Generalized Anvil Combination Ordering: A Shortest-Path Formulation for Pre-Enchanted Items and Multi-Enchantment Books in Minecraft

Daniel Charisma Christian - 13525052
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: danielcharismac0809@gmail.com, 13525052@std.stei.ac.id

***Abstract*—Combining enchanted items on an anvil in Minecraft is constrained by an XP cost formula and a per-item prior-work penalty that grows with each use. Existing calculators for finding the cheapest combination order often assume every input starts blank, paired with single-enchantment books that have never touched an anvil. This assumption breaks down whenever a player already owns partially enchanted gear or a book carrying more than one enchantment, which is the common case in practice. This paper models the problem as a shortest path search over a state space where each state is a multiset of items and each transition is one legal anvil combination weighted by its XP cost. Because every transition cost is non-negative, Dijkstra's algorithm applies directly and returns both the minimum total cost and the combination order that achieves it. A working implementation is tested against worked examples from the Minecraft Wiki and matches the documented costs, including cases with pre-enchanted items and multi-enchantment books that prior tools often cannot represent.**

Keywords—Dijkstra's algorithm, shortest path, graph search, anvil mechanics, enchantment combination, Minecraft

I. Introduction

Items in Minecraft may have attributes applied to them such as enchantments, custom names, durability spent, or other items it stores just to name a few. Anvils in Minecraft can be used to combine two items into a new item that absorbs the attributes of both items resulting in an item with favorably higher and better attributes. However, using an item that has previously been produced from combining two items in an anvil in conjunction with another item to produce an item with even better attributes incurs a prior-work penalty cost that grows exponentially [1]. Thus, properly planning ahead of time the sequence of item merges in order to minimize the exponentially growing total XP cost is paramount. There are existing tools to aid in determining this sequence, however they each rely on some assumptions rendering those tools unusable for certain common scenarios in the game where you aren't able to meet those criteria [4][5][6].

A single enchantment already changes how a tool or weapon performs, but the value of an anvil comes from stacking several at once onto one item. A pickaxe with only the Efficiency enchantment mines faster but the same pickaxe with Efficiency, Unbreaking, and Mending mines faster, survives far longer, and repairs itself from collected experience (XP), which matters because survival mode gives the player no second copy of that pickaxe if it breaks. Each enchantment on its own solves one problem. A fully combined item solves several at once, and that is what makes the combination worth the XP it costs [1].

This matters more in survival mode specifically because XP is not free to come by. Earning experience points requires killing mobs, mining ores, smelting items, breeding animals, or trading with villagers, each of which costs time and exposes the player to risk [1]. An item combination that turns out to be wasteful does not just cost levels on a counter; it costs the hours of activity it took to earn those levels in the first place, with no way to get that specific XP back except by grinding more of the same activities. This is the practical stake behind treating the merge order as an optimization problem rather than something to improvise on the spot.

This paper will primarily be focusing on the problem of minimizing total XP cost required, which consists of enchantment XP cost and item prior-work penalty, for item combination on anvils in Minecraft's survival mode whilst aiming to provide a solution with looser restraints by using less assumptions than what pre-existing tools demand, such as, among others, requiring one nonbook item while every other item is an enchanted book.

This paper will not be accounting for the XP cost of durability repairs, nor for renaming items when combining items on anvils as those require at most only one additional XP point of cost and may even be entirely circumvented with clever use of another object in the game called a grindstone [1]. This paper will also assume that no pair of starting items have enchantments that are incompatible with the other.

II. Theoretical Foundations

A. Sets and Multisets

A Set is defined as a possibly-empty unordered collection of distinct objects. These objects within a set may be referred

to as elements or members. Notationally, sets are written as $\{o_1, o_2, o_3, \dots, o_n\}$, where o_i represents each object of the set. For instance, $\{1, 2, 3, 4, 5, 6\}$ is a set with elements 1, 2, 3, 4, 5, and 6 [2].

While a set cannot have repeating elements as all of them must be distinct, a multiset is defined as a possibly-empty unordered collection of objects of which are unnecessarily distinct. For instance, $\{1, 1, 2, 2, 3\}$, $\{2, 2, 2\}$, $\{2, 3, 4\}$, $\{\}$ are all multisets respectively [2].

The multiplicity of an element of a multiset is defined as the number of times said element appears in the multiset. For instance in the set $\{0, 1, 1, 1, 0, 0, 0, 1\}$, 0 appears 4 times and thus has a multiplicity of 4 [2].

B. Graph Theory

Graphs are commonly used to represent discrete objects and relations between objects. A graph is defined as a pair (V, E), where V is a nonempty set of vertices $\{v_1, v_2, v_3, \dots, v_n\}$, representing the objects, and E is a possibly-empty set of edges $\{e_1, e_2, e_3, \dots, e_n\}$ representing the relationships between the objects. Each edge connects two vertices, indicating a relationship or interaction between them [2].

Based on whether a graph contains multiple edges between the same pair of vertices or not, it can be classified as a simple graph or a non-simple graph. A simple graph is a graph that does not contain multiple edges between the same pair of vertices, while a non-simple graph allows multiple edges between the same pair of vertices [2].

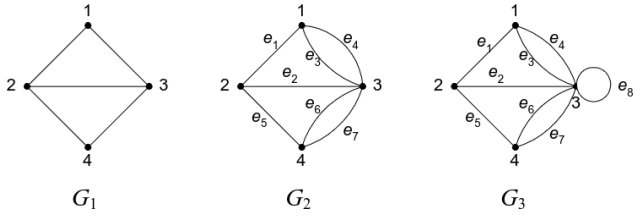


Figure 2.2.1 Simple and non simple graphs. Source: [2]

In the figure above, is a graph with vertices $\{1, 2, 3, 4\}$ and edges $\{(1, 2), (1, 3), (2, 3), (2, 4), (3, 4)\}$. It is a simple graph since it doesn't have multiple edges between the same pair of vertices whereas and are non-simple graphs because both of them have multiple edges between the same pair of vertices, and are both connecting and on both graphs [2].

A graph can also be directed or undirected. A directed graph is a graph where the edges have a direction, indicating a one-way relationship between the vertices. An undirected graph is a graph where the edges do not have a direction, indicating a two-way relationship between the vertices [2].

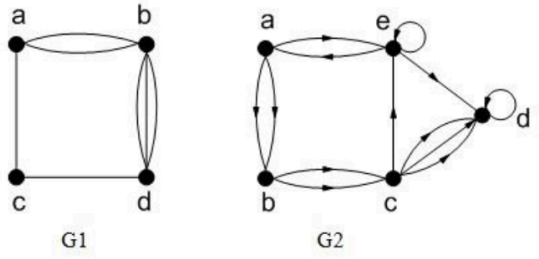


Figure 2.2.2 Directed and Undirected graphs. Source: [2]

In the figure above, G_1 is a non-simple undirected graph as it provides no visual indicator of a direction on its edges, which means its edges do not dictate a direction between the vertex it connects to. G_2 is a non-simple directed graph where every edge has a direction on which vertex it's directed to [2].

A graph may also be weighted, i.e. its edges may hold values [2].

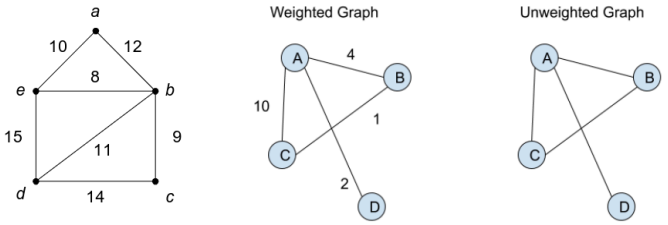


Figure 2.2.3 Weighted and Unweighted Graphs. Source: [2]

As shown in figure 2.2.3, despite the weighted graph and the unweighted graph having the same edges and vertices, the edges of the weighted graph are associated with an additional information called its weight.

C. Minecraft Anvil Mechanics

When using an anvil in Minecraft, there are two slots to place the two items to be combined in. The first slot will in this paper be considered the target item, and the second slot as the sacrifice item. However, not all items can be combined in anvils: only tools, armor, weapons, and books. Thus, only item attributes that pertain to those items are relevant to how the anvil produces the new item. Such attributes that may appear on the aforementioned items are, enchantments, custom names, durability spent, and anvil-use count. Additionally, two items may be combined if the target item and the sacrificed item are of the same item type, or if the sacrificed item is a book and has at least one enchantment [1].

For every enchantment on the sacrifice item which we will call x and has level p, if the target item has an enchantment that is incompatible with enchantment x, then the combined item will not have enchantment x. Otherwise, if the target item has enchantment x with level q then

- If $p > q$ then the combined item will have enchantment x with level p
- If $p = q$ then the combined item will have enchantment x with level $min(p + 1, M)$, where M

is the enchantment's maximum level (refer to Table 2.3.1)
- If $p < q$ then the combined item will have enchantment x with level q

And if the target item does not have enchantment x, then the combined item must have enchantment x with level p. After all of that, the combined item that is produced must have every enchantment from the target item with either the same or higher level than the target item. If the target item has an anvil-use count of n, and the sacrifice item has an anvil-use count of m, then the combined item will have an anvil-use count of $max(n, m) + 1$ [1].

For every enchantment on the sacrifice item which we will call x, has level p, and has cost multiplier of c, if the target item has an enchantment that is incompatible with enchantment x, add 1 to the total enchantment cost. Otherwise if the target item has enchantment x with level q then

- If $p > q$, add $p \cdot c$ to the total enchantment cost
- If $p = q$, add $min(p + 1, M) \cdot c$ to the total enchantment cost, where M is the enchantment's maximum level (refer to Table 2.3.1)
- If $p < q$, add $q \cdot c$ to the total enchantment cost

The cost multiplier c is the book enchantment XP cost multiplier if the sacrifice item is a book item, otherwise it is the item enchantment XP cost multiplier (refer to table 2.3.1) [1].

The target item's prior-work penalty is calculated as $2^n - 1$ where n is the anvil-use count of the target item. Likewise, the sacrifice item's prior-work penalty is calculated as $2^m - 1$ where m is the anvil-use count of the sacrifice item [1].

The total XP cost of combining the two items is then the sum of the total enchantment cost, the target item's prior work penalty, and the sacrifice item's prior work penalty. If the total cost for combination is more than 39, then the items cannot be combined in anvil within the survival mode of the game, so it must be accounted for later in the algorithm [1].

The following is a table detailing every enchantment as of Minecraft 26.2 along with their maximum enchantment level, item enchantment XP cost multiplier, and book enchantment XP cost multiplier.

TABLE 2.3.1: Enchantments, their maximum level, and their cost multiplier

Name of Enchantment	Maximum Enchantment Level	Item enchantment XP cost multiplier	Book enchantment XP cost multiplier
Protection	4	1	1
Fire Protection	4	2	1
Blast Protection	4	4	2
Projectile Protection	4	2	1
Thorns	3	8	4
Respiration	3	4	2
Aqua Affinity	1	4	2
Feather Falling	4	2	1
Depth Strider	3	4	2
Frost Walker	2	4	2
Soul Speed	3	8	4
Swift Sneak	3	4	2
Sharpness	5	1	1
Smite	5	2	1
Bane of Arthropods	5	2	1
Knockback	2	2	1
Fire Aspect	2	4	2
Looting	3	4	2
Sweeping Edge	3	4	2
Lunge	3	2	1
Wind Burst	3	4	2
Density	5	2	1
Breach	4	2	1
Efficiency	5	1	1
Silk Touch	1	8	4
Fortune	3	4	2
Luck of the Sea	3	4	2
Lure	3	4	2
Power	5	1	1
Punch	2	4	2
Flame	1	4	2
Infinity	1	8	4
Impaling	5	4	2
Riptide	3	4	2
Loyalty	3	1	1
Channeling	1	8	4
Multishot	1	4	2
Piercing	4	1	1
Quick Charge	3	2	1
Mending	1	4	2
Unbreaking	3	2	1
Curse of Binding	1	8	4
Curse of Vanishing	1	8	4

D. Dijkstra's Algorithm

Dijkstra's algorithm solves the single-source shortest path problem on a weighted graph with non-negative edge weights [3]. Given a starting vertex, it finds the minimum-cost path from that vertex to every other reachable vertex, using a greedy strategy: at each step, it commits to the closest vertex not yet finalized, then updates the known distances to that vertex's neighbors.

The algorithm tracks two things for each vertex: a tentative distance from the source, and whether the vertex has been finalized. It starts with the source at distance 0 and every other vertex at infinity. On each iteration, it removes the unfinalized vertex with the smallest tentative distance from a priority queue, marks it finalized, and relaxes each outgoing edge: if reaching a neighbor through this vertex is cheaper than the neighbor's current recorded distance, that distance is updated. The algorithm can stop as soon as a specific goal vertex is finalized, rather than running until every vertex is, when only one destination matters.

The correctness argument rests on one fact: once a vertex is popped from the priority queue, its tentative distance is already its true shortest distance from the source. This only holds because edge weights are non-negative. If an edge could carry negative weight, some longer route discovered later could undercut a distance already declared final, which breaks the greedy assumption the algorithm depends on. Mehlhorn and Sanders give the standard correctness proof for this invariant, along with a priority-queue implementation that runs in $O((m + n)log\, n)$ time for a graph with n vertices and m edges [3].

This same non-negativity requirement is why Dijkstra's algorithm applies cleanly to the anvil problem in this paper. Every transition between item-multiset states (refer back to the

previous subsection on Anvil Mechanics) carries a cost built from enchantment level contributions and prior-work penalties, both of which are non-negative by construction, so the finalize-on-pop guarantee holds here too: the first goal state popped from the open set is provably optimal.

III. Problem Statement

Suppose we have a multiset of items each with a non-empty set of enchantments each having some arbitrary level. What is the minimum XP cost to combine all of those items into a singular item, and the order of combinations to be made in order to achieve that? If there is no way to combine all of these items then the algorithm we construct must report such too.

However, the problem then requires two cases to be considered. The first case where there are no pair of enchantments from two items in the multiset that are incompatible with each other, and the second case where there exists some pair of enchantments from two items in the multiset that are incompatible with each other. In this second case, we find the minimum XP cost and the order of combinations, for some arbitrary set of enchantments we wish to retain for the final combined product from the set of enchantments from all items in the multiset. For instance, we could have the enchantment Silk Touch on item A, and the enchantment Fortune on item B, and we may choose to retain the Silk Touch enchantment for the final product, or alternatively choose to retain the Fortune enchantment.

This paper will assume the first case; the second will not be considered as it involves a trickier outer combinatorial selection over an inner shortest-path problem. This trickier second case may hopefully in a future work be tackled.

IV. State Space Formulation

Let an item be a tuple $i = (\tau, E, n)$ where τ is the item type (sword, boots, book, etc.), E is a set of enchantments which is defined to be a tuple (e, l) where e and l are the enchantment type and enchantment level respectively, with n as the anvil-use count, from which the prior-work penalty is derived as $2^n - 1$.

A state S is a multiset of items, $S = \{i_1, i_2, i_3, \dots, i_k\}$. The initial state S_0 is the multiset of input items given to the algorithm. A state is a goal state if $|S| = 1$, i.e. every item has been merged into a single final item.

From state S a transition is defined by choosing an ordered pair (t, s) of distinct items in S where t is the target item and s is the sacrifice item such that t and s share the same item type or that s is a book; and that s has at least one enchantment. The transition then removes t and s from S and inserts the product of the two items, producing a successor state S' with $|S'| = |S| - 1$.

The XP cost for this transition is used as the edge weight and if it exceeds 39 then the transition cannot be made as anvils only allow combinations that cost less than 39 XP at a time in Minecraft's survival mode. Crucially, this XP cost is also always non-negative due to the cost multiplier always being positive (refer to table 2.3.1), every valid enchantment in the game having a positive valued level, and the prior-work penalty always being non-negative as the value n in the exponential always starts at 0 and increments by at least one on every combination, which is a precondition Dijkstra's Algorithm requires to correctly produce the shortest path, or in this case, the least total XP cost.

Alternatively, the sequence of transitions (combination orders) may be used as the state. However, consider how any two different merge orders that happen to produce the same intermediate multiset of items have, from that point onward, identical subproblems. i.e. the sequence of merge orders after until the end is invariant regardless of the sequence of merge orders before, or in other words, how you got here is irrelevant. The multiset of items may also then be used as a key for the closed set, thus pruning any paths that revisit an already explored state.

The full state space forms a directed graph: nodes are reachable multisets, while edges are valid anvil transitions weighted by XP cost. The problem "find the minimum total XP cost to combine all items into one, and the order to do it in" is therefore a shortest-path problem from S_0 to the set of goal states (any single-item multiset) in this graph. And because edge weights are non-negative, Dijkstra's algorithm finds this shortest path correctly and is guaranteed to terminate with an optimal answer the first time it pops a goal state from the priority queue.

V. Implementation

The implementation will be written in Rust as opposed to Python for the purpose of utilizing the speed given from a language that compiles down to machine code. ViRb3's Anvil Calculator similarly also uses Rust to benefit from the additional speed provided from a compiled language, albeit primarily for WebAssembly instead of machine code [5].

As with any code implementation, the data types and auxiliary functions must be defined first. However, as there exists a hefty amount of enchantments and their numeric attributes, they won't be shown completely on the following figures. For the full code, refer to the Appendix.

```rust
#[derive(Debug, PartialEq, Eq, PartialOrd, Ord, Clone, Copy, Hash)]
enum EnchantmentType {
    // General
    Mending,
    Unbreaking,

    // Armor
    Protection,
    FireProtection,
    BlastProtection,
    ProjectileProtection,
    Thorns,

    AquaAffinity,
    Respiration
```

Figure 5.1 Enchantment Types that are available (truncated)

```rust
#[derive(Debug, PartialEq, Eq, PartialOrd, Ord, Clone, Copy, Hash)]
struct Enchantment {
    enchantment_type: EnchantmentType,
    level: u32,
}

impl Enchantment {
    fn new(enchantment_type: EnchantmentType, level: u32) -> Self {
        Self {
            enchantment_type,
            level,
        }
    }
}
```

Figure 5.2 Implementation of the Enchantment tuple

```rust
#[derive(Debug, PartialEq, PartialOrd, Ord, Eq, Clone, Copy, Hash)]
enum ItemType {
    Pickaxe,
    Axe,
    Shovel,
    Hoe,

    CarvedPumpkin,
    Elytra,
    Helmet,
    Chestplate,
    Leggings,
    Boots,

    Sword,
    Bow,
    Crossbow,
    Trident,
    Mace,
    Spear,

    FishingRod,
    FlintAndSteel,

    Book,
}
```

Figure 5.3 The item types capable of being enchanted

```rust
#[derive(Debug, PartialEq, Eq, PartialOrd, Ord, Hash, Clone)]
struct Item {
    item_type: ItemType,
    enchants: BTreeSet<Enchantment>,
    /// Prior Work Penalty is calculated as 2^n - 1 where n is this value.
    anvil_use_count: u32,
}
```

Figure 5.4 Implementation of the Item tuple

```rust
impl Item {
    /// Assumes not used in an anvil before
    fn new_no_enchants(item_type: ItemType) -> Self {
        Self {
            item_type,
            enchants: Default::default(),
            anvil_use_count: 0,
        }
    }

    /// Assumes not used in an anvil before
    fn new_with_enchants(item_type: ItemType, enchants: Vec<Enchantment>) -> Self {
        Self {
            item_type,
            enchants: enchants.into_iter().collect(),
            anvil_use_count: 0,
        }
    }

    fn is_book(&self) -> bool {
        self.item_type == ItemType::Book
    }

    fn can_combine(&self, sacrifice: &Self) -> bool {
        ((self.item_type == sacrifice.item_type) || (sacrifice.is_book()))
            && !sacrifice.enchants.is_empty()
    }
```

Figure 5.5 Auxiliary functions to create or retrieve information from an Item

```rust
/// Assumes can_combine is true
fn combine(&self, sacrifice: &Self) -> Self {
    let mut enchants = self.enchants.clone();
    for sench in &sacrifice.enchants {
        if let Some(tench) = self
            .enchants
            .iter()
            .find(|e| e.enchantment_type == sench.enchantment_type)
        {
            let new_level = if tench.level == sench.level {
                (tench.level + 1).min(tench.enchantment_type.max_level())
            } else if tench.level > sench.level {
                tench.level
            } else {
                sench.level
            };
            let new_ench = Enchantment::new(tench.enchantment_type, new_level);
            enchants.insert(new_ench);
        } else if EnchantmentType::get_mutual_exclusion_groups()
            .iter()
            .find(|g| {
                g.contains(&sench.enchantment_type)
                    && g.iter()
                        .find(|se| {
                            **se != sench.enchantment_type
                                && self
                                    .enchants
                                    .iter()
                                    .find(|te| te.enchantment_type == **se)
                                    .is_some()
                        })
                        .is_some()
            })
            .is_none()
        {
            enchants.insert(*sench);
        }
    }
    let anvil_use_count = self.anvil_use_count.max(sacrifice.anvil_use_count) + 1;
    let item_type = self.item_type;
    Self {
        item_type,
        enchants,
        anvil_use_count,
    }
}
```

Figure 5.6 Auxiliary function to simulate the combination two items

```rust
/// Assumes can_combine is true
fn combination_cost(&self, sacrifice: &Self) -> u32 {
    // Includes cost for compatible and incompatible enchantments
    let mut enchant_cost = 0;
    for sench in &sacrifice.enchants {
        let new_level = if let Some(tench) = self
            .enchants
            .iter()
            .find(|e| e.enchantment_type == sench.enchantment_type)
        {
            if sench.level == tench.level {
                (sench.level + 1).min(sench.enchantment_type.max_level())
            } else if sench.level > tench.level {
                sench.level
            } else {
                tench.level
            }
        } else if EnchantmentType::get_mutual_exclusion_groups()
            .iter()
            .find(|g| {
                g.contains(&sench.enchantment_type)
                    && g.iter()
                        .find(|se| {
                            **se != sench.enchantment_type
                                && self
                                    .enchants
                                    .iter()
                                    .find(|te| te.enchantment_type == **se)
                                    .is_some()
                        })
                        .is_some()
            })
            .is_some()
        {
            1
        } else {
            sench.level
        };

        let multiplier = if sacrifice.is_book() {
            sench.enchantment_type.book_multiplier()
        } else {
            sench.enchantment_type.item_multiplier()
        };

        enchant_cost += new_level * multiplier;
    }
    let tpwp = (1 << self.anvil_use_count) - 1;
    let spwp = (1 << sacrifice.anvil_use_count) - 1;
    enchant_cost + tpwp + spwp
}
```

Figure 5.7 Auxiliary function to simulate the cost of combining two items

Each queue entry is a DijkstraState consisting of the current item multiset represented using Rust's BTreeMap [8], the accumulated cost to reach it, and the path of (target_index, sacrifice_index) pairs taken so far. The Ord implementation reverses the natural ordering on cost so that Rust's max-heap BinaryHeap [8] behaves as the min-heap Dijkstra requires, i.e. the lowest-cost frontier state is always popped first.

```rust
#[derive(PartialEq, Eq)]
struct DijkstraState {
    items: BTreeMap<Item, usize>,
    cost: u32,
    path: Vec<(usize, usize)>,
}
impl Ord for DijkstraState {
    fn cmp(&self, other: &Self) -> std::cmp::Ordering {
        // std BinaryHeap is a max heap, so reverse the cost to
        // make it behave like a min heap. We want to minimize cost after all.
        self.cost.cmp(&other.cost).reverse()
    }
}
impl PartialOrd for DijkstraState {
    fn partial_cmp(&self, other: &Self) -> Option<std::cmp::Ordering> {
        Some(self.cmp(other))
    }
}
```

Figure 5.8 DijkstraState definition

The algorithm proceeds as standard Dijkstra over an implicit graph (edges are generated on demand rather than precomputed, since the full state space is too large to materialize upfront for any non-trivial input):

- Pop the lowest-cost state from the open set.
- If it is a goal state ($|S| = 1$), return its cost and path by the Dijkstra invariant, this is optimal, since every other state still in the open set has cost ≥ this one.
- If this exact multiset has already been finalized (present in closed), discard it as a cheaper or equal path to this state has already been explored.
- Otherwise, mark it closed, and generate successor states: for every ordered pair of (target, sacrifice) indices into the current multiset where combination is legal, compute the combined item and its cost, and push the resulting state onto the open queue with updated cost and path.

```rust
fn itemslice_to_btmap(items: &[Item]) -> BTreeMap<Item, usize> {
    let mut result = BTreeMap::new();
    for item in items {
        if result.contains_key(item) {
            *result.get_mut(item).unwrap() += 1;
        } else {
            result.insert(item.clone(), 1);
        }
    }
    return result;
}

fn remove_item_from_btmap(btmap: &mut BTreeMap<Item, usize>, item: &Item) {
    if *btmap.get(item).unwrap() == 1 {
        btmap.remove(item);
    } else {
        *btmap.get_mut(item).unwrap() -= 1;
    }
}

fn add_item_to_btmap(btmap: &mut BTreeMap<Item, usize>, item: Item) {
    if btmap.contains_key(&item) {
        *btmap.get_mut(&item).unwrap() += 1;
    } else {
        btmap.insert(item, 1);
    }
}
```

Figure 5.9 Auxiliary functions to help manipulate the item multiset

```rust
// Assumes first case: that there are no pair of enchantments from two items in the list that are
// incompatible (mutually exclusive) with each other.
fn solve(items: &[Item]) -> Option<(u32, Vec<(usize, usize)>)> {
    let mut open = BinaryHeap::new();
    open.push(DijkstraState {
        items: itemslice_to_btmap(items),
        cost: 0,
        path: vec![],
    });
    let mut closed = HashSet::new();

    while let Some(popped) = open.pop() {
        if popped.items.values().fold(0, |acc, x| acc + x) == 1 {
            dbg!(open.len(), closed.len());
            return Some((popped.cost, popped.path));
        }

        if closed.contains(&popped.items) {
            continue;
        }
        closed.insert(popped.items.clone());

        for (target_idx, (target, target_count)) in popped.items.iter().enumerate() {
            for (sacrifice_idx, sacrifice) in popped.items.keys().enumerate() {
                if target_idx == sacrifice_idx && *target_count == 1 {
                    continue;
                }

                if !target.can_combine(sacrifice) {
                    continue;
                }
                let comb_cost = target.combination_cost(sacrifice);
                if comb_cost > 39 {
                    // Anvil will not work
                    continue;
                }

                let mut new_items = popped.items.clone();

                remove_item_from_btmap(&mut new_items, target);
                remove_item_from_btmap(&mut new_items, sacrifice);

                let combined = target.combine(sacrifice);
                add_item_to_btmap(&mut new_items, combined);

                let mut new_path = popped.path.clone();
                new_path.push((target_idx, sacrifice_idx));

                open.push(DijkstraState {
                    items: new_items,
                    cost: popped.cost + comb_cost,
                    path: new_path,
                })
            }
        }
    }

    None
}
```

Figure 5.10 Implementation of Dijkstra's Algorithm

Because the state is a multiset with multiplicities rather than a flat list, a single item with `count == 1` must not be paired with itself as both target and sacrifice. This is the purpose of the `target_idx == sacrifice_idx && *target_count == 1` check. (Note: an item type with multiplicity ≥ 2, e.g. two identical books, can legitimately combine with "itself", in other words the check only forbids using the same single physical item as both the target and sacrifice in one operation.)

The path is stored not as item values but as index pairs into the multiset's key ordering at each step (`BTreeMap` iteration order is deterministic, sorted by Item's derived `Ord`). This keeps the `DijkstraState` cheap to clone when pushed onto the heap repeatedly. simulate_path replays these index pairs against the original input afterward to recover the actual sequence of item combinations for display, which is why it duplicates the same multiset bookkeeping (remove_item_from_btmap / add_item_to_btmap) as the solver while leveraging BTreeMap's deterministic iteration order for replaying.

```rust
fn simulate_path(items: &[Item], idx_path: &[(usize, usize)]) -> Vec<(Item, Item)> {
    let mut items = itemslice_to_btmap(items);
    let mut item_path = vec![];
    for (tidx, sidx) in idx_path {
        let mut new_items = items.clone();

        let target = items.keys().nth(*tidx).unwrap();
        let sacrifice = items.keys().nth(*sidx).unwrap();

        item_path.push((target.clone(), sacrifice.clone()));
        remove_item_from_btmap(&mut new_items, target);
        remove_item_from_btmap(&mut new_items, sacrifice);

        let combined = target.combine(sacrifice);
        add_item_to_btmap(&mut new_items, combined);

        items = new_items;
    }

    item_path
}

fn print_enchanted_item(item: &Item) {
    println!("{:?}", item.item_type);
    for e in &item.enchants {
        println!("- {:?} {}", e.enchantment_type, e.level);
    }
}

fn print_item_path(item_path: &[(Item, Item)]) {
    for (i, (t, s)) in item_path.into_iter().enumerate() {
        println!("Step {}", i + 1);
        print_enchanted_item(t);
        print_enchanted_item(s);
        println!("Cost: {}", t.combination_cost(&s));
        println!("AUC: {}", t.combine(&s).anvil_use_count);
        println!();
    }
}
```

Figure 5.11 Functions for replaying anvil combination sequence

Because all edge weights are non-negative, the first goal state popped from the priority queue is guaranteed optimal by Dijkstra's Algorithm and every remaining state in the open set may then be discarded. However, if the open set is exhausted without reaching a goal state, then solve returns None. This occurs when the input multiset cannot be fully merged in, for instance, the case of two books with no enchantments, since can_combine requires the sacrifice to have at least one enchantment, or if the individual merge XP cost requirement cannot always avoid veering above 39 thus making it impossible to combine on the anvil in survival mode.

VI. Results

Taking the example of a pair of unenchanted boots, 7 enchanted books with enchantments Soul Speed 3, Thorns 3, Feather Falling 4, Depth Strider 3, Protection 4, Unbreaking 3, Mending with each book holding one of the respective enchantments. The minimum XP cost required to combine all of these is 66 [4].

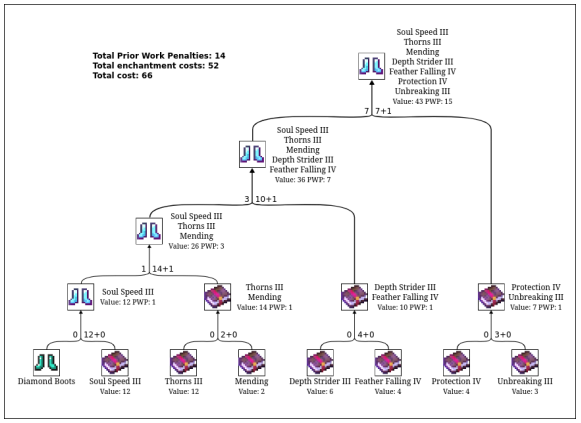


Figure 6.1 Optimal anvil combination sequence example, Source: [4]

```
Step 1
Book
- SoulSpeed 3
Book
- Unbreaking 3
Cost: 3
AUC: 1

Step 2
Boots
Book
- Thorns 3
Cost: 12
AUC: 1

Step 3
Boots
- Thorns 3
Book
- Unbreaking 3
- SoulSpeed 3
Cost: 17
AUC: 2

Step 4
Book
- FeatherFalling 4
Book
- Mending 1
Cost: 2
AUC: 1

Step 5
Book
- DepthStrider 3
Book
- Protection 4
Cost: 4
AUC: 1

Step 6
Boots
- Unbreaking 3
- Thorns 3
- SoulSpeed 3
Book
- Protection 4
- DepthStrider 3
Cost: 14
AUC: 3

Step 7
Boots
- Unbreaking 3
- Protection 4
- Thorns 3
- DepthStrider 3
- SoulSpeed 3
Book
- Mending 1
- FeatherFalling 4
Cost: 14
AUC: 4

Total cost: 66
```

Figure 6.2 Program terminal output

Although the solver obtained a different sequence of merges to Figure 6.1 as shown in Figure 6.2, the final XP cost sum still remains equal at 66 and thus proves this algorithm is correct.

There also exists a complete test suite available in the written code implementation (see Appendix) utilizing Rust's test cases [7] that among other things also cross checks the solver's results with the Minecraft Wiki's examples [4].

VII. Comparison to Prior Tools

Most public anvil calculators solve a narrower version of this problem. Tools like iamcal's enchant-order and the fast mode of kkchengaf's calculator start from one blank item plus a set of single-enchantment books, none of which have ever touched an anvil. That's a reasonable default for someone planning a fresh enchant from scratch, but it can't represent a sword that already has Sharpness III and Unbreaking II, or a book holding three enchantments at once.

Two tools come closer. The brute-force mode of kkchengaf's calculator accepts multiple items and books as input, including items with a nonzero prior-work penalty, computed by checking the rename cost in an anvil without actually renaming anything [5]. ViRb3's anvil-calc goes further: a Rust implementation compiled to WebAssembly that takes up to fifteen items or books, each with its own custom prior-work penalty and arbitrary enchantment set, including support for enchantments outside the vanilla list for modded play [6].

Both use depth-first search with branch-and-bound pruning rather than Dijkstra. The state is the same: a multiset of items still waiting to be merged. The search strategy differs in one respect that matters. DFS with branch-and-bound has to explore until the open set is empty, tracking a running best-known cost and discarding any branch once its partial cost exceeds that bound. Dijkstra, because every edge weight is non-negative, can stop the moment it pops a single-item state off the queue, since nothing still in the queue can possibly be cheaper. In practice this is less a question of one algorithm beating the other and more a question of search order: a priority queue explores cheapest-first, so it tends to tighten the best-known bound earlier, which gives branch-and-bound pruning more to work with sooner. With memoization on the closed set, the two approaches end up doing comparable work for the input sizes these tools target. The real gap between this implementation and the existing ones isn't the search algorithm. It's what counts as valid input. None of the surveyed tools take an item that's already enchanted and combine it with other already-enchanted items or multi-enchantment books while accounting for its existing anvil-use count. This implementation treats arbitrary pre-enchanted items, multi-enchantment books, and nonzero prior-work penalties as first-class inputs from the start, because the item representation, (type, enchantment set, anvil-use count), never assumes the item is freshly minted.

This paper only solves what the problem statement calls the first case: no two enchantments in the entire input multiset are mutually incompatible. The second case, where the multiset contains incompatible pairs like Silk Touch and Fortune, isn't a pure shortest-path problem anymore. It splits into two: first choose which enchantments to keep for the final item, then find the cheapest merge order for that choice. With k incompatible pairs, the number of valid retention choices grows combinatorially, and each one needs its own shortest-path search.

A few other costs are out of scope here. Renaming an item on an anvil costs 1 XP level on top of any enchantment cost, and this implementation doesn't account for it, on the assumption that renaming is usually optional and decided separately from enchantment planning. Repair cost, which depends on durability and material, is left out for the same reason: it's a cost a player can choose whether to pay, not a fixed part of reaching a target enchantment set on top of costing barely any [1][4].

There's also a practical scaling question. The closed set keeps the algorithm from repeating the same subproblem, but the number of distinct multisets reachable from n starting items still grows fast, since the search effectively explores something like the space of binary merge trees on n leaves before the closed set starts collapsing equivalent states together. For the examples tested here, ranging up to eight items, this stayed manageable. Whether the same approach holds up cleanly past fifteen or twenty items, where ViRb3's tool advertises support, is worth checking directly. Finally, the current implementation prefers to discard the highest enchantment levels obtainable in cases such as having a sword with Sharpness 4 and two books with Sharpness 3 (where the maximum level is 5): the current implementation may prefer to combine 4 + 3 to get 4 and again 4 + 3 after that to get a final level of 4 in order to minimize the total cost following the problem statement while as established in the introduction section, a higher level is often more favorable to a player.

VIII. Conclusion

This paper modeled anvil combination as a shortest-path problem over a state space where each state is a multiset of items and each edge is a single legal anvil combination weighted by its XP cost. Because that cost is always non-negative, Dijkstra's algorithm applies directly and finds the minimum-cost merge order, along with the sequence of combinations needed to reach it. The implementation handles inputs that existing public tools don't: items that already carry enchantments and a nonzero prior-work penalty, and books holding more than one enchantment at once. Test cases built from the Minecraft Wiki's worked examples, single combinations and full merge sequences alike, matched the expected costs, which is reasonable evidence the cost and combination logic match the game's actual rules. The main piece left out is the case where the input contains incompatible enchantments and a choice has to be made about which ones to keep. That turns a single shortest-path search into a search over which enchantments to retain, each retention choice spawning its own shortest-path problem underneath it. Solving that case fully is the natural next step for this line of work.

IX. Appendix

As an additional resource, the following is a Github Repository containing the code implementation of the algorithm in Rust. https://github.com/sbxte/anvil-calculator. The implementation used within this paper is available on the Git commit whose commit hash is f1be74dd (truncated), f1be74dd67b9d266ec2bf0ea27e94940b47adb5e (full).

References

[1] Mojang Studios, Minecraft: Java Edition, version 26.2. Stockholm, Sweden: Mojang Studios, 2026.

[2] Munir, Rinaldi. Accessed: 17 June 2026 [Online], http://informatika.stei.itb.ac.id/~rinaldi.munir/

[3] K. Mehlhorn and P. Sanders, Algorithms and Data Structures: The Basic Toolbox. Berlin: Springer, 2008, ch. 10.

[4] Minecraft Wiki, Anvil Mechanics, Accessed: 18 June 2026 [Online], https://minecraft.wiki/w/Anvil_mechanics/

[5] Kkchengaf, Enchantment Order Calculator, Accessed: 6 June 2026 [Online], https://kkchengaf.github.io/Minecraft-Enchantment-Order-Calculator/

[6] ViRb3, Anvil Calculator, Accessed: 6 June 2026 [Online], https://virb3.github.io/anvil-calc/

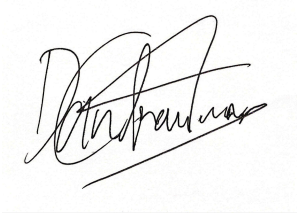
[7] Rust Foundation, Testing Attributes, Accessed: 19 June 2026 [Online], https://doc.rust-lang.org/reference/attributes/testing.html/

[8] Rust Foundation, Rust Standard Library, Accessed: 17 June 2026 [Online], https://doc.rust-lang.org/stable/std/

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 19 Juni 2026



Daniel Charisma Christian

13525052