Variants of Enforced Hill Climbing

Eda Kaynar

Mat No.: 2023-052-319

Examiner: Prof. Dr. Malte Helmert

Supervisor: Dr. Clemens Büchner

July 11, 2026

Abstract

Enforced Hill Climbing is a heuristic search strategy in classical planning introduced first in 2001 by Hoffmann and Nebel, as part of the Fast Forward planning system. It combines a greedy search algorithm called Hill Climbing with Breadth-First Search to escape local minima. In this work, we compare the differences between the original Fast Forward implementation and an alternate implementation in the Fast Downward planning system, introduced first in 2006 by Helmert. The two implementations differ in regard to the use of duplicate detection, laziness of heuristic evaluation, and the usage of helpful actions/preferred operators to focus the search. We extend the configurable and parameterized system in Fast Downward to support both variants, Fast Downward and Fast Forward. We then analyze the impact of these implementation differences on search performance and highlight the trade-offs between the different approaches.

University of Basel

Switzerland

Table of Contents

1 Introduction

This thesis studies a heuristic search algorithm called *Enforced Hill Climbing (EHC)* (Hoffmann and Nebel, 2001), used in automated planning with the goal of solving problems by combining traditional Hill Climbing (Russell and Norvig, 1995) with Breadth-First Search to escape local minima. The main focus of this thesis is to explore different variants of EHC and the key differences between the EHC implementation in Fast Forward and the EHC implementation in the Fast Downward planning system, both introduced in the early 2000s.

1.1 Enforced Hill Climbing

The idea of *Enforced Hill Climbing (EHC)* was first introduced by Hoffmann and Nebel (2001). The introduced search algorithm EHC is a heuristic, greedy, incomplete search strategy designed for forward state-space planning. It is based on the traditional *Hill Climbing* algorithm, which incrementally moves from a current state to a neighbouring state with a strictly closer distance to the goal (Samuel, 1959). While standard Hill Climbing is fast, it can get stuck in local minima or plateaus where no immediate improvement is visible (Appleby et al., 1961); therefore, it might not reach a global minimum. However, Hill Climbing always terminates, although with no guarantee that the result returned will be a solution (in this case it will return the best state it has reached).

EHC extends this idea by introducing a systematic mechanism for escaping local minima. Instead of choosing the best immediate successor, like Hill Climbing, it performs a *Breadth-First Search* starting from the current state until it finds a successor with a strictly closer distance to the goal. This means that rather than relying only on local information, the algorithm temporarily explores the surrounding transition system (see Definition 2.3) with a blind (without knowing how close it is to the goal) search algorithm (Breadth-First Search). Once a better state is found (i.e. a state closest to the goal currently), the search moves to that state and repeats the process. This combination of greedy guidance and local Breadth-First exploration makes EHC significantly better at escaping local minima than pure Hill Climbing. The main concept of *Breadth-First Search* was first introduced by Moore (1959), as a blind search algorithm that traverses graphs layer by layer and uses a queue with a first-in-first-out (FIFO) structure. The effectiveness of EHC depends heavily on the quality of the arbitrary function guiding the search. If this arbitrary function misleads the search or if the problem contains states from which the goal is unreachable (dead ends), EHC may fail

to make progress. Because of cases like these, EHC is incomplete.

1.2 Fast Forward

Using this idea of EHC, Hoffmann and Nebel (2001) introduced the *Fast Forward (FF) planning system*, which uses EHC as a core search component. FF builds directly on EHC and extends it with powerful heuristic and pruning techniques. Its arbitrary function (heuristic) is derived from a relaxed version of the planning problem (defined in section 2.3). In this relaxed version, certain constraints of the original problem are simplified, allowing the system to estimate how far a state is from the goal more efficiently. FF combines the heuristic evaluation with a relaxed planning procedure that resembles a simplified version of GRAPHPLAN (Blum and Furst, 1997). This allows it to extract guidance information, such as estimates of goal distance and sets of helpful actions, which are the specific operators extracted from the planning graph to help achieve a goal state. These are then used by EHC to focus the search on directions that are more likely to lead to a solution.

These helpful actions, which are introduced alongside FF as one of the pruning techniques, help make the search more efficient. They restrict the set of successor states considered during search to those that appear promising (helpful) according to the relaxed plan. Together, these mechanisms can significantly reduce the branching factor and guide the search more directly towards goal states but are not guaranteed to do so.

1.3 Fast Downward

The *Fast Downward (FD) planning system* is a classical planner framework that solves deterministic planning problems and implements different algorithms (Helmert, 2006). While FF and FD follow the same basic ideas of EHC, FD introduces several important modifications to how the search is conducted. For instance, FD uses preferred operators, which help guide the search towards promising actions to reduce computational overhead in large transition systems.

Similar to FF, FD employs forward state-space search, where planning is formulated as a search problem over states and actions. However, the framework is designed to allow different search configurations to be combined easily. For the purposes of this thesis, the FD implementation of EHC serves as the basis for experimentation. The extension done to FD in this thesis exposes several configurable aspects of the search procedure, including the

choice between lazy and non-lazy heuristic evaluation, different duplicate detection strategies through global and local closed lists, and additional pruning techniques such as preferred operators and optional dead-end pruning. The flexibility and modularity of FD therefore make it particularly suitable for studying the behaviour of EHC under different implementation choices and for evaluating the impact of these choices on search efficiency and overall planning performance.

1.4 Key Differences in Search Strategy

FD introduces several important changes from the earlier heuristic planning system FF. These differences are most clearly seen in three aspects of the search process: the use of closed lists (how explored states are managed through local vs. global closed lists), the timing of heuristic evaluation (either eagerly at generation time or lazily at expansion time), and the integration of heuristic guidance (how the search is guided, through helpful actions in FF or preferred operators in FD). Each of these choices introduces different trade-offs between computational overhead, coverage, and completeness, which are analysed in detail in sections 4 and 7.

2 Background

This section establishes the formal notation used throughout the thesis. We define STRIPS planning tasks (Fikes and Nilsson, 1971), transition systems, heuristic functions, and the relaxed planning graph used by the FF heuristic. We further define the notions of helpful actions, preferred operators, and dead ends.

2.1 Planning Tasks, Transition System, and Actions

These definitions are presented in the order and manner that is most helpful in understanding the FF heuristic and other main topics mentioned in this work.

Definition 2.1 (STRIPS Planning Task)**.** A *STRIPS planning task* is a 4 tuple $\Pi = \langle V, I, G, A \rangle$, where:

- V is a finite set of *state variables*,
- $I \subseteq V$ is the *initial state*,
- $G \subseteq V$ is the set of *goals*,

- A is a finite set of *actions*, where for all actions $a \in A$, the following are defined:
  - $pre(a) \subseteq V$ are the *preconditions* of a,
  - $add(a) \subseteq V$ are the *add effects* of a,
  - $del(a) \subseteq V$ are the *delete effects* of a,
  - $cost(a) \subseteq \mathbb{N}_0$ is the *cost* of a.

The STRIPS planning task sets the background for the definitions of a state, applicable actions, and a goal state. For these definitions, let $\Pi = \langle V, I, G, A \rangle$ be a planning task. A *state* s of Π is a subset of V, i.e. $s \subseteq V$, representing the set of propositions that are true in that state. An action $a \in A$ is *applicable* in state s of Π if $pre(a) \subseteq s$. Applying an applicable action a to s yields the successor state $s' = \Big(s \setminus del(a)\Big) \cup add(a)$. A state s of Π is a *goal state* if $G \subseteq s$.

Definition 2.2 (Plan)**.** Let $\Pi = \langle V, I, G, A \rangle$ be a planning task. A *plan* for Π is a sequence of actions $\pi = \langle a_1, \ldots, a_n \rangle$ such that applying $a_1, \ldots, a_n$ successively to state I is well defined (every a_i is applicable in the state reached after applying $a_1, \ldots, a_{i-1}$) and the resulting state is a goal state.

Definition 2.3 (Transition System)**.** A *transition system* is a 6 tuple $\mathcal{T} = \langle S, L, c, T, s_0, S_G \rangle$, with:

- S is a finite set of *states*,
- L is a finite set of (transition) *labels*,
- c: $L \to \mathbb{R}_0^+$ is a label *cost* function,
- $T \subseteq S \times L \times S$ is the *transition relation*,
- $s_0 \in S$ is the *initial state*, and
- $S_G \subseteq S$ is the set of *goal states*.

Every STRIPS planning task $\Pi = \langle V, I, G, A \rangle$ induces a transition system in the natural way: the states are all subsets of V, the actions are the transition labels, the transition relation is defined by applicability, and the successor-state formula is represented in the applicable actions definition above, $s_0 = I$, and S_G is the set of all $s \subseteq V$ with $G \subseteq s$.

Definition 2.4 (Dead-end State)**.** A state s of Π is called a *dead end* if no plan exists from s to a goal state G, i.e. the goal is unreachable from s in the underlying transition system.

2.2 Heuristic Functions

Definition 2.5 (Heuristic Function)**.** Let $\mathcal{T}$ be a transition system with states S. A *heuristic function* for $\mathcal{T}$ is a function $h : S \to \mathbb{R}_0^+ \cup \{\infty\}$ mapping each state $s \in S$ to a non-negative number or infinity and estimates the distance from s to the closest goal state.

A heuristic h is *goal-aware* if $h(s) = 0$ for all goal states S_G, and a heuristic h is *admissible* if $h(s)$ never overestimates the true distance to the goal. By convention, $h(s) = \infty$ indicates that the heuristic has determined that s is a dead end, i.e. no goal state is reachable from s according to the heuristic's model of the problem.

2.3 Relaxed Planning Tasks and the Relaxed Planning Graph

Definition 2.6 (Relaxed Planning Task Π^+)**.** Given a planning task $\Pi = \langle V, I, G, A \rangle$, its *delete relaxation* is $\Pi^+ = \langle V, I, G, A^+ \rangle$, where for every action $a \in A$ there is a corresponding relaxed action a^+ with $pre(a^+) = pre(a)$, $add(a^+) = add(a)$, $cost(a^+) = cost(a)$ and $del(a^+) = \emptyset$. Once a proposition becomes true in the relaxed task, it remains true forever; actions can only add facts, never remove them.

Definition 2.7 (Relaxed Plan in STRIPS planning)**.** Let $\Pi = \langle V, I, G, A \rangle$ be a planning task. A *relaxed plan* for a state s of Π is a plan π^+ that achieves the goal state G from state s, within the relaxed planning task Π^+.

Definition 2.8 (Relaxed Planning Graph (RPG))**.** Let $\Pi^+ = \langle V, I, G, A^+ \rangle$ be a relaxed planning task. The *relaxed planning graph* of Π^+ represents *which* variables in Π^+ can be reached and *how*. It is a layered AND/OR graph consisting of alternating *variable layers* $V^0, V^1, V^2, \ldots$ and *action layers* $A^1, A^2, \ldots$, constructed as follows:

- The variable layer V^0 contains the variable vertex v^0 for all $v \in I$.
- The action layer A^{i+1} contains the action vertex a^{i+1} for action a if V^i contains the vertex v^i for all $v \in pre(a)$.
- The variable layer V^{i+1} contains the variable vertex v^{i+1} if the previous variable layer contains v^i, or the previous action layer contains a^{i+1} with $v \in add(a)$.

The graph contains a *goal vertex* g if $v^n \in V^n$ for all $v \in G$, where n is the last layer. The graph can be constructed for arbitrarily many layers, but stabilizes after a bounded number of layers, i.e. $V^{i+1} = V^i$ and $A^{i+1} = A^i$.

The layers are connected by the following directed edges:

- from v^i to a^{i+1} if $v \in pre(a)$ (*precondition edges*),
- from a^i to v^i if $v \in add(a)$ (*effect edges*),
- from v^i to v^{i+1} (*no-op edges*),
- from v^n to g if $v \in G$ (*goal edges*).

A *proposition* is a boolean state variable that can either be `TRUE` or `FALSE`. A *no-op* is an action whose precondition is a proposition and whose add effect is the same proposition.

In addition to ordinary actions, the RPG construction makes use of *no-op* actions. The graph is expanded layer by layer until either (a) $G \subseteq V_i$ for some layer i (the task is relaxed-solvable from s), or (b) $V_{i+1} = V_i$ (the graph has leveled off without covering G, meaning Π^+ and therefore Π has no solution). Because every fact, once introduced, persists in all subsequent layers (empty delete effects), the graph necessarily levels off after at most $|V|$ layers, which bounds the construction to polynomial time in the size of the task.

2.4 The FF Heuristic and GRAPHPLAN

Definition 2.9 (The h^{add} Heuristic)**.** Let *RPG* be a relaxed planning graph of Π^+. h^{add} is the heuristic function h that sums the predecessor vertex costs for action vertices a^i starting from the initial to the goal state (Bonet and Geffner, 2001).

h^{add} is pessimistic and assumes that there is no positive synergy when achieving action preconditions (Corrêa et al., 2022); therefore, it might overestimate the distance to the goal. To solve this issue, Hoffmann and Nebel (2001) introduced the FF heuristic, which estimates the distance to the goal more accurately than h^{add}.

Definition 2.10 (FF Heuristic)**.** The *FF heuristic* (h^{FF}) (Hoffmann and Nebel, 2001) calculates $h(s)$ by solving a *delete-relaxed* version of the planning task starting from the goal G until the initial state I is reached, marking either all predecessors if its a marked action/goal vertex, or marking one predecessor with minimal h^{add} value if the marked variable vertex v^i is in layer $i \geq 1$. The actions corresponding to the marked action vertices build a relaxed plan. The cost of this plan is the FF heuristic value.

The FF heuristic is *goal-aware* but *not admissible*. h^{FF} starts from the goal state and traces the paths back through the predecessors to the initial state, while adding all the action costs together to get the heuristic value. If no relaxed plan exists (i.e. Π^+ is unsolvable from s), the FF heuristic value of that state s is infinite and s is reported as a dead end. Because Π^+ is solvable whenever Π is solvable, an infinite heuristic value for s implies s is also a dead end in the original (non-relaxed) task. The converse does not hold: a state can be a dead end in Π while still appearing solvable in Π^+, since the relaxation ignores negative interactions between actions. This incompleteness of dead-ends is the reason EHC is also *incomplete*. It never misses a truly solvable state, but it can fail to recognise some genuinely unsolvable states.

Definition 2.11 (Backward Extraction)**.** Once a proposition layer V_i with $G \subseteq V_i$ is found, a relaxed plan is extracted by a backward pass. Starting from layer i with the goal-layer subgoals $G_i(s) := G$, for each layer $j = i, i-1, \ldots, 1$ the algorithm selects, for every proposition $v \in G_j(s)$ that is not already true in V_{j-1}, one action $a \in A_{j-1}$ with $v \in add(a)$. Every such selected action a is marked as part of the relaxed plan. The preconditions of all selected actions at layer $j-1$ become the new subgoal set $G_{j-1}(s)$:

$$G_{j-1}(s) = \bigcup_{a \text{ selected at layer } j} pre(a).$$

This process is repeated until layer 0 is reached, where $G_0(s) \subseteq V_0 = s$ holds by construction.

2.5 Goal Sets $G_i(s)$ and Helpful Actions

The backward-extraction process above produces, for each layer j, a set of *relevant subgoals* $G_j(s)$, the propositions that the relaxed plan must still achieve by layer j. In particular, $G_1(s)$ is the set of subgoals that the relaxed plan requires to be achieved already by the *first* fact layer V_1 (e.g. within one relaxed step of the current state s).

Definition 2.12 (Helpful Actions)**.** An action a applicable in s (i.e. $pre(a) \subseteq s$) is called a *helpful action* (Hoffmann and Nebel, 2001) for s if

$$add(a) \cap G_1(s) \neq \emptyset,$$

that is, a achieves *at least one* proposition the extracted relaxed plan requires at the first layer.

2.6 Preferred Operators

Preferred operators are a strict subset of *helpful actions*. A *preferred operator* is any applicable operator that *some* heuristic marks as promising, based on whatever internal structure that heuristic computes (for h^{FF} in Fast Downward, this coincides with the operators selected during relaxed-plan extraction, e.g. operators $a \in \pi^+$ with $pre(a) \subseteq s$) (Helmert, 2006).

Helpful actions are a strict generalization of preferred operators: every preferred operator is a helpful action. Both helpful actions and preferred operators are applicable actions.

3 EHC in Fast Forward

Standard Hill Climbing evaluates neighboring states and immediately moves to a successor with a strictly lower heuristic value if one exists (if no such state exists, then Hill Climbing terminates and treats this state as a local minimum). While this approach is computationally inexpensive, it becomes incomplete in the presence of local minima, dead ends, or flat heuristic regions where no strictly improving successor exists. EHC addresses this limitation by replacing the single-step successor selection with a Breadth-First Search whenever no immediate improvement is found. The algorithm operates in phases. Starting from the current state, all successor states are generated and evaluated. If a successor with a strictly lower heuristic value is found, the search immediately moves to that state and begins a new phase. Otherwise, the algorithm performs a systematic Breadth-First exploration outward from the current state until an improving state is discovered. Once an improving state is found, the temporary search frontier is discarded and the process repeats. The effectiveness of EHC depends heavily on the quality of the heuristic function. In the FF planner, EHC was combined with the FF heuristic, which is derived from a relaxed planning graph where delete effects are ignored. This heuristic estimates goal distance by extracting a relaxed plan and summing up costs of the actions required to achieve the goal under the relaxed problem formulation. Even though with this improvement, EHC can now overcome plateaus and local minima, the algorithm still remains incomplete because it can still become trapped in transition systems with dead ends (known source of incompleteness, see section 3.1) or fail to terminate in domains where heuristic guidance is misleading (not fitting choice of heuristic function). Some known reasons for EHC to be incomplete are listed in section 3.1.

The FF planner implements EHC using a local closed list, non-lazy heuristic evaluation, and helpful actions. Our implementation of EHC (FF-style), in the FD configurable system,

consists of two functions: an expansion function *expand* that generates and immediately evaluates all successors of the current state, and a main search loop *ehc* that drives the transitions between the Breadth-First Search and EHC iterations. Algorithm 1 shows the expansion function and the non-lazy EHC main loop.

Algorithm 1 EHC with Local Closed List and Non-Lazy Evaluation

```
1: function expand_nolazy(s)
2: for all applicable operators op in s do
3:    s' ← successor(s, op)
4:    if s' is new then
5:       open search node for s'
6:    end if
7:    compute h(s')
8:    insert (s', op) into OPEN
9: end for
10: mark s as closed
11: end function

12: function ehc_nolazy(s)
13: open ← create_open_list
14: closed_list ← ∅
15: while open list is not empty do
16:    (s, op) ← remove first entry from FIFO open list
17:    s' ← successor(s, op)
18:    if s' is closed then
19:       continue
20:    end if
21:    insert s' into closed_list
22:    evaluate heuristic h(s')
23:    if h(s') < h(current) then
24:       clear open list
25:       clear closed_list
26:       set current state to s'
27:       return ehc_nolazy(s')
28:    else
29:       expand s'
30:    end if
31: end while
32: end function
```

This local closed list only stores states generated during the current Breadth-First Search phase. Once a better heuristic state is found, the local closed list is cleared before the next phase begins. This implementation preserves duplicate detection locally while still allowing states to be revisited across different EHC phases.

Forward state-space search maintains a *search space* that tracks, for every state encoun-

tered, whether it has been *newly discovered* (status `new`), *generated but not yet expanded* (status `open`), *expanded* (status `closed`), or *identified as a dead end* (status `dead_end`). Each state starts with status `new`; it transitions to `open` when first inserted into the open list, to `closed` when expanded, and to `dead_end` when its heuristic value is found to be ∞. These four statuses are mutually exclusive at any point in time. The *open list* is the central frontier data structure of a forward search. It stores a set of entries representing search nodes that have been generated but not yet expanded. These concepts were used in all the pseudo-codes in this work.

The plan in EHC is kept track of by always appending the path that is followed in Breadth-First Search to the current plan (Hoffmann and Nebel, 2001). This process starts with an empty plan. Whenever progress is made in h-value, we take the segment that leads to this state and append the sequence of actions that brought us to that state. We repeat this until a dead end is reached or we have found a goal state.

3.1 Sources of Incompleteness in EHC

Enforced Hill Climbing is, in general, an *incomplete* search strategy, which means it is possible for EHC to terminate without finding a solution even though a plan exists. The fundamental source of incompleteness in EHC was identified by Hoffmann and Nebel (2001) shortly before Proposition 4 of their original paper, which states:

> *"If in one iteration breadth-first search for a better state fails, then enforced hill-climbing stops without finding a solution. This can happen because once enforced hill-climbing has chosen to include an action in the plan, it never takes this decision back. The method is therefore only complete on tasks where no fatally wrong decisions can be made. These are the tasks that do not contain "dead ends"."*

Formally, Hoffmann and Nebel (2001) prove that EHC is complete if the planning task is *dead-end free* (Proposition 4). In total, we have identified three sources of incompleteness, two of which were not mentioned in the original FF paper. These sources are independent of one another and can co-occur. The sources of incompleteness listed here are not exclusive; therefore there may be more that we are not aware of.

1. **Irreversibility/ "no way back".** If the underlying transition system is not symmetric (e.g. some actions have no inverse), then EHC might go down a path that it can never

get out from (dead-end) (Hoffmann and Nebel, 2001). Domains in which every action has an inverse (e.g. Blocks World, where every `stack` has a corresponding `unstack`) rule out this source of incompleteness, since any state visited remains reachable from any later state via the inverse actions.

2. **Permanent duplicate elimination (global closed list).** Marking every visited state as permanently closed can prevent EHC from ever revisiting a state through which the only solution path passes. Even though a solution might exist through the closed state, a global closed list would never be able to reach that solution. This is a property of the search algorithm, not of the problem instance.

3. **Pruning by helpful actions/ preferred operators.** If `PRUNE_BY_PREFERRED` is used and the relaxed plan extraction happens to mark only operators that lead away from the only solution path as preferred/helpful, all other successors including ones on a genuine solution path are discarded and never inserted into the open list.

4 Key Differences

This chapter summarizes and discusses the main differences in the implementation of EHC in FF and FD, which can be divided into three subcategories: global vs. local closed list, lazy vs. non-lazy heuristic evaluation, and preferred operators vs. helpful actions.

4.1 Global vs. Local Closed List

One of the most important implementation differences concerns duplicate detection. Each Breadth-First Search phase in FF uses a local closed list. In contrast, FD implementation uses a global closed list, meaning that every expanded state is marked and stored permanently in the closed list and will never be reconsidered during the search after it has been added to this list. This avoids redundant exploration and improves efficiency in large transition systems where many paths lead to the same state. The pseudo-code shown in section 6, Algorithm 1, can mostly be reused for showing the difference between local and global closed lists. For comparison, below is the pseudo-code for the implementation of a global closed list used to implement EHC in FD. Everything else remains the same as Algorithm 1 but line 25 is removed.

Algorithm 2 EHC with Global Closed List (with non-lazy heuristic evaluation)

1: **if** $h(s') < h(current)$ **then**
2: clear open list
3: set current state to s'
4: **return** ehc_nolazy(s')
5: **else**
6: expand s'
7: **end if**

However, global duplicate detection can also prevent the planner from revisiting states that may later become relevant through alternative search paths in order to reach the goal (see section 3.1). In certain domains, like Blocks World, this can lead to not finding a solution despite all actions being reversible. The following Example 4.1 demonstrates such a scenario.

Example 4.1 (Global/Local Closed List)**.** *Let* $\mathcal{S} = \langle S, A, cost, T, s_I, S_G \rangle$ *be a symmetric transition system and* $h(s)$ *a heuristic function where:* $S = \{s_I, s_2, s_3, s_4, s_5, s_G\}$,
$A = \{a, b, c, d, e, f, g\}$,
Heuristics ($h(s)$): $h(s_I) = 4, h(s_2) = 5, \quad h(s_3) = 1, \quad h(s_4) = 2, h(s_5) = 2, \quad h(s_G) = 0$,
Transition Function (T):$(s_I, b) = s_2, (s_2, b) = s_I, (s_I, a) = s_4, (s_4, a) = s_I, (s_2, c) = s_3, (s_3, c) = s_2, (s_3, d) = s_4, (s_4, d) = s_3, (s_4, e) = s_3, (s_3, e) = s_4, (s_4, f) = s_5, (s_5, f) = s_4, (s_5, g) = s_G, (s_G, g) = s_5$
$s_I = s_I$ *and* $S_G = s_G$

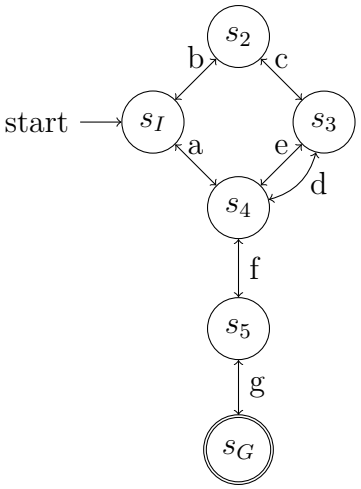


Figure 1: Example 4.1 "No Solution" for Global Closed List

Running EHC (FD-style) on this example, the FD algorithm would do the following:

1. expand s_I: open list: $\{s_2(h = 5), s_4(h = 2)\}$, closed list: $\{s_I\}$,
2. expand s_4: open list: $\{s_3(h = 1), s_5(h = 2)\}$, closed list: $\{s_I, s_4\}$,

3. expand s_3: closed list: $\{s_I, s_4, s_3\}$.

Since there is no way to the goal state from s_3 without going through s_4 again, the search is unsuccessful in reaching a goal state, even though s_3 is *not* a dead-end. Therefore, the final actions taken from s_I by FD would be:

$$a, e$$

EHC (FD-style) would not return to s_4 with action d because this state has already been seen and therefore been added to the global closed list. This leads to no solution (when there is clearly a solution) because the only way to get to the solution is to revisit closed states. With a local closed list, EHC (FF-style) can traverse through s_4 again, which results in the solution path:

$$a, e, d, f, g$$

With this Example 4.1 in mind, we can say that what originally holds in the FF paper about completeness of EHC described in section 3.1 by Hoffmann and Nebel (2001) is no longer true when using the global closed list. Knowing this, we can conclude:

Theorem 4.1. *When using global closed lists, there is no guarantee of completeness with EHC (even if the planning task is dead-end free).* ***Proof****: Shown by counterexample: Example 4.1.*

One key observation worth noting that can be extracted from *Figure 1* is that the solution cost and path found by the two variants, FF and FD, are indeed *different*. This is because they take different paths through the closed list (meaning FD is not able to revisit states marked closed). Nevertheless, this global closed list behaviour motivates the use of a local closed list. Instead of permanently storing all visited states, the local closed list is cleared after every Breadth-First Search phase of EHC.

4.2 Lazy vs. Non-Lazy Heuristic Evaluation

As already mentioned, FF uses non-lazy heuristic evaluation, where heuristic values are computed immediately when successors are generated. While this simplifies state ranking and search control, it may waste computation on states that are later discarded. By contrast, FD employs lazy heuristic evaluation, meaning that heuristic values are only computed when a state is selected for expansion. This results in reduced number of expensive heuristic computations in domains with large branching factors. For better understanding of what the lazy heuristic evaluation in FD does, below is some pseudo-code to help clarify this

implementation. Algorithm 3 is really similar to Algorithm 1 but there are some slight changes regarding the timing and content of insertion into the open list:

Algorithm 3 EHC with Local Closed List and Lazy Evaluation

```
1: function expand_lazy(s)
2: for all applicable operators op in s do
3:     s' ← successor(parent, op)
4:     insert (s', op) into OPEN
5: end for
6: mark s as closed
7: end function

8: function ehc_lazy(s)
9: open ← create_open_list
10: closed_list ← ∅
11: while open list is not empty do
12:     (parent, op) ← remove first entry from FIFO open list
13:     s' ← successor(parent, op)
14:     if s' is not new then
15:         continue
16:     end if
17:     insert s' into closed_list
18:     if h(parent) < h(current) then
19:         clear open list
20:         clear closed_list
21:         set current state to s'
22:         return ehc_lazy(s')
23:     else
24:         expand s'
25:     end if
26: end while
27: end function
```

This strategy avoids unnecessary heuristic computations because many generated states may never actually be expanded. A further difference between lazy vs. non-lazy is the pairs that are inserted into the open list. In the lazy implementation, the open list stores only the parent state together with the operator:

$$(parent\_state, op)$$

The successor state is reconstructed later when the entry is popped from the open list. In contrast, non-lazy heuristic evaluation open list only stores:

$$(successor\_state, op)$$

This way, the parent state never gets used for heuristic evaluation. The pseudo-code for this implementation of non-lazy heuristic evaluation was previously shown in section 3, in Algorithm 1. While non-lazy evaluation may perform unnecessary computations for states that are never expanded, it allows successor ranking to be based directly on the successor heuristic values rather than on deferred parent information. This means all successors in the lazy version have the same h-value and look equally promising.

4.3 Preferred Operators vs. Helpful Actions

Another important difference concerns search guidance and pruning. FF introduces helpful actions, which are actions selected from the relaxed plan generated during heuristic computation. Only actions considered promising according to the relaxed plan are expanded, significantly reducing the branching factor.

FD specializes this idea through preferred operators. Instead of strictly pruning non-preferred successors, preferred operators are prioritized within the open list while still allowing alternate successors to remain reachable. Preferred operators in FD are generated from the relaxed plan by checking whether an operator achieves a proposition whose preconditions are directly reachable from the current state. Both methods attempt to guide the search toward promising actions extracted from the relaxed plan. The pseudo-code for the original implementation of preferred operators used in FD can be seen below. The following snippets of helpful actions and preferred operators belong in the FF heuristic implementation as two separate functions.

Algorithm 4 Snippet of EHC with Preferred Operators

```
1: function mark_preferred_operators(s, goal)
2: for all goal propositions do
3:     recursively trace relaxed plan
4: end for
5: for all operators in relaxed plan do
6:     if all preconditions reached directly from current state then
7:         mark operator as preferred
8:     end if
9: end for
10: end function
```

The crucial differences between preferred operators and helpful actions lie in both *how* the set is computed and *how it is used*, which can be seen as two dimensions. Both helpful actions and preferred operators can be used for pruning and ranking.

For the first dimension, both FF and FD use the relaxed plan to identify a promising subset of applicable actions. FF computes *helpful actions*, the set of applicable operators satisfying $add(a) \cap G_1(s) \neq \emptyset$, while FD computes *preferred operators*, the set of applicable operators selected during relaxed-plan extraction with preconditions satisfied in the current state. Every preferred operator is a helpful action but not the other way around.

For the second dimension, once a promising subset has been identified, both systems support two ways of using it, controlled by the parameter `preferred_usage`:

- `PRUNE_BY_PREFERRED`: only successors reached via the selected subset are inserted into the open list. Non-selected successors are discarded entirely. This can significantly reduce the branching factor but may remove valid solution paths (see section 3.1), making EHC incomplete even on dead-end free tasks.

- `RANK_PREFERRED_FIRST`: all applicable operators are inserted into the open list, but successors reached via the selected subset are ranked ahead of the rest at the same h-value. No successor is ever discarded, so this mode preserves completeness with respect to the underlying Breadth-First Search.

FF originally uses helpful actions to prune in order to guide the search. In this implementation, an action is only considered helpful if at least one effect contributes to the relaxed plan and when the action achieves something that leads to reaching the goal.

Algorithm 5 Snippet of EHC with Helpful Actions

```
1:  function compute_helpful_actions(s)
2:  for all applicable operators o in current state s do
3:      helpful ← false
4:      for all effects e ∈ add(o) do
5:          if e ∈ G_1(S) then
6:              helpful ← true
7:              break
8:          end if
9:      end for
10:     if helpful then
11:         mark o as preferred
12:     end if
13: end for
14: end function
```

The pseudo-code for helpful actions can be seen above. Helpful actions aggressively reduce the branching factor and can significantly improve runtime performance in many benchmark

domains. However, unlike preferred operators, they may prune away necessary actions and therefore reduce search robustness in some planning tasks.

Lazy heuristic evaluation and preferred operators are connected through what information is actually available to the open list at insertion time. Under lazy evaluation, h is not computed when a successor is generated but only when it is later popped for expansion; every entry inserted during one expansion step therefore carries the *same* heuristic value, namely the h-value of the parent state, since no successor-specific evaluation has happened yet. This means the open list cannot use h itself to distinguish between successors of the same parent. Preferred operators fill this gap, because they are extracted from the same relaxed plan that produced the parent's h-value; marking a successor as preferred/non-preferred costs no additional heuristic computation, yet still gives the open list a way to order otherwise heuristically indistinguishable successors, which is precisely the role played by the `RANK_PREFERRED_FIRST` mode. The same connection holds in the pruning variant: since individual successors have not yet been evaluated, `PRUNE_BY_PREFERRED` acts as a substitute filter, restricting the open list to relaxed-plan operators before any real heuristic computation is spent on them, rather than filtering *after* evaluation. In both modes, preferred-operator information is what lets lazy evaluation retain useful successor ordering despite deferring the heuristic computation that would normally provide it.

5 Dead-End Pruning

In addition to the original FD implementation, this work introduces an *optional dead-end pruning* mechanism for EHC. The purpose of this extension is to avoid spamming the open list with dead-end states which is relevant for both FF and FD (but with minor effects, see section 7.4). A dead end is considered to be a state from which the planner cannot reach the goal according to the heuristic estimate, represented internally by an infinite heuristic value. In the original FD implementation, dead-end states are detected during heuristic evaluation and marked accordingly, but they may still be regenerated later through different paths in the search space. The implemented extension introduces a dedicated dead-end closed list, called `dead_end_list`, which stores the identifiers of all states previously classified as dead ends. Before expanding a newly generated successor state, the search checks whether the state already exists in this list. If it does, the state is immediately discarded without re-evaluation.

This modification is implemented directly inside the EHC search loop and integrates

naturally with both the global and local closed list variants. The implementation is controlled by the boolean parameter *dead_end*. When enabled, the planner maintains a persistent set of dead-end states across all EHC phases. The pruning mechanism is implemented in two stages; First, before evaluating a generated successor state, the planner checks whether the state has already been identified as a dead end. If the state already exists in the dead-end list, it is skipped immediately. Second, after heuristic evaluation, states with infinite heuristic values are inserted into the dead-end list. This ensures that every future occurrence of the same state can be pruned efficiently without recomputing the heuristic. This implementation can be made clear in the pseudo-code below.

Algorithm 6 Snippet of EHC with Dead-End Pruning

```
1: if dead_end_pruning then
2:    if state ∈ dead_end_list then
3:       continue
4:    end if
5: end if
6: if h(state) = ∞ then
7:    mark state as dead end
8:    dead_end_list.insert(state)
9:    continue
10: end if
```

The snippet shown above is intended to be in the ehc main loop of both lazy and non-lazy variants, where lines 1-5 from the snippet go before both s' checks in Algorithms 1 (line 18) and 3 (line 14), while the rest of the snippet goes after these lines. The primary advantage of dead-end pruning is the reduction of repeated heuristic evaluations. Since heuristic computation can be computationally expensive, avoiding repeated evaluation of known dead-end states improves search efficiency significantly in domains with many unsolvable subregions.

Another advantage is the reduction of unnecessary state expansions. Without pruning, cyclic domains may repeatedly revisit the same dead-end states and get stuck in an endless cycle. The dead-end list prevents these states from being reconsidered. This mechanism is particularly beneficial when combined with local closed lists. Since local closed lists are reset after every successful EHC phase, previously explored dead-end states could otherwise be revisited repeatedly in later phases.

However, the effectiveness of this approach depends on the reliability of the heuristic's dead-end detection. If the heuristic incorrectly classifies reachable states as dead ends, pruning may remove valid solution paths. For this reason, dead-end pruning should always first

be implemented as an optional configuration parameter rather than being enabled by default (this is not the case with the FD planning system; FD has reliable dead-end detection).

6 EHC in Fast Downward

Fast Downward (FD) is a domain-independent classical planning system first introduced by Helmert (2006). One of the major strengths of FD is its modular design, which enables researchers to modify individual components and combine different options for search independently. Although it may seem like FD's focus is heavily on implementations of EHC, FD more commonly provides implementations for other heuristic search algorithms such as eager and lazy best-first search, A* and many more. FD includes an implementation of EHC as a satisficing search strategy integrated within its modular search framework. The implementation follows the same core principles described in the FF planner while adapting them to FD's internal state representation and heuristic infrastructure. The FD implementation maintains a current search state and attempts to find a successor with a lower heuristic value. Once a state with an improved heuristic value is found, the open list is cleared and the search restarts from the improved state. Preferred operators can either be used to rank successors before non-preferred operators or to prune all non-preferred operators entirely. This behaviour is configurable through the `preferred_usage` parameter introduced in section 4. FD additionally supports lazy heuristic evaluation. As explained in section 4, this reduces heuristic computation overhead when many generated states are never expanded.

7 Performance Evaluation

This section presents the empirical results obtained from running the configurable EHC implementation described in section 4 on a set of classical planning benchmark domains. Each of the three implementation differences; local/global closed list, lazy/non-lazy evaluation, helpful actions/preferred operators, and dead-end pruning, are first evaluated in isolation, then the better-performing option is fixed, and the next parameter is then evaluated on top of the accumulated choice. These results are then checked for plausibility against the literature expectations from section 6. Before discussing the results, we clarify the two outcome categories that appear throughout the tables.

Definition 7.1 (Unsolved-Incomplete in FD)**.** The planner terminates with "No Solution -

FAILED" but also provides *no proof* of unsolvability. However, for this instance, no solution can be found by EHC because it's currently in an irreversible state (dead end) or cannot exit a local minimum.

Definition 7.2 (Unsolvable in FD)**.** The planner has *proven* that no plan exists for the given instance or this given domain.

As mentioned previously in multiple sections, there is a clear trade-off between the local and global closed list variants that is also visible in the data acquired. While the global closed list gets rid of redundant states and reduces overhead, it also falsely reports `unsolved-incomplete` when there clearly is a solution in the reported transition system. This is clearly visible in reversible domains with no dead-ends such as BLOCKS and TERMES that report `unsolved-incomplete` for the variants that use a global closed list and successful for the other variants with a local closed list. This is due to the only path to the solution in some transition systems being through closed states to escape local minima of h-values, which the global closed list variants fail to go through.

A natural question motivating the entire evaluation is whether any single configuration never produces an `unsolved-incomplete` outcome on a genuinely solvable instance, or whether one can be identified as the overall best or worst combination. We return to these questions in section 7.5. The metrics reported throughout are: *cost* (sum of total plan cost across commonly solved instances, lower is better), *coverage* (the number of instances solved across domains, higher is better), *unsolved-incomplete count* (lower is better), *evaluations* (the number of evaluations of states as geometric mean, lower is better), *expansions* (the number of expansions of states as geometric mean, lower is better), and *generated states* (the number of generated states as geometric mean, lower is better), and *total time* (geometric mean in seconds, sum of planner time and search time, lower is better).

7.1 Local vs. Global Closed List

Table 1 compares local closed list (`gc=0`) and global closed list (`gc=1`) variants, each reported at its best sub-configuration.

The results reveal a clear trade-off. The local closed list produces lower average plan cost and fewer `unsolved-incomplete` results, whereas the global closed list achieves substantially higher coverage and is significantly more efficient: evaluations, expansions, and generated states are each reduced by a factor of roughly $5 - 7\times$, and total time is reduced by 0.37 s.

Metric	Local	Global
Cost (sum)	**8,955,381**	8,963,266
Coverage	1,037	**1,279**
Unsolved-inc.	**487**	574
Evals (geo mean)	3,337	**715**
Expansions (geo mean)	3,296	**562**
Generated (geo mean)	10,734	**1,431**
Total time (geo mean)	0.57 s	**0.20 s**

Table 1: Local vs. global closed list at each variant's best sub-configuration.

This matches the knowledge that the global closed list eliminates re-expansion of already-visited states across EHC phases, which under the local list inflate the open list with stale edges. The higher `unsolved-incomplete` count under the global closed list is consistent with Example 4.1 in section 4: permanently pruning visited states can block the only path to a solution in domains without genuine dead ends, such as Blocks and Termes, where all `unsolved-incomplete` results under the global closed list are false reports (of unsolvability) rather than genuine unsolvability.

Fixed choice. We fix `gc=1` (global closed list) as the primary configuration, prioritising coverage and efficiency. If minimising `unsolved-incomplete` results or average plan cost is the primary objective, `gc=0` is the better choice; the two objectives are in genuine conflict and no single closed list setting dominates on both.

7.2 Lazy vs. Non-Lazy Heuristic Evaluation

Table 2 compares non-lazy (`l=0`) and lazy (`l=1`) heuristic evaluation with `gc=1` fixed. Both variants (gc=0/1) achieve their best sub-configuration with `l=1` (lazy evaluation).

Metric	Non-lazy	Lazy
Cost (sum)	**9,074,728**	11,294,933
Coverage	1,031	**1,279**
Unsolved-inc.	**566**	574
Evals (geo mean)	4,595	**779**
Expansions (geo mean)	1,148	**605**
Generated (geo mean)	12,461	**1,591**
Total time (geo mean)	0.54 s	**0.19 s**

Table 2: Non-lazy vs. lazy heuristic evaluation with `gc=1` fixed.

Lazy evaluation dominates on every metric except `unsolved-incomplete` and average cost (sum). Coverage is substantially higher, and computational overhead is reduced by

roughly 6× in evaluations and 2× in expansions. The total time improvement from 0.54 s to 0.19 s confirms that avoiding unnecessary heuristic computations translates directly into wall-clock performance.

The negligible `unsolved-incomplete` difference reflects a tie-breaking effect: non-lazy evaluation attaches heuristic values to successors at generation time rather than at pop time, which occasionally produces a different expansion order within a phase that happens to find an improvement. This effect is instance-specific and not reliably reproducible across the benchmark set, hence the near-tie.

Fixed choice. We fix `l=1` (lazy evaluation), due to the clear domination of the lazy variant. The non-lazy variant can also be fixed here if always finding the least average cost (sum) is a priority. Running configuration: `l=1, gc=1`.

7.3 Preferred Operators vs. Helpful Actions

The preferred operators and helpful actions results are analysed first with the ranking setting and then with pruning to help understand the difference between these search guidance methods.

7.3.1 With `RANK_PREFERRED_FIRST`

Table 3 compares preferred operators (`ha=0`) and helpful actions (`ha=1`) with `l=1, gc=1` fixed.

Metric	Pref. ops	Helpful actions
Cost (sum)	8,958,776	**8,958,464**
Coverage	**1,279**	1,264
Unsolved-inc.	**474**	479
Evals (geo mean)	713	**698**
Expansions (geo mean)	554	**542**
Generated (geo mean)	1,442	**1,409**
Total time (geo mean)	**0.17 s**	0.19 s

Table 3: Preferred operators vs. helpful actions under `RANK_PREFERRED_FIRST`, with `l=1,gc=1` fixed.

In this mode, all applicable successors are inserted into the open list regardless of the action set used; the only difference is which successors receive higher priority in the open list.

Preferred operators achieve higher coverage and fewer `unsolved-incomplete` results, at the cost of slightly more evaluations, expansions, and generated states. The lower evaluation

count under helpful actions does not reflect more efficient search but is instead a consequence of earlier termination. Because helpful actions promote a broader set of successors than preferred operators, EHC is occasionally guided toward successors that do not lead to an improvement. This is also a direct consequence of the increased number of `unsolved-incomplete` problems while using helpful actions.

Fixed choice. Both options are viable depending on the objective: `ha=0` (preferred operators) for maximum coverage; `ha=1` (helpful actions) for maximum per-instance efficiency. We fix `ha=0` (preferred operators) for maximum coverage.

7.3.2 With `PRUNE_BY_PREFERRED`

The original FF paper uses helpful actions to prune (Hoffmann and Nebel, 2001), corresponding to the setting `PRUNE_BY_PREFERRED` in FD. To isolate the effect of this rank/prune setting, we additionally ran all of the configurations with `PRUNE_BY_PREFERRED`, with all other parameters held fixed at `l=1, gc=1`.

Metric	Pref. ops	Helpful actions
Cost (sum)	8,952,213	**8,951,933**
Coverage	**1,279**	1,266
Unsolved-inc.	482	**479**
Evals (geo mean)	716	**693**
Expansions (geo mean)	557	**538**
Generated (geo mean)	1,449	**1,396**
Total time (geo mean)	0.17 s	0.17 s

Table 4: Preferred operators vs. helpful actions under `PRUNE_BY_PREFERRED`, with `l=1,gc=1` fixed.

There are two crucial findings that are visible in comparing pruning vs. ranking. First, the one substantial difference is in the slightly higher `unsolved-incomplete` numbers while pruning. This is due to the last source of incompleteness mentioned in section 3.1, where pruning may prune away branches to the goal prematurely and therefore cause EHC to not find a solution. Since coverage is almost identical in both preferred operators and helpful actions (pruning vs. ranking), the slight difference in `unsolved-incomplete` is not a case where ranking succeeds and pruning fails; rather, because ranking retains every successor instead of discarding non-preferred ones, EHC performs substantially more work per phase before concluding failure. On instances where pruning would exhaust its restricted frontier and report `unsolved-incomplete` quickly, ranking instead continues searching until it hits

the search's time or memory bound. This indicates that the pruning/ranking setting affects *how* EHC fails on already-unsolved instances without necessarily changing *whether* it fails on them.

Second, comparing Tables 3 and 4 shows that pruning results in slightly better values in all the metrics than ranking when used in combination with helpful actions, and slightly worse values in all the metrics when pruning is used with preferred operators. This observation that pruning improves metrics under helpful actions but worsens them under preferred operators follows from the relative sizes of the two action sets. Since preferred operators are a strict subset of helpful actions, pruning with preferred operators is more aggressive: it discards a larger fraction of successors per expansion. This over-restriction means EHC more frequently exhausts its pruned frontier without finding an improvement, producing slightly more evaluations and expansions as Breadth-First Search repeatedly searches a restricted space before failing, and more `unsolved-incomplete` outcomes as some solution paths are permanently cut off. Helpful actions, being the broader set, prune less aggressively when used for pruning: enough successors are retained that Breadth-First Search finds improvements reliably while still reducing unnecessary work compared to ranking, which explains the marginal gains in evaluations, expansions, and even coverage over ranking mode.

Fixed choice. Pruning is better when it comes to efficiency and reducing the branching factor. Pruning also works more efficiently with helpful actions; therefore, we fix `ha=1` (helpful actions).

7.4 Dead-End Pruning

Table 5 compares dead-end pruning enabled (`de=1`) and disabled (`de=0`) with `l=1,gc=1` fixed. We do not fix any specific `ha` value because it mostly depends on the preference of the user and the purpose of the experiment (`ha=0/1` and prune/rank).

Dead-end pruning has no measurable aggregate effect under `gc=1`. Every metric is either identical or differs only in the final three digits. This is expected since the global closed list already guarantees that every state is expanded at most once. The separate `dead_end_list` maintained by `de=1` is therefore redundant under `gc=1`, any state it would early-reject is already rejected by the closed list check that follows immediately after. However, dead-end pruning is most beneficial when combined with local closed lists. This is because dead-end pruning helps prune away dead-end states that would otherwise be reconsidered after clearing

Metric	No pruning	Pruning
Cost (sum)	8,958,529	**8,958,464**
Coverage	1,279	1,279
Unsolved-inc.	574	574
Evals (geo mean)	698	698
Expansions (geo)	542	542
Generated (geo)	**1,408**	1,409
Total time (geo)	0.17 s	0.17 s

Table 5: Dead-end pruning enabled vs. disabled with `l=1,gc=1` fixed.

the local closed list.

Fixed choice. We fix `de=0` as default for `gc=1` configurations as it does not make a big difference. For `gc=0`, `de=1` is preferable as it provides minor robustness and cost (sum) improvement.

7.5 Best and Worst Overall Configurations

Table 6 summarises the key configurations resulting from the greedy tuning procedure, together with the original FF and FD reference points.

Configuration	Description	Coverage	Time	Evals	Unsolved-inc.
l=1, gc=1, ha=0, de=0	Fast Downward	1,279	0.17 s	713-716	474-482
l=1, gc=1, ha=1, de=1	Hybrid	1,264-1,266	0.17 s-0.19 s	693-698	479
l=0, gc=0, ha=1, de=0/1	Fast Forward	998–999	1.28–1.29 s	9,400–9,418	479
l=0, gc=0, ha=0, de=0	Worst Overall	1,002	1.07 s	9,673	481

Table 6: Best to worst overall configurations across all satisficing benchmark domains.

Best overall: `l=1,gc=1,ha=0,de=0`. This is the Fast Downward style configuration. It achieves the highest coverage, the fastest runtime, and the second-lowest evaluations. Its only relative weakness is the highest `unsolved-incomplete` count (482), a direct consequence of the global closed list permanently blocking states in some solvable instances, as described in Example 4.1 and section 4. The result confirms that the design choices in Fast Downward are well-calibrated for the benchmark set used here.

Best for completeness: `l=1,gc=1,ha=1,de=1`. This hybrid trades slightly less coverage for the lowest `unsolved-incomplete` count among all `gc=1` configurations. The reason for the lower coverage count could be multiple reasons, such as time and memory outs. Helpful-action

pruning avoids some cases where preferred-operator ranking leads EHC down an unhelpful branch, and the dead-end cache provides a marginal additional guard. This configuration is preferred when maximising the fraction of solvable instances that are actually solved matters more than raw coverage.

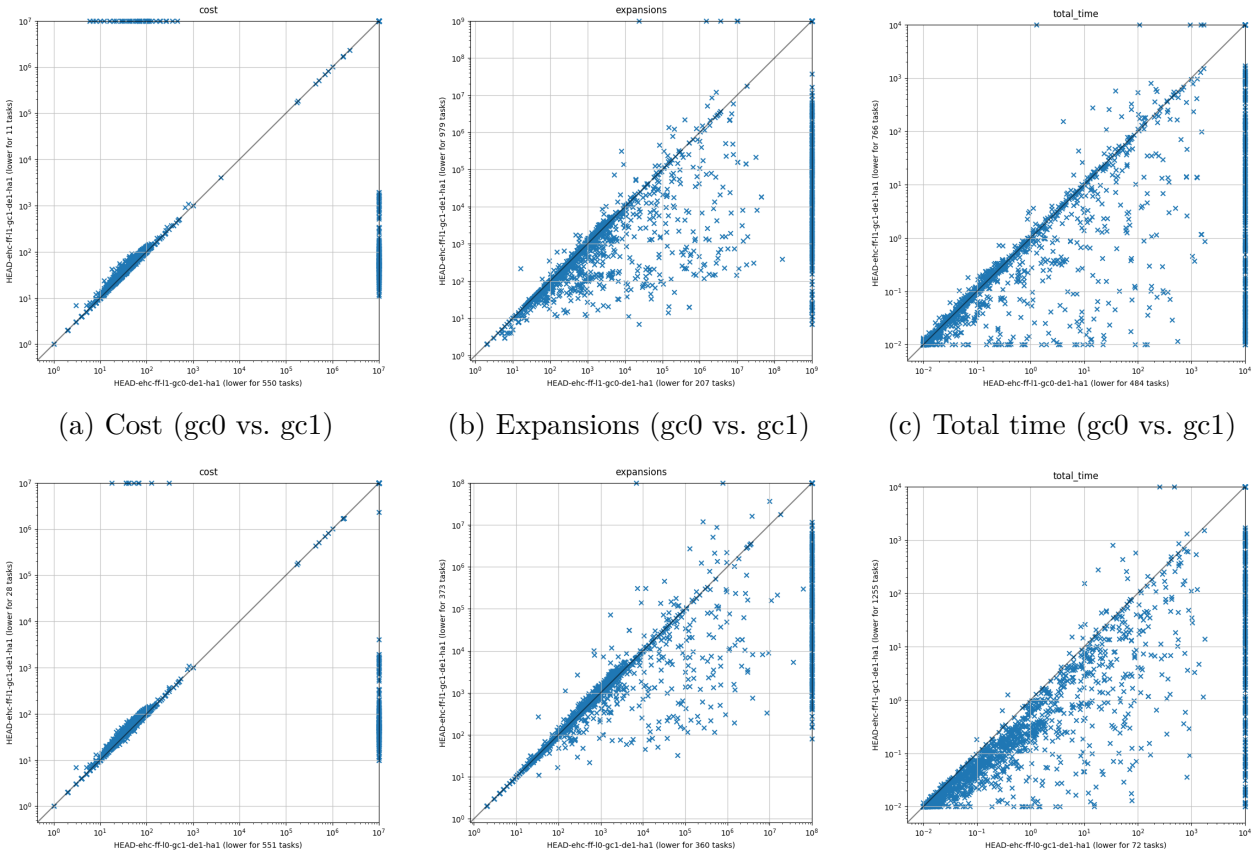
Fast Forward style: `l=0,gc=0,ha=1,de=0/1`. The closest configuration to the original FF planner also achieves 479 `unsolved-incomplete` results but at far higher computational cost compared to the efficiency results for the hybrid. The reason for the lower coverage relative to the hybrid (despite the same completeness count) could be that the repeated re-expansion of previously visited states across phases burns through the search's time/memory budget, so unsolved instances are not converted into clean `unsolved-incomplete` verdicts as often, which results in these time/memory outs.

Worst overall: `l=0,gc=0,ha=0,de=0`. This combination is strictly dominated on almost every metric. It achieves neither the completeness benefits of helpful-action pruning nor the efficiency of lazy evaluation or the global closed list. Its preferred-operator ranking without pruning combines the open list flooding of `gc=0` with the unnecessary heuristic-computation overhead of `l=0`.

No universally complete configuration exists. The minimum `unsolved-incomplete` count observed across all configurations is 479, achieved by both the FF-style and hybrid configurations. No configuration reduces this to zero or can reduce this because EHC is a known incomplete algorithm. Using local closed lists helps reduce this count (recovers paths the global list blocks).

Comparing Scatter-Plots. Figure 2 shows scatter plots comparing pairs of configurations across all benchmark tasks for cost, expansions, and total time. Each point represents one task; points above the diagonal indicate the y-axis configuration performs worse on that task, and points below indicate the x-axis configuration performs worse. Points on or near the diagonal indicate the two configurations perform similarly.

The cost plots confirm the findings in Tables 1 and 2: the local closed list (`gc=0`) achieves lower plan cost than the global variant on 550 tasks compared to only 11 where `gc=1` wins, and non-lazy evaluation (`l=0`) similarly achieves lower cost on 551 tasks versus 28 for lazy. In both cost plots the majority of points lie above the diagonal and cluster tightly near it



(a) Cost (gc0 vs. gc1) (b) Expansions (gc0 vs. gc1) (c) Total time (gc0 vs. gc1)

(d) Cost (l0 vs. l1) (e) Expansions (l0 vs. l1) (f) Total time (l0 vs. l1)

Figure 2: Scatter plots comparing `cost`, `expansions` and `total time` over all the domains in the satisficing suite. Each point represents one planning instance.

for small cost values, indicating that when the two configurations produce similar-cost plans they agree closely, but the divergence grows for larger, more complex instances where the differences in search trajectory become more pronounced.

The expansions plots tell a more nuanced story. For the closed list comparison, points are heavily scattered below the diagonal with the global variant winning on 979 tasks against 207 for local, which is consistent with the geometric mean reduction reported in Table 1. The wide scatter reflects that for some tasks the local list performs similarly to the global, while for others it expands more states. For the lazy vs. non-lazy expansions comparison, points are nearly evenly split (373 vs 360 tasks), with most points under the diagonal. This matches the results from Table 2 where the lazy variant dominates in the number of expansions.

The total time plots show the clearest separation. The lazy vs. non-lazy comparison is the most decisive of all six plots: lazy evaluation wins on 1255 tasks against only 72 for non-lazy, with nearly all points lying below the diagonal. This confirms that the overhead of computing heuristic values at generation time in non-lazy evaluation translates directly and consistently into slower wall-clock performance across almost every instance in the bench-

mark set. The closed list time comparison shows a more moderate advantage for the global variant (766 vs 484 tasks), reflecting that while the global list is faster on average, the local list occasionally benefits from shorter individual phases when re-expansion happens to lead quickly to an improving state, which is also why most of the points lie below the diagonal, favoring the global closed list. Plots not shown had all data points approximately on the diagonal, indicating negligible per-instance differences consistent with the ties reported in Table 5.

8 Conclusion and Outlook

This work investigated the key implementation differences between the Fast Forward and Fast Downward variants of Enforced Hill Climbing, specifically the choice of closed list (local vs. global), the timing of heuristic evaluation (lazy vs. non-lazy), the type of search guidance used (preferred operators vs. helpful actions), and the optional dead-end pruning mechanism. A configurable EHC system supporting all combinations of these parameters was used in order to evaluate the differences across several satisficing planning benchmark domains.

The most significant finding concerns the global closed list and its effect on completeness. As demonstrated by Example 4.1 in section 4, the global closed list can cause FD to report no solution on genuinely solvable instances when the only path to the goal passes through a previously visited state. This is not a theoretical edge case but a measurable phenomenon in practice, accounting for a substantial share of the unsolved-incomplete outcomes observed in the Blocks domain. The local closed list avoids this at the cost of significantly higher computational overhead, as states and their heuristic values are re-evaluated across EHC phases. Lazy heuristic evaluation was one of the most impactful efficiency improvements, reducing evaluations with negligible effect on completeness. Preferred operators in `RANK_PREFERRED_FIRST` mode outperformed helpful-action pruning in `PRUNE_BY_PREFERRED` mode in unsolved-incomplete instances, confirming that aggressive pruning introduces its own source of incompleteness. Dead-end pruning had no measurable effect under the global closed list, since the closed list already subsumes its function, but is beneficial when combined with a local closed list. The overall best configuration by coverage and runtime is the Fast Downward default (l=1, gc=1, ha=0, de=0), while the closest equivalent to Fast Forward (l=0, gc=0, ha=1, de=0/1) achieves fewer unsolved-incomplete outcomes but at a substantially higher computational cost and lower coverage.

Several directions remain open for future investigation. First, the plan quality difference between the local and global closed list variants deserves closer study. As visible in Example 4.1, the two configurations not only differ in whether a solution is found, but also in which solution is returned, the global closed list takes the path (a, e) and fails, while the local list finds (a, e, d, f, g). Extending the implementation to print and compare the actual solution sequences for both configurations across all benchmark instances would make this difference concrete and measurable, and could reveal systematic biases toward shorter or longer plans under each variant. Second, this work only considers the FF heuristic; evaluating whether the same implementation trade-offs hold under alternative heuristics such as h^{add} or landmarks would test the generality of the findings. Lastly, the relationship between domain structure, particularly reversibility of actions, and the relative performance of the configurations studied here is only qualitatively understood; a more systematic domain-feature analysis could predict which configuration to prefer without running all variants.

9 Acknowledgments

I would like to express my sincerest gratitude to Dr. Clemens Büchner for his continuous guidance, support, and valuable feedback throughout the development of this thesis. His insights and feedback were critical in shaping both the direction and quality of this work.

I am also deeply grateful for Prof. Dr. Malte Helmert for providing the opportunity to work on this topic. Furthermore, I would like to thank all members of the research group and department for their support and for providing me with the necessary resources.

References

J.S. Appleby, D.V. Blake, and E.A. Newman. Techniques for producing school timetables on a computer and their application to other scheduling problems. *The Computer Journal*, 3 (4):237–245, 1961.

Avrim L. Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1-2):281–300, 1997.

Blai Bonet and Héctor Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1-2): 5–33, 2001.

Augusto B. Corrêa, Florian Pommerening, Malte Helmert, and Guillem Frances. The FF heuristic for lifted classical planning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 9716–9723, 2022.

Richard E. Fikes and Nils J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4):189–208, 1971.

Malte Helmert. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

Edward F. Moore. The shortest path through a maze. In *Proc. of the International Symposium on the Theory of Switching*, pages 285–292. Harvard University Press, 1959.

Stuart Russell and Peter Norvig. A modern approach. *Artificial Intelligence. Prentice-Hall, Englewood Cliffs*, 25(27):79–80, 1995.

Arthur L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229, 1959.