Automated Creation of Work Distribution Functions for Parallel Best-First Search

Yuu Jinnai  Alex Fukunaga

The University of Tokyo

ICAPS-2016
Hash Distributed A* (HDA*) is parallel A* which distributes nodes according to a hash function which assigns each state to a unique process. As HDA* relies on the hash function for load balancing, the choice of hash function is significant to its performance!
Zobrist hashing (ZHDA*)
(Zobrist 1970; Kishimoto et al. 2013)
+ good load balance
- high communication overhead

State abstraction (AHDA*)
(Burns et al. 2010)
- worse load balance
+ low communication overhead

Abstract Zobrist Hashing (AZHDA*)
(Jinnai & Fukunaga 2016)
+ good load balance
+ low communication overhead
* requires feature abstraction as a parameter

This presentation proposes a method to automatically generate efficient feature abstraction for Abstract Zobrist hashing
Hash Function for HDA*

- State \((s)\) is given as a set of features \(x_i\):
  \[
  \text{state } s = (x_1, x_2, \ldots, x_n)
  \]

- Given a state \(s\), a hash function \(H(s)\) returns the process which owns the state \(s\)
Hash Function for HDA*

- We want $H(s)$ to be balanced
  → load balance

![State space graph](image-url)
Hash Function for HDA*

- We want $H(s)$ to be balanced → load balance
- We want the value of $H(s)$ to not change frequently → communication overhead
Zobrist Hashing (ZHDA*)
Zobrist (1970); Kishimoto et al. (2009)

- Goal: Distribute nodes uniformly among processes
- Method: Initialize a table of random bit strings $R$, XOR the hash value $R_i[x_i]$ for each feature

$$Z(s) = R_1[x_1] \text{ xor } R_2[x_2] \text{ xor } ... \text{ xor } R_n[x_n]$$
Zobrist Hashing (ZHDA*)
Zobrist (1970); Kishimoto et al. (2009)

\[ Z(s) = R_1[x_1] \ xor \ R_2[x_2] \ xor \ ... \ xor \ R_n[x_n] \]

(xi represents the position of tile i)
Zobrist Hashing (ZHDA*)
Zobrist (1970); Kishimoto et al. (2009)

- Strength: good load balance
- Limitation: high communication overhead

![State space graph and process communication diagram](image)
State abstraction (AHDA*)
Burns et al. (2010)

- Goal: Assign neighbor nodes to the same process
- Method: Project states into abstract states, and abstract states are assigned to processors

\[ A(s) = R[s'] \]

Example: \( s' \) only considers the position of tile 1, 2, and 3:

<table>
<thead>
<tr>
<th>State ( s )</th>
<th>Abstract State ( s' )</th>
<th>State Hash ( R[s'] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 1 6</td>
<td>3 1 2</td>
<td>101000001</td>
</tr>
</tbody>
</table>
State abstraction (AHDA*)
Burns et al. (2010)

- Strength: low communication overhead
- Limitation: worse load balance
Goal: Distributes nodes uniformly while assigning neighbor nodes to the same process

Method: Apply feature abstraction $A_i(x_i)$ to project features into abstract features and $XOR$ the hash value of each abstract feature

$$AZ(s) = R_1[A_1(x_1)] \text{xor } R_2[A_2(x_2)] \text{xor } ... \text{xor } R_n[A_n(x_n)]$$

or

$$AZ(s) = Z(s'), \text{ where } s' = (A_1(x_1), A_2(x_2),..., A_n(x_n))$$
Abstract Zobrist Hashing (AZHDA*)
Jinnai&Fukunaga (2016)

\[ AZ(s) = R_1[A_1(x_1)] \text{xor} R_2[A_2(x_2)] \text{xor} \ldots \text{xor} R_n[A_n(x_n)] \]
Zobrist Hashing (ZHDA*)
Zobrist (1970); Kishimoto et al. (2009)

\[ Z(s) = R_1[x_1] \text{xor} R_2[x_2] \text{xor} \ldots \text{xor} R_n[x_n] \]
State abstraction (AHDA*)
Burns et al. (2010)

$$A(s) = R[s']$$

Example: $s'$ only considers the position of tile 1, 2, and 3:
Abstract Zobrist Hashing (AZHDA*)
Jinnai&Fukunaga (2016)

\[ AZ(s) = R_1[A_1(x_1)] \text{xor} R_2[A_2(x_2)] \text{xor} ... \text{xor} R_n[A_n(x_n)] \]
Abstract Zobrist Hashing (*AZHDA*)
Jinnai & Fukunaga (2016)

- Achieves good load balancing using Zobrist hashing
- Reduces communication overhead using feature abstraction
The performance of AZHDA* with hand-crafted abstract feature

- (Jinnai & Fukunaga, 2016) showed that Abstract Zobrist hashing using hand-crafted feature abstraction significantly outperformed previous methods (Zobrist hashing and Abstraction)

AZHDA*: Abstract Zobrist hashing + HDA*

AHDA*: State abstraction + HDA*

ZHDA*: Zobrist hashing + HDA*

24-puzzle (Jinnai & Fukunaga, 2016)
Zobrist hashing for planning

We can use SAS+ variables for Zobrist hashing

\[ Z(s) = R_1[x_1] \text{ xor } R_2[x_2] \text{ xor } \ldots \text{ xor } R_n[x_n] \]

Example: blocks world

\[ s = (x_1, x_2, x_3) \]
Abstract Zobrist hashing for planning

To apply AZHDA* on domain-independent planning, we have to generate feature abstraction $A_i(x_i)$ automatically.

$AZ(s) = R_1[A_1(x_1)] \xor R_2[A_2(x_2)] \xor \ldots \xor R_n[A_n(x_n)]$

Example: blocks world

Grey squares represent feature abstraction

$s = (x_1, x_2, x_3)$

Example: blocks world

Grey squares represent feature abstraction
Greedy abstract feature generation  
(Jinnai&Fukunaga 2016)

Approach: maps each SAS+ variable $x_i$ to abstract feature $S_1$ and $S_2$ based on $x_i$'s domain transition graphs (nodes are values, edges are transitions)
Greedy abstract feature generation  
(Jinnai&Fukunaga 2016)

Approach: maps each SAS+ variable $x_i$ to abstract feature $S_1$ and $S_2$ based on $x_i$'s domain transition graphs (nodes are values, edges are transitions)

1. Assign the minimal degree node to $S_1$

```
S1
x1=1 -> x2=2 -> x3=3
      |     |
      v     v
x4=4 -> x5=5 -> x6=6
      |     |
      v     v
x7=7 -> x8=8 -> x9=9
```

DTG of a variable $x_i$ represents the transition of the value

GreedyAFG applied to DTG of 8-puzzle
Greedy abstract feature generation
(Jinnai&Fukunaga 2016)

Approach: maps each SAS+ variable $x_i$ to abstract feature $S_1$ and $S_2$ based on $x_i$'s domain transition graphs (nodes are values, edges are transitions)

1. Assign the minimal degree node to $S_1$

2. Add to $S_1$ the unassigned node which shares the most edges with node in $S_1$

---

GreedyAFG applied to DTG of 8-puzzle

DTG of a variable $x_i$ represents the transition of the value
Approach: maps each SAS+ variable $x_i$ to abstract feature $S_1$ and $S_2$ based on $x_i$'s domain transition graphs (nodes are values, edges are transitions)

1. Assign the minimal degree node to $S_1$
2. Add to $S_1$ the unassigned node which shares the most edges with node in $S_1$
3. Until $|S_1|$ reaches the half of the DTG, repreat step 2.

Greedy AFGeneration (Jinnai&Fukunaga 2016)

GreedyAFG applied to DTG of 8-puzzle
Greedy abstract feature generation
(Jinnai&Fukunaga 2016)

Approach: maps each SAS+ variable $x_i$ to abstract feature $S_1$ and $S_2$ based on $x_i$'s domain transition graphs (nodes are values, edges are transitions)

1. Assign the minimal degree node to $S_1$
2. Add to $S_1$ the unassigned node which shares the most edges with node in $S_1$
3. Until $|S_1|$ reaches the half of the DTG, repeat step 2.
4. Assign all unassigned nodes to $S_2$

$$A_i(x_i) = \begin{cases} 1 & \text{if } x_i \in S_1 \\ 2 & \text{if } x_i \in S_2 \end{cases}$$

DTG of a variable $x_i$ represents the transition of the value

GreedyAFG applied to DTG of 8-puzzle
The performance of GreedyAFG
(Jinnai&Fukunaga 2016)

- Evaluated on IPC benchmarks
- Single multicore machine (8 cores)
- Pattern database heuristics
- AZHDA* using GreedyAFG achieved only a modest improvement over previous methods

<table>
<thead>
<tr>
<th></th>
<th>AZH/GreedyAFG</th>
<th>Zobrist</th>
<th>Abstraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Walltime (sec)</td>
<td>282</td>
<td>298</td>
<td>341</td>
</tr>
<tr>
<td>Speedup efficiency</td>
<td>0.797</td>
<td>0.766</td>
<td>0.729</td>
</tr>
<tr>
<td>Search overhead</td>
<td>0.01</td>
<td>0.01</td>
<td>0.34</td>
</tr>
<tr>
<td>Comm. overhead</td>
<td>0.62</td>
<td>0.86</td>
<td>0.40</td>
</tr>
</tbody>
</table>

→ What the problem of GreedyAFG?
Problem of GreedyAFG

- GreedyAFG incurs communication overhead if **ANY** of the abstract feature changes its value from the parent node (because a hash value is a function of a set of abstract features)

\[ AZ(s) = R_1[A_1(x_1)] \text{xor} R_2[A_2(x_2)] \text{xor} \ldots \text{xor} R_n[A_n(x_n)] \]

- If any of the \( A_i(x_i) \) changes, then the value of \( R_i[A_i(x_i)] \) changes, then \( AZ(s) \) changes (thus incurs communication overhead)
Problem of GreedyAFG

- GreedyAFG incurs communication overhead if **ANY** of the abstract feature changes its value from the parent node (because a hash value is a function of a set of abstract features)

Grey squares are the abstract features generated by GreedyAFG
Problem of GreedyAFG

- GreedyAFG incurs communication overhead if **ANY** of the abstract feature changes its value from the parent node (because a hash value is a function of a set of abstract features)

Grey squares are the abstract features generated by GreedyAFG
Problem of GreedyAFG

- GreedyAFG incurs communication overhead if \textbf{ANY} of the abstract feature changes its value from the parent node (because a hash value is a function of a set of abstract features)

Grey squares are the abstract features generated by GreedyAFG
Problem of GreedyAFG

- GreedyAFG incurs communication overhead if ANY of the abstract feature changes its value from the parent node (because a hash value is a function of a set of abstract features).

Grey squares are the abstract features generated by GreedyAFG.
Problem of GreedyAFG

- GreedyAFG incurs communication overhead if ANY of the abstract feature changes its value from the parent node (because a hash value is a function of a set of abstract features)

Grey squares are the abstract features generated by GreedyAFG
Problem of GreedyAFG

- GreedyAFG incurs communication overhead if **ANY** of the abstract feature changes its value from the parent node (because a hash value is a function of a set of abstract features)

This abstract feature **ALWAYS** changes its value!
Thus **ALL** node generations may incur communication overhead!
We propose *Fluency-based filtering* which ignores features which change their values too frequently.

We apply GreedyAFG to the rest of the features.
Fluency-Based Filtering

- We define \textit{fluency} of a variable $x$

$$
fluency(x) := \frac{\text{number of ground actions which change the value of } x}{\text{total number of ground actions}}
$$

- Our implementation ignores variables whose fluency is in the top 30% of the variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Fluency</th>
</tr>
</thead>
<tbody>
<tr>
<td>handempty</td>
<td>1.0</td>
</tr>
<tr>
<td>not handempty</td>
<td>0.5</td>
</tr>
<tr>
<td>holding(a)</td>
<td>0.5</td>
</tr>
<tr>
<td>ontable(a)</td>
<td>0.5</td>
</tr>
<tr>
<td>holding(b)</td>
<td>0.5</td>
</tr>
<tr>
<td>ontable(b)</td>
<td>0.5</td>
</tr>
<tr>
<td>on(a,b)</td>
<td>0.5</td>
</tr>
<tr>
<td>on(b,a)</td>
<td>0.5</td>
</tr>
</tbody>
</table>
In fact, variables with high fluency are common in wide range of domains.

For example, in domains modeling agent-environment, variables modeling the state of agents tend to have high fluency.

- blocks world
- gripper
- sokoban
Operator-based Zobrist hashing

- Zobrist hashing incurs significant communication overhead
- Method: Preinitialize the random table so that the given operator does not change the hash value

\[
\begin{align*}
H(s) &= 1011 \\
H(a) &= 0000 \\
H(s') &= H(s) \text{ xor } H(a) \\
&= 1011 \\
&= H(s) 
\end{align*}
\]
Dynamic AHDA*

- In previous work, AHDA* used a fix threshold to the number of the abstract nodes
- This leads to suboptimal performance to instance set with varying difficulty (especially in distributed memory cluster)
- Dynamic AHDA* set the threshold according to the size of the problem difficulty
- Our current implementation set the threshold of the total number of features in the abstract state space to be 30% of the total number of features in the problem instance
Experiments

- We evaluated HDA* variants on IPC benchmarks (21 instances)
- 48 cores (6 machines with 8 cores)
- Based on FastDownward and MPICH3
- merge&shrink heuristic (LFPA)
Experiments

- FAZHDA*: AZHDA* using GreedyAFG with fluency filtering
- OZHDA*: Operator-based Zobrist hashing
- DAHDA*: Dynamic AHDA*
- GAZHDA*: AZHDA* using GreedyAFG without fluency filtering

$\rightarrow$ FAZHDA* outperformed GAZHDA* and other HDA* variants
Summary of Paper

GreedyAFG (GAZHDA*)

Zobrist hashing (ZHDA*)

State abstraction (AHDA*)
Summary of Paper

- We proposed Fluency-based filtering for AZHDA* which ignores variables which frequently change their values.
• We proposed Fluency-based filtering for AZHDA* which ignores variables which frequently change their values.

• We proposed Operator-based Zobrist hashing which generates Zobrist hashing bitstrings that ensures reduced communication overhead.
Summary of Paper

- We proposed Fluency-based filtering for AZHDA* which ignores variables which frequently change their values.
- We proposed Operator-based Zobrist hashing which generate Zobrist hashing bitstrings that ensures reduced communication overhead.
- We implemented Dynamic AHDA* to determine the size of abstract state space according to the instance difficulty.
We proposed Fluency-based filtering for AZHDA* which ignores variables which frequently change their values.

We proposed Operator-based Zobrist hashing which generate Zobrist hashing bitstrings that ensures reduced communication overhead.

We implemented Dynamic AHDA* to determine the size of abstract state space according to the instance difficulty.

AZHDA*+Fluency-based filtering performed the best.
Operator-based Zobrist hashing

\[ Z(s) = R[x_1] \text{xor } R[x_2] \text{xor } \ldots \text{xor } R[x_n] \]

- Let \( s' \) be the child node of \( s \) using operator \( a \)
- Assume all effects in add&delete effect take place
- Zobrist hash value of \( s' \) is

\[ Z(s') = Z(a) \text{xor } Z(s) \]

where \( Z(a) = R[p_1] \text{xor } R[p_1] \text{xor } \ldots \text{xor } R[p_1] \) for all propositions \( p_i \) in add&delete effect in \( a \)

\[ \rightarrow \text{If } Z(a) = 0, \text{ then } Z(s') = Z(s) \]
Operator-based Zobrist hashing

\[ Z(s) = R_1[x_1] \oplus R_2[x_2] \oplus \ldots \oplus R_n[x_n] \]

→ If \( Z(a) = 0 \), then \( Z(s') = Z(s) \)

1. Select one operator
2. Modify a value of \( R_i[x_i] \) value without a flag so that \( Z(a) = 0 \)
3. Set flags to all propositions in a so that they won't be modified later
4. Repeat from 1

- We select the operator with least preconditions (future work)
Dynamic AHDA* construction

- Follows the construction of Structured Duplicate Detection (SDD) (Zhou&Hansen 2007)
- Idea: Add an atom group which preserve the locality the best

- Select an atom group (= SAS+ variable) which retains the maximum-degree of the abstract state graph smallest compared to the graph size
- Add the atom group into the abstract state representation
- Terminate if the size of the abstract state reaches a threshold Nmax
- Abstract state is represented using the selected atom groups, and the projection from a state to an abstract state simple ignores all features not in the atom groups
● Each thread has its own open/closed list
● Each thread sends generated nodes to its owner (assigned by the hash function)
● Other than sending/recieving each thread runs A* search
Summary of Paper

- GreedyAFG generates abstract features for Abstract Zobrist hashing but fails to reduce communication overhead due to variables with high fluency.
- We introduced a notation of fluency and proposed Fluency-based filtering which ignores variables which frequently change their values.
- We proposed Operator-based Zobrist hashing which generate Zobrist hashing bitstrings that ensures reduced communication overhead.
- We implemented Dynamic AHDA* to determine the size of abstract state space according to the instance difficulty.
- AZHDA*+Fluency-based filtering performed the best.
We define a metric to estimate the walltime efficiency $\text{eff}_{\text{esti}}$ and actual walltime efficiency:

$$
\text{eff}_{\text{esti}} := \frac{1}{(1+cCO)(1+SO)}
$$