To Push or Not to Push - Part II
Practical Pushing Planning for Rearrangement Tasks

Ohad Ben Shahar
Ehud Rivlin
Center for Intelligent Systems
Department of Computer Science
Technion – Israel Institute of Technology
Haifa, Israel
[obs,ehudr]@cs.technion.ac.il

Abstract

Rearrangement of objects by pushing is a basic manipulation task. In [5] we presented a resolution-complete algorithm that plans optimal pushing manipulations for rearrangement tasks but operates in high time and space complexity. In this paper we address the issue of practical planning for the same kind of problems. Rather than using a classical heuristic method, we propose an alternative approach. We present a hierarchical classification of the pushing problems' domain into several classes, each characterized by properties of the plans that can solve it. Such a classification allows us to consider each class individually, analyze and exploit properties of each class, and suggest individual planning methods. Algorithms for two of the defined classes are presented. Both algorithms were tested in a simulated environment, with up to 32 movable objects and 66 combined DOF. Some of these simulations (with up to 8 moveables), as well as some experimental results using a real platform, and statistical data from more than 1000 randomly generated problems are presented here.
1 Introduction

Rearrangement of objects by pushing is a basic manipulation task. In [5] we presented an algorithm which, when given a set of goal configurations, plans a pushing plan to the "cheapest" one (or announces that no such plan exists). Although allowing optimal solutions and great flexibility, this algorithm is characterized by time and space complexities which do not allow its activation for most practical scenarios. This was not totally unexpected due to the high complexity of the underlying problem [15] and the unique properties of our domain: huge multidimensional state-space; unique transition operators, some of which are irreversible; nonintuitive solutions; and a need for planning with a deep lookahead in order to avoid irrecoverable states (which we refer to as trap points).

Since pushing problems are of high practical orientation, practical planning algorithms are required, even at the expense of completeness. Realizing that, one should seek for some heuristic approach to search the space and hope to find a solution within resource bounds\(^1\). However, the pushing domain imposes a common problem on all classical heuristic methods, since it does not allow an easy definition of good heuristic functions. Hence, beam-search might often prune solution-leading search-states due to poor evaluation. Similarly, best-first and \(A^*\), which keep the whole search tree, might reach the memory bound long before a solution is found.

Rather than using a classical heuristic method, in this paper we propose an alternative approach. We propose a classification of the pushing problems domain into several special classes, each characterized by properties of the plans that can solve it. Such a classification allows us to consider each class individually, to analyze and exploit properties of each class, and to suggest individual planning methods. Algorithms for two of the defined classes are presented. We also present some auxiliary tools which allow a simple graph representation of essential information and properties of pushing (and general manipulation) problems.

The rest of this paper is organized as follows: Section 2 discusses related work. Section 3 contains basic definitions and formulates our problem. Section 4 presents the classification of the pushing problems and section 5 defines elementary building functions and data structures needed for our methods. Sections 6 and 7 discuss and present the algorithms for two of the proposed classes, while sections 8 and 9 present some simulation results and real world experiments. In conclusion, section 10 discusses performance issues and section 11 summarizes our work and future research.

2 Related Work

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 [15]. Barraquand and Latombe [4, 3] 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,

\(^1\)We should note that non heuristic search might be considered an option too. Depth-first search, for example, seems to be particularly reasonable as it requires only linear space. However, as argued in [14], DFS might be impractical and even "dangerous" for such a huge search space as ours.
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 [7] and multi-arm manipulation planning [8, 9]. However, this method has a few limitations which seem to make it unsuitable for our problem. Hill climbing cannot be used due to the existence of trap-points. 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 most pushing problems. Similarly, problems which have all their solutions pass through narrow passages are major candidates for the failure of a randomized search. Furthermore, any pushing path has many segments which lie on the boundary of the configuration’s obstacles (see section 3). A direct usage of Barraquand and Latombe’s method for pushing tasks thus requires a unique potential field which both retracts the robot from obstacles and movable objects (to avoid collisions) and attracts it to them (to allow the pushing action).

Despite the great deal of motion planning research, not much work has been done directly on the area of pushing planning. Akella and Mason [1] analyzed the series of pushes needed to bring a convex polygon to a desired configuration. They used linear pushes, carried out by a fence, which may translate the object when stable edge contact is maintained, and rotate it while meeting an unstable edge or a corner. In their work, uniform friction distribution was assumed and no attention was paid to the pusher’s geometry or to any other geometrical constraints (e.g., obstacles). A somewhat comprehensive study was carried out by Lynch and Mason [13] 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. In their work they considered only limited DOF by allowing only one movable object and assuming that the pusher can change the contact configuration (chosen from a discrete set) at any time, with no restrictions. Chen and Hwang [6] presented a practical, heuristic and inexact solution for many movable obstacles. Their method is primarily a motion planning method (rather than pushing planning) by which movable obstacles can be pushed away by the robot whenever they stand in its way. Their method uses an approximate-cell-like decomposition to predict some optimal global path. This path is then executed using a generate-and-test paradigm, incorporated with local activities to move disturbing obstacles.

In our foregoing paper [5], 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. The pushing plan is found using a two phase procedure: context sensitive back propagation of a cost function, and a pushing path restoration phase. The latter is based on a gradient descent procedure which considers, in each step, only admissible neighboring configurations. The admissibility mechanism gives a primary tool for expressing the special characteristics of the pushing manipulation, using local considerations alone. It also allows a full integration of any geometrical constraints imposed by the pushing robot, the pushed objects and the environment. Although being (resolution-) complete, this algorithm has only limited practical use when many movable objects are involved due to its high complexity.
3 Definitions

Let $B = \{ \mathcal{R}, \mathcal{I}, \mathcal{M}_1, \ldots, \mathcal{M}_n \}$ 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 $\{\mathcal{M}_1, \ldots, \mathcal{M}_n\}$ is a collection of movable rigid objects, being able to move by external pushing force which might be applied by $\mathcal{R}$.

Each of the participating dynamic objects has its own configuration space. Let $CS_{\mathcal{R}}$ be the configuration space of $\mathcal{R}$ and $CS_{\mathcal{M}_i}$ the configuration space of $\mathcal{M}_i$. Since any pushing task must be carried out in a context of some support surface, we consider $CS_{\mathcal{M}_i}$ to be of 2 or 3 dimensions only. Let $CCS$ be the composite configuration space of $\mathcal{R}$ and all movable objects. Each vector in $CCS$ is a composite configuration $Q = (q_R, q_{M_1}, \ldots, q_{M_n})$ for $q_R \in CS_{\mathcal{R}}$ and $q_{M_i} \in CS_{\mathcal{M}_i}$. Let $CCO$ be the Cobstacle set in $CCS$, i.e., the set of all composite configurations in which at least two bodies of $B$ overlap (note that $CCO \neq \emptyset$ even if $\mathcal{I} = \emptyset$). Each composite configuration not in $CCO$ is some legal common configuration of $B$’s bodies and the set of all such configurations will be denoted by $CCF = CCS \setminus CCO$.

Along with the above configuration spaces we will also use the following projection operators:

$$
\Pi_{\mathcal{R}} : CCS \mapsto CS_{\mathcal{R}} \quad \Pi_{\mathcal{R}}(Q) = \Pi_{\mathcal{R}}(q_R, q_{M_1}, \ldots, q_{M_n}) = q_R
$$

$$
\Pi_{\mathcal{M}_i} : CCS \mapsto CS_{\mathcal{M}_i} \quad \Pi_{\mathcal{M}_i}(Q) = \Pi_{\mathcal{M}_i}(q_R, q_{M_1}, \ldots, q_{M_n}) = q_{M_i}
$$

Let $P(Q_1, Q_2)$ denote a configuration path (C-path) between $Q_1$ and $Q_2$. Similarly, let $P(Q_1, G)$ denote a C-path from $Q_1$ to one of the configurations in the set $G$. It is clear that not every C-path $P(Q_1, Q_2)$ is a pushing C-path from $Q_1$ to $Q_2$. Thus, some constrained definition is needed. As discussed in [5], each pushing path is a manipulation path [12, 11, 2] (i.e., an alternate sequence of transit paths and transfer paths) with each transfer path lying on the boundary of $CCF$ (with at least one projection, on some $CS_{\mathcal{M}_i}$, being non constant). In addition, it is further constrained by the directions it can move along the boundary of $CCF$. This is due to the fact that pushing is applicable by applying force only in specific directions.

Using all the above we define our basic pushing planning problem as follows:

**Given $B$ (a description of the environment), its corresponding $CCS$ space and $CCF$ set, with 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.**

When many movable objects are involved, the problem above incorporates a configuration space of many dimensions. As mentioned in the subsequent sections, many problems of this kind may be practically solved by breaking them into many low dimensional problems, each being solved by some underlying pushing planner. While the presented algorithms can use any such planner, we use the algorithm presented in [5].
4 Pushing Problems Classification and Examples

Let $\mathcal{PP}$ be the set of all pushing problems. As practical systems might assume some constraints on the feasible pushing problems they can handle, it may be beneficial to define accordingly some constrained subsets of $\mathcal{PP}$. In this section we divide $\mathcal{PP}$ into some classes which serve as a basis to a somewhat gradual study of the general pushing problem.

We define a pushing of a movable object $M_i$ to be non-preemptive if, while pushing $M_i$ from its initial configuration to its goal configuration, the robot pushes no other movable $M_j$. Using this definition, we define our classes of pushing problems as follows (see figure 1):

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{classification.png}
\caption{Classification of pushing problems.}
\end{figure}

**Definition 1:** A pushing plan is called **sequential** if it can be described as a sequence of robot operations in which at most one movable is pushed at a time. The class of all pushing problems which can be solved by a sequential plan will be denoted by $\mathcal{SP}$.

**Definition 2:** A pushing plan is called **linear** if it can be described as a sequence of non-preemptive pushings. The class of all pushing problems which can be solved by a linear plan will be denoted by $\mathcal{LP}$.

**Definition 3:** A pushing problem is called **flat** if it is linear and the pusher can choose any permutation of non-preemptive pushings in order to achieve the goal. The class of all pushing problems which can be solved by a flat plan will be denoted by $\mathcal{FP}$.

As the problems in $\mathcal{SP} \setminus \mathcal{LP}$ can still vary a lot (with respect to difficulty), we find a further classification of $\mathcal{SP}$ as beneficial. Hence, we define the set $\mathcal{LP}_e$ of those non-linear problems for which the task of finding a solution is rather "reasonable". A more precise definition, which motivates the proposed algorithm for non-linear problems, is presented in section 7. Figure 2 presents a set of test problems representing our classification. This set of problems will accompany us throughout this paper.
Figure 2: Examples of pushing problems (each problem is stated by its initial and goal configurations). P1 is a flat problem. P2 and P3 are both linear problems, although the latter seems more difficult. P4, which seems similar to P2, is no longer linear. P5, P6 and P7 are all non-linear too. Finally, P8 is an example of a non-sequential problem.
5 Basic Tools

When trying to deal with the various problem classes, we will use some tools and functions as described below:

Definition 4: Given $I$, $R$, and one movable $M$, the predicate $\text{IsPushable}(R, I, M, q_0, q_G)$ indicates whether $M$ is pushable\(^2\) from $q_0$ to $q_G$.

Definition 5: Given $I$, $R$, and one movable $M$, the function $\text{PushCPath}(R, I, M, Q_0, G)$ returns either (1) a pushing C-path that describes a plan for pushing $M$ by $R$ from the composite configuration $Q_0$ to one of the composite configurations in $G$ (each belongs to $CS_R \times CS_M$), or (2) NULL - if no such pushing C-path exists.

Definition 6: Given $I$, $R$, a set of movable objects $S = \{M_1, \ldots, M_n\}$, and their composite configuration $Q$, the function $\text{Reachable}(R, I, S, Q)$ returns a subset $S' \subseteq S$. Each $M_i \in S'$ is such that $R$ can reach it (i.e., obtain some contact mode) after leaving from $\Pi_R(Q)$ and without moving any movable.

Definition 7: Given $I$, $R$, $M_i$ and $M_j$, the function $\text{RCM}(R, I, M_i, q_i, M_j, q_j)$ returns all contact modes of $R$ and $M_i$ (when it is placed at $q_i$) which allow $R$ to reach $M_j$ (when it is placed at $q_j$).

The above four functions serve as building blocks for the algorithms presented, and we put no restrictions on the way that one implements them. One way to implement them is on the basis of the basic algorithm developed in [5]. Using this basic algorithm and assuming a predetermined discretization of the composite configuration space, most of these functions can operate in a constant time and space. The only exception is Reachable(), which needs linear time.

When implementing the defined routines with the basic algorithm of [5], $\text{PushCPath}()$ becomes a pure implementation for one movable object. $\text{IsPushable}()$ can be easily implemented by some variation of the cost-mapping phase of the same algorithm. All we should do is to set $\text{GOAL}$, the set of goal configurations given to the cost mapping phase, to be all configurations in which the movable lies at $q_G$ while the robot maintains some contact with it. The propagation can be terminated as soon as the wave-front accepts a configuration which its projection on $CS_M$ is $q_0$ (or fail if no such event occurs). $\text{Reachable}()$ can be implemented by considering all movable objects as obstacles while flooding the free space of the robot. Each configuration in that flooded area should then be checked whether it represents a contact mode with some object. $\text{RCM}()$ can work very similarly to $\text{Reachable}()$. It propagates a cost wave originating from all $R$’s configurations which touches $M_1$, and records each wave-front’s configuration which serves as a contact mode of $R$ and $M_0$ (when placed at $q_{M_0}$).

Definition 8: The function $\text{FREEZE}(I, M, q_M)$ freezes the movable object $M$ at the configuration $q_M$, as if it was an obstacle. It returns the new set $I' = I \cup M(q_M)$, with $M(q_M)$ being $M$ configured at $q_M$.

\(^2\)Note that such a predicate is not concerned with the initial or final configuration of the robot.

\(^3\)RCM stands for Reachable-Contact-Modes.
Some functions, like PushCPath(), are designed to handle one movable only. Using FREEZE() we allow those functions to consider geometrical restrictions also imposed by the other movables.

**Definition 9**: Given a finite set \( S = \{m_1, \ldots, m_n\} \), 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 \( v_i \xrightarrow{m_k} v_j \) if \( v_i \cup \{m_k\} = v_j \).

![Figure 3: The \( N_P(\{a, b, c, d\}) \) permutation net.](image)

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 \( S \). Figure 3 illustrates the permutation net of a four elements set.

**Definition 10**: Given a finite set of events \( S = \{\text{event}_1, \ldots, \text{event}_n\} \), which should all occur in order to achieve some goal \( G \), a precedence graph of \( S \) under \( G \) is a directed graph \( G_P(S, G) = (V, E) \) which has a node for each possible event (i.e., \( V = S \)) and there is a directed edge from \( \text{event}_i \) to \( \text{event}_j \) if \( \text{event}_i \) must precede \( \text{event}_j \), regardless of any other event, or else the goal \( G \) will become impossible.

A directed edge from \( \text{event}_i \) to \( \text{event}_j \) does not guarantee a successful occurrence of \( \text{event}_j \) after \( \text{event}_i \) (such a success might depend on a third event or some global context), but it certainly indicates that \( \text{event}_j \) can not happen before \( \text{event}_i \), regardless of any other event or criterion. A precedence graph which expresses all precedences between all pairs of events will be called maximal and will be denoted by \( G_P^M(S, G) \). It is clear that for each set of events and a corresponding goal there is exactly one such graph.
6 Dealing with $LP$ Problems

Given a flat pushing problem as $P1$, one can easily find an appropriate pushing plan that solves it. Given the set of $n$ movables, choose an arbitrary permutation of its elements and then create an independent plan for each (using the PushCPath() function) while treating all others as stationary obstacles (using the FREEZE() function). Since each sub-plan is constant in time and space, we get an $O(n)$ time complexity and $O(1)$ space complexity. Nevertheless, the $LP$ class, which does not seem to be much more difficult, requires much more time resources. This phenomena was already mentioned in the AI planning literature [10] and indeed, a naive algorithm similar to the above will have to scan all $n!$ possible permutations until encountering the one that solves the problem. Hence an exponential time in the average case. This section is aimed at proposing a planning algorithm that is far more efficient for most practical scenarios.

Having such a linear problem in hand, let us examine $N_P(S)$ - the permutation net of $S$. Since we are looking for a linear plan we can view $N_P(S)$ as a search graph in which each node represents the set of objects already pushed to their goal configuration, and each edge $\nu_i \rightarrow \nu_j$ represents the action of non-preemptive pushing of $M_i$. 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 $CCF$, it is still exponentially large in $n$, which makes all forms of exhaustive search unrealistic. However, in many cases we can dramatically narrow the search by using the information of the problem’s precedence graph, which can be built within a comparably modest time.

Let us define the auxiliary set of events $E = \{E_{M_1}, \ldots, E_{M_n}\}$, with event $E_{M_i}$ being defined as the non-preemptive pushing of $M_i$, and let $G^M(E, Q_G)$ be the maximal precedence graph of $E$ under the goal $Q_G$ (i.e., the goal of placing all movables in their goal configurations). Each edge $E_{M_i} \rightarrow E_{M_j}$ in $G^M(E, Q_G)$ represents the fact that any attempt to push $M_j$ to its goal before $M_i$ is pushed will not allow the realization of the goal $Q_G$. One way of constructing $G^M(E, Q_G)$ is by using the following simple algorithm:

$$\text{MPG}(I, R, E, S, Q_0, Q_G)$$

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

Each iteration of the above routine treats only two movables, one is considered an obstacle while a pushing plan is searched for the other. Two situations are checked for each such combination - first in which $M_i$ is frozen in its initial configuration, and second in which $M_i$ is frozen in its goal configuration. For each of the two situations a pushing path is searched for the "free" movable $M_j$ and an appropriate edge is constructed upon a failure. One should
note that because the two events $E_{M_i}$ and $E_{M_j}$ 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 ignore those configurations and test for an arbitrary pushing path using IsPushable() instead of using PushCPath(). Each iteration of the routine is of constant time complexity, hence the whole routine is $O(n^2)$, the same as the space complexity of $G^M_P(E, Q_G)$. Moreover, most of the computations can be done in parallel as each precedence edge is totally independent of the others. Figure 4 shows some maximal precedence graphs as produced by MPG().

![Maximal precedence graphs.](image)

Figure 4: Maximal precedence graphs. (A), (B), and (C) relate to problems P2, P3, and P7, respectively.

The following lemma shows that the precedence graph has a true potential in dramatically narrowing the search. It is then followed by two equivalent lemmas which allow us to efficiently use the precedence graph.

**Lemma 1**: 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.

**proof**: Let $E_{M_i} \rightarrow E_{M_j}$ be a precedence edge in $G_P(E, Q_G)$ and let $V_l$ and $E_l$ be the number of permutation net’s nodes and edges that become inadmissible as a result of it, respectively. Since such an edge states that $M_j$ cannot be pushed to its goal before $M_i$, we get that each node of the corresponding permutation net, which ”contains” $M_j$ but does not ”contain” $M_i$, cannot be part of a solution path. The number of such nodes in level $k$ of the permutation net is $(\frac{n-2}{k-1})$, leading to a total of

$$V_l = \sum_{k=1}^{n-1} \left(\frac{n-2}{k-1}\right) = 2^{n-2}.$$  

Each such node of level $k$ is connected by $k$ entering edges and $(n - k)$ leaving edges, all of which cannot be part of a solution path. However, we cannot calculate $E_l$ by simply multiply $V_l$ by $n$ since many such inadmissible edges connect two inadmissible nodes, and might be counted twice. Let us examine an inadmissible node $v$ at level $k$. Such a node is labeled by \{${M_{p_1}, M_{p_2}, \ldots, M_{p_{k-1}}, M_j}$\} with $p_m \neq i$ for all $1 \leq m < k$. The number of entering edges to $v$ is $k$, all of which come from nodes of the previous level. Each such edge is labeled with some movable included in $v$. An edge labeled with $M_{p_m}$ comes from a node labeled with \{${M_{p_1}, \ldots, M_{p_{m-1}}, M_{p_{m+1}}, \ldots, M_{p_{k-1}}, M_j}$\} and it is easy to see that such a node is inadmissible too (since it contains $M_j$ while not containing $M_i$). The edge labeled with
Each inadmissible node in level \( k \) has

1. \( k \) entering edges, from which \( k - 1 \) come from an inadmissible node of the previous level and one edge comes from an admissible node.
2. \( n - k \) leaving edges, from which \( (n - k - 1) \) enter an inadmissible node of the next level and one edge goes to some admissible node.

Utilizing this conclusion, we get that the number of inadmissible edges can be counted by considering each entering edge and only one leaving edge of every inadmissible node. All other leaving edges can be ignored since they will be counted as entering edges of inadmissible nodes in the next level (see figure 5). Formalizing that we get the following total number of inadmissible edges

\[
E_I = \sum_{k=1}^{n-1} (k + 1) \cdot \binom{n-k}{k} = 2 \cdot \sum_{k=0}^{n-2} \binom{n-2}{k} + \sum_{k=0}^{n-2} k \cdot \binom{n-2}{k} = 2^{n-2} \cdot \left(1 + \frac{n}{2}\right).
\]

The ratios of the above two results for \( V_I \) and \( E_I \) to the total number of permutation net’s nodes and edges, are

\[
\frac{V_I}{2^n} = \frac{1}{4}, \quad \frac{E_I}{n \cdot 2^{n-1}} = \frac{1}{4} \cdot \left(1 + \frac{2}{n}\right) > \frac{1}{4}
\]

thus, proving the lemma. \( \Box \)

Figure 5: The left graph shows the permutation net for a pushing problem with 4 movable. The figure shows all nodes (filled) and edges (thick) that can be marked as inadmissible due to the existence of a single precedence edge from \( E_a \) to \( E_k \). All the solid (thick) edges correspond to edges that enter some inadmissible node while the dashed edges enter an inadmissible node towards some admissible one. Each admissible node has exactly one such dashed edge. The right graph is the permutation net after the removal of all inadmissible elements.

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.
Lemma 2: Given a pushing problem $P_0$ and its maximal precedence graph $G^M_P(E, Q_G)$

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

**proof:** Assume that $E_{M_i} \in E$ has $d_{in}(E_{M_i}) > 0$. Then there must be some $M_j \neq M_i$ that precedes $M_i$ in any permutation that solves $P_0$. In practical terms, no permutation that solves $P_0$ can have $M_i$ as its first element. Hence, if each node of $G^M_P(E, Q_G)$ maintains that property then no movable can be the first in any pushing permutation that solves $P_0$, i.e., no linear solution exists. □

Lemma 3: Given a pushing problem $P_0$ and its maximal precedence graph $G^M_P(E, Q_G)$

$$G^M_P(E, Q_G) \text{ has a directed cycle } \Rightarrow P_0 \in \mathcal{PP} \setminus \mathcal{LP}.$$  

**proof:** Assume that $P_0 \in \mathcal{LP}$ and its maximal precedence graph $G^M_P(E, Q_G)$ contains the cycle $(E_{M_{i_0}} \rightarrow E_{M_{i_1}} \rightarrow \ldots \rightarrow E_{M_{i_m}} = E_{M_{i_0}})$. As a linear problem, $P_0$ can be solved by a linear plan, represented as a permutation of $S$. Yet, in any such a permutation each $M_{i_{(k \mod m)}}$ must precede $M_{i_{(k+1 \mod m)}}$. Since no permutation can satisfy such a constraint we get that either (1) $P_0 \notin \mathcal{LP}$ or (2) no such cycle exists. □

The latter two lemmas provide us with elementary tools to detect non-linear problems. However, it is important to note that the converse relations do not necessarily hold and non-linear problems can have a cycle-free maximal precedence graph. Example for such a case is illustrated in figure 6. Figure 6 illustrates another characteristic of pushing problems: it’s linearity, and sometimes the ability to solve it at all, depend very much on the initial and goal configurations of the pusher. These configurations define additional reachability constraints which are not represented in the precedence graph. Such constraints are temporary in nature and are not mutually independent. Hence, representing them using precedence edges is faulty. However, since we are looking for linear plans, we can safely claim that the first movable to be pushed must be such that it is reachable for $R$ from its initial (goal) configuration. While these constraints can be easily observed, it is not possible to formally deduce similar constraints on the other moveables too. However, a careless planner might cause the robot to trap itself, with no way to reach the next "correct" movable to be pushed, as illustrated in figure 7. We handle this problem using a one level lookahead, as detailed below.

![Figure 6](image-url)
Figure 7: Reachability problems in pushing planning. Let us assume all movable in the figure are allowed to be pushed after the robot obtains contact with the middle point of any of their edges. (1) shows, from left to right, an example where \textit{R} chooses a wrong final configuration after pushing \textit{X}, yet it can still find a way to the next movable (\textit{W}) without violating linearity (by further pushing \textit{X} outside the room, changing the contact mode and pushing it back to the passage). (2) shows an example where such a wrong decision can lead to a failure. Note that if the final configuration of the robot after pushing \textit{X} was chosen to be as described by the dashed line, then the task was solved. However, the wrong decision prevents the planner from reaching movable \textit{Y}, nor it allows the planning of a correction step for reaching the dashed contact mode. Hence, the only good permutation (\textit{W} \rightarrow \textit{X} \rightarrow \textit{Y}) could not be fully examined and a failure should be announced.

### 6.1 The LPLAN Algorithm

Following all of the above, we can now derive a precedences guided search over the permutation net of a given problem. The basic idea is that if we locate a movable object (say \( \mathcal{M}_i \)), where its corresponding pushing event \( E_{\mathcal{M}_i} \) has a zero in-degree, then there is no other event that should precede it. After pushing \( \mathcal{M}_i \) to its target configuration we can safely remove \( E_{\mathcal{M}_i} \) and its corresponding (in-going and out-going) edges from the precedence graph, and regard the remaining graph as the precedence graph of the remaining sub-problem. The guided search can then be based on an iterative inspection of the precedence graph of the problem, looking for a node with a zero in-degree, and removing it along with its corresponding edges before the next iteration. The problem of reachability is handled by a one step lookahead: given a movable to be pushed, we allow \textit{R} to finish the pushing C-path only in those configurations that allow it to reach the next movable to be pushed.

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

#### Algorithm LPLAN \( (I, R, S = \{ \mathcal{M}_1, \ldots, \mathcal{M}_n \}, Q_b, Q_G) \)

1. \( E \leftarrow \{ E_{\mathcal{M}_i} : \mathcal{M}_i \in S \} \)
2. \( G = (G_V, G_E) \leftarrow MPG(I, R, E, S, Q_b, Q_G) \).
3. \( REACH_{init} \leftarrow \text{Reachable}(R, I, S, Q_b) \)
4. \( REACH_{goal} \leftarrow \text{Reachable}(R, I, S, Q_G) \)
5. \( Perm \leftarrow \text{NULL} \)
6. Call Plan\( (I, R, S, Q_b, Q_G, G, 1) \)

end
First, the maximal precedence graph and the two reachability sets are constructed. The planning itself is carried out by the `Plan()` procedure, which handles the planning of the next non-preemptive C-path (which corresponds to following one edge of the permutation net):

**Procedure Plan** \((\mathcal{I}, \mathcal{R}, S, Q_{\text{current}}, Q_{\text{goal}}, G, \text{Level})\)

```plaintext
if (Level == n) then begin
  Let \(\mathcal{M}_C\) be the only element of \(S\)
  \(\text{Perm}[n] \leftarrow \mathcal{M}_C\)
  \(\text{FROM} \leftarrow (\Pi_R(Q_{\text{current}}), \Pi_{M_C}(Q_{\text{current}}))\)
  \(\text{GOAL} \leftarrow (\Pi_R(Q_{\text{goal}}), \Pi_{M_C}(Q_{\text{goal}}))\)
  \(\text{PATH} \leftarrow \text{PushCPath}(\mathcal{R}, \mathcal{I}, \mathcal{M}_C, \text{FROM}, \text{GOAL})\)
  if (PATH == Null) then return FAILURE
  else return SUCCESS
end if

CURRENT \leftarrow \text{Next-Candidates}(S, G, \text{Level})
if (CURRENT == \emptyset) then return FAILURE

for each \(\mathcal{M}_C \in \text{CURRENT} \) do
  \(\text{Perm}[\text{Level}] \leftarrow \mathcal{M}_C\)
  \(G' = (G'_V, G'_E) \leftarrow (G_V \setminus \{E_{M_C}\}, \{e \in G_E : e \text{ not connected to } E_{M_C}\})\)
  \(\text{NEXT} \leftarrow \text{Next-Candidates}(S \setminus \{\mathcal{M}_C\}, G', \text{Level} + 1)\)

for each \(\mathcal{M}_N \in \text{NEXT} \) do
  \(\mathcal{I}_{\text{imp}} \leftarrow \mathcal{I}\)
  for each \(\mathcal{M}_j \in \{\mathcal{M} \in S : \mathcal{M} \neq \mathcal{M}_C \land \mathcal{M} \neq \mathcal{M}_N\} \) do
    \(\mathcal{I}_{\text{imp}} \leftarrow \text{FREEZE}(\mathcal{I}_{\text{imp}}, \mathcal{M}_j, \Pi_{M_j}(Q_{\text{current}}))\)
    \(\text{GOAL} \leftarrow \text{RCM}(\mathcal{I}_{\text{imp}}, \mathcal{M}_C, \Pi_{M_C}(Q_{\text{goal}}), \mathcal{M}_N, \Pi_{M_N}(Q_{\text{current}}))\)
    \(\mathcal{I}_{\text{imp}} \leftarrow \text{FREEZE}(\mathcal{I}_{\text{imp}}, \mathcal{M}_N, \Pi_{M_N}(Q_{\text{current}}))\)
    \(\text{FROM} \leftarrow (\Pi_R(Q_{\text{current}}), \Pi_{M_C}(Q_{\text{current}}))\)
    \(\text{PATH} \leftarrow \text{PushCPath}(\mathcal{R}, \mathcal{I}_{\text{imp}}, \mathcal{M}_C, \text{FROM}, \text{GOAL})\)
    if (\(\text{PATH} == \text{Null}\)) then continue to next iteration
    \(Q_{\text{new}} \leftarrow Q_{\text{current}}\)
    \(\Pi_R(Q_{\text{new}}) \leftarrow \mathcal{R}'s \text{ goal configuration } "\text{chosen}" \text{ by PushCPath()}.\)
    \(\Pi_{M_C}(Q_{\text{new}}) \leftarrow \Pi_{M_C}(Q_{\text{goal}})\)
    \(\mathcal{I}_{\text{imp}} \leftarrow \text{FREEZE}(\mathcal{I}_{\text{imp}}, \mathcal{M}_C, \Pi_{M_C}(Q_{\text{goal}}))\)
    if (\(\text{Plan}(\mathcal{I}_{\text{imp}}, \mathcal{R}, S \setminus \{\mathcal{M}_C\}, Q_{\text{new}}, Q_G, G', \text{Level} + 1) = \text{SUCCESS}\))
      then return SUCCESS
  end for
end for
return FAILURE
end
```

Plan() receives the current configuration achieved so far \(Q_{\text{current}}\) and the precedence graph of the remaining sub-problem \((G)\). If an appropriate C-path is found, Plan() invokes itself.
recursively. Special treatment is given to the last movable (when Level = n). This treatment, besides serving as a termination criterion for the recursion, is necessary since at this level the NEXT set is guaranteed to be empty, the lookahead cannot function any more, and the inner loop cannot be executed. A notable auxiliary procedure used by Plan() is Next-Candidates(). It receives a precedence graph and calculates the set of all candidate moveables which can be considered next, while giving special treatment to the initial and final objects in the permutation, as implied by the reachability constraints. It is used by Plan() to find the current candidates, and in order to allow a lookahead of all the candidates expected in the next level.

\textbf{Procedure Next-Candidates} \((S, G = (G_V, G_E), Level)\)
\[\text{NEXT} \leftarrow \{M_i : E_{M_i} \in G_V \land d_{in}(E_{M_i}) = 0\}.\]
\[\text{if (Level == 1) then} \quad \text{NEXT} \leftarrow \text{NEXT} \cap \text{REACH}_{\text{init}}.\]
\[\text{if (Level == n) then} \quad \text{NEXT} \leftarrow \text{NEXT} \cap \text{REACH}_{\text{goal}}.\]
\[\text{return NEXT.}\]

The LPLAN search algorithm is guided by the precedence graph’s edges which impose only \textit{necessary} conditions that may not be sufficient, as argued before (p.12). However, the use of the information in the precedence graph is embedded within a depth-first like search over the permutation net. Consequently, a linear solution is guaranteed to be found, if one exists.

One should note that additional precedence constraints limit the search and allow LPLAN to exhibit better worst-case behavior. As a result, some easy looking linear problems, which their precedence graph contains no edges, might cause LPLAN to work harder. Such a degenerate graph is expected, for example, for every problem which incorporates several (i.e., more than one) 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 non-flat and LPLAN becomes no more than a DFS, which is expected to backtrack until encountering a solution. A typical example is presented in figure 8. Nevertheless, some planning time can still be spared by using a simple additional test in the inner loop of Plan(). Upon a failure of PushCPath(), we check whether or not the current candidate \(M_C\) can be pushed to its goal (with IsPushable()). If not, then it is guaranteed that such a failure will occur for every \(M_N\) of the lookahead. Hence, we can terminate the inner loop and start checking the next candidate.

Finally, we should mention that a necessary condition for LPLAN to succeed is the existence of a pushing C-path for each movable alone (i.e., while ignoring all others). Such test can be realized in linear time as part of the main routine of LPLAN. However, it is easy to see that when a movable is trapped, the precedence graph of the problem is guaranteed to contain many 2-hops cycles, all involving the same problematic object, as illustrated in figure 9. Following lemma 3, we can conclude that the problem is not linearly solvable.
7 Dealing with Reasonable SP Problems

Given a non-linear pushing problem, its solution can no longer be represented as a path in the permutation net. One approach for solving such problems might be to try and act in a linear, accumulative fashion. As before, we may push each movable to its goal in its turn. However, while doing that, the robot should be allowed to manipulate "disturbing" moveables. This approach, of subgoal definition, is widely used in AI planning problems [10]. However, it has a static nature which makes it less attractive for our domain.

An examination of "reasonable" non-linear problems such as P4, P5 and even P6, reveals a unique characteristic. Given such a problem, specified by $Q_0$ and $Q_G$, it may be transformed into a linear problem by introducing some small change (which we refer to as "noise") in the initial configuration of one or more moveable objects. Moreover, the realization of such a noise can usually be considered as a linear problem by itself, solvable by a linear plan. Hence, in

---

$^4$A classic example is the subgoals' definition for the $n$-puzzle problem, in which each tile should be brought to its goal in order. This subgoal ordering is predefined and ignores the specific instance of the problem at hand, something which is unacceptable in the pushing domain due to the irreversibility of the pushing manipulation.
Problem P6 (SP)

Figure 10: Reducing a non-linear pushing problem into two linear problems.

In many practical cases, the given non-linear problem \( Q_0 \rightarrow_{SP} Q_G \) can be reduced to a sequence of two linear problems \( Q_0 \leftrightarrow Q_{\text{noised}} \leftrightarrow Q_G \). An example of such a reduction for problem P6 is demonstrated in figure 10. Such a common sense approach, of rearrangement by decomposition, is often used by humans in a variety of rearrangement tasks, and might be feasible for many practical pushing problems. In order to formalize this approach, we define the following family of pushing problems:

**Definition 11**: A pushing plan is called \( \epsilon \)-linear if it is sequential, and can be described as a sequence of two linear plans \( Q_0 \leftrightarrow Q_{\text{noised}} \leftrightarrow Q_G \), with \( Q_{\text{noised}} \in \epsilon \)-neighborhood\((Q_0)\). The class of all pushing problems which can be solved by a \( \epsilon \)-linear plan will be denoted by \( \mathcal{LP}_\epsilon \).

The \( \epsilon \)-neighborhood of a configuration is defined as its neighborhood of "radius" \( \epsilon \):

\[
\epsilon \text{-neighborhood}\((q_r^0, q_{m_1}^0, \ldots, q_{m_n}^0)\) \triangleq \{ (q_r, q_{m_1}, \ldots, q_{m_n}) : |q_{m_i} - q_{m_i}^0| \leq \epsilon \ \forall 1 \leq i \leq n \}.
\]

Each value of \( \epsilon \) defines a different class of pushing problems. It is clear that \( \mathcal{LP}_{\epsilon_0} \subseteq \mathcal{LP}_{\epsilon_1} \) if \( \epsilon_0 < \epsilon_1 \) and that \( \epsilon = 0 \) creates the class \( \mathcal{LP} \) of linear problems. We consider each \( \epsilon \)-linear pushing problem to be a sequential problem whose solution is rather "reasonable".

Realizing the above arguments, we try to deal with reasonable sequential problems as if they were \( \epsilon \)-linear. Hence, given such a problem we have to answer two main questions: what movable should be noised ? ; and what noise should be applied ? Of course, both answers should be such that they allow us to transform the problem into a sequence of two linear problems, if possible. The following two subsections propose one way to answer these questions and present an algorithm that can handle many practical scenarios.
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 $G_I^p(E, Q_G)$, represents all precedences following from the initial configuration. Similarly, the goal precedence graph, denoted by $G_G^p(E, Q_G)$, represents all precedences following from the goal configuration. The routines used to construct those graphs are both derived from the MPG() routine (section 6), each preserving only the relevant tests. Following is the code for building the initial precedence graph. The goal precedence graph is built similarly.

```
IPG(I, R, E, S, Q_G)
    for each element of \{(M_i, M_j) : M_i, M_j \in S \land i \neq j\} do begin
        q_0 \leftarrow \Pi_{M_i}(Q_0)
        q_G \leftarrow \Pi_{M_j}(Q_G)
        T' \leftarrow FREEZE(I, M_i, \Pi_{M_i}(Q_0))
        if (IsPushable(R, T', M_j, q_0, q_G) == FALSE) then
            construct the edge $E_{M_i} \rightarrow E_{M_j}$
    end
```

Using $G_I^p(E, Q_G)$, the initial precedence graph, we define the following measure:

$$\beta(M_i) = d_{out}(E_{M_i} \in G_I^p(E, Q_G)) .$$

Literally, $\beta(M_i)$ counts the number of movables that $M_i$ stands in their way to the goal. Figure 11 illustrates the initial, goal and maximal precedence graphs of problems P4, P5, and P6. The values of the corresponding $\beta(M_i)$ are presented in table 1.

<table>
<thead>
<tr>
<th>Problem</th>
<th>P4</th>
<th>P4</th>
<th>P5</th>
<th>P6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objetc</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>$\beta(\cdot)$</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 1: Examples of $\beta(M_i)$

Having both subgraphs and the values of $\beta(M_i)$, we can deal with the question of which movables should we noise. One possible criterion for noising $M_i$ might be $\beta(M_i) > 0$. However, this condition is somewhat too strong, since sometimes we can achieve an appropriate configuration by noising only a subset of those candidates (e.g., in problem P5 only one movable should be noised). Consequently, we create the noisy configuration in an iterative manner. During each iteration we choose the movable $M_i$ with the maximal $\beta(M_i)$ and noise its configuration. Then we update both precedence graphs and move to the next iteration. The loop is terminated as soon as a cycle-free maximal precedence graph is obtained or no appropriate noise is found. Following is a description of the routine that creates the noisy configuration. The $\cup$ operator is such that it "unites" two graphs. The Handled set keeps track of the movables which were already noised. This information is used by the Noise() routine, which introduces the actual noise to a given movable and updates both precedence graphs accordingly.
Figure 11: Initial, goal and maximal precedence graphs of the discussed problems.

**Procedure Noise-Configuration** \((I, R, S, Q_{init}, Q_{goal})\)

\[
E \leftarrow \{E_{M_i} : M_i \in S\}
\]

\[
Q_{current} \leftarrow Q_{init}
\]

\[
G^I = (G^I_V, G^I_E) = G^I_P(E, Q_G) \leftarrow \text{IPG}(I, R, E, S, Q_{init}, Q_G)
\]

\[
G^G = (G^G_V, G^G_E) = G^G_P(E, Q_G) \leftarrow \text{GPG}(I, R, E, S, Q_{init}, Q_G)
\]

\[
\text{Handled} \leftarrow \emptyset
\]

**loop**

if \((G^I_V, G^I_E) \cup (G^G_V, G^G_E)\) contains no cycles then

return \(Q_{current}\) and **SUCCESS**

\[
D \leftarrow \text{Max}(\{\beta(M_i) : M_i \in \text{Reachable}(R, I, S, Q_{current}) \setminus \text{Handled}\})
\]

if \((D == 0)\) then return **FAILURE**

\[
M_{cand} \leftarrow \text{Select}(Candidates)
\]

\[
Q_{current} \leftarrow \text{Noise}(M_{cand}, I, R, S, Q_{current}, Q_{goal}, G^I, G^G, \text{Handled})
\]

if \((Q_{current} == NULL)\) then return **FAILURE**

\[
\text{Handled} \leftarrow \text{Handled} \cup \{M_{cand}\}
\]

**end loop**

Figure 12 demonstrates the action of Noise-Configuration() on problem P6. For P6, success is achieved only after the initial precedence graph becomes edge-less. However, this is not a must, and the termination can be obtained earlier. After creating the noisy configuration,
we are supposed to be left with a cycle-free maximal precedence graph. Such a graph can now be given to LPLAN in order to check whether a linear plan can solve it.

Figure 12: Running Noise-Configuration() on problem P6. The remaining initial precedence graph becomes edge-less and the maximal precedence graph becomes cycle-free after $n$ iterations (at this point the problem can be passed to LPLAN which may find a solution like $D \rightarrow A \rightarrow B \rightarrow C$).

### 7.2 Introducing an Appropriate "Noise"

The next question we should address is what noise should be introduced to a given candidate $M_{\text{cand}}$ after it was chosen by Noise-Configuration(). As we consider the problem to be $\epsilon$-linear, we assume that an appropriate noise, which fulfills the followings, does exist:

1. It should open the way for all other movables which must precede $M_{\text{cand}}$ when pushed to their goal, and whose path to the goal is currently blocked by $M_{\text{cand}}$.

2. It must not create new blocks for movables which are already able to reach their goal.

3. It must be realizable by pushing $M_{\text{cand}}$ from its current configuration.

4. It must not be a trap point (unless it coincides with the goal configuration of $M_{\text{cand}}$).

5. It should be a member of the $\epsilon$-neighborhood of the initial configuration.
Given $\epsilon$, the $\epsilon$-neighborhood of the initial configuration contains no more than $O(\epsilon^{M\text{'s DOF}})$ configurations of $M_{\text{ cand}}$. We currently choose to check all those configurations although it is 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.

Every configuration surviving the initial filtering should be checked for consistency with the other requirements. This can be done by updating both precedence graphs with accordance to the 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 ($M_{\text{ cand}}$) might have been changed.

Utilizing that we can sketch the Noise() routine as follows:

**Procedure Noise**($M_{\text{ cand}}, I, R, S, Q_{\text{ current}}, Q_{\text{ goal}}, G^I, G^G, Handled$)

1. $T' \leftarrow T$
2. For each element of $\{M_j : M_j \in S \land M_j \neq M_{\text{ cand}}\}$ do
   1. $T' \leftarrow \text{FREEZE}(T', M_j, \Pi_{M_j}(Q_{\text{ current}}))$
   2. $N \leftarrow \epsilon$-Neighborhood($M_{\text{ cand}}, \Pi_{M_{\text{ cand}}}(Q_{\text{ current}}), T'$)
3. For each $Q_{\text{ noise}} \in N$ do
   1. UIP($M_{\text{ cand}}, G^I, I, R, S, Q_{\text{ noise}}, Q_{\text{ goal}}$)
   2. UGP($M_{\text{ cand}}, G^G, I, R, S, Q_{\text{ noise}}, Q_{\text{ goal}}$)
   3. If (Is-Noise-Good($I, M_{\text{ cand}}, G^I, G^G, Handled$)) then
      1. return ($Q_{\text{ noise}}$)
4. Return (NULL)

UIP() and IGP() update the initial and goal precedence graphs, respectively. The $\epsilon$-Neighborhood() routine finds all $M_{\text{ cand}}$'s configurations which lie in the $\epsilon$-neighborhood of a given configuration, and returns a list of $CS_R \times CS_{M_{\text{ cand}}}$ configurations, sorted by their "pushing distance" from $Q_{\text{ current}}$. 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 $R$ and $M_{\text{ cand}}$ (when all other movables are frozen during this propagation). Since only $\epsilon$-neighborhood’s configurations are needed, the propagation can terminate at any configuration which lies farther than $\epsilon$.

An important part of Noise() is the Is-Noise-Good() predicate, which checks whether a specific configuration in the $\epsilon$-neighborhood can serve as an appropriate noise. A primary test carried out by Is-Noise-Good() is whether $\beta(M_{\text{ cand}})$ became zero. However, this is an insufficient condition. Additional tests must be considered in order to verify that the noisy configuration didn’t invalidate previous noises. This can be done by checking the updated maximal precedence graph (obtained by a union of $G^I$ and $G^G$) and looking for 2-hop cycles between $E_{M_{\text{ cand}}}$ and the events corresponding to each of the already noised movables (stored in the Handled set).
The sequential planner for reasonable problems, SPLAN, is a concatenation of Noise-Configuration() and LPLAN:

\[
\text{Algorithm SPLAN} \left( \mathcal{I}, \mathcal{R}, S = \{M_1, \ldots, M_n\}, Q_0, Q_G \right)
\]
\[
Q_{\text{Noise}} \leftarrow \text{Noise-Configuration} \left( \mathcal{I}, \mathcal{R}, S, Q_0, Q_G \right)
\]
\[
\text{LPLAN} \left( \mathcal{I}, \mathcal{R}, S, Q_{\text{noise}}, Q_G \right)
\]

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

Figure 13: A linear pushing plan for problem P3.

8 Simulated Examples

In order to allow for an examination of our algorithms, they were implemented and tested over a simulated environment. Illustrated below are the results of some test cases which
involve pushing tasks with a circular robot and up to eight movable objects, all having 2 DOF. Problems with up to 32 movables, that have up to 66 combined degrees of freedom, were successfully tested too.

Although the control environment allowed us to use each of the three discussed algorithms (FPLAN, LPLAN or SPLAN), all problems were submitted to SPLAN, which called LPLAN as soon as the task seemed to be linear. All runs used an $\epsilon$-neighborhood of radius $\epsilon = 3$. Each movable’s shape was defined by the user. Pushing contact modes were derived automatically using a COF (center of friction) which was specified by the user. The underlying implementation of the functions defined in section 5 was based upon the algorithm from [5].

Figures 13 to 19 show planning results as produced by the algorithm. All planning steps are ordered left to right and top to bottom. Transit paths are marked by arrows. Some planning steps, which correspond to non-preemptive pushings, were omitted (marked by two dots between adjacent steps). Figure 13 shows the plan found for problem P3 (from figure 2). Figures 14 and 15 show a linear problem with 8 movables and its solution. As these problems are linear, SPLAN introduced no noise before calling LPLAN. Figures 16, 17 and 19 show the solutions for successfully handled sequential problems. Included are problems P4, P6 and the 8-movable sequential problem presented in figure 18. The noisy configuration, from which each problem became linear, was marked by a brighter frame.

9 Real World Implementation

The experiments so far were executed in a simulated environment. However, as our goal is a real working system, we tested our algorithms in a realistic scenario using a mobile platform. The planner was run on a Sun 4/670 computer where it was integrated with the control environment of the NOMAD-200 mobile robot from Nomadic. The robot was directed remotely via a wireless Ethernet link. The planner used a coarse representation of the lab where the robot carried out a rearrangement plan for 5 chairs. Figure 20 shows some snapshots from one execution while figure 21 details the corresponding plan.

The experiments showed the applicability of the algorithm for real scenarios involving

![Figure 14: A linear problem with 8 movables.](image-url)
many movable objects. However, as discussed in [5], it also emphasized the great importance of sensory feedback. Lacking such a feedback, each such experiment must be preceded with an accurate calibration of the robot and the moveables, 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. Moreover,
other moveables (e.g., boxes) showed different behaviors when pushed and demonstrated the importance of the integration of a sensory information with a mechanical model of the manipulation. We see these directions as avenues for future research.

10 Complexity and Performance

The algorithms presented in this paper are motivated by the need for practical pushing planners. To achieve this goal the algorithms we presented break the multidimensional problem into a set of low dimensional subproblems, each of which is solved by the PushCPath() planning routine. Our implementation of PushCPath() is based on the basic algorithm [5] which has an exponential time and space complexity. However, here it is used to plan pushing C-paths for one movable at a time, (using FREEZE to consider all others as obstacles), giving a constant running time for each movable.

The LPLAN algorithm conducts a precedence-based DFS, with a one level lookahead, over the permutations net. When no precedence edges exist and still only few non-preemptive pushing permutations can solve the problem or part of it, LPLAN might carry out an exhaustive search over the permutation net. Nevertheless, in many practical cases LPLAN is expected to rarely backtrack, requiring only $O(n^2)$ time resources. In practical terms, it is
the construction of the maximal precedence graph that consumes the major part of the time resources needed by LPLAN, since each of the $O(n^2)$ tests of MPG() requires a rather large amount of constant processing time.

Considering SPLAN, its worst case complexity is dominated by that of LPLAN. The preceding phase of Noise-Configuration() includes at most $n$ iterations, each of which activates a cycle detector, a reachability scanner, and a “noise generator”. The first operates in $O(n^2)$, the second is linear, and the Noise() routine has an $O(e^{M^sDOF}n)$ time complexity. Hence, the complexity of Noise-Configuration() is $O(n^3)$, which outperform the exponential worst case complexity of LPLAN. However, as discussed above, LPLAN exhibits $O(n^2)$ performance in many cases. In those cases SPLAN’s complexity is dominated by that of Noise-Configuration(), i.e., $O(n^3)$. One should note that the Noise() routine, although linear with $n$, is a resource consuming routine. This is due to the fact that the linear time needed to update both precedence graphs (UIP() and UGP() routines) is factored by a relatively large constant. In addition, if $n$ is relatively small, $e^{M^sDOF}$ may exceed $n^2$, turning Noise() to the most demanding part of Noise-Configuration().

In order to verify that our planning algorithms exhibit a practical behavior, we tested them over a large set of randomly generated problems, using the simulated environment described in section 8. Our problem generator created maze-like problems with "rooms" and "corridors", and chose the initial and goal configurations randomly. The complexity of the environment (i.e., average corridors density) and the number of movables were defined by the user. In order to reduce the number of trivial (i.e., flat) problems, problematic areas (e.g., narrow passages) were allocated higher probability for being occupied with movables. Examples of such randomly generated problems is given in figure 22.

<table>
<thead>
<tr>
<th>Measure</th>
<th>Number OfMovables</th>
<th>Total Avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Linear problems</td>
<td>98%</td>
<td>85%</td>
</tr>
<tr>
<td>Sequential problems - treated linearly</td>
<td>1%</td>
<td>9%</td>
</tr>
<tr>
<td>Sequential problems - treated sequentially</td>
<td>1%</td>
<td>6%</td>
</tr>
<tr>
<td>Sequential planning success rate</td>
<td>100%</td>
<td>84%</td>
</tr>
<tr>
<td>Total success rate</td>
<td>99%</td>
<td>90%</td>
</tr>
</tbody>
</table>

Table 2: Execution characteristics for 1000 randomly generated problems (2 to 8 movables)

Using the above mechanism, more than 1000 problems, with up to 8 movables, were tested. All problems were generated with the same environmental complexity and the same discretization density, while the number of movables was gradually increased. Tables 2 summarizes some characteristics of the problems5, with each column corresponds to the same number of tests (about 150). The classification of problems into the appropriate class was based on their maximal precedence graph. In practical terms, every problem with a cycle free precedence graph, and which LPLAN had solved, was identified as linear, while all others were counted as non-linear.

Only 1% of all problems exhibited anomalous behavior due to LPLAN’s exhaustive search over the permutation net, and confirm our claim about LPLAN’s practical performance. These problems were not included in the analysis.
Table 2 shows that most of the problems we encountered were linear (83% in our 1000 problems experiment), however, as expected, their number tended to decrease with the growth in the number of movables. The tests reinforce our planning approach in two ways. First, they confirm our assumption that the precedence graph may serve as a decent measure for a problem's linearity. On the average, only in 5% of the tests this assumption was found to be erroneous and the amount of such errors was found to be insensitive to the number of participating movables. Second, they show that our approach, which tries to reduce sequential problems to a sequence of two linear problems, can be quite successful, solving here more than 80% of all the sequential problems.

Tables 3 and 4 summarize running times for both the entire set of problems and for the sequential problems alone. One should note that the time to construct the precedence graphs, which is class-independent, is similar in both tables. Nevertheless, the total running time for sequential problems is much higher due to the preceding phase of Noise-Configuration(). The average time spent by LPLAN in sequential problems is usually smaller since SPLAN might fail to find an appropriate noise and terminate before LPLAN is invoked. Figure 23 presents this information in two graphs which emphasize the fact that the non-polynomial component of our algorithm - the LPLAN planning procedure - is only a modest time consumer compared with the polynomial components. This fact reinforces our claim that LPLAN usually exhibits efficient and practical performance, with backtracking being rare.

<table>
<thead>
<tr>
<th>Measure</th>
<th>Number Of Movables</th>
<th>Total Avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Avg. time to handle a problem (seconds)</td>
<td>2.54</td>
<td>5.37</td>
</tr>
<tr>
<td>Avg. time to build precedence graphs</td>
<td>1.00</td>
<td>2.58</td>
</tr>
<tr>
<td>Avg. time to create noisy configuration</td>
<td>0.53</td>
<td>1.47</td>
</tr>
<tr>
<td>Avg. Time spent by LPLAN</td>
<td>1.00</td>
<td>1.31</td>
</tr>
</tbody>
</table>

Table 3: Performance results for all problems.

<table>
<thead>
<tr>
<th>Measure</th>
<th>Number Of Movables</th>
<th>Total Avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Avg. time to handle a problem (seconds)</td>
<td>5.14</td>
<td>9.97</td>
</tr>
<tr>
<td>Avg. time to build precedence graphs</td>
<td>0.90</td>
<td>2.49</td>
</tr>
<tr>
<td>Avg. time to create a noisy configuration</td>
<td>3.47</td>
<td>6.68</td>
</tr>
<tr>
<td>Avg. Time spent by LPLAN</td>
<td>0.77</td>
<td>0.80</td>
</tr>
</tbody>
</table>

Table 4: Performance results for sequential problems.
11 Conclusions

We addressed the problem of practical planning of a pushing manipulation for rearrangement tasks with many movable objects. We presented a hierarchical classification of the pushing problems domain into several special classes, each characterized by properties of the plans that can solve it. Based on this classification, two algorithms were presented for two of the defined classes. Our methods break the multidimensional problem into a set of low dimensional subproblems, each of which is solved by the PushCPath() planning routine. In addition, the algorithms are primarily based on the ability to efficiently extract precedence constraints directly from the given problem, and to use those constraints in order to narrow the search possibilities. We presented two tools - the permutation net and the precedence graph - which allow a clear and simple graph representation for some aspects of our problems. We tested our methods in a simulated environment with problems of up to 32 movable objects and 66 combined DOF, and carried out an extensive experiment, of more than 1000 randomly generated problems, to verify their practicality.

Although being able to deal with many movables, our methods can still be enhanced in several ways. As SPLAN’s performance is mostly affected by the Noise() routine, it may be wise to look for heuristics that focus first on the most promising members of the ε-neighborhood of the configuration to be noised. Moreover, SPLAN is not complete and might fail to solve ε-linear problems (e.g., P7 in figure 2, taken from the archive of the Sokoban computer game). We believe that some variation of the Noise-Configuration() routine, which chooses candidates by a more complex criterion, may serve that purpose. Similarly, although very complex sequential problems are less common in everyday life, it might be desirable to formulate a practical planner for them too. A further enhancement of our pushing problems classification, and the addition of classes as ε\(^k\)-linear, which permit the representation of a given problem as a chain of k linear problems, seems to be the first step towards that aim.

Finally, as observed from our real world experiments and discussed in [5], there are many issues that must be addressed before a robust rearrangement system can be implemented. These include the ability to handle arbitrary pushing manipulations (i.e., pushing that both rotate and translate the movable), a method to deal with incomplete knowledge and a scheme to exploit sensory information in order to achieve stable and predictable pushing manipulations.

Acknowledgment

We would like to thank Shaul Markovitch for his fruitful remarks with regard to the formalization of the \( \mathcal{LP}\epsilon \) class and the family of ε-linear pushing problems.
Figure 17: A sequential plan for problem P6. Note that unless movable C was noised to the left, no linear solution could have been found. The noisy configuration, from which the solution is linear, is marked by a brighter frame.

Figure 18: A sequential problem with 8 movables.
Figure 19: A pushing plan for the sequential problem with 8 movables. The noisy configuration, from which the solution is linear, is marked by a brighter frame.
Figure 20: Some snapshots from a real world experiment.

Figure 21: The pushing plan for the real-world experiment.
Figure 22: Examples of randomly generated pushing problems.

Figure 23: Planning time as a function of the number of movables.
References


