Qualitative Evaluation of Stability in Disassembling Block Structures with Robot Manipulator

Przemysław Andrzej Wałęga
Institute of Philosophy
University of Warsaw, Poland
p.a.walega@gmail.com

Michał Zawidzki
Department of Logic
University of Łódź, Poland
zawidzki@filozof.uni.lodz.pl

Jakub Możaryn
Institute of Automatic Control and Robotics
Warsaw University of Technology, Poland
mozaryn@mchtr.pw.edu.pl

Abstract

In this paper, we present an approach to disassembling block structures, i.e., taking subsequent blocks off from the pile one by one, so that we can safely pull out a concrete block from the remaining structure without making it collapse. At the heart of the approach lies an algorithm that determines whether removing a particular block from the pile makes it lose its stability. Our algorithm is based on a qualitative representation of a structure and exploits such notions as "lying on", "lying under", and "being stable", as well as numerical data, e.g., positions of blocks' mass centers. In order to perform the task of disassembling block structure, we devised a method searching for a minimal sequence of blocks that need to be taken off from the pile before a designated block in order to safely pull out this block. We have implemented this method in the Robotic Operating System (ROS) framework and have used a 3D computer simulator V-REP to test its performance. Although the approach is based on a qualitative representation, in most of the conducted simulations the algorithm correctly predicted stability of an investigated structure and thus determined a way of its safe dissembling.

Introduction

By *disassembling block structure* we mean the following task: given a structure of blocks use a single robotic arm that can manipulate only one block at a time to pull out a selected block without making any blocks fall down. An example of a block structure before and after disassembling is depicted in Fig. 1.

In order to disassemble a block structure with a guarantee that it will not collapse, it is essential to evaluate its stability while we perform this task, e.g., it should be determined that removing a selected block (e.g., b_6 in Fig. 1) will not result in a collapse of any other block in the structure.

It appears that solving the stability checking task is crucial in disassembling structures. Checking whether a structure is stable is a thorougly investigated problem with multiple references to the literature. In most cases it takes the form of one of the following tasks:

• *Block-piling* – use a single robotic arm that can manipulate only one block at a time to pile a copy of a given structure of blocks (Blum, Griffith, and Neumann 1970);

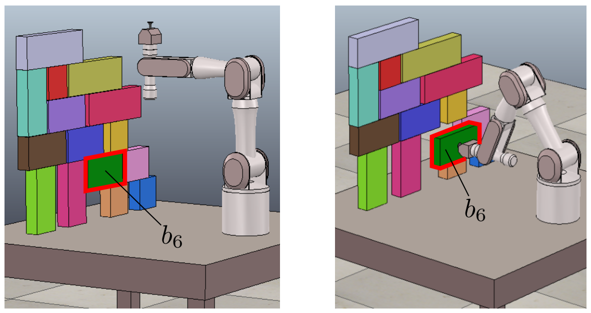

Figure 1: The initial (a) and the final (b) stage of a task of disassembling a block structure. Specifically, the task was to take off the block b_6 which was recognized not to affect the stability of the whole structure and as such could be removed directly.

• *Product assembling* – plan an order in which product's subparts need to be assembled, where one of the main constraints imposed on the order is that at each stage a current subassembly needs to remain stable against the gravity force and the insertion force of the next part to be assembled (Boneschanscher et al. 1988);

• *Image understanding* – given a 2D image infer relations occurring between the recognized objects. In particular, inferring physical relations between objects often requires stability checking, e.g., in order to determine which object "supports" others (Gupta, Efros, and Hebert 2010; Zhang and Renz 2014).

A number of theoretical results and algorithms for stability checking have been established within the abovementioned fields. It has been shown that checking if a configuration of polygonal objects in a 2D plane is stable is co-NP-complete regardless of whether friction is taken into account (Palmer 1989). On the other hand, when we impose certain restrictions on the types of contact between polygons, the problem becomes tractable (i.e., in P) (Palmer 1989; Boneschanscher et al. 1988).

One of the best known algorithms for stability checking has been proposed in (Blum, Griffith, and Neumann 1970). The approach is based on linear programming – for each block in a structure a force balance equation is constructed and additional linear inequalities are introduced to the system in order to encompass friction. The method consists in searching for a solution to the system of equations (and inequalities). A structure is said to be stable iff there exists a solution such that all forces acting on objects take non-negative values.

Another numerical approach has been introduced in (Siskind 2000; Siskind 2004), where a kinematic analysis of a two-dimensional configuration of pentagonal blocks is undertaken. Depending on the type of movement that is possible between two blocks: rotating, sliding, and others, the approach distinguishes 3 types of joints, namely revolute, prismatic, and rigid, respectively. To each segment of the structure a linear and angular velocity is ascribed and a system of linear equations (and inequalities) is constructed. Checking if the whole scene is stable is equivalent to the existence of a solution of this system, such that the so-called entropy variable introduced to the scheme can be equated to zero.

In contrast to numerical approaches for stability checking, there are also approaches based on *qualitative* methods, where representation rests upon symbolic rather than numerical values (Cohn and Renz 2008). For instance, in (Zhang and Renz 2014) the authors present a list of intuitive rules using which one can determine if a block in a 2D scene is stable. The rules take into account the relative position of the block's center of mass and its supports. For instance, the main rule states that "an object is stable and will not topple if the vertical projection of the centre of mass of an object falls into the area of support base." This approach has been recently used for automatic generation of stable scenes in a computer game (Stephenson and Renz 2016).

In this paper, we present a qualitative approach in which we construct a symbolic representation of a configuration of blocks using notions such as "lying on", "lying under", and "being stable." The representation and assumptions we made are presented in Sect. "Assumptions and Representation."

For stability control we exploit a qualitative method from (Wałęga, Zawidzki, and Lechowski 2016) which was previously applied to reason about physical properties of a 2D scene in a computer game environment. Afterwards, we embed this stability checking method into an algorithm for disassembling block structures. Both methods are described in Sect. "Disassembling Algorithm."

The entire method was implemented in Robotic Operating System (ROS) (Quigley et al. 2009) which is a standard framework used to program real robots as well as their computer models. The algorithm has been tested in a powerful 3D computer simulator V-REP (Rohmer, Singh, and Freese 2013) in a scenario in which a robotic manipulator equipped with a vacuum gripper is used to take off selected blocks from a given structure. Implementation, simulations, and discussion of obtained results are presented in Sect. "Implementation and Simulations."

Finally, concluding remarks and directions of the future work are formulated in Sect. "Conclusions."

Assumptions and Representation

In this paper, we consider the disassembling block structure task in a scenario, where blocks are 3D cuboids and a robotic manipulator is capable of manipulating only one block at a time. Given an initial block structure and a specified block to be eventually pulled out, the disassembling task is to determine a minimal sequence of blocks that need to be taken off in order to securely pull out the specified block.

In order to make the task simpler we put the following restrictions on the scenario:

- Blocks are convex and their depth dimension is the same (which is often the case in warehouses);
- No block lies aslant on the pile, i.e., all of them are aligned in parallel to the ground line (which is a natural assumption for a warehouse use-case);
- The mass of each block is known in advance (we can assume that it was weighted before piling) and distributed uniformly among the whole block;
- We have an access to a two-dimensional representation of the structure which enables us to determine the arrangement of the blocks (in real applications such a representation may be provided by machine vision algorithms);
- While taking off the blocks no insertion force occurs and friction is disregarded (in practice such a situation may be achieved by immobilizing all blocks touching the block that is being manipulated at the moment).

We are aware that the abovementioned assumptions are idealistic and in practice it may happen that not all of them will be satisfied simultaneously, however we believe that the approach presented in this paper can be successfully extended onto more general cases in the future.

We assume that the two-dimensional representation of a structure is a bitmap enabling us to distinguish individual blocks. We construct a qualitative representation of the blocks' arrangement as follows. Let P be the set of all pixels in this representation, let each block b_i be a subset of P, and denote a set of all blocks by B. We say that a block b_1 $lies\_on$ a block b_2 whenever b_1 is adherent to b_2 from above. Formally, for any blocks $b_1, b_2 \in B$ let $lies\_on(b_1, b_2)$ if:

$$b_1 \neq b_2 \wedge \exists p_1, p_2 \in P\Big(p_1 \in b_1 \wedge p_2 \in b_2$$
$$\wedge x(p_1) = x(p_2) \wedge y(p_1) < y(p_2) < y(p_1) + c\Big),$$

where c is a constant value, and $x(p_i)$, $y(p_i)$ are the x- and y-coordinates of a pixel p_i, respectively. In words, b_1 $lies\_on$ b_2 if they are different blocks and there are pixels p_1, p_2 such that p_1 belongs to b_1, p_2 belongs to b_2, the x-coordinate of these pixels is the same and p_1 has a bigger y-coordinate than p_2, but p_1 is not further way from p_2 than a fixed constant c. In perfect conditions p_1 would be a pixel strictly adhering p_2 from above (then we would set $c = 2$). However, in practice blocks' contours may be inaccurately represented and hence we set a constant $c > 2$ as the maximal value of possible error in determining contours of blocks. We denote the transitive closure of the relation $lies\_on$ by $lies\_on^*$. If b_1 $lies\_on$ b_2 we say that b_2 $lies\_under$ b_1.

The $lies\_on$ relation is irreflexive and for convex polygons also asymmetric (this asymmetry breaks down in the case of non-convex polygons). It also resembles the intuitive concept of "being supported by" and under the assumptions we previously made (namely the convexity of blocks and a non-slanting arrangement of all of them) these two notions indeed collapse.

For each two blocks b_1, b_2 such that $lies\_on(b_1, b_2)$, we define the left- and rightmost pixels lying on a border between b_1 and b_2, denoted by $left\_p(b_1, b_2)$ and $right\_p(b_1, b_2)$, respectively.

Disassembling Algorithm

Our algorithm for disassembling block structures consists of a method for stability checking (we call it *Qualitative Stability* – QS in short) and a method determining which blocks need to be removed in order to securely pull out a selected one. (we call it *Pile Decomposition* – PD in short)

Stability Checking Algorithm

First, we explain how the QS algorithm works. Due to space restrictions the description we provide below is concise. For a fully-fledged characterization of the algorithm we refer the reader to (Wałęga, Zawidzki, and Lechowski 2016). Throughout the presentation of the algorithm, we use Fig. 2(b) as our running example.

We apply QS by taking as an input a concrete block b we intend to pull out from the pile B. Afterwards, we proceed in the following steps.

First, we determine a set A_b of all blocks lying directly or indirectly on b[1]. Afterwards we find a set U_b of all blocks lying (directly) under $\mathfrak{A}_b$ except for b itself. Intuitively, after pulling out b $\mathfrak{A}_b$ rests on $\mathfrak{U}_b$. Note that $\mathfrak{U}_b$ need not be connected. In our figure, if we want to pull out b_6, then $A_{b_6} = \{b_7, b_8, b_{10}, b_{11}, b_{13}, b_{14}, b_{15}\}$ and $U_{b_6} = \{b_3, b_5, b_9, b_{12}\}$.

Second, we fix three points:

1. the center of mass for $\mathfrak{A}_b$ ($center(\mathfrak{A}_b)$, p_c in short),
2. the leftmost connection point between $\mathfrak{A}_b$ and $\mathfrak{U}_b$ ($left\_p(\mathfrak{A}_b, \mathfrak{U}_b)$, p_l in short),
3. the rightmost connection point between $\mathfrak{A}_b$ and $\mathfrak{U}_b$ ($right\_p(\mathfrak{A}_b, \mathfrak{U}_b)$, p_r in short),

We call the last two the *pivot points* of $\mathfrak{A}_b$. In Fig. 2(b) p_c, p_l, p_r are represented by a red, blue and green hollow dot, respectively.

Third, we check whether the vertical projection[2] of $\mathfrak{A}_b$'s center of mass protrudes beyond any of the pivot points. If it is not the case, we proceed to step **Fourth**. Otherwise, we perform Step **Sixth**. In the figure, $x(p_l) < x(p_c) < x(p_r)$.

Fourth, If the abscissa of $\mathfrak{A}_b$'s center of mass is between the abscissas of both pivot points (which is the case in our figure), it means that $\mathfrak{A}_b$ would remain stable as a single object, however the stability of some components of A_b could have been violated. To verify the latter, we check, one by one, each block which was lying on b before it was pulled out (in the figure, b_7 and b_8). For each such block b_i we fix two sets: LO_{b_i} of all blocks lying directly on b_i and LU_{b_i} of all blocks lying directly under b_i. If LU_{b_i} is empty, it means that b_i has no supporting blocks and falls. Therefore, we launch the QS method for b_i as an input and for the pile of blocks reduced by the current list of fallen blocks. In our figure, $LO_{b_7} = \{b_{10}\}$, $LU_{b_7} = \{b_5\}$, $LO_{b_8} = \{b_{10}, b_{11}\}$, and $LU_{b_8} = \{b_3\}$.

Fifth, If LU_{b_i} is not empty, which means that there exist blocks supporting b_i, it is still possible that b_i has lost its stability. We verify it by checking vertical projections of $\mathfrak{LO}_{b_i}$ and b_i's mass center. If both of them entirely protrude beyond (the vertical projection of) $\mathfrak{LU}_{b_i}$ from one side (either left or right), it means that there is no counterbalance for the forces acting on b_i on this side from above. Consequently, b_i looses its stability and falls. Thus, we launch the QS method for b_i as an input and for the pile of blocks reduced by the current list of fallen blocks.

Sixth If the vertical projection of $\mathfrak{A}_b$'s center of mass protrudes beyond one of the pivot points, $\mathfrak{A}_b$ must have lost its balance and in effect all elements A_b fall. Consequently, we add them to the list of fallen blocks. Roughly, if a centre of mass of the structure $\mathfrak{A}_b$ is located horizontally between utmost connection points between $\mathfrak{A}_b$ and its underlying structure $\mathfrak{U}_b$, in most cases it remains stable (unless the procedure from steps **Fourth** and **Fifth**, called *Check*, proved the opposite). Otherwise we assume that all elements of A_b fall.

Seventh, we begin with constructing a set U_b^+ consisting of the elements of U_b and all blocks lying directly under b (in the figure there exists one such block, namely o_2). If $\mathfrak{A}_b$ has not collapsed, we further consider only the blocks located directly under b. Otherwise we consider all elements of U_b^+. For each such element u_i we proceed analogously to step **Second**, namely we check whether the vertical projection of its center of mass does not protrude beyond any of the pivot point between this element and the underlying structure. If it is the case, u_i falls we launch the QS method for u_i as an input and for for the pile of blocks reduced by the current list of fallen blocks. Otherwise, it remains stable, so we proceed to the next U_b^+'s element u_{i+1} and we repeat the current step.

A pseudocode of the QS method is demonstrated in Alg. 1 (QUALSTAB), Alg. 2 (FALL) and Alg. 3 (CHECK). Intuitively, Alg. 1 serves to initialize a list $fall$ that is to be filled. Alg. 2 investigates whether whole sets of blocks treated as monolithic objects will fall once a particular block has fallen or been pulled out. Alg. 3 enables us to check if particular elements of a pile that Alg. 2 evaluated as stable will indeed remain stable.

The QS method always terminates. Indeed, by the fact that the number of blocks in the pile is limited, the FALL algorithm can be recursively called only a finite number of times – it cannot be called for the same block twice, which is secured in lines: 2 (which prevents Alg. 3 from calling FALL infinitely many times in lines 3 and 8) and 13 of Alg. 2 (which prevents it from calling FALL infinitely many times in lines 22 and 41). Moreover, each call of the FALL method consists of a finite number of steps since all lists: A_b, U_b,

[1]Henceforth, we will denote an object composed of all elements of a set S, i.e., a mereological sum of pixels constituting the images of all elements of S, by a corresponding symbol $\mathfrak{S}$ written in gothic font. For example, $\mathfrak{A}_b$ will denote an object built of all pixels of all elements of A_b.

[2]By "vertical projection" we mean the projection in the direction of the x-axis.

U_b^+ and C_{b_i} are finite (again, by the fact that there is only a finite number of blocks in the pile). The same applies to the CHECK method. It can only be called finitely often since all lists A_b that CHECK relies on are finite and since there are only finitely many blocks in the pile, only a finite number of such lists can be created during operation of the method.

Pile Decomposition Algorithm

The algorithm presented in Sect. "Stability Checking Algorithm" is an intermediate step in applying the whole framework to some real-life scenarios (like, e.g., rearranging piles of blocks). Once we have picked a block b we want to pull out and launched the QS algorithm which returned a list of blocks that will potentially fall afterwards, we would like to find out which blocks should we take off from the pile B to get able to securely pull out b. One answer is that if all elements of the list *fall*(b, B) were taken off, we could definitely take out b without a risk. This estimation, however, seems exaggerated. The proposition formulated below comprises the following observation: if we want to remove the smallest possible number of blocks to safely pull out b, we can confine ourselves to blocks from *fall*(b, B).

Proposition 1. *Let s be a sequence of blocks such that removing them from a pile B in the same order they occur in the sequence suffices to safely pull out b. Then the sequence $s \cap$ fall(b, B) also enjoys this property.*

Proof. According to the proposition if there is a combination of blocks whose removal from the pile B ensures that b can be securely pulled out, the same can be obtained if we take off from the pile only those blocks from this combination which are elements of *fall*(b, B). We will prove that the proposition holds by showing that if b_- is a block from outside *fall*(b, B), then *fall*(b, B) = *fall*$(b, B \setminus \{b_-\})$, i.e., if b_i falls down upon pulling out b from B, then b_i will also collapse if we pull out b from the pile reduced by a block b_- from outside *fall*(b).
Case 1: Suppose that b_i was added to *fall*(b, B) by the algorithm CHECK (executing line 3). Then by the structure of CHECK (line 2 and 6), we can derive that b_i has lost its balance because both its center of mass and forces acting on it from above protrude beyond its supports on the same side. Since both of the above conditions are satisfied conjunctively, even if we remove blocks pressing b_i from above, its center of mass still protrudes beyond its supports, thus making it fall. If, in turn, we remove those supports, b_i will fall by lines 2-3 of CHECK.
Case 2: Suppose that b_i was added to *fall*(b, B) by the execution of line 23 of FALL. It means that b had previously pressed it from above, but after we pulled out b, b_i's center of mass protrudes beyond b_i's supports having no counterbalance. Therefore, removal of no block can prevent it from falling. It means that after we pull out b, all elements of *fall*(b, B) under any conditions.
Case 3: Suppose that b_i was added to *fall*(b, B) by the execution of line 15 of FALL. It means that after pulling out b, b_i fell out with all blocks placed directly and indirectly above it (and possibly with some other blocks it rested on).

Algorithm 1 QUALSTAB(b,B)

Input: a block b and a pile B such that $b \in B$
Output: a list $fall$ of all blocks that fall if b is pulled out from B

```
1: Initialize an empty list of blocks $fall$;
2: Run FALL($b$, $fall$);
```

Algorithm 2 FALL(b, $fall$)

Input: a block b and an initialized (possibly non-empty) list $fall$
Output: an updated list $fall$

```
1: Add $b$ to $fall$;
2: Set a list of blocks $A_b \leftarrow \{b' \in B \mid lies\_on^*(b', b) \wedge b' \notin fall\}$;
3: Initialize a new pseudo-block $\mathfrak{A}_b$; // $\mathfrak{A}_b$ is understood as a mereological sum of the elements of $A_b$
4: Set a float $x(\mathfrak{A}_b) \leftarrow$ weighted average of abscissas of all $A_b$'s elements' mass centers; // an abscissa of $\mathfrak{A}_b$'s mass center
5: Set lists of blocks $U_b \leftarrow \{b' \in B \mid b' \notin fall \cup A_b \wedge \exists b'' \in A_b\ lies\_under(b', b'')\}$, $U_b^+ \leftarrow U_b \cup \{b' \in B \mid lies\_under(b', b) \wedge b' \notin fall\}$;
6: if $U_b$ is empty then // if $b$ turned out to be the only block lying under $\mathfrak{A}_b$
7:     Set $fall \leftarrow fall \cup \bigcup A_b$;
8: else
9:     Initialize a new pseudo-block $\mathfrak{U}_b$; // $\mathfrak{U}_b$ is understood as a mereological sum of the elements of $U_b$
10:    if $x(\mathfrak{A}_b)$ lies between the pivot points between $\mathfrak{A}_b$ and $\mathfrak{U}_b$ then // if the $\mathfrak{A}_b$' mass center lies between both utmost connection points between $\mathfrak{A}_b$ and the structure $\mathfrak{U}_b$ lying under $\mathfrak{A}_b$
11:        for each block $b_i$ such that $lies\_on(b_i, b)$ do
12:            Run CHECK($b_i$, $fall$);
13:        end for
14:    else
15:        Set $fall \leftarrow fall \cup \bigcup A_b$;
16:    end if
17: end if
18: for each block $b_i$ from $U_b^+$ do
19:    Set a list $C_{b_i} \leftarrow \{b' \in B \mid lies\_under(b', b_i)\}$;
20:    if $C_{b_i}$ is not empty then
21:        Initialize a new pseudo-block $\mathfrak{C}_{b_i}$; // $\mathfrak{C}_{b_i}$ is understood as a mereological sum of the elements of $C_{b_i}$
22:        if $x(b_i)$ does not lie between the pivot points between $b_i$ and $\mathfrak{C}_{b_i}$ then
23:        Run FALL($b_i$, $fall$);
24:        end if
25:    end if
26: end for
```

Algorithm 3 CHECK(b, $fall$)

Input: a block b and an initialized list $fall$
Output: an updated list $fall$

```
1: Set lists $LO_b \leftarrow \{b' \in B \mid lies\_on(b', b)\}$, $LU_b \leftarrow \{b' \in B \mid lies\_under(b', b)\}$;
2: if $LU_b$ is empty then // it means that $b$ has no support from below
3:     Run FALL($b$, $fall$);
4: else
5:     Initialize new pseudo-blocks $\mathfrak{LO}_b$, $\mathfrak{LU}_b$; // $\mathfrak{LO}_b$, $\mathfrak{LU}_b$ are understood as mereological sums of $LO_b$, $LU_b$, respectively;
6:     if vertical projections of both $x(b)$ and $\mathfrak{LO}_b$ entirely protrude beyond $\mathfrak{LU}_b$ from one side then // if two conditions are jointly satisfied: 1) $b$'s mass center protrudes beyond the adherent structure $\mathfrak{LU}_b$ lying under $b$ from the left (right), 2) the adherent structure $\mathfrak{LO}_b$ pressing $b$ from above also protrudes beyond $\mathfrak{LU}_b$ from the left (right)
7:         Run FALL($b$, $fall$);
8:     end if
9: end if
```

By lines 9-10 and 15 of FALL the only blocks that can affect the stability of the structure b_i has fallen within, and

Algorithm 4 PILEDEC(b)

Input: a block b

Output: a shortest sequence of blocks one can safely remove in order to take off b, or $\emptyset$ if no such sequence exists.

```
 1: $fall \leftarrow$ QUALSTAB(b,B) $\setminus \{b\}$;
 2: if $fall = \emptyset$ then
 3:     return $b$; // If $fall = \emptyset$, then $b$ may be straightforwardly pulled out
 4: else
 5:     Initialize an empty list $failed$; // $failed$ is a list of sequences of blocks whose removal leads to a collapse of the pile
 6:     for each $i = 1, i \leq |fall|, i{+}{+}$ do
 7:         for each sequence $s$ of $i$ distinct elements from $fall$ do
 8:             if $s \setminus s[i] \in failed$ then // $s[i]$ denotes the $i$-th element of the sequence $s$. Hence, $s \setminus s[i]$ is the sequence $s$ without its last element
 9:                 add $s$ to the list $failed$;
10:             else
11:                 if QUALSTAB($s[i], B \setminus \{s \setminus s[i]\}$) $\neq \{s[i]\}$ then // the QS is launched for the last element of $s$ and $B$ reduced by the elements of $s \setminus s[i]$
12:                     add $s$ to the list $failed$;
13:                 else
14:                     if QUALSTAB($b, B \setminus \{s\}$) $= \{b\}$ then // the QS method is launched for $b$ and the pile $B$ reduced by the elements of $s$
15:                         return $s^\frown b$; // where $s^\frown b$ is $s$ with $b$ added in the end
16:                     end if
17:                 end if
18:             end if
19:         end for
20:     end for
21:     return $\emptyset$;
22: end if
```

are possibly from outside *fall*(b, B), are elements of the set U_b. Since the aforementioned structure collapsed, its center of mass must had protruded beyond one of pivot points, i.e., p_l or p_r. Without loss of generality, let's assume that p_l was that point. If we remove from U_b a block that constituted p_l, the new p_l will move rightwards, thus making the center of mass of the structure protrude beyond it from the left even more than before. Therefore, the structure will collapse. On the other hand, if we remove from U_b an element which does not constitute any of the pivot points, call it b_j, the QS method executed for b_j might show that a part of the previously considered structure will fall after such a move. This might mean that if we remove b_j together with the part that should collapse from the pile, then after pulling out b the so reduced pile will remain stable. But we assumed that $b_j \notin$ *fall*(b, B), which means that we can remove just the part of the structure, leaving b_j, and obtain the same upshot.

The 3 abovementioned cases exhaust all possible genesis of blocks being included in *fall*(b, B). They showed that removing blocks from outside *fall*(b, B) prior to pulling out b cannot reduce *fall*(b, B) itself. Consequently, when considering which blocks should we simultaneously remove from the pile for it to remain stable after pulling out b, we can confine ourselves to the elements of *fall*(b, B). Hence the conslusion. □

Our ultimate goal is to find an algorithm that will assist us in decomposing a concrete pile of blocks, namely it will indicate a sequence of blocks which has to be taken off from the pile in order to pull out b, if we want to avoid collapsing at least a part of the initial pile. Let us denote $fall(b, B) \setminus \{b\}$ by $fall$. Then by Prop. 1 we can restrict our search space to sequences of elements of $fall$. It follows that only sequences of the length at most $|fall|$ need to be examined, where by $|fall|$ is the number of elements of $fall$. At the current stage of research, we propose a brute-force method which seeks for the best way to disassemble the pile in order to securely pull out the block b, i.e., a shortest sequence of blocks whose removal makes pulling out b safe.

Below, we are presenting a concise description of a proposed brute-force algorithm searching for the best (shortest) way to decompose a pile of blocks. We call this algorithm Pile Decomposition (PD in short). A more formal and precise presentation of PD can be found in the form of pseudocode in Alg. 4 (PILEDEC).

Roughly speaking, within the PD algorithm for each $i \leq |fall|$ we investigate all sequences of distinct blocks from the list $fall$, starting from shortest ones (i.e., for $i = 1$). Let s denote a sequence of length i of distinct elements from $fall$. For any each tested sequence we address two questions:

1. Does taking off blocks from a sequence s from the pile with preservation of the order in which they occur in s lead to a collapse of (a part of) the pile?
2. Does pulling out our target block b from the pile reduced by the elements of s lead to a collapse of (a part of) such a pile?

If s is the first sequence investigated by the algoirithm for which the answers to both questions are negative, then the algorithm returns $s^\frown b$, i.e., a sequence obtained by adding b in the end of s, as the solution. Since we start our search from shortest sequences, gradually incrementing their length, it guarantees that if the algorithm provides a sequence as the solution, it is a shortest solution (not necessarily unique). If for no sequence the answers to both abovementioned questions are negative, it means that there exists no solution to our problem of pile decomposition, i.e., we cannot disassemble it by removing blocks one by one, and the algorithm returns $\emptyset$. For instance, if you consider blocks b_2, b_5, b_6, and b_8 from Fig. 2(c) as constituting a separate pile, then it is infeasible to safely pull out b_2 – since b_6 and b_8 are balancing each other, one cannot take these blocks off one by one without causing a collapse (in the pseudocode of Alg. 4 we automatically add such a sequence to the *failed* list).

For each possible length i of a sequence of blocks ($1 \leq i \leq |fall|$) there exist $\frac{|fall|!}{(|fall|-i)!}$ different sequences consisting of distinct elements. It means that in the worst case our PD algorithm will launch the QS algorithm $1 + 2 \cdot \sum_{i=1}^{|fall|-1} \frac{|fall|!}{(|fall|-i)!}$ times. In practice, the number of QS calls can be significantly lower from this estimation. In particular, our algorithm constructs a list $failed$ of sequences of blocks, whose removal lead to a collapse of the pile. While checking if taking off blocks from a sequence s leads to a collapse of a structure, at first, we check if some prefix of s is already in the list $failed$, i.e., is known to lead to a collapse. If it is the case, we can quickly conclude that s should be added to $failed$.

Implementation and Simulations

The simulation environment consisted of 2 parts: Virtual Robot Evaluation Platform (V-REP) and Robot Operating System (ROS). **V-REP** is one of the most advanced yet intuitive robot simulation environments that allows creating or importing robot 3D CAD models, equipping them with sensors and effectors, and testing them in a prebuilt simulation environment. **ROS** is a flexible environment for writing programs for different types of robots comprising a set of tools, libraries and programming methods designed to simplify the task of creating complex programs running on different hardware platforms.

The scene modelled in V-REP is shown in Fig. 1(a). It consists of a 7-axis robotic manipulator equipped with a vacuum gripper and 15 blocks numbered from b_1 to b_{15}. Blocks were arranged in a pile with few restrictions described in Sect. "Assumptions and Representation." Moreover, the structure of blocks had to be stable in the beginning and not move during the simulation. The robotic manipulator was set at a distance allowing access to and gripping of each block. The gripping process was carried out by means of an astrictive end-effector in form of vacuum gripper (Fig. 1(b)).

All simulations were performed at the standard PC station with Ubuntu 14.04, ROS Indigo 1.11.20 and V-REP PRO EDU v3.3.0. In the proposed solution client $\leftrightarrow$ server communication was used, while the program written in ROS was the client and the V-REP simulator was the responding server. The main code was written in a V-REP environment in LUA scripting language with C-style pseudocode (interpreter v5.1.4) (Ierusalimschy, de Figueiredo, and Celes 1996). It allowed moving simulated robot arm, performing gripping and disassembling the block structure, and changing the simulation parameters.

QS Algorithm Verification

Simulations conducted in the simulation environment were designed to test the performance of the algorithm. At the beginning, for the evaluated scene, structural stability was checked by running the simulation while the simulated robot was off. It was assumed that the block layout is stable if none of the blocks are moving. Then, one after another the stability of the system was checked for every block (or, to be more precise, for the structure resulting in pulling out this block). In the case when the block layout lost its stability and the part of the structure collapsed, information about the displaced blocks had been recorded, and these blocks were considered "collapsed." Finally, the QS algorithm was tested. With the simulation enabled, the algorithm analyzed the block layout and instructed the robot to pull out and then insert back blocks that would not collapse the structure. In the logs available in the ROS console information was saved about which blocks collapsed after pulling one of them.

For each configuration, based on the results collected for the QS algorithm and simulations, we created a classification matrix (Fig. 3), where a row number denotes the index of a removed block, and in columns the information about the block's collapse is given. In each cell, green color ("+") means collapse predicted by the QS algorithm and observed, red color means unpredicted collapse of the block ("-"). Collapses predicted but not observed are marked in blue.

Configuration C1 The first considered configuration (Fig. 2(a)) represents the situation in which the blocks are tightly packed within a rectangle. This may happen in real-life cases where blocks are placed and removed from shelves in a warehouse.

It can be seen in classification matrix (Fig. 3(a)) that, for the considered layout of blocks, QS algorithm classified as "safe" (situation without stability loss) and drew all possible blocks, i.e., b_2, b_3, b_5, b_6, b_{13}, and b_{15}. For b_4 and b_8 QS algorithm assumed that more blocks would collapse. In both cases, this is due to the fact that the centre of gravity of the block group, over b_4 or b_8 blocks, is beyond the support points. For example, the algorithm recognizes that all blocks over b_4 will collapse, which in this particular situation is not the case. In this situation, b_{11} and b_{14} should be considered independently – their centre of gravity is located between the support points. On the other hand, for b_7 and b_{11} QS algorithm did not predict the collapse of all the blocks. This is due to the fact that when the construction collapsed, then the dynamics of changes had not been considered.

Configuration C2 Configuration C2 (Fig. 2(b)) reflects the situation where layers of blocks are shifted to the previous layers.

QS algorithm classified as stable situations and drew all possible blocks: b_1, b_3, b_5, b_6, b_8, b_{12}, b_{13}, b_{14}, and b_{15}. Additionally, by the analysis of the classification matrix (Fig. 3(b)), it can be seen that the block b_{10} was incorrectly considered as removable and caused a collapse of b_6 and b_7. This is due to the fact that when searching the structure, the algorithm checked the block b_7 and then finished the search. To detect the fall of b_6 and b_7, the algorithm should also check b_6, which will lose its stability and as a consequence, also b_7 will collapse. Moreover, QS algorithm predicted that more blocks would collapse after removing b_4 and b_9 than it was in the simulation. The reason for this false inference is the same as for the configuration C1.

Configuration C3 The last configuration C3 (Fig.2(c)) represents an irregular pile of blocks. Such a system could arise as a result of external disturbances or by removing the blocks from the pile.

QS algorithm classified as stable situations and drew all possible blocks: b_1, b_2, b_3, b_4, b_7, and b_{15}. However, as it can be seen in the classification matrix (Fig. 2(c)), it improperly classified, as possible to remove, b_6 and b_{14}. Drawing-off these blocks caused the collapse of the structure. For b_6, the reason of false classification is that the centre of gravity of the block group $\{b_{10}, b_{12}, b_{14}\}$ lays between the support points and QS algorithm recognized that the block group will be stable. The incorrect classification for b_{14} is due to the fact, that only the position of the centre of gravity of b_{13}, in relation to the support points, was taken into account during the run of CHECK (see Alg. 3). The algorithm does not include block b_{15} laying on b_{13} - both will collapse after removing b_{14}.

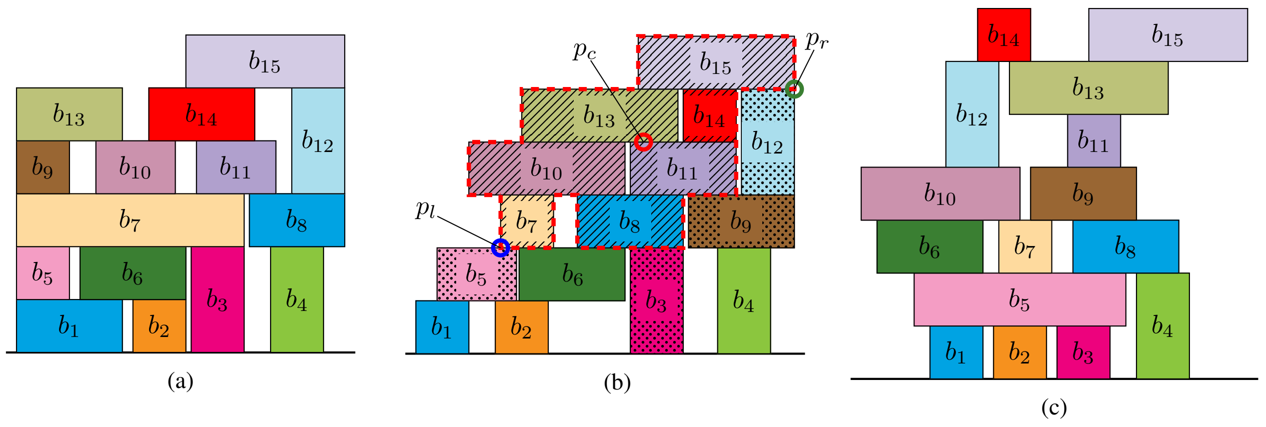

Figure 2: Pile configurations (a) C1, (b) C2, (c) C3. In (b) the striped blocks are the components of $\mathfrak{A}_{b_6}$, whereas the dotted ones are the components of $\mathfrak{U}_{b_6}$.

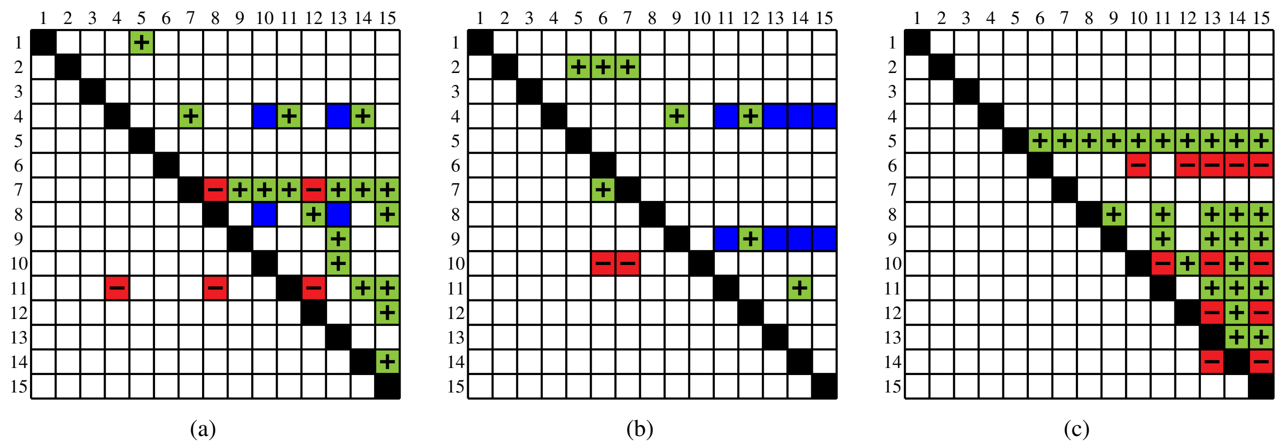

Figure 3: Classification matrices for configurations (a) C1, (b) C2, and (c) C3. Blue, red ("-"), and green ("+") cells denote collapse predicted by QS which did not occur in simulation, unpredicted collapse, and fulfilled predictions, respectively.

As depicted in Fig. 3, the number of proper / possible to draw / false classifications for configurations C1 – C3 is 6/9/6, 6/9/6, and 0/1/2, respectively. The proposed algorithm classified all blocks possible to draw with few mistakes. Based on the results obtained, it can be stated that the predictions of QS algorithm are accurate, but it does not entirely prevent critical situations.

PD Algorithm Verification

The PD method is at the early stage of development but we have already conducted simulations of its performance for various configurations. In what follows we present two interesting cases, namely PD launched for the block b_2 from the configuration C2 and for b_{11} from C3. The former presents a correct and non-trivial decomposition, whereas the latter shows that PD may provide incorrect answers due to inaccuracy of the QS algorithm which is used within PD. The results are presented in Tab. 1.

The output of the PD algorithm is a shortest sequence of elements removing which enables to pull out the input block. Since there might be more than one such sequence, PD returns a sequence that was found first. Consequently, the output depends on the searching method, which was not specified in the pseudocode (see line 7 of Alg. 4).

Table 1: Simulation of PD algorithm. In both cases the shortest decomposition is of length 4, so we list all sequences of length at most 3 which belong to the list $failed$.

	C2, block b_2	C3, block b_{11}
Output:	(b_6, b_7, b_5, b_2)	$(b_{14}, b_{15}, b_{13}, b_{11})$ or $(b_{15}, b_{14}, b_{13}, b_{11})$
$failed$:	(b_7),(b_5, b_6),(b_5, b_7), (b_6, b_5),(b_7, b_5),(b_7, b_6), (b_5, b_6, b_7),(b_5, b_7, b_6), (b_6, b_5, b_7),(b_7, b_5, b_6), (b_7, b_6, b_5),(b_7, b_5, b_6), (b_7, b_6, b_5)	(b_{13}),(b_{13}, b_{14}),(b_{13}, b_{15}), (b_{14}, b_{13}),(b_{15}, b_{13}), (b_{13}, b_{14}, b_{15}), (b_{14}, b_{13}, b_{15}), (b_{15}, b_{13}, b_{14}), (b_{13}, b_{15}, b_{14})

In the simulation of PD for the configuration C3 and block b_{11} we put in Tab. 1 all possible outputs which may occur as a result of applying different searching methods. The first one, i.e., $(b_{14}, b_{15}, b_{13}, b_{11})$ is unsafe, namely after pulling out b_{14}, b_{15} falls to the right. The sequence is treated as a correct one by PD due to inaccuracy of the QS algorithm – see row 14 of Fig. 3c: according to QS pulling out b_{14} is safe but in reality it leads to the fall of b_{13} and b_{15}.

Conclusions

ROS / V-REP environment enabled the simulation of industrial robot performance in an isolated robotic cell to investigate the quality of QS and PD algorithms. This significantly accelerated our research in qualitative reasoning. It allowed focusing on the essence of the qualitative reasoning without considering some aspects of working with real robots, e.g., machine vision. In addition, the simplicity of the simulator allowed quick reproduction and analysis of different scenarios – which is hard to achieve at the real laboratory stand.

The qualitative reasoning in the form of, e.g., QS or PD algorithms, can be used to assist human employees in making a complex decision with collaborative robots (*cobots*) in a shared workspace (Peshkin et al. 2001). Examples of such human-machine collaboration are: unloading of a parcel storage in warehouses (Wurman and Romano 2015) or unloading of transport trucks (Bonkenburg 2016). In such cases, it is necessary for the robot to be able to quickly identify and make a collaborative decision with human employees on the particular item from the group of other objects without losing the stability of the entire system. Moreover, for workers to accept robots as collaborators, their designers must ensure smooth human-machine interaction. It can be realized by teaching robots tasks by demonstration (Rozo et al. 2016) or explanation (Suddrey et al. 2017). In such research and applications the use of qualitative reasoning can be an interesting approach.

Results of the simulation proved that QS algorithm fulfils its role – it allows to predict the stability of the examined configuration of blocks. Important advantage of QS algorithm is the exchange of complex equations into simple rules. However, simulations also showed that this algorithm is not an ideal solution and it occasionally incorrectly indicated the blocks that can be pulled out. Therefore, for industrial applications, that require the system's reliability, it is necessary to extend the QS algorithm with additional qualitative rules to increase its efficiency and robustness.

The most natural directions for future research lead towards optimizing QS algorithm by reducing the number of incorrect guesses and of PD method by reducing the number of sequences of blocks the algorithm thoroughly investigates. One of possible heuristics, that could supplement the current version of the latter algorithm, exploits the proof of Prop. 1 (cases 1 and 2) according to which blocks added to the list by execution of line 23 of FALL or line 3 or 7 of CHECK fall upon pulling out b regardless of other conditions. The revised version of the PD method could then consider only those sequences which include all such blocks.

Acknowledgments. The authors are supported by the National Science Centre, Poland (grant number: DEC-2011/02/A/HS1/00395). We acknowledge substantial contribution from Marcin Jakubowski for work on simulations.

References

[Blum, Griffith, and Neumann 1970] Blum, M.; Griffith, A.; and Neumann, B. 1970. A stability test for configurations of blocks - AI Memo 188. Technical report, MIT Artificial Intelligence Laboratory.

[Boneschanscher et al. 1988] Boneschanscher, N.; van der Drift, H.; Buckley, S. J.; and Taylor, R. H. 1988. Subassembly stability. In *AAAI 88*, 780–785.

[Bonkenburg 2016] Bonkenburg, T. 2016. *Robotics In Logistics: A DPDHL perspective on implications and use cases for the logistics industry*. DHL Customer Solutions and Innovation.

[Cohn and Renz 2008] Cohn, A. G., and Renz, J. 2008. Qualitative spatial representation and reasoning. *Handbook of knowledge representation* 3:551–596.

[Gupta, Efros, and Hebert 2010] Gupta, A.; Efros, A.; and Hebert, M. 2010. Blocks world revisited: Image understanding using qualitative geometry and mechanics. *Computer Vision–ECCV 2010* 482–496.

[Ierusalimschy, de Figueiredo, and Celes 1996] Ierusalimschy, R.; de Figueiredo, L. H.; and Celes, W. 1996. Lua – an extensible extension language. *Software Pract Exper* 26:635–652.

[Palmer 1989] Palmer, R. S. 1989. Computational complexity of motion and stability of polygons. Technical report, Cornell University.

[Peshkin et al. 2001] Peshkin, M.; Colgate, J. E.; Wannasuphoprasit, W.; Moore, C.; Gillespie, B.; and Akella, P. 2001. Cobot architecture. *IEEE T Robotic Autom* 17(4):377–390.

[Quigley et al. 2009] Quigley, M.; Conley, K.; Gerkey, B.; Faust, J.; Foote, T.; Leibs, J.; Wheeler, R.; and Ng, A. 2009. ROS: an open-source robot operating system. In *ICRA–OSS 09*, volume 3.

[Rohmer, Singh, and Freese 2013] Rohmer, E.; Singh, S. P.; and Freese, M. 2013. V-REP: A versatile and scalable robot simulation framework. In *IROS 2013*, 1321–1326. IEEE.

[Rozo et al. 2016] Rozo, L.; Calinon, S.; Caldwell, D. G.; Jiménez, P.; and Torras, C. 2016. Learning physical collaborative robot behaviors from human demonstrations. *IEEE T Robot* 32(3):513–527.

[Siskind 2000] Siskind, J. M. 2000. Visual event classification via force dynamics. In *AAAI 2000*, 149–155.

[Siskind 2004] Siskind, J. M. 2004. Method of determining the stability of two dimensional polygonal scenes. United States Patent no. US 6,693,630 B1.

[Stephenson and Renz 2016] Stephenson, M., and Renz, J. 2016. Procedural generation of complex stable structures for Angry Birds levels. In *IEEE–CIG 2016*.

[Suddrey et al. 2017] Suddrey, G.; Lehnert, C.; Eich, M.; Maire, F.; and Roberts, J. 2017. Teaching robots generalizable hierarchical tasks through natural language instruction. *IEEE Robot Autom Let* 2(1):201–208.

[Wałęga, Zawidzki, and Lechowski 2016] Wałęga, P. A.; Zawidzki, M.; and Lechowski, T. 2016. Qualitative physics in Angry Birds. *IEEE T Comp Intel AI* 8(2):152–165.

[Wurman and Romano 2015] Wurman, P. R., and Romano, J. M. 2015. The amazon picking challenge 2015 [competitions]. *IEEE Robot Autom Mag* 22(3):10–12.

[Zhang and Renz 2014] Zhang, P., and Renz, J. 2014. Qualitative spatial representation and reasoning in Angry Birds: The extended rectangle algebra. In *14th International Conference on the Principles of Knowledge Representation and Reasoning*.