Optimal Unlabeled Pebble Motion on Trees

Pierre Le Bodic, Edward Lam

Department of Data Science and Artificial Intelligence, Monash University, Melbourne, Australia
pierre.lebodic@monash.edu, edward.lam@monash.edu

Abstract

Given a tree, a set of pebbles initially stationed at some nodes of the tree and a set of target nodes, the Unlabeled Pebble Motion on Trees problem (UPMT) asks to find a plan to move the pebbles one-at-a-time from the starting nodes to the target nodes along the edges of the tree while minimizing the number of moves. This paper proposes the first optimal algorithm for UPMT that is asymptotically as fast as possible, as it runs in a time *linear* in the size of the input (the tree) and the size of the output (the optimal plan).

Introduction

The input of the Unlabeled Pebble Motion on Trees problem (UPMT) is a tree $T = (V, E)$, with $n = |V|$. A set of k pebbles is initially located at k distinct nodes of the tree, called *starting nodes*. There are also k nodes described as *targets*, some of which may coincide with the starting nodes. A *move* is an action that moves a single pebble from its current node to an adjacent pebble-less node. A *plan* is a sequence of moves. A *feasible* plan moves every pebble to a target. The *length* of a plan is the number of moves it uses. The length of an optimal plan is denoted OPT.

In the *labeled* version of the problem, every starting node and every target node is labeled with a specific pebble, and every target node must be reached by its designated pebble. In *unlabeled* (or *anonymous*) Pebble Motion problems, a target can be reached by any pebble.

Pebble Motion is related and has been used (see e.g. Kulich, Novák, and Přeucil 2019) to tackle Multi-Agent Pathfinding (MAPF) (Stern 2019), a problem in which pebbles/agents can move synchronously, and hence lends itself well to real-life problems such as automated warehouses.

This paper proposes a novel optimal algorithm for UPMT with time complexity $O(n \log n + OPT \log n)$, where the encoding size of the input is $O(n \log n)$, and the encoding size of the plan is $O(OPT \log n)$. We also show that $OPT \leq k(n-1)$. The algorithm traverses the tree top-down, and, at each node, moves pebbles within its subtree to ensure that the subtrees rooted at its children contain one pebble per target.

The paper is organized as follows. First, we review related work. Second, we derive a tight lower bound on the length of an optimal plan. Third, we introduce an optimal algorithm using this lower bound. Naturally, we end with conclusions.

Related Work

Kornhauser, Miller, and Spirakis (1984) propose a sub-optimal algorithm for UPMT with plans of length $O(n^3)$. Ardizzoni et al. (2024) provide a more readable description, as well as an improvement to path lengths in $O(knc + n^2)$, where c is the length of a *corridor*, which is at worst $O(n)$.

Auletta et al. (1999) provide an algorithm linear in n to determine the feasibility of the labeled problem. In the feasible cases, they also provide an algorithm that runs in $O(n + OPT_l)$, where OPT_l refers to the length of a plan for the labeled problem, which is in $O(k^2(n-k))$.

Călinescu, Dumitrescu, and Pach (2008) study (un)labeled problems similar to the Pebble Motion problem on various classes of graphs, but define a move to be a consecutive sequence of edge moves made by a single pebble. They also give an optimal algorithm for UPMT. This algorithm first computes all shortest paths between pebbles and targets, and then solves a minimum-weight matching problem to find an assignment of pebbles to targets that minimizes the total distance between pebbles and targets. The time complexity of solving a minimum matching in a complete bipartite graph with $2k$ vertices and edge weights of at most n is $O(k^{2.5} \log n)$ (Gabow and Tarjan 1989) or $O(k^3)$ (Edmonds and Karp 1972).

Lower Bound on the Optimal Plan Length

Pick an arbitrary root node $r \in V$. Let T_u denote the subtree rooted at a node $u \in V$. We write $v \in T_u$ to mean that v is a node of T_u. We define $pebble(u) \in \{0, 1\}$ and $target(u) \in \{0, 1\}$ as the current number of pebbles and targets at a node $u \in V$, respectively. Note that $pebble(u)$ is updated as pebbles move, while $target(u)$ is a fixed input. We define the *demand* $d : V \to \mathbb{Z}$, such that, for $u \in V$,

$$d(u) = \sum_{v \in T_u} target(v) - \sum_{v \in T_u} pebble(v), \qquad (1)$$

as the number of targets minus the current number of pebbles in T_u, or, recursively,

$$d(u) = target(u) - pebble(u) + \sum_{v \text{ child of } u} d(v). \qquad (2)$$

Lemma 1. *The problem is solved if and only if $d(u) = 0$ for every $u \in V$.*

Proof. ($\Rightarrow$) Suppose the problem is solved, i.e. $target(u) = pebble(u)\ \forall u \in V$, then (1) implies $d = 0$. ($\Leftarrow$) By induction. At every leaf u, (2) gives $0 = target(u) - pebble(u) + 0$. Furthermore, for every non-leaf node u, suppose that $d(v) = 0$ for each child v of u. Again, (2) simplifies to $0 = target(u) - pebble(u) + 0$. Therefore, at every node u, $target(u) = pebble(u)$. □

Lemma 2. *A feasible plan has length at least $\sum_{u \in V} |d(u)|$.*

Proof. Consider a non-root node $v \in V \setminus \{r\}$ and its parent $u \in V$. Suppose that $d(v) \geq 0$, i.e. there are $d(v)$ more targets than pebbles in the subtree T_v. Any feasible plan must include at least $d(v)$ moves from u through v, otherwise the subtree T_v will have missing pebbles after the execution of the plan. In the case where T_v has extra pebbles, i.e. $d(v) \leq 0$, there must be $-d(v)$ moves from v through u. Therefore, in a feasible plan, the number of moves on edge uv is at least $|d(v)|$. Finally, observe that $d(r) = 0$. □

Optimal Unlabeled Pebble Motion on Trees

We now use the demand d to design an algorithm that makes no unnecessary moves. To do so, only moves that decrease the lower bound (Lemma 2) are performed. We present a recursive top-down algorithm that realizes this.

Algorithm Description

Algorithm 1, *balance_subtrees*, starts at a node u with zero demand (Precondition 1.b). The first part of *balance_subtrees*, Lines 1-7, ensures that every child of u has zero demand, by balancing the pebbles between the children. A balance between children is achievable because $d(u) = 0$. For the same reason, *balance_subtrees* implicitly handles the case of u being a target because ensuring all children are balanced means that a pebble is left on u if and only if u is a target. Node u interacts with the subtree of a child v via the functions $inject\_pebble(v)$ and $extract\_pebble(v)$, not by direct moves between u and v, as moving pebbles from one subtree to another can require moves inside these subtrees. The second part of *balance_subtrees*, Lines 8-9, recursively calls *balance_subtrees* on all children to ensure that all descendants have zero demand (Postcondition 1.c).

Algorithm 2, *inject_pebble*, is called on v by its parent u when a pebble needs to be moved from u to v. It handles the case where v already holds a pebble. If so, v must first *inject_pebble* its pebble to one of its children, before the pebble on u is moved to v. Because $d(v) > 0$ (Precondition 2.c), this is guaranteed to succeed.

Algorithm 3, *extract_pebble*, moves a pebble from v to its parent u. It handles the case where there is no pebble on v by recursively extracting a pebble from within T_v. Thanks to Precondition 3.c, this is guaranteed to succeed.

Algorithm 4, *move_pebble*, updates $pebble(\cdot)$ and $d(\cdot)$, and finally performs a move.

Algorithm 1: $balance\_subtrees(u)$

Precondition 1.a: A node $u \in V$
Precondition 1.b: $d(u) = 0$
Postcondition 1.c: $d(v) = 0$ for all $v \in T_u$
1: **while** a child v of u with $d(v) \neq 0$ exists **do**
2: **if** $pebble(u) = 1$ **then**
3: Pick a child v of u with $d(v) > 0$
4: $inject\_pebble(v)$
5: **else**
6: Pick a child v of u with $d(v) < 0$
7: $extract\_pebble(v)$
8: **for** v child of u **do**
9: $balance\_subtrees(v)$

Algorithm 2: $inject\_pebble(v)$

Precondition 2.a: A node $v \in V \setminus \{r\}$ with parent u
Precondition 2.b: $pebble(u) = 1$
Precondition 2.c: $d(v) > 0$
Postcondition 2.d: $pebble(u) = 0$
Postcondition 2.e: $pebble(v) = 1$
1: **if** $pebble(v) = 1$ **then**
2: Pick a child w of v such that $d(w) > 0$
3: $inject\_pebble(w)$
4: $move\_pebble(u, v)$

Algorithm 3: $extract\_pebble(v)$

Precondition 3.a: A node $v \in V \setminus \{r\}$ with parent u
Precondition 3.b: $pebble(u) = 0$
Precondition 3.c: $d(v) < 0$
Postcondition 3.d: $pebble(u) = 1$
Postcondition 3.e: $pebble(v) = 0$
1: **if** $pebble(v) = 0$ **then**
2: Pick a child w of v such that $d(w) < 0$
3: $extract\_pebble(w)$
4: $move\_pebble(v, u)$

Algorithm 4: $move\_pebble(u, v)$

Precondition 4.a: A node $u \in V$ adjacent to a node $v \in V$
Precondition 4.b: $pebble(u) = 1$
Precondition 4.c: $pebble(v) = 0$
Precondition 4.d: if v is a child of u, then $d(v) > 0$
Precondition 4.e: if u is a child of v, then $d(u) < 0$
Postcondition 4.f: $pebble(u) = 0$
Postcondition 4.g: $pebble(v) = 1$
1: $pebble(u) \leftarrow 0$
2: $pebble(v) \leftarrow 1$
3: **if** v is a child of u **then**
4: $d(v) \leftarrow d(v) - 1$
5: **else**
6: $d(u) \leftarrow d(u) + 1$
7: Output move (u, v)

Algorithm Correctness

We prove that the algorithm terminates after having ensured that all pebbles are located at a target.

Correctness of $move\_pebble$

Lemma 3. *A call to* $move\_pebble$ *correctly updates* d*.*

Proof. Assuming that d is correct when $move\_pebble$ is called, we prove it is still correct after the call. A move from u to v can only change $d(u)$ and $d(v)$. If v is a child of u, moving a pebble from u to v does not change the number of pebbles in T_u. But the number of pebbles in T_v increases by 1, therefore the demand $d(v)$ decreases by 1. Similarly, if v is the parent of u, moving a pebble from u to v increases the demand $d(u)$ by 1. □

Lemma 4. *A call to* $move\_pebble$ *decrements* $\sum_{u \in V} |d(u)|$ *by 1.*

Proof. If v is a child of u, then by Precondition 4.d, $d(v) > 0$, and it is decremented on Line 4. Otherwise, by Precondition 4.e, $d(u) < 0$, and it is incremented on Line 6. The value of d at other nodes are unchanged. □

Lemma 5. *Postconditions 4.f and 4.g are satisfied.*

Correctness of $inject\_pebble$

Lemma 6. *Preconditions 2.a to 2.c are satisfied when* $inject\_pebble$ *calls itself recursively on Line 3.*

Proof. Note that $pebble(v) = 1$ on Line 3. Precondition 2.a and Precondition 2.b are clearly satisfied. To show Precondition 2.c, we check that there must be a child w of v such that $d(w) > 0$ on Line 2. Evaluating (2) for v gives:

$$\begin{aligned} & d(v) = target(v) - pebble(v) + \sum_{w \text{ child of } v} d(w) \\ \implies & 0 < d(v) \leq \max(0,1) - 1 + \sum_{w \text{ child of } v} d(w) \\ \implies & 0 < \sum_{w \text{ child of } v} d(w). \end{aligned}$$

□

Lemma 7. *Preconditions 4.a to 4.e are satisfied when* $move\_pebble$ *is called by* $inject\_pebble$ *on Line 4.*

Proof. Precondition 4.a: node u is the parent of v, therefore they are adjacent.

Precondition 4.b: Precondition 2.b remains true until $move\_pebble$ is called on Line 4, as only pebbles within T_v may have moved before Line 4.

Precondition 4.c: if $pebble(v) = 1$ on Line 1, then $inject\_pebble$ is called on Line 3, which, by Postcondition 2.d, ensures $pebble(v) = 0$ on Line 4.

Precondition 4.d: Precondition 2.c remains true until Line 4, since the number of pebbles in T_v does not change before Line 4.

□

Lemma 8. *Postconditions 2.d and 2.e are satisfied.*

Proof. Direct consequence of Line 4 and Lemma 5. □

Correctness of $extract\_pebble$ The results and proofs are similar to those of $inject\_pebble$.

Correctness of $balance\_subtrees$

Lemma 9. *Preconditions 2.a to 2.c are satisfied when* $inject\_pebble$ *is called by* $balance\_subtrees$*.*

Proof. Precondition 2.a is satisfied, as v has a parent u. Precondition 2.b is satisfied, since $pebble(u) = 1$ on Line 2. To show that Precondition 2.c is satisfied, we show that there is indeed a child v of u with $d(v) > 0$. Suppose there is not. Then, applying (2) at u,

$$\begin{aligned} d(u) &= target(u) - pebble(u) + \sum_{v \text{ child of } u} d(v) \\ &\leq \max(0,1) - 1 - \sum_{v \text{ child of } u} |d(v)| \\ &\leq - \sum_{v \text{ child of } u} |d(v)| \\ &< 0, \end{aligned}$$

where the last inequality holds because the while condition on Line 1 is true. Since $d(u) = 0$ (by Precondition 1.b, and no pebble moving out of T_u), this is a contradiction. □

Lemma 10. *Preconditions 3.a to 3.c are satisfied when* $extract\_pebble$ *is called by* $balance\_subtrees$*.*

Proof. The proof is similar to that of Lemma 9. □

Lemma 11. *Preconditions 1.a and 1.b are satisfied for all children* v *of* u *when* $balance\_subtrees$ *calls itself recursively.*

Proof. The while condition on Line 1 is false on Line 8, therefore $d(v) = 0$ for every child v of u. No pebble moves between u and any of its children on or after Line 8, hence this remains true for all recursive calls. □

Lemma 12. *Postcondition 1.c is satisfied.*

Proof. We first observe that, after each iteration of the while loop, at least one call to $move\_pebble$ is made via $inject\_pebble$ or $extract\_pebble$, therefore, by Lemma 4, $\sum_{v \text{ child of } u} |d(v)|$ is decremented by at least 1. Hence, Line 8 is ultimately reached. Since $balance\_subtrees$ is then called on all descendants of u via recursion, and since Precondition 1.b is satisfied for all these calls (Lemma 11), all nodes $v \in T_u$ satisfy $d(v) = 0$ at the end of the call. □

Theorem 13. *Algorithm* $balance\_subtrees$ *produces a feasible plan for UPMT.*

Proof. Direct from Lemma 1 and Lemma 12. □

Algorithm Optimality

We show that the plan output by *balance_subtrees* has minimum size, as it meets the bound given in Lemma 2.

Theorem 14. *Algorithm balance_subtrees outputs a plan of optimal length* $OPT = \sum_{u \in V} |d(u)|$.

Proof. Each pebble move is the result of exactly one call to $move\_pebble$. Lemma 4 ensures that $move\_pebble$ can be called at most $\sum_{u \in V} |d(u)|$ times. Thus, the plan produced by $balance\_subtrees$ has the length at most OPT. Since the plan is feasible (Theorem 13), its length is OPT (Lemma 2). □

The results below further characterize optimal plans.

Corollary 15. $OPT \leq k(n-1)$

Proof. Direct from Theorem 14, the fact that $|d(u)| \leq k$ for all $u \in V$, and $d(r) = 0$. □

Theorem 16. *In an optimal plan, every pebble reaches its target via a shortest path.*

Proof. Preconditions 4.d and 4.e ensure that a move between two nodes can occur in at most one direction throughout the algorithm. Therefore, since a feasible plan is returned, all paths are simple, i.e. do not cycle. It remains only to recall that there is a unique simple path between two nodes in a tree (Diestel 2017), and thus it is shortest.

While this proof only shows that optimal plans produced by $balance\_subtrees$ have this property, notice that if a different optimal plan contained a longer-than-shortest path for one pebble, then the path of another pebble in the plan would need to be shorter-than-shortest, which is a contradiction. □

Algorithm Complexity

We suppose that a reference (e.g. index or pointer) to a node uses $O(\log n)$ bits, and thus reading or writing a reference to a node takes $O(\log n)$ time. Therefore, the size of the encoding of an n-ary tree on n nodes is $n \log n$. Furthermore, the encoding of a plan is the number of moves times two node indices, one for each endpoint, so its size is $O(OPT \log n)$ if it is optimal.

We prove that the runtime of this algorithm is asymptotically optimal, as it runs in a time linear in the encoding size of the input plus the encoding size of the plan, i.e. $O(n \log n + OPT \log n)$.

In the proofs that follow, observe that, if, for the analysis, we chose a computational model where all numbers could be encoded in constant size, all $\log$ terms would disappear. The runtime would then simplify to $O(n + OPT)$, which, in this model, would also be the encoding size of the input and the encoding size of the output.

Lemma 17. *Function d can be computed in $O(n \log n)$ time.*

Proof. Note that d holds numbers of encoding size at most $O(\log k)$. The base case of the recursive definition (2) of d occurs at the leaves, thus d can be computed by a postorder traversal of T. At each node u, we access all of its n_u children, which takes $O(n_u \log n)$ time, and add up the $n_u + 2$ terms of (2), which takes $O(n_u \log k)$ time. Since $k \leq n$, this is $O(n_u \log n)$ per node. Since there are $\sum_{u \in V} n_u = n - 1$ children in T, the runtime is in $O(n \log n)$. □

Lemma 18. $move\_pebble$ *runs in* $O(\log n)$.

Proof. This is simply the time to output u and v, access u, v and related data, as well as the addition in $O(\log k)$ time to update d. □

Lemma 19. *The total number of calls to* $inject\_pebble$ *and* $extract\_pebble$ *is* OPT.

Proof. Both $inject\_pebble$ and $extract\_pebble$ (whether it is a recursive call or not) call $move\_pebble$ exactly once. Theorem 14 shows that this happens OPT times. □

Lemma 20. *At a node u, picking a child v with $d(v) > 0$ or $d(v) < 0$ takes $O(\log n)$ time.*

Proof. When creating the tree, store each child v of u in one of three lists at u according to whether $d(v) > 0$, $d(v) = 0$ or $d(v) < 0$. Return the first node of each list as required, which takes $O(\log n)$. If, after a move, $d(v)$ is now 0, pop v out of its list and place it in the $d = 0$ list, which is in $O(1)$. □

Corollary 21. *The total runtime for all calls to* $inject\_pebble$ *and* $extract\_pebble$ *is in* $O(OPT \log n)$.

Lemma 22. *Across all calls, the total number of while (resp. for) loop iterations of* $balance\_subtrees$ *is at most* OPT *(resp.* $n - 1$*).*

Proof. Each while loop iteration leads to at least one pebble move, of which there is at most OPT across all calls to $balance\_subtrees$. □

Theorem 23. *The runtime of* $balance\_subtrees$ *is in* $O(n \log n + OPT \log n)$.

Proof. Reading the input and computing d (Lemma 17) takes $O(n \log n)$ time. All calls to functions $move\_pebble$, $inject\_pebble$ and $extract\_pebble$ together take $O(OPT \log n)$ time (Corollary 21). The total number of iterations in all calls to $balance\_subtrees$ is $O(OPT + n)$ (Lemma 22). Lemma 20 shows that each iteration takes $O(\log n)$ time. □

Conclusions

Remarkably, the runtime of the algorithm is equal to the runtime of reading the input and writing the output. Another interesting property of this algorithm is that for any node $u \in V$ such that $d(u) = 0$, UPMT can be solved in T_u independently of the rest of the tree, as no pebble will move from u to its parent (if any). In particular, this means that the for loop at Line 8 of $balance\_subtrees$ is "embarrassingly parallelizable". Independent subproblems are also interesting in the context of MAPF, as at least one move per independent subproblem can take place simultaneously without collision. Interesting future directions include using this work to solve Pebble Motion problems on general graphs, for example via so-called "trans-shipment" nodes (Ardizzoni et al. 2024), as well as MAPF (see e.g. Kulich, Novák, and Přeucil 2019).

Acknowledgments

This research is partially supported by the Australian Research Council under the Discovery Early Career Researcher Award DE240100042.

References

Ardizzoni, S.; Saccani, I.; Consolini, L.; Locatelli, M.; and Nebel, B. 2024. An Algorithm with Improved Complexity for Pebble Motion/Multi-Agent Path Finding on Trees. *Journal of Artificial Intelligence Research*, 79.

Auletta, V.; Monti, A.; Parente, M.; and Persiano, P. 1999. A Linear-Time Algorithm for the Feasibility of Pebble Motion on Trees. *Algorithmica*, 23: 223–245.

Călinescu, G.; Dumitrescu, A.; and Pach, J. 2008. Reconfigurations in Graphs and Grids. *SIAM Journal on Discrete Mathematics*, 22(1): 124–138.

Diestel, R. 2017. *Graph Theory*. Springer Publishing Company, Incorporated, 5th edition. ISBN 3662536218.

Edmonds, J.; and Karp, R. M. 1972. Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *J. ACM*, 19(2): 248–264.

Gabow, H. N.; and Tarjan, R. E. 1989. Faster Scaling Algorithms for Network Problems. *SIAM Journal on Computing*, 18(5): 1013–1036.

Kornhauser, D.; Miller, G.; and Spirakis, P. 1984. Coordinating Pebble Motion On Graphs, The Diameter Of Permutation Groups, And Applications. In *25th Annual Symposium on-Foundations of Computer Science, 1984.*, 241–250.

Kulich, M.; Novák, T.; and Přeucil, L. 2019. Push, Stop, and Replan: An Application of Pebble Motion on Graphs to Planning in Automated Warehouses. In *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*, 4456–4463.

Stern, R. 2019. *Multi-Agent Path Finding – An Overview*, 96–115. Cham: Springer International Publishing. ISBN 978-3-030-33274-7.