Block-Level Goal Recognition Design

Tsz-Chiu Au

Department of Computer Science and Engineering, Ulsan National Institute of Science and Technology
chiu@unist.ac.kr

Abstract

Existing works on goal recognition design (GRD) consider the underlying domain as a classical planning domain and apply modifications to the domain to minimize the worst case distinctiveness. In this paper, we propose replacing existing modifications with blocks, which group several closely related modifications together such that a block can modify a region in the search space with respect to some design constraints. Moreover, there could be blocks within blocks such that the design space becomes hierarchical for modifications at different levels of granularity. We present 1) a new version of pruned-reduce, a successful pruning rule for GRD, for block-level GRD, and 2) a new pruning rule for pruning some branches in both hierarchical and non-hierarchical design space. Our experiments show that searching in hierarchical design spaces greatly speeds up the redesign process.

Introduction

In goal recognition, an observer infers the goal of an agent acting in an environment from online observations of the agent’s behavior (Kautz 1987; Carberry 2001; Ramirez and Geffner 2010; Sukthankar et al. 2014; Vered and Kaminka 2017; Pereira, Oren, and Meneguzzi 2017). Goal recognition design (GRD) aims to redesign an environment such that an observer can recognize an agent’s goal as early as possible (Keren, Gal, and Karpas 2014, 2020). A popular performance measure in GRD is the worst case distinctiveness (WCD), which is the number of observations an observer needs to ascertain an agent’s goal in the worst case. Minimizing the WCD has practical usage in real-world applications. For example, in the airport security domain as described in (Keren, Gal, and Karpas 2014, 2020), we want to redesign the layout of an airport such that a security guard can be one step ahead of an intruder by recognizing an inappropriate goal pursued by the intruder as soon as possible.

Existing works on GRD consider environments as classical planning domains and redesign environments by modifying the actions in the planning domains using action removals (Keren, Gal, and Karpas 2014) or action conditioning (Keren, Gal, and Karpas 2018). In some environments, however, several modifications must be applied simultaneously to achieve the desired effect for goal recognition. For example, Figure 1 extends the example in (Keren, Gal, and Karpas 2014) by including all the walls such that an agent is not allowed to deviate from the given paths (the red lines). We can substitute one of the three yellow blocks for the red region in the grid to disallow an agent to choose some of the given paths. All walls in a yellow block must be added to the red region simultaneously; otherwise, an agent could deviate from the given paths via some missing walls in the red region. Action removals and action conditioning ignore this design constraint, causing the GRD algorithms to enumerate all combinations of the modifications for adding these walls.

In this paper, we introduce block-level GRD that uses blocks as the primary mean of environmental modifications. A block is a group of closely related modifications that can simultaneously modify multiple edges in the search space of classical planning problems. In Figure 1, there are three blocks, each of which modifies every edge in the red region in the grid. A block-level GRD algorithm chooses one of these blocks to minimize the WCD. If it chooses block $b_1$, the remaining paths an agent can choose are $p_2$, $p_3$, and $p_4$, and the WCD is 6, which is the length of the longest common prefix of $p_2$ and $p_3$. If the algorithm chooses $b_2$, the WCD is also 6. But if it chooses $b_3$, the remaining paths are $p_1$ and $p_4$, and the WCD is 3. Therefore, the algorithm should return $b_3$.

We formulate a hierarchical design model that allows blocks within blocks. The block at the top level of the hierarchy lays out the structure of the environment, whereas the blocks at the bottom level implement the design. The hierarchical design model resembles hierarchical task network (HTN) planning (Erol, Hendler, and Nau 1996; Nau...
et al. 2003, 2005) and can greatly reduce the search space of the GRD algorithms, making our approach suitable for large-scale GRD problems. In summary, our contributions are:

- We define a hierarchical design model for block-level goal recognition design.
- We modify pruned-reduce (Keren, Gal, and Karpas 2014), a highly successful pruning rule for GRD, for block-level GRD, and state the sufficient condition for returning an optimal design in a breadth-first search.
- We introduce a new pruning rule that avoids the expansion of some regions in a hierarchical design space. This pruning rule is applicable to non-hierarchical design models as well.
- We devise a local search algorithm to solve much larger GRD problems than the breadth-first search.

This paper is organized as follows. After presenting the related work, we define the problem of block-level goal recognition design. Next, we describe the block-based pruned-reduce and the design subtree pruning rule. Lastly, we present the experimental results and conclude this paper.

Related Work


Some works focus on improving the performance of GRD algorithms. Keren et al. introduced pruned-reduce in their seminal paper on WCD-based GRD (Keren, Gal, and Karpas 2014). Son et al. proposed solving GRD problems by Answer Set Programming solvers, which outperforms the existing approaches for fully observable models with optimal agents (Son et al. 2016). Ang et al. formulated the problem of computing optimal strategies as a mixed integer program in game-theoretic settings of GRD (Ang et al. 2017). Keren et al. framed existing pruning methods in the context of strong stubborn sets (Keren, Gal, and Karpas 2018). However, we still need better techniques to solve large GRD problems.

Our hierarchical design model resembles hierarchical task network (HTN) planning, which is more expressive than classical planning and can achieve an exponential speedup with sufficient domain knowledge to guide the planning process (Erol, Hendler, and Nau 1996). Our hierarchical design model also contains domain knowledge of which blocks fit a region. Like methods in SHOP2 (Nau et al. 2003, 2005), this knowledge can reduce the design space tremendously. However, our hierarchical design model provides little guidance on reducing the WCD.

Preliminary Definition: Classical Planning

This work is based on STRIPS planning for fully observable and deterministic environments (Fikes and Nilsson 1971). A state $s$ is a set of fluents, each of which is a ground, functionless atom that is true in $s$. A planning problem is a tuple $\langle F, s_0, A, G, \text{cost} \rangle$, where $F$ is a set of fluents, $s_0 \subseteq F$ is the initial state, $G$ is a set of goal states, $A$ is a set of actions and cost is the cost function of actions. Each action is a triple $a = (\text{pre}(a), \text{add}(a), \text{del}(a))$, where $\text{pre}(a) \subseteq F$, $\text{add}(a) \subseteq F$, $\text{del}(a) \subseteq F$ are the preconditions, the add list, and the delete list, respectively. $a$ is applicable to state $s$ if and only if $\text{pre}(a) \subseteq s$. The resultant state of applying $a$ to $s$ is $\text{apply}(s, a) = (s \setminus \text{del}(a)) \cup \text{add}(a)$. For simplicity, we assume the costs of all actions are the same (i.e., $\text{cost}(a) = 1$ for all $a \in A$). A plan $\pi$ is a sequence of actions $\langle a_1, a_2, \ldots, a_k \rangle$, where $a_i \in A$ for $1 \leq i \leq k$. $\pi$ is valid if and only if $a_{i+1}$ is applicable to $s_i$ where $s_{i+1} = \text{apply}(s_i, a_{i+1})$ for $0 \leq i < k$. The path of a valid $\pi$ is the sequence of states $p(\pi) = (s_0, s_1, \ldots, s_k)$ visited by an agent when executing $\pi$. We say a state $s$ is reachable by a valid plan $\pi$ if and only if $s \in p(\pi)$. Hence, a goal state $g \in G$ is reachable by $\pi$ if and only if $g \in p(\pi)$. Let $\text{prefix}(p_1, p_2)$ be the longest common prefix of $p_1$ and $p_2$.

Our GRD problem revolves around modifications to the search space of classical planning problems. Let $S$ be the set of all states reachable by some valid plans in a planning problem $\langle F, s_0, A, G, \text{cost} \rangle$. The search space of the planning problem is a directed graph $(V, E)$, where $V$ is a set of vertices and $E$ is a set of directed edges. Each state $s \in S$ has a vertex $v$ in $V$, and we denote $v$ by $v(s)$. Likewise, we denote $s[v]$ for a given $v$. There is an edge $(v_1, v_2)$ in $E$ whose cost is $\text{cost}(v_1, v_2)$ if there is an action $a \in A$ applicable to $s[v_1]$ and the resultant state is $s[v_2] = \text{apply}(s[v_1], a)$. We denote an edge by $a[v_1, v_2]$. Then we can formulate the planning problem as a graph search problem $(V, E, v_0, V_{\text{goal}}, \text{cost})$, where $(V, E)$ is the search space, $v_0 = v(s_0) \in V$ is the initial vertex, and $V_{\text{goal}}$ is the set of goal vertices representing the goal states in $G$. The objective is to find a path $p$ that connects $v_0$ to one of the goal vertices while minimizing the total cost of the path. Then, we can convert $p$ into a plan $\pi$ that solves the planning problem.

Block-Level Goal Recognition Design

This section defines the goal recognition design problem that uses blocks as environmental modifications.

Blocks and Regions

Previous works introduced two modifications for fully observable domains: action removals (Keren, Gal, and Karpas 2014) and action conditioning (Keren, Gal, and Karpas 2018). An action removal removes an action in $A$, whereas an action conditioning adds additional fluents to $\text{pre}(a)$. The
effect of these modifications to the search space is the removal of some edges such that some legal plans become infeasible. In this paper, we consider modifications that directly update a search space by a block, which represents a search space that can be put in a region of another search space. A block can alter multiple connectivity in a search space at once. Typically, only a few configurations in a region conform to the design constraints in an environment (see Figure 1). A block can produce these configurations directly, making the modification process more efficient.

We introduce a new type of vertices called regional vertices, which represent regions that can be substituted by blocks. Let us extend the definition of search spaces to include regional vertices. An extended search space is $G = (V, V', E, E')$, where $V$ is a set of ordinary vertices that represents a set $S$ of states (i.e., $s[v]$ exists for all $v \in V$), $V'$ is a set of regional vertices that do not represent any state (i.e., $s[v]$ does not exist for any regional vertex $v \in V'$), $E$ is a set of edges that represents actions applicable to the ordinary states in $S$ (i.e., $a[v_1, v_2]$ exists for any $(v_1, v_2) \in E$ if $v_1 \in V$), and $E'$ is a set of outgoing edges of the regional vertices in $V'$ that do not represent any action (i.e., $a[v_1, v_2]$ does not exist for any $(v_1, v_2) \in E'$ if $v_1 \in V'$ is a regional vertex). In short, an extended search space is a graph in which the set of vertices is $(V \cup V')$ and the set of edges is $(E \cup E')$. Moreover, $E$ is the set of all outgoing edges of the vertices in $V$, and $E'$ is the set of all outgoing edges of the regional vertices in $V'$. We shall assume the root vertex $v_{\text{root}}$ and the goal vertices in $V_{\text{goal}}$ are not regional vertices. All extended search spaces must satisfy the following condition in order to maintain the semantics of the planning problem: for all edge $(v_1, v_2)$ such that both $v_1$ and $v_2$ are ordinary vertices, $s[v_2] = \text{apply}(s[v_1], a[v_1, v_2])$.

**Substituting Blocks for Regional Vertices**

The specification of a block $b_1$ is $(V_1, V'_1, E_1, E'_1, V_{\text{entry}}^1, V_{\text{out}}^1)$, where $G_1 = (V_1, V'_1, E_1, E'_1)$ is an extended search space, $V_{\text{entry}}^1 \subseteq (E_1 \cup E'_1)$ is a set of entry vertices, and $V_{\text{out}}^1 \subseteq (E_1 \cup E'_1)$ is a set of exit vertices. Let us consider another extended search space $G_2 = (V_2, V'_2, E_2, E'_2)$ with a regional vertex $v_{\text{root}} \in V'_2$, and we are going to substitute $b_1$ for $v_{\text{root}}$. Let $E_{\text{in}} = \{ (v_1, v_2) : v_1 \in V'_2 \}$ be the set of incoming edges of $v_{\text{root}}$, and $E_{\text{out}} = \{ (v_1, v_2) : v_2 \in V'_2 \}$ be the set of outgoing edges of $v_{\text{root}}$ in $G_2$. Let $V_{\text{in}}^m = \{ v_1 \} \cup (V_{\text{in}} \cap V_{\text{out}}$ and $V_{\text{out}} = \{ v_2 \} \cup (V_{\text{in}} \cap V_{\text{out}}$, respectively. We say the vertices in $V_{\text{in}}$ and $V_{\text{out}}$ the incoming vertices of $r$ and the outgoing vertices of $r$, respectively.

After the substitution of $b_1$ for $v_{\text{root}}$, $G_1$ and $G_2$ are merged into a new search space $G_3$ in which $v_{\text{root}}$ is replaced with the $G_1$ with some new edges between $V_{\text{in}}$ and $V_{\text{entry}}$ as well as between $V_{\text{out}}$ and $V_{\text{out}}$. These new edges are specified by three given parameters: 1) $K_{\text{in}} \subseteq V_{\text{in}} \times V_{\text{entry}}$ is a set of edges that enter $b_1$; 2) $K_{\text{out}} \subseteq V_{\text{out}} \times V_{\text{out}}$ is a set of edges that leave $b_1$; and 3) $K_{\text{action}}$ is a mapping from $K_{\text{out}}$ to $\mathcal{A}$, where $K_{\text{out}} = \{ v_1 \} \times V_{\text{out}}$ is the set of edges in $K_{\text{out}}$ whose start vertices are ordinary vertices in $V_1$.

**Definition 1** We say $b_1$ is fit for $v_r$ in $G_2$ with $K_{\text{in}}, K_{\text{out}}$, and $K_{\text{action}}$ if and only if 1) for all $(v_1, v_2) \in K_{\text{in}}$ s.t. $v_1 \in V_2$ and $v_2 \in V_1$, apply(s[v_1], a[v_1, v_2]) = s[v_2]; and 2) for all $(v_1, v_2) \in K_{\text{out}}$ s.t. $v_1 \in V_1$ and $v_2 \in V_2$, apply(s[v_1], K_{\text{action}}((v_1, v_2))) = s[v_2].

These conditions are used to maintain the semantics of the planning problem: for any edge $(v_1, v_2)$ in $K_{\text{in}}$ or $K_{\text{out}}$, if both $v_1$ and $v_2$ are ordinary vertices, the resultant state after applying the action represented by the edge must be the same as $s[v_2]$. Note that there is no restriction on any edge $(v_1, v_2)$ in $K_{\text{in}}$ or $K_{\text{out}}$ if either $v_1$ or $v_2$ is a regional vertex.

Given a block $b_1$ that is fit for $v_r$ in $G_2$ with $K_{\text{in}}, K_{\text{out}}$, and $K_{\text{action}}$, we can substitute $b_1$ for $v_r$ to create a new extended search space $G_3 = (V_3, V'_3, E_3, E'_3)$, where $V_3 = V_1 \cup V_2$, $V'_3 = (V'_1 \cup V'_2) \setminus \{ v_r \}$, $E_3 = (E_1 \cup E_2) \setminus (E_6 \cup E_7)$, $E'_3 = (E'_1 \cup E'_2) \setminus (E'_1 \cup E'_5) \cup (E'_6 \cup E'_7)$, $E_4 = \{ (v_1, v_r) : v_r \in E_{\text{in}} \} \cup \{ v_r : v_r \in V_{\text{in}} \}$, $E'_4 = \{ (v_r, v_1) : v_1 \in E_{\text{in}} \}$, $E_5 = \{ (v_1, v_2) : v_1 \in V_{\text{out}} \}$, $E'_5 = \{ (v_1, v_2) : v_1 \in V_{\text{out}} \}$, $E_6 = \{ (v_1, v_2) : v_1 \in V_{\text{in}} \}$, $E'_6 = \{ (v_1, v_2) : v_1 \in V_{\text{out}} \}$, $E_7 = \{ (v_1, v_r) : v_r \in E_{\text{in}} \}$, $E'_7 = \{ (v_r, v_1) : v_r \in E_{\text{in}} \}$

This substitution removes the edges in $E_4$, $E'_4$, and $E'_5$ from $G_1$ and $G_2$, and then adds the edges in $E_6$, $E_7$, and $E'_6$ to $G_3$. The only vertex that is removed is $v_r$. We denote this substitution by $G_3 = \text{Subst}(G_2, v_r, b_1, K_{\text{in}}, K_{\text{out}}, K_{\text{action}})$.

**Hierarchical Design Models**

We define a hierarchical design model based on blocks and regions as follows. If the extended search space $G = (V, V', E, E')$ of a block $b$ has some regional vertices (i.e., $|V'| \geq 1$), every $v_r \in V'$ can be substituted by another block $b_j$ for $v_r$ with given $K_{\text{in}}^j$, $K_{\text{out}}^j$, and $K_{\text{action}}^j$. We say $b_j$ is a feasible block for $v_r$, and $b_j$ is a child block of $b$. In our design model, the set of all feasible blocks for $v_r$, together with the corresponding $K_{\text{in}}^j$, $K_{\text{out}}^j$, and $K_{\text{action}}^j$ that make the feasible blocks fit for $v_r$, is given. We denote the set of feasible blocks for $v_r$ by dom($v_r$), called the domain of $v_r$. We assume the domain of a regional vertex must have at least one feasible block. We also assume each block can be uniquely identified. Hence, each block $b_j$ belongs to the domain of only one regional vertex that is denoted by $v[b_j]$. Since $K_{\text{in}}^j$, $K_{\text{out}}^j$, and $K_{\text{action}}^j$ for every $b_j$ are given, we shall omit them when we substitute $b_j$ by dom($v[b_j]$) for $v[b_j]$ from now on. Thus, we write Subst($G, v[b_j], b_j, K_{\text{in}}^j, K_{\text{out}}^j, K_{\text{action}}^j$) as Subst($G$, $b_j$).

Figure 2 shows a hierarchical design model. At the top level, there is a root block $b_{\text{root}}$. For the completeness of our definition, we define a special vertex $v_{\text{dummy}} = v[b_{\text{root}}]$ and its domain contains $b_{\text{root}}$ only. In the specification of $b_{\text{root}}$,
we set \( V^{\text{entry}} = \{v_0\} \), \( V^\text{exit} = \{v|g \in G\} \), \( V_r^\text{in} = \{v_{\text{dummy}}\} \) and \( V_r^\text{out} = \{v_{\text{dummy}}\} \), where \( v_0 = v[s_0] \) is the initial vertex. Note that \( v_0 \) and the goal vertices in \( V_{\text{goal}} \) must be ordinary vertices in \( b_{\text{root}} \). Every block except \( b_{\text{root}} \) has a parent \( \text{parent}[b] \). Similarly, every block can have child blocks unless the block is a terminal block with no regional vertex. We shall assume there is no cycle of “blocks within blocks”;

thus, a hierarchical design model is finite. A design path from \( b_0 \) to \( b_m \) is a sequence \( \langle b_0, b_1, \ldots, b_m \rangle \) of blocks, where \( b_i = \text{parent}[b_{i+1}] \) for \( 1 \leq i < m \). Since every block \( b \) is uniquely identified, there is one and only one design path \( \langle b_{\text{root}}, b_1, \ldots, b_m, b \rangle \) from \( b_{\text{root}} \) to \( b \). Let ancestor\[b] = \langle b_{\text{root}}, b_1, \ldots, b_m, b \rangle \) be the ancestors of \( b \), where \( \text{parent}[b] = b_{m-1} \).

A design tree \( \Theta \) is a set of blocks such that \( \Theta \) contains all blocks on the design path from \( b_{\text{root}} \) to every block in \( \Theta \). Another way to define \( \Theta \) is that \( \Theta \) is a design tree if and only if \( \text{parent}[b] \in \Theta \) for every \( b \in (\Theta \setminus \{b_{\text{root}}\}) \). In Figure 2, a red arrow \( (v, b) \) denotes the choice of a feasible block \( b \) in \( \text{dom}(v) \) of a regional vertex \( v \). The set of all red arrows constitutes a design tree. A design tree’s root is always \( b_{\text{root}} \). By contrast, the root of a design subtree can be any block. A design subtree of \( b \in \Theta(b) \), which is a set of blocks that contains all blocks on the design path from \( b \) to every block in \( \Theta(b) \). If \( \Theta(b) \) is a design subtree of \( b \), parent\[b'[b] = \Theta(b) \) for every \( b' \in (\Theta(b) \setminus \{b\}) \). A design tree is a design subtree of \( b_{\text{root}} \).

We denote the set of child blocks of a block \( b' \in \Theta(b) \) by children\[b'] = \{b''|b'' \in \Theta(b) \text{ and parent}[b''] = b'\}. We say \( b' \) is full if |children\[b']| = |\text{VT}| \) where \( \text{VT} \) is the set of regional vertices in \( b' \). A terminal block \( b \) is always full since it has no regional vertex and children\[b] \) is empty. A design subtree \( \Theta(b) \) is full if every block in \( \Theta(b) \) is full; otherwise, \( \Theta(b) \) is partial. The design tree in Figure 2 is full since all outgoing arrows of all blocks in the design tree are blue arrows.

We say the blocks in \( \Theta(b) \), excluding \( b \), are subblocks of \( b \) in \( \Theta(b) \). A block \( b' \) is a possible subblock of \( b \) if and only if there exists a design subtree \( \Theta(b) \) such that \( b \) is a subblock of \( \Theta(b) \). Let \( B_{\text{sub}}(b) \) be the set of all possible subblocks of \( b \). We can construct \( B_{\text{sub}}(b) \) by a depth-first search in the hierarchical design model starting from \( b \): whenever the depth-first search visits a block \( b' \) for the first time, it adds \( b' \) to \( B_{\text{sub}}(b) \) if \( b' \neq b \), and then visits all child blocks of \( b' \). Let \( B_{\text{all}} = B_{\text{sub}}(b_{\text{root}}) \cup \{b_{\text{root}}\} \) be the set of all blocks.

A design tree \( \Theta \) specifies which feasible block in \( \text{dom}(v) \) should substitute for the corresponding regional vertex \( v \) in \( \Theta \). The result of the substitutions is a new extended search space \( G(\Theta) \). \( G(\Theta) \) can be constructed by a depth-first search in \( \Theta \) starting from the root of \( \Theta \). Initially, \( G \) is the extended search space of \( b_{\text{root}} \). Whenever the depth-first search visits a new block \( b \) in \( \Theta \), it replaces \( G \) with \( \text{Subst}(G, b) \). The replacement would succeed since all ancestors of \( b \) have been visited previously, and \( v[b] \) must be present in \( G \). Ultimately, the depth-first search returns \( G(\Theta) \). If \( \Theta \) is full, \( G(\Theta) \) has no regional vertex. If \( \Theta \) is partial, we say the regional vertices in \( G(\Theta) \) are open in \( \Theta \).

Legal Paths

An assumption in GRD research is that an agent cannot move freely or follow a given set \( \Pi_{\text{leg}} \) of legal plans whose paths can be either the shortest paths to some goals (Keren, Gal, and Karpas 2014), some feasible paths subject to physical constraints (Au 2022), or some paths in a path library (Mirskey et al. 2019). Note that legal plans do not have to be optimal or near-optimal. Let \( \Pi_{\text{leg}} \) be the set of legal paths traversed by an agent when executing the legal plans in \( \Pi_{\text{leg}} \) starting at the initial state \( s_0 \). The last state \( \text{last}[p] \) of a legal plan \( p \in \Pi_{\text{leg}} \) must be a goal state. By definition of goal states, no two goal states share the same state. Hence, each vertex on a legal path \( p \in \Pi_{\text{leg}} \) has at most one goal state. We assume an agent who chooses \( p \) aims for \( g[p] \) in the last state \( \text{last}[p] \) only and ignores the other goal states before \( \text{last}[p] \) on \( p \). We also assume that a legal path has no cycle (i.e., \( v_1 \neq v_2 \) for any \( v_1, v_2 \in p \).

Given a design tree \( \Theta \), a legal path \( p \in \Pi_{\text{leg}} \) is valid in \( \Theta \) if and only if \( p \) exists in \( G(\Theta) = (V, V', E, E') \), i.e., \( \langle v_1, v_2 \rangle \in E \) for all \( v_1, v_2 \in E \) where \( E \) is the set of edges in \( p \). Given a valid legal path \( p \) in \( \Theta \), we say the goal \( g(p) \) is reachable by \( p \) in \( \Theta \). A design tree \( \Theta \) is encompassing if every goal in \( \Pi_{\text{leg}} \) is reachable by some valid legal path \( p \in \Pi_{\text{leg}} \) in \( \Theta \), i.e., \( G = \{g[p]|p \in \Pi_{\text{leg}} \text{ and } p \text{ is valid in } \Theta\} \). The encompassment of design trees is important because we only consider design trees where all goals are reachable by some legal paths.

The Block-Level GRD Problem

According to (Keren, Gal, and Karpas 2020), a goal recognition design (GRD) model is a pair \( T = (R_0, D) \), where \( R_0 \) is an initial goal recognition model, and \( D \) is a design model, which is a tuple \( \langle M, \delta, \phi, \mathcal{U} \rangle \), where \( M \) is a set of atomic modifications, \( \delta \) is a modification transition function, \( \phi \) is a constraint indicator, and \( \mathcal{U} \) is an evaluation function for the goal recognition models. In this paper, a goal recognition model \( R \) is a design tree \( \Theta \), which can be either full or partial. The initial goal recognition model \( R_0 \) is \( R_0 = \{b_{\text{root}}\} \). In our context, \( D = \langle M, \delta, \phi, \mathcal{U} \rangle \), where \( 1) \mathcal{M} = \{B_{\text{all}} \setminus \{b_{\text{root}}\}\} \) is the set of all blocks except \( b_{\text{root}} \); 2) \( \delta \) is a transition function that maps a design tree \( \Theta \) to a new design tree after adding a new block to \( \Theta \) to replace an open regional vertex in \( \Theta \); 3) \( \phi \) checks whether a block adding to \( \Theta \) is in the domain of an open regional vertex in \( \Theta \); and 4)
\( \mathcal{U} \) is the worst case distinctiveness \( \text{WCD}(R) = \text{WCD}(\Theta) \), which represents the maximal cost of a non-distinctive path, which is a path an agent can follow without revealing its goal (Keren, Gal, and Karpas 2020). In this paper, we assume the costs of all actions are the same, and therefore the WCD is the maximal length of a non-distinctive path. More precisely,

\[
\text{WCD}(\Theta) = \text{WCD}(\Theta, P^{\text{valid}}) = \begin{cases} 
\max_{p_1, p_2 \in P^{\text{valid}} \text{ and } p_1 \neq p_2} |\text{prefix}(p_1, p_2)| & \text{if } \Theta \text{ is encompassing} \text{,} \\
\infty & \text{otherwise},
\end{cases}
\]

where \( P^{\text{valid}} \) is the set of all valid legal paths in \( \Theta \) and \( \text{prefix}(p_1, p_2) \) is the longest common prefix of the paths \( p_1 \) and \( p_2 \). The objective of a design algorithm is to find an optimal design tree \( \Theta^* \) such that \( \text{WCD}(\Theta) \) is minimized (i.e., for all \( \Theta \neq \Theta^* \), \( \text{WCD}(\Theta^*) \leq \text{WCD}(\Theta) \)).

**Block-Based Pruned-Reduce**

The seminal paper on GRD introduced two algorithms for finding an optimal design with the minimum WCD (Keren, Gal, and Karpas 2014). The first algorithm is a breadth-first search (BFS) called exhaustive-reduce, and the second algorithm extends exhaustive-reduce with pruned-reduce, a pruning rule that avoids expanding nodes that cannot reduce the WCD. This section presents a new version of pruned-reduce for block-level GRD problems.

First, we devise a BFS to explore the space of design trees. Initially, the queue in the BFS contains \( \Theta_0 = \{b_{\text{root}}\} \). In each iteration, the BFS extracts the first design tree \( \Theta \) from the queue. If \( \Theta \) is encompassing (i.e., all goals are reachable in \( \Theta \) and \( \text{WCD}(\Theta) < \text{WCD}(\Theta^*) \) where \( \Theta^* \) is the current best design tree, the BFS sets \( \Theta^* \) to \( \Theta \). Next, if \( \Theta \) is partial, the BFS creates a new design tree \( \Theta' \) for every feasible block for every open regional vertex in \( \Theta \), and adds \( \Theta' \) to the end of the queue. The BFS continues until the queue is empty and then returns \( \Theta^* \) whose \( \text{WCD}(\Theta^*) \) is minimum. The algorithm’s time complexity is \( O(m^4) \), where \( m \) is the maximum number of regional vertices in a block and \( h \) is the maximum height of design trees. The pseudocode of the BFS can be found in the technical appendix.

Pruned-reduce is an effective pruning rule for GRD algorithms. According to Theorem 5 in (Keren, Gal, and Karpas 2014), a successor node in a BFS with pruned-reduce is created only for actions that appear in the legal paths that yield the WCD. However, we cannot directly apply pruned-reduce to block-level GRD due to two issues. First, pruned-reduce is applicable only when modifications do not add new actions to the goal recognition model because the new edges in the search space could increase the WCD. However, when a block substitutes for a region, the block can contain new edges that are added to the search space. Second, even if a block \( b \) does not contain any action that can modify the legal paths that yield the WCD, some of its possible subblocks may contain such action. Therefore, it is necessary to look into the possible design subtrees of \( b \) to decide whether \( b \) can reduce the WCD.

Based on our definition of blocks and regional vertices, the first issue does not exist. Since regional vertices do not invalidate any legal paths, a regional vertex can be considered an empty search space that allows legal paths to pass through. When we substitute a block \( b \) for a regional vertex \( v \), the “new” edges in \( b \) still permit the legal paths that rely on the edges to pass through. By contrast, if an edge \( e \) in a legal path \( p \) is missing in \( b \) and \( e \) does not present in some possible subblocks of \( b \) even \( e \) is in the subpath of \( p \) that lies inside \( b \), the legal path will be invalidated. Therefore, the effect of the substitution is like removing these missing edges from the search space covered by the regional vertex. When the BFS extends the first design tree \( \Theta \) in the queue where \( \Theta \) is partial and has some open regional vertex \( v \), the substitution of a feasible block \( b \in \text{dom}(v) \) for \( v \) is like the removal of these missing edges in \( b \) simultaneously. If an edge \( e \) in a legal path \( p \) is missing in \( b \) and \( e \) presents in some possible subblocks of \( b \), the BFS will later expand the design subtrees that contain these possible subblocks of \( b \) unless these design subtrees are pruned.

**Possible Invalidation of Legal Paths**

To address the second issue, we consider the exact condition under which a block could remove some actions in legal paths. A block \( b \) with an extended search space \( G = (V, V', E, E') \) supports a path \( p \) if and only if for all \((v, v') \in \text{edges}(p) \) s.t. \( v \in V \), \((v, v') \in (E \cup K^{\text{out}}) \). That is, if the start vertex of an edge in \( p \) is in \( G \), the edge should also be in \( G \) or \( K^{\text{out}} \). Note that \( b \) still supports \( p \) even if no edge in \( \text{edges}(p) \) has a start vertex in \( V \). By contrast, \( b \) invalidates \( p \) if and only if \( b \) does not support \( p \). More precisely,

**Definition 2** \( b \) invalidates a legal path \( p \) if and only if there exists \((v, v') \in \text{edges}(p) \) s.t. \( v \in V \) but \((v, v') \notin (E \cup K^{\text{out}}) \).

If \( b \) invalidates \( p \) and \( b \) is added to a full design tree \( \Theta \), \( p \) cannot exist in \( G(\Theta) \) due to the missing edges in \( b \).

Even if a block \( b \) does not invalidate a path \( p \), some of the possible subblocks of \( b \) may still invalidate \( p \) if \( p \) has edges that enter the regional vertices of \( b \). Hence, the exact condition under which a block will certainly invalidate a legal path \( p \) regardless of the choice of the design subtree of \( b \) is:

**Definition 3** A block \( b \) necessarily invalidates \( p \) if and only if there exists at least one subblock \( b' \) in every possible design subtree \( \Theta(b) \) such that \( b' \) invalidates \( p \).

We can use necessary invalidation for pruned-reduce: when the BFS creates a new design tree \( \Theta' = \Theta \cup \{b\} \) for an open regional vertex \( v \) in \( \Theta \), the BFS chooses not to add \( \Theta' \) to the queue if \( b \) does not necessarily invalidate the legal paths that yield the WCD in \( \Theta \) since \( \Theta' \) cannot reduce the WCD. In fact, \( \text{WCD}(\Theta') = \text{WCD}(\Theta) \) according to Theorem 5 in (Keren, Gal, and Karpas 2014).

However, it is costly to check the condition of necessary invalidation since this involves an enumeration of all possible design subtrees of \( b \). Hence, we propose a weaker condition that can be computed quickly. A block \( b \) necessarily supports a path \( p \) if and only if all the blocks in all possible design subtrees of \( b \) support \( p \). By contrast, \( b \) possibly invalidates \( p \) if \( b \) does not necessarily support \( p \). More precisely,
Definition 4 A block \( b \) possibly invalidates a legal path \( p \) if and only if there exists a design subtree of \( b \) in which there exists a block that invalidates \( p \).

We can precompute the set \( B_{\text{invalid}}(p) \) of blocks that possibly invalidate a legal path \( p \in P_{\text{leg}} \). Initially, \( B_{\text{invalid}}(p) \) is empty for all \( p \in P_{\text{leg}} \). For all block \( b \in B_{\text{alt}} \), find \( P_{\text{leg}}(b) \subseteq P_{\text{leg}} \), which is the subset of legal paths invalidated by \( p \) according to Definition 2. For every \( p \in P_{\text{leg}}(b) \), we add \( b \) and all blocks in ancestor \( [b] \) to \( B_{\text{invalid}}(p) \). Finally, we obtain \( B_{\text{invalid}}(p) \) for every \( p \in P_{\text{leg}} \).

We use \( P_{\text{wcd}} = \bigcup_{p \in P_{\text{wcd}}} B_{\text{invalid}}(p) \) for pruned-reduce, where \( P_{\text{wcd}} \) is the current set of legal paths that yield the current minimum WCD. As shown in its pseudo-code, the BFS prunes the new design tree \( \Theta' = (\Theta \cup \{b_k\}) \) if \( b_k \) is not in \( B_{\text{invalid}} \). If \( \Theta' \) is not pruned, \( b_k \) possibly invalidates the legal paths that yield the WCD but does not necessarily invalidate these legal paths, and \( \Theta' \) may or may not reduce the WCD. However, it is safe to keep more design subtrees in the queue so that the BFS can later find out whether some extensions of these design subtrees can reduce the WCD.

### Pruning Design Subtrees

In this section, we devise a new pruning rule to avoid the expansion of the design subtrees for some regional vertices.

#### The Lower and Upper Bounds of Relative WCDs

Let \( P \) be a set of subpaths of legal paths that go through the extended search space of \( b \) s.t. all subpaths in \( P \) enter \( b \) from the same entry \( v_1 \) of \( b \) and exit \( b \) from some exits of \( b \). We combine the subpaths in \( P \) to form a legal path tree \( T \) by merging the longest common prefix \( \text{prefix}(p_1, p_2) \) of every pair of subpaths \( (p_1, p_2) \) in \( P \). The root of \( T \) is \( v_1 \), whereas the terminal vertices of \( T \) are some exits of \( b \). We say the last vertex of \( \text{prefix}(p_1, p_2) \) a junction in \( T \) for any \( p_1, p_2 \in P \).

We want to compute the lower and upper bounds of the relative WCD of every vertex \( v \) in \( T \) regardless of design trees. The relative WCD is defined as follows:

**Definition 5** The relative WCD of a vertex \( v \) in a legal path tree \( T \) for a given design tree \( \Theta \) is \( \text{RWCD}(\Theta, v) = \text{WCD}(\Theta, P') + 1 \), where \( P' \) is the set of subpaths in the subtree \( T' \) of \( v \) in \( T \) such that every \( p' \in P' \) is a path from \( v \) to a terminal vertex in \( T' \) and \( p' \) is valid in \( \Theta \).

If \( v \) is the initial vertex \( v_\text{root} \) and the legal path tree \( T \) is formed by \( P_{\text{leg}} \), we have \( \text{WCD}(\Theta, v_\text{root}) = \text{RWCD}(\Theta, v_\text{root}) - 1 \).

Suppose the relative WCDs of the terminal vertices of \( T \) for a design tree \( \Theta \) are given. We can recursively compute the relative WCDs of the internal vertices of \( T \) for \( \Theta \) as follows. Let \( P' \subseteq P \) be the set of subpaths that are valid in \( \Theta \). Let \( T' \) be a legal path tree formed by merging the subpaths in \( P' \). For every \( v \in (T \setminus T') \), \( \text{RWCD}(\Theta, v) \) is undefined since \( v \) is unreachable from \( v_1 \) in \( \Theta \). The relative WCD of a vertex \( v \) in \( T' \) is:

\[
\text{RWCD}(\Theta, v) = \begin{cases} 
1 + \max_{v' \in \text{children}(v)} \text{RWCD}(\Theta, v') & \text{if } |P'(v)| > 1 \\
0 & \text{otherwise},
\end{cases}
\]

where \( \text{children}(v) \) is the child vertices of \( v \) in \( T' \) and \( P'(v) \) is the set of paths in \( P' \) s.t. \( v \in P' \). Note that \( P'(v) \) is non-empty since \( v \) is in \( T' \). Since \( \text{RWCD}(\Theta, v') \) is given for every terminal vertex \( v' \in T' \), we can use Equation 3 to compute \( \text{RWCD}(\Theta, v) \) for every internal vertex \( v \) in a bottom-up fashion by a depth-first search. If \( |P'(v)| > 1 \), the paths in \( P'(v) \) cannot be distinguished at \( v \) yet, and hence the relative WCD of \( v \) is one plus the maximum of the relative WCDs of its children. If \( |P'(v)| = 1 \), there is only one path \( p \) in \( P' \) that goes through \( v \) and \( p \) can be identified at \( v \) and the relative WCD of \( v \) is 0.

If we do not know the relative WCDs of the terminal vertices of \( T \) given a design tree \( \Theta \), we cannot calculate the relative WCDs by Equation 3. However, if we are given the lower bound \( L_i \) and the upper bound \( U_i \) of the relative WCDs of every terminal vertex \( v_i \) in \( T \), we can use Equation 3 to compute the lower and upper bounds of the relative WCDs of the internal vertices in \( T \). Let \( B_{\text{sub}}(b) \) be the set of all possible subblocks of \( b \). \( B_{\text{sub}}(b) \) can be constructed by a depth-first search as described previously. Let \( P_{\text{min}} \subseteq P \) be the subset of subpaths in \( P \) that are not invalidated by any block in \( B_{\text{sub}}(b) \). Let \( T_{\text{min}} \) be the legal path tree formed by merging the subpaths in \( P_{\text{min}} \). Since \( P_{\text{min}} \) is the smallest set of subpaths in \( P \) that is not invalidated by any block in any design tree, the relative WCDs of every internal vertex \( v_j \in T_{\text{min}} \) computed by Equation 3 using \( L_i \) as the relative WCDs of every terminal vertex \( v_i \) is the lower bound \( L_i \) of the relative WCD of \( v_j \) in any design tree. For every vertex \( v_j \) in \( T \) but not in \( T_{\text{min}} \), we set the lower bound \( L_j \) of the relative WCD of \( v_j \) to 0. By contrast, since \( P \) is the largest set of subpaths before considering any invalidation, the relative WCDs of every internal vertex \( v_j \in T \) computed by Equation 3 using \( U_i \) as the relative WCDs of every terminal vertex \( v_i \) is the upper bound \( U_j \) of the relative WCD of \( v_j \) in any design tree. The technical appendix contains an example illustrating the calculation of the lower and upper bounds.

The calculation of \( L_i \) and \( U_i \) depends on the \( L_i \) and \( U_i \) of every terminal vertex \( v_i \) in \( T \). Both \( L_i \) and \( U_i \) can be calculated by Equation 3 using the lower and upper bounds of the relative WCDs of the vertices \( \text{children}(v_i) \) in the legal paths that go through \( v_i \). If we compute the bounds of the relative WCDs of the blocks in the hierarchical design model in a top-down fashion, \( \text{parent}(b) \) of a block \( b \) is evaluated before \( b \). Then we can obtain the lower and upper bounds of the relative WCDs of the vertices \( \text{children}(v_i) \) from \( T_{\text{out}} \) of the regional vertex \( v(b) \) in \( \text{parent}(b) \).

### Design Subtree Pruning Rule

We can use the lower and upper bounds of relative WCDs in \( b \) to avoid the expansion of the design subtrees in \( b \). Let \( L_i \) and \( U_i \) be the lower bound and the upper bound of a vertex \( v_i \) in \( T \) formed by a set \( P \) of subpaths of legal paths that go through the extended search space of \( b \). Let \( T_{\text{compact}} \) be the compact path tree of \( T \). \( T_{\text{compact}} \) is like \( T \) except that some ordinary vertices in \( T \) are replaced by regional vertices in \( b \) when these ordinary vertices lie inside these regional vertices. Due to space limits, we put the discussion on how to construct \( T_{\text{compact}} \) from \( T \) in the technical appendix. First, for every junction \( v \) in \( T \), if there are \( v_1, v_2 \in \text{children}(v) \)
such that $L_1 > U_2$, we mark all regional vertices in the subtree of $v_2$ in $T^{compact}$ as "pruned". For every regional vertex $v_r$ that is marked as "pruned", if all instances of $v_r$ in $T^{compact}$ are also marked as "pruned", we insert $v_r$ into a set $PrunedRegions(b)$. Then, the BFS prunes the design subtrees by not extending any open regional vertex in $PrunedRegions(b)$. We can avoid extending these regional vertices because the relative WCD of $v_2$ is always lower than the relative WCD of $v_1$ regardless of the design subtrees of the regional vertices in $PrunedRegions(b)$. Therefore, the vertex that yields the minimum WCD will not be found in the subtree of $v_2$ in $T$. Hence, there is no need to expand the design subtrees of the pruned regional vertices in the subtree of $v_2$ in $T$. If we can avoid expanding some large design subtrees, the GRD algorithms can speed up tremendously. The pseudocode of the BFS with the design subtree pruning rule can be found in the technical appendix.

### Empirical Evaluation

We conducted two experiments to evaluate the pruning rules and compare the BFS with a local search algorithm. The local search algorithm is based on the min-conflict heuristics (Minton et al. 1992; Sosić and Gu 1994), a highly effective heuristics for constrained optimization problems, and it can return a suboptimal design tree quickly. The local search algorithm’s pseudocode can be found in the technical appendix. Experiment 1 compared the local search algorithm with the original BFS in (Keren, Gal, and Karpas 2014) and our BFS. Experiment 2 evaluated the performance of the pruned-reduce and the design subtree pruning rule.

#### Experimental Setup

We adopted four domains in the International Planning Competition: LOGISTICS, GRID, DEPOTS, and DRIVERLOG. We selected these domains because we could easily add hierarchical design models to them. For each domain, we implemented a problem generator that can also generate a hierarchical design model for each problem instance. In each domain, we generated 30 problem instances of different sizes, and for each problem instance, we generated $n$ goals, where $2 \leq n \leq 10$ and $n$ increases with the problem size. Then, we used a planner called Fast Downward (Helmert 2006) to find a legal plan and the corresponding legal path for each goal. We randomly deleted some actions on the legal plans and ran Fast Downward again to find legal plans that were slightly different from the previous ones. Ultimately, the number of legal paths in $P^{leg}$ was around $5 \times n$. In Experiment 1, we used three GRD algorithms: 1) the original BFS with pruned-reduce as described in (Keren, Gal, and Karpas 2014), 2) our BFS that utilizes hierarchical design models and uses both pruned-reduce and design subtree pruning, and 3) the local search algorithm with design subtree pruning and a pruned-reduce-like heuristic. We ran the algorithms to solve the problem instances and reported the average execution times and the average minimum WCDs in Tables 1 and 2. In Experiment 2, we ran our BFS in three scenarios: 1) without pruning rules, 2) with pruned-reduce only, and 3) with both pruned-reduce and design subtree pruning. We ran the algorithm to solve the problem instances and reported the average execution times in Table 2. Both experiments were conducted on an Apple laptop with an M1 CPU and 16GB RAM.

#### Results

In Table 1, we can see that our BFS with pruned-reduce and design subtree pruning outperforms the original BFS with pruned-reduce by around an order of magnitude. Hence, our hierarchical design model can substantially reduce the size of the search space. The local search algorithm was much faster than the original BFS and our BFS, whereas its average minimum WCDs were not much larger. In Table 2, we can see that both pruning rules can speed up the BFS, but the performance improvement from design subtree pruning was much smaller than that from pruned-reduce. The effectiveness of design subtree pruning depends on how many design subtrees are pruned and how large the pruned design subtrees are. Perhaps there were not many junctions in the legal path trees that satisfy the condition of the design subtree pruning rule in our hierarchical design models. Therefore, the performance gain of the design subtree pruning rule is less than that of block-based pruned-reduce.

### Conclusions and Future Work

In this paper, we introduced the concept of blocks, which allows us to enforce some design constraints among modifications in deterministic GRD. Based on blocks and regions, we defined a hierarchical design model that can greatly reduce the search space of GRD problems. Our block-based pruned-reduce is highly effective, whereas the effectiveness of the design subtree pruning rule depends on the locations of junctions in the legal path trees and the size of the design subtrees. Despite its name, the design subtree pruning rule is also applicable to non-hierarchical design models, which can be viewed as hierarchical design models with one layer of blocks, each of which contains exactly one modification. For many years, pruned-reduce has been the only pruning rule we know for GRD. Our design subtree pruning rule is another pruning rule for hierarchical and non-hierarchical design models. Block-level GRD offers a new way to think about the structure of environments for GRD. In the future, we intend to combine our hierarchical design model with hierarchical plans in HTN planning and extend our model with sensing actions for partial observability.
Acknowledgments

This work has been taken place at UNIST and was supported by NRF (2022R1A2C101216812).

References


