



Complexity of action path finding with small precondition sets

Downloaded from: <https://research.chalmers.se>, 2026-05-16 04:49 UTC

Citation for the original published paper (version of record):

Damaschke, P., Ramirez-Amaro, K. (2026). Complexity of action path finding with small precondition sets. *Discrete Applied Mathematics*, 381: 339-348.
<http://dx.doi.org/10.1016/j.dam.2025.12.009>

N.B. When citing this work, cite the original published paper.



Complexity of action path finding with small precondition sets

Peter Damaschke^{a,*}, Karinne Ramirez-Amaro^b

^a Department of Computer Science and Engineering, Chalmers University of Technology and University of Gothenburg, Gothenburg, 41296, Sweden

^b Department of Electrical Engineering, Chalmers University of Technology, Gothenburg, 41296, Sweden



ARTICLE INFO

Article history:

Received 30 January 2025

Received in revised form 4 September 2025

Accepted 5 December 2025

Keywords:

State space

Reachability

Token sliding

Topological ordering

Feedback vertex set

Perfect matching

ABSTRACT

Suppose that we are given a finite set of Boolean attributes and a set of actions defined on them. Every action has the effect of changing some attribute values and may also depend on further attribute values which are, however, not changed. The subset of attributes affected by an action is known as precondition. The goal is to find some sequence of executable actions that transform a given initial state into a desired target state. This type of problem appears, e.g., in robot motion planning. In this paper, we study cases of the problem where the precondition of every action only depends on a conjunction of terms with at most two attributes. We classify a number of cases as polynomial-time solvable or NP-complete. They amount to extended versions of some classic graph problems, among them topological orderings and perfect matchings. This appears to be the first systematic study of preconditions, despite the rich literature on many aspects of path finding in finite state spaces. A complete dichotomy of polynomial-time and NP-complete cases remains an open question.

© 2025 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY license (<http://creativecommons.org/licenses/by/4.0/>).

1. Introduction

Consider a finite discrete state space where states are described by the values of Boolean attributes. (Finite discrete variables can always be split into binary ones.) Moreover, a set of actions is available. Every action changes the state. We wish to find some sequence of actions that transforms an initial state into a target state, maybe even a shortest such sequence. Since the number of states is exponential in the number of attributes, it would be impractical to turn the problem simply into a reachability or shortest-path problem in a directed graph of states and actions. There are not only too many states, but every action would also produce exponentially many arcs. More specifically, we are interested in actions each of which changes only a few attributes while the values of other attributes are irrelevant, as this is the typical case in reality.

We encountered this type of problem within an interdisciplinary project dealing with teaching robots to perform tasks by watching human demonstrations. A system records the demonstrations and infers states, actions, and their preconditions. However, some actions are misinterpreted, and invalid sequences proposed which would not even be executable. It arises the (sub-)problem to get from some correct state to another prescribed state by a valid sequence of actions that can replace an erroneously inferred sequence.

The complexity of path finding in huge implicit graphs has been extensively studied in the literature under the label of reconfiguration problems, as in, e.g., [5,6,9,10]. However, this has been mainly considered for transformations of solutions of specific graph problems to other solutions by certain permitted steps. Similarly, rearrangement problems for objects

* Corresponding author.

E-mail addresses: ptr@chalmers.se (P. Damaschke), karinne@chalmers.se (K. Ramirez-Amaro).

have been studied mainly in special domains, like genome rearrangements (with a vast amount of articles), puzzles [4], and composing trains from wagons at a shunting yard, as in [3,7]. We are not aware of a systematic study of path finding considering all possible abstract forms of actions and preconditions (or at least such involving only a few attributes). Our techniques in some cases resemble problems known as routing permutations via matchings [1] and as token swapping and token sliding [2,8,12], but, briefly said, the results in the literature deal with other questions and aspects.

Overview of contributions. We initiate a systematic study of the complexity of finding action paths in state spaces where every available action changes only a few attributes, and the preconditions are conjunctions of terms, each depending on only a few other attributes. Actually, we focus on at most two attributes per action and term, which already gives rise to many nontrivial cases. In Section 2 we introduce the setup and formal notation.

Now we summarize the technical part. In Sections 3 and 4 we solve some cases that can be reduced to the perfect matching problem. They also reveal some general feature of our problem: while some classic graph algorithms are used to pick a subset of actions from an instance, we must also schedule these actions to obtain a transformation from the initial state into the target state that respects the preconditions at all times. This adds new “dynamics” to some classic graph structures. Sections 5 to 8 deal with cases that involve breaking of directed cycles of actions. Section 9 reports on a case that can be solved by viewing it as some token production and transportation problem. Section 10 shows the technically hardest case we could solve, by a combination of a generalized perfect matching problem with (again) token production and transportation along certain paths.

2. Preliminaries

2.1. Actions and preconditions

Let A be our set of n attributes. There are 2^n possible states, each described by the binary values, 0 or 1, of all attributes. By renaming the values, all attribute values can be assumed to be 0 in the *initial state*. For some given bipartition $A = A_0 \cup A_1$, $A_0 \cap A_1 = \emptyset$, all values of the attributes in A_i ($i = 0, 1$) must be i in the *target state*.

An *action* changes the values of some attributes, but it may also depend on further attributes that are not changed. That is, the action is executable only for restricted values (or combinations of values) of further attributes. For instance, an action may depend on the presence of some tools or on other conditions that are not changed themselves.

Example. In order to get a nail into a board, first the board and a still unused nail must be ready on site, but also a hammer must be present (precondition, being a conjunction of some attribute values describing properties or spatial relationships of objects). After the action of hammering in, the nail is attached to the board (changed attribute value), but also the hammer is still on the spot and is not affected (unchanged attribute value). As the next action, the hammer may be moved to another place where it is needed, and so on.

An action is, therefore, formally described by the following ingredients.

- a set $V \subseteq A$ of attributes whose values are changed by the action,
- the values of all attributes in V before the action,
- another set $U \subseteq A$ of attributes with $U \cap V = \emptyset$ (and possibly $U = \emptyset$),
- a Boolean function p of the attributes in U , indicating that the action is executable if and only if $p = 1$ holds for the current attribute values in U .

We call p together with the fixed values in V the *precondition* of the action.

2.2. Conjunctive preconditions

A particularly relevant scenario is that the precondition of every action

- has only a few attributes whose values are changed by the action (that is, a small V),
- and a function p being a **conjunction** of functions each of which only depends on a few variables, too.

Sometimes we refer to these Boolean functions joined by conjunction as the *terms* of p .

The emphasis on conjunctions may need some motivation. (But the reader may skip this paragraph without losing the thread.) Preconditions might be known from real-world knowledge, but they might also be inferred by the system from observations and statistics: The action under consideration has been observed only in certain states, and under the assumption that only a few attributes are relevant, e.g., limited by some small constant r , one can infer a set U with $|U| \leq r$ and the function p on it. But U is in general not uniquely determined, and to be on the safe side we can then take the union of all candidate sets and take the conjunction of the mentioned functions, or in simpler words, all potential preconditions must be satisfied.

But also preconditions as such are often just conjunctions of small terms, or even of single attribute values: some object is present, some device is switched on, some tool is ready for use, some object is at some place or in a robot's hand, some place is free, etc.

Since an immense number of different Boolean functions may appear in preconditions, but conjunctions of single-attribute terms cover already many such practical cases, we will only study the latter ones in the present paper, in order to make a meaningful start.

2.3. Notation for simple conjunctive preconditions

We introduce some handy notation and terminology suited for the mentioned cases.

Consider any pair of (a) an action and (b) one of the terms in its precondition. The *signature* of such a pair consists of the following ingredients.

- the number of attributes of value 0 changed into 1, each denoted by a + symbol,
- the number of attributes of value 1 changed into 0, each denoted by a – symbol,
- the type of the term, that is, the function up to permutations of its attributes.

By the definition of V , the total number of + and – symbols is $|V|$. We write every signature as a string of + and – symbols, each indicating a changed attribute value, and a symbol for the Boolean function. The ordering of symbols in the string is arbitrary but will be chosen conveniently.

Terms depending on only a single attribute can simply be denoted by 1 and 0: The only (non-constant) such functions are the identity and the negation, that is, the single attribute must have value 1 and value 0, respectively.

For clarity and referencing we also state our problem formally:

ACTION SEQUENCE

Instance: a finite set A of attributes divided into A_0 and A_1 , and a set of available actions.

Find: a sequence of actions that transforms the state where all attribute values are 0 into the state where all attribute values in A_0 and A_1 are 0 and 1, respectively.

The preconditions of the given actions have signatures, as introduced above. We write the signatures that may occur in an instance in parentheses. For example, “case (–, ++ 0)” of ACTION SEQUENCE means: There may exist actions that change a single attribute from 1 to 0, and other actions that change two attributes from 0 to 1, but are executable only if some other attributes all have value 0.

2.4. Attribute graphs

In some cases it is helpful to think of a problem instance as a graph that we call an *attribute graph*. We will define it in an ad-hoc way for each case. Its vertex set is A , and every edge, arc, loop, hyperedge, etc., represents an action along with one term in its precondition. But remember that all terms must be satisfied before an action can be executed; see, e.g., example (b) in Fig. 1.

A *loop* is an edge on one vertex. An undirected edge joining two vertices is simply called an *edge*, whereas a directed edge is called an *arc*. We call u the *tail* and v the *head* of the arc uv from u to v . For every action with signature (++) or (––), we may create an edge joining the two changed attributes, also called a *positive* or *negative edge*, respectively. For every signature with two different symbols, we may create an arc whose meaning depends on the symbols.

Often we do not distinguish between an attribute, its value, and its vertex, without risk of confusion. For instance, by “changing an attribute” we mean changing its value.

Sometimes it can be convenient to think of attribute values as *tokens*. A vertex free of tokens has attribute value 0, whereas a token on a vertex indicates attribute value 1. A (–+) arc uv then means that a token can be moved from u to v , when v is free. See example (c) in Fig. 1.

The phrase “case (.)” of the ACTION SEQUENCE problem means that only actions with the mentioned signatures appear in the instances.

Special remark. It is important to realize that preconditions must not be contradictory. For instance, a vertex v with (+) loop cannot be the head of a (1+) arc as well: This would not be consistent, because the action that changes $v = 0$ into $v = 1$ either depends on another attribute or not.

Fig. 1 may help the reader keep the notation and terminology in mind.

2.5. Some general observations

The following observations help limit the possible cases somewhat.

For a solution to an instance to exist, some signature must contain more + than –. The reason is simple: In the initial state, all attributes were 0 by definition, whereas the final state has some 1s, hence there must exist some type of action that increases the number of 1s. When an action is applied, every + turns some 0 into 1, but every – turns some 1 into 0.

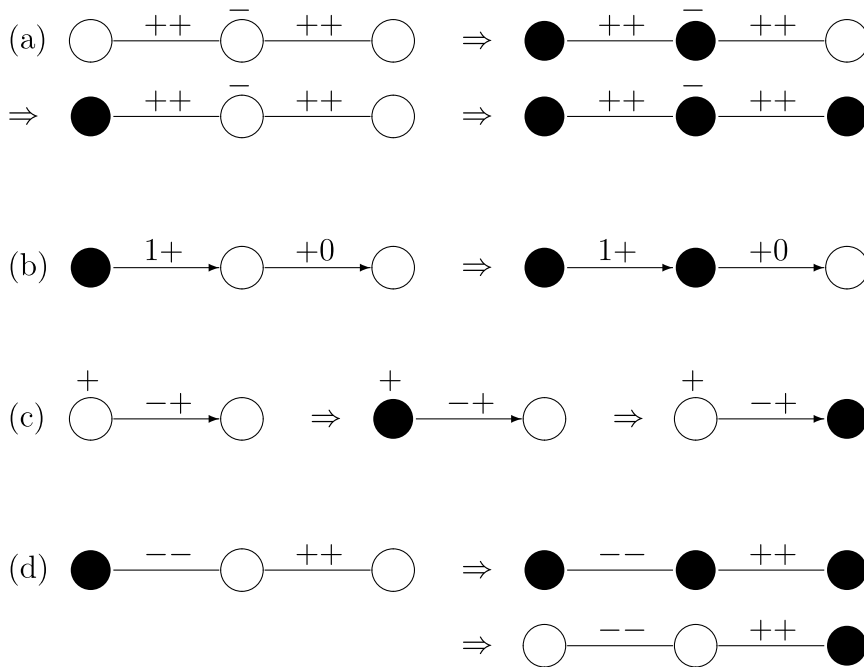


Fig. 1. This illustrates how attribute values change by the application of various actions, related to some of the cases we will solve. White and black circles represent values 0 and 1, respectively. Loops are indicated by a + or – sign at the vertex. The arrows do not mean much; they only indicate directed edges in some graph algorithms we will apply. In (a), the three actions are executed from left to right. In (b), the action changing the value in the middle depends on a conjunction of two terms, and the action is applicable only because all attributes involved have the requested values. That is, one must check all incident edges. Such conjunctions can have arbitrarily many terms. In (c), a token is generated and then moved by one step. In (d), two actions simulate the move of a token, which will be the building block of solving the $(++, --)$ case.

If $(+)$ is the only signature with a symbol $+$, then a $(+)$ action must exist for every attribute in A_1 , these actions can be done in any ordering (as they do not depend on anything), and further actions are useless. Furthermore, some signature must contain at least one symbol $+$ and no symbol $-$ or 1 , since otherwise no action is executable in the initial state.

If no signature contains a symbol $-$, then attributes in A_0 must not be changed to 1, because it would be impossible to reset them to 0. We say in such cases that A_0 is *inert*.

Theorem 1. ACTION SEQUENCE in case $(+++)$ is NP-complete.

Proof. A polynomial-time reduction from 3-DIMENSIONAL MATCHING (3DM for short) to ACTION SEQUENCE with $(+++)$ actions is pretty obvious. Given a 3DM instance, we create an instance of ACTION SEQUENCE where $A_0 = \emptyset$, A_1 is the given set of elements, and every given triple is interpreted as a $(+++)$ action. If the 3DM instance has a solution, we can execute the corresponding actions in an arbitrary order, to cover A_1 exactly. Conversely, if the ACTION SEQUENCE instance has a solution, its actions can change only pairwise disjoint triples of attributes from 0 to 1, and they must cover A_1 , hence they form a solution to the 3DM instance. \square

This NP-completeness observation for a signature of size 3 suggests that it should be interesting to begin a systematic study of the complexity of cases where every signature involves at most two attributes, which we will do in the following.

3. Case $(+, ++)$

For any instance, let the attribute graph on vertex set A consist of loops and edges for the actions with signature $+$ and $++$, respectively. We call a subset of the edges and loops that covers every vertex exactly once a perfect matching (slightly generalizing the usual notion of a perfect matching that includes only edges). The following statement is evident:

Lemma 2. An instance of ACTION SEQUENCE in case $(+, ++)$ has a solution if and only if the subgraph of the attribute graph induced by A_1 has a perfect matching of $(++)$ edges and $(+)$ loops. The actions can then be done in any ordering.

The problem of computing a perfect matching of loops and edges can be reduced to the usual PERFECT MATCHING problem as follows. (This might be folklore, but we give a reduction for the sake of completeness.) Let G be a graph with

n vertices, usual edges, and ℓ loops. We create ℓ fresh vertices if $n - \ell$ is even, and $\ell + 1$ fresh vertices if $n - \ell$ is odd. All fresh vertices are connected to each other, and to all vertices with loops. Let H be the graph obtained in this way.

We show equivalence of the problem instances. Define $r = (n - \ell) \bmod 2$. Thus, H contains $\ell + r$ fresh vertices. Let M be some perfect matching in G with λ loops. We replace every loop in M with some edge to a fresh vertex, disjoint from the other edges in M . The number of fresh vertices not yet covered by M is now equal to $(\ell + r) - \lambda = (n - \ell) - (n - \ell) + r$. Since $n - \ell$ is even, this number is always even. Thus we can add pairwise disjoint edges to M that cover the remaining fresh vertices, which yields a perfect matching in H . Conversely, let M be some perfect matching in H . We replace every edge vv' in M between a loop vertex v and a fresh vertex v' with the loop at v . These loops together with the edges from M in G form a perfect matching in G .

Together with [Lemma 2](#) this yields:

Theorem 3. ACTION SEQUENCE in case $(+, ++)$ is solvable in polynomial time.

As a remark, one can even find the minimum number of actions, by assigning unit costs to the original edges and the edges between loop vertices and fresh vertices, and zero costs to the edges between fresh vertices, and compute a minimum-cost perfect matching.

4. Case $(-, ++)$

For any instance of this case, we construct an attribute graph in a less obvious way: We create an edge for every pair of vertices with a $(++)$ action, and a loop at every vertex u which has an incident $(++)$ edge uv with a $(-)$ action on v . (A loop is like a “virtual” $+$ action.) The proof of the following result shows why this definition is meaningful.

Theorem 4. ACTION SEQUENCE in case $(-, ++)$ is solvable in polynomial time.

Proof. Suppose that a given instance has a solution. Consider any $(++)$ action on uv performed there. If some $(-)$ actions on u are done later, then the next such $(-)$ action can be done immediately after the $(++)$ action. (As long as u keeps value 1, no other $(++)$ actions can involve u anyhow.) The same statement holds for v . Hence, if later $(-)$ actions are done on both u and v , then we can delete three actions from the sequence. We conclude: Either no later actions are done on u and v at all, or some later $(-)$ action is done immediately thereafter on only one vertex, say v , with the combined effect that the value of u changes from 0 to 1 and does not change further. This yields a perfect matching of edges and loops on A_1 .

Conversely, suppose that M is a perfect matching of edges and loops on A_1 . We show that the instance has a solution, by giving an algorithm that constructs it.

We compute a perfect matching as in Section 3. As long as two loops in M are joined by some $(++)$ edge, we pick such a pair and replace the two loops with that edge. This step is repeated until the set L of loops in M is an independent set. Now, actions are scheduled as follows. For every loop $u \in L$, we choose some incident edge uv and perform $(++)$ on uv and $(-)$ on v . This is done for all loops in any sequential order. Since L is independent, we have $v \notin L$, hence these actions do not interfere and are always executable. After that, we perform the $(++)$ actions on all edges of M , again in any sequential order. \square

Finally we remark:

Theorem 5. ACTION SEQUENCE in case $(+, -, ++)$ is solvable in polynomial time.

This can be shown by combining the cases $(+, ++)$ and $(-, ++)$. But since the ideas are quite different, we have presented those two cases separately for better readability.

5. Cases $(+0, 1+, \dots)$ without $-$ symbols in other signatures

In this section we collect some general observations for the mentioned cases, which will be applied in subsequent sections.

An “arc” uv from vertex u to vertex v refers to a $(+0)$ or a $(1+)$ arc, if nothing else is specified. In a $(+0)$ arc, u allows a $+$ action if v has value 0, and in a $(1+)$ arc, v allows a $+$ action if u has value 1.

Since A_0 is inert, it holds for all actions that the attributes with $+$ and 1 symbols are in A_1 . Furthermore, any $(+0)$ arc uv from A_1 to A_0 can be removed, and if u is not the tail of any further $(+0)$ arc uv to some $v \in A_1$, then a $(+)$ loop at u must be created instead. Hence, we can assume that all arcs are in A_1 .

Consider any arc uv applied in a solution, that is, some action changes exactly one attribute and depends on the value of the other attribute. If uv is a $(1+)$ arc, then $u = 1$ must hold before we can change v . If uv is a $(+0)$ arc, then u can be changed only as long as still $v = 0$. In summary, the value of u must be changed earlier than the value of v .

Due to these precedence constraints, the set of these arcs cannot contain a directed cycle. In that case, the subgraph spanned by all arcs is a directed acyclic graph (DAG), and the attributes therein must be changed by following some topological ordering. But these are only necessary conditions for a solution.

Further details as well as the complexity depend on the signatures being present, as we shall see. The demand to destroy directed cycles of arcs leads to variants of the DIRECTED FEEDBACK VERTEX SET (DFVS) problem. A DFVS is a set of vertices that hits every directed cycle. Given a directed graph with n vertices and an integer k , the DIRECTED FEEDBACK VERTEX SET problem asks to find a DFVS of k vertices. The problem is known to be NP-complete, and it can be trivially solved in $O^*(2^n)$ time. Remarkably, not much progress has been made in improving this time bound; the state-of-the-art result is still the one in [11].

6. Case (+0, 1+, +)

Theorem 6. ACTION SEQUENCE in case (+0, 1+, +) is solvable in polynomial time.

Proof. As seen in Section 5, we can assume that all arcs and loops are in A_1 . Moreover, every vertex of A_1 is either a (+) loop, or the tail of some (+0) arcs or the head of some (1+) arcs (or both), since otherwise no solution can exist. As we know from Section 5, a directed cycle of arcs rules out any solution. Therefore, suppose that the arcs form a DAG. Then we can change all attributes in A_1 from 0 to 1 by following any topological order. This will not violate any precondition. \square

7. Cases (1+, ...) (+0, ...) without further + symbols in other signatures

Theorem 7. ACTION SEQUENCE in cases (1+, ...) and (+0, ...) without further + symbols in other signatures is solvable in polynomial time.

Proof. Trivially, no solution can exist for instances of case (1+, ...), because every action with a + has some 1 in its precondition, hence we cannot start changing any attribute values from 0 to 1.

We turn to case (+0, ...). If some (+0) arcs in A_1 form a directed cycle, then no solution exists, because we cannot change the last value 0 to 1 in such a cycle. Hence, suppose that the (+0) arcs in A_1 form a DAG. Then some vertices lack outgoing (+0) arcs in A_1 . Suppose that each of them has some outgoing (+0) arc with head in A_0 though. (Otherwise, no solution exists.) Then we can change all values in A_1 from 0 to 1 by following any topological ordering and applying +0 actions only. This also implies that other available actions are not needed here. \square

8. Cases (++, +0) and (++, 1+)

Theorem 8. ACTION SEQUENCE in the cases (++, +0) and (++, 1+) is NP-complete.

Proof. We give a polynomial-time reduction from DIRECTED FEEDBACK VERTEX SET.

Given a directed graph G and an integer k , being an instance of the DIRECTED FEEDBACK VERTEX SET problem, we can first delete all vertices not being in directed cycles (and all their incident arcs). This does not change the problem, since such vertices are useless in a DFVS. Thus, we can assume that every vertex in G is contained in some directed cycle.

Next, we create k fresh vertices and a (++) edge for every pair of a fresh vertex and an original vertex. We also interpret all given arcs either as (+0) arcs or as (1+) arcs. All vertices are in A_1 , that is, A_0 is empty. The construction can be done in polynomial time.

Suppose that some DFVS X of size k exists. Let Y be the set of the other original vertices not being in X . We fix some matching M of (++) edges that covers all fresh vertices and the set X . Since X is a DFVS, the subgraph induced by Y is a DAG. To give all attributes the value 1, we schedule actions as follows. In case (++, +0), we follow any topological ordering to raise all values in Y to 1. Since every vertex $v \in Y$ was in some directed cycle in G , it has some outgoing (+0) arc that can be used for v , moreover, the heads of all these outgoing arcs in $Y \cup X$ still have value 0 when v is processed. After that, we do all (++) actions in M . In case (++, 1+), we first perform all (++) actions in M . Then we follow any topological ordering to raise all values in Y to 1. Since every vertex $v \in Y$ was in some directed cycle in G , it has some incoming (1+) arc that can be used for v , moreover, the tails of all these incoming arcs in $Y \cup X$ have already the value 1 when v is processed.

Conversely, suppose that some action sequence raises all attribute values to 1. The (++) edges used to serve the k fresh vertices must be pairwise disjoint, and no further (++) actions may be applied, since otherwise some vertex would obtain a value larger than 1. Let X be the set of the k original vertices covered by the (++) actions, and Y the remainder. The attribute values in Y can be raised by arc actions only. If the subgraph induced by Y contains some directed cycle, then we cannot set all values to 1 therein, without violating some precondition (as observed in Section 5). It follows that X is a DFVS. \square

In an instance of ACTION SEQUENCE in our considered cases, let n denote the total number of vertices in A_1 that appear in directed cycles. Despite NP-hardness, upper bounds on the time complexity $t(n)$ remain of practical interest. As mentioned in Section 5, as far as is known, even the plain DIRECTED FEEDBACK VERTEX SET problem cannot be solved much faster than in $O^*(2^n)$ time in graphs with n vertices. Furthermore, our reduction in Theorem 8 shows that any path finding algorithm for the case (++, +0) or (++, 1+) yields an DFVS algorithm running in $t(n)$ time. Hence we can hardly expect algorithms for these cases being much faster than $O^*(2^n)$. Therefore, it may be interesting to notice:

Theorem 9. ACTION SEQUENCE in cases $(++, +0)$ and $(++, 1+)$ is solvable in $O^*(2^n)$ time.

Proof. Consider an instance of either problem. Recall that all arcs and all $(++)$ edges can be assumed to be in A_1 , as well as some $(+)$ loops created in a preprocessing phase. Let $Z \subseteq A_1$ be the union of all directed cycles in A_1 . For each of the 2^n subsets $X \subseteq Z$ we do the following. We check in polynomial time whether (i) X is a DFVS and (ii) some matching of $(++)$ edges and $(+)$ loops covers exactly X . If these conditions are fulfilled, we can raise the values of all vertices in $A_1 \setminus X$ by following some topological ordering, and we can raise the values in X before and after that, in case $(++, 1+)$ and $(++, +0)$, respectively (as we did in our reduction for Theorem 8). This yields $O^*(2^n)$ -time algorithms for both cases. \square

The combined case $(++, +0, 1+)$ is NP-complete due to Theorem 8, but the question of an exact algorithm is more tricky: Suppose that we have decided on some matching that covers an DFVS X . Consider two vertices $x, y \in X$ such that xy is a $(++)$ edge, and there exists some $(+0)$ arc ux and some $(1+)$ arc yv . Then u must be changed before the $(++)$ action on xy , and v must be changed afterwards. This imposes a new arc uv that expresses some precedence constraint rather than an action. All arcs together must form a DAG. Thus, not only the chosen DFVS matters, but also the chosen matching on it. Enumerating and trying all possible matchings would increase the time bound to some $O^*(2^{n \log n})$. Here, improvements might be possible by a deeper problem analysis.

9. Case $(+, -+)$

Theorem 10. ACTION SEQUENCE in case $(+, -+)$ is solvable in polynomial time, even with the optimal number of actions.

Proof. For treating this case, we call the vertices with $(+)$ loops *sources*, as they can produce tokens that must eventually cover all vertices of A_1 . (Recall the notion of tokens from Section 2.) We define $d(u, v)$ as the length, i.e., number of arcs, of some shortest directed path of $(-+)$ arcs from u to v , if some exists. We denote the sources in arbitrary order u_1, u_2, u_3 , etc.

To every $v \in A_1$ we assign the source u that minimizes $d(u, v)$, and if several sources do, we take the source with minimum index among them. (This implies $u = v$ in the case when v itself is a source.) If some $v \in A_1$ is not reachable from any source, then no solution exists. For every $v \in A_1$ and its assigned source u , we also fix (arbitrarily) some shortest path from u to v which we call the *delivery path* to v .

Now, assume that P is the delivery path to p , starting in u_i , and Q is the delivery path to q , starting in u_j , where $i < j$, and that P and Q intersect in some vertex x . If $d(u_i, x) > d(u_j, x)$, then $d(u_j, p) \leq d(u_j, x) + d(x, p) < d(u_i, x) + d(x, p) = d(u_i, p)$, hence we would have rather assigned u_j to p . If $d(u_i, x) \leq d(u_j, x)$, then $d(u_i, q) \leq d(u_i, x) + d(x, q) \leq d(u_j, x) + d(x, q) = d(u_j, q)$, hence we would have rather assigned u_i to q . Since both cases yield contradictions, we conclude that delivery paths from distinct sources do not intersect. Hence, a token from one source can never block the delivery paths from other sources.

From every source u we send tokens to the assigned vertices $v \in A_1$ in the order of decreasing $d(u, v)$; ties are broken arbitrarily.

The latter rule ensures that each delivery path is free when it is needed. The obtained solution also uses the optimal number of actions, since the token that is finally placed on any vertex $v \in A_1$ must come from some source, and choosing a nearest source for every v is trivially optimal. Furthermore, all computations can be done in polynomial time. \square

If further actions exist, however without $+$ symbols, then these additional actions are useless, hence these cases can be solved exactly as case $(+, -+)$. The reason is as follows. As seen above, any solution must generate tokens in the sources and move them along directed paths to all vertices of A_1 . If they are all reachable, then we can obviously ignore any additional actions. If some are not reachable, then additional actions that only delete tokens do not make them reachable.

10. Case $(++, --)$

This is by far the most intricate case that we have settled. We begin with a high-level outline of this section. First we show that every solvable instance has a so-called quasi-matching with alternating paths (Definition 13). The proof (of Lemma 14) includes an algorithm which will not be part of the final algorithm. The actual algorithm (Theorem 17) starts instead with constructing a quasi-matching from scratch and uses its alternating paths to move tokens to the vertices of A_1 . For this, some further preparations are needed, which are provided in the preceding lemmas.

Lemma 11. If a solution exists, then $|A_1|$ is even.

Proof. The sum of values is initially 0, hence it is even, and every action changes it by $+2$ or -2 , hence it always remains even. \square

Consider any solution to a problem instance. We create t unweighted positive edges between two vertices if the corresponding $(++)$ action is performed t times. Similarly, we create t unweighted negative edges between two vertices if the corresponding $(--)$ action is performed t times. Let the weight of a vertex be the number of incident positive edges minus the number of incident negative edges. We say that a multiset of positive and negative edges, built from copies of the $(++)$ and $(--)$ edges of the given instance, is *feasible* if every vertex in A_i has the weight i . Clearly, the multiset coming from a solution is feasible. Hence we can state immediately:

Lemma 12. *If a solution exists, then a feasible multiset of edges exists.*

Definition 13. A walk is any path that traverses every edge at most once. But a walk may cross itself, i.e., visit vertices repeatedly.

An *alternating path* is a walk consisting of positive and negative edges that appear alternately, and with positive edges at both ends. We remark that any single positive edge also satisfies this condition and is therefore considered an alternating path between its end vertices.

A *quasi-matching* is a partitioning of A_1 into disjoint pairs of vertices $\{u, v\}$ such that, for each of these pairs, there exists an alternating path joining u and v .

We remark that these alternating paths are not required to be disjoint.

Lemma 14. *If a solution exists, then any fixed feasible multiset F of edges permits a quasi-matching.*

Proof. By Lemmas 11 and 12, $|A_1|$ is even, and a feasible multiset F of edges exists. Fix any such F . Besides its edges, we will also create tokens, with the following semantics: A token on a vertex increases its weight by 1, and at most one token is allowed on every vertex. In the beginning, all nodes are free of tokens.

With these notions, we can apply the following procedure to F . We start a walk in any free vertex $u \in A_1$. Since u has weight 1, there exists some positive edge uv . We first traverse this edge, and we continue our walk according to the following rules that exhaust all possible events.

- When we arrive on a positive edge at some free vertex $u' \in A_1$ with $u' \neq u$, we stop.
- When we arrive on a positive edge at u , we continue on some negative edge. It exists, since u has weight 1, and we have traversed two incident positive edges, the first one and the current one, that are not yet compensated by traversed negative edges incident to u .
- When we arrive on a positive edge at some vertex of A_0 , we continue on some unused negative edge. It exists, since the vertex weight is 0.
- When we arrive on a positive edge at some vertex of A_1 with a token, we continue on some unused negative edge. It exists, since the vertex weight is 1 but a token is there, too.
- When we arrive on a negative edge at some vertex, we continue on some unused positive edge. It exists, since the vertex weight is non-negative.

Since we stop only in the first event, we obtain an alternating path that connects two distinct free vertices u and u' of A_1 . Finally, we delete all its edges and place tokens on u and u' . This preserves all vertex weights, since every inner vertex of an alternating path is incident to one positive and one negative edge of that path.

This procedure is repeated as long as possible. Since $|A_1|$ is even, and we always start a walk on some free vertex in A_1 , all vertices in A_1 will eventually have a token. In other words, we have partitioned A_1 into pairs of vertices connected by alternating paths. We remark that there may also remain edges in F that are not used in the alternating paths of the quasi-matching. \square

Remember that the multiple edges came from a given solution and were only a means to prove the existence of a quasi-matching. Given a problem instance (rather than a solution), we create instead an attribute graph with only one positive edge for every $(++)$ action and only one negative edge for every $(--)$ action. From now on, we use tokens to indicate attribute values 1, whereas vertices free of tokens have value 0.

The alternating paths in a quasi-matching were not required to be vertex- or edge-disjoint. Each of them may now also traverse vertices and edges repeatedly (since only one copy of every edge is retained), but the two end vertices of every alternating path are distinct.

Assuming for the moment that we know a quasi-matching on A_1 in the attribute graph, we would like to construct an action sequence that eventually places one token on every vertex of A_1 . This would establish the converse of Lemma 14. As a preparation we first study this task on any single alternating path, that is, placing two tokens at its end vertices.

Lemma 15. *Consider any two distinct vertices connected by some shortest alternating path P . By using only actions on P , we can bring two tokens to the end vertices of P , and keep all other vertices of P free, upon termination of this procedure.*

Proof. P cannot contain an edge xy followed by an edge yx , because we could then remove these two edges from P , contradicting the minimum length. Hence, any three consecutive vertices x, y, z on P are distinct. Next, suppose that xy is negative, yz is positive, x has a token, and y and z are free. A $(++)$ action on yz followed by a $(--)$ action on xy moves the token from x to z .

In the following we think of the path P as the sequence of its edges. Again, remember that some vertices and edges on P may be identical (even on a *shortest* alternating path, due to the condition of being alternating; it is easy to find examples). We denote the vertices on P by u_0, \dots, u_k , where k is odd. We use the equation symbol to indicate identical vertices. Two vertices u_i and u_j where i, j are both even or both odd cannot be identical, $u_i = u_j$, since some alternating path shorter than P would then connect u_0 and u_k .

Let i be the largest index with $u_i = u_0$, and j the smallest index with $u_j = u_k$. If $i > 0$, then i is odd, and if $j < k$, then j is even. If $j < i$, then the subpath from u_j to u_i would be an alternating path shorter than P . This shows $i < j$. We initially pick some positive edge on P that is incident to u_i or u_j , and place two tokens at its vertices by a $(++)$ action.

More generally, consider a situation where the tokens are on u_p and u_q , with the following properties: p is even, q is odd, $p < q$, $i < q + 2$, $p - 2 < j$. These invariants are satisfied initially. Since $p < q$, we have $u_{q+1} \neq u_p$. If also $u_{q+2} \neq u_p$, then both u_{q+1} and u_{q+2} are free, and we can move the token from u_q to u_{q+2} as described above. Similarly, we have $u_{p-1} \neq u_q$, and if also $u_{p-2} \neq u_q$, then we can move the token from u_p to u_{p-2} as described above. If both $u_p = u_{q+2}$ and $u_q = u_{p-2}$, then the tokens are already on u_{p-2} and u_{q+2} . If $p = 0$, then $u_p = u_0 \neq u_{q+2}$, since $i < q + 2$. Similarly, if $q = k$ then $u_q = u_k \neq u_{p-2}$, since $p - 2 < j$. Hence we can always move some token outwards, and the aforementioned invariants are preserved. By iterating this step we obtain a sequence of valid actions that results in two tokens at the ends of P . Moreover, all inner vertices of P which are not identical to end vertices are eventually free again. \square

In the following, let $m := |A_1|/2$ (which is integer by Lemma 11). Consider any quasi-matching with $m := |A_1|/2$ pairs of vertices, fix some alternating path for each pair, and denote these paths by P_1, \dots, P_m arbitrarily. We call such a sequence P_1, \dots, P_m compatible if no vertex of any path P_j is an end vertex of an earlier path P_i , $i < j$.

Lemma 16. *Given a sequence P_1, \dots, P_m of the alternating paths from any quasi-matching on A_1 , we can transform it in polynomial time into a compatible sequence without increasing the total length of these paths.*

Proof. Assume that P_i connects u and v , and P_j connects x and y , where $i < j$. By the definition of a quasi-matching, u, v, x, y are distinct. Further assume that at least one end of P_i , say v , is an inner vertex of P_j . Since P_j is alternating, one edge of P_j incident to v is negative. Say, this edge points towards the end y of P_j . We append the subpath of P_j from v to y to P_i , to form an alternating path from u to y . The remainder of P_j is an alternating path from x to v ; in fact, both ends are positive edges. This procedure transfers edges from some path P_j to some earlier path P_i . Hence we can iterate it only finitely often, and we eventually obtain a sequence of alternating paths which is compatible. The time is obviously polynomial. \square

Theorem 17. ACTION SEQUENCE in case $(++, --)$ is solvable in polynomial time.

Proof. Given a problem instance, we can compute some shortest alternating path between every pair of vertices of A_1 where such a path exists, straightforwardly in polynomial time. Using some perfect-matching algorithm we can then also compute a quasi-matching. If none exists, then there is no solution, due to Lemmas 12 and 14.

In the affirmative case, we take such a quasi-matching with $m := |A_1|/2$ pairs of vertices, fix some shortest alternating path for each pair, and denote these paths by P_1, \dots, P_m arbitrarily. We make this sequence compatible by using Lemma 16. The alternating paths obtained are not necessarily shortest ones. In that case, we replace every alternating path P_i with a shortest one between the same vertices, thus strictly decreasing the total length of all our paths. Now, the paths may no longer be compatible. Then we rearrange them again as in Lemma 16, and so on. Eventually we obtain a compatible sequence of shortest alternating paths, still in polynomial time. Finally, we take these paths P_1, \dots, P_m one by one and place tokens exactly at their end vertices using valid actions, as in Lemma 15. Since the sequence is compatible, all vertices of the current path are free of tokens when we start processing the path. \square

11. Conclusions

We have solved a number of cases of the ACTION SEQUENCE problem, for various signatures of actions, but this can only be a start. We had to limit our scope to actions that change at most two values and whose preconditions are conjunctions of single values. One example of an open case is $(++, -+)$ where tokens can be generated only in certain pairs, which seems to destroy the simple ideas of the case $(+, -+)$. Within the scope of this paper, we could classify the complexity of all other cases containing a $++$ signature. It could be interesting and worthwhile to simplify the algorithm and the proof for the case $(++, --)$. However, the ultimate goal would be a dichotomy, that is, a complete classification of the polynomial and NP-complete cases. Preconditions that contain disjunctions are not yet studied here and will be addressed by further research as well. Disjunctions appear, e.g., if an action requires some tool being ready to hand, but there exist two or more options for it. Furthermore, one could extend the problem, e.g., by a set $A_2 \subset A$ of “don’t care” attributes that can have any value in the target state. An extended study would apparently also need more refined notation, perhaps a whole taxonomy of cases. Finally, this study is only a small part of an interdisciplinary endeavour. For further research we can imagine to prioritize cases that arise in the development of robot systems for executing collaborative tasks.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This work was supported by the Information and Communication Technology Area of Advance (ICT AoA) at Chalmers University of Technology through the seed project “Raw2Rich Graphs: Learning on-demand human actions based on graph theory” in 2024.

This article is also dedicated to the 60th birthday of Ralf Klasing. We believe that it has a similar spirit as parts of his oeuvre dealing with search and motion planning in graphs, although the exact type of problems and the technical contents are, of course, very different.

The authors thank the anonymous reviewers for their constructive comments and suggestions that helped clarify the framework and a number of details.

Data availability

No data was used for the research described in the article.

References

- [1] N. Alon, F.R.K. Chung, R.L. Graham, Routing permutations on graphs via matchings, *SIAM J. Discrete Math.* 7 (3) (1994) 513–530, <http://dx.doi.org/10.1137/S0895480192236628>.
- [2] V. Bartier, N. Bousquet, A.E. Mouawad, Galactic token sliding, *J. Comput. System Sci.* 136 (2023) 220–248, <http://dx.doi.org/10.1016/j.jcss.2023.03.008>.
- [3] M. Gatto, J. Maue, M. Mihalák, P. Widmayer, Shunting for dummies: An introductory algorithmic survey, in: R.K. Ahuja, R.H. Möhring, C.D. Zoroiliagis (Eds.), *Robust and Online Large-Scale Optimization: Models and Techniques for Transportation Systems*, in: *Lecture Notes in Computer Science*, vol. 5868, Springer, 2009, pp. 310–337, http://dx.doi.org/10.1007/978-3-642-05465-5_13.
- [4] O. Goldreich, Finding the shortest move-sequence in the graph-generalized 15-puzzle is NP-hard, in: O. Goldreich (Ed.), *Studies in Complexity and Cryptography. Miscellanea on the Interplay between Randomness and Computation - In Collaboration with Lidor Avigad, Mihir Bellare, Zvika Brakerski, Shafi Goldwasser, Shai Halevi, Tali Kaufman, Leonid Levin, Noam Nisan, Dana Ron, Madhu Sudan, Luca Trevisan, Salil Vadhan, Avi Wigderson, David Zuckerman*, in: *Lecture Notes in Computer Science*, vol. 6650, Springer, 2011, pp. 1–5, http://dx.doi.org/10.1007/978-3-642-22670-0_1.
- [5] M. Grobler, S. Maaz, N. Megow, A.E. Mouawad, V. Ramamoorthi, D. Schmand, S. Siebertz, Solution discovery via reconfiguration for problems in P, in: K. Bringmann, M. Grohe, G. Puppis, O. Svensson (Eds.), *51st International Colloquium on Automata, Languages, and Programming, ICALP 2024*, July 8–12, 2024, Tallinn, Estonia, in: *LIPIcs*, vol. 297, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024, pp. 76:1–76:20, <http://dx.doi.org/10.4230/LIPIcs.ICALP.2024.76>.
- [6] T. Ito, E.D. Demaine, N.J.A. Harvey, C.H. Papadimitriou, M. Sideri, R. Uehara, Y. Uno, On the complexity of reconfiguration problems, *Theoret. Comput. Sci.* 412 (12–14) (2011) 1054–1065, <http://dx.doi.org/10.1016/j.tcs.2010.12.005>.
- [7] F. Kamenga, P. Pellegrini, J. Rodriguez, B. Merabet, Solution algorithms for the generalized train unit shunting problem, *EURO J. Transp. Logist.* 10 (2021) 100042, <http://dx.doi.org/10.1016/j.ejtl.2021.100042>.
- [8] H. Kiya, Y. Okada, H. Ono, Y. Otachi, Sequentially swapping tokens: Further on graph classes, in: L. Gasieniec (Ed.), *SOFSEM 2023: Theory and Practice of Computer Science - 48th International Conference on Current Trends in Theory and Practice of Computer Science, SOFSEM 2023, Nový Smokovec, Slovakia, January 15–18, 2023, Proceedings*, in: *Lecture Notes in Computer Science*, vol. 13878, Springer, 2023, pp. 222–235, http://dx.doi.org/10.1007/978-3-031-23101-8_15.
- [9] A.E. Mouawad, N. Nishimura, V. Raman, N. Simjour, A. Suzuki, On the parameterized complexity of reconfiguration problems, in: G.Z. Gutin, S. Szeider (Eds.), *Parameterized and Exact Computation - 8th International Symposium, IPEC 2013, Sophia Antipolis, France, September 4–6, 2013, Revised Selected Papers*, in: *Lecture Notes in Computer Science*, vol. 8246, Springer, 2013, pp. 281–294, http://dx.doi.org/10.1007/978-3-319-03898-8_24.
- [10] N. Ohsaka, Gap amplification for reconfiguration problems, in: D.P. Woodruff (Ed.), *Proceedings of the 2024 ACM-SIAM Symposium on Discrete Algorithms, SODA 2024, Alexandria, VA, USA, January 7–10, 2024, SIAM, 2024*, pp. 1345–1366, <http://dx.doi.org/10.1137/1.9781611977912.54>.
- [11] I. Razgon, Computing minimum directed feedback vertex set in $o(1.9977^{11})$, in: G.F. Italiano, E. Moggi, L. Laura (Eds.), *Theoretical Computer Science, 10th Italian Conference, ICTCS 2007, Rome, Italy, October 3–5, 2007, Proceedings, World Scientific, 2007*, pp. 70–81.
- [12] K. Yamanaka, E.D. Demaine, T. Ito, J. Kawahara, M. Kiyomi, Y. Okamoto, T. Saitoh, A. Suzuki, K. Uchizawa, T. Uno, Swapping labeled tokens on graphs, *Theoret. Comput. Sci.* 586 (2015) 81–94, <http://dx.doi.org/10.1016/j.tcs.2015.01.052>.