prizes. For a generic graph, this choice is usually infeasible because the nodes with best prizes cannot be visited in a path with length $K$.

A. Application example

We illustrate a popular instance of the prize accumulation problem. The parameter of interest is denoted by $\theta \in \mathbb{R}$ and sensor $i$ reads it embedded in gaussian noise:

$$y_i = \theta + \eta_i,$$
where $y_i$ is the observation available at sensor $i$ and $\eta_i \sim \mathcal{N}(0, \sigma^2_i)$ stands for a gaussian random variable with zero mean and variance $\sigma^2_i$, i.e., with probability density function

$$p(\eta_i) = \frac{1}{\sqrt{2\pi\sigma_i^2}} e^{-\frac{1}{2} \frac{\eta_i^2}{\sigma_i^2}}.$$ 

The variance sets the sensor accuracy: the smaller the noise strength $\sigma^2$, the better the sensor accuracy.

Suppose we activate $K$ sensors. To simplify notation, we assume that we activate sensors $1$ to $K$ and that those sensors constitute a feasible path in the graph. The observation vector $y := (y_1, y_2, \ldots, y_K) \in \mathbb{R}^K$ is given by

$$y = \theta 1_K + \eta$$

where $1_K := (1, 1, \ldots, 1)$ and $\eta := (\eta_1, \eta_2, \ldots, \eta_K)$.

Assuming the measurements are independent, the maximum-likelihood (ML) estimator (executed at the end of the token’s path) is

$$\hat{\theta}_{\text{ML}} = \arg\max_{\theta} \log p(y; \theta)$$

$$= \arg\min_{\theta} \sum_{i=1}^{K} \frac{y_i - \theta)^2}{2\sigma_i^2}$$

$$= \sum_{i=1}^{K} \frac{y_i}{\sigma_i^2}$$

It is straightforward to show that this estimator is unbiased and its variance is

$$\text{var}(\hat{\theta}_{\text{ML}}) = \mathbb{E} \left\{ (\theta - \hat{\theta}_{\text{ML}})^2 \right\}$$

$$= \frac{1}{\sum_{i=1}^{K} \frac{1}{\sigma_i^2}}.$$ 

We see that the precision of the estimator, i.e., the inverse of its variance, is given by the sum of the precisions of the sensors involved in the path:

$$\frac{1}{\text{var}(\hat{\theta}_{\text{ML}})} = \sum_{i=1}^{K} p_i$$

with

$$p_i := \frac{1}{\sigma_i^2}.$$ 

This conclusion applies to any other path of length $K$. Thus, in order to reduce the variance of the estimator, we must select the path of length $K$ that maximizes the sum of the visited $p_i$’s—an optimization challenge that matches our problem formulation.

### B. Related literature and contributions

The research literature addresses several flavors of the generic problem of selecting $K$ most informative sensors.

Works close to ours are [3], [4], [5], [6] and [7]. They differ from our problem statement in that either the merit of each subset of $K$ nodes cannot be cast as a sum of node prizes, or the set of $K$ nodes are only required to form a connected subgraph—a path always generate a connected subgraph, but a general connected subgraph with $K$ nodes cannot always be generated by a path with $K$ nodes. The latter flavor is known as the Maximum Weight Connected Graph Problem.

Our problem statement falls into the scope of the work in [8] which proposes an heuristic to optimize a submodular function over the paths of $K$ nodes.

In this paper we propose several approaches to tackle our problem statement. The algorithms we develop apply tools from linear programming (LP) and dynamic programming and offer different trade-offs in terms of solution sub-optimality and computational complexity.

The key finding is that our idea of combining the rollout framework developed in [9], [10] and [11] with the heuristic in [8] gives rise to a powerful solution method that matches the performance of state of art integer linear program (ILP) solvers packed with a profusion of sophisticated branch-and-bound shortcuts. Our approach is far simpler to implement (a modest number of lines in Matlab) and, contrary to the ILP solver, has a predictable running time.

### C. Organization

The organisation of this paper is as follows: section II models the representation of paths and formulates the Prize Accumulation Problem as a Integer Linear Program (ILP). section III presents a greedy search algorithm in its original and improved versions, section IV wraps the latter algorithm in the context of dynamic programming, section V extends the problem formulation to the multi-category prize context. Sections VI and VII describe the implementation of presented algorithms and discuss the provided execution results. Finally, section VIII summarizes the achievements made.

### II. FORMULATION AS AN INTEGER LINEAR PROGRAM

The Prize Accumulation Problem (PAP) can be formulated as an Integer Linear Program (ILP). To reach the ILP formulation, we need to introduce the concept of space-time graph (STG) — an operation that produces a large graph by creating several time folds of a given one. The Prize Accumulation Problem (PAP) seeks the path of length $K$ that reaps the largest sum of prizes. This problem statement evokes a search over a discrete domain—the set of all paths of length $K$ in a given graph $G = (V, E)$. To escape the combinatorial nature of the problem we need to represent paths in an optimization-friendlier setting. The concept of network flows on STGs provides a suitable new mathematical home for our optimization variable.

#### A. How to represent a path?

A path in a given graph, say, $12 \rightarrow 7 \rightarrow 9 \rightarrow 20$, can be thought of as the route traced by one unit of water that enters the graph at node 12 and leaves it at node 20. In this example, we visualize the water entering the graph at node 12, flowing through the edge $12 \rightarrow 7$, then through the edge $7 \rightarrow 9$, and finally through the edge $9 \rightarrow 20$, after which it leaves the graph. The edges act as a water plumbing system; a path corresponds to a particular journey through the pipelines.
a) **Network flows:** The example generalizes, i.e., we can express any path via a network flow. By definition, a network flow is any feasible distribution of water across the edges of a network\(^1\) —by feasible distribution we mean a distribution that conserves water at each node like Kirchoff’s law for currents in electrical networks.

As we want to consider general entry and exit nodes (i.e., not fixed \(a \text{ priori}\)) we need to work with an augmented graph. We now detail the procedure.

b) **Augmented graph:** Let \(G = (V, E)\) be the given graph. We start by creating two extra nodes: a source node \(s\) and a sink node \(t\). Those nodes represent the virtual source and virtual sink of the unit of water that will move around \(G\). We then introduce new arcs\(^2\) from \(s\) to every node of \(G\), and from those to \(t\). We denote the new graph by \(G_t = (V_t, E_t)\), where the \(f\) subscript stands for flow. Note that \(V_t\) has two more nodes than \(V\) and \(E_t\) has a total of \(2|V| + |E|\) arcs.

Let \(x_f \in \mathbb{R}^{|E_t|}\) be a vector that represents a flow: the entry \(x_{fj}\) represents the amount of flow through the \(j\)th arc of \(G_t\).

To support a feasible distribution of water, the vector \(x_f\) cannot be arbitrary: it must obey the flow conservation laws. These are encoded in the linear system

\[
A_fx_f = b_t
\]

where \(A_f\) is the node-arc incidence matrix associated to \(G_t\) and \(b_t\) is the supply-demand vector. Row \(j\) of \(A_f\) and the corresponding \(j\)-th entry of \(b_t\) define the flow conservation at node \(j\); each row of \(A_f\) sums out all variable flows (represented by \(x_f\)) entering or exiting the associated node, and \(b_t\) represents the corresponding external supply or extraction (demand). If the \(j\)-th entry of \(b_t\) equals zero then node \(j\) is a transportation node (neither receives nor leaks water from and to the exterior); the sum of flow through the incoming edges matches the sum of flow through the outgoing edges.

In sum, a nonnegative vector \(x_f\) is called a network flow if it obeys (1).

\[\begin{bmatrix}
-1 & 1 & 0 \\
0 & -1 & 1 \\
\end{bmatrix}
\begin{bmatrix}
x_f_1 \\
x_f_2 \\
x_f_3 \\
\end{bmatrix}
= 
\begin{bmatrix}
3 \\
-1 \\
\end{bmatrix} \]  

\[\text{Note that } x_{f_j} \text{ is the amount of water traversing edge } e_j. \text{ The first row of the equation defines the flow balance at node } v_1, \]

\[-x_{f_1} + x_{f_2} = 3, \]

and the second row the flow balance at node \(v_2,\)

\[-x_{f_2} + x_{f_3} = -1. \]

Fig. 2. A graph \(G = (V, E)\) with \(V = \{v_1, v_2, v_3, v_4\}\) and \(E = \{e_1, e_2, e_3, e_4, e_5, e_6\}\).

d) **Example of an augmented graph:** Let \(G\) be the graph in figure 2. The node-arc incidence matrix \(A\) and supply-demand vector \(b_t\) of the graph \(G\) are:

\[
A = \begin{bmatrix}
1 & -1 & 1 & 1 & 0 & 0 \\
-1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & -1 & 1 & 1 & -1 \\
0 & 0 & 0 & 0 & -1 & 1 \\
\end{bmatrix}
\]

and

\[
b_t = (0, 0, 0, 0). \]

In figure 3, we display the associated augmented graph \(G_t = (V_t, E_t)\); note that \(V_t = \{s, V, t\}\) and \(E_t = \{s_1, s_2, s_3, s_4, E, t_1, t_2, t_3, t_4\}\). The pair \(A_f, b_t\) of the augmented graph \(G_t\) are:

\[
A_f = \begin{bmatrix}
I_{[V]}^T & 0_{[E]}^T & 0_{[V]}^T \\
-I_{[V]}^T & A & I_{[V]}^T \\
0_{[V]}^T & 0_{[E]}^T & -I_{[V]}^T \\
\end{bmatrix}
\]

and

\[
b_f = (1, 0, 0, 0, 0, -1). \]

---

\(^1\)Network is a synonym for graph.

\(^2\)An arc is sometimes defined in the literature as being a directed edge, or a ordered pair of nodes. We use both definitions interchangeably.
B. How to represent a path? Part 2

The space-time graph (STG) spans many instances of $G = (V, E)$ through time. If we are interested in paths of length $K$, the STG spans $K$ copies of $G$.

a) Construction of the STG: We start by creating $K$ copies of the set of nodes of $G$: $V$. Each one of the $K$ replicas is a temporal layer of the STG and is denoted by $l_k$, $k = 1, \ldots, K$. Finally, we link the temporal layers: for each arc $v_i \rightarrow v_j$ in $G$, we insert an arc connecting every pair of consecutive temporal layers $l_k$ and $l_{k+1}$ through nodes $v_i$ at $l_k$ and $v_j$ at $l_{k+1}$, for $k = 1, \ldots, K-1$.

Finally, we add a source node $s$ and a sink node $t$, and join them to the first and last temporal layer, respectively.

b) Example: In figure 4, we show the STG corresponding to the graph in figure 2—we omit the replications of arcs $e_1$, $e_2$, $e_4$ and $e_5$, to avoid overloading the figure.

c) Conservation system: We denote the STG by $G_{st} = (V_{st}, E_{st})$, where $st$ subscript stands for temporal flow. A flow in $G_{st}$ is any nonnegative vector $x_{st}$ that obeys the conservation system

$$A_{st} x_{st} = b_{st},$$

where $A_{st}$ is the node-arc incidence matrix of the STG and follows from the node-arc incidence matrix $A$ of the graph $G$ as follows:

$$A_{st} = \begin{pmatrix}
1_{|V|}^T & 0_{|E|}^T & 0_{|E|}^T & \cdots & 0_{|E|}^T & 0_{|E|}^T & 0_{|V|}^T \\
-I_{|V|} & A^+ & 0 & \cdots & 0 & 0 & 0 \\
0 & A^- & A^+ & \cdots & 0 & 0 & 0 \\
0 & 0 & A^- & \cdots & 0 & 0 & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & \cdots & A^- & A^+ & 0 \\
0 & 0 & 0 & \cdots & 0 & A^- & I_{|V|} \\
0_{|V|} & 0_{|E|}^T & 0_{|E|}^T & \cdots & 0_{|E|}^T & 0_{|E|}^T & -1_{|V|}^T
\end{pmatrix},$$

where $1_{|V|}$ and $0_{|V|}$ are column-vectors of $|V|$ ones or zeros, respectively, $I_{|V|}$ is the $|V|$-by-$|V|$ identity matrix. Also, $A^+$ is a matrix with the same dimension of $A$ that keeps only its positive entries; the remaining are set to zero. The matrix $A^-$ follows the same definition but for the negative entries of $A$. The pattern-column with descending entries of $A^+$ and $A^-$ entries appears $K-1$ times; the same number of temporal layer interconnections in $G_{st}$. For example, for the STG in figure 4,
Let \( A_{\ell f} \) be:

\[
A_{\ell f} = \begin{bmatrix}
1_{|V|}^T & 0_{|E|}^T & 0_{|E|}^T & 0_{|V|}^T \\
-I_{|V|} & A^+ & 0 & 0 \\
0 & 0 & A^+ & 0 \\
0_{|V|}^T & 0_{|E|}^T & 0_{|E|}^T & -I_{|V|}^T
\end{bmatrix}
\]  \( (4) \)

Note that \( x_{af} \) has now a total of \( 2|V| + (K-1)|E| \) entries, corresponding to the same total of arcs in the STG.

Also, \( b_{tf} \) is the supply-demand for the STG, and we define it (as before) equal to +1 at the first entry, -1 at the last one, and zero elsewhere.

d) Properties of the STG: The STG of any graph is acyclic, i.e., it is loopless. As such, it prohibits the presence of circulations—one of the main drawbacks of augmented graphs. Indeed, in a STG, we have arcs only pointing downwards, from layer to layer. So, it is impossible to traverse a cycle within it, i.e., to start a path at any node \( v \), and end at that same node \( v \); we would need to get upwards among the layers.

It follows that STGs cannot support disconnected paths. The flow balance equation \( A_{af}x_{af} = b_{tf} \) makes any flow start at the source node \( s \) and end at the node \( t \).

Finally, if \( z_{af} \) is a binary flow in a STG it is easy to recover the corresponding path in the original graph: the trajectory (encoded in \( z_{af} \)) of the unit of water crossing the STG passes through exactly one node of each temporal layer; if we drop the time stamp of each such visited node, we trace a path in the original graph.

C. The prize accumulation problem as an integer linear program

a) Preliminaries: the matrix of visits: Let \( G = (V, E) \) be the given graph and suppose \( x_{fi} \) represents a path of \( G \) in the augmented graph \( G_{ti} \), i.e., \( x_{fi} \) is a flow in \( G_{ti} \) that contains no disconnected paths nor circulations and it is a binary vector: the \( j \)th entry of \( x_{fi} \) is 1 if arc \( j \) belongs to the path, and 0 otherwise. The number of visits that the path embedded in \( x_{fi} \) pays to a general node \( v \) can be obtained as

\[
b^T_{vf}x_{fi}
\]

where \( b_{vf} \) is an \( |E| \)-dimensional binary vector whose \( j \)th entry is

\[
b_{vfj} = \begin{cases} 
1, & \text{if arc } j \text{ points into node } v \\
0, & \text{otherwise}
\end{cases}
\]  \( (5) \)

We define the matrix of visits as the \(|V|\)-by-\(|E|\) binary matrix that stacks all those vectors; its \( v \)-th row is \( b^T_{vf} \):

\[
B = \begin{bmatrix}
b_{v1}^T \\
b_{v2}^T \\
\vdots \\
b_{v|V|}^T
\end{bmatrix}
\]  \( (6) \)

This matrix is an useful device to define the integer linear program formulation.

b) Integer linear program (ILP) formulation: We work in terms of the STG of the given graph \( G = (V, E) \). The path length \( K \) determines the number of temporal layers of the STG.

Any flow \( x_{af} \) in the STG starts at the source \( s \) and ends at the sink \( t \), crossing all layers in between.

To retrieve the number of visits that the path encoded in \( x_{af} \) pays to a general node \( v \) in \( G \), we have to adapt the matrix of visits \( B \) to the STG, as follows:

\[
B_{tf} = \begin{bmatrix} I_{|V| \times |V|} & B & \cdots & B & 0_{|V| \times |V|} \end{bmatrix} \]  \( (7) \)

We concatenate horizontally a \(|V|\)-by-\(|V|\) identity matrix, a replication of \( B \) by \( K-1 \) times, and finally a \(|V|\)-by-\(|V|\) matrix of zeros. The identity matrix accounts for the arcs of \( G_{tf} \) pointing from the source node into every node in the first layer, and the zero matrix is due to the fact that every node in the last layer has an arc pointing into the sink node. The replication of \( B \) is needed because, when walking a path along \( G_{tf} \), a visit made to \( m \) replications of a same node \( v \), in the different layers, counts as \( m \) visits made to the same node \( v \) in \( G \), at different time instants.

We define the node-visit map

\[
s(x_{af}) = B_{af}x_{af}
\]  \( (8) \)

that returns the number of visits paid to each node in \( G \) by the path embedded in the flow \( x_{af} \) in \( G_{af} \). Note that \( s(x_{af}) \in \mathbb{R}^{[V]} \) is indexed by the nodes of \( V \); for \( v \in V \), the entry \( s_v(x_{af}) \) indicates how many times node \( v \) is visited by the path underlying \( x_{af} \). Also, the map \( s(\cdot) \) is linear.

Let \( p_v \) denote the prize available at node \( v \in G \). We can formulate the prize accumulation problem as

\[
\begin{align*}
\text{maximize} & \quad \sum_{v \in V} p_v \min\{1, s_v( x_{af} ) \} \\
\text{subject to} & \quad A_{af}x_{af} = b_{tf} \\
& \quad x_{afj} \in \{0,1\}, \quad \forall j = 1, \ldots, |E_{af}|.
\end{align*}
\]  \( (9) \)

Each entry of the optimization variable \( x_{af} \) is constrained to the binary alphabet \( \{0,1\} \): we switch on or off the arcs of the STG. The conservation system \( A_{af}x_{af} = b_{tf} \) guarantees that a valid path is formed. In the objective function we must insert the \( \min\{\cdot\} \) operator in order to collect at most once the node prizes.

Problem (9) is not an ILP since the objective is not linear in the optimization variable. However, we can cast it as an ILP by introducing epigraph variables \( t = (t_v)_{v \in V} \):

\[
\begin{align*}
\text{maximize} & \quad \sum_{v \in V} p_v t_v \\
\text{subject to} & \quad t_v \leq 1, \quad \forall v \in V \\
& \quad t_v \leq s_v( x_{af} ), \quad \forall v \in V \\
& \quad A_{af}x_{af} = b_{tf} \\
& \quad x_{afj} \in \{0,1\}, \quad \forall j = 1, \ldots, |E_{af}|.
\end{align*}
\]  \( (10) \)

Problem 10 can be efficiently solved to optimality with a ILP solver, which would most certainly make use of the branch-and-bound technique, temporarily redefining the constraint on the \( x_{afj} \) entries to:

\[
0 \leq x_{afj} \leq 1
\]
III. Recursive Greedy Search

We transform the Recursive Greedy Algorithm for Walks in Directed Graphs (RGAWDG)—introduced in [8]—into a highly competitive solution method for our problem. In this section we just introduce the raw version of RGAWDG (as presented in [8]) and suggest a small implementation improvement. We achieve the major performance leap in the next section.

a) The algorithm RGAWDG: This algorithm was conceived as a sub-optimal approach to maximize some merit function \( f \) over the set of paths on a given graph. The authors assume the merit function to be monotonous and submodular in order to secure nice approximation properties for the output of RGAWDG. However, the algorithm works for generic functions, only the theoretical guarantees may not hold \( a \) priori; in any case, for our prize accumulation problem, the merit function can be shown to be modular—a special case of submodularity.

We present the RGAWDG as Algorithm 1. Its parameters are defined as follows: \( s, t \) and \( K \) indicate that we seek an \( s-t \) path of at most \( K \) arcs, i.e., \( K+1 \) nodes (allowing repetition); the parameter \( X \) indicates that we seek to find a path \( P \) that augments the set \( X \) and maximizes some merit function over \( P \); \( f_X(P) = f(V(P) \cup X) - f(X) \); the parameter \( i \) indicates the depth of the recursion; \( f(X) \) is the merit of the \( X \) node-set and \( V(P) \) is the set of nodes visited by the path \( P \).

Informally, the algorithm works as follows:
- determine the middle node \( v \) and the length \( K' \) to reach \( v \) from \( s \);
- recursively find a walk \( P_1 \) from \( s \) to \( v \) with at most length \( K' \);
- recursively find a walk \( P_2 \) from \( v \) to \( t \) with at most length \( K - K' \) to augment the nodes visited by \( P_1 \);
- output the walk obtained by the concatenation of \( P_1 \) and \( P_2 \).

On the first call, we use \( X = \{ \} \). The symbol \( P_1 \cdot P_2 \) denotes the binding of paths \( P_1 \) and \( P_2 \) to form the concatenated path \( P \). The method \( \text{PATH}(s,t) \) was implemented using a naive yet fast method using the recursive path reconstruction technique from the solution provided by the Floyd-Warshall algorithm.

b) Improved implementation: A close look at the algorithm 1 reveals an immediate improvement: before dividing the procedure to separately look for \( P_1 \) and \( P_2 \), at lines 10–11, we can include a sentinel before line 10 that detects the impossibility of finding a \( s-v-t \) path.

The sentinel avoids the undesirable effect of constructing first the \( P_1 \) path to find only that it is impossible to append to it a \( P_2 \) (the budget of steps do not allow to reach the destination). With the inclusion of that sentinel, one may find that line 2 becomes useless, due to the fact that the next level of recursion is not made if the sentinel previously detects that an \( s-v-t \) path is impossible. Despite that, we should include it, in order to test for the impossibility of finding an \( s-t \) path at the very first call of the procedure. The computation time related to that test can be considered negligible if we previously store the distances between the nodes of the graph in matrix form, indexed by the nodes themselves. This can be achieved by resorting to the Floyd-Warshall algorithm.

We present our improved version as algorithm 2. For simplicity, we refer to this version as recursive greedy search (RGS). In the presented algorithm, note that \( D(v_1, v_2) \) denotes the distance between nodes \( v_1 \) and \( v_2 \).

Algorithm 1 Recursive Greedy for Walks in Directed Graphs – Original Version

1. procedure \( \text{RG}(s,t,K,X,i) \)
2. \hspace{1em} if \( \text{DISTANCE}(s,t) > K \) then
3. \hspace{2em} return Infeasible
4. end if
5. \hspace{1em} \( P \leftarrow \text{PATH}(s,t) \)
6. \hspace{1em} Base case: \( i = 0 \). return \( P \)
7. \hspace{1em} \( m \leftarrow f_X(P) \)
8. \hspace{1em} for all \( v \in V \) do
9. \hspace{2em} for \( 1 \leq K' \leq K \) do
10. \hspace{3em} \( P_1 \leftarrow \text{RG}(s,v,K',X,i-1) \)
11. \hspace{3em} \( P_2 \leftarrow \text{RG}(v,t,K-K',X \cup V(P_1),i-1) \)
12. \hspace{3em} if \( f_X(P_1 \cdot P_2) > m \) then
13. \hspace{4em} \( P \leftarrow P_1 \cdot P_2 \)
14. \hspace{4em} \( m \leftarrow f_X(P) \)
15. \hspace{3em} end if
16. \hspace{2em} end for
17. \hspace{1em} end for
18. return \( P \)
19. end procedure

are defined as follows: \( s, t \) and \( K \) indicate that we seek an \( s-t \) path of at most \( K \) arcs, i.e., \( K+1 \) nodes (allowing repetition); the parameter \( X \) indicates that we seek to find a path \( P \) that augments the set \( X \) and maximizes some merit function over \( P \); \( f_X(P) = f(V(P) \cup X) - f(X) \); the parameter \( i \) indicates the depth of the recursion; \( f(X) \) is the merit of the \( X \) node-set and \( V(P) \) is the set of nodes visited by the path \( P \).

Informally, the algorithm works as follows:
- determine the middle node \( v \) and the length \( K' \) to reach \( v \) from \( s \);
- recursively find a walk \( P_1 \) from \( s \) to \( v \) with at most length \( K' \);
- recursively find a walk \( P_2 \) from \( v \) to \( t \) with at most length \( K - K' \) to augment the nodes visited by \( P_1 \);
- output the walk obtained by the concatenation of \( P_1 \) and \( P_2 \).

on the first call, we use \( X = \{ \} \). The symbol \( P_1 \cdot P_2 \) denotes the binding of paths \( P_1 \) and \( P_2 \) to form the concatenated path \( P \). The method \( \text{PATH}(s,t) \) was implemented using a naive yet fast method using the recursive path reconstruction technique from the solution provided by the Floyd-Warshall algorithm.

b) Improved implementation: A close look at the algorithm 1 reveals an immediate improvement: before dividing the procedure to separately look for \( P_1 \) and \( P_2 \), at lines 10–11, we can include a sentinel before line 10 that detects the impossibility of finding a \( s-v-t \) path.

The sentinel avoids the undesirable effect of constructing first the \( P_1 \) path to find only that it is impossible to append to it a \( P_2 \) (the budget of steps do not allow to reach the destination). With the inclusion of that sentinel, one may find that line 2 becomes useless, due to the fact that the next level of recursion is not made if the sentinel previously detects that an \( s-v-t \) path is impossible. Despite that, we should include it, in order to test for the impossibility of finding an \( s-t \) path at the very first call of the procedure. The computation time related to that test can be considered negligible if we previously store the distances between the nodes of the graph in matrix form, indexed by the nodes themselves. This can be achieved by resorting to the Floyd-Warshall algorithm.

We present our improved version as algorithm 2. For simplicity, we refer to this version as recursive greedy search (RGS). In the presented algorithm, note that \( D(v_1, v_2) \) denotes the distance between nodes \( v_1 \) and \( v_2 \).

Algorithm 2 Recursive Greedy for Walks in Directed Graphs – Improved Version

1. procedure \( \text{RG}(s,t,K,X,i) \)
2. \hspace{1em} if \( \text{DISTANCE}(s,t) > K \) then
3. \hspace{2em} return Infeasible
4. end if
5. \hspace{1em} \( P \leftarrow \text{PATH}(s,t) \)
6. \hspace{1em} Base case: \( i = 0 \). return \( \{ P, m \} \)
7. \hspace{1em} \( m \leftarrow f_X(P) \)
8. \hspace{1em} for all \( v \in V \) do
9. \hspace{2em} for \( 1 \leq K' \leq K \) do
10. \hspace{3em} if \( D(s,v) \leq K', D(v,t) \leq K - K' \) then
11. \hspace{4em} \( P_1 \leftarrow \text{RG}(s,v,K',X,i-1) \)
12. \hspace{4em} \( P_2 \leftarrow \text{RG}(v,t,K-K',X \cup V(P_1),i-1) \)
13. \hspace{4em} if \( f_X(P_1 \cdot P_2) > m \) then
14. \hspace{5em} \( P \leftarrow P_1 \cdot P_2 \)
15. \hspace{5em} \( m \leftarrow f_X(P) \)
16. \hspace{4em} end if
17. \hspace{3em} end if
18. \hspace{2em} end for
19. \hspace{1em} end for
20. return \( \{ P, m \} \)
21. end procedure

IV. Fortified Recursive Greedy Search

This section contains the main contribution of this paper. We boost the performance of the RGS method from the previous

3The method is naive since it ignores the prizes on the nodes.
section III: our enhancement produces a method that matches the performance of an optimal ILP solver on large graphs (100 nodes), yet it is far easier to implement and has a predictable running time (in fact, a tunable one).

Our key idea consists in embedding the RGS in the framework of fortified rollouts—a general-purpose heuristic for discrete optimization [11].

The essence of fortified rollout is as follows: given a set of variables to be assigned values (from some alphabet) and some merit function on the set of assignments, we gradually build an assignment by fixing one variable at a time. We set the first variable, then the second, and so on, until all variables are assigned values. To choose a value for a new variable, we score all possible choices via an user-provided heuristic to complete the current assignment. This is a greedy method since we optimize the remaining variables with the previous ones fixed—we do not optimize over all possible combinations of values for all the variables (thus avoiding exhaustive enumeration of the search space).

In the context of the prize accumulation problem, we have \( K \) variables to assign values to: the \( K \) nodes that the path must visit. Our alphabet is \( V \), the node set, and the user-provided heuristic is the RGS.

The details are given in Algorithm 3, where we have the following meaning for algorithm inner variables:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( obp )</td>
<td>overall best path</td>
</tr>
<tr>
<td>( obm )</td>
<td>overall best merit</td>
</tr>
<tr>
<td>( v )</td>
<td>token node</td>
</tr>
<tr>
<td>( lbm )</td>
<td>local best merit</td>
</tr>
<tr>
<td>( ldn )</td>
<td>local destination node</td>
</tr>
<tr>
<td>( a )</td>
<td>arc</td>
</tr>
<tr>
<td>( outarcs )</td>
<td>outgoing arcs</td>
</tr>
</tbody>
</table>

Note that, inside that algorithm, \( obp[j] \) means that we are accessing the node at \( j \)-th position of the overall best path, and \( obp[j]:end \) means that we are accessing a fraction of the overall best path, from position \( j \) to the last position of it.

We refer to this algorithm as the fortified recursive greedy search (FRGS).

V. OPTIMIZATION WITH SEVERAL PRIZE CATEGORIES

In this section, we briefly discuss the necessary adaptations to address a natural extension of the prize accumulation problem in which nodes hold prizes from several categories.

In this setup, a journey along nodes will accumulate prizes along several dimensions and we adopt a worst-case formulation. We seek the path, of given length, that maximizes the lowest accumulated prize across the categories.

a) Relation to sensor networks: In the context of sensor networks, this problem statement arise as soon as we need to estimate several parameters at once. For example, the network may be composed of nodes capable of measuring air pressure, temperature and humidity. Nodes exhibit different accuracies for each measurement category: the different accuracies set the prizes, as in the example from section I-A. By looking at the worst-case formulation, we try to find a path that does well across all parameters, i.e., yields a reasonable accuracy for all the parameter estimates.

A. Integer linear program formulation

We start by modifying the optimization problem in (9):

\[
\text{maximize } \min \left\{ \sum_{v \in V} p_v^{(i)} \min\{1,s_v(x_{td})\} : i \in I \right\}
\]

subject to

\[
A_{td}x_{td} = b_{td}, \quad x_{td} \in \{0,1\}, \quad \text{for all } j = 1, \ldots, |E_{td}|
\]

(11)

where is \( I \) the set of categories and \( p_v^{(i)} \) denotes the prize from category \( i \in I \) that node \( v \) holds. The linear map \( s_v(\cdot) \) was defined in (8).

By introducing an epigraph variable, we can reformulate (11) into

\[
\text{maximize } x_{sd,y}
\]

subject to

\[
y \leq \sum_{v \in V} p_v^{(i)} \min\{1,s_v(x_{td})\}, \quad \text{for } i \in I
\]

\[
A_{td}x_{td} = b_{td}, \quad x_{td} \in \{0,1\}, \quad \text{for all } j = 1, \ldots, |E_{td}|
\]

(12)

where \( y \) is the (scalar) epigraph variable.

Finally, we can get rid of the min operators via an additional set of variables \( t = \{t_v\}_{v \in V} \):

\[
\text{maximize } x_{sd,y,t}
\]

subject to

\[
y \leq \sum_{v \in V} p_v^{(i)} t_v, \quad \text{for } i \in I
\]

\[
t_v \leq s_v(x_{td}), \quad \text{for } v \in V
\]

\[
A_{td}x_{td} = b_{td}, \quad x_{td} \in \{0,1\}, \quad \text{for all } j = 1, \ldots, |E_{td}|
\]

(13)

Formulation (13) is an integer linear optimization problem.

Algorithm 3 Fortified Rollout with Recursive Greedy
procedure \( RH=RG \)

\( obp,obm \) \( \leftarrow \) RG\((s,t,K,i,X = \{\})\)

\( v \leftarrow s, B \leftarrow K, j = 1, X \leftarrow \{\} \)

while \( v \not\in t \) do

\( X \leftarrow X \cup v, B \leftarrow B - 1 \)

\( outarcs = \text{OUTGOINGARCSFROM}(v) \)

\( lbm \leftarrow 0 \)

for all \( a \in outarcs \) do

\( ldn \leftarrow NODEPOINTEDBY(a) \)

if \( lbm > obm \) then

\( obm \leftarrow lbm \)

\( bnn \leftarrow ldn \)

\( obp[j]:end \leftarrow lbp \)

else

\( bnn \leftarrow obp[j] \)

end if

end for

\( v \leftarrow bnn \)

\( j \leftarrow j + 1 \)

end while

end procedure

return \( \{obp,obm\} \)
B. Recursive greedy search

It is straightforward to extend the recursive greedy search (RGS) methods from section III to the new multi-category prize setting. We just need to modify the oracle function, i.e., the auxiliary method that computes the merit of a given sequence of nodes \( P \), given that we have already visited a set of nodes \( X \).

The output of this oracle function will now be the minimum value returned by the original oracle functions in each of the measurement categories:

\[
\text{oracle}_X(P, X, \Psi) = \min_{i \in \mathcal{I}} \{\text{oracle}_X(P, X, p^{(i)})\}
\]  

(14)

where \( i \) indexes the set \( \mathcal{I} \) of measurement categories, \( \Psi = \{p^{(1)}, \ldots, p^{(i)}, \ldots, p^{(|\mathcal{I}|)}\} \) denotes the set of all prize distributions for all categories of measurement, and \( p^{(i)} \) refers to the \( i \)-th prize distribution.

VI. IMPLEMENTATION

In order to establish a baseline in terms of performance and computational complexity, a brute-force search was implemented, so that we could compare the other algorithms in terms of performance and computational complexity. A branch-and-bound search that solves the Linear Orienteering Problem to optimality was implemented, making use of Armadillo Linear Algebra and Boost libraries and of SCIP – Solving Constraint Integer Programs solver. The presented Recursive Greedy Search and its wrapped version in Fortified Rollout technique were also implemented in the same conditions. All the presented algorithms and associated procedures were implemented in the C++ language.

VII. RESULTS

After implementation, all the algorithms were executed in the same computer, so that the comparison is as fair as possible. Two input graphs were used, containing either 20 and 100 nodes, both with a maximum degree of 4. The computation times and obtained merits are present in tables I, II, III and IV. The merits were rounded. For all tables, we explicit the meaning of the column headers:

- \( K \): The length of the path, expressed as the number of nodes.
- \( \text{Opt.} \): The optimal attained merit of the best path for a given length \( K \).
- \( \text{RG}_{i=2} \): The Recursive Greedy Search (RGS) with recursion depth equal to some value \( r \).
- \( \text{BF} \): Brute force search.
- \( \text{ILP} \): SCIP solver computing the solution of the ILP to optimality.
- \( \text{FRGS}_{i=2} \): Fortified Recursive Greedy Search, with recursion depth equal to 2.

We analyze the results first for the search over the graph with 20 nodes, and then over the graph for 100 nodes.

### Table I

<table>
<thead>
<tr>
<th>( K )</th>
<th>Opt.</th>
<th>( \text{RG}_{i=2} )</th>
<th>( \text{RG}_{i=3} )</th>
<th>( \text{RG}_{i=4} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>2372</td>
<td>2372</td>
<td>2372</td>
<td>2372</td>
</tr>
<tr>
<td>12</td>
<td>2387</td>
<td>2387</td>
<td>2387</td>
<td>2387</td>
</tr>
<tr>
<td>13</td>
<td>2401</td>
<td>2386</td>
<td>2400</td>
<td>2401</td>
</tr>
<tr>
<td>14</td>
<td>2415</td>
<td>2400</td>
<td>2401</td>
<td>2415</td>
</tr>
<tr>
<td>15</td>
<td>2427</td>
<td>2400</td>
<td>2415</td>
<td>2427</td>
</tr>
<tr>
<td>16</td>
<td>2436</td>
<td>2400</td>
<td>2427</td>
<td>2427</td>
</tr>
<tr>
<td>17</td>
<td>2448</td>
<td>2400</td>
<td>2434</td>
<td>2436</td>
</tr>
<tr>
<td>18</td>
<td>2456</td>
<td>2400</td>
<td>2436</td>
<td>2448</td>
</tr>
<tr>
<td>19</td>
<td>2462</td>
<td>2405</td>
<td>2448</td>
<td>2454</td>
</tr>
<tr>
<td>20</td>
<td>2464</td>
<td>2405</td>
<td>2448</td>
<td>2456</td>
</tr>
</tbody>
</table>

### Table II

<table>
<thead>
<tr>
<th>( K )</th>
<th>BF</th>
<th>ILP</th>
<th>( \text{RG}_{i=2} )</th>
<th>( \text{RG}_{i=3} )</th>
<th>( \text{RG}_{i=4} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>40</td>
</tr>
<tr>
<td>12</td>
<td>8</td>
<td>1</td>
<td>0</td>
<td>7</td>
<td>82</td>
</tr>
<tr>
<td>13</td>
<td>24</td>
<td>3</td>
<td>0</td>
<td>11</td>
<td>159</td>
</tr>
<tr>
<td>14</td>
<td>77</td>
<td>4</td>
<td>0</td>
<td>16</td>
<td>289</td>
</tr>
<tr>
<td>15</td>
<td>243</td>
<td>3</td>
<td>0</td>
<td>23</td>
<td>500</td>
</tr>
<tr>
<td>16</td>
<td>777</td>
<td>4</td>
<td>0</td>
<td>32</td>
<td>827</td>
</tr>
<tr>
<td>17</td>
<td>2901</td>
<td>3</td>
<td>1</td>
<td>43</td>
<td>1319</td>
</tr>
<tr>
<td>18</td>
<td>9193</td>
<td>2</td>
<td>1</td>
<td>58</td>
<td>2030</td>
</tr>
<tr>
<td>19</td>
<td>28963</td>
<td>3</td>
<td>1</td>
<td>76</td>
<td>3037</td>
</tr>
<tr>
<td>20</td>
<td>91328</td>
<td>23</td>
<td>1</td>
<td>97</td>
<td>4411</td>
</tr>
</tbody>
</table>

A. Graph with 20 nodes

The good performance of the RGS is notable: with only a two-level recursion depth it is already possible to attain near-optimal performance, while taking almost no time when compared to other search approaches. Other recursion-depths are inefficient, because the RGS takes more time that ILP while at the same time not producing the optimal solutions.

We can clearly see the expected exponential growth in computation time inherent to the brute force search, which confirms the combinatorial nature of the Prize Accumulation Problem. Also, the non-monotonous time for ILP reminds us of the intrinsic unpredictability of the heuristics that a branch-and-bound solver uses to manage its search tree. Even so, we can clearly see that ILP allows us to solve our problem to optimality while avoiding its combinatorial complexity.

Also, we can see how the complexity of RGS evolves with the increase of its recursion depth parameter, which is expected, due to a greater number of computations.

### Table III

<table>
<thead>
<tr>
<th>( K )</th>
<th>( \text{RG}_{i=2} )</th>
<th>( \text{RG}_{i=3} )</th>
<th>( \text{RG}_{i=4} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>2372</td>
<td>2372</td>
<td>2372</td>
</tr>
<tr>
<td>12</td>
<td>2387</td>
<td>2387</td>
<td>2387</td>
</tr>
<tr>
<td>13</td>
<td>2401</td>
<td>2386</td>
<td>2400</td>
</tr>
<tr>
<td>14</td>
<td>2415</td>
<td>2400</td>
<td>2401</td>
</tr>
<tr>
<td>15</td>
<td>2427</td>
<td>2400</td>
<td>2415</td>
</tr>
<tr>
<td>16</td>
<td>2436</td>
<td>2400</td>
<td>2427</td>
</tr>
<tr>
<td>17</td>
<td>2448</td>
<td>2400</td>
<td>2434</td>
</tr>
<tr>
<td>18</td>
<td>2456</td>
<td>2400</td>
<td>2436</td>
</tr>
<tr>
<td>19</td>
<td>2462</td>
<td>2405</td>
<td>2448</td>
</tr>
<tr>
<td>20</td>
<td>2464</td>
<td>2405</td>
<td>2448</td>
</tr>
</tbody>
</table>

B. Graph with 100 nodes

It is interesting to note the lower performance of RGS: the algorithm struggles to achieve a good performance for longer paths, at a two-level recursion depth. The FRGS and RG\( (i=3) \) attain very similar merits, this may indicate that both tend to take similar paths in the graph.

We can clearly see the superiority of FRGS over plain RGS: it attains very similar performance while at the same time taking much less computational time to run.
VIII. CONCLUSIONS
We explored some approaches for addressing the prize accumulation problem. We found that the two best approaches are:

- our integer linear program (ILP) formulation—from subsection II-C—which opens the door for solving the prize accumulation problem via any ILP solver. Computer simulations have shown that an ILP solver can outperform a brute-force search;

- our fortified recursive greedy search method (FRGS)—from section IV—which boosts the performance of the algorithm in [8] by wrapping it in a fortified rollout framework [9]. Computer simulations show that the FRGS performs close to the ILP solver; yet, it has a predictable running time and is much simpler to implement.

REFERENCES