THE MINIMUM CUT PITCH SPELLING ALGORITHM: SIMPLIFICATIONS AND DEVELOPMENTS

Benjamin Wetherfield
University of Cambridge
bsw28@cam.ac.uk

ABSTRACT

This paper describes and refines the Minimum Cut Pitch Spelling Algorithm, designed for flexible use in modern software and contemporary music settings. In the process of composing notated music, a decision must be made by the composer as to which enharmonic spelling should be assigned to each represented pitch. Spelling assignments in close proximity on the page are interrelated, with each choice exerting a pull on the surrounding choices. Hence, the complexity of the problem can proliferate, especially where tonal centers are contextually ambiguous or even non-existent. The minimum cut approach herein presents a model for spelling pitches efficiently based on their intervallic relationships. Building on the previous presentation of the model (in the author's bachelor's thesis), simplifications and extensions of both the workings of the algorithm and its exposition are given. Among the simplified components of the presentation are the system of encoding applied to pitch spellings, the approach taken to avoid double-accidentals, and a decoupling of the full complexity of the model from its simplest pitch relationships. A new practical inverting (or 'learning') process for generating algorithm parameters from collections of spelled pitches (based on the Edmonds-Karp Maximum Flow algorithm) is also introduced.

1. INTRODUCTION

In the process of composing music, pitch spellings make up a conduit for sound, an intermediate representation. Without due care, however, spellings can lead to a score that is difficult to read, rehearse or tune. Secondary as it is to musical sound, the assignment of "correct" spellings – or at least a good "default" – is an obvious candidate for automation in modern software. Various pitch spelling procedures have been proposed, with a number successfully reproducing the spellings of corpora of canonical works to a high degree of accuracy [1, 2, 3], but there are further measures of utility, and indeed further use cases, that can be applied to pitch spelling algorithms.

In [3], Honigh proposes four criteria for identifying a useful pitch spelling algorithm: *accuracy* in reproducing the spelling choices present in well-spelled scores, *parsimony* – a term introduced by Cambouropoulos to signify the avoidance of accidentals, and particularly doubled accidentals [4], *generalizability* – the value of the model in illuminating other features of music, and *cognitive plausibility* – the extent to which the pitch spelling approach mirrors the mental process of spelling done by a musician. To these criteria we can add several more. *Applicability* of the algorithm to different musical styles and contexts, *flexibility* of the algorithm to factor in user-preference and interaction, and *extensibility* of the algorithm to perform spellings with microtones. The latter three criteria are emphasized in the construction of the minimum cut pitch spelling algorithm, along with a reliance on the efficiency of algorithms studied in graph theory and operations research. A deep look at the construction of microtonal pitch spelling from the basic algorithm proves to be beyond the scope of this paper, but further details can be found in [5].

Despite seeking efficiency, this algorithm does not seek to be "real-time" in the sense that it could be used to perform real-time transcription or other stream-input/output operations. It should be noted that streaming pitch spelling algorithms are achieved elsewhere [2, 6]. Rather, the algorithm seeks to make strong use of a static score, taking advantage of the potential for pitches to exert a tug on each other forwards *and backwards* across the page. Interestingly, the static view allows us to abandon "windowing", a common feature in pitch spelling algorithms whereby a moving frame of reference restricts the pitches that can interact with each other as we move through the score from left to right. Instead, we make use of an extended network of pitches, the strength of whose connections is dampened over larger distances in the score.

In this paper, I will describe simplifications to previous modeling steps, which in turn clarify the exposition of the algorithm as a whole (see [5] for a comparison). In particular, I decouple musical context and proximity information from the pitch relationships governing spelling choices, leading to a further level of flexibility in allowing a user fine-grained control over the weight and adjacency structure of the pitch network. I examine inverting procedures, by way of which algorithm parameters can be extracted from spelled input data, and analyze a new inverse approach, which brings several practical benefits.

2. MINIMUM CUT

A **weighted directed graph** is a set of *nodes*, together with a set of *arcs* that denote one-way (directed) connections be-

↓	↑	**Encoded spelling**
0	0	Flattest spelling
1	1	Sharpest spelling
0	1	Natural spelling

Table 1. Encoding system for all pitch classes except 8, for which there is no 'natural' spelling (only G♯ and A♭), and hence, for which the $(0, 1)$ entry is removed (see Table 2).

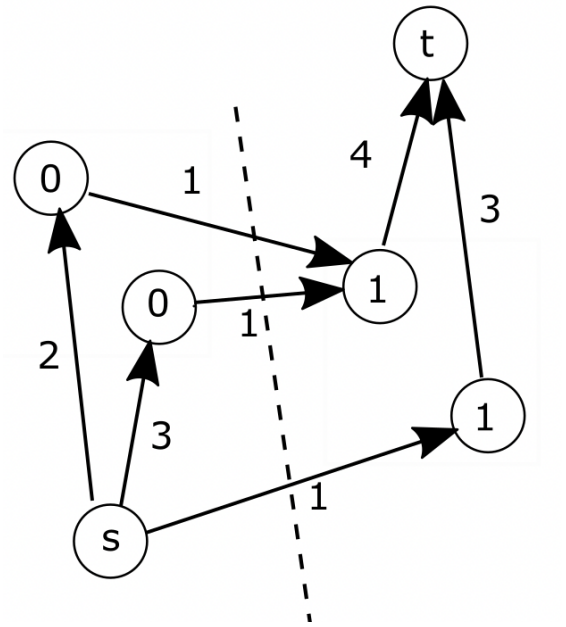

Figure 1. A flow network, with source s and sink t. The minimum cut is indicated with a dashed line. Numbers beside arcs indicate their weights. Numbers inside of nodes indicate on which side of the cut they fall: nodes with a 0 lie on the source side, while those with a 1 lie on the sink side.

tween nodes; each arc is given a numerical *weight*. A **flow network** denotes a weighted directed graph with two special nodes, a *source* and a *sink*. By convention, no arc starts at the sink and none ends at the source. Solving for the **minimum cut** of a flow network partitions the set of nodes into nodes strongly connected to the source, nodes strongly connected to the sink, and (possibly) nodes strongly connected to neither. The first class of nodes can all be assigned the value 0, the second class the value 1, and the third class arbitrarily 0 or 1 (assigned to the class in its entirety). The set of arcs that go from a node of value 0 to a node of value 1 is referred to as a cut. The cut is a *minimum* cut, when the total weight of its arcs is the least of all possible cuts in the network. Fast algorithms for solving for the minimum cut, either directly, or via the 'dual' maximum flow property, are well-known [7, 8, 9].

3. PITCH SPELLING MODEL

The strategy for constructing the pitch spelling model proceeds by encoding spelled pitches using a pair of binary

Pitch Class	Flattest Spelling	Sharpest Spelling	Natural Spelling
0	D𝄫	B♯	C♮
1	D♭	B𝄪	C♯
2	E𝄫	C𝄪	D♮
3	F𝄫	D♯	E♭
4	F♭	D𝄪	E♮
5	G𝄫	E♯	F♮
6	G♭	E𝄪	F♯
7	A𝄫	F𝄪	G♮
8	A♭	G♯	NONE
9	B𝄫	G𝄪	A♮
10	C𝄫	A♯	B♭
11	C♭	A𝄪	B♮

Table 2. Mapping from the 'Flattest', 'Sharpest' and 'Natural' signifiers in Table 1 to explicit spellings of pitch classes 0-11.

variables (or bits). The assignment of 0's and 1's to these bits arises from the solution to a carefully constructed minimum cut problem instance.

3.1 Binary Encoding of Spelled Pitches

Slightly adjusting the approach taken in [5], possible spellings for a given pitch can be encoded by a pair of binary variables. The system of encoding summarized in Table 1 provides a significantly simplified approach, reducing the rule-set needed from five tables of cases to a single table of the same size [5]. I will, moreover, diverge from [5] by referring to the two bits used to encode a spelled pitch respectively as ↓ ("down") and ↑ ("up").

3.2 Constructing a Flow Network

We start with a collection of pitches that we wish to spell. We split each pitch into a ↓ and ↑ bit, and for each bit, we introduce a corresponding node into the flow network. Ending at each ↓ node is an arc from the source, and starting at each ↑ node is an arc to the sink. Further arcs are drawn between internal (non-sink and non-source) nodes with weights based on rules addressed in Section 3.2.3. Weights are assigned to the arcs based on empirically derived relationships between pitch classes, explored heuristically below and in greater depth in Section 4. Solving for the minimum cut gives us a 0 or 1 assignment for each ↑ and ↓. Reading off the encoding system via the mapping in Table 1 gives us a spelling for each pitch.

3.2.1 Arc-Weight Heuristics

To enhance the intuitive intelligibility of the model, we can give a heuristic derivation of relationships required between arc weights. To start, for pitches that permit a 𝄫 spelling and no 𝄪 spelling (pitch classes 0, 3, 5, 10) we ensure that the weight of the arc from ↑ to the sink is heavier than the weight of the arc from the source to ↓ (see Fig. 2). This way, the 𝄫 spelling is the most costly option to include in a cut. Analogously, for pitches that permit a 𝄪 spelling

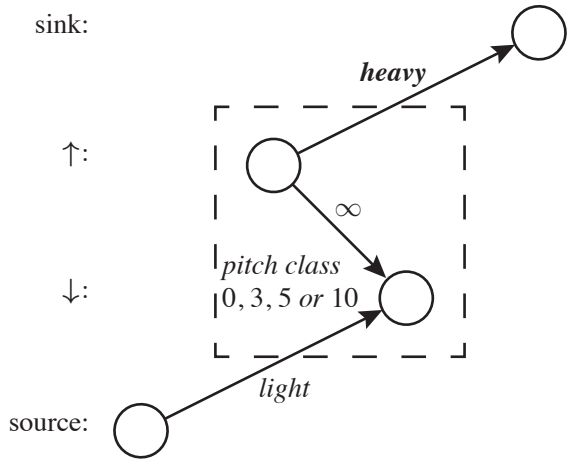

Figure 2. Visual representation of the relative arc weights needed in connecting the ↑ and ↓ nodes corresponding to pitches of class 0, 3, 5 or 10. The *heavy* and *light* arcs are discussed in Section 3.2.1, while the arc with weight ∞ is discussed in Section 3.2.2.

and no 𝄫 spelling (pitch classes 1, 4, 6, 11), we ensure that the weight of the arc from the source to ↓ is heavier than the weight of the arc from ↑ to the sink. For pitches that permit both a 𝄫 spelling and a 𝄪 spelling (pitch classes 2, 7, 9), we merely ensure that the weights from the source to ↓ and from ↑ to the sink are both relatively heavy so that the natural (indeed, ♮) spelling is almost always favored.

By penalizing double accidentals, we enforce the parsimony condition sought extensively in the literature. [1]

3.2.2 Avoiding Invalid Encodings

As Table 1 dictates, the assignment (↓: 1, ↑: 0) does not correspond to a valid spelling for any pitch. We add a further condition to our flow network construction to ensure that no pitch arrives at this state.

Between the ↑ and ↓ nodes of a *single pitch* in the flow network, we add an arc from ↑ to ↓ of weight 'infinity'. [2] Now if, for a given pitch, the ↑ node has state 0 and the ↓ has state 1, the total weight across the cut blows up (since the infinite weight gets included), and hence our cut is not minimal.

For pitches of pitch class 8 we can introduce an infinite weight arc in *both* directions, enforcing that the ↑ and ↓ nodes associated with the pitch class 8 pitch are always equal in value.

3.2.3 Connecting Internal Nodes

For each pair of pitches, we add arcs according to one of two cases.

1. We add arcs in both directions that connect the ↑ node of the first pitch to the ↑ node of the second pitch and likewise for the ↓ nodes.

[1] Penalizing accidentals directly in this way removes the need for a special 'parsimony pivot' as described in the original exposition [5].

[2] In our number system we include a value *infinity* that is greater than all numbers in the system but itself.

Pitch Class Pair	Lowest Cost Spellings
(0, 1)	(C♮, D♭) (B♯, C♯)
(0, 4)	(C♮, E♮)
(1, 3)	(C♯, D♯) (D♭, E♭)
(1, 5)	(C♯, E♯) (D♭, F♮)
(1, 10)	(C♯, B♮) (D♭, C♭)
(3, 4)	(D♯, E♮)
(3, 6)	(D♯, F♯) (E♭, G♭)
(3, 11)	(D♯, B♮) (E♭, C♭)
(4, 5)	(E♮, F♮)
(5, 6)	(E♯, F♯) (F♮, G♭)
(5, 11)	(F♮, B♮) (E♯, B♮) (F♮, C♭)
(6, 10)	(F♯, A♯) (G♭, B♭)
(10, 11)	(A♯, B♮) (B♭, C♭)

Table 3. Pitch class pairs for which ↑ node is connected to ↓ and vice versa.

2. We add arcs in both directions that connect the ↑ node of the first pitch to the ↓ node of the second pitch and vice versa.

Case 1 makes sense for all pairs of pitches where the respective natural spellings of the two pitches form a well-spelled interval and likewise for the respective sharpest spellings and flattest spellings (per Table 1). With appropriate weights added to the arcs, the ↓ nodes will tend to be coupled, and end up in the same state (unless there is a strong pull from another intervallic relationship) and likewise for the ↑ nodes.

Case 2 makes sense when we need to keep the pair of pitches away from problematic "natural" spellings per Table 2 (e.g. (C♮, C♯) as opposed to (C♮, D♭)). With the effects of sufficiently heavy source and sink incident arc weights, as discussed in Section 3.2.1, double spellings are avoided, and hence Table 3 gives the set of lowest cost spellings for pairs of pitch classes that should have ↑ nodes and ↓ nodes connected.

For pairs of pitch classes not listed in Table 3, case 1 applies (as can be checked exhaustively), and hence we connect the respective ↑ nodes to each other, and likewise for the respective ↓ nodes.

3.2.4 Phantom Pitches

To pull preferred spellings in one or another direction (more sharps, more flats, or more balance), we can add phantom pitches to the graph that do not come from the score in question, but nevertheless affect its spelling. The source and sink arcs of phantom nodes can be set to infinity so that these pitches are always spelled in their 'natural' form. Hence adding 0 as a phantom pitch would function the same as having an already-spelled C♮ among the unspelled pitches.

3.3 Injecting Musical Context

So far, we have only considered collections of pitches to spell in a musical vacuum, without temporal or cross-part information. This boils the problem of pitch spelling down

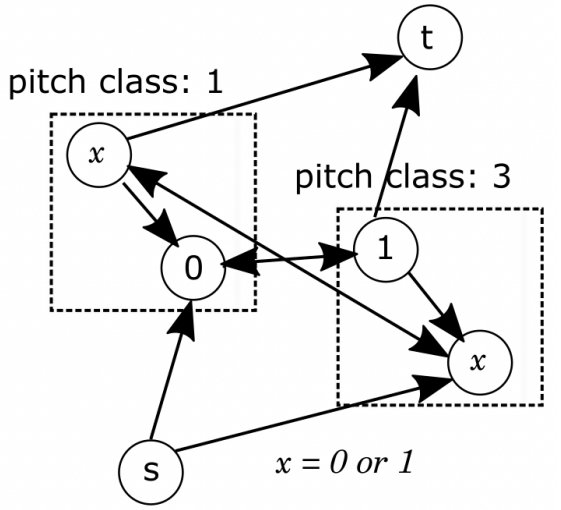

Figure 3. A fully connected network for spelling a pitch class 1 note against a pitch class 3 note. Arc weights need to be assigned such that the nodes fall on the sides of the cut indicated (as per Figure 1). The problem of finding the right arc weights is addressed in Section 4. When $x = 0$, the spelling (C♯, D♯) results. When $x = 1$, the spelling (D♭, E♭) results (cross-reference with Tables 1 and 2).

to its bare essentials of pitch relationships. But the model can also be easily be modified to factor in more contextual information, as would be appropriate in the spelling of a real score. By scaling arcs, we can increase or decrease the importance in certain interval relationships in determining the final spelling. In particular, scaling up the arcs between two pitches will increase the likelihood that those pitches are spelled harmoniously as an interval in a given context of other pitches.

3.3.1 Proximity Information

Pitches, as they occur in a score, can be independent from each other or exert a distant pull on each other. For pitches that should not be interrelated in terms of their spelling, [3] the arcs between their nodes can be removed from the flow network. For pitches that exert a distant pull, the arcs between their nodes can be scaled down appropriately, and indeed, adjacency in the score can be treated on a sliding scale whereby closer pitches have a larger scale factor less than 1 and more distant pitches have a smaller scale factor. For example, in [5], the adjacency scale factor – call it t_{pq} – for pitches in the same part, was implemented as an exponential factor for some base $\alpha > 1$:

$$t_{pq} = \alpha^{-(p-q)} \tag{1}$$

with $p \geq q \geq 0$ proportional to the sequential position of the two pitches in the score. A simple scale factor $0 < \beta < 1$ can also be added similarly, for pitches p and q that are simultaneous but in separate parts:

$$t'_{pq} = \beta \tag{2}$$

[3] Notes that should not be interrelated could include those either sids of a double barline, or simultaneous but in different sub-tonalities of a bitonal chord etc..

In a practical software implementation, adjacency information, or relevance of different pitches to each other across parts can be edited by a user, to provide an interactive, computer-*aided* spelling of a complex harmonic landscape. However, rules for removing arcs for pitches that should not exert an influence on each other are reasonable to generate. In [5], methods are proposed, such as treating double barlines and long rests as barriers between pitches, and removing any arcs for pitches on either side of these barriers.

4. INVERSE PROBLEM

While the existence of arcs based on pitch relationships can be determined heuristically, specific pitch-based weights can be determined through a solution to the *inverse problem*. The inverse problem poses the challenge of extracting arc-weight information from a collection of spelled pitches, rather than using the arc weights in the context of the flow network to derive assignments and hence spellings.

4.1 Full Complexity Solution

The original inverse problem solution is conceived by leveraging the primal-dual relationship of the maximum flow problem and the minimum cut problem. In linear optimization, a known result states that if a solution to the primal problem is equal in value to the value of a solution to the dual problem, then those solutions are optimal [10]. Both the maximum flow problem and the minimum cut problem can be stated as linear optimization problems, namely, with linear objective functions and linear constraints.

4.1.1 Defining Linear Constraints

For the purpose of defining the **maximum flow** problem, we assign to each arc a non-negative variable called the *flow* f_a of the arc a, bounded above by the weight w_a of a. The flow constraint for each node states that the sum of the flow variables for arcs going into the node should be equal to the sum of the flow variables for arcs going out from the node. The value of the total flow through the network as a whole is the sum of the flow variables for arcs going out from the source (or equivalently, going into the sink).

By satisfying flow constraints (in the maximum flow problem) and putting an upper bound B on the size of each pitch-based arc weight w_a, we can maximize the arc-weights to obtain a valid flow with non-zero arc-weights. By adding a duality constraint that sets the flow network's cut value equal to its total flow value, we ensure that the arc-weights found result in an optimal solution with pitches as they are found in the source score or corpus of scores.

From (3)-(8) we give a simplified statement of the constrained linear optimization labeled EXACTINVERSE in [5]. Contextual information, which can be injected into the inverse problem analogously to the approach discussed in Section 3.3, is omitted in this statement, for clarity and brevity. We define an *arc type* by the pitch classes and ↑/↓ types of its start and end node. To satisfy the model, we require a constraint, (6), that specifies that arcs with the

same type have the same weight. Meanwhile, the objective in (3) is to maximize the sum of weights w_a over all combinations of pitch classes and ↑/↓ start and end nodes, indexed by a. (4), (5), (7) and (8) express the flow, duality and bounding constraints straightforwardly.

$$\max_{w_a, f_a} \sum_{\text{arcs types } \alpha} w_\alpha \tag{3}$$

subject to

flow constraints:

$$\sum_{\text{ingoing arcs } a} f_a = \sum_{\text{outgoing arcs } b} f_b \quad \text{for each node} \tag{4}$$

$$0 \leq f_a \leq w_a \quad \text{for each arc } a \tag{5}$$

model constraints:

$$w_a = w_b \quad \text{up to context-based scaling}\,[4] \quad \text{for } a, b \text{ of the same arc type} \tag{6}$$

duality constraint:

$$\sum_{\text{source arcs: } s} f_s = \sum_{\text{cut arcs: } c} w_c \tag{7}$$

upper bound:

$$0 \leq w_a \leq B \quad \text{for each arc } a \tag{8}$$

[5] provides a mathematical proof that a solution of this optimization functions as an inverse to the pitch spelling problem; that is, once weights are extracted using the optimization described in (3)-(8), solving for the minimum cut *using those weights* will once again return the same set of spellings.

The simplex algorithm is, in practical computations, extremely efficient, and can be used to solve linear optimization problems, or linear programs, of the form of (3)-(8) [10]. Hence, in cases where the intervallic spelling rules of a score or corpus are consistent, we can compute a set of weights for constructing pitch spelling flow networks efficiently.

4.1.2 Robustness Through Approximation

[5] also provides an *approximate* solution to the inverse problem, as the existence of a set of weights consistent with the spelling of a given corpus of music is not guaranteed. By loosening the duality constraint and adjusting a constant λ to set how strict an inverse we seek, we can guarantee an output even for a corpus with inconsistent intervallic spelling.

$$\max_{w_a, f_a, \Delta} \sum_{\text{arcs types } \alpha} w_\alpha - \lambda\Delta \tag{9}$$

subject to

flow constraints:

$$\sum_{\text{ingoing arcs } a} f_a = \sum_{\text{outgoing arcs } b} f_b \quad \text{for each node} \tag{10}$$

$$0 \leq f_a \leq w_a \quad \text{for each arc } a \tag{11}$$

model constraints:

$$w_a = w_b \quad \text{up to context-based scaling}\,[5] \quad \text{for } a, b \text{ of the same arc type} \tag{12}$$

approx. duality:

$$\sum_{\text{source arcs: } s} f_s \geq \sum_{\text{cut arcs: } c} w_c - \Delta \tag{13}$$

upper bound:

$$0 \leq w_a \leq B \quad \text{for each arc } a \tag{14}$$

Indeed, the Δ value provides an interesting measure on *how consistent* in terms of the pitch spelling model the spelling of a corpus of music is. By adding a further constraint,

$$\Delta \leq pB \tag{15}$$

for some constant $0 < p < 1$, we can test whether the corpus of music can produce a duality of gap of less than or equal to pB, at which point, according to the terminology of [5] we could say that the corpus of music is 'p-consistent' in spelling. B is the upper bound on arc weights, as given in (15).

Constants such as λ, α and β in (1), (2) and (9) can be characterized empirically through repeated trials, using a systematic 'grid-search' method or random hyper parameter search method (see [11, 12]). Work is underway to characterize these constants empirically. [6]

4.2 Practical Inverse

In the production of practical software, a reliance on the simplex algorithm for inverse solutions can be limiting. The fastest simplex implementations tend to be commercial or, at the opposite extreme, use 'copyleft' licensing, with each paradigm placing restrictions on how the pitch spelling algorithm can be distributed and deployed downstream [13]. The optimization of a simplex algorithm, moreover, is highly technical, making it especially difficult to effectively implement without the required background in specialized operations research. In learning from a corpus of music, moreover, it is hard to guarantee that there is enough saturation of all the intervals to generate a representative set of arc weights. As a result, this paper proposes a more contained implementation for practically generating a set of arc weights with lower software production and data wrangling overheads.

[4] per Section 3.3

[5] per Section 3.3

[6] A work-in-progress Python implementation with these aims can be found here: https://github.com/bwetherfield/pitchspell

The principle of this inverse problem approach is to generate arc weights not from a corpus of scores, but rather from a collection of spelled dyads, triads and/or larger groups of pitches. Mirroring the context-free approach to pitch spelling in the 'forward' direction, we generate inverses only from pitch relationships.

We define an **unweighted network** to be a flow network where the arcs do not have weights. Using a collection of spelled groups of pitches, we can construct an unweighted network with the use of the pitch spelling model outlined in Section 3. Nodes are given values 0 or 1 according to the encoding rules in Tables 1 and 2. We stipulate the following condition.

Condition 1. Flattest spellings and sharpest spellings should never be adjacent in inputs to the practical inverse.

For example, we cannot feed in a chord containing both D♭ and B♯.

4.2.1 Adapting the Edmonds-Karp Algorithm

The Edmonds-Karp Algorithm solves for the maximum flow of a network. As we have noted, the maximum flow problem is the dual of the minimum cut, and, as such, the minimum cut solution can be obtained easily from a solved maximum flow. I will here give a brief description of the Edmonds-Karp algorithm and then explain how it can be modified for the purposes of performing the first part of an inverse solving approach.

In the Edmonds-Karp algorithm, we begin by finding the shortest path from source to sink such that all arcs in the path have *residual capacity*; the path must trace nodes connected in the network, either forward or backward across arcs. A backward arc has residual capacity if its current flow value is nonzero. A forward arc has residual capacity if its flow value is strictly less than its weight. Having found a path, we *push flow* through it. To push flow, we increase the flow of all forward arcs and decrease the flow of all backward arcs in the path by the same amount. In the algorithm, we push flow equal to the *minimum residual capacity* in the path, namely the minimum difference between the upper bound of a forward arc or the lower bound of a backward arc and its respective flow. Figure 4 shows a possible setup. When the flow of an arc equals its weight, we say the arc is *saturated*. When an arc is saturated, we remove it and insert a reversed arc of the same weight in its place. In the course of the algorithm, another shortest path with flow capacity is now found and we iterate until no shortest path with flow capacity remains. It can be shown that the process terminates after at most as many iterations as there are nodes [9].

FILLSTORAGE (Fig 5) is a modified Edmonds-Karp algorithm that operates on the unweighted network constructed in Section 4.2. By analogy with the Edmonds-Karp algorithm, we iterate on finding the shortest available path from source to sink. For each *arc type* (characterized by the pitch classes and ↑/↓ of its endpoints), we store a set of other arc types. Each arc type's weight will have to exceed or equal the weights of the arc types in its storage. We find the arc in the current path for which the start node has value 0 and the end node has value 1. As we will prove below

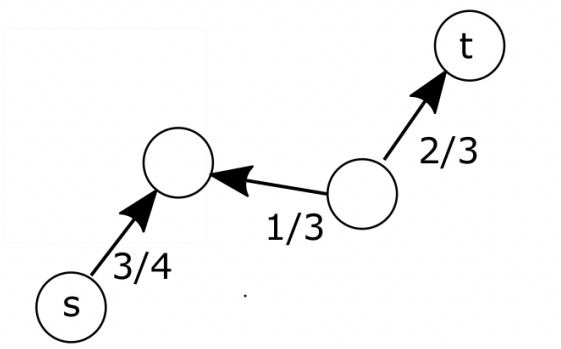

Figure 4. A path in a residual network. 3/4 represents a flow of 3 and a capacity of 4 for the given arc. The above path can have 1 unit of flow pushed through it, as each of the arcs has a residual capacity of 1. The first and last arcs will be saturated and replaced by reversed arcs with 0 flow, and respectively 4 and 3 as capacity.

(Claim 2), by the construction of the network, there will only be one such arc for each path found in the algorithm. For each path, we add this $0 \to 1$ arc's type to the STORAGE corresponding to all the other arc types present in the path. As in the Edmonds-Karp algorithm, we remove the 'saturated' arc and insert a reversed copy.

Claim 1. *The following arcs cannot be taken in paths found by the modified Edmonds-Karp algorithm:*

1. *↓ with value 1 to ↓ with value 0*
2. *↑ with value 1 to ↑ with value 0*
3. *↑ with value 1 to ↓ with value 0*
4. *↓ with value 1 to ↑ with value 0*

Proof.

1. There is a shorter path directly through the node with value 0.
2. There is a shorter path through the first node directly to the sink.
3. There is a shorter path through the first node directly to the sink.
4. The first node must belong to a (↑: 1, ↓: 1) encoding, and the second to a (↑: 0, ↓: 0), hence a sharpest spelling and a flattest spelling are connected, violating Condition 1.

□

Claim 2. *There is exactly one arc from a value 0 node to a value 1 node in each path found by the modified Edmonds-Karp algorithm described.*

Proof. No arc with values $1 \to 0$ can occur between internal nodes by Claim 1. Nor can a source or sink arc have weights $1 \to 0$ as source and sink always have value 0 and 1 respectively. Since there is no $1 \to 0$ arc in the path, and since a $0 \to 1$ arc must appear between source and sink, there is exactly one $0 \to 1$ arc in each path found by the algorithm. □

```
function FILLSTORAGE
    STORAGE(a) ← an empty set for each arc type a
    while p ← AUGMENTINGPATH do
        for a in arcs of p do
            if a is a "0 → 1" arc then
                remove a from the network
                insert REVERSED(a) into the network
                for b ≠ a in p do
                    append type of a to STORAGE(b)
                end for
            end if
        end for
    end while
    return STORAGE
end function

function AUGMENTINGPATH
    return shortest path from source to sink
end function
```

Figure 5. Modified Edmonds-Karp algorithm used to populate the storage assigned to each arc type present in the network.

4.2.2 Concrete Weight Generation Steps

For each arc we have stored the set of arcs on which its weight must depend. This forms a linked directed graph structure (where the arcs are playing the role of nodes in the new directed graph structure). In the MAIN function loop (Fig. 6), we can use standard graph algorithms to detect cycles of dependencies (for instance, a modified depth first search), and consolidate arcs in the storage into groups of connected components as needed (using Tarjan's algorithm) [7]. Now the storage is free of cyclic dependencies, we can recursively ensure each arc type is greater than the sum of all arc types in its storage, by adding 1 at each level of the recursion. If Tarjan's algorithm was called, we need to undo the grouping of arcs with UNDAGIFY, giving each arc in the same group the arc weight that was computed for that group. Since all arcs in the same group have the same arc weight, they are all greater than *or equal to* each other in weight, as needed.

4.2.3 Analysis of Correctness

We now imagine constructing a flow network from the arc weights derived in the inverse procedure. The flow network represents the same pitches in the same relationships, so it has the same adjacency structure as the unweighted network used to generate weights. We wish to show that it can generate a spelling consistent with the input spelled pitches.

To show the consistency of the flow network with the input pitches, note that we can run the Edmonds-Karp algorithm and follow the same sequence of paths found in the inverse procedure, which we will refer to as the *route*. If there are no cyclic dependencies in the storage structure, then the order of cut arcs being saturated and reversed follows the order of the inverse process, by construction. Hence the desired minimum cut is found.

```
function MAIN
    STORAGE ← FILLSTORAGE
    if DETECTCYCLE then
        DAGIFY
        GENERATEWEIGHTS
        UNDAGIFY
    end if
    GENERATEWEIGHTS
    return STORAGE
end function

procedure GENERATEWEIGHTS
    map over STORAGE calling WEIGHT on arcs, or arc groups if DAGIFY has been called
    function WEIGHT(a)
        if WEIGHT(a) has not already been called then
            w ← 0
            for b in STORAGE(a) do
                add WEIGHT(b) to w
            end for
            STORAGE(a) ← w + 1
        end if
    end function
end procedure

function DETECTCYCLE
    perform a cycle detection on STORAGE
end function

procedure DAGIFY
    group arcs in STORAGE that have cyclic dependencies using Tarjan's algorithm for finding strongly connected components
end procedure

procedure UNDAGIFY
    unwrap arc groups so that STORAGE maps from arcs to weights instead of arc groups to weights.
end procedure
```

Figure 6. The entire practical inverse procedure, including a call to the modified Edmonds-Karp algorithm.

When there is a cyclic dependency, we can show, by a contradiction argument that the forward solver will find at least one consistent minimum cut. If not, there would be a path in the faulty forward solution that contained an unseen 0→1 arc, which contradicts the construction of the inverse.

Informal experiments have shown that all three of the following methods can increase the robustness of an inverse solution, such that *all* minimum cuts that were fed in can be recovered with a forward solver.

1. Supplying more input spelled collections to the inverter, including, for example, all correctly spelled dyads.

2. "Warm starting" the STORAGE data structure with preset arc type dependencies before running FILLSTORAGE, using the arc heuristics in Section 3.2.1

to enforce the equality of certain pairs of "heavy" and "light" weights.

3. Supplying a suitable phantom pitch set to the forward problem instance (see Section 3.2.4).

4.2.4 Time Complexity

For the first part of the algorithm, the time complexity can be reduced to the time complexity of the Edmonds-Karp algorithm, which is $O(VE^2)$ – as it is described in this paper – where V denotes the number of nodes in the network, and E the number of arcs [9]. For the second part of the algorithm, note that the number of arc types is fixed, and so the number of steps needed is bounded (albeit by a large bound). Though at practical scales, the impact of the second half of the algorithm may be felt, its contribution in asymptotic terms is constant. Hence, we can fully characterize the complexity of the algorithm by that of Edmonds-Karp.

4.2.5 Simple Set of Results

Running the Practical Inverse procedure against an exhaustive list of well-spelled dyads, we obtain the arc weights laid out in tables 4, 5, 6 and 7. The exhaustive set of "good" input spellings contains all plausible spellings of semitones ([C, D♭], [C♯, D], and so on), tones ([C, D], [C♯, D♯] *and* [D♭, E♭], and so on), minor thirds, major thirds and perfect fourths *that do not contain any double accidentals*. [7]

4.2.6 Insufficiency of Results

It is worthy of note that Table 4, unlike Table 5, only features 0's and 1's. Where ↓ nodes are connected only to ↓ nodes and ↑ nodes are connected only to ↑ in the construction of a spelling network, the minimum cut set is *empty* as the source and sink are already strongly disconnected! Hence, the algorithm runs only trivially, adding 1 to internal connected arcs only once. Moreover, the (0, 0)-(11, 11) diagonal contains only 0's. The set of dyads used lacks unisons ([C, C], [C♯, C♯] and so on), to fill in this diagonal, in part to cope with limitations in the implementation. At best, with unisons included, however, we would have seen a diagonal full of 1's, which does not give proper weight to these pitched spelling relationships.

Intuitively, the size of the entries in Table 4 should reflect how important it is for pitch classes to be spelled in the same direction when side by side. Hence, it encodes the pull of pitch class 10 to B♭ when near a C♮, or conversely, to A♯ when near a B♯.

Analysis of these results demonstrate, therefore, the insufficiency of dyads alone for characterizing all arc weights in the inverse problem. The dyads prove adequate, however, for inducing differentiation for *half* of the internal arc weights, along with the outer weights to and from the source and sink nodes.

[7] These results were obtained using an open source Swift implementation hosted on github: https://github.com/bwetherfield/PitchSpellingModel

pc	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	1	1	0	1	0	1	0	1	1	1
1	0	0	1	0	1	0	1	0	1	1	0	1
2	1	1	0	1	1	1	1	1	0	1	1	1
3	1	0	1	0	0	1	0	1	0	0	1	0
4	0	1	1	0	0	0	1	1	1	1	0	1
5	1	0	1	1	0	0	0	1	0	1	1	0
6	0	1	1	0	1	0	0	1	1	1	0	1
7	1	0	1	1	1	1	1	0	0	1	1	1
8	0	1	0	0	1	0	1	0	0	1	0	1
9	1	1	1	0	1	1	1	1	1	0	1	1
10	1	0	1	1	0	1	0	1	0	1	0	0
11	1	1	1	0	1	0	1	1	1	1	0	0

Table 4. Empirical arc weights derived for $(\downarrow, \downarrow)$ arc types. The empirical results for $(\uparrow, \uparrow)$ arc types are identical, thanks to the symmetry of the input set of spellings.

pc	0	1	2	3	4	5	6	7	8	9	10	11
0	1	2	0	0	1	0	0	0	2	0	0	0
1	1	1	0	1	0	1	0	0	0	0	1	0
2	0	0	1	0	0	0	0	0	0	0	0	0
3	0	3	0	1	2	0	3	0	2	0	0	2
4	1	0	0	1	1	1	0	0	0	0	0	0
5	0	2	0	0	1	1	2	0	2	0	0	0
6	0	0	0	1	0	1	1	0	0	0	1	0
7	0	0	0	0	0	0	0	1	2	0	0	0
8	2	0	0	2	0	2	0	2	1	0	3	0
9	0	0	0	0	0	0	0	0	0	1	0	0
10	0	3	0	0	0	0	3	0	3	0	1	2
11	0	0	0	1	0	0	0	0	0	0	1	1

Table 5. Empirical arc weights derived for $(\downarrow, \uparrow)$ arc types. The empirical results for $(\uparrow, \downarrow)$ arc types are the exact matrix transposition (i.e. reflection along the (0,0)-(11,11) diagonal) thanks to the symmetry of the input set of spellings.

pc	0	1	2	3	4	5	6	7	8	9	10	11
	13	26	3	1	13	13	26	3	0	3	1	13

Table 6. Empirical arc weights derived for (source, ↓) arc types.

pc	0	1	2	3	4	5	6	7	8	9	10	11
	13	1	3	26	13	13	1	3	0	3	26	13

Table 7. Empirical arc weights derived for (↑, sink) arc types.

5. CONCLUSIONS AND FURTHER WORK

This paper has summarized the structure and composition of the minimum cut pitch spelling algorithm, while presenting some theoretical simplifications and extensions to the original exposition given in [5]. Clarifications to the system of encoding for pitch spellings, along with the de-

coupling of context information from the simple pitch relationships allow us to reason more directly about the core functionality of the algorithm. The new inverse problem approach, moreover, reduces the potentially unwieldy dependency on the simplex algorithm in generating sensible arc weights for the model.

There is still plenty of work to do before the algorithm presented is practically useful in production software. More heuristic methods, or larger spelling sets, are needed to populate the full set of pitch-based arc weights (in response to the limitations described in Section 4.2.6). Moreover, an empirical study on a large corpus of scores is still needed to tune hyper-parameters (as mentioned in Section 4.1.2), check the validity of heuristic measures (such as those described in Section 4.2.3) and compare the accuracy of this and other algorithms on corpora of canonical scores.

6. REFERENCES

[1] D. Meredith, "Comparing pitch spelling algorithms on a large corpus of tonal music," *Computer Music Modeling And Retrieval*, vol. 3310, pp. 173–192, 2005.

[2] E. Chew and Y.-C. Chen, "Real-Time Pitch Spelling Using the Spiral Array," *Computer Music Journal*, vol. 29, no. 2, pp. 61–76, June 2005.

[3] A. K. Honingh, "Compactness in the Euler-Lattice: A Parsimonious Pitch Spelling Model," *Musicae Scientiae*, vol. 13, no. 1, pp. 117–138, March 2009.

[4] E. Cambouropoulos, "A general pitch interval representation: Theory and applications," *Journal of New Music Research*, vol. 25, no. 3, pp. 231–251, September 1996.

[5] B. Wetherfield, "A graphical theory of musical pitch spelling," Bachelor's Thesis, Harvard University, 2017. [Online]. Available: https://dash.harvard.edu/handle/1/38779539

[6] D. Meredith, "Optimizing Chew and Chen's Pitch-Spelling Algorithm," *Computer Music Journal*, vol. 31, no. 2, pp. 54–72, June 2007.

[7] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM Journal on Computing*, vol. 1, no. 2, pp. 146–160, 1972. [Online]. Available: https://doi.org/10.1137/0201010

[8] D. S. Hochbaum, "The pseudoflow algorithm: A new algorithm for the maximum-flow problem," *Operations Research*, vol. 56, no. 4, pp. 992–1009, August 2008.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.

[10] D. Bertsimas, *Introduction to linear optimization*, ser. Athena Scientific series in optimization and neural computation. Belmont, Mass.: Athena Scientific, 1997.

[11] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly, B. Holt, and G. Varoquaux, "API design for machine learning software: experiences from the scikit-learn project," *CoRR*, vol. abs/1309.0238, 2013. [Online]. Available: http://arxiv.org/abs/1309.0238

[12] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization." *J. Mach. Learn. Res.*, vol. 13, pp. 281–305, 2012. [Online]. Available: http://dblp.uni-trier.de/db/journals/jmlr/jmlr13.html#BergstraB12

[13] J. L. Gearhart, K. L. Adair, R. J. Detry, J. D. Durfee, K. A. Jones, and N. Martin, "Comparison of open-source linear programming solvers."