A Formal Analysis of Hierarchical Planning with Multi-Level Abstraction

Michael Staud[1,2]

[1]StaudSoft UG, D-88213 Ravensburg, Germany
[2]University of Ulm, Institute of Artificial Intelligence
D-89069 Ulm, Germany
michael.staud@staudsoft.com

Abstract

We analyze Hierarchical World State Planning (HWSP), a novel algorithm that tackles the scalability limitations of Hierarchical Task Network (HTN) planning. By combining multi-level state abstraction with predictive task decomposition, HWSP reduces exponential search space growth. We formalize predictive separable tasks, classify planning domains, and derive tight bounds on search complexity. Our results show that HWSP transforms exponential interactions into linear coordination, enabling polynomial-time top-level planning in favorable domains where sibling tasks are independent. This enables efficient planning in complex domains that were previously intractable, opening up new possibilities for real-world applications.

Introduction

Many real-world domains – from urban planning to large-scale logistics – require reasoning over vast, structured state spaces. Hierarchical Task Network (HTN) planning mitigates this growth by decomposing high-level tasks, yet even totally ordered HTN search can be overwhelmed: the number of sibling-task combinations often grows exponentially, straining both memory and computation. Existing scalability techniques, such as macro-operators, factored planning, and state abstraction, either sacrifice HTN semantics or still require exhaustive inter-sibling search. To address this, we build on the recently proposed Hierarchical World State Planning (HWSP) algorithm (Staud 2023), which introduces multi-level state abstraction and separable abstract tasks to break large problems into smaller, soundly connected subproblems.

We ground our work in the standard HTN planning formalism (Erol, Hendler, and Nau 1994), which models domains in terms of *primitive* tasks (directly executable actions) and *abstract* tasks (compound tasks) that are refined via *methods* into constrained task networks with ordering, binding, and dependency constraints. Our analysis focuses on **totally ordered HTN, acyclic (TO-HTN) planning**, where all task networks are strictly ordered. This restriction is motivated by two factors: (1) TO-HTN underlies many widely deployed planning systems, and (2) its decidability and complexity are well-characterized (Geier and Bercher 2011; Alford et al. 2012), making it a solid baseline for isolating the effects of multi-level state abstraction in HWSP.

In this paper, we provide a theoretical analysis of HWSP's planning efficiency across varying domain structures. We introduce a classification of domains – EX1, EX2, and EX3 – that captures how separable tasks interact and when backtracking is necessary. Our results establish tight upper bounds on the number of search nodes visited by HWSP under each domain class. We contrast these results with a progression-based HTN search algorithm, showing that under certain structural assumptions (e.g., sibling task independence), HWSP avoids the exponential exploration of sibling task combinations that characterizes interleaved HTN search. Furthermore, we incorporate a probabilistic analysis of binary decomposition predictors, showing how predictor quality impacts search effort in EX2 domains.

We organize the paper as follows: Section "Formal Framework" introduces the formal framework of Hierarchical World State Planning (HWSP), including its planning model, semantics, and interface definitions. We also present a city planning domain that is used to illustrate the key concepts. In Section "Domain Classes", we define three domain classes (EX1 – EX3) that constrain how separable abstract tasks behave, particularly with respect to decomposition and backtracking. Section "Backtracking" analyzes the backtracking behavior of HWSP under these domain classes. Section "Results" presents the theoretical results comparing HWSP and classical HTN planning.

Related Work

The basic idea of HWSP is to divide a planning problem into multiple smaller ones.

Early work by Korf (1987) demonstrated that decomposing a planning problem into subproblems can yield significant improvements in search efficiency. This idea was further developed by Knoblock (1990), who introduced algorithms for automatically creating abstraction hierarchies and proved that solving problems on multiple levels of abstraction can be exponentially more efficient than flat planning. The concept of factored planning, introduced by Brafman and Domshlak (Brafman and Domshlak 2006), also shows that problem factorization can lead to exponential reductions in effort.

Prior work has recognized macro-operators as an effective tool for improving planning efficiency. In classical STRIPS planning, a macro-operator is a precompiled or learned high-level action that encapsulates a sequence of primitive actions, thereby reducing search depth. In HWSP, separable abstract tasks play a related role: they encapsulate recurring hierarchical substructures, offering a similar reduction in search depth while also supporting richer ordering constraints. Korf's (1985) showed that, in certain domains, appropriately chosen macro-operators can substantially reduce effective search depth, turning exponential-time search into polynomial-time search for those cases. More recent work by Botea et al. (2005) demonstrated the effectiveness of automatically learned macro-operators in improving planning performance.

Hierarchical reinforcement learning (HRL) leverages both temporal and state abstraction to accelerate decision-making in long-horizon tasks. Like HWSP, HRL maintains representations of the environment at multiple abstraction levels; in HRL, these representations are coupled with a hierarchy of policies that operate over different temporal scales. Dietterich (2000)'s foundational MAXQ framework formalizes how a Markov Decision Process (MDP) can be decomposed into a hierarchy of subtasks, each with its own value function and (potentially abstracted) state representation. More recent work by Kulkarni et al. (2016) demonstrated that such multi-level abstractions can achieve strong performance in challenging, visually rich game environments.

Beyond classical HTN planning, several lines of work have addressed hierarchical abstraction, including independent subproblem identification (Lotem and Nau 2000), decomposition axioms (Biundo and Schattenberg 2001), and hierarchical goal networks (Shivashankar et al. 2012). HWSP builds on this tradition, but focuses on formalizing multi-level world state abstraction for scalability while retaining the soundness guarantees of HTN semantics. Sacerdoti's ABSTRIPS system (Sacerdoti 1974) introduced a form of state-space abstraction in which selected predicates are omitted at higher levels, yielding an abstract search space whose solutions are incrementally refined – though refinements can require backtracking. Angelic hierarchical planning (Marthi, Russell, and Wolfe 2008) instead abstracts actions, associating them with optimistic and pessimistic effect summaries to produce provable cost bounds during search. While this resembles separable abstract tasks in summarizing lower-level effects, focuses on bounded-cost optimality rather than multi-level state abstraction for scalability. More recent advances include compiling HTNs to SAT for optimal search (Behnke, Höller, and Biundo 2019), defining standard hierarchical languages such as HDDL (Höller et al. 2020a), and incorporating HTN planning into the IPC competition track to support empirical evaluation.

The interaction between task ordering, commitment strategies, and backtracking in HTN planning has been extensively investigated. Olz and Bercher (2023) proposed a look-ahead evaluation method for partially decomposed plans, enabling early pruning of unpromising decompositions and thereby reducing backtracking. Tsuneto et al. (1996) examined variable and method commitment policies in HTN planning, comparing early-commitment and delayed-commitment strategies and their impact on search efficiency.

Formal Framework

This Section describes the Hierarchical World State Planning (HWSP) algorithm, which extends the traditional planning paradigm by introducing a multi-level world state (Staud 2022, 2023). The HWSP algorithm is based on the concept of *separable abstract tasks*, which enable more efficient planning by partitioning the planning into smaller sub-planning processes across multiple levels of abstraction. The following version of the HWSP algorithm is a more formal one compared to the original paper. We assume a progressive planning strategy.

Let A denote the set of propositional atoms. A literal is an atom or its negation. Each atom $a \in A$ represents a basic proposition that can be TRUE or FALSE. For any set of literals E, we define two functions $E^+ = \{a \in A \mid a \in E\}$ and $E^- = \{a \in A \mid \neg a \in E\}$.

Planning Domain

A planning domain is denoted as $D = (A, T_a,\ T_p,\ M,\ L,\ L_d,\ D_{\text{der}},\ \Phi)$ where A is the set of propositional atoms, T_a is the set of all abstract tasks (including both standard and separable tasks), T_p is the set of primitive tasks, $T_S \subset T_a$ denotes the set of separable abstract tasks, M is the set of methods, and $L \in \mathbb{N}$ the number of detail levels (i.e., abstraction levels) present in the domain. The detail level function $L_d : A \cup T_a \cup T_p \rightarrow \{1, \ldots, L\}$ assigns each atom or task a detail level ℓ. The derived atoms (derived predicates (Edelkamp and Hoffmann 2004)) are defined in D_{der}, with their corresponding logical formulas specified in Φ.

Tasks are tuples in the form $t = \langle prec_t, \mathit{eff}_t \rangle$. For primitive tasks and separable abstract tasks, $prec_t$ and eff_t are sets of propositional literals with semantic meaning in the planning process. For non-separable abstract tasks, $prec_t = \mathit{eff}_t = \emptyset$; we maintain the tuple structure for formal consistency, but these tasks derive their semantics solely through decomposition methods.

For separable abstract tasks $t \in T_S$, we write $\mathit{eff}_t = \mathit{meff}_t \cup \mathit{peff}_t$ where meff_t are mandatory effects and peff_t defines the set of *possible* non-guaranteed effects. During planning, the prediction function $\varepsilon(s, t_S) \subseteq \mathit{peff}_t$ selects which subset of these possible effects will actually occur (see Definition 1). Let $\mathcal{L}$ be a set of unique identifiers (labels), which we assume to be countable and disjoint from all other syntactic entities in the domain. A plan step is a uniquely labeled task $l : t,\ l \in \mathcal{L}$. A *method* is a tuple $m = \langle t_a, P_m \rangle$, where t_a is the abstract task it can decompose, and $P_m = (l_1 : s_1, l_2 : s_2, \ldots, l_k : s_k)$ is a sequence of labeled plan steps (totally ordered).

A *state* s is an element of $\mathcal{P}(A)$, where $\mathcal{P}(A)$ denotes the power set of A. The *step function* $\mathsf{step} : \mathcal{P}(A) \times (T_p \cup T_S) \rightarrow \mathcal{P}(A)$ applies a task's effects to the world state: $\mathsf{step}(s, t) = (s \cup \mathit{eff}_t^+) \setminus \mathit{eff}_t^-$. We require that a separable abstract task t_S at detail level ℓ decomposes only into tasks at detail level $\ell + 1$, while other tasks decompose at the same level ℓ. Hence, separable tasks must occur at levels $\ell < L$. Let $s \downarrow \ell = \{a \in s | L_d(a) = \ell\}$ denote the projection of the state s to the detail level ℓ. A **planning problem** is defined as a tuple $\Pi = (D, s_0, t_0)$, where D is the planning domain, $s_0 \in \mathcal{P}(A)$ is the initial world state, and $t_0 \in T_a$ is the initial abstract task to be decomposed. If a goal is needed, it is added as an end task.

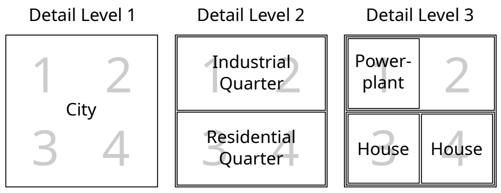

Figure 1: The different detail levels of the example domain in the goal state.

All detail layers in the world state with index $\ell < L$ consist solely of derived atoms and depend only on the atoms at the next finer detail level $\ell + 1$, thereby determining how information is abstracted in the world state. Let $D_{\text{der}} \subset A$ be the set of derived atoms (Edelkamp and Hoffmann 2004), which includes all atoms that are not on the most concrete (non-abstract) detail level. For each derived atom $a_{\text{der}} \in D_{\text{der}}$ at detail level ℓ, there exists a propositional formula $\phi_{a_{\text{der}}} \in \Phi$ involving only atoms from the next detail level ($\ell + 1$). Formally:

$$a^{\ell}_{\text{der}} := \phi_{a_{\text{der}}}(a^{\ell+1}_1, a^{\ell+1}_2, \ldots, a^{\ell+1}_k)$$

where $a^{\ell+1}_i$ denotes atoms at detail level $\ell + 1$.

This definition of derived atoms is more restrictive than in PDDL 2.2 (Edelkamp and Hoffmann 2004), and no dependency cycles are allowed, even across multiple predicates.

We restrict our attention to domains with *finite task depth*. The *task depth* of a task is defined as the length of the longest sequence of tasks obtainable through successive method applications. For any domain D, let $\mathrm{depth}_{\max}(D)$ denote its maximum task depth. We assume that there exists a fixed natural number $\Delta \in \mathbb{N}$ such that $\mathrm{depth}_{\max}(D) \leq \Delta$ for every domain D considered in this paper.

Example Domain To illustrate HWSP concepts, we consider a city planning domain with three abstraction levels. The domain models a 2×2 city containing 2 quarters, each with 2 building cells. Planning tasks include constructing the city at level 1, individual quarters at level 2 (industrial or residential), and placing specific buildings at level 3. Industrial quarters require a single power plant, while residential quarters need two houses. Certain cells may be blocked, preventing construction, and power requirements can create dependencies between quarters depending on the preconditions.

We formalize this as domain $D = (A, T_a, T_p, M, 3, L_d, D_{\text{der}}, \Phi)$ for a 2×2 city with quarters $q_j = \{2j-1, 2j\}$ for $j \in \{1, 2\}$. The atoms are distributed across three levels: level 1 contains $\{\text{city\_built}, \text{has\_power\_city}\}$, level 2 contains $\{\text{industrial\_built}_{q_j}, \text{ residential\_built}_{q_j}, \text{ has\_power}_{q_j}\}$ for $j \in \{1, 2\}$, and level 3 contains $\{\text{house}_i, \text{ powerplant}_i, \text{ blocked}_i\}$ for $i \in \{1, 2, 3, 4\}$.

The tasks include build_city : $\langle \emptyset, \{\text{city\_built}\} \rangle$ (separable), a universal quarter task (separable) build_quarter$_{q_j}$ and quarter tasks (abstract) build_$type_{q_j}$: $\langle \emptyset, \{type\text{\_built}_{q_j}\} \rangle$ for $type \in \{\text{industrial, residential}\}$ and $j \in \{1, 2\}$. Primitive tasks are place_$building_i$: $\langle \{\neg\text{blocked}_i\}, \{building_i\} \rangle$ for $building \in \{\text{house, powerplant}\}$ and $i \in \{1, 2, 3, 4\}$.

The derived atoms capture hierarchical relationships: $\text{industrial\_built}_{q_j} \equiv \vee_{i \in q_j} \text{ powerplant}_i$; $\text{residential\_built}_{q_j} \equiv \wedge_{i \in q_j} \text{ house}_i$; $\text{has\_power}_{q_j} \equiv \vee_{i \in q_j} \text{ powerplant}_i$; $\text{has\_power\_city} \equiv \vee_{j \in \{1,2\}} \text{ has\_power}_{q_j}$; $\text{city\_built} \equiv (\exists j \text{ industrial\_built}_{q_j}) \wedge (\exists k \text{ residential\_built}_{q_k})$. Each separable task has mandatory effects *meff* corresponding to its completion predicate, while houses can additionally have preconditions $prec_t = \{\text{has\_power\_city}\}$.

Methods for the domain (totally ordered) are: m_{city}: build_city $\Rightarrow$ (build_quarter$_{q_1}$, build_quarter$_{q_2}$); $m_{q_j,\text{res}}$: build_quarter$_{q_j}$ $\Rightarrow$ (build_residential$_{q_j}$); $m_{q_j,\text{ind}}$: build_quarter$_{q_j}$ $\Rightarrow$ (build_industrial$_{q_j}$);
m_{res,q_j}: build_residential$_{q_j}$ $\Rightarrow$ (place_house$_{i_1}$, place_house$_{i_2}$), $i_1, i_2 \in q_j$; m_{ind,q_j}: build_industrial$_{q_j}$ $\Rightarrow$ (place_powerplant$_i$), $i \in q_j$.

Predictor To enable efficient decomposition, HWSP incorporates predictor functions (Staud 2023) that estimate separable abstract task outcomes.

Definition 1 (Predictor Functions)**.** *Given a world state $s \in \mathcal{P}(A)$ and a separable task $t_S \in T_S$, the predictor function returns*

$$\pi(s, t_S) := \big(\varepsilon(s, t_S), \sigma(s, t_S)\big) \in \mathcal{P}(A) \times \{\text{TRUE}, \text{FALSE}\},$$

where

- $\varepsilon : \mathcal{P}(A) \times T_S \to \mathcal{P}(A)$ *is an* effect predictor *that proposes a set of* non-mandatory *effects (an estimate for $peff_{t_S}$), and*
- $\sigma : \mathcal{P}(A) \times T_S \to \{\text{TRUE}, \text{FALSE}\}$ *is the* decomposition predictor *whose output is to decide in a planning process if a separable abstract task can be decomposed or not.*

The predictor can be fixed-effect rules, Conv2D, ResNet or any other approximation function (Staud 2023).

HTN plans with labeled occurrences For proofs of the upcoming theorems, we need a description of a plan that stores the decomposition tree:

Definition 2 (HTN plan with explicit decomposition tree)**.** *A* plan (or partial plan) *is a tuple*

$$Q := \langle V, E, r, \lambda_T, \lambda_M, \prec \rangle$$

that is obtainable from the initial task network through valid method decompositions, where

- *(V, E, r) is a finite rooted tree whose root is $r \in V$,*
- *V is partitioned into the set of* task nodes V_T *and the set of* method nodes V_M*,*
- *$\lambda_T : V_T \to \mathcal{L} \times (T_a \cup T_p)$ is a labeling function assigning each task node a* plan step $l : t$ *with $l \in \mathcal{L}$ and $t \in T_a \cup T_p$,* task : $V_T \to (T_a \cup T_p)$ *will return the task of a plan step,*
- *$\lambda_M : V_M \to M$ is a labeling function assigning each method node a method $m \in M$,*
- *$\prec$ is a strict total order on V_T (the task nodes) that is* consistent with the child order of each method node*,*

Edges alternate strictly between task and method nodes: if $(u, v) \in E$ with $u \in V_T$, then $v \in V_M$ and $\lambda_M(v) = \langle t, P_m \rangle$ where $\lambda_T(u) = l : t$ for some $l \in \mathcal{L}$; conversely, all children of a method node $u \in V_M$ are task nodes $v \in V_T$ whose plan steps $\lambda_T(v)$ correspond exactly, in order, to the plan steps in P_m.

Let $\mathsf{state\_at} : \mathcal{Q} \times V_T \to \mathcal{P}(A)$ be the function $\mathsf{state\_at}$ (Q, v) that returns the world state obtained from the initial state s_0 by applying, in the order $\prec$, the effects of all primitive task nodes $w \in V_T$ such that $w \prec v$ in plan Q. By Definition 5, all task nodes w such that $w \prec v$ must be fully decomposed before v is considered for decomposition. Hence, the cumulative world state $\mathsf{state\_at}(Q, v)$ is well-defined and includes only effects of primitive tasks. We define:

Definition 3 (Decomposability Test)**.** *Let Q be a plan and $v \in V_T$ a task node with $\lambda_T(v) = l : t$, and let $s := \mathsf{state\_at}(Q, v)$. We define $\mathsf{can\_decompose} : \mathcal{Q} \times V_T \to \{\text{TRUE}, \text{FALSE}\}$ by:*

$$\mathsf{can\_decompose}(Q, v) := \text{TRUE}$$

iff there exists a sequence of method applications starting from t such that:

1. *for every task t' in the resulting decomposition tree, the precondition of t' holds in the state in which t' is applied, and*
2. *all leaves in the decomposition subtree rooted at v are primitive tasks.*

Sibling-Rewriting Relation For two plans Q, Q' and a separable abstract task node $n_s \in V_T$ in Q, we write $Q \rightsquigarrow_{n_s} Q'$ iff Q' is obtained from Q by replacing only the subtree rooted at n_s. That is, the subtree at n_s is first removed (inverting its decomposition) and then replaced by a newly constructed decomposition (possibly different from the original one).

Definition 4 (Fully Decomposed Task)**.** *Let Q be a plan and $v \in V_T$ a task node. We define:* $\mathsf{is\_decomposed}(Q, v) := \forall u \in \text{leaves}_Q(v)\ \mathsf{task}(u) \in T_p$ *That is, v is* fully decomposed *in Q iff all of its leaf task nodes in the subtree rooted at v are primitive tasks.*

Definition 5 (Progressive Method Decomposition)**.** *Let $Q = \langle V, E, r, \lambda_T, \lambda_M, \prec \rangle$, $u \in V_M$ with parent $t \in V_T$, and method $\lambda_M(u) = \langle t_a, (l_1 : s_1, \ldots, l_k : s_k) \rangle$.* $\mathsf{decompose}(Q, u)$ *is defined* $\iff \forall w \in V_T : w \prec t \Rightarrow \mathsf{is\_decomposed}(Q, w)$.

If defined, $\mathsf{decompose}(Q, u) = Q' = \langle V', E', r, \lambda'_T, \lambda_M, \prec' \rangle$ *where:*

$$V' = V \cup \{v_1, \ldots, v_k\} \text{ (fresh task nodes)}$$

$$E' = E \cup \{(u, v_i) : i = 1, \ldots, k\}$$

$$\lambda'_T(v_i) = \mathsf{newlabel}(l_i) : s_i \text{ for } i = 1, \ldots, k;\ \lambda'_T(v) = \lambda_T(v)$$

$$\begin{aligned} \prec' &= \big(\prec \setminus \big(\{(w, t) \mid w \prec t\} \cup \{(t, z) \mid t \prec z\}\big)\big) \\ &\cup \quad \{(w, v_i) : w \prec t, i = 1 \ldots k\} \\ &\cup \quad \{(v_i, z) : t \prec z, i = 1 \ldots k\} \\ &\cup \quad \{(v_i, v_{i+1}) : i = 1, \ldots, k-1\} \end{aligned}$$

The above definitions will be used in subsequent proofs.

HWSP Planning Algorithm

The most important difference between HTN planning and HWSP is the planning process (Staud 2023). In HTN planning, we decompose abstract tasks until only primitive tasks remain. In HWSP, decomposition stops at separable abstract tasks, unless we are on the most detailed level. This allows the planner to generate a top-level plan composed of separable abstract tasks, which are expected to work under typical conditions; otherwise, backtracking is performed.

For each separable abstract task emitted, a new planning process is launched to generate a subplan that achieves its mandatory effects $\mathit{meff}_t \subseteq D_{\text{der}}$. As the effects are derived predicates, the actual goal is the union of their logical formulas $\phi_{a_{\text{der}}}, a_{\text{der}} \in \mathit{meff}_t$.

Definition 6 (Planning Process Solution)**.** *A solution for planning process at level ℓ is a plan Q where every leaf node is either: (1) a primitive task (*$\mathsf{task}(v) \in T_p$*) if $\ell + 1 = L$, or (2) a separable abstract task (*$\mathsf{task}(v) \in T_S$*) if $\ell + 1 < L$ and all preconditions of these tasks must hold at execution time.*

Definition 7 (Planning Process Interface)**.** *A* planning process *(PP) is a 5-tuple $(D, I, \ell, \mathcal{I}, \mathcal{F})$ where:*

- *D is the planning domain*
- *$I \in I_S$ is the planning state of the planning process, where I_S is the set of all planning states. Given a separable task t,* $\mathsf{Init}(t) \in I_S$ *produces the initial planning state for planning t,*
- *$\ell \in \{1, ..., L\}$ is the current detail level*
- $\mathcal{I} : I_S \times \mathcal{P}(A) \to I_S \times \big((\mathcal{L} \times (T_a \cup T_S \cup T_p)) \cup \{\mathit{prec\_failure}, \mathit{proc\_failure}, \mathit{finished}\}\big)$
  Given the current planning state and the world state, this function returns the updated planning state and either the next plan step $l : t$ to be added to the plan or a status indicator.
- $\mathcal{F} : I_S \times \mathcal{P}(A) \times T_S \to I_S \times \mathbb{N}$
  Handles the failure of a previously generated separable task $t \in T_S$ under the given world state. Returns the updated planning state and the number of previously generated tasks to retract (i.e., undo). The next call to $\mathcal{I}$ will generate a new alternative task or proc_failure which indicates that the planning process failed.

Internally, depending on the implementation, $\mathcal{F}$ may trigger backtracking or replanning.

Implementation Notes. The interface can be implemented using various search strategies. In an *MCTS-based* approach (Browne et al. 2012), backtracking simply returns to a previous node and selects a task with a lower visit count than the one discarded. An alternative is *HTN with repair* (Höller et al. 2020b), which rolls back the failed plan, removes the faulty task, and tries to complete a new plan by reusing parts of the old one.

Example In the city example, a planning process generates a plan to construct the buildings at the highest detail level 3 when given a quarter task as input.

Relationship to classical HTN planning. Unlike classical HTN problem spaces (Alford et al. 2012) where nodes represent single planning problems, HWSP operates over a hierarchy of interconnected planning processes, each working at different abstraction levels. If the domain provides only one level ($L = 1$) and contains no separable abstract tasks, HWSP degenerates into standard HTN planning: the single PP is invoked exactly once, no spawning occurs, and the algorithm behaves identically to a progression-based HTN search (i.e., standard forward-chaining HTN planning, as used in planners like SHOP2 (Nau et al. 2003)).

Algorithm 1: HTN planning with stack-based algorithm. Detailed backtracking procedures using $\mathcal{F}$ are implementation-dependent and are therefore omitted.

```
1:  **Input:** Domain $D = (A, T_a, T_p, M, L, L_d, D_{\text{der}}, \Phi)$
2:  **Input:** Problem $\Pi = (D, s, t_0)$
3:  **Initialize:** Stack $S \leftarrow \langle(D, \mathsf{Init}(t_0), 1, \mathcal{I}, \mathcal{F})\rangle$ , MainPlan $p_M \leftarrow \langle\ \rangle$
4:  **while** $S \neq \emptyset$ **do**
5:      $\langle D, I, \ell, \mathcal{I}, \mathcal{F}\rangle \leftarrow S.top()$
6:      $\langle I', x\rangle \leftarrow \mathcal{I}(I,\ s \downarrow \ell)$
7:      update $S.top()$ by replacing I with I'
8:      **if** $x =$ finished **then**
9:          $S.pop()$
10:     **else if** $x =$ prec_failure **or** $x =$ proc_failure **then**
11:         **backtrack** via using $\mathcal{F}$ or by selecting new
12:         planning processes.
13:     **else**                                   ▷ $x$ is a labeled step
14:         let $x = (l : t)$
15:         **if** $t \in T_p$ **then**
16:             $p_M \leftarrow p_M \oplus \langle l : t\rangle$
17:             $s \leftarrow \mathsf{step}(s, t)$
18:         **else if** $t \in T_S$ **then**      ▷ assume $L_d(t) + 1 < L$
19:             $S.push((D, \mathsf{Init}(t), L_d(t) + 1, \mathcal{I}, \mathcal{F}))$
20:         **end if**
21:     **end if**
22: **end while**
23: **return** MainPlan $p_M$
```

Definition 8 (Overall Solution)**.** *A solution to the overall planning problem* $\Pi = (D, s_0, t_0)$ *is a plan Q where: (1) every leaf node is a primitive task (*$\mathsf{task}(v) \in T_p$*), and (2) all primitive task preconditions hold during sequential execution from* s_0.

Example In the city domain, the main planning algorithm would be coordinating the planning processes so that a complete city is built.

Domain Classes

The performance of planning algorithms, such as Hierarchical Task Network (HTN) planning and its extensions like Hierarchical World State Planning (HWSP), can be significantly influenced by the structure of the planning domain. This observation aligns with the downward refinement property introduced by Bacchus and Yang (Bacchus and Yang 1991), which emphasizes how domain structure impacts the

Algorithm 2: HWSP Planning Process Implementation: Core search procedure that handles three task types: primitive tasks (add to plan directly), separable abstract tasks (handled via predictive delegation to subprocesses), and non-separable abstract tasks (expanded using method decomposition). The interface functions $\mathcal{I}$ and $\mathcal{F}$ coordinate between the main algorithm and individual planning processes.

```
1:  **function** $\mathcal{I}$(I, s) *The planning process maintains a local plan and incrementally returns tasks to the main algorithm. If no plan exists, it calls `SolveLevel` to generate one using the current world state projection. Returns the next unexecuted task from the plan, or status indicators: `finished` (all tasks complete), `prec_failure` (preconditions violated), or `proc_failure` (no valid plan exists).*
2:  **end function**
3:  **function** $\mathcal{F}$(I, s, t) *Called when separable task t fails during execution at a lower abstraction level and attempts to backtrack to find a new solution. Returns the number of previously-generated tasks that must be retracted from the main algorithm's plan.*
4:  **end function**
5:  **function** SOLVELEVEL($Q, s$)
6:      $v = \mathsf{nextTaskNode}(Q)$, $\langle l : t\rangle \leftarrow \lambda_T(v)$
7:      **if** $t \in T_p$ **then** ▷ primitive task, add to plan directly
8:          $s' \leftarrow \mathsf{step}(s, t)$
9:          $(p_R, s'') \leftarrow$ SOLVELEVEL$(Q, s')$
10:         **if** $p_R =$ **failure then**
11:             **return failure**
12:         **end if**
13:         **return** $(\langle l : t\rangle \oplus p_R, s'')$
14:     **else if** $t \in T_S$ **then** ▷ separable abstract, treated like primitive task in planning process
15:         $(\varepsilon, \sigma) \leftarrow \pi(s, t)$ ▷ Get predicted effects
16:         **if** $\sigma =$ FALSE **then**
17:             ▷ Predictor says decomposition will fail
18:             **return failure**
19:         **end if**
20:         ▷ Apply mandatory + predicted effects
21:         $s' \leftarrow \mathsf{step}(s, \mathit{meff}_t \cup \varepsilon)$
22:         $(p_R, s'') \leftarrow$ SOLVELEVEL$(Q, s')$
23:         **if** $p_R =$ **failure then**
24:             **return failure**
25:         **end if**
26:         ▷ Include separable task in solution
27:         **return** $(\langle l : t\rangle \oplus p_R, s'')$
28:     **else**    ▷ $t \in T_a \setminus T_S$ = non-separable abstract
29:         ▷ Select method and decompose it.
30:         $\mathcal{M} \leftarrow \mathsf{methods}(t)$    $m = \mathsf{select}(\mathcal{M})$
31:         $u := \mathsf{makeMethodNode}(v, m)$
32:         $Q' \leftarrow \mathsf{decompose}(Q, u)$
33:         $(p', s') \leftarrow$ SOLVELEVEL$(Q', s)$
34:         **if** $p' \neq$ **failure then**
35:             **return** $(p', s')$
36:         **end if**
37:         **return failure**
38:     **end if**
39: **end function**
```

feasibility and efficiency of hierarchical refinement. To formalize these influences in the context of separable abstract tasks, we introduce domain classes based on task interaction and backtracking behavior. In particular, we analyze how sibling tasks (parallel subtasks at the same abstraction level) may or may not interfere with one another.

Let us consider our urban planning example again, where a city manager plans construction in different quarters. We illustrate the three domain classes (EX1 – EX3) by toggling certain constraints in the same base domain, as summarized in Table 1.

Class	Active constraints
EX1	No blocked cells; no electricity requirement
EX2	Blocked cells present; no electricity requirement
EX3	Blocked cells present; we require has_power_city

Table 1: Configurations of the running example corresponding to the domain classes.

1. **EX1: Execution Guaranteed**
   Intuitive Definition: Execution (decomposition of the separable abstract task) is guaranteed once its preconditions hold in the current state.
   Example: In our urban planning scenario, when there are no blocked cells and no electricity requirement, the quarter can always be constructed.

   Definition 9 (Domain class EX1)**.** *A domain D is in **EX1** iff for every plan Q and every separable task occurrence $v \in V_T$ of Q, preconditions of* $\mathsf{task}(v)$ *are satisfied in* $\mathsf{state\_at}(Q, v)$*, then it holds* $\mathsf{can\_decompose}(Q, v) =$ TRUE

2. **EX2: Execution Can Fail but Won't Benefit from Backtracking into Siblings**
   Intuitive Definition: Execution can fail, but a different solution from another planning process on the same detail level won't help.
   Example: If we have blocked cells in the initial state (e.g., $\{\text{blocked}_2\} \subset s_0$), some quarter constructions can fail. For instance, build_residential$_{q_1}$ fails because it requires both cells 1 and 2 to be unblocked for house placement, but blocked$_2 \in s_0$. However, build_industrial$_{q_1}$ can still succeed using only cell 1 for power plant placement. Crucially, the decomposability of $build\_residential\_q_1$ depends only on $\{\text{blocked}_1, \text{blocked}_2\}$ and is independent of any choices made for build_industrial$_{q_2}$ or build_residential$_{q_2}$ in quarter q_2. When build_residential$_{q_1}$ fails, the planner backtracks to the parent build_city task and tries build_industrial$_{q_1}$ instead, without reconsidering previously completed quarters.

   Definition 10 (Domain class EX2)**.** *D is in **EX2** iff for all plans Q, separable occurrences $v \in V_T$ and for all sibling abstract task occurrences s located* strictly before *v in the depth-first ordering with* $L_d(\mathsf{task}(s)) = L_d(\mathsf{task}(v))$, $\mathsf{task}(s) \in T_S \wedge \mathsf{task}(v) \in T_S \wedge Q \rightsquigarrow_s Q'$ $\implies \mathsf{can\_decompose}(Q, v) = \mathsf{can\_decompose}(Q', v)$.

   In words, replacing the internal subtree of an earlier separable sibling s does not affect the decomposability of a later separable task at the same detail level.

3. **EX3: Execution Can Fail and May Benefit from Backtracking into Siblings**
   Intuitive Definition: Execution can fail, and another solution from a prior planning process might help.
   Example: When residential quarters require power to become buildable, it creates dependencies between quarter choices. Attempting build_residential$_{q_1}$ as the first quarter fails because the precondition requires has_power_city to be true, but no power plants exist yet. The planner must backtrack to build_quarter$_{q_1}$ and try build_industrial$_{q_1}$ first to establish power generation, then build_residential$_{q_2}$ can succeed.

   Definition 11 (Domain class EX3)**.** *The class **EX3** includes all HWSP planning domains.*

   This formulation represents the general case without assuming task independence. Note, by propagating power availability upward via derived atoms (e.g., has_power$_{q_j}$, has_power_city), remove build_quarter$_{q_j}$, making build_residential$_{q_j}$, build_industrial$_{q_j}$ to separable tasks and including the electricity check in preconditions of, cross-sibling dependencies are captured at higher levels. This effectively transforms the domain from EX3 to EX2.

Separable abstract tasks isolate planning to local *islands* of the world state. How strongly such islands interact (EX1 – EX3) determines the degree to which global backtracking is required (in Algorithm 1).

Proposition 1 (Independence chain)**.**

$$EX1 \subset EX2 \subset EX3.$$

Proof. Immediate from the definitions □

Domain Conditions

Instead of a full classification, we formulate sufficient conditions which analyze how the preconditions of subtasks relate to the guarantees provided by their abstract parent. If every subtask t in a separable abstract task T has $\text{prec}(t) \subseteq \text{prec}(T)$, or $\text{prec}(t)$ can be established by effects of other subtasks of T (according to the decomposition defined by the chosen method), then the decomposition is self-contained: all requirements are either inherited or locally achievable. In this case the domain behaves like **EX1**, since successful execution is guaranteed once $\text{prec}(T)$ holds.

If, more generally, each subtask may also rely on derived predicates from a lower abstraction level, the task network is still independent of its siblings. This setting corresponds to **EX2**: execution may fail locally, but feasibility does not depend on choices made in other sibling tasks.

Backtracking

In HWSP, backtracking occurs when a planning process fails to find a viable solution. At this point, the algorithm must reconsider its previous decisions and explore alternative paths.

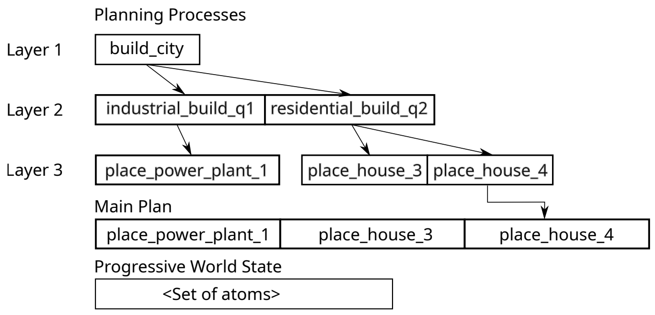

Figure 2: A typical state of the HWSP algorithm.

This is where the domain classes (EX1, EX2, and EX3) come into play, as they significantly influence the backtracking behavior.

- **EX1**: In domains where execution is guaranteed once preconditions of a separable abstract task are met, backtracking is relatively straightforward. Since each task's success is assured, the main planner never needs to backtrack due to separable abstract tasks.
- **EX2**: In EX2 domains, the failure of a separable abstract task requires restricted backtracking behavior.

  Proposition 2 (EX2 Backtracking Behavior)**.** *In an EX2 domain, if a separable abstract task v fails during decomposition, the planner can backtrack to the parent planning process without considering siblings of v.*

  Proof. By Definition 10, for any plan Q and separable task v, modifying any sibling s (with $s \prec v$) does not affect the decomposability of v. Formally: $\mathsf{can\_decompose}(Q, v) = \mathsf{can\_decompose}(Q', v)$ where $Q \rightsquigarrow_s Q'$.
  If v fails ($\mathsf{can\_decompose}(Q, v) = \text{FALSE}$), altering any prior sibling s through backtracking cannot make $\mathsf{can\_decompose}(Q', v) = \text{TRUE}$. Thus, the only recourse is to inform the parent process of v's failure. The parent must then generate a new separable abstract task to replace v or one of its siblings. If the parent exhausts all alternatives and fails, the failure propagates upward recursively. This eliminates the need to explore siblings of v, as their modifications cannot resolve v's failure under EX2's independence. The algorithm remains sound and complete (Staud 2022). □

- **EX3**: In these domains, where execution can fail and backtracking may help, we must perform full backtracking as in the original HWSP algorithm (Staud 2023).

Results

We compare *Hierarchical World-State Planning* (HWSP) on the three domain classes EX1 $\subset$ EX2 $\subset$ EX3 with a progression-based HTN algorithm.

Complexity model

Consider a fixed detail level ℓ. Let n_ℓ denote the number of *sibling* separable task occurrences emitted at that level, and let each sibling have at most k_ℓ decomposition methods. The refinement of *one* sibling induces a local search tree of branching factor $b_{\ell+1}$ and depth $h_{\ell+1}$; we abbreviate $B_{\ell+1} = b_{\ell+1}^{h_{\ell+1}}$.

Let N denote the total input size (i.e., the size of the domain and the initial task network). We assume that $n_\ell = \mathcal{O}(N)$; that is, the number of sibling tasks at each level grows at most linearly with the input.

We analyze the number of states visited during search, denoted as:

- S_{HTN}: states visited by classical HTN planning
- S_{EX1}, S_{EX2}, S_{EX3}: states visited by HWSP in the respective domain classes

State-space bounds

Theorem 3 (EX3 versus progression-based HTN)**.** *For any level ℓ*

$$S_{\text{HTN}} = B_{\ell+1}^{n_\ell}, S_{\text{EX3}} = n_\ell\, k_\ell^{n_\ell}\, B_{\ell+1}, \frac{S_{\text{HTN}}}{S_{\text{EX3}}} = \Theta\left(\frac{B_{\ell+1}^{n_\ell - 1}}{n_\ell}\right)$$

Proof. Classical HTN planning *inlines* the local search of each sibling. Consequently the global search tree contains $B_{\ell+1}$ possibilities for the first sibling, $B_{\ell+1}$ for the second, and so on, for a total of $B_{\ell+1}^{n_\ell}$ states.

In EX3, the parent process still explores every combination of high-level alternatives, hence $k_\ell^{n_\ell}$ parent choices. Once such a combination of high-level method choices $(m_1, \ldots, m_{n_\ell})$ has been fixed, each sibling is then solved *independently* by its own local search; at this stage the parent does not interleave those searches. The resulting state count is therefore $n_\ell\, k_\ell^{n_\ell}\, B_{\ell+1}$. Taking the quotient of the two expressions yields the claimed factor. □

Theorem 4 (EX2 versus EX3)**.** *For any level ℓ*

$$S_{\text{EX2}} = n_\ell\, k_\ell\, B_{\ell+1}, \qquad \frac{S_{\text{EX3}}}{S_{\text{EX2}}} = \Theta\left(\frac{k_\ell^{n_\ell - 1}}{n_\ell}\right)$$

Proof. Definition 10 states that substituting an *earlier* sibling by an alternative decomposition never changes the decomposability of a *later* sibling. Hence, when a sibling fails, the planner backtracks exactly one level up, replaces *only* that sibling, and leaves all others untouched. Worst case is it will test every one of the k_ℓ methods before a success or final failure is found. Since there are n_ℓ siblings, at most $n_\ell\, k_\ell$ alternatives are generated, each of which spawns one local search of size $B_{\ell+1}$. The product yields S_{EX2}. Dividing by S_{EX3} establishes the gap. □

The bounds given in Theorems 3 and 4 are tight in the following sense: for each setting of n_ℓ, k_ℓ, and $B_{\ell+1}$, there exists a family of domains and initial task networks for which the number of visited states matches the bound asymptotically.

HTN Tightness Example. Let the initial task be a sequence of n_ℓ abstract tasks, each with k_ℓ methods, where each method expands to a local plan space of size $B_{\ell+1}$ (e.g., a binary tree of depth $\log_2 B_{\ell+1}$). If no heuristic guidance is used, the planner must exhaustively search all $B_{\ell+1}^{n_\ell}$ combinations.

EX3 Tightness Example. Consider a domain with n_ℓ sibling tasks (e.g., city quarters). Each has k_ℓ decomposition methods, exactly one of which builds a power plant. Suppose that every quarter requires electricity to decompose, and power flows only from left to right: quarter q_j can be decomposed only if some earlier sibling q_i with $i < j$ contains a `power plant`. This induces a causal dependency chain across siblings: choosing non-power plant methods early may prevent later decompositions, forcing backtracking and revision of earlier choices. In the worst case, it explores all $k_\ell^{n_\ell}$ global method assignments; for each one, it performs n_ℓ independent subsearches of size $B_{\ell+1}$. The total number of visited states is therefore $n_\ell\, k_\ell^{n_\ell}\, B_{\ell+1}$, which matches the upper bound for EX3. The cross-sibling dependency violates the EX2 invariance condition (Definition 10), so this domain lies strictly in EX3. Thus, the upper bound for S_{EX3} is *tight*.

EX2 Tightness Example. Construct a domain with n_ℓ separable sibling tasks (e.g., one per city quarter), each having k_ℓ distinct task alternatives. For each sibling, only one of its k_ℓ alternatives is decomposable, while the others are designed to fail due to local, hidden constraints (e.g., unobservable predicates or infeasible layouts). These constraints are specific to each sibling and cannot be inferred or affected by the other siblings' choices. When a sibling task fails, the parent planning process must replace it with another alternative, up to k_ℓ times in the worst case. Each attempted task triggers a local planning process that explores a subsearch space of size $B_{\ell+1}$. Since all sibling tasks operate over disjoint parts of the world state and have no shared predicates, changing one sibling's decomposition does not influence the decomposability of another – satisfying the EX2 invariance condition (Definition 10). The total number of visited states in this construction is $n_\ell k_\ell B_{\ell+1}$, matching the upper bound for EX2 domains and proving its tightness asymptotically in the worst case.

On EX1. Once the preconditions of a separable task are satisfied, its decomposition cannot fail. Inside the planning processes, however, non-monotonic search (e.g., local backtracking) is still permitted. Therefore EX1 improves the constant factors of Theorem 4 but does not change its asymptotic form. More precisely, the number of search states in EX1 is $S_{\text{EX1}} = n_\ell\, B_{\ell+1}$, as each of the n_ℓ separable tasks is successfully decomposed on the first attempt, requiring a single local search of size $B_{\ell+1}$ per task.

Corollary 5 (Strict inclusion chain)**.** *It holds* $S_{\text{EX1}} < S_{\text{EX2}} < S_{\text{EX3}} < S_{\text{HTN}}$ *as each transition removes one exponential factor in the input size* N*, assuming* $n_\ell = O(N)$ *and* $L > 1$.

Connection to classical complexity theory

Bacchus and Yang's *Downward Refinement Property* (DRP) (Bacchus and Yang 1991) formalizes a key structural criterion for efficient hierarchical planning: once a high-level plan is found, it can be refined monotonically without modifying earlier decisions. Our EX1 domain class satisfies DRP by construction in Algorithm 1, as successful decomposition is guaranteed once preconditions are met. This is however not the case in Algorithm 2.

EX2 domains do not satisfy DRP in the strict sense, since the planner may backtrack and replace earlier siblings. However, they enforce a weaker property: the feasibility of decomposing a task is invariant under changes to the internal decomposition of prior siblings at the same abstraction level. Korf's macro-operator theorem (Korf 1985) predicts an exponential-to-linear collapse in ideal DRP-compatible settings, which applies to EX1 with respect to Algorithm 1 and partially to EX2. Note that while prior work (Alford et al. 2012) establishes bounds on HTN problem space size for termination guarantees, our analysis focuses on search effort complexity.

Effect of a (fallible) binary predictor

In the following, N is the combined size of the initial task network and domain description. Let $\hat{p}$ denote the predictor accuracy introduced below (to avoid notational conflict with earlier polynomials). In HWSP every separable task t_S comes with a binary *decomposition predictor* $\sigma : \mathcal{P}(A) \times T_S \to \{\text{TRUE}, \text{FALSE}\}$. Intuitively, $\sigma(s, t_S) = \text{TRUE}$ says that local planning *ought* to succeed.[1] Mis-predictions are the only reason why an EX2 planner ever needs to backtrack.

Error model. For the sake of analysis we assume that, *conditioned on the current state*, the predictor returns the correct answer with probability $\hat{p}$ (*accuracy*) and the wrong answer with probability $\bar{p} = 1 - \hat{p}$.

Theorem 6 (Expected search effort with a fallible predictor)**.** *Let* $S^\star_{\text{EX2}} = n_\ell B_{\ell+1}$ *be the number of search states when the predictor is* perfect*, so that each task is solved on the first attempt* ($k_\ell = 1$ *effectively).*

Assume that, each time a separable task is encountered, the predictor is correct with probability $\hat{p}$ *and wrong with probability* $\bar{p} = 1 - \hat{p}$ *(independently of previous calls). If the predictor errors, the parent process tries a different alternative; at most* $k_\ell - 1$ *further alternatives exist. Then the expected number of visited states is bounded by*

$$\mathbb{E}[S_{\text{EX2}}(p)] \;\le\; S^\star_{\text{EX2}}\,\big(1 + \bar{p}\,(k_\ell - 1)\big).$$

Proof. Let n_ℓ be the number of separable task occurrences at level ℓ, and let each task require a local search of size $B_{\ell+1}$ once a valid decomposition is selected. Under a perfect predictor, each task is decomposed successfully on the first try, so the total number of visited states is: $S^\star_{\text{EX2}} = n_\ell B_{\ell+1}$. Now consider a fallible predictor with accuracy $\hat{p}$, and let $\bar{p} = 1 - \hat{p}$ denote the probability of an incorrect prediction. If the predictor fails (predicts incorrectly, whether a false positive or false negative), the planner must backtrack and try a different decomposition. Since each task has at most k_ℓ possible alternatives, the number of additional alternatives to try after a misprediction is at most $k_\ell - 1$.

Thus, the expected number of task attempts for each separable task is at most: $1 \cdot \hat{p} + (1 + \beta) \cdot \bar{p}$, with

[1] We treat σ as a black box: it may be a learned classifier, a logical test or a mix thereof.

$\beta \leq k_\ell - 1$. Substituting, we obtain the upper bound: Expected attempts per task $\leq 1 + \bar{p} \cdot \beta \leq 1 + \bar{p} \cdot (k_\ell - 1)$. Multiplying this overhead by the base cost $S^\star_{\mathrm{EX2}}$ yields: $\mathbb{E}\left[S_{\mathrm{EX2}}(p)\right] \leq S^\star_{\mathrm{EX2}} \cdot (1 + \bar{p} \cdot (k_\ell - 1))$, which proves the claim. □

A perfect predictor collapses EX2 to EX1. Setting $\hat{p} = 1$ makes the overhead term $(1 + \bar{p}\,\beta)$ equal to 1, so every separable task is decomposed successfully on the very first try. This eliminates *all* backtracking in Algorithm 1 – precisely the hallmark of the EX1 class (Def. 9). Formally, the planner's behavior on this instance becomes behaviorally indistinguishable from running HWSP on an EX1 domain, although the structural EX1 property may not hold for all instances. □

Interpretation. Theorem 6 shows that the predictor's accuracy enters the search complexity as a *linear* scalar factor. Even modest accuracies (say $\hat{p} = 0.9$) leave the $\Theta\big(k_\ell^{n_\ell - 1}\big)$ gap of Theorem 4 largely intact (here treating n_ℓ as fixed for ratio analysis), because $\bar{p} \leq 0.1$ (Staud 2023).

Conclusion

We analyzed Hierarchical World State Planning (HWSP), a planning framework that extends classical HTN planning through multi-level state abstraction and separable abstract tasks. While HWSP was introduced in prior work, its theoretical properties and complexity characteristics had not been formally studied.

References

Alford, R.; Shivashankar, V.; Kuter, U.; and Nau, D. 2012. HTN Problem Spaces: Structure, Algorithms, Termination. In *Proceedings of the International Symposium on Combinatorial Search*, volume 3, 2–9.

Bacchus, F.; and Yang, Q. 1991. The Downward Refinement Property. In *IJCAI*, 286–293.

Behnke, G.; Höller, D.; and Biundo, S. 2019. Finding Optimal Solutions in HTN Planning-a SAT-based Approach. In *IJCAI*, 5500–5508.

Biundo, S.; and Schattenberg, B. 2001. From Abstract Crisis to Concrete Relief – A Preliminary Report on Combining State Abstraction and HTN Planning. In *ECP 2001*, 157–168.

Botea, A.; Enzenberger, M.; Müller, M.; and Schaeffer, J. 2005. Macro-FF: Improving AI Planning with Automatically Learned Macro-Operators. *Journal of Artificial Intelligence Research*, 24: 581–621.

Brafman, R. I.; and Domshlak, C. 2006. Factored Planning: How, when, and When Not. In *AAAI*, volume 6, 809–814.

Browne, C. B.; Powley, E.; Whitehouse, D.; Lucas, S. M.; Cowling, P. I.; Rohlfshagen, P.; Tavener, S.; Perez, D.; Samothrakis, S.; and Colton, S. 2012. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1): 1–43.

Dietterich, T. G. 2000. Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition. *Journal of Artificial Intelligence Research*, 13: 227–303.

Edelkamp, S.; and Hoffmann, J. 2004. PDDL 2.2: The Language for the Classical Part of IPC-4. In *International Planning Competition*.

Erol, K.; Hendler, J. A.; and Nau, D. S. 1994. HTN Planning: Complexity and Expressivity. In Hayes-Roth, B.; and Korf, R. E., eds., *Proceedings of the 12th National Conference on Artificial Intelligence, Volume 2*, 1123–1128.

Geier, T.; and Bercher, P. 2011. On the Decidability of HTN Planning with Task Insertion. In *IJCAI*, 1955–1961.

Höller, D.; Behnke, G.; Bercher, P.; Biundo, S.; Fiorino, H.; Pellier, D.; and Alford, R. 2020a. HDDL: An Extension to PDDL for Expressing Hierarchical Planning Problems. In *Proc. of the AAAI Conference on AI*, volume 34, 9883–9891.

Höller, D.; Bercher, P.; Behnke, G.; and Biundo, S. 2020b. HTN Plan Repair Via Model Transformation. In *German Conference on Artificial Intelligence (Künstliche Intelligenz)*, 88–101.

Knoblock, C. A. 1990. Learning Abstraction Hierarchies for Problem Solving. In *AAAI*, 923–928.

Korf, R. E. 1985. Macro-Operators: A Weak Method for Learning. *Artificial Intelligence*, 26(1): 35–77.

Korf, R. E. 1987. Planning as Search: A Quantitative Approach. *AI 87*, 33(1): 65–88.

Kulkarni, T. D.; Narasimhan, K.; Saeedi, A.; and Tenenbaum, J. 2016. Hierarchical Deep Reinforcement Learning: Integrating Temporal Abstraction and Intrinsic Motivation. *Advances in Neural Information Processing Systems*, 29.

Lotem, A.; and Nau, D. S. 2000. New Advances in GraphHTN: Identifying Independent Subproblems. In *Proc. AIPS 2000*, 206–215.

Marthi, B.; Russell, S.; and Wolfe, J. A. 2008. Angelic Hierarchical Planning: Optimal and Online Algorithms. In *ICAPS*, 222–231.

Nau, D. S.; Au, T.-C.; Ilghami, O.; Kuter, U.; Murdock, J. W.; Wu, D.; and Yaman, F. 2003. SHOP2: An HTN Planning System. *Journal of Artificial Intelligence Research*, 20: 379–404.

Olz, C.; and Bercher, P. 2023. A Look-Ahead Technique for Search-Based HTN Planning: Reducing the Branching Factor by Identifying Inevitable Task Refinements. In *Proceedings of the International Symposium on Combinatorial Search*, volume 16, 65–73.

Sacerdoti, E. D. 1974. Planning in a Hierarchy of Abstraction Spaces. *Artificial intelligence*, 5(2): 115–135.

Shivashankar, V.; Kuter, U.; Nau, D. S.; and Alford, R. 2012. Hierarchical Goal Networks: Initial Results. In *Proc. ICAPS 2012*, 235–243.

Staud, M. 2022. Urban Modeling via Hierarchical Task Network Planning. In *HPLAN Workshop*.

Staud, M. 2023. Integrating Deep Learning Techniques into Hierarchical Task Planning for Effect and Heuristic Predictions in 2D Domains. In *HPLAN Workshop*.

Tsuneto, R.; Erol, K.; Hendler, J.; and Nau, D. 1996. Commitment Strategies in Hierarchical Task Network Planning. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 536–542.