A Graph-Partitioning Based Approach for Parallel Best-First Search

Yuu Jinnai [1,2]
[1] Center for Advanced Intelligence Project
RIKEN

Alex Fukunaga[2]
[2] Graduate School of Arts and Sciences
The University of Tokyo

Abstract

Parallel best-first search algorithms such as HDA* distribute work among the processes using a global hash function. Previous work distribution strategies seek to find a good wall-time efficiency by reducing search overhead and/or communication overhead, but there was no unified, quantitative analysis on the effects of the methods on both overheads. We 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 evaluate GRAZHDA* on domain-independent planning as well as a domain specific solver for the 24-puzzle and show that GRAZHDA* outperforms previous methods.

1 Introduction

The A* algorithm (Hart, Nilsson, and Raphael 1968) 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 1 machine to be solved. Thus, designing scalable, parallel search algorithms is an important goal.

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, and 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 and Hansen 2007) and PBNF (Burns et al. 2010). Abstraction-based work distribution achieves low communication overhead, but at the cost of high search overhead. Abstract Zobrist hashing (AZH) (Jinnai and Fukunaga 2016a) 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. Improvements to domain-independent, automated abstract feature generation methods for AZHDA* were proposed in (Jinnai and Fukunaga 2016a). Although these methods seek to reduce search/communication overheads in the HDA* framework, these methods can be characterized as *bottom-up, ad hoc* approaches that introduce new mechanisms to address some particular problem within the HDA*/AZHDA* framework, but these methods do not allow *a priori* prediction of the communication and search overheads that will be incurred.

This paper proposes a new, *top-down* approach to minimizing overheads in parallel best-first search. Instead of addressing specific problems/limitations within the AZHDA* framework, we formulate an objective function which defines exactly what we seek in terms of minimizing both search and communications overheads, enabling a predictive model of these overheads. We then propose an algorithm which directly synthesizes a work distribution function approximating the optimal behavior according to this objective. The resulting algorithm, GRAZHDA* significantly outperforms all previous variants of HDA*. We first review HDA* and previous work distribution methods (Sec. 2). We then describe the relationship between the work distribution method, search overhead, communication overheads and time efficiency, and propose an objective function for directly maximizing efficiency, which corresponds to the problem of partitioning the state space graph according to a sparsest-cut objective (Sec. 4-5). Next, 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* (Sec. 6). We evaluate GRAZHDA* experimentally on domain-independent planning using a commodity cluster (48 cores) as well as a cloud

cluster (128 cores), and show that it outperforms previous methods (Sec. 7). We also evaluate GRAZHDA* for a domain-specific, 24-puzzle solver on a multicore machine.

This paper summarizes work which will appear in a JAIR article (Jinnai and Fukunaga 2017).

2 Background

Hash Distributed A* (HDA*) (Kishimoto, Fukunaga, and Botea 2013) is a parallel A* algorithm where each processor has its own OPEN and CLOSED. A global hash function assigns a unique owner thread to every search node. Each thread T repeatedly executes the following: (1) For all new nodes n in T's message queue, if it is not in CLOSED (not a duplicate), put n in OPEN. (2) Expand node n with highest priority in OPEN. For every generated node c, compute hash value $H(c)$, and send c to the thread that owns $H(c)$.

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

Communication Overhead (CO): Communication overhead is 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 structure such as message queues. HDA* incurs communication overhead when transferring a node from the thread where it is generated to its owner according to the hash function. In general, CO 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 paper 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). 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.

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.

Zobrist Hashing, Abstraction, and Abstract Zobrist Hashing In the original work on HDA*, Kishimoto et al. (2013) used Zobrist hashing (1970). 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.

$$Z(s) := R[x_0] \; xor \; R[x_1] \; xor \; \cdots \; xor \; R[x_n]$$

Zobrist hashing seeks to distribute nodes uniformly among all threads, 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.

In order to minimize communication overhead in HDA*, Burns et al (2010) proposed AHDA*, which uses *abstraction* based node assignment. AHDA* applies the state space partitioning technique used in PBNF (Burns et al. 2010), which in turn is based on PSDD (Zhou and Hansen 2007). Abstraction projects nodes in the state space into *abstract states*, and abstract states are assigned to processors using 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. Abstraction is generally constructed by ignoring subset of features. It has been shown that abstraction has roughly 2-4 times the search overhead of Zobrist hashing on the 24-puzzle (Jinnai and Fukunaga 2016a).

Dynamic AHDA* (DAHDA*), dynamically sets the threshold of the abstract graph size according to the instance's state space size (Jinnai and Fukunaga 2016b). DAHDA* was shown to significantly improve upon AHDA* in distributed memory clusters, in cases where AHDA* fails to solve many instances because of poor load balancing.

Abstract Zobrist hashing (AZH) (Jinnai and Fukunaga 2016a) 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:

$$AZ(s) := R[A(x_0)] \; xor \; R[A(x_1)] \; xor \; \cdots \; xor \; R[A(x_n)] \quad (1)$$

where A is a *feature projection function*, a many-to-one mapping from from each raw feature to an *abstract feature*, and R is a precomputed 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. Figure 1 illustrates the computation of AZH for the 8-puzzle.

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 states 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 for 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.

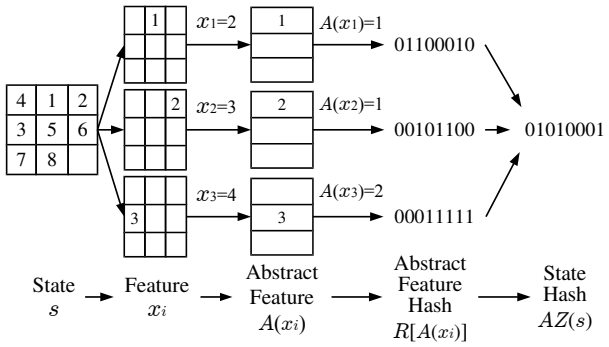


Figure 1: Calculation of abstract Zobrist hash (AZH) value $AZ(s)$ for the 8-puzzle: State $s = (t_1, t_2, ..., t_8)$, where $t_i = 1, 2, ..., 9$. The Zobrist hash value of s is the result of xor'ing a preinitialized random bit vector $R[t_i]$ for each feature (tile) t_i. AZH incorporates an additional step which projects features to abstract features (for each feature t_i, look up $R[A(t_i)]$ instead of $R[t_i]$).

Domain-Independent Feature Projection Functions for Abstract Zobrist Hashing The feature projection function plays a critical role in determining the performance of AZH, because AZH relies on the feature projection in order to reduce communications overhead. Below, we review two recently proposed domain-independent abstract feature generation methods, GreedyAFG and FluencyAFG.

Greedy Abstract Feature Generation (Jinnai and Fukunaga 2016a) *Greedy abstract feature generation* (GreedyAFG) is a simple, domain-independent abstract feature generation method, which partitions each feature into 2 abstract features (Jinnai and Fukunaga 2016a). GreedyAFG first identifies *atom groups* (sets 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 and Nebel 1995)). Each atom group G is partitioned into 2 abstract features S_1 and S_2, based G's undirected transition graph (nodes are propositions, edges are transitions), as follows: (1) assign the minimal degree node to S_1; (2) greedily add to S_1 the unassigned node which shares the most edges with nodes in S_1; (3) while $|S_1| < |G|/2$ repeat step (2) to guarantee ; (4) assign all unassigned nodes to S_2. This procedure guarantees $|S_2| \leq |S_1| \leq |S_2| + 1$.

Fluency-Dependent Abstract Feature Generation (Jinnai and Fukunaga 2016b) 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. Fluency-dependent abstract feature generation (FluencyAFG) overcomes this limitation (Jinnai and Fukunaga 2016b). The *fluency* of a variable v is the # of ground actions which change the value of the v divided by the total # 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). For example, in the `grid` domain, the atom group for the p_{rob}, the SAS+ variable representing the robot's position has high fluency (~ 1.0), so FluencyAFG marks it for exclusion, but the value of the abstract feature for p_{rob} seldom changes because the size of the grid is very large.

3 Work Distribution as a Graph Partitioning

Although previous research on work distribution for HDA* proposed methods which reduce CO or SO, there was no explicit model which enabled the prediction of the actual efficiency achieved during search.

In this section, we show that a work distribution method can be modelled 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.

Work distribution methods for hash-based parallel search distribute nodes by assigning a process to each node in the state space.

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.

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 number of total edges, i.e.,

$$CO = \frac{\sum_i^p \sum_{j>i}^p E(S_i, S_j)}{\sum_i^p \sum_{j\geq i}^p E(S_i, S_j)}, \quad LB = \frac{|S_{max}|}{mean|S_i|}, \qquad (2)$$

[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.

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:

$$SO = \sum_{i}^{p}(|S_{max}| - |S_i|) = p(LB - 1). \quad (3)$$

4 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* $\mathit{eff}_{actual} := \frac{speedup}{\#cores}$, where $speedup = T_n/T_1$, T_n is the runtime on N cores. Our ultimate goal is to maximize eff_{actual}.

Communication Efficiency: Assume that the communication cost between every pair of processors is identical. Then communication efficiency, the degradation of efficiency by communication cost, is $\mathit{eff}_c = \frac{1}{1+cCO}$, where $c = \frac{\text{time for sending a node}}{\text{time for generating a node}}$.

Search Efficiency: Assuming every core expands 1 node at a time and 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, 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), but we assume that CO and SO are the dominant factors in determining efficiency.

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

$$\mathit{eff}_{esti} = \mathit{eff}_c \cdot \mathit{eff}_s = 1/\big(\ (1 + cCO)(1 + SO)\ \big) = 1/\big(\ (1 + cCO)(1 + p(LB - 1))\ \big) \quad (4)$$

Experiment: eff_{esti} model vs. actual efficiency We evaluated the performance of the following HDA* variants on domain-independent planning.

- FAZHDA*: AZHDA* using fluency-based filtering (FluencyAFG) (Jinnai and Fukunaga 2016b).
- GAZHDA*: AZHDA* using greedy abstract feature generation (GreedyAFG) (Jinnai and Fukunaga 2016a).
- OZHDA*: HDA* with Operator-based Zobrist hashing (Jinnai and Fukunaga 2016b).
- DAHDA*: AHDA* (Burns et al. 2010) with dynamic abstraction size threshold (Jinnai and Fukunaga 2016b).
- ZHDA*: HDA* using Zobrist hashing (Kishimoto, Fukunaga, and Botea 2013).

We implemented these HDA* variants on Fast Downward (parallelized implementation using MPICH 3) using the merge&shrink heuristic (Helmert et al. 2014) (abstraction size =1000). 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), 16GB RAM, and 1000Mbps Ethernet interconnect.We packed 100 states per MPI message.

Table 1 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 M&S 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 2-4. Figure 2b, 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 $\mathit{eff}_{actual} = a \cdot \mathit{eff}_{esti}$, $a = 0.86$ 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).

5 Sparsest Cut Objective Function

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 and Kolda 2000; Buluç et al. 2013).

In Sec. 4, we showed that eff_{esti} can be used to effectively predict the actual efficiency of a work distribution. By defining a graph cut objective such that the partitioning the nodes in the search space (with $f < f^*$) according to this graph cut

[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.

objective corresponds to maximizing eff_{esti}, we would have a method of generating an optimal workload distribution.

A *sparsest cut* objective for graph partitioning problem seeks to maximize the *sparsity* of the graph (Leighton and Rao 1999).We define sparsity as

$$Sparsity := \frac{\prod_i^k |S_i|}{\sum_i^k \sum_{j>i}^k E(S_i, S_j)}, \quad (5)$$

where $|S_i|$ is the sum of nodes weights in partition S_i, $E(S_i, S_j)$ 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 4 and 2, Sparsity simultaneously considers both LB and CO, as the numerator $\prod_i^k |S_i|$ corresponds to LB and the denominator $\sum_i^k \sum_{j>i}^k E(S_i, S_j)$ corresponds to CO.

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

Experiment: Relationship between Sparsity and eff_{esti} To validate the correlation between sparsity and estimated efficiency eff_{esti}, we used METIS (approximate) graph partitioning package (Karypis and Kumar 1998) to partition modified versions of the search spaces of the instances used in Fig. 2a. 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 2c 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.

6 Graph Partitioning-Based Abstract Feature Generation (GRAZHDA*)

Since eff_{esti} model accurately estimates actual efficiency, and sparsity has a strong correlation with eff_{esti}, a partition of the state space graph which minimize sparsity should be a (near) optimal work distribution which maximizes eff_{esti}. Unfortunately, it is impractical to directly apply standard graph partitioning algorithms to the state space graph because the state space graph is a huge *implicit* graph, and the partitioner needs as input the explicit representation of the relevant state space graph (a solution to the search problem itself!).

Therefore, to generate a work distribution method for parallel A*, we have to partition some graph which is easily accessible from the domain description (e.g. PDDL, SAS+). We propose *Graph partitioning-based Abstract Zobrist HDA** (**GRAZHDA**), which approximates the optimal strategy by partitioning *domain transition graphs*.

Given an atom group $x \in X$, its domain transition graph (DTG) $\mathcal{D}_x(E, V)$ is a directed graph where vertices V corresponds to the value of the atom group and edges E to their transitions, where $(v, v') \in E$ if and only if there is an operator o with $v \in del(o)$ and $v' \in add(o)$ (Jonsson and Bäckström 1998). We used DTGs of SAS+ variables.

Figure 3 shows the partitioning of a DTG (for the variable representing package location) in the standard `logistics` domain using sparsest cut objective function. Maximizing sparsity results in cutting only 1 edge (i.e., good load balance).

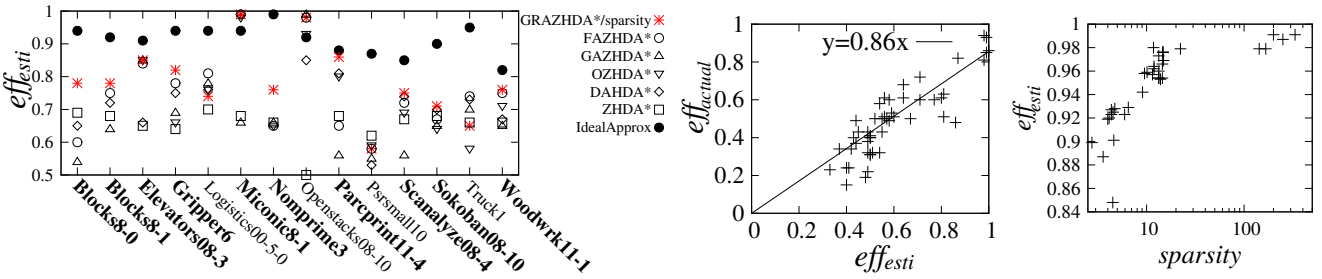
GRAZHDA* treats each partition of the DTG as an abstract feature in the AZH framework, assigning a hash value to each abstract feature. Since the AZH value of a state is the XOR of the hash values of the abstract features (Eqn 1), 2 nodes in the state space are in different partitions if and only if they are partitioned in *any* of the DTGs. (Figure 4). Therefore, GRAZHDA generates 2^n partitions from n DTGs, which are then projected to the p processors (by taking the partition ID modulo p). To make it likely that partitioning over the DTGs is a good approximation for partitioning the actual state space graph, we set a weight for each edge $e = \frac{\text{\# ground actions which correspond to the transition}}{\text{\# ground actions}}$. As DTGs typically have < 10 nodes, we compute the optimal sparsest cut with a straightforward branch-and-bound procedure.

7 Evaluation of GRAZHDA*

Figure 2a shows eff_{esti} for the various work distribution methods, including GRAZHDA*/Sparsity (see Sec. 4 for experimental setup and list of methods included in comparison). To evaluate how these methods compare to an ideal (but impractical) model which actually 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 and 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 (Eqn. 5) 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 is a useful approximation for an ideal work distribution.

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 Sec. 4 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 1, GRAZHDA*/sparsity achieved a good balance between CO and SO and had the highest actual speedup overall, significantly outperforming all other previous methods.

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



(a) Comparison eff_{esti} for various work distribution methods (b) eff_{esti} vs. eff_{actual} (c) *sparsity* vs. eff_{esti}

Figure 2: Figure 2a compares eff_{esti} when $c = 1.0$, $p = 48$. **Bold** indicates that GRAZHDA* has the best eff_{esti} (except for IdealApprox). Figure 2b compares eff_{esti} and the actual experimental efficiency when $c = 1.0$, $p = 48$. $eff_{actual} = 0.86 \cdot eff_{esti}$ with variance of residuals = 0.013 (least-squares regression). Figure 2c 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}.

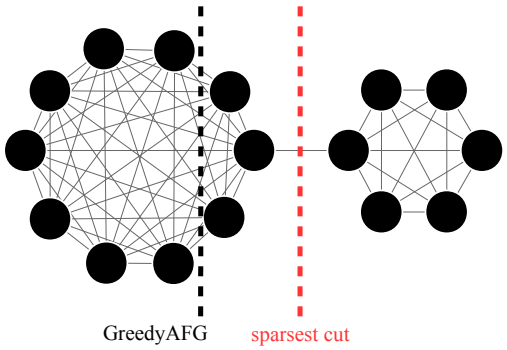


Figure 3: Example of sparsest cut and GreedyAFG to a domain transition graph in logistic domain.

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 et al. 2011). We intentionally chose this configuration to evaluate work distribution methods in environment which is significantly different from our other experiments. Table 2 shows that as with the smaller-scale cluster results, GRAZHDA*/sparsity outperformed other methods in this large-scale cloud environment.

24-Puzzle Results: We evaluated GRAZHDA*/sparsity on the 24-puzzle using a high-performance, domain specific 24-puzzle solver using a disjoint PDB heuristic (Korf and Felner 2002) (node generation rate = 367,645 nodes/sec/core). We compared GRAZHDA*/sparsity (automated abstract feature generation) vs. AZHDA* with the hand-crafted work distribution (AZHDA*/HandCrafted) used in (Jinnai and Fukunaga 2016a) and ZHDA* (Kishimoto, Fukunaga, and Botea 2013) on 100 random instances on a single Xeon E5-2650 v2 2.60 GHz CPU. The average runtime of sequential A* on the instances was 219 secs. With 8 cores, the speedups were 7.84(GRAZHDA*/sparsity), 7.85(AZHDA*/HandCrafted), and 5.95(ZHDA*). Thus, the completely automated GRAZHDA*/sparsity is competitive with a carefully hand-designed work distribution method.

8 Previous Methods as Graph Partitioning

Previous work distribution methods for parallel best-first search can be understood in terms of the graph partitioning framework proposed in this paper. ZHDA*, the original Zobrist-hashing based HDA* (Kishimoto, Fukunaga, and Botea 2013), corresponds to an extreme case of the AZH framework where every node is assigned to a different partition. Abstraction-based work partitioning in AHDA* (Burns et al. 2010) can be described as partitioning to a subset of DTGs such that each node is assigned to a different partition. Previous instances of the AZH framework (Jinnai and Fukunaga 2016a) can be viewed as the generation abstract features based on *bisections* of DTGs according to some objective. Consider *weighted sparsity*, a generalization of the sparsity objective:

$$WSparsity := \frac{\prod_i^k |S_i| + w_{co}}{\sum_i^k \sum_{j>i}^k E(S_i, S_j) + w_{lb}}. \quad (6)$$

Then, GreedyAFG (Jinnai and Fukunaga 2016a) can be described as optimizing weighted sparsity with weights $w_{co} = 0, w_{lb} = +\infty$. Because it only optimizes LB, GAZHDA* often results in significantly suboptimal CO. For example, Figure 3 shows that for this logistics domain DTG, GreedyAFG ends up cutting 2 edges while SparsestAFG cuts only 1. We evaluated eff_{esti} for various values of these weights, and observed that peak eff_{esti} was in the vicinity of $w_{co} = w_{lb} = 0$ (i.e., same as Eqn. 5), while overweighting CO or LB ($w_{co} > 0.2$ or $w_{lb} > 0.2$) resulted in significantly degraded eff_{esti}.

FAZHDA* (Jinnai and Fukunaga 2016b) can be described as an extension of GAZHDA* which generates the partition $S_1 = G, S_2 = \emptyset$ when the optimal sparsity is lower than some threshold (control parameter).

Thus, by casting previous work distribution methods as instances of the graph partitioning framework, it can be seen that from the perspective of graph partitioning, previous methods are *ad hoc* solutions to the problem of work distribution. In contrast, GRAZHDA*/sparsity explicitly seeks a work distribution which addresses both LB and CO, and our experiments validate the effectiveness of this top-down approach.

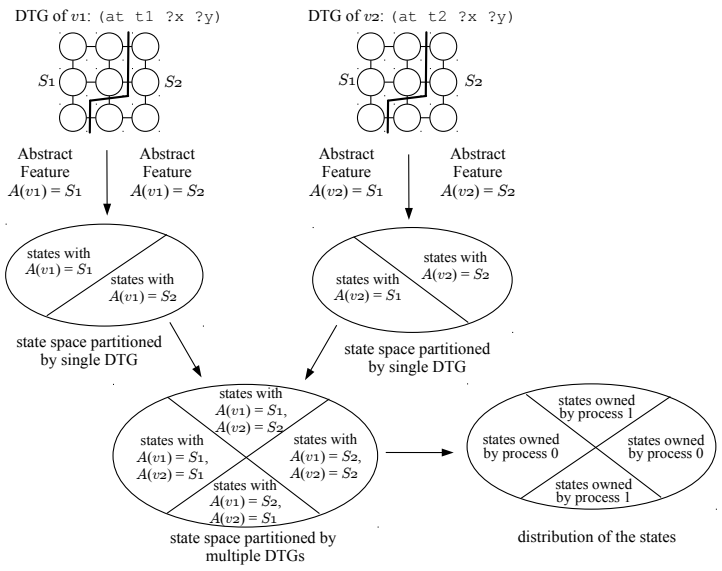


Figure 4: Partitioned DTGs and the resulting partitioniong of the state space by XORing the hash values of abstract features.

9 Conclusions

We proposed and evaluated a new, domain-independent approach to work distribution for parallel best-first search in the HDA* framework. The main contributions are (1) proposal and validation of eff_{esti}, a model of search and communication overheads for HDA* which can be used to predict actual walltime efficiency, (2) formulating the optimization of eff_{esti} as a graph partitioning problem with a sparsity objective, and validating the relationship between eff_{esti} and the sparsity objective, and (3) GRAZHDA*, a new work distribution method which approximate the optimal strategy by partitioning domain transition graphs. We experimentally showed that GRAZHDA*/sparsity significantly improves both estimated efficiency (eff_{esti}) as well as actual performance (walltime efficiency) compared to previous work distribution methods. Our results demonstrate the viability of approximating the partitioning of the entire search space by applying graph partitioning to an abstraction of the state space (i.e., the DTG).

Despite significant improvements compared to previous work distribution approaches, there is room for improvement. The gap between the eff_{esti} metric for GRAZHDA*/sparsity and a ideal model (IdealApprox) represents the gap between actually partitioning the state space graph (as IdealApprox does) vs. the approximation obtained by the GRAZHDA*/sparsity DTG partitioning. Closing this gap in eff_{esti} is a direction for future work.

References

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

Buluç, A.; Meyerhenke, H.; Safro, I.; Sanders, P.; and Schulz, C. 2013. Recent advances in graph partitioning. *Preprint*.

Burns, E. A.; Lemons, S.; Ruml, W.; and Zhou, R. 2010. Best-first heuristic search for multicore machines. *Journal of Artificial Intelligence Research* 39:689–743.

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

Helmert, M.; Haslum, P.; Hoffmann, J.; and 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.

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

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

Jinnai, Y., and Fukunaga, A. 2016a. Abstract zobrist hash: An efficient work distribution method for parallel best-first search. In *Proceedings of the Thritieth AAAI Conference on Artificial Intelligence (AAAI)*.

Jinnai, Y., and Fukunaga, A. 2016b. Automated creation of efficient work distribution functions for parallel best-first search. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling, ICAPS 2016*.

Jinnai, Y., and Fukunaga, A. 2017. On hash-based work distribution methods for parallel best-first search. *J. Artif. Intell. Res.(JAIR)*. (to appear).

Jonsson, P., and 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.; and Kolla, A. 2014.

Table 1: Comparison of eff_{actual}, eff_{esti}, average speedups (spdup), communication/search overhead (CO, SO) on 10 runs on a commodity cluster with 6 nodes, 48 processes using merge&shrink heuristic. 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}.

Instance	A*		GRAZHDA*/sparsity					FAZHDA*				
	time	expd	eff_{actual}	eff_{esti}	spdup	CO	SO	eff_{actual}	eff_{esti}	spdup	CO	SO
Blocks10-0	129.29	11065451	**0.57**	**0.57**	**27.17**	0.28	0.38	0.54	0.43	26.02	0.70	0.35
Blocks11-1	813.86	52736900	**0.71**	**0.53**	**34.25**	0.66	0.15	0.71	0.50	34.25	0.66	0.15
Elevators08-5	165.22	7620122	0.34	0.51	16.43	0.47	0.33	0.26	0.49	12.34	0.32	0.51
Elevators08-6	453.21	18632725	0.45	0.50	21.47	0.49	0.37	0.38	0.36	18.05	0.52	0.81
Gripper8	517.41	50068801	0.56	0.60	26.67	0.50	0.15	**0.57**	**0.63**	**27.45**	0.43	0.10
Logistics00-10-1	559.45	38720710	**0.94**	**0.70**	**45.16**	0.43	0.01	0.91	0.61	43.85	0.57	0.02
Miconic11-0	232.07	12704945	0.87	0.95	41.97	0.01	0.07	0.88	0.91	42.43	0.01	0.06
Miconic11-2	262.01	14188388	**0.94**	**0.97**	**45.26**	0.01	0.05	0.93	0.92	44.87	0.01	0.05
NoMprime5	309.14	4160871	**0.50**	**0.58**	**23.95**	0.80	-0.04	0.48	0.53	22.87	0.79	-0.05
NoMystery10	179.52	1372207	**0.72**	0.61	**34.80**	0.51	0.12	0.48	**0.75**	22.99	0.24	-0.44
Openstacks08-19	282.45	15116713	0.51	0.59	24.67	0.27	0.34	0.42	0.58	20.00	0.24	0.37
Openstacks08-21	554.63	19901601	0.53	0.65	25.23	0.17	0.35	0.52	0.62	24.97	0.15	0.35
Parcprinter11-11	307.19	6587422	**0.42**	**0.54**	**20.26**	0.26	0.55	0.27	0.49	13.08	0.26	0.61
Parking11-5	237.05	2940453	**0.62**	0.55	**29.75**	0.40	0.34	0.62	0.54	29.67	0.63	0.11
Pegsol11-18	801.37	106473019	0.44	**0.72**	21.03	0.39	0.02	0.44	0.71	20.97	0.39	0.00
PipesNoTk10	157.31	2991859	**0.33**	**0.52**	**15.73**	0.98	0.01	0.33	0.49	15.64	0.98	0.01
PipesTk12	321.55	15990349	0.70	**0.66**	33.78	0.46	0.05	**0.83**	0.65	**39.65**	0.46	0.03
PipesTk17	356.14	18046744	0.92	**0.65**	43.92	0.54	0.01	**0.94**	0.63	**45.03**	0.54	0.01
Rovers6	1042.69	36787877	**0.86**	**0.79**	**41.17**	0.15	0.14	0.84	0.72	40.48	0.15	0.17
Scanalyzer08-6	195.49	10202667	**0.69**	**0.92**	**32.92**	0.12	0.01	0.63	0.86	30.31	0.12	0.01
Scanalyzer11-6	152.92	6404098	**0.91**	**0.78**	**43.83**	0.16	0.13	0.57	0.63	27.31	0.18	0.34
Average	382.38	21557805	**0.64**	**0.62**	**30.92**	0.38	0.17	0.60	0.61	28.68	0.40	0.17
Total walltime	8029.97	452713922					**277.91**					301.38

	GAZHDA*					OZHDA*					DAHDA*					ZHDA*				
	eff_{actual}	eff_{esti}	spdup	CO	SO	eff_{actual}	eff_{esti}	spdup	CO	SO	eff_{actual}	eff_{esti}	spdup	CO	SO	eff_{actual}	eff_{esti}	spdup	CO	SO
Blocks10-0	0.45	0.44	21.81	0.99	0.12	0.32	0.37	15.47	0.98	0.34	0.52	0.47	25.11	0.88	0.08	0.31	0.48	14.93	0.98	0.30
Blocks11-1	0.61	0.48	29.20	0.99	0.03	0.61	0.47	29.20	0.99	0.03	0.52	0.43	24.88	0.91	0.21	0.58	0.48	27.98	0.98	0.07
Elevators08-5	**0.61**	0.58	**29.35**	0.65	-0.00	0.46	**0.64**	21.86	0.09	0.44	0.57	0.51	27.59	0.83	-0.03	0.57	0.47	27.54	0.98	-0.03
Elevators08-6	**0.72**	**0.76**	**34.52**	0.24	-0.09	0.68	0.56	32.70	0.41	0.22	0.32	0.39	15.28	0.88	0.31	0.38	0.49	18.19	0.96	0.06
Gripper8	0.46	0.50	21.86	0.81	0.06	0.52	0.44	24.77	0.98	0.14	0.45	0.45	21.80	0.98	0.08	0.45	0.47	21.66	0.98	0.08
Logistics00-10-1	0.24	0.42	11.68	0.85	0.25	0.24	0.43	11.68	0.85	0.25	0.36	0.53	17.52	0.84	0.00	0.34	0.48	16.09	0.99	0.00
Miconic11-0	0.27	0.53	13.15	0.53	0.24	0.79	**0.96**	37.86	0.02	0.02	**0.96**	0.91	**46.05**	0.01	0.08	0.15	0.48	7.40	0.96	0.13
Miconic11-2	0.18	0.37	8.53	0.53	0.74	0.77	0.90	36.86	0.02	0.07	0.70	0.81	33.81	0.01	0.18	0.31	0.48	14.67	0.96	0.05
NoMprime5	0.39	0.48	18.55	0.95	-0.06	0.35	0.51	16.66	0.94	0.00	0.38	0.49	18.46	0.90	-0.05	0.35	0.47	16.63	0.98	-0.02
NoMystery10	0.40	0.66	18.98	0.42	-0.07	0.45	0.50	21.61	0.74	0.11	0.59	0.60	28.41	0.60	-0.07	0.45	0.49	21.68	0.99	-0.07
Openstacks08-19	0.46	0.58	22.14	0.38	0.21	0.36	0.55	17.11	0.34	0.32	0.51	**0.66**	24.54	0.24	0.18	**0.54**	0.47	**25.99**	0.99	-0.05
Openstacks08-21	0.53	0.65	25.67	0.15	0.31	**0.82**	0.49	**39.34**	0.92	0.05	0.56	**0.68**	26.72	0.13	0.28	0.81	0.51	39.06	0.92	-0.00
Parcprinter11-11	0.35	0.40	16.85	0.74	0.41	0.33	0.34	15.98	0.82	0.56	0.15	0.15	7.00	0.19	4.38	0.40	0.48	19.15	0.97	0.08
Parking11-5	0.59	0.49	28.43	0.98	0.02	0.56	0.46	26.76	0.97	0.07	0.60	**0.59**	28.84	0.52	0.07	0.56	0.47	27.09	0.98	0.04
Pegsol11-18	0.34	0.53	16.22	0.77	0.05	**0.55**	0.71	**26.17**	0.34	-0.03	0.46	0.70	22.16	0.34	-0.01	0.35	0.47	16.97	0.98	0.03
PipesNoTk10	0.32	0.50	15.58	0.98	0.01	0.32	0.48	15.22	0.98	0.02	0.32	0.48	15.58	0.98	0.01	0.07	0.48	3.22	0.98	-0.44
PipesTk12	0.41	0.48	19.84	0.99	0.01	0.45	0.49	21.40	0.88	0.04	0.52	0.57	25.12	0.67	0.00	0.41	0.48	19.78	0.98	0.00
PipesTk17	0.56	0.50	26.64	0.98	0.00	0.60	0.52	28.82	0.88	0.00	0.65	0.60	31.16	0.60	0.01	0.55	0.49	26.27	0.98	0.00
Rovers6	0.70	0.61	33.49	0.56	0.01	0.85	0.71	41.00	0.31	0.03	0.53	0.73	25.48	0.05	0.26	0.63	0.53	30.01	0.76	0.00
Scanalyzer08-6	0.42	0.54	20.28	0.77	0.01	0.49	0.58	23.70	0.66	0.01	0.44	0.51	21.23	0.94	0.00	0.34	0.48	16.54	0.98	0.01
Scanalyzer11-6	0.34	0.41	16.36	0.65	0.49	0.81	0.68	38.82	0.30	0.09	0.41	0.44	19.51	0.50	0.46	0.42	0.48	20.36	0.98	0.05
Average	0.45	0.51	21.39	0.71	0.13	0.54	0.53	25.86	0.64	0.13	0.50	0.47	24.11	0.57	0.31	0.43	0.49	20.53	0.96	0.01
Total walltime					398.75					331.18					377.86					433.23

Measuring and understanding throughput of network topologies. *arXiv preprint arXiv:1402.2531*.

Karypis, G., and 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. S.; and Botea, A. 2009. Scalable, parallel best-first search for optimal sequential planning. In *Proc. 19th International Conference on Automated Planning and Scheduling (ICAPS)*, 201–208.

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

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

Leighton, T., and 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.

Zhou, R., and Hansen, E. A. 2007. Parallel structured duplicate detection. In *Proc. 22nd AAAI Conference on Artificial Intelligence (AAAI)*, 1217–1223.

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.

Table 2: 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			FAZHDA*		
	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*			OZHDA*			DAHDA*			ZHDA*		
	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