Masaryk University
Faculty of Informatics

Characterizing DAG-depth of directed graphs

Bachelor's Thesis

Matúš Bezek

Brno, Spring 2016

Masaryk University
Faculty of Informatics

Characterizing DAG-depth of directed graphs

Bachelor's Thesis

Matúš Bezek

Brno, Spring 2016

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Matúš Bezek

Advisor: prof. RNDr. Petr Hliněný, Ph.D.

Acknowledgement

I would like to thank prof. RNDr. Petr Hliněný, Ph.D. for suggesting this interesting topic and for his help. I would also like to thank my family and friends for their support.

Abstract

We study DAG-depth, a structural depth measure of directed graphs, which naturally extends the tree-depth of ordinary graphs. We define a DAG-depth decomposition as a strategy for the cop player in the lift-free version of the cops-and-robber game on directed graphs. The DAG-depth decomposition is related to DAG-depth in a similar way as an elimination tree is related to the tree-depth. We provide a definition of the mergeable and optimally mergeable vertices in order to make the decomposition smaller and acceptable for the cop player as a strategy. We also define a closure, which gives the largest digraph for which the given decomposition represents a winning strategy for the cop player.

Keywords

DAG-depth, DAG-depth decomposition, graph theory, directed graph, cops-and-robber

Contents

1 **Introduction** 1
2 **Definitions** 3
3 **Cops-and-robber game** 5
3.1 *Original game* 5
3.2 *Lift-free version* 5
3.2.1 Solution on undirected graphs 6
3.3 *Extension to directed graphs* 7
4 **DAG-depth decomposition** 9
4.1 *Basic properties of a DAG-depth decomposition* 10
4.2 *Computing DAG-depth decomposition* 15
4.3 *Merging the vertices* 19
5 **Closure and determinism** 25
5.1 *How to get the closure* 25
5.2 *Determinism* 26
5.3 *Modifying the decomposition* 28
5.3.1 Unrooting 28
5.3.2 Prolonging a branch 30
5.3.3 Removing a "double cover" 32
6 **Conclusion** 35
Bibliography 37

1 Introduction

A graph is a data structure which represents the set of objects and the relations between them. The objects are represented as vertices and the relation is represented as the edges between them.

Structural width parameters are numeric parameters associated with graphs. They represent different properties of graphs. Examples of such parameters are path-width [1], tree-width [2] and clique-width [3]. The first two were defined by Robertson and Seymour in 1980s, the clique-width was defined by Courcelle et al. in 1991. For instance path-width measures how close a graph is to a path and similarly with the others.

Some problems are NP-complete to solve in general but easy to solve on trees. Tree-width and tree-width decomposition provide a way to find efficient algorithms [2] for graphs with bounded tree-width.

As a directed analog of tree-width, directed tree-width [4] was defined by Johnson et al. in 1998. This line of research continued by Obdržálek's definition of DAG-width [5] in 2006. Another digraph measure Kelly-width [6] was defined by Hunter and Kreutzer in 2007. In 2010 Ganian et al. analyzed [7] digraph width measures and reasons why the search for the "perfect" directed analog of tree-width has not been successful so far.

These parameters are tightly correlated with different versions of a cops-and-robber game with an infinitely fast visible robber. The purpose of this game is to catch the robber by placing the cops in the vertices and moving them.

Structural depth parameters are correlated with the lift-free version of the game, defined in Section 3.2. Example of such parameter is tree-depth [8], defined by Nešetřil and Ossona de Mendez in 2005.

In 2012 Adler et al. defined [9] a hypergraph analog of tree-depth. In this work Adler et al. also studied the generalization of the tree-depth decomposition for hypergraphs.

A directed analog of tree-depth was defined as DAG-depth [10] by Ganian et al. in 2009. Its definition however did not provide any structural insight into DAG-depth decomposition because of difficul-

ties with such notion. In this thesis, we are looking for a definition and structure of such decomposition.

We define a DAG-depth decomposition of a digraph and show that it can be used as a winning strategy for the cop player in the lift-free version of the cops-and-robber game in directed graph. The main problem is that an optimal decomposition usually has to contain more copies of a single vertex.

Chapter 2, following this introduction, shows the definitions and basic terminology of relevant areas of graph theory. Chapter 3 formally defines the cops-and-robber game and its versions that we use. In Chapter 4 we define a DAG-depth decomposition of a digraph and prove it can be used as a winning strategy. This chapter also contains an algorithm to find such decomposition for any digraph and definition of the optimally mergeable vertices to make this decomposition smaller. Chapter 5 provides the definition of closure as a largest digraph for which a given DAG-depth decomposition represents the winning strategy. Also some modifications of a decomposition to make it stronger are defined in this chapter and it talks about determinism in the decomposition. Lastly, Chapter 6 concludes this thesis and suggests the possible directions for future work.

2 Definitions

A *simple undirected graph* is defined as a pair $G = (V, E)$, where V is the non-empty vertex set and $E \subseteq \binom{V}{2}$ is the edge set. The notation $V(G)$ ($E(G)$) represents the vertex (edge) set of the graph G.

An edge between vertices $u, v \in V(G)$ is denoted $\{u, v\}$. If a graph contains an edge $e = \{u, v\}$, the vertices u, v are adjacent. Edge e is incident to given vertices and the vertices are the ends of e.

The *degree* of a vertex v in a graph G, denoted $d_G(v)$ is the number of the edges of G incident to the vertex v. The *neighborhood* $N_G(v)$ of a vertex v in a graph G is the set of vertices adjacent to v.

A *path* P_n of the length n is a graph that consists of n edges which connect $n + 1$ distinct vertices. A *cycle* C_n of length $n \geq 3$ is a graph consisting of n vertices joined in a circular fashion with n edges.

A *complete graph* K_n is the graph consisting of n vertices such that there is an edge between every pair of vertices.

An *isomorphism* between two graphs G and H is a bijective function $f : V(G) \to V(H)$ such that for every $u, v \in V(G)$ edge $\{u, v\} \in E(G)$ exists if and only if edge $\{f(u), f(v)\} \in E(H)$ exists. Two graphs are called *isomorphic* if there exists an isomorphism between them.

A *subgraph* of a graph $G = (V, E)$ is a graph $H = (V', E')$ such that $V' \subseteq V$, $E' \subseteq E$ and for every edge $e \in E'$, both of its ends are in V'. The notation $H \subseteq G$ means that H is a subgraph of G.

Induced subgraph H of the graph G is a subgraph such that for every pair of vertices $u, v \in V(H)$, if edge $\{u, v\}$ exists in G, it must also exist in H. For a graph G and vertex $v \in V(G)$, the notation $G - v$ represents the graph H such that H is the induced subgraph of G and $V(H) = V(G) \setminus v$. A graph H is a *spanning* subgraph of G if $H \subseteq G$ and $V(H) = V(G)$.

A set of vertices such that there exists a path between each pair of them is called a *connected component*. We call a graph *connected* if it contains exactly one connected component.

A *tree* is a simple connected undirected graph which does not contain any cycle as a subgraph. A *rooted* tree is coupled with one selected vertex, called the root. A vertex v is a *parent* of a vertex u if there exists an edge between them and v is closer to root. We say v is

an *ancestor* of u if either v is u's parent, or v is an ancestor of u's parent. If v is an ancestor of u, then u is v's *descendant*.

A *forest* is a simple undirected graph not containing any cycle. Therefore, all of its connected components are trees. A *rooted forest* is a disjoint union of rooted trees.

A *disjoint union* of two graphs G_1, G_2, denoted $G_1 \cup\!\!\!\cdot\, G_2$ is a graph whose vertex set copies the vertex sets of G_1 and G_2 such that they become disjoint. The edge set $E(G_1 \cup\!\!\!\cdot\, G_2) = E(G_1) \cup E(G_2)$.

A *directed graph*, also called *digraph*, is a pair $D = (V, E)$, where V is the set of vertices and $E \subseteq V \times V$ is the set of oriented edges. An edge (u, v) goes from u to v. It has a *tail* in u and a *head* in v.

Paths, cycles, complete graphs, subgraphs, isomorphism and a disjoint union can be naturally extended to directed graphs.

An *outdegree* $d_D^+(v)$ of the vertex $v \in V(D)$ is the number of edges going from v. An *indegree* $d_D^-(v)$ of the vertex $v \in V(D)$ is the number of edges going to v. An *out-neighborhood*, denoted $N_D^+(v)$ is the set of vertices x such that the edge (v, x) exists in D.

A *directed acycled graph (DAG)* is a digraph which does not contain any directed cycle as a subgraph. In DAG, vertex u is a parent of v if the digraph contains an edge (u, v). Vertex v is then a child of u. The vertex u is an ancestor of v if the digraph contains a path from u to v. If u is an ancestor of v, then v is a descendant of u.

One of the ways to extend connectivity to directed graphs is the concept of *reachable fragments*. Reachable fragments are maximal, by inclusion, sets of vertices such that every fragment R contains a vertex called the source, from which there is a path to every vertex of R.

Two reachable fragments R_1, R_2 are *disjoint* if they do not have any vertices in common.

An *associated undirected graph* of digraph D is the graph G obtained from D by ignoring the direction of its edges. A *weak components* of directed graph are connected components of its associated undirected graph.

3 Cops-and-robber game

3.1 Original game

The *cops-and-robber* game was first introduced [11] in 1982 by LaPaugh. The variant we are interested in was introduced [12] in 1989 by Seymour and Thomas. The main difference between them is that in the version by Seymour and Thomas the robber is infinitely fast, while in LaPaugh's version he is not.

The game by Seymour and Thomas is played by one player on a finite undirected graph G. The player controls k cops. At any time each of them either stays on some vertex or is temporarily removed from the graph. The player moves the cops, he can remove them from the graph and place them back into any vertex he wants.

The robber always stands on some vertex of G. During the game, he can move at any time along the edges at infinite speed. He is not allowed to run through a cop but he can see when the cop is being placed on some vertex and he can run through that vertex before the cop lands.

The cop player wins when the cops catch a robber, i.e. when the robber is in some vertex v such that there is a cop placed in all vertices from $N^+(v)$ and also on v. The player loses if the robber is able to avoid getting caught.

The robber is always aware of cops' position and the player is aware of robber's position.

The minimal number of cops needed to catch a robber on a graph is called the *cop number* of the graph.

3.2 Lift-free version

In the *lift-free* version of the cops-and-robber game there is an additional rule; once the cop is placed to some vertex, he stays there until the end of the game. In each turn the cop player puts a cop onto some vertex of the graph. The game ends when the robber is caught or the cop player runs out of cops. If the robber is caught, the cop player wins, otherwise he loses.

The cop number can be naturally extended to this version of the game.

3.2.1 Solution on undirected graphs

The concept of tree-depth was introduced by Nešetřil and Ossona de Mendez [8]. A *tree-depth* of a graph G, denoted $td(G)$, represents the number of the cops needed to catch the robber on a graph G. An *elimination tree* for a graph G represents a game plan for the cop player to catch the robber.

Let G be an undirected graph and $G_1, \ldots, G_p$ the connected components of G. A tree-depth is defined as:

$$td(G) = \begin{cases} 1 & \text{if } |V(G)| = 1 \\ 1 + \min_{v \in V(G)}(td(G - v)) & \text{if } p = 1 \text{ and } |V(G)| > 1 \\ \max_{1 \leq i \leq p} td(G_i) & \text{otherwise} \end{cases}$$

Tree-depth can be also defined as a minimal depth of an elimitation tree of a graph G.

An elimination tree for a connected graph G is a rooted tree Y with a vertex set $V(Y) = V(G)$. If $|V(G)| = 1$, then $E(G) = \emptyset$, otherwise some vertex r is chosen as a root. Then, elimination tree is constructed for each connected component of the graph $G - r$ and the root of each of these elimination trees is connected by an edge with r. The elimination tree with the depth $td(G)$ can be naturally used as a game plan for the cop player.

For a graph with more connected components, elimination trees can be constructed for every component. Merging them into a single rooted forest will create a graph which the cop player can again use as a game plan.

The idea of using the elimination tree as a game plan is that the player begins by placing a cop on the appropriate root and continues by some path of the elimination tree to a leaf. The vertex covered is always a child of the previously covered vertex. The player always chooses a vertex which is occupied by the robber or contains such

a vertex as a descendant in the elimination tree. We will be more formal in the coming sections.

3.3 Extension to directed graphs

The lift-free version of the cops-and-robber game can be naturally extended to directed graphs. The robber can only move along the edges in the way the edges are directed.

The directed analog of tree-depth is DAG-depth [10] introduced by Ganian et al. in 2009.

Definition 1 (DAG-depth)**.** Let D be a digraph and $R_1, \ldots, R_p$ the reachable fragments of D. The DAG-depth $ddp(D)$ is inductively defined:

$$ddp(D) = \begin{cases} 1 & \text{if } |V(D)| = 1 \\ 1 + \min_{v \in V(D)}(ddp(D - v)) & \text{if } p = 1 \text{ and } |V(D)| > 1 \\ \max_{1 \leq i \leq p} ddp(R_i) & \text{otherwise} \end{cases}$$

We now relate these two concepts tightly to each other.

Theorem 3.3.1. *Let D be a digraph of DAG-depth k. Then there exists a lift-free winning strategy for the cop player by using at most k cops.*

Proof. In case that $|V(D)| = 1$, it is $k = 1$. The player puts the cop on the only vertex of D and therefore catches the robber.

If D consists of one reachable fragment, DAG-depth is defined as $1 + ddp(D - v)$ for some $v \in V(D)$ such that $ddp(D - v)$ is minimal. After placing the cop on such vertex v there are $ddp(D - v) = k - 1$ cops left to use on the graph $D - v$, which suffices to win by induction.

If D consists of more reachable fragments, each of its fragments can be considered as a separate smaller graph in the game. When the robber is in some vertex v, to catch him the player can consider the recursive solution for any fragment which contains v. □

Theorem 3.3.2. *Let D be a digraph. If there exists a lift-free winning strategy for k cops, then the DAG-depth of D is at most k.*

Proof. We use induction on the size of D.

If $|V(D)| = 1$, then the robber is immediately caught by the first cop. This is consistent with $ddp(D) = 1$.

Assume now $|V(D)| > 1$. If D consists of one reachable fragment, then consider the robber placed to its source. The winning strategy then says which vertex v is to be covered by a cop. After v is covered, the rest of the strategy talks about how to win on the graph $D - v$ with $k - 1$ cops. Hence, by induction, $ddp(D - v) \leq k - 1$. By Definition 1, it follows that $ddp(D) = 1 + \min_{u \in V(D)}(ddp(D - u)) \leq 1 + k - 1 = k$.

If D consists of more than one reachable fragment, then let $s_1, \ldots, s_m$ be the sources of those where $m \geq 2$. For $i = 1, \ldots, m$, consider the robber placed to s_i. The cop player can win with k cops on the fragment R_i reachable from s_i. Since $|V(R_i)| < |V(D)|$, by induction $ddp(R_i) \leq k$, and hence $ddp(D) = \max_{1 \leq i \leq p} ddp(R_i) \leq k$. $\square$

4 DAG-depth decomposition

The aim of this chapter is to define a DAG-depth decomposition from the recursive definition of DAG-depth (Definition 1) the same way as an elimination tree is obtained from the definition of tree-depth. The decomposition aims to represent a game plan for the cop player.

The main difference between the tree-depth and DAG-depth cases is that in undirected graphs two connected components cannot have any vertices in common, while distinct maximal reachable fragments in directed graphs can have some vertices in common. This naturally brings complications and ambiguity.

There could be two ways to resolve this. Either just ignore it and let the decomposition have more copies of one vertex. But that would mean the decomposition could grow exponentially large, and exponentially large decompositions would be practically useless for the player as a game plan and for algorithmic purposes.

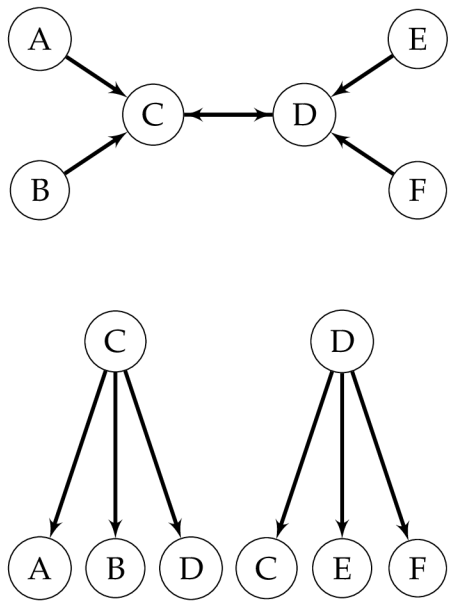

Figure 4.1: A simple digraph and its decomposition, showing that DAG-depth decomposition cannot always be done optimally without repetition of vertices (see repeated C,D)

The other extreme solution would be to merge all the copies of one vertex. This cannot always be done, as the graph in Figure 4.1 shows.

In that graph, the robber can be caught by using two cops. The idea is that if the robber starts on the vertices A or B, the player places the first cop on the vertex C. Then the robber has either stayed on the vertex he was on, or ran to D. Placing the cop on the robber will catch him, since there is no edge between A and D or B and D. Symmetrically, if the robber starts on E or F, the first cop is placed to D and second catches the robber. If the robber starts on C or D, he can not go into any other vertex and covering these two vertices in any order will result into a win.

In the given decomposition, the two copies of the vertex C can not be merged since their merging would create a path of length 2 and therefore the decomposition would not be optimal anymore. For the same reason the copies of D can not be merged, too.

Balancing these two extreme approaches would give us decompositions with some vertices merged. Its size could therefore not be that huge. Now the core question is, which vertices can be merged and why.

4.1 Basic properties of a DAG-depth decomposition

As argued above, in a decomposition P some vertices will be copies of the same original vertex $v \in D$. To properly work with this fact, there is a need to formally distinguish the two vertex sets and map between them.

This is the first difference from an elimination tree of tree-depth where the vertex sets are identical. Formally, this can be done by defining the function $org : V(P) \to V(D)$ which takes a vertex from the decomposition and returns its original from the digraph D. Vertices $x, y \in V(P)$ are copies of the same vertex if and only if $org(x) = org(y)$.

Roots of a DAG are all of its vertices whose indegree is zero. Vertices whose outdegree is zero are called *leafs*.

The *level* of a vertex in a DAG is the maximal length of a directed path from a root to this vertex. The *depth* of a vertex is the maximal length of a path from this vertex to a leaf. The *depth* of a DAG is the maximum depth over its vertices.

Due to the complicated situation (and the coming proofs), we introduce a DAG-depth decomposition in two stages – a generic one and then the so-called valid one.

Definition 2 (DAG-depth decomposition)**.** A *DAG-depth decomposition* of a digraph D is a DAG P and a surjective function $org : V(P) \to V(D)$. Furthermore, a DAG-depth decomposition is called *valid* if the following Neighbor cover condition holds.

The *Neighbor cover* condition states that for every vertex $v' \in V(P)$ such that $org(v') = v$, the following holds:
For every $u \in N_D^+(v)$,

1. there exists $u' \in V(P)$ such that $org(u') = u$ and u' is a descendant of v' in P, or
2. every path from any root of P to v' contains a vertex $u' \in V(P)$ such that $org(u') = u$.

Let P be a DAG-depth decomposition of some graph D such that the depth of P is equal to the DAG-depth of D. P is then called an *optimal* decomposition. Such decomposition exists for any digraph D, as Theorem 4.1.2 shows.

The following example shows how the DAG-depth decomposition can be used as a strategy for the cop player to catch the robber.

If the player is to use the decomposition in Figure 4.2 as a game plan, he starts by covering the vertex E, since its copy is the only root.

Then, if the robber is on the vertex A or B, the player continues by covering the vertex B. If the robber was on this vertex, he has been caught. Otherwise he is on the vertex A, which the player will cover by the third cop and therefore catch the robber.

If the robber was not on the vertex A or B, the player places the second cop on the vertex G, whose copy is on the same level as the copy of B. There are now three possibilities where the robber can be.

The first one is that he is on one of the vertices C, D. The second one is that he is on the vertex F. The last one is that the robber is hiding on one of the vertices H, I, or J.

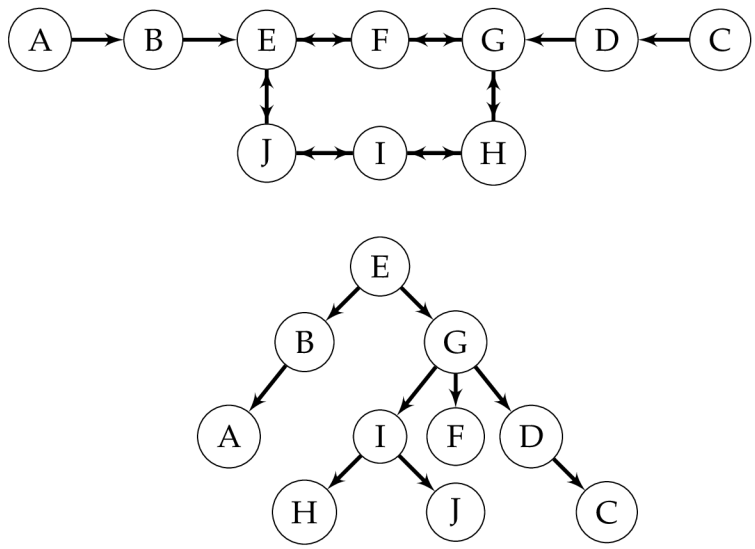

Figure 4.2: A simple graph and its valid and optimal decomposition

If it is the first case, the player continues by covering the vertex D, whose copy is the child of the copy of the last covered vertex. If the robber was here, he is caught, otherwise the cop is placed to C and catches the robber.

If it is the second case and the robber is hiding on the vertex F, the player simply covers the vertex F and catches the robber.

If the robber is on one of the vertices H, I, or J, the player covers the vertex I, whose copy is the child of the copy of the last covered vertex G. The robber then escapes either to vertex H or to J.

In the last step the player simply covers the vertex robber is on and catches him.

The game rules outlined in this example are formally defined next.

Definition 3. Given a DAG-depth decomposition (P, org) of a digraph D, the cop player's strategy is as follows. The cops are placed on the vertices of D and every cop is placed "because of" some vertex of P.

The following convention is observed: if we say a cop is to be placed to a vertex $v' \in V(P)$, he is actually placed to $v \in V(D)$ such that $org(v') = v$, unless the vertex has been covered by another cop before. In that case, no cop is placed in this step. Then, the strategy is described by two simple rules:

1. The first cop is always placed to one of the roots of P. Each subsequent cop is placed to the out-neighborhood of the previous cop in P.

2. Among the possible positions from 1, the actually chosen one must have in P a directed path to a copy of robber's current position.

The choice of the next vertex to be covered by a cop in this strategy is generally non-deterministic. Since the decomposition can contain more copies of a single vertex, more vertices can contain them as a descendant. The example of such choice is choosing the root of the decomposition, shown in an example in Figure 5.1. Section 5.2 talks more about determinism in DAG-depth decompositions.

Theorem 4.1.1. *Let D be a digraph and P its DAG-depth decomposition of depth k. Then the decomposition is valid if and only if the strategy based on P by Definition 3 is winning for k cops.*

Proof. ($\Leftarrow$) The decomposition is valid if the Neighbor cover condition holds by Definition 2. Suppose the contrary: there exist a pair of vertices $u, v \in V(D)$ such that edge $(u, v) \in E(D)$ exists. Also a vertex $u' \in V(P)$ exists such that $org(u') = u$ and u' does not contain any copy of the vertex v as a descendant. Since none of the conditions in the definition of the Neighbor cover condition holds, there exists a path p from some root to u' such that p does not contain any copy of the vertex v.

Let the player use the decomposition according to rules specified in Definition 3. If the robber started on the vertex $u \in V(D)$, then the player could proceed along the path p, since all of its vertices contain the copy $u' \in V(P)$ as a descendant. When the player covers the vertex u because of u', the robber can escape to the vertex $v \in V(D)$ since the path p does not contain any copy of that vertex and therefore it is

not covered by a cop. Since the vertex u' does not contain any copy of v as a descendant, the player playing according the Definition 3 can not cover v and the robber wins. The given decomposition therefore represents a strategy which is not winning.

($\Rightarrow$) The other direction is the subject of subsequent claims and will follow from Theorem 4.1.3. □

Theorem 4.1.2. *If the DAG-depth of a digraph D is k, then there exists a valid DAG-depth decomposition of D of depth k.*

Proof. If $|V(D)| = 1$, then $ddp(D) = 1$. A decomposition with depth one exists, since it consists also of the only vertex.
A decomposition that consists of one vertex is always valid, since the original graph did not contain any edges and the Neighbor cover condition therefore always holds.

If $|V(D)| > 1$ and D consists of the only reachable fragment, then the DAG-depth is computed as $ddp(d) = 1 + \min_{v \in V(D)}(ddp(D - v))$. Such vertex v is then the root of the decomposition and is connected to the roots of the recursive decomposition of the rest of a graph.
Since the vertex v was chosen to be the root, all other vertices are its descendants. Therefore all the vertices of its out-neighborhood are his descendants, and for the rest of the graph the Neighbor covercondition holds by induction. That means the decomposition is valid.

Otherwise, D consists of more reachable fragments. The decomposition of each of them can be computed separately. When a disjoint union of them is made to a single graph, its depth will be equal to the maximum of the decompositions of the fragments. This is in accordance with Definition 1.
Since the decomposition is a disjoint union of the decompositions of the fragments, by induction the decompositions of the fragments are valid. Therefore their disjoint union is a valid decomposition too. □

Theorem 4.1.3. *Let a DAG P be a valid DAG-depth decomposition of the depth k. Then, if the player places the cops by the rules of Definition 3, then this defines a winning strategy for at most k cops.*

Proof. The decomposition P is valid if the Neighbor cover condition from Definition 2 holds. The cop player wins when the cop is placed on top of the robber to a vertex r and all vertices from $N_D^+(r)$ are already covered by the cops.

Let the last move of the cop be to vertex $u \in V(D)$, because of its copy $u' \in V(P)$ as in Definition 3. We claim that the robber may move only to vertices of D whose copies in P are reachable from u' in P.

Before the robber moves, the previous statement holds because of the rule 2 of Definition 3.

Let the robber be on a vertex $r \in V(D)$ and $r' \in V(P)$ its copy such that it is a descendant of u'. The statement still holds if the robber moves along an edge $(r, v) \in E(D)$ to vertex $v \in V(D)$ which has not yet been covered by a cop. From Definition 2 we know that in the decomposition P every path from a root to r' contains a copy of v or P contains a vertex $v' \in V(P)$ such that it is a descendant of r'. If v' is a descendant of r', it is also a descendant of u' since r' is its descendant. If every path from a root to r' contains a copy of v, such copy must be a descendant of u'. If it was not, then v' would have to lie on some path from a root to u', and v would have already been covered by a cop by Definition 3.

The previous invariant allows the player to always fulfill the rules of Definition 3.

The rules in Definition 3 end with covering a vertex from $V(D)$ because of its copy which is a leaf in decomposition P. That means that all the neighbors of the covered vertex have been covered before and the robber is caught. The decomposition therefore represents a winning strategy.

In every move, the vertex $v \in V(D)$ is covered because of some $v' \in V(P)$. Such vertex v' is always a child of the previous v' and therefore all such vertices create a path in P. If the player used more than k cops, the path would need to be longer than k. Since the depth of P is k, such path can not exist. Therefore, the decomposition represents a strategy for at most k cops. □

4.2 Computing DAG-depth decomposition

The following algorithm returns the valid and optimal DAG-depth decomposition without merging any of the possible duplicated vertices.

Algorithm 1 : procedure $ddpdec(D)$

Input: Digraph D
Output: DAG-depth decomposition of D

```
1:  if |V(D)| = 1 then
2:      P := empty directed graph
3:      v := the only vertex of D
4:      v' := vertex such that org(v') = v
5:      P.addVertex(v')
6:  else if |rf(D)| = 1 then
7:      v := v ∈ V(D), ddp(D − v) = min{ddp(D − u) : u ∈ V(D)}
8:      P := ddpdec(D − v)
9:      v' := vertex such that org(v') = v
10:     P.addVertex(v')
11:     for all u : P.roots do
12:         P.addEdge(v', u)
13:     end for
14: else
15:     P := empty directed graph
16:     for all H : H is a reachable fragment of D do
17:         H' := ddpdec(H)
18:         P := P ⊎ H'          ▷ symbol ⊎ represents disjoint union
19:     end for
20: end if
21: return (P, org)
```

Lemma 4.2.1. *Any call of Algorithm 1 will eventually lead to a recursive call on a digraph D' such that $|V(D')| = 1$.*

Proof. If $|V(D)| > 1$ and D consists of one reachable fragment, on line 8, the algorithm is recursively called on the digraph $D - v$. Since $|V(D - v)| = |V(D)| - 1$, the claim holds by induction.

If D consists of more reachable fragments, the algorithm is recursively called on each of its reachable fragments $R_1, \ldots, R_p$ for $p > 1$. Each of its fragments must be smaller than D, therefore all recursive calls are performed on a smaller digraphs and the claim holds by induction. □

Theorem 4.2.2. *Algorithm 1 will always end.*

Proof. Lemma 4.2.1 shows that any call will lead to a recursive call on a digraph D' such that $|V(D')| = 1$. When the algorithm is called on such digraph, it returns the decomposition with only one vertex and does not do any recursive call. □

Theorem 4.2.3. *Let D be a simple directed graph. Algorithm 1 with input D will return a DAG P and function org, which form a valid and optimal DAG-depth decomposition of D.*

Proof. If there is only one vertex in digraph D, its decomposition also consists of only one vertex, the copy of the only vertex of D.

If D has a single reachable fragment, its DAG-depth is computed as $1 + min\{ddp(G - v) : v \in V(G)\}$. On line `7` algorithm chooses such vertex v that DAG-depth of graph $(G - v)$ is minimal. The `for`-cycle on lines `11-13` adds only edges from v to roots of the decomposition of the digraph $(G - v)$, which increases the depth of decomposition by 1. For the digraph $(G - v)$ the claim holds by induction.

If there are $p > 1$ reachable fragments, the algorithm computes their decompositions and joins them all to a single graph without merging any vertices. Therefore the resulting graph contains p weak components, each of them being a DAG-depth decomposition of one reachable fragment. The depth of the whole decomposition is equal to the maximum of depths of its weak components, which is the maximum of DAG-depths of its reachable fragments. □

This decomposition, without merging any vertices, can be exponentially large and therefore not practically usable as an explicit strategy for a player.

Theorem 4.2.4. *There exists a digraph D such that its only valid and optimal DAG-depth decomposition without merging any vertices is exponentially large.*

Proof. Let D be a digraph such that $V(D) = \{a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_n\}$ for $n \in \mathbb{N}$ and $E(D) = \{(a_i, a_j) \cup (a_i, b_j) \cup (b_i, a_j) \cup (b_i, b_j)\}$ for every $1 \leq i < j \leq n$. See Figure 4.3.

The digraph D consists of two isomorphic reachable fragments – $V(D) \setminus \{a_1\}$ and $V(D) \setminus \{b_1\}$.

In a reachable fragment $V(D) \setminus \{b_1\}$ the only optimal first move of the cop player is placing the cop onto a_1 since if he made any other

move, one of the subgraphs $\{a_1, a_2, \ldots, a_n\}$ and $\{a_1, b_2, \ldots, b_n\}$ is left uncovered. These subgraphs each require another n cops to catch the robber, while the DAG-depth of D is $n = 1 + n - 1$.

After covering a_1, the remaining digraph has the vertex set $\{a_2, \ldots, a_n, b_2, \ldots, b_n\}$. Its decomposition can be found by induction.

The reachable fragment $V(D) \setminus \{a_1\}$ is isomorphic to $V(D) \setminus \{b_1\}$ and therefore the only optimal move is to cover b_1 and the remaining digraph is the same as for the first reachable fragment.

The decomposition of D will contain decomposition of the remaining digraph two times – as a descendant of a_1 and as a descendant of b_1. The decomposition therefore consists of $\sum_{i=1}^{n} 2^i = 2^{n+1} - 2$ vertices. □

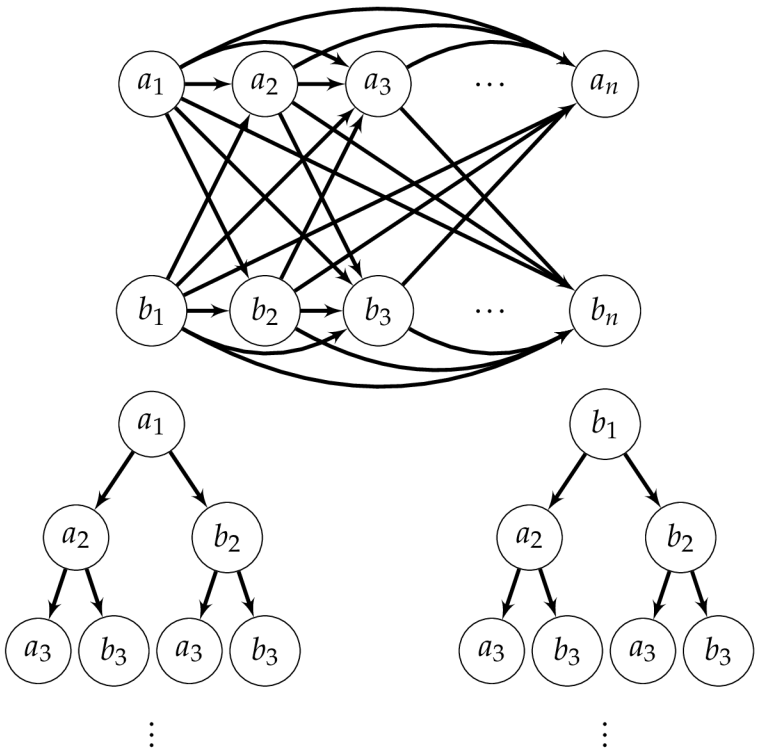

Figure 4.3: A digraph and its exponentially large valid and optimal DAG-depth decomposition without merging of any vertices

4.3 Merging the vertices

We now return to the size aspect of a DAG-depth decomposition. To reduce the size of the decomposition, some copies of the same vertex should be merged while preserving validity of the decomposition. Not all vertices with the same original can be merged though, as shown at the beginning of this chapter, in Figure 4.1.

Merging can be illustrated by the following example.

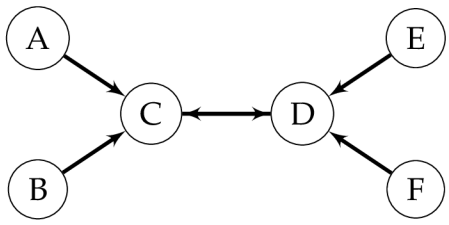

Figure 4.4: A simple digraph

Graph in Figure 4.4 consists of 4 reachable fragments, starting from A, B, E, F. Its decomposition obtained without merging any vertices will therefore also be composed of 4 disjoint reachable fragments, each of them representing the decomposition for a single reachable fragment of the original graph.

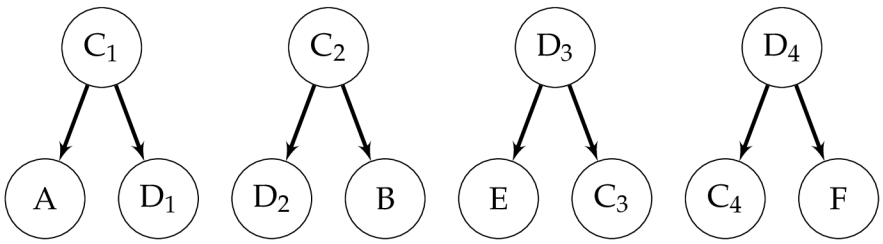

Figure 4.5: An optimal and valid decomposition of the graph from Figure 4.4 before merging any vertices. The indexed letters refer to the original vertices in Figure 4.4.

The graph in Figure 4.5 represents a valid and optimal decomposition of the graph in Figure 4.4 according to the Definition 2.

The vertices D_1 and D_2 can be merged, since they both have a copy of C as their only ancestor and neither of them has any descendants. For the same reason the vertices C_3 and C_4 can be merged.

Vertices C_1 and C_2 can be merged too. In the original graph, the only vertex reachable from C is D, and both C_1 and C_2 contain a copy of D as a descendant. Symmetrically, D_3 and D_4 can be merged.

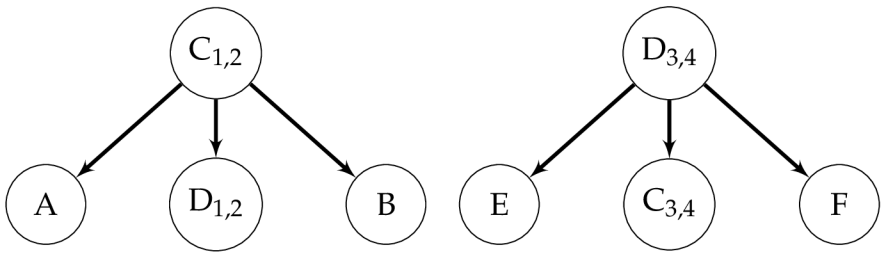

Figure 4.6: The decomposition from Figure 4.5 after merging some vertices.

No further changes can be made if the decomposition is to stay valid and optimal. To see this, note that either the copies of C or the copies of D can be merged, but not both. Since the vertex $C_{1,2}$ is the parent of $D_{1,2}$ and $D_{3,4}$ is the parent of $C_{3,4}$, merging the copies of C as well as the copies of D would mean that the resulting digraph would contain only one copy of each of these vertices. Therefore, the digraph would contain both edges (C, D) and (D, C). That would create a cycle, and the DAG-depth decomposition is defined as a DAG.

On the other hand, note that merging of all the copies of, say, D, preserves validity, but not optimality, as can be seen in Figure 4.7.

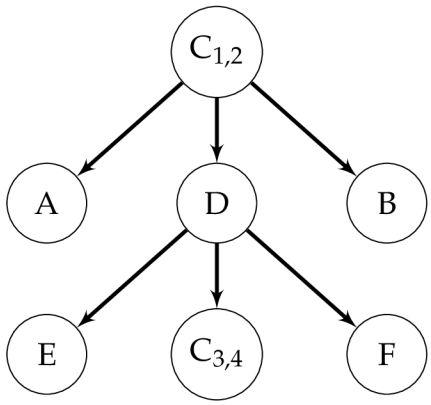

Figure 4.7: The decomposition from the bottom part of Figure 4.5 after more merging. This decomposition is no longer optimal.

The graph in Figure 4.7 contains the vertex $C_{3,4}$, which is not necessary, since the vertex C from the original graph will be covered because of the vertex $C_{1,2}$.

Since every path from any root to the vertex $C_{3,4}$ contains another copy of C, no cop will ever be placed because of $C_{3,4}$. The vertex C will always be covered because of the another copies of C which the player will reach in a decomposition before $C_{3,4}$.

Section 5.3.3 is dedicated to the behavior of the decomposition where it contains such vertices that are useless since their original will always be covered before the cop player reaches them in the decomposition.

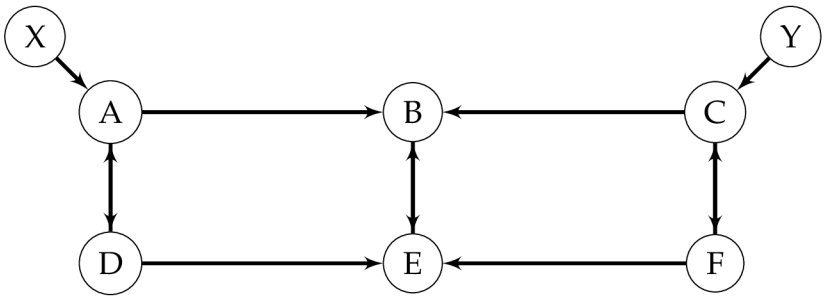

Figure 4.8: A simple digraph

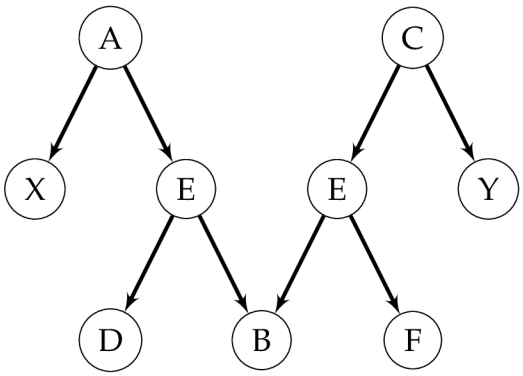

Figure 4.9: An optimal and valid decomposition of the graph from Figure 4.8, in which every mergeable pair of vertices is already merged.

The decomposition in Figure 4.9 is an example of why the Neighbor cover condition is needed. Vertices E cannot be merged even though they are on the same level in decomposition. Merging them would break the Neighbor cover condition, since the original graph contains an edge (D, A), and if the copies of the vertex E were merged, the copy of the vertex A would no longer need to be covered by a cop before D. In the same way merging them would break a Neighbor cover condition because of the edge (F, C) in the original graph. Therefore after merging the copies of E the decomposition would no longer be valid.

In the game terms, the problem is as follows. Let the player proceed by the rules defined in Definition 3. Then if the robber is initially in the vertex D, the player could choose to place a cop on C, since D would become its descendant in the decomposition. If then robber moved to the vertex A and the cops would proceed by this game plan, they would not catch him.

Definition 4. Two vertices $u, v \in V(P)$ are *mergeable* in the DAG-depth decomposition (P, org) if the conditions $1 - 3$ hold. Furthermore, u and v are *optimally mergeable* if they are mergeable and also the condition 4 holds.

1. $org(u) = org(v)$
2. After merging u with v, P remains a DAG.

3. Merging u with v does not break the Neighbor cover property from Definition 2.

4. Merging u with v does not increase the depth of the decomposition.

Lemma 4.3.1. *Let P be a valid and optimal DAG-depth decomposition of some graph and $u, v \in V(P)$ an optimally mergeable pair of vertices of P. Then, after merging u and v, P is still a valid and optimal decomposition.*

Proof. The decomposition is valid when the Neighbor cover property holds. The condition 3 from the Definition 4 states that two vertices are mergeable if the neighbor cover property will still hold. Therefore the decomposition has to be valid.

The condition 4 states that merging them does not increase the depth of the decomposition. Since the depth has not been increased, the decomposition is still optimal. □

Not following Definition 4 and merging a pair of vertices which are not mergeable will make the resulting decomposition no longer valid. The definition of the pairs of mergeable vertices only assures that after merging them the decomposition fulfills the definition of a DAG-depth decomposition.

The trivial lower bound on the size of a valid decomposition is equal to the number of vertices of the original graph, but such a decomposition may not be optimal. The question of minimizing the size of the valid and optimal decomposition is left for further investigations.

5 Closure and determinism

While previous text focused on how to construct a DAG-depth decomposition, or a game plan, for a given digraph, now we look at the other side. Roughly, having a game plan, can we easily say on which digraphs we can win with it?

Recall that in the case of tree-depth this was trivial - already the definition of the tree-depth decomposition worked with the concept of a closure of a rooted forest, which, at the same time, represents the unique maximal graph on which the cop player always wins when following the decomposition as the game plan.

However, in the case of digraphs and DAG-depth, we again face unprecedented complications. A DAG-depth decomposition, unlike an elimination tree, can contain more copies of a single vertex of the original digraph. Therefore a problem, which was trivial in undirected graphs, becomes complex.

In the closure obtained from an elimination tree, each vertex is connected with all of its former ancestors and descendants. In a DAG-depth decomposition, more copies of a vertex can have different ancestors and descendants.

5.1 How to get the closure

Definition 5 (Closure)**.** A *partial closure* C is a directed graph obtained from a DAG-depth decomposition P of some graph D, such that D is a spanning subgraph of C and (P, org) is still a valid DAG-depth decomposition for the digraph C. A *closure* is a maximal partial closure by inclusion.

Theorem 5.1.1. *For a DAG-depth decomposition (P, org) of a digraph D, we construct a digraph C, such that $V(C) = V(D)$ by iteratively adding edges (u, v) for $u, v \in V(C)$ if for every $u' \in V(P)$ which is a copy of u*

1. *there exists $v' \in V(P)$ such that v' is a copy of v and v' is a descendant of u', or*

2. *every path from a root to u' contains a copy of v.*

Then, C is a closure of P.

Proof. The conditions in this lemma are the same as the Neighbor cover property in Definition 2 and so C is clearly a partial closure. On the other hand, every other edge not in $E(C)$ would, by its own, violate Definition 2 and so C is maximal. □

5.2 Determinism

For a digraph, there is generally no unique valid and optimal DAG-depth decomposition. There can be more optimal valid winning strategies for the cop player on the given graph. For example, when playing on any bi-directed complete graph K_n the robber can be caught only by using n cops and covering all the vertices of the graph. In this case it does not matter in which order the vertices are covered, any strategy that covers all vertices is optimal and valid. Figure 5.2 shows another digraph, its two different decompositions such that the closures obtained from them are different.

A closure for a particular decomposition, on the other hand, is unique since it is the maximal digraph such that the Neighbor cover condition holds – see Theorem 5.1.1. Whether any ordered pair of vertices fulfills the Neighbor cover condition depends only on the decomposition. The definition of the Neighbor cover condition is deterministic and therefore it is a deterministic problem to state whether some edge $(u, v) \in V(C)$ will exist in the closure.

Observation 5.2.1. *Choosing a vertex v on line 7 of Algorithm 1 is nondeterministic.*

Proof. Let D be a digraph such that $V(D) = \{A, B\}$ and $E(D) = \emptyset$. Then $ddp(D - A) = ddp(D - B) = 1$ and both vertices can be chosen as v. □

Since Algorithm 1 is not deterministic, there can be more valid optimal DAG-depth decompositions for a single graph, as can be seen in Figure 5.1.

Observation 5.2.2. *There exists a digraph D which has two different DAG-depth decompositions P_1 and P_2 such that closures C_1 and C_2 obtained from decompositions are different.*

Proof. Such a graph is for example circle C_4, which is shown at figure 5.2 along with its decompositions and their closures. □

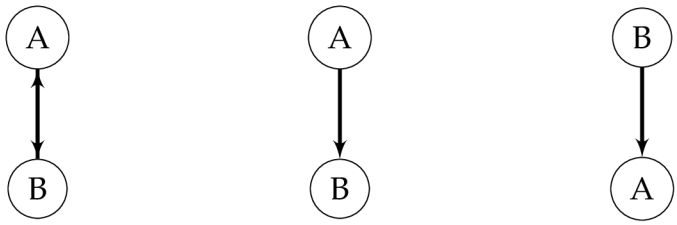

Figure 5.1: A simple digraph and its two different valid DAG-depth decompositions

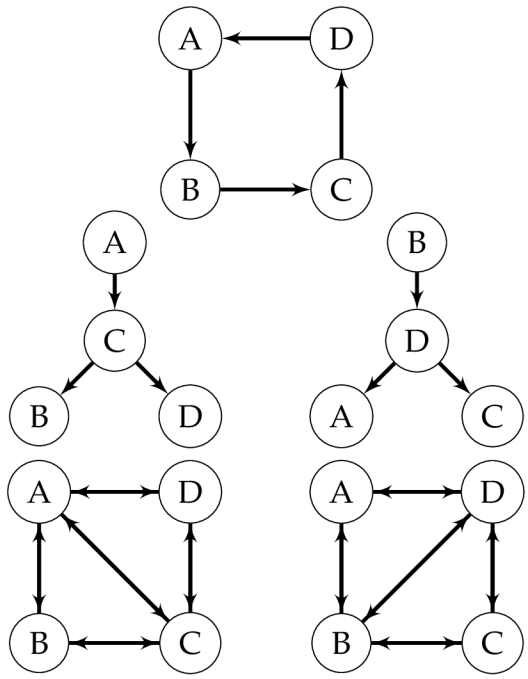

Figure 5.2: A simple digraph, its two different, valid and optimal decompositions and two different closures obtained from the decompositions

5.3 Modifying the decomposition

This section suggests a few modifications of the decomposition. After these modifications the decomposition will remain valid for the original digraph while the closure will represent the larger class of graphs.

Different decompositions of the same graph can be compared. We say a decomposition P_1 is stronger than a decomposition P_2 if the closure C_2 obtained from P_2 is a subgraph of a closure C_1 obtained from P_1. There are also incomparable decompositions – for example the decompositions presented in Figure 5.2.

If a decomposition has been optimal, after these modification it will remain optimal. Note that while the decomposition remains optimal and the cop player can use it to catch a robber with the optimal number of cops, in the worst case, the strategy for the original will not remain optimal in every case. If the robber is in the part of the graph where he can be caught by less cops, the cop player might cover some unnecessary vertex in order to make the decomposition stronger.

The other purpose of these modifications is that some vertices may not be mergeable in the original decomposition but modifying the decomposition can make them mergeable. That way can make the decomposition smaller and more acceptable for the player as a usable strategy.

We plan to study this interesting aspect of DAG-depth in our future work.

5.3.1 Unrooting

If some root of the decomposition has the depth smaller than the depth of the decomposition, whenever the player uses this branch of decomposition, he catches the robber while still having some cops left.

Theorem 5.3.1. *Let P be a valid and optimal DAG-depth decomposition of some digraph D and r its root such that its depth is smaller than the depth of P. Then after unrooting this vertex, see Algorithm 2, P is still valid and optimal DAG-depth decomposition of D.*

Proof. Let l be the depth of r and ddp be the depth of D. If the player used affected branch to catch a robber, in the end he would have

$ddp - l$ cops left.
After modification, the player uses $ddp - l$ cops by one of the other branches, and then either continues by the branch he already started, or starts using the unrooted branch, in which the remaining cops are enough to catch the robber. □

Algorithm 2 : procedure $unroot(P, r)$

Input: Decomposition P, vertex r which is a root of P
Output: Modified decomposition P

```
l := depth of r
ddp := depth of P
for all v : V(P), v ≠ r do
    if level of v = ddp − l then
        P.addEdge(v, r)
    end if
end for
return D
```

Adding just one edge from any vertex of desired level to r would unroot the vertex r as well. But there is no reason why an edge could not be added from all such vertices, and adding them all will produce a stronger decomposition.

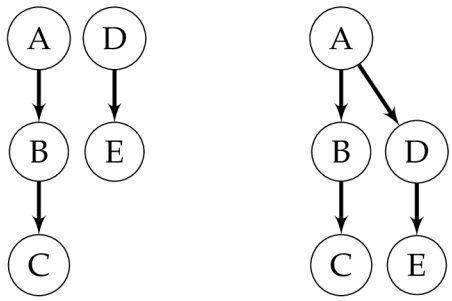

Figure 5.3: Example of unrooting

Figure 5.3 presents two DAG-depth decompositions – the unmodified one on the left and the decomposition after unrooting.

The unmodified decomposition suggests that the original graph consists of two disjoint reachable fragments – $\{A, B, C\}$ and $\{D, E\}$. If the robber is in any of these reachable fragments, he is caught only if the whole fragment is covered by a cop.

Therefore if the robber starts in the reachable fragment that contains the vertices D,E, he can be caught by only two cops. That means the player has one cop left, since the depth of the unmodified decomposition is 3.

If the cop player uses the unmodified DAG-depth decomposition as a strategy and the robber starts in the reachable fragment that contains the vertices D,E, the cop player covers these two vertices and catches the robber.

If the player plays the game on the same graph but uses the decomposition after unrooting, and the robber starts in the vertex D or E, the first cop is unnecessary placed in a vertex A. Then the player catches the robber in the same way as in the case of using the unmodified decomposition.

Unrooted decomposition is therefore stronger than the original decomposition, since a closure obtained from it contains edges (A, D), (D, A), (A, E) and (E, A).

5.3.2 Prolonging a branch

A vertex v of decomposition P can be moved if $v.level + v.depth < P.depth$. If that happens, the vertex is a part of a branch that catches the robber without using all the cops.
This modification has a lot in common with unrooting, which was described in Section 5.3.1.

Theorem 5.3.2. *Let P be a valid and optimal DAG-depth decomposition of some digraph D and v its vertex such that $v.level + v.depth < P.depth$. Then after prolonging the branch containing this vertex, see Algorithm 3, P is still valid and optimal DAG-depth decomposition of D.*

Proof. If $v.level + v.depth < P.depth$ holds then by every strategy that uses the vertex v catches the robber while still having some cops left. That means using one more cop will not matter.

Making the vertex a child of all his former siblings will make the player to place one cop to one of the siblings and then either continue with one of the sibling's original children, or then return to vertex v and continue through it. If the player continues through one of the original children of the sibling, this modification to graph did not change anything for the given game. If he continues through v, the only change is the one cop placed to v's sibling. □

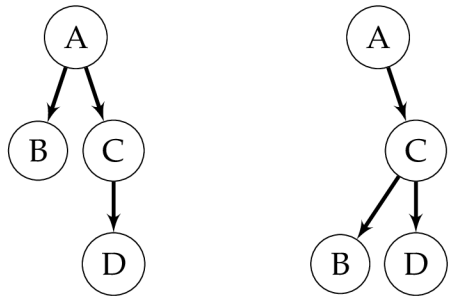

Figure 5.4: Example of prolonging the branch

Figure 5.4 contains two DAG-depth decompositions – the unmodified on the left and the decomposition after prolonging the branch that ends with the vertex B.

The unmodified decomposition suggests that after the vertex A has been covered in the original graph, the remaining graph consists of two disjoint reachable fragments – $\{B\}$ and $\{C, D\}$. If the robber is in any of these fragments, he can be caught only by covering the whole fragment by cops.

Therefore if after covering the vertex A, the robber escapes to the vertex B, he can not escape anywhere from this vertex and he can be caught by the next – second cop. While if he escapes to the other reachable fragment, two cops are needed to catch him. That means the DAG-depth of the original graph is 3.

If the player proceeds by any of these decompositions, he starts by covering the vertex A. Let the robber escape to B. He can now be covered by the next cop, since he can not escape anywhere from this vertex.

If the player uses the unmodified decomposition, he catches the robber by the second cop and ends the game while still having one more cop left.

If the player uses the decomposition with the prolonged branch, he unnecessarily covers the vertex C before catching the robber. Thanks to this unnecessary move, the closure obtained from this decomposition contains edges between the vertices B and C and therefore the modified decomposition is stronger than the original one.

Algorithm 3 : procedure $prolong(P, v)$

Input: Decomposition P, vertex v such that $v.depth + v.level < P.depth$

Output: Modified decomposition P

```
s := number of siblings of v
if s = 0 then
    for all p : parents of v do
        P := prolong(P, p)
    end for
else
    for all b : siblings of v do
        P.addEdge(b, v)
    end for
    for all p : parents of v do
        P.removeEdge(p, v)
    end for
end if
return D
```

5.3.3 Removing a "double cover"

Modifying the decomposition may cause that some copy v' of vertex v can be covered only if some other copy of the same vertex have been covered before. In that case v' can be removed from the decomposition.

Algorithm 4 : procedure $dCover(P, v')$

Input: Decomposition P, vertex v' such that v' is a copy of v and every path from root to v' contains at least one copy of the vertex v

Output: Modified decomposition P

```
for all f ∈ parents of v' do
    for all c ∈ children of v' do
        P.addEdge(f, c)
    end for
end for
P.removeVertex(v')
return D
```

Theorem 5.3.3. *Let P be a valid and optimal DAG-depth decomposition of some digraph D and $v' \in V(P)$ its vertex such that $org(v') = v$ and every path from some root of P to v' contains at least one other copy of the vertex v. Then after removing v' from P (see Algorithm 4), P is still valid and optimal DAG-depth decomposition.*

Proof. Since every path from some root of P to v' contains at least one other copy of v, the vertex v has to be covered by a cop before the cop player reaches v' in the decomposition. If the cop player wants to continue to descendants of v', he does not cover the original of v', since it is already covered. He skips one step and covers one of its children. Therefore removing v' from the decomposition by Algorithm 4 does not make any change. □

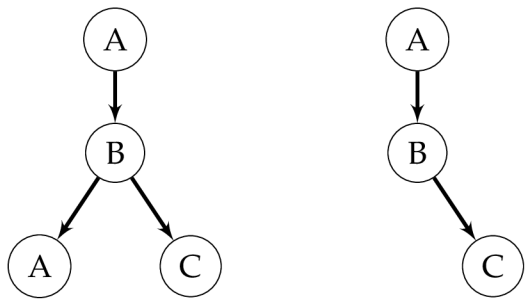

Figure 5.5: Example of removing the double cover of the vertex A

Figure 5.5 presents two DAG-depth decompositions – on the left side is the unmodified and on the right side is the decomposition where the unnecessary second copy of the vertex A is removed.

The second copy of the vertex A is obviously unnecessary since if the player gets to it in the decomposition, the vertex A is already covered because of the first copy of that vertex. Therefore removing it from the decomposition does not make any difference in the strategy.

Note that the modified decomposition is stronger, since the closure obtained from it can contain an edge (A, C). The closure obtained from the original decomposition does not contain it because the second, unnecessary, copy of A does not contain any copy of C neither as a descendant nor as an ancestor.

6 Conclusion

We present a definition of a DAG-depth decomposition which extends the concept of an elimination tree as a winning strategy for the cop player in the lift-free version of the cops-and-robber game. Unlike in the case of an elimination tree, the vertex set of a DAG-depth decomposition is not equal to the vertex set of the original graph. That requires us to deal with the two vertex sets and to find a way to map between them. Since the vertex sets are not equal, the size aspect of the DAG-depth decomposition becomes a problem. In the primitive handling, the size of the decomposition can grow exponentially. To make a decomposition smaller and therefore acceptable as a strategy for the cop player, we provide a definition of mergeable and optimally mergeable vertices. We also provide an algorithm for computing a decomposition of any digraph.

Secondly, we present a definition of the closure as a biggest graph where the given decomposition works as a winning strategy. We also provide a way to deterministically obtain a closure for a given decomposition. The definition of the stronger decomposition is provided as a way to compare different decompositions of the same digraph. This also shows that there can exist more valid and optimal decompositions of a single digraph and therefore a single digraph can also produce more optimal closures.

Lastly, we start studying ways to modify the decomposition to make it stronger and smaller in order to make it more acceptable for the player. We provide three modifications in order to achieve this – unrooting, prolonging a branch and removing a "double cover". These modifications can make a valid and optimal decomposition stronger and smaller while the decomposition remains valid and optimal.

We find two main directions for possible future improvements and extension of our results. The first one is to study the lower bounds on the size of a valid and optimal DAG-depth decomposition of a digraph and a relationship between this bounds and the properties of given digraphs. The second one is to continue investigation of modifications of the decomposition to make the decomposition stronger in regard of its closure.

Bibliography

[1] N. Robertson and P. D. Seymour. "Graph minors. I. Excluding a forest". In: *Journal of Combinatorial Theory, Series B* 35.1 (Aug. 1983), pp. 39–61.

[2] N. Robertson and P. D. Seymour. "Graph minors. II. Algorithmic aspects of tree-width". In: *Journal of Algorithms* 7.3 (Sept. 1986), pp. 309–322.

[3] B. Courcelle, J. Engelfriet, and G. Rozenberg. "Handle-rewriting hypergraph grammars". In: *Journal of Computer and System Sciences* 46.2 (Apr. 1993), pp. 218–270.

[4] T. Johnson, N. Robertson, P. D. Seymour, and R. Thomas. "Directed Tree-Width". In: *Journal of Combinatorial Theory, Series B* 81.1 (May 2001), pp. 138–154.

[5] J. Obdržálek. "DAG-width". In: *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithms* (2006), pp. 814–821.

[6] P. Hunter and S. Kreutzer. "Digraph measures: Kelly decompositions, games, and orderings". In: *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms* (2007), pp. 637–644.

[7] R. Ganian, P. Hliněný, J. Kneis, D. Meister, J. Obdržálek, P. Rossmanith, and S. Sikdar. "Are there any good digraph width measures?" In: *Journal of Combinatorial Theory, Series B* 116 (Jan. 2016), pp. 250–286.

[8] J. Nešetřil and P. Ossona de Mendez. "Tree-depth, subgraph coloring and homomorphism bounds". In: *European Journal of Combinatorics* 27.6 (Aug. 2006), pp. 1022–1041.

[9] I. Adler, T. Gavenčiak, and T. Klimošová. "Hypertree-depth and minors in hypergraphs". In: *Theoretical Computer Science* 463 (Dec. 2012), pp. 84–95.

[10] R. Ganian, P. Hliněný, J. Kneis, A. Langer, J. Obdržálek, and P. Rossmanith. "Digraph width measures in parametrized algorithmics". In: *Discrete applied mathematics* 168 (May 2014), pp. 88–107.

[11] A. S. LaPaugh. "Recontamination does not help to search a graph". In: *Journal of the ACM* 40.2 (Apr. 1993), pp. 224–245.

[12] P. D. Seymour and R. Thomas. "Graph Searching and a Min-Max Theorem for Tree-Width". In: *Journal of Combinatorial Theory, Series B* 58.1 (May 1993), pp. 22–33.