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 a 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 prevent an agent from choosing 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.

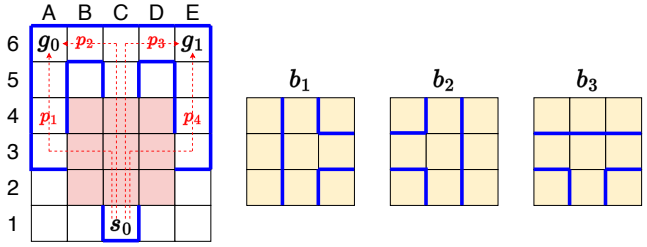

Figure 1: An example of block-level GRD. An agent can choose one of the four legal paths (the red lines) to reach either g_1 or g_2. The blue lines are the walls.

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 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 choose 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 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 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

GRD is a special case of environment design (Zhang, Chen, and Parkes 2009). Keren et al. proposed using the WCD as a performance measure in GRD (Keren, Gal, and Karpas 2014, 2019). Subsequently, many works extended the WCD-based GRD model to deal with non-optimal agents (Keren, Gal, and Karpas 2015), non-observable actions (Keren, Gal, and Karpas 2016a), privacy preserving in GRD (Keren, Gal, and Karpas 2016b), stochastic domains (Wayllace et al. 2016; Wayllace, Hou, and Yeoh 2017), game-theoretic GRD (Ang et al. 2017), GRD for plan libraries (Mirsky et al. 2019), partially-observable states (Wayllace et al. 2020), incomplete information (Keren 2019), information shaping (Keren et al. 2020), stochastic domains with sub-optimal agents (Wayllace and Yeoh 2022), interleaving between agents and observers (Gall, Ruml, and Keren 2021), and agents with multiple goals (Au 2022). (Keren, Gal, and Karpas 2020) is a survey of the works on GRD before 2020.

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 $\langle \mathcal{F}, s_0, \mathcal{A}, G, \mathsf{cost} \rangle$, where $\mathcal{F}$ is a set of fluents, $s_0 \subseteq \mathcal{F}$ is the initial state, G is a set of goal states, $\mathcal{A}$ is a set of actions and cost is the cost function of actions. Each action is a triple $a = \langle \mathsf{pre}(a), \mathsf{add}(a), \mathsf{del}(a) \rangle$, where $\mathsf{pre}(a) \subseteq \mathcal{F}$, $\mathsf{add}(a) \subseteq \mathcal{F}$, $\mathsf{del}(a) \subseteq \mathcal{F}$ are the precondition, the add list, and the delete list, respectively. a is *applicable* to state s if and only if $\mathsf{pre}(a) \subseteq s$. The *resultant* state of applying a to s is $\mathsf{apply}(s, a) = (s \setminus \mathsf{del}(a)) \cup \mathsf{add}(a)$. For simplicity, we assume the costs of all actions are the same (i.e., $\mathsf{cost}(a) = 1$ for all $a \in \mathcal{A}$). A plan π is a sequence of actions $\langle a_1, a_2, \ldots, a_k \rangle$, where $a_i \in \mathcal{A}$ for $1 \leq i \leq k$. π is *valid* if and only if a_{i+1} is applicable to s_i where $s_{i+1} = \mathsf{apply}(s_i, a_{i+1})$ for $0 \leq i < k$. The *path* of a valid π is the sequence of states $p(\pi) = \langle s_0, s_1, \ldots, s_k \rangle$ visited by an agent when executing π. We say a state s is *reachable* by a valid plan π if and only if $s \in p(\pi)$. Hence, a goal state $g \in G$ is reachable by π if and only if $g \in p(\pi)$. Let $\mathsf{prefix}(p_1, p_2)$ 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 $\langle \mathcal{F}, s_0, \mathcal{A}, G, \mathsf{cost} \rangle$. 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 by $s[v]$ for a given v. There is an edge (v_1, v_2) in E whose cost is $\mathsf{cost}[v_1, v_2]$ if there is an action $a \in \mathcal{A}$ applicable to $s[v_1]$ and the resultant state is $s[v_2] = \mathsf{apply}(s[v_1], a)$. We denote a by $a[v_1, v_2]$. Then we can formulate the planning problem as a graph search problem $(V, E, v_0, V_{\mathsf{goal}}, \mathsf{cost})$, where (V, E) is the search space, $v_0 = v[s_0] \in V$ is the *initial vertex*, and V_{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 π 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 $\mathcal{A}$, whereas an action conditioning adds additional fluents to $\mathsf{pre}(a)$. 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 $\mathcal{G} = (V, V^r, E, E^r)$, 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^r is a set of regional vertices that do *not* represent any state (i.e., $s[v]$ does not exist for any regional vertex $v \in V^r$), 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^r is a set of outgoing edges of the regional vertices in V^r that do *not* represent any action (i.e., $a[v_1, v_2]$ does not exist for any $(v_1, v_2) \in E^r$ if $v_1 \in V^r$ is a regional vertex). In short, an extended search space is a graph in which the set of vertices is $(V \cup V^r)$ and the set of edges is $(E \cup E^r)$. Moreover, E is the set of all outgoing edges of the vertices in V, and E^r is the set of all outgoing edges of the regional vertices in V^r. We shall assume the root vertex v_{root} and the goal vertices in V_{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, $s[v_2] = \mathsf{apply}(s[v_1], a[v_1, v_2])$.

Substituting Blocks for Regional Vertices

The *specification* of a block b_1 is $(V_1, V_1^r, E_1, E_1^r, V_1^{\mathsf{entry}}, V_1^{\mathsf{exit}})$, where $\mathcal{G}_1 = (V_1, V_1^r, E_1, E_1^r)$ is an extended search space, $V_1^{\mathsf{entry}} \subseteq (E_1 \cup E_1^r)$ is a set of *entry* vertices, and $V_1^{\mathsf{exit}} \subseteq (E_1 \cup E_1^r)$ is a set of *exit* vertices. Let us consider another extended search space $\mathcal{G}_2 = (V_2, V_2^r, E_2, E_2^r)$ with a regional vertex $v_r \in V_2^r$, and we are going to substitute b_1 for v_r. Let $E_r^{\mathsf{in}} = \{(v_1, v_r)\}_{v_1 \in (V_2 \cup V_2^r)}$ be the set of incoming edges of v_r and $E_r^{\mathsf{out}} = \{(v_r, v_2)\}_{v_2 \in (V_2 \cup V_2^r)}$ be the set of outgoing edges of v_r in $\mathcal{G}_2$. Let $V_r^{\mathsf{in}} = \{v_1\}_{(v_1, v_r) \in E_r^{\mathsf{in}}}$ and $V_r^{\mathsf{out}} = \{v_2\}_{(v_r, v_2) \in E_r^{\mathsf{out}}}$ be the sets of vertices on the edges, excluding v_r, in E_r^{in} and E_r^{out}, respectively. We say the vertices in V_r^{in} and V_r^{out} the *incoming vertices* of r and the *outcoming vertices* of r, respectively.

After the substitution of b_1 for v_r, $\mathcal{G}_1$ and $\mathcal{G}_2$ are merged into a new search space $\mathcal{G}_3$ in which v_r is replaced with the $\mathcal{G}_1$ with some new edges between V_r^{in} and V_1^{entry} as well as between V_1^{exit} and V_r^{out}. These new edges are specified by three given parameters: 1) $K^{\mathsf{in}} \subseteq V_r^{\mathsf{in}} \times V_1^{\mathsf{entry}}$ is a set of edges that enter b_1; 2) $K^{\mathsf{out}} \subseteq V_1^{\mathsf{exit}} \times V_r^{\mathsf{out}}$ is a set of edges that leave b_1; and 3) K^{action} is a mapping from $K^{\mathsf{out}}_{\mathsf{ordinary}}$ to $\mathcal{A}$, where $K^{\mathsf{out}}_{\mathsf{ordinary}} = K^{\mathsf{out}} \cap (V_1 \times V_r^{\mathsf{out}})$ is the set of edges in K^{out} whose start vertices are ordinary vertices in V_1.

Definition 1 *We say b_1 is* fit *for v_r in $\mathcal{G}_2$ with K^{in}, K^{out}, and K^{action} if and only if 1) for all $(v_1, v_2) \in K^{\mathsf{in}}$ s.t. $v_1 \in V_2$ and $v_2 \in V_1$, $\mathsf{apply}(s[v_1], a[v_1, v_r]) = s[v_2]$; and 2) for all $(v_1, v_2) \in K^{\mathsf{out}}$ s.t. $v_1 \in V_1$ and $v_2 \in V_2$, $\mathsf{apply}(s[v_1], K^{\mathsf{action}}((v_1, v_2)) = s[v_2]$.*

These conditions are used to maintain the semantics of the planning problem: for any edge (v_1, v_2) in K^{in} or K^{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^{in} or K^{out} if either v_1 or v_2 is a regional vertex.

Given a block b_1 that is fit for v_r in $\mathcal{G}_2$ with K^{in}, K^{out}, and K^{action}, we can substitute b_1 for v_r to create a new extended search space $\mathcal{G}_3 = (V_3, V_3^r, E_3, E_3^r)$, where

$$
\begin{aligned}
V_3 &= V_1 \cup V_2, \\
V_3^r &= (V_1^r \cup V_2^r) \setminus \{v_r\}, \\
E_3 &= [(E_1 \cup E_2) \setminus E_4] \cup (E_6 \cup E_7), \\
E_3^r &= [(E_1^r \cup E_2^r) \setminus (E_4^r \cup E_5^r)] \cup (E_6^r \cup E_7^r), \\
E_4 &= \{(v_1, v_r)\}_{(v_1, v_r) \in E_r^{\mathsf{in}} \text{ and } v_1 \in V_2} \\
E_4^r &= \{(v_1, v_r)\}_{(v_1, v_r) \in E_r^{\mathsf{in}} \text{ and } v_1 \in V_2^r} \\
E_5^r &= \{(v_r, v_1)\}_{(v_r, v_1) \in E_r^{\mathsf{out}}} \\
E_6 &= \{(v_1, v_2)\}_{(v_1, v_2) \in K^{\mathsf{in}} \text{ and } v_1 \in V_2} \\
E_6^r &= \{(v_1, v_2)\}_{(v_1, v_2) \in K^{\mathsf{in}} \text{ and } v_1 \in V_2^r} \\
E_7 &= \{(v_1, v_2)\}_{(v_1, v_2) \in K^{\mathsf{out}} \text{ and } v_1 \in V_1} \\
E_7^r &= \{(v_1, v_2)\}_{(v_1, v_2) \in K^{\mathsf{out}} \text{ and } v_1 \in V_1^r}
\end{aligned}
\tag{1}
$$

This substititon removes the edges in E_4, E_4^r, and E_5^r from $\mathcal{G}_1$ and $\mathcal{G}_2$, and then adds the edges in E_6, E_6^r, E_7, and E_7^r to $\mathcal{G}_3$. The only vertex that is removed is v_r. We denote this substitution by $\mathcal{G}_3 = \mathsf{Subst}(\mathcal{G}_2, v_r, b_1, K^{\mathsf{in}}, K^{\mathsf{out}}, K^{\mathsf{action}})$.

Hierarchical Design Models

We define a *hierarchical design model* based on blocks and regions as follows. If the extended search space $\mathcal{G} = (V, V^r, E, E^r)$ of a block b has some regional vertices (i.e., $|V^r| \geq 1$), every $v_i \in V^r$ can be substituted by another block b_j fit for v_i with given K_j^{in}, K_j^{out}, and K_j^{action}. We say b_j is a *feasible* block for v_i, and b_j is a *child* block of b. In our design model, the set of all feasible blocks for v_i, together with the corresponding K^{in}, K^{out}, and K^{action} that make the feasible blocks fit for v_i, is given. We denote the set of feasible blocks for v_i by $\mathsf{dom}(v_i)$, called the *domain* of v_i. 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_j^{in}, K_j^{out}, and K_j^{action} for every b_j are given, we shall omit them when we substitute $b_j \in \mathsf{dom}(v[b_j])$ for $v[b_j]$ from now on. Thus, we write $\mathsf{Subst}(\mathcal{G}, v[b_j], b_j, K_j^{\mathsf{in}}, K_j^{\mathsf{out}}, K_j^{\mathsf{action}})$ as $\mathsf{Subst}(\mathcal{G}, b_j)$.

Figure 2 shows a hierarchical design model. At the top level, there is a *root block* b_{root}. For the completeness of our definition, we define a special vertex $v_{\mathsf{dummy}} = v[b_{\mathsf{root}}]$ and its domain contains b_{root} only. In the specification of b_{root},

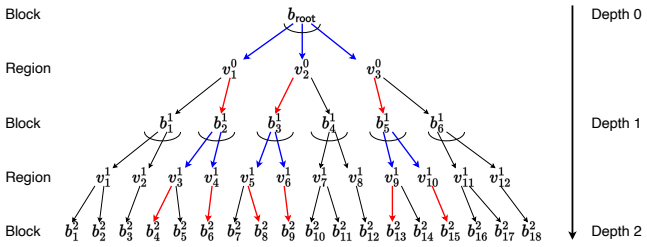

Figure 2: A hierarchical design model. The red arrows represent a design tree in the model. The blue arrows denote the inclusion of non-open regional vertices in the design tree.

we set $V^{\mathsf{entry}} = \{v_0\}$, $V^{\mathsf{exit}} = \{v[g]\}_{g \in G}$, $V_r^{\mathsf{in}} = \{v_{\mathsf{dummy}}\}$ and $V_r^{\mathsf{out}} = \{v_{\mathsf{dummy}}\}$, where $v_0 = v[s_0]$ is the initial vertex. Note that v_0 and the goal vertices in V_{goal} must be ordinary vertices in b_{root}. Every block b except b_{root} has a parent $\mathsf{parent}[b]$. 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 $b_i = \mathsf{parent}[b_{i+1}]$ for $1 \leq i < m$. Since every block b is uniquely identified, there is one and only one design path $\langle b_{\mathsf{root}}, b_1, \ldots, b_{m-1}, b \rangle$ from b_{root} to b. Let $\mathsf{ancestor}[b] = \langle b_{\mathsf{root}}, b_1, \ldots, b_{m-1} \rangle$ be the *ancestors* of b, where $\mathsf{parent}[b] = b_{m-1}$.

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

We denote the set of child blocks of a block b' in $\Theta(b)$ by $\mathsf{children}[b'] = \{b''\}_{b' \in \Theta(b) \text{ and } \mathsf{parent}[b''] = b'}$. We say b' is *full* if $|\mathsf{children}[b']| = |V^r|$ where V^r is the set of regional vertices in b'. A terminal block b is always full since it has no regional vertex and $\mathsf{children}[b]$ 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 $B^{\mathsf{sub}}(b)$ be the set of all possible subblocks of b. We can construct $B^{\mathsf{sub}}(b)$ by 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 $B^{\mathsf{sub}}(b)$ if $b' \neq b$, and then visits all child blocks of b'. Let $B_{\mathsf{all}} = B^{\mathsf{sub}}(b_{\mathsf{root}}) \cup \{b_{\mathsf{root}}\}$ be the set of all blocks.

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

Legal Paths

An assumption in GRD research is that an agent cannot move freely but follow a given set Π^{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 (Mirsky et al. 2019). Note that legal plans do not have to be optimal or near-optimal. Let P^{leg} be the set of *legal paths* traversed by an agent when executing the legal plans in Π^{leg} starting at the initial state s_0. The last state $\mathsf{last}[p]$ of a legal plan $p \in \Pi^{leg}$ must be a goal state, denoted by $g[p]$. By the definition of goal states, no two goal states share the same state. Hence, each vertex on a legal path $p \in P^{leg}$ has at most one goal state. We assume an agent who chooses p aims for $g[p]$ in the last state $\mathsf{last}[p]$ only and ignores the other goal states before $\mathsf{last}[p]$ on p. We also assume that a legal path p has no cycle (i.e., $v_1 \neq v_2$ for any $v_1, v_2 \in p$).

Given a design tree Θ, a legal path $p \in P^{leg}$ is *valid* in Θ if and only if p *exists* in $\mathcal{G}(\Theta) = (V, V^r, E, E^r)$, i.e., $(v_1, v_2) \in E$ for all $(v_1, v_2) \in \mathsf{edges}(p)$ where $\mathsf{edges}(p)$ is the set of edges in p. Given a valid legal path p in Θ, we say the goal $g(p)$ is *reachable* by p in Θ. A design tree Θ is *encompassing* if every goal in G is reachable by some valid legal path $p \in P^{leg}$ in Θ, i.e., $G = \{g[p]\}_{p \in P^{leg} \text{ and } p \text{ is valid in } \Theta}$. 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 $T = \langle R_0, D \rangle$, where R_0 is an initial goal recognition model, and D is a design model, which is a tuple $\langle \mathcal{M}, \delta, \phi, \mathcal{U} \rangle$, where $\mathcal{M}$ is a set of atomic modifications, δ is a modification transition function, ϕ 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 Θ, which can be either full or partial. The initial goal recognition model R_0 is $\Theta_0 = \{b_{\mathsf{root}}\}$. In our context, D is $\langle \mathcal{M}, \delta, \phi, \mathcal{U} \rangle$, where 1) $\mathcal{M} = (B_{\mathsf{all}} \setminus \{b_{\mathsf{root}}\})$ is the set of all blocks except b_{root}; 2) δ is a transition function that maps a design tree Θ to a new design tree after adding a new block to Θ to replace an open regional vertex in Θ; 3) ϕ checks whether a block adding to Θ is in the domain of an open regional vertex in Θ; and 4)

$\mathcal{U}$ is the *worst case distinctiveness* $\mathrm{WCD}(R) = \mathrm{WCD}(\Theta)$, 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,

$$\mathrm{WCD}(\Theta) = \mathrm{WCD}(\Theta, P^{leg}_{\mathsf{valid}}) = \tag{2}$$
$$\begin{cases} \max_{p_1, p_2 \in P^{leg}_{\mathsf{valid}} \text{ and } p_1 \neq p_2} |\mathsf{prefix}(p_1, p_2)| & \text{if } \Theta \text{ is encompassing} \\ \infty & \text{otherwise,} \end{cases}$$

where P^{leg}_{valid} is the set of all valid legal paths in Θ and $\mathsf{prefix}(p_1, p_2)$ 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 Θ^* such that $\mathrm{WCD}(\Theta)$ is minimized (i.e., for all $\Theta \neq \Theta^*$, $\mathrm{WCD}(\Theta^*) \leq \mathrm{WCD}(\Theta)$).

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 $\Theta_0 = \{b_{\mathsf{root}}\}$. In each iteration, the BFS extracts the first design tree Θ from the queue. If Θ is encompassing (i.e., all goals are reachable in Θ) and $\mathrm{WCD}(\Theta) < \mathrm{WCD}(\Theta^*)$ where Θ^* is the current best design tree, the BFS sets Θ^* to Θ. Next, if Θ is partial, the BFS creates a new design tree Θ' for every feasible block for every open regional vertex in Θ, and adds Θ' to the end of the queue. The BFS continues until the queue is empty and then returns Θ^* whose $\mathrm{WCD}(\Theta^*)$ is minimum. The algorithm's time complexity is $O(m^h)$, 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 Θ in the queue where Θ is partial and has some open regional vertex v, the substitution of a feasible block $b \in \mathsf{dom}(v)$ 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 $\mathcal{G} = (V, V^r, E, E^r)$ *supports* a path p if and only if for all $(v, v') \in \mathsf{edges}(p)$ s.t. $v \in V$, $(v, v') \in (E \cup K^{\mathsf{out}})$. That is, if the start vertex of an edge in p is in $\mathcal{G}$, the edge should also be in $\mathcal{G}$ or K^{out}. Note that b still supports p even if no edge in $\mathsf{edges}(p)$ 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 $(v, v') \in \mathsf{edges}(p)$ s.t. $v \in V$ but $(v, v') \notin (E \cup K^{\mathsf{out}})$.*

If b invalidates p and b is added to a full design tree Θ, p cannot exist in $\mathcal{G}(\Theta)$ 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 $\Theta' = \Theta \cup \{b\}$ where $b \in \mathsf{dom}(v)$ for an open regional vertex v in Θ, the BFS chooses *not* to add Θ' to the queue if b does *not* necessarily invalidate the legal paths that yield the WCD in Θ since Θ' cannot reduce the WCD. In fact, $\mathrm{WCD}(\Theta') = \mathrm{WCD}(\Theta)$ 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 $B^{\mathsf{invalid}}(p)$ of blocks that possibly invalidate a legal path $p \in P^{leg}$. Initially, $B^{\mathsf{invalid}}(p)$ is empty for all $p \in P^{leg}$. For all block $b \in B_{\mathsf{all}}$, find $P^{leg}_{\mathsf{invalid}}(b) \subseteq P^{leg}$, which is the subset of legal paths invalidated by p according to Definition 2. For every $p \in P^{leg}_{\mathsf{invalid}}(b)$, we add b and all blocks in $\mathsf{ancestor}[b]$ to $B^{\mathsf{invalid}}(p)$. Finally, we obtain $B^{\mathsf{invalid}}(p)$ for every $p \in P^{leg}$.

We use $B^{\mathsf{invalid}}_{\mathsf{wcd}} = \bigcup_{p \in P_{\mathsf{wcd}}} B^{\mathsf{invalid}}(p)$ for pruned-reduce, where P_{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 $B^{\mathsf{invalid}}_{\mathsf{wcd}}$. If Θ' is not pruned, b_k possibly invalidates the legal paths that yield the WCD but does not necessarily invalidate these legal paths, and Θ' 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 $\mathsf{prefix}(p_1, p_2)$ 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 $\mathsf{prefix}(p_1, p_2)$ 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 Θ is RWCD$(\Theta, v) =$ 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 Θ.*

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

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

$$\mathrm{RWCD}(\Theta, v) = \tag{3}$$
$$\begin{cases} 1 + \max_{v' \in \mathsf{children}[v]} \mathrm{RWCD}(\Theta, v') & \text{if } |P'(v)| > 1 \\ 0 & \text{otherwise,} \end{cases}$$

where $\mathsf{children}[v]$ is the child vertices of v in T' and $P'(v)$ is the set of paths in P' s.t. $v \in p \in P'$. Note that $P'(v)$ is non-empty since v is in T'. Since $\mathrm{RWCD}(\Theta, v'')$ is given for every terminal vertex v'' in T', we can use Equation 3 to compute $\mathrm{RWCD}(\Theta, v)$ for every internal vertex v in a bottom-up fashion by 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 Θ, 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 $B^{\mathsf{sub}}(b)$ be the set of all possible subblocks of b. $B^{\mathsf{sub}}(b)$ can be constructed by depth-first search as described previously. Let $P^{\mathsf{min}} \subseteq P$ be the subset of subpaths in P that are *not* invalidated by any block in $B^{\mathsf{sub}}(b)$. Let T^{min} be the legal path tree formed by merging the subpaths in P^{min}. Since P^{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 $v_j \in T^{\mathsf{min}}$ computed by Equation 3 using L_i as the relative WCDs of every terminal vertex v_i is the lower bound L_j of the relative WCD of v_j in *any* design tree. For every vertex v_j in T but not in T^{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_j and U_j 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 $\mathsf{children}(v_i)$ 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, $\mathsf{parent}[b]$ of a block b is evaluated before b. Then we can obtain the lower and upper bounds of the relative WCDs of the vertices in $\mathsf{children}(v_i)$ from V^{out} of the regional vertex $v(b)$ in $\mathsf{parent}[b]$.

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^{compact} be the *compact path tree* of T. T^{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^{compact} from T in the technical appendix. First, for every junction v in T, if there are $v_1, v_2 \in \mathsf{children}(v)$

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 $\mathsf{PrunedRegions}(b)$. Then, the BFS prunes the design subtrees by not extending any open regional vertex in $\mathsf{PrunedRegions}(b)$. 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 $\mathsf{PrunedRegions}(b)$. 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; Sosič 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. 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.

	Original BFS		Our BFS		Local Search	
	Time	WCD	Time	WCD	Time	WCD
LOGISTICS	23.67	8.0	2.48	8.0	0.43	8.5
DEPOTS	17.29	4.2	0.80	4.2	0.24	4.2
GRID	37.64	8.5	3.24	8.5	0.52	8.6
DRIVERLOG	16.80	7.7	0.77	7.7	0.15	7.9

Table 1: The average execution times of the BFS (in second) and the average minimum WCDs in Experiment 1.

	No Pruning	Pruned-reduce	P.R. + D.S.P.
LOGISTICS	20.80	2.61	2.48
DEPOTS	5.25	0.89	0.80
GRID	33.02	3.55	3.24
DRIVERLOG	5.17	0.82	0.77

Table 2: The average execution times of the algorithms (in second) in Experiment 2.

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 prune-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

Ang, S.; Chan, H.; Jiang, A. X.; and Yeoh, W. 2017. Game-Theoretic Goal Recognition Models with Applications to Security Domains. In *GameSec 2017*, 256–272.

Au, T.-C. 2022. Extended Goal Recognition Design with First-Order Computation Tree Logic. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 9661–9668.

Carberry, S. 2001. Techniques for Plan Recognition. *User Modeling and User-Adapted Interaction*, 11(1–2): 31–48.

Erol, K.; Hendler, J.; and Nau, D. S. 1996. Complexity results for hierarchical task-network planning. *Annals of Mathematics and Artificial Intelligence*, 18: 69–93.

Fikes, R. E.; and Nilsson, N. J. 1971. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 2: 189–208.

Gall, K. C.; Ruml, W.; and Keren, S. 2021. Active Goal Recognition Design. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*.

Helmert, M. 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research*, 26: 191–246.

Kautz, H. 1987. *A Formal Theory of Plan Recognition*. Ph.D. thesis, Department of Computer Science, University of Rochester.

Keren, S. 2019. Information Shaping for Enhanced Goal Recognition. In *AAAI Workshop on Plan, Activity, and Intent Recognition*.

Keren, S.; Gal, A.; and Karpas, E. 2014. Goal Recognition Design. In *International Conference on Automated Planning and Scheduling (ICAPS)*, 154–162.

Keren, S.; Gal, A.; and Karpas, E. 2015. Goal Recognition Design for Non-Optimal Agents. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 3298–3304.

Keren, S.; Gal, A.; and Karpas, E. 2016a. Goal Recognition Design with Non-Observable Actions. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 3152–3158.

Keren, S.; Gal, A.; and Karpas, E. 2016b. Privacy Preserving Plans in Partially Observable Environments. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 3170–3176.

Keren, S.; Gal, A.; and Karpas, E. 2018. Strong Stubborn Sets for Efficient Goal Recognition Design. In *International Conference on Automated Planning and Scheduling (ICAPS)*, 141–149.

Keren, S.; Gal, A.; and Karpas, E. 2019. Goal Recognition Design in Deterministic Environments. *Journal of Artificial Intelligence Research (JAIR)*, 65: 209–269.

Keren, S.; Gal, A.; and Karpas, E. 2020. Goal Recognition Design - Survey. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 4847–4853.

Keren, S.; Xu, H.; Kwapong, K.; Parkes, D.; and Grosz, B. 2020. Information Shaping for Enhanced Goal Recognition of Partially-Informed Agents. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 9908–9915.

Minton, S.; Johnston, M. D.; Philips, A. B.; and Laird, P. 1992. Minimizing Conflicts: A Heuristic Repair Method for Constraint-Satisfaction and Scheduling Problems. *Artificial Intelligence*, 58(1–3): 161–205.

Mirsky, R.; Gal, K.; Stern, R.; and Kalech, M. 2019. Goal and Plan Recognition Design for Plan Libraries. *ACM Transactions on Intelligent Systems and Technology*, 10(2).

Nau, D.; Au, T.-C.; Ilghami, O.; Kuter, U.; Muñoz-Avila, H.; Murdock, W.; Wu, D.; and Yaman, F. 2005. Applications of SHOP and SHOP2. *IEEE Intelligent Systems*, 20(2): 34–41.

Nau, D.; Au, T.-C.; Ilghami, O.; Kuter, U.; Murdock, W.; Wu, D.; and Yaman, F. 2003. SHOP2: An HTN Planning System. *JAIR*, 20: 379–404.

Pereira, R. F.; Oren, N.; and Meneguzzi, F. 2017. Landmark-Based Heuristics for Goal Recognition. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 3622–3628.

Ramirez, M.; and Geffner, H. 2010. Probabilistic Plan Recognition Using Off-the-Shelf Classical Planners. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 1121–1126.

Son, T. C.; Sabuncu, O.; Schulz-Hanke, C.; Schaub, T.; and Yeoh, W. 2016. Solving Goal Recognition Design Using ASP. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 3181–3187.

Sosič, R.; and Gu, J. 1994. Efficient Local Search with Conflict Minimization: A Case Study of the N-Queens Problem. *Knowledge and Data Engineering*, 6(5): 661–668.

Sukthankar, G.; Geib, C.; Bui, H. H.; Pynadath, D.; and Goldman, R. P. 2014. *Plan, Activity, and Intent Recognition: Theory and Practice*. Newnes.

Vered, M.; and Kaminka, G. A. 2017. Heuristic Online Goal Recognition in Continuous Domains. *arXiv:1709.09839*.

Wayllace, C.; Hou, P.; and Yeoh, W. 2017. New Metrics and Algorithms for Stochastic Goal Recognition Design Problems. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 4455–4462.

Wayllace, C.; Hou, P.; Yeoh, W.; and Son, T. C. 2016. Goal Recognition Design with Stochastic Agent Action Outcomes. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 3279–3285.

Wayllace, C.; Keren, S.; Gal, A.; Karpas, E.; Yeoh, W.; and Zilberstein, S. 2020. Accounting for Observer's Partial Observability in Stochastic Goal Recognition Design: Messing with the Marauder's Map. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, 2394–2401.

Wayllace, C.; and Yeoh, W. 2022. Stochastic Goal Recognition Design Problems with Suboptimal Agents. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 9953–9961.

Zhang, H.; Chen, Y.; and Parkes, D. 2009. A general approach to environment design with one agent. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2002–2008.