NATIONAL INSTITUTE OF APPLIED SCIENCES OF LYON

RESEARCH REPORT

Toward social navigation among movable obstacles

Author:
Benoit RENAULT

Supervisors:
Dr. Fabrice JUMEL
Dr. Jacques SARAYDARYAN
Dr. Olivier SIMONIN

Examiners:
Dr. Jean-François BOULICAUT
Dr. Vasile-Marian SCUTURICI
Dr. Christine SOLNON

INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

A report written for the INRIA Chroma team and submitted in fulfillment of the requirements of the Engineering Degree of the Computer Sciences and Engineering Department (IF)

August 6, 2018

Résumé: L'objectif est de permettre la navigation optimale dun seul robot dans un environnement intérieur partiellement connu, dynamique et modifiable, en respectant les conventions sociales.

Mots-clefs: NAMO, Navigation en milieu modifiable, Navigation Sociale

Abstract: The objective is to allow optimal navigation among movable obstacles of a single robot in a partially known interior setting, while respecting social conventions.

Key-words: NAMO, Navigation Among Movable Obstacles, Social Navigation

Contents

Chapter 1

Introduction

1.1 Problem

1.2 Existing approaches

Chapter 2

Navigation Among Movable Obstacles: state of the art

2.1 Determining appropriate comparison criteria

Hypotheses	Approaches	Performance criteria
Previous knowledge of the map	Path Planning Algorithm(s) and heuristics	Evaluation in a simulated/real setting
Obstacle characteristics	Evaluation and evolution of an obstacle's "movable" characteristic and its associated cost	Computation time
Robot characteristics	Object manipulation maneuver planning	Local/global Optimality
Problem type	Planning taking uncertainty into account	Optimality in distance/time/energy...
		Social acceptability
		Number and Density of obstacles

FIGURE 2.1: Comparison criteria, sorted by type: hypotheses, approaches and performance criterias

2.1.1 Hypotheses

2.1.2 Approaches

2.1.3 Performance criteria

2.2 Comparison and cross-comparison

2.2.1 Comparison tables

2.2.2 Conclusions

2.2.3 Situating our work in the established context

manipulation point

Chapter 3

Study of Wu et. al.'s algorithm for locally optimal NAMO in unknown environments

Given the state of the art presented in the previous chapter and what we aimed to achieve in the available time, we chose to build upon Wu et. al.'s algorithm for locally optimal NAMO in unknown environments.

3.1 Original algorithms

In Wu et. al.'s proposition, the basic idea is to consider either a plan that doesn't involve interacting with obstacles (which can be achieved with any pre-existing path finding algorithm), or a three-steps plan that consists in, first, reaching the obstacle (c_1), second, pushing it in a single direction (c_2), and finally reaching the goal from the position we left the obstacle at (c_3). This is best shown in figure 3.1.

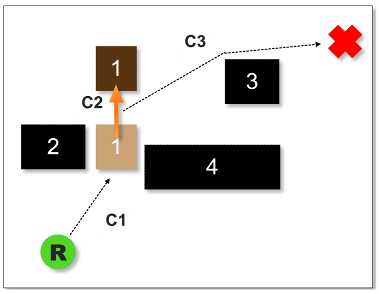


FIGURE 3.1: Figure describing the three plan components when pushing an object, as published in [3].

In their first article [3], Wu et. al. describe two versions of their algorithm: a naive but locally optimal baseline one, and an optimized one, built upon the logic foundation of the first, but that loses its local optimality. In their second article [1], several other improvements on performance are proposed that restore the local optimality of the proposition. By locally optimal, we mean that the algorithm will **always** choose the **best** plan given its current, limited knowledge of the environment.

The baseline algorithm is very straightforward : it uses an A* path finding subroutine to determine the optimal path between the current robot's position and the goal, avoiding all known obstacles (none at the beginning). As the robot moves forward (Figure 3.2a, line 17), and therefore gains new information (same figure, line 5), whenever a new obstacle is encountered, every push action for every obstacle is re-simulated (Figure 3.2b) and compared to the current optimal plan (Figure 3.2a, line 11). Local optimality is guaranteed for this approach, since the A* algorithm is used with the admissible Euclidean heuristic, thus returning optimal solutions for a given state of the map and goal, and also because whenever a new obstacle is detected (= the map is in a new state), **all** possible plans are re-evaluated and compared to check if a better plan than the current one can be found or not before moving the robot again.

```
Algorithm 1 BASELINE($R_{Init}$, $R_{Goal}$)
1:  $R \Leftarrow R_{Init}$;
2:  $\mathcal{O} \Leftarrow \emptyset$; {set of objects}
3:  $p_{opt} \Leftarrow A^*(R_{Init}, R_{Goal})$;
4:  while $R \neq R_{Goal}$ do
5:    $\mathcal{O}_{new} \Leftarrow$GET-NEW-INFORMATION();
6:    if $\mathcal{O}_{new} \neq \emptyset$ then
7:      $\mathcal{O} = \mathcal{O} \cup \mathcal{O}_{new}$;
8:      for each $o \in \mathcal{O}$ do
9:        for each possible push direction $d$ on $o$ do
10:         $p \Leftarrow$ EVALUATE-ACTION($o$,$d$);
11:         if $p.cost < p_{opt}.cost$ then
12:           $p_{opt} = p$;
13:         end if
14:       end for
15:     end for
16:   end if
17:   $R \Leftarrow$ Next step in $p_{opt}$;
18: end while
```

(A) Main loop that evaluates all plans containing the manipulation of an obstacle every time a new one is found

```
Algorithm 2 EVALUATE-ACTION($o$, $d$)
1:  $p_{o,d} \Leftarrow \emptyset$
2:  $c_1 = |A^*(R, o.init)|$;
3:  $o.position = o.init$;
4:  while push on $o$ in $d$ possible do
5:    $o.postion = o.postion + one\_push\_in\_d$;
6:    $c_2 = (o.postion - o.init)$;
7:    $c_3 = |A^*(o.position, R_{Goal})|$;
8:    $p = c_1 + c_2 + c_3$;
9:    $p.cost = c_1 * moveCost + c_2 * pushCost + c_3 * moveCost$;
10:   $P_{o,d} \Leftarrow P_{o,d} \cup \{p\}$;
11: end while
12: return $p \in P_{o,d}$ with min $p.cost$;
```

(B) Subroutine for evaluating all possible plans for each manipulation direction allowed on an obstacle

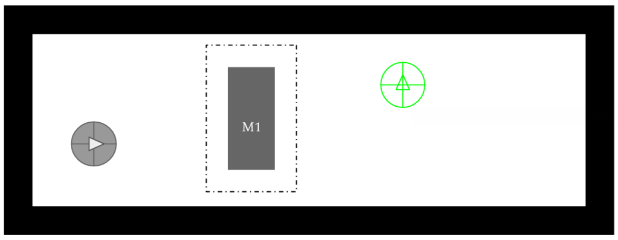
FIGURE 3.2: Baseline Algorithm as published by Wu et. al. in [3]

The optimized version of the algorithm published in the first article offers four optimization steps :

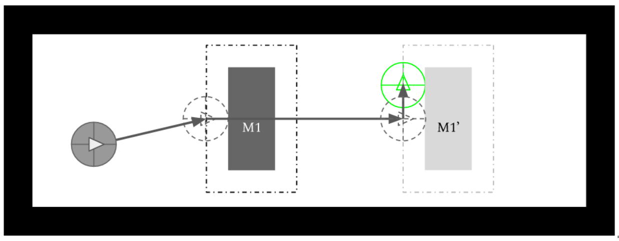
First Optimization Only consider computing a new plan if the current one is actually blocked by a new obstacle (Figure 3.5a, line 6). This keeps the local optimality, since a newly detected obstacle can only imply a costlier plan either moving around it or moving it (**assuming obstacles don't move by themselves, which could open new, better routes**): given that we can assume that our current optimal plan was optimal before the discovery of the new obstacle, we need only reconsider it if the new obstacle actually prevents it from coming to fruition.

Second Optimization Stop simulating pushes in a given direction before it becomes costlier than the current valid optimal plan, thanks to a bound (Figure 3.5b, line 5). This also keeps optimality, since there are no reasons to continue evaluating actions that are already costlier than simply following the current valid optimal plan.

Third Optimization Only compute the third path component (from obstacle to goal) with A* if the simulated movement actually creates an opening (Figure 3.5b, line 7), since checking an opening creation is less computing time-consuming than running a search algorithm to the goal. However, this step also causes the loss of local optimality, because it prevents the algorithm to fully evaluate some plans that could improve the cost. Below are examples figures of cases where this happens, assuming the cost of following a path is the same wheter it implies moving an obstacle or not. In Figure 3.3 and 3.4, no new opening is ever found, either because the blocking areas around the obstacle don't change (**??**) or because there were no blocking areas to begin with (**??**). Theses case are tentatively adressed in Chapter 4.

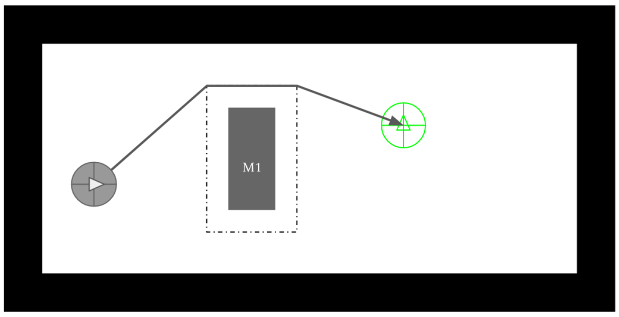


(A) Initial situation.

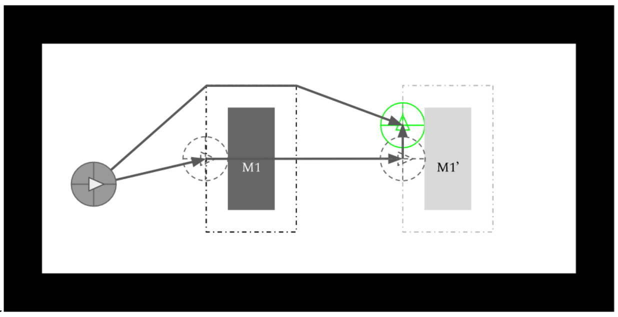


(B) Expected plan.

FIGURE 3.3: "Corridor" case where the original algorithm will not even find a plan when it should.



(A) Initial situation and suboptimal plan avoiding the obstacle that will be returned by the original algorithm.

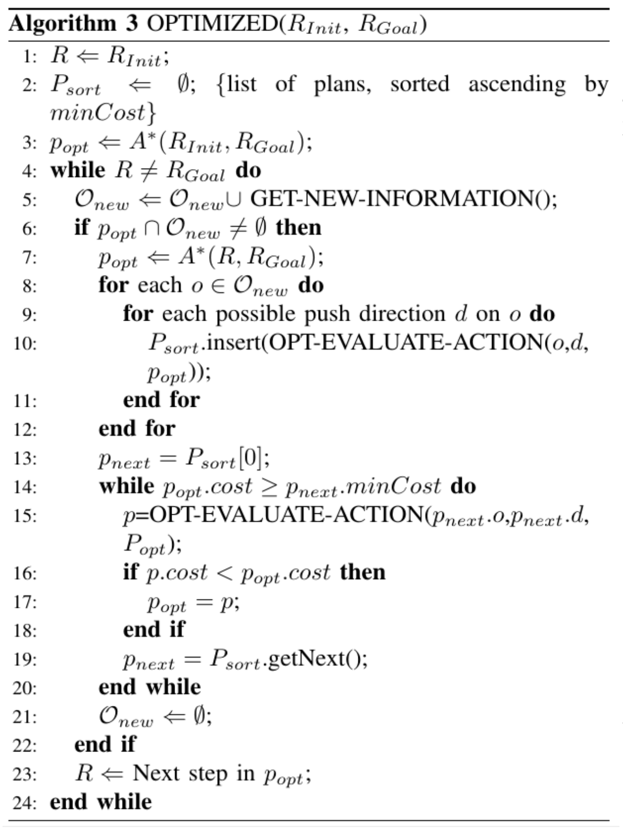


(B) Expected plan is the one that pushes the obstacle.

FIGURE 3.4: "Open space" case where the original algorithm will find only a suboptimal plan.

Fourth Optimization Finally, when the plan needs to be re-evaluated, new obstacles are evaluated first(Figure 3.5a, lines 8 to 12) and the resulting paths are saved into a list (Figure 3.5a, lines 2 and 10) by growing order of an underestimated heuristic cost that corresponds to sum of the costs of the plan components c_2 and c_3 (Figure 3.5b, line 12). Then, this list is used to iterate over all obstacle/push direction combinations, and re-evaluate the corresponding plan (Figure 3.5a, lines 13 to 20). The re-evaluation can be stopped as soon as the next pair to consider has a heuristic cost greater than the cost of the current optimal plan, improving execution performance (Figure 3.5b, line 14). However, this optimization step causes the loss of the guarantee of optimality. The heuristic cost depends on c_2 and c_3, and when moving an obstacle, these components' costs may get lower. As the algorithm never updates the heuristic cost (*minCost*) in the list according to this possibility, there is therefore no guarantee that the heuristic cost will always be an underestimate. In the words of Levihn, main author of the second article: *"Second, as the algorithm does not acknowledge the fact that free-space can be created during the execution (e.g. by moving objects), which can lower c2 or c3 for some objects, this optimization steps sacrifices local optimality."* [1]

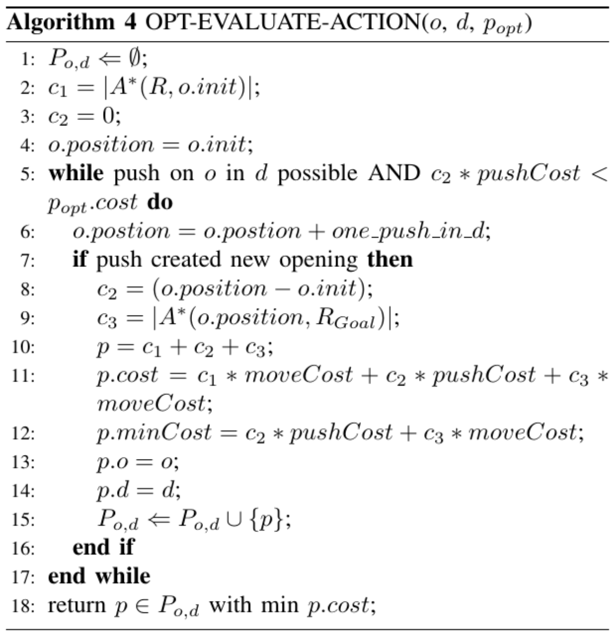
Note on the use of A* in the main loop In Figure 3.5a, lines 3 and 7, a call to the A* algorithm is made. For line 3, it determines the optimal path from the initial position to the goal, supposing no obstacle has been detected yet. In line 7, after the current optimal path has been invalidated by the detection of a collision between this path and a new obstacle, this call to A* star allows to get a new optimal path that avoids all obstacles that will serve as basis for the comparison with paths that consider moving obstacles.



Algorithm 3 OPTIMIZED(R_{Init}, R_{Goal})

```
1: $R \Leftarrow R_{Init}$;
2: $P_{sort} \Leftarrow \emptyset$; {list of plans, sorted ascending by $minCost$}
3: $p_{opt} \Leftarrow A^*(R_{Init}, R_{Goal})$;
4: while $R \neq R_{Goal}$ do
5:     $\mathcal{O}_{new} \Leftarrow \mathcal{O}_{new} \cup$ GET-NEW-INFORMATION();
6:     if $p_{opt} \cap \mathcal{O}_{new} \neq \emptyset$ then
7:         $p_{opt} \Leftarrow A^*(R, R_{Goal})$;
8:         for each $o \in \mathcal{O}_{new}$ do
9:             for each possible push direction $d$ on $o$ do
10:                $P_{sort}$.insert(OPT-EVALUATE-ACTION($o$,$d$, $p_{opt}$));
11:            end for
12:        end for
13:        $p_{next} = P_{sort}[0]$;
14:        while $p_{opt}.cost \geq p_{next}.minCost$ do
15:            $p$=OPT-EVALUATE-ACTION($p_{next}.o$,$p_{next}.d$, $P_{opt}$);
16:            if $p.cost < p_{opt}.cost$ then
17:                $p_{opt} = p$;
18:            end if
19:            $p_{next} = P_{sort}$.getNext();
20:        end while
21:        $\mathcal{O}_{new} \Leftarrow \emptyset$;
22:    end if
23:    $R \Leftarrow$ Next step in $p_{opt}$;
24: end while
```

(A) Main loop



Algorithm 4 OPT-EVALUATE-ACTION(o, d, p_{opt})

```
1: $P_{o,d} \Leftarrow \emptyset$;
2: $c_1 = |A^*(R, o.init)|$;
3: $c_2 = 0$;
4: $o.position = o.init$;
5: while push on $o$ in $d$ possible AND $c_2 * pushCost < p_{opt}.cost$ do
6:     $o.postion = o.postion + one\_push\_in\_d$;
7:     if push created new opening then
8:         $c_2 = (o.position - o.init)$;
9:         $c_3 = |A^*(o.position, R_{Goal})|$;
10:        $p = c_1 + c_2 + c_3$;
11:        $p.cost = c_1 * moveCost + c_2 * pushCost + c_3 * moveCost$;
12:        $p.minCost = c_2 * pushCost + c_3 * moveCost$;
13:        $p.o = o$;
14:        $p.d = d$;
15:        $P_{o,d} \Leftarrow P_{o,d} \cup \{p\}$;
16:    end if
17: end while
18: return $p \in P_{o,d}$ with min $p.cost$;
```

(B) Subroutine

FIGURE 3.5: Optimized Algorithm as published by Wu et. al. in [3]

3.2 Removing ambiguity

The original pseudocode presented above has quite a few typos, implicit or incongruous notations that create ambiguity (e.g., storing costs and paths in the same variable, having two different variable affectation operators ($\leftarrow$ and $=$), ...), confusing the reading. Since no pseudocode is provided by the authors for the second article's improvements propositions, it was necessary to first fix the pseudocode of the first article. The following paragraphs and pseudocode aim at fixing this.

A foreword on notations:

- Paths are ordered sets of "steps", which are themselves robot poses.
- Calling the A* or D*Lite algorithms returns A PATH. If no path is found, the returned set is empty: $\emptyset$.
- Plans are to be noted with a lowercase p. Lists or sets of plans will be noted with an uppercase P. A plan is a data structure with a "components" list attribute in which paths are stored in order of execution, and a "cost" attribute that represents the cost of executing the plan.
- Components of a plan are noted with a lowercase c.
- Assuming we suppose a cost in distance, the norm of a path $path$ (written $|path|$) corresponds to the sum of the euclidean distances between consecutive steps, and $+\infty$ if the path is empty.
- $moveCost$ and $pushCost$ are constants without dimension.
- The current robot pose is to be noted with an uppercase R, the initial pose with R_{init} and the goal pose with R_{goal}
- Obstacles are to be noted with a lowercase o. Lists or sets of obstacles will be noted with an uppercase O.

Note on update In the original paper, it is not explicit how the GET-NEW-INFORMATION method works. We therefore changed it to UPDATE-FROM-NEW-INFORMATION() method, that updates the world representation I given in parameter, with new information about obstacles that is collected in parallel in a different execution thread. By that we mean, if this new information includes modifications to known obstacles, they are updated, and if there are new obstacles, they are added to I. We assume that the attribute I.occGrid corresponds to the occupation grid with inflated obstacles in the current state, so that the path finding routine may run with it. In the same way, I.newObstacles corresponds to the list of newly observed obstacles since the last call of the same method.

Note on intersection detection The $p_{opt} \bigcap \mathcal{O}_{new} \neq \emptyset$ notation is used here as shorthand for checking that any of the new obstacles do not intersect with the robot's path. In implementation, this could be done for example by checking whether every pose in the path is not comprised in the inflated obstacles representation.

Note on re-evaluation The way the algorithm is written now, even new obstacles that have just been evaluated might be reevaluated, since they have been inserted into the P_{sort} list right before. Consequently, we should add a condition $p_{next}.o \notin \mathcal{O}_{new}$ around the line 16 of Algorithm **??** to further optimize the algorithm.

Note on getNext This is a helper method to traverse a list. It returns null when all elements have been traversed.

Note on stop condition The algorithm's stop condition is not explicit in none of the articles. If no path has been found even after considering all relevant obstacles, then the algorithm must return a success value.

Note on plan following In the article, the hypothesis is given that if the robot tries to move an obstacle but does not succeed, this obstacle will never be considered for manipulation again. It is meant to allow the robot to detect unmovable obstacles and to avoid an infinite loop caused by an endless evaluation of a same obstacle that cannot be moved. This hypothesis is translated in pseudocode by replacing the vague "$R \leftarrow$ Next step in p_{opt}" statement by checking whether the robot actually checking whether robot succeeded or not the desired movement by comparing the actual pose R_{real} after execution and the one we wished to reach R_{next} , and using the *blockedObsL* set to remember obstacles that should never be evaluated again.

Note on obstacle push pose For a given obstacle, the algorithm iterates over every push direction applicable to it, but doesn't iterate over every point from which it could apply said push direction. We must deduce that there is an **implicit hypothesis that for a given push direction, only one point around the obstacle is a valid manipulation start point**. Therefore, we will assume that *o.init* corresponds to the pose the robot must get into in order to move obstacle o in direction d. From the video that presents an implementation of the original algorithm, this pose :

- Is orientated in the given direction, toward the side of the obstacle that allows to push in the given direction,
- Is situated on precomputed "manipulation points" that are at a robot radius distance from the side; often it seems that this point is in front of the side's middle point,
- Among these "manipulation points" it seems the one closest to the current position of the robot is chosen. This is fine since the algorithm seems to operate under the **implicit hypotheses that the friction between the ground and the obstacle is negligible, and that the robot's width is always smaller than the length of the obstacle's side being pushed**. Thus, if the obstacle is movable and not blocked by surrounding obstacles, it will move in the direction it is being pushed in, whatever the accessible "manipulation point" on the appropriate side may be.

Note on $c_1 \neq \emptyset$ This condition is not in the original pseudocode, but is necessary to avoid the limit case where no path from the current robot pose to the obstacle is found, in which case, the manipulation plan cannot exist.

Note on bounding It is not said in the article how the "push on o in d possible" condition is verified, but we can assume that it is checked by verifying at each step that the obstacle's new occupied space doesn't intersect with any other obstacle. The $c_2 * pushCost$ bound here is not tight at all, which is fixed in the new optimization steps proposed in the second article. Still, it allows to cut down on unnecessary evaluations of extra pushes.

Note on elementary push It is not explicit in the article how an elementary *one_push_in_d* is computed, but we can assume it is the multiplication of a distance constant by the unit direction vector for the given direction d. We will make this more explicit in our own algorithm.

Note on opening detection Opening detection is not adressed in this paper and the method used is not explicit. A technical paper later written by co-authors Levihn and Stilman [2] however clears this ambiguity: *"The algorithm did not rely on search but simply observed the amount of adjacent free spaces on corners of the manipulated obstacle. While efficient, this algorithm is only applicable for world configurations populated with simple rectangular shaped static and movable obstacles. This is not realistic."*.

Note on c_2 As we only admit pushes in straight lines, and because the previous "push on o in d possible" condition means that there will be no collision on the manipulation path, c_2 simply is a path made from the pose to start moving the object and the pose where the robot leaves it.

Note on $c_3 \neq \emptyset$ This condition is not in the original pseudocode, but is necessary to avoid the limit case where no path from the obstacle to the goal is found, in which case, the manipulation plan cannot exist.

Algorithm 1 Optimized algorithm for NAMO in unknown environments of Wu et. al. (2010), fixed - MAIN LOOP

```
procedure OPTIMIZED($R_{init}$, $R_{goal}$)
    $R \leftarrow R_{init}$
    $blockedObsL \leftarrow \emptyset$
    $P_{sort} \leftarrow \emptyset$
    $\mathcal{O}_{new} \leftarrow \emptyset$
    $isManipSuccess \leftarrow True$
    $I \leftarrow empty\_occupation\_grid$
    $p_{opt}.components \leftarrow [A^*(R_{init}, R_{goal}, I.occGrid)]$
    $p_{opt}.cost \leftarrow |p_{opt}.components[0]| * moveCost$
    while $R \neq R_{goal}$ do
        UPDATE-FROM-NEW-INFORMATION($I$)                    ▷ Note on update
        $\mathcal{O}_{new} \leftarrow \mathcal{O}_{new} \bigcup I.newObstacles$
        $isPathFree \leftarrow p_{opt} \bigcap \mathcal{O}_{new} \neq \emptyset$      ▷ Note on intersection detection
        if not($isPathFree$ AND $isManipSuccess$) then
            $p_{opt}.components \leftarrow [A^*(R, R_{goal}, I.occGrid)]$
            $p_{opt}.cost \leftarrow |p_{opt}.components[0]| * moveCost$
            for each $o \in \mathcal{O}_{new}$ do
                for each possible push direction $d$ on $o$ do
                    $p \leftarrow$ OPT-EVALUATE-ACTION($o, d, p_{opt}, I, R, R_{goal}, blockedObsL$)
                    if $p \neq$ null then
                        $P_{sort}$.insert(p)
                    end if
                end for
            end for
            if $P_{sort} \neq \emptyset$ then
                $p_{next} \leftarrow P_{sort}[0]$
                while $p_{opt}.cost \geq p_{next}.minCost$ AND $p_{next} \neq$ null do    ▷ Note on re-evaluation
                    $p \leftarrow$ OPT-EVALUATE-ACTION($p_{next}.o$, $p_{next}.d$, $p_{opt}$, $I$, $R$, $R_{goal}$,
    $blockedObsL$)
                    if $p \neq$ null AND $p.cost \leq p_{opt}.cost$ then
                        $p_{opt} \leftarrow p$
                    end if
                    $p_{next} \leftarrow P_{sort}$.getNext()                ▷ Note on getNext
                end while
            end if
            $\mathcal{O}_{new} \leftarrow \emptyset$
        end if
        if $p_{opt}.components = \emptyset$ then
            return $False$                                  ▷ Note on stop condition
        end if
        $isManipSuccess \leftarrow True$
        $R_{next} \leftarrow p_{opt}$.getNextStep()                  ▷ Note on plan following
        $c_{next} \leftarrow p_{opt}$.getNextStepComponent()
        $R_{real} \leftarrow$ ROBOT-GOTO($R_{next}$)
        if $c_{next} = c_2$ AND $R_{real} \neq R_{next}$ then
            $isManipSuccess \leftarrow False$
            $blockedObsL \leftarrow blockedObsL \bigcup p_{opt}.o$
        end if
        $R \leftarrow R_{real}$
    end while
    return $True$
end procedure
```

Algorithm 2 Optimized algorithm for NAMO in unknown environments of Wu et. al. (2010), fixed - ACTION EVALUATION SUBROUTINE

```
procedure OPT-EVALUATE-ACTION($o$, $d$, $p_{opt}$, $I$, $R$, $R_{goal}$, $blockedObsL$)
    if $o \in blockedObsL$ then
        return null
    end if
    $P_{o,d} \leftarrow \emptyset$
    $c_1 \leftarrow A^*(R, o.init, I.occGrid)$                          ▷ Note on obstacle push pose
    if $c_1 = \emptyset$ then                                            ▷ Note on $c_1 \neq \emptyset$
        return null
    end if
    $c_2 \leftarrow \emptyset$
    $oSimPose \leftarrow o.init$
    while push on $o$ in $d$ possible AND $|c_2| * pushCost \leq p_{opt}.cost$ do    ▷ Note on bounding
        $oSimPose \leftarrow oSimPose + one\_push\_in\_d$              ▷ Note on elementary push
        if push created new opening then                                 ▷ Note on opening detection
            $c_2 \leftarrow \{o.init, oSimPose\}$                       ▷ Note on $c_2$
            $c_3 \leftarrow A^*(oSimPose, R_{goal}, I.occGrid)$
            if $c_3 \neq \emptyset$ then                                 ▷ Note on $c_3 \neq \emptyset$
                $p.components \leftarrow [c_1, c_2, c_3]$
                $p.cost \leftarrow (|c_1| + |c_3|) * moveCost + |c_2| * pushCost$
                $p.minCost \leftarrow |c_2| * pushCost + |c_3| * moveCost$
                $p.o, p.d \leftarrow o, d$
                $P_{o,d} \leftarrow P_{o,d} \bigcup \{p\}$
            end if
        end if
    end while
    return $p \in P_{o,d}$ with minimal $p.cost$ or null if $P_{o,d} = \emptyset$
end procedure
```

3.3 Pseudocode expression of Levihn's recommendations

In the second article [1], Levihn brings alternate solutions for the Second Optimization, Third Optimization and Fourth Optimization, reducing the computational effort and enlarging the scope of problems the algorithm can manage. For the Fourth Optimization, the changes make it so optimality is not affected by this optimization step.

Second Optimization In this article, the authors precise that they are using an improved opening detection algorithm, detailed in their separate technical paper [2]. Since contrary to the previous one, this new algorithm doesn't rely on obstacles being rectangles, but accepts any kind of polygon, it extends the capability of the overall algorithm to any convex polygon.

However, in the same way we proved that using opening detection for considering the computation of a full plan affected optimality before, since no measures are proposed in this new article to take this into account, we must assume that local optimality is not restored. In Chapter 4, we propose a measure for restoring optimality.

Third Optimization The bound that allows to reduce the number of unnecessary evaluations of extra pushes is tightened by adding to the current value of $|c_2|$ the cost of the first plan component

c_1 and an underestimate of the cost of the third plan component c_3. This underestimate is the euclidean distance between the last position of the simulated push pose $oSimPose$ and the goal pose R_{goal}. This bound is thus proved to be an underestimate of the real cost, keeping optimality.

Fourth Optimization Last but not least, a new heuristic is proposed alongside a modified version of the previous one. Basically, all obstacles that haven't been evaluated at least once are ordered in a separate list $euCostL$ (Algorithm **??**, lines 3, 14 to 16) by a heuristic cost that is independent from c_2 and c_3: the euclidean distance between the goal pose R_{goal} and the obstacle's nearest "manipulation point" (computation is done in Algorithm **??**, line 15) at which the robot could manipulate it. When an obstacle has been evaluated, it is added to another list $minCostL$ (Algorithm **??**, lines 3, 23 and 33) ordered by the usual $minCost$. Since this heuristic is more informed, $minCostL$ is used first when available. If not, $euCostL$ is used, the obstacle is re-evaluated and naturally added to the list ordered by $minCost$. This is achieved through the use of separate indexes for traversing the lists: i_e and i_m. This heuristic is invalidated anytime an obstacle has changed of place (Algorithm **??**, lines 8 to 10), potentially lowering c_2 or c_3. Thus, local optimality is no longer affected. For a more detailed explanation, please consult the following pseudocode or the original article [1].

Reordering the algorithm Since this is an interpretation, and for easier understanding, we took the liberty of cutting the "OPTIMIZED" algorithm (main loop) into two algorithms : the main loop where the plan is executed and knowledge about the environment updated (i.e. Algorithm 3: "MAKE-AND-EXECUTE-PLAN"), and the subroutine where the iteration over the obstacles is done to generate a new plan when necessary (i.e Algorithm 4: "MAKE-PLAN"). We also renamed the "OPT-EVALUATE-ACTION" subroutine (i.e Algorithm 2) into "PLAN-FOR-OBSTACLE" (i.e Algorithm 5). Note that the parameters p_{opt}, $euCostL$ and $minCostL$ are directly modified during the execution of the MAKE-PLAN() method, hence no return statement in it.

Note on D*Lite In the second article, it is mentionned twice that the path finding subroutine has been changed from A* to D*Lite. However, not even a hint of an explanation is given as to why this change, or what difference in the implementation it makes. Therefore, in our later final implementation, we shall stick with A*.

Note on update In addition to the previous note (3.2) We assume that I.freeSpaceCreated is True if any obstacle's occupied space has been reduced, False otherwise, and that I.allObstacles is the list of all observed obstacles in the current state.

Note on getting the list element and limit cases For the sake of readability in the pseudocode, if the list element that is asked for is out of bounds (empty list or reached end of list), the "[ ]" operator shall return a "fake" tuple with a null obstacle reference, and infinite cost: {null, $+\infty$}. This could easily be implented in code by either using a ternary operator (for example, "$minCostL[i_m].minCost$" would become "$minCostL[i_m] =$ null ? $+\infty$: $minCostL[i_m].minCost$") or implementing a custom array object with the wanted behaviour.

Traversal note Stop condition: if the next entry from $minCostL$ or $euCostL$ to be considered (the one with the lowest cost) is associated with a cost that is greater than the current optimal plan, it is not worth trying to evaluate any more options, and therefore the loop must end.

Note on priority If the current lowest cost entry is minCostL, evaluate the associated obstacle.

Note on postponing If i_e points to a lower cost than the one pointed by i_m, we only evaluate the associated object if $minCostL$ doesn't already contain an evaluation for it (because there is tighter bound for the cost of moving the object): the evaluation is thus postponed until the obstacle is reached through $minCostL$.

Note on manipulation points & c_1's computation In the article, the authors claim that *"c1 only needs to be calculated once for the entire process of evaluating the current object."*. With the same reasoning as in Note on obstacle push pose, this affirmation can only be true if the algorithm operates under the **implicit hypothesis that for all given manipulation directions, only one point around the obstacle is considered a valid manipulation start point**. From the video accompanying the article, this point seems to be the nearest point from the robot, situated at a radius distance from the middle of a side of the obstacle. However, this hypothesis actually hinders optimality: if there is in fact a valid manipulation point for each side of the obstacle, and the algorithm knowingly doesn't consider them because they are further from the current robot's position, it will ignore the fact that a same manipulation direction could end up in opening a better path if the obstacle were moved from another manipulation point (see figures below). To guarantee optimality, we would have to simulate the manipulation in the given direction for every reachable manipulation point, thus re-evaluating c_1 for each. That would result in adding an extra "for" loop englobing the existing one. This is illustrated by the figures below.

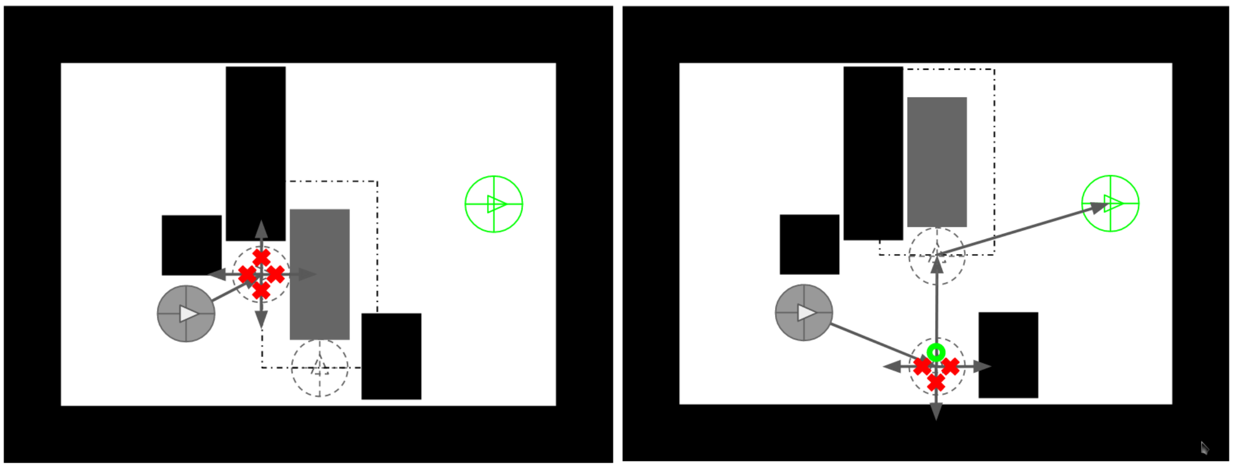

(A) Nearest manipulation pose doesn't allow moving the obstacle at all. (B) But if we also consider the other pose, a valid plan can be found.

FIGURE 3.6: Illustration of the importance of considering all possibles manipulations for all manipulation poses and not just considering the nearest one.

Note on BA The BA variable in the OPT-EVALUATE-ACTION() function allows to remember the initial blocking areas when using the new algorithm for more efficient opening detection. On the first called, the variable is initialized with the initial blocking areas, and at each following call, it is passed as a parameter to reduce computational overhead. This measure is recommended by the technical paper.

Note on allowed manipulations In the second article, the authors assert that they don't limit manipulation to pushes. However, it is clear, especially from the video, that manipulations are restricted to translations in a single direction.

Note on *seq* Seq corresponds to the number of unit translations that have been simulated.

Algorithm 3 Optimized algorithm for NAMO in unknown environments of Wu et. al. adapted according to M.Levihn et. al.'s (2014) recommandations - EXECUTION LOOP

```
1: procedure MAKE-AND-EXECUTE-PLAN($R_{init}$, $R_{goal}$)
2:     $R \leftarrow R_{init}$
3:     $\mathcal{O}_{new} \leftarrow \emptyset$
4:     $isManipSuccess \leftarrow True$
5:     $blockedObsL \leftarrow \emptyset$
6:     $euCostL, minCostL \leftarrow \emptyset, \emptyset$
7:     $I \leftarrow empty\_occupation\_grid$
8:     $p_{opt}.components \leftarrow [D$*Lite($R_{init}$, $R_{goal}$, $I.occGrid$)]          ▷ Note on D*Lite
9:     $p_{opt}.cost \leftarrow |p_{opt}| * moveCost$
10:    while $R \neq R_{goal}$ do
11:        UPDATE-FROM-NEW-INFORMATION($I$)          ▷ Note on update
12:        if $I$.freeSpaceCreated() then
13:            $minCostL \leftarrow \emptyset$
14:        end if
15:        $\mathcal{O}_{new} \leftarrow \mathcal{O}_{new} \bigcup I.newObstacles$
16:        $\mathcal{O} \leftarrow I.allObstacles$
17:        $isPathFree \leftarrow p_{opt} \bigcap \mathcal{O}_{new} \neq \emptyset$
18:        if not($isPathFree$ AND $isManipSuccess$) then
19:            $p_{opt}.components \leftarrow [D$*Lite($R$, $R_{goal}$, $I.occGrid$)]
20:            $p_{opt}.cost \leftarrow |p_{opt}| * moveCost$
21:            MAKE-PLAN($R$, $R_{goal}$, $I$, $\mathcal{O}$, $blockedObsL$, $p_{opt}$, $euCostL$, $minCostL$)
22:            $\mathcal{O}_{new} \leftarrow \emptyset$
23:        end if
24:        if $p_{opt}.components = \emptyset$ then
25:            return $False$
26:        end if
27:        $isManipSuccess \leftarrow True$
28:        $R_{next} \leftarrow p_{opt}$.getNextStep()
29:        $c_{next} \leftarrow p_{opt}$.getNextStepComponent()
30:        $R_{real} \leftarrow$ ROBOT-GOTO($R_{next}$)
31:        if $c_{next} = c_2$ AND $R_{real} \neq R_{next}$ then
32:            $isManipSuccess \leftarrow False$
33:            $blockedObsL \leftarrow blockedObsL \bigcup p_{opt}.o$
34:        end if
35:        $R \leftarrow R_{real}$
36:    end while
37:    return $True$
38: end procedure
```

Algorithm 4 Optimized algorithm for NAMO in unknown environments of Wu et. al. adapted according to M.Levihn et. al.'s (2014) recommandations - PLAN COMPUTATION

```
procedure MAKE-PLAN($R$, $R_{goal}$, $I$, $\mathcal{O}$, $blockedObsL$, $p_{opt}$, $euCostL$, $minCostL$)
    for each $o \in \mathcal{O}$ do
        $C_{3_{(Est)}} \leftarrow \min(\{\forall graspPoint \in o.graspPoints \ || \ |\{graspPoint, R_{goal}\}|\})$
        $euCostL$.insertOrUpdate($\{o, C_{3_{(Est)}}\}$)
    end for
    $i_e, i_m \leftarrow 0, 0$                                        ▷ Note on getting the list element and limit cases
    while min($minCostL[i_m].minCost, euCostL[i_e].c_{3_{est}}$) $< p_{opt}.cost$ do          ▷ Traversal note
        if $minCostL[i_m].minCost < euCostL[i_e].c_{3_{est}}$ then                   ▷ Note on priority
            $p \leftarrow$ PLAN-FOR-OBSTACLE($minCostL[i_m].obstacle$, $p_{opt}$, $I$, $R$, $R_{goal}$, $blockedObsL$)
            if $p \neq$ null then
                $minCostL$.insertOrUpdate($\{minCostL[i_m].obstacle, p.minCost\}$)
                $i_m \leftarrow i_m + 1$
                if $p.cost < p_{opt}.cost$ then
                    $p_{opt} \leftarrow p$
                end if
            else
                $minCostL$.insertOrUpdate($\{minCostL[i_m].obstacle, +\infty\}$)
                $i_m \leftarrow i_m + 1$
            end if
        else
            if not $minCostL$.contains($euCostL[i_e].obstacle$) then            ▷ Note on postponing
                $p \leftarrow$ PLAN-FOR-OBSTACLE($euCostL[i_e].obstacle$, $p_{opt}$, $I$, $R$, $R_{goal}$, $blockedObsL$)
                if $p \neq$ null then
                    $minCostL$.insertOrUpdate($\{euCostL[i_e].obstacle, p.minCost\}$)
                    $i_m \leftarrow i_m + 1$
                    if $p.cost < p_{opt}.cost$ then
                        $p_{opt} \leftarrow p$
                    end if
                else                                   ▷ Corresponds to the "If $p \neq$ null" statement.
                    $minCostL$.insertOrUpdate($\{euCostL[i_e].obstacle, +\infty\}$)
                    $i_m \leftarrow i_m + 1$
                end if
            end if
            $i_e \leftarrow i_e + 1$
        end if
    end while
end procedure
```

Algorithm 5 Optimized algorithm for NAMO in unknown environments of Wu et. al. adapted according to M.Levihn et. al.'s (2014) recommandations - PLAN EVALUATION FOR A SINGLE OBSTACLE

```
procedure PLAN-FOR-OBSTACLE($o$, $p_{opt}$, $I$, $R$, $R_{goal}$, $blockedObsL$)
    if $o \in blockedObsL$ then
        return null
    end if
    $P_{o,d} \leftarrow \emptyset$
    $c_1 \leftarrow D$*Lite$(R, o.init, I.occGrid)$        ▷ Note on manipulation points & $c_1$'s computation
    if $c_1 = \emptyset$ then
        return null
    end if
    $BA \leftarrow null$                                    ▷ Note on BA
    for each possible manipulation direction $d$ on $o$ do    ▷ Note on allowed manipulations
        $seq \leftarrow 1$                                  ▷ Note on $seq$
        $oSimPose \leftarrow o.init + one\_translation\_in\_d$
        $c_{3_{(Est)}} \leftarrow \{oSimPose, R_{goal}\}$
        $C_{est} \leftarrow (|c_1| + |c_{3_{(Est)}}|) * moveCost + seq * one\_translation\_in\_d * pushCost$
        while $C_{est} \leq p_{opt}.cost$ AND manipulation on $o$ possible do
            if CHECK-NEW-OPENING($I.occGrid$, $o$, $seq * one\_translation\_in\_d$, $BA$) then
                $c_2 \leftarrow \{o.init, oSimPose\}$
                $c_3 \leftarrow D$*Lite($oSimPose$, $R_{goal}$, $I.occGrid$)
                if $c_3 \neq \emptyset$ then
                    $p.components \leftarrow [c_1, c_2, c_3]$
                    $p.cost \leftarrow (|c_1| + |c_3|) * moveCost + |c_2| * pushCost$
                    $p.minCost \leftarrow |c_2| * pushCost + |c_3| * moveCost$
                    $p.o, p.d \leftarrow o, d$
                    $P_{o,d} \leftarrow P_{o,d} \bigcup \{p\}$
                end if
            end if
            $seq \leftarrow seq + 1$
            $oSimPose \leftarrow oSimPose + one\_translation\_in\_d$
            $c_{3_{(Est)}} \leftarrow \{oSimPose, R_{goal}\}$
            $C_{est} \leftarrow (|c_1| + |c_{3_{(Est)}}|) * moveCost + seq * one\_translation\_in\_d * pushCost$
        end while
    end for
    return $p \in P_{o,d}$ with minimal $p.cost$ or null if $P_{o,d} = \emptyset$
end procedure
```

Chapter 4

Extension of Wu et. al.'s algorithm toward dynamic and social navigation

4.1 Discussion on the original hypotheses in the light of tests with the Pepper robot

As eventually our experimental platform is to be a Pepper robot, in the context of the Robocup@Home challenge that emulates a home setting, several hypotheses from the original algorithms have to be reconsidered :

- **Initial knowledge of the environment** is partial, in that all static obstacles (i.e. objects that are not meant to be moved by any actor, like walls or very heavy furniture) have already been mapped. In the context of the Robocup@Home Challenge, participants are allowed to build such a map prior to the actual trials. This hypothesis is actually quite justified since in a home setting, it is very likely that the robot has undergone a configuration phase prior to its daily use, when it is provided with a manually drawn map of the home, or at least allowed to roam about and map the static obstacles. Having a map of static obstacles is very important for standard localization algorithms used in ROS, like AMCL, since they use this environment knowledge to compensate for odometry error.

- **Manipulation actions**, for the moment, are to be limited to pushes in a perpendicular direction to the obstacle's side being pushed. Given the many problematics related to grasping objects (e.g., appropriate positioning of the robot joints, keeping the robot balanced, ...), it is best for a first iteration not to dwell on these.

- **Manipulation poses** are a key concept of manipulating obstacles, as we have shown in the previous chapter, and in contrary to the original algorithms we will explicitly explain our hypotheses as to them. Experimentations with the Pepper Robot and carboard boxes as movable obstacles have shown that a good first approximation that guarantees quasi-systematic push manipulation successes are poses situated at the middle of the object's sides. This is, of course, supposing that we are only considering light objects with negligible friction against the ground, and with no other cinematic constraint than a plan-plan link between one of the obstacle's faces and the ground (a perfect plane).

- **Manipulation cost** A constant *pushCost* has been used in the previously shown algorithm to allow weighting of the manipulation action in regard to a simple move action. Semantically, it makes more sense that this constant be related to the object (the difficulty of moving a specific object depending mainly of its physical properties), so we will store it as an obstacle attribute.

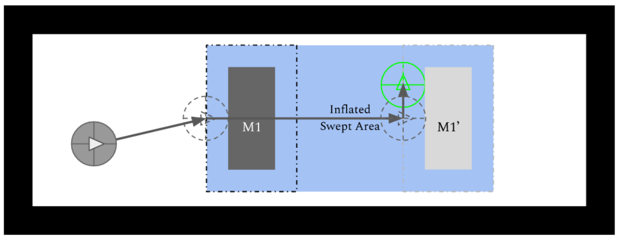
- **Manipulation possibility check** Checking whether a manipulation is possible or not is the same as checking whether the area covered by the robot and the obstacle as they move together is not in intersection with any other obstacle. As we limit our action set to pushes in a specific direction, this area can be defined of the convex hull containing both the robot's and the obstacle's polygonal representation at their initial and final position. According to the existing litterature, we will call this the "safe-swept area" if no other obstacle is in intersection with it. In the following pseudocode, this is done by the "GET-SAFE-SWEPT-AREA" method, which returns null if any obstacle is in intersection with the manipulation area. This area is saved as part of the plan so that when the plan is being executed, checking for a collision is as simple as checking if an obstacle appeared in this area.

- **Obstacle discovery** As the robot approaches obstacles, their geometrical representation is updated according to what the robot's sensors can see. When executing a plan that includes the manipulation of an obstacle, said obstacle can actually change during the execution of the c_1 component, which is problematic for the preservation of optimality, since the obstacle's push poses may change (it is defined with a dependency to the side's middle point). Therefore re-evaluation should not only be triggered if a new obstacle intersects with the current optimal plan, but also if the current optimal plan includes the manipulation of an obstacle and if said obstacle has changed in a way that makes the originally targeted *pushPose* unavailable.

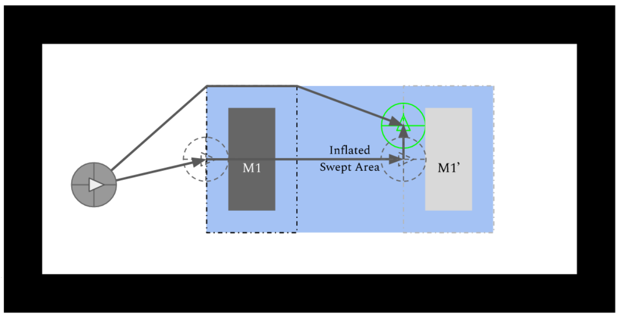
Below, we propose, a way for restoring the optimality, assuming we are under the hypothesis of sole translations in a single direction:

A new opening detection is defined by the disparition of at least one blocking area thanks to the considered manipulation. A new opening is never detected if :

- Not a single blocking area disappears thanks to the considered manipulation, because the blocking areas do not vary enough or at all ("corridor" case),
- There are no blocking areas to begin with ("open space" case).



(A) "Corridor" case



(B) "Open space" case

FIGURE 4.1: Limit cases of the original algorithm with illustration of the "Inflated swept area"

In both cases, it is only interesting to consider the manipulation if it actually creates any chance of finding a path that has a lower cost than the one that avoids the obstacle. If no new opening is detected, and if, like in the "corridor" case, no path avoiding the obstacle was found, or, like in the "open space" case, a path avoiding the obstacle was found, we should only consider the manipulation if it allows us to push the obstacle through the goal pose; in more precise terms, **if the goal pose is within the "inflated swept area" and the obstacle in its final position does not intersect with the goal pose**. The inflated swept area is defined as the area covered by the inflated (by the robot's radius) obstacle when moved. In the end, the overall check condition should be :

If CHECK-NEW-OPENING($I.occGrid$, o, $translation$, BA) AND $goalPose \in$ GET-INFLATED-SWEPT-AREA(o, $translation$, I) AND $goalPose \notin o.inflatedArea$

We have an intuition that these two extra verifications steps are sufficient allow to restore optimality, but this is no proper demonstration. Since performance is not the main focus of our work here, but optimality is, we will prefer not to use the opening check optimization step in our following algorithms propositions, and postpone a proof to later work.

Note on continue The **continue** statement returns the control to the beginning of the loop, and simply won't execute any of the remaining statements in the current iteration of the loop. This is done because a plan with a manipulation cannot exist without an empty c_1 component.

Note on COPY Here, $p_{opt}.o$ is a copy of object o, and not the same object, so that when o is updated because of the call to UPDATE-FROM-NEW-INFORMATION() on I, we can compare the difference between the two. We do the same for $p_{opt}.pushPose$ for the same reason. This allows us to trigger re-evaluation if the obstacle's push poses change and the one the robot aimed for no longer exists.

Note on [] and $\neq$ Here, the [] operator is used as a short handle for "get the obstacle that corresponds in $\mathcal{O}$ that corresponds to the saved obstacle $p_{opt}.o$. The $\neq$ operator checks if the two states of the obstacle are the same or not (i.e, if the obstacle changed).

Note on the use of $\bigcap$ The notation $\bigcap$ means here that we check for possible collisions between the swept area and any obstacle, since they may have changed.

Note on saving *translation* in p_{opt} The $translation$ necessary for manipulating the obstacle is saved to easily recompute the safe swept area when the obstacle changes.

Algorithm 6 Execution loop taking our hypotheses into account.

```
procedure MAKE-AND-EXECUTE-PLAN($R_{init}$, $R_{goal}$, $I_{init}$)
    ...                                              ▷ Initialization (lines 2 to 6 in Algorithm 3)
    $I \leftarrow I_{init}$
    $p_{opt}.components \leftarrow [A^*(R_{init}, R_{goal}, I.occGrid)]$
    $p_{opt}.cost \leftarrow |p_{opt}| * moveCost$
    while $R \neq R_{goal}$ do
        UPDATE-FROM-NEW-INFORMATION($I$)
        if $I$.freeSpaceCreated then
            $minCostL \leftarrow \emptyset$
        end if
        $\mathcal{O}_{new} \leftarrow \mathcal{O}_{new} \bigcup I.newObstacles$
        $\mathcal{O} \leftarrow I.allObstacles$
        $isPathFree \leftarrow p_{opt} \bigcap \mathcal{O}_{new} \neq \emptyset$
        $isPushPoseValid \leftarrow True$
        $isManipSafe \leftarrow True$
        $isObstacleSame \leftarrow True$
        if $p_{opt}.o$ exists then                   ▷ If $p_{opt}$ includes the manipulation of an obstacle.
            $isObstacleSame \leftarrow \mathcal{O}[p_{opt}.o] = p_{opt}.o$          ▷ Note on [] and $\neq$
            if not $isObstacleSame$ then
                $p_{opt}.o \leftarrow \mathcal{O}[p_{opt}.o.id]$                  ▷ Update the copy.
                $p_{opt}.safeSweptArea \leftarrow$ GET-SAFE-SWEPT-AREA($p_{opt}.o$, $p_{opt}.translation$, $I$)
                if $p_{opt}.pushPose \notin p_{opt}.o.pushPoses$ then
                    $isPushPoseValid \leftarrow False$
                end if
            end if
            if $p_{opt}.safeSweptArea \bigcap \mathcal{O} \neq \emptyset$ then            ▷ Note on the use of $\bigcap$
                $isManipSafe \leftarrow False$
            end if
        end if
        if not($isPathFree$ AND $isManipSuccess$ AND $isManipSafe$ AND $isPushPoseValid$)
    then
            $p_{opt}.components \leftarrow [A^*(R, R_{goal}, I.occGrid)]$
            $p_{opt}.cost \leftarrow |p_{opt}| * moveCost$
            MAKE-PLAN($R$, $R_{goal}$, $I$, $\mathcal{O}$, $blockedObsL$, $p_{opt}$, $euCostL$, $minCostL$)
            $\mathcal{O}_{new} \leftarrow \emptyset$
        end if
        ...                                          ▷ Execution (lines 24 to 35 in Algorithm 3)
    end while
    return $True$
end procedure
```

Algorithm 7 Obstacle evaluation subroutine taking our hypotheses into account.

```
1: procedure PLAN-FOR-OBSTACLE($o$, $p_{opt}$, $I$, $R$, $R_{goal}$, $blockedObsL$)
2:     if $o \in blockedObsL$ then
3:         return null
4:     end if
5:     $P_{o,d} \leftarrow \emptyset$
6:     for each $pushPose$ in $o.pushPoses$ do
7:         $pushUnit \leftarrow (cos(pushPose.yaw), sin(pushPose.yaw))$    ▷ Unit vector for push direction
8:         $c_1 \leftarrow A^*(R, pushPose, I.occGrid)$    ▷ $c_1$ is computed for each push pose
9:         if $c_1 = \emptyset$ then
10:            continue    ▷ Note on continue
11:        end if
12:        $seq \leftarrow 1$
13:        $translation \leftarrow pushUnit * onePushDist * seq$    ▷ onePushDist is a distance constant
14:        $safeSweptArea \leftarrow$ GET-SAFE-SWEPT-AREA($o$, $translation$, $I$)
15:        $oSimPose \leftarrow pushPose + translation$
16:        $c_{3_{(Est)}} \leftarrow \{oSimPose, R_{goal}\}$
17:        $C_{est} \leftarrow (|c_1| + |c_{3_{(Est)}}|) * moveCost + |translation| * o.pushCost$
18:        while $C_{est} \leq p_{opt}.cost$ AND $safeSweptArea \neq$ null do
19:            $c_2 \leftarrow \{pushPose, oSimPose\}$
20:            $c_3 \leftarrow A^*(oSimPose, R_{goal}, I.occGrid)$
21:            if $c_3 \neq \emptyset$ then
22:                $p.components \leftarrow [c_1, c_2, c_3]$
23:                $p.cost \leftarrow (|c_1| + |c_3|) * moveCost + |c_2| * o.pushCost$
24:                $p.minCost \leftarrow |c_2| * o.pushCost + |c_3| * moveCost$
25:                $p.o \leftarrow$ COPY($o$)    ▷ Note on COPY
26:                $p.translation \leftarrow translation$    ▷ Note on saving $translation$ in $p_{opt}$
27:                $p.safeSweptArea \leftarrow safeSweptArea$
28:                $P_{o,d} \leftarrow P_{o,d} \bigcup \{p\}$
29:            end if
30:            $seq \leftarrow seq + 1$
31:            $translation \leftarrow pushUnit * onePushDist * seq$
32:            $safeSweptArea \leftarrow$ GET-SAFE-SWEPT-AREA($o$, $translation$, $I$)
33:            $oSimPose \leftarrow pushPose + translation$
34:            $c_{3_{(Est)}} \leftarrow \{oSimPose, R_{goal}\}$
35:            $C_{est} \leftarrow (|c_1| + |c_{3_{(Est)}}|) * moveCost + |translation| * o.pushCost$
36:        end while
37:    end for
38:    return $p \in P_{o,d}$ with minimal $p.cost$ or null if $P_{o,d} = \emptyset$
39: end procedure
```

4.2 Social awareness through risk consideration

As shown in Chapter 2, to the best of our knowledge, the current NAMO litterature has never covered the idea of socially-aware navigation. Then, we must ask: what makes the action of moving an obstacle socially-aware or not ?

The first thing that comes to mind would be to consider that some objects are better not be moved because:

- they are too fragile (e.g. flower pot),
- they have a high value in the humans eye (e.g. a costly vase)
- they might cause the robot to break if it fails to move them properly (i.e. heavy or unstable objects)
- they are not supposed to be moved (i.e. exhibited objects)

Thus comes the notion of risk, either to the robot or to the manipulated objects. To mitigate this risk, we propose to modify our base algorithm so that an obstacle is not to be moved unless identified as belonging to a provided whitelist of "movable" obstacles.

This identification of the obstacle's nature is to be done through computer vision, since it is one of the most efficient and most common ways to detect specific objects, by using trained neural networks for example. However, robots come with all sort of sensors to detect obstacles: laser range finders, RGB(D) cameras, sonars, ... And often, as with Pepper, their fields of vision do not perfectly overlap: typically, an obstacle may be detected by the laser range finders or the sonars, but not be within the field of vision of the RGB(D) camera, because it is in its blind spot or simply too close or too far away. This creates a situation where the robot knows an obstacle is there, but cannot definitely categorize it as "movable" or "unmovable" since it is not in the camera's field of vision.

Then, it means that the algorithm must be adapted not only to manage the fact that an object should be considered for manipulation if and only if it is deemed "movable", but also to eventually adapt the robot's trajectory **in an optimal way** so that an "unidentified" / "potentially movable" object can be identified with certainty before engaging with the manipulation procedure.

For that, when we evaluate an obstacle, we first check whether the obstacle has already been identified or not. If it has been identified as "movable", the algorithm does not change. If it has been identified as "unmovable", the obstacle evaluation routine simply stops before actually evaluating. And finally, if the evaluated obstacle is "unidentified":

- The c_1 plan component that goes from the current robot pose to the push pose is evaluated,
- If a pose comprised in the computed c_1 component allows the camera field of vision (which is condidered to be cutoff cone) to encompass the obstacle's currently known geometry, keep the precomputed c_1 component,
- Else we must find a path shortest path component c_0 from the current robot pose to an "observation pose" where we know we can identify the obstacle as "movable" or not with certainty and recompute c_1 as the path from this "observation pose" to the push pose.
- The observation poses are updated in the same way that push poses are: automatically, whenever an obstacle is updated. These poses are situated at every grid point for which the field of vision of the robot sensor(s) dedicated to obstacle recognition covers the entire known obstacle's geometry. Though the presented algorithm is not affected by the representation of the identification sensor's field of vision, in our experimentation, we will consider a single RGB(D) camera, and approximate its field of vision by the difference between a circular sector and a disk of same center, coincident with the robot's center, which is an acceptable representation for the Pepper robot. The circular sector has a radius r_{max},

central angle θ and is equally partitioned around the robot's orientation direction line. The disk has a radius r_{min}.

- To find c_0 and c_1, a heuristic cost defined as the sum of the euclidian distance between the current pose and observation pose, and euclidean distance between observation pose and currently evaluated push pose is computed for every observation pose, and allows to order them in a list $obsPoseL$, sorted by ascending heuristic cost. The list is then traversed until the heuristic cost of the current element is greater than the current optimal cost of $c_0 + c_1$.

Algorithm 8 Execution loop modified for allowing observation.

```
1: procedure MAKE-AND-EXECUTE-PLAN($R_{init}$, $R_{goal}$, $I_{init}$)
2:     ...                                  ▷ Initialization (lines 2 to 5 in Algorithm 6)
3:     $isObservable \leftarrow True$
4:     while $R \neq R_{goal}$ do
5:         UPDATE-FROM-NEW-INFORMATION($I$)
6:         ...                  ▷ Knowledge update and checks (lines 7 to 29 in Algorithm 6)
7:         if not($isPathFree$ AND $isManipSuccess$ AND $isManipSafe$ AND $isPushPoseValid$
   AND $isObservable$) then
8:             $p_{opt}.components \leftarrow [A^*(R, R_{goal}, I.occGrid)]$
9:             $p_{opt}.cost \leftarrow |p_{opt}| * moveCost$
10:            MAKE-PLAN($R$, $R_{goal}$, $I$, $\mathcal{O}$, $blockedObsL$, $p_{opt}$, $euCostL$, $minCostL$)
11:            $\mathcal{O}_{new} \leftarrow \emptyset$
12:        end if
13:        if $p_{opt}.components = \emptyset$ then
14:            return $False$
15:        end if
16:        $isManipSuccess \leftarrow True$
17:        $isObservable \leftarrow True$
18:        $R_{next} \leftarrow p_{opt}$.getNextStep()
19:        $c_{next} \leftarrow p_{opt}$.getNextStepComponent()
20:        if $p_{opt}.o.movableStatus = IS\_MAYBE\_MOVABLE$ AND not $isObstacleSame$ AND
   ($c_{next} = o_1$ OR $c_{next} = c_1$) then
21:            $fObsPose \leftarrow$ GET-FIRST-PATH-OBSPOSE($p_{opt}.o$, $p_{opt}$.get-$o_1$() + $p_{opt}$.get-$c_1$(), $I$)
22:            if $fObsPose$ = null then
23:                $isObservable \leftarrow False$
24:                continue
25:            end if
26:        end if
27:        $R_{real} \leftarrow$ ROBOT-GOTO($R_{next}$)
28:        if $c_{next} = c_2$ AND $R_{real} \neq R_{next}$ then
29:            $isManipSuccess \leftarrow False$
30:            $blockedObsL \leftarrow blockedObsL \bigcup p_{opt}.o$
31:        end if
32:        $R \leftarrow R_{real}$
33:    end while
34:    return $True$
35: end procedure
```

Algorithm 9 Obstacle evaluation subroutine modified for allowing observation.

```
procedure PLAN-FOR-OBSTACLE($o$, $p_{opt}$, $I$, $R$, $R_{goal}$, $blockedObsL$)
    if $o \in blockedObsL$ OR $o.movableStatus = IS\_NOT\_MOVABLE$ then
        return null
    end if
    $P_{o,d} \leftarrow \emptyset$
    for each $pushPose$ in $o.pushPoses$ do
        $pushUnit \leftarrow (cos(pushPose.yaw), sin(pushPose.yaw))$
        $c_1 \leftarrow A^*(R, pushPose, I.occGrid)$
        if $c_1 = \emptyset$ then
            continue
        end if
        $o_1 \leftarrow \emptyset$
        if $o.movableStatus = IS\_MAYBE\_MOVABLE$ then
            $obsPose \leftarrow$ GET-FIRST-PATH-OBSPOSE($o$, $c_1$, $I$)
            if $obsPose \neq$ null then
                $o_1, c_1 \leftarrow c_1[c_1.firstPose : obsPose], c_1[obsPose : c_1.lastPose]$        ▷
            else
                COMPUTE-O1-C1($o$, $I$, $R$, $pushPose$, $o_1$, $c_1$)
                if $o_1 = \emptyset$ OR $c_1 = \emptyset$ then
                    continue
                end if
            end if
        end if
        $seq \leftarrow 1$
        $translation \leftarrow pushUnit * onePushDist * seq$
        $safeSweptArea \leftarrow$GET-SAFE-SWEPT-AREA($o$, $translation$, $I$)
        $oSimPose \leftarrow pushPose + translation$
        $c_{3_{(Est)}} \leftarrow \{oSimPose, R_{goal}\}$
        $C_{est} \leftarrow ((o_1 \neq \emptyset ? |o_1| : 0) + |c_1| + |c_{3_{(Est)}}|) * moveCost + |translation| * o.pushCost$
        while $C_{est} \leq p_{opt}.cost$ AND $safeSweptArea \neq$ null do
            $c_2 \leftarrow \{pushPose, oSimPose\}$
            $c_3 \leftarrow A^*(oSimPose, R_{goal}, I.occGrid)$
            if $c_3 \neq \emptyset$ then
                $p.components \leftarrow o_1 \neq \emptyset$ ? $[o_1, c_1, c_2, c_3]$ : $[c_1, c_2, c_3]$
                $p.cost \leftarrow ((o_1 \neq \emptyset ? |o_1| : 0) + |c_1| + |c_3|) * moveCost + |c_2| * o.pushCost$
                $p.minCost \leftarrow |c_2| * o.pushCost + |c_3| * moveCost$
                $p.o \leftarrow$ COPY($o$)
                $p.translation \leftarrow translation$
                $p.safeSweptArea \leftarrow safeSweptArea$
                $P_{o,d} \leftarrow P_{o,d} \bigcup \{p\}$
            end if
            $seq \leftarrow seq + 1$
            $translation \leftarrow pushUnit * onePushDist * seq$
            $safeSweptArea \leftarrow$GET-SAFE-SWEPT-AREA($o$, $translation$, $I$)
            $oSimPose \leftarrow pushPose + translation$
            $c_{3_{(Est)}} \leftarrow \{oSimPose, R_{goal}\}$
            $C_{est} \leftarrow ((o_1 \neq \emptyset ? |o_1| : 0) + |c_1| + |c_{3_{(Est)}}|) * moveCost + |translation| * o.pushCost$
        end while
    end for
    return $p \in P_{o,d}$ with minimal $p.cost$ or null if $P_{o,d} = \emptyset$
end procedure
```

Algorithm 10 Subroutine for checking if *path* allows to identify the obstacle.

```
procedure GET-FIRST-PATH-OBSPOSE($o$, $path$, $I$)
    for each $pose$ in $path$ do
        if IS-OBS-IN-FOV-FOR-POSE($o$, $pose$, $I$) then
            return $pose$
        end if
    end for
    return null
end procedure
```

Algorithm 11 Subroutine for computing o_1 and c_1 if c_1 is not already valid.

```
procedure COMPUTE-O1-C1($o$, $I$, $R$, $pushPose$, $o_1$, $c_1$)
    $c_1 \leftarrow \emptyset$
    $totalCost \leftarrow +\infty$
    for each $obsPose$ in $o.obsPoses$ do
        $o \leftarrow A^*(R, obsPose, I.occGrid)$
        $c \leftarrow A^*(obsPose, pushPose, I.occGrid)$
        $newTotalCost = |o| + |c|$
        if $newTotalCost < +\infty$ AND ($newTotalCost < totalCost$ OR ($newTotalCost = totalCost$ AND $|o| < |o_1|$)) then
            $o_1 = o$
            $c_1 = c$
            $totalCost = |o_1| + |c_1|$
        end if
    end for
end procedure
```

Algorithm 12 Optimized subroutine for computing o_1 and c_1 if c_1 is not already valid.

```
1: procedure OPT-COMPUTE-O1-C1($o$, $I$, $R$, $pushPose$, $o_1$, $c_1$)
2:     $c_1 \leftarrow \emptyset$
3:     $totalCost \leftarrow +\infty$
4:     $euPosesCostL \leftarrow \emptyset$        ▷ Sort observation poses by ascending heuristic cost.
5:     for each $obsPose$ in $o.obsPoses$ do
6:         $euPosesCostL$.insert($\{|R, obsPose| + |obsPose, pushPose|\}$)
7:     end for
8:     $pathToObsL \leftarrow \emptyset$
9:     if $euPosesCostL \neq \emptyset$ then
10:        $op_{next} \leftarrow euPosesCostL[0]$
11:        while $totalCost \geq op_{next}.cost$ AND $op_{next} \neq$ null do
12:            if $pathToObsL[op_{next}.obsPose] \neq$ null then
13:                $o \leftarrow pathToObsL[op_{next}.obsPose]$
14:            else
15:                $o \leftarrow A^*(R, op_{next}.obsPose, I.occGrid)$
16:                $pathToObsL$.insert($\{op_{next}.obsPose, o\}$)
17:            end if
18:            $c \leftarrow A^*(op_{next}.obsPose, pushPose, I.occGrid)$
19:            $newTotalCost = |o| + |c|$
20:            if $totalCost \geq newTotalCost$ AND $newTotalCost < +\infty$ then
21:                $o_1 = o$
22:                $c_1 = c$
23:                $totalCost = |o_1| + |c_1|$
24:            end if
25:            $op_{next} \leftarrow euPosesCostL$.getNext()
26:        end while
27:    end if
28: end procedure
```

4.3 Social awareness through placement consideration

Algorithm 13 Obstacle evaluation subroutine modified for considering placement.

```
1: procedure PLAN-FOR-OBSTACLE($o$, $p_{opt}$, $I$, $R$, $R_{goal}$, $blockedObsL$, $occCostGrid$)
2:     ...                                              ▷ Initialization (lines 2 to 5 in Algorithm 7)
3:     for each $pushPose$ in $o.pushPoses$ do
4:         $pushUnit \leftarrow (cos(pushPose.yaw), sin(pushPose.yaw))$
5:         $c_1 \leftarrow A*(R, pushPose, I.occGrid)$
6:         if $c_1 = \emptyset$ then
7:             continue
8:         end if
9:         $seq \leftarrow 1$
10:        $translation \leftarrow pushUnit * onePushDist * seq$
11:        $safeSweptArea \leftarrow$GET-SAFE-SWEPT-AREA($o$, $translation$, $I$)
12:        $oSimPose \leftarrow pushPose + translation$
13:        $suppC_M \leftarrow$GET-OCC-COST(GET-OBS-POINTS($o$, $translation$), $occCostGrid$)
14:        $c3_{(Est)} \leftarrow \{oSimPose, R_{goal}\}$
15:        $C_{est} \leftarrow (|c_1| + |c3_{(Est)}|) * moveCost + |translation| * o.pushCost * suppC_M$
16:        while $C_{est} \leq p_{opt}.cost$ AND $safeSweptArea \neq$ null do
17:            $c_2 \leftarrow \{pushPose, oSimPose\}$
18:            $c_3 \leftarrow A*(oSimPose, R_{goal}, I.occGrid)$
19:            if $c_3 \neq \emptyset$ then
20:                $p.components \leftarrow [c_1, c_2, c_3]$
21:                $p.cost \leftarrow (|c_1| + |c_3|) * moveCost + |c_2| * o.pushCost * suppC_M$
22:                $p.minCost \leftarrow |c_2| * o.pushCost * suppC_M + |c_3| * moveCost$
23:                $p.o \leftarrow$ COPY($o$)
24:                $p.translation \leftarrow translation$
25:                $p.safeSweptArea \leftarrow safeSweptArea$
26:                $P_{o,d} \leftarrow P_{o,d} \bigcup \{p\}$
27:            end if
28:            $seq \leftarrow seq + 1$
29:            $translation \leftarrow pushUnit * onePushDist * seq$
30:            $safeSweptArea \leftarrow$GET-SAFE-SWEPT-AREA($o$, $translation$, $I$)
31:            $oSimPose \leftarrow pushPose + translation$
32:            $suppC_M \leftarrow$GET-OCC-COST(GET-OBS-POINTS($o$, $translation$), $occCostGrid$)
33:            $c3_{(Est)} \leftarrow \{oSimPose, R_{goal}\}$
34:            $C_{est} \leftarrow (|c_1| + |c3_{(Est)}|) * moveCost + |translation| * o.pushCost * suppC_M$
35:        end while
36:    end for
37:    return $p \in P_{o,d}$ with minimal $p.cost$ or null if $P_{o,d} = \emptyset$
38: end procedure
39: procedure GET-OCC-COST($simOccPoints$, $occCostGrid$)
40:    $cost \leftarrow 0$
41:    for each $point$ in $simOccPoints$ do
42:        $valueForPoint \leftarrow occCostGrid[point]$
43:        if $valueForPoint = FORBIDDEN\_VALUE$ then
44:            return $INF$
45:        end if
46:        $cost \leftarrow cost + valueForPoint/MAX\_VALUE$
47:    end for
48:    return $cost$
49: end procedure
```

4.4 Taking dynamic obstacles into account

Algorithm 14 Execution loop taking dynamic obstacles into account.

```
1: procedure MAKE-AND-EXECUTE-PLAN($R_{init}$, $R_{goal}$, $I_{init}$)
2:     ...                                  ▷ Initialization (lines 2 and 4 to 6 in Algorithm 3)
3:     $p_{opt}.components \leftarrow [A^*(R_{init}, R_{goal}, I.occGrid)]$
4:     $p_{opt}.cost \leftarrow |p_{opt}| * moveCost$
5:     $blockingObsL \leftarrow \emptyset$
6:     $isBlockingObsMoved \leftarrow False$
7:     $I \leftarrow I_{init}$
8:     while $R \neq R_{goal}$ do
9:         UPDATE-FROM-NEW-INFORMATION($I$)
10:        if $I$.freeSpaceCreated then
11:            $minCostL \leftarrow \emptyset$
12:        end if
13:        $\mathcal{O} \leftarrow I.allObstacles$
14:        $latestBlockingObsL \leftarrow p_{opt} \bigcap \mathcal{O}$
15:        $isPathFree \leftarrow latestBlockingObsL \neq \emptyset$
16:        $blockingObsL \leftarrow blockingObsL \bigcup latestBlockingObsL$    ▷ Now we must consider all
           obstacles, since not only new obstacles can get in the way : $\mathcal{O}_{new}$ is no longer useful.
17:        $isBlockingObsMoved \leftarrow False$
18:        for each $o$ in $blockingObsL$ do
19:            if not $isBlockingObsMoved$ AND $o \in I.movedObstacles$ then    ▷ When updated,
           $I$ saves in list the obstacles that have moved by checking whether they don't occupy some of
           the space they previously occupied.
20:                $isBlockingObsMoved \leftarrow True$
21:            end if
22:        end for
23:        ...                              ▷ Plan validity checks (lines 14 to 29 in Algorithm 6)
24:        $isPlanValid \leftarrow (isPathFree$ AND $isManipSuccess$ AND $isManipSafe$ AND
           $isPushPoseValid)$
25:        if not $isPlanValid$ then
26:            $p_{opt}.components \leftarrow [A^*(R, R_{goal}, I.occGrid)]$
27:            $p_{opt}.cost \leftarrow |p_{opt}| * moveCost$
28:        end if
29:        if not $isPlanValid$ OR ($isBlockingObsMoved$) then
30:            MAKE-PLAN($R$, $R_{goal}$, $I$, $\mathcal{O}$, $blockedObsL$, $p_{opt}$, $euCostL$, $minCostL$)
31:            $isManipSuccess \leftarrow True$         ▷ Line 30 in Algorithm 3 is moved here.
32:            continue                     ▷ Go back to beginning of the loop to check for
           new information because MAKE-PLAN may take a long time to run and obstacles may have
           moved since we began the evaluation. If the world state hasn't changed in the meantime, the
           produced plan will still be valid and MAKE-PLAN won't be called again. We don't forget to
           reset $isManipSuccess$ sooner.
33:        end if
34:        ...                              ▷ Execution (lines 24 to 29 and 21 to 35 in Algorithm 3)
35:    end while
36:    return $True$
37: end procedure
```

4.5 Algorithm proposition

Chapter 5

Validation through simulation & Tooling Development

5.1 Portable Linux ROS Workspace for Pepper with Docker

5.2 ROS-Standards Compatible Simulator

5.3 Simulation results

Re-reading the three papers about NAMO in unknown environments also offered the opportunity to reconsider three criteria for evaluating our experiment, in addition to the overall runtime:

- The average number of obstacle evaluations: characterizes the gain of ordering candidate objects for evaluation.
- The average number of path planning algorithm calls: characterizes the gain of the choice of the bound during obstacle evaluation.
- The most efficient move/manipulation ratio: characterizes the density of the environment for which the algorithm gives the best performance.

Chapter 6

Conclusion and Perspectives

Appendix A

Algorithms

A.1 Efficient Opening Detection, Levihn M. and Stilman M. (2011), Commented

Note on GET-M_i'-MATRIX The world W is represented by an occupancy grid: this procedure extends the obstacle M_i by the robot's diameter, giving us M_i', then represented as a binary matrix M, which has the size of the bounding box of M_i'.

Note on GET-NEW-X/Y-POS Simulate the set of manipulation actions A_M and get the world coordinates of the object.

Note on interpreting the value of Z If Z is the 0-matrix, it means no new openings were detected, as all blocking areas still are blocking after the manipulation. Else, it means that one intersecting area has disappeared, meaning a possible new opening.

Note on detecting BAs Check for blockage between M_i' (represented by M) and other obstacles (which data is contained in the occupancy grid G). For that we only call ASSIGN-NR if at least one of the two current elements of both matrices signals an obstacle ($\neq 0$).

Note on deletion If an index has already been deleted in an element of BS, delete it everywhere else because if part of a previous blocking area is detected, it means that the robot is still blocked by the same area. If for a same element of BS, there is an index in $BA[x][y] and BA_s^*[x][y]$, then it means that the blocking area still exists, thus we zero it in BS.

Algorithm 15 Efficient Local Opening Detection algorithm, Levihn et. al. (2011), commented

```
procedure CHECK-NEW-OPENING($G$, $M_i$, $A_M$, $BA$)
    $M \leftarrow$ GET-$M_i'$-MATRIX($M_i$)    ▷ Note on GET-$M_i'$-MATRIX
    $x_{offset} \leftarrow M_i.x$    ▷ $M_i.x$ and $M_i.y$ are the map coordinates of $M_i$'s top left corner.
    $y_{offset} \leftarrow M_i.y$
    if $BA$ is null then    ▷ BA needs not be recomputed if the environment did not change.
        $BA \leftarrow$ GET-BLOCKING-AREAS($x_{offset}$, $y_{offset}$, $M$, $G$)    ▷ Blocking areas before manip.
    end if
    $x_{offset} \leftarrow$ GET-NEW-X-POS($A_M$, $M_i$)    ▷ Note on GET-NEW-X/Y-POS
    $y_{offset} \leftarrow$ GET-NEW-Y-POS($A_M$, $M_i$)
    $BA_s \leftarrow$ GET-BLOCKING-AREAS($x_{offset}$, $y_{offset}$, $M$, $G$)    ▷ Blocking areas after manip.
    $BA_s^* \leftarrow [0][0](dim(M))$    ▷ Since the window of $BA_s$ shifted with the manipulation of $M_i$, ...
    for $k$ from 0 to $|BA_s^*|$ do    ▷ ... we shift it back for the future comparison with $BA$.
        for $l$ from 0 to $|BA_s^*[i]|$ do
            $x \leftarrow (x_{offset} - M_i.x) + k$
            $y \leftarrow (y_{offset} - M_i.y) + l$
            if $0 < x < |BA_s^*|$ AND $0 < y < |BA_s^*[x]|$ then
                $BA_s^*[x][y] \leftarrow |BA_s^*|[k][l]$
            end if
        end for
    end for
    $Z \leftarrow$ COMPARE($BA$, $BA_s^*$)    ▷ Finally, compare the two blocking aread configurations.
    if $Z = [0][0](dim(M))$ then    ▷ Note on interpreting the value of Z
        return $false$
    end if
    return $true$
end procedure

procedure GET-BLOCKING-AREAS($x_{off}$, $y_{off}$, $M$, $G$)
    $index \leftarrow 1$
    $BA \leftarrow [0][0](dim(M))$    ▷ $[0][0](dim(M))$ represents the 0-Matrix of dimensions = $dim(M)$
    for $x$ from 0 to $|M|$ do    ▷ Iterate over $M$ to detect and tag blocking areas.
        for $y$ from 0 to $|M[x]|$ do
            if $M[x][y] \neq 0$ AND $G[x + x_{off}][y + y_{off}] \neq 0$ then    ▷ Note on detecting BAs
                ASSIGN-NR($BA$, $x$, $y$, $index$)    ▷ $BA$ and $index$ are directly modified in the call.
            end if
        end for
    end for
    return $BA$    ▷ Return the saved information on the blocked areas.
end procedure
```

```
procedure ASSIGN-NR($BA, x, y, index$)
    for $i$ from -1 to 1 do            ▷ Assignment is performed based on the 3*3 neighborhood.
        for $j$ from -1 to 1 do
            if $BA[x+i][y+j] \neq 0$ then            ▷ If a number is already in the neighborhood, ...
                $BA[x][y] \leftarrow BA[x+i][y+j]$            ▷ ... the same number is assigned to the element.
                return
            end if
        end for
    end for
    $BA[x][y] \leftarrow index$            ▷ Else a new number is assigned, equal to the new index.
    $index \leftarrow index + 1$            ▷ Only increment index if new intersection is created.
    return
end procedure

procedure COMPARE($BA, BA_s^*$)            ▷ This function checks for non-zero entries in both matrices.
    $BS \leftarrow$ COPY($BA$)
    $del_{num} \leftarrow \emptyset$            ▷ Set of the indexes of obstacles to delete from $BS$.
    for $x$ from 0 to $|BS|$ do            ▷ Iterate over $BS$.
        for $y$ from 0 to $|BS[x]|$ do
            if $BA[x][y] \in del_{num}$ then            ▷ Note on deletion
                $BS[x][y] \leftarrow 0$
            end if
            if $BA[x][y] \neq 0$ AND $BA_s^*[x][y] \neq 0$ then            ▷ Note on deletion
                $del_{num} = del_{num} \bigcup BS[x][y]$
                $BS[x][y] \leftarrow 0$
            end if
        end for
    end for
    return $BS$
end procedure
```

Bibliography

[1] M. Levihn, M. Stilman, and H. Christensen. "Locally optimal navigation among movable obstacles in unknown environments". In: *2014 IEEE-RAS International Conference on Humanoid Robots*. 2014 IEEE-RAS International Conference on Humanoid Robots. Nov. 2014, pp. 86–91. DOI: 10.1109/HUMANOIDS.2014.7041342.

[2] Martin Levihn and Mike Stilman. *Efficient Opening Detection*. Technical Report. Georgia Institute of Technology, 2011. URL: https://smartech.gatech.edu/handle/1853/40954 (visited on 03/25/2018).

[3] Hai-Ning Wu, M. Levihn, and M. Stilman. "Navigation Among Movable Obstacles in unknown environments". In: *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2010 IEEE/RSJ International Conference on Intelligent Robots and Systems. Oct. 2010, pp. 1433–1438. DOI: 10.1109/IROS.2010.5649744.