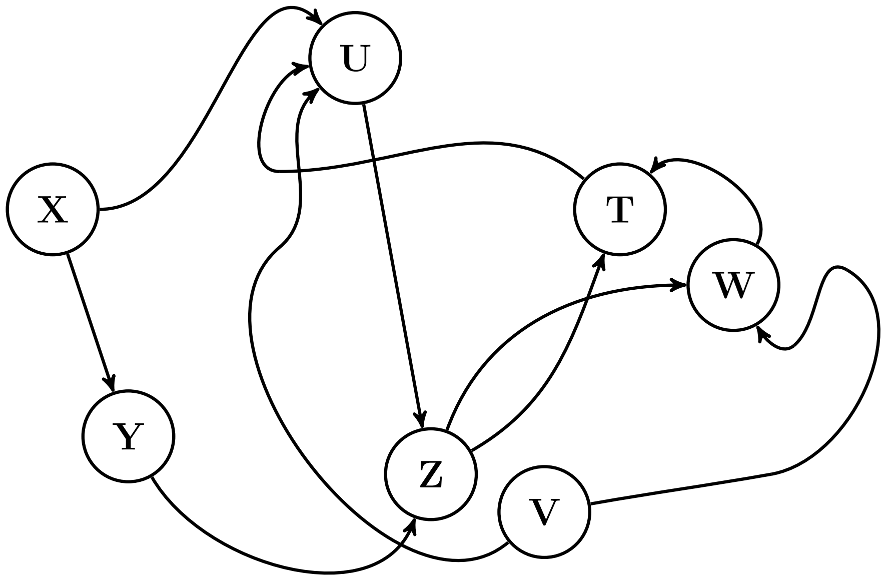
Searching for Objects in Graphs

Gabi Maduro Joris van der Meulen Matthijs Tijink

June 14, 2014



Contents

1 Introduction

Our bachelor project is about finding hidden objects in graphs as fast as possible. Almost everyone frequently searches in graphs, consider for example large networks as google or twitter. These networks can be described as graph and the node you search for is a website or a person on twitter. The everyday use is what makes our research relevant. The next example from another bachelor thesis ([8]) inspired and motivated us to work on our problem. Consider a building with a lot of rooms and hallways in which a bomb is hidden. The bomb is hidden in a certain room or hallway with a certain probability. If the building is now looked at as a graph, with its rooms and hallways being the vertices and the doors between rooms or hallways being the edges of the graph, finding the fastest way to the bomb is an example of our problem.

To make the problem easier to handle, we made some assumptions. Firstly, we limit the graph to be a tree. We do this because only a single path exists between every pair of nodes, which makes analyzing the problem easier. Furthermore, we will assume that the probability that the object is hidden in a certain node is proportional to the degree of that node. This probability proportional to the degree of nodes is used troughout the report except for section 5.In this section we use general probabilities.

In this report we first describe in section 2 the literature we found and what we searched for. Next you will find our problem statement with mathematical notation in section 3.

In our research on this problem we worked on three different aspects. The first aspect, described in section 4 and 5, describes a way to convert the problem to the travelling repairman problem and dynamic programming algorithm to find the optimal walk which needs to be taken in order to find the hidden object the quickest. We convert our original problem while the travelling repairman problem is a known and studied problem and this may help us finding good solutions.

After that a mathematical proof is given in section 6, which is the second aspect. The property we prove in this section helps us to find our hidden object faster by implementing this property in our computer program. The propery is about visiting leaf nodes.

The last aspect is described in sections 7, 8, 9 and 10. In section 7 we came up with algorithms to tackle the problem and in section 10 the results from this are shown. We want to test these algorithms on different kind of trees. In section 8 we describe how we generate trees. We generate branching process trees and preferential attachment trees which both have a power law distribution for the degrees of the nodes. This is a property often seen in real graphs. Furthermore we analyze the graphs in section 9. This is to compare our numerical results with analytical predictions.

We will finish our report with a conclusion and a discussion.

2 Research on Literature

This bachelor project is about searching in graphs, finding an object hidden in the graph as fast as possible. Since the assignment was defined in a general and broad sense, the first step in the project was then setting a better defined problem, which is more specific. This also allows us to get more useful results.

For our first literature research we got a few sources from our supervisor to inspire us for our problem statement. One of the sources was a bachelor thesis ([8]) where the problem of finding a bomb, which is described in our introduction, was given as an example. The methods in this thesis are unapplicable to our problem though. This is because the probability that a object is hidden at a certain node is equal and that is not the case we study.

We also received a presentation about quick detection of nodes with large degrees ([1]). This presentation describes how a node with largest degree could be found when not all information in the graph was known. Interesting for us is the power law distribution used in this paper. In our problem we generate trees using Preferential Attachment ([2]) and a Branching Process, such that we get graphs with node degrees following a power law distribution.

These two sources - the bachelor thesis ([8]) and the presentation about finding the node with the largest degree in a unknown graph - inspired us to do research on trees which have power law distribution and find hidden objects in such trees.

Now knowing on what problem we want to do research on we could search for scientific articles. For this we used programs as google scholar and scopus. First we came up with a search question. This is a question which hopefully gives us the sources we need. Then we filtered the most important words in this question and used google scholar or scopus to search for the articles. When we got too many hits we would also try other restrictions such as the date of publishing or a writer. In this way we looked up our literature.

Our problem is closely related to the Travelling Repairman Problem ([9]). The Travelling Repairman Problem (TRP) is the problem to find a walk which visits all nodes in a graph with minimum average arrival time. Our problem is similar, because it also finds a walk in the graph which visits all nodes and minimizes the time. In our problem we also want to minimize the time, the difference is that our problem maximizes the probability of finding an object. The source ([9]) is therefore relevant in understanding the Travelling Repairman Problem and converting our problem to it.

First we use the source [4]. It is about the TRP on a line, which makes the problem easier. It is relevant for understanding the more simple case of the TRP. When later on we can try to extend to trees or other graphs.

[9] also gives an algorithm for solving a special case of the TRP, which might be adaptable to our problem. In [3] is written about latency tours. In the article is stated that they find some improvements for the TRP, because latency tours also have similiarities to TRP. However we could not use these similiarities, while they speak about multiple paths in the TRP which we did not use.

The trees in this research are randomly generated using two methods: Preferential Attachment (as described in [2]) and a Branching Process ([6]). We needed these sources to know how to generate these trees and to get some knowledge on the trees to make conclusions on the results we get from these trees. We chose for these types of trees while they have a different structure, we want that our solutions work for not just one type of tree. After constructing trees, several heuristics are used to find a walk which solved our problem. We did not find any resources on heuristics applicable to this problem, but in chapter six of ([7]) several heuristics are described which inspired our work on finding an optimal solution to our problem. The reason again why there were no heuristics applicable to our problem is that we use the fact that the probability of an object in a node is proportional to the degree of this node. The source ([7]) is however relevant, because it showed many algorithms which we could try to change to let them work for our problem. One of our heuristics looks a lot like Katz' centrality measure, so that is why we needed the source [5].

3 Problem Statement

An object is hidden in $G = (V, E)$, a known unweighted tree (see the definition below). The node where the object is hidden is $X \in V$. The goal is to find this object as fast as possible by starting at some node and following edges.

Definition. The set of nodes V contains all nodes in the graph.

Definition. The set of edges E contains all edges in the graph. An edge $uv \in E$ represents an undirected edge from node $u \in V$ to node $v \in V$. Edge $uv \in E$ implies that $vu \in E$ too.

Definition. The tree $G = (V, E)$ is the tree with nodes V and edges E. Since G is a tree, there are no cycles in G, which implies that there is an unique path between every pair of nodes $u, v \in V$.

All we know about the object is that it stays at the same node. We assume that the probability that the object is hidden at node $v \in V$ is follows formula (1). If we reach node X, we assume that we immediately find the object. Let δ_v be the degree of node $v \in V$.

$$P(X = v) = p_v = c \cdot \delta_v \quad \text{with} \quad v \in V \tag{1}$$

To find the object, we choose a walk Γ through G which reaches all nodes. This walk, with J steps, is defined as (2):

$$\begin{aligned}
&\pi(j) && \text{node visited at step } j,\\
&\pi(j) \in V && j = 1, 2, \ldots, J,\\
&(\pi(j), \pi(j+1)) \in E && j = 1, 2, \ldots, J-1,\\
&\Gamma = (\pi(1), \pi(2), \ldots, \pi(J)).
\end{aligned} \tag{2}$$

We define the first arrival time in a node $v \in V$ as:

$$T_v(\Gamma) = \min \{n \geq 1 : \pi(n) = v\}.$$

Since the goal is to minimize the expected time to find the object, the objective function is (3):

$$\begin{aligned}
T(\Gamma) &= T_X(\Gamma),\\
\mathrm{E}[T(\Gamma)] &= \sum_{v \in V} P(X = v) T_v(\Gamma) = \sum_{v \in V} p_v T_v(\Gamma) = c \sum_{v \in V} \delta_v T_v(\Gamma).
\end{aligned} \tag{3}$$

The problem considered in this paper is as follows:

$$\begin{aligned}
\underset{\Gamma}{\text{minimize}} \quad & \mathrm{E}[T(\Gamma)]\\
\text{subject to} \quad & (2).
\end{aligned} \tag{4}$$

An optimal walk Γ^* is a walk which is a solution to this problem. In the analysis of this problem we use several definitions:

Definition. $L^0 = \{v \in V : \delta_v = 1\}$. L^0 is the set of leaf nodes (nodes with only one neighbour).

Definition. $L^1 = \{v \in V : uv \in E, u \in L^0\}$. L^1 is the set of near-leaf nodes (nodes with a leaf node as neighbour)

Definition. $N(v) = \{u \in L^0 : uv \in E\}$. $N(v)$ is the set of leaf node neighbours of $v \in V$.

Definition. $R_t(\Gamma) = \{v \in V : T_v(\Gamma) \leq t\}$. $R_t(\Gamma)$ is the set of already visited nodes at time t for walk Γ.

Definition. $d(u, v)$ is the distance between nodes $u, v \in V$.

4 The Travelling Repairman Problem

In section 2 we already mention the analogy between our problem the Travelling Repairman Problem. Now our goal is to exploit this analogy to find an optimal solution for 4 on a tree. The motivation for this approach is that the Travelling Repairman Problem is well studied, so every breakthrough in the Travelling Repairman Problem can be applied to our problem this way.

The Travelling Repairman Problem can be described as follows: We have one repairman who needs to repair a number of machines located at different points. Each edge has a time required to walk over the edge. The TRP is about finding the walk which minimizes the mean waiting time of the machines. Since our problem has weights on nodes instead of weights on edges, our problem is not a subset of the TRP.

To convert our problem to the Travelling Repairman Problem, use the following algorithm (see figure 1):

1. Copy graph $G(V,E)$ to $G'(V',E')$, renaming $v_i \in V$ to $v_{i,1}$.
2. Give all edges $e \in E(G')$ weight 1.
3. For every $v_i \in V$ with $\delta_{v_i} \geq 2$:
   Add a node $v_{i,j}$ and an edge $(v_{i,1}, v_{i,j})$ to G' with weight 0 for every $j = 2, \ldots, \delta(v_i)$.

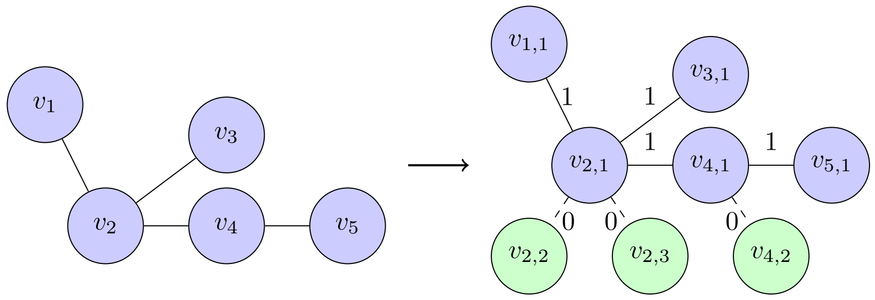


Figure 1: Converting a graph G to a Travelling Repairman Problem graph G'

To convert an optimal solution Γ' of the Travelling Repairman Problem graph G' back to the original problem, use the following steps:

1. Remove all nodes $v_{i,j}$ with $j \neq 1$ from walk Γ'. Also remove multiple sequential nodes $v_{i,1}$. This is still a valid walk (will be proved later on)
2. Rename all nodes $v_{i,1}$ back to v_i to get walk Γ in G.
3. Γ is an optimal solution to the original problem.

Example for figure 1:

$$\Gamma' = (v_{2,2}, v_{2,1}, v_{4,1}, v_{4,2}, v_{4,1}, v_{5,1}, v_{4,1}, v_{2,1}, v_{1,1}, v_{2,1}, v_{3,1}),$$
$$\Gamma = (v_2, v_4, v_5, v_4, v_2, v_1, v_2, v_3).$$

Theorem. *The walk* Γ*, as constructed using the algorithm above, is an optimal solution to (4).*

Proof. We start by proving that $T_{v_{i,j}} = T_{v_{i,k}}$, for $j,k = 1,\ldots,\delta(v_i)$.

Suppose that this statement is not true. Then there are i, j and k such that $T_{v_{i,j}} < T_{v_{i,k}}$. Let P be a shortest path from $v_{i,j}$ to $v_{i,k}$. It is trivial to see that the length of this path is 0. Let $\tilde{\Gamma}$ be the walk with PP^{-1} added after the first occurence of $v_{i,j}$. In this new walk the only change is $T_{v_{i,k}}$: $T_{v_{i,k}} = T_{v_{i,j}}$. So $\tilde{\Gamma}$ is a better solution than Γ', so Γ' is not optimal. Now we have a contradiction, so our assumption that $T_{v_{i,j}} < T_{v_{i,k}}$ is not true. Therefore the statement $T_{v_{i,j}} = T_{v_{i,k}}$, for $j,k = 1,\ldots,\delta(v_i)$ is correct.

The walk Γ' without all nodes $v_{i,j}$ with $j \neq 1$, and removing multiple sequential occurrences of $v_{i,1}$ is a valid walk, because all $v_{i,j}$ with $j \neq 1$ are all leaf nodes. All nodes $v_{i,1}$ are visited in walk Γ', so Γ is a solution for the original problem.

Every valid solution Γ in G has a corresponding solution Γ' in G', because all nodes $v_{i,j}$ with $j \neq 1$ can be reached without going through $v_{\lambda,1}$ nodes ($i \neq \lambda$). This means that any solution for G can be found using this method.

Now we prove that the walk Γ is optimal. The objective function for the TRP is (5):

$$\begin{aligned} \mathrm{E}[T'(\Gamma')] =& c\sum_{i=1}^{n}\sum_{j=1}^{\delta_{v_i}} T_{v_{i,j}}(\Gamma'), \\ =& c\sum_{i=1}^{n} \delta_{v_i} T_{v_{i,1}}(\Gamma'). \end{aligned} \tag{5}$$

This is the same objective function as the original problem, so the walk Γ is optimal in G.

□

5 A Dynamic Programming Algorithm

In this section we will describe a dynamic programming algorithm that finds the optimal walk. In [2] an algorithm is described that solves the travelling repairman problem on a graph if this graph is a line. We noticed that if we make some alterations to this algorithm, we can make it feasible for our problem as well. Provided at least that we let our graph be a special case of the tree, namely a line. To make this algorithm a little less trivial, it considers arbitrary probabilities instead of probabilities proportional to the degree of a node. After that we extend the algorithm so it works on all trees. Finally, we consider the runtime of the algorithm, in this case assuming that probabilities are proportional to the degree, in contrast to the rest of this section.

First of all, note that if the object is hidden in a certain node with a probability proportional to the degree of this node, the solution is trivial. It is then optimal to start in one end of the line and visit all the nodes one by one until we are at the other end of the line. The problem gets more interesting if the probability that the object is hidden in a certain node is indepent of the degree of this node. Let us give a more formal definition of this problem.

In the line there are n vertices. Every vertex v is given a certain value $f(v)$ (proportional to the probability that the object we are looking for is hidden in that vertex) and every line has weight 1. That is, the distance between every two vertices that are connected via an edge is 1. The line consists of vertices $v_1, v_2, \ldots, v_n$ and we choose an optimal starting node v_s with $s \in \{1, ..., n\}$. Let us represent by $[v_i, v_j]_l$, $i < j$ the fact that the leftmost location visited is v_i, the rightmost location visited is v_j and that we are currently in the leftmost location visited, i.e. v_i. Similarly, $[v_i, v_j]_r$ denotes the fact that the leftmost location visited is v_i, the rightmost location visited is v_j and that we are currently in the rightmost location visited, i.e. v_j. The initial state will be $[v_s, v_s]_l$ or $[v_s, v_s]_r$, these states are identical. If the current state is one of $\{[v_i, v_j]_l, [v_i, v_j]_r\}$, the nodes $v_x, i \leq x \leq j$ have all been visited already. The final state will be one of $\{[v_1, v_n]_l, [v_1, v_n]_r\}$.

We let $S_l(v_i) = \sum_{k=0}^{i-1} f(v_k)$ and $S_r(v_j) = \sum_{k=j+1}^{n} f(v_k)$. This is the sum of weights of unvisited nodes to the left and right of the visited nodes, respectively. Furthermore, we denote with $DP[v_i, v_j]_l$ the cost of getting from the initial state into state $[v_i, v_j]_l$ in the optimal walk. Observe that $[v_i, v_j]_l$ can only be reached from the states $[v_{i+1}, v_j]_l$ and $[v_{i+1}, v_j]_r$. Similar reasoning works on the right side, for $[v_i, v_j]_r$. That is $[v_i, v_j]_r$ can only be reached from the states $[v_i, v_{j-1}]_r$ and $[v_i, v_{j-1}]_l$. These observations lead to the following equations for computing the cost DP:

$$\begin{aligned}
DP[v_i, v_j]_l &= \min \begin{cases} S_l(v_{i+1}) + S_r(v_j) + DP[v_{i+1}, v_j]_l \\ (j-i)(S_l(v_{i+1}) + S_r(v_j)) + DP[v_{i+1}, v_j]_r \end{cases}, \\
DP[v_i, v_j]_r &= \min \begin{cases} S_l(v_i) + S_r(v_{j-1}) + DP[v_i, v_{j-1}]_r \\ (j-i)(S_l(v_i) + S_r(v_{j-1})) + DP[v_i, v_{j-1}]_l \end{cases}, \\
DP[v_s, v_s]_l &= 0, \\
DP[v_s, v_s]_r &= 0, \\
DP[v_{s+k}, v_s]_l &= \infty, \quad k > 0, \\
DP[v_{s+k}, v_s]_r &= \infty, \quad k > 0, \\
DP_{\text{optimal}} &= \min\{DP[v_1, v_n]_l, DP[v_1, v_n]_r\}.
\end{aligned}$$

The walk that will be found to obtain DP_{optimal} using the above equation is the optimal walk, i.e. the walk that needs to be taken for finding the hidden object as soon as possible. Let us clarify the above algorithm by working out a simple example. For this we will use the line in figure 2.

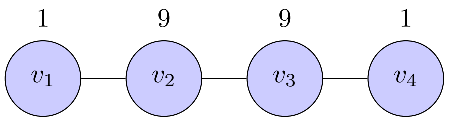


Figure 2: A line tree with values $f(v_x)$ on the nodes

We will choose one of the possible final states and look at the different ways to get to that state. If we choose final state $[v_1, v_4]_l$, we will receive figure 3.

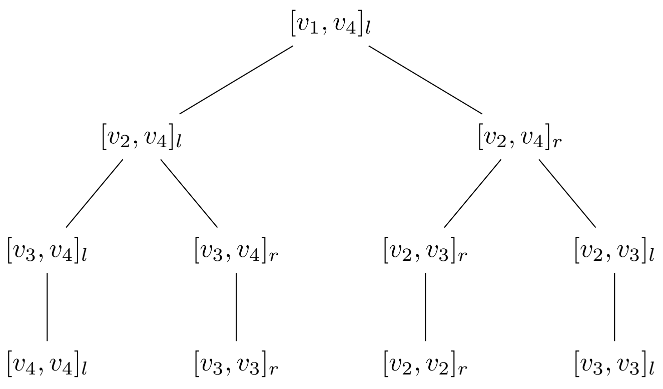


Figure 3: The different ways to get to final state $[v_1, v_4]_l$

The costs of the states in the last generation in figure 3 are all equal to zero. For the other costs in the figure, the equations are as follows:

$$DP[v_1,v_4]_l = \min\begin{cases} S_l(v_2)+S_r(v_4)+DP[v_2,v_4]_l = 1+DP[v_2,v_4]_l \\ (4-1)(S_l(v_2)+S_r(v_4))+DP[v_2,v_4]_r = 3+DP[v_2,v_4]_r \end{cases},$$
$$DP[v_2,v_4]_l = \min\begin{cases} S_l(v_3)+S_r(v_4)+DP[v_3,v_4]_l = 10+DP[v_3,v_4]_l \\ (4-2)(S_l(v_3)+S_r(v_4))+DP[v_3,v_4]_r = 20+DP[v_3,v_4]_r \end{cases},$$
$$DP[v_2,v_4]_r = \min\begin{cases} S_l(v_2)+S_r(v_3)+DP[v_2,v_3]_l = 2+DP[v_2,v_3]_r \\ (2-4)(S_l(v_2)+S_r(v_3))+DP[v_2,v_3]_r = 4+DP[v_2,v_3]_l \end{cases},$$
$$DP[v_3,v_4]_l = \min\begin{cases} S_l(v_4)+S_r(v_4)+DP[v_4,v_4]_l = 19+DP[v_4,v_4]_l = 19 \\ (3-4)(S_l(v_4)+S_r(v_4))+DP[v_4,v_4]_r = 19+DP[v_4,v_4]_r = 19 \end{cases},$$
$$DP[v_3,v_4]_r = \min\begin{cases} S_l(v_3)+S_r(v_3)+DP[v_3,v_3]_l = 11+DP[v_4,v_4]_l = 11 \\ (3-4)(S_l(v_4)+S_r(v_4))+DP[v_4,v_4]_r = 11+DP[v_4,v_4]_r = 11 \end{cases},$$
$$DP[v_2,v_3]_r = \min\begin{cases} S_l(v_2)+S_r(v_2)+DP[v_2,v_2]_l = 11+DP[v_2,v_2]_r = 11 \\ (2-3)(S_l(v_2)+S_r(v_2))+DP[v_2,v_2]_r = 11+DP[v_2,v_2]_l = 11 \end{cases},$$
$$DP[v_2,v_3]_l = \min\begin{cases} S_l(v_3)+S_r(v_3)+DP[v_3,v_3]_l = 2+DP[v_3,v_3]_l = 2 \\ (2-3)(S_l(v_3)+S_r(v_3))+DP[v_3,v_3]_r = 2+DP[v_3,v_3]_r = 2 \end{cases}.$$

Now we can fill in the costs beloning to the states in figure 3, starting with the initial states and working our way up. This yields us figure 4:

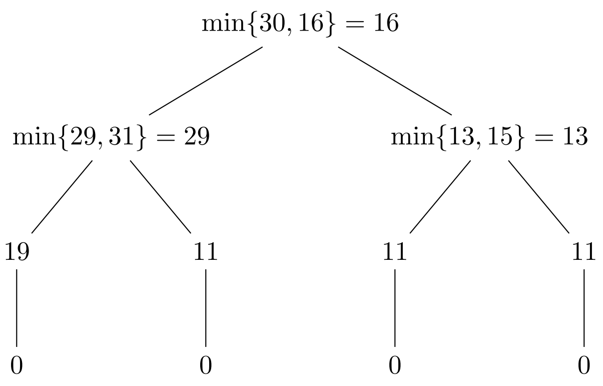


Figure 4: Calculating the cost of the optimal walk

Looking at figure 4, we see that the optimal walk in the line has value 16. Comparing figure 4 with figure 3 we see that the optimal walk uses v_2 as starting node, then visits node v_3 and v_4 and ends in node v_1.

Normally it would now have been necessary to look at the other possible final state, $[v_1, v_4]_r$, and do something similar. However, since the line in this example is symmetrical, this is not necessary. Because of the symmetry we know that there is one other optimal walk; the walk that uses v_3 as starting node, then visits node v_2 and v_1 and ends in node v_4.

The algorithm used for the line has complexity $\mathcal{O}(n^2)$. It can be transformed to make it feasible for trees as well, for this we will let R be the set of all visited nodes and introduce the following definitions:

Definition. $W_R = \{w \in R : \exists_{v \in V \setminus R}\, d(v,w) = 1\}$. W_R is the set of all visited nodes that are neighbouring a not yet visited node.

Definition. $\Theta_S = \sum_{v \in S} p_v$, $X \in V$. Θ_S is the total value of all nodes in any subset S of V.

Now we are ready to formulate the equations for the algorithm feasible for trees:

$$DP[R]_v = \begin{cases} \min_{w \in W_R}\{d(v,w)\Theta_{V \setminus R} + DP[R \setminus \{v\}]_w\} & \text{if} \quad |R| > 1 \\ 0 & \text{if} \quad |R| = 1 \\ \infty & \text{if} \quad |R| < 1 \end{cases} \tag{6}$$

$$DP_{\text{optimal}} = \min_{l \in L^0}\{DP[V \setminus \{l\}]_l\} \tag{7}$$

As was the case with the line algorithm, the walk that will be found to obtain the minimal value DP_{optimal} is the optimal walk. In order to find the order of complexity of this algorithm, we will look at $|W_R|$, the size of the set of all visited nodes that are neigbouring a not yet visited node, at all time steps. We will inspect the worst-case scenario for this algorithm, which is a star tree. This is the worst-case scenario because the number of possible sets R and the amount of choices W_R as large as possible at all times. An example of such a tree can be seen in figure 5. By inspecting this worst-case scenario we will find that the dynamic programming algorithm has complexity $\mathcal{O}\big((n-1)!\big)$. This is because in a star tree at the first time step, $|W_R|$ will be 1, but at the second time step it will be $n-2$, followed by $n-3$ at the third time step, $n-4$ at the third time step and so on.

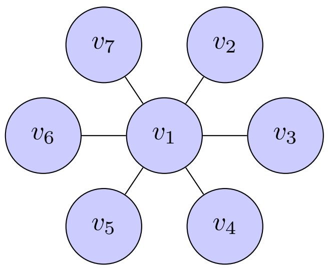


Figure 5: Example of a star tree

$\mathcal{O}\big((n-1)!)\big)$ is the same order of complexity as any non-dynamic programming algorithm for finding the optimal walk will have. However, although they have the same order of complexity, we expect that the dynamic programming algorithm will be faster but on the other hand will use more memory, since all values $DP[R]_v$ have to be remembered.

In contrast what we previously assumed in this section, we will assume that the probabilities are distributed proportionally to the degree of the nodes from now on. If the theory of combining leaf nodes, as will be explained and proved in section 6, is applied (which requires proportional probablities), the star is trivial in the algorithm. So we will consider the now worst-case scenario, which is an extended star tree. This will result in a much smaller complexity. An example of an extended star tree can be seen in figure 6,

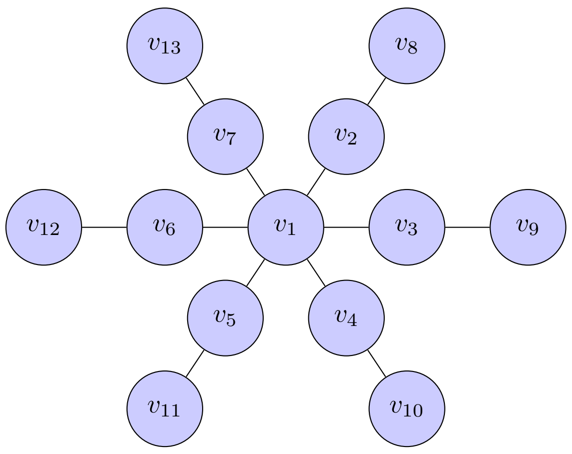


Figure 6: Example of an extanded star tree

At both the first and second time step, $|W_R|$ will be 1, assuming we start at a leaf node, and at both the third and fourth time step it will be $\left\lfloor \frac{n}{2} \right\rfloor - 1$. At the fifth and sixth time step $|W_R|$ will be $\left\lfloor \frac{n}{2} \right\rfloor - 2$, at the seventh and eigth time step $|W_R|$ will be $\left\lfloor \frac{n}{2} \right\rfloor - 3$ and so on. This will result in a complexity of order $\mathcal{O}\left(\left(\frac{n-1}{2}!\right)^2\right)$.

6 Visiting Leaf Nodes

In this section we want to prove a theorem to improve the searching time for a hidden object in a tree.

Theorem. *If an optimal walk visits a leaf node at a certain time, except as initial node, then it is optimal to proceed by visiting all unvisited leaf nodes with the same ancestor before visiting any other nodes.*

Proof. We will prove this theorem using mathematical induction. The first step is proving that after visiting a leaf node, it is (also) optimal to visit another leaf node with the same ancestor directly.

Let Γ be an optimal solution of 4. Now take any time t such that $\pi(t) \in L^1$, $|N(\pi(t)) \setminus R_t(\Gamma)| \geq 2$ and $\pi(t+1) \in N(\pi(t))$. This means that we visit a leaf node at time $t+1$ with at least one leaf node with the same ancestor still unvisited. Since $\pi(t+1)$ is a leaf node, we have $\pi(t) = \pi(t+2)$.

Suppose $\pi(t+3) \notin N(\pi(t))$. Since Γ has to contain $v \in N(\pi(t))$ after time $t+1$, at a certain time $t+s$ we must visit $\pi(t)$ and its unvisited leaf node. Let P be the walk of $s-3$ timesteps after $\pi(t+2)$, with P not empty. We choose the smallest s such that $\pi(t+s) = \pi(t)$ and $\pi(t+s+1)$ is a leaf node $v \in N(\pi(t))$. It is trivial to show that v is unvisited, because the walk is optimal. Let Q be the walk which starts 3 timesteps after P. Q either is empty or begins with $\pi(t)$, since the last node visited before Q is a leaf node neighbouring $\pi(t)$. This gives Γ the following structure:

$$\Gamma = (\pi(1), \pi(2), \ldots, \pi(t), \pi(t+1), \pi(t), P, \pi(t), \pi(t+s+1), Q).$$

To prove the theorem, we need to be able to know something about $\mathrm{E}[T(\Gamma)]$:

$$\begin{aligned}
\mathrm{E}[T(\Gamma)] &= \sum\nolimits_{v\in V} p_v T_v(\Gamma) \\
&= \sum\nolimits_{v\in R_t(\Gamma)} p_v T_v(\Gamma) \\
&+ p_{\pi(t+1)} T_{\pi(t+1)}(\Gamma) \\
&+ \sum\nolimits_{v\in (P\setminus R_t(\Gamma))} p_v T_v(\Gamma) \\
&+ p_{\pi(t+s+1)} T_{\pi(t+s+1)}(\Gamma) \\
&+ \sum\nolimits_{v\in (Q\setminus R_{t+s+2}(\Gamma))} p_v T_v(\Gamma)
\end{aligned} \tag{8}$$

Now we construct $\tilde{\Gamma}$ as follows:

$$\tilde{\Gamma} = (\pi(1), \pi(2), \ldots, \pi(t), P, \pi(t), \pi(t+1), \pi(t), \pi(t+s+1), Q)$$

The only changes between Γ and $\tilde{\Gamma}$ is the position of P. P has been moved two timesteps earlier, and the two nodes visited before P in Γ have been moved $s-3$ timesteps later. All nodes in $\tilde{\Gamma}$ have been denoted using the $\pi(i)$'s of Γ. This allows $\mathrm{E}[T(\tilde{\Gamma})]$ to be calculated in terms of Γ.

$$
\begin{aligned}
E[T(\tilde{\Gamma})] &= \sum\nolimits_{v \in V} p_v T_v(\tilde{\Gamma}) \\
&= \sum\nolimits_{v \in R_t(\Gamma)} p_v T_v(\Gamma) \\
&+ p_{\pi(t+1)} (T_{\pi(t+1)}(\Gamma) + s - 3) \\
&+ \sum\nolimits_{v \in (P \setminus R_t(\Gamma))} p_v (T_v(\Gamma) - 2) \\
&+ p_{\pi(t+s+1)} T_{\pi(t+s+1)}(\Gamma) \\
&+ \sum\nolimits_{v \in (Q \setminus R_{t+s+2}(\Gamma))} p_v T_v(\Gamma)
\end{aligned} \tag{9}
$$

Since Γ is optimal, $\mathrm{E}[T(\Gamma)] \leq \mathrm{E}[T(\tilde{\Gamma})]$ should hold:

$$\mathrm{E}[T(\Gamma)] \leq \mathrm{E}[T(\tilde{\Gamma})].$$

Substituting (8) and (9) we get

$$
\begin{pmatrix}
\sum\nolimits_{v \in R_t(\Gamma)} p_v T_v(\Gamma) \\
+ p_{\pi(t+1)} T_{\pi(t+1)}(\Gamma) \\
+ \sum\nolimits_{v \in (P \setminus R_t(\Gamma))} p_v T_v(\Gamma) \\
+ p_{\pi(t+s+1)} T_{\pi(t+s+1)}(\Gamma) \\
+ \sum\nolimits_{v \in (Q \setminus R_{t+s+2}(\Gamma))} p_v T_v(\Gamma)
\end{pmatrix}
\leq
\begin{pmatrix}
\sum\nolimits_{v \in R_t(\Gamma)} p_v T_v(\Gamma) \\
+ p_{\pi(t+1)} (T_{\pi(t+1)}(\Gamma) + s - 3) \\
+ \sum\nolimits_{v \in (P \setminus R_t(\Gamma))} p_v (T_v(\Gamma) - 2) \\
+ p_{\pi(t+s+1)} T_{\pi(t+s+1)}(\Gamma) \\
+ \sum\nolimits_{v \in (Q \setminus R_{t+s+2}(\Gamma))} p_v T_v(\Gamma)
\end{pmatrix}.
$$

It follows that

$$
\begin{pmatrix}
p_{\pi(t+1)} T_{\pi(t+1)}(\Gamma) \\
+ \sum\nolimits_{v \in (P \setminus R_t(\Gamma))} p_v T_v(\Gamma)
\end{pmatrix}
\leq
\begin{pmatrix}
p_{\pi(t+1)} (T_{\pi(t+1)}(\Gamma) + s - 3) \\
+ \sum\nolimits_{v \in (P \setminus R_t(\Gamma))} p_v (T_v(\Gamma) - 2)
\end{pmatrix},
$$

which reduces to

$$2 \sum\nolimits_{v \in (P \setminus R_t(\Gamma))} p_v \leq (s-3) p_{\pi(t+1)}. \tag{10}$$

Now we construct walk W:

$$W = (\pi(1), \pi(2), \ldots, \pi(t), \pi(t+1), \pi(t), \pi(t+s+1), \pi(t), P, Q).$$

Note that we still use the names for the nodes given in Γ. Note that W is a valid walk, even though the last node before Q is not a leaf node of $\pi(t)$. This is because Q is either empty or starts with $\pi(t)$. As can be seen in the structure of Γ, P ends with a node adjacent to $\pi(t)$. We calculate $\mathrm{E}[T(W)]$ in terms of Γ:

$$\begin{aligned}
\mathrm{E}[T(W)] &= \sum\nolimits_{v\in V} p_v T_v(W) \\
&= \sum\nolimits_{v\in R_t(\Gamma)} p_v T_v(\Gamma) \\
&\quad + p_{\pi(t+1)} T_{\pi(t+1)}(\Gamma) \\
&\quad + \sum\nolimits_{v\in(P\setminus R_t(\Gamma))} p_v(T_v(\Gamma)+2) \quad . \\
&\quad + p_{\pi(t+s+1)}(T_{\pi(t+s+1)}(\Gamma) - s + 3) \\
&\quad + \sum\nolimits_{v\in(Q\setminus R_{t+s+2}(\Gamma))} p_v T_v(\Gamma)
\end{aligned}$$

Comparing $\mathrm{E}[T(W)]$ and $\mathrm{E}[T(\Gamma)]$, we obtain:

$$\begin{aligned}
&\mathrm{E}[T(W)] - \mathrm{E}[T(\Gamma)] = \\
&\begin{pmatrix} \sum_{v\in R_t(\Gamma)} p_v T_v(\Gamma) \\ + p_{\pi(t+1)} T_{\pi(t+1)}(\Gamma) \\ + \sum_{v\in(P\setminus R_t(\Gamma))} p_v(T_v(\Gamma)+2) \\ + p_{\pi(t+s+1)}(T_{\pi(t+s+1)}(\Gamma) - s + 3) \\ + \sum_{v\in(Q\setminus R_{t+s+2}(\Gamma))} p_v T_v(\Gamma) \end{pmatrix} - \begin{pmatrix} \sum_{v\in R_t(\Gamma)} p_v T_v(\Gamma) \\ + p_{\pi(t+1)} T_{\pi(t+1)}(\Gamma) \\ + \sum_{v\in(P\setminus R_t(\Gamma))} p_v T_v(\Gamma) \\ + p_{\pi(t+s+1)} T_{\pi(t+s+1)}(\Gamma) \\ + \sum_{v\in(Q\setminus R_{t+s+2}(\Gamma))} p_v T_v(\Gamma) \end{pmatrix} = \\
&\begin{pmatrix} \sum_{v\in(P\setminus R_t(\Gamma))} p_v(T_v(\Gamma)+2) \\ + p_{\pi(t+s+1)}(T_{\pi(t+s+1)}(\Gamma) - s + 3) \end{pmatrix} - \begin{pmatrix} \sum_{v\in(P\setminus R_t(\Gamma))} p_v T_v(\Gamma) \\ + p_{\pi(t+s+1)} T_{\pi(t+s+1)}(\Gamma) \end{pmatrix} = \\
&2\sum\nolimits_{v\in(P\setminus R_t(\Gamma))} p_v - (s-3)p_{\pi(t+s+1)}.
\end{aligned}$$

Since $\pi(t+1)$ and $\pi(t+s+1)$ both are nodes with degree one, we have $p_{\pi(t+1)} = p_{\pi(t+s+1)}$ and we can use inequality (10) to simplify this equation:

$$\begin{aligned}
\mathrm{E}[T(W)] - \mathrm{E}[T(\Gamma)] &= \\
2\sum\nolimits_{v\in(P\setminus R_t(\Gamma))} p_v - (s-3)p_{\pi(t+s+1)} &\le \\
0.
\end{aligned}$$

Hence:

$$\mathrm{E}[T(W)] \le \mathrm{E}[T(\Gamma)].$$

The assumption that $\pi(t+3) \notin N(\pi(t))$ can be true or false:

- If it is true, $\mathrm{E}[T(W)] = \mathrm{E}[T(\Gamma)]$ which means the walk W is optimal too.
- If it is false, the optimal walk has $\pi(t+3) \in N(\pi(t))$.

In both cases there is an optimal walk such that $\pi(t+3) \in N(\pi(t))$ for that walk. This concludes this first part of the proof, the initial condition part of mathematical induction. The rest of the proof assumes optimal walks with this property.

For the induction step of the proof, take an optimal walk Γ such that $|N(\pi(t))| = n \geq 3$. Now assume that the optimal walk looks like this:

$$\Gamma = (\pi(1), \ldots, \pi(t), \pi(t+1), \ldots, \pi(t+2k-3), \pi(t), \ldots).$$

with $\pi(t+1), \pi(t+3), \ldots, \pi(t+2k-3) \in N(\pi(t))$ and $\pi(x) \notin N(\pi(t))$ for $x \leq t$. That means $k-1$ leaf nodes have been visited. Now take $\tilde{t} = t + 2k - 4$. Notice that $\pi(\tilde{t}) = \pi(t)$ so $\pi(\tilde{t}) \in L^1$. $\left|N(\pi(\tilde{t})) \setminus R_{\tilde{t}}(\Gamma)\right| = 2 \geq 2$. All conditions for the first part of the proof are fulfilled, so $\pi(t+2k-1) \in N(\pi(t))$. Then Γ is expressed as follows:

$$\Gamma = (\pi(1), \ldots, \pi(t), \pi(t+1), \ldots, \pi(t+2k-3), \pi(t), \pi(t+2k-1), \pi(t), \ldots).$$

Using mathematical induction, Γ has this form for any n, meaning that all leaf nodes with the same ancestor are visited at the same time. $\square$

7 Numerical Solutions

7.1 Optimal solution

The optimal solution for our problem is hard to find. Here we outline the algorithm we used to find the optimal solution. The basis for the algorithm is a brute force search for the best walk.

7.1.1 Brute force search

The algorithm tries every possible walk, by iterating over all walks recursively. The $\mathrm{E}[T(\Gamma)]$ of this walk can be found using formula 3. Since every walk is tried, the optimal solution will be found.

To make this algorithm faster, a several strategies have been used. First of all we estimate the $\mathrm{E}[T(\Gamma)]$ of the complete walk with the current walk prefix, using a so-called admissible heuristic. This admissible heuristic (explained in section 7.1.2) makes an underestimation of the value of all walks with the current walk prefix. If this value is more than the value of the currently best known walk, all walks with the current walk prefix are worse than that walk. This allows the algorithm to skip evaluation of a number of walks

Secondly, the algorithm exploits isomorphism in the graph by skipping any node of degree one if another node of degree one with the same ancestor node has already been tried. This would give the same optimal solution, so they can safely be skipped.

Finally, the initial best value of a walk can be chosen by using a heuristic to find a solution to the problem (as described in section 7.2). This gives a reasonably good value, but might not be optimal. This allows even more non-optimal walks to be skipped.

7.1.2 Admissible heuristic

Admissible heuristics are useful for stopping a search early on if it is not going to be better than the current best solution. This is possible because an admissible heuristic is an underestimate of the value, so if the admissible heuristic is worse than the current best solution, any solution that can be reached from the current point is also worse.

In the algorithm for calculating the walk, we used two different lower bounds: A_1 and A_2.

A_1 The sum $\Delta = \sum_{v \in V \setminus R_t(\Gamma)} \delta_v$ is used for calculating the value of the remaining nodes. All nodes still left to visit cannot be reached any sooner than the next time step. That means that $A_1 = \Delta(t+1)$.

A_2 The number of remaining nodes k cannot all be reached in one time step, it takes k more steps to visit them all. Assuming they all have degree one, this yields the following heuristic:
$A_2 = (t+1) + (t+2) + \ldots + (t+k) = k(t + \frac{k+1}{2}) = \frac{1}{2}k^2 + \frac{1}{2}k + kt$.

7.2 Heuristics

We have tried several heuristics, which are explained in the following sections. The first four heuristics (7.2.1 until and including 7.2.4) choose a node to visit, walk the path towards that node (also visiting all nodes in between while walking the path) and then choose the node to visit next.

7.2.1 Largest degree

This heuristic begins at a node with the largest value p_v. The next node to visit is a node in the set $H_{7.2.1}(v)$, where v is the node at time t and $\pi(t+1)$ is chosen such that $\pi(t+1) \in H_{7.2.1}(v)$ (see (11)).

$$\begin{aligned} C &= \operatorname*{arg\,max}_{w \in V \setminus R_t(\Gamma)} p_w \\ H_{7.2.1}(v) &= \operatorname*{arg\,min}_{w \in C} d(v, w) \end{aligned} \tag{11}$$

In words this heuristic means that we visit the node with the largest degree then the node which has second largest degree and all nodes which are on the path between these two nodes etcetera. If two nodes have the same degree we choose to go to the node that has the smallest distance from the node we are in now. We chose this heuristic because the object has a larger chance of being hidden at a node with a large degree, so if we visit those nodes early on, we might be finding it quite fast.

7.2.2 Degree divided by distance

This heuristic also begins at a node with the largest value p_v. The next node to visit is a node in the set $H_{7.2.2}(v)$, where v is the node at time t and $\pi(t+1)$ is chosen such that $\pi(t+1) \in H_{7.2.2}(v)$ (see (12)).

$$H_{7.2.2}(v) = \operatorname*{arg\,max}_{w \in V \setminus R_t(\Gamma)} \frac{p_w}{d^\alpha(v, w)} \tag{12}$$

The idea behind this heuristic is that although it has advantages to go to nodes with a large degree early on, it also has disadvantages. The previous heuristic did not take the distance between nodes into account (except for the tie-breaking), this heuristic prefers close nodes with a relatively large degree over far away nodes with a larger degree. We have used several powers of α: 0.5, 1, 2 and 3.

7.2.3 Summed degree divided by distance

This heuristic also begins at a node with the largest value p_v. The next node to visit is a node in the set $H_{7.2.3}(v)$, where v is the node at time t and $\pi(t+1)$ is chosen such that $\pi(t+1) \in H_{7.2.3}(v)$ (see (13)).

$$H_{7.2.3}(v) = \operatorname*{arg\,max}_{w \in V \setminus R_t(\Gamma)} \frac{S(w)}{d^\alpha(v, w)} \tag{13}$$

In this formula, $S(w)$ denotes the sum of the degrees of the nodes that get visited while walking from node v to node w. If the set of nodes that get visited while walking from node v to node w is denoted by P, then $S(w) = \sum_{s \in P} p_s$. This heuristic takes into account, as opposed to the two heuristics above, also at the nodes visited along the way when choosing a new node to visit. We have used several powers of α: 0.5, 1, 2 and 3.

7.2.4 Importance of nearby nodes

This heuristic also begins at a node with the largest value p_v. The next node to visit is a node in the set $H_{7.2.4}(v)$, where v is the node at time t and $\pi(t+1)$ is chosen such that $\pi(t+1) \in H_{7.2.4}(v)$ (see (14)).

$$\begin{aligned} f(w) &= \sum_{u \in V \setminus R_t(\Gamma)} \delta_u \cdot \beta^{d(u,w)} \\ H_{7.2.4}(v) &= \operatorname*{arg\,max}_{w \in V \setminus R_t(\Gamma)} \beta^{d(v,w)} \cdot f(w) \end{aligned} \tag{14}$$

This heuristic assigns a value to every node. A higher value indicates a node with more nodes with large degree nearby. The factor $\beta \in (0,1)$ makes sure that nodes which are further away contribute less towards the value assigned to the node. Finally the assigned value is corrected for the distance to the node at time t. We have used $\beta = 0.2$ in our experiments.

This heuristic has similarities to Katz' centrality measure (see [5]), which also assigns a value to each node and also corrects for distance by applying a factor with the distance as exponent. Yet this heuristic has several key differences to fit our problem in a better way.

Katz' centrality measure lets each node contribute multiple times to the assigned value by counting all walks which exist between the nodes, so back-and-forth walks are counted too. This heuristic does not do this. Furthermore, Katz' centrality measure does not have a concept of already visited nodes, so it naturally uses all nodes, which does not make sense for this heuristic.

Finally, since Katz' centrality measure does not have a concept of a current position, it does not correct for the distance to the current node, which this heuristic does.

7.2.5 Minimal total time

This heuristic finds a closed walk which minimizes the total time for visiting all nodes. This is trivial to compute, since it simply means all edges have to be visited exactly twice. We expect that this heuristic will not be very good, since it does not prioritize nodes with large degrees, but it will provide an easy to compute upper bound to $\mathrm{E}[T(\Gamma^*)]$.

8 Generating Trees

For the sake of testing the heuristics, we are going to need to be able to generate trees. We do this in two different ways, using a branching process and using preferential attachment model. Both models are described and explained in the sections below.

8.1 Branching Processes

We can generate trees by using a branching process, as described in [6] A branching process can be seen as a population consisting of individuals able to produce offspring of the same kind. In order to create a tree this way, we let the population start with 1 individual, r, and let every individual produce j new offspring with probabilty $p_j, j \geq 1$. The number of offspring produced by each individual are i.i.d.; indepent of each other and identically distributed. The individual in the zeroth generation, r, will produce individuals in the first generation, $r\_0, r\_1, ...,$ which will produce individuals in the second generation, $r\_0\_0, r\_0\_1, ..., r\_1\_0, r\_1\_1, ..., ...,$ and so on.

The expected number of nodes in generation n will be μ^n, where μ is the expected number of offspring produced by each individual. If the process is stopped after n generations, the expected number of nodes in the tree will be $\sum_{i=0}^{n} \mu^n$. In these formulas μ denotes $\sum_{j=1}^{\infty} jp_j$.

The distribution used in the branching process is the Zeta distribution (a discrete power law distribution) with parameter 2.6, note that this yields a different distribution in degrees for the preferential attachment model. We The distribution function for the Zeta distribution is $f_s(k) = \frac{k^{-s}}{\zeta(s)}$, where $\zeta(s) = \sum_{k=1}^{\infty} k^{-s}$ is the Riemann zeta function. An example tree is shown in figure 7.

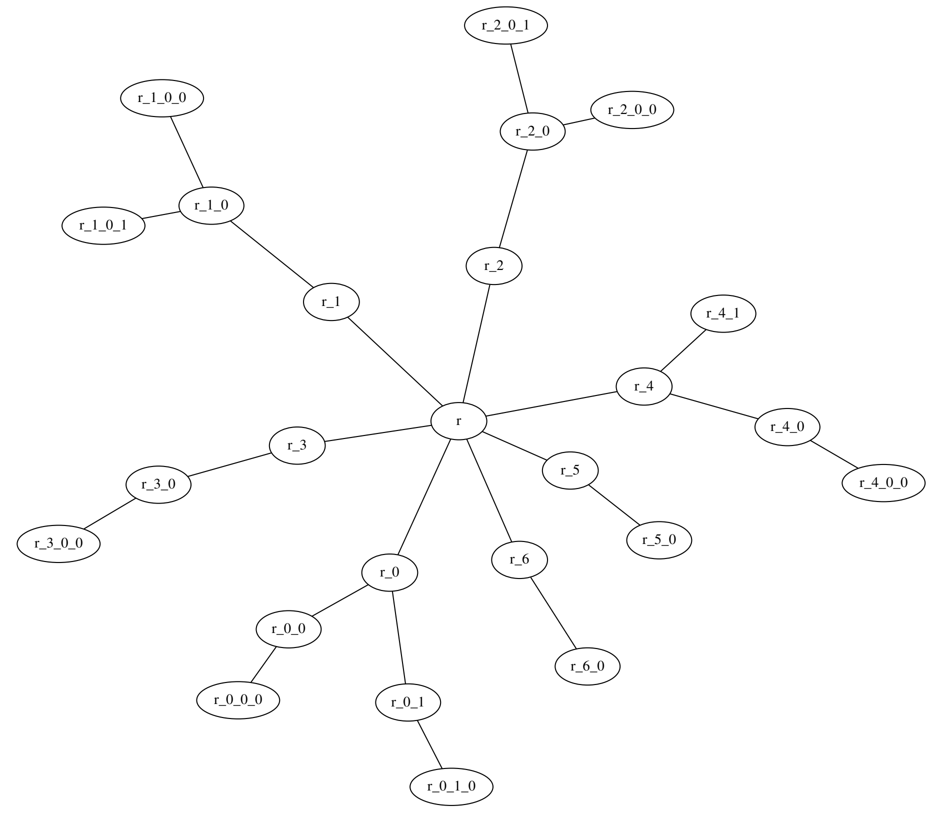


Figure 7: A tree generated by a Branching Process with 25 nodes

8.2 Preferential Attachment

Another method we use to generate trees, is by using preferential attachment model ([2]). Here we also start with just one node, and then start adding other nodes one by one that will be attached to one of the already existing nodes. The probability that the newly added node will be attached to a certain already existing node is proportional to the degree of that already existing node. Except the probability that a new node will attach to the starting node is proportional to the degree plus one. This is in order to get the process started in the first place. So if there are n already existing nodes in the tree, with a total degree of $\sum_{v \in V} \delta(v) = 2n - 2$ and therefore a total weight of $\sum_{v \in V} w(v) = \sum_{v \in V} \delta_v + 1 = 2n - 2 + 1 = 2n - 1$, then a newly added node will be attached to node v that with probability $\frac{\delta_v + \mathbb{1}\{v \text{ is initial node}\}}{2n-1}$.

This way a tree will be constructed that has nodes with high degrees in the first generations, whereas in the trees constructed by using a branching process, nodes with high degree can occur at any generation. An example tree can be seen in figure 8.

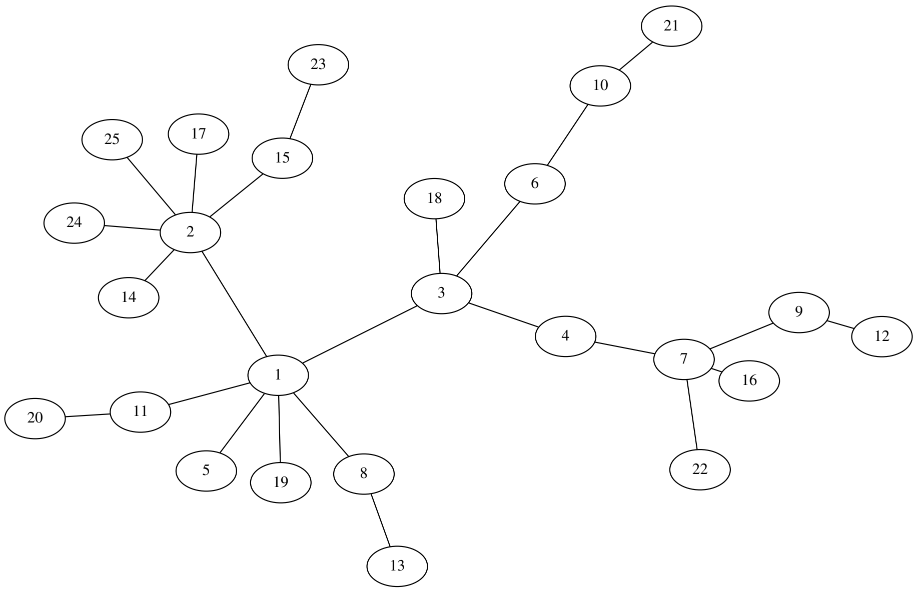


Figure 8: A tree generated by Preferential Attachment with 25 nodes
(Nodes are numbered in the order of their arrival.)

9 Predicting the Performance of Heuristics

All our heuristics are based on visiting nodes with large degrees first, because at a node with high degree the probability of finding the hidden object is the highest. The heuristics we use are in most cases not optimal, especially not in large trees, and results will be different with each heuristic. We want to predict which heuristic is better by looking at the structure of the graphs, but also by looking at the performance of the heuristics.

With this performance we mean: with what probability we will find the hidden object when visiting the nodes with large degrees. We can do this because we know with which distribution the branching trees are generated and we know how the preferential attachment trees are generated. With the quality of the heuristics we want to test the idea whether visiting nodes with large degrees is a good idea, while in our graphs a lot of nodes have degree one, so a small degree.

In the branching process trees the nodes with large degree can be anywhere because every node has the same probability of getting many children. To determine the diameter of the graph, we need to look at the number of generations. If we generate a tree of $20,000$ node, we solve the following equation:

$$\sum_{i=0}^{n} \mu^i = 20,000.$$

This is because the expected number of nodes in a generation is μ^n, with μ the expected number of children of any node and n the generation. In our case the number of expected children for every node μ is $1,7509$. Choosing $n = 16$ we get an expected number of 18201 nodes and for $n = 17$ we get an expected number of 31048 nodes. This means that for a tree of 20.000 nodes the expected number of generations is 17. So the largest distance between two nodes is about 34.

To this end, we use a heuristic argument based on average generation sizes. This will not give an average depth of the tree, but it is a reasonable approximation.

Furthermore we can say something about the number of nodes with degree one. In the last generation the nodes all have degree one, calling this last generation the n^{th} generation we can say there is an expected number of μ^n nodes in this last generation. If we look at the fraction of nodes in the last generation we can express this as follows:

$$\mu^n = 1 + (\mu - 1) \sum_{i=0}^{n-1} \mu^i.$$

The result follows from the geometric series. This means there are almost $\mu - 1$ times more nodes in the last generation than all other nodes. With μ is 1.7509, this means that around $\frac{0.7509}{1.7509}$, which is 42 percent of the nodes, has degree one.

It is more difficult to say something like this about the preferential attachment graph. We do know that nodes with a higher degree have a high probablity of getting connected to a new node. Therefore we expect nodes with large degrees to be near each other. Because these nodes are more likely to be near to each other the number of nodes of degree one will be large. While adding a new node to the tree, meaning that this node gets degree one, it is much more likely that the next new node will connect to a node with a higher degree than one so that this node also gets degree one. Because these nodes with large degrees have smaller distances to each other we expect our heuristics to do better on preferential attachment graphs than on branching trees.

We know that many nodes in our generated trees have nodes with degree one. The question is how profitable it is to visit the nodes with large degree. In the figure beneath you can see how the probability grows of finding the object when we visit the nodes in order with highest degree first and lowest degree at last. This is optimistic of course while it suggests that it is possible to jump from the node with largest degree to the node with the second largest degree to the node with the third highest degree etcetera.

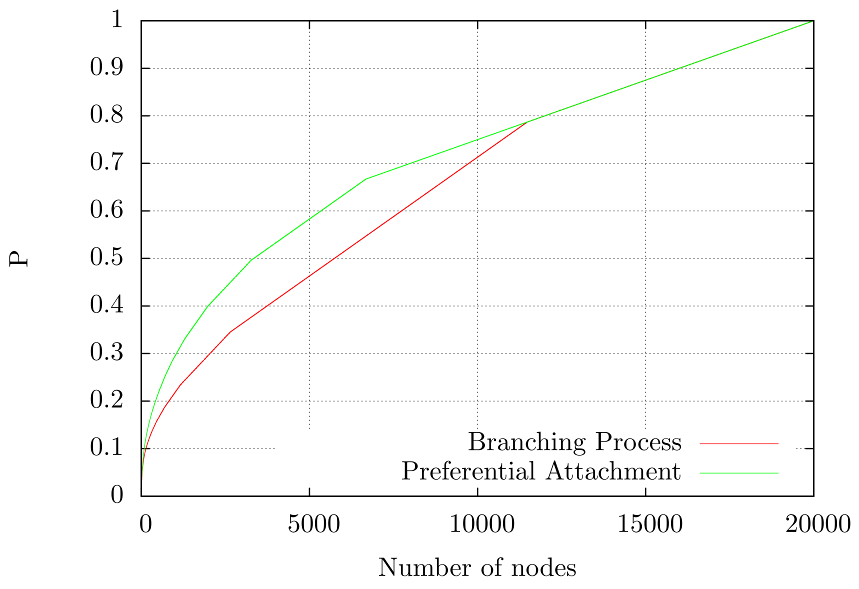


Figure 9: Number of nodes (sorted on degree from high to low) and probability the object is hidden there

Observing this graph we can see that especially in the beginning the probability grows quickly. With a probability of almost 0.3 we find the object in the preferential attachment graph by visiting only five percent of the nodes with largest degree. What we also notice from the figure is that the preferential attachment graph has far more nodes with degree one than the branching process graph. We can see this by looking to the straight line at the right side of the graph. For preferential attachment this line starts at 6700 nodes, while this line starts at 11500 nodes for the branching process. This is due to the different distributions we used in both processes. If we used other parameters for instance for the branching process it might be that the branching process tree has more nodes of degree one.

This suggests that our heuristics will work better on preferential attachment graphs than on branching process graphs.

10 Numerical Results

We have tested the heuristics described in section 7, both on trees created with branching processes and with preferential attachment. We have tested the heuristics on five small trees, consisting of 25 nodes, and one hundred large trees, consisting of 20, 000 nodes. These results can be seen in table 1 and 2 respectively.

Method		Branching process		Preferential attachment	
		ET	σ_T	ET	σ_T
7.1.2		15.18	1.12	13.8	0.22
7.2.1		18.41	1.31	15.53	1.04
7.2.2	$\alpha = 0.5$	18.07	1.54	15.18	0.86
	$\alpha = 1$	18.47	1.62	14.9	0.65
	$\alpha = 2$	18.61	1.73	15.23	0.78
	$\alpha = 3$	18.61	1.73	15.1	0.66
7.2.3	$\alpha = 0.5$	18.49	1.55	15.22	1.45
	$\alpha = 1$	18.39	1.76	15.13	1.16
	$\alpha = 2$	18.6	1.73	15.03	0.68
	$\alpha = 3$	18.61	1.73	15.1	0.66
7.2.4		46.73	18.36	17.28	2.21
7.2.5		15.29	1.15	13.99	0.47

Table 1: The results for small trees (25 nodes) with 5 samples

Method		Branching process		Preferential attachment	
		ET	σ_T	ET	σ_T
7.2.1		24045	1259	19204	152
7.2.2	$\alpha = 0.5$	18919	538	16347	63
	$\alpha = 1$	17171	436	14501	55
	$\alpha = 2$	18028	497	14438	68
	$\alpha = 3$	18962	566	15014	98
7.2.3	$\alpha = 0.5$	19513	722	15898	82
	$\alpha = 1$	18178	480	15076	57
	$\alpha = 2$	17393	483	14350	57
	$\alpha = 3$	18671	575	14820	85
7.2.4		54064	32172	20531	1281
7.2.5		19420	553	19231	64

Table 2: The results for large trees (20000 nodes) with 100 samples

In the first column of these tables we list the heuristics used, these are described in section 7. We only searched for the optimal solution on small trees, since this method is very slow. The small and large trees consist of 25 and 20000 nodes respectively. The third and fourth, as well as the fifth and sixth columns of the tables, denote the average time and the standard deviation of $\mathrm{E}[T(\Gamma)]$ for different graphs. The trees are created using either a branching process or preferential attachment.

In the second table it can be seen that for large trees created by preferential attachment, heuristic 7.2.2 with $\alpha = 1$ works better than any other heuristic. For branching processes, the standard deviation is too high to draw any conclusions, so table 3 shows results (5 samples) with the $E[T(\Gamma)]$ of heuristic 7.2.2 subtracted (using $\alpha = 1$). This removes a lot of the influence of the shape of the graph, because the result of the best heuristic is subtracted before averaging with different graphs. This shows that heuristic 7.2.2 using $\alpha = 2$ is still reasonably good, but $\alpha = 1$ is a bit better. Note that the difference between heuristic 7.2.2 (using $\alpha = 1$) and heuristic 7.2.3 (using $\alpha = 2$) is not significant, so both methods give similar results.

Method		$E(T - \kappa)$	$\sigma_{T-\kappa}$
7.2.1		7370	314
7.2.2	$\alpha = 0.5$	1869	113
	$\alpha = 1$	0	0
	$\alpha = 2$	888	88
	$\alpha = 3$	1881	262
7.2.3	$\alpha = 0.5$	2592	167
	$\alpha = 1$	1060	60
	$\alpha = 2$	125	127
	$\alpha = 3$	1536	86
7.2.4		59794	30333
7.2.5		2437	257

Table 3: The results for large trees with κ equal to the $E[T(\Gamma)]$ of heuristic 7.2.2 (per sample)

The results obtained on the small trees with the different heuristics are quite in line though a little bit different then the results obtained on the large trees with the different heuristics. These results however are insignificant, due to the high standard deviations. There is one particular aspect that we could test on the small trees but not on the large trees, the optimal solution to our problem obtained via brute force search. As can be seen from the results, there is still improvement on small graphs possible for the heuristics, and we expect that this is true for larger graphs as well.

Figure 10 shows for a sample graph how much time is needed for both a branching process graph and a preferential attachment graph to find the object with a certain probability.

This figure allows several more observations if we compare it with figure 9. We notice that given the degree distribution of figure 9 our heuristic performs reasonably well (figure 9 shows the time needed to find the object assuming all nodes can be visited in optimal order directly after each other).

We also notice that the preferential attachment graph has more nodes with higher degree, making it possible to find the object faster on average.

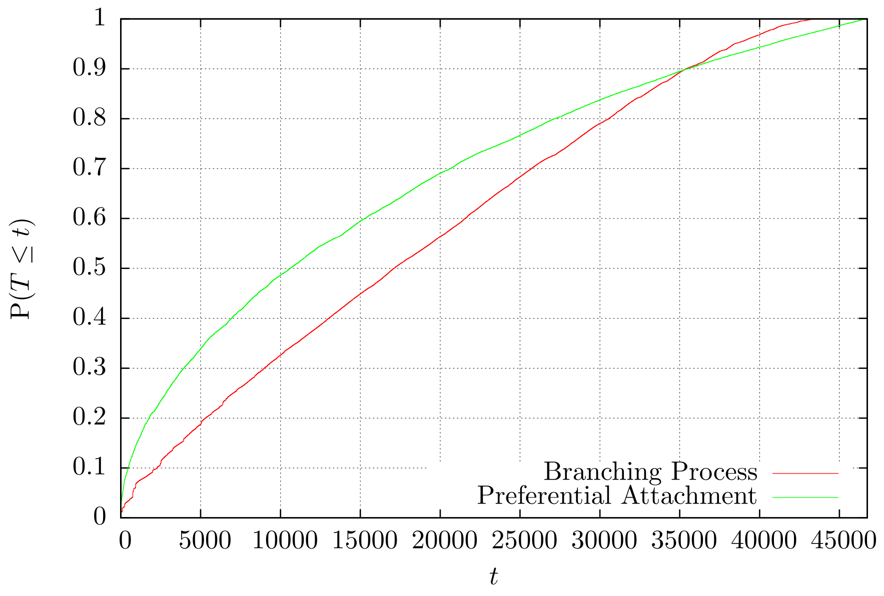


Figure 10: Graph of how fast the object is found
(Computed with heuristic 7.2.2 using $\alpha = 2$)

11 Conclusion & Discussion

We did research on finding a hidden object in a tree, when the probability that an object is hidden at a node is proportional with the degree of the node. This means that when a node has a relatively large degree that the probability that a object is hidden in this node is also relatively high. We used trees instead of graphs to make the problem easier to analyze, but for generality further research is needed to see what results and observations in this paper hold for general graphs too.

What we did was searching for the optimal walk in the tree which would visit all nodes. Using brute force, so trying every possible walk, we could only find the optimal walk in trees with a maximum of ten nodes. We then implemented some strategies to improve our search. With this we could calculate the optimal walk in trees with a maximum of 25 nodes in a reasonable time.

One strategy we used is visiting all leaf nodes connected to a comment ancestor subsequently one after another, and only after that visiting other nodes in the tree. We proved that this strategy is optimal.
One strategy we used and also proved correct in our report is, if it is optimal to visit a leaf node(node with degree one) it is optimal to visit all leaf nodes connected to the neighbour of the leaf node directly after visiting this leaf node.

In most networks you work with much larger graphs than say 25 nodes. To solve the problem for larger trees, of a few thousand nodes, we came up with several heuristics which try to find the optimal walk. However in smaller trees where we can calculate the optimal walk we discovered that these heuristics are not optimal in most cases.

We have created a way to convert our problem to a Travelling Repairman Problem and an algorithm to find the optimal walk in a tree for our problem. This algorithm uses dynamic programming.
To try the heuristics, we first needed to generate large trees. We chose to generate two different types of trees: branching trees and preferential attachment trees. Our heuristics are based on visiting nodes with large degree first, because at the nodes with a large degree the probability of finding the object is larger than in nodes with a small degree. We expected that in the preferential attachment trees a hidden object is found faster than in a branching process tree, our results also show this. Preferential attachment trees have, in our experiments, far more nodes with degree one than branching process trees in most cases, this means also more nodes with a large degree. Secondly the nodes with large degree can be anywhere in a branching process tree and in the preferential attachment trees these nodes mostly arise in the first generations of the tree.

From the results we do not get one best heuristic. What we can say is that the heuristics 'Degree divided by distance' and 'Summed degree divided by distance' with an $\alpha = 1,2$ give the best results. For further research we recommend trying other heuristics, for example adapting heuristics used for the Travelling Repairman Problem.

Since our problem has a lot of related problems, there are a lot of ways to try and extend our problem. As noted above, instead of minimizing the expected time in trees, one could minimize the expected time in a connected graph. Another current limitation is the probability distribution for the location of the object. Instead of a distribution proportional to the degree of a node, one could have a distribution which increases with the degree of a node. Alternatively, any restrictions on this distribution can be dropped.

In our problem walks are used to reach every node, but one could consider jumping to a chosen node or to a random node. This even is recommended in the case that the graph or tree is not fully known, but only a (set of) local section(s) is visible at any given time.

A more hands-on improvement of this paper is looking at more ways to generate graphs, using different parameters, or making sure that the Branching Process graph has an distribution of degrees identical to that of a Preferential Attachment graph.

12 References

References

[1] Konstantin Avrachenkov, Nelly Litvak, Marina Sokol, and Don Towsley. "Quick Detection of Nodes with Large Degrees". In: *CoRR* abs/1202.3261 (2012). URL: `http://dblp.uni-trier.de/db/journals/corr/corr1202.html#abs-1202-3261`.

[2] Albert-László Barabási and Réka Albert. "Emergence of Scaling in Random Networks". In: *Science* 286.5439 (1999), pp. 509–512. DOI: `10.1126/science.286.5439.509`. eprint: `http://www.sciencemag.org/content/286/5439/509.full.pdf`. URL: `http://www.sciencemag.org/content/286/5439/509.abstract`.

[3] K. Chaudhuri, B. Godfrey, S. Rao, and K. Talwar. "Paths, trees, and minimum latency tours". In: cited By (since 1996)34. 2003, pp. 36–45. URL: `http://www.scopus.com/inward/record.url?eid=2-s2.0-0344981511&partnerID=40&md5=17e26a41cd30173f0a4bcd6cf1ff449e`.

[4] AFRATL F., COSMADAKIS S., PAPADIMITRIOU C. H., PAPAGEORGIOU G., and PAPAKOSTANTINOU N. "The complexity of the travelling repairman problem". In: *R.A.I.R.O. Informatique théorique* 20.1 (1986). Ed. by Dunod. eng, pp. 79–87. ISSN: 0399-0540. URL: `http://www.refdoc.fr/Detailnotice?idarticle=13090207`.

[5] Leo Katz. "A new status index derived from sociometric analysis". In: *Psychometrika* 18.1 (1953), pp. 39–43.

[6] Sheldon M. Ross. *Introduction to Probability Models, Ninth Edition*. Orlando, FL, USA: Academic Press, Inc., 2006. ISBN: 0125980620.

[7] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (2nd Edition)*. Prentice Hall, Dec. 2002. ISBN: 0137903952. URL: `http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20\&amp;path=ASIN/0137903952`.

[8] Colin Schepers. "Searching in Networks". June 2010.

[9] John N. Tsitsiklis. "Special cases of traveling salesman and repairman problems with time windows". In: *Networks* 22 (1992), pp. 263–282.