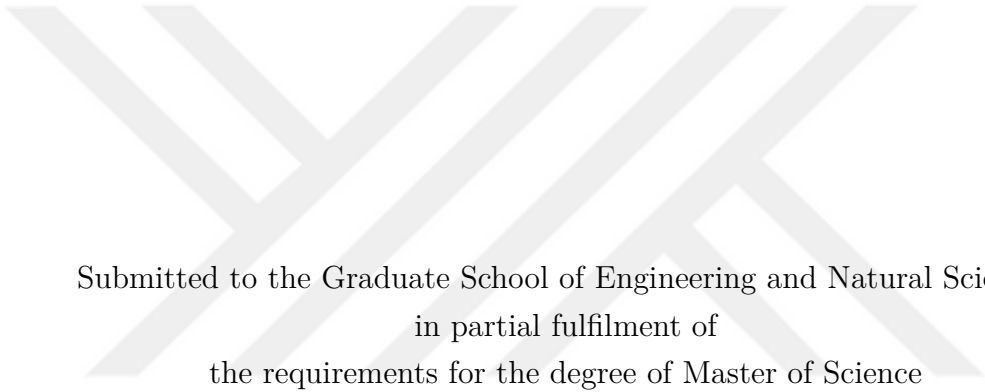


**PLACEMENT GENERATION AND HYBRID PLANNING FOR
ROBOTIC REARRANGEMENT ON CLUTTERED SURFACES**

by
ABDUL RAHMAN DABBOUR



Submitted to the Graduate School of Engineering and Natural Sciences
in partial fulfilment of
the requirements for the degree of Master of Science

Sabanci University
July 2019

PLACEMENT GENERATION AND HYBRID PLANNING FOR
ROBOTIC REARRANGEMENT ON CLUTTERED SURFACES

Approved by:

Assoc. Prof. Dr. Volkan Patoğlu
(Thesis Advisor)



Assoc. Prof. Dr. Esra Erdem Patoğlu
(Thesis Co-Advisor)



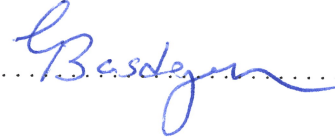
Assoc. Prof. Dr. Güllü Kızıldaş Şendur



Asst. Prof. Dr. Öznür Taştan



Prof. Dr. Çağatay Başdoğan
(Koç University)



Date of approval: 19 July 2019



ABDUL RAHMAN DABBOUR 2019 ©

All Rights Reserved

ABSTRACT

PLACEMENT GENERATION AND HYBRID PLANNING FOR ROBOTIC REARRANGEMENT ON CLUTTERED SURFACES

ABDUL RAHMAN DABBOUR

MECHATRONICS ENGINEERING

MSc. THESIS

JULY 2019

Thesis Advisor: Assoc. Prof. Dr. Volkan Patoğlu

Thesis Co-Advisor: Assoc. Prof. Dr. Esra Erdem Patoğlu

Keywords: Placement generation, rearrangement planning of multiple movable objects, hybrid planning, service robotics.

Rearranging multiple moving objects across surfaces, e.g. from a table to kitchen shelves as it arises in the context of service robotics, is a challenging problem. The rearrangement problem consists of two subproblems: placement generation and rearrangement planning. Firstly, the collision-free goal poses of the objects to be moved need to be determined subject to the arbitrary geometries of the objects and the state of the surface that already includes movable objects (clutter) and immovable obstacles on it. Secondly, after the goal poses of all objects have been determined, a plan of physical actions must be computed to achieve these goal poses. Computation of such a rearrangement plan is difficult in that it necessitates not only high-level task planning, but also low-level feasibility checks to be integrated with this task plan to ensure that each step of the plan is collision-free.

In this thesis, we propose a general solution to the rearrangement of multiple arbitrarily-shaped objects on a cluttered flat surface with multiple movable objects and obstacles. In particular, we introduce a novel method to solve the object placement problem, utilizing nested local searches guided by intelligent heuristics to efficiently perform multi-objective optimizations. The solutions computed by our

method satisfy the collision-freeness constraint, and involves minimal movements of the clutter. Based on such a solution, we introduce a hybrid method to generate an optimal feasible rearrangement plan, by integrating ASP-based high-level task planning with low-level feasibility checks. Our hybrid planner is capable of solving challenging non-monotone rearrangement planning instances that cannot be solved by the existing geometric rearrangement approaches.

The proposed algorithms have been systematically evaluated in terms of computational efficiency, solution quality, success rate, and scalability. Furthermore, several challenging benchmark instances have been introduced that demonstrate the capabilities of these methods. The real-life applicability of the proposed approaches have also been verified through physical implementation using a BAXTER robot.



ÖZET

DAĞINIK YÜZEYLERDE ROBOTİK DÜZENLEME İÇİN YERLEŞİM OLUŞTURMA VE HİBRİT PLANLAMA

ABDUL RAHMAN DABBOUR

MEKATRONİK MÜHENDİSLİĞİ
YÜKSEK LİSANS TEZİ
TEMMUZ 2019

Tez Danışmanı: Doç. Dr. Volkan Patoğlu
Tez Eş Danışmanı: Doç. Dr. Esra Erdem Patoğlu

Anahtar Kelimeler: Yerleşim oluşturma, hareketli nesneleri yeniden düzenleme, hibrit planlama, hizmet robotlar.

Hareket ettirilebilen pek çok nesnenin farklı yüzeyler üzerinde yeniden düzenlenmesi zor bir problemdir. Örneğin, servis robotığı bağlamında, nesnelerin bir masadan kalabalık mutfak raflarına taşınması bu tür bir problemdir. Yeniden düzenleme problemi iki alt problemden oluşmaktadır: yerleşim oluşturma ve düzenleme planı oluşturma. İlk olarak, hareket ettirilecek (kalabalık) nesnelerle geometrilerine ve yüzeyin üzerinde bulunan hareket ettirilebilen ve hareket ettirilemeyen (engel) nesneler arasında çarpışmalarla sonuçlanmayacak konumlar ve yönelimler belirlenmelidir. İkinci olarak, nesnelerin hedef durumları belirlendikten sonra, bu hedef durumlara ulaşmak için gerekli eylem planı hesaplanmalıdır. Böyle bir planın hesaplanması güçtür, çünkü sadece yüksek seviye planlama yeterli değildir, aynı zamanda planın her bir adımının çarpışma içermemesini sağlamak için düşük seviyede uygulanabilirlik kontrolü yapılması gereklidir.

Bu tezde, farklı şekillere sahip nesneleri ve engellerin dağınık bir yüzey üzerinde yeniden düzenlemesi için genel bir çözüm önermekteyiz. Öncelikle, nesne yerleştirme problemini sezgisel yöntemler ile yönlendirilen iç içe yerel aramalar kullanan birçok

kriterli optimazasyon problemi olarak çözmekteyiz. Metodumuzla hesaplanan çözümler yüzey üzerinde halihazırda var olan nesnelerin hareketlerini en aza indirip çarpışmasızlık kısıtı sağlamaktadır. Bu çözüme dayanarak, optimum ve fiziksel olarak uygulanabilen bir düzenleme planı hesaplamak için, ASP-tabanlı yüksek seviye eylem planlama ile düşük seviye uygulanabilirlik kontrolünü entegre ederek hibrit bir metod önermekteyiz. Hibrit planlama yaklaşımımız, mevcut geometrik düzenleme yaklaşımları ile çözülemeyen, monoton nitelikte olmayan zor planlama problemlerini çözebilmektedir.

Önerilen yöntemler hesaplama verimliliği, çözüm kalitesi, başarı oranı ve ölçeklenebilirlik açılarından sistematik olarak değerlendirilmiştir. Ayrıca, bu yöntemlerin yetkinliklerini gösteren birkaç zor test problemi de sunulmuştur. Önerilen yaklaşımların gerçek hayatta uygulanabilirliği bir BAXTER robot kullanarak gerçekleştirilen bir fiziksel uygulama ile doğrulanmıştır.



ACKNOWLEDGEMENTS

For their continuous guidance, support, and encouragement, I would like to express my deepest gratitude to my advisors, Assoc. Prof. Dr. Volkan Patoğlu and Assoc. Prof. Dr. Esra Erdem Patoğlu, who taught me the principles of research, and gave me all the freedom in my work while directing me with sound advice throughout.

For their unconditional and unrelenting love and support throughout this journey, I would like to convey my thanks to my father, mother, and three siblings.

For always providing much needed relief in stressful times and widening my horizons – academic or otherwise – and for making my Sabancı experience that much more colorful, I would like to express my gratitude to my friends.

For their thought-provoking discussions, feedback, and friendship, I would like to convey my thanks to the members of the Cognitive Robotics Lab, robots included.

For sharing with me, and the general public, high-quality tools and deep knowledge free of charge, I would like to extend my appreciation to the open-source community.

For their time and feedback, I would like to express my regards to the jury members.

When I first entered Sabancı University in the winter of 2017, I could not imagine the amount of knowledge and experience that I would be able to acquire in such a short time. This work would not be possible if not for the collaboration of numerous anonymous parties working to my – and other students’ – advantage. Thank you.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Approach	2
1.3	Challenges	3
1.4	Contributions	4
1.5	Thesis Outline	5
2	Related Work	6
2.1	Placement Generation	6
2.2	Rearrangement Planning	8
3	Preliminaries	11
3.1	Search	11
3.2	Answer Set Programming	13
3.3	Collision Resolution	15
4	Placement Generation	17
4.1	Problem Definition	17
4.2	Methods	18
4.2.1	Penetration Minimization	18
4.2.2	Collision Minimization	19
4.2.3	Rearrangement Minimization	21
5	Rearrangement Planning	25
5.1	Problem Definition	25
5.2	Methods	25
5.2.1	Configuration Discretization	26
5.2.2	High-Level Planning	28
5.2.3	Low-Level Feasibility Check	29
6	Experimental Evaluation	32

6.1	Placement Generation	32
6.1.1	Benchmark Instances	38
6.2	Rearrangement Planning	40
6.2.1	Benchmark Instances	45
6.2.2	Comparison of Hybrid Planning Approaches	46
6.3	Physical Implementation	47
7	Conclusion	51
	Bibliography	51
A	Detailed Results	57



List of Figures

3.1	Petersen graph initial coloring.	14
3.2	Petersen graph final coloring.	14
4.1	A sample problem instance and a possible solution.	18
4.2	Intermediate local search algorithm	22
4.3	Outermost local search algorithm progression	24
5.1	An example of configuration discretization.	27
6.1	Plots summarizing the results of Experiment 1.	34
6.2	Plots summarizing the results of Experiment 2.	35
6.3	Plots summarizing the results of Experiment 3.	36
6.4	Confined Placement Scenario.	38
6.5	Tight placement scenario.	39
6.6	Elongated objects scenario.	39
6.7	The tight L-shapes scenario with concavity across the x-y plane. . . .	40
6.8	The tight L-shapes scenario with concavity across the z-y plane. . . .	40
6.9	Plots summarizing the planning scalability based on Experiments 4-6. . .	41
6.10	Plots summarizing the results of Experiment 4.	42
6.11	Plots summarizing the results of Experiment 5.	43
6.12	Plots summarizing the results of Experiment 6.	44
6.13	Enforced Swaps Scenario.	45
6.14	Tight Placement Scenario.	46
6.15	Plots summarizing the results of Experiment 7.	47
6.16	BAXTER executing a plan with placement generated using <i>Outer</i>	48
6.17	BAXTER executing a plan with placement generated using <i>Intermediate</i> . .	49
6.18	BAXTER executing a nonmonotone plan with a forced swap.	50

List of Tables

2.1	Comparison of selected related works for placement generation.	8
2.2	Comparison of selected related works for rearrangement planning. . .	10
A.1	Experiments 1, 2, and 3 results: placement generation performance. .	58
A.2	Experiment 4, 5, and 6 results: effect of placement generation algo- rithm on rearrangement planning performance.	59

List of Algorithms

1	HILLCLIMBING	12
2	INNERMOSTSEARCH	19
3	INTERMEDIATESEARCH	21
4	OUTERMOSTSEARCH	23
5	NAIVEDISCRETIZER	26
6	ISCOLLISION	29
7	FEASIBILITYSEARCH	30
8	PROPAGATE	31

Chapter 1

Introduction

1.1 Motivation

For useful integration of robotic systems into everyday life, they must be capable of performing high-complexity real-life tasks efficiently. For instance, typical human environments, such as table tops, kitchen shelves, or office desks, are usually cluttered; and manipulating the environment to deal with such clutter is integral to performing everyday chores in social environments, whether that means rearranging objects upon a surface or across multiple surfaces.

Geometric rearrangement of multiple movable objects on a surface is a difficult problem, because it requires the manipulation of existing objects on the surface, as well as the placement of new objects to be put on the surface. The order of manipulation actions carried out is also relevant, e.g. an elongated object might need to be rotated or moved away first before another object can take its place.

Generally, solving such a problem requires task planning to decide for the order of manipulation actions (e.g., when to pick, place, move objects), and feasibility checks are required to check the execution of each manipulation action against geometric/kinematic constraints (e.g., to avoid collisions). These requirements usually lead to hybrid planning solutions that combine high-level task planning and low-level feasibility checks.

However, before attempting to solve such a planning problem for the rearrangement of objects in a clutter, one needs to know the goal configuration (i.e., how the objects are arranged on the surface at the end of the plan). But, in real life scenarios, this information is not available most of the time, especially for heavily-cluttered or tightly-packed scenarios. Therefore, it is crucial to determine a geometrically feasible goal configuration of objects on the surface before planning for rearrangements.

Placement generation, generating such a goal configuration, is an understudied problem, especially in the robotics domain. To the best of the author’s knowledge, outside of [1], there exists no work considering the problem of generating a goal state for the placement of multiple movable objects on top of a continuous surface that is already cluttered with other multiple movable and immovable objects. While this is briefly mentioned in [1], the method is not explained in detail, and there are no quantitative evaluations offered.

On the contrary, rearrangement planning has enjoyed attention within the robotics research community. However, as is discussed in Chapter 2, methods that generalize to non-monotonic planners are rare, and methods that further generalize to using any point on the entire continuous surface for *buffer poses* – poses to use for non-monotonic plan steps – are even rarer.

With this motivation, we study:

- the **placement generation** problem: given a surface cluttered with (unmovable) obstacles and a set of existing movable objects on it, and a set of new objects to be placed on the surface, the goal is to find a collision-free placement of all objects on the surface while minimizing the total number and amount of displacements of the existing movable objects.
- the **rearrangement planning** problem: given an initial placement of obstacles and movable objects, and a final placement of obstacles, movable objects, and new objects, the goal is to find the minimum set of actions to be taken by a robot to transform the initial placement to the final placement.

1.2 Approach

The overall approach is as follows:

1. The **placement generation** finds a collision-free final configuration for all objects (all the new objects together with all other objects in the clutter) while also trying to minimize the number of object relocations, and the amount of movement each object is relocated. At this stage, the number, type, order, and feasibility of the move actions required to achieve this goal configuration are not considered.
2. The **rearrangement planning** is divided into two stages:
 - (a) The **discrete placement** stage takes, as input, the initial configurations and the final configurations of all objects on the cluttered surface, and

divides the surface into a minimum number of non-uniform grid cells. During gridization of the continuous plane, an object is allowed to span multiple grid cells as long as each grid cell contains the centroid of a single object only.

- (b) The **hybrid planning** stage aims to find a sequence of collision-free steps through feasible move actions (i.e. pick-and-place actions) to achieve the final placement of the objects in the clutter from their initial discrete placement, while simultaneously minimizing the number of actions.

1.3 Challenges

Generating a goal placement and a rearrangement plan introduces tough challenges:

- **The curse of dimensionality:** In the context of the problems we have motivated, there is no upper bound on the number of objects for which we are attempting to find a placement for, or trying to plan for the rearrangement of. Even after fixing the plane of contact between the objects and surface, searching for a placement for each object is still equivalent to searching for a pose vector of 2 translational and a rotational dimension per object. Since the curse of dimensionality is a specially difficult problem to attack from a planning perspective, it is imperative that a solution developed for the placement generation is both scalable and capable of reducing the strain on the following planning algorithm.
- **Arbitrary Geometry:** Unlike some industrial environments, we cannot make assumptions about the geometry of objects in service robotics scenarios. This means that our algorithms must be able to deal with multiple heterogeneous objects of any size, orientation, or shape. Concave and elongated geometries especially introduce difficulties by constraining the poses of other objects in challenging ways.
- **Continuous Domain:** In the interest of generalization, we tackle the problem in its real-life domain, continuous in all dimensions. Note that while our algorithms use discretizations as heuristics during placement and planning, the final step is always based on a continuous surface.

1.4 Contributions

Robotic rearrangement of objects on cluttered surfaces has two main challenges: (i) goal generation, i.e., computing a collision-free placement of objects and (ii) planning, i.e., computing a sequence of feasible manipulation actions to realise the generated goal. In light of these challenges associated with the problems, we summarize our contributions as follows:

- We have introduced a novel, efficient nested local search algorithm for generating a collision-free placement of multiple arbitrarily-shaped objects on a flat but otherwise arbitrarily-shaped, crowded surface with multiple movable objects and static obstacles.
- We have introduced a hybrid method for planning for the actions of the robot rearranging the objects on the cluttered surface to reach a collision-free placement computed by our nested local search algorithm. In this method, we discretize the continuous rearrangement problem, and combine high-level task planning with low-level feasibility checks. In particular, for ensuring connectivity of actions executed one after another we introduce a novel feasibility check based on local search.
- We have designed and generated a set of object placement problem benchmarks to experimentally evaluate our object placement algorithm. These benchmarks include randomly generated problem instances with non-convex objects, extremely confined surfaces, and where the solutions are confined to very specific regions.
- We have systematically evaluated our object placement algorithm to show the usefulness of each local search layer, from the perspectives of computational efficiency, success rate, and solution quality.
- We have designed and generated a set of rearrangement planning benchmarks to experimentally evaluate our hybrid method. The benchmark problems require multiple and circular swaps of objects.
- We have evaluated our hybrid planning algorithm to investigate its scalability in terms of computation time. We have also investigated the effect of using object placements generated by the outermost layer of our placement algorithm against those generated by the intermediate layer to better understand the usefulness of minimizing the number of rearranged objects during placement.
- We have implemented the object placement and rearrangement planning algorithms in PYTHON: <https://github.com/ardabbour/rearrangement>.

- We have shown the applicability of our methods with a physical implementation using a BAXTER robot. We have considered several scenarios where different shaped objects are placed on a cluttered surface. The BAXTER robot has autonomously identified the objects in the clutter using the cameras embedded in its arm and computed a collision-free placement of the objects in the clutter, found a hybrid plan to reach it, and executed it.

1.5 Thesis Outline

The rest of the thesis is organized as follows:

Chapter 2 discusses related works. this section is divided into two: works relating to placement generation and works relating to rearrangement planning. After each discussion, a table summarizing the related works and comparing them to this thesis is presented.

Chapter 3 introduces the main concepts upon which this thesis is built and provides examples to aid the reader in understanding the main use cases, then focuses on more relevant elements of these concepts.

Chapter 4 explains the placement generation algorithms, which constitute the major novelty of this thesis.

Chapter 5 describes how the rearrangement planning is accomplished in light of the hybrid planning framework.

Chapter 6 highlights the results achieved using the methods described in Chapters 4–5 and discusses them in detail.

Finally, Chapter 7 recaps the thesis and discusses future works.

Chapter 2

Related Work

Rearrangement of multiple movable objects, a challenging problem that involves planning, manipulation and geometric reasoning, has received much attention in robotics. In particular, planning for geometric rearrangement with multiple movable objects and its variations, such as navigation among movable obstacles [2], [3], have been studied using various approaches. Since even a simplified variant the rearrangement problem with only one movable obstacle has been proved to be NP-hard [4], [5], most studies introduce several important restrictions to the problem, like monotonicity of plans [6]–[10], where each object can be moved at most once. Recent work has focused on generating non-monotonic plans [1], [11]–[15]. However, in most of these studies [6]–[9], [11]–[14], [16], [17], it is assumed that the goal configuration is known. Finding suitable arrangements for objects on a cluttered surface has received relatively less attention.

Tables 2.1 and 2.2 show a comparison of selected related works with this thesis.

2.1 Placement Generation

Cosgun et al. [10] propose an algorithm that searches for a suitable placement for a single object on a cluttered surface by discretizing the possible orientations of the object, convolving object pixels with the ones on the table, and identifying candidate regions for the object placement that result in minimal penetration with other objects. A placement is then produced by sampling these regions; however, this placement may not be collision-free. Then, they plan for a sequence of linear push actions to rearrange the clutter and clear space for the new object such that this placement becomes collision-free. Note that there are several limitations in this approach; multiple new objects are not considered, the surface and object orientations

are discretized, and the final configuration is not necessarily collision-free.

Yu et al. [18] aim to find sensible placements for furniture by initially generating a random arrangement, then rearranging it to minimize a cost function that measures the difference between the current arrangement and several positive examples provided by the user. Kang et al. [15] also follow on this idea, and modify the algorithm so it becomes more suitable for robotic applications. Neither study considers heavily cluttered scenes or utilizes high resolution collision checks. In [15], the task is to rearrange objects currently available in the scene to achieve a more tidy arrangement; no new objects are added and there exists no constraints that force a certain set of objects to be on certain surfaces. Furthermore, in these studies, even the initial state is a feasible (collision-free) configuration, and the goal is to improve it in terms of a measure of tidiness.

Jiang et al. [19]–[21] extract object-to-object and object-to-human features from databases of 3D environments and learn semantic/geometric preferences for object surface pairs. Then, they discretize the surfaces’ point cloud into placing areas by random sampling and solve an maximum matching problem to assign each object’s pose to a suitable placing area. This approach only considers placements to a pre-determined set of discrete configurations and does not address the more challenging continuous version of the problem.

In [1], a placement generation algorithm based on a local search guided with heuristics and random restarts is proposed. This work significantly extends this earlier study by introducing an innermost search layer that uses established collision resolution methods, as well as two nested local searches wrapped around this basic search algorithm, to improve upon the efficiency and quality of solutions, as well as the success rate. The results in chapter 6.1 indicate orders of magnitude difference in terms of CPU time and success rate in cluttered scenarios.

A closely related problem to placement generation, studied in computer graphics and operations research, is the packing problem (also known as the knapsack problem), where the goal is to place as many objects as possible in a non-overlapping configuration within a given empty container. The packing problem is NP-hard [22]. It has been widely studied in 2D context (cf. the survey [23]). It has been also studied in 3D under various conditions/restrictions [24]–[26] (e.g., packing a set of polyhedrons into a fixed size polyhedron without considering rotations [27], orthogonal packing of tetris-like items into rectangular bins [28], [29]).

However, the placement generation problem is quite different from these packing problems. First of all, since the placement generation problem is motivated by the geometric rearrangement of objects on a cluttered surface, the surface does not have

Reference	Placement Problem Tackled					Placement Goals	
	Domain	DOF	New Objects	Stacking	Surfaces	Collisions	Arrangement
Cosgun et al., 2011	Discrete	6	Single	No	Single	Avoid then resolve at motion planning	Avoid rearrangements
Jiang et al., 2012	Discrete	6	Multiple	Yes	Multiple	Avoid using cost fn	Relative to positive examples
Jiang et al., 2012	Discrete	6	Multiple	Yes	Multiple	Avoid using cost fn	Relative to positive examples
Jiang et al., 2013	Discrete	6	Multiple	Yes	Multiple	Avoid using cost fn	Relative to positive examples
Yu et al., 2011	Continuous	3	Multiple	Yes	Single	Avoid using cost fn	Relative to positive examples
Kang et al., 2018	Continuous	3	None	Yes	Multiple	No	Relative to positive examples
Havur et al., 2014	Continuous	3	Multiple	No	Multiple	Resolve using guided re-placements	Avoid rearrangements implicitly
This Thesis	Continuous	3	Multiple	No	Multiple	Resolve using guided re-placements and collision resolution methods	Reduce rearranged objects and their movement

Table 2.1: Comparison of selected related works for placement generation.

to be empty and contains movable objects. The packing problem, on the other hand, assumes that the fixed size container is empty. Also the objective function for the placement generation problem is different: the goal is to find a collision-free configuration of all objects, so as to minimize the total number and amount of displacements of the existing objects on the surface. The packing problem, on the other hand, aims to find a collision-free configuration of some objects, so as to maximize the coverage rate (i.e., the total volume of the objects packed in the container). Along these lines, the packing problem and the placement problem are different computational problems with different optimization goals. The methods to attack these problems are significantly different from each other and do not allow for direct comparisons.

In this thesis, we focus on finding collision-free configurations for two specific sets of objects, objects on the surface and objects to be added onto the surface, taking into account the continuous nature of the domain.

2.2 Rearrangement Planning

A rearrangement planning algorithm proposed [1], upon which this work is based. This algorithm is executed sequentially as follows (i) first, a discrete representation of the initial and goal configurations is obtained by dividing the surface into the minimum number of non-uniform grids, where each grid cannot contain more than one object centroid, using an Answer Set Programming formulation, (ii) a plan is computed on this discrete representation using the hybrid planning paradigm [30], [31], where the high-level planner finds a discretely-feasible plan step, utilizing low-

level checks to verify the continuous feasibility of each step, and (iii), a local search algorithm is used to ensure collision-free configurations for all steps.

A hybrid planning paradigm is also employed by Dantam et al. [17], [32], where they utilize a SAT solver as their high-level task planner, and integrate it with low level motion planners. As in [1], the motion planner is able to add or remove constraints to the task planner to contract or expand the search space iteratively, depending on the feasibility of the actions queried by the task planner. Crucially, they prove their method to be probabilistically complete, when coupled with a probabilistically complete motion planner such as RRT-Connect [33]–[35]. Unlike in [30], [31], it is important to note that they do not claim the ability to deal with non-monotone plans. Non-monotone plans may be computed only if an additional buffer poses are introduced. Due to the method of discretization, the motion planner is allowed to inform the task planner of the infeasibility of a query placement, but has no way of informing the task planner of an alternative feasible placement in the region around the queried placement.

Han et al. [14] consider the problem of planning for the rearrangement of objects using only overhead grasps. They first discretize the surface into a set of points made of the initial and final centroid positions of the objects, as well as several buffer points to place objects for non-monotone scenarios. A graph is then constructed using these points as vertices, with the shortest line between them forming edges. They then reduce the Euclidean Traveling Salesperson Problem [36] – if the rearrangement plan is monotonic – or the Feedback Vertex Problem [37] to their own graph-based problem definition. This allows them to use highly efficient methods previously established in the literature to acquire a rearrangement plan. Importantly, the planner here provides no interaction between the high-level task plan and the low-level motion plan, since all tasks considered by the task planner are known to be feasible.

King et al. [38] introduce a planner based on a Monte-Carlo Tree Search (MCTS) [39] to create plans for the rearrangement of a single movable objects using non-prehensile actions, such as pushing. In this framework, the search is to find a sequence of actions whose probability of success (measured as expected value) is maximized. Using noisy physics simulations as black-box successor state generators, a tree is built where each node represents a sequence of actions; this tree is traversed using 3 policies: random, demonstration-based, and subsearch-based. The demonstration-based policy uses a machine learning algorithm based on human choices, whereas the subsearch-based policy searches through a discretized and simplified space of the problem, starting from the current node to the goal, for the optimal action. Note that this method is only tested in scenarios where clutter is minimal (i.e., surface

Reference	Action Space	Object Type	Non-Monotone Plans	Approach	Collision Resolution
Han et al., 2017	Discrete	Uniform	Yes, given buffer poses	Reduce to TSP or FVSP, solve with ILP	No
King et al., 2017	Discrete	Arbitrary	Not Applicable	MCTS	Not Applicable
Dantam et al., 2018	Discrete	Arbitrary	Yes, given buffer poses	PDDL with feasibility checks	No
Zagoruyko et al., 2019	Discrete	Arbitrary	Yes	MCTS with neural network	Rudimentary
Havur et al., 2014	Continuous	Arbitrary	Yes	Guided replanning with ASP	Yes
This Thesis	Continuous	Arbitrary	Yes	ASP with integrated feasibility checker	Yes

Table 2.2: Comparison of selected related works for rearrangement planning.

coverage is less than 50%). As a planner that uses a non-prehensile action set, it is capable of handling non-monotone instances.

Zagoruyko et al. [16] also use an MCTS to create a plan from a pre-defined set of discrete actions. While searching, a multilayer perceptron [40] that is trained in simulation is used to predict values of actions to guide the exploration of the tree, which the authors claim maintains the scalability of their approach to be better than others, with the downside of having to train a different network for every different number of objects to be rearranged. This framework also allows solving non-monotone instances.

This thesis extends the approach introduced in [1] by integrating the final local search algorithm into the hybrid planner and introducing improvements to result in significant computational speed ups. In this regard, it describes a generic planner capable of utilizing any set of robot actions to solve both monotone and non-monotone scenarios rearrangement scenarios. Unlike the planners described in [14], [16], [17], [38], our planner is capable of identifying multiple buffer poses on its own due to the approaches taken in discretization and task-motion planning.

Chapter 3

Preliminaries

3.1 Search

Search is one of the cornerstone ideas used in artificial intelligence [41] used for a very wide range of problems.

Usually, **classical search** is used when a sequence of locations and transitions (represented as states and actions) are required for defining a solution [41]. Commonly, the problem domain is fully observable, deterministic, and known, e.g., the traveling salesperson problem [42].

A classical search problem can be defined as a 7-tuple $\langle S, A, T, G, C, P, s_0 \rangle$ where:

- S is the set of all **states**,
- A is the set of all **actions**,
- $T : S \times A \rightarrow S$ is the **transition model** that maps a state-action tuple $\langle s_1, a_1 \rangle$ to a consequent state s_2 ,
- $G : S \rightarrow \{0, 1\}$ is the **goal test** to determine whether a state is a goal state,
- $C : S \times A \times S \rightarrow \mathbb{R}_+$ is the **step cost** of transitioning from state s_1 to state s_2 using action a ,
- $P : L \rightarrow \mathbb{R}_+$ is the **path cost** that sums the step costs of the sequence of state-action-state triplets L , and
- s_0 is the **initial state**.

Once the problem is defined, a search algorithm is chosen with suitable properties, such as completeness, optimality, and domain type to attack it.

Local search is a variant of search where the assumptions made earlier about the search space, such as observability and determinism, are relaxed, and where a solution does not include the path to find it [41]. The primary difference between local search and classical search is the lack of a goal test, step cost, and a path cost. Instead, a local search employs an **objective function** that determines the value of the state, making it an optimization framework. Note that while a local search problem does not keep track of the path, or the path cost, it can still be used to tackle planning problems by modeling a state as a sequence of transitions between configurations. Formally, we can define a local search problem as a 5-tuple $\langle S, A, T, O, s_0 \rangle$, where $O : S \rightarrow \mathbb{R}$ is the objective function.

The most basic approach towards local search, described in [41], is given in Algorithm 1, where the algorithm attempts to find higher-valued neighbors of a given state, using the transition model T , until it no higher-valued neighbor can be found. This approach is prone to result in suboptimal solutions because it has no way of escaping local minima, and is therefore highly dependent on the proximity and reachability – in terms of the transition model – between the initial and goal states.

One of the simplest ways to mitigate this is to use random restarts; i.e., rather than stopping the search, a random state is generated if no higher valued state exists, and the search is restarted.

Integrating this stochasticity into to hill-climbing by associating a probability distribution with the set of successors is another option. Higher state values correspond to higher probabilities of being chosen as successors. This way, some ‘downhill’ moves are allowed which could result in exploring more regions of the search space, at the cost of slower conversion to some solution. One such popular stochastic hill-climbing method is known as *simulated annealing*, where the variance of the probability distribution decreased with time. The ratio of exploitation to exploration increases with time, and the rate at which it does so becomes a parameter.

Algorithm 1 HILLCLIMBING

Input: *problem*

Output: a state that is a local maximum.

```

1: current  $\leftarrow$  problem.INITIAL
2: loop
3:   neighbor  $\leftarrow$  a highest-valued successor of current
4:   if VALUE(neighbor)  $\leq$  VALUE(current) then
5:     return current
6:   end if
7:   current  $\leftarrow$  neighbor
8: end loop
```

3.2 Answer Set Programming

Answer Set Programming, hereafter referred to as ASP, is a declarative programming paradigm [43]–[47] primarily used to solve difficult, often NP-hard, search problems. With ASP, a problem is represented as a finite set of **rules**, called a **program** P whose answer set corresponds to a solution. The answer sets for a program can be computed by ASP solvers. This thesis makes use of the ASP solvers CLINGO [48] and DLVHEX [49].

In this thesis, we consider programs that consist of rules of the form

$$\alpha_1 \vee \dots \vee \alpha_k \leftarrow \beta_1, \dots, \beta_n, \text{ not } \beta_{n+1}, \dots, \text{ not } \beta_m.$$

where $k \geq 0$, $n \geq 0$, $m \geq n$, each α_i is a propositional atom, and each β_j is a propositional atom or an external atom. $\alpha_1, \dots, \alpha_k$ is the **head** of the rule, and $\beta_1, \dots, \beta_n, \text{ not } \beta_{n+1}, \dots, \text{ not } \beta_m$ is the **body** of the rule. A headless body is called a **constraint**, and a bodyless head is called a **fact**.

An **external atom** is an expression of the form

$$\&g[Y_1, \dots, Y_n](X_1, \dots, X_m).$$

Here, g is a function defined outside the ASP program that takes the inputs $Y_1, \dots, Y_n$ and returns the outputs $X_1, \dots, X_m$. This is especially useful for integrating computations of functions in continuous domains into the ASP program.

ASP supports two types of constraints:

- **Hard constraints**, or integrity constraints, are rules of the form

$$\leftarrow \beta_1, \dots, \beta_n, \text{ not } \beta_{n+1}, \dots, \text{ not } \beta_m.$$

Intuitively, this means the body must not hold.

- **Weak constraints** are rules of the form

$$\leftarrow \beta_1, \dots, \beta_n, \text{ not } \beta_{n+1}, \dots, \text{ not } \beta_m. [w@p, t_1, \dots, t_n]$$

Intuitively, this means that we prefer the body not to hold, and we add a cost w at the priority level p when the body holds for the **terms** $t_1, \dots, t_n$. Different weak constraints can have different priority levels, allowing for very powerful multi-objective optimization.

ASP allows **aggregates** to express properties on a specific set of elements. They are expressions of the form

$$s_1 <_1 \alpha\{t_1, \dots, t_n : L_1, \dots, L_m\} <_2 s_2$$

Here t_i are terms, L_i are atoms, α denotes an aggregate function, (e.g., $\#count$, $\#max$, and $\#sum$). $<_1$ and $<_2$ are comparison operators (e.g., $=$, $>$, $<$), and s_1 and s_2 are terms.

Consider for instance the problem of finding the chromatic number of a graph. Recall that the chromatic number of a graph is the minimum number of colors needed to color the vertices such that no two adjacent vertices share the same color. We represent this problem in ASP and present it to clingo (using the notation described in [50]) as follows:

```

1 % Generate
2 { vertexcolor(X,C) : color(C) } = 1 :- vertex(X).
3 % Test
4 :- edge(X,Y), vertexcolor(X,C), vertexcolor(Y,C).
5 % Optimize
6 :~ vertexcolor(X,C). [1@1,C]
7 % Display
8 #show vertexcolor/2.

```

This ASP program consists of four parts: generate (line 2), test (line 4), optimize (line 6), and display (line 8). In line 2, all colorings are generated; in line 4 the colorings where adjacent vertices have the same color are eliminated. Line 6 minimizes the number of different vertex colors available on the graph. Line 8 displays an answer set.

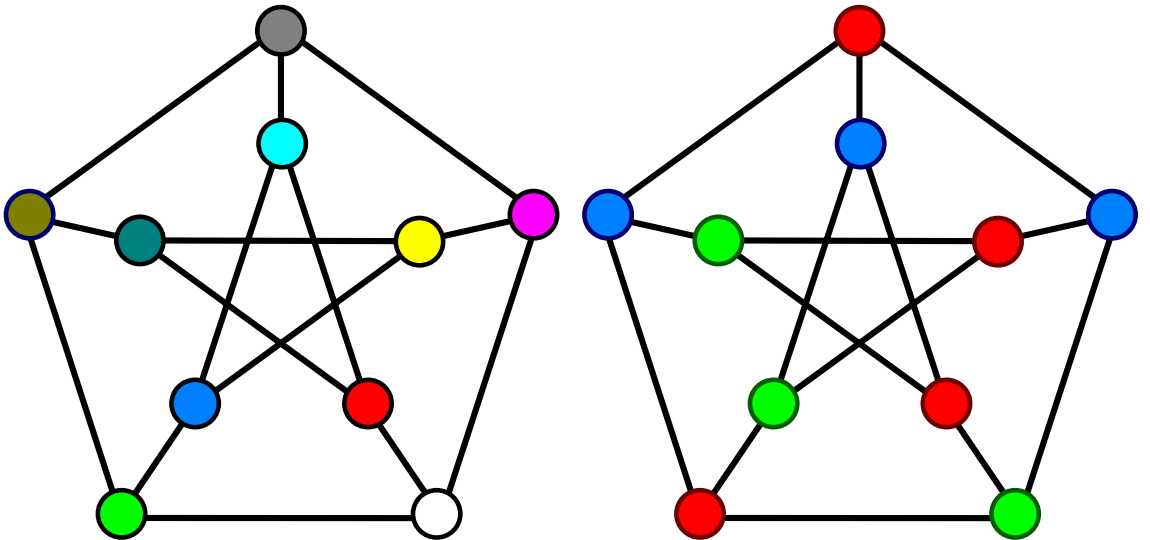


Fig. 3.1. Petersen graph initial coloring. **Fig. 3.2.** Petersen graph final coloring.

We describe a problem instance, like the Petersen graph , as follows:

```

1 % Vertices
2 vertex(1..10).
3 % (Directed) Edges
4 edge(1,(2;4;6)). edge(2,(1;3;7)). edge(3,(2;5;9)).
5 edge(4,(1;8;9)). edge(5,(3;6;8)). edge(6,(1;5;10)).
6 edge(7,(2;8;10)). edge(8,(4;5;7)). edge(9,(3;4;10)).
7 edge(10,(6;7;9)).
8 % Make graph undirected
9 edge(X,Y) :- edge(Y,X).
10 % Colors
11 color(1..10).
```

A solution to this instance can be found by clingo as follows:

```

Answer: 1
vertexcolor(2,8) vertexcolor(1,10) vertexcolor(4,9) vertexcolor(6,9)
vertexcolor(3,10) vertexcolor(7,9) vertexcolor(5,8) vertexcolor(9,8)
vertexcolor(8,10) vertexcolor(10,10)
Optimization: 3
OPTIMUM FOUND
```

This output describes which vertex carries which color, what the chromatic number found at the point of termination was, and whether the global optimum was found.

3.3 Collision Resolution

When simulating objects, it is crucial to handle situations where they collide. The handling of such situations consists of two stages: detecting the collision, i.e., calculating if there exists an overlap of the object pair geometries, and responding to the collisions in a realistic way, i.e., predicting the resulting dynamics of the simulated system after the collision accurately [51].

Much research has been done in this line of work, primarily with interests in realistic computer simulation and animation. We restrict this brief review to the work done on calculating the dynamic response to the collisions of rigid bodies. Most of the literature regarding collision resolution in rigid body systems can be categorised as analytic, penalty or impulse methods [52].

Analytic methods retract the simulation to the point just before any penetrations were detected. Then, for every contact point, constraint equations are generated. Once all the constraint equations have been defined, solving them results in very

accurate reactionary forces and impulses for every contact point [53]. This method is accepted to be the slowest, but most accurate [52], [54].

Penalty-based methods model a virtual spring-damper between every pair of colliding objects at each time-step to penalize the collision [54], then the system dynamics are calculated as a set of optimization problems [52]. This is computationally inexpensive and easy to implement, but requires the setting of the spring stiffness and damping as parameters [52], [54]. This method suffers from lacking a straightforward way of dealing with objects of large inertia coming to contact, which can lead to instability in the simulation due to very large forces being created [52], [54].

Impulse-based methods are some of the earliest [55], and are based on simulating infinite forces occurring in an infinitesimally small time-step between contact points [56], [57]. This method is both computationally inexpensive and accurate [52], [54]. Unlike penalty-based methods, impulse-based methods do not need the specification of stiffness or damping parameters, and unlike both analytic and penalty-based methods, impulse-based methods can handle each collision locally, enabling significant computational advantages through parallelization [52].

Most modern physics engines such as Bullet [58], V-REP [59], ODE [60], and MuJoCo [61], use a combination of these methods intelligently, depending on factors such as the object pair contact type, shapes, velocities, etc. In the context of this thesis, the underlying physics of these methods can be considered as a black-box function that resolves collisions until the system reaches a static equilibrium.

Chapter 4

Placement Generation

As described earlier in section 1.2, the first step of our approach is to generate a goal placement for the rearrangement problem. Two criteria are considered:

- the number and intensity collisions, and
- the number of original objects moved and by how much.

4.1 Problem Definition

The *placement generation* (PG) problem is defined by

- a surface S and its geometric model $g(S)$ that details its size and shape,
- the sets O_O and O_C of non-movable (obstacles) and movable (clutter) objects on the surface S and the sets $g(O_O)$ and $g(O_C)$ of their geometric models,
- the set O_N of new objects to be placed on the surface S and the set $g(O_N)$ of their geometric models,
- a set W of continuous placement constraints on objects (e.g., a monitor may be forced to be in the corner of the table), and
- an initial collision-free configuration C_I of all objects in $O_C \cup O_O$ on the surface S relative to $g(S)$.

A solution for a placement generation problem $\langle g(S), O_O, g(O_O), O_C, g(O_C), O_N, g(O_N), C_I, W \rangle$ is a collision-free final configuration C_F of all objects $O_N \cup O_C$ on the surface S relative to $g(S)$.

Figure 4.1 presents a sample problem instance: initially in Figure 4.1(a), a cylindrical obstacle (yellow), and four geometrically different movable objects (red) are placed

on a surface (green); the goal is to find a collision-free configuration of these objects and some more new objects (blue), like in Figure 4.1(c).

4.2 Methods

We propose a 3-layer deep nested local search algorithm (Algorithm 4) to compute a solution for a placement generation problem. Intuitively; the innermost search applies a collision resolution method using a physics engine, with the goal of minimizing the total penetration depth of objects, the intermediate local search further aims to minimize the number of collisions by allowing re-placements of objects, and the outermost local search further tries to minimize the number and amount of displacements of movable objects with respect to their initial configurations.

4.2.1 Penetration Minimization

The innermost layer is a local search employing a dynamic simulator that uses any of the methods described in section 3.3 as its transition model. The search starts with a configuration $C_{current}$ obtained from C_I by randomly placing the new objects O_N on the surface S , with zero velocities. It searches over configurations of all objects $O_O \cup O_C \cup O_N$ on the surface S , with the goal of minimizing the objective optimization function $cost_p$ defined as the sum of maximum penetration depth [62] of pairs of objects in collisions. The physics engine returns a configuration C_{next} of all objects, minimizing $cost_p(C_{next})$.

In particular, the chosen collision resolution method is applied by passing $C_{current}$ to a physics engine. For each simulation time-step, this typically generates a pair of forces between every object pair in collision along the contact normal until the system converges to a static equilibrium.

In the simulation, we only regard the force component parallel to the surface, constraining objects to moving along the surface plane only. We also employ position

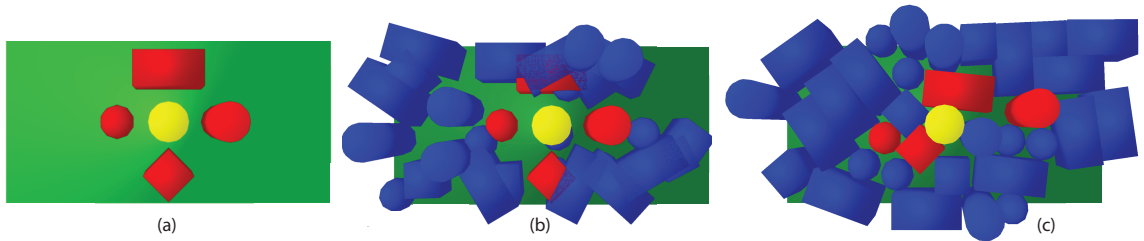


Fig. 4.1. A sample problem instance and a possible solution.

constraints on the centroids of objects, forcing them to be on top of the surface. Additionally, when required, we simulate immovable walls to force all object meshes to be constrained within the surface area, which can be arbitrarily shaped.

For instance, consider the initial configuration C_I of obstacles (yellow) and movable objects (red) shown in Figure 4.1(a). A configuration $C_{current}$ of objects obtained by randomly placing all the new objects (blue) on the surface can be seen in Figure 4.1(b). Note that there are many collisions in $C_{current}$, and objects penetrate with each other. By applying this innermost search, the configuration shown in Figure 4.1(c) may be generated, where the total penetration is zero.

The only action that the physics engine can perform (due to the nature of the collision resolution methods) is to push objects in collision outside of each other; so, for instance, it cannot swap locations of two objects. This may lead to local minima where the objects can no longer be pushed, and there may be still some collisions in C_{next} . This motivates us towards the intermediate local search algorithm that utilizes some heuristics to re-place all the objects in collision.

Algorithm 2 INNERMOSTSEARCH

Input: PG problem $P = \langle g(S), O_O, g(O_O), O_C, g(O_C), O_N, g(O_N), C_I, W \rangle$; placement constraints V ; and, if provided, a configuration $C_{current}$ of all objects $O_O \cup O_C \cup O_N$ on the surface S ($C_{current}$ is obtained from $C_{current}$ in INTERMEDIATE-SEARCH).

Output: A configuration of all objects $O_N \cup O_C$ on the surface S .

```

    //  $cost_p$ : cost function characterizes the total amount of penetration depth of
    // pairs of objects in collisions considering the placement constraints  $V$ .
1:  $C_{current} \leftarrow$  if not provided, generate a configuration of all objects  $O_O \cup O_C \cup O_N$ 
   on the surface  $S$  obtained from  $C_I$  by randomly placing  $O_N$  on  $S$ .
   // Load the problem  $P$  to the dynamic simulator according to  $C_{current}$ .
2: LOAD( $P, C_{current}$ );
   // Call the dynamic simulator to reduce the total penetration depth.
3: loop
4:    $C_{next} \leftarrow$  STEP( $V$ );
5:   if  $cost_p(C_{current}) \leq cost_p(C_{next})$  then
6:     return  $C_{current}$ ;
7:   else
8:      $C_{current} = C_{next}$ ;
9:   end if
10: end loop

```

4.2.2 Collision Minimization

The intermediate local search algorithm (Algorithm 3) utilizes heuristically-guided re-placements to avoid local minima. Intuitively, for each object o in collision,

the heuristic suggests (i) dividing the surface S into cells and (ii) re-initiating the placement of the object o into the center one of the cells that is unoccupied (i.e., has no other object centroids); the heuristics is used to guide generation of new configurations. For (i), the heuristic imposes a grid on the surface S . If a grid cell has no object centroids in it, it is marked as free. If no free cell exists, a new refined grid is imposed on S with smaller but more numerous cells. This refinement process continues until there is at least one free cell in the imposed grid.

The intermediate local search algorithm also starts with a configuration $C_{current}$ of all objects $O_O \cup O_C \cup O_N$ on the surface S obtained from C_I by randomly placing the new objects O_N on the surface S . It calls the innermost search algorithm described above to find a better configuration, but starting with the configurations obtained from $C_{current}$ as suggested by the heuristics. The goal is to minimize the objective optimization function $cost_c$ defined as a tuple $\langle \#col, cost_p \rangle$, where $\#col$ is the total number of collisions and $cost_p$ is the total amount of penetration depth of pairs of objects in collisions. Here, lexicographic ordering is used to find the minimum of two tuples: $\langle a, b \rangle < \langle a', b' \rangle$ if either $(a < a')$ or $(a = a' \text{ and } b < b')$. In this way, priority is given to $\#col$ and then to $cost_p$: among multiple configurations of all objects with the same minimum number of collisions, the configuration C_{next} with the least cumulative penetration depth is returned.

An example is presented in Figure 4.2 to illustrate the usefulness of the heuristics in the intermediate local search algorithm. The search starts with a configuration $C_{current}$, where $\#col(C_{current}) = 24$ and $cost_p(C_{current}) = 1.75$. Note that the innermost search alone cannot find a better configuration with less value of $cost_p$: the physics engine gets stuck, as it can no longer push objects on the right half of the surface. First, the heuristic is utilized to re-place the cyan-colored object, which is in collision with some other object, in $C_{current}$. For that, a grid is imposed over the surface with two free cells, labelled A and B, that do not contain any object centroids. Then, two new configurations, C_A and C_B , are obtained from $C_{current}$ by randomly re-placing the cyan-colored object in A and in B, respectively. Here, $cost_c(C_A) = \langle 19, 1.43 \rangle$ and $cost_c(C_B) = \langle 20, 1.37 \rangle$. At this point, the intermediate local search algorithm calls the innermost search algorithm for each of these two configurations.

The intermediate local search algorithm with heuristically-guided re-placements is useful for minimizing the number of collisions on a surface, but the objects in O_C may end up being displaced and rotated too much with respect to their original configurations in C_I . This is undesirable from the perspective of rearrangement planning, because it will require more number of manipulation actions to rearrange such objects. This motivates us towards the outermost local search algorithm, which

Algorithm 3 INTERMEDIATESEARCH

Input: PG problem $P = \langle g(S), O_O, g(O_O), O_C, g(O_C), O_N, g(O_N), C_I, W \rangle$; placement constraints V ; and, if provided, a configuration $C_{current}$ of all objects $O_O \cup O_C \cup O_N$ on the surface S ($C_{current}$ is obtained from C_I in OUTERMOSTSEARCH)

Output: A configuration of all objects $O_N \cup O_C$ on the surface S .

```
//  $H$ : a set of free cells, suggested re-placements of objects  $o \in O_C$  in collision
//  $cost_c$ : cost function characterizes the total number of collisions ( $\#col$ ) and
// the total amount of penetration depth of pairs of objects in collisions ( $cost_p$ )
// considering the placement constraints  $V$ 
1:  $C_{current} \leftarrow$  if not provided, generate a configuration of all objects  $O_O \cup O_C \cup O_N$ 
   on the surface  $S$  obtained from  $C_I$  by randomly placing the new objects  $O_N$  on
   the surface  $S$ 
   // Call the dynamic simulator to reduce the total penetration depth, and check
   if it also decreases the number of collisions
2:  $C_{next} \leftarrow \text{INNERMOSTSEARCH}(P, V, C_{current})$ ;
3: if  $cost_c(C_{next}) < cost_c(C_{current})$  then
4:    $C_{current} = C_{next}$ ;
5: end if
   // Re-place objects in collisions and call the dynamic simulator until no better
   configuration can be found
6:  $C_{best} = C_{current}$ ;
7: loop
8:    $H \leftarrow$  refine discretization, identify free cells
9:   for  $o \in O_C \cup O_N$  in collision do
10:    for every free cell  $e$  in  $H$  do
11:       $C_o \leftarrow$  re-place  $o$  in  $C_{current}$  in  $e$ 
12:       $C_x \leftarrow \text{INNERMOSTSEARCH}(P, V_o, C_o)$ ;
13:      if  $cost_c(C_x) < cost_c(C_{best})$  then
14:         $C_{best} = C_x$ ;
15:      end if
16:    end for
17:  end for
18:  if  $cost_c(C_{current}) \leq cost_c(C_{best})$  then
19:    return  $C_{current}$ ;
20:  else
21:     $C_{current} = C_{best}$ ;
22:  end if
23: end loop
```

utilizes some constraints on the placements of objects to limit their movements.

4.2.3 Rearrangement Minimization

The outermost local search algorithm (Algorithm 4) utilizes placement constraints to minimize displacements. Intuitively, the amount of displacement of a movable

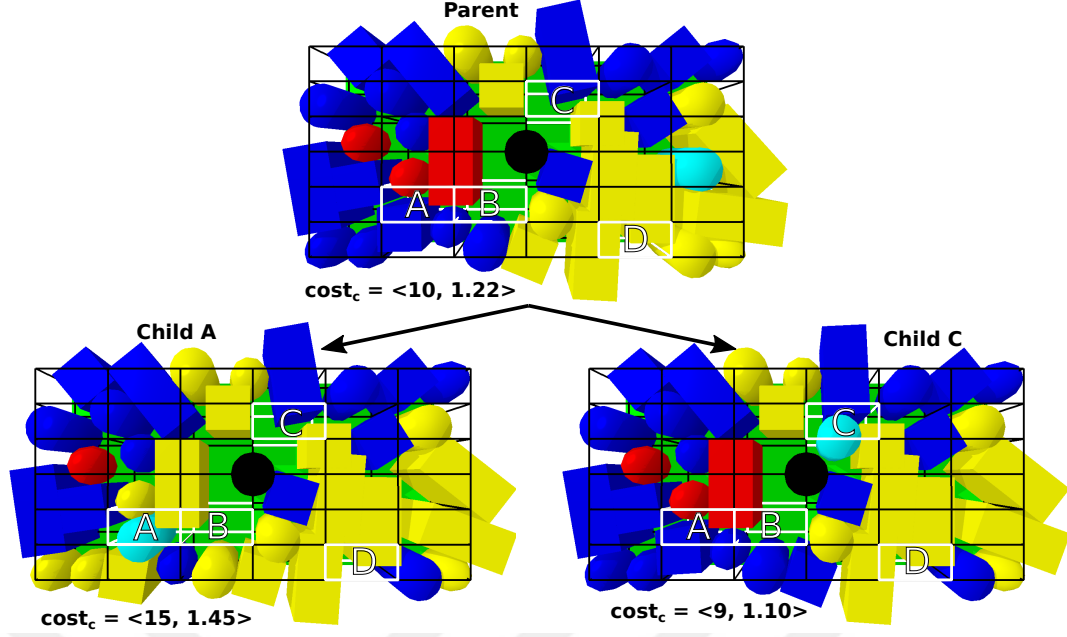


Fig. 4.2. Intermediate local search algorithm

object $o \in O_C$ is constrained to a ball whose centroid is the object's centroid in the given initial configuration in C_I . Initially, $radius_o$ is set to 0; if the outermost local search algorithm cannot find a better configuration under these constraints, then $radius_o$ is increased slightly.

The outermost local search algorithm starts with a configuration $C_{current}$ of all objects $O_O \cup O_C \cup O_N$ on the surface S obtained from C_I by randomly placing the new objects O_N on the surface S . It calls the intermediate local search algorithm described above to find a better configuration, but with respect to the given set V of placement constraints. The goal is to minimize the objective optimization function $cost_r$ defined as a triple $\langle \#col, \#move, cost_d \rangle$, where $\#col$ is the total number of collisions, $\#move$ is the total number of moves of objects in O_C from their original places, and $cost_d$ is the total amount of change in configurations of objects in O_C with respect to C_I . Here, $cost_d$ for a configuration C_x is computed as the sum of the total amount of displacements of objects in O_C (i.e., $\sum_{o \in O_C} dist_o(C_x)$, where $dist_o(C_x)$ is the distance between the centroid of o in C_I and the centroid of o in C_x) and the total amount of change in orientations of objects in O_C (i.e., $\sum_{o \in O_C} arc_o(C_x)$, where $arc_o(C_x)$ is the arc length due to the change of configuration of o from C_I to C_x). The outermost local search algorithm also uses lexicographic ordering to find the minimum of two triples. In this way, priority is given first to $\#col$, then to $\#move$, and then to $cost_d$.

An example is presented in Figure 4.3 to illustrate the usefulness of the placement constraints in the outermost local search algorithm. The search starts with a con-

Algorithm 4 OUTERMOSTSEARCH

Input: PG problem $P = \langle g(S), O_O, g(O_O), O_C, g(O_C), O_N, g(O_N), C_I, W \rangle$.

Output: A configuration of all objects $O_N \cup O_C$ on the surface S .

```
// V: a set of placement constraints for every object  $o \in O_C$ , specifying  $radius_o$ 
// of the balls where  $o$  can be placed in
//  $cost_r$ : cost function characterizes the total amount of changes of object poses
// with respect to  $C_I$ 
1:  $C_{current} \leftarrow$  a configuration of all objects  $O_O \cup O_C \cup O_N$  on the surface  $S$  obtained
   from  $C_I$  by randomly placing the new objects  $O_N$  on the surface  $S$ ;
2:  $V \leftarrow$  for every  $o \in O_C$ ,  $radius_o=0$ 
   // Call the intermediate local search to reduce number of collisions and check if
   it also decreases the total pose changes
3:  $C_{next} \leftarrow \text{INTERMEDIATESEARCH}(P, V, C_{current})$ ;
4: if  $cost_r(C_{next}) < cost_r(C_{current})$  then
5:    $C_{current} = C_{next}$ ;
6: end if
   // Relax the placement constraints and call the intermediate local search until
   no better configuration can be found
7:  $C_{best} = C_{current}$ ;
8: loop
9:   for  $o \in O_C$  where  $radius_o < 1$  do
10:     $radius_o \leftarrow$  increase the radius slightly;
11:     $V_o \leftarrow$  update  $V$  with  $radius_o$ ;
12:     $C_o \leftarrow \text{INTERMEDIATESEARCH}(P, V_o, C_{current})$ ;
13:    if  $cost_r(C_o) < cost_r(C_{best})$  then
14:       $C_{best} = C_o$ ;
15:    end if
16:   end for
17:   if  $cost_r(C_{current}) \leq cost_r(C_{best})$  then
18:     return  $C_{current}$ ;
19:   else
20:      $C_{current} = C_{best}$ ;
21:   end if
22: end loop
```

figuration $C_{current}$, where $\#col(C_{current}) = 4$, $\#move(C_{current}) = cost_d(C_{current}) = 0$. Initially, the radius of the placement balls for every object that is initially on the table is zero. With these placements constraints, the intermediate local search cannot find a better configuration to optimize $cost_r$. Then, for each object, the outermost local search algorithm relaxes its placement constraints by increasing the radius of its placement ball by a certain percentage, and calls the intermediate local search again. With such relaxed constraints, the intermediate local search algorithm returns better configurations with less costs. For example, when it is allowed to place the cuboid object within a small circle around its initial configuration, the intermediate local search algorithm returns a configuration C_{cuboid} with $cost_r(C_{cuboid}) = \langle 2, 1, 2.82 \rangle$;

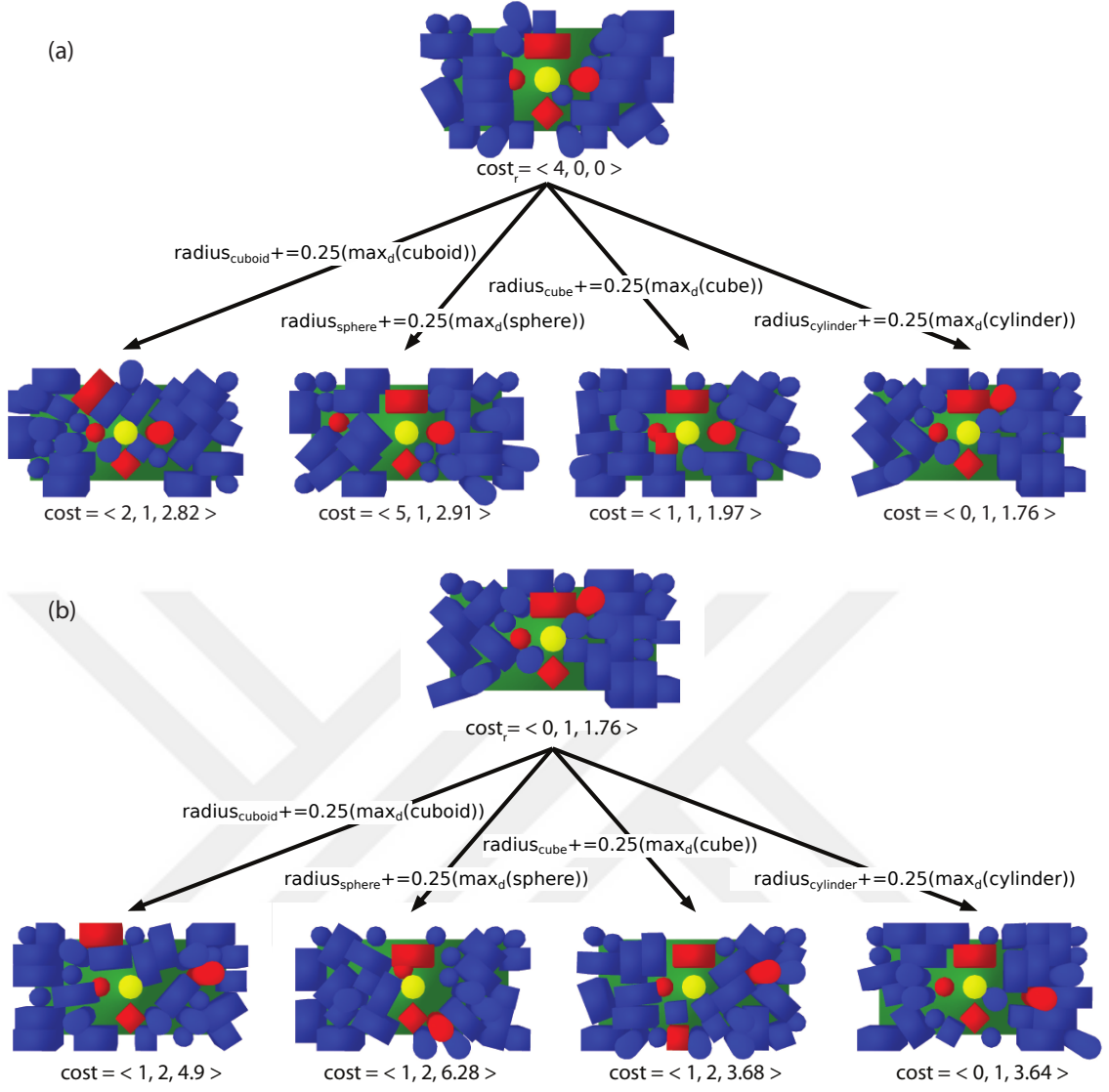


Fig. 4.3. Outermost local search algorithm progression

whereas, when it is allowed to place the cylindrical object by a small amount, it returns a configuration C_{cyl} with $cost_r(C_{cyl}) = \langle 0, 1, 1.76 \rangle$ (Figure 4.3(a)). The outermost local search algorithm continues search from C_{cyl} , with even more relaxed placement constraints, but cannot find a better configuration with less cost; so the outermost search stops (Figure 4.3(b)).

Chapter 5

Rearrangement Planning

In Chapter 4, we discussed methods used to acquire a feasible goal placement, where the configuration is collision-free, and the number of original objects moved, as well as their total displacement, is minimized. In this chapter, we detail the hybrid planning framework used to arrive to a feasible rearrangement plan by which a robot can achieve the goal placement computed earlier.

5.1 Problem Definition

The rearrangement planning (RP) problem is defined by

- a grid of non-uniform cells, where each cell contains at most the centroid of a single object.
- a unique grid cell identifier for each object in $O_C \cup O_N$ w.r.t. C_I and C_F
- a set H of high level task planning constraints on manipulation of objects in the clutter.

A solution to a RP problem is an optimal task plan P^* with the *minimum* number of manipulation actions that rearranges objects from C_I to C_F .

5.2 Methods

To find such a rearrangement plan, we first discretize the continuous domain utilizing the solution provided by our goal placement method. Then, we use the hybrid planning framework of [31], [63] to embed low-level feasibility checks into a high-

level representation of actions, such that a rearrangement plan that is physically feasible can be computed.

5.2.1 Configuration Discretization

We begin the discretization step by pre-processing the solution of placement generation such that the centroid of each object is occupied by a single cell in the grid. The cells of the grid are non-uniform, and it is important to determine the minimum number of cells required to represent such a grid, as this greatly improves the performance of plan computation. The discretization step is performed in two steps: first, a naive discretization is computed by introducing horizontal and vertical lines among object centroids as detailed in Algorithm 5. Second, an optimization is performed on the naive discretization to compute the grid with minimum number of cells. In particular, we define the minimization as a problem declaratively in ASP, and use the efficient ASP solver CLASP to find the global minimum.

Figure 5.1 presents the progression of configuration discretization: (a) shows the solution computed by the placement generation algorithm, (b) shows the grid produced by Algorithm 5, resulting in 729 cells, and (c) shows the grid optimized using ASP, resulting in 64 cells. Note that while there are 25 objects, the discrete domain contains 29 centroids because it considers the final centroid locations of $O_O \cup O_C \cup O_N$, shown in (a), as well as the initial centroid locations of $O_O \cup O_C$, colored yellow in (b) and (c).

While we include O_N in Algorithm 5, it is important to note that for configurations where each object has the same cross-sectional area across the axis perpendicular to the surface, we can completely omit O_N from the discretization and planning

Algorithm 5 NAIVEDISCRETIZER

Input: A configuration C of all objects $O_N \cup O_C \cup O_O$ on the surface S .

Output: Sets of horizontal lines H and vertical lines V , together describing a grid.

```

    // Get ordered lists of the x and y coordinates of  $O_N \cup O_C \cup O_O$ 
1:  $X, Y \leftarrow \text{SORTASCENDING}(\text{GETCENTERSOFMASS}(C));$ 
2:  $H \leftarrow \emptyset;$ 
3: for  $i \in \{0, \dots, |X| - 2\}$  do
4:    $V \leftarrow V \cup \{\frac{1}{2}(X[i] + X[i + 1])\}$ 
5: end for
6:  $V \leftarrow \emptyset;$ 
7: for  $i \in \{0, \dots, |Y| - 2\}$  do
8:    $H \leftarrow H \cup \{\frac{1}{2}(Y[i] + Y[i + 1])\}$ 
9: end for
10: return  $H, V;$ 

```

altogether, since it is sufficient to rearrange the original objects to their goal poses as the new objects can be simply placed to their goal locations after this re-arrangement is performed.

After a naive discretization of the domain, we represent the optimization problem to ASP format by, first, defining the set of horizontal and vertical grid lines, where each line is represented by the atoms $hline(i)$ and $vline(i)$

$$hline(0).hline(m).vline(0).vline(n). \\ \{hline(y) : 0 \leq y \leq m\}. \{vline(x) : 0 \leq x \leq n\}.$$

and where $m = |V|$ and $n = |H|$, with V and H following their definitions in Algorithm 5. Then, the grid cells can be defined as follows

$$cell(x_1, y_1, x_2, y_2) \leftarrow \\ hline(y_1), hline(y_2), vline(x_1), vline(x_2), \\ \{hline(y) : y_1 < y < y_2\}0, \{vline(x) : x_1 < x < x_2\}0.$$

where $0 \leq x_1, x_2 \leq n$ and $0 \leq y_1, y_2 \leq m$. We define object centroids by atoms of the form $obj(x_1, y_1, x_2, y_2)$. Next, we introduce a constraint that prevents more than one distinct object centroid from being located in the same cell:

$$\leftarrow cell(x_1, y_1, x_2, y_2), \\ 2\{obj(x_3, y_3, x_4, y_4) : x_1 \leq x_3, x_2 \geq x_4, y_1 \leq y_3, y_2 \geq y_4\}.$$

Finally, we define weak constraints to minimize the number of grid lines:

$$\sim hline(y).[1@1, y] \\ \sim vline(x).[1@1, x]$$

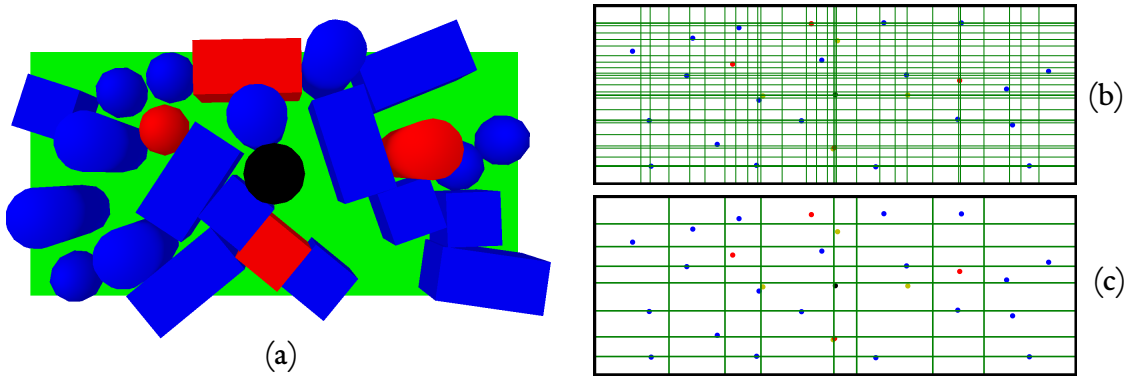


Fig. 5.1. An example of configuration discretization.

5.2.2 High-Level Planning

After an optimal non-uniform grid for discretization of object configurations is found, a rearrangement plan is computed using ASP. The planning domain is described as follows.

The fluent $loc(o, c, t)$ is used to represent the location of the object o to be in the grid cell c at timestep t , and the fluent $moveTo(o, c, t)$ is used to represent the movement of object o from its current grid cell to grid cell c at timestep t .

The preconditions of $moveTo(o, c, t)$ action are defined as follows:

$$\begin{aligned} &\leftarrow moveTo(o, c, t), obj(o), obs(o), cell(c), time(t). \\ &\leftarrow moveTo(o, c, t), obj(o), cell(c), time(t), loc(o, c, t). \\ &\leftarrow moveTo(o, c, t), obj(o), obj(o_1), o \neq o_1, cell(c), time(t), loc(o_1, c, t). \end{aligned}$$

where the first precondition prevents the movement of obstacles ($obs(o)$ indicates that the object o is an obstacle), the second precondition prevents the movement of an object into its current cell, and the third precondition prevents the movement of an object into an occupied cell.

The direct effect of $moveTo(o, c, t)$ action is defined as follows:

$$loc(o, c, t_1) \leftarrow moveTo(o, c, t), obj(o), time(t), t_1 = t + 1, cell(c).$$

This rule describes the direct changes that happen to the state as a result of the $moveTo$ action; in particular, the location of object o becomes cell c in the timestep t_1 , which is the timestep that follows the one the action is executed.

The ramification of $moveTo(o, c, t)$ action is defined as follows:

$$\begin{aligned} &-loc(o, c, t_1) \leftarrow \\ &\quad loc(o, c, t), loc(o, c_1, t_1), \\ &\quad obj(o), cell(c), cell(c_1), c \neq c_1, time(t), t_1 = t + 1. \end{aligned}$$

This describes the indirect changes that occur as a result of the $moveTo$ action; in particular, since the location of object o becomes cell c_1 in the next timestep t_1 , it is explicitly defined as not being in its previous cell c during t_1 .

Finally, since the high-level planner relies on a discretized state representation based on object centroids, it is necessary to make use of an external atom to perform

collision checks of each action of the plan to ensure feasibility.

$$\leftarrow \&isCollision[loc]().$$

The input to the external atom is the *loc* predicate with all its extensions at all timesteps. That is, the external computation inputs the discrete location, in cell index, of every object at each timestep. The external computation returns true if there exists a collision among objects at any timestep – otherwise, it returns false.

This external atom must be implemented carefully to ensure the continuity of the domain, including orientation, when checking for collisions. The next Subsection details this feasibility check.

5.2.3 Low-Level Feasibility Check

As explained in Section 3.2, an external atom calls a function that is defined outside the ASP solver. In this case, the *&isCollision* external atom calls a function that first attempts to naively determine feasibility by, for every step of the plan, completely constraining the non-moving objects between the two configurations before and after the step, and constraining the moving object of the step to within its grid cells. Then, Algorithm 3 is called to find a solution for this step with these constraints. This process is repeated for each step, using information from the previous step to retain continuity, except for the final, since it is already collision-free.

If a solution for an earlier step prevents a solution from being found for a later step, this naive method fails, making it especially susceptible to non-monotone instances.

Algorithm 6 ISCOLLISION

Input: *loc*, the sequence of discrete configurations queried by the planner.

Output: success or failure.

```

    // D: the configuration discretization
    // P: the placement generation problem
    // CI: the initial configuration.
    // CG: the goal configuration
1: load P, D, CI, and CG from memory
2:  $E_{feasible} \leftarrow \text{FEASIBILITYSEARCH}(loc, P, D, C_I, C_G)$ 
3: if  $cost_f(E_{feasible}) = \langle 0, 0, 0 \rangle$  then
4:     save plan based on  $E_{feasible}$  to memory
5:     return success
6: else
7:     return failure
8: end if

```

This is similar to the implementation described in [1], where continuous information between each consecutive pair of steps is not retained. Instead each step is checked for independently, which necessitates succeeding the hybrid planning step with a layer to adjust the plan steps to overall plan consistency. Note that this layer may fail to find a consistent plan, which will require calling the hybrid planner again with new task-level constraints.

We propose retaining the continuous information when checking between each consecutive pair of steps by checking the plan in its entirety, rather than one step at a time. When the naive method fails, we propose using a local search based on the placement generation intermediate layer to act as a sophisticated feasibility checker.

Algorithm 7 FEASIBILITYSEARCH

Input: loc , the sequence of discrete configurations queried by the planner; D , the configuration discretization; P , the placement problem; C_I and C_G , the initial and goal configurations.

Output: A sequence of configurations and their constraints

```

1:  $E_{current} \leftarrow \text{CREATESTATE}(loc, P, D, C_I, C_G)$ 
2:  $E_{best} \leftarrow E_{current}$ 
3: loop
4:   for  $\langle C_i, V_i \rangle \in E_{current}$  do
5:      $C_o \leftarrow \text{INTERMEDIATESEARCH}(P, V_i, C_i)$ 
6:      $E_o \leftarrow \text{PROPAGATE}(E_{current}, C_o)$ 
7:     if  $\text{cost}_f(E_o) < \text{cost}_f(E_{best})$  then
8:        $E_{best} = E_o$ ;
9:     end if
10:  end for
11:  if  $\text{cost}_f(E_{current}) \leq \text{cost}_f(E_{best})$  then
12:    return  $E_{current}$ ;
13:  else
14:     $E_{current} = E_{best}$ ;
15:  end if
16: end loop

```

Consider the definitions established earlier in Chapter 4.1 and Section 5.1. We define the state for our sophisticated feasibility checker as the n-tuple

$$E = \langle \langle C_0, V_0 \rangle, \langle C_1, V_1 \rangle, \dots, \langle C_n, V_n \rangle \rangle$$

where $n = |P^*| + 1$ is the number of configurations that will be seen while executing plan P^* . Then we can define the cost as

$$\text{cost}_f(E) = \left\langle \sum_{i=1}^n \text{col}(C_i), \sum_{i=1}^n \# \text{col}(C_i), \sum_{i=1}^n \text{cost}_p(C_i) \right\rangle$$

where $col(C_i)$ returns 1 or 0 describing the presence or absence of a collision in C_i .

The search begins with an initial state E_0 , generated using the naive method described earlier. The constraints $V_0, \dots, V_n$ for each configuration are defined such that the objects that are in their initial or goal grid cells are completely constrained to their initial or goal continuous poses, respectively, and objects in buffer cells are constrained to their corresponding grid cells.

The transition between a state E_i and its successor E_{i+1} is detailed in Algorithm 7, which uses Algorithm 3 to solve collisions in a configuration C_j with constraints V_j . The pose changes resulting from this are then propagated across all other configurations $\{C_0, \dots, C_{j-1}\} \cup \{C_{j+1}, \dots, C_n\}$, as detailed in Algorithm 8.

Algorithm 8 PROPAGATE

Input: E_{curr} , the current sequence of configurations and their constraints; C_{ref} , the configuration we would like to propagate and its constraints V_{ref} .

Output: A new sequence of configurations and their constraints, where the effect of modifying C_{ref} is propagated.

```

1:  $O_M^{ref} \leftarrow O_C \cup O_N$  from  $C_{ref}$ 
2: for  $o \in O_M^{ref}$  do
  // Forward propagation
3:   for  $i \in \langle j, j+1, \dots, |E_{curr}| \rangle$  do
    // Acquire the object  $o$  as it is defined in configuration  $C_i$ 
4:      $b \leftarrow getObject(o, C_i)$ 
    // Propagate the pose if the constraints acting on objects  $o$  and  $b$  are equal.
5:     if  $getConstraint(b, V_i) = getConstraint(o, V_{ref})$  then
6:        $b.pose \leftarrow o.pose$ 
    // Exit the loop once a constraint change is detected – this means the cell
    // definition has changed and we should no longer propagate forward.
7:     else
8:       break
9:     end if
10:  end for
  // Backward propagation
11:  for  $i \in \langle j-1, j-2, \dots, 0 \rangle$  do
12:     $b \leftarrow getObject(o, C_i)$ 
13:    if  $getConstraint(b, V_i) = getConstraint(o, V_{ref})$  then
14:       $b.pose \leftarrow o.pose$ 
15:    else
16:      break
17:    end if
18:  end for
19: end for
20: return  $E_{curr}$ 

```

Chapter 6

Experimental Evaluation

6.1 Placement Generation

The success the methods described in Chapter 4 is evaluated with quantitative and qualitative assessments. Quantitatively, we attempt to get an understanding of the scalability of the algorithms by solving problems with varying numbers of obstacle, original, and new objects. Qualitatively, we test our method’s ability to solve particularly difficult instances with non-convex objects and highly confined spaces, some introduced in the literature and others introduced in this thesis.

We have conducted three sets of experiments to evaluate and compare performance of each search level of our goal generation algorithm. All simulations were executed on workstation with an Intel Xeon W-2155 CPU running at 3.30 GHz using a single thread and 32 GB RAM. All algorithms were implemented in Python, with Bullet [58] as the back-end physics engine. Each instance of each experiment was run 60 times to allow for averaging of the results. A timeout of 300 s per trial was imposed.

Experiment 1 started with the initial configuration depicted in Figure 4.1(a), where 4 objects and an obstacle were on a surface. The number of new objects to be added to the table was the control variable in this condition. The number of new objects was increased from 4 to 36, such that the total footprint area covered by all objects and the obstacle were gradually increased up to 95% of the total surface area.

Experiment 2 involved an obstacle and 4 new objects to be placed on the table. The number of movable objects that were initially on the table was the control variable in this condition. The number of movable objects on the table was increased from 4 to 36, such that the total footprint area covered by all objects and the obstacle were gradually increased up to 95% of the total surface area.

Experiment 3 involved 4 initial objects randomly placed on the surface in collision-free configurations and 4 new objects to be added to the surface. The number of obstacles on the table was the control variable in this condition. The number of obstacles was increased from 4 to 32, such that the total footprint area covered by all objects and obstacles was gradually increased up to 95% of the total surface area.

Six algorithms were compared, where three of them correspond to the different levels of our nested local search:

- innermost search (*Inner*) based on collision resolution methods,
- intermediate local search (*Intermediate*) wrapped around *Inner*, and
- the outermost local search (*Outer*) wrapped around *Intermediate*.

Remaining three algorithms are taken as baselines to demonstrate effectiveness of our approach with respect to naive implementations and earlier methods presented in the literature. In particular,

- to highlight the benefits of using the innermost search, we have tested *Random Sample* approach, which randomly samples new configurations for objects with uniform distribution until a collision-free placement is found.
- To highlight the benefits of using our intermediate local search, we have tested *Random Restart* which is implemented as the innermost search *Inner* with random restarts.
- To highlight the differences between the placement generation algorithm introduced in this thesis and the one introduced in [1], we have evaluated its performance under the label *Havur*.

The performance of the algorithms were compared based on three metrics:

- efficiency, measured by the average CPU time spend to calculate a solution,
- quality, measured by the average number of objects moved that were initially on the table and their total movement, and
- success rate, measured by percentage of trials that converge to a collision-free final configuration before a time-out is reached.

Efficiency and quality metrics are computed for partial solutions of unsuccessful trials that return configurations with collision(s).

Figures 6.1–6.3 graphically summarizes the data collected from Experiments 1–3, respectively. In each figure, the left column represent the quality of the solution,

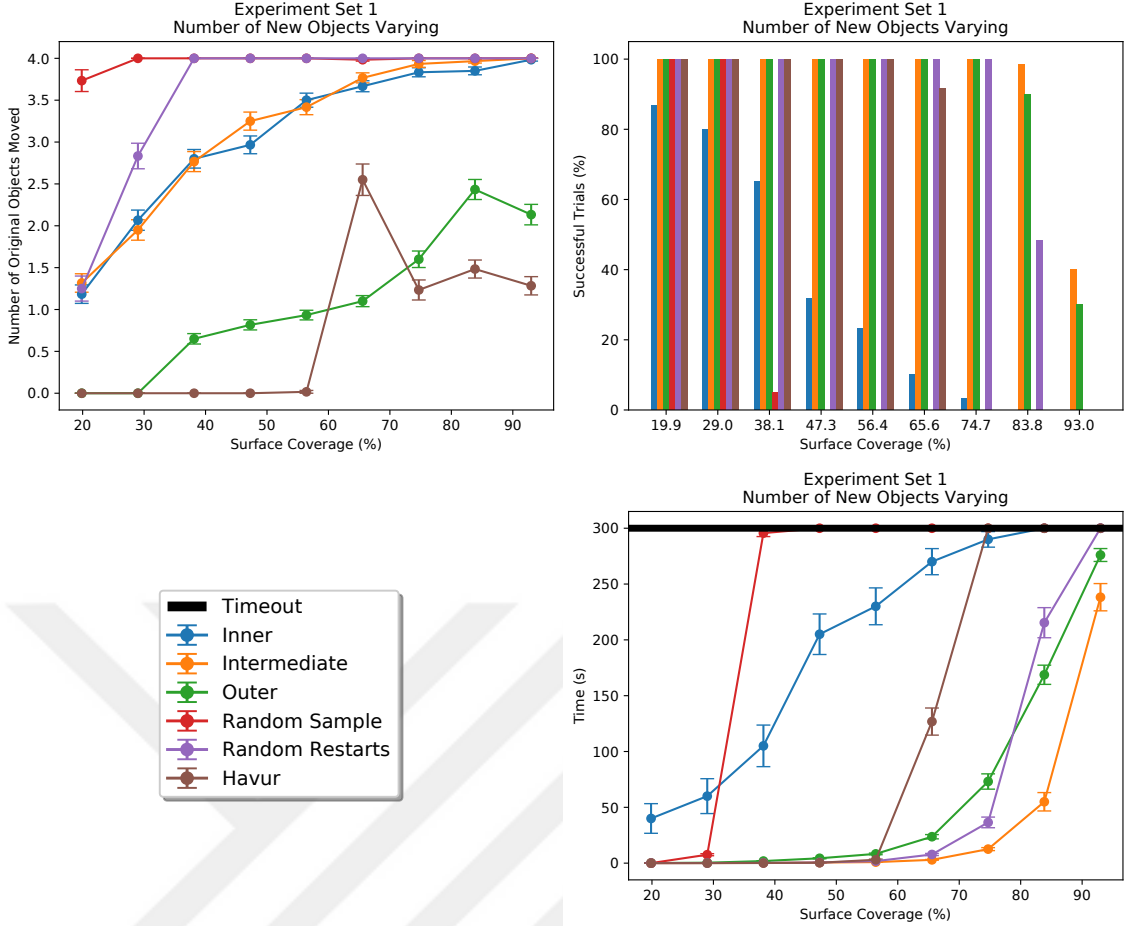


Fig. 6.1. Plots summarizing the results of Experiment 1.

while the right column shows the CPU time and success rate metrics. These results are also detailed in Table A.1.

We can observe the effect of increasing the number of new objects to be put on the table from Figure 6.1.

- In terms of average CPU time, *Inner* outperforms *Random Sample*, while *Intermediate* consistently outperforms all other algorithms, including *Random Restart*. While *Outer* is in general slower than *Random Restart* for problems with surface coverage less than 75%, *Outer* outperforms *Random Restart* in highly cluttered environments, since it relies on *Intermediate* when stuck at local minima. *Havur* is observed to perform very well up until about 60% surface coverage, after which it is outperformed by even the *Random Restart* baseline.
- In terms of solution quality, *Inner* and *Intermediate* perform quite similarly and move more objects as the table gets more cluttered, while *Outer* performs significantly better than both. In particular, up to 30% surface coverage, *Outer* could find solutions that do not require any movements of the objects

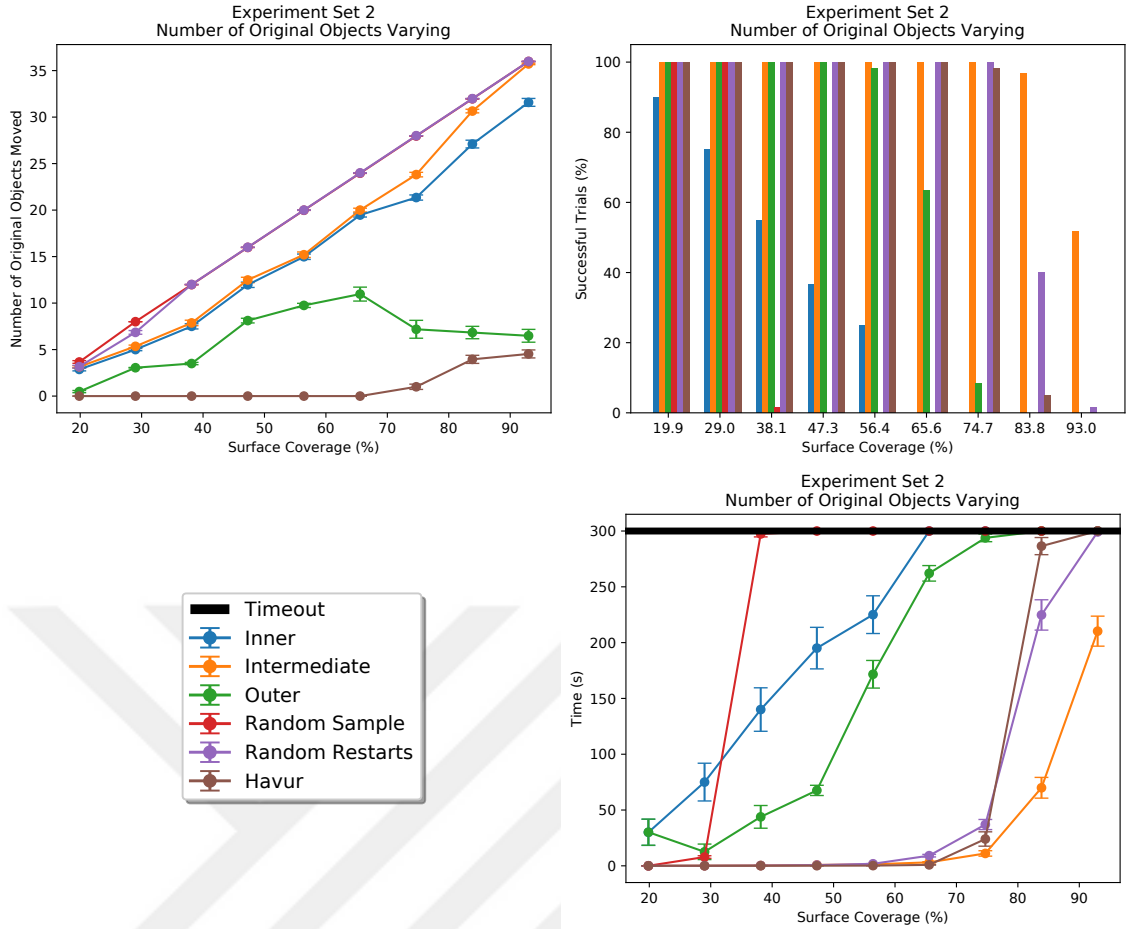


Fig. 6.2. Plots summarizing the results of Experiment 2.

initially on the table. As the clutter increases, it becomes necessary to move some of these objects, but the number of moved objects stays significantly less than those in the solutions computed by *Inner* and *Intermediate*. It is important to note that the baseline algorithms, *Random Sample* and *Random Restart*, consistently result in low solution quality, even when compared to algorithms that do not distinguish between original and new objects, such as *Intermediate*. *Havur* produces marginally better quality solutions than *Outer* until 60% surface coverage, after which the quality becomes worse.

- In terms of success rate, we note that *Intermediate* and *Outer* can solve almost all problems up to 85%, and are the only algorithms capable of solving problems more cluttered than 85%, although the success rate drops to about 40%. While *Random Sample* outperforms *Inner* for simple problems where the surface coverage is less than 30%, *Inner* can solve some problems of up to 75% surface coverage, unlike *Havur* which fails completely after 65%, while *Random Sample* cannot solve any problems that has more than 40% surface coverage.

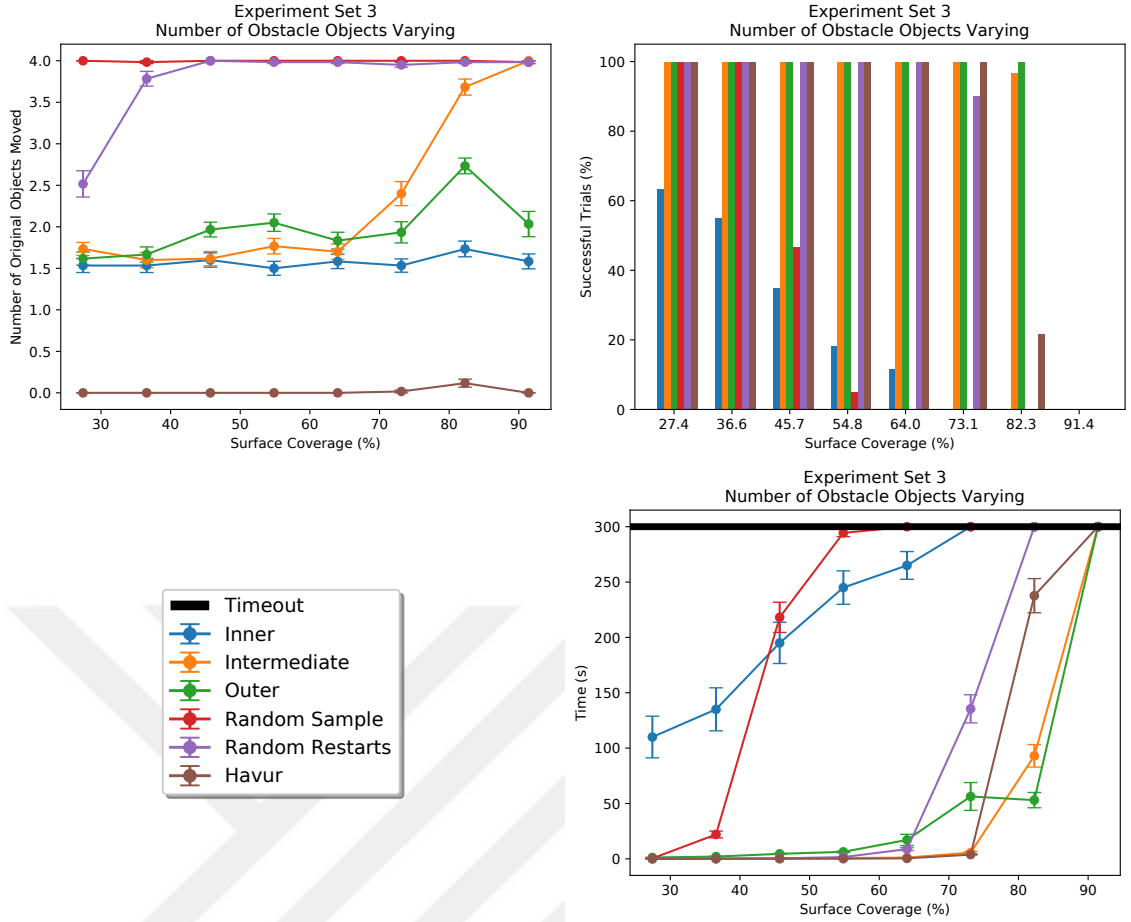


Fig. 6.3. Plots summarizing the results of Experiment 3.

We can observe the effect of increasing the number of original objects on the table from Figure 6.2. Note that these objects can be moved but are desired not to be relocated; hence, the quality of solutions becomes more emphasized in these experiments.

- In terms of average CPU time, once again, *Inner* outperforms the *Random Sample*, and *Intermediate* outperforms *Random Restart*. From these experiments, we can observe how increasing the number of original objects affects *Outer*, causing it to have a sharp increase in computation time at about 50% surface coverage. *Havur* has a time efficiency profile similar to *Random Restart*, and is outperformed by *Intermediate*.
- The trend observed from the first set of experiments regarding solution quality can again be seen here. As *Outer* and *Havur* are the only two algorithms that attempt to improve solution quality, they consistently move fewer original objects across all surface coverage levels.
- The success rate of *Outer* varies over surface coverage levels, but is always worse than *Intermediate* and *Havur*. In particular, given that *Outer* aims to

solve a more constrained version of the problem solved by *Intermediate*, it is not surprising that it fails more often. In this experiment set, *Intermediate* is the only algorithm that is capable of solving a significant portion of problems with 95% surface coverage.

We can observe the effect of increasing the number of obstacles on the table from Figure 6.3. Note that since obstacles cannot be moved, these instances prove to be much harder than the previous experiments, as the percentage of surface coverage increases.

- In terms of average CPU time, *Random Restart*, *Intermediate*, and *Outer* perform similarly up to 65% clutter, after which *Random Restart* baseline falls behind in computation time. The computation time for *Outer*, *Havur*, and *Intermediate* display great similarity until about 85%, after which no algorithm can solve even a single instance. Note that as the obstacles increase, the problem become highly constrained and objects initially located on the table are trapped; hence, object movements are much less in this experiment.
- In terms of solution quality, *Random Sample* and *Random Restart* baselines perform quite similarly and move almost all objects initially on the table, while *Outer* and *Intermediate* perform slightly better than both baselines. *Inner* and *Havur* perform the best in terms of solution quality, but this is mainly due to the constrained nature of the problem mentioned earlier.
- *Inner* and *Intermediate* display significantly different trends in terms of success rate. In particular *Inner* starts failing quite early and can only solve about 30% of instances at 55% clutter, while *Intermediate*, *Havur*, and *Outer* consistently find over 80% of the solutions up to 85% clutter, after which all approaches fail.

The results of these experiments indicate the usefulness of all three nested local searches proposed by our method. *Outer* significantly improves solution quality by minimizing movements of objects on the table, *Intermediate* improves success rate by allowing replacements when stuck in local optima, and *Inner* improves the CPU time over *Random Sampling* especially for cluttered scenarios. It is important to highlight that *Havur* fails in all experiment sets once the surface coverage goes over 80%, and it is especially susceptible to failure if the number of new objects to be placed is high. The reason for both these failures is that *Havur* places objects sequentially, which can constrain the algorithm’s ability once the surface coverage and the number of new objects is too high.

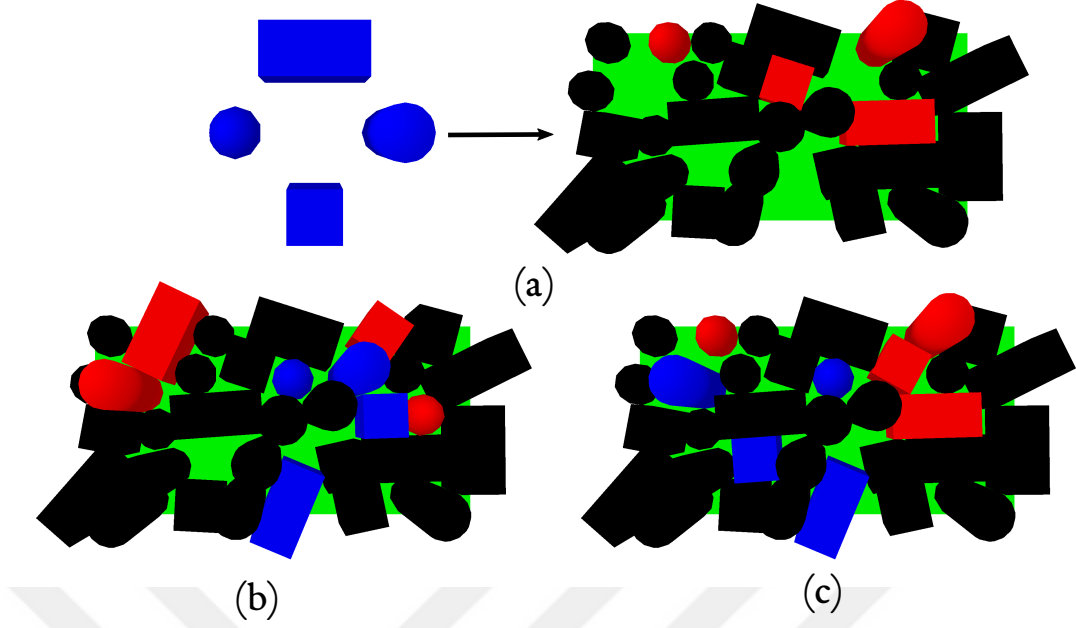


Fig. 6.4. Confined Placement Scenario.

6.1.1 Benchmark Instances

To demonstrate the ability of our local search approach to solve a large variety of placement problems, we have also tested it with several difficult benchmark scenarios. These benchmarks have been engineered to result in difficult instances, by introduction of non-convex objects, confined surfaces, and very specific configurations that result in feasible placements.

Confined Placement Scenario

In this scenario, we have 4 original objects, and a very large number of randomly placed obstacles, all clones of the same 4 basic shapes we have seen in Figure 4.1. The goal is to place 4 more of these objects. We show this to demonstrate the ability of our algorithm to handle highly restricted spaces. Figure 6.4 shows our problem with two solutions: one found using *Intermediate*, and the other using *Outer*.

Tight Placement Scenario

We test our algorithms ability to deal with concave shapes with the tight placement scenario introduced in [1] as seen in Figure 6.5. The solution found is similar to the original problem; indeed, the problem's difficulty lies in that all possible solutions would be similar, with the ellipse fitting into the circle's hole and having been pushed

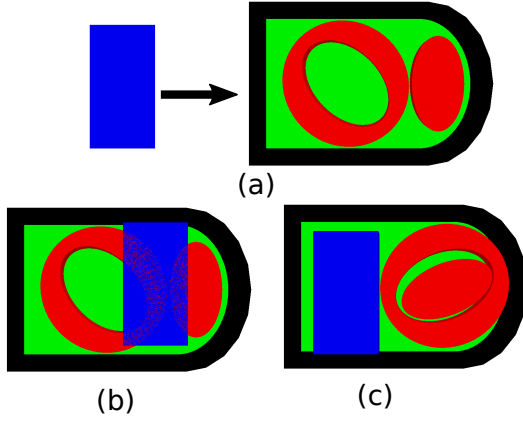


Fig. 6.5. Tight placement scenario.

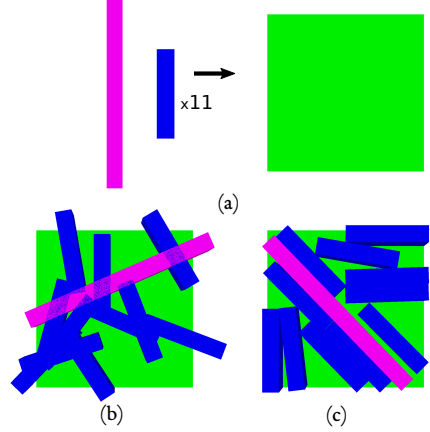


Fig. 6.6. Elongated objects scenario.

to the top of the surface, and the rectangular box lying horizontally at the bottom of the surface. Notably, our algorithm is able to find a solution in a matter of minutes, which is a significant improvement over the previous work.

Elongated Objects Scenario

In the elongated object benchmark, the length of slender objects are designed to pose a challenge, as shown in Figure 6.6(a). In particular, one of the objects is selected to have a length greater than the width of the square surface, while 11 other slender objects are set to have a length that is equal to the half the width of the square surface. This benchmark is difficult, as the placement of the longest object introduces unusable space on the square surface, rendering convergence to a solution significantly more difficult. Figure 6.6(b) depicts a random placement for this problem, demonstrating the unusable area introduced by the long object. Figure 6.6(c) presents the solution computed by our algorithm.

Tight L-Shapes Scenarios

L-shapes are a popular method of demonstrating the capabilities of a placement algorithm. Here we present two variations. In the first, our L-shapes are uniform along the z-axis, and the emphasis is to show that our algorithm is able to handle such a common benchmark. In the second, the uniformity is in the x and y axes, to demonstrate our algorithm's ability to deal with problems in 3D.

These problems are defined such that the total contact area of the L-shapes with the surface is equivalent to the total surface area of the surface, so the range of possible solutions is very limited. Our solutions can be seen in Figures 6.7 and 6.8.

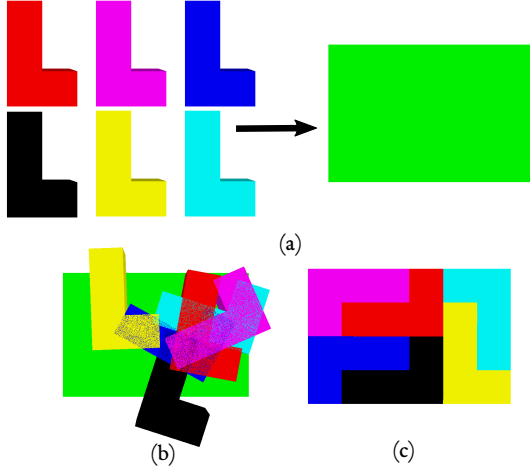


Fig. 6.7. The tight L-shapes scenario with concavity across the x-y plane.

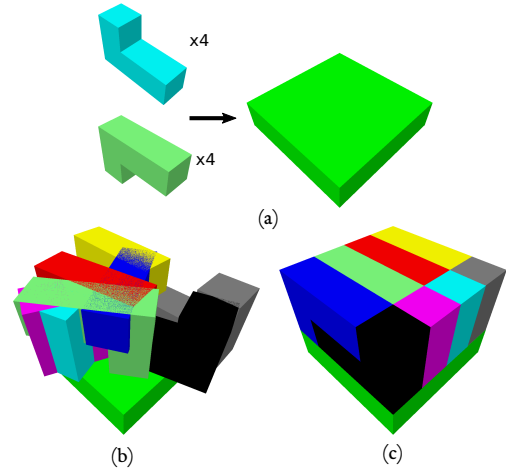


Fig. 6.8. The tight L-shapes scenario with concavity across the z-y plane.

6.2 Rearrangement Planning

The methods described in Chapter 5 are evaluated with quantitative and qualitative assessments. Quantitatively, we analyze the scalability of the planning algorithm by solving problems with varying numbers of original objects on several ranges of surface coverage. In particular, we investigate the effect of using our outermost local search on the planning stage. Qualitatively, we test our method’s ability to solve particularly difficult instances of non-monotone plans that involve multiple enforced swaps.

An Intel Xeon W-2155 CPU running at 3.30 GHz using a single thread and 32 GB RAM was used for this analysis, with all algorithms implemented in Python, and Bullet [58] as the back-end physics engine. For discretization, and high-level planning, CLINGO and DLVHEX, respectively, are used and connected to our Python implementation via external atoms.

To evaluate the effect of minimizing the number of rearranged objects on the surface for the planner, experiments 4-6 are presented. Each instance was run 20 times and no timeout was imposed.

Experiment 4 uses a single obstacle and 12 new objects which are all random clones of the objects shown in Figure 4.1. The number of original objects is controlled and incremented from 1 to 9 to study the performance of the algorithms in environments with low surface coverage.

Experiments 5 and 6 use the same setting as Experiment 4, but utilize 20 and 28 new objects to study the performance of the algorithms in environments with moderate and high surface coverage, respectively.

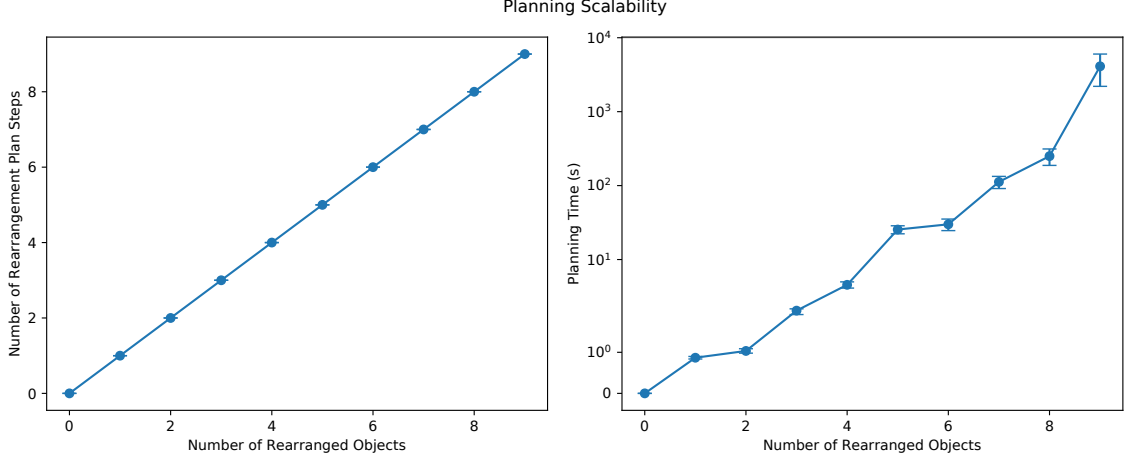


Fig. 6.9. Plots summarizing the planning scalability based on Experiments 4-6.

To evaluate the difference in performance between using interleaved and disconnected feasibility checks for hybrid planning, experiment 7 is presented. Experiment 7 uses a set of instances with enforced swaps in tight spaces to force the feasibility checker to activate. Each instance was run 60 times and no timeout was imposed.

Two placement algorithms are compared, corresponding to different levels of the overall placement generation nested local search:

- intermediate local search (*Intermediate*) wrapped around *Inner*, and
- the outermost local search (*Outer*) wrapped around *Intermediate*.

The performance of hybrid planning method based on the solutions produced by these algorithms as compared based on two metrics:

- computational efficiency, measured by the mean CPU time spent calculating a solution, and
- quality, measured by the number of steps in the resulting minimal plan.

Comparing the effect of using *Intermediate* and *Outer* on the resulting planning process is the main focus of these experiment sets. Figures 6.10-6.12 show the results of Experiments 4-6, respectively, where the mean plan length, placement time, planning time, and the total time are illustrated. Note that the mean plan length includes the number of placement steps, and not just the rearrangement steps. These results are further detailed in Table A.2.

The overall planner performance in terms of computational efficiency and solution length can be seen in Figure 6.9:

- The planner is able to consistently find monotone (and therefore minimal) plans. This is mostly due to the similarity in sizes of the objects being consid-

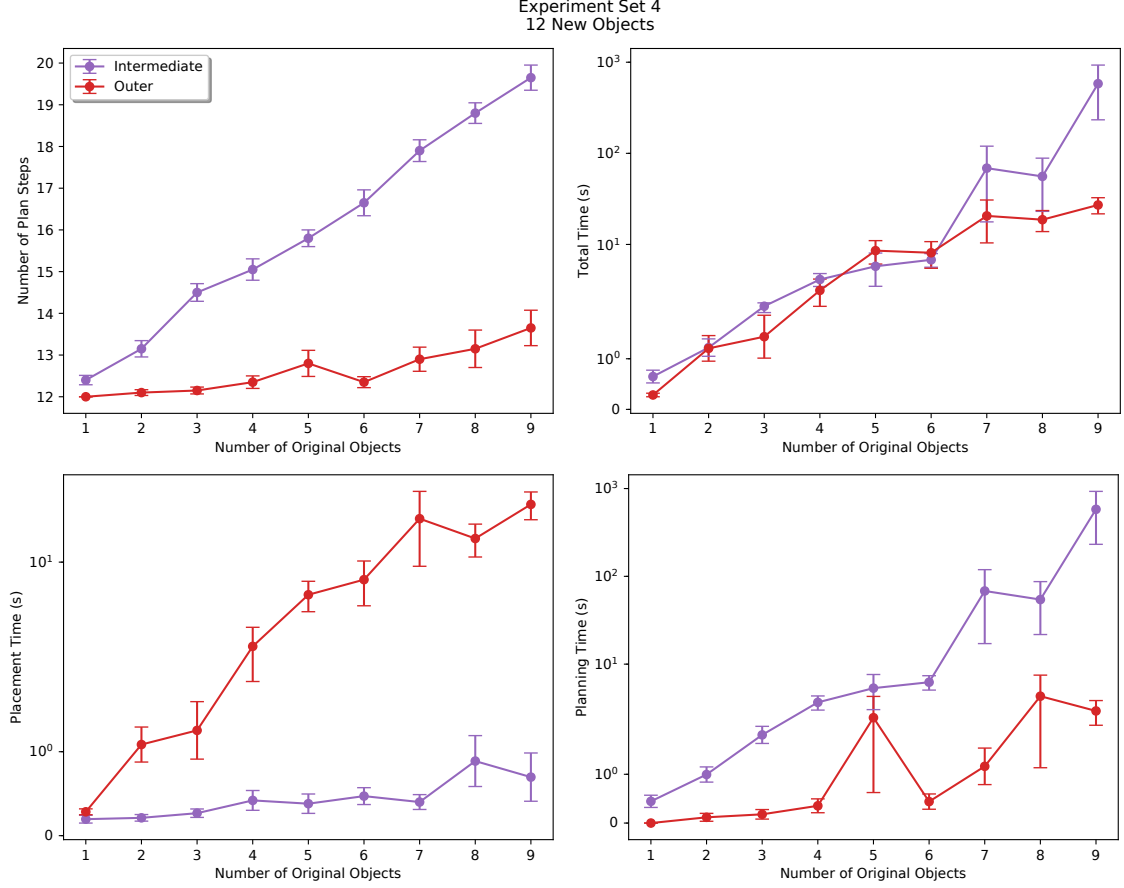


Fig. 6.10. Plots summarizing the results of Experiment 4.

ered, but can also be partially attributed to using collision resolution methods for placement generation; since objects are pushed out of collision to find a solution, a swap between similarly-sized is very rarely necessitated.

- The planning time increases exponentially with the number of rearranged objects, which reinforces our motivation to minimize the number of rearranged objects in the placement stage.

The effect of using *Intermediate* and *Outer* to generate a goal placement for configurations with a low surface coverage (approx. 30% - 50%) is illustrated in Figure 6.10:

- There is no clear difference in terms of the overall computational efficiency between *Intermediate* or *Outer* for simpler problems where the number of original objects is less than 7. However, it can be argued that given the difference in the plan lengths, the overall efficiency of *Outer* is higher due to the robot execution time.
- When the number of original objects is greater than 6, *Intermediate* shows better computational efficiency in computing placement solutions, but *Outer* outperforms it in terms of the overall efficiency by reducing the number of

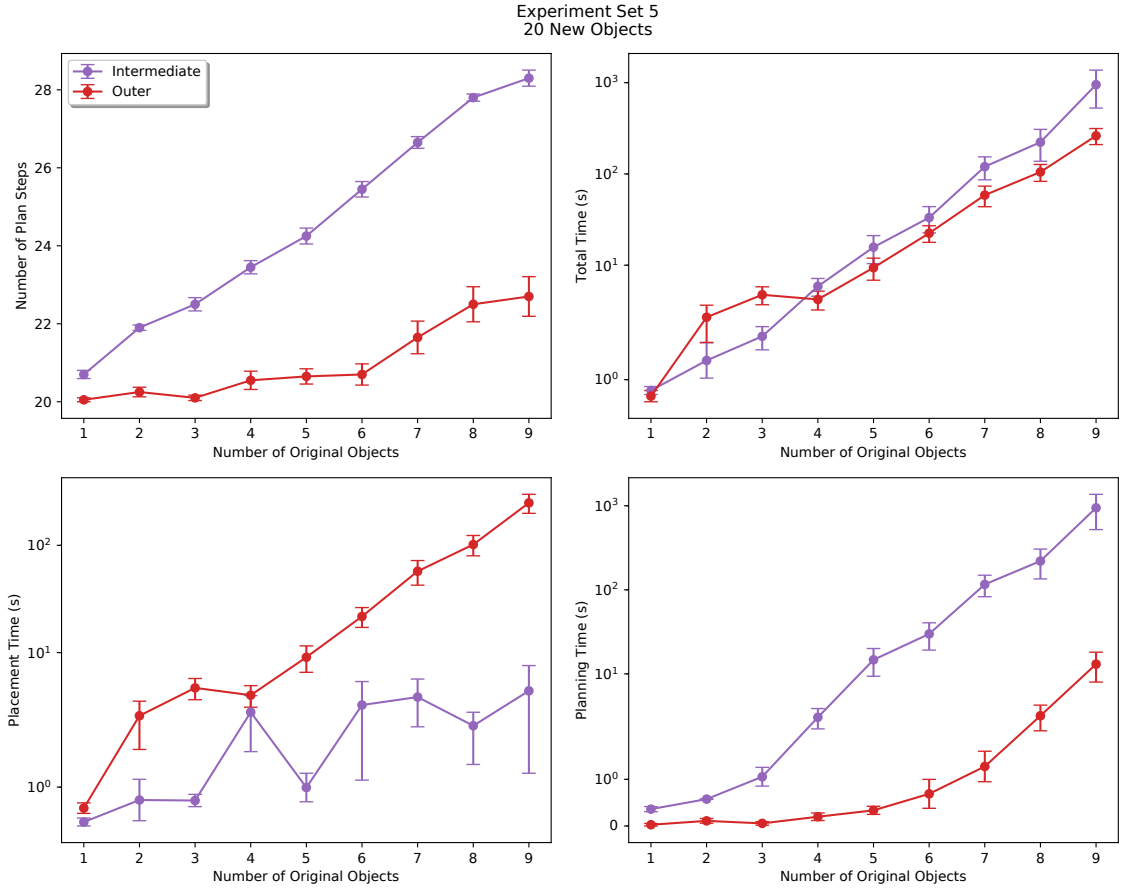


Fig. 6.11. Plots summarizing the results of Experiment 5.

rearranged objects significantly, and thus giving the planner a simpler problem.

- The number of plan steps required to compute a solution using *Intermediate* increases almost linearly with respect to the original objects, whereas the solution computed using *Outer* results in much shorter plans.

The effect of using *Intermediate* and *Outer* to generate a goal placement for configurations with a moderate surface coverage (approx. 50% - 70%) is illustrated in Figure 6.11:

- Using *Intermediate* to compute a placement solution for problems with a small number of original objects (< 4) is somewhat advantageous, but the difference might prove too little to mitigate the amount of time the robot would spend executing the plan.
- When the number of original objects is greater than 3, using *Outer* for placement generation results in approximately 1.5 times improvement in the planning time, but the overall efficiency improvement is more modest.
- As expected, the number of plan steps is consistently lower when using the solution provided by *Outer*, which is significant in terms of place execution.

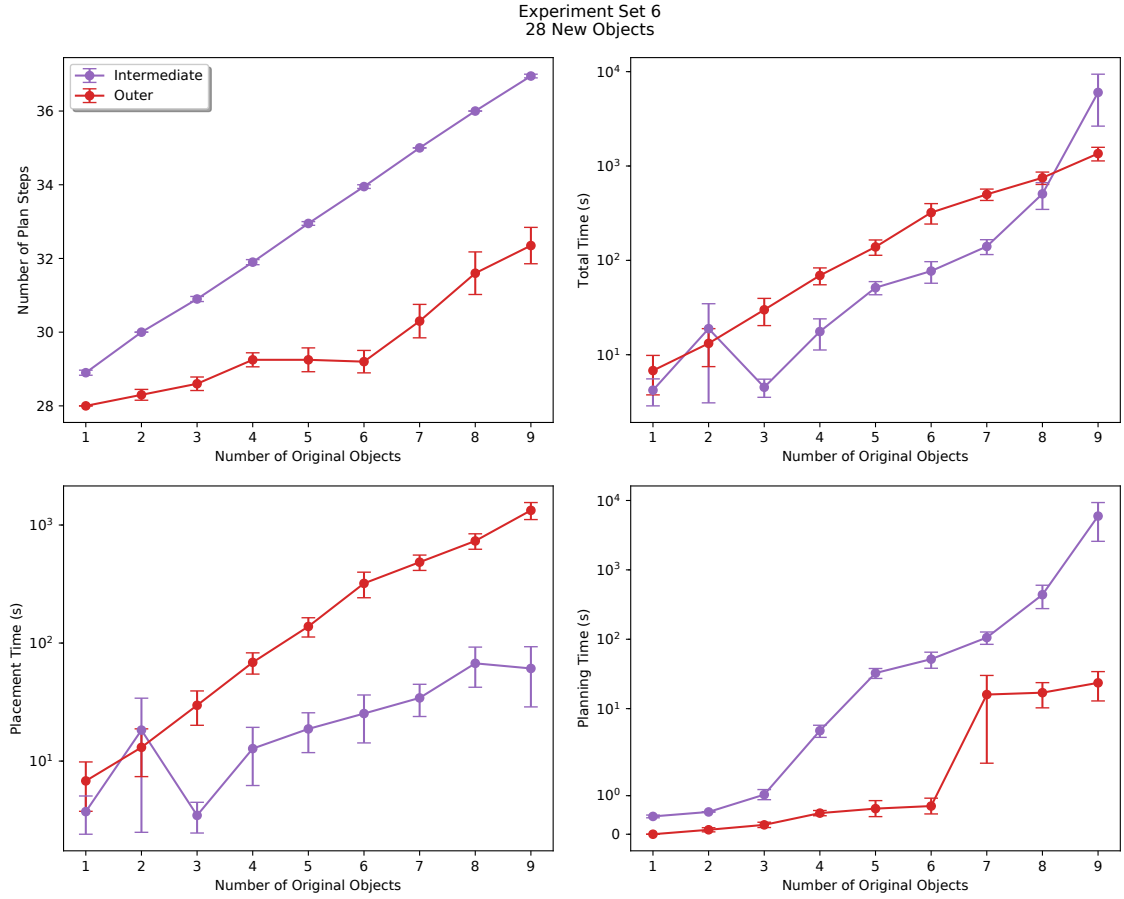


Fig. 6.12. Plots summarizing the results of Experiment 6.

The effect of using *Intermediate* and *Outer* to generate a goal placement for configurations with a high surface coverage (approx. 70% - 85%) is illustrated in Figure 6.12:

- Unlike the previous cases, *Intermediate* shows a significantly better overall efficiency when compared to *Outer*, except for the case where there are 9 original objects.
- Even with a high surface coverage rate, *Outer* is able to reduce the number of moved objects, and therefore the number of plan steps, significantly.
- In such highly cluttered environments, the difference between placement times of *Outer* and *Intermediate* is approximately an order of magnitude, and the difference in planning time is not enough to consistently offset this difference. Hence, selection of the proper approach can be decided based on the potential gain, during execution of the plan.

6.2.1 Benchmark Instances

Experiments 4-6 demonstrate the ability of our planner to solve monotone planning instances. To demonstrate its ability with non-monotone problems, we have also tested it with several difficult benchmark scenarios based on enforced swaps. Importantly, the planner does not use explicitly defined buffer poses, which is how planners in [11]–[15] relax the main challenge of this problem. Instead, our planner suggests regions of placement, based on the minimal discretization, and the low-level feasibility checker locally searches within those regions for appropriate buffer poses.

Enforced Swaps Scenario

The enforced swap benchmark is used to show the ability of the planner to handle non-monotone planning instances with multiple swaps, which require moving multiple objects to buffer poses. Note that, since the planner aims to optimize the number of plan steps, it may require the use of multiple buffer poses, which would be difficult to define before running the experiment. In this benchmark, shown in Figure 6.13, we demonstrate our planner’s ability to compute and utilize multiple buffer poses for multiple swaps.

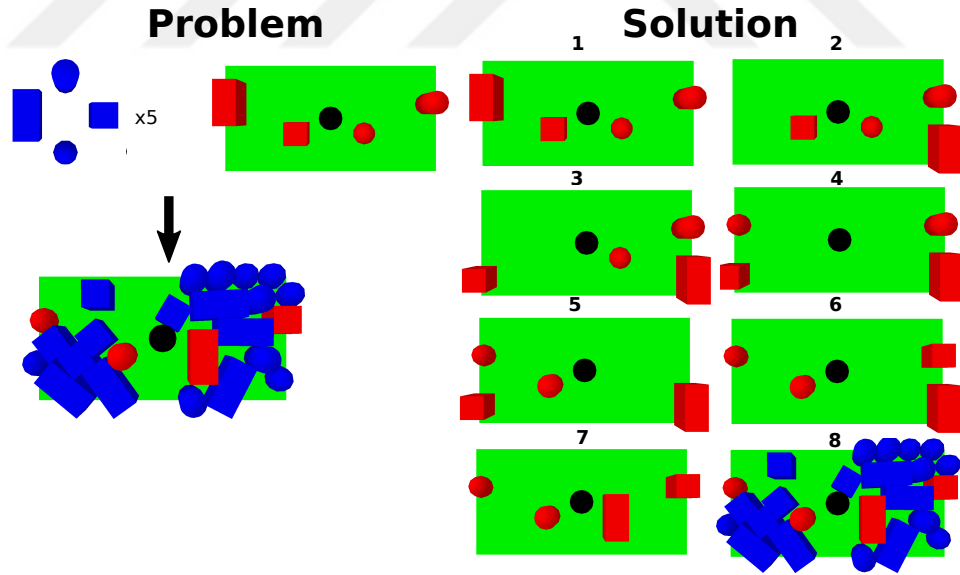


Fig. 6.13. Enforced Swaps Scenario.

Tight Placement Scenario

The tight placement benchmark introduced in [1] is used to show the ability of the planner to handle non-monotone plans in very confined spaces. Note that in this benchmark, even defining the buffer poses beforehand manually is not a trivial task.

In this benchmark, shown in Figure 6.14, we demonstrate our planner’s ability to compute buffer poses in highly constrained continuous spaces.

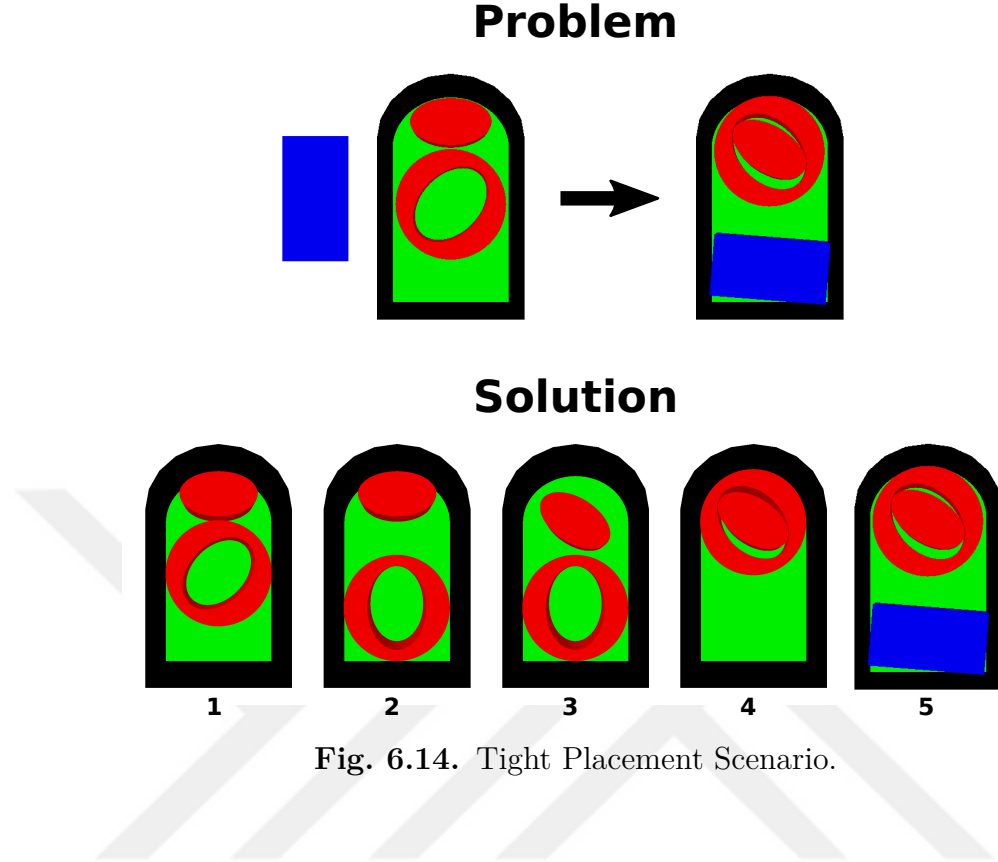


Fig. 6.14. Tight Placement Scenario.

6.2.2 Comparison of Hybrid Planning Approaches

Given that our method embeds all the feasibility checks into the high-level planner directly using external atoms (*interleaved computation*), it is expected to be more efficient in terms of computation time than the methods introduced in [1], which relies on *guided replanning*.

To that end, we introduce several benchmarks as in Figure 6.15 that specifically target the feasibility checker. In particular, we introduced low-dimensional problems from the high-level planner’s perspective, but require the objects to compute feasible solutions.

In Figure 6.15, the red objects are movable, the black objects are immovable, and all objects are allowed to overhang from the surface, provided their center of mass lies on top of the surface. In all of these instances, our rearrangement planning method is evaluated to be faster by a significant margin ranging from 10%-50%. which is consistent with the observations in [64].

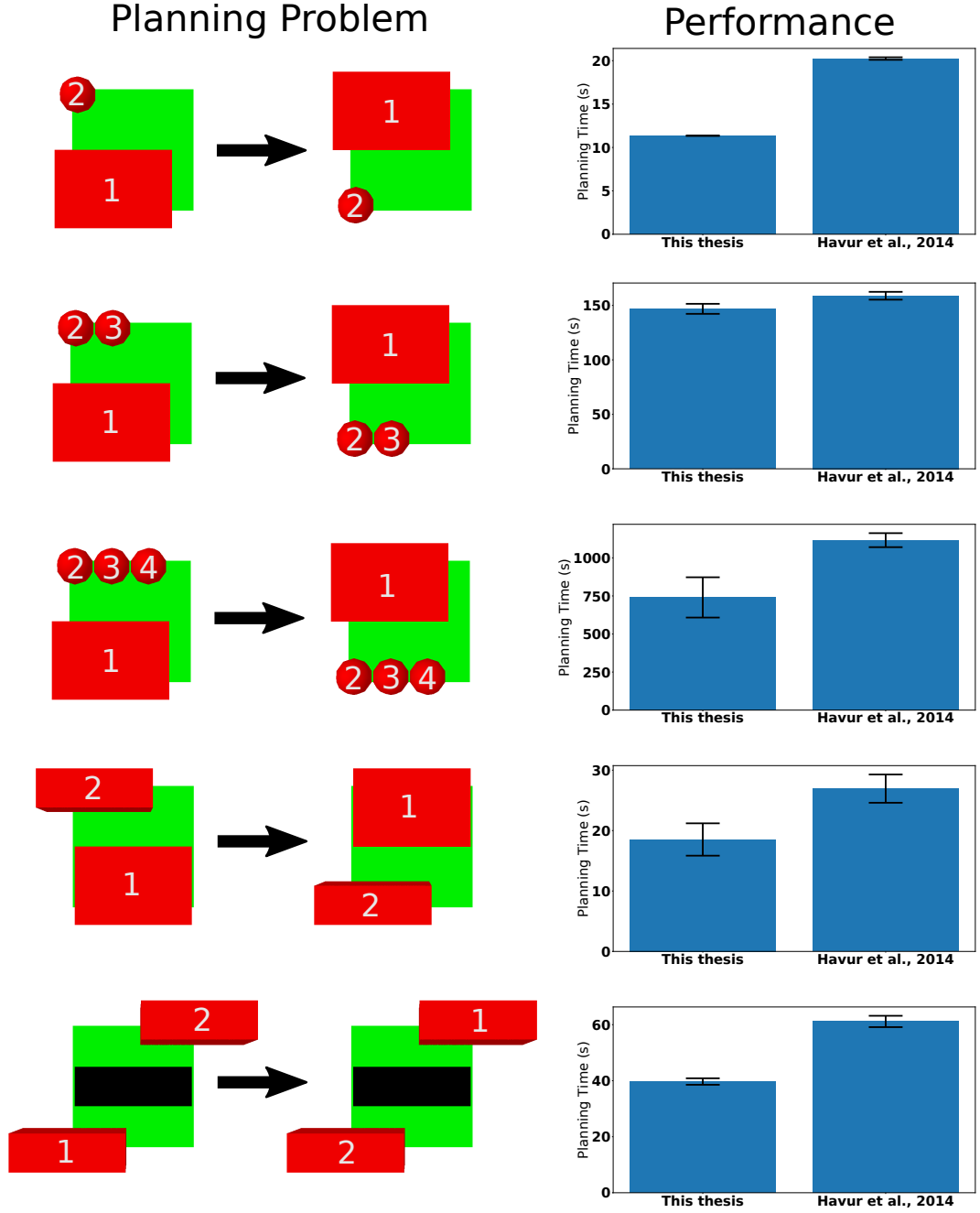


Fig. 6.15. Plots summarizing the results of Experiment 7.

6.3 Physical Implementation

To verify the applicability of our methods in real-life situations, we use a BAXTER robot to rearrange objects that of arbitrary shapes. Using the camera embedded into the robot's end effector, a simple visual servoing algorithm with color thresholding is used to detect and localize objects in its surroundings. A single arm of the robot is used for the rearrangement with overhand grasps. Three instances are explored:

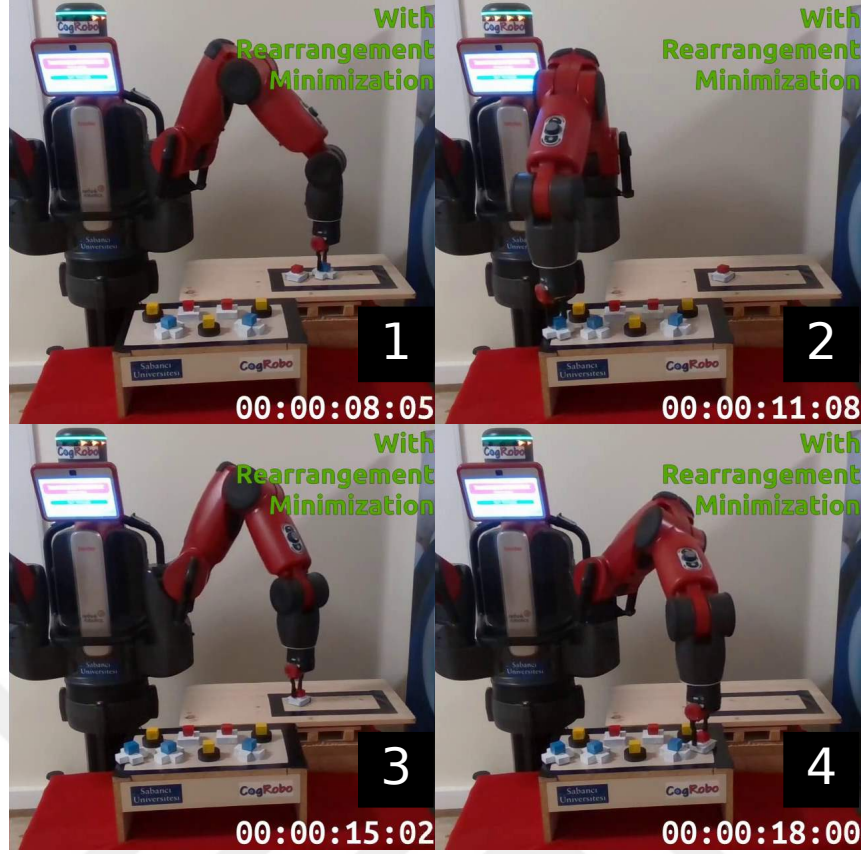


Fig. 6.16. BAXTER executing a plan with placement generated using *Outer*.

1. Figure 6.17 shows the execution of a monotone plan, whose goal state was calculated using *Intermediate*.
2. Figure 6.16 shows the execution of a monotone plan, whose goal state was calculated using *Outer*.
3. Figure 6.18 shows the execution of a non-monotone plan with forced swaps.

In all instances, the red-white objects are convex movable objects, the blue-white objects are non-convex movable objects, and the yellow-black objects are obstacles.

In Figures 6.17 and 6.16, the same instance is used. Timestamps are included to highlight the cost of rearranging objects. In particular, minimizing the number of rearranged objects by using *Outer* at the placement stage resulted in no rearrangements, so the entire operation took approximately 20 seconds, whereas using *Intermediate* resulting in a plan that involved two rearrangements and took approximately 30 seconds.

In Figure 6.18, a nonmonotone instance is given where a red-white and a blue-white object must swap positions. Note, in particular, the tightness of the configurations shown at steps 4-5. Generating such a plan requires a continuous feasibility check.

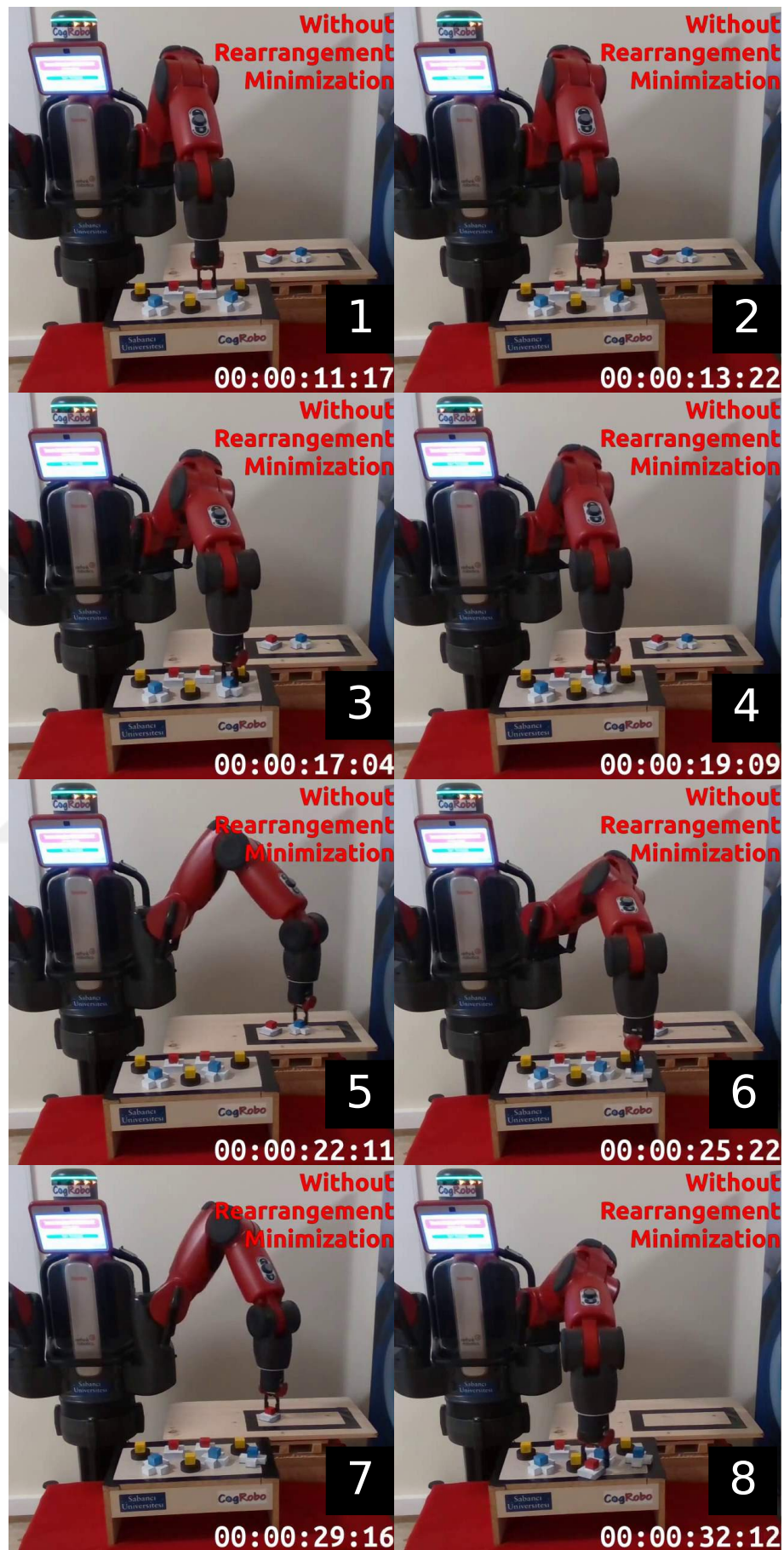


Fig. 6.17. BAXTER executing a plan with placement generated using *Intermediate*.

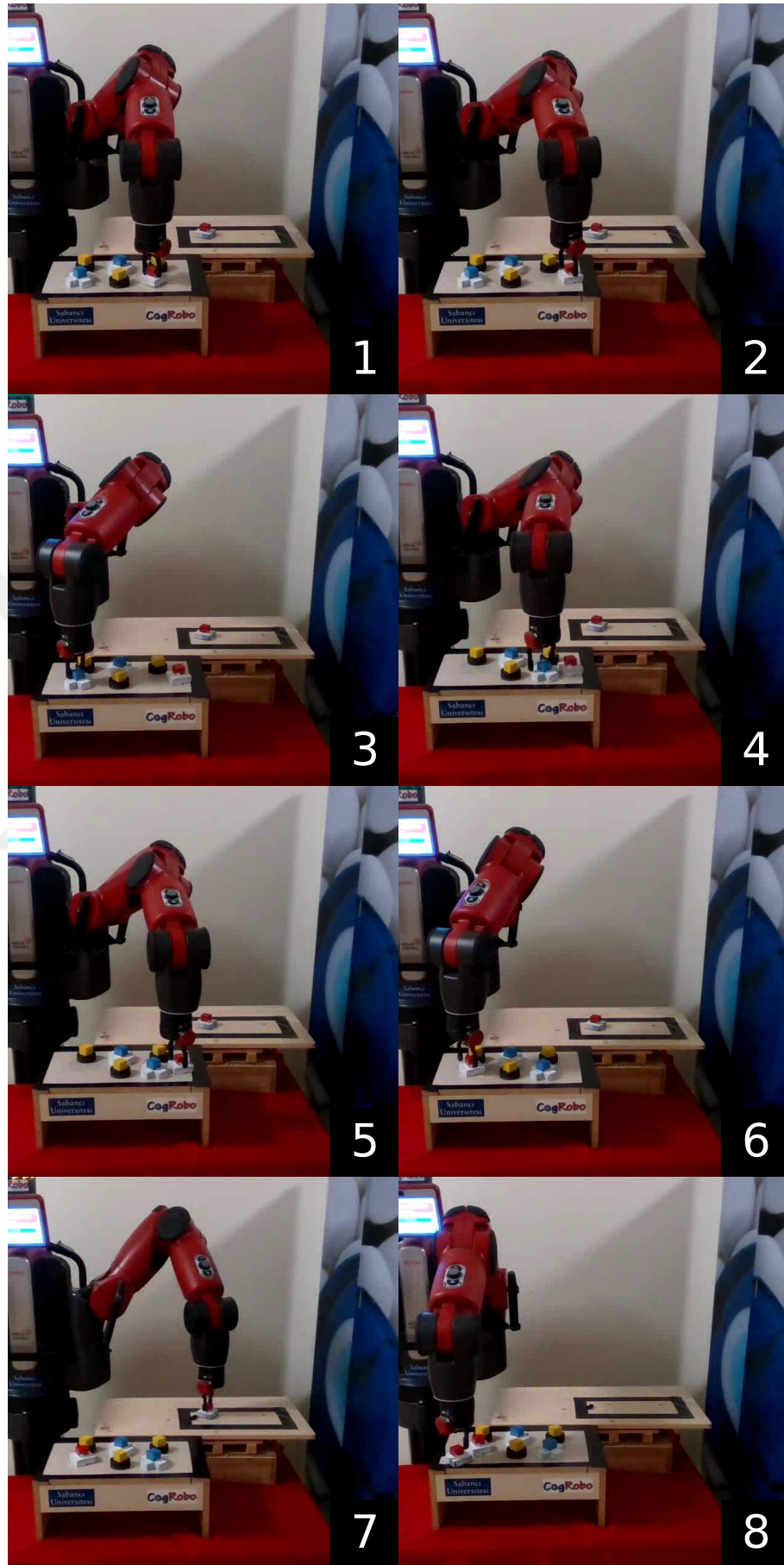


Fig. 6.18. BAXTER executing a nonmonotone plan with a forced swap.

Chapter 7

Conclusion

Given a surface cluttered with (immovable) obstacles and movable objects, and a new set of objects, the object placement problem asks for a collision-free placement of all the objects on the surface. We have introduced a novel algorithm to solve this problem, utilizing nested local search algorithms with multi-objective optimizations: the innermost search tries to minimize the total penetration depths of objects, the intermediate local search further tries to minimize the number of collisions, and the outermost local search further tries to minimize the changes in object poses. Each level of the search is guided by heuristics: the innermost search utilizes a potential field method over a physics-based engine, the intermediate local search gradually utilizes re-placements of objects to avoid local minima, and the outermost local search gradually relaxes the constraints that specify how far objects can be displaced. In that sense, our method introduces novel mathematical search models with nested multi-objective optimizations, and algorithms that further utilize heuristics to avoid local minima. On the other hand, due to local searches, our algorithm is more about local optimization and is likely not to reach the global optima. Comprehensive experimental evaluations demonstrate high computational efficiency and success rate of our method, as well as good quality of solutions. Furthermore, difficult benchmark scenarios that include non-convex objects, confined surfaces, and very small solution spaces can also be solved with our method.

Our placement generation method is shown to improve planning times while using the hybrid planning framework. Furthermore, improvements to hybrid rearrangement planning are also introduced by integrating a more sophisticated feasibility checker based on object placement that also ensures plan connectivity. We demonstrate the applicability of our methods by testing them with simulated benchmark scenarios and with real-life executions using a BAXTER robot.

Bibliography

- [1] G. Havur, G. Ozbilgin, E. Erdem, and V. Patoglu, “Geometric rearrangement of multiple movable objects on cluttered surfaces: A hybrid reasoning approach,” in *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, IEEE, 2014, pp. 445–452.
- [2] M. Stilman and J. J. Kuffner, “Navigation among movable obstacles: Real-time reasoning in complex environments,” *International Journal of Humanoid Robotics*, vol. 2, no. 04, pp. 479–503, 2005.
- [3] M. Stilman and J. Kuffner, “Planning among movable obstacles with artificial constraints,” *The International Journal of Robotics Research*, vol. 27, no. 11-12, pp. 1295–1307, 2008.
- [4] G. Wilfong, “Motion planning in the presence of movable obstacles,” *Annals of Mathematics and Artificial Intelligence*, vol. 3, no. 1, pp. 131–150, 1991.
- [5] E. D. Demaine, M. L. Demaine, M. Hoffmann, and J. O’Rourke, “Pushing blocks is hard,” *Computational Geometry*, vol. 26, no. 1, pp. 21–36, 2003.
- [6] K. Okada, A. Haneda, H. Nakai, M. Inaba, and H. Inoue, “Environment manipulation planner for humanoid robots using task graph that generates action sequence,” in *Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, IEEE, vol. 2, 2004, pp. 1174–1179.
- [7] M. Stilman, J.-U. Schamburek, J. Kuffner, and T. Asfour, “Manipulation planning among movable obstacles,” Georgia Institute of Technology, 2007.
- [8] M. R. Dogar and S. S. Srinivasa, “A planning framework for non-prehensile manipulation under clutter and uncertainty,” *Autonomous Robots*, vol. 33, no. 3, pp. 217–236, 2012.
- [9] J. Barry, K. Hsiao, L. P. Kaelbling, and T. Lozano-Pérez, “Manipulation with multiple action types,” in *Experimental Robotics*, Springer, 2013, pp. 531–545.
- [10] A. Cosgun, T. Hermans, V. Emeli, and M. Stilman, “Push planning for object placement on cluttered table surfaces,” in *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, IEEE, 2011, pp. 4627–4632.
- [11] A. Krontiris, R. Shome, A. Dobson, A. Kimmel, and K. Bekris, “Rearranging similar objects with a manipulator using pebble graphs,” in *Humanoid Robots (Humanoids), 2014 14th IEEE-RAS International Conference on*, IEEE, 2014, pp. 1081–1087.

- [12] A. Kroutiris and K. E. Bekris, “Dealing with difficult instances of object rearrangement,” in *Robotics: Science and Systems*, 2015.
- [13] —, “Efficiently solving general rearrangement tasks: A fast extension primitive for an incremental sampling-based planner,” in *Robotics and Automation (ICRA), 2016 IEEE International Conference on*, IEEE, 2016, pp. 3924–3931.
- [14] S. D. Han, N. M. Stiffler, A. Kroutiris, K. E. Bekris, and J. Yu, “High-quality tabletop rearrangement with overhand grasps: Hardness results and fast methods,” in *Proceedings of Robotics: Science and Systems*, 2017.
- [15] M. Kang, Y. Kwon, and S.-E. Yoon, “Automated task planning using object arrangement optimization,” in *2018 15th International Conference on Ubiquitous Robots (UR)*, IEEE, 2018, pp. 334–341.
- [16] S. Zagoruyko, Y. Labbé, I. Kalevatykh, I. Laptev, J. Carpentier, M. Aubry, and J. Sivic, “Monte-carlo tree search for efficient visually guided rearrangement planning,” *arXiv preprint arXiv:1904.10348*, 2019.
- [17] N. T. Dantam, Z. K. Kingston, S. Chaudhuri, and L. E. Kavraki, “An incremental constraint-based framework for task and motion planning,” *The International Journal of Robotics Research*, vol. 37, no. 10, pp. 1134–1151, 2018.
- [18] L. F. Yu, S. K. Yeung, C. K. Tang, D. Terzopoulos, T. F. Chan, and S. J. Osher, “Make it home: Automatic optimization of furniture arrangement,” 2011.
- [19] Y. Jiang, M. Lim, C. Zheng, and A. Saxena, “Learning to place new objects in a scene,” *The International Journal of Robotics Research*, vol. 31, no. 9, pp. 1021–1043, 2012.
- [20] Y. Jiang, M. Lim, and A. Saxena, “Learning object arrangements in 3d scenes using human context,” in *Proceedings of the 29th International Conference on Machine Learning*, Omnipress, 2012, pp. 907–914.
- [21] Y. Jiang and A. Saxena, “Hallucinating humans for learning robotic placement of objects,” in *Experimental Robotics*, Springer, 2013, pp. 921–937.
- [22] B. Chazelle, H. Edelsbrunner, and L. J. Guibas, “The complexity of cutting complexes,” *Discrete & Computational Geometry*, vol. 4, no. 2, pp. 139–181, 1989.
- [23] H. Dyckhoff, “A typology of cutting and packing problems,” *European Journal of Operational Research*, vol. 44, no. 2, pp. 145–159, 1990.
- [24] X. Liu, J.-m. Liu, A.-x. Cao, and Z.-l. Yao, “Hape3d—a new constructive algorithm for the 3d irregular packing problem,” *Frontiers of Information Technology & Electronic Engineering*, vol. 16, no. 5, pp. 380–390, 2015.
- [25] T. Romanova, J. Bennell, Y. Stoyan, and A. Pankratov, “Packing of concave polyhedra with continuous rotations using nonlinear optimisation,” *European Journal of Operational Research*, vol. 268, no. 1, pp. 37–53, 2018.
- [26] Y. Ma, Z. Chen, W. Hu, and W. Wang, “Packing irregular objects in 3d space via hybrid optimization,” *Computer Graphics Forum*, vol. 37, no. 5, pp. 49–59, 2018.

- [27] J. Egeblad, B. K. Nielsen, and M. Brazil, “Translational packing of arbitrary polytopes,” *Computational Geometry*, vol. 42, no. 4, pp. 269–288, 2009.
- [28] S. Martello, D. Pisinger, and D. Vigo, “The three-dimensional bin packing problem,” *Operations Research*, vol. 48, no. 2, pp. 256–267, 2000.
- [29] G. Fasano, “A global optimization point of view to handle non-standard object packing problems,” *Journal of Global Optimization*, vol. 55, no. 2, pp. 279–299, 2013.
- [30] E. Erdem, K. Haspalamutgil, C. Palaz, V. Patoglu, and T. Uras, “Combining high-level causal reasoning with low-level geometric reasoning and motion planning for robotic manipulation,” in *2011 IEEE International Conference on Robotics and Automation*, IEEE, 2011, pp. 4575–4581.
- [31] E. Erdem, E. Aker, and V. Patoglu, “Answer set programming for collaborative housekeeping robotics: Representation, reasoning, and execution,” *Intelligent Service Robotics*, vol. 5, no. 4, pp. 275–291, 2012.
- [32] N. T. Dantam, Z. K. Kingston, S. Chaudhuri, and L. E. Kavraki, “Incremental task and motion planning: A constraint-based approach..”
- [33] S. M. LaValle, “Rapidly-exploring random trees: A new tool for path planning,” 1998.
- [34] J. J. Kuffner Jr and S. M. LaValle, “Rrt-connect: An efficient approach to single-query path planning,” in *ICRA*, vol. 2, 2000.
- [35] S. M. LaValle and J. J. Kuffner Jr, “Randomized kinodynamic planning,” *The international journal of robotics research*, vol. 20, no. 5, pp. 378–400, 2001.
- [36] C. H. Papadimitriou, “The euclidean travelling salesman problem is np-complete,” *Theoretical computer science*, vol. 4, no. 3, pp. 237–244, 1977.
- [37] R. M. Karp, “Reducibility among combinatorial problems,” in *Complexity of computer computations*, Springer, 1972, pp. 85–103.
- [38] J. E. King, V. Ranganeni, and S. S. Srinivasa, “Unobservable monte carlo planning for nonprehensile rearrangement tasks,” in *Robotics and Automation (ICRA), 2017 IEEE International Conference on*, IEEE, 2017, pp. 4681–4688.
- [39] R. Coulom, “Efficient selectivity and backup operators in monte-carlo tree search,” in *International conference on computers and games*, Springer, 2006, pp. 72–83.
- [40] M. W. Gardner and S. Dorling, “Artificial neural networks (the multilayer perceptron)—a review of applications in the atmospheric sciences,” *Atmospheric environment*, vol. 32, no. 14-15, pp. 2627–2636, 1998.
- [41] S. J. Russell and P. Norvig, *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited, 2016.
- [42] E. L. Lawler, J. K. Lenstra, A. R. Kan, D. B. Shmoys, *et al.*, *The traveling salesman problem: a guided tour of combinatorial optimization*, vol. 3.
- [43] V. Lifschitz, “Answer set programming and plan generation,” *Artif. Intell.*, vol. 138, no. 1-2, pp. 39–54, 2002.
- [44] G. Brewka, T. Eiter, and M. Truszczyński, “Answer set programming at a glance,” *Commun. ACM*, vol. 54, no. 12, pp. 92–103, 2011.

- [45] M. Gelfond and V. Lifschitz, “Classical negation in logic programs and disjunctive databases,” *New Generation Comput.*, vol. 9, no. 3/4, pp. 365–386, 1991.
- [46] M. Gelfond and Y. Kahl, *Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach*. New York, NY, USA: Cambridge University Press, 2014.
- [47] V. Lifschitz, “What is answer set programming?” In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, 2008, pp. 1594–1597.
- [48] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub, “Clingo = ASP + control: Preliminary report,” *CoRR*, vol. abs/1405.3694, 2014.
- [49] T. Eiter, G. Ianni, R. Schindlauer, and H. Tompits, “Dlvhex: A prover for semantic-web reasoning under the answer-set semantics,” in *2006 IEEE / WIC / ACM International Conference on Web Intelligence (WI 2006 Main Conference Proceedings)(WI’06)*, IEEE, 2006, pp. 1073–1074.
- [50] M. Gebser, B. Kaufmann, R. Kaminski, M. Ostrowski, T. Schaub, and M. Schneider, “Potassco: The potsdam answer set solving collection,” *Ai Communications*, vol. 24, no. 2, pp. 107–124, 2011.
- [51] M. Moore and J. Wilhelms, “Collision detection and response for computer animation,” in *ACM Siggraph Computer Graphics*, ACM, vol. 22, 1988, pp. 289–298.
- [52] X. Tang, A. Paluszny, and R. Zimmerman, “Energy conservative property of impulse-based methods for collision resolution,” *International Journal for Numerical Methods in Engineering*, vol. 95, no. 6, pp. 529–540, 2013.
- [53] D. Baraff, “Analytical methods for dynamic simulation of non-penetrating rigid bodies,” in *ACM SIGGRAPH Computer Graphics*, ACM, vol. 23, 1989, pp. 223–232.
- [54] A. Seth, J. M. Vance, and J. H. Oliver, “Virtual reality for assembly methods prototyping: A review,” *Virtual reality*, vol. 15, no. 1, pp. 5–20, 2011.
- [55] J. K. Hahn, “Realistic animation of rigid bodies,” in *Acm Siggraph Computer Graphics*, ACM, vol. 22, 1988, pp. 299–308.
- [56] B. Mirtich and J. Canny, “Impulse-based simulation of rigid bodies,” in *Proceedings of the 1995 symposium on Interactive 3D graphics*, ACM, 1995, 181–ff.
- [57] B. Mirtich and J. Canny, “Hybrid simulation: Combining constraints and impulses,” in *Proceedings of First Workshop on Simulation and Interaction in Virtual Environments*, 1996.
- [58] E. Coumans *et al.*, “Bullet physics library,” *Open source: bulletphysics.org*, vol. 15, no. 49, p. 5, 2013.
- [59] E. Rohmer, S. P. Singh, and M. Freese, “V-rep: A versatile and scalable robot simulation framework,” in *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, IEEE, 2013, pp. 1321–1326.
- [60] R. Smith *et al.*, “Open dynamics engine,” 2005.

- [61] E. Todorov, T. Erez, and Y. Tassa, “Mujoco: A physics engine for model-based control,” in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, IEEE, 2012, pp. 5026–5033.
- [62] V. Patoglu and R. B. Gillespie, “A feedback stabilized minimum distance maintenance for convex parametric surfaces,” *IEEE Transactions on Robotics*, vol. 25, no. 5, 2005.
- [63] E. Aker, V. Patoglu, and E. Erdem, “Answer set programming for reasoning with semantic knowledge in collaborative housekeeping robotics,” *IFAC Proceedings Volumes*, vol. 45, no. 22, pp. 77–83, 2012.
- [64] E. Erdem, V. Patoglu, and P. Schüller, “A systematic analysis of levels of integration between high-level task planning and low-level feasibility checks,” *AI Communications*, vol. 29, no. 2, pp. 319–349, 2016.



Appendix A

Detailed Results



Surface Coverage (%)	Random Sample			Inner			Random Restart			Intermediate			Outer			Havur		
	Time (s)	Rearranged Objects	Success (%)	Time (s)	Rearranged Objects	Success (%)	Time (s)	Rearranged Objects	Success (%)	Time (s)	Rearranged Objects	Success (%)	Time (s)	Rearranged Objects	Success (%)	Time (s)	Rearranged Objects	Success (%)
Experiment set 1: controlling surface coverage by varying the number of new objects																		
19.85	0.03	3.73	100.00	20.04	1.18	0.00	0.04	1.25	100.00	0.02	1.32	100.00	0.13	0.00	100.00	0.01	0.00	100.00
29.00	7.56	4.00	100.00	30.09	2.07	0.00	0.10	2.83	100.00	0.09	1.95	100.00	0.41	0.00	100.00	0.04	0.00	100.00
38.14	295.63	4.00	0.00	35.14	2.80	0.00	0.28	4.00	100.00	0.19	2.77	100.00	1.88	0.65	100.00	0.11	0.00	100.00
47.28	300.00	4.00	0.00	75.25	2.97	0.00	0.59	4.00	100.00	0.53	3.25	100.00	4.36	0.82	100.00	0.38	0.00	100.00
56.42	300.00	4.00	0.00	80.36	3.50	0.00	1.81	4.00	100.00	0.94	3.42	100.00	8.29	0.93	100.00	3.04	0.02	100.00
65.56	300.00	3.98	0.00	300.00	3.67	0.00	7.76	4.00	100.00	2.95	3.77	100.00	23.68	1.10	100.00	126.82	2.55	0.00
74.70	300.00	4.00	0.00	300.00	3.83	0.00	36.49	4.00	100.00	12.57	3.93	100.00	73.13	1.60	100.00	300.00	1.23	0.00
83.85	300.00	4.00	0.00	300.00	3.85	0.00	215.40	4.00	0.00	54.99	3.97	0.00	168.78	2.43	0.00	300.00	1.48	0.00
92.99	300.00	4.00	0.00	300.00	3.98	0.00	300.00	4.00	0.00	239.28	4.00	0.00	276.75	2.13	0.00	300.00	1.28	0.00
Experiment set 2: controlling surface coverage by varying the number of original objects																		
19.85	0.03	3.67	100.00	5.02	2.87	0.00	0.04	3.17	100.00	0.03	3.15	100.00	0.17	0.48	100.00	0.02	0.00	100.00
29.00	7.87	8.00	100.00	10.04	5.00	0.00	0.11	6.85	100.00	0.12	5.35	100.00	2.59	3.05	100.00	0.04	0.00	100.00
38.14	297.38	11.98	0.00	50.08	7.50	0.00	0.29	12.00	100.00	0.29	7.87	100.00	22.60	3.50	100.00	0.07	0.00	100.00
47.28	300.00	15.98	0.00	75.19	11.97	0.00	0.67	16.00	100.00	0.78	12.48	100.00	67.59	8.12	100.00	0.12	0.00	100.00
56.42	300.00	19.98	0.00	85.24	14.98	0.00	1.82	20.00	100.00	1.35	15.20	100.00	122.78	9.75	0.00	0.20	0.00	100.00
65.56	300.00	23.97	0.00	300.00	19.47	0.00	9.10	24.00	100.00	3.13	20.00	100.00	243.89	10.97	0.00	0.86	0.00	100.00
74.70	300.00	27.97	0.00	300.00	21.35	0.00	37.00	28.00	100.00	11.11	23.82	100.00	292.47	7.18	0.00	24.03	1.00	0.00
83.85	300.00	31.97	0.00	300.00	27.10	0.00	224.82	31.98	0.00	69.98	30.65	0.00	300.00	6.83	0.00	286.49	3.95	0.00
92.99	300.00	35.98	0.00	300.00	31.58	0.00	299.21	35.98	0.00	210.80	35.72	0.00	300.00	6.48	0.00	300.00	4.53	0.00
Experiment set 3: controlling surface coverage by varying the number of obstacle objects																		
27.42	0.37	4.00	100.00	40.03	1.53	0.00	0.08	2.52	100.00	0.11	1.73	100.00	1.19	1.62	100.00	0.03	0.00	100.00
36.57	22.00	3.98	100.00	45.04	1.53	0.00	0.16	3.78	100.00	0.20	1.60	100.00	2.03	1.67	100.00	0.04	0.00	100.00
45.71	218.12	4.00	0.00	60.07	1.60	0.00	0.41	4.00	100.00	0.49	1.62	100.00	4.49	1.97	100.00	0.05	0.00	100.00
54.85	294.39	4.00	0.00	80.07	1.50	0.00	1.56	3.98	100.00	0.69	1.77	100.00	6.29	2.05	100.00	0.11	0.00	100.00
63.99	300.00	4.00	0.00	95.10	1.58	0.00	8.76	3.98	100.00	1.08	1.70	100.00	12.09	1.83	100.00	0.40	0.00	100.00
73.13	300.00	4.00	0.00	300.00	1.53	0.00	135.49	3.95	0.00	5.59	2.40	100.00	19.12	1.93	100.00	3.80	0.02	100.00
82.27	300.00	4.00	0.00	300.00	1.73	0.00	300.00	3.98	0.00	93.04	3.68	0.00	52.17	2.73	100.00	237.69	0.12	0.00
91.42	300.00	3.98	0.00	300.00	1.58	0.00	300.00	3.98	0.00	300.00	4.00	0.00	300.00	2.03	0.00	300.00	0.00	0.00

Table A.1: Experiments 1, 2, and 3 results: placement generation performance.

Original Objects	Surface Coverage (%)	Intermediate					Outer				
		Rearranged Objects	Plan Steps	Placement Time (s)	Planning Time (s)	Total Time (s)	Rearranged Objects	Plan Steps	Placement Time (s)	Planning Time (s)	Total Time (s)
Experiment set 4: low surface coverage levels.											
1	31.00	0.40	12.40	0.20	0.45	0.65	0.00	12.00	0.28	0.00	0.28
2	35.00	1.15	13.15	0.21	1.00	1.22	0.10	12.10	1.09	0.12	1.21
3	36.57	2.50	14.50	0.27	1.81	2.10	0.15	12.15	1.25	0.18	1.44
4	38.14	3.05	15.05	0.42	3.67	4.12	0.35	12.35	2.77	0.35	3.13
5	40.14	3.80	15.80	0.38	5.33	5.76	0.80	12.80	6.08	2.46	8.55
6	44.14	4.65	16.65	0.47	6.22	6.78	0.35	12.35	7.66	0.44	8.11
7	45.71	5.90	17.90	0.40	68.15	68.76	0.90	12.90	19.39	1.16	20.58
8	47.28	6.80	18.80	0.89	54.41	55.80	1.15	13.15	14.33	4.32	18.70
9	49.28	7.650	19.65	0.70	579.96	581.75	1.65	13.65	24.10	2.93	27.13
Experiment set 5: moderate surface coverage levels.											
1	49.28	0.70	20.70	0.20	0.36	0.79	0.05	20.05	0.65	0.03	0.68
2	53.28	1.90	21.90	0.21	0.58	1.38	0.25	20.25	2.58	0.11	2.69
3	54.85	2.50	22.50	0.27	1.05	1.86	0.10	20.10	4.68	0.06	4.74
4	56.42	3.45	23.45	0.42	3.04	5.86	0.55	20.55	4.01	0.20	4.21
5	58.42	4.25	24.25	0.38	14.70	15.75	0.65	20.65	9.04	0.34	9.39
6	62.42	5.45	25.45	0.47	29.87	33.21	0.70	20.70	21.72	0.69	22.42
7	63.99	6.65	26.65	0.40	116.02	120.11	1.65	21.65	57.29	1.27	58.61
8	65.56	7.80	27.80	0.89	219.98	222.79	2.50	22.50	101.67	3.17	104.94
9	67.56	8.30	28.30	0.70	944.94	950.55	2.70	22.70	248.79	13.05	262.04
Experiment set 6: high surface coverage levels.											
1	67.56	0.90	28.90	3.73	0.46	4.21	0.00	28.00	6.79	0.00	6.79
2	71.56	2.00	30.00	18.30	0.58	18.90	0.30	28.30	13.07	0.12	13.19
3	73.13	2.90	30.90	3.47	1.03	4.52	0.60	28.60	29.70	0.24	29.95
4	74.70	3.90	31.90	12.76	4.81	17.61	1.25	29.25	68.59	0.55	69.16
5	76.70	4.95	32.95	18.73	32.57	51.37	1.25	29.25	138.17	0.66	138.85
6	80.70	5.95	33.95	25.28	51.66	77.07	1.20	29.20	320.27	0.73	321.03
7	82.27	7.00	35.00	34.30	105.85	140.41	2.30	30.30	484.38	15.97	500.42
8	83.85	8.00	36.00	67.32	438.47	506.47	3.60	31.60	732.87	16.98	750.03
9	85.85	8.95	36.95	60.94	5958.91	6021.38	4.35	32.35	1331.17	23.54	1355.02

Table A.2: Experiment 4, 5, and 6 results: effect of placement generation algorithm on rearrangement planning performance.