On Hash-Based Work Distribution Methods for Parallel Best-First Search

Yuu Jinnai

Thesis Advisor: Alex Fukunaga

Graduate School of Arts and Sciences

The University of Tokyo

Tokyo, Japan

Abstract

Parallel best-first search algorithms such as Hash Distributed A* (HDA*) distribute work among the processes using a global hash function. We analyze the search and communication overheads of state-of-the-art hash-based parallel best-first search algorithms, and show that although Zobrist hashing, the standard hash function used by HDA*, achieves good load balance for many domains, it incurs significant communication overhead since almost all generated nodes are transferred to a different processor than their parents. We propose Abstract Zobrist hashing, a new work distribution method for parallel search which, instead of computing a hash value based on the raw features of a state, uses a feature projection function to generate a set of abstract features which results in a higher locality, resulting in reduced communications overhead. We show that Abstract Zobrist hashing outperforms previous methods on search domains using hand-coded, domain specific feature projection functions. We then propose GRAZHDA*, a graph-partitioning based approach to automatically generating feature projection functions. GRAZHDA* seeks to approximate the partitioning of the actual search space graph by partitioning the domain transition graph, an abstraction of the state space graph. We show that GRAZHDA* outperforms previous methods on domain-independent planning.

Acknowledgements

I would like to express my sincere gratitude to my advisor Dr. Alex Fukunaga for the continuous support of my M.S. and related research, for his patience, motivation, and knowledge. He has been supportive since I began working on computer science as an undergraduate. Despite I was from a different field and know almost nothing about computer science, he patiently guided me of my study and research.

Besides my advisor, I am grateful to my fellow labmates for stimulating my scientific curiosity in areas which I was otherwise unaware of. In particular, I would like to thank Asai Masataro, Satoru Horie, Yasutaka Tanaka, and Shunji Lin.

Last I would like to thank my family for supporting my decision to go to Graduate school and my life in general, despite all the difficulties.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The A* algorithm (Hart, Nilsson, & Raphael, 1968b) is used in many areas of AI, including planning, scheduling, path-finding, and sequence alignment. Parallelization of A* can yield speedups as well as a way to overcome memory limitations – the aggregate memory available in a cluster can allow problems that can't be solved using a single machine to be solved. Thus, designing scalable, parallel search algorithms is an important goal. The major issues which need to be addressed when designing parallel search algorithms are search overhead (states which are unnecessarily generated by parallel search but not by sequential search), communications overhead (overheads associated with moving work among threads), and coordination overhead (synchronization overhead).

Hash Distributed A* (HDA*) is a parallel best-first search algorithm in which each processor executes A* using local OPEN/CLOSED lists, and generated nodes are assigned (sent) to processors according to a global hash function (Kishimoto, Fukunaga, & Botea, 2013). HDA* can be used in distributed memory systems as well as multi-core, shared memory machines, and has been shown to scale up to hundreds of cores with little search overhead.

The performance of HDA* depends on the hash function used for assigning nodes to processors. Kishimoto et al (2009, 2013) showed that using the Zobrist hash function (1970), HDA* could achieve good load balance and low search overhead. Burns et al

(2010) noted that Zobrist hashing incurs a heavy communication overhead because many nodes are assigned to processes that are different from their parents, and proposed AHDA*, which used an abstraction-based hash function originally designed for use with PSDD (Zhou & Hansen, 2007) and PBNF (Burns et al., 2010). Abstraction-based work distribution achieves low communication overhead, but at the cost of high search overhead.

In this thesis, we investigate node distribution methods for HDA*. We start by reviewing previous approaches to work distribution in parallel best-first search, including the HDA* framework (Section 2). Then, in Section 3, we present an in-depth investigation of parallel overheads in state-of-the-art parallel best-first search methods. We begin by investigating *why* search overhead occurs on parallel best-first search by analyzing how node expansion order in HDA* diverges from that of A*. If the expansion order of a parallel search algorithm is strictly the same as A*, there is no search overhead, so divergence in expansion order is a useful indicator for understanding search overhead. We show that although HDA* incurs some search overhead due to load imbalance and startup overhead, HDA* using the Zobrist Hash function incurs significantly less search overhead than other methods. However, while HDA* with Zobrist hashing successfully achieves low search overhead, we show that communication overhead is actually as important as search overhead in determining the overall efficiency for parallel search, and Zobrist hashing results in very high communications overhead, resulting in poor performance on the grid pathfinding problem.

Next, in Section 4, we propose Abstract Zobrist hashing (AZH), which achieves both low search overhead and communication overhead by incorporating the strengths of both Zobrist hashing and abstraction. While the Zobrist hash value of a state is computed by applying an incremental hash function to the set of features of a state, AZH first applies a feature projection function mapping features to abstract features, and the Zobrist hash value of the abstract features (instead of the raw features) is computed. We show that on the 24-puzzle, 15-puzzle, and multiple sequence problem, AZH with hand-crafted, domain-

specific feature projection function significantly outperform previous methods on a multi-core machine with up to 16 cores.

Then, we discuss a domain-independent method to automatically generate an efficient feature projection function for abstract Zobrist hashing framework. We first show that a work distribution can be modeled as graph partitioning (Section 5). However, standard graph partitioning techniques for workload distribution in scientific computation are inapplicable to heuristic search because the state space is only defined implicitly. Then, in Section 6, we propose GRAZHDA*, a new domain-independent method for automatically generating a work distribution function, which, instead of partitioning the actual state space graph (which is impractical), generates an approximation by partitioning a *domain transition graph*. We then discuss what objective function to optimize in GRAZHDA* to achieve a good performance and propose sparsity as an objective function. We experimentally show that GRAZHDA* optimizing *sparsity* objective function outperforms all previous variants of HDA* on domain-independent planning, using experiments run on a 48-core cluster as well as a cloud-based cluster with 128 cores. We conclude the thesis with a summary of our results and directions for future work (Section 7).

Portions of this work has been previously presented in two conference papers (Jinnai & Fukunaga, 2016a, 2016b), corresponding to Section 4, as well as parts of Section 2, and in a journal paper (Jinnai & Fukunaga, 2017b), corresponding to Section 4-6.

Chapter 2

Preliminaries and Background

In this section, we first review A* search and classic planning problem, and define the three major classes of overheads that pose a challenge for parallel search (Section 3). We then survey parallel best-first search algorithms (Section 4) and review the HDA* framework (Section 5). We then review the two previous approaches which have been proposed for the HDA* framework, Zobrist hashing (Section 6) and abstraction (Section 7).

2.1 A* search

Most of the parallel search algorithms presented in this thesis are based on A* search algorithm (Hart, Nilsson, & Raphael, 1968a). Given a weighted directed Graph $G = (V, E, w)$, an initial node s_0, goal nodes $T \subset V$, A* returns a path from the initial node s_0 to one of the goal nodes T. A* keeps two sets of nodes, the OPEN list and the CLOSED list. The OPEN contains the set of nodes that have been generated and yet to be expanded. The CLOSED is the set of *expanded* nodes. A* selects a node to expand from the OPEN with the smallest f-value, an estimation of the cost of a shortest solution path including node n. The f-value of node n is defined as $f(n) = g(n) + h(n)$. The path cost $g(n)$ is the cost of the best known path from the initial node s_0 to the node n. The heuristic value $h(n)$ is an estimation of the cost from n to a goal node. A heuristic function h is an admissible

heuristic if it is a lower bound for the optimal solution costs; that is, $h(s) \leq C*(n)$ for all $n \in V$. For admissible heursitic h, A* returns the minimal cost path.

Algorithm 1: A*

```
1  Initialize OPEN to {s_0}, CLOSED to {∅};
2  f(s_0) ← h(s);
3  while OPEN ≠ ∅ do
4      Remove u from OPEN with minimum f(u);
5      Insert u in CLOSED;
6      if Goal(u) then
7          Return Path(u);
8      else
9          Succ(u) ← Expand(u);
10         for each v ∈ Succ(u) do
11             if v ∈ OPEN then
12                 if g(u) + w(u, v) < g(v) then
13                     parent(v) ← u;
14                     f(v) ← g(u) + w(u, v) + h(v);
15             else if v ∈ CLOSED then
16                 if g(u) + w(u, v) < g(v) then
17                     parent(v) ← u;
18                     f(v) ← g(u) + w(u, v) + h(v);
19                     Remove v from CLOSED;
20                     Insert v into OPEN with f(v);
21             else
22                 parent(v) ← u;
23                 Initialize f(v) ← g(u) + w(u, v) + h(v);
24                 Insert v into OPEN with f(v);
25 Return ∅ (failure, no path exist);
```

2.2 Classical Planning

Classical planning is a framework in which many application problems are modelled, including logistics (Helmert & Lasinger, 2010; Sousa & Tavares, 2013), cell assembly (Asai & Fukunaga, 2014), genome rearrangement (Erdem & Tillier, 2005), and arcade games (Lipovetzky, Ramirez, & Geffner, 2015; Jinnai & Fukunaga, 2017a). A world in classical planning is described in logic (Fikes & Nilsson, 1971). Atomic propositions AP describe

what can be true of false in each state of the world. By applying operations to a state, the state transition to another state where different atomic propositions might be true or false. The goal of a classical planning problem is to find a sequence of operations which leads to goal condition from the initial state. We follow the definition by (Edelkamp & Schroedl, 2010):

Definition 1 *A classical planning problem is a finite-state space problem* $P = (S, A, s_0, T)$ *where* $S \subseteq 2^{AP}$ *is the set of states,* $s_0 \in S$ *is the initial state,* $T \subseteq S$ *is the set of goal states, and* A *is the set of actions (operations) that transform states into states.*

Specifically, in STRIPS formalization, a goal is described as a list of propositions $Goal \subseteq AP$. T is a set of states which all propositions in $Goal$ are true. Actions $a \in A$ have propositional preconditions $pre(a)$, and propositional effects ($add(a)$, $del(a)$), where $pre(a) \subseteq AP$ is the precondition of a, $add(a) \subseteq AP$ is the add list, $del(a) \subseteq AP$ is the delete list. Given a state s with $pre(a) \subseteq s$, then its successor $s' = succ(s, a)$ is defined as $s' = (s\ del(a)) \cup add(a)$. As such, a classical planning problem can be solved by an A* search ($G(V', E', w'), s'_0, T'$); $V' = S$, $e(v_i, v_j) \in E'$ exists if there exists a such that $v_j = succ(v_i, a)$, $s'_0 = s_0$, $T' = T$. We discuss classical planning in detail in Section 2.

2.3 Parallel Overheads

Although an ideal parallel best-first search algorithm would achieve an n-fold speedup on n threads, several overheads can prevent parallel search from achieving linear speedup.

Communication Overhead (CO): [1] Communication overhead refers to the cost of exchanging information between threads. In this thesis we define communication overhead as the ratio of nodes transferred to other threads: $CO := \frac{\text{\# nodes sent to other threads}}{\text{\# nodes generated}}$. CO is detrimental to performance because of delays due to message transfers (e.g., network communications), as well as access to data structures such as message queues. In general, CO

1. In this thesis, CO stands for communication overhead, not coordination overhead.

increases with the number of threads. If nodes are assigned randomly to the threads, CO will be proportional to $1 - \frac{1}{\#thread}$.

Search Overhead (SO): Parallel search usually expands more nodes than sequential A*. In this thesis we define search overhead as $SO := \frac{\text{\# nodes expanded in parallel}}{\text{\#nodes expanded in sequential search}} - 1$. SO can arise due to inefficient load balance (LB), where we define load balance as $LB := \frac{\text{Maximum number of nodes assigned to a thread}}{\text{Average number of nodes assigned to a thread}}$. If load balance is poor, a thread which is assigned more nodes than others will become a bottleneck – other threads spend their time expanding less promising nodes, resulting in search overhead. Search overhead is not only critical to the walltime performance, but also to the space efficiency. Even in distributed memory environment, RAM per core is still an important issue to consider.

Coordination (Synchronization) Overhead: In parallel search, coordination overhead occurs when a thread has to wait in idle for an operation of other threads. Even when a parallel search itself does not require synchronization, coordination overhead can be incurred due to contention for the memory bus (Burns et al., 2010; Kishimoto et al., 2013).

There is a fundamental trade-off between CO and SO. Increasing communication can reduce search overhead at the cost of communication overhead, and vice-versa.

2.4 Parallel Best-First Search Algorithms

The key to achieving a good speedup in parallel best-first search is to minimize communication, search, and coordination overhead. In this section, we survey previous approaches. Figure 2-1 presents a visual classification of these approaches which summarizes the discussion below.

Parallel A* (PA*) (Irani & Shih, 1986) is a straightforward parallelization of A* which uses a single, shared open list (in this thesis, we refer to this algorithm as "PA*", and use "parallel A*" to refer to the family of parallel algorithms based on A*). Since worker processes always expand the best node from the shared open list, this minimizes search overhead by eliminating the burst effects. However, node reexpansions are possible in PA* because (as with most other parallel A* variants including HDA*) PA* does not guarantee

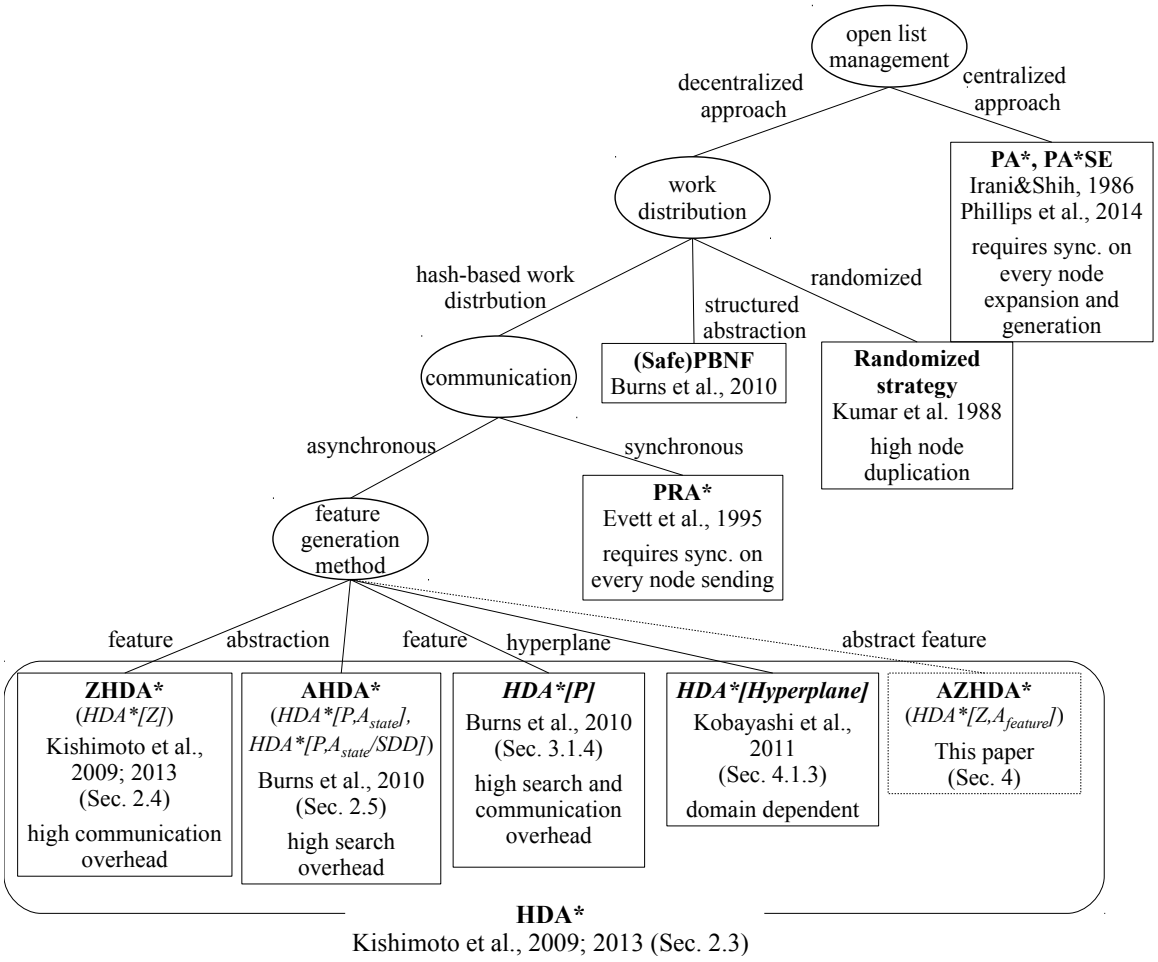


Figure 2-1: Classification of parallel best-first searches.

that a state has an optimal g-value when expanded. Phillips, Likhachev, and Koenig have proposed PA*SE, a mechanism for reducing node reexpansions in PA* (2014) which only expands nodes when their g-values are optimal, ensuring that nodes are not reexpanded.

Kumar, Ramesh, and Rao (1988) identified two classes of approaches to open list management in parallel A*. PA* and its variants are instances of a *centralized approach* which shares a single open list among all processes. However, concurrent access to the shared open list becomes a bottleneck and inherently limits the scalability of this approach unless the cost of expanding each node is extremely expensive, even if lock-free data structures are used (Burns et al., 2010). A *decentralized approach* addresses this bottleneck by assigning each process to a separate open list. Each process executes a best-first search using its own

local open list. While decentralized approaches eliminate coordination overhead incurred by a shared open list, load balancing becomes a problem.

There are several approaches to load balancing in decentralized best-first search. The simplest approach is a randomized strategy which sends generated states to a randomly selected neighbor processes (Kumar et al., 1988). The problem with this strategy is that duplicate nodes are not detected unless they are fortuitously sent to the same process, which can result in a tremendous amount of search overhead due to nodes which are redundantly expanded by multiple processors.

Parallel Retracting A* (PRA*) (Evett, Hendler, Mahanti, & Nau, 1995) uses a *hash-based work distribution* to address simultaneously address both load balancing and duplicate detection. In PRA*, each process owns its local open and closed list. A global hash function maps each state to exactly one process which owns the state. Thus, hash-based work distribution solves the problem of duplicate detection and elimination, because each state has exactly one owner. When generating a state, PRA* distributes it to the corresponding owner synchronously. However, synchronous node sending was shown to degrade performance on domains with fast node expansion, such as grid pathfinding and sliding-tile puzzle (Burns et al., 2010).

Transposition-Table Driven Work Scheduling (TDS) (Romein, Plaat, Bal, & Schaeffer, 1999) is a distributed memory, parallel IDA* with hash-based work distribution. In contrast to PRA*, TDS sends a state to its owner process asynchronously.

An alternate approach for load balancing, which originated in a line work for using multiple processes in external memory search (Korf & Schultze, 2005; ?, ?), is based on *structured abstraction*. Given a state space graph and a projection function, an abstract state graph is (implicitly) generated by projecting states from the original state space graph into abstract nodes. For example, an abstract space for the sliding tile puzzle domain can be created by projecting all nodes with the blank tile at position b to the same abstract state. While the use of abstractions as the basis for heuristic functions has a long history (Pearl, 1984), the use of abstractions as a mechanism for partitioning search states originated in

Structured Duplicate Detection (SDD), an external memory search which stores explored states on disk (Zhou & Hansen, 2004). In SDD, an n-block is defined as the set of all nodes which map to the same abstract node. SDD uses n-blocks to provide a solution to duplicate detection. For any node n which belongs to n-block B, the *duplicate detection scope* of n is defined as the set of n-blocks which can possibly contain duplicates of n, and duplicate checks can be restricted to the duplication detection scope, thereby avoiding the need to look for a duplicate of n outside this scope. SDD exploits this property for external memory search by expanding nodes within a single n-block B at a time and keeping the duplicate detection scope of the nodes in B in RAM, avoiding costly I/O. Unlike stack-slicing, which requires leveled search space, SDD is applicable to any state-space search problem. Parallel Structured Duplicate Detection (PSDD) is a parallel search algorithm which exploits n-blocks to address both synchronization overhead and communication overhead (Zhou & Hansen, 2007). Each processor is exclusively assigned to an n-block and its neighboring n-blocks (which are the duplication detection scopes). By exclusively assigning n-blocks with disjoint duplicate detection scopes to each processor, synchronization during duplicate detection is eliminated. While PSDD used disjoint duplicate detection scopes to parallelize breadth-first heuristic search (Zhou & Hansen, 2006a), Parallel Best-NBlocks First (PBNF) (Burns et al., 2010) extends PSDD to best-first search on multicore machine by ensuring that n-blocks with the best current f-values are assigned to processors. Since livelock is possible in PBNF on domains with infinite state spaces, Burns et al proposed SafePBNF, a livelock-free version of PBNF (2010). Burns et al (2010) also proposed AHDA*, a variant of HDA* which uses an abstraction-based node distribution function. AHDA* is described below in Section 7.

Efficient abstractions can also be generated by exploiting prior knowledge of the structure of the state-space and/or machines on which search is performed. Stack-slicing projects states to their path costs to achieve efficient communication in depth-first search (?), and is useful in domains with *levelled graphs*, where each state can be reached only by a unique path cost, such as model checking (Holzmann & Bošnaĉki, 2007) (thus enabling dupicate

detection). LOcal HAshing of nodes (LOHA) applies path cost-based partitioning in A* search to reduce the number of inter-node communication in a hypercube multiprocessor (Mahapatra & Dutt, 1997).

2.5 Hash Distributed A* (HDA*)

Hash Distributed A* (HDA*) (Kishimoto et al., 2013) is a parallel A* algorithm which incorporates the idea of hash-based work distribution from PRA* (Evett et al., 1995) and asynchronous communication from TDS (Romein et al., 1999). In HDA*, each processor has its own open/closed lists. A global hash function assigns a unique owner thread to every search node. Each thread T repeatedly executes the following:

1. T checks its message queue if any new nodes are in. For all new nodes n in T's message queue, if it is not in the open list (not a duplicate), put n in the open list.

2. Expand node n with the highest priority in the open list. For every generated node c, compute hash value $H(c)$, and send c to the thread that owns $H(c)$.

HDA* has two features which make it attractive as a parallel search algorithm. First, there is little coordination overhead because HDA* communicates asynchronously, and locks for an access to shared open/closed lists are not required because each thread has its own local open/closed list. Second, the work distribution mechanism is simple, requiring only a hash function. However, the effect of the hash function was not evaluated empirically, and the importance of the choice of hash function may not have been fully understood or appreciated – at least one subsequent work which evaluated HDA* used an implementation of HDA* which failed to achieve uniform distribution of the nodes (see Section 2).

2.6 Zobrist Hashing ($HDA^*[Z]$) and Operator-Based Zobrist Hashing ($HDA^*[Z_{operator}]$)

Since the work distribution in HDA* is completely determined by a global hash function, the choice of the hash function is crucial to its performance. ? (?, 2013) noted that it was desirable to use a hash function which uniformly distributed nodes among processors, and used the Zobrist hash function (1970), described below. The Zobrist hash value of a state s, $Z(s)$, is calculated as follows. For simplicity, assume that s is represented as an array of n propositions, $s = (x_0, x_1, ..., x_n)$. Let R be a table containing preinitialized random bit strings (Algorithm 3).

$$Z(s) := R[x_0] \textit{ xor } R[x_1] \textit{ xor } \cdots \textit{ xor } R[x_n] \tag{2.1}$$

Algorithm 2: $HDA^*[Z]$

Input: $s = (x_0, x_1, ..., x_n)$
1 $hash \leftarrow 0$;
2 **for** *each* $x_i \in s$ **do**
3 $\quad hash \leftarrow hash\ xor\ R[x_i]$;
4 **Return** $hash$;

Algorithm 3: Initialize $HDA^*[Z]$

Input: F: a set of features
1 **for** *each* $x \in F$ **do**
2 $\quad R[x] \leftarrow random()$;
3 **Return** R

In the rest of the thesis, we refer to the original version of HDA* by ? (?, 2013), which used Zobrist hashing, as ZHDA* or $HDA^*[Z]$.

Zobrist hashing seeks to distribute nodes uniformly among all processes, without any consideration of the neighborhood structure of the search space graph. As a consequence,

communication overhead is high. Assume an ideal implementation that assigns nodes uniformly among threads. Every generated node is sent to another thread with probability $1 - \frac{1}{\#threads}$. Therefore, with 16 threads, $> 90\%$ of the nodes are sent to other threads, so communication costs are incurred for the vast majority of node generations.

Operator-based Zobrist hashing (OZHDA*) (Jinnai & Fukunaga, 2016b) partially addresses this problem by manipulating the random bit strings in the randomized bitstring table R such that for some selected states S, there are some operators $A(s)$ for $s \in S$ such that the successors of s which are generated when $a \in A(s)$ is applied to s are guaranteed to have the same Zobrist hash value as s, forcing them to be assigned the same processor as s. Although Jinnai and Fukunaga showed that OZHDA* reduces communication overhead compared to Zobrist hashing (2016b), it may result in increased search overhead compared to $HDA^*[Z]$(the extent of which is unpredictable).

2.7 Abstraction ($HDA^*[P, A_{state}]$)

In order to minimize communication overhead in HDA*, Burns et al. (2010) proposed AHDA*, which uses *abstraction* based node assignment. The abstraction strategy in AHDA* applies the state space partitioning technique used in PBNF (Burns et al., 2010) and PSDD (Zhou & Hansen, 2007), which projects nodes in the state space to *abstract states*. After mapping states to abstract states, the AHDA* implementation by Burns et al. (2010) assigns abstract states to processors using a perfect hashing and a modulus operator.

Thus, nodes that are projected to the same abstract state are assigned to the same thread. If the abstraction function is defined so that children of node n are usually in the same abstract state as n, then communication overhead is minimized. The drawback of this method is that it focuses solely on minimizing communication overhead, and there is no mechanism for equalizing load balance, which can lead to high search overhead.

HDA* with abstraction can be characterized by two parameters to decide its behavior – a hashing strategy and an abstraction strategy. The AHDA* implementation by Burns et al. (2010) implemented the hashing strategy using a perfect hashing and a modulus

operator, and an abstraction strategy following the construction for SDD (Zhou & Hansen, 2006b) (for domain-independent planning), or a hand-crafted abstraction (for the sliding tiles puzzle and grid path-finding domains). Note that an abstraction strategy can itself be seen as a type of hashing strategy, but in this thesis, we make the distinction between the method used to project states onto some cluster of states (abstraction) and methods which are used to map states (or abstract states) to processors (hashing).

Jinnai and Fukunaga (2016b) showed that AHDA* with a static N_{max} threshold performed poorly for a benchmark set with varying difficulty because a fixed size abstract graph results in very poor load balance, and implemented Dynamic AHDA* (DAHDA*) which dynamically sets the size of the abstract graph according to the number of features (the state space size is exponential in the number of features). We evaluate DAHDA* in detail in Appendix A.

2.8 Classification of HDA* variants and a Uniform Notation for HDA* variants ($HDA^*[hash, abstraction]$)

At least 12 variants of HDA* have been proposed and evaluated in the previous literature. Each variant of HDA* can be characterized according to two parameters: a hashing strategy used (e.g., Zobrist hashing or perfect hashing), and an abstraction strategy (which corresponds to the strategy used to cluster states or features before the hashing, e.g., state projection based on SDD).

Table 2.1 shows all of the HDA* variants that are discussed in this thesis. In order to be able to clearly distinguish among these variants, we use the notation $HDA^*[hash, abstraction]$ throughout this thesis, where "$hash$" is the hashing strategy of HDA* and "$abstraction$" is the abstraction strategy. Variants that do not use any abstraction strategy are denoted by $HDA^*[hash]$. In cases where the unified notation is lengthy, we use the abbreviated name in the text (e.g., "FAZHDA*" for $HDA^*[Z, A_{feature}/DTG_{fluency}]$).

For example, we denote AHDA* (Burns et al., 2010) using a perfect hashing and a hand-crafted abstraction as $HDA^*[P, A_{state}]$, and AHDA* using a perfect hashing and a SDD abstraction as $HDA^*[P, A_{state}/SDD]$. We denote HDA* with Zobrist hashing without any clustering (i.e., the original version of HDA* by ? ?, 2013) as $HDA^*[Z]$. We denote OZHDA* as $HDA^*[Z_{operator}]$, where $Z_{operator}$ stands for Zobrist hashing using operator-based initialization.

Table 2.1: Overview of all HDA* variants mentioned in this thesis

Algorithms Evaluated With Domain-Specific Solvers Using Domain-Specific, Feature Generation Techniques		
method		First proposed in
$HDA^*[Z]$	ZHDA* : Original version, using Zobrist hashing [Sec 6]	(?)
$HDA^*[P]$	Perfect hashing. [Sec 1.4]	(Burns et al., 2010)
$HDA^*[P, A_{state}]$	AHDA* with perfect hashing and state-based abstraction [Sec 7]	(Burns et al., 2010)
$HDA^*[Z, A_{state}]$	AHDA* with Zobrist hashing and state-based abstraction [Sec 7]	trivial variant of $HDA^*[P, A_{state}]$
$HDA^*[Hyperplane]$	Hyperplane work distribution (Sec 1.3)	(Kobayashi et al., 2011)
$HDA^*[Z, A_{feature}]$	Abstract Zobrist Hashing (feature abstraction) [Sec 4]	(Jinnai & Fukunaga, 2016a)

Automated, Domain-Independent Feature Generation Methods Implemented for Parallelized, Classical Planner		
method		First proposed in
$HDA^*[Z]$	Original version, using Zobrist hashing [Sec 2]	(?)
$HDA^*[Z, A_{state}/SDD]$	AHDA* with Zobrist hashing and SDD-based abstraction [Sec 6]	trivial variant of $HDA^*[P, A_{state}/SDD]$, which was ussed for classical planning in (Burns et al., 2010); uses Zobrist-based hashing instead of perfect hashing.
$HDA^*[Z, A_{state}/SDD_{dynamic}]$	DAHDA*: Dynamic AHDA* [Sec 7 & Append. A]	(Jinnai & Fukunaga, 2016b)
$HDA^*[Z, A_{feature}/DTG_{greedy}]$	GAZHDA*: Greedy Abstract Feature Generation [Sec 2.1]	(Jinnai & Fukunaga, 2016a)
$HDA^*[Z, A_{feature}/DTG_{fluency}]$	FAZHDA*: Fluency-Dependent Abstract Feature Generation [Sec 2.2]	(Jinnai & Fukunaga, 2016b)
$HDA^*[Z_{operator}]$	OZHDA*: Operator-based Zobrist [Sec 6]	(Jinnai & Fukunaga, 2016b)
$HDA^*[Z, A_{feature}/DTG_{sparsity}]$	GRAZHDA*/sparsity: Graph partitioning-based Abstract Feature Generation using the sparsity cut objective [Sec 6]	This thesis

Chapter 3

Analysis of Parallel Overheads in Multicore Best-First Search

As discussed in Section 3, there are three broad classes of parallel overheads in parallel search: search overhead (SO), communications overhead (CO), and coordination (synchronization) overhead. Since state-of-the-art parallel search algorithms such as HDA* and PBNF have successfully eliminated coordination overhead, the remaining overheads are SO and CO. Previous work has focused on evaluating SO quantitatively because SO is fundamental overhead to the algorithm itself whereas CO is due to machine environment which is difficult to evaluate and control. Thus, in this section, we first evaluate the SO of $HDA^*[Z]$ and SafePBNF.

Kishimoto et al. previously analyzed search overhead for $HDA^*[Z]$ (2013). They measured $R_<$, $R_=$, and $R_>$, the fraction of expanded nodes with $f < f^*$, $f = f^*$, and $f > f^*$ (where f^* is optimal cost), respectively. They also measured R_r, the fraction of nodes which were reexpanded. All admissible search algorithms must expand all nodes with $f < f^*$ in order to guarantee optimality. In addition, some of the nodes with $f = f^*$ nodes are expanded. Thus, SO is the sum of $R_>$, R_r, and some fraction of $R_=$. These metrics enable estimating the SO on instances which are too hard to solve in sequential A*. Burns et al. analyzed the quality of nodes expanded by SafePBNF and $HDA^*[P, A_{state}]$

by comparing the number of nodes expanded according to their f values, and showed that $HDA^*[P, A_{state}]$ expands nodes with larger f value (lower quality nodes) compared to SafePBNF (2010).

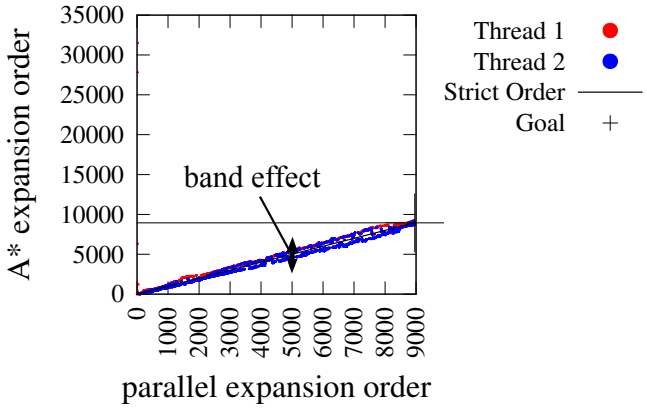
While these previous works measure the amount of search overhead, they do not provide a quantitative explanation for *why* such overheads occur. In addition, previous work has not directly compared $HDA^*[Z]$ and SafePBNF, as Burns et al. (2010) compared SafePBNF to —$HDA^*[P, A_{state}]$ and another variation of HDA* which uses a suboptimal hash function, which we refer to as $HDA^*[P]$ in this thesis.

In this section, we propose a method to analyze SO and explain search overhead in HDA* and SafePBNF. In light of the observation of this analysis, we revisit the comparison of HDA* vs. SafePBNF on sliding-tile puzzle and grid path finding. We then analyze the impact communications overhead has on overall performance.

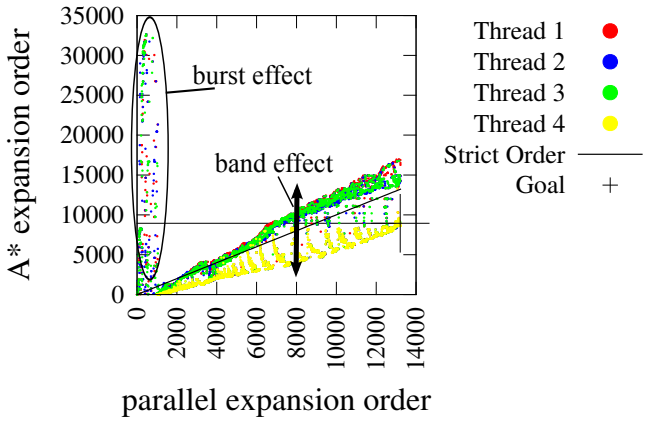
3.1 Search Overhead and the Order of Node Expansion on Combinatorial Search

Consider the global order in which states are expanded by a parallel search algorithm. If a parallel A* algorithm expands states in exactly the same order as A*, then by definition, there is no search overhead. We ran A* and $HDA^*[Z]$ on 100 randomly generated instances of the 15-puzzle on Intel Xeon E5410 2.33 GHz CPU with 16 GB RAM, using a 15-puzzle solver based on the solver code used in the work of Burns et al. (2010). We recorded the order in which states were expanded. We used a random generator by Burns to generate random instances[1]. The results from runs on 2 representative instances (one "easy" instance which A* solves after 8966 expansions, and one "difficult" instance which A* solves after 4265772 expansions), are shown in Figure 3-1, 3-2 and 3-3 (The results on the other difficult/easy problems were similar to these representative instances – aggregate results are presented in Sections 1.3-1.4).

1. The instance generator is at `https://github.com/eaburns/pbnf/tree/master/tile-gen`



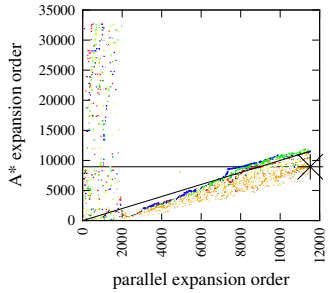
(a) $HDA^*[Z]$ on an easy instance with 2 threads. $HDA^*[Z]$ slightly diverges from A* expansion order with 2 threads (band effect).



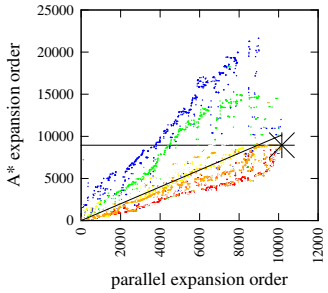
(b) $HDA^*[Z]$ on an easy instance with 4 threads. At the beginning of the search, $HDA^*[Z]$ significantly diverges from A* expansion order, which mostly results in search overhead (burst effect). The band effect is larger with 4 threads than with 2 threads.

Figure 3-1: Illustration of Band Effect: Comparison node expansion order on an **easy instance of the 15-Puzzle**. The vertical axis represents the order in which state s is expanded by parallel search, and the horizontal axis represents the order in which s is expanded by A*. The line $y = x$ corresponds to an ideal, strict A* ordering in which the parallel expansion order is identical to the A* expansion order. The cross marks ("Goal") represents the (optimal) solution, and the vertical line from the goal shows the total number of node expansions by A*. Thus, all nodes above this line result in SO.

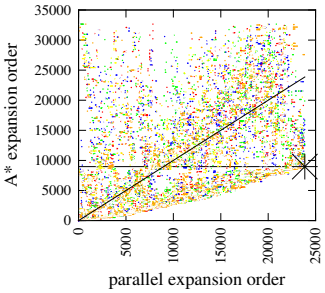
In Figures 3-1, 3-2 and 3-3, the horizontal axis represents the order in which state s is expanded by parallel search (HDA* or SafePBNF). The vertical axis represents the



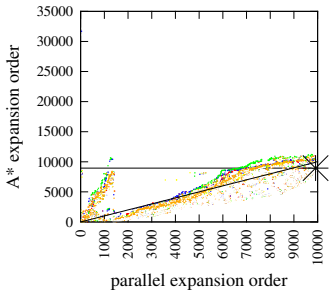
(a) $HDA^*[Z]$ on an easy instance with 8 threads. Both band and burst effects are more significant than with 4 threads.



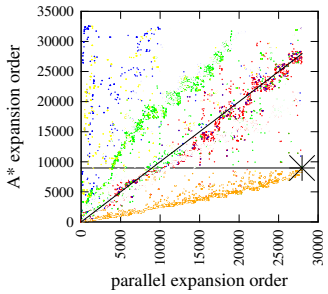
(b) $HDA^*[P, A_{state}]$ on an easy instance with 8 threads. $HDA^*[P, A_{state}]$ has a significantly bigger band compared to $HDA^*[Z]$.



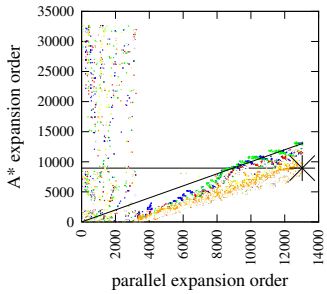
(c) SafePBNF on an easy instance with 8 threads. As threads in SafePBNF requires exclusive access to nblocks, the expansion order differs significantly from A* (and HDA* variants).



(d) $HDA^*[Z]$ on an easy instance with 8 threads with artificially slowed expansion rate. The band effect remains clear, indicating that the band effect is not an accidental overhead cause by communications or lock contention.

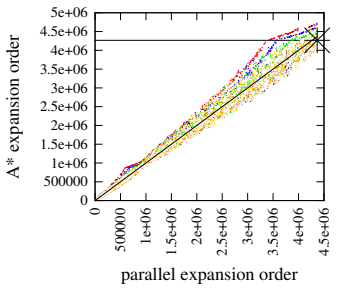


(e) $HDA^*[P]$ on an easy instance with 8 threads. $HDA^*[P]$ has a significantly bigger band compared to other methods and many threads are expanding unpromising (high f value) nodes. As a result, $HDA^*[P]$ expands > 25000 nodes to solve the instance which A* solves with 8966 expansions.

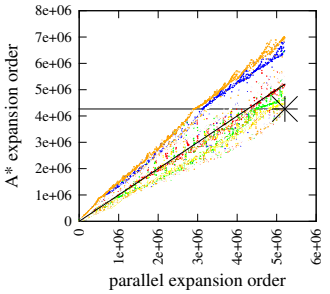


(f) $HDA^*[Z]$ using FIFO tiebreaking on an easy instance with 8 threads (vs. A* using FIFO tiebreaking).

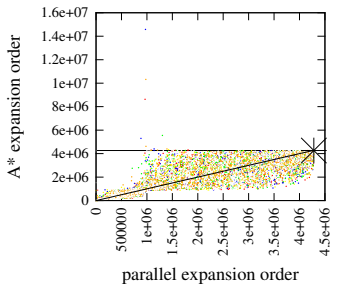
Figure 3-2: Comparison of parallel vs sequential node expansion order on an **easy instance of the 15-Puzzle** with 8 threads.



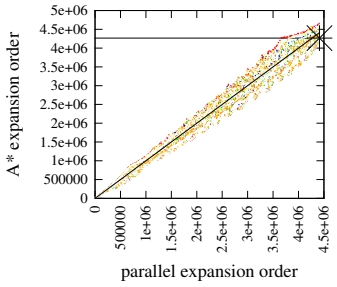
(a) $HDA^*[Z]$ on a difficult instance with 8 threads. As the instance is difficult enough, the relative significance of burst effect becomes negligible.



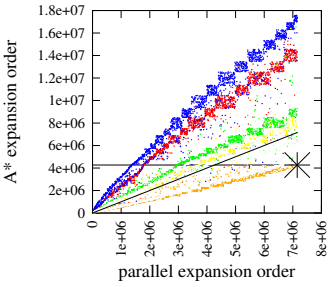
(b) $HDA^*[P, A_{state}]$ on a difficult instance with 8 threads. As with the easy instance, $HDA^*[P, A_{state}]$ has a bigger band than $HDA^*[Z]$ on a difficult instance.



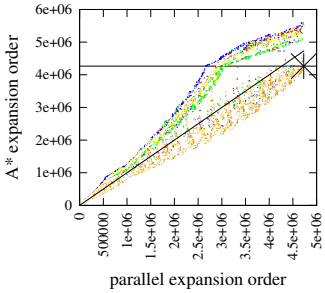
(c) SafePBNF on a difficult instance with 8 threads. Because SafePBNF requires each thread to explore each nblock exclusively, the order of node expansion differs significantly from A*. SafePBNF retains exploring promising nodes by switching nblocks at the cost of communication and coordination overhead.



(d) $HDA^*[Z]$ on a difficult instance with 8 threads with artificially slowed expansion rate. We did not observe a significant difference from $HDA^*[Z]$without slow expansion.



(e) $HDA^*[P]$ on a difficult instance with 8 threads. $HDA^*[P]$ has the biggest band effect, significantly diverged from A*. $HDA^*[P]$ expands $> 7,000,000$ nodes to solve the instance which A* solves with $4,000,000$ expansions.



(f) $HDA^*[Z]$ using FIFO tiebreaking on a difficult instance with 8 threads (vs. A* using FIFO tiebreaking).

Figure 3-3: Comparison of node expansion order on a **difficult instance of the 15-Puzzle** with 8 threads. The average node expansion order divergence of scores are $HDA^*[Z]$: $\bar{d} = 10,330.6$, $HDA^*[Z]$ (slowed): $\bar{d} = 8,812.1$, $HDA^*[P, A_{state}]$: $\bar{d} = 245,818$, $HDA^*[P]$: $\bar{d} = 4,469,340$, SafePBNF: $\bar{d} = 140,629.4$.

A expansion order* of state s, which is the order in which sequential A* expands node s. Note that although standard A* would terminate after finding an optimal solution, we modified sequential A* for this set of experiments so that it continues to search even after the optimal solution has been found. This is because parallel search expands nodes that are not expanded by sequential A* (i.e., search overhead), and we want to know for all states expanded by parallel search which are not usually expanded by sequential A*, how much the parallel search has diverged from the behavior of sequential A*.

The line $y = x$ corresponds to an ideal, *strict* A* ordering in which the parallel expansion ordering is identical to the A* expansion order. The cross marks ("Goal") in the figures represents the (optimal) solution found by A*, and the vertical line from the goal shows the total number of node expansions in A*. Thus, all nodes above this line results in SO. Note that unlike sequential A*, parallel A* can not terminate immediately after finding a solution, even if the heuristic is consistent, because when parallel A* finds an optimal solution it is possible that some nodes with $f < f^*$ have not been expanded (because they are assigned to a processor which is different from the processor where the solution was found).

Although the traditional definition of A* (Hart et al., 1968b) specifies that nodes are expanded in order of nondecreasing f-value (i.e., best-first ordering), this is not sufficient to define a canonical node expansion ordering for sequential A* because many nodes can have the same f-value. A tie-breaking policy can be used to impose a unique, canonical expansion ordering for sequential A*. Our sequential A* uses a LIFO tie-breaking policy, which has been shown to result in good performance on the 15-puzzle (Burns, Hatem, Leighton, & Ruml, 2012), as well as domain-independent planning (Asai & Fukunaga, 2016). In addition, all of our HDA* variants, as well as SafePBNF uses LIFO tie-breaking for each local open list. Thus, by "strict A*" order, we mean "the order in which A* with LIFO tie-breaking expands nodes", and in Figures 3-1, 3-2 and 3-3 compare the expansion ordering of this ordering vs. HDA*/SafePBNF with local LIFO tiebreaking.

To verify that the results are not dependent on the particular tie-breaking policy, Figures 3-2f and 3-3f show results where both sequential A* and the parallel algorithms use FIFO tie-breaking. These show that the results are not qualitatively affected by the choice of tie-breaking policy.

By analyzing the results, we observed three causes of search overhead on HDA*, (1) *Band Effect*, the divergence from the A* order due to load imbalance, (2) *Burst Effect*, an initialization overhead, and (3) node reexpansions. Below, we explain and discuss each of these overheads.

3.1.1 Band Effect

The order in which states are expanded by $HDA^*[Z]$ is fairly consistent with sequential A*. However, there is some divergence from the strict A* ordering, within a "band" that is symmetrical around the strict A* ordering line. For example, in Figure 3-1a, we have highlighted a band showing that the (approximately) 5000'th state expanded by HDA* corresponds a strict A* order between 4500-5500 (i.e., a band width of approximately 1000 at this point in the search). The width of the band tends to increase as the number of threads increases (see the bands in Figure 3-1a, 3-1b, 3-2a). Although the width of the band tends to increase as the search progresses, the rate of growth is relatively small. Also, the harder the instance (i.e., the larger the number of nodes expanded by A*), the narrower the band tends to be (Figure 3-3a).

A simple explanation for this band effect is load imbalance. Suppose we use 2 threads, and assume that threads t_1 and t_2 share p and $1-p$ of the nodes with f value $= f_i$ for each f_i. Consider the n'th node expanded by t_1. This should roughly correspond to the $\frac{n}{p}$'th node expanded by sequential A*; at the same time, t_2 should expand the node which roughly corresponds to the $\frac{n}{1-p}$'th node expanded by sequential A*. In this case, the band size is $|\frac{n}{p} - \frac{n}{1-p}|$. Therefore, if $p = 0.5$ (perfect load balance), the band is small, and as p diverges from 0.5, the band size becomes larger.

One possible, alternative interpretation of the band effect is that it is somehow related to or caused by other factors such as communications overhead or lock contention. To test this, we ran $HDA^*[Z]$ on 8 cores where the state expansion code was intentionally slowed down by adding a meaningless but time-consuming computation to each state expansion.[2] If the band effect was caused by communications or lock contention related issues, it should not manifest itself if the node expansion rate is so slow that the relative cost of communications and synchronization is very small. However, as shown in Figure 3-2d and 3-3d, the band effect remains clearly visible even when the node expansion rate is very slow, indicating that the band effect is not an accidental overhead caused by communications or lock contention (similar results were obtained for other instances).

Observation 1 *The band effect on $HDA^*[Z]$ represents load imbalance between threads. The width of the band determines the extent to which superlinear speedup or search overhead (compared to sequential A*) can occur. Furthermore, the band effect is independent of node evaluation rate.*

The expansion order of SafePBNF is shown in Figure 3-2c and 3-3c. Because SafePBNF requires each thread to explore each nblock (and duplicated detection scope) exclusively, the order of node expansion is significantly different from A*. However, SafePBNF tries to explore promising nodes by switching among nblocks to focus on nblocks which contain the most promising nodes. This requires communication and coordination overhead, which increases the walltime by about $<10\%$ of the time on the 15-puzzle (Burns et al., 2010).

3.1.2 Burst Effect

At the beginning of the search, it is possible for the node expansion order of HDA* to deviate significantly from strict A* order due to a temporary "burst effect". Since there is some variation in the amount of time it takes to initialize each individual thread and

2. At the beginning of the search on each thread, we initialize a thread-local, global integer i to 7. On each thread, after each node expansion, we perform the following computation 100,000 times: $j = 11i \bmod 9999943$, and then set $i \leftarrow j$. This is a heavy computation with a small memory footprint and is intended to occupy the thread without causing additional memory accesses.

populate all of the thread open lists with "good" nodes, it is possible that some threads may start out expanding nodes in poor regions of the search space because good nodes have not yet been sent to their open lists from threads that have not yet completed their initialization. For example, suppose that n_1 is a child of the root node n_0, and n_1 has a significantly worse f-value than other descendants of n_0. Sequential A* will not expand n_1 until all nodes with lower f-values have been expanded. However, at the beginning of search, n_1 may be assigned to a thread t_1 whose queue q_1 is empty, in which case t_1 will immediately expand n_1. The children of n_1 may also have f-values which are significantly worse than other descendants of n_0, but if those children of n_1 are in turn assigned to threads with queues that are (near) empty or otherwise populated by other "bad" nodes with poor f-values, then those children will get expanded, and so on. Thus, at the beginning of the search, many such bad nodes will be expanded because all queues are initially empty, bad nodes will continue to be expanded until the queues are filled with "good" nodes. As the search progresses, all queues will be filled with good nodes, and the search order will more closely approximate that of sequential A*.

Furthermore, these burst-overhead nodes tend to be reached through suboptimal paths (because states necessary for better paths are unavailable during the burst phase), and therefore tend to be revisited later via shorter paths, contributing to revisited node overhead.

The burst phenomenon is clearly illustrated in Figure 3-1b and 3-2a, which shows the behavior of $HDA^*[Z]$ with 8 threads on a small 15-puzzle problem (solved by A* in 8966 expansions). The large vertically oriented cluster at the left of the figure shows that states with a strict A* order of over 30,000 are being expanded within the first 2,000 expansions by HDA*. The A* implementation we used expands over 85,248 nodes per second (the node expansion includes overhead for storing node information in the local data structure, thus slower than base implementation by Burns et al.), this burst phenomenon is occurring within the first 0.023 seconds of search.

Figure 3-3a shows that on a harder problem instance which requires $>$ 4,000,000 state expansions by A*, the overall effect of this initial burst overhead is negligible.

Figure 3-2d shows that when the node expansion rate is artificially slowed down, the burst effect is not noticeable even if the number of states expansions necessary to solve the problem with A* is small ($<$ 10,000). This is consistent with our explanation above that the burst effect is caused by brief, staggered initialization of the threads – when state expansions are slow, the staggered start becomes irrelevant.

From the above, we can conclude that the burst effect is only significant when the problem can be solved very quickly ($<$ 0.88 seconds) by A* and the node expansion rate is fast enough that the staggered initialization can cause a measurable effect.

The practical significance of the burst effect depends on the characteristics of the application domain. In puzzle-solving domains, the time scales are usually such that the burst effect is inconsequential. However, in domains such as real-time path planning, the total time available for planning can be just as a fraction of a second, so the burst effect can have a significant effect.

Observation 2 *The burst effect in $HDA^*[Z]$ can dominate search behavior on easy problems, resulting in large search overhead. However, the burst effect is insignificant on harder problems, as well as when node expansion rate is slow.*

The burst effect is less pronounced in SafePBNF compared to $HDA^*[Z]$, because a thread in SafePBNF prohibits other threads from exploring its duplicate detection scope. The nodes shown in Figure 3-2c are actually band effect, which means that it is persistent through the search (Figure 3-3c).

3.1.3 Node Reexpansions

With a consistent heuristic, A* never reexpands a node once it is saved in the closed list, because the first time a node is expanded, we are guaranteed to have reached through a lowest-cost path to that node. However, in parallel best-first search, nodes may need to be reexpanded even if they are in the closed list. For example, in HDA*, each processor selects the best (lowest f-cost) node in its local open list, but the selected node may not have the current globally lowest f-value. As a result, although HDA* tends to find shortest

paths to a node first, the paths may not be lowest-cost paths, and some node n which is expanded by some thread in HDA* may have been reached through a suboptimal path, and must later be reexpanded after it is reached through a lower-cost path.

This is not a significant overhead for unit-cost domains because shorter paths always have smaller cost. In fact, we observed that $HDA^*[Z]$, $HDA^*[P, A_{state}]$ and SafePBNF had low reexpansion rates for on the 15-puzzle. For $HDA^*[Z]$ with 8 threads, the average reexpansion rate R_r was 2.61×10^{-5} for 100 instances.

Node reexpansions are more problematic in non-unit cost domains, because a shorter path does not always mean a smaller cost. (Kobayashi et al., 2011) analyzed node reexpansion on multiple sequence alignment which $HDA^*[Z]$ suffers from high node duplication rate. We discuss node reexpansions by HDA* on the multiple sequence alignment problem in Section 1.3.

3.1.4 The Impact of Work Distribution Method on the Order of Node Expansion

In addition to $HDA^*[Z]$, we investigated the order of node expansion on $HDA^*[P, A_{state}]$, $HDA^*[P]$, and SafePBNF. The abstraction used for $HDA^*[P, A_{state}]$ ignores the positions of all tiles except tiles 1,2, and 3 (we tried (1) ignoring all tiles except tiles 1,2, and 3, (2) ignoring all tiles except tiles 1,2,3, and 4, (3) mapping cells to rows, and (5) mapping cells to the blocks , and chose (1) because it performed the best). $HDA^*[P]$ is an instance of HDA* which is called "HDA*" in the work of Burns et al. (2010). Unlike the original HDA* by ? (?), which uses Zobrist hashing, $HDA^*[P]$ uses a perfect hashing scheme which maps permutations (tile positions) to lexicographic indices (thread IDs) by Korf and Schultze (2005). A perfect hashing scheme computes a unique mapping from permutations (abstract state encoding) to lexicographic indices (thread ID)[3]. While this encoding is effective for its original purpose of efficient representation of states for external-memory

3. The permutation encoding used by HDA*[P] is defined as: $H(s) = c_1k! + c_2(k-1)! + ... + c_k1!$ where the position of tile $p(i)$ is the c_i-th smallest number in the set $\{1, 2, 3, ..., 16\} \setminus \{c_1, c_2, ...c_{i-1}\}$. State s is sent to a process with process id $H(s)\ mod\ n$, where n is the number of processes. Therefore, if $n = 8$ then $H(s) mod n = \{c_{k-2}3! + c_{k-1}2! + c_k1!\}$, thus it only depends on the relative positions of tiles 12, 13, and 14. In addition, processes with odd/even id only send nodes to processes with odd/even id unless the position of 14 changes.

Table 3.1: Comparison of the average divergence ($\bar{d}$) and premature expansions ($\bar{p}$) for the 50 most difficult 15-puzzle instances.

	$\bar{d}$	$\bar{p}$
$HDA^*[Z]$	10,330.6	563,605
SafePBNF	140,629.4	598,759
$HDA^*[P, A_{state}]$	245,818.0	2,595,540
$HDA^*[P]$	4,469,340.0	3,725,942

search, it was *not* designed for the purpose of work distribution. For SafePBNF, we used the configuration used in (Burns et al., 2010).

Figures 3-2 and 3-3 compare the expansion orders of $HDA^*[Z]$, $HDA^*[P, A_{state}]$, $HDA^*[P]$, and SafePBNF. Although some trends are obvious by visual inspection, e.g., the band effect is larger for $HDA^*[P, A_{state}]$ than on $HDA^*[Z]$, a quantitative comparison is useful to gain more insight.

Thus, we calculated the average *divergence* of each algorithm, where divergence of a parallel search algorithm B on a problem instance I is defined as follows: Let $N_{A*}(s)$ be the order in which state s is expanded by A*, and let $N_B(s)$ be order in which s is expanded by B, and let $V(A^*, B)$ be the set of all states expanded by both A* and P. In case s is reexpanded by an algorithm, we use the first expansion order. Then the divergence of B from A* on instance I is $d(I) = \sum_{s \in V(A^*,B)} |N_{A^*}(s) - N_B(s)| \ / \ |V(A^*, B)|$. We computed the average divergence $\bar{d}$ for 50 most difficult instances in the instance set.

In addition to the divergence d, we calculated the average number of *premature expansions* p, which is the number of nodes expanded before all nodes with lower f value than that node are expanded. Unlike the divergence, the number of premature expansions is not significantly influenced by the expansion order within the same f value.

The average divergence and premature expansions for these difficult instances are shown in Table 3.1. These results indicate that the order of node expansion of $HDA^*[Z]$ is the most similar to that of A*. Therefore, $HDA^*[Z]$ is expected to have the least SO. The abstraction-based methods, $HDA^*[P, A_{state}]$ and SafePBNF, have significantly higher divergence than $HDA^*[Z]$, which is not surprising, since by design, these methods do not seek to simulate

A* expansion order. Finally, $HDA^*[P]$ has a huge divergence, and is expected to have very high SO – it is somewhat surprising that a work distribution function can have divergence (and search overhead) which is so much higher than methods that focus entirely on reducing communications overhead such as $HDA^*[P, A_{state}]$. We evaluate the SO and speedup of each method below in Section 2.

3.2 Revisiting HDA* ($HDA^*[Z]$, $HDA^*[P, A_{state}]$, $HDA^*[Z, A_{state}]$, $HDA^*[P]$) vs. SafePBNF for Admissible Search

Previous work compared $HDA^*[P]$, $HDA^*[P, A_{state}]$, and SafePBNF on the 15-puzzle and grid pathfinding problems (Burns et al., 2010). They also compared SafePBNF with $HDA^*[P, A_{state}]$ on domain-independent planning. The overall conclusion of this previous study was that among the algorithms evaluated, SafePBNF performed best for optimal search. We now revisit this evaluation, in light of the results in the previous section, as well as recent improvements to implementation techniques. There are three issues to note regarding the experimental settings used by Burns et al.:

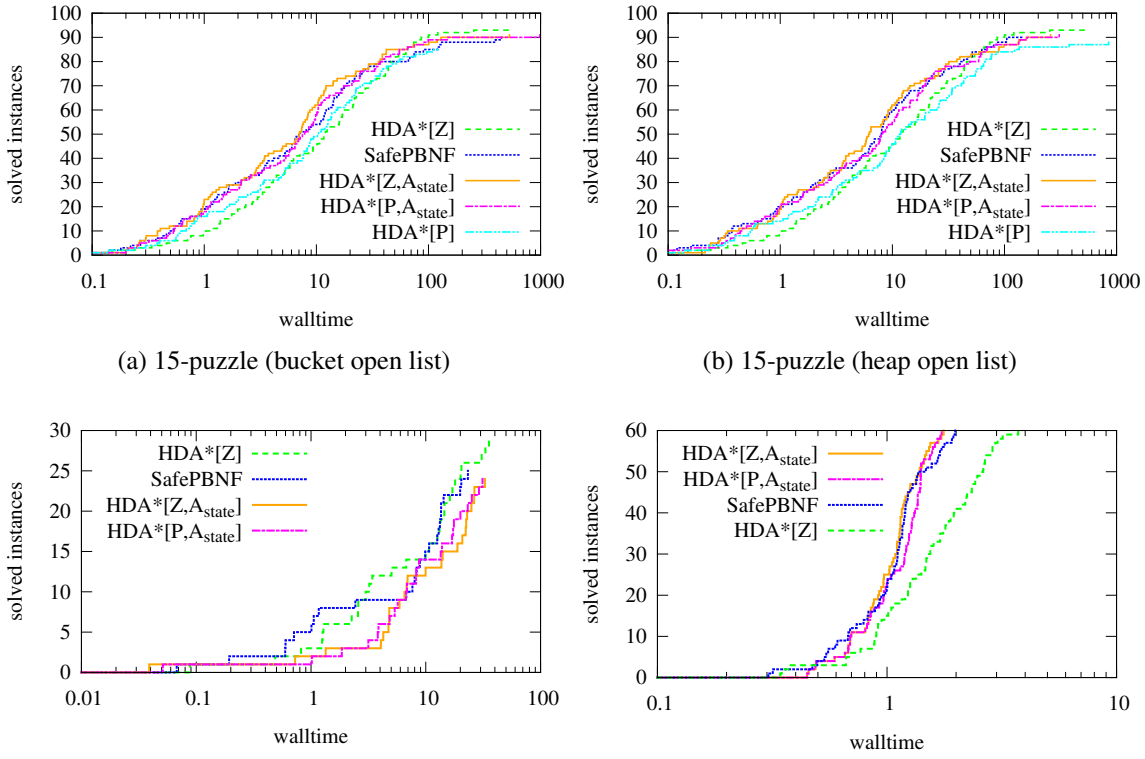
Firstly, *the previous comparison did not include $HDA^*[Z]$, the original HDA* which uses Zobrist hashing* (?; Kishimoto et al., 2013). Burns et al. evaluated two variants of HDA*: $HDA^*[P]$ (which was called "HDA*" in their paper) and $HDA^*[P, A_{state}]$ (called "AHDA*" in their paper). As shown above, the node expansion order of $HDA^*[Z]$ has a much smaller divergence from A* compared to SafePBNF and $HDA^*[P, A_{state}]$. While $HDA^*[Z]$ seeks to minimizes search overhead and both $HDA^*[P, A_{state}]$ as well as SafePBNF seeks to reduce communications overhead, $HDA^*[P]$ minimizes neither communications nor search overheads (as shown above, it has much higher expansion order divergence than all other methods), so $HDA^*[P]$ is not a good representative of the HDA* framework. Therefore, a direct comparison of SafePBNF and $HDA^*[P, A_{state}]$ (which minimize communications overhead) to $HDA^*[Z]$ (which minimizes search overhead) is necessary in order to understand how these opposing objectives affect performance.

Secondly, the 15-puzzle and grid search instances used in the previous study only required a small amount of search, so the behavior of these algorithms on difficult problems has not been compared. In the previous study, the grid domains consisted of 5000x5000 grids, and the 15-puzzle instances were all solvable within 3 million expansions by A*. Since grid pathfinding solvers can generate 10^6 nodes per second, and 15-puzzle solvers can generate 0.5×10^6 nodes per second, these instances are solvable in under a second by a 8-core parallel search algorithm. As shown in section 1, when the search only takes a fraction of a second, HDA* incurs significant search overhead due to the burst effect, but the burst effect is a startup overhead whose impact is negligible on problem instances that require more search.

Thirdly, in the previous study, for all algorithms, a binary heap implementation for the open list priority queue was used, which incurs $O(logN)$ costs for insertion. This introduces a bias for PBNF over all of the HDA* variants. PBNF uses a separate binary heap for each n-block – splitting the open list into many binary heaps greatly decreases the N in the $O(logN)$ cost node insertions compared to algorithms such as HDA* which use a single open list per thread. However, it has been shown that a bucket implementation ($O(1)$ for all operations) results in significantly faster performance on state-of-the-art A* implementations (Burns et al., 2012).

Therefore, we revisit the comparison of HDA* and SafePBNF by (1) using Zobrist hashing for HDA* (i.e., $HDA^*[Z]$) in order to minimize search overhead (2) using both easy instances (solvable in < 1 second) and hard instances (requiring up to 1000 seconds to solve with sequential A*) of the sliding tiles and grid path-finding domains in order to isolate the startup costs associated with the burst effect, and (3) using both bucket and heap implementations of the open list in order to isolate the effect of data structure efficiency (as opposed to search efficiency).

For the 15-puzzle, we used the standard set of 100 instances by Korf (1985), and used the Manhattan Distance heuristic. We used the same configuration used in Section 1.4 for all algorithms (except without the instrumentation to storing the expansion order in-



(a) 15-puzzle (bucket open list)

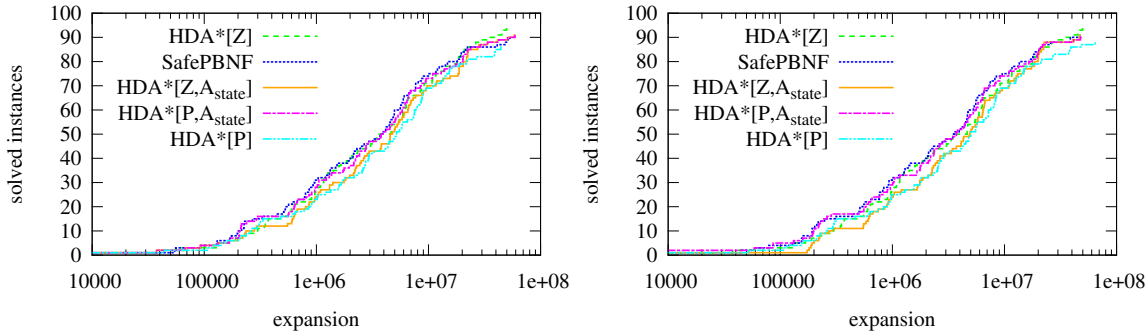
(b) 15-puzzle (heap open list)

(c) 24-puzzle (bucket open list)

(d) Grid Pathfinding (bucket open list)

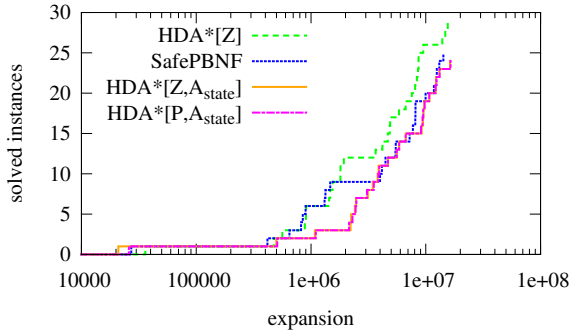
Figure 3-4: Comparison of the number of instances solved within given walltime. The x axis shows the walltime and y axis shows the number of instances solved by the given walltime. In general, $HDA^*[Z]$ outperforms SafePBNF on difficult instances ($>$ 10 seconds) and SafePBNF outperforms $HDA^*[Z]$ on easy instances ($<$ 10 seconds).

formation for each state). For the 24-puzzle, we used 30 instances randomly generated which could be solved within 1000 seconds by sequential A*, and used the pattern database heuristic (Korf & Felner, 2002). The abstraction used by $HDA^*[P, A_{state}]$, $HDA^*[Z, A_{state}]$, and SafePBNF ignores the numbers on all of the tiles except tiles 1,2,3,4, and 5 (we tried (1) ignoring all tiles except tiles 1-5, (2) ignoring all tiles except tiles 1-6, (3) ignoring all tiles except tiles 1-4, (4) mapping cells to rows, and (5) mapping cells to the blocks, and chose (1), the best performer). For (4-way unit-cost) grid path finding, we used 60 instances based obtained by randomly generating 5000x5000 grids where 0.45 of the cells are obstacles. We used Manhattan distance as a heuristic. The abstraction used for $HDA^*[P, A_{state}]$ and $HDA^*[Z, A_{state}]$ maps 100x100 nodes to an abstract node, which performed the best

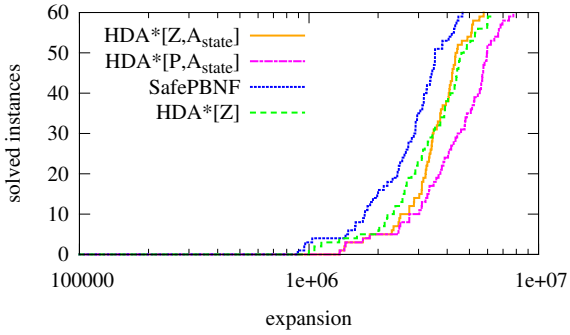


(a) 15-puzzle (bucket open list)

(b) 15-puzzle (heap open list)



(c) 24-puzzle (bucket open list)



(d) Grid Pathfinding (bucket open list)

Figure 3-5: Comparison of the number of instances solved within given number of node expansions. The x axis shows the walltime and y axis shows the number of instances solved by the given node expansion. Overall, $HDA^*[Z]$ has the lowest SO except in grid pathfinding, where $HDA^*[Z]$ suffers from high node duplication because the node expansion is extremely fast in grid pathfinding. $HDA^*[Z, A_{state}]$ and $HDA^*[P, A_{state}]$ expanded almost identical number of nodes in 24-puzzle.

among 5x5, 10x10, 50x50, 100x100, and 500x500 (Section 3). For SafePBNF we used the same configuration used in previous work (Burns et al., 2010). The queue of free nblocks is implemented using binary tree as there were no significant difference in performance using vector implementation.

Figure 3-4 compares the number of instances solved as a function of wall-clock time by $HDA^*[Z]$, $HDA^*[P, A_{state}]$, $HDA^*[P]$, and SafePBNF. The results show that on the 15-puzzle and 24-puzzle, grid pathfinding, PBNF initially outperforms $HDA^*[Z]$, but as more time is consumed, $HDA^*[Z]$ solves more instances than PBNF, i.e., PBNF outperforms

$HDA^*[Z]$ on easier problems due to the burst effect (Section 1.2), while $HDA^*[Z]$ outperforms SafePBNF on more difficult instances because after the initial burst effect subsides, $HDA^*[Z]$ diverges less from A* node expansion order and therefore incurs less search overhead.

Observation 3 *$HDA^*[Z]$ significantly outperforms SafePBNF on 15-puzzle and 24-puzzle instances that require a significant amount of search. On instances that can be solved quickly, SafePBNF outperforms $HDA^*[Z]$ due to the burst effect.*

Comparing the results for the 15-puzzle for the bucket open list implementation (Figure 3-4a) and the heap open list implementation (Figure 3-4b), we observe that all of the HDA* variants benefit from using a bucket open list implementation. Not surprisingly, for the more difficult problems, the benefit of the more efficient data structure ($O(1)$ vs. $O(logN)$ insertion for N states) becomes more significant. PBNF does not benefit as much from the bucket open list because in PBNF, there is a separate queue associated with each n-block, so the difference between bucket and heap implementations is $O(1)$ vs $O(logN/B)$, where B is the number of n-blocks.

Figure 3-5 compares the number of solved instances within the number of node expanded. Due to the burst effect, with small number of expansions, $HDA^*[Z]$ solves fewer instances compared to SafePBNF, especially in grid domain.

3.2.1 On the effect of hashing strategy in AHDA* ($HDA^*[Z, A_{state}]$ vs. $HDA^*[P, A_{state}]$)

In addition to the original implementation of AHDA* (Burns et al., 2010), which distributes abstract states using a perfect hashing ($HDA^*[P, A_{state}]$), we implemented $HDA^*[Z, A_{state}]$ which uses Zobrist hashing to distribute. Interestingly, Figure 3-5 shows that both $HDA^*[Z, A_{state}]$ and $HDA^*[P, A_{state}]$ achieved lower search overhead than $HDA^*[P]$ in 15-puzzle. A possible explanation is that the abstraction is hand-crafted so that the abstract nodes are sized equally and distributed evenly in the search space. On the other hand, as an abstract state is already a large set of nodes, distributing abstract states using Zobrist hashing ($HDA^*[Z, A_{state}]$) does not yield significantly better search overhead compared to $HDA^*[P, A_{state}]$.

3.3 The Effect of Communication Overhead on Speedup

Although $HDA^*[Z]$ is competitive with the abstraction-based methods ($HDA^*[P, A_{state}]$ and SafePBNF) on the sliding tile puzzle domains, Figure 3-4d shows that $HDA^*[P, A_{state}]$ and SafePBNF significantly outperformed $HDA^*[Z]$ in the grid path-finding domain. Interestingly, Figure 3-5d shows that $HDA^*[P, A_{state}]$ and $HDA^*[Z]$ solve roughly the same number problems, given the same number of node expansions. This indicates that the performance difference between $HDA^*[P, A_{state}]$ and $HDA^*[Z]$ on the grid domain is *not* due to search overhead, but rather due to the fact that $HDA^*[P, A_{state}]$ is able to expand nodes faster than $HDA^*[Z]$. In previous work, Burns et al showed that $HDA^*[P]$ suffers from high communications overhead on the grid domain (2010).[4]

Although HDA* uses asynchronous communication, sending/receiving message require access to data structure such as message queues. Communication costs is crucial in grid path finding because the node expansion rate is extremely high in grid path-finding. Fast node expansion means that the relative time to send a node is higher. Our grid solver expands 955,789 node/second, much faster than our 15-puzzle (bucket) solver (565,721 node/second). Thus, the relative cost of communication in grid domain is twice as high as that of 15-puzzle.

To understand the impact of communications overhead, we evaluated the speedup, communications overhead (CO), and search overhead (SO) of $HDA^*[P, A_{state}]$ with different abstraction sizes. The abstraction used for $HDA^*[P, A_{state}]$ maps $k \times k$ blocks in the grid to a single abstract state. Note that in this domain, an abstraction size of 1 corresponds to $HDA^*[P]$. Table 3.2 shows the results. As the size of the $k \times k$ block increases, communications is reduced, and as a result, 100x100 $HDA^*[P, A_{state}]$ is faster than $HDA^*[Z]$ and

4. Burns et al. evaluated HDA* ($HDA^*[P]$) on the grid problem using a perfect hash function *processor(s)* $= (x \cdot y_{max} + y) \mod p$ (p is the number of processes) of the state location for work distribution. This hash function results in different behavior according to the number of processes. If $(y_{max} \mod p) = 0$, then all cells in each row have the same hash value, but all pairs of adjacent rows are guaranteed to have different hash values. If $(y_{max} \mod p) \neq 0$, all pairs adjacent cells are guaranteed to have different hash values. Both conditions result in high communication overhead, thus $HDA^*[P, A_{state}]$ (100x100) significantly outperformed both condition.

$HDA^*[P]$ although it has the same amount of SO. However, there is a point of diminishing returns due to load imbalance – in the extreme case when the entire $N \times N$ grid is mapped to a single abstract state, there would be no communications but only 1 processor would have work. Thus, a 500x500 abstraction results in worse performance than a 100x100 abstraction.

Table 3.2: Comparison of speedup, communication overhead, and search overhead of $HDA^*[P, A_{state}]$ on grid path finding using different abstraction size. CO: communication overhead ($= \frac{\text{\# nodes sent to other threads}}{\text{\# nodes generated}}$), SO: search overhead ($= \frac{\text{\# nodes expanded in parallel}}{\text{\#nodes expanded in sequential search}} - 1$).

abstraction size	speedup	CO	SO
$HDA^*[Z]$	2.61	0.87	0.05
1x1 (= $HDA^*[P]$)	2.57	0.87	0.05
5x5	3.50	0.19	0.05
10x10	3.82	0.10	0.06
50x50	4.16	0.02	0.06
100x100	4.22	0.01	0.05
500x500	3.24	0.01	0.42

Note that while this experiment was run on a a single multicore machine using pthreads and low-level instructions (try_lock) for moving states among processors, communications overhead becomes an even more serious issue using interprocess communication (e.g. MPI) on distributed environment because the communication cost for each message is higher on such environments.

Observation 4 *SafePBNF and $HDA^*[P, A_{state}]$ outperform $HDA^*[Z]$ on the grid pathfinding problem, even though SafePBNF and $HDA^*[P, A_{state}]$ require more node expansions than $HDA^*[Z]$. Communications overhead accounts for the poor performance of $HDA^*[Z]$ on grid pathfinding.*

3.4 Summary of the Parallel Overheads for $HDA^*[Z]$ and $HDA^*[P, A_{state}]$

Table 3.3 summarizes the comparison of the Zobrist hashing based $HDA^*[Z]$ and structured abstraction based $HDA^*[P, A_{state}]$ work distribution strategies on the sliding-tile puzzle and grid pathfinding domains. As we showed in Section 1.4 and 2, $HDA^*[Z]$ outperforms $HDA^*[P, A_{state}]$ on sliding-tile puzzle domain because $HDA^*[P, A_{state}]$ suffers from high SO. On the other hand, $HDA^*[P, A_{state}]$ outperforms $HDA^*[Z]$ on grid pathfinding because $HDA^*[Z]$ has high CO (Section 3). In summary, both $HDA^*[Z]$ and $HDA^*[P, A_{state}]$ have clear weakness – $HDA^*[Z]$ has no mechanism which explicitly seeks to reduce the amount of communication, whereas $HDA^*[P, A_{state}]$ has no mechanism which explicitly minimizes load balancing.

Table 3.3: Comparison of speedup, communication overhead, and search overhead of $HDA^*[Z]$ and $HDA^*[P, A_{state}]$ on 15-puzzle, 24-puzzle, and grid pathfinding with 8 threads. CO: communication overhead, SO: search overhead. $HDA^*[Z]$ outperformed $HDA^*[P, A_{state}]$ in 15-puzzle and 24-puzzle while $HDA^*[P, A_{state}]$ outperformed $HDA^*[Z]$ in grid pathfinding.

15-puzzle	speedup	CO	SO
$HDA^*[Z]$	**5.10**	0.86	0.03
$HDA^*[P, A_{state}]$	3.90	0.22	0.13
24-puzzle	speedup	CO	SO
$HDA^*[Z]$	**6.28**	0.85	0.04
$HDA^*[P, A_{state}]$	4.20	0.38	0.14
grid	speedup	CO	SO
$HDA^*[Z]$	2.57	0.87	0.05
$HDA^*[P, A_{state}]$	**4.22**	0.01	0.05

Chapter 4

Abstract Zobrist Hashing(AZH)

As we discussed in Section 3, both search and communication overheads have a significant impact on the performance of HDA*, and methods that only address one of these overheads are insufficient. $HDA^*[Z]$, which uses Zobrist hashing, assigns nodes uniformly to processors, achieving near-perfect load balance, but at the cost of incurring communications costs on almost all state generations. On the other hand, abstraction-based methods such as PBNF and $HDA^*[P, A_{state}]$ significantly reduce communications overhead by trying to keep generated states at the same processor as where they were generated, but this results in significant search overhead because all of the productive search may be performed at 1 node, while all other nodes are searching unproductive nodes which would not be expanded by A*. Thus, we need a more balanced approach which simultaneously addresses both search and communication overheads.

Abstract Zobrist hashing (AZH) is a hybrid hashing strategy which augments the Zobrist hashing framework with the idea of projection from abstraction, incorporating the strengths of both methods. The AZH value of a state, $AZ(s)$ is:

$$AZ(s) := R[A(x_0)] \; xor \; R[A(x_1)] \; xor \; \cdots \; xor \; R[A(x_n)] \tag{4.1}$$

where A is a *feature projection function*, a many-to-one mapping from each raw feature to an *abstract feature*, and R is a pre-computed table for each abstract feature.

Thus, AZH is a 2-level, hierarchical hash, where raw features are first projected to abstract features, and Zobrist hashing is applied to the abstract features. In other words, we project state s to an abstract state $s' = (A(x_0), A(x_1), ..., A(x_n))$, and $AZ(s) = Z(s')$. Figure 4-1 illustrates the computation of the AZH value for an 8-puzzle state.

AZH seeks to combine the advantages of both abstraction and Zobrist hashing. Communication overhead is minimized by building abstract features that share the same hash value (abstract features are analogous to how abstraction projects state to abstract states), and load balance is achieved by applying Zobrist hashing to the abstract features of each state.

Compared to Zobrist hashing, AZH incurs less CO due to abstract feature-based hashing. While Zobrist hashing assigns a hash value for each node independently, AZH assigns the same hash value to all nodes which share the same abstract features for all features, reducing the number of node transfers. Also, in contrast to abstraction-based node assignment, which minimizes communications but does not optimize load balance and search overhead, AZH seeks good load balance, because the node assignment considers all features in the state, rather than just a subset.

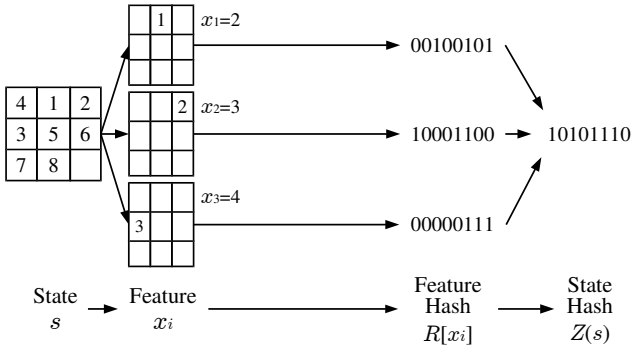
Algorithm 4: Initialize $HDA^*[Z, A_{feature}]$

Input: F: a set of features, A: a mapping from features to abstract features (abstraction strategy)

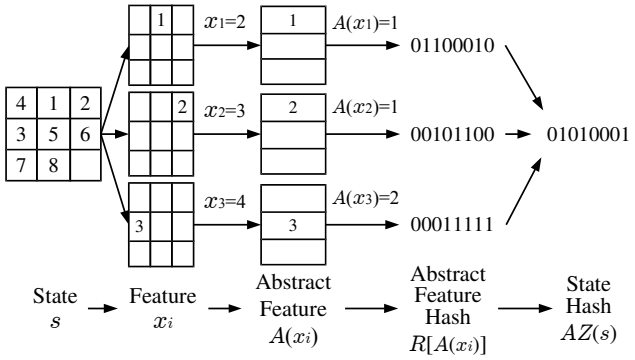
```
for each a ∈ {A(x)|x ∈ F} do
    R'[a] ← random();
for each x ∈ F do
    R[x] ← R'[A(x)];
Return R
```

AZH is simple to implement, requiring only an additional projection per feature compared to Zobrist hashing, and we can pre-compute this projection at initialization (Algorithm 4). Thus, there is no additional runtime overhead per node during the search. In fact,

except for initialization, the same code to Zobrist hashing can be used (Algorithm 2). The projection function $A(x)$ can be generated either hand-crafted or automated. Following the notation of AHDA* in Section 7, we denote AZHDA* with hand crafted feature abstraction as $HDA^*[Z, A_{feature}]$, where $A_{feature}$ stands for feature abstraction. The key difference of $HDA^*[Z, A_{feature}]$ from $HDA^*[Z, A_{state}]$ is that $HDA^*[Z, A_{feature}]$ applies abstraction to each feature and applies Zobrist hashing to abstract features, whereas $HDA^*[Z, A_{state}]$ applies abstraction to a state and applies Zobrist hashing to the abstract state.



(a) Zobrist hashing



(b) Abstract Zobrist hashing

Figure 4-1: Calculation of Abstract Zobrist Hash (AZH) value $AZ(s)$ for the 8-puzzle: State $s = (x_1, x_2, ..., x_8)$, where $x_i = 1, 2, ..., 9$ ($x_i = j$ means tile i is placed at position j). The Zobrist hash value of s is the result of xor'ing a preinitialized random bit vector $R[x_i]$ for each feature (tile) x_i. AZH incorporates an additional step which projects features to abstract features (for each feature x_i, look up $R[A(x_i)]$ instead of $R[x_i]$).

4.1 Evaluation of Work Distribution Methods on Domain-Specific Solvers

We evaluated the performance of the following HDA* variants on several standard benchmark domains with different characteristics.

- $HDA^*[Z, A_{feature}]$: HDA* using AZH
- $HDA^*[Z]$: HDA* using Zobrist hashing (?)
- $HDA^*[P, A_{state}]$: HDA* using Abstraction based work distribution (Burns et al., 2010)
- $HDA^*[P]$: HDA* using a perfect hash function (Burns et al., 2010)

The experiments were run on an Intel Xeon E5-2650 v2 2.60 GHz CPU with 128 GB RAM, using up to 16 cores.

The 15-puzzle experiments in Section 1.1 incorporated enhancements from the more recent work by Burns et al. Burns et al. to the code used in Section 1, which is based on the code by Burns et al. (2010), which includes $HDA^*[P]$, $HDA^*[P, A_{state}]$, and SafePBNF (we implemented 15-puzzle $HDA^*[Z]$ and $HDA^*[Z, A_{feature}]$ as an extension of their code).

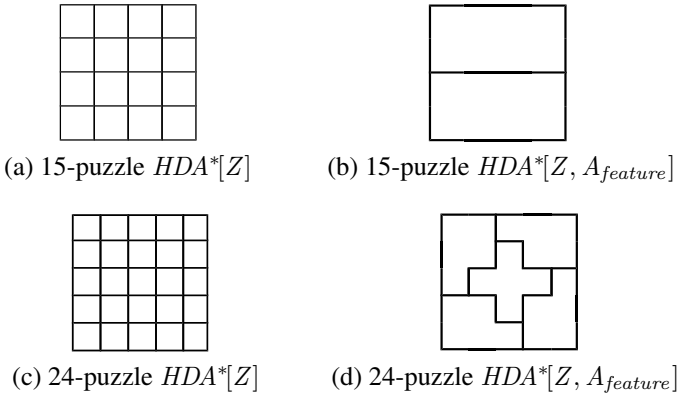
For the 24-puzzle and multiple sequence alignment (MSA), we used our own implementation of HDA* for overall performance (different from the code used in Section 2), using the Pthreads library, try_lock for asynchronous communication, and the Jemalloc memory allocator (Evans, 2006). We implemented the open list as a 2-level bucket (Burns et al., 2012) for the 15-puzzle and 24-puzzle, and a binary heap for MSA (binary heap was faster for MSA).

Note that although we evaluated $HDA^*[Z]$, $HDA^*[P, A_{state}]$, and SafePBNF on the on the grid pathfinding problem in Section 3, we do not evaluate $HDA^*[Z, A_{feature}]$ on the grid pathfinding problem because in the case of grid pathfinding, the obvious feature projection function for $HDA^*[Z, A_{feature}]$ corresponds to the abstraction used by $HDA^*[P, A_{state}]$.

4.1.1 15-Puzzle

We solved 100 randomly generated instances with solvers using the Manhattan distance heuristic. These are not the same instances as the 100 instances used in Section 1 because the solver used for this experiment was faster than the solver used in Section 1[1], and some of the instances used in Section 1 were too easy for an evaluation of parallel efficiency.[2] We selected instances which were sufficiently difficult enough to avoid the results being dominated by the initial startup overhead of the burst effect (Section 1.2) – sequential A* required an average of 52.3 seconds to solve these instances. In addition to $HDA^*[Z, A_{feature}]$, $HDA^*[Z]$, and $HDA^*[P, A_{state}]$, we also evaluated SafePBNF (Burns et al., 2010) and $HDA^*[P]$.

The projections $A(x_i)$ (abstract features) we used for AZH in $HDA^*[Z, A_{feature}]$ are shown in Figure 4-2b. The configurations for the other work distribution methods ($HDA^*[Z]$, $HDA^*[P, A_{state}]$, SafePBNF, and $HDA^*[P]$) were the same as in Section 1.



(a) 15-puzzle $HDA^*[Z]$ (b) 15-puzzle $HDA^*[Z, A_{feature}]$

(c) 24-puzzle $HDA^*[Z]$ (d) 24-puzzle $HDA^*[Z, A_{feature}]$

Figure 4-2: The hand-crafted abstract features used by AZH for 15 and 24-puzzle.

First, as discussed in Section 2, high search overhead is correlated with load balance. Figure 4-3, which shows the relationship between load balance and search overhead, in-

1. In Section 1, the code is based on the code used in the work of Burns et al. (2010), while the code used in this section incorporated all of the enhancements from their more recent work on efficient sliding tile solver code (Burns et al., 2012)
2. This was intentional – in Section 1, we needed a distribution of instances that included easy instances to highlight the burst effect (Section 1.2) as well as for comparison with other methods 2.

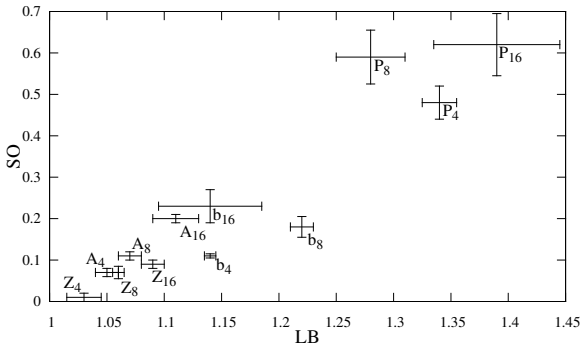


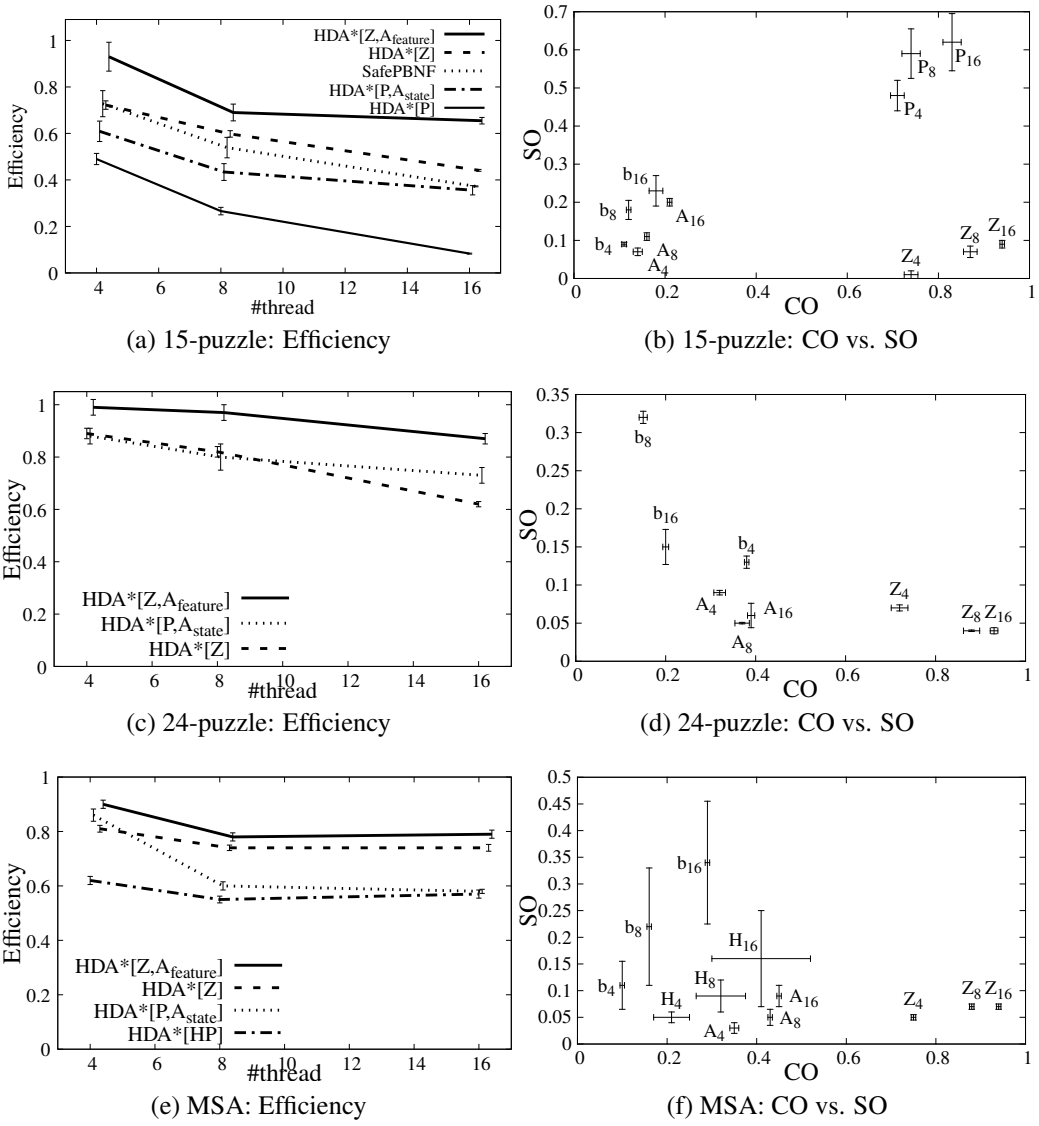
Figure 4-3: Load balance (LB) and search overhead (SO) on 100 instances of the 15-Puzzle for 4/8/16 threads. "A" = $HDA^*[Z, A_{feature}]$, "Z" = $HDA^*[Z]$, "b" = $HDA^*[P, A_{state}]$, "P" = $HDA^*[P]$, e.g., "Z_8" is the LB and SO for Zobrist hashing on 8 threads. 2-D error bars show standard error of the mean for both SO and LB.

dicates a very strong correlation between high load imbalance and search overhead. We discuss the relationship of load balance and search overhead in detail in Section 2.

Figure 4-4a shows the efficiency ($= \frac{speedup}{\#cores}$) of each method. $HDA^*[P]$ performed extremely poorly compared to all other HDA* variants and SafePBNF. The reason is clear from Figure 4-4b, which shows the communication and search overheads. $HDA^*[P]$ has both extremely high search overhead and communication overhead compared to all other methods. This shows that the hash function used by $HDA^*[P]$ is not well-suited as a work distribution function.

$HDA^*[P, A_{state}]$ had the lowest CO among HDA* variants (Figure 4-4b), and significantly outperformed $HDA^*[P]$. However, $HDA^*[P, A_{state}]$ has worse LB than $HDA^*[Z]$ (Figure 4-3), resulting in higher SO. For the 15-puzzle, this tradeoff is not favorable for $HDA^*[P, A_{state}]$, and Figures 4-4a-4-3 show that $HDA^*[Z]$, which has significantly better LB and SO, outperforms $HDA^*[P, A_{state}]$.

According to Figure 4-4a, SafePBNF outperforms $HDA^*[P, A_{state}]$, and is comparable to $HDA^*[Z]$ on the 15-puzzle. Although our definition of communication overhead does not apply to SafePBNF, SO for SafePBNF was comparable to $HDA^*[P, A_{state}]$, 0.11/0.17/0.24 on 4/8/16 threads.



(a) 15-puzzle: Efficiency

(b) 15-puzzle: CO vs. SO

(c) 24-puzzle: Efficiency

(d) 24-puzzle: CO vs. SO

(e) MSA: Efficiency

(f) MSA: CO vs. SO

Figure 4-4: Efficiency ($= \frac{speedup}{\#cores}$), Communication Overhead (CO), and Search Overhead (SO) for 15-puzzle (100 instances), 24-puzzle (100 instances), and MSA (60 instances) on 4/8/16 threads. The open list is implemented using a 2-level bucket for sliding-tiles, and as a binary heap for MSA. In the CO vs SO plot, "A" = $HDA^*[Z, A_{feature}]$ (AZHDA*), "Z" = $HDA^*[Z]$ (ZHDA*), "b" = $HDA^*[P, A_{state}]$ (AHDA*), "P" = $HDA^*[P]$, "H" = $HDA^*[Hyperplane]$, e.g., "Z_8" is the CO and SO for Zobrist hashing on 8 threads. Error bars show standard error of the mean.

$HDA^*[Z, A_{feature}]$ significantly outperformed $HDA^*[Z]$, $HDA^*[P, A_{state}]$, and SafePBNF. As shown in Figure 4-4b, although $HDA^*[Z, A_{feature}]$ had higher SO than $HDA^*[Z]$ and higher CO than $HDA^*[P, A_{state}]$, it achieved a balance between these overheads which resulted in high overall efficiency. The tradeoff between CO and SO depends on each domain and instance. By tuning the size of the abstract feature, we can choose a suitable tradeoff.

4.1.2 24-Puzzle

We generated a set of 100 random instances that could be solved by A* within 1000 seconds. For the same reason as with the 15-puzzle experiments above in Section 1.1, these are different from the 24-puzzle instances used in 2. We chose the hardest instances solvable given the memory limitation (128GB). The average runtime of sequential A* on these instances was 219.0 seconds. The average solution length of our 24-puzzle instances was 92.9 (the average solution length in th epreious work by Korf and Felner (2002) was 100.8). We used a disjoint pattern database heuristic (Korf & Felner, 2002). For the sliding-tile puzzle, the disjoint pattern database heuristic is much more efficient than Manhattan distance, thus the average walltime of 24-puzzle with disjoint pattern database heuristic was much faster than that of 15-puzzle with Manhattan distance heuristic, even though the 24-puzzle search space is much larger than the 15-puzzle search space. Figure 4-2d shows the feature projections we used for 24-puzzle. For $HDA^*[Z]$ and $HDA^*[P, A_{state}]$, we used same configurations as in Section 2. The abstraction used by SafePBNF ignores the numbers on all of the tiles except tiles 1,2,3,4, and 5 (we tried (1) ignoring all tiles except blank and tiles 1-2, (2) ignoring all tiles except blank and tiles 1-3, (3) ignoring all tiles except blank and tiles 1-4, (4) ignoring all tiles except tiles 1-3, (5) ignoring all tiles except tiles 1-4, (6) ignoring all tiles except tiles 1-5, and chose (6), the best performer).

Figure 4-4c shows the efficiency of each method. As with the 15-puzzle, $HDA^*[Z, A_{feature}]$ significantly outperformed $HDA^*[Z]$ and $HDA^*[P, A_{state}]$, and Figure 4-4d shows that as with the 15-puzzle, $HDA^*[Z]$ and $HDA^*[P, A_{state}]$ succeed in mitigating only one of the overheads (SO or CO). In contrast, $HDA^*[Z, A_{feature}]$ outperformed both $HDA^*[Z]$ and

$HDA^*[P, A_{state}]$ as its SO was comparable to that of $HDA^*[Z]$ while its CO was roughly equal to that of $HDA^*[P, A_{state}]$.

4.1.3 Multiple Sequence Alignment

Multiple Sequence Alignment (MSA) is the problem of finding a minimum-cost alignment of a set of DNA or amino acid sequences by inserting gaps in each sequence. MSA can be solved by finding the min-cost path between corners in a n-dimensional grid, where each dimension corresponds to the position of each sequence. We used 60 benchmark instances, consisting of 10 actual amino acid sequences from BAliBASE 3.0 (Thompson, Koehl, Ripp, & Poch, 2005), and 50 randomly generated instances. The BAliBASE instances we used are: BB12021, BB12022, BB12036, BBS11010, BBS11026, BBS11035, BBS11037, BBS12016, BBS12023, BBS12032. We generated random instances by 1. select number of sequences n from 4 to 9 uniformly randomly, 2. For each sequence select a number of acids l from $5000/n * 0.9 < l < 5000/n * 1.1$, 3. choose each acid uniformly random from 20 acids. Edge costs are based on the PAM250 matrix score with gap penalty 8 (Pearson, 1990). Since there was no significant difference between the behavior of HDA* among actual and random instances, we report the average of all 60 instances. We used the pairwise sequence alignment heuristic (Korf, Zhang, Thayer, & Hohwald, 2005).

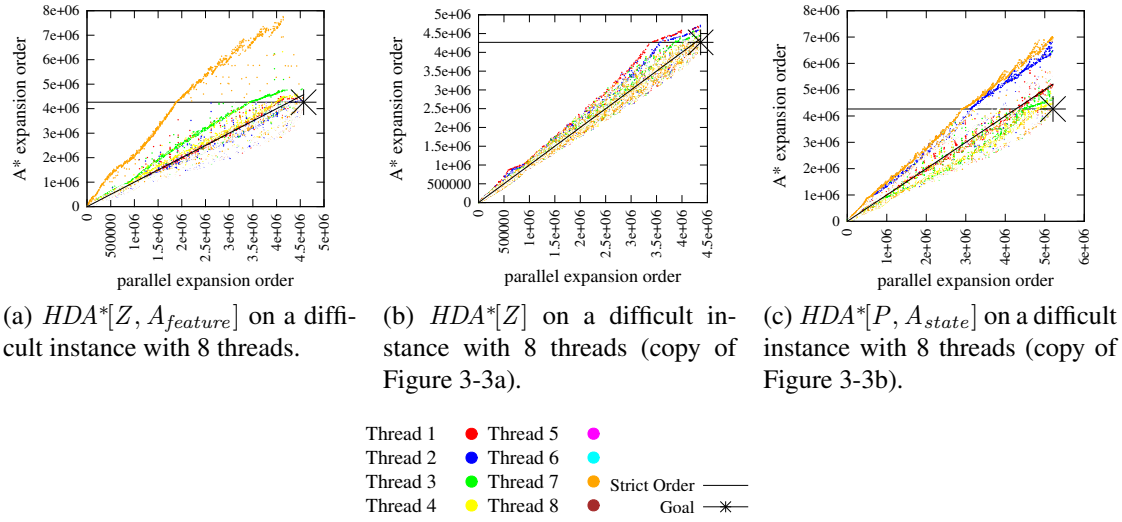
The features for Zobrist hashing and AZH were the positions of each sequence. For AZH, we grouped 4 positions per row into an abstract feature. Thus, with n sequences, nodes in the n-dimensional hypercube with edge length l share the same hash value. The abstraction used by $HDA^*[P, A_{state}]$ only considers the position of the longest sequence and ignores the others. We chose this abstraction for $HDA^*[P, A_{state}]$ as it performed the best among (1) only considering the position of the longest sequence, (2) only considering the two longest sequences, and (3) only considering the three longest sequences. We also evaluated the performance of Hyperplane Work Distribution (Kobayashi et al., 2011). $HDA^*[Z]$ suffers from node reexpansion in non-unit cost domains such as MSA. Hyperplane work distribution seeks to reduce node reexpansions by mapping the n-dimension grid to hyper-

planes (denoted as $HDA^*[Hyperplane]$). For $HDA^*[Hyperplane]$, we determined the plane thickness d using the tuning method by Kobayashi et al. (2011) where $\lambda = 0.003$, which yielded the best performance among 0.0003, 0.003, 0.03, and 0.3.

Figure 4-4e compares the efficiency of each method, and Figure 4-4f shows the CO and SO. $HDA^*[Z, A_{feature}]$ outperformed the other methods. With 4 or 8 threads, $HDA^*[Z, A_{feature}]$ had smaller SO than $HDA^*[Z]$. This is because like $HDA^*[Hyperplane]$, $HDA^*[Z, A_{feature}]$ reduced the amount of duplicated nodes in some domains compared to $HDA^*[Z]$. Our MSA solver expands 300,000 nodes/second, which is relatively slow compared to, e.g., our 24-puzzle solver, which expands 1,400,000 node/sec. When node expansions are slow, the relative importance of CO decreases, and SO has a more significant impact on performance in MSA than in the 15/24-Puzzles. Thus, $HDA^*[P, A_{state}]$, which incurs higher SO, did not perform well compared to $HDA^*[Z]$. $HDA^*[Hyperplane]$ did not perform well, but it was designed for large-scale, distributed search, and we observed $HDA^*[Hyperplane]$ to be more efficient on difficult instances than on easier instances – it is included in this evaluation only to provide another point of reference for evaluating $HDA^*[Z]$ and $HDA^*[Z, A_{feature}]$.

4.1.4 Node Expansion Order of $HDA^*[Z, A_{feature}]$

In Section 1.4, in order to see why search overhead occurs in HDA* and PBNF, we analyzed how the node expansion order of parallel search diverges from that of sequential A*. Figure 4-5 shows the expansion order of $HDA^*[Z, A_{feature}]$ on a difficult instance ($HDA^*[Z]$ and $HDA^*[P, A_{state}]$ are included for comparison). $HDA^*[Z, A_{feature}]$ has a bigger band effect than $HDA^*[Z]$, but smaller than that of $HDA^*[P, A_{state}]$. The average divergence of nodes for difficult instances are $HDA^*[Z]$: $\bar{d} = 10330.6$, $HDA^*[P, A_{state}]$: $\bar{d} = 245818$, $HDA^*[Z, A_{feature}]$: $\bar{d} = 76932.2$. Note that although the band effect of $HDA^*[Z, A_{feature}]$ in Figure 4-5a appears to be as large as the band effect of $HDA^*[P, A_{state}]$ in Figure 3-3b, the actual divergence score $\bar{d}$ is significantly higher on $HDA^*[P, A_{state}]$ ($\bar{d} = 245818$) than on $HDA^*[Z, A_{feature}]$ ($\bar{d} = 76932.2$) because $HDA^*[P, A_{state}]$ expanded more nodes



(a) $HDA^*[Z, A_{feature}]$ on a difficult instance with 8 threads.

(b) $HDA^*[Z]$ on a difficult instance with 8 threads (copy of Figure 3-3a).

(c) $HDA^*[P, A_{state}]$ on a difficult instance with 8 threads (copy of Figure 3-3b).

Figure 4-5: Comparison of $HDA^*[Z, A_{feature}]$ node expansion order vs. sequential A* node expansion order on a **difficult instance of the 15-puzzle** with 8 threads. The average node expansion order divergence scores for difficult instances are $HDA^*[Z]$: $\bar{d} = 10330.6$, $HDA^*[P, A_{state}]$: $\bar{d} = 245818$, $HDA^*[Z, A_{feature}]$: $\bar{d} = 76932.2$. AZHDA has a bigger band effect than $HDA^*[Z]$, but smaller than $HDA^*[P, A_{state}]$. Although the band of $HDA^*[Z, A_{feature}]$ appears to be as large as $HDA^*[P, A_{state}]$, the actual divergence score $\bar{d}$ is higher on $HDA^*[P, A_{state}]$ as $HDA^*[P, A_{state}]$ expands more nodes.

(>5,000,000 nodes) than $HDA^*[Z, A_{feature}]$ (>4,500,000 nodes), $HDA^*[P, A_{state}]$ has significantly larger divergence than $HDA^*[Z, A_{feature}]$.

4.2 Automated, Domain Independent Abstract Feature Generation

In Section 1, we evaluated hand-crafted, domain-specific feature projection functions for instances of the HDA* framework ($HDA^*[Z]$, $HDA^*[P]$, $HDA^*[P, A_{state}]$, $HDA^*[Z, A_{feature}]$), and showed that AZH outperformed previous methods. Next, we turn our focus to fully automated, domain-independent methods for generating feature projection functions which can be used when a formal model of a domain (such as PDDL/SAS+ for classical planning) is available.

From now on, we discuss domain-independent methods for work distribution. Table 4.1 summarizes the previously proposed methods and their abbreviations.

Table 4.1: Comparison of **previous automated domain-independent feature generation methods** for HDA*. CO: communication overhead, SO: search overhead, "optimized": the method explicitly optimizes the overhead (approximately). "ad hoc": the method seeks to mitigate the overhead but without an explicit objective function. "not addressed": the method does not address the overhead.

abbreviation	method	CO	SO
FAZHDA*	$HDA^*[Z, A_{feature}/DTG_{fluency}]$ (Sec. 2.2) (Jinnai & Fukunaga, 2016b)	ad hoc	ad hoc
GAZHDA*	$HDA^*[Z, A_{feature}/DTG_{greedy}]$ (Sec. 2.1) (Jinnai & Fukunaga, 2016b)	ad hoc	ad hoc
OZHDA*	$HDA^*[Z_{operator}]$ (Sec. 6) (Jinnai & Fukunaga, 2016b)	ad hoc	ad hoc
DAHDA*	$HDA^*[Z, A_{state}/SDD_{dynamic}]$ (Sec. 7, Appendix A) (Jinnai & Fukunaga, 2016b)	optimized	not addressed
AHDA*	$HDA^*[Z, A_{state}/SDD]$ (Sec. 7) (Burns et al., 2010)	optimized	not addressed
ZHDA*	$HDA^*[Z]$ (Sec. 6) (?)	not addressed	optimized

For $HDA^*[Z]$, automated domain-independent feature generation for classical planning problems represented in the SAS+ representation (Bäckström & Nebel, 1995) is straightforward (Kishimoto et al., 2013). For each possible assignment of value k to variable v_i in a SAS+ representation, e.g., $v_i = k$, there is a binary proposition $x_{i,k}$ (i.e., the corresponding STRIPS propositional representation). Each such proposition $x_{i,k}$ is a feature to which a randomly generated bit string is assigned, and the Zobrist hash value of the state can be computed by xor'ing the propositions that describe a state, as in Equation 2.1.

For AHDA*, the abstract representation of the state space can be generated by ignoring some of the features (SAS+ variables) and using the rest of the features to represent the abstraction. Burns et al. used the greedy abstraction algorithm by Zhou and Hansen (2006b) to select the subset of features, which we refer to as SDD abstraction. It adds one atom group to the abstract graph at a time, choosing the atom group which minimizes the maximum out-degree of the abstract graph, until the graph size (number of abstract nodes)

reaches the threshold given by a parameter. As we saw in Section 1, the hashing strategy for abstract state has little effect on the performance. We used the implementation of AHDA* with Zobrist hashing and SDD abstraction ($HDA^*[Z, A_{state}/SDD]$).

For AZHDA* ($HDA^*[Z, A_{feature}]$), the feature projection function which generates abstract features from raw features plays a critical role in determining the performance of AZHDA*, because AZHDA* relies on the feature projection in order to reduce communications overhead. In this section, we discuss two methods to automatically generate the feature projection function for AZH. Greedy abstract feature generation (GreedyAFG), which partitions each *domain transition graph* (DTG) into 2 abstract features, and fluency-based abstract feature generation (FluencyAFG), an extension of GreedyAFG which filters the DTGs to partition according to a fluency-based criterion. GreedyAFG and FluencyAFG seek to generate efficient feature projection functions without an explicit model of what to optimize. Further details on GreedyAFG and FluencyAFG can be found in our previous conference paper (Jinnai & Fukunaga, 2016b).

4.2.1 Greedy Abstract Feature Generation (GAZHDA*)

Greedy abstract feature generation (GreedyAFG) is a simple, domain-independent abstract feature generation method, which partitions each feature into 2 abstract features (Jinnai & Fukunaga, 2016a). GreedyAFG first identifies *atom groups* (?) and its domain transition graph (DTG). Atom group is a set of mutually exclusive propositions from which exactly one will be true for each reachable state, e.g., the values of a SAS+ multi-valued variable (Bäckström & Nebel, 1995). GreedyAFG maps each atom group X into 2 abstract features S_1 and S_2, based on X's undirected DTG (nodes are values, edges are transitions), as follows: (1) assign the minimal degree node (node with the least number of edges between other nodes) to S_1; (2) greedily add to S_1 the unassigned node which shares the most edges with nodes in S_1; (3) while $|S_1| < |X|/2$ repeat step 2; (4) assign all unassigned nodes to S_2. Due to the loop criterion in step 3, this procedure guarantees a perfectly balanced bisection of the DTGs, i.e., $|S_2| \leq |S_1| \leq |S_2| + 1$, so load balancing is minimized. $A(x_i)$

in Equation 4.1 corresponds to the mapping from x_i to S_1, S_2, and R_i is defined over S_1 and S_2. We denote GAZHDA* as $HDA^*[Z, A_{feature}/DTG_{greedy}]$, as it applies feature abstraction (FA) by cutting DTGs using GreedyAFG.

Algorithm 5: Greedy Abstract Feature Generation

Input: X: an atom group

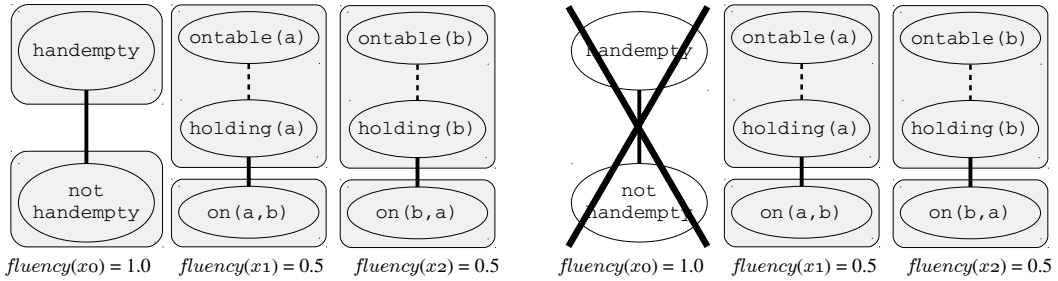
1 Assign the minimal degree node (node with the least number of edges between other nodes) to S_1;
2 **while** $|S_1| < |G|/2$ **do**
3 Greedily add to S_1 the unassigned node which shares the most edges with nodes in S_1;
4 Assign all unassigned nodes to S_2.;
5 **Return** (S_1, S_2);

4.2.2 Fluency-Dependent Abstract Feature Generation (FAZHDA*)

Since the hash value of the state changes if any abstract feature value changes, GreedyAFG fails to prevent high CO when any abstract feature changes its value very frequently, e.g., in the blocks domain, every operator in the domain changes the value of the SAS+ variable representing the state of the robot's hand (handempty $\leftrightarrow$ not-handempty). *Fluency-dependent abstract feature generation* (FluencyAFG) overcomes this limitation (Jinnai & Fukunaga, 2016b). The *fluency* of a variable v is the number of ground actions which change the value of the v divided by the total number of ground actions in the problem. By ignoring variables with high fluency, FluencyAFG was shown to be quite successful in reducing CO and increasing speedup compared to GreedyAFG.

A problem with fluency is that in the AZHDA* framework, CO is associated with a change in value of an abstract feature, not the feature itself. However, FluencyAFG is based on the frequency with which features (not abstract features) change. This leads FluencyAFG to exclude variables from consideration unnecessarily, making it difficult to achieve good LB (in general, the more variables are excluded, the more difficult it becomes to reduce LB). Figure 4-6 shows how fluency-based filtering is applied to the blocks domain. The process of fluency-based filtering which ignores a subset of features can be described as

an instance of abstraction. Therefore, we denote FAZHDA* as $HDA^*[Z, A_{feature}/DTG_{fluency}]$, as it applies fluency-based abstraction, and then GAZHDA*.



(a) GreedyAFG (b) FluencyAFG

Figure 4-6: Greedy abstract feature generation (GreedyAFG) and Fluency-dependent abstract feature generation (FluencyAFG) applied to blocksworld domain. The hash value for a state $s = (x_0, x_1, x_2)$ is given by $AZ(s) = R[A(x_0)]$ *xor* $R[A(x_1)]$ *xor* $R[A(x_2)]$. Grey squares are abstract features A generated by GreedyAFG, so all propositions in the same square have same hash value (e.g. $R[A(\mathsf{holding(a)})] = R[A(\mathsf{ontable(a)})]$). $fluency(x_0) = 1$ since all actions in the blocks world domain change its value. In this case, any abstract features based on the other variables are rendered useless, as all actions change x_0 and thus change the state's hash value. In this example, Fluency-dependent AFG will filter x_0 before calling GreedyAFG to compute abstract features based on the remaining variables (thus $AZ(s) = R[A(x_1)]$ *xor* $R[A(x_2)]$).

Chapter 5

A Graph Partitioning-Based Model for Work Distribution

Although GAZHDA* and FAZHDA*, the domain-independent abstract feature generation methods discussed in Section 2, seek to reduce communications overhead compared to $HDA^*[Z]$, they are not based on an explicit model which enables the prediction of the actual communications overhead achieved during the search. Furthermore, the impact of these methods on search overhead is completely unspecified, and thus, it is not possible to predict the parallel efficiency achieved during the search. Previous work relied on ad hoc, control parameter tuning in order to achieve good performance (Jinnai & Fukunaga, 2016b). In this section, we first show that a work distribution method can be modeled as a partition of the search space graph, and that communication overhead and load balance can be understood as the number of cut edges and balance of the partition, respectively. Using this model, we introduce a metric, *estimated efficiency*, and we experimentally show that the metric has a strong correlation to the actual efficiency. This leads to the GRAZHDA* feature generation method described in Section 6.

5.1 Work Distribution as Graph Partitioning

Work distribution methods for hash-based parallel search distribute nodes by assigning a process to each node in the state space. Our goal is to design a work distribution method which maximizes efficiency by reducing CO, SO, and load balance (LB). In particular, given a problem instance, we seek a principled method of quickly, automatically generating a work distribution method (hash function) for HDA* for that particular problem instance. We propose an approach which is based on optimizing *a priori* estimates of CO, SO, and LB. In our approach, given a problem, we search a space of hash functions, using these estimates of CO, SO, LB as the basis for a (cheap) evaluation function for this search in the space of hash functions. To enable this, we first develop a model for estimating algorithm performance based on the notion of a workload graph.

To guarantee the optimality of a solution, a parallel search method needs to expand a goal node and all nodes with $f < f^*$ (relevant nodes S). The workload distribution of a parallel search can be modeled as a partitioning of an undirected, unit-cost *workload graph* G_W which is isomorphic to the *relevant* search space graph, i.e., nodes in G_W correspond to states in the search space with $f < f^*$ and goal nodes, and edges in the workload graph correspond to edges in the search space between nodes with $f < f^*$ and goal nodes. The distribution of nodes among p processors corresponds to a p-way partition of G_W, where nodes in partition S_i are assigned to process p_i.

The workload graph G_W only includes nodes with $f < f^*$, for the following reason. We are ultimately trying to develop a method for quickly estimating SO, CO, and LB for a work distribution scheme S without actually running S. In principle, if we knew exactly the actual portion of the graph which is explored by HDA* with a particular partitioning scheme, then this would allow us to accurately compute search efficiency. However, that requires running HDA* until a solution is found, so this is impractical, and we need an approximation of the actual explored nodes. The set of nodes with $f < f^*$ is a reasonable approximation to the nodes which are explored by HDA*, because these are the set of nodes

which must be expanded regardless of the hash function (partitioning method). Depending on the hash function, some nodes with $f \geq f^*$ are expanded, but it is not possible to know how many such nodes will be expanded without actually running HDA* with that hash function. Therefore, although the workload graph underestimates the size of the actual relevant search space, it is a reasonable approximation. While underestimating the relevant search space is not ideal, the converse (considering states which are irrelevant to the actual HDA*) is problematic. For example, if we consider the entire search space (i.e., including all nodes with $f \geq f^*$) would be mapped to processors if the search algorithm continued to execute until the space is exhausted, then $HDA^*[P]$ (Section 1.1) successfully partitions the space evenly, i.e., "perfect load balance". However, as shown in Figure 4-3, $HDA^*[P]$ has the worst load balance in the actual experiment. This is because the distribution of $HDA^*[P]$ is highly biased in the search space so that the relevant state space ($f \leq f^*$), which is a small fraction of the state space, is distributed unevenly. Considering only the nodes with $f < f^*$ allows us to capture this bias. This example also illustrates how using a perfect hashing which balances the partitions for the entire search space does not does not achieve good performance unless the partitions are also balanced with respect to portion of the the search space which is actually explored by the search algorithm.

Given a partitioning of G_W, LB and CO can be estimated directly from the structure of the graph, without having to run HDA and measure LB and CO experimentally, i.e., it is possible to predict and analyze the efficiency of a workload distribution method without actually executing HDA*.* Therefore, although it is necessary to run A* or HDA* once to generate a workload graph,[1] we can subsequently compare the LB and CO of many partitioning methods without re-running HDA* for each partitioning method. LB corresponds to load balance of the partitions and CO is the number of edges between partitions over the

1. Hence, this is not yet a practical method for automatic hash function generation – a further approximation of this model which does not require generating the workload graph, and yields a practical method is described in Section 6.

number of total edges, i.e.,

$$CO = \frac{\sum_i^p \sum_{j>i}^p E(S_i, S_j)}{\sum_i^p \sum_{j\geq i}^p E(S_i, S_j)}, \quad LB = \frac{|S_{max}|}{mean|S_i|}, \tag{5.1}$$

where $|S_i|$ is the number of nodes in partition S_i, $E(S_i, S_j)$ is the number of edges between S_i and S_j, $|S_{max}|$ is the maximum of $|S_i|$ over all processes, and $mean|S| = \frac{|S|}{p}$.

Next, consider the relationship between SO and LB. It has been shown experimentally that an inefficient LB leads to high SO, but to our knowledge, there has been no previous analysis on *how* LB leads to SO in parallel best-first search. Assume that the number of duplicate nodes is negligible[2], and every process expands nodes at the same rate. Since HDA* needs to expand all nodes in S, each process expands $|S_{max}|$ nodes before HDA* terminates. As a consequence, process p_i expands $|S_{max}| - |S_i|$ nodes not in the relevant set of nodes S. By definition, such irrelevant nodes are search overhead, and therefore, we can express the overall search overhead as:

$$\begin{aligned} SO &= \sum_i^p (|S_{max}| - |S_i|) \\ &= p(LB - 1). \end{aligned} \tag{5.2}$$

5.2 Parallel Efficiency and Graph Partitioning

In this section we develop a metric to estimate the walltime efficiency as a function of CO and SO. First, we define *time efficiency* $eff_{actual} := \frac{speedup}{\#cores}$, where $speedup = T_N/T_1$, T_n is the runtime on N cores and T_1 the runtime on 1 core. Our ultimate goal is to maximize eff_{actual}.

2. The number of duplicate node is closely related to LB and CO. If the order of node expansion is exactly the same as A*, then the number of duplicate is 0. The duplicate nodes occur when LB is suboptimal and the order of node expansion diverges from A*. The other cause of duplicate is CO. Even if the load balance is optimal, the optimal path may be disturbed by communication latency and suboptimal path may be discovered first, resulting in duplicate nodes. Therefore, optimizing LB and CO leads to reducing duplicate nodes.

Communication Efficiency: Assume that the communication cost between every pair of processors is identical. If t_{com} is the time spent sending nodes from one core to another[3], and t_{proc} is the time spent processing nodes (including node generation and evaluation). Hence communication efficiency, the degradation of efficiency by communication cost, is $\mathit{eff}_c = \frac{1}{1+cCO}$, where $c = \frac{t_{com}}{t_{proc}}$.

Search Efficiency: Assuming all cores expand nodes at the same rate and that there are no idle cores, HDA* with p processes expands np nodes in the same wall-clock time A* requires to expand n nodes. Therefore, search efficiency, the degradation of efficiency by search overhead, is $\mathit{eff}_s = \frac{1}{1+SO}$.

Using CO and LB (and SO from Equation 5.2), we can estimate the time efficiency eff_{actual}. eff_{actual} is proportional to the product of communication and search efficiency: $\mathit{eff}_{actual} \propto \mathit{eff}_c \cdot \mathit{eff}_s$. There are overheads other than CO and SO such as hardware overhead (i.e. memory bus contention) that affect performance (Burns et al., 2010; Kishimoto et al., 2013), but we assume that CO and SO are the dominant factors in determining efficiency.

We define *estimated efficiency* eff_{esti} as $\mathit{eff}_{esti} := \mathit{eff}_c \cdot \mathit{eff}_s$, and we use this metric to estimate the actual performance (efficiency) of a work distribution method.

$$\begin{aligned} \mathit{eff}_{esti} &= \mathit{eff}_c \cdot \mathit{eff}_s = \frac{1}{(1+cCO)(1+SO)} \\ &= \frac{1}{(1+cCO)(1+p(LB-1))} \end{aligned} \tag{5.3}$$

5.2.1 Experiment: eff_{esti} model vs. actual efficiency

To validate the usefulness of eff_{esti}, we evaluated the correlation of eff_{esti} and actual efficiency on the following HDA* variants discussed in Section 1 on domain-independent planning.

- FAZHDA*: $HDA^*[Z, A_{feature}/DTG_{fluency}]$, AZHDA* using fluency-based filtering (FluencyAFG).

3. In a multicore environment, the cost of "sending" a node from thread p_1 to p_2 is the time required to obtain access to the incoming queue for p_2 (via a successful `try_lock` instruction).

- GAZHDA*: $HDA^*[Z, A_{feature}/DTG_{greedy}]$, AZHDA* using greedy abstract feature generation (GreedyAFG).
- OZHDA*: $HDA^*[Z_{operator}]$, Operator-based Zobrist hashing (Sec. 6).
- DAHDA*: $HDA^*[Z, A_{state}/SDD_{dynamic}]$, AHDA* (Burns et al., 2010) with dynamic abstraction size threshold (Appendix A).
- ZHDA*: $HDA^*[Z]$, HDA* using Zobrist hashing (Kishimoto et al., 2013) (Sec. 6).

We implemented these HDA* variants on top of the Fast Downward classical planner using the merge&shrink heuristic (Helmert, Haslum, Hoffmann, & Nissim, 2014) (abstraction size =1000). We parallelized Fast Downward using using MPICH3. We selected a set of IPC benchmark instances that are difficult enough so that parallel performance differences could be observed. We ran experiments on a cluster of 6 machines, each with an 8-core Intel Xeon E5410 2.33 GHz CPU with 16 GB RAM, and 1000Mbps Ethernet interconnect. For FAZHDA*, we ignored 30% of the variables with the highest fluency as it performed the best out of 10%, 20%, 30%, 50%, and 70%. DAHDA* uses at most 30% of the total number of features in the problem instance (we tested 10%, 30%, 50%, and 70% and found that 30% performed the best). We packed 100 states per MPI message in order to reduce the number of messages (Romein et al., 1999).

Table 6.2 shows the speedups (time for 1 process / time for 48 processes). We included the time for initializing work distribution methods (for all runs, the initializations completed in ≤ 1 second), but excluded the time for initializing the abstraction table for the merge&shrink heuristic. From the measured runtimes, we can compute actual efficiency eff_{actual}. Then, we calculated the performance estimated eff_{esti} as follows. We generated the workload graph G_W for each instance (i.e., enumerated all nodes with $f \leq f^*$ and edges between these nodes), and calculated LB, CO, SO, and eff_{esti} using Eqs 5.1-5.3. Figure 5-1, which compares estimated efficiency eff_{esti} vs. the actual measured efficiency eff_{actual}, indicates a strong correlation between eff_{esti} and eff_{actual}. Using least-square regression to estimate the coefficient a in $\mathit{eff}_{actual} = a \cdot \mathit{eff}_{esti}$, we obtained $a = 0.86$ with variance of residuals 0.013. Note that $a < 1.0$ because there are other sources of overhead which not

accounted for in eff_{esti}, (e.g. memory bus contention) which affect performance (Burns et al., 2010; Kishimoto et al., 2013).

Observation 5 *The eff_{esti} metric for a partitioning scheme, which can be computed from the workload distribution graph (without running HDA* using that partitioning scheme), is strongly correlated with the actual measured efficiency eff_{actual} of HDA*.*

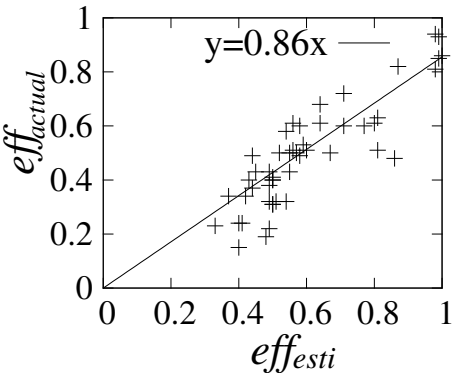


Figure 5-1: Comparison of eff_{esti} and the actual experimental efficiency when communication cost $c = 1.0$ and the number of processes $p = 48$. The figure aggregates the data points of FAZHDA*, GAZHDA*, OZHDA*, DAHDA*, and ZHDA* shown in Figure 6.1. $eff_{actual} = 0.86 \cdot eff_{esti}$ with variance of residuals = 0.013 (least-squares regression).

Chapter 6

Graph Partitioning-Based Abstract Feature Generation (GRAZHDA*)

A standard approach to workload balancing in parallel scientific computing is graph partitioning, where the workload is represented as a graph, and a partitioning of the graph according to some objective (usually the cut-edge ratio metric) represents the allocation of the workload among the processors (Hendrickson & Kolda, 2000; Buluc, Meyerhenke, Safro, Sanders, & Schulz, 2015).

In Section 5, we showed that work distributions for parallel search on an implicit graph can be modeled as partitions of a workload graph which is isomorphic to the search space, and that this workload graph can be used to estimate the CO and LB of a work distribution. If we were given a workload graph, then by defining a graph cut objective such that partitioning the nodes in the search space (with $f \leq f^*$) corresponds to maximizing the efficiency, we would have a method of generating an optimal workload distribution. Unfortunately, this is impractical as the workload graph is an explicit representation of the relevant state space graph, i.e., this a solution to the search problem itself!

However, a practical alternative is to apply graph partitioning to a graph which serves an approximate, proxy for the actual state space graph. We propose *GRaph partitioning-based Abstract Zobrist HDA** (**GRAZHDA***), which approximates the optimal graph partitioning-

based strategy by partitioning *domain transition graphs* (DTG). Given a classical planning problem represented in SAS+, the domain transition graph (DTG) of a SAS+ variable X, $\mathcal{D}_X(E, V)$, is a directed graph where vertices V corresponds to the possible values of a variable X, edges E represent transitions among the values of X, and $(v, v') \in E$ iff there is an operator (action) o with $v \in del(o)$ and $v' \in add(o)$ (Jonsson & Bäckström, 1998).

Listing 6.1: Sliding-tile puzzle PDDL

```
(define (domain strips-sliding-tile)
  (:requirements :strips)
  (:predicates
   (tile ?x) (position ?x)
   (at ?t ?x ?y) (blank ?x ?y)
   (inc ?p ?pp) (dec ?p ?pp))
  (:action move-up
    :parameters (?omf ?px ?py ?by)
    :precondition (and
                    (tile ?omf) (position ?px) (position ?py) (position ?by)
                    (dec ?by ?py) (blank ?px ?by) (at ?omf ?px ?py))
    :effect (and (not (blank ?px ?by)) (not (at ?omf ?px ?py))
                 (blank ?px ?py) (at ?omf ?px ?by)))
  (:action move-left
        .
        .
```

The DTGs for a problem provide a highly compressed representation which reflects the structure of the search space, and is easily extracted automatically from the formal domain description (e.g., PDDL/SAS+). We expect DTGs to be good proxies for the search space because DTGs tend to be orthogonal to each other – otherwise the propositions of the DTG is redundant (this is not always true as PDDL may contain dual representations, e.g. sokoban).

GRAZHDA* partitions each DTG into two abstract features according to an objective function. That is, each DTG is partitioned into two subsets S_1 and S_2. Projection $A(x)$ is defined on the value of the DTG, and returns 1 or 0 depending on whether S_1 or S_2 it is included in. Abstract Zobrist hashing is then applied using these abstract features

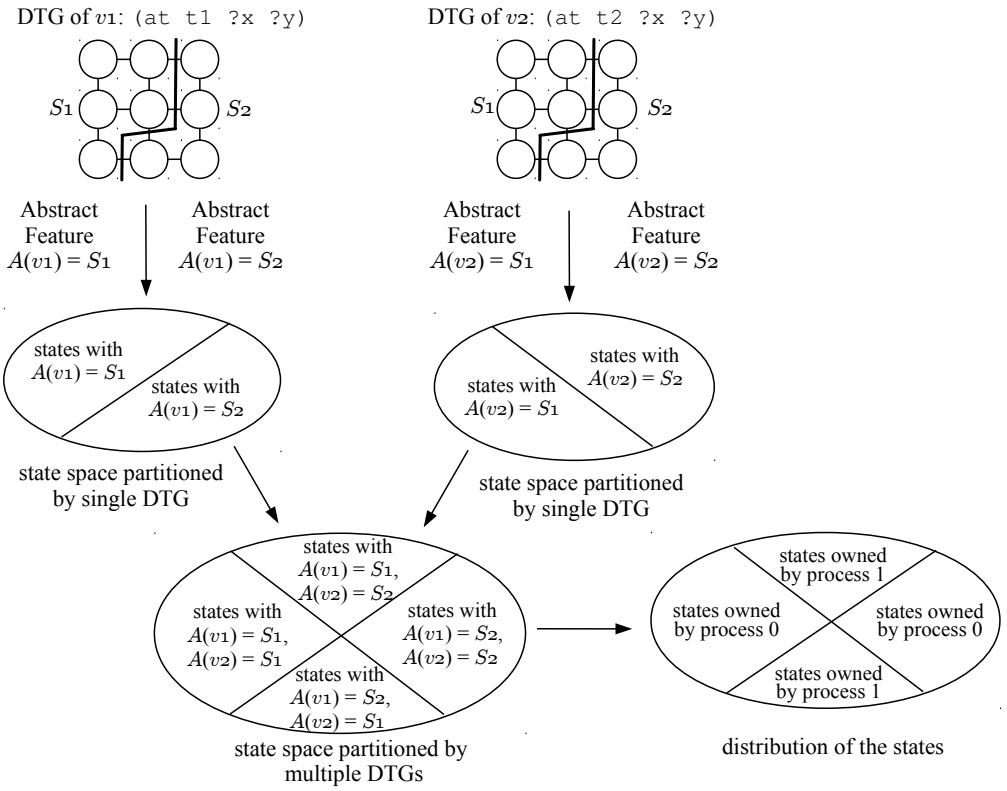


Figure 6-1: GRAZHDA* applied to 8 puzzle domain. The SAS+ variable v_1 and v_2 correspond to the position of tile 1 and 2. The domain transition graphs (DTGs) of v_1 and v_2 are shown in the top of the figure (e.g. $v_1 = \{$`(at t1 x1 y1), (at t1 x1 y2), (at t1 x1 y3), ...`$\}$). GRAZHDA* partitions each DTG with given objective function to generate abstract feature S_1 and S_2, and $A(v_1) = S_1, S_2$. Thus, the hash value of abstract feature $R[A(v_1)]$ corresponds to which partition v_1 belongs to. As DTGs are compressed representation of the state space graph, partitioning a DTG corresponds to partitioning a state space graph. By xor'ing $R[A(v_1)], R[A(v_2)]$, ..., the hash value $AZ(s)$ represents for each variable v_i which partition it belongs to.

(random table R in Equation 4.1 is defined on S_1 and S_2). In GRAZHDA*, AZH uses each partition of the DTG as an abstract feature, assigning a hash value to each abstract feature (Figure 6-1). Since the AZH value of a state is the XOR of the hash values of the abstract features (Equation 4.1), 2 nodes in the state space are in different partitions if and only if they are partitioned in *any* of the DTGs. Therefore, GRAZHDA* generates 2^n partitions from n DTGs, which are then projected to the p processors (by taking the

hash value modulo p, *processor(s) = hashvalue(s)* $\bmod p$).[1] We denote GRAZHDA* as $HDA^*[Z, A_{feature}/DTG]$, where DTG stands for DTG-partitioning.

6.1 Previous Methods and Their Relationship to GRAZHDA*

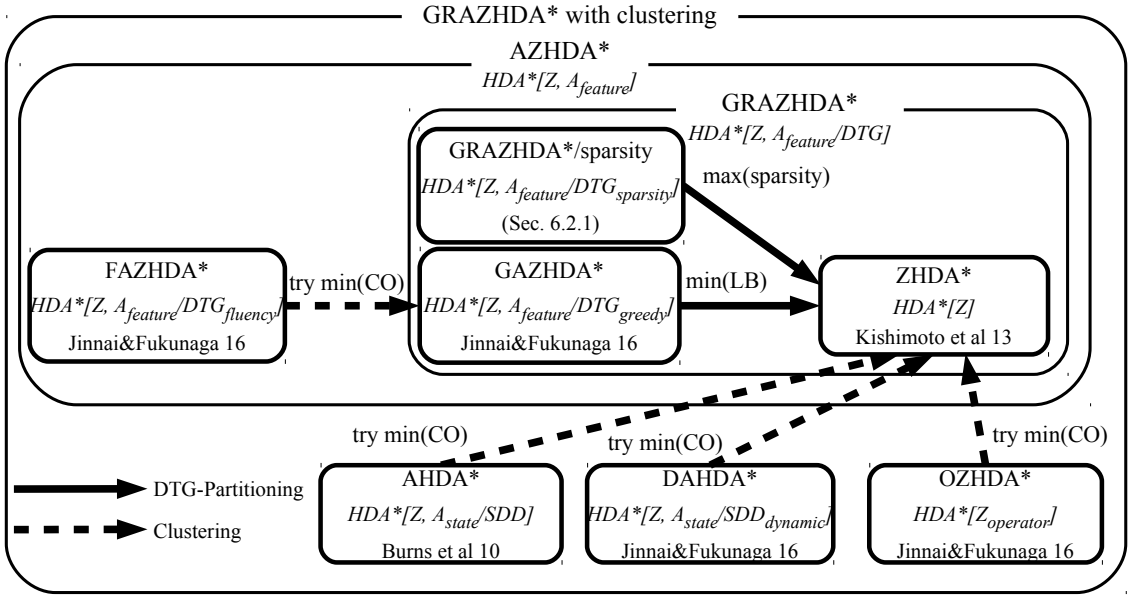


Figure 6-2: Work distribution methods described as an instances of GRAZHDA* with clustering. Previous methods can be seen as GRAZHDA* + clustering with suboptimal objective function. The arrows represent the relationship of methods. For example, FAZHDA* applies fluency-based filtering to ignore some variables, and then applies GreedyAFG to partition DTGs. This can be described as applying clustering, partitioning, and then Zobrist hashing. As such, all previous methods discussed in this thesis can be explained as instances of GRAZHDA* (with clustering).

In this section we show that previously proposed methods for the HDA* framework can be interpreted as instances of GRAZHDA*. First, we define a DTG-partitioning as follows: given $s = (v_0, v_1, ..., v_n)$, a DTG-partitioning maps a state s to an abstract state

1. In HDA* the owner of a state is computed as *processor(s) = hashvalue(s)* $\bmod p$, so it is possible that states with different hash values are assigned to the same thread. Also, while extremely unlikely, it is theoretically possible that s and s' may have the same hash value even if they have different abstract features due to the randomized nature of Zobrist hashing (in all our HDA* variants, we detect such collisions by always comparing the values stored in the hash table whenever hash keys point to a nonempty hash table entry).

$s' = (A_0[v_0], A_1[v_1], ..., A_n[v_n])$, where $A_i[v_i]$ is defined by a graph partitioning on each DTG while optimizing given objective function. DTG-partitioning corresponds to AF/DTG for an abstraction strategy. Then, in order to model non-DTG based methods, we refer to all other methods which map a state space to an abstract state space with or without objectives a *clustering*. For example, by ignoring subset of the variables, we get an abstract state $s' = (v_0, ..., v_m)$ where $m < n$. Clustering corresponds to any abstraction strategy other than DTG-partitioning. Using this terminology, the relationship between GRAZHDA* and previous methods is summarized in Figure 6-2.

First, $HDA^*[Z]$, the original Zobrist-hashing based HDA* (?; Kishimoto et al., 2013), corresponds to an extreme case where every node in DTG is assigned to a different partition (for all A_i, $A_i[v_i] \neq A_i[v'_i]$ if $v_i \neq v'_i$).

GAZHDA* (GreedyAFG) (Jinnai & Fukunaga, 2016a), described in Section 2.1 is in fact applying DTG-partitioning whose objective function is to minimize LB as the primary objective, with a secondary objective of (greedily) minimizing CO, as it tries to assign the most connected node but does not optimize. Thus, GAZHDA* an instance of GRAZHDA*.

AHDA* (Burns et al., 2010) (Section 7), FAZHDA* (Jinnai & Fukunaga, 2016b) (Section 2.2), OZHDA* (Jinnai & Fukunaga, 2016b) (Section 6), and DAHDA* (Jinnai & Fukunaga, 2016b) (Section7), are instances of GRAZHDA* with *clustering*, which map the state space graph to an abstract state space graph, and then apply DTG-partitioning to the abstract state space graph so that the nodes mapped to the same abstract state are guaranteed to be assigned to the same partition, so that there no communication overhead is incurred when generating a node that is in the same abstract state as its parent.

AHDA* generates an abstract state space by ignoring some of the features (DTGs) in the state representation and then it applies hashing to the abstract state space. Ignoring part of the state representation can be interpreted as a *clustering* of nodes so that all of the nodes in a cluster are allocated to the same processor. The problem with AHDA* is the criteria used to determine which features to ignore (conversely, which features to take into account). It minimizes the highest degree of the abstract nodes, as the abstraction

method used by AHDA* was originally proposed for duplicate detection of external search (Zhou & Hansen, 2006b). However, this doe not correspond to a natural objective function which optimizes parallel work distribution objective such as edge cut or load balancing. Therefore, although the projection of AHDA* result in significantly reduced CO, it does not explicitly try to optimize it; CO is reduced as a fortunate side-effect of generating efficient abstract state space for external search. DAHDA* (Jinnai & Fukunaga, 2016b) improves upon AHDA* by dynamically tuning the number of DTGs which are ignored (see Appendix A), but the state projection mechanism is the same as AHDA*.

FAZHDA* is a variant of GAZHDA*, which, instead of using all the variables as GAZHDA* does, FAZHDA* ignores some of the variables in the state based on their *fluency*, which is defined as the number of ground actions which change the value of the variable divided by the total number of ground actions in the problem. As we pointed out above for AHDA*, ignoring variables can be described as a clustering. Although fluency-based filtering is intended to reduce CO, ignoring high fluency variables is only a heuristic which sometimes succeeds in reducing CO, but sometimes fails, since fluency is defined on the frequency of the change of the feature (value), but the change of abstract feature is what incurs CO. Even if the fluency of a variable is 1.0, the value may change within an abstract feature, thus eliminating the DTG does not improve any CO whatsoever. Fluency-based filtering only takes into account of the fluency of the variable, whereas GRAZHDA* framework looks into each transition in the DTG to choose how to treat the variable.

OZHDA* clusters nodes connected with selected operators and applies Zobrist hashing, so that the selected operator does not cost communication. The clustering of OZHDA* is bottom-up, in the sense that state space nodes connected with selected operators are directly clustered, instead of using SAS+ variables or DTGs. The problem with OZHDA* is that the clustering is ad hoc and unbalanced – some of the nodes are clustered but the others are not, and the choice of which nodes to cluster or not is not explicitly optimized. The clustered nodes are then partitioned by assigning each node to a separate partition, as with ZHDA* (see above), but this is dangerous, since OZHDA* ends up treating clustered nodes

and original nodes equally, without considering that the clustered nodes should have larger edge cut costs than original single nodes. Thus, although the clustering done by OZHDA* is intended to reduce CO, it comes at the price of load balance – the edge costs for the (implicit) workload graph are not aggregated when the clusters are formed, so load balance is being sacrificed without an explicit objective function controlling the tradeoff.

Thus, we have shown that all previous methods for work distribution in the HDA* framework can be viewed as instances of GRAZHDA* using *ad hoc* criteria for clustering and optimization.

6.2 Effective Objective Functions for GRAZHDA*

In the previous section, we showed that previous variants of HDA* can be seen as instances of GRAZHDA* which partitioned the workload graph based on *ad hoc* criteria. However, since the GRAZHDA* framework formulates workload distribution as a graph partitioning problem, a natural idea is to design an objective function for the partitioning which directly leads to a desired tradeoff between search and communication overheads, resulting in good overall efficiency. Fortunately, a metric which can be used as the basis for such an objective is available: eff_{esti}.

In Section 2.1, we showed that eff_{esti}, based on the workload is an effective predictor for the actual efficiency of a work distribution strategy. In this section, we propose approximations to eff_{esti} which can be used as objective functions for the DTG partitioning in GRAZHDA*.

In principle, in order to maximize the performance of GRAZHDA*, it is desirable to have a function which approximates eff_{esti} as closely as possible. However, since GRAZHDA* partitions the domain transition graph as opposed to the actual workload graph (which is isomorphic to the search space graph), and the DTG is only an approximation to the actual workload graph, a perfect approximation of eff_{esti} is not feasible. Fortunately, in practice, it turns out that using a straightforward approximation of eff_{esti} as an objective function

for GRAZHDA* result in good performance when compared to previous work distribution methods.

6.2.1 Sparsest Cut Objective Function (GRAZHDA*/sparsity)

One straightforward objective function which is clearly related to eff_{esti} is a *sparsest cut* objective, which maximizes *sparsity*, defined as

$$sparsity := \frac{\prod_i^p |S_i|}{\sum_i^p \sum_{j>i}^p E(S_i, S_j)}, \tag{6.1}$$

where p is the number of partitions (= number of processors), $|S_i|$ is the number of nodes in partition S_i divided by the total number of nodes, $E(S_i, S_j)$ is the sum of edge weights between partition S_i and S_j. Consider the relationship between the sparsity of a state space graph for a search problem and the eff_{esti} metric defined in the previous section. By equations 5.3 and 5.1, sparsity simultaneously considers both LB and CO, as the numerator $\prod_i^p |S_i|$ corresponds to LB and the denominator $\sum_i^p \sum_{j>i}^p E(S_i, S_j)$ corresponds to CO.

Sparsity is used as a metric for parallel workloads in computer networks (Leighton & Rao, 1999; Jyothi, Singla, Godfrey, & Kolla, 2014), but to our knowledge this is the first proposal to use sparsity in the context of parallel search of an implicit graph.

Figure 6-3 shows the sparsest cut of a DTG (for the variable representing package location) in the standard `logistics` domain. Each edge in a DTG corresponds to a transition of its value. Edge costs w_e represent the ratio of operators which corresponds to its transition over the total number of operators in the DTG. For example in logistics, each edge corresponds to 2 operators, one in each direction ((drive-truck ?truck pos0 pos1) and (drive-truck ?truck pos1 pos0), or (fly-airplane ?plane pos0 pos1) and (fly-airplane ?plane pos1 pos0)). The total number of operator in the graph is 120, thus w_e for each edge is $2/120 = 1/60$. We use this to calculate sparsity (Equation 6.1). Maximizing sparsity results in cutting only 1 edge (Figure 6-3): it cuts the graph with $|S_1| \cdot |S_2| = 10/16 \cdot 6/16$, and edge cuts $E(S_1, S_2) = 1 \cdot w_e$, thus $sparsity = \frac{|S_1| \cdot |S_2|}{E(S_1, S_2)} = 26.72$, whereas the partition by GreedyAFG

results in cutting 21 edges ($sparsity = 0.71$). The problem with GreedyAFG is that it imposes a hard constraint requiring the partition to be perfectly balanced. While this optimizes load balance, locality (i.e., the number of cut edges) is sacrificed. GRAZHDA*/sparsity takes into account both load balance and CO without the hard constraint of bisection, resulting in a partitioning which preserves more locality.

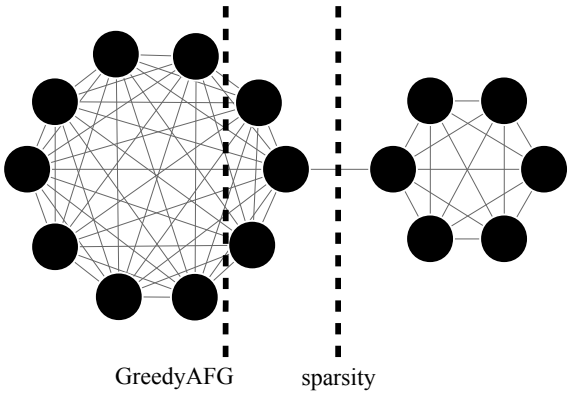
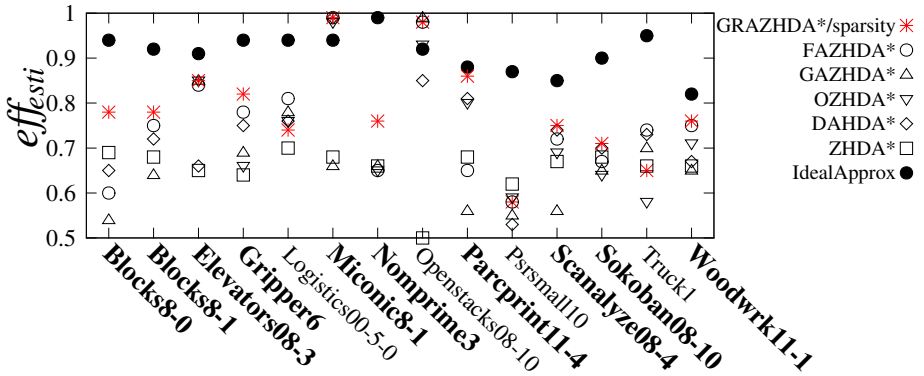


Figure 6-3: GRAZHDA*/sparsity and Greedy abstract feature generation (GreedyAFG) applied to DTG on logistics domain of 2 cities with 10/6 locations. Each node in the domain transition graph above corresponds to a location of the package (at obj12 ?loc). GreedyAFG potentially cuts many edges because it requires the best load balance possible for the cut (bisection), while GRAZHDA*/sparsity takes into account of the number of edge cut as an objective function.

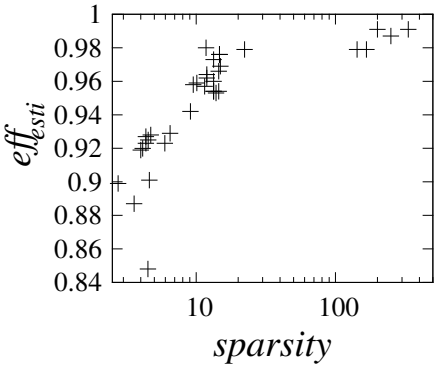
6.2.2 Experiment: Validating the Relationship between Sparsity and eff_{esti}

To validate the correlation between sparsity and estimated efficiency eff_{esti}, we used the METIS (approximate) graph partitioning package (Karypis & Kumar, 1998) to partition modified versions of the search spaces of the instances used in Fig. 6-4a. We partitioned each instance 3 times, where each run had a different set of random, artificial constraints added to the instance (we chose 50% of the nodes randomly and forced METIS to distribute them equally among the partitions – these constraints degrade the achievable sparsity). Figure 6-4b compares sparsity vs. eff_{esti} on partitions generated by METIS with random constraints. There is a clear correlation between sparsity and eff_{esti}. Thus, partitioning a

graph to maximize $sparsity$ should maximize the eff_{esti} objective, which should in turn maximize actual walltime efficiency.



(a) Comparison of eff_{esti} for various work distribution methods



(b) *sparsity* vs. eff_{esti}

Figure 6-4: Figure 6-4a compares eff_{esti} when communication cost $c = 1.0$, the number of processes $p = 48$. **Bold** indicates that GRAZHDA*/sparsity has the best eff_{esti} (except for IdealApprox). Figure 6-4b compares sparsity vs. eff_{esti}. For each instance, we generated 3 different partitions using METIS with load balancing constraints which force METIS to balance randomly selected nodes, to see how degraded sparsity affects eff_{esti}. There was no partition with $eff_{esti} < 0.84$.

6.2.3 Partitioning the DTGs

Given an objective function such as sparsity, GRAZHDA* partitions each DTG into two abstract features, as described above in Section 6. Since each domain transition graph typically only has fewer than 10 nodes, we compute the optimal partition for both objective functions with a straightforward depth-first branch-and-bound procedure. It is possible that

branch-and-bound becomes impractical in case a domain has very large DTGs, or we may develop a more complicated objective function for partitioning the DTGs. In such cases, we can use heuristic partitioning methods such as the FM algorithm (Fiduccia & Mattheyses, 1982). However, to date, branch-and-bound has been sufficient – in all of the standard IPC benchmark domains we evaluated, the abstract feature generation procedure (which includes partitioning all of the DTGs) take less than 4 seconds on every instance we tested (most instances take < 1 second).

6.3 Evaluation of Automated, Domain-Independent Work Distribution Methods

In addition to the methods in Section 2.1, we evaluated the performance of GRAZHDA*/sparsity. We used CGL-B (CausalGraph-Goal-Level&Bisimulation) merge&shrink heuristic (Helmert et al., 2014), which is more efficient and recently proposed than LFPA merge&shrink (Helmert, Haslum, & Hoffmann, 2007) used in a previous conference paper which evaluated GAZHDA* and FAZHDA* (Jinnai & Fukunaga, 2016b). For example in Block10-1, CGL-B expands 11,065,451 nodes while LFPA 51,781,104 expands nodes. We set the abstraction size for merge&shrink to 1000. The choice of heuristic affects the behavior of parallel search if the heuristics have different node expansion rate, because it affects the relative cost of communication. As CGL-B and LFPA have roughly the same node expansion rate, we did not observe a significant difference on the effect of work distribution methods. Therefore, we show the result using CGL-B because it runs faster on sequential A*. We discuss the effect of node expansion rate in Section 3.4. We did not apply fluency-based filtering (Section 2.2) and used all DTGs in GRAZHDA*/sparsity because it did not improve the performance.

Figure 6-4a shows eff_{esti} for the various work distribution methods, including GRAZHDA* (see Section 2.1 for experimental setup and list of methods included in comparison). To evaluate how these methods compare to an ideal (but impractical) model which actually

Table 6.1: Comparison of eff_{actual} and eff_{esti} on a commodity cluster with 6 nodes, 48 processes. eff_{esti} (eff_{actual}) with **bold** font indicates the method has the best eff_{esti} (eff_{actual}). Instance name with **bold** indicates that the best eff_{esti} method has the best eff_{actual}. Speedup, CO, SO on experimental run are shown in Table 6.2.

Instance	A*		GRAZHDA*/ sparsity $[Z, A_{feauture}/DTG_{sparsity}]$		FAZHDA* $[Z, A_{feauture}/DTG_{fluency}]$	
	time	expd	eff_{actual}	eff_{esti}	eff_{actual}	eff_{esti}
Blocks10-0	129.29	11065451	**0.57**	**0.57**	0.54	0.43
Blocks11-1	813.86	52736900	**0.71**	**0.53**	0.71	0.50
Elevators08-5	165.22	7620122	0.34	0.51	0.26	0.49
Elevators08-6	453.21	18632725	0.45	0.50	0.38	0.36
Gripper8	517.41	50068801	0.56	0.60	**0.57**	**0.63**
Logistics00-10-1	559.45	38720710	**0.94**	**0.70**	0.91	0.61
Miconic11-0	232.07	12704945	0.87	0.95	0.88	0.91
Miconic11-2	262.01	14188388	**0.94**	**0.97**	0.93	0.92
NoMprime5	309.14	4160871	**0.50**	**0.58**	0.48	0.53
NoMystery10	179.52	1372207	**0.72**	0.61	0.48	**0.75**
Openstacks08-19	282.45	15116713	0.51	0.59	0.42	0.58
Openstacks08-21	554.63	19901601	0.53	0.65	0.52	0.62
Parcprinter11-11	307.19	6587422	**0.42**	**0.54**	0.27	0.49
Parking11-5	237.05	2940453	**0.62**	0.55	0.62	0.54
Pegsol11-18	801.37	106473019	0.44	**0.72**	0.44	0.71
PipesNoTk10	157.31	2991859	**0.33**	**0.52**	0.33	0.49
PipesTk12	321.55	15990349	0.70	**0.66**	**0.83**	0.65
PipesTk17	356.14	18046744	0.92	**0.65**	**0.94**	0.63
Rovers6	1042.69	36787877	**0.86**	**0.79**	0.84	0.72
Scanalyzer08-6	195.49	10202667	**0.69**	**0.92**	0.63	0.86
Scanalyzer11-6	152.92	6404098	**0.91**	**0.78**	0.57	0.63
Average	382.38	21557805	**0.64**	**0.62**	0.60	0.61

Instance	GAZHDA* $[Z, A_{feauture}/DTG_{greedy}]$		OZHDA* $[Z_{operator}]$		DAHDA* $[Z, A_{state}/SDD_{dynamic}]$		ZHDA* $[Z]$	
	eff_{actual}	eff_{esti}	eff_{actual}	eff_{esti}	eff_{actual}	eff_{esti}	eff_{actual}	eff_{esti}
Blocks10-0	0.45	0.44	0.32	0.37	0.52	0.47	0.31	0.48
Blocks11-1	0.61	0.48	0.61	0.47	0.52	0.43	0.58	0.48
Elevators08-5	**0.61**	0.58	0.46	**0.64**	0.57	0.51	0.57	0.47
Elevators08-6	**0.72**	**0.76**	0.68	0.56	0.32	0.39	0.38	0.49
Gripper8	0.46	0.50	0.52	0.44	0.45	0.45	0.45	0.47
Logistics00-10-1	0.24	0.42	0.24	0.43	0.36	0.53	0.34	0.48
Miconic11-0	0.27	0.53	0.79	**0.96**	**0.96**	0.91	0.15	0.48
Miconic11-2	0.18	0.37	0.77	0.90	0.70	0.81	0.31	0.48
NoMprime5	0.39	0.48	0.35	0.51	0.38	0.49	0.35	0.47
NoMystery10	0.40	0.66	0.45	0.50	0.59	0.60	0.45	0.49
Openstacks08-19	0.46	0.58	0.36	0.55	0.51	**0.66**	**0.54**	0.47
Openstacks08-21	0.53	0.65	**0.82**	0.49	0.56	**0.68**	0.81	0.51
Parcprinter11-11	0.35	0.40	0.33	0.34	0.15	0.15	0.40	0.48
Parking11-5	0.59	0.49	0.56	0.46	0.60	**0.59**	0.56	0.47
Pegsol11-18	0.34	0.53	**0.55**	0.71	0.46	0.70	0.35	0.47
PipesNoTk10	0.32	0.50	0.32	0.48	0.32	0.48	0.07	0.48
PipesTk12	0.41	0.48	0.45	0.49	0.52	0.57	0.41	0.48
PipesTk17	0.56	0.50	0.60	0.52	0.65	0.60	0.55	0.49
Rovers6	0.70	0.61	0.85	0.71	0.53	0.73	0.63	0.53
Scanalyzer08-6	0.42	0.54	0.49	0.58	0.44	0.51	0.34	0.48
Scanalyzer11-6	0.34	0.41	0.81	0.68	0.41	0.44	0.42	0.48
Average	0.45	0.51	0.54	0.53	0.50	0.47	0.43	0.49

Table 6.2: Comparison of average speedups, communication/search overhead (CO, SO) on 10 runs on a commodity cluster with 6 nodes, 48 processes using merge&shrink heuristic. The results with standard deviation are shown in appendix.

Instance	A*		GRAZHDA*/ sparsity $[Z, A_{feauture}/DTG_{sparsity}]$			FAZHDA* $[Z, A_{feauture}/DTG_{fluency}]$		
	expd	time	speedup	CO	SO	speedup	CO	SO
Blocks10-0	129.29	11065451	**27.17**	0.28	0.38	26.02	0.70	0.35
Blocks11-1	813.86	52736900	**34.25**	0.66	0.15	34.25	0.66	0.15
Elevators08-5	165.22	7620122	16.43	0.47	0.33	12.34	0.32	0.51
Elevators08-6	453.21	18632725	21.47	0.49	0.37	18.05	0.52	0.81
Gripper8	517.41	50068801	26.67	0.50	0.15	**27.45**	0.43	0.10
Logistics00-10-1	559.45	38720710	**45.16**	0.43	0.01	43.85	0.57	0.02
Miconic11-0	232.07	12704945	41.97	0.01	0.07	42.43	0.01	0.06
Miconic11-2	262.01	14188388	**45.26**	0.01	0.05	44.87	0.01	0.05
NoMprime5	309.14	4160871	**23.95**	0.80	-0.04	22.87	0.79	-0.05
NoMystery10	179.52	1372207	**34.80**	0.51	0.12	22.99	0.24	-0.44
Openstacks08-19	282.45	15116713	24.67	0.27	0.34	20.00	0.24	0.37
Openstacks08-21	554.63	19901601	25.23	0.17	0.35	24.97	0.15	0.35
Parcprinter11-11	307.19	6587422	**20.26**	0.26	0.55	13.08	0.26	0.61
Parking11-5	237.05	2940453	**29.75**	0.40	0.34	29.67	0.63	0.11
Pegsol11-18	801.37	106473019	21.03	0.39	0.02	20.97	0.39	0.00
PipesNoTk10	157.31	2991859	**15.73**	0.98	0.01	15.64	0.98	0.01
PipesTk12	321.55	15990349	33.78	0.46	0.05	**39.65**	0.46	0.03
PipesTk17	356.14	18046744	43.92	0.54	0.01	**45.03**	0.54	0.01
Rovers6	1042.69	36787877	**41.17**	0.15	0.14	40.48	0.15	0.17
Scanalyzer08-6	195.49	10202667	**32.92**	0.12	0.01	30.31	0.12	0.01
Scanalyzer11-6	152.92	6404098	**43.83**	0.16	0.13	27.31	0.18	0.34
Average	382.38	21557805	**30.92**	0.38	0.17	28.68	0.40	0.17
Total walltime	8029.97	452713922			**277.91**			301.38

Instance	GAZHDA* $[Z, A_{feauture}/DTG_{greedy}]$			OZHDA* $[Z_{operator}]$			DAHDA* $[Z, A_{state}/SDD_{dynamic}]$			ZHDA* $[Z]$		
	speedup	CO	SO	speedup	CO	SO	speedup	CO	SO	speedup	CO	SO
Blocks10-0	21.81	0.99	0.12	15.47	0.98	0.34	25.11	0.88	0.08	14.93	0.98	0.30
Blocks11-1	29.20	0.99	0.03	29.20	0.99	0.03	24.88	0.91	0.21	27.98	0.98	0.07
Elevators08-5	**29.35**	0.65	-0.00	21.86	0.09	0.44	27.59	0.83	-0.03	27.54	0.98	-0.03
Elevators08-6	**34.52**	0.24	-0.09	32.70	0.41	0.22	15.28	0.88	0.31	18.19	0.96	0.06
Gripper8	21.86	0.81	0.06	24.77	0.98	0.14	21.80	0.98	0.08	21.66	0.98	0.08
Logistics00-10-1	11.68	0.85	0.25	11.68	0.85	0.25	17.52	0.84	0.00	16.09	0.99	0.00
Miconic11-0	13.15	0.53	0.24	37.86	0.02	0.02	**46.05**	0.01	0.08	7.40	0.96	0.13
Miconic11-2	8.53	0.53	0.74	36.86	0.02	0.07	33.81	0.01	0.18	14.67	0.96	0.05
NoMprime5	18.55	0.95	-0.06	16.66	0.94	0.00	18.46	0.90	-0.05	16.63	0.98	-0.02
NoMystery10	18.98	0.42	-0.07	21.61	0.74	0.11	28.41	0.60	-0.07	21.68	0.99	-0.07
Openstacks08-19	22.14	0.38	0.21	17.11	0.34	0.32	24.54	0.24	0.18	**25.99**	0.99	-0.05
Openstacks08-21	25.67	0.15	0.31	**39.34**	0.92	0.05	26.72	0.13	0.28	39.06	0.92	-0.00
Parcprinter11-11	16.85	0.74	0.41	15.98	0.82	0.56	7.00	0.19	4.38	19.15	0.97	0.08
Parking11-5	28.43	0.98	0.02	26.76	0.97	0.07	28.84	0.52	0.07	27.09	0.98	0.04
Pegsol11-18	16.22	0.77	0.05	**26.17**	0.34	-0.03	22.16	0.34	-0.01	16.97	0.98	0.03
PipesNoTk10	15.58	0.98	0.01	15.22	0.98	0.02	15.58	0.98	0.01	3.22	0.98	-0.44
PipesTk12	19.84	0.99	0.01	21.40	0.88	0.04	25.12	0.67	0.00	19.78	0.98	0.00
PipesTk17	26.64	0.98	0.00	28.82	0.88	0.00	31.16	0.60	0.01	26.27	0.98	0.00
Rovers6	33.49	0.56	0.01	41.00	0.31	0.03	25.48	0.05	0.26	30.01	0.76	0.00
Scanalyzer08-6	20.28	0.77	0.01	23.70	0.66	0.01	21.23	0.94	0.00	16.54	0.98	0.01
Scanalyzer11-6	16.36	0.65	0.49	38.82	0.30	0.09	19.51	0.50	0.46	20.36	0.98	0.05
Average	21.39	0.71	0.13	25.86	0.64	0.13	24.11	0.57	0.31	20.53	0.96	0.01
Total walltime			398.75			331.18			377.86			433.23

applies graph partitioning to the entire search space (instead of partitioning DTG as done by GRAZHDA*), we also evaluated *IdealApprox*, a model which partitions the entire state space graph using the METIS (approximate) graph partitioner (Karypis & Kumar, 1998). IdealApprox first enumerates a graph containing all nodes with $f \leq f^*$ and edges between these nodes and ran METIS with the sparsity objective (Equation 6.1) to generate the partition for the work distribution. Generating the input graph for METIS takes an enormous amount of time (much longer than the search itself), so IdealApprox is clearly an impractical model, but it provides a useful approximation for an ideal work distribution which can be used to evaluate practical methods.

Not surprisingly, IdealApprox has the highest eff_{esti}, but among all of the practical methods, GRAZHDA*/sparsity has the highest eff_{esti} overall. As we saw in Section 2.1 that eff_{esti} is a good estimate of actual efficiency, the result suggest that GRAZHDA*/sparsity outperforms other methods. In fact, as shown in Table 6.1 and 6.2, GRAZHDA*/sparsity achieved a good balance between CO and SO and had the highest actual speedup overall, significantly outperforming all other previous methods. Note that as IdealApprox is only an approximation of the sparsest-cut, other methods can sometimes achieve better eff_{esti}.

6.3.1 The effect of the number of cores on speedup

Figure 6-5 shows the speedup of the algorithms as the number of cores increased from 8 to 48. GRAZHDA*/sparsity outperformed consistently outperformed the other methods. The performance gap between the better methods (GRAZHDA*/sparsity, FAZHDA*, OZHDA*, DAHDA*) and the baseline ZHDA* increases with the number of the cores. This is because as the number of cores increases, communications overheads increases with the number of cores, and our new work distribution method successfully mitigates communications overhead.

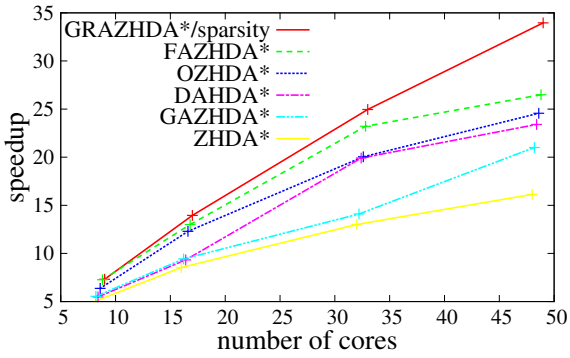


Figure 6-5: Speedup of HDA* variants (average over all instances in Table 6.2. Results are for 1 node (8 cores), 2 nodes (16 cores), 4 nodes (32 cores) and 6 nodes (48 cores).

6.3.2 Cloud Environment Results

In addition to the 48 core cluster, we evaluated GRAZHDA*/sparsity on an Amazon EC2 cloud cluster with 128 virtual cores (vCPUs) and 480GB aggregated RAM (a cluster of 32 m1.xlarge EC2 instances, each with 4 vCPUs, 3.75 GB RAM/core. This is a less favorable environment for parallel search compared to a "bare-metal" cluster because physical processors are shared with other users and network performance is inconsistent (Iosup, Ostermann, Yigitbasi, Prodan, Fahringer, & Epema, 2011). We intentionally chose this configuration to evaluate work distribution methods in an environment which is significantly different from our other experiments. Table 6.3 shows that as with the smaller-scale cluster results, GRAZHDA*/sparsity outperformed other methods in this large-scale cloud environment.

6.3.3 24-Puzzle Experiments

We evaluated GRAZHDA*/sparsity on the 24-puzzle using the same configuration as Section 1.2. Abstract feature generated by GRAZHDA*/sparsity is shown in Figure 6-6d. We compared GRAZHDA*/sparsity (automated abstract feature generation) vs. AZHDA* with the hand-crafted work distribution ($HDA^*[Z, A_{feature}]$) (Figure 4-2d) and $HDA^*[Z]$. With 8 cores, the speedups were 7.84 (GRAZHDA*/sparsity), 7.85 ($HDA^*[Z, A_{feature}]$), and 5.95

Table 6.3: Comparison of walltime, communication/search overhead (CO, SO) on a cloud cluster (EC2) with 128 virtual cores (32 m1.xlarge EC2 instances) using the merge&shrink heuristic. We run sequential A* on a different machine with 128 GB memory because some of the instances cannot be solved by A* on a single m1.xlarge instance due to memory limits. Therefore we report walltime instead of speedup.

Instance	A*	GRAZHDA*/sparsity $[Z, A_{feauture}/DTG_{sparsity}]$			FAZHDA* $[Z, A_{feauture}/DTG_{fluency}]$		
	expd	time	CO	SO	time	CO	SO
Airport18	48782782	102.34	0.59	0.49	**95.48**	0.59	0.29
Blocks11-0	28664755	**12.40**	0.42	0.37	22.86	0.68	0.53
Blocks11-1	45713730	**17.21**	0.42	0.25	32.60	0.66	0.82
Elevators08-7	74610558	**51.90**	0.54	0.25	121.90	0.55	0.26
Gripper9	243268770	**78.90**	0.42	0.01	82.90	0.43	0.06
Openstacks08-21	19901601	6.30	0.23	0.06	5.76	0.19	-0.05
Openstacks11-18	115632865	**33.10**	0.24	-0.14	33.25	0.23	-0.12
Pegsol08-29	287232276	58.85	0.44	0.16	81.75	0.42	0.55
PipesNoTk16	60116156	120.64	0.94	0.84	**106.28**	0.94	0.72
Trucks6	19109329	**8.01**	0.17	0.46	51.51	0.19	0.34
Average	99361115	**43.03**	0.42	0.25	59.87	0.48	0.39
Total walltime	894250040			**387.31**			538.81

Instance	GAZHDA* $[Z, A_{feauture}/DTG_{greedy}]$			OZHDA* $[Z_{operator}]$			DAHDA* $[Z, A_{state}/SDD_{dynamic}]$			ZHDA* $[Z]$		
	time	CO	SO	time	CO	SO	time	CO	SO	time	CO	SO
Airport18	128.22	0.98	0.02	123.09	0.90	0.56	143.27	0.92	0.36	106.80	0.99	0.02
Blocks11-0	21.75	0.98	0.65	21.70	0.99	0.70	20.29	0.95	0.88	29.19	0.99	0.35
Blocks11-1	25.84	0.98	0.56	24.84	0.86	0.78	29.52	0.94	0.83	36.04	1.00	0.52
Elevators08-7	61.16	0.70	0.05	86.65	0.07	0.22	52.09	0.96	0.18	59.88	1.00	0.04
Gripper9	85.98	1.00	0.16	90.98	0.98	0.20	95.72	1.00	0.15	105.78	1.00	0.17
Openstacks08-21	**5.67**	0.71	-0.35	40.06	0.96	0.00	6.94	0.69	-0.17	14.65	1.00	-0.09
Openstacks11-18	71.34	0.77	-0.09	79.34	0.81	-0.00	84.67	0.76	0.01	49.97	1.00	-0.53
Pegsol08-29	98.53	0.98	0.06	**54.13**	0.34	0.13	108.17	1.00	0.11	120.27	0.98	0.16
PipesNoTk16	108.28	0.95	0.78	120.21	0.99	0.73	125.37	1.00	0.72	149.96	1.00	0.73
Trucks6	30.22	0.94	0.41	32.22	0.96	0.57	17.19	0.53	0.43	28.22	1.00	0.34
Average	56.53	0.89	0.29	61.13	0.77	0.41	60.00	0.87	0.36	66.00	1.00	0.29
Total walltime			508.77			550.13			539.96			593.96

($HDA^*[Z]$). Thus, the completely automated GRAZHDA*/sparsity is competitive with a carefully hand-designed work distribution method. For the 15-puzzle, the partition generated by GRAZHDA*/sparsity exactly corresponds to the hand-crafted hash function of Figure 4-2b, so the performance is identical.

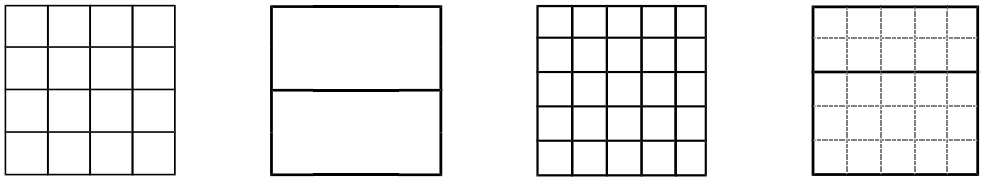
Table 6.4: Comparison of speedups, communication/search overheads (CO, SO) using expensive heuristic (LM-cut).

(a) Single multicore machine (8 cores)

Instance	A*		GRAZHDA*/sparsity $[Z, A_{feauture}/DTG_{sparsity}]$			DAHDA* $[Z, A_{state}/SDD_{dynamic}]$			ZHDA* $[Z]$		
	time	expd	speedup	CO	SO	speedup	CO	SO	speedup	CO	SO
Blocks14-1	351.03	191948	**5.53**	0.33	0.19	2.08	0.30	1.82	5.40	0.90	0.05
Elevators08-7	1182.92	1465914	3.47	0.48	1.70	3.30	0.73	0.04	**3.75**	0.88	0.00
Elevators08-8	742.65	344304	**7.19**	0.41	0.06	4.80	0.72	0.03	5.65	0.82	0.00
Floortile11-4	1783.44	2876492	3.54	0.50	0.28	**4.03**	0.41	0.01	3.17	0.96	0.00
Gripper7	903.96	10082501	1.41	0.68	0.27	**2.60**	0.56	0.00	2.27	0.94	0.00
Openstacks08-15	707.31	11309809	**4.95**	0.32	-0.07	4.26	0.27	0.03	3.91	0.88	-0.04
Openstacks11-12	309.49	4250213	**4.59**	0.38	-0.00	4.40	0.29	-0.00	3.94	0.92	-0.01
Openstacks11-15	1187.58	13457961	4.04	0.36	0.10	**4.09**	0.28	0.01	3.59	0.89	0.00
PipesNoTk10	997.62	662717	2.65	0.86	0.00	2.10	0.96	0.01	**3.02**	0.89	0.00
PipesNoTk12	201.07	200502	4.36	0.84	0.00	4.65	0.47	0.09	**4.69**	0.90	0.00
PipesNoTk15	323.59	212678	4.61	0.85	0.00	3.83	0.57	0.22	**4.91**	0.89	0.01
PipesTk11	572.00	382587	**6.45**	0.37	0.01	3.57	0.64	0.00	3.69	0.86	0.00
Scanalyzer11-6	1149.31	699932	**5.89**	0.13	-0.01	3.14	0.44	-0.00	2.75	0.88	-0.00
Storage15	330.79	155979	4.70	0.70	0.04	4.67	0.68	0.01	**4.95**	0.85	0.00
Trucks9	199.02	65531	**7.38**	0.05	-0.04	3.72	0.06	0.42	3.40	0.87	-0.01
Trucks10	800.02	384585	**6.85**	0.04	0.01	4.42	0.04	0.15	2.03	0.91	0.03
Visitall11-7half	181.05	519064	**6.59**	0.14	0.24	5.62	0.16	0.15	6.09	0.87	0.00
Woodwrk11-6	283.73	172077	**7.10**	0.39	-0.00	5.97	0.27	-0.00	3.25	0.94	-0.00
Average	678.14	2635266	**5.07**	0.43	0.15	3.96	0.44	0.17	3.91	0.89	0.00
Total walltime	12206.58	47434794			**3215.00**			3513.00			3711.51

(b) Commodity Cluster with 6 nodes (48 cores)

Instance	A*		GRAZHDA*/sparsity $[Z, A_{feauture}/DTG_{sparsity}]$			DAHDA* $[Z, A_{state}/SDD_{dynamic}]$			ZHDA* $[Z]$		
	time	expd	speedup	CO	SO	speedup	CO	SO	speedup	CO	SO
Blocks14-1	351.03	191948	**22.86**	0.34	0.50	20.22	0.32	0.65	16.79	0.98	0.18
Elevators08-7	1182.92	1465914	18.17	0.53	0.36	**22.25**	0.81	0.07	20.91	0.97	0.02
Elevators08-8	742.65	344304	25.58	0.45	0.51	30.78	0.84	0.10	**31.43**	0.95	0.05
Floortile11-4	1783.44	2876492	18.25	0.99	0.09	**24.65**	0.46	0.10	21.56	0.99	0.02
Gripper7	903.96	10082501	12.59	0.66	0.02	**16.17**	0.61	0.07	12.59	0.99	0.01
Openstacks11-11	721.30	11309809	**43.09**	0.36	-0.43	10.19	0.25	1.22	20.75	0.99	-0.02
Openstacks11-15	1187.58	13457961	15.50	0.28	0.28	17.39	0.23	0.25	**19.53**	0.99	0.01
Parcprinter11-12	195.51	218595	**46.90**	0.04	0.02	44.14	0.05	0.01	18.02	0.99	0.24
PipesNoTk10	997.62	662717	**15.57**	0.98	0.01	14.80	0.99	0.01	15.38	0.98	0.01
PipesNoTk12	201.07	200502	26.05	0.89	0.28	**32.86**	0.52	0.34	22.03	0.98	0.30
PipesNoTk15	323.59	212678	**25.18**	0.94	0.18	19.54	0.62	0.72	21.11	0.98	0.39
PipesTk8	1141.00	145828	17.96	0.98	0.04	17.38	0.98	0.06	**18.99**	0.98	0.03
PipesTk11	572.00	382587	**30.35**	0.41	0.16	23.62	0.65	0.06	19.31	0.98	0.04
Scanalyzer11-06	1149.31	699932	**42.21**	0.13	0.04	20.18	0.49	0.01	15.45	0.98	0.00
Storage15	330.79	155979	22.50	0.88	0.22	**30.35**	0.74	0.09	24.15	0.96	0.19
Trucks9	199.02	65531	**24.82**	0.05	0.78	18.92	0.06	1.42	12.24	0.98	0.96
Trucks10	800.02	384585	17.61	0.05	0.60	**41.74**	0.04	0.25	12.53	1.00	0.04
Visitall11-07half	181.05	519064	12.97	0.16	2.59	12.88	0.17	2.60	**22.14**	0.98	0.58
Woodwrk08-7	819.62	33871	**36.12**	0.74	0.07	31.91	0.74	0.37	26.71	1.00	0.07
Woodwrk11-6	283.73	172077	**42.67**	0.42	0.07	21.38	0.29	0.03	16.81	0.99	0.05
Average	756.91	2527318	**26.06**	0.55	0.17	23.98	0.52	0.24	19.26	0.98	0.09
Total walltime	12867.52	42964409			**637.57**			646.66			709.89



(a) 15-puzzle $HDA^*[Z]$ (b) 15-puzzle GRAZHDA*/sparsity (c) 24-puzzle $HDA^*[Z]$ (d) 24-puzzle GRAZHDA*/sparsity

Figure 6-6: Abstract features generated by GRAZHDA*/sparsity ($HDA^*[Z, A_{feature}/DTG_{sparsity}]$) for 15-puzzle and 24-puzzle. Abstract features generated on 15-puzzle exactly corresponds to the hand-crafted hash function of Figure 4-2b.

6.3.4 Evaluation of Parallel Search Overheads and Performance in Low Communications-Cost Environments

In previous experiments, we compared work distribution functions using domain-specific solvers with very fast node generation rates (Section 1), as well as domain-independent planning using a fast heuristic function (Section 3). Next, we evaluate search overheads and performance when node generation rates are low due to expensive node evaluations. In such domains, the impact of communications overheads is minimal because overheads for queue insertion, buffering, etc. are negligible compared to the computation costs associated with node generation and evaluation. As a consequence, search overhead is the dominant factor which determines search performance.

In particular, we evaluate different parallel work distribution strategies when applied to domain-independent planning using the landmark-cut (LM-cut) heuristic, a state-of-the-art heuristic which is a relatively expensive heuristic. While there is no dominance relationship among planners using cheap heuristics such as merge&shrink heuristics (which require only a table lookup during search) and expensive heuristics such as LM-cut, recent work in forward-search based planning has focused on heuristics which tend to be slow, such as heuristics that require the solution of a linear program at every search node (Pommerening, Röger, Helmert, & Bonet, 2014; Imai & Fukunaga, 2015), so parallel strategies that focus on minimizing search overheads is of practical importance. Previous evaluations of parallel

work distribution strategies in domain-independent planning used relatively fast heuristics. Kishimoto et al. (2013), as well as Jinnai and Fukunaga (2016a, 2016b) used merge&shrink abstraction based heuristics. Zhou and Hansen (2007) and Burns et al. (2010)used the max-pair heuristic (Haslum & Geffner, 2000).Thus, this is the first evaluation of parallel forward search for domain-independent planning using an expensive heuristic.

To evaluate the effect of SO and CO with the LM-cut heuristic, we compared the performance of ZHDA*, DAHDA*, and GRAZHDA*/sparsity as representatives of methods which optimize SO, CO, and both SO and CO, respectively. The instances used for this experiment are different from the experiments using merge&shrink (Table 6.2), because some of the instances used for the merge&shrink experiments were too easy to solve with LM-cut and not suitable for evaluating parallel algorithms. The average node expansion rate by sequential A* on the selected instances was 3886.02 node/sec. Compared to the expansion rate with merge&shrink heuristic used in Section 3 (56378.03 node/sec), the expansion rate is 14.5 times slower. Therefore, the relative cost of communication is expected to be smaller with LM-cut than merge&shrink heuristic.

Table 6.4a shows the results on a single multicore machine with 8 cores. Overall, GRAZHDA*/sparsity outperformed ZHDA* and DAHDA*. Interestingly, although GRAZHDA*/sparsity has higher SO, it was still faster than ZHDA* because of lower CO. Even with this low communication cost environment, CO continues to be one of the major overhead for HDA*.

Table 6.4b shows the results on a commodity cluster with 48 cores. As in the multicore environment, GRAZHDA*/sparsity outperformed ZHDA* and DAHDA*. However, the relative speedup of ZHDA* to GRAZHDA*/sparsity is higher with LM-cut (0.75) than with merge&shrink (0.66) (note that we used different instance set, so it may due to other factors). Some of the instances (trucks9, visitall11-07-half) are too easy for a distributed environment, and therefore on these instances, high SO is incurred due to the burst effect (Section 1.2). Therefore, some of the instances have high SO even in ZHDA* where good LB is achieved.

Chapter 7

Conclusions

We investigated node distribution methods for HDA*, and showed that previous methods suffered from high communication overhead ($HDA^*[Z]$), high search overhead ($HDA^*[P, A_{state}]$), or both ($HDA^*[P]$), which limited their efficiency. We proposed Abstract Zobrist hashing, a new distribution method which combines the strengths of both Zobrist hashing and abstraction, and AZHDA* ($HDA^*[Z, A_{feature}]$), a new variant of HDA* which is based on AZH. Our experimental results showed that AZHDA* achieves a successful trade-off between communication and search overheads, resulting in better performance than previous work distribution methods with hand-crafted abstract features.

We then extended the investigation to automated, domain-independent approaches for generate work distribution. We formulated work distribution as graph partitioning, and proposed and validated eff_{esti}, a model of search and communication overheads for HDA* which can be used to predict the actual walltime efficiency. We proposed and evaluated GRAZHDA*, a new top-down approach to work distribution for parallel best-first search in the HDA* framework which approximate the optimal graph partitioning by partitioning domain transition graphs according to an objective function such as sparsity.

We experimentally showed that GRAZHDA*/sparsity significantly improves both estimated efficiency (eff_{esti}) as well as the actual performance (walltime efficiency) compared to previous work distribution methods. Our results demonstrate the viability of approx-

imating the partitioning of the entire search space by applying graph partitioning to an abstraction of the state space (i.e., the DTG). While our results indicate that sparsity works well as a partitioning objective for GRAHZDA*, it is possible that a different objective function might yield better results, since DTG-partitioning is only an approximation to G_W partitioning. We have experimented with another objective MIN(CO+LB) objective, which minimizes $(CO + LB)$, and found that the performance is comparable to sparsity. Investigation of other objective functions is a direction for future work.

Despite significant improvements compared to previous work distribution approaches, there is room for improvement. The gap between the eff_{esti} metric for GRAZHDA* and an ideal model (IdealApprox) in Figure 6-4a represents the gap between actually partitioning the state space graph (as IdealApprox does) vs. the approximation obtained by the GRAZHDA* DTG partitioning. Closing this gap in eff_{esti} should lead to corresponding improvements in actual walltime efficiency, and poses challenges for future work. One possible approach to closing this gap is to partition a merged DTG which represents multiple SAS+ variables instead of partitioning a DTG of a single SAS+ variable. As merged DTGs have a richer representation of the state space graph, partitioning them using an objective function may result in a better approximation of the ideal partitioning. This approach is similar to merge-and-shrink heuristic (Helmert et al., 2014) which merging multiple DTGs into abstract state space to better estimate the state-space graph.

In this thesis, we assumed identical distance between each two cores. However, communication costs vary among pairs of processors in distributed search, especially in cloud cluster environments. Furthermore, as the number of cores scales to thousands or tens of thousands or more, some consideration of core locality is likely to be necessary. Incorporating the technique to distribute nodes considering the locality of processors such as LOHA&QE (Mahapatra & Dutt, 1997) may further improve the performance.

Implementing intra-node communications as interthread communication (OpenMP) is shown to improve the performance on a hash-based parallel suboptimal search (Vidal, Vernhes, & Infantes, 2012). The technique should also improve the performance of HDA*.

Dynamic adjustment of the partitioning on Structured Duplicate Detection has shown to be effective for external search (Zhou & Hansen, 2011). We may further improve the performance of HDA* by adjusting the hash function in the course of the search.

Finally, GPU-based massively parallel search has recently been shown to be successful (Zhou & Zeng, 2015). Investigation of tradeoffs between communication and search overhead in a heterogeneous algorithm which seeks to effectively utilize all normal cores as well as GPU cores using a framework based on abstract feature-based hashing is a direction for future work.

Appendix A. Dynamic AHDA* (DAHDA*), an improvement to AHDA* for distributed memory systems

This section presents an improvement to AHDA* (Burns et al., 2010). In our experiments, we used AHDA* as one of the baselines for evaluating our new AZHDA* strategies. The baseline implementation of AHDA* ($HDA^*[Z, A_{state}/SDD]$) is based on the greedy abstraction algorithm described in (Zhou & Hansen, 2006b), and selects a subset of DTGs (atom groups). The greedy abstraction algorithm adds one DTG to the abstract graph (G) at a time, choosing the DTG which minimizes the maximum out-degree of the abstract graph, until the graph size (# of nodes) reaches the threshold given by a parameter N_{max}. PSDD requires a N_{max} to be derived from the size of the available RAM. We found that AHDA* with a static N_{max} threshold as in PSDD performed poorly for a benchmark set with varying difficulty because a fixed size abstract graph results in very poor load balance. While poor load balance can lead to low efficiency and poor performance, a bad choice for N_{max} can be catastrophic when the system has a relatively small amount of RAM per core, as poor load balance causes concentrated memory usage in the overloaded processors, resulting in early memory exhaustion (i.e., AHDA* crashes because a thread/process which is allocated a large number of states exhausts its local heap).

The AHDA* results in Table 1 are for a 48-core cluster, 2GB/core, and uses $N_{max} = 10^2, 10^3, 10^4, 10^5, 10^6$ nodes based on Fast-Downward (Helmert, 2006) using merge&shrink

heuristic (Helmert et al., 2014). Smaller N_{max} results in lower CO, but when N_{max} is too small for the problem, load imbalance results in a concentration of the nodes and memory exhaustion. Although the total amount of RAM in current systems is growing, the amount of RAM per core has remained relatively small because the number of cores has also been increasing (and is expected to continue increasing). Thus, this is a significant issue with the straightforward implementation of AHDA* which uses a static N_{max}. To avoid this problem, N_{max} must be set dynamically according to the size of the state space for each instance. Thus, we implemented Dynamic AHDA* (DAHDA* = $HDA^*[Z, A_{state}/SDD_{dynamic}]$), which dynamically set the size of the abstract graph according to the number of DTGs (the state space size is exponential in the # of DTGs). We set the threshold of the total number of features in the DTGs to be 30% of the total number of features in the problem instance (we tested 10%, 30%, 50%, and 70% and found that 30% performed best). Note that the threshold is relative to the number of features, not the state space size as in AHDA*, which is exponential in the # features. Therefore, DAHDA* tries to take into account of certain amount of features, whereas AHDA* sometimes use only a fraction of features.

Table 1: Performance of AHDA* with fixed threshold (on 48 cores). Note that $|G| > |G'|$ does not indicate that all atom groups used in G are used in G'. DAHDA* limits the size of abstract graph according to the number of features in abstract graph, whereas AHDA* set maximum to N_{max}. Due to this difference, DAHDA* tends to end up with a different set of atom groups than AHDA*.

Instance	A*		DAHDA* $[Z, A_{state}/SDD_{dynamic}]$				AHDA* $[Z, A_{state}/SDD]$ $N_{max} = 100$		
	time	expd	speedup	CO	SO	$\|G\|$	speedup	CO	SO
Blocks10-0	129.29	11065451	**25.11**	0.88	0.08	14641	5.61	0.48	4.72
Blocks11-1	621.74	52736900	**24.88**	0.91	0.21	20736	memory exhaustion		
Elevators08-5	165.22	7620122	**27.59**	0.83	-0.03	1500	6.54	0.61	1.84
Elevators08-6	453.21	18632725	15.28	0.88	0.31	73125	memory exhaustion		
Gripper8	517.41	50068801	21.80	0.98	0.08	39366	16.84	0.39	0.58
Logistics00-10-1	559.45	38720710	**17.52**	0.84	0.00	140608	memory exhaustion		
Miconic11-0	232.07	12704945	**46.05**	0.01	0.08	2048	7.61	0.00	5.39
Nomprime5	309.14	4160871	18.46	0.90	-0.05	4194304	memory exhaustion		
Openstacks08-21	554.63	19901601	**26.72**	0.13	0.28	8388608	memory exhaustion		
PipesNoTk10	157.31	2991859	15.58	0.98	0.01	32768	5.89	0.17	1.15
Scanalyzer08-6	195.49	10202667	21.23	0.94	0.00	16384	**40.15**	0.02	-0.07

	AHDA* $[Z, A_{state}/SDD]$											
	$N_{max} = 1000$			$N_{max} = 10000$			$N_{max} = 100000$			$N_{max} = 1000000$		
	speedup	CO	SO	speedup	CO	SO	speedup	CO	SO	speedup	CO	SO
Blocks10-0	memory exhaustion			18.24	0.39	0.29	16.72	0.47	0.25	14.77	0.54	0.24
Blocks11-1	memory exhaustion			memory exhaustion			21.38	0.65	0.12	15.38	0.68	0.10
Elevators08-5	18.02	0.79	0.65	19.30	0.87	0.44	18.20	0.90	0.37	18.84	0.92	0.35
Elevators08-6	17.86	0.67	0.50	18.38	0.86	0.23	16.99	0.91	0.09	**22.66**	0.89	-0.02
Gripper8	memory exhaustion			**30.17**	0.53	0.31	25.31	0.65	0.21	24.65	0.70	0.16
Logistics00-10-1	memory exhaustion			memory exhaustion			memory exhaustion			memory exhaustion		
Miconic11-0	6.75	0.00	5.60	26.90	0.01	0.19	26.22	0.01	0.25	25.77	0.02	0.40
Nomprime5	18.28	0.31	0.07	16.47	0.43	0.10	**19.92**	0.58	0.01	17.07	0.60	0.00
Openstacks08-21	memory exhaustion			memory exhaustion			memory exhaustion			memory exhaustion		
PipesNoTk10	18.38	0.30	0.10	**21.74**	0.43	0.04	18.50	0.56	0.03	14.36	0.64	0.03
Scanalyzer08-6	38.11	0.03	-0.03	39.26	0.26	-0.07	30.17	0.47	-0.07	26.46	0.64	-0.07

Appendix B. Experimental results with standard deviations

Table 9: Comparison of speedups, communication/search overhead (CO, SO) and their standard deviations on a commodity cluster with 6 nodes, 48 processes using merge&shrink heuristic (Extension of Table 6.2).

Instance	A*		GRAZHDA*/sparsity $[Z, A_{feauture}/DTG_{sparsity}]$			FAZHDA* $[Z, A_{feauture}/DTG_{fluency}]$		
	time	expd	speedup	CO	SO	speedup	CO	SO
Blocks10-0	129.29	11065451	**27.17** (4.11)	0.28 (0.02)	0.38 (0.41)	26.02 (0.74)	0.70 (0.00)	0.35 (0.04)
Blocks11-1	813.86	52736900	**34.25** (3.54)	0.66 (0.00)	0.15 (0.13)	34.25 (0.64)	0.66 (0.00)	0.15 (0.03)
Elevators08-5	165.22	7620122	16.43 (2.81)	0.47 (0.01)	0.33 (0.06)	12.34 (0.24)	0.32 (0.00)	0.51 (0.01)
Elevators08-6	453.21	18632725	21.47 (0.90)	0.49 (0.00)	0.37 (0.04)	18.05 (0.61)	0.52 (0.00)	0.81 (0.09)
Gripper8	517.41	50068801	26.67 (0.75)	0.50 (0.00)	0.15 (0.08)	**27.45** (0.73)	0.43 (0.00)	0.10 (0.12)
Logistics00-10-1	559.45	38720710	**45.16** (3.28)	0.43 (0.00)	0.01 (0.03)	43.85 (3.05)	0.57 (0.00)	0.02 (0.00)
Miconic11-0	232.07	12704945	41.97 (0.54)	0.01 (0.00)	0.07 (0.01)	42.43 (0.57)	0.01 (0.00)	0.06 (0.01)
Miconic11-2	262.01	14188388	**45.26** (0.60)	0.01 (0.00)	0.05 (0.00)	44.87 (1.18)	0.01 (0.00)	0.05 (0.01)
NoMprime5	309.14	4160871	**23.95** (0.85)	0.80 (0.00)	-0.04 (0.02)	22.87 (2.98)	0.79 (0.00)	-0.05 (0.03)
Nomystery10	179.52	1372207	**34.80** (0.87)	0.51 (0.00)	0.12 (0.03)	22.99 (4.55)	0.24 (0.00)	-0.44 (0.10)
Openstacks08-19	282.45	15116713	24.67 (1.25)	0.27 (0.01)	0.34 (0.05)	20.00 (0.86)	0.24 (0.00)	0.37 (0.04)
Openstacks08-21	554.63	19901601	25.23 (0.51)	0.17 (0.00)	0.35 (0.03)	24.97 (0.42)	0.15 (0.00)	0.35 (0.02)
Parcprinter11-11	307.19	6587422	**20.26** (0.93)	0.26 (0.00)	0.55 (0.29)	13.08 (4.09)	0.26 (0.03)	0.61 (0.67)
Parking11	237.05	2940453	**29.75** (0.48)	0.40 (0.00)	0.34 (0.01)	29.67 (4.12)	0.63 (0.00)	0.11 (0.10)
Pegsol11-18	801.37	106473019	21.03 (0.65)	0.39 (0.00)	0.02 (0.01)	20.97 (0.21)	0.39 (0.00)	0.00 (0.01)
PipesNoTk10	157.31	2991859	**15.73** (0.38)	0.98 (0.00)	0.01 (0.00)	15.64 (0.35)	0.98 (0.00)	0.01 (0.00)
PipesTk12	321.55	15990349	33.78 (4.22)	0.46 (0.00)	0.05 (0.01)	**39.65** (2.65)	0.46 (0.00)	0.03 (0.01)
PipesTk17	356.14	18046744	43.92 (2.69)	0.54 (0.00)	0.01 (0.00)	**45.03** (3.81)	0.54 (0.00)	0.01 (0.00)
Rovers6	1042.69	36787877	**41.17** (2.51)	0.15 (0.00)	0.14 (0.09)	40.48 (1.40)	0.15 (0.00)	0.17 (0.04)
Scanalyzer08-6	195.49	10202667	**32.92** (0.74)	0.12 (0.00)	0.01 (0.00)	30.31 (0.56)	0.12 (0.00)	0.01 (0.00)
Scanalyzer11-6	152.92	6404098	**43.83** (0.54)	0.16 (0.00)	0.13 (0.00)	27.31 (1.68)	0.18 (0.00)	0.34 (0.05)
Average	382.38	21557805	**30.92** (1.58)	0.38 (0.00)	0.17 (0.06)	28.68 (1.69)	0.40 (0.00)	0.17 (0.07)
Total walltime	8029.97	452713922			**277.91** (14.20)			301.38 (17.65)

Cont. Table 9.

Instance	GAZHDA* $[Z, A_{feauture}/DTG_{greedy}]$			OZHDA* $[Z_{operator}]$		
	speedup	CO	SO	speedup	CO	SO
Blocks10-0	21.81 (3.26)	0.99 (0.00)	0.12 (0.30)	15.47 (4.37)	0.98 (0.00)	0.34 (0.34)
Blocks11-1	29.20 (3.22)	0.99 (0.00)	0.03 (0.16)	29.20 (4.99)	0.99 (0.00)	0.03 (0.21)
Elevators08-5	**29.35** (2.77)	0.65 (0.04)	-0.00 (0.36)	21.86 (0.47)	0.09 (0.00)	0.44 (0.03)
Elevators08-6	**34.52** (4.09)	0.24 (0.00)	-0.09 (0.00)	32.70 (2.96)	0.41 (0.00)	0.22 (0.03)
Gripper8	21.86 (0.58)	0.81 (0.00)	0.06 (0.02)	24.77 (3.56)	0.98 (0.04)	0.14 (0.00)
Logistics00-10-1	11.68 (0.95)	0.85 (0.00)	0.25 (0.00)	11.68 (2.14)	0.85 (0.00)	0.25 (0.05)
Miconic11-0	13.15 (3.27)	0.53 (0.00)	0.24 (0.16)	37.86 (0.81)	0.02 (0.00)	0.02 (0.02)
Miconic11-2	8.53 (0.97)	0.53 (0.00)	0.74 (0.16)	36.86 (0.65)	0.02 (0.00)	0.07 (0.01)
NoMprime5	18.55 (0.69)	0.95 (0.00)	-0.06 (0.01)	16.66 (0.44)	0.94 (0.00)	0.00 (0.02)
Nomystery10	18.98 (4.04)	0.42 (0.00)	-0.07 (0.06)	21.61 (1.44)	0.74 (0.00)	0.11 (0.04)
Openstacks08-19	22.14 (1.19)	0.38 (0.01)	0.21 (0.05)	17.11 (1.28)	0.34 (0.00)	0.32 (0.13)
Openstacks08-21	25.67 (0.82)	0.15 (0.00)	0.31 (0.04)	**39.34** (0.52)	0.92 (0.00)	0.05 (0.11)
Parcprinter11-11	16.85 (2.71)	0.74 (0.00)	0.41 (0.49)	15.98 (1.44)	0.82 (0.00)	0.56 (0.03)
Parking11	28.43 (1.01)	0.98 (0.00)	0.02 (0.03)	26.76 (3.07)	0.97 (0.00)	0.07 (0.14)
Pegsol11-18	16.22 (0.27)	0.77 (0.00)	0.05 (0.01)	**26.17** (0.26)	0.34 (0.00)	-0.03 (0.00)
PipesNoTk10	15.58 (0.36)	0.98 (0.00)	0.01 (0.00)	15.22 (0.35)	0.98 (0.00)	0.02 (0.00)
PipesTk12	19.84 (3.18)	0.99 (0.01)	0.01 (0.00)	21.40 (0.94)	0.88 (0.00)	0.04 (0.02)
PipesTk17	26.64 (0.20)	0.98 (0.00)	0.00 (0.00)	28.82 (0.13)	0.88 (0.00)	0.00 (0.00)
Rovers6	33.49 (1.01)	0.56 (0.00)	0.01 (0.02)	41.00 (2.13)	0.31 (0.00)	0.03 (0.02)
Scanalyzer08-6	20.28 (2.22)	0.77 (0.00)	0.01 (0.00)	23.70 (1.53)	0.66 (0.00)	0.01 (0.00)
Scanalyzer11-6	16.36 (3.89)	0.65 (0.00)	0.49 (0.16)	38.82 (1.64)	0.30 (0.00)	0.09 (0.01)
Average	21.39 (1.94)	0.71 (0.00)	0.13 (0.10)	25.86 (1.67)	0.64 (0.00)	0.13 (0.06)
Total walltime			398.75 (36.16)			331.18 (21.39)
Instance	DAHDA* $[Z, A_{state}/SDD_{dynamic}]$			ZHDA* $[Z]$		
	speedup	CO	SO	speedup	CO	SO
Blocks10-0	25.11 (4.89)	0.88 (0.00)	0.08 (0.05)	14.93 (4.05)	0.98 (0.00)	0.30 (0.25)
Blocks11-1	24.88 (2.00)	0.91 (0.00)	0.21 (0.01)	27.98 (2.28)	0.98 (0.00)	0.07 (0.09)
Elevators08-5	27.59 (4.07)	0.83 (0.01)	-0.03 (0.05)	27.54 (2.72)	0.98 (0.01)	-0.03 (0.03)
Elevators08-6	15.28 (1.77)	0.88 (0.00)	0.31 (0.06)	18.19 (3.15)	0.96 (0.00)	0.06 (0.14)
Gripper8	21.80 (2.92)	0.98 (0.04)	0.08 (0.05)	21.66 (3.42)	0.98 (0.01)	0.08 (0.03)
Logistics00-10-1	17.52 (0.80)	0.84 (0.00)	0.00 (0.00)	16.09 (0.56)	0.99 (0.00)	0.00 (0.02)
Miconic11-0	**46.05** (0.87)	0.01 (0.00)	0.08 (0.01)	7.40 (2.74)	0.96 (0.00)	0.13 (0.04)
Miconic11-2	33.81 (1.35)	0.01 (0.00)	0.18 (0.00)	14.67 (2.65)	0.96 (0.00)	0.05 (0.06)
NoMprime5	18.46 (0.59)	0.90 (0.00)	-0.05 (0.01)	16.63 (0.57)	0.98 (0.00)	-0.02 (0.01)
Nomystery10	28.41 (2.29)	0.60 (0.00)	-0.07 (0.10)	21.68 (3.30)	0.99 (0.00)	-0.07 (0.22)
Openstacks08-19	24.54 (1.05)	0.24 (0.00)	0.18 (0.03)	**25.99** (3.40)	0.99 (0.00)	-0.05 (0.19)
Openstacks08-21	26.72 (1.06)	0.13 (0.00)	0.28 (0.05)	39.06 (2.71)	0.92 (0.00)	-0.00 (0.12)
Parcprinter11-11	7.00 (2.91)	0.19 (0.01)	4.38 (1.54)	19.15 (2.95)	0.97 (0.00)	0.08 (0.16)
Parking11	28.84 (0.82)	0.52 (0.00)	0.07 (0.02)	27.09 (3.55)	0.98 (0.00)	0.04 (0.16)
Pegsol11-18	22.16 (0.83)	0.34 (0.00)	-0.01 (0.02)	16.97 (1.05)	0.98 (0.00)	0.03 (0.03)
PipesNoTk10	15.58 (0.46)	0.98 (0.00)	0.01 (0.00)	11.22 (0.38)	0.98 (0.00)	0.03 (0.00)
PipesTk12	25.12 (0.31)	0.67 (0.00)	0.00 (0.00)	19.78 (0.36)	0.98 (0.00)	0.00 (0.00)
PipesTk17	31.16 (0.58)	0.60 (0.00)	0.01 (0.00)	26.27 (4.15)	0.98 (0.01)	0.00 (0.00)
Rovers6	25.48 (2.86)	0.05 (0.00)	0.26 (0.07)	30.01 (2.50)	0.76 (0.00)	0.00 (0.07)
Scanalyzer08-6	21.23 (2.62)	0.94 (0.00)	0.00 (0.00)	16.54 (0.43)	0.98 (0.00)	0.01 (0.00)
Scanalyzer11-6	19.51 (3.55)	0.50 (0.00)	0.46 (0.14)	20.36 (0.66)	0.98 (0.00)	0.05 (0.01)
Average	24.11 (1.84)	0.57 (0.00)	0.31 (0.11)	20.53 (2.27)	0.96 (0.00)	0.01 (0.08)
Total walltime			377.86 (28.85)			433.23 (47.90)

Bibliography

Asai, M., & Fukunaga, A. (2014). Fully automated cyclic planning for large-scale manufacturing domains.. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*.

Asai, M., & Fukunaga, A. (2016). Tiebreaking strategies for a* search: How to explore the final frontier. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*.

Bäckström, C., & Nebel, B. (1995). Complexity results for SAS+ planning. *Computational Intelligence*, *11*(4), 625–655.

Buluc, A., Meyerhenke, H., Safro, I., Sanders, P., & Schulz, C. (2015). Recent advances in graph partitioning. *arXiv preprint arXiv:1311.3144*.

Burns, E., Lemons, S., Ruml, W., & Zhou, R. (2010). Best-first heuristic search for multi-core machines. *Journal of Artificial Intelligence Research (JAIR)*, *39*, 689–743.

Burns, E. A., Hatem, M., Leighton, M. J., & Ruml, W. (2012). Implementing fast heuristic search code. In *Proceedings of the Annual Symposium on Combinatorial Search*, pp. 25–32.

Edelkamp, S., & Schroedl, S. (2010). *Heuristic Search: Theory and Applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Erdem, E., & Tillier, E. (2005). Genome rearrangement and planning. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pp. 1139–1144.

Evans, J. (2006). A scalable concurrent malloc (3) implementation for FreeBSD. In *Proc. BSDCan Conference*.

Evett, M., Hendler, J., Mahanti, A., & Nau, D. (1995). PRA*: Massively parallel heuristic search. *Journal of Parallel and Distributed Computing*, *25*(2), 133–143.

Fiduccia, C. M., & Mattheyses, R. M. (1982). A linear-time heuristic for improving network partitions. In *Conference on Design Automation*, pp. 175–181.

Fikes, R. E., & Nilsson, N. (1971). STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, *5*(2), 189–208.

Hart, P., Nilsson, N., & Raphael, B. (1968a). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on System Sciences and Cybernetics*, *SSC-4*(2), 100–107.

Hart, P. E., Nilsson, N. J., & Raphael, B. (1968b). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, *4*(2), 100–107.

Haslum, P., & Geffner, H. (2000). Admissible heuristics for optimal planning. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 140–149.

Helmert, M. (2006). The Fast Downward planning system. *Journal of Artificial Intelligence Research*, *26*, 191–246.

Helmert, M., Haslum, P., & Hoffmann, J. (2007). Flexible abstraction heuristics for optimal sequential planning. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 176–183.

Helmert, M., Haslum, P., Hoffmann, J., & Nissim, R. (2014). Merge-and-shrink abstraction: A method for generating lower bounds in factored state spaces. *Journal of the ACM (JACM)*, *61*(3), 16.

Helmert, M., & Lasinger, H. (2010). The scanalyzer domain: Greenhouse logistics as a planning problem.. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*.

Hendrickson, B., & Kolda, T. G. (2000). Graph partitioning models for parallel computing. *Parallel computing*, *26*(12), 1519–1534.

Holzmann, G. J., & Bošnaĉki, D. (2007). The design of a multicore extension of the SPIN model checker. *IEEE Transactions on Software Engineering*, *33*(10), 659–674.

Imai, T., & Fukunaga, A. (2015). On a practical, integer-linear programming model for delete-free tasks and its use as a heuristic for cost-optimal planning. *Journal of Artificial Intelligence Research*, *54*, 631–677.

Iosup, A., Ostermann, S., Yigitbasi, M. N., Prodan, R., Fahringer, T., & Epema, D. H. (2011). Performance analysis of cloud computing services for many-tasks scientific computing. *IEEE Transactions on Parallel and Distributed Systems*, *22*(6), 931–945.

Irani, K., & Shih, Y. (1986). Parallel A* and AO* algorithms: An optimality criterion and performance evaluation. In *International Conference on Parallel Processing*, pp. 274–277.

Jinnai, Y., & Fukunaga, A. (2016a). Abstract Zobrist hash: An efficient work distribution method for parallel best-first search. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pp. 717–723.

Jinnai, Y., & Fukunaga, A. (2016b). Automated creation of efficient work distribution functions for parallel best-first search. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*.

Jinnai, Y., & Fukunaga, A. (2017a). Learning to prune dominated action sequences in online black-box domain. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*. (to appear).

Jinnai, Y., & Fukunaga, A. (2017b). On work distribution functions for parallel best-first search. *Journal of Artificial Intelligence Research*. (to appear).

Jonsson, P., & Bäckström, C. (1998). State-variable planning under structural restrictions: Algorithms and complexity. *Artificial Intelligence*, *100*(1), 125–176.

Jyothi, S. A., Singla, A., Godfrey, P., & Kolla, A. (2014). Measuring and understanding throughput of network topologies. *arXiv preprint arXiv:1402.2531*.

Karypis, G., & Kumar, V. (1998). A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, *20*(1), 359–392.

Kishimoto, A., Fukunaga, A., & Botea, A. (2013). Evaluation of a simple, scalable, parallel best-first search strategy. *Artificial Intelligence*, *195*, 222–248.

Kishimoto, A., Fukunaga, A. S., & Botea, A. (2009). Scalable, parallel best-first search for optimal sequential planning. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 201–208.

Kobayashi, Y., Kishimoto, A., & Watanabe, O. (2011). Evaluations of Hash Distributed A* in optimal sequence alignment. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 584–590.

Korf, R. (1985). Depth-first iterative deepening: An optimal admissible tree search. *Artificial Intelligence*, *97*, 97–109.

Korf, R. E., & Felner, A. (2002). Disjoint pattern database heuristics. *Artificial Intelligence*, *134*(1), 9–22.

Korf, R. E., & Schultze, P. (2005). Large-scale parallel breadth-first search. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pp. 1380–1385.

Korf, R. E., Zhang, W., Thayer, I., & Hohwald, H. (2005). Frontier search. *Journal of the ACM (JACM)*, *52*(5), 715–748.

Kumar, V., Ramesh, K., & Rao, V. N. (1988). Parallel best-first search of state-space graphs: A summary of results.. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, Vol. 88, pp. 122–127.

Leighton, T., & Rao, S. (1999). Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. *Journal of the ACM (JACM)*, *46*(6), 787–832.

Lipovetzky, N., Ramirez, M., & Geffner, H. (2015). Classical planning with simulators: Results on the Atari video games. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 1610–1616.

Mahapatra, N. R., & Dutt, S. (1997). Scalable global and local hashing strategies for duplicate pruning in parallel A* graph search. *IEEE Transactions on Parallel and Distributed Systems*, *8*(7), 738–756.

Pearl, J. (1984). *Heuristics - Intelligent Search Strategies for Computer Problem Solving*. Addison–Wesley.

Pearson, W. R. (1990). Rapid and sensitive sequence comparison with FASTP and FASTA. *Methods in enzymology*, *183*, 63–98. Matrix score is available at http://prowl.rockefeller.edu/aainfo/pam250.htm.

Phillips, M., Likhachev, M., & Koenig, S. (2014). PA*SE: Parallel A* for slow expansions. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*.

Pommerening, F., Röger, G., Helmert, M., & Bonet, B. (2014). LP-based heuristics for cost-optimal planning. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*.

Romein, J. W., Plaat, A., Bal, H. E., & Schaeffer, J. (1999). Transposition table driven work scheduling in distributed search. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pp. 725–731.

Sousa, A., & Tavares, J. (2013). Toward automated planning algorithms applied to production and logistics. *IFAC Proceedings Volumes*, *46*(24), 165–170.

Thompson, J. D., Koehl, P., Ripp, R., & Poch, O. (2005). BAliBASE 3.0: Latest developments of the multiple sequence alignment benchmark. *Proteins: Structure, Function and Genetics (PROTEINS)*, *61*(1), 127–136.

Vidal, V., Vernhes, S., & Infantes, G. (2012). Parallel AI planning on the SCC. In *4th Many-core Applications Research Community (MARC) Symposium*, p. 15.

Zhou, R., & Hansen, E. A. (2004). Structured duplicate detection in external-memory graph search. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pp. 683–689.

Zhou, R., & Hansen, E. A. (2006a). Breadth-first heuristic search. *Artificial Intelligence*, *170*(4), 385–408.

Zhou, R., & Hansen, E. A. (2006b). Domain-independent structured duplicate detection. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pp. 1082–1087.

Zhou, R., & Hansen, E. A. (2007). Parallel structured duplicate detection. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pp. 1217–1223.

Zhou, R., & Hansen, E. A. (2011). Dynamic state-space partitioning in external-memory graph search. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 290–297.

Zhou, Y., & Zeng, J. (2015). Massively parallel A* search on a GPU. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pp. 1248–1255.

Zobrist, A. L. (1970). A new hashing method with application for game playing. *reprinted in International Computer Chess Association Journal (ICCA)*, *13*(2), 69–73.