A linear time algorithm for finding an Euler walk in a strongly connected 3-uniform hypergraph

Zbigniew Lonc[†] and Paweł Naroski[‡]

Faculty of Mathematics and Information Science, Warsaw University of Technology, Warsaw, Poland

received 7th October 2011, revised 19th April 2012, accepted 14th May 2012.

By an Euler walk in a 3-uniform hypergraph H we mean an alternating sequence $v_0, e_1, v_1, e_2, v_2, \ldots, v_{m-1}, e_m, v_m$ of vertices and edges in H such that each edge of H appears in this sequence exactly once and $v_{i-1}, v_i \in e_i$, $v_{i-1} \neq v_i$, for every $i = 1, 2, \ldots, m$. This concept is a natural extension of the graph theoretic notion of an Euler walk to the case of 3-uniform hypergraphs. We say that a 3-uniform hypergraph H is *strongly connected* if it has no isolated vertices and for each two edges e and f in H there is a sequence of edges starting with e and ending with f such that each two consecutive edges in this sequence have two vertices in common. In this paper we give an algorithm that constructs an Euler walk in a strongly connected 3-uniform hypergraph (it is known that such a walk in such a hypergraph always exists). The algorithm runs in time $O(m)$, where m is the number of edges in the input hypergraph.

Keywords: 3-uniform hypergraph, Euler walk, strongly connected hypergraph, linear time algorithm

1 Introduction

In this paper we consider a generalization of the graph-theoretic concept of an Euler walk to 3-uniform hypergraphs. This notion is well-understood from both theoretical and algorithmic point of view in the case of graphs. There are several possible ways to generalize this concept to the case of hypergraphs. The way we do it is motivated by some past research and potential applications.

By a *hypergraph* we mean a pair $H = (V, E)$, where V is a finite set and E is a family of some subsets of V. The elements of the set V are called *vertices* of H and the elements of E its *edges*. In this paper we deal with k-*uniform* hypergraphs only, i.e. hypergraphs whose all edges have the same cardinality equal to k. Clearly, 2-uniform hypergraphs are graphs.

By a *walk* (or a v_0v_ℓ-*walk*) of length ℓ in a hypergraph H we mean an alternating sequence

$$v_0, e_1, v_1, e_2, v_2, ..., v_{\ell-1}, e_\ell, v_\ell$$

of vertices and edges of this hypergraph satisfying the following conditions:

[†]zblonc@mini.pw.edu.pl
[‡]p.naroski@mini.pw.edu.pl

(i) the edges $e_1, e_2, \ldots, e_\ell$ are pairwise different,

(ii) $v_{i-1}, v_i \in e_i$, for $i = 1, ..., \ell$, and

(iii) $v_{i-1} \neq v_i$, for $i = 1, ..., \ell$.

A walk is called a *tour* if

(iv) $v_0 = v_\ell$.

We observe that the conditions (iii) and (iv) in the definition of a tour imply that $\ell \geq 2$. If, for some walk (respectively tour), $e_1, e_2, \ldots, e_\ell$ are all the edges of a hypergraph H, then we call such a walk (resp. tour) an *Euler walk* (resp. *tour*) in H. Let us observe that if H is a 2-uniform hypergraph (i.e. a graph), then the notions of an Euler walk and tour coincide with their traditional meanings in graph theory.

As we have already mentioned, there are several possible ways to extend the graph-theoretic concept of a tour to the case of hypergraphs. The tours we have just defined are most closely related to so-called Berge-cycles (see Berge [2], p. 155). More precisely, a tour defined in this paper in which all vertices $v_1, v_2, \ldots, v_\ell$ are pairwise different is a Berge-cycle.

It is natural to ask questions how fast we can check if an Euler walk and an Euler tour in a k-uniform hypergraph exist and, if they exist, how fast we can construct them. It is not surprising (and quite simple to prove; see [8]) that unlike in the case of graphs, the problems of existence of an Euler walk and tour in a k-uniform hypergraph are NP-complete, when $k > 2$.

However, if we restrict our attention to some important smaller classes of hypergraphs, the problems become tractable. A good example of such a class is the class of so-called triangulated irregular networks. Following the terminology we use in this paper, a *triangulated irregular network* (or a *TIN*) is a 3-uniform hypergraph whose edges are the vertex sets of all the triangular faces of some planar 2-connected graph in which all faces, except possibly the outer face, are triangles. TINs are structures widely used in geographic information systems (GIS) and computer graphics to represent terrains and surfaces (see for example [7]). An important issue addressed in these areas is to order triangles of a TIN into a sequence such that the consecutive triangles in this sequence share 1 or 2 vertices. Such ordering enables to compress the data and speeds up transmission and processing of triangulations. Clearly, we obtain a more efficient compression if consecutive triangles in the ordering share 2 vertices. However, it is not difficult to show that the problem of existence of such ordering for a TIN is NP-complete. Therefore, we relax the restriction and require that consecutive triangles share at least 1 vertex. This relaxation leads naturally to the concept of an Euler walk in a TIN (see [1] and [5] for a more thorough discussion of these issues).

Euler walks and tours in TINs have been intensively studied, in the context described in the preceding paragraph, in a series of papers by Bartholdi and Goldsman [3, 4, 5, 6] and Arkin *et al.* [1]. Among others it was shown in [5] that all TINs have Euler walks. Moreover a TIN has an Euler tour unless it belongs to some special family of exceptions (described in [5]). For TINs in which every edge has 2-element intersections with at least 2 other edges, the authors gave an algorithm constructing Euler tours. The running time of this algorithm is $O(m^2)$, where m is the number of edges in the TIN.

The problem of existence of an Euler walk in TINs has also been studied in mixed integer programming in connection with the problem of approximation of a two-variable function with a piecewise linear function (see Wilson [11] and Vielma *et al.* [10]).

In [8] we studied the problems of existence of Euler walks and tours in strongly connected k-uniform hypergraphs. A k-uniform hypergraph H is *strongly connected* if it has no isolated vertices and for each two edges e and f in H there is a sequence of edges starting with e and ending with f such that each two consecutive edges in this sequence have $k-1$ vertices in common. One can easily observe that the class of strongly connected 3-uniform hypergraphs contains the class of TINs as a proper subclass. For this reason this case (of $k=3$) seems to be more interesting that the case of $k>3$. Another reason is that, as we demonstrated in [8], for $k>3$ the problems of existence of an Euler walk or tour in a strongly connected hypergraph become trivial. Therefore in this paper we consider 3-uniform hypergraphs only. In [8] we characterized all strongly connected 3-uniform hypergraphs that have an Euler tour and proved that all 3-uniform strongly connected hypergraphs have an Euler walk.

In the present paper we describe an algorithm that constructs an Euler walk in a strongly connected 3-uniform hypergraph that runs in time $O(m)$, where m is the number of edges in the hypergraph. It improves over the already mentioned algorithm given in [5], designed for some special strongly connected TINs only, that runs in time $O(m^2)$.

Using similar ideas we can also design an algorithm constructing an Euler tour in a strongly connected 3-uniform hypergraph, whenever it exists (see Naroski [9]), that runs in time $O(m)$. The algorithm (which we do not discuss in this paper) is, however, much more technically complicated.

The paper is organized as follows. In Section 2 we present our algorithm and prove its correctness. All aspects regarding the data structures and the running time of our algorithm are discussed in Section 3. Finally in Section 4 we give some final remarks.

2 The algorithm and its correctness

Our algorithm (that we call $EulerWalk$) works in two main steps. First, we partition the edge set of the input strongly connected 3-uniform hypergraph into a certain number of walks such that all the walks, except possibly one, are tours (this is basically done by the algorithm $Dismantle$, see Lemma 2). Next, we glue the walks together and create a single walk containing all edges of the input hypergraph (this step is done by the algorithm $Walk$, see Lemma 4).

Let us introduce definitions and notation that we will use in this paper. For a hypergraph H, we denote by $E(H)$ the set of edges of H. Let, for a 3-uniform hypergraph H, $F(H)$ denote the set of all 2-element sets that are contained in at least one edge of H. Clearly, $|F(H)| \leq 3|E(H)|$. Let A be a set of 3-element sets. We say that a hypergraph is *induced by the set of edges* A if A is the edge set of the hypergraph and the union of the edges in A is its vertex set. For a hypergraph H and an edge f in H, we denote by $H-f$ the hypergraph induced by the set of edges $E(H)-\{f\}$.

By the *edge set of a walk* $W = v_0, e_1, v_1, e_2, v_2, ..., v_{\ell-1}, e_\ell, v_\ell$ we mean the set $\{e_1, e_2, \ldots, e_\ell\}$ and we denote it by $E(W)$. We say that a walk (or a tour) W *traverses an edge* e_i *using vertices* v_{i-1} *and* v_i if the sequence (v_{i-1}, e_i, v_i) is a subsequence of consecutive terms of the alternating sequence W. By a *partition* of the edge set of a hypergraph H *into walks* we mean a partition of the edge set of H into subsets which are edge sets of some walks.

For a 3-uniform strongly connected hypergraph H we define a graph $G(H)$ whose vertex set is the set of edges of H. The set of edges of $G(H)$ consists of pairs of edges of H which intersect on 2 elements. As H is strongly connected, the graph $G(H)$ is connected. We would like to construct a depth-first tree in $G(H)$ (starting with an arbitrary vertex e in $G(H)$). However, we do not want to construct the graph $G(H)$ explicitly because the number of edges in $G(H)$ may be quadratic with respect to $m=|E(H)|$. The algorithm $DF\text{-}tree(H,e)$ described below constructs a depth-first tree T_e in $G(H)$ without explicitly constructing $G(H)$. We shall prove in Section 3 that the algorithm $DF\text{-}tree(H, e)$ runs in time $O(m)$ if we choose an appropriate data structure to represent the hypergraph H.

In the algorithms $DF\text{-}tree$ and $Dismantle$ discussed beneath, S is a stack. We use standard stack operations $push(S, g)$ to insert an element g onto S, $pop(S)$ to remove the top element from S, and $top(S)$ to return the top element of S without removing it from S. We also use a Boolean array $visited$ indexed with all edges of H. If, for some edge f, $visited[f] =$ **true**, then we call the edge f *visited*. Otherwise, when $visited[f] =$ **false**, we call f *unvisited*. By E_e we denote the set of edges of the tree constructed by the algorithm $DF\text{-}tree$.

Algorithm $DF\text{-}tree(H, e)$
1 **for** every edge f in H **do** $visited[f] \leftarrow$ **false**
2 $S \leftarrow \emptyset$
3 $E_e \leftarrow \emptyset$
4 $push(S, e)$
5 $visited[e] \leftarrow$ **true**
6 **while** $S \neq \emptyset$
7 **do** $f \leftarrow top(S)$
8 **if** $|f \cap g| = 2$, for some edge g such that $visited[g] =$ **false**
9 **then** $push(S, g)$
10 $visited[g] \leftarrow$ **true**
11 $E_e \leftarrow E_e \cup \{fg\}$
12 **else** $pop(S)$
13 **return** the tree T_e induced by the set of edges E_e

Clearly, line 8 of the algorithm $DF\text{-}tree$ can be equivalently written as follows

8' **if** fg is an edge in $G(H)$, for some edge g such that $visited[g] =$ **false**

so $DF\text{-}tree$ is a standard algorithm constructing a depth-first tree (rooted in e) in $G(H)$.

Lemma 1 *Let f be a vertex in the tree T_e constructed by the algorithm $DF\text{-}tree(H, e)$ and let f_1, f_2 be two different children of f in T_e. Then $|f_1 \cap f_2| = 1$.*

Proof: As both ff_1 and ff_2 are edges in T_e, $|f \cap f_1| = |f \cap f_2| = 2$, so $|f_1 \cap f_2| \geq 1$. Suppose the lemma is not true, so $|f_1 \cap f_2| = 2$ and, consequently, f_1f_2 is an edge in $G(H)$. Clearly, the vertex f had been visited before f_1 and f_2 were visited in the execution of the algorithm $DF\text{-}tree(H, e)$. We can assume without loos of generality that f_1 had been visited before f_2 was visited. As f_1f_2 is an edge in $G(H)$, the vertex f_2 was visited while f_1 was still at the stack S (so in particular f was not at the top of S). Thus, f_2 is not a child of f in T_e, a contradiction. Hence, $|f_1 \cap f_2| = 1$. □

As every child of a vertex f in T_e is a 3-element set sharing 2 elements with f, Lemma 1 immediately implies the following statement.

Corollary 1 *Each vertex in the tree T_e has at most* 3 *children.* □

Let H be a 3-uniform strongly connected hypergraph and let e be an edge in H. The algorithm $Dismantle(H, e)$, that we give below, constructs a partition of the edge set of H or $H - e$ into tours. It will turn out that the tours in the partition are in fact short (each of them has 2 or 3 edges).

In the algorithm $Dismantle$, $\mathcal{C}$ is a list of tours in H. Let $\{e_1,\dots,e_k\}$ be the edge set of some walk in H. By $L(e_1,\dots,e_k)$ we denote any walk (of possibly many) whose edge set is $\{e_1,\dots,e_k\}$.

Algorithm $Dismantle(H,e)$
1 $T_e \leftarrow$ $DF\text{-}tree(H,e)$
2 **for** every vertex f in T_e **do** $visited[f] \leftarrow$ **false**
3 $\mathcal{C} \leftarrow \emptyset$
4 $S \leftarrow \emptyset$
5 $push(S,e)$
6 $visited[e] \leftarrow$ **true**
7 **while** $S \neq \emptyset$
8 **do** $f \leftarrow top(S)$
9 **if** $visited[g] =$ **false**, for some child g of f in T_e
10 **then** $push(S,g)$
11 $visited[g] \leftarrow$ **true**
12 **else** **if** f has 3 children f_1, f_2, f_3 and all of them are leaves in T_e
13 **then** $\mathcal{C} \leftarrow \mathcal{C} \cup \{L(f_1,f_2,f_3)\}$
14 remove vertices f_1, f_2, f_3 from T_e
15 **if** f has 2 children f_1, f_2 and both of them are leaves in T_e
16 **then** $\mathcal{C} \leftarrow \mathcal{C} \cup \{L(f_1,f,f_2)\}$
17 remove vertices f_1, f, f_2 from T_e
18 **if** f has 1 child f_1 and it is a leaf in T_e
19 **then** $\mathcal{C} \leftarrow \mathcal{C} \cup \{L(f_1,f)\}$
20 remove vertices f_1, f from T_e
21 $pop(S)$
22 **return** $\mathcal{C}$

Lemma 2 *Let H be a strongly connected 3-uniform hypergraph and e an edge in H. When the algorithm $Dismantle(H,e)$ stops, the list $\mathcal{C}$ consists of tours whose sets of edges form a partition of the edge set of H or $H-e$.*

Proof: We observe that the list $\mathcal{C}$ is updated in lines 13, 16 and 19 of the algorithm only. It follows from Lemma 1 that the edges f_1, f_2, f_3 of H defined in line 12 form a tour. Similarly, by Lemma 1, the edges f_1, f_2 defined in line 15 (resp. f_1 defined in line 18) and f form a tour in H (see Figure 1). Thus, the elements added to $\mathcal{C}$ in lines 13, 16, and 19 are tours. Consequently, the algorithm returns a list of tours in H.

It is clear that the tours stored in the list $\mathcal{C}$ do not have common edges because in the algorithm $Dismantle(H,e)$ right after adding a tour to the list $\mathcal{C}$ the edges that form the tour are removed from the set of vertices of T_e (lines 14, 17 and 20).

It suffices to show that all vertices of T_e (which are edges of H), except possibly its root e, belong to some tour in the list $\mathcal{C}$ when the algorithm stops. Let us denote by E the set of edges of H that do not belong to any tour in the list $\mathcal{C}$ when the algorithm stops and assume that $E-\{e\} \neq \emptyset$. We notice that the set of vertices not removed from the tree T_e when the algorithm stops is equal to E too (see lines 13-14, 16-17 and 19-20). Next, we observe that if a vertex is removed from T_e, then all its children are removed (see lines 12-20). Thus, if some vertex f of T_e is in the set E, then so is its parent. Consequently, the vertices of T_e which belong to E form a subtree T'_e of T_e rooted at e. The tree T'_e has a vertex different from e because $E - \{e\} \neq \emptyset$. Let f be a lowest leaf in T'_e and let f_0 be its parent. Obviously, each vertex of T_e is inserted onto the stack S exactly once and removed from S exactly once. Let us consider the pass of the **while** loop when the vertex f_0 is removed from S. One can easily observe that at this moment all children of f_0 are leaves in the current tree T_e. As f_0 has at least one child (f is child of f_0), one of the conditions in lines 12, 15 or 18 is satisfied. Consequently, f is removed from T_e in this pass of the **while** loop, so $f \notin E$, a contradiction. Thus, $E - \{e\} = \emptyset$ and the lemma holds. □

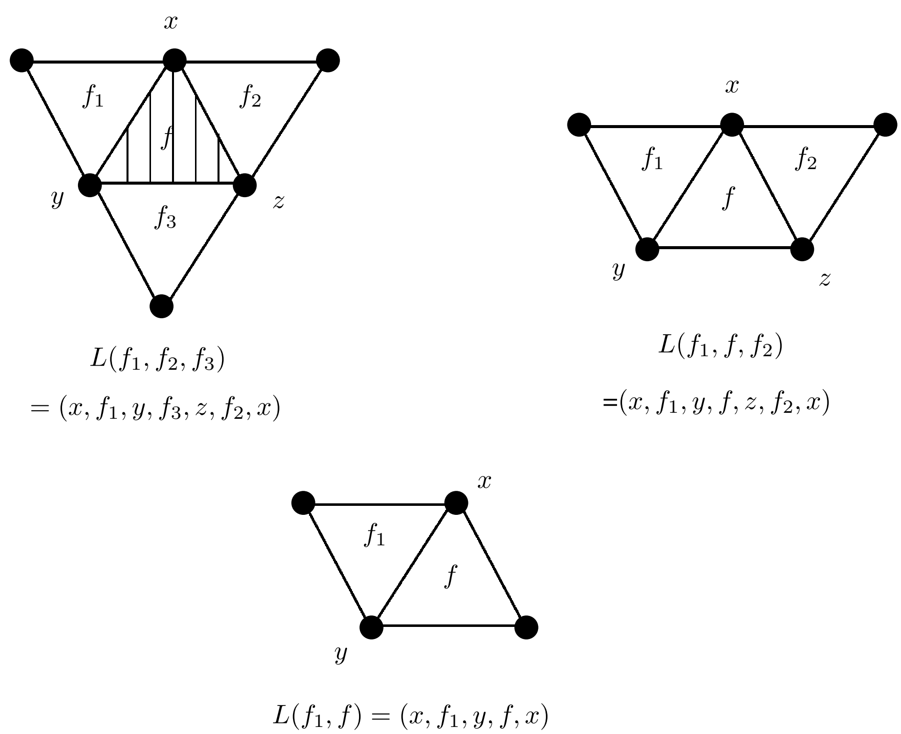

Fig. 1: Tours added to the list $\mathcal{C}$ in lines 13, 16, and 19. The shaded edge f does not belong to the tour.

The algorithm $Walk$ that we will describe later constructs an Euler tour (or walk) in a strongly connected hypergraph from an already constructed partition of the edge set of the hypergraph into tours (or tours and one walk). In this algorithm we will need an auxiliary algorithm $Connect$. The input to the algorithm $Connect$ consists of a walk P, a tour C, an edge e in P, and an edge f in C. The walk P and the tour C are edge-disjoint and the edges e and f intersect on a 2-element set. The algorithm $Connect$ produces a walk W whose edge set is the union of edge sets of the walk P and the tour C.

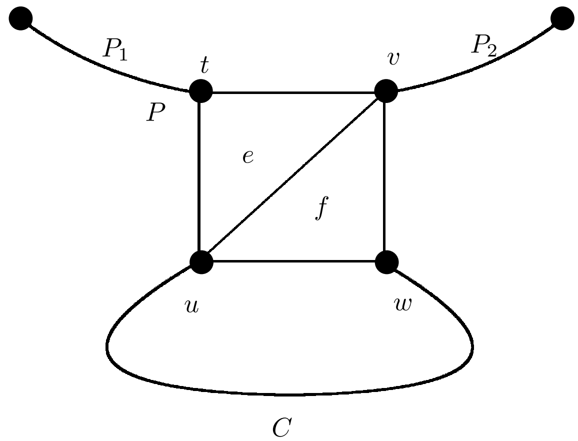

Fig. 2: Construction of the walk W in the option **else** of the algorithm $Connect$.

Algorithm $Connect(P, e, C, f)$
1 **if** there is a vertex x such that P traverses the edge e using x
and C traverses the edge f using x
2 **then** $P_1 \leftarrow$ the part of P from one of its ends to x
3 $P_2 \leftarrow$ the remaining part of P from x to the other end of P
4 $W \leftarrow (P_1, x, C, x, P_2)$
5 **else** $u \leftarrow$ the vertex of e, which is not used by P while traversing e
6 $v \leftarrow$ the vertex of f, which is not used by C while traversing f
/⋆ u and v are the common vertices of the edges e and f ⋆/
7 $t \leftarrow$ the vertex of e different from u and v
8 $w \leftarrow$ the vertex of f different from u and v
9 $P_1 \leftarrow$ the part of P from one of the ends of P to t
obtained by removing e from P (see Figure 2)
10 $P_2 \leftarrow$ the part of P from one of the ends of P to v
obtained by removing e from P (see Figure 2)
11 $C' \leftarrow$ the walk from u to w obtained from C by removing f
12 $W \leftarrow (P_1, t, e, u, C', w, f, v, P_2)$
13 **return** W

Lemma 3 *Let P be an ab-walk and C a tour in a hypergraph H. If the sets of edges of P and C are disjoint and there exist edges $e \in E(P)$ and $f \in E(C)$ which have 2 vertices in common, then the algorithm $Connect(P, e, C, f)$ returns an ab-walk in H whose edge set is $E(P) \cup E(C)$.*

Proof: It suffices to observe that the sequences assigned to the variable W in lines 4 and 12 of the algorithm $Connect(P, e, C, f)$ are ab-walks with the edge set $E(P) \cup E(C)$. □

The input $\mathcal{C}$ of the algorithm $Walk$ written below is a list of walks whose sets of edges form a partition of the edge set of a strongly connected hypergraph. At most one of the walks is not a tour. We shall show (see Lemma 4) that this algorithm returns a walk whose edge set is the union of the edge sets of the walks in $\mathcal{C}$.

Algorithm $Walk(\mathcal{C})$
1 $P \leftarrow$ the only walk in $\mathcal{C}$ which is not a tour, if it exists, or any element
  of $\mathcal{C}$ otherwise
2 $\mathcal{C} \leftarrow \mathcal{C} - \{P\}$
3 **while** $\mathcal{C} \neq \emptyset$
4   **do** Find a pair of edges e and f such that e is an edge of P, but
      f is not an edge of P and $|e \cap f| = 2$.
5   Find a tour $C \in \mathcal{C}$ containing f.
6     $P \leftarrow Connect(P, e, C, f)$
7     $\mathcal{C} \leftarrow \mathcal{C} - \{C\}$
8 **return** P

Lemma 4 *Let H be a strongly connected hypergraph and $\mathcal{C}$ a list of walks whose sets of edges form a partition of the edge set of H. At most one of the walks in $\mathcal{C}$ is an ab-walk which is not a tour. The algorithm $Walk(\mathcal{C})$ returns an Euler ab-walk in H. This walk is an Euler tour in H, if the list $\mathcal{C}$ contains tours only.*

Proof: Let a and b be the ends of the walk assigned to the variable P in line 1 of the algorithm. (If this walk is a tour, then $a = b$ is any vertex of this tour.) After executing of lines 1-2 of the algorithm $Walk(\mathcal{C})$, the list $\mathcal{C}$ contains tours in H only and the variable P contains a walk in H (which may be a tour too).

At this moment the edge sets of the tours in the list $\mathcal{C}$ together with the edge set of P form a partition of the edge set of H and P is an ab-walk. We claim that this property is an invariant of the **while** loop in lines 3-7. We observe that as long as the list $\mathcal{C}$ is nonempty, the instructions in lines 4-5 are feasible. Indeed, let $E(P)$ be the set of edges of P and let $E(\mathcal{C})$ be union of the sets of edges of the walks in the list $\mathcal{C}$ at start of some pass of the **while** loop. As $E(P) \cup E(\mathcal{C}) = E(H)$, by the strong connectivity of H, there is a pair of edges $e \in E(P)$ and $f \in E(\mathcal{C})$ such that $|e \cap f| = 2$. So, the instructions in lines 4-5 are feasible as long as $\mathcal{C} \neq \emptyset$. By Lemma 3, after execution of line 6, the variable P contains an ab-walk whose edge set is the sum of the edge sets of the walk which was the value of P right before execution of this line and the edge set of the tour C. Since the tour C is removed from the list $\mathcal{C}$ in line 7, right after execution of the analyzed pass of the **while** loop, the edge sets of the tours in the updated list $\mathcal{C}$ together with the edge set of the updated walk P form a partition of the edge set of H. This completes the proof of our claim.

At the end of the last pass of the **while** loop the list $\mathcal{C}$ is empty, so by the claim proved in the preceding paragraph, the walk P returned by the algorithm $Walk$ is an ab-walk containing all edges of H, so it is an Euler ab-walk in H. If, at start of the algorithm, the list $\mathcal{C}$ contains tours only, then $a = b$, so the algorithm returns an Euler walk. □

We are ready now to give an algorithm which constructs an Euler walk in a strongly connected hypergraph. Recall that, for a single edge e, by $L(e)$, we mean any walk whose edge set is $\{e\}$.

Algorithm $EulerWalk(H)$

$e \leftarrow$ any edge of H
$\mathcal{C} \leftarrow Dismantle(H, e)$
if the edge e is in none of the tours in $\mathcal{C}$ **then** $\mathcal{C} \leftarrow \mathcal{C} \cup \{L(e)\}$
return $Walk(\mathcal{C})$

Theorem 1 *If H is a strongly connected 3-uniform hypergraph, then the algorithm $EulerWalk(H)$ returns an Euler walk in H.*

Proof: The theorem follows immediately from Lemmas 2 and 4. □

3 The running time

We shall first discuss the running time of the algorithm $Dismantle$. We start with defining a data structure to represent the input hypergraph.

Let H be a 3-uniform hypergraph. For every $c \in F(H)$, we construct a doubly linked list $I[c]$ of edges of H which contain c. Clearly, each edge occurs in three different such lists. We assume that the occurrences of the same edge in different lists are appropriately joint by pointers to enable removal of the edge from all three lists in a constant time. More precisely, if we want to remove an edge, say e, from a list, we first find using the pointers (in a constant time) the remaining two occurrences of e in the lists. To remove the edge e from each of these list, it suffices to redirect appropriately the links of the predecessor and the successor of e. It can be done in a constant time as the lists are double linked. The lists $I[c]$ can be easily constructed in time $O(m)$, where $m = |E(H)|$, if H is given in any standard way (e.g. by a list of edges or by incidence lists, i.e. lists defined for every vertex and consisting of all the edges containing this vertex).

Lemma 5 *The algorithm $DF\text{-}tree$ can be implemented to run in time $O(m)$, where m is the number of edges in the input hypergraph.*

Proof: We assume that the input hypergraph H is represented by the lists $I[c]$, for every $c \in F(H)$.

Clearly, the number of passes of the loop in line 1 is equal to m and each pass of this loop takes a constant time. In each pass of the **while** loop either one edge is inserted onto S (see line 9) or one is removed (see line 12). Thus, as the graph $G(H)$ is connected, the number of passes of the loop in lines 6-12 is $2m - 1$ because each edge of H is one time inserted onto S and one time removed.

In each pass of this loop checking if the condition in line 8 is satisfied and finding the appropriate edge g can be implemented to run in a constant time by using the lists $I[c]$, $c \in F(f)$. After finding an unvisited edge g in one of these three lists $I[c]$, we remove the edge g from the lists containing it. This way each time the condition in line 8 is checked for an edge f, each list $I[c]$, $c \in F(f)$, contains unvisited edges only (or is empty). So, every execution of line 8 takes a constant time. It is quite straightforward to check that all the remaining instructions inside the loop in lines 6-12 take constant times.

Thus, the running time of the algorithm $DF\text{-}tree$ is indeed $O(m)$. □

Lemma 6 *The algorithm $Dismantle$ can be implemented to run in time $O(m)$, where m is the number of edges in the input hypergraph.*

Proof: By Lemma 5, line 1 of the algorithm $Dismantle$ runs in time $O(m)$. Moreover by the same argument as in the proof of Lemma 5, we show that the number of passes of the loop in lines 7-21 of the algorithm $Dismantle$ is $2m-1$. Each instruction in this loop takes a constant time (for the instruction in line 9 it follows from Corollary 1), so the running time of the algorithm $Dismantle$ is indeed $O(m)$. □

Let $\mathcal{C}$ be the input list for the algorithm $Walk$ and let m be the number of edges in the strongly connected hypergraph induced by the union of the edges of the walks in the list $\mathcal{C}$. We shall show now that the algorithm $Walk$ can be implemented to run in time $O(m)$. We represent walks as doubly linked lists and tours as doubly linked cyclic lists of consecutive vertices and edges of these walks or tours. To reach a linear running time of the algorithm, we need a data structure which, for a given edge and a given partition of the edge set of a hypergraph into walks, enables us to find in a constant time the unique walk containing this edge and the position of this edge in this walk. Therefore, for a partition $\mathcal{C}$ of the set of edges of a hypergraph into walks, we define an array $\mathcal{J}_{\mathcal{C}}$ indexed with edges of the hypergraph. For every edge e of the hypergraph, $\mathcal{J}_{\mathcal{C}}[e] = (W_e, p_e)$, where W_e is the walk in $\mathcal{C}$ which contains the edge e and p_e is a pointer to the edge e on the list representing the walk W_e.

Lemma 7 *Let H be a strongly connected hypergraph and $\mathcal{C}$ a list of walks whose sets of edges form a partition of the edge set of H. Moreover, at most one of the walks in $\mathcal{C}$ is not a tour. The algorithm $Walk(\mathcal{C})$ can be implemented to run in time $O(m)$, where m is the number of edges in H.*

Proof: We assume that the hypergraph H is represented by the lists $I[c]$, for every $c \in F(H)$. At start of the algorithm we create the data structure $\mathcal{J}_{\mathcal{C}}$. Clearly, it can be done in time $O(m)$.

Obviously, finding the walk in line 1 of the algorithm $Walk(\mathcal{C})$ takes $O(m)$ time and executing the line 2 can be implemented to run in a constant time.

We observe that in each pass of the **while** loop exactly one walk is removed from the list $\mathcal{C}$, so the number of passes of this loop is bounded by m. After removing a walk from the list $\mathcal{C}$ (in lines 2 and 7), we shall delete the edges of this walk from all the lists $I[c]$, $c \in F(H)$. Since every edge of H belongs to exactly one walk in $\mathcal{C}$, the total number of such deletions is bounded by $3m$. Due to these removals, at start of every execution of the **while** loop, the lists $I[c]$ contain only the edges of the hypergraph H which do not belong to the current walk P.

We shall show now that the total time needed for all executions of line 4 of the algorithm $Walk$ is $O(m)$. We use the observation made in the preceding paragraph that e and f is a pair of edges described in line 4 of the algorithm if and only if $|e \cap f| = 2$ and at the moment of execution of line 4, f is in one of the lists $I[c]$, where $c \in F(e)$, and e is an edge of the walk P. Thus, in each execution of line 4, we search for an edge e in the walk P such that for some $c \in F(e)$, the list $I[c]$ is nonempty and if it is, we choose f to be the first edge in this list. We do this search for such an edge e in the following order. First we check the edges of the walk assigned to the variable P in line 1, next the edges added to the walk P (in line 6) in the first pass of the **while** loop, next the edges added to the walk P in the second pass of the loop, etc. We observe that, if for an edge e' in P, $I[c] = \emptyset$, for some $c \in F(e')$, then $I[c] = \emptyset$ in all executions of line 4 in the next passes of the **while** loop. Therefore, in executions of line 4 in each of the remaining passes of this loop, we do not check such lists $I[c]$ any more. We start the search for an edge e satisfying the condition in line 4 from the next list $I[c]$, for $c \in F(e')$, or from a list $I(c')$, for $c' \in F(e'')$, where e'' is the next edge in our order, if all the lists $I[c]$, for $c \in F(e')$, are empty. This way no element in the lists $I[c]$, $c \in F(H)$, is checked more than once. As the sum of numbers of elements in the lists $I[c]$ is $3m$, the amount of time needed for all executions of line 4 in the algorithm is $O(m)$.

Let us consider now line 6 of the algorithm $Walk$ which contains a call of the algorithm $Connect$. The data structure $\mathcal{J}_\mathcal{C}$ allows us to find all the vertices (x, u, v, t, and w) needed in the procedure $Connect$ in a constant time. The construction of the walk W in lines 4 and 12 of the algorithm $Connect$ can be done by redirecting a constant number of pointers in the linked lists representing the walk P and the tour C. All these operations take a constant time, however after executing line 6 of the algorithm $Walk$, the data structure $\mathcal{J}_\mathcal{C}$ has to be updated because we get a new partition of the edge set of H into walks. For each edge e in the tour C, we have to change the value W_e of the walk containing the edge e (from C to P). Since, for every edge of the hypergraph H such an update is needed at most once, the total time necessary for such updates is $O(m)$.

Each execution of line 5 of the algorithm $Walk$ takes a constant time because we use the data structure $\mathcal{J}_\mathcal{C}$. Moreover, obviously, we need a constant time to execute line 7. Therefore, as the number of passes of the **while** loop is bounded by m, the running time of the algorithm $Walk$ is $O(m)$. □

The linearity of the algorithm $EulerWalk$ constructing an Euler walk in a strongly connected hypergraph follows now immediately from Lemmas 5 and 7.

Theorem 2 *The algorithm $EulerWalk$ finding an Euler walk in a strongly connected hypergraph can be implemented to run in time $O(m)$, where m is the number of edges in this hypergraph.* □

4 Final remarks

The linear time algorithm $EulerWalk$ provides a constructive proof of existence of an Euler walk in a strongly connected hypergraph. Let us observe however that we do not impose any conditions regarding the ends of the constructed Euler walk. In Naroski [9] a linear time algorithm constructing an Euler tour in a strongly connected hypergraph (if such a tour exists) was given. The algorithm is more technically complicated and requires some new ideas. This algorithm can also be used to construct in linear time an Euler ab-walk in a strongly connected hypergraph, for a specified pair of vertices a and b (if such a walk exists).

Acknowledgements

We thank Armin Fügenschuh for introducing us to the subject of this paper and for pointing out some relevant references. We are also grateful to an anonymous referee whose remarks allowed us to simplify presentation of some parts of the paper.

This research was supported by the Polish Ministry of Science and Higher Education in 2009–2010 under grant No. 3674/B/H03/2009/36.

References

[1] E. M. Arkin, M. Held, J. S. B. Mitchell, and S. S. Skiena. Hamilton triangulations for fast rendering. *Algorithms – ESA '94 (Utrecht), Lecture Notes in Comput. Sci.*, 855:36–47, 1994.

[2] C. Berge. *Hypergraphs, Combinatorics of Finite Sets*. North-Holland, 1989.

[3] J. J. Bartholdi III and P. Goldsman. Continuous indexing of hierarchical subdivisions of the globe. *Int. J. Geographical Information Science*, 15 (6):489–522, 2001.

[4] J. J. Bartholdi III and P. Goldsman. Vertex-labeling algorithms for the hilbert spacefilling curve. *Softw. Pract. Exper.*, 31:395–408., 2001.

[5] J. J. Bartholdi III and P. Goldsman. Multiresolution indexing of triangulated irregular networks. *IEEE Transactions on Visualization and Computer Graphics*, 10 (3):484–495, 2004.

[6] J. J. Bartholdi III and P. Goldsman. The vertex-adjacency dual of a triangulated irregular network has a hamiltonian tour. *Operations Research Letters*, 32:304–308, 2004.

[7] R. Laurini and D. Thompson. *Fundamentals of Spatial Information Systems*. Academic Press, 1992.

[8] Z. Lonc and P. Naroski. On tours that contain all edges of a hypergraph. *Electronic Journal of Combinatorics*, 17 (1)(#R144):1–31, 2010.

[9] P. Naroski. *Long tours in hypergraphs*. PhD thesis, Warsaw University of Technology, Warsaw, 2010. (in Polish).

[10] J. P. Vielma, S. Ahmed, and G. Nemhauser. Mixed-integer models for nonseparable piecewise-linear optimization: unifying framework and extensions. *Operations Research*, 58:303–315, 2010.

[11] D. L. Wilson. *Polyhedral Methods for Piecewise-Linear Functions*. PhD thesis, University of Kentucky, Lexington, 1998.