Practical Pushing Planning for Rearrangement Tasks

Ohad Ben-Shahar and Ehud Rivlin

***Abstract*— In this paper, we address the problem of *practical* manipulation planning for rearrangement tasks of *many* movable objects. We study a special case of the rearrangement task, where the only allowed manipulation is pushing. While problems of this kind are known to be PSPACE-hard, we search for algorithms that can provide practical planning time for most common scenarios. We present a hierarchical classification of manipulation problems into several classes, each characterized by properties of the plans that can solve it. Such a classification allows one to consider each class individually, to analyze and exploit properties of each class, and to suggest individual planning methods accordingly. Following this classification, we suggest algorithms for two of the defined classes. Both items have been tested in a simulated environment, with up to 32 movable objects and 66 combined DOF. We present simulations results (with up to 10 movables), statistical data from 1000 randomly generated problems, as well as some experimental results using a real platform.**

I. Introduction

REARRANGEMENT of objects in a given workspace is a basic manipulation task. The applications of a rearrangement system are numerous, ranging from fine assembly planning to household maintenance tasks. However, the high complexity of the underlying planning problem prevents the realization of a complete system that can handle scenarios of *many* objects. In this paper, we address the problem of planning a rearrangement plan while considering several special features.

In the light of the above, the first feature we are interested in deals with the number of objects that the system can handle. Wilfong [17] showed that motion planning in the presence of movable objects is PSPACE-hard. This result convinced other researchers to concentrate first on constrained versions of the problem, namely with cases of one or few movable objects only [1], [3], [6], [8], [12], [14], [17]. Different from previous works, in this study we are interested in problems which engage *many* movable objects, as happens in most practical scenarios. Undoubtedly, complete/optimal algorithms that solve such problems are not expected to be practical, hence we seek for practical methods even at the expense of completeness/optimality.

Manuscript received January 10, 1997; revised January 15, 1998. This paper was first published in the *Proceedings of the 13th International Conference on Robotics and Automation*, Minneapolis, MN, 1996, pp. 172–177. This paper was recommended for publication by Associate Editor M. Peshkin and Editor V. Lumelsky upon evaluation of the reviewers' comments.

The authors are with the Center for Intelligent Systems, Department of Computer Science, Technion, Israel (e-mail: obs@cs.technion.ac.il; ehudr@cs.technion.ac.il).

Publisher Item Identifier S 1042-296X(98)04615-1.

A second feature which we introduce into our study is the fact that we allow our robot to manipulate the objects only by *pushing* them (rather than grasping them). This constraint is motivated by several observations: The action of pushing allows easier manipulation of larger and heavier objects. It permits easier simultaneous manipulation of groups of objects, and most importantly, it can be realized with simple and cheap robot structures. In that sense, the pushing manipulation allows any mobile robot to be the manipulator of the system. Then again, the action of pushing has many disadvantages over grasping, which make it less widely used after all. The action of pushing is inherently restricted to a support surface and does not allow the robot to exploit the third dimension while manipulating the object. In addition, pushing is different from grasping by the fact that it might bring the object into irreversible configurations (e.g., corners) which we call *trap points* [6]. Hence, unlike with grasping, planning is essential for systems that rearrange objects by pushing. Finally, the action of pushing is mechanically unstable and thus various control problems arise. As it is emphasized in the subsequent sections, this study focuses on planning issues that emerge from the former two points, while assuming that the control problems have been solved separately by an external component. Note that some undesirable aspects of the mechanics of pushing might be minimized by an appropriate choice of the pusher geometry.

The major contribution of this paper is twofold—the presentation of a novel formal classification of manipulation/rearrangement problems and the development of two practical planning algorithms. The presented classification allows to consider each class individually, to analyze and exploit properties of each class, and to suggest individual practical planning methods accordingly. The suggested algorithms correspond to two of the defined classes. Both algorithms can solve rearrangement problems of *many* movable objects amidst cluttered environment. Different from some previous works in the area [1], [16], our methods provide detailed manipulation plans, including any intermediate motion of the pusher while changing contact configuration with the pushed movables. While being specifically designed for the pushing manipulation, both algorithms are fully compatible with other manipulations too (e.g., grasping).

II. Problem Formulation

Let $\mathcal{B} = \{\mathcal{R}, \mathcal{I}, \mathcal{M}_1, \cdots, \mathcal{M}_n\}$ be a set of bodies composing the environment. $\mathcal{R}$ is a *robot* (i.e., capable of self movement), $\mathcal{I}$ represents the union of all *immovable* static

bodies (i.e., obstacles), and $\{\mathcal{M}_1, \cdots, \mathcal{M}_n\}$ is a collection of *movable* rigid objects.

Each of the participating dynamic objects has its own configuration space. Let CS_R and CS_{M_i} be the configuration spaces of $\mathcal{R}$ and $\mathcal{M}_i$, respectively. Since any pushing manipulation must be carried out in a context of some support surface, we consider CS_{M_i} to have two or three dimensions only. Let CCS be the composite configuration space of $\mathcal{R}$ and all movable objects. Each vector in CCS is a *composite configuration* $Q = (\mathbf{q_R}, \mathbf{q_{M_1}}, \cdots, \mathbf{q_{M}}_n)$ for $\mathbf{q_R} \in CS_R$ and $\mathbf{q_{M}}_i \in CS_{M_i}$. Along with the above configuration spaces we will also use the following projection operators:

$$\Pi_R\text{: } CCS \mapsto CS_R$$
$$\Pi_R(Q) = \Pi_R(\mathbf{q_R}, \mathbf{q_{M_1}}, \cdots, \mathbf{q_{M}}_n) = \mathbf{q_R}$$
$$\Pi_{M_i}\text{: } CCS \mapsto CS_{M_i}$$
$$\Pi_{M_i}(Q) = \Pi_{M_i}(\mathbf{q_R}, \mathbf{q_{M_1}}, \cdots, \mathbf{q_{M}}_n) = \mathbf{q_{M}}_i.$$

Let $P(Q_1, Q_2)$ denote a *configuration path* (*C-path*) between Q_1 and Q_2. It is clear that not every C-path $P(Q_1, Q_2)$ is a *pushing C-path* from Q_1 to Q_2 and that a constrained definition is needed. A brief discussion on that issue is outlined in Section IV.

Using most of the above we define the rearrangement planning problem as follows:

> Given $\mathcal{B}$ **(a description of the environment), an initial composite configuration** Q_0 **and some goal composite configuration** Q_G**, find a pushing C-path** $P(Q_0, Q_G)$**, or report whether no such path exists.**

Rather than an AI decision problem, this definition is phrased in terms of motion planning problems since we are interested in solutions that fully describe the motion of the robot while it is realizing the rearrangement plan. In addition to any pushing motion, these solutions should include any intermediate motions that change the contact configuration of the robot with the currently pushed object, as well as any motion that moves the robot from one object to another. Note that nothing in this definition constraints the robot from pushing more than one object simultaneously.

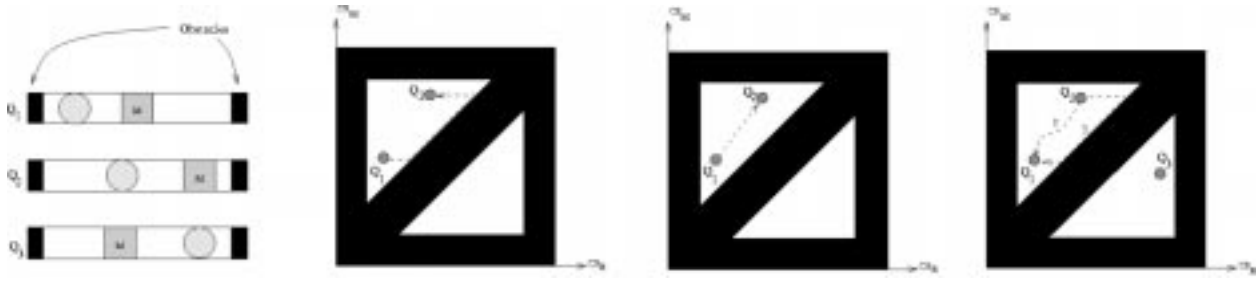
III. Related Work

The problem of object rearrangement in general, and the special variation addressed in this paper, relate to several research areas in robotics and AI. The most notable area is the one of *manipulation planning*, or motion planning in the presence of movable objects. As mentioned earlier, this variation of the basic motion planning problem was proved to be PSPACE-hard [17]. This result convinced other researchers to concentrate first on constrained versions of the problem, namely with cases of one or few movable objects alone. A generalized approach for the manipulation planning was proposed in [2], [3], [14], and [15]. They defined the solution as a special path—a *manipulation path*—in the composite configuration space of the robot and all movable objects, and applied the exact cell decomposition methodology in order to calculate that path. While being exact and complete, this approach has several limitations. It allows the manipulation of only one movable at a time and it is inherently limited to grasping manipulation. Even more important is the fact that this approach cannot practically handle scenarios of *many* movable objects due to its high (exponential) computational complexity.

When dealing with rearrangement problems, one may find many common aspects to *assembly planning* too. However, the state of the art research in the area of assembly planning [9] addresses problems with different characteristics than ours. Most such research ignores the manipulator, its geometry, and any constraints on the allowed manipulation (e.g., pushing only). The ultimate assembly planner should be able to generate plans directly from a CAD model of the goal assembly [18]. consequently, assembly planning tends to ignore the initial configuration of the parts and assumes that they come from infinity. We, on the other hand, are interested in *rearrangement* of parts, i.e., changing their common configuration from a given *initial* configuration to a given *goal* configuration. In that sense, as well in the others, our rearrangement planning problem is a generalization of the assembly planning problem.

When dealing with rearrangement problems of many movable objects, one has to handle configuration spaces of many DOF. Such scenarios received less attention due to the high complexity involved. Barraquand and Latombe [4], [5] addressed the large DOF motion planning problem by a *probabilistically resolution-complete* stochastic approach. They defined simple numerical potential fields over a discretized version of the composite configuration space, searched for a solution using a hill-climbing like search, and used a Monte-Carlo algorithm to escape from local minima. Such a method can provide fast solutions for some large DOF problems (the authors experimented with up to 31 DOF) and served as the basis for several other studies with large DOF robots [10] and multi-arm manipulation planning [11], [12]. However, the fact that the core of this method is a hill climbing search makes it problematic for our kind of problems due to the irreversibility of the pushing manipulation and the existence of trap points. In addition, a success in the random search seems to need a solution subspace which is comparably large, something which is not true for cluttered environments or rearrangement problems of many movable objects.

Despite the great deal of motion planning research, not much work has been done directly in the area of pushing planning. Akella and Mason [1] analyzed the series of pushes needed to bring a convex polygon to a desired configuration. While using pushing manipulation, this problem is a very constrained version of the rearrangement problem. They allowed only *one* convex movable object, used a simplified fence-like pusher, and ignored any other geometrical constraints (e.g., obstacles). A comprehensive study was carried out by Lynch and Mason [16] where both mechanics, control and planning issues were considered. Their planning method was based on a *best-first* search over an inexact representation of the configuration space, which aimed at finding a path to some neighborhood of the specified goal. They considered again only limited number of DOF by allowing only one movable object. It was also assumed that the fence-like pusher can change the contact configuration (chosen from a discrete



(a) (b) (c) (d)

Fig. 1. Pushing C-paths versus general configuration paths. (a) Three placements of a robot and one movable object, both with one degree of freedom (one axis of motion). (b) The composite configuration space of the problem and a pushing C-path from Q_1 to Q_2. (c) Another C-path from Q_1 to Q_2 which is not a pushing C-path. Note that this C-path includes self movement of M. (d) No pushing path can be found from Q_2 to Q_1 (note that the longer path is a pulling path). Similar conclusion can be drawn regarding a pushing path from Q_2 to Q_3, though different reasons apply.

set) at any time, with no restrictions. As mentioned before, in this paper we are interested in multi-object problems, where the solutions inherently integrate the motion of the pusher, including all intermediate motions between contact configurations. On the other hand, we are less interested in the mechanics of pushing, and assume that its effects are handled by an external component.

Finally, a somewhat different problem was addressed by Chen and Hwang [7] who presented a practical, heuristic, and inexact solution for many movable *obstacles*. Their method is primarily a motion planning method (rather than rearrangement planning) in which movable obstacles can be pushed away by the robot whenever they stand in its way to the goal.

IV. Basic Pushing Planning

In our foregoing paper [6], we presented a potential-field method for planning a pushing manipulation by a mobile robot which tries to rearrange several movable objects in its work space. That method was designed to solve rearrangement problems as defined in Section II, and it is resolution complete, optimal and flexible. However, it has only limited practical use due to the high complexity involved. This section briefly describes that algorithm since we use it as a building block in the practical algorithms described herein.

A. Pushing C-Paths

Given a pushing rearrangement problem as formulated in Section II, a planner that can solve it should find an appropriate C-path that represents the rearrangement plan. As mentioned earlier, it is clear that not every C-path $P(Q_1, Q_2)$ is a *pushing C-path* from Q_1 to Q_2, and that a constrained/refined definition is needed. Examples of C-paths and pushing C-paths are given in Fig. 1.

In general, we expect an appropriate C-path to have a structure of a *manipulation path*, as defined in [2], [3], [14], and [15]. However, several differences do apply. First, one should note that pushing force can be applied only in specific directions (i.e., one cannot push an object by moving away from it). This observation implies that any C-path segment that corresponds to pushing a movable object, cannot have an *arbitrary* direction in the configuration space. Second, while Laumond *et al.* defined each *transfer path* to manipulate only *one* movable object, the general pushing C-path should allow, in our view, a simultaneous manipulation of several objects. Finally, while Laumond *et al.* defined each transfer path to represent a *rigid manipulation* (i.e., a manipulation during which the geometric relationship between the manipulator and the object remains constant), we find *non rigid* manipulations to be more realistic, especially in the context of pushing manipulation.

B. Basic Algorithm

Having the above loosely described properties of the required pushing C-path, we solve the pushing rearrangement problem with a two-phase procedure, while using a *discretized* version of the composite configuration space of the robot and all movable objects.

The first phase, called the *cost mapping* phase, carries out a Dijkstra like propagation procedure which assigns a cost value to each cell of the *free* configuration space. The propagation is carried out *backward*, originating from the goal configuration(s). Each step of the propagation selects one cell, the one with the minimal cost, from the wave front and "floods" a subset of its neighbors. This subset, which we call the *admissible neighbors*, is carefully determined using several factors such as the mechanical model of the manipulation (which is assumed to be analyzed separately by an external component), constraints on the allowed manipulations (e.g., what contact points between the robot and the objects are allowed during the manipulation), and the maximum number of movables that the robot can push simultaneously. In addition, artificial constraints can be integrated into that mechanism, as elaborated in [6]. When the propagation is finished, every free cell in the configuration space is assigned the cost of the *cheapest* pushing C-path that connects it to the "closest" goal configuration. The cost metric itself can integrate many useful factors, such as the total pushed weight, the local passability of the support surface, and others.

The cost mapping phase is a preprocessing phase that must be executed only upon a change in the environment $(\mathcal{B})$ or the set of goal configurations. Otherwise, it can be executed only once, producing a fully mapped free space (or a potential field). Given such space and an initial composite configuration of the robot and all movable objects, a specific rearrangement plan is constructed by the *restoration phase*. That phase carries out a

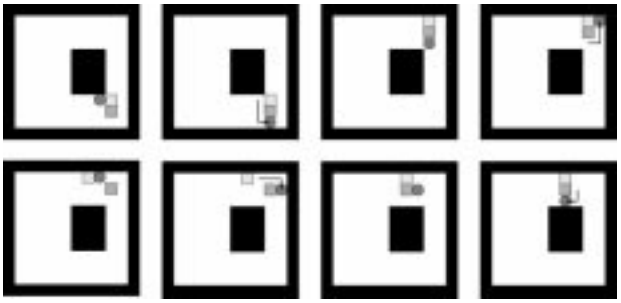


Fig. 2. Pushing plan for rearrangement of two movable objects, where the robot is allowed to push both objects simultaneously, if such a manipulation is found useful.

variation of a hill climbing procedure which starts at the initial configuration and ends at the first reached local minimum after following only admissible neighbors. The mapping process guarantees that every local minimum in the composite space is also a global minimum, hence one of the goals. The result of the restoration phase is a description of an appropriate C-path which represents the rearrangement plan. The projection of this C-path over CS_R yields the motion plan which the robot should execute in order to realize the solution.

Figs. 2 and 3 illustrate two selected results of the basic planning algorithm, as produced by our planner in [6]. The first is a pushing rearrangement plan where the robot could choose to push more than one object simultaneously. The second corresponds to a pushing rearrangement problem for which non rigid rotational pushings are required in order to achieve the goal. Both examples demonstrate the apparent differences between a manipulation path and a pushing C-path, as discussed in the previous subsection.

The above rearrangement planning method is resolution-complete and provides optimal solutions. Equally important is the fact that this method is easily implementable and very flexible, allowing constraints to be described using *local* terms, via admissible neighborhood relationships between configuration cells, rather than using global terms. Consequently, spatially *varying* constraints can be integrated as easily as spatially invariant ones[1]. Despite its advantages, this method is impractical due to its exponential space and time complexities. A complexity analysis, and a discussion of other limitations of the method can be found in [6]. It is this impractical behavior that motivated the study in this paper.

V. Problems Classification and Examples

Providing practical solutions for the general problem defined in Section II might be a difficult task. Hence, we take an alternative approach that may allow a gradual study of the general problem and provide practical solutions for problems of increasing difficulty. In this section, we divide the set of all rearrangement problems into some hierarchical classes. This classification is practically independent of the specific manipulation chosen to rearrange the objects, thus it may be useful for manipulations other than pushing too. Following this classification, we present practical algorithms and planners for two of the defined classes.

[1]For example, this feature allows to model a variable friction distribution, something which has been avoided in most related pushing research.

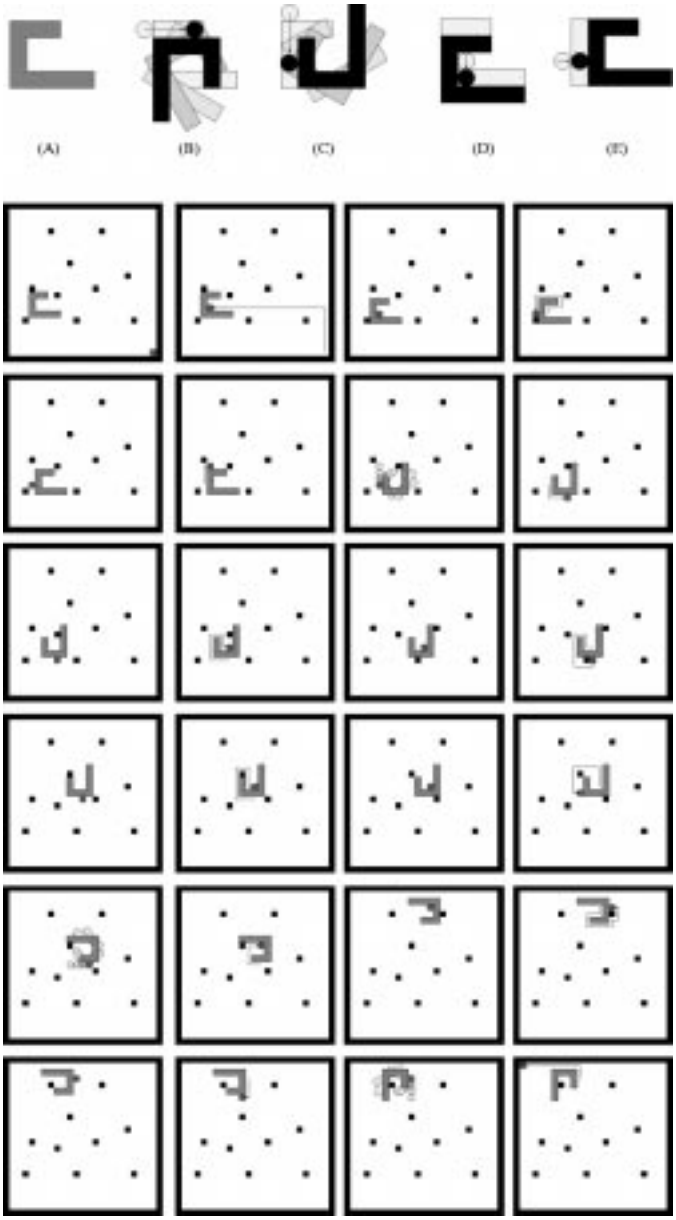


Fig. 3. Planning with non rigid rotational pushings: (a) details the movable object while (b)–(e) show all the allowed pushings (relative contact point and the pushing outcome). The rest of the figure outlines a pushing plan that maneuvers the object in a cluttered environment.

In what follows we use the following auxiliary definition:

Definition 1: A manipulation of a movable object $\mathcal{M}_i$ is called **nonpreemptive** iff (1) $\mathcal{M}_i$ is manipulated from its *initial configuration* to its *goal configuration*, and (2) while doing so, no other movable $\mathcal{M}_j$ changes its configuration (i.e., it does not move).

Let $\mathcal{PP}^n$ be the set of all *pushing* rearrangement problems of n movable objects. We start the classification of $\mathcal{PP}^n$ with a definition of two basic classes. These classes, as well as the others to come, are based on characteristics of the rearrangement *plans* that solve their problems.

Definition 2: A pushing rearrangement problem is called **flat** if the pusher can choose **any** permutation of nonpreemp-

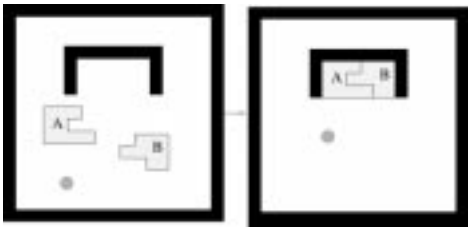


Fig. 4. $\mathcal{SP}^2$ problem.

tive pushings in order to achieve the goal. The class of all flat pushing problems will be denoted by $\mathcal{FP}$.

Definition 3: A pushing rearrangement plan is called m-**sequential** if it can be described as a sequence of robot operations, each of which pushes at most m movable objects simultaneously. A pushing problem is called m-sequential if it can be solved by an m-sequential plan. The class of all m-sequential problems will be denoted by $\mathcal{SP}^m$.

Intuitively, $\mathcal{FP}$ represents very easy problems while $\mathcal{SP}^m$ might contain very difficult problems (in terms of finding a rearrangement solution). Indeed, both classes bound our problems' domain on both extremes—the most constrained and the most general (note that $\mathcal{SP}^n = \mathcal{PP}^n$). Being the more general, $\mathcal{SP}^m$ is probably the more interesting class to deal with. However, we found *no* previous *motion planning* work that addresses $\mathcal{SP}^{m>1}$-like problems[2]. In the same spirit, and because of the high complexity involved, this paper will deal only with $\mathcal{SP}^1$ problems. We hope that the formalization of $\mathcal{SP}^m$ may lay the ground for future research in that direction An example for an $\mathcal{SP}^2$ problem is given in Fig. 4.

Following the discussion above, we next define a series of classes for problems of increasing difficulties, all are subsets of $\mathcal{SP}^1$.

Definition 4: A pushing rearrangement plan is called **linear** if it can be described as a sequence of nonpreemptive pushings. A pushing problem is called linear if it can be solved by a linear plan. The class of all linear problems will be denoted by $\mathcal{LP}$.

Both $\mathcal{FP}$ and $\mathcal{LP}$ try to constrain the problem by forcing it to be decomposed of similar *serializable* rearrangement *subgoals* [13]. However, $\mathcal{FP} \subset \mathcal{LP}$ since an appropriate permutation of nonpreemptive pushings is not known a priori for linear problems. Examples of flat and linear problems are given in Fig. 5. The first practical algorithm suggested in this paper is designed for linear problems.

The next step in classifying our problems is focused at $\mathcal{SP}^1 \setminus \mathcal{LP}$, a class which contains problems in various degrees of difficulty. Following our guideline to define classes by characteristics of rearrangement plans, we formalize a rearrangement approach which seems to apply for many practical cases. Given a non linear rearrangement task, it if often

[2]This common practice doesn't hold for the assembly planning literature (e.g., [9]), where subassemblies are often used. Nevertheless, even in that context, assembly plans are not general $\mathcal{SP}^{m>1}$ sequences, since once they are assembled, subassemblies are usually not broken. A more general approach, and one that is most appropriate for $\mathcal{SP}^{m>1}$ problems, should allow to break down a subassembly and manipulate its individual components, if such a manipulation is proven useful. Such a general $\mathcal{SP}^m$ manipulation was illustrated in Fig. 2.

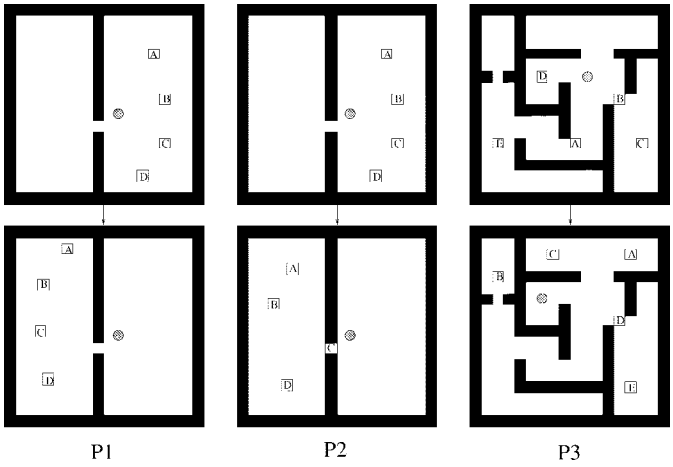


Fig. 5. A Flat (P1) and two Linear (P2, P3) problems.

possible to reduce it to a linear problem by applying a small "*perturbation*" to the initial configuration of some of the movable objects. More precisely, given a 1-sequential problem $Q_0 \overset{\mathcal{SP}^1}{\rightarrow} Q_G$, it is often possible to reduce it to a sequence of two linear problems $Q_0 \overset{\mathcal{LP}}{\rightarrow} Q_{\text{perturb}} \overset{\mathcal{LP}}{\rightarrow} Q_G$, with Q_{perturb} being rather "close" to Q_0. Examples of such reduction are given in Fig. 6.

Formalizing this approach, we define the set $\mathcal{LP}\epsilon$ as follows:

Definition 5: A pushing rearrangement plan is called ϵ-**linear** if it can be described as a sequence of **two** linear plans $Q_0 \overset{\mathcal{LP}}{\rightarrow} Q_{\text{perturb}} \overset{\mathcal{LP}}{\rightarrow} Q_G$, with $Q_{\text{perturb}} \in \epsilon$-neighborhood$(Q_0)$. A pushing problem is called ϵ-linear if it can be solved by an ϵ-linear plan. The class of all ϵ-linear problems will be denoted by $\mathcal{LP}\epsilon$.

The ϵ-**neighborhood** of a composite configuration $Q_0 = (q_r^0, q_{m_1}^0, \cdots, q_{m_n}^0)$ is defined as its neighborhood of "radius" ϵ:

$$\epsilon\text{-neighborhood}\left(q_r^0, q_{m_1}^0, \cdots, q_{m_n}^0\right) \triangleq \left\{\left(q_r, q_{m_1}, \cdots, q_{m_n}\right) : \left|q_{m_i} - q_{m_i}^0\right| \leq \epsilon \quad \forall 1 \leq i \leq n\right\}.$$

Each value of ϵ defines a different class of pushing problems. It is clear that $\mathcal{LP}\epsilon_0 \subset \mathcal{LP}\epsilon_1$ if $\epsilon_0 < \epsilon_1$ and that $\epsilon = 0$ creates the class $\mathcal{LP}$ of linear problems. It is also clear that $\mathcal{SP}^1 \setminus \mathcal{LP}_\infty \neq \emptyset$.

The following generalization of $\mathcal{LP}\epsilon$ does allow us to fill up the gap between $\mathcal{LP}_\infty$ and $\mathcal{SP}^1$:

Definition 6: A pushing rearrangement plan is called ϵ^k-**linear** (pronounced ϵ-k-linear) if it can be described as a sequence of k linear plans $Q_0 \overset{\mathcal{LP}}{\rightarrow} Q_1 \overset{\mathcal{LP}}{\rightarrow} Q_2 \overset{\mathcal{LP}}{\rightarrow} \cdots \overset{\mathcal{LP}}{\rightarrow} Q_{k-1} \overset{\mathcal{LP}}{\rightarrow} Q_G$, with $Q_i \in \epsilon$-neighborhood(Q_{i-1}). A pushing problem is called ϵ^k-linear if it can be solved by an ϵ^k-linear plan. The class of all ϵ^k-linear problems will be denoted by $\mathcal{LP}\epsilon^k$.

Naturally, $\mathcal{LP}\epsilon^2 \triangleq \mathcal{LP}\epsilon$. It is also clear that any $\mathcal{SP}^1$ rearrangement plan of length K (i.e., a plan of K robot steps) can be described as a $\mathcal{LP}\epsilon^k$ plan, for $k \leq K$.

Concluding this section, a general view of our classification is illustrated in Fig. 7.

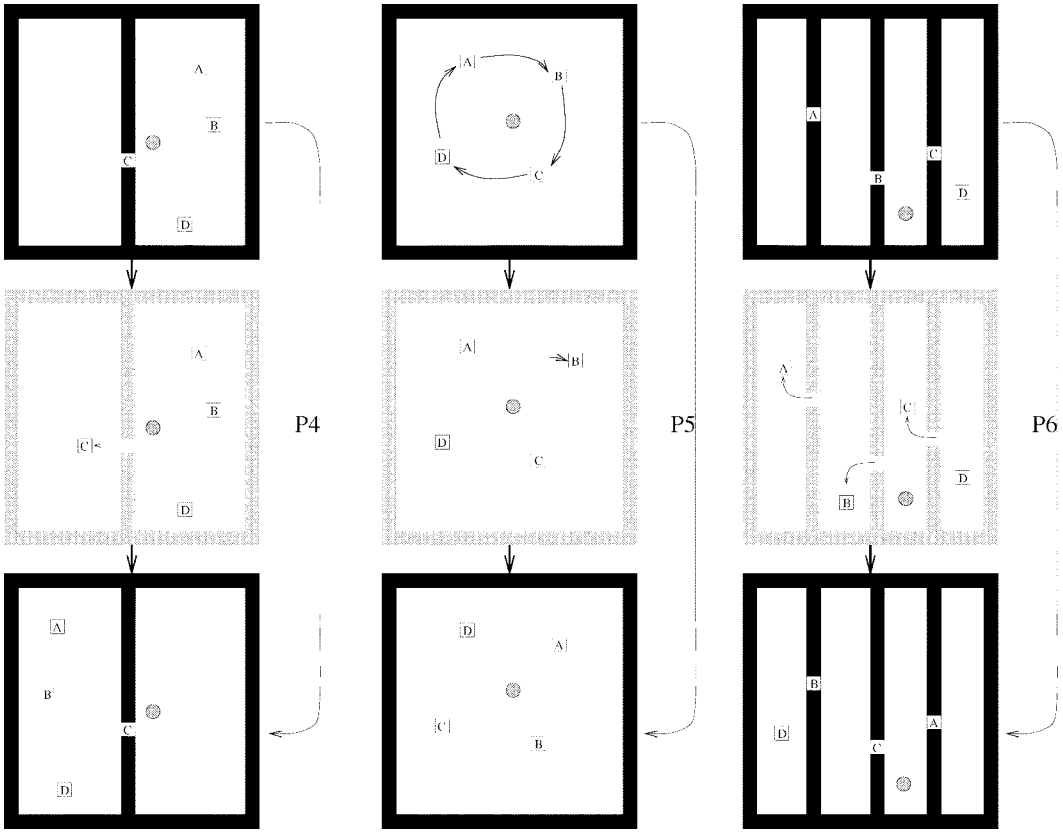


Fig. 6. $\mathcal{LP}\epsilon$ problems. All figures include an illustration of Q_{perturb} (middle row, shaded), from which the problem becomes linear.

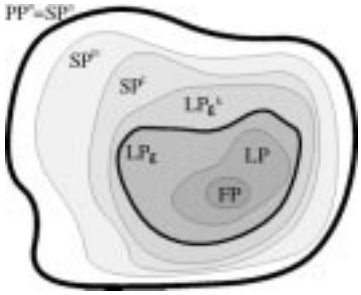


Fig. 7. Classification of $\mathcal{PP}^n$.

VI. Dealing With $\mathcal{LP}$ Problems

Given a flat pushing problem as $P1$, one can easily find an appropriate pushing plan that solves it. Assuming that n movables are involved, choose an arbitrary permutation of its elements and then create an independent plan for each, while treating all others as stationary obstacles. Each such sub-plan can be constructed using our basic pushing planner [6] (see Section IV). Since it operates on one movable object at a time, each run of the basic pushing planner is constant in time and space (for a predetermined discretization rate), hence we get $O(n)$ total time complexity and $O(1)$ total space complexity.

Although solved easily, flat problems are less common in real world scenarios. In this section, we consider the class of linear problems and provide a practical, complete planner that solves them.

A. Planning Issues and Overview

Given a linear problem, a naive way of solving it would be to scan all possible non preemptive pushing permutations, looking for one that solves the problem. The pushing of each movable object in a permutation can be planned with the planner of Section IV. However, this naive approach has two main problems which we need to address.

The first problem is the existence of $n!$ permutations which yields an exponential search time in the average case. A remedy might come from the following observation. Many linear rearrangement problems contain *inherent* constraints on the order of non preemptive pushings that may solve them. For example, let us consider problem P2 in Fig. 5. Considering movables A and C alone (i.e., when ignoring all other movable objects), we can confidently say that C must not be pushed before A. Hence, any permutation in which C stands before A cannot be a solution. Similarly, we can examine *any* two movables and extract a list of precedence constraints which can be represented in a graph. After constructing such a precedence graph in polynomial time, it can be used to filter out many illegal permutations, using a topological-sort like procedure. While worst case behavior remains exponential, it appears that this approach allows many linear problems to be solved out within practical time (see Section IX).

A second problem, which we call the *contact-mode* problem, is unique to the pushing domain. As mentioned above, each non preemptive pushing in a given permutation can be planned with an underlying planner, as the one in Section IV. However,

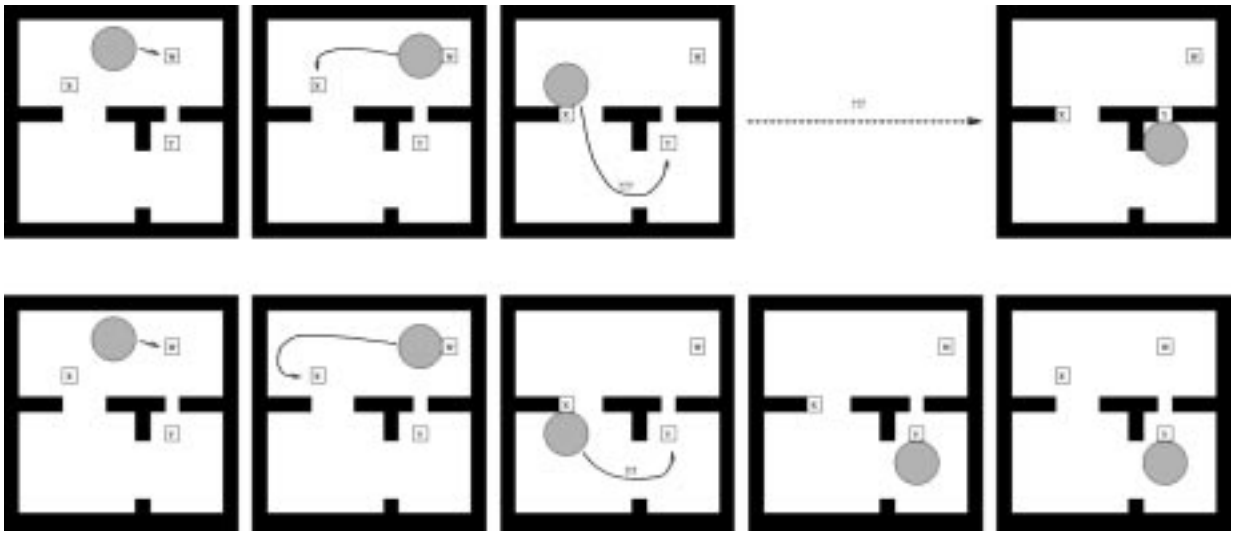


Fig. 8. Contact mode problems. While executing $W \rightarrow X \rightarrow Y$, the contact mode configuration of the pusher with movable X, prior to handling movable Y, is crucial (in that example, pushing is allowed only from the middle point of a movable's edge).

a careless usage of such a planner might result in a planning failure, even if the given permutation is a valid solution (see Fig. 8).

B. Precedence Graphs of Rearrangement Problems

Formalizing the above discussion, let us first define the graph of precedence constraints which we extract from a given problem.

Definition 7: Given an initial configuration Q_0 and a set of events $S = \{E_{\mathcal{M}_1}, \cdots, E_{\mathcal{M}_n}\}$ which should all occur in order to achieve some goal configuration Q_G, a **precedence graph of** S **under** Q_0 **and** Q_G is a directed graph $G_P(S, Q_0, Q_G) = (V, E)$ which has a node for each possible event (i.e., $V = S$) and a directed edge from $E_{\mathcal{M}_i}$ to $E_{\mathcal{M}_j}$ if $E_{\mathcal{M}_i}$ **must precede** $E_{\mathcal{M}_j}$ in order to achieve Q_G, regardless of any other event.

A directed edge from $E_{\mathcal{M}_i}$ to $E_{\mathcal{M}_j}$ **does not** guarantee a successful occurrence of $E_{\mathcal{M}_j}$ after $E_{\mathcal{M}_i}$ (such a success might depend on a third event or some global context), but it certainly indicates that $E_{\mathcal{M}_j}$ must not happen **before** $E_{\mathcal{M}_i}$, regardless of any other event or criterion. A precedence graph which expresses all inherent precedences, between all pairs of events in a given problem will be called **maximal** and will be denoted by $G_P^M(S, Q_0, Q_G)$. Naturally, $E_{\mathcal{M}_i}$ in the definition corresponds to the non preemptive pushing of movable $\mathcal{M}_i$. Intuitively, unless $G_P^M(S, Q_0, Q_G)$ is edgeless, the corresponding problem cannot be flat. Similarly, if $G_P^M(S, Q_0, Q_G)$ is not a DAG, the problem cannot be linear. This last observation is formally expressed in the following two equivalent Lemmas ($d_{\text{in}}(\cdot)$ denotes the in-degree of a node):

Lemma 1: Given a rearrangement problem P_0 and its maximal precedence graph $G_P^M(S, Q_0, Q_G)$

$$P_0 \in \mathcal{LP} \Rightarrow \exists\, E_{\mathcal{M}_i} \in E \quad \text{s.t.} \quad d_{\text{in}}(E_{\mathcal{M}_i}) = 0.$$

Lemma 2: Given a pushing problem P_0 and its maximal precedence graph $G_P^M(S, Q_0, Q_G)$

$G_P^M(S, Q_0, Q_G)$ has a directed cycle $\Rightarrow P_0 \in \mathcal{PP} \setminus \mathcal{LP}$.

Proof: Assume $d_{\text{in}}(E_{\mathcal{M}_i}) > 0$. Then there must be some $\mathcal{M}_j \neq \mathcal{M}_i$ that precedes $\mathcal{M}_i$ in *any* permutation that solves P_0. In practical terms, no permutation that solves P_0 can have $\mathcal{M}_i$ as its first element. If every node of $G_P^M(S, Q_0, Q_G)$ maintains that property then **no** movable can be the first in any non preemptive pushing permutation that solves P_0, i.e., no linear solution exists. □

In order to algorithmically construct the maximal precedence graph of a given problem, we need two auxiliary functions. The first, which we call FREEZE$(\mathcal{I}, \mathcal{M}_i, q)$, freezes a movable object at a given configuration q, and returns a new environment $\mathcal{I}'$ which incorporates that change. The second, IsPushable(), is formally defined as follows:

Definition 8: Given $\mathcal{I}$, $\mathcal{R}$, and one movable $\mathcal{M}$, the predicate **IsPushable**$(\mathcal{R}, \mathcal{I}, \mathcal{M}, q_0, q_G)$ indicates whether $\mathcal{M}$ is pushable from q_0 to q_G, i.e., whether or not there exists a pushing plan that brings $\mathcal{M}$ from q_0 to q_G.

Using these two functions, the maximal precedence graph of a given problem can be constructed by the following procedure. Note again that this graph, and its precedence constraints, are extracted *from* the problem, rather than imposed on it, as happens in many other search problems

MPG$(\mathcal{I}, \mathcal{R}, S, Q_0, Q_G)$
for each element of $\{(\mathcal{M}_i, \mathcal{M}_j) : \mathcal{M}_i, \mathcal{M}_j \in S \wedge i \neq j\}$**do**
begin
$q_0 \leftarrow \Pi_{M_j}(Q_0)$
$q_G \leftarrow \Pi_{M_j}(Q_G)$
$\mathcal{I}' \leftarrow$ FREEZE$(\mathcal{I}, \mathcal{M}_i, \Pi_{M_i}(Q_0))$
if (IsPushable$(\mathcal{R}, \mathcal{I}', \mathcal{M}_j, q_0, q_G) ==$ FALSE) **then**
construct the edge $E_{\mathcal{M}_i} \rightarrow E_{\mathcal{M}_j}$
$\mathcal{I}' \leftarrow$ FREEZE$(\mathcal{I}, \mathcal{M}_i, \Pi_{M_i}(Q_G))$
if (IsPushable$(\mathcal{R}, \mathcal{I}', \mathcal{M}_j, q_0, q_G) ==$ FALSE) **then**
construct the edge $E_{\mathcal{M}_j} \rightarrow E_{\mathcal{M}_i}$
end.

Each iteration of the above routine treats *only two* movables, one is considered as an obstacle while a pushing plan is searched for the other. Two situations are checked for each such combination—first in which $\mathcal{M}_i$ is frozen in its initial

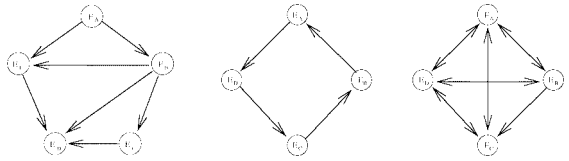


Fig. 9. Maximal precedence graph of problems P3, P5, and P6.

configuration, and second in which $\mathcal{M}_i$ is frozen in its goal configuration. For each of the two situations a pushing path is searched for the "free" movable $\mathcal{M}_j$ and an appropriate precedence edge is constructed upon a failure.

The observant reader would note that IsPushable(), by definition, is not concerned with the initial or goal configuration of the *robot*. Careful examination shows that this is exactly the case when we build the maximal precedence graph. As the two events $E_{\mathcal{M}_i}$ and $E_{\mathcal{M}_j}$ are considered in isolation, we cannot predict the exact a priori configuration of the robot before actually trying to push a specific movable, nor can we know the best posteriori configuration for it. Thus, the best we can do is to ignore these configurations and test for an arbitrary pushing path, as done by IsPushable().

The implementation of IsPushable() is fairly a variation of the cost mapping phase of the basic algorithm in Section IV, with both time and space complexities being constant (for a predetermined discretization rate). Consequently, each iteration of the MPG() routine is of constant time complexity, leading to a total complexity of $O(n^2)$, the same as the space complexity of $G_P^M(S, Q_0, Q_G)$. Moreover, most of the computations can be done in parallel as each precedence edge is totally independent of the others. Fig. 9 shows the maximal precedence graphs of problems P2, P5, and P6, as produced by MPG().

As mentioned before, we are going to use $G_P^M(S, Q_0, Q_G)$ to sort the non preemptive pushing events via a topological sort like procedure. While its value in saving search time is not definite, the following discussion shows that the maximal precedence graph has a true potential in dramatically narrowing the search. In order to do that we first observe that since our solution is a *permutation* of the non preemptive pushing events, we don't have to use CCS as our search space. Rather, we can use an alternative, more "compact" space, which we call the *permutation net* and define as follows:

Definition 9: Given a finite set $S = \{m_1, \cdots, m_n\}$, its **permutation net** is a directed, labeled, acyclic graph $N_P(S) = (V, E)$. $V = 2^S$ is the set of all subsets of S. Each two nodes v_i and v_j are connected by the directed edge $v_i \stackrel{m_k}{\rightarrow} v_j$ if $v_i \cup \{m_k\} = v_j$.

A permutation net has $\sum_{i=0}^{n} \binom{n}{i} = 2^n$ nodes and $\sum_{i=0}^{n} i \cdot \binom{n}{i} = n \cdot 2^{n-1}$ edges. It is obvious that such a graph contains exactly one root (the empty subset) and exactly one sink (the whole set), and that any path from the root to the sink is a unique *permutation* of S. Fig. 10 illustrates the permutation net of a four elements set.

Having a linear problem in hand, each node of $N_P(S)$ represents the set of objects already pushed to their goal configuration, and each edge $v_i \stackrel{\mathcal{M}_k}{\rightarrow} v_j$ represents the action of non preemptive pushing of $\mathcal{M}_k$ (in that way, each *edge* of the permutation net corresponds to a specific *node* of the precedence graph). Our problem's solution, if it exists, must reside as a directed path of n edges in $N_P(S)$ so the problem of finding a linear pushing plan can thus be regarded as finding such a path. Although $N_P(S)$ seems to be a relatively limited search space when compared to CCS, it is still exponentially large in n, which makes all forms of exhaustive search unrealistic. While $G_P^M(S, Q_0, Q_G)$ cannot make the size of $N_P(S)$ polynomial, it does allow for major reduction in the search space, as shown by the following Lemma (see [19] for the proof):

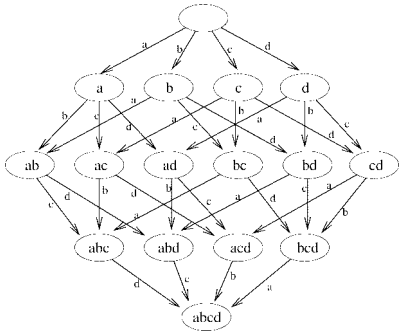


Fig. 10. The $N_P\{a, b, c, d\}$ permutation net.

Lemma 3: Each precedence graph's edge marks one fourth of the number of permutation net's nodes and at least one fourth of its edges as inadmissible for the search.

Unfortunately, different precedence edges do not necessarily mark distinct sets of inadmissible elements of the permutation net. It is clear, however, that $n-1$ precedence edges, forming a Hamiltonian path, transform the permutation net into a linear list which preserves the only permutation that might solve the problem.

Fig. 11 illustrates the result of applying the maximal precedence graph of P2 to its permutation net. In addition to a meaningful reduction in size, it is important to note that in this case the remaining net contains *only* solution permutations. In other words, searching the remaining net in a depth first search (DFS) like approach will require no backtracking. Although this property is not guaranteed, we found that many practical problems requires no or only little backtracking while searching the *reduced* permutation net, allowing a very fast planning. Naturally, if optimal solutions are required, and memory resources are available, the DFS can be replaced with a breadth first search (BFS) over the reduced permutations net. In doing that, one has to take into consideration that the precedence graph might contain no or only small number of edges, leading to relatively large size reduced permutation net.

C. The LPLAN Algorithm

Following the discussion above, we can now derive a practical linear planner as a precedence guided DFS over the permutation net of a given problem. Instead of building the reduced net *before* the search, we will do it *during* the search—by preceding each step of the DFS with an appropriate test over the maximal precedence graph. Following Lemma 1, such a test should allow the DFS to move down the

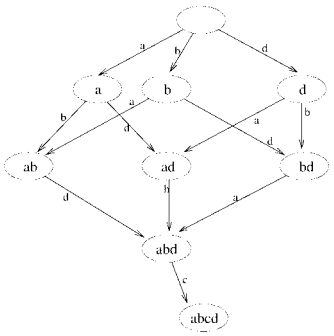


Fig. 11. The reduced permutation net of P2.

permutation net only through those edges whose corresponding node in the precedence graph has zero in-degree. After going down such an edge and planning the non preemptive pushing of the chosen movable (say $\mathcal{M}_i$), we can safely remove $E_{\mathcal{M}_i}$ and its corresponding (in-going and out-going) edges from the precedence graph, and use the remaining graph as the precedence graph of the remaining sub-problem (the one that excludes $\mathcal{M}_i$). If backtracking is needed, the precedence graph should be reconstructed accordingly.

While the above scheme represents the skeleton of **LPLAN**, the planner for linear problems, it must be further refined to handle the contact-mode problem (see p. 17 of this paper). As mentioned before, the configuration that the robot achieves after pushing one movable might play a crucial role in the success to push the next movable. Furthermore, the effect of the chosen configuration might be noticed only much latter in the pushing permutation. Hence, we found it necessary to consider each of the possible final configurations that the robot can achieve after pushing the current movable. Since the pushing plan for each movable object is done with a basic planner which works in a discretized configuration space, the number of such configurations is finite.

Following is the high level code for LPLAN, the algorithm for solving $\mathcal{LP}$ problems

Algorithm LPLAN $(\mathcal{I}, \mathcal{R}, S = \{\mathcal{M}_1, \cdots, \mathcal{M}_n\}, Q_0, Q_G)$
  $G = (G_V, G_E) \leftarrow$ MPG$(\mathcal{I}, \mathcal{R}, S, Q_0, Q_G)$.
  Perm $\leftarrow$ *NULL*
  Call Plan$(\mathcal{I}, \mathcal{R}, S, Q_0, Q_G, G, 1)$
end.

The main part of the initialization phase is the construction of the maximal precedence graph. The core planning routine is the **Plan()** procedure, which recursively handles the planning of the next nonpreemptive C-path. As mentioned before, each such nonpreemptive pushing C-path corresponds to following one edge of the permutation net. Algorithm continues on the bottom of the next page.

Plan() receives the current configuration achieved so far (Q_{current}) and the precedence graph of the remaining sub-problem (G). It first filters out all movables that, according to the precedence graph, must not be manipulated at the current level. For each remaining movable, Plan() tries to plan a pushing C-Path from its initial configuration to its goal configuration. The non preemptive pushing planning of each movables is handled by an underlying planner, referenced here as the PushCPath() function, and defined formally as follows.

Definition 10: Given $\mathcal{I}$, $\mathcal{R}$, and one movable $\mathcal{M}$, the function **PushCPath**$(\mathcal{R}, \mathcal{I}, \mathcal{M}, Q_0, Q_G)$ returns either

1) a pushing C-path that represents the pushing of $\mathcal{M}$ with $\mathcal{R}$ from Q_0 to Q_G (both belong to $CS_R \times CS_M$);
2) NULL—if no such pushing C-path exists.

While PushCPath() can be implemented with *any* planner that agrees with the above definition, we use the planner from our previous work [6]. Although its complexity is exponential with the number of objects, using it for one movable alone yields a $O(1)$ complexity.

Finally, in order to avoid the contact-mode problem, LPLAN considers *every* possible contact configuration that $\mathcal{R}$ may achieve after pushing the current candidate $\mathcal{M}_C$. If an appropriate C-path is found, Plan() invokes itself recursively.

D. Discussion

The LPLAN search algorithm is guided by the precedence graph's edges which impose only *necessary* conditions and may be insufficient. However, the use of the precedences information is embedded within a depth-first like search over the permutation net so a linear solution is guaranteed to be found, if one exists. Naturally, since our underlying planner works in a discretized configuration space, LPLAN is only *resolution complete*.

Following Lemma 3, it is clear that additional precedence constraints limit the search further and allow LPLAN to exhibit better worst-case behavior. On the other hand, some easy looking linear problems that have precedence graphs with no edges, might cause LPLAN to work harder. Such a degenerate graph is expected, for example, for every problem which incorporates **several** narrow passages connecting two adjacent meadows. If all those passages are occupied by movable objects during one of the initial or goal configurations, then the problem becomes nonflat and LPLAN becomes no more than a DFS, which is expected to backtrack until encountering a solution.

A necessary condition for LPLAN to succeed is the ability to push each movable *alone* (i.e., while ignoring all others) to its goal. Such a test can be realized in linear time as part of the high level code of LPLAN, using the IsPushable() function. However, it is easy to see that when a movable is trapped in such a way, the precedence graph of the problem is guaranteed to contain many 2-edges cycles, all involving the same "problematic" object. Following Lemma 2, we shall conclude that the problem is not "linearly solvable".

Additional Improvements: While the above formal code of Plan() provides the core algorithm, it does not specify several important details that prevent LPLAN from producing inefficient pushing plans. Most notable is the loosely defined order by which LPLAN selects candidate movables from the *CURRENT* set. While other criteria are possible too, we choose to sort the candidates in *CURRENT* by their distance from $\Pi_R(Q_{\text{current}})$. This allows the planner to prefer close movables, minimizing the "length" of the planned C-path.

VII. Dealing with $\mathcal{LP}\epsilon$ Problems

Given an ϵ-linear pushing rearrangement problem, its solution can no longer be represented as a path in the permutation

net, unless an appropriate *perturbation* has preceded it. Following Definition 5, the perturbed configuration Q_{perturb} should be searched in a small neighborhood of the initial configuration. However, even if we are using a discretized version of the configuration space, this neighborhood is exponential in size, making any kind of exhaustive search impractical. The following two subsections propose a practical, heuristic way, of finding Q_{perturb}. Instead of searching the whole ϵ-neighborhood of Q_0, we try to decide which movable should be perturbed, and incrementally search for an appropriate perturbation for each of these candidates. Since the projection of the ϵ-neighborhood on each CS_{M_i} is constant in size (for any given ϵ), the total searching time becomes polynomial with the number of movables. A complete complexity discussion is included in Section IX.

A. Choosing Movables to be Perturbed

Let us examine two specific *partial* precedence graphs, which we call the *initial* precedence graph and the *goal* precedence graph. The initial precedence graph, denoted by $G_P^I(S, Q_0, Q_G)$, represents all precedences following from the *initial* configuration. Similarly, the goal precedence graph, denoted by $G_P^G(S, Q_0, Q_G)$, represents all precedences following from the *goal* configuration. The routines used to construct those graphs are both derived from the MPG() routine (Section VI-B), each preserving only the relevant tests. On the bottom of the next page is the code for building the initial precedence graph. The goal precedence graph is built similarly. Using $G_P^I(E, Q_0, Q_G)$, the initial precedence graph, we define the following measure:

$$\beta(\mathcal{M}_i) = d_{\text{out}}\big(E_{\mathcal{M}_i} \in G_P^I(S, Q_0, Q_G)\big).$$

Literally, $\beta(\mathcal{M}_i)$ counts the number of movables such that $\mathcal{M}_i$ stands in their way to the goal. For example, the values of $\beta(\mathcal{M}_i)$ for the movables of problems P4, P5, and P6 (from Fig. 6) are presented in Table I.

TABLE I
EXAMPLES OF $\beta(\mathcal{M}_i)$

Problem	P4				P5				P6			
Object	A	B	C	D	A	B	C	D	A	B	C	D
$\beta(\cdot)$	0	0	3	0	1	1	1	1	2	3	2	0

Having the values of $\beta(\mathcal{M}_i)$, we heuristically mark the candidates for perturbation by looking for movables whose $\beta(\mathcal{M}_i) > 0$. While this criterion is effective, it is somewhat too strong, since sometimes we can achieve an appropriate perturbation by affecting only a subset of those candidates (e.g., in problem P5 only one movable should be perturbed in order to make the problem linear). Consequently, we build the perturbation in an iterative manner. During each iteration we choose the movable $\mathcal{M}_i$ with the maximal $\beta(\mathcal{M}_i)$ and perturb its configuration. Then we update both precedence graphs and move to the next iteration. The loop is terminated as soon

---

Procedure Plan $(\mathcal{I}, \mathcal{R}, S, Q_{\text{current}}, Q_{\text{goal}}, G = (G_V, G_E)$, *Level*)
  CURRENT $\leftarrow \{\mathcal{M}_i$: $E_{\mathcal{M}_i} \in G_V \wedge d_{\text{in}}(E_{\mathcal{M}_i}) = 0\}$.
  if (*CURRENT* $== \emptyset$) **then**
    return **FAILURE**
  for each $\mathcal{M}_C \in$ *CURRENT* **do**
    Perm[*Level*] $\leftarrow \mathcal{M}_C$
    $G' = (G'_V, G'_E) \leftarrow (G_V \setminus \{E_{\mathcal{M}_C}\},$
    $\{\ \ e \in G_E$: e not connected to $E_{\mathcal{M}_C}\})$
    $\mathcal{I}_{\text{tmp}} \leftarrow \mathcal{I}$
    for each $\mathcal{M}_j \in \{\mathcal{M} \in S : \mathcal{M} \neq \mathcal{M}_C\}$ **do**
      $\mathcal{I}_{\text{tmp}} \leftarrow$FREEZE$(\mathcal{I}_{\text{tmp}}, \mathcal{M}_j, \Pi_{M_j}(Q_{\text{current}}))$
    if (*Level* $< n$) **then**
      CONTACTS $\leftarrow$All-Contacts$(\mathcal{R}, \mathcal{M}_C, \Pi_{M_C}(Q_{\text{goal}}))$
    else
      CONTACTS $\leftarrow (\Pi_R(Q_{\text{goal}}), \Pi_{M_C}(Q_{\text{goal}}))$
    FROM $\leftarrow (\Pi_R(Q_{\text{current}}), \Pi_{M_C}(Q_{\text{current}}))$
    for each configuration *GOAL* $\in$ *CONTACTS* **do**
      PATH $\leftarrow$ PushCPath$(\mathcal{R}, \mathcal{I}_{\text{tmp}}, \mathcal{M}_C, FROM, GOAL)$
      if (*PATH* $==$ *Null*) **then** continue to next iteration
      $Q_{\text{new}} \leftarrow Q_{\text{current}}$
      $\Pi_R(Q_{\text{new}}) \leftarrow \Pi_R(GOAL)$.
      $\Pi_{M_C}(Q_{\text{new}}) \leftarrow \Pi_{M_C}(Q_{\text{goal}})$
      $\mathcal{I}_{\text{tmp}} \leftarrow$FREEZE$(\mathcal{I}, \mathcal{M}_C, \Pi_{M_C}(Q_{\text{goal}}))$
      if (Plan$(\mathcal{I}_{\text{tmp}}, \mathcal{R}, S \setminus \{\mathcal{M}_C\}, Q_{\text{new}}, Q_{\text{goal}}, G',$ *Level* $+ 1) =$ **SUCCESS**) **then**
        return **SUCCESS**
    end for
  end for
  return **FAILURE**
end

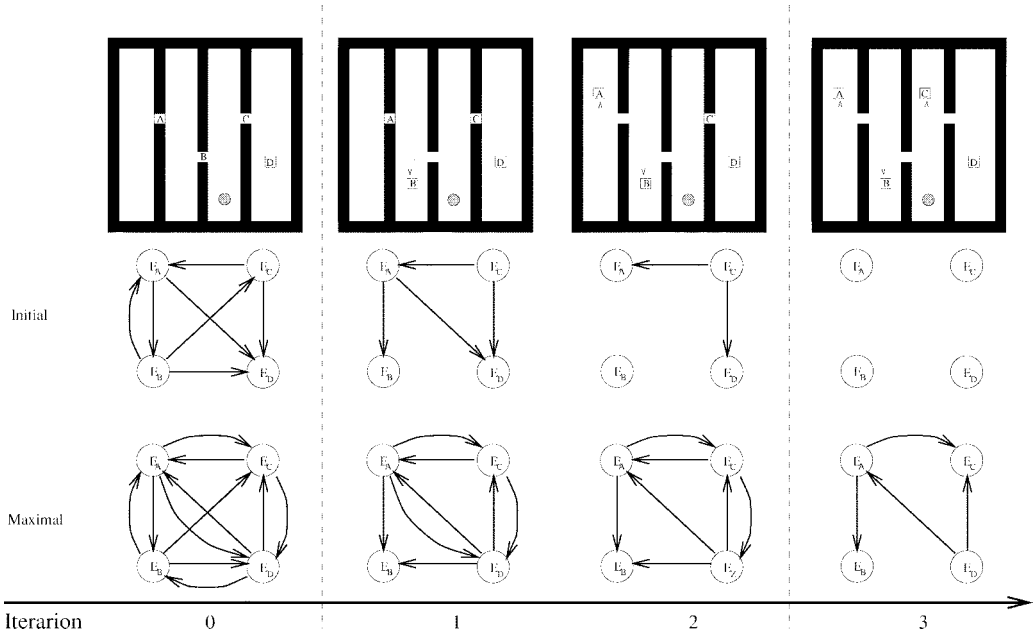


Fig. 12. Running Perturb-Configuration() on problem P6. Applying LPLAN to the result will yield the linear solution $(D \rightarrow A \rightarrow B \rightarrow C)$.

as a cycle-free maximal precedence graph is obtained or no appropriate perturbation is found.

On the bottom of the next page is a description of the routine that creates the perturbed configuration. The *Handled* set keeps track of the movables which were already perturbed. This information is used by the **Perturb()** routine, which introduces the actual perturbation to a given movable and updates both precedence graphs to reflect the new configuration accordingly. The *select*() operator arbitrary selects a member from a given set. As mentioned for LPLAN, this selection may be replaced with a more deterministic one in order to prevent inefficient results. The $\overset{\text{graph}}{\cup}$ operator "unites" two graphs, and it is used in order to build the maximal precedence graph from both partial graphs. Finally, the Reachable() operator returns a set of all movables that the robot can reach from a given configuration. This can be easily calculated by flooding the free space of the robot from its current configuration. Fig. 12 demonstrates the action of Perturb-Configuration() on problem P6. For P6, success is achieved only after the initial precedence graph becomes edge-less. However, this is not a necessary condition, and termination can be obtained earlier. After creating the perturbation, we are supposed to be left with a cycle-free maximal precedence graph. Such a graph, and its corresponding rearrangement problem, can now be given to LPLAN in order to check whether a linear plan can solve it.

B. Introducing an Appropriate Perturbation

After a candidate was chosen, it should be assigned an appropriate perturbation. Definition 5 constrains the perturbation to a well defined neighborhood, yet not every configuration in that neighborhood can serve for our purpose. The required perturbed configuration should fulfill the followings conditions.

1) It should open the way for all other movables which must precede $\mathcal{M}_{\text{cand}}$ when pushed to their goal, and whose path to the goal is currently blocked by $\mathcal{M}_{\text{cand}}$.
2) It must not create new blocks for movables which are already able to reach their goal.
3) It must be realizable by *pushing* $\mathcal{M}_{\text{cand}}$ from its current configuration.
4) It must not be a trap point (see Section I), unless it coincides with the goal configuration of $\mathcal{M}_{\text{cand}}$.

Given ϵ, and assuming a discretized configuration space, the ϵ-neighborhood of the initial configuration contains no more than $O(\epsilon^{\mathcal{M}'s\ \text{DOF}})$ configurations of $\mathcal{M}_{\text{cand}}$. We currently choose to check all those configurations although it may be possible to use heuristics to focus only on some of them. In any case, many of those configurations can be effortlessly filtered out during the search as they represent collisions with obstacles or they cannot be realized by pushing. Furthermore, we build the list of ϵ-neighborhood configurations while sorting them according to their "pushing distance" from the initial

---

IPG$(\mathcal{I}, \mathcal{R}, S, Q_0, Q_G)$
  for each element of $\{(\mathcal{M}_i, \mathcal{M}_j)$: $\mathcal{M}_i, \mathcal{M}_j \in S \wedge i \neq j\}$ **do begin**
    $q_0 \leftarrow \Pi_{M_j}(Q_0)$
    $q_G \leftarrow \Pi_{M_j}(Q_G)$
    $\mathcal{I}' \leftarrow$FREEZE$(\mathcal{I}, \mathcal{M}_i, \Pi_{M_i}(Q_0))$
    if (IsPushable$(\mathcal{R}, \mathcal{I}', \mathcal{M}_j, q_0, q_G) ==$ FALSE) **then**
      construct the edge $E_{\mathcal{M}_i} \rightarrow E_{\mathcal{M}_j}$
  end

configuration of $\mathcal{M}_{\text{cand}}$. This can be done by a *forward* propagation of a cost wave function in $CS_R \times CS_{M_{\text{cand}}}$, which originates from the current composite configuration of $\mathcal{R}$ and $\mathcal{M}_{\text{cand}}$ (when all other movables are frozen during this propagation). Since only ϵ-neighborhood's configurations should be considered, the propagation can terminate at any configuration which lies farther than ϵ.

Every configuration surviving the initial filtering should be checked for consistency with the other requirements. This can be done by updating both the initial and goal precedence graphs in accordance with the currently checked configuration, an action which can be realized in *linear time*. There is no need to build both graphs from scratch since only those precedence edges which enter or leave $E_{\mathcal{M}_{\text{cand}}}$ might have been changed. After updating the graphs, the evaluation of the current configuration is primarily done by checking whether $\beta(\mathcal{M}_{\text{cand}})$ became zero. However, this is an insufficient condition. Additional tests must be considered in order to verify that the perturbed configuration didn't invalidate previous perturbations. This can be done by checking the updated *maximal* precedence graph (obtained by a union of G^I and G^G) and looking for cycles between $E_{\mathcal{M}_{\text{cand}}}$ and the nodes of previously handled movables (stored in the *Handled* set). We choose to test only for cycles of two nodes in order to keep this test linear in complexity. Empirically, this still allows good detection for most cases.

C. The ELPAN Algorithm

Following the subsections above, our practical algorithm for ϵ-linear pushing rearrangement problems, **ELPLAN**, is a concatenation of **Perturb-Configuration()** and **LPLAN**

Algorithm ELPLAN $(\mathcal{I}, \mathcal{R}, S = \{\mathcal{M}_1, \cdots, \mathcal{M}_n\}, Q_0, Q_G)$
  $Q_{\text{perturb}} \leftarrow$ Perturb-Configuration$(\mathcal{I}, \mathcal{R}, S, Q_0, Q_G)$
  if $Q_{\text{perturb}} \neq$ *NULL* **then**
    LPLAN$(\mathcal{I}, \mathcal{R}, S, Q_{\text{perturb}}, Q_G)$
end.

While LPLAN is complete with regard to linear problems, ELPLAN is opportunistic and may fail for several reasons. First, Perturb-Configuration() may fail due to a bad choice of ϵ. Second, it may succeed and create a perturbation for which the maximal precedence graph is free of cycles, while the problem remains nonlinear. Finally, Perturb-Configuration() does not include any backtracking or lookahead. Hence, it might choose a "wrong perturbation" for a movable, in such a way that it prevents a later candidate from being perturbed appropriately. We consider a practical method to handle that problem a future research topic.

VIII. Implementation and Results

In this section, we present several simulated planning results of LPLAN and ELPLAN, as obtained with our implemented planner. Since we primarily study issues of high level planning and planning practicality, we find simulations to be as informative as real world experiments. Furthermore, a simulator allows us to avoid many other problems not relevant to our study but such that tend to appear in real world experiments. We do, however, present one real world experiment with a mobile robot and several chairs acting as movable objects.

A. The Planner

The practical pushing planner (PPP) is implemented in C and runs on a Sun 4/460 computer, equipped with a 50 MHz SuperSPARC processor. Since its two supported algorithms, ELPLAN and LPLAN, are hierarchically depended, PPP is not a priori required to decide what class a problem belongs to. Rather, every problem is submitted to ELPLAN which decides whether or not to apply some perturbation and then transfers the result to LPLAN. The input of PPP is a graphical description of the static environment $(\mathcal{I})$, the shape of all movables, and the two composite configuration, Q_0 and Q_G, which define the rearrangement problem. The output of PPP is a full animated rearrangement solution, if exists, including any intermediate motion of the robot which is required while changing contact mode or moving from one movable to another.[3]

[3]While here we give the pushing plans in figures, the interested reader may view PPP's actual animated output at http://www.cs.technion.ac.il/~obs/projects.html#PPP.

Procedure Perturb-Configuration $(\mathcal{I}, \mathcal{R}, S, Q_{\text{init}}, Q_{\text{goal}})$
  $Q_{\text{current}} \leftarrow Q_{\text{init}}$
  $G^I \leftarrow$ IPG$(\mathcal{I}, \mathcal{R}, S, Q_{\text{init}}, Q_{\text{goal}})$
  $G^G \leftarrow$ GPG$(\mathcal{I}, \mathcal{R}, S, Q_{\text{init}}, Q_{\text{goal}})$
  $Handled \leftarrow \emptyset$
  loop
    if $G^I \overset{\text{graph}}{\cup} G^G$ contains no cycles **then**
    return Q_{current} and **SUCCESS**
    $D \leftarrow \text{Max}(\{\beta(\mathcal{M}_i)$: $\mathcal{M}_i \in$Reachable$(\mathcal{R}, \mathcal{I}, S, Q_{\text{current}}) \setminus$ *Handled*$\})$
    if $(D == 0)$ **then** return **FAILURE**
    $\mathcal{M}_{\text{cand}} \leftarrow$ *Select*$(\{\mathcal{M}_i : \beta(\mathcal{M}_i) == D\})$
    $Q_{\text{current}} \leftarrow$ Perturb$(\mathcal{M}_{\text{cand}}, \mathcal{I}, \mathcal{R}, S, Q_{\text{current}}, Q_{\text{goal}}, G^I, G^G,$ *Handled*$)$
    if $(Q_{\text{current}} ==$ *NULL*$)$ **then** return **FAILURE**
    Handled $\leftarrow$ *Handled* $\cup \{\mathcal{M}_{\text{cand}}\}$
  end loop
end

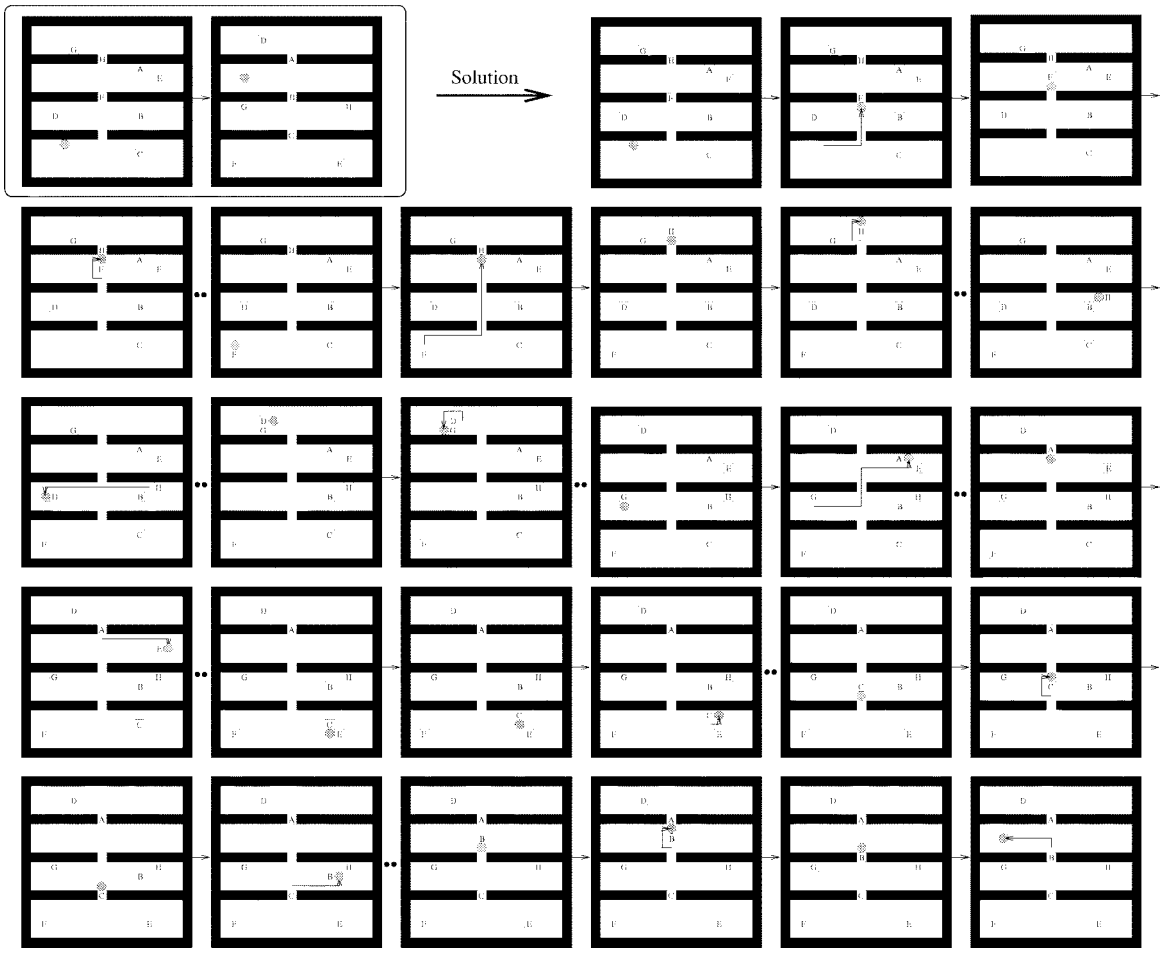


Fig. 13. A linear problem with eight movables and its solution.

PPP uses a discretized space to describe all its geometric objects. This is mainly since it uses a discretized configuration space planner [6] to handle individual objects (see Section IV). Hence, while the planner allows the user to define the shape of each movable, their geometry is subjected to the discretization too.

The robot in PPP's simulations is illustrated as a circular one, occupying one cell of the discretized environment. This was done in order to better discriminate it from the movable objects, and it is not a constraint imposed by the algorithm. The pushing contact modes of the robot with each movable object are derived automatically using a center of friction (COF) which is specified by the user. All pushing manipulations were constrained to translations only due to a similar limitation imposed by the underlying planner. Consequently, all movable objects, as well as the robot, were defined as having 2 DOF. **In no way, however, this is a limitation of the algorithms presented in this paper**. As shown in Fig. 3, limited nonrigid rotational pushing capabilities can be integrated into PPP even with the underlying planner of [6].

B. Simulations Results

We present here selected results of problems that have up to eight movable objects (18 combined DOF). However, problems with up to 32 movable objects (66 combined DOF) were successfully tested too. These results are omitted here due to space limitation and presented only in [6]. All figures show major planning steps of each solution. All planning steps are ordered left to right and top to bottom. Transit paths [15] are marked by arrows. Some planning steps, which correspond to nonpreemptive pushes, were omitted (marked by two dots between adjacent steps).

Fig. 13 shows a linear problem with eight movables and its solution. As this problem is linear, ELPLAN introduced *no* perturbation before calling LPLAN. Planning time for this problem was 26.32 s. The Plan() routine was called nine times during the execution, which means that it backtracked only once.

Fig. 14 shows the solution for problem P4 (see Fig. 6). As expected, $\mathcal{R}$ first handles movable B. It clears the passage and pushes the other three movables to their goal. Only then $\mathcal{R}$ returns to B in order to push it back to its goal configuration. Note the somewhat redundant manipulation of movable B, needed in order to allow $\mathcal{R}$ to reach its own final configuration. For P4, as well as for the other ϵ-linear problems presented here, we choose an ϵ-neighborhood of radius $\epsilon = 3 \cdot \mathcal{R}$'s diameter. The perturbed configuration, from which the problem becomes linear, is marked by a brighter frame. Planning time for this problem was 11.65 s. The Plan() routine was called four times during the execution, which means that no backtracking occurred.

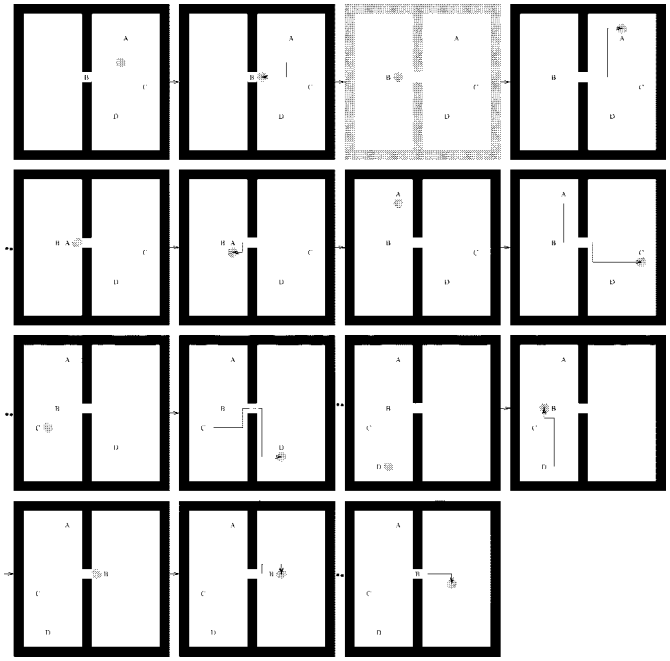


Fig. 14. An $LP\epsilon$ plan for problem P4. The perturbed configuration, from which the solution is linear, is marked by a brighter frame.

Fig. 15 shows the solution for problem P6 (see Fig. 6). Note that if movable C was perturbed to the right, no linear solution could have been found. PPP was able to find that correct perturbation in spite of the fact that $\mathcal{R}$ approached movable C from the left, being able to first push it only to the right. Planning time for this problem was 97 s and the Plan() routine was called seven times during the execution, which means that it backtracked three times.

C. Real World Experiment

While the results presented above are all simulations, we also tested our planner in a realistic scenario using a mobile platform. For this purpose, we integrated PPP with the control environment of the NOMAD-200 mobile robot from Nomadic. The planner used a coarse representation of the lab where the robot carried out a rearrangement plan for five chairs. The robot was directed remotely via a wireless Ethernet link. Some snapshots from one execution are presented in Fig. 16. The pushing plan is illustrated in Fig. 17.

The experiments showed the applicability of the planner for real scenarios involving many movable objects. However, they also emphasized the great importance of sensory feedback. Lacking such a feedback, each experiment must be preceded with an accurate calibration of the robot and the movables, both for position and orientation. Although small odometry errors are acceptable in most cases of indoor *navigation*, this is not the case for pushing. Missing the correct contact mode (for a push) can be critical. While in this study we assumed that the mechanical model is handled separately, there is no doubt that the integration of sensory information with the mechanical model is an issue of major importance as well. We see all these directions as avenues for future research.

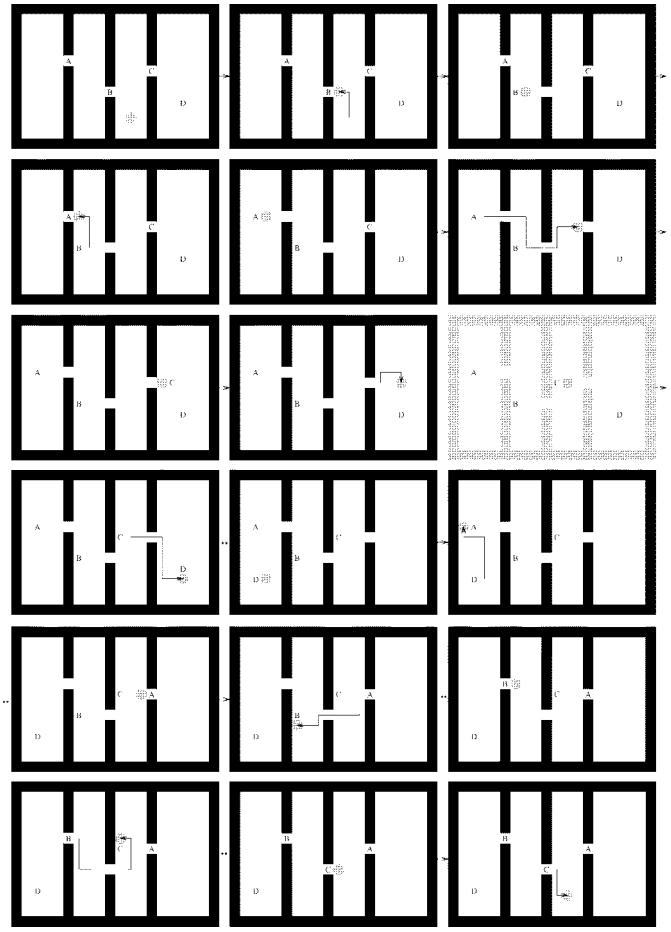


Fig. 15. An $LP\epsilon$ plan for problem P6. The perturbed configuration, from which the solution is linear, is marked by a brighter frame.

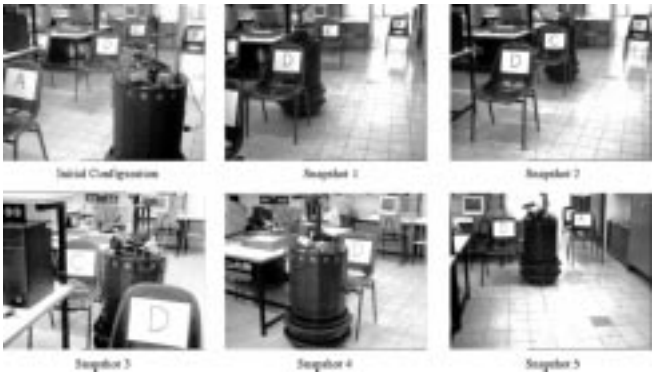


Fig. 16. Some snapshots from a real world experiment.

IX. Complexity, Performance and Practicality

The algorithms presented in this paper are motivated by the need for practical planners. To achieve this goal, both algorithms break the multidimensional problem into a set of low dimensional subproblems, each of which is solved by an underlying planner. While our underlying planner [6] has an exponential time and space complexity in the general case, we have used it to plan pushing C-paths for one movable at a time, having a constant running time for each movable. In addition, the core of both algorithms is a constraints based search. Here

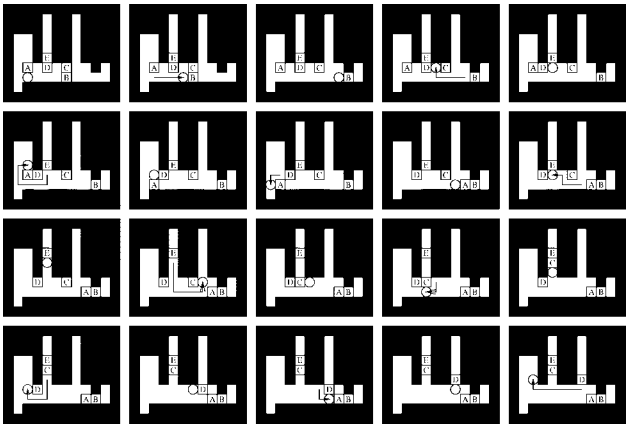


Fig. 17. The pushing plan for the real world experiment. Note that the workspace shown in the previous snapshots was roughly discretized and cells were marked as obstacles even if only part of them contains an actual obstacle.

we do not *impose* constraints *on* the problem but rather *extract* them *from* the problem during a preprocessing phase. Although no new information is added during this phase, the extraction of inherent precedence constraints dramatically narrows the search.

Although the theoretical complexity of LPLAN, and consequently of ELPLAN, is exponential in the number of movables, in many practical cases LPLAN is expected to backtrack *rarely*, requiring only n iterations of the Plan() routine. In these cases, the construction of the maximal precedence graph consumes most of the planning time, leading to a $O(n^2)$ time complexity. ELPLAN's worst case complexity is dominated by that of LPLAN. The preceding phase of Perturb-Configuration() includes at most n iterations, each of which activates a cycle detector, a reachability scanner, and a "perturbation generator." The Perturb() routine dominates the overall performance of Perturb-Configuration(), though its theoretical complexity outperforms the complexity of the cycle detector. Hence, while the *theoretical* complexity of Perturb-Configuration() is $O(n^3)$, its *practical* performance becomes $O(\epsilon^{\mathcal{M}'s\mathrm{DOF}}n^2)$ as a result of the $O(\epsilon^{\mathcal{M}'s\mathrm{DOF}}n)$ complexity of the Perturb() routine.

In order to verify that our planning algorithms exhibit practical behavior, we tested them over a large set of randomly generated problems (see Fig. 18), using simulated environment like the one described in Section VIII. Our problem generator created maze-like problems with "rooms" and "corridors," and chose the initial and goal configurations randomly. In order to reduce the number of trivial problems, problematic areas (e.g., narrow passages) were given a higher probability for being occupied with movables. Furthermore, each generated problem was first submitted to a flat-problem planner and considered non trivial only upon its failure.

Using this mechanism, we tried to confirm not only the claim for *practical planning time*, but the following claims as well:

1) The precedence graph may serve as a decent detector of linear problems, i.e., in most cases, a cycle free precedence graph correctly indicates a linear problem.

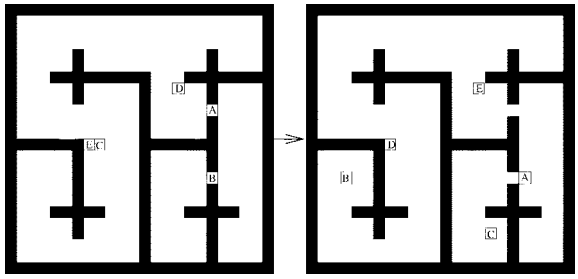


Fig. 18. Example of a randomly generated pushing rearrangement problem.

2) The precedence graph has the ability to filter out most of the erroneous search paths in the permutation net. Consequently, in most cases, the precedence graph can prevent backtracking during the search, allowing the planner to find a solution within practical time limits.
3) Although it is exponential in principle, the time needed to search the reduced permutation net is usually much smaller than the polynomial components of the algorithms. Hence, in practical terms, both algorithms present practical, polynomial behavior.
4) The perturbation approach has a good ability to handle non linear problems.

The following tables summarize a comprehensive test of about 1 000 problems with up to ten movable objects (22 combined DOF). All problems were generated with the same environmental complexity (e.g., density of corridors) and the same discretization rate, while the number of movables was gradually increased. Each problem was allocated up to n^3 Plan() calls. Had the planner reached that limit, a failure was announced.

Table II summarizes some characteristics of the problems. The classification of problems into the appropriate class was based on their maximal precedence graph and the completeness of LPLAN, i.e., every problem with a cycle free precedence graph, *and* which LPLAN had solved, was identified as linear, while all others were counted as nonlinear.

Table II shows that on the average, only 9% of all problems were classified incorrectly by the algorithm. This reinforces claim 1. It is interesting to note that the number of such incorrect classifications was *insensitive* to the number of participating movables. Also shown in Table II is the fact that on the average, 75% of all sequential problems were successfully solved by the perturbation approach, while the total success rate was 85%. One should note that many planning failures were caused by incorrect classification of sequential problems into linear ones. The normalized success rate row presents the expected success rate for a correct classification. While all these results reinforce claim 4 above, it is important to add that as expected, the success rate tends to decrease as more movables share the problem.

Table III represents the second part of our experiment: measuring the role of the precedence graph in the overall performance of PPP. For that purpose we submitted each problem to PPP twice. First with a full calculated precedence graph, and secondly with a null precedence graph. We were mainly interested in three measures—how many times the

TABLE II
EXECUTION CHARACTERISTICS FOR THE RANDOMLY GENERATED PROBLEMS (THREE TO TEN MOVABLES). THE NORMALIZED SUCCESS RATE, CALCULATED AS (% *LINEAR* +% *SEQUENTIAL* ·% *SUCCESS* $_{\text{sequential}}$), REFLECTS THE SUCCESS RATE FOR PERFECT CLASSIFICATION, WITH NO LINEARLY TREATED SEQUENTIAL PROBLEMS

Number Of Movables	3	4	5	6	7	8	9	10	Total Avg.
Percent of Linear problems	80%	76%	66%	82%	56%	62%	74%	70%	71%
Percent of Sequential problems	20%	24%	34%	18%	44%	38%	26%	30%	29%
Planning success rate - total	94%	88%	84%	92%	72%	80%	90%	82%	85%
Linearly treated sequential problems	4%	6%	6%	6%	22%	14%	6%	8%	9%
Planning success rate - sequential problems	88%	77%	71%	80%	73%	75%	80%	54%	75%
Planning success rate - normalized total	98%	94%	90%	96%	88%	91%	95%	86%	92%

TABLE III
PERFORMANCE MEASURES WITH AND WITHOUT PRECEDENCE GRAPHS

Performance with precedence graphs								
Number Of Movables	3	4	5	6	7	8	9	10
Zero Plan() backtracks	97%	93%	93%	91%	89%	81%	85%	77%
1-n Plan() backtracks	2%	4%	4%	4%	7%	7%	5%	4%
n-n^2 Plan() backtracks	1%	0%	0%	4%	0%	5%	3%	4%
More than n^2 Plan() backtracks	0%	3%	3%	1%	4%	7%	7%	15%
Average number of Plan() calls	2.9	5.5	7.3	10.8	7.1	47.0	72.9	172.8
Average number of PushCPath() calls	3.0	7.3	9.9	20.2	9.6	142.7	361.0	823.0
Normalized number of PushCPath() calls	9.0	19.3	29.9	50.2	51.6	198.7	433.0	913.0
Problems that reached resource limit	0%	2%	2%	2%	0%	6%	6%	14%
Performance without precedence graphs								
Number Of Movables	3	4	5	6	7	8	9	10
Zero Plan() backtracks	92%	80%	76%	70%	46%	46%	50%	50%
1-n Plan() backtracks	0%	10%	6%	8%	12%	8%	8%	6%
n-n^2 Plan() backtracks	4%	0%	4%	4%	16%	6%	6%	2%
More than n^2 Plan() backtracks	4%	10%	14%	18%	26%	40%	40%	40%
Average number of Plan() calls	3.8	7.2	20.7	46.1	97.2	184.3	168.6	373.8
Average number of PushCPath() calls	9.0	21	76.5	244.7	532.1	1318.7	1684.4	3234.5
Problems that reached resource limit	4%	2%	10%	16%	22%	28%	20%	32%

TABLE IV
TIME PERFORMANCE OF THE ALGORITHM

Planning time								
Number Of Movables	3	4	5	6	7	8	9	10
Total planning time (seconds)	20.2	35.2	61.2	74.8	112	204	178	289
For the precedence graph calculation	53%	55%	60%	72%	70%	47%	67%	55%
For the PerturbConfiguration() routine	24%	22%	17%	14%	17%	12%	12%	25%
For the Plan() search routine	23%	23%	23%	14%	13%	41%	21%	20%

planner called the Plan() routine, how many times did it call the underlying planner PushCPath(), and what portion of problems reached their resource limit. All of the above, unlike PPP's execution time, are platform independent and represent relevant issues of performance and practicality. Note that for the precedence-graph based performance we have normalized the number of PushCPath() calls by adding all calls during the building of the precedence graphs.

Table III clearly shows that unless a problem is easy, if not trivial, the time penalty for calculating its precedence graph is only a fraction of a brute force search over a non constrained permutation net. The same holds for the number of Plan()'s backtracks, which drops back to zero as soon as a precedence graph participates in the planning. Equally important is the fact that without the precedence graph, an increasing portion of problems reached their resource limit and failed. All these results strongly support claim 2 and the crucial role of the precedence graph in making our method a practical one.

Finally, Table IV represents the average planning time spent on each problem and how was that time divided between the major parts of the computation: the precedence graph construction routine, the PerturbConfiguration() routine and the Plan() search routine. One should note that the (theoretically exponential) search time of Plan() was always much smaller than the (polynomial) time needed to build the precedence graph. In practice, this leads to practical planning time for most problems, as stated in claim 3 above.

X. CONCLUSION

Planning a sequence of pushing manipulations is a PSPACE-hard problem effortlessly solved by humans in everyday life. In this paper, we have suggested a practical approach to handle pushing planning for rearrangement tasks with many movable objects.

The contributions of our study span several topics. We have introduced a novel hierarchical classification of the manipulation problems domain. In our classification, problems are characterized by properties of the plans that can solve them. This allows for a closer, more direct, link between the different classes and their planning algorithms, as well as providing some insight into the algorithms themselves.

Based on this classification, we have presented new manipulation/pushing planning algorithms for problems that belong to two of the defined classes. Our methods break the multidimensional problem into a set of low dimensional subproblems, extract precedence constraints directly from the given problem, and use those to narrow the search. This allows to solve problems of many movable objects with joint number of DOF never handled before.

While being fully compatible with any manipulation, our algorithms are specifically designed to support the particular manipulation of pushing, whose unique characteristics require special care even at the planning level (trap-points, contact-mode problems, etc.). In addition, our algorithms differ from previous works by providing complete description of the pushing plan, including any intermediate motion of the robot between different contact modes, or different movable objects.

Finally, we have presented two tools—the permutation net and the precedence graph—which allow a clear and simple graph representation for different aspects of our problems. Both tools are manipulation independent, hence may be applied to other manipulations as well.

We have implemented our algorithms into PPP, a practical pushing planner. We have tested it in a simulated environment with problems of up to 32 movable objects and a 66 combined DOF. The planner has been integrated with the Nomad-200 environment, and experiments testing the applicability of the planner in real scenarios involving several movable objects have been run.

While PPP was implemented using an underlying planner from our previous study, the planning algorithms themselves (LPLAN and ELPLAN) are totally independent of that choice. In that sense, a better underlying pushing planner may provide more complicated pushing manipulations (such as rotational pushing), hence contributes to PPP's overall performance. In the same spirit, PPP may easily be extended to support grasping simply by plugging a planner that supports grasping manipulation into PushCPath(). Changing PPP in that way, it has easily solved nonlinear manipulation problems, including the frequently addressed Sussman anomaly, and assembly-planning like problems.

The work presented in this paper embodies many directions for future research. On the planning level, these include the development of a practical algorithm for non $\mathcal{LP}\epsilon$ sequential problems, and research into the yet unexplored (in the motion planning context) area of $\mathcal{SP}^{n>1}$ problems. On the implementation level, we have no doubt that many issues must be addressed before a robust, real world rearrangement system can be realized. Developing a method to deal with incomplete knowledge and a scheme to exploit sensory information in order to achieve stable and predictable pushing manipulations are some of these future research directions.

REFERENCES

[1] S. Akella and M. T. Mason, "Posing polygonal objects in the plane by pushing," in *Proc. IEEE Int. Conf. Robot. Automat.*, May, 1992, pp. 2255–2262.

[2] R. Alami, J. P. Laumond, and T. Simeon, "Two manipulation planning algorithms," *The Algorithmic Foundations of Robotics*, K. Goldberg, D. Halpern, J. C. Latombe, and R. Wolson, Eds. Boston, MA: A. K. Peters, 1995.

[3] R. Alami, T. Simeon, and J. P. Laumond, "A geometrical approach to planing manipulation tasks, the case of discrete placements and grasps," *Int. Symp. Robot. Res.*, 1989, pp. 453–463.

[4] J. Barraquand and J. C. Latombe, "A monte-carlo algorithm for path planning with many degrees of freedom," *IEEE Int. Conf. Robot. Automat.*, 1990, pp. 1712–1717.

[5] ———, "Robot motion planning: A distributed representation approach," *Int. J. Robot. Res.*, vol. 6, no. 10, pp. 628–649, Dec. 1991.

[6] O. Ben-Shahar and E. Rivlin, "To push or not to push—Part I. On the rearrangement of movable objects by a mobile robot," CIS Rep. 9516, Technion, Haifa, Israel, July 1995.

[7] P. C. Chen and Y. K. Hwang, "Practical path planning among movable obstacles," *IEEE Int. Conf. Robot. Automat.*, 1991, pp. 444–449.

[8] B. Dacre-Wright, J. P. Laumond, and R. Alami, "Motion planning for a robot and a movable object amidst polygonal obstacles," *IEEE Int. Conf. Robot. Automat.*, May 1992, pp. 2474–2480.

[9] L. S. Homem de Mello and S. Lee, Eds., *Computer-Aided Mechanical Assembly Planning*. Norwell, MA: Kluwer, 1991.

[10] L. Kavraki and J. C. Latombe, "Randomized preprocessing of configuration space for fast path planning," *IEEE Int. Conf. Robot. Automat.*, 1994, pp. 2138–2145.

[11] Y. Koga, T. Lastennet, J. C. Latobme, and T. Y. Li, "Multi-arm manipulation planning," *9th Int. Symp. Automat. Robot. Contruction*, June 1992.

[12] Y. Koga and J. C. Latombe, "On multi-arm manipulation planning," *IEEE Int. Conf. Robot. Automat.*, 1994, pp. 945–952.

[13] R. E. Korf, "Planning as search: A quantitative approach," *Artificial Intell.*, vol. 33, pp. 65–89, 1987.

[14] J. P. Laumond and R. Alami, "A new geometrical approach to planing manipulation tasks, the case of a circular robot and a movable circular object amidst polygonal obstacles," Tech. Rep. 88314, LAAS, Toulouse, France, 1988,

[15] ———, "A geometrical approach to planing manipulation tasks in robotics," Tech. Rep. 89261, LAAS, Toulouse, France, 1989,

[16] K. M. Lynch and M. T. Mason, "Stabe pushing: Mechanics, controllability, and planning," *1st Workshop Algorithmic Foundation Robot.*, 1995.

[17] G. T. Wilfong, "Motion planning in the presence of movable obstacles," in *Proc. ACM Symp. Computat. Geometry ASCG*, 1988, pp. 279–288.

[18] J. D. Wolter, "On automatic generation of assembly plans," *Computer-Aided Mechanical Assembly Planning*, L. S. Homem de Mello and S. Lee, Eds. Norwell, MA: Kluwer, 1991, ch. 11, pp. 263–288.

[19] O. Ben-Shahar and E. Rivlin, "To push or not to push—Part III," CIS Rep., Technion, Haifa, Israel, Oct. 1995.

Ohad Ben-Shahar received the B.Sc. and M.Sc. degrees in computer science from the Technion, Israel Institute of Technology, Haifa, in 1989 and 1996, respectively, and is currently pursuing the Ph.D. degree in computer science at Yale University, New Haven, CT.

His research interests are computer vision and robot motion/manipulation planning.

Ehud Rivlin received the B.Sc. and M.Sc. degrees in computer science and the M.B.A. degree from Hebrew University, Jerusalem, Israel, and the Ph.D. degree from the University of Maryland, College Park.

He is an Assistant Professor in the Computer Science Department, Technion, Israel Institute of Technology, Haifa. His current research interests are in machine vision and robot navigation.