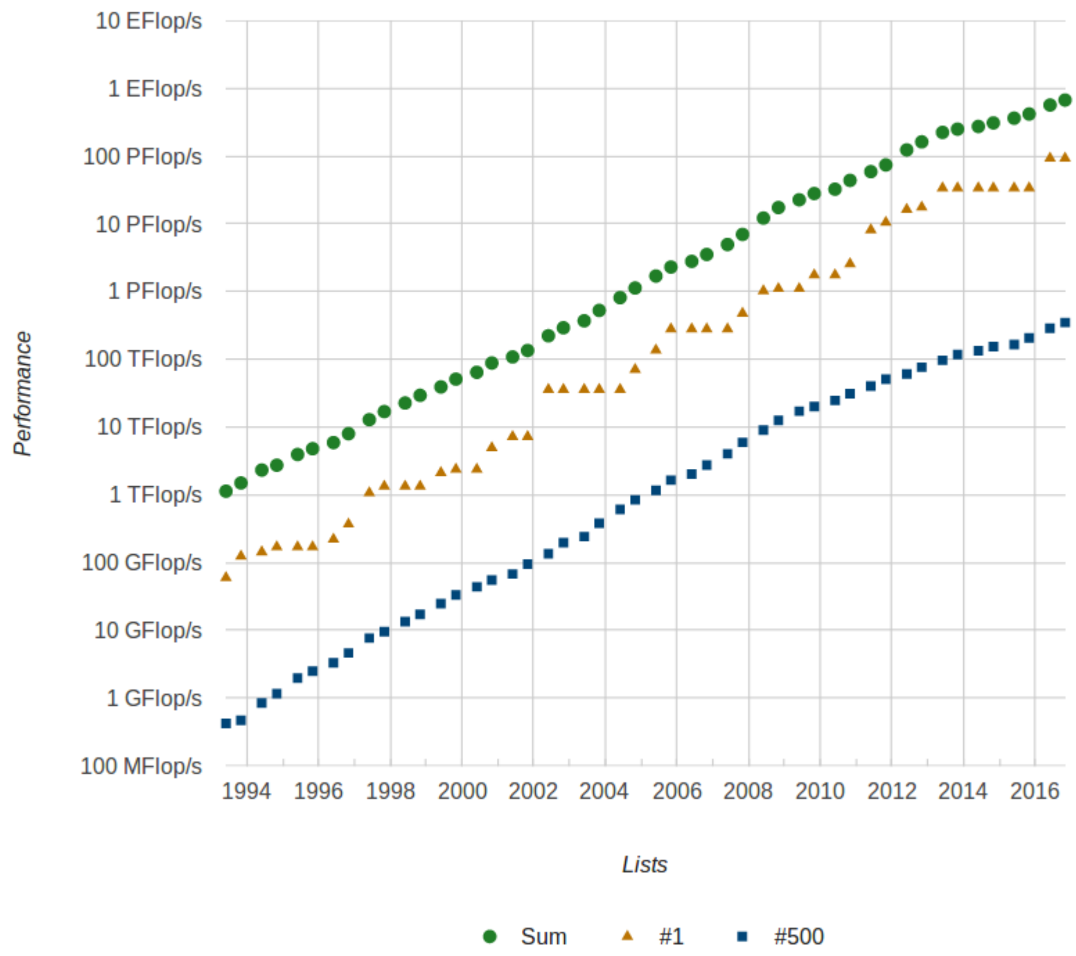
A Graph Partitioning-Based Approach for Parallel Best-First Search

Yuu Jinnai[1]   Alex Fukunaga[2]

The University of Tokyo[1,2]
RIKEN Center for Advanced Intelligence Project[1]

*This presentation is based on Section 5 of the journal paper (Jinnai&Fukunaga'17)

Why Parallel Algorithms?



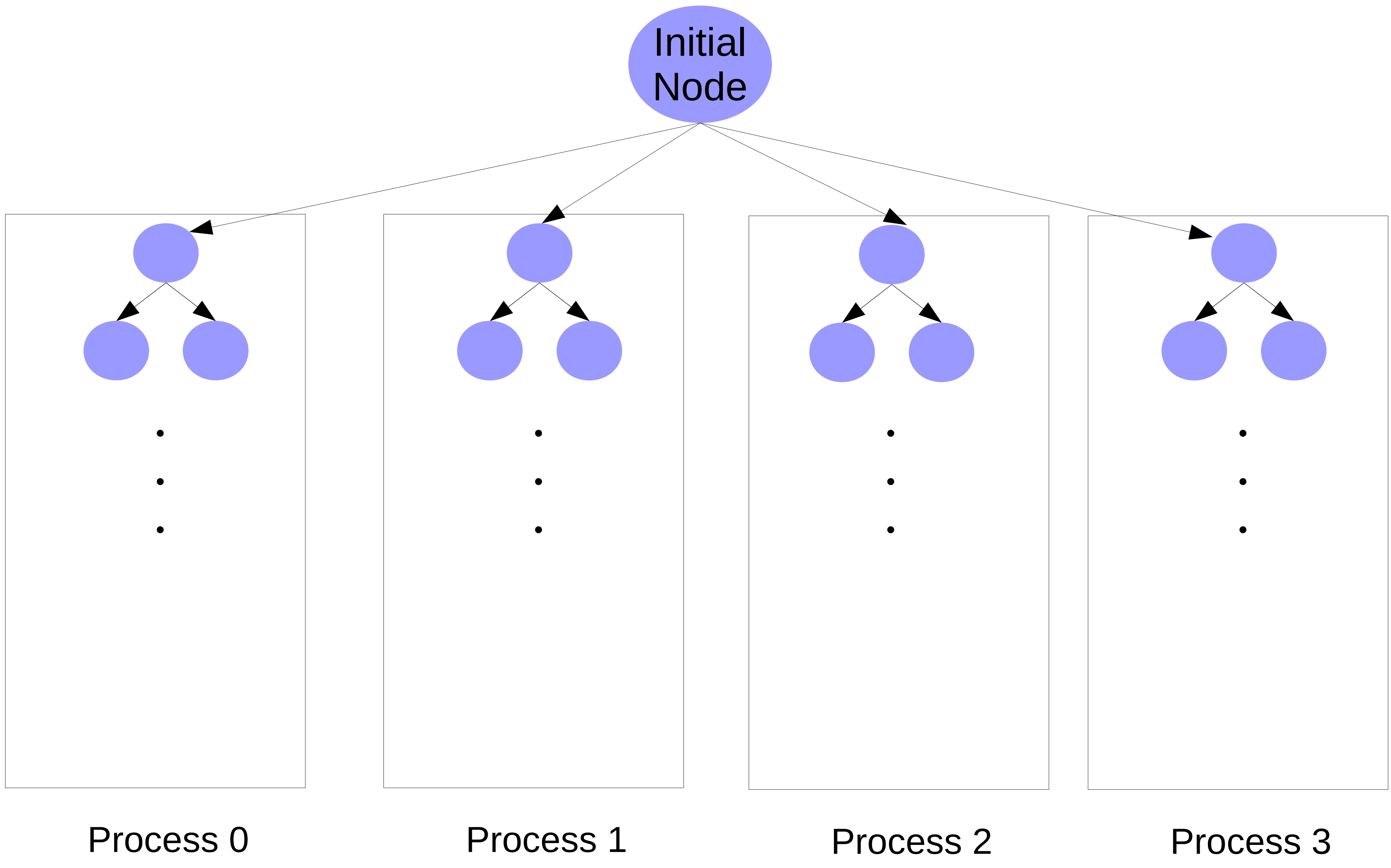
Supercomputer (TOP 500)

https://www.top500.org/statistics/perfdevel/

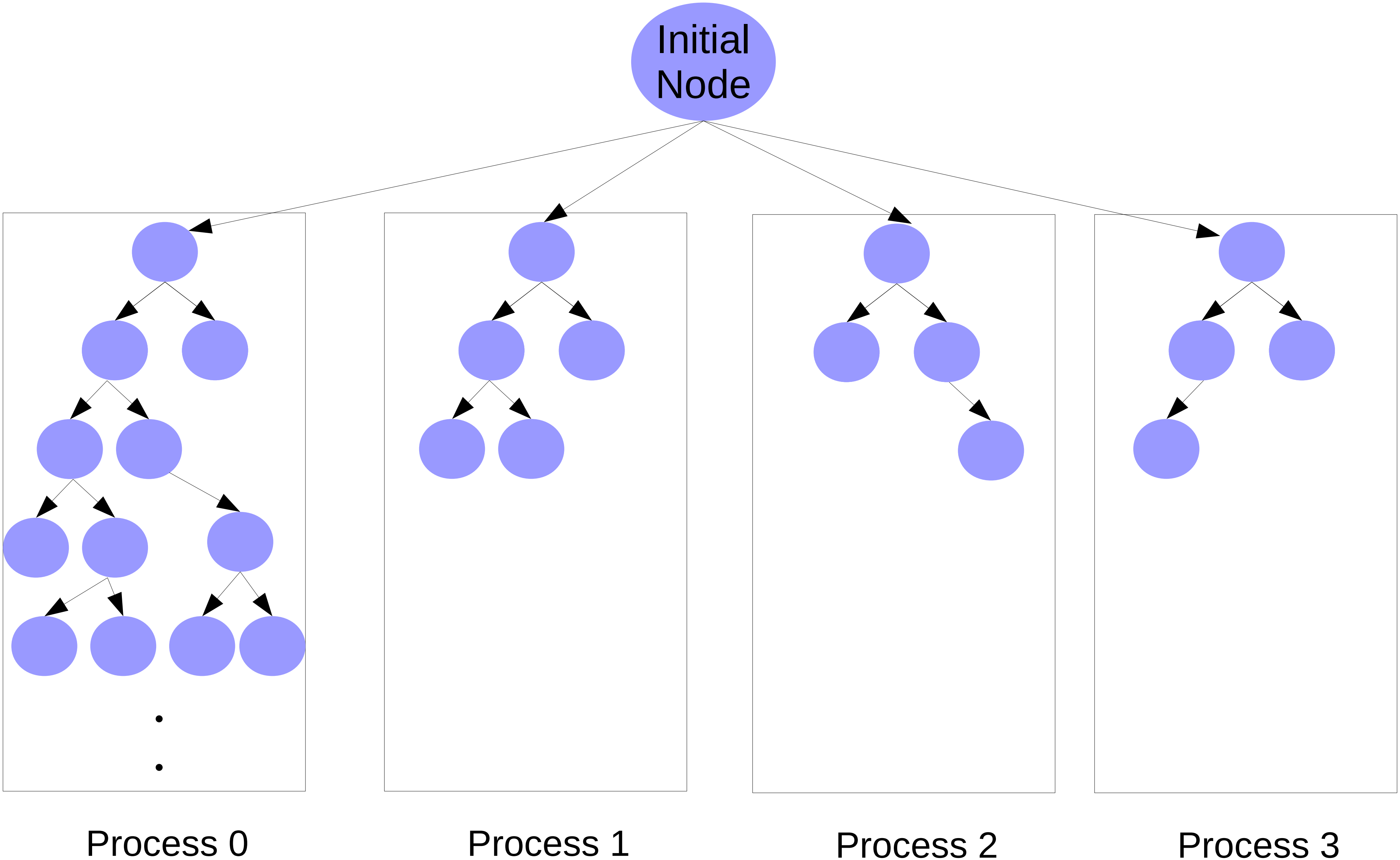
Why Parallel A*?

- Larger aggregated memory (with distributed environment)
  - potentially able to solve instances which sequential A* cannot solve due to memory limitation
- Walltime speedup

Subtree Distribution

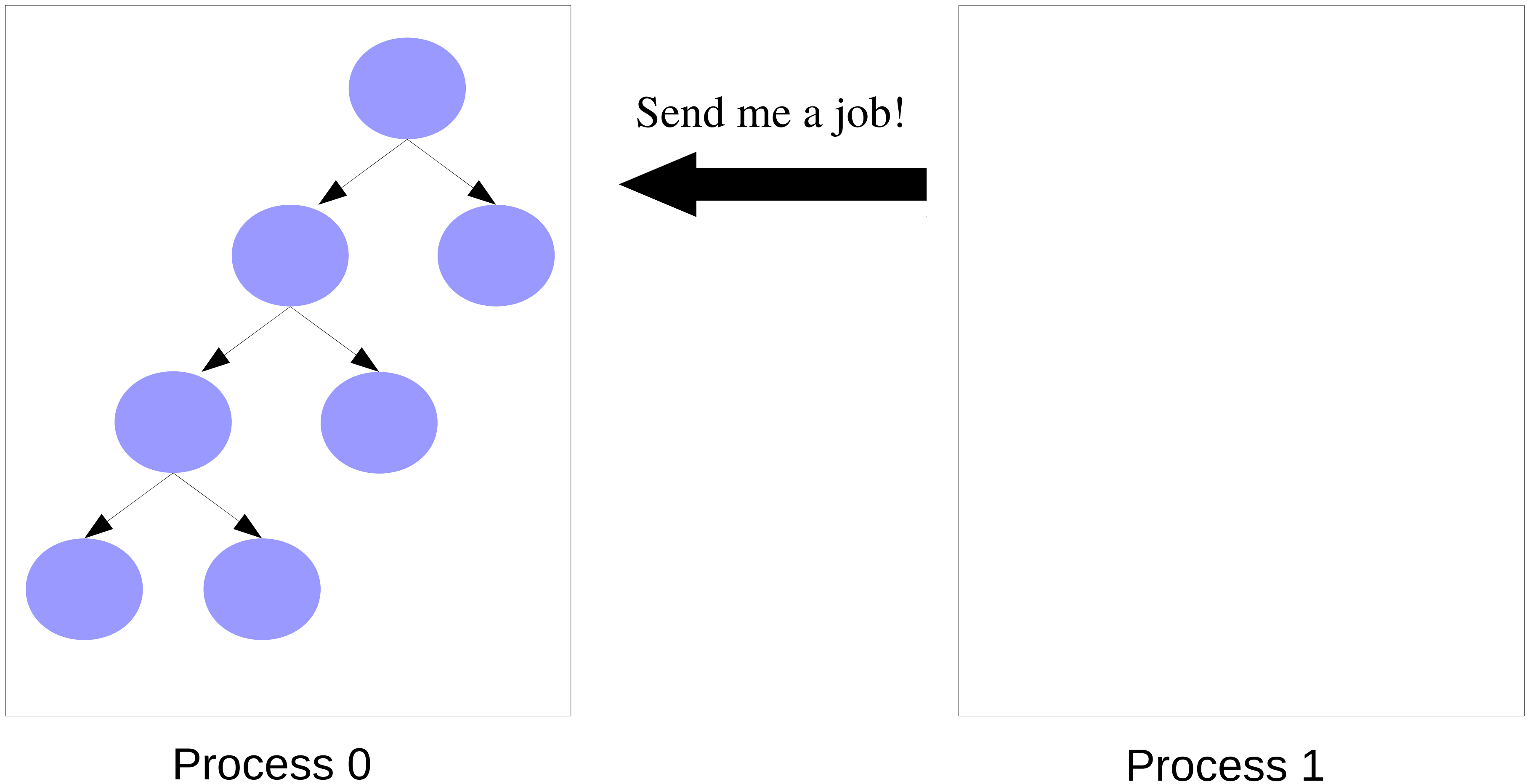


Subtree Distribution



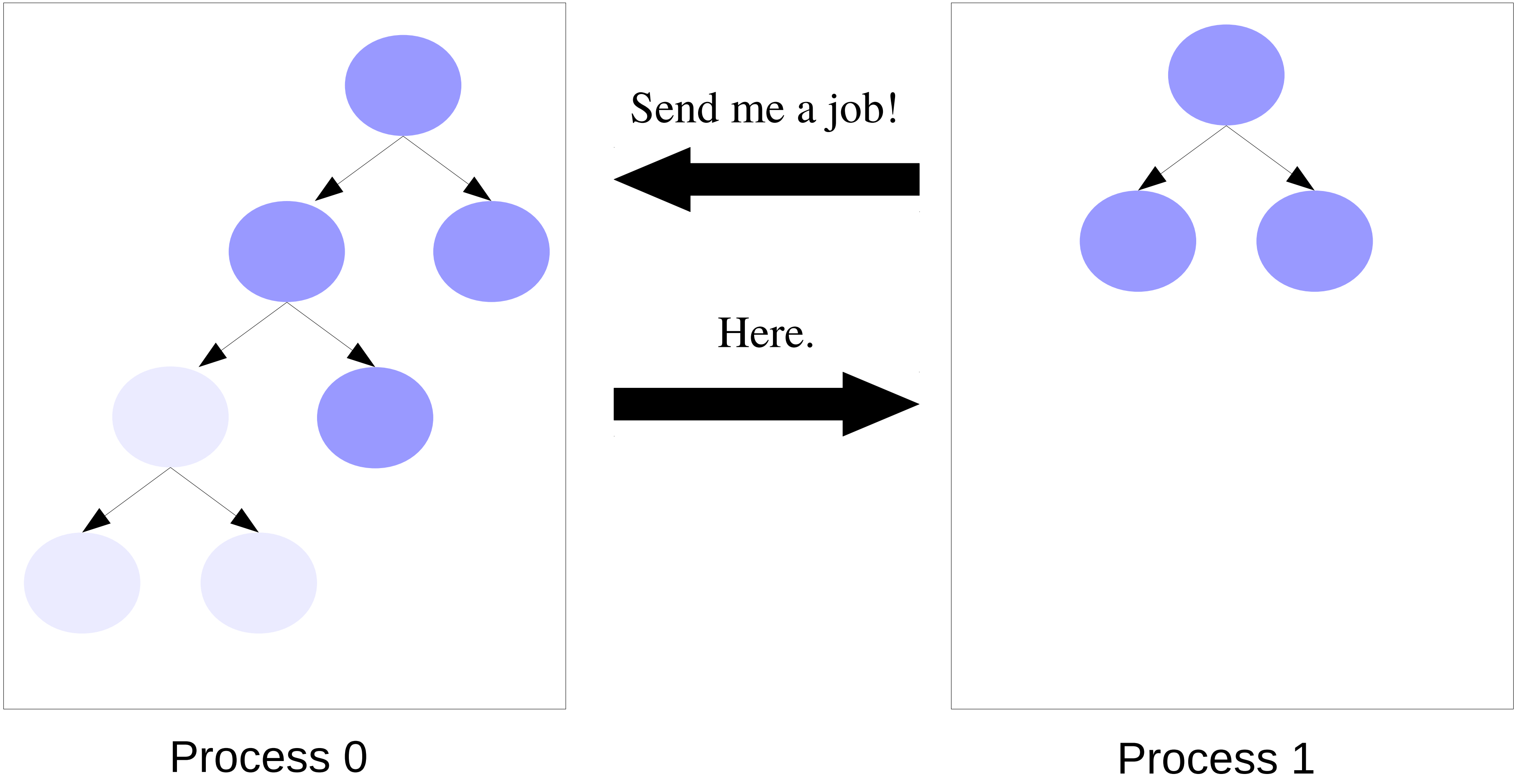
Dynamic Load Balancing Approach

Work Stealing Approach (Rao&Kumar'87)



Dynamic Load Balancing Approach

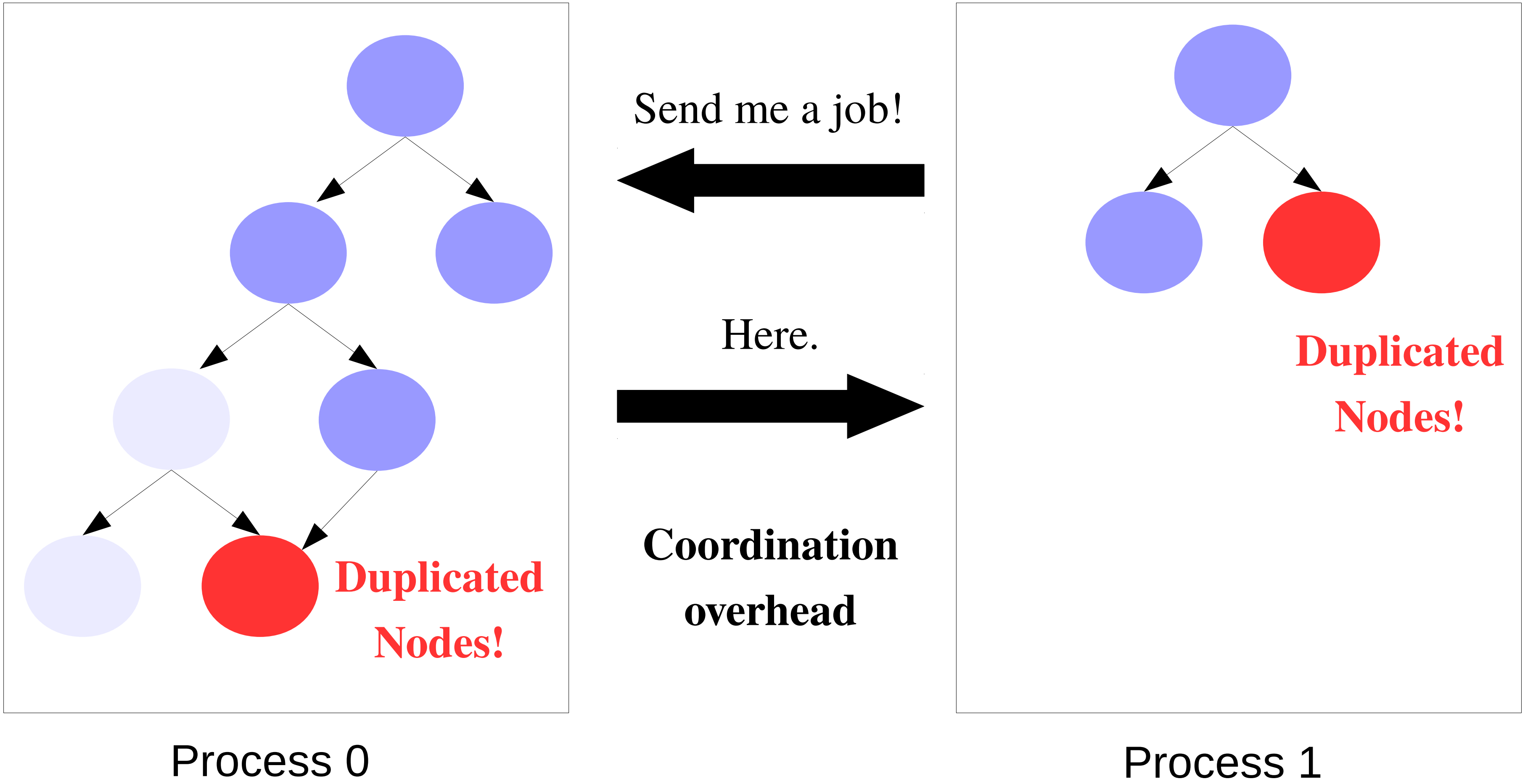
Work Stealing Approach (Rao&Kumar'87)



Dynamic Load Balancing Approach

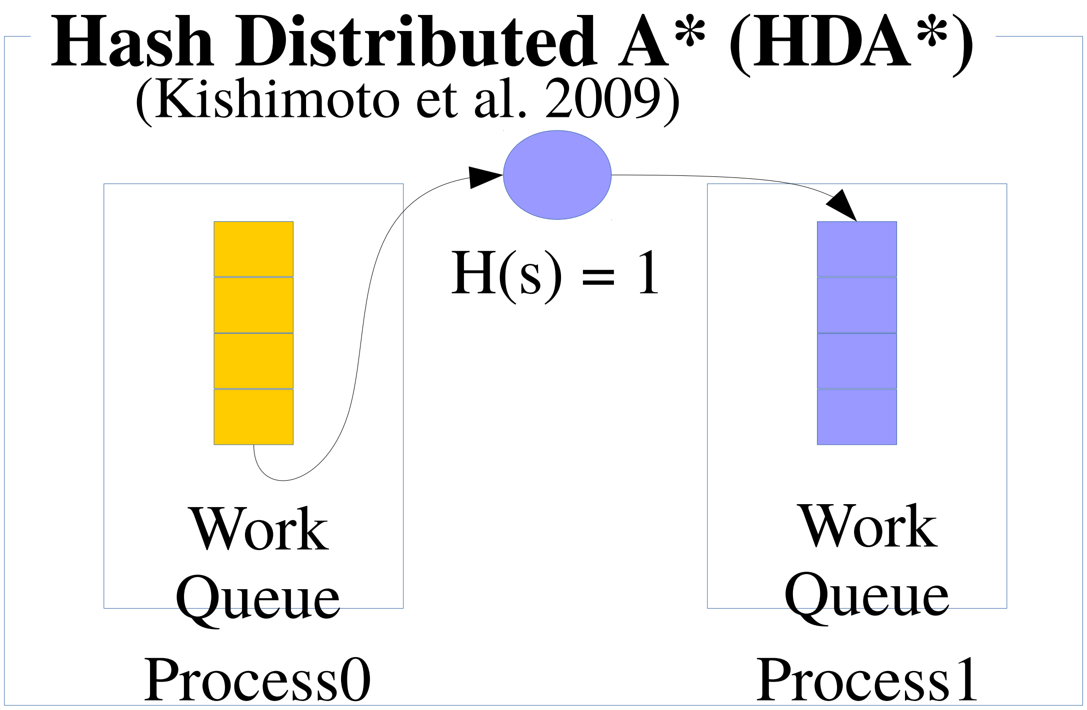
Work Stealing Approach (Rao&Kumar'87)

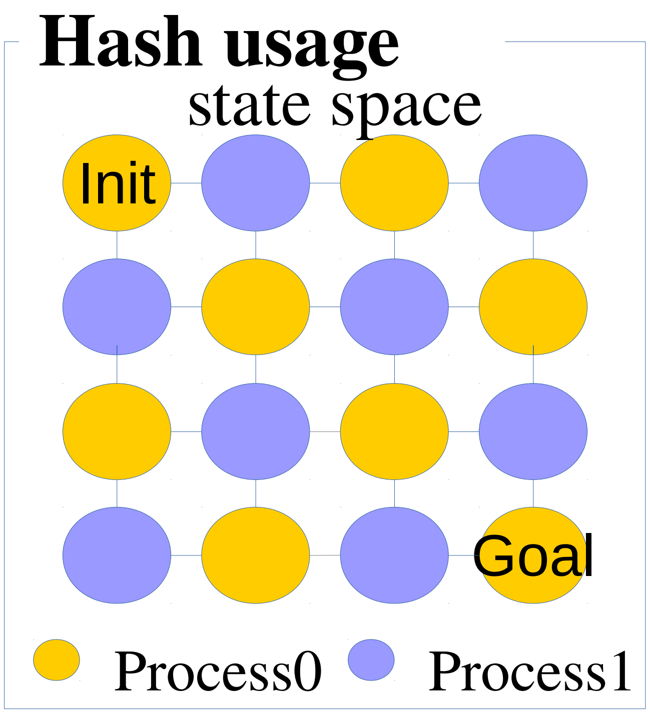
- Incurs duplicated nodes (for graph search)
- Incurs coordination overhead



Static Load Balancing Approach (Hashing)

- A global hash function assigns each state to a unique process
- A process sends generated nodes to their owner processes

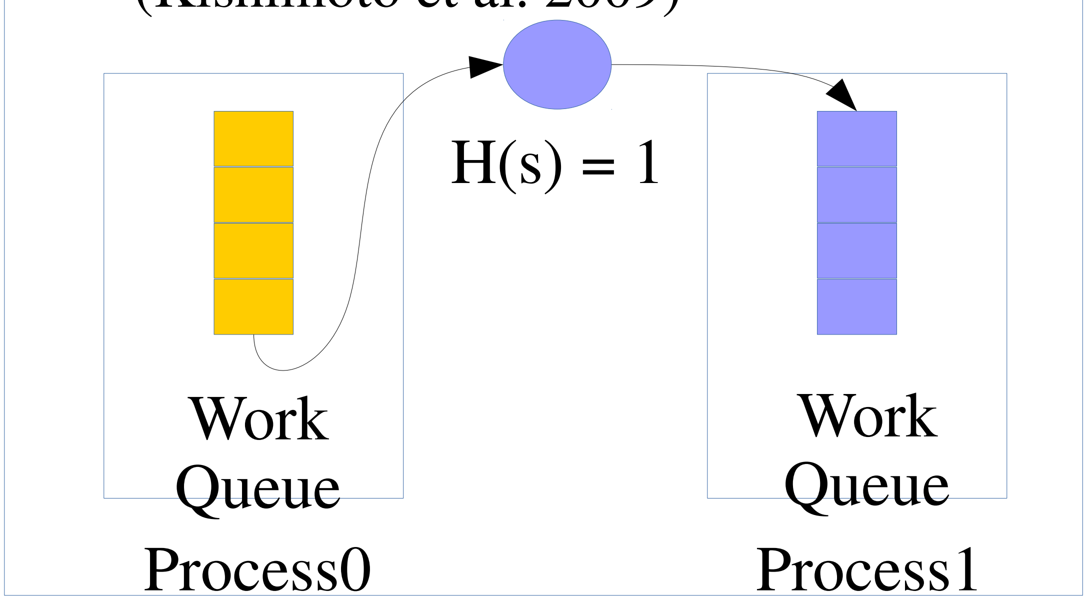




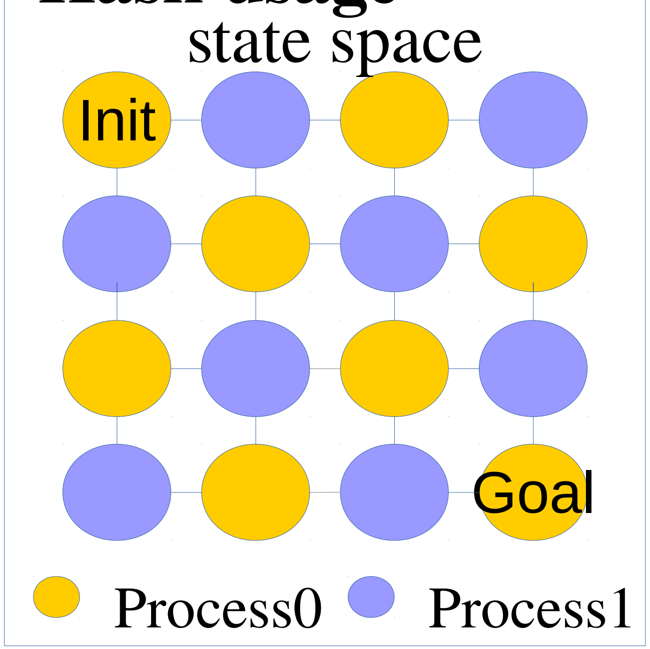
Static Load Balancing Approach (Hashing)

As HDA* relies on the hash function for load balancing,
the choice of hash function is significant to its performance!

Hash Distributed A* (HDA*)
(Kishimoto et al. 2009)



Hash usage

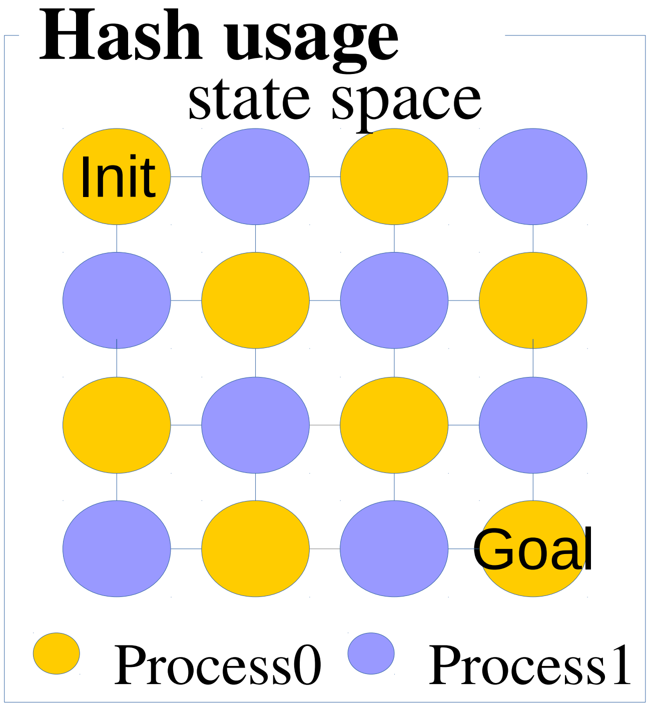


Hash Function for HDA*

- State (s) is given as a set of features x_i:

  state s = (x_1, x_2,...,x_n)

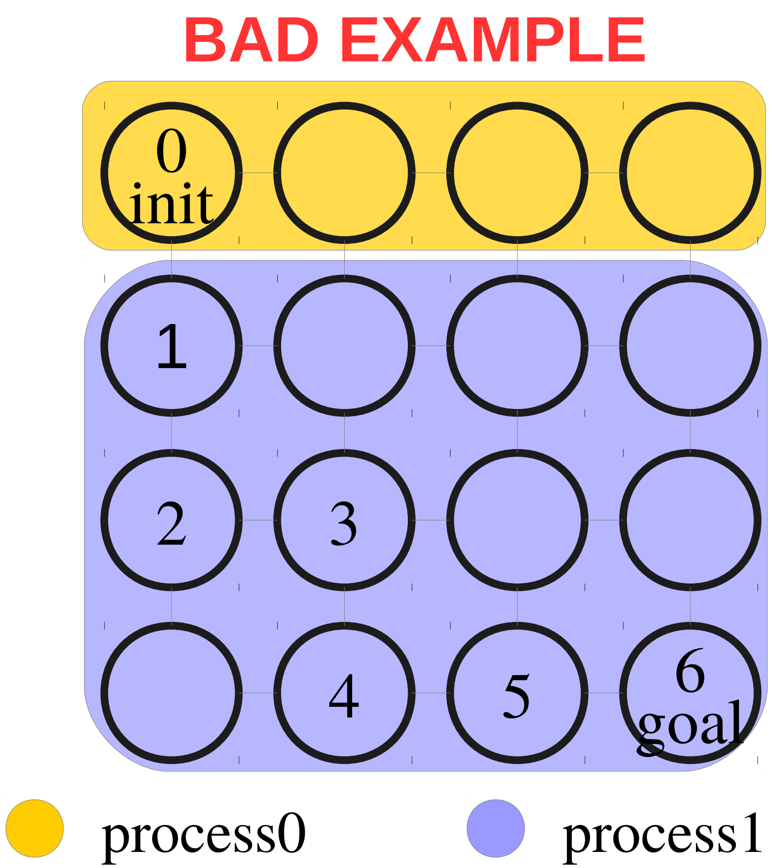
- Given a state s, a hash function $H(s)$ returns the process which owns the state s



Properties of Hash Function

We want $H(s)$ to be balanced

$\rightarrow$ load balance (LB)

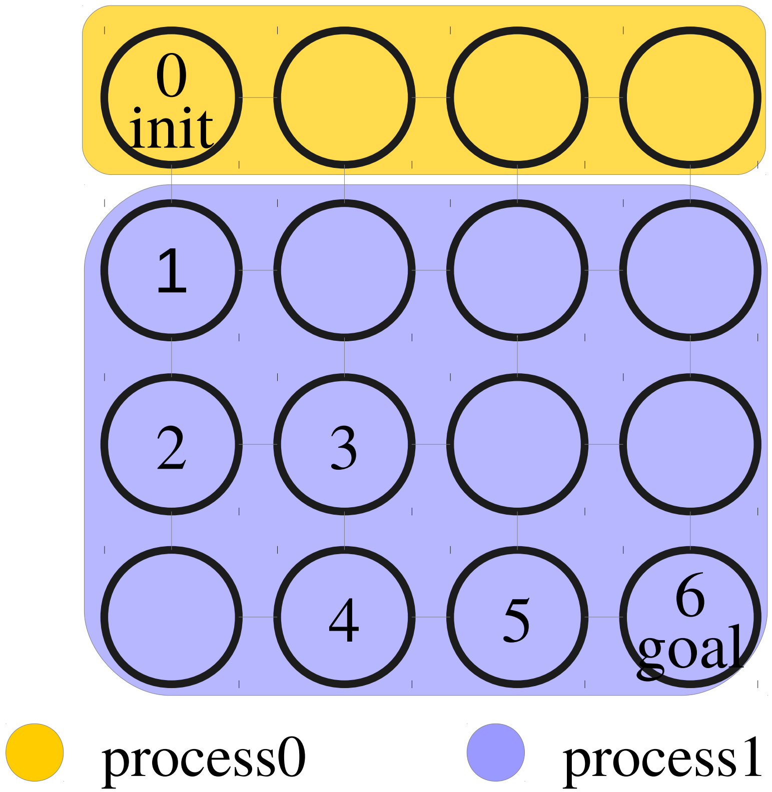


Properties of Hash Function

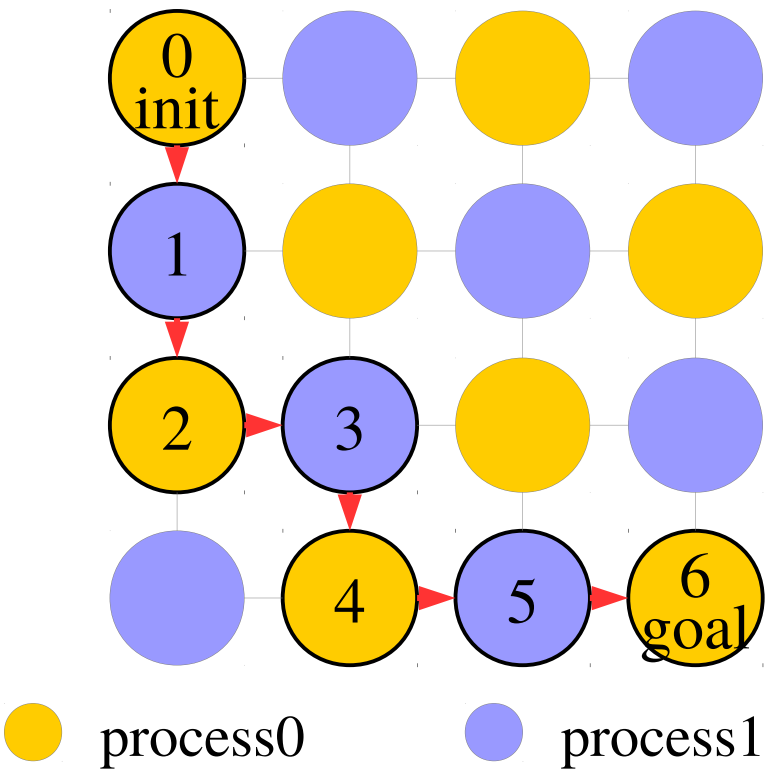
We want $H(s)$ to be balanced
$\rightarrow$ load balance (LB)

We want the value of $H(s)$ to not change frequently
$\rightarrow$ communication overhead (CO)

BAD EXAMPLE



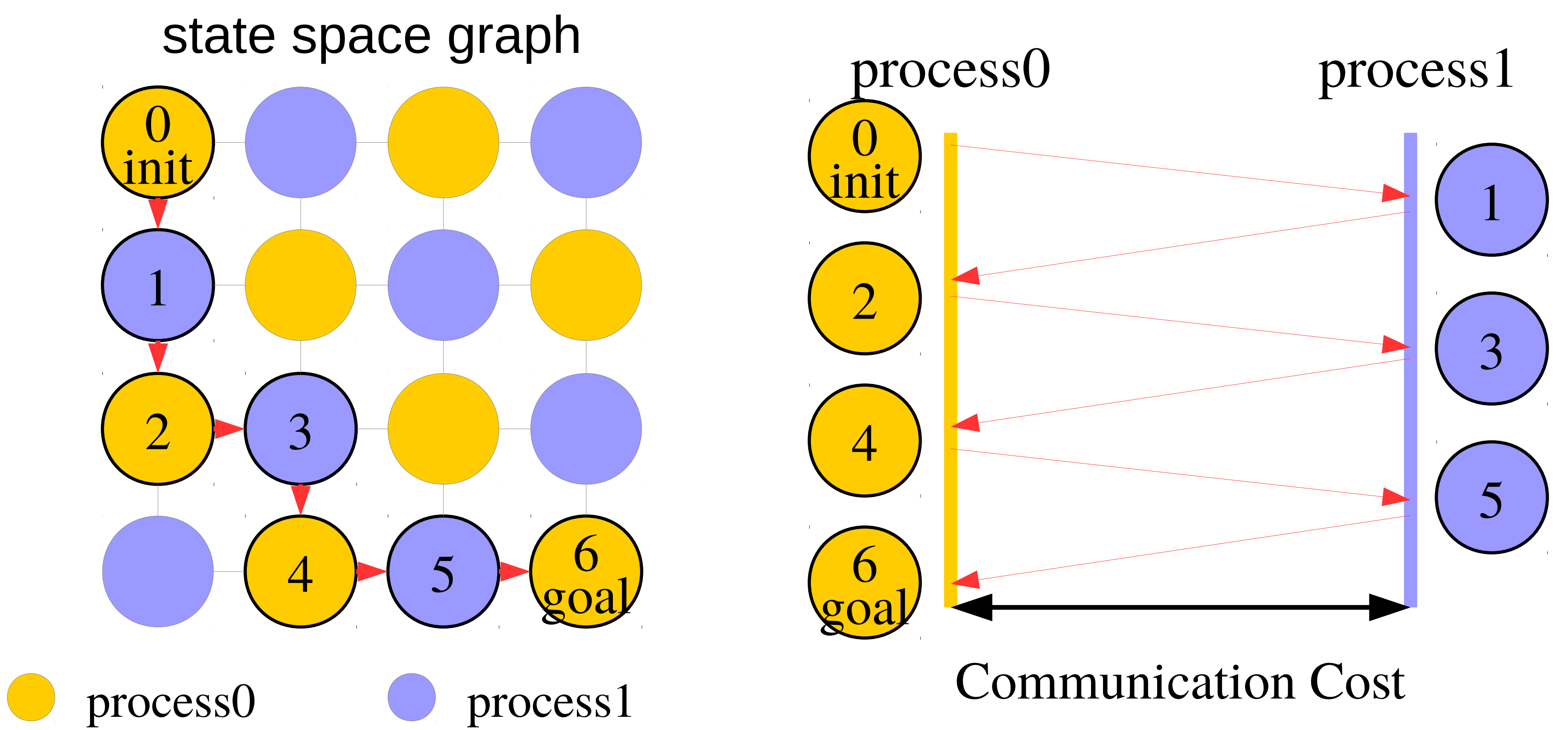
BAD EXAMPLE



Zobrist Hashing (ZHDA*)

Zobrist (1970); Kishimoto et al. (2009)

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

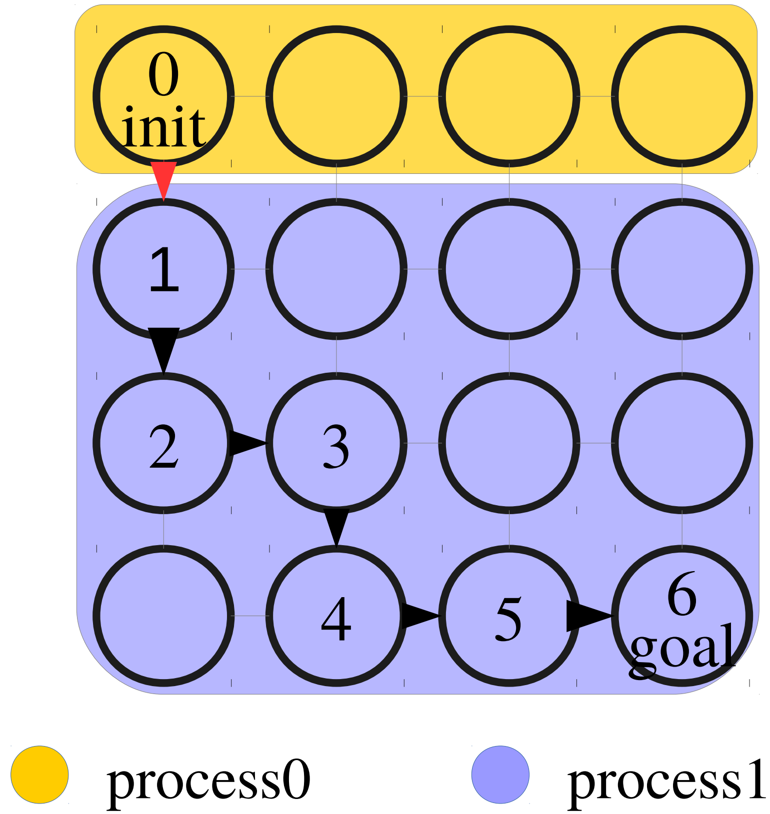


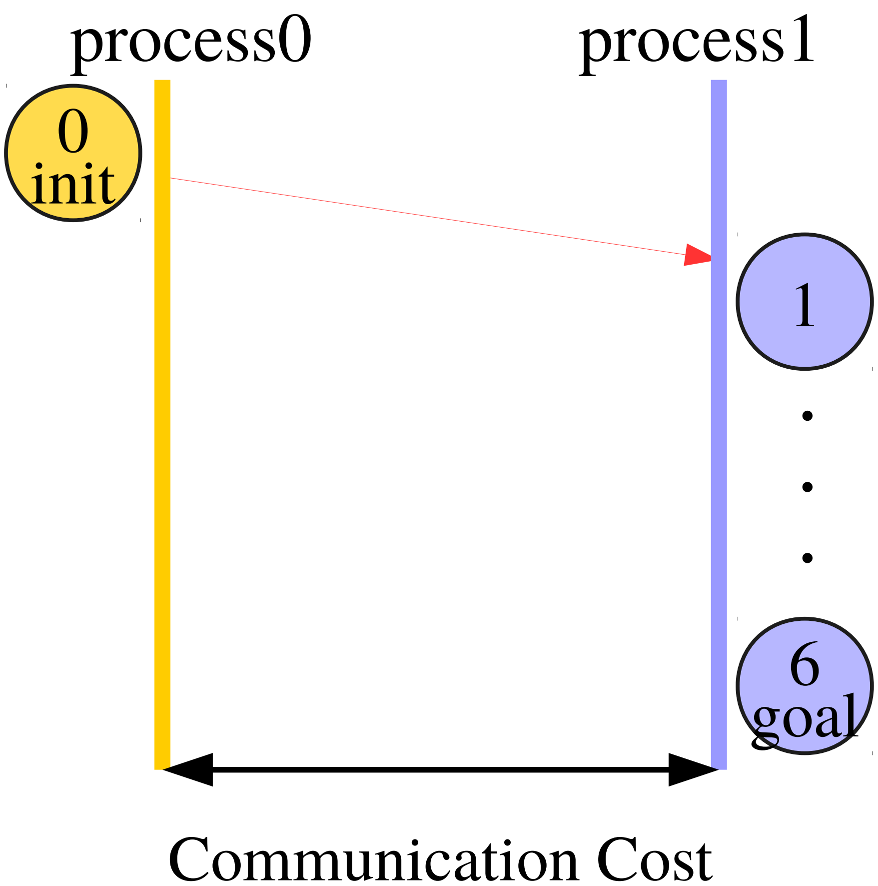
State abstraction (AHDA*)

Burns et al. (2010)

- Strenght: low communication overhead
- Limitation: worse load balance

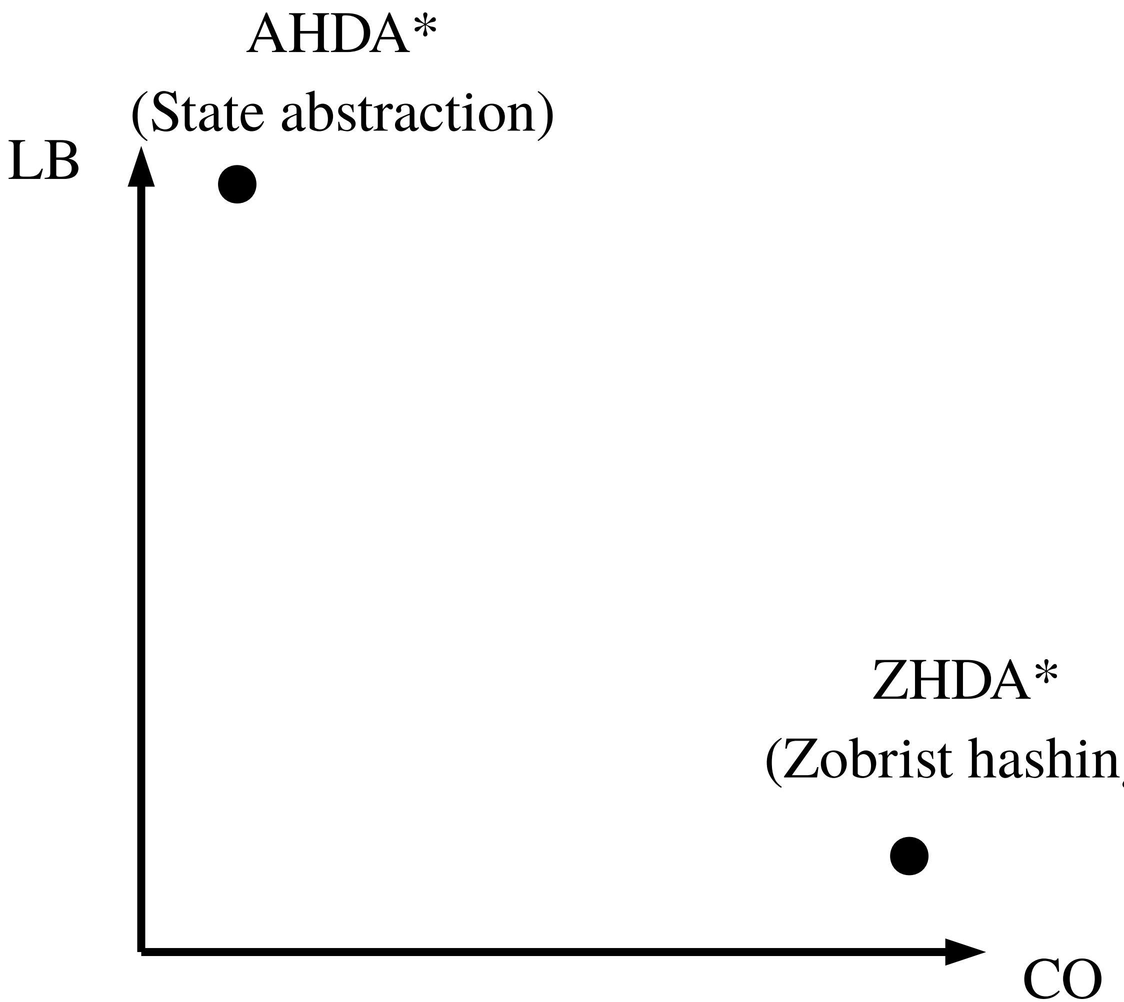
state space graph





Two Extremes

Both ZHDA* and AHDA* have a clear weakness and do not scale well in large-scale cluster

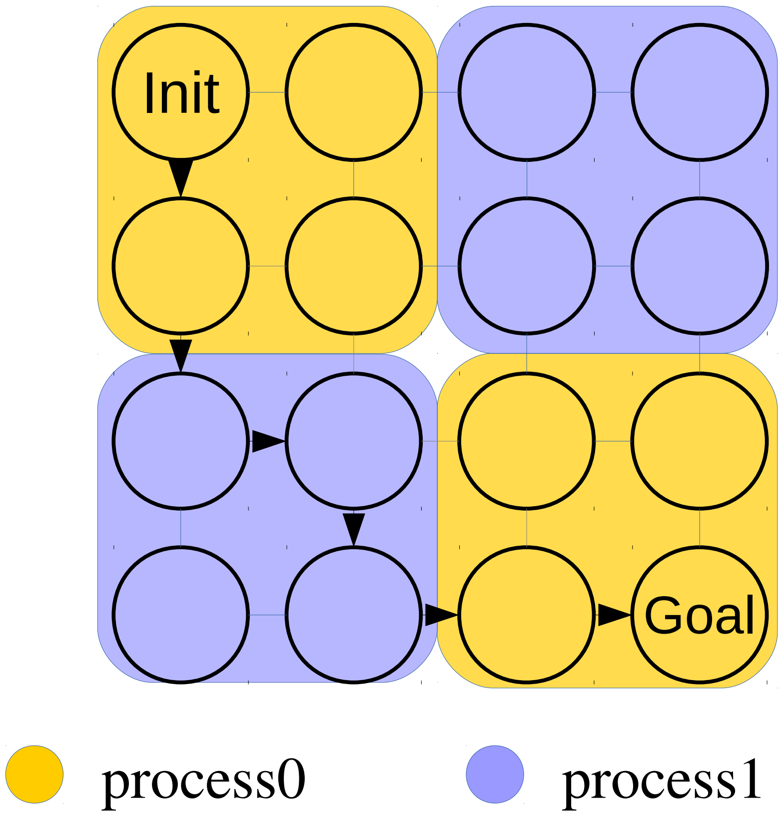


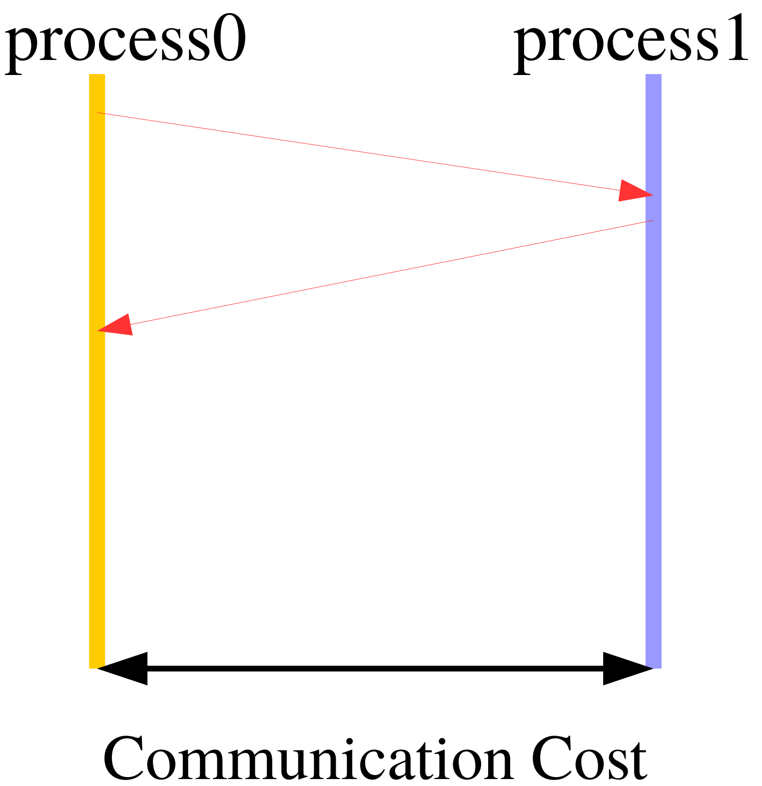
Abstract Zobrist Hashing (AZHDA*)

Jinnai&Fukunaga (2016)

- A hybrid of Abstraction and Zobrist hashing
- Can balance the trade-off of LB and CO by a parameter

state space graph

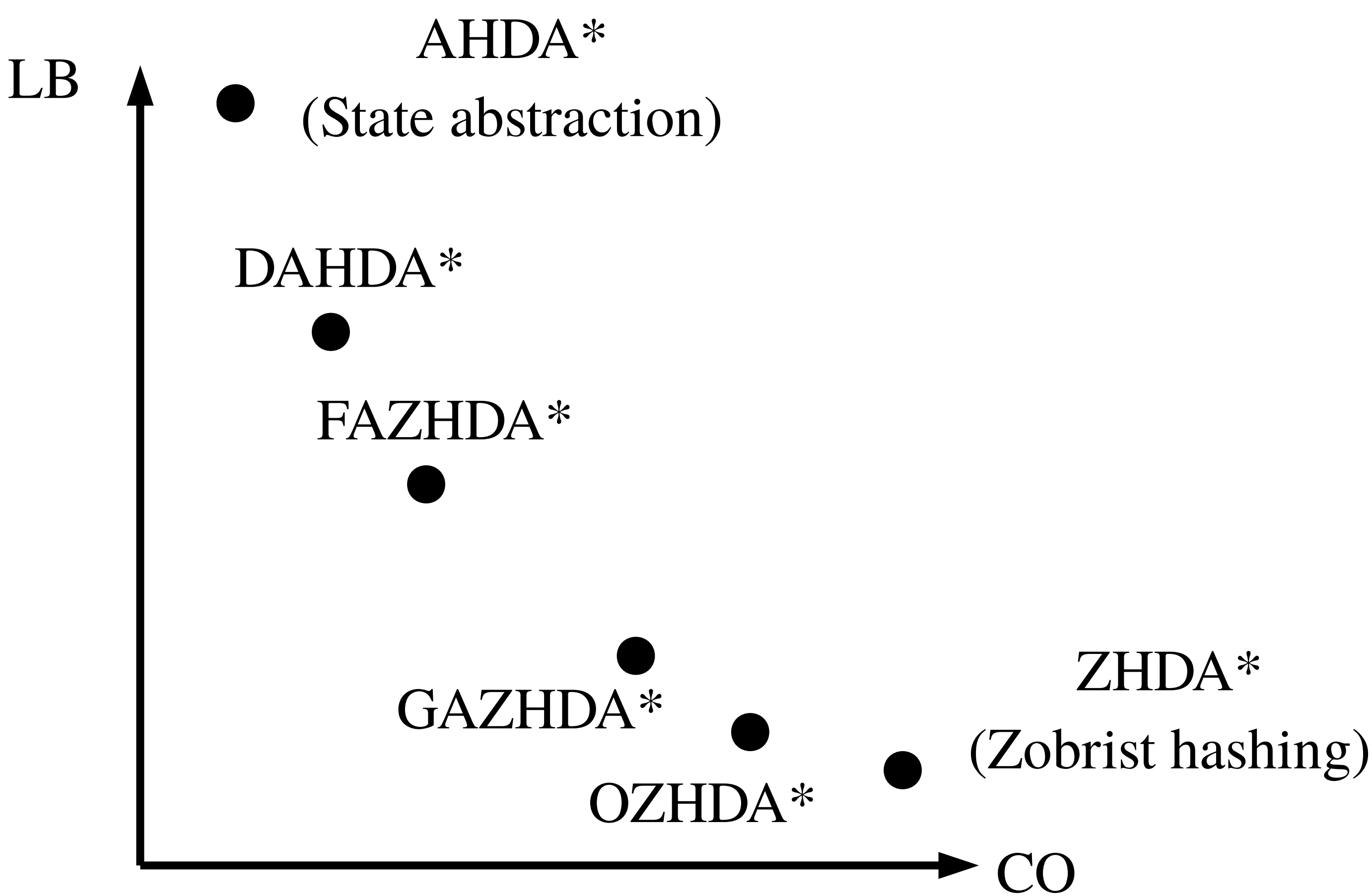




Variants of HDA*

Jinnai&Fukunaga (2016)

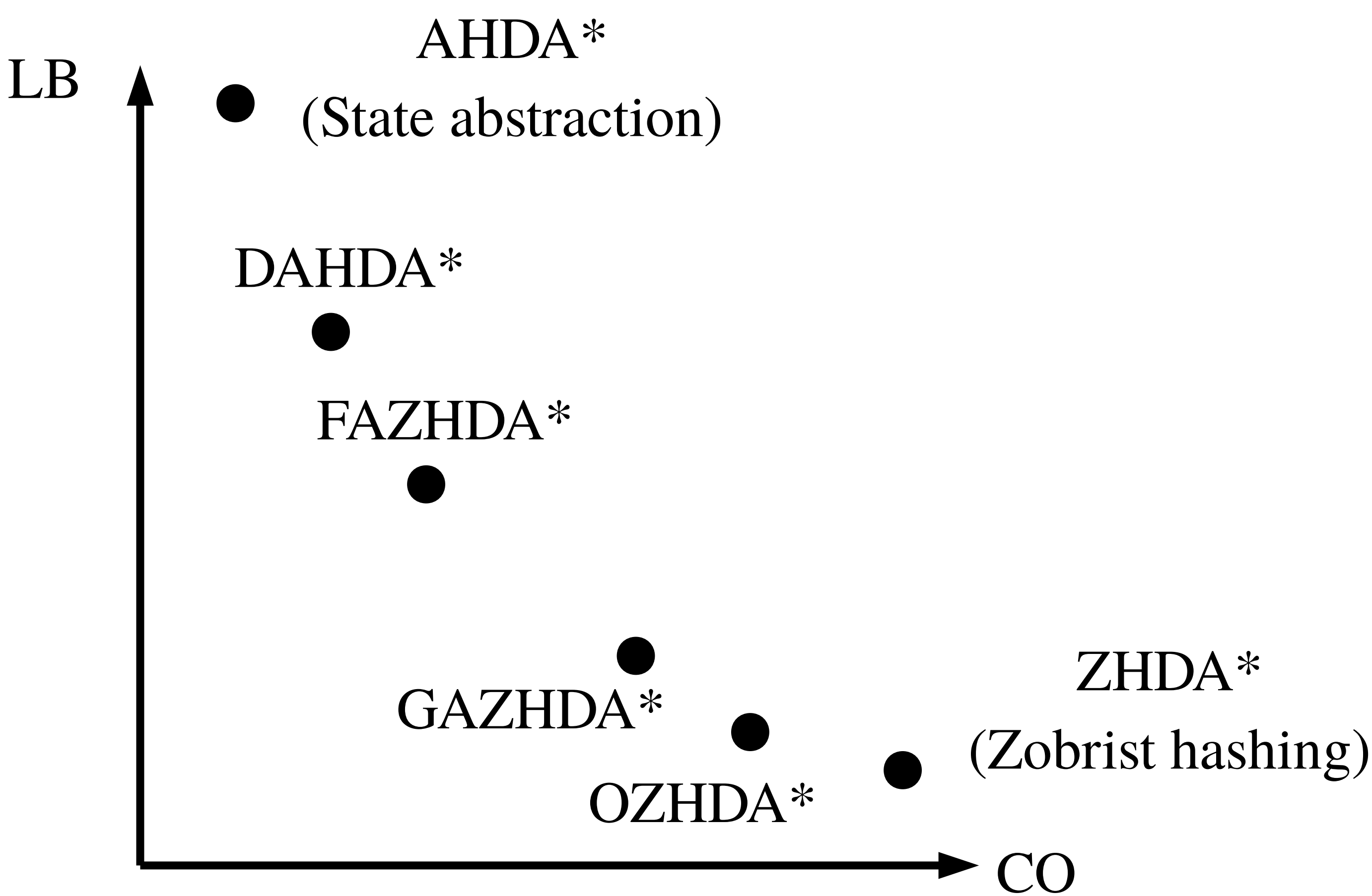
- Bunch of variants… so which one is the best and **why**?



Variants of HDA*

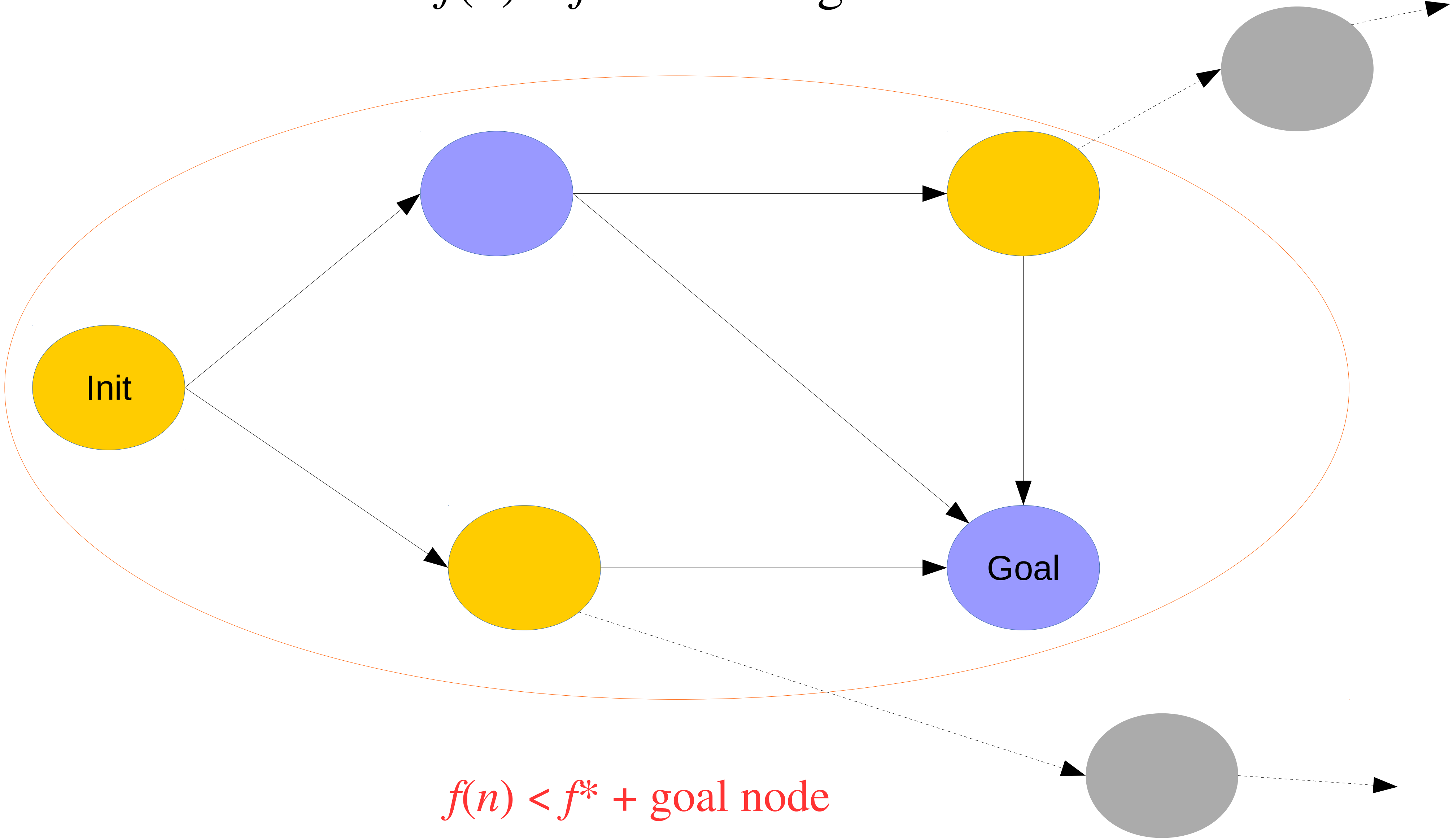
Jinnai&Fukunaga (2016)

- Bunch of variants… so which one is the best and **why**?
- In this work we developed a model for HDA* so that we can evaluate which method is likely to perform the best

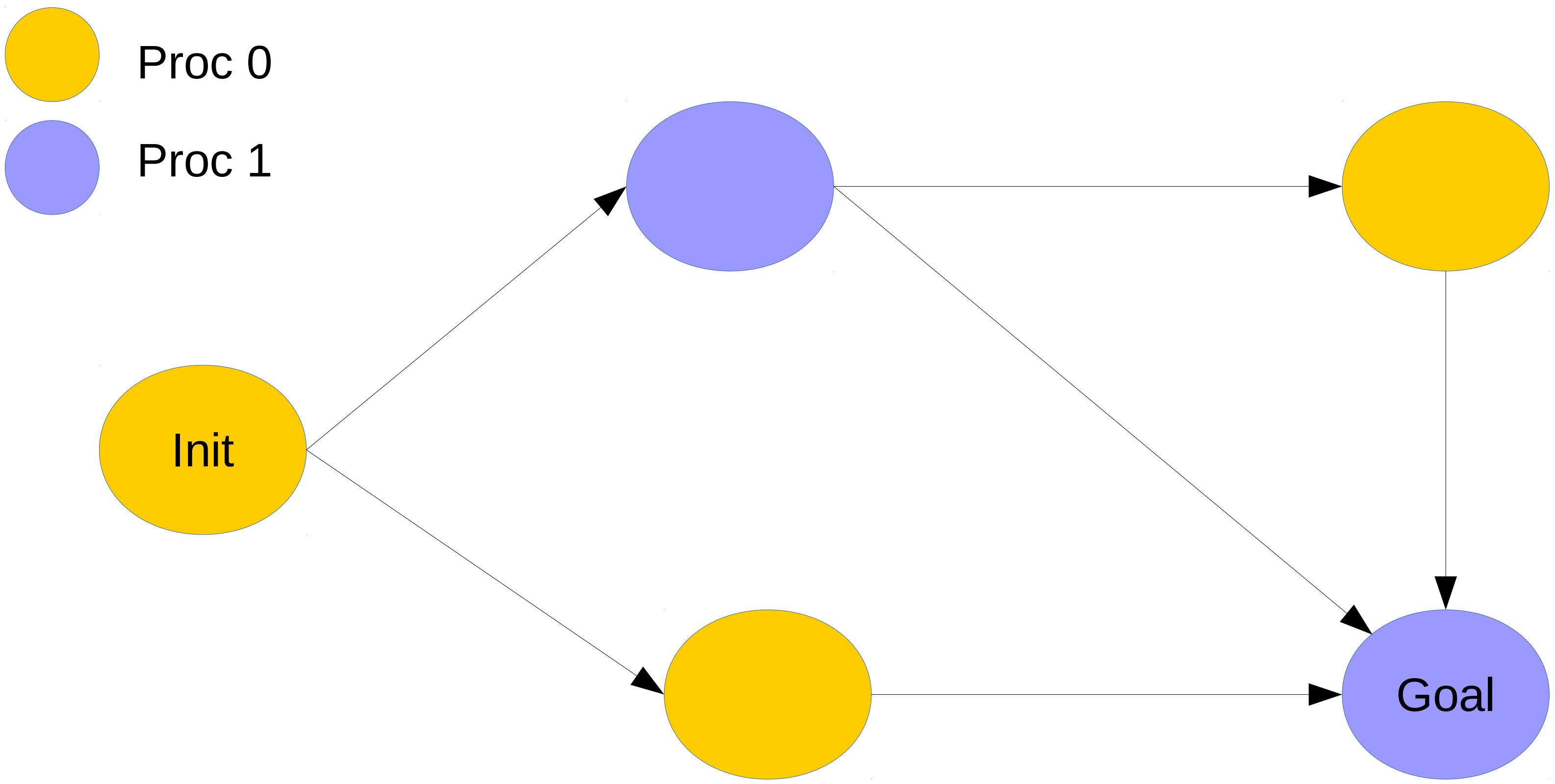


Workload Graph

- A subset of state-space graph which includes node n iff $f(n) < f^*$ or n is a goal node

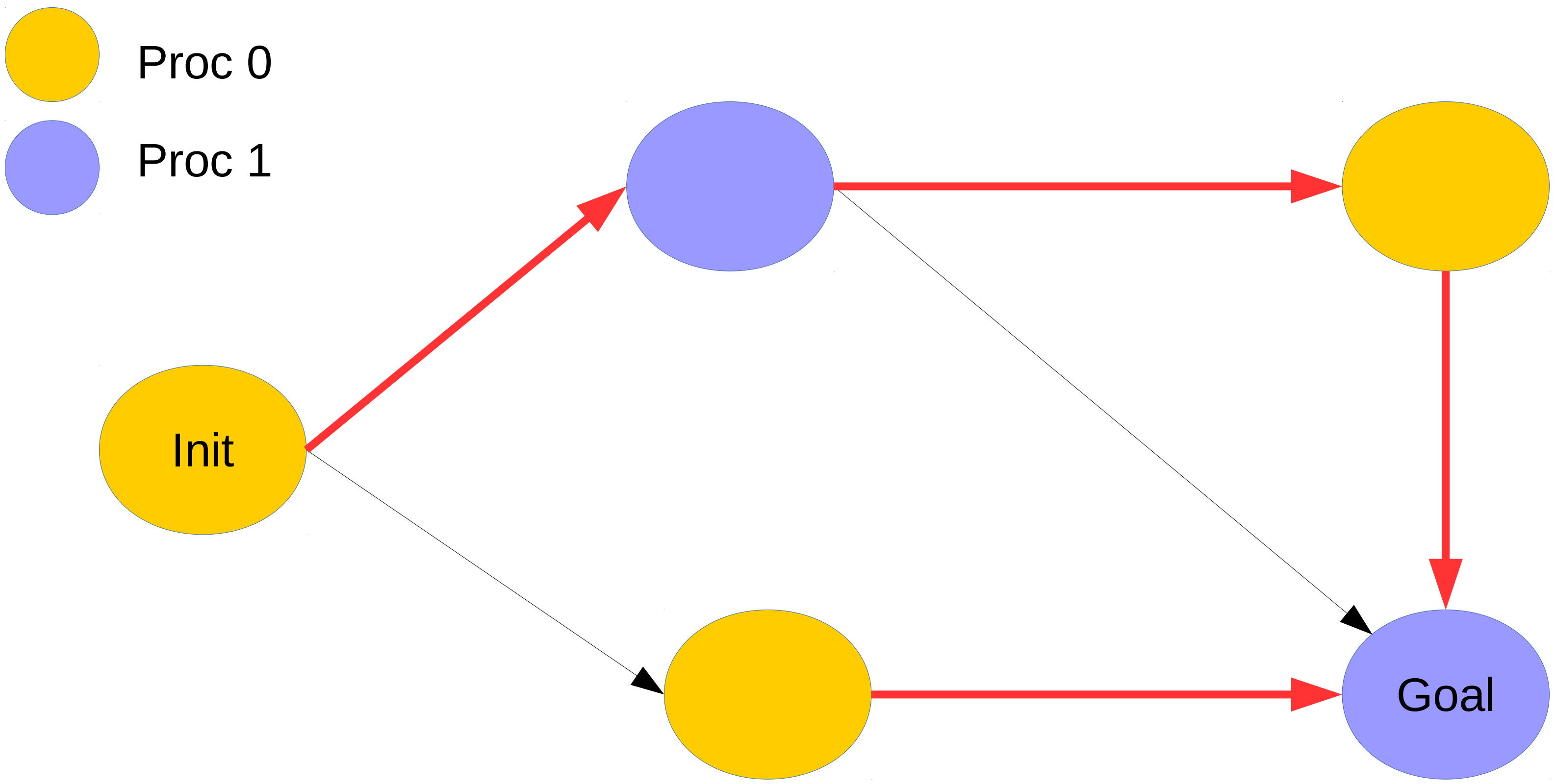


Model of Workloads



1. Expand a node owned by the process ($t = t_{proc}$)
2. Send child nodes to their owners ($t = t_{com}$)
3. Terminates when all nodes are expanded and sent (to ensure optimality)

Model of Overheads



Communication Overhead (CO):

$$CO := \frac{\text{number of edges which require communication}}{\text{total number of edges}}$$

Load Balance (LB):

$$LB := \frac{\text{maximum number of nodes owned by a process}}{\text{average number of nodes owned by a process}}$$

Communication/Search Efficiency

- Communication Efficiency
  - The degradation of walltime efficiency by communication
  - Assume communication cost for every pair of processors are identical

$$eff_c := \frac{1}{cCO} \qquad \text{where} \qquad c := \frac{t_{com}}{t_{proc}}$$

- Search Efficiency
  - The degradation of walltime efficiency by load balance

(proceedings)

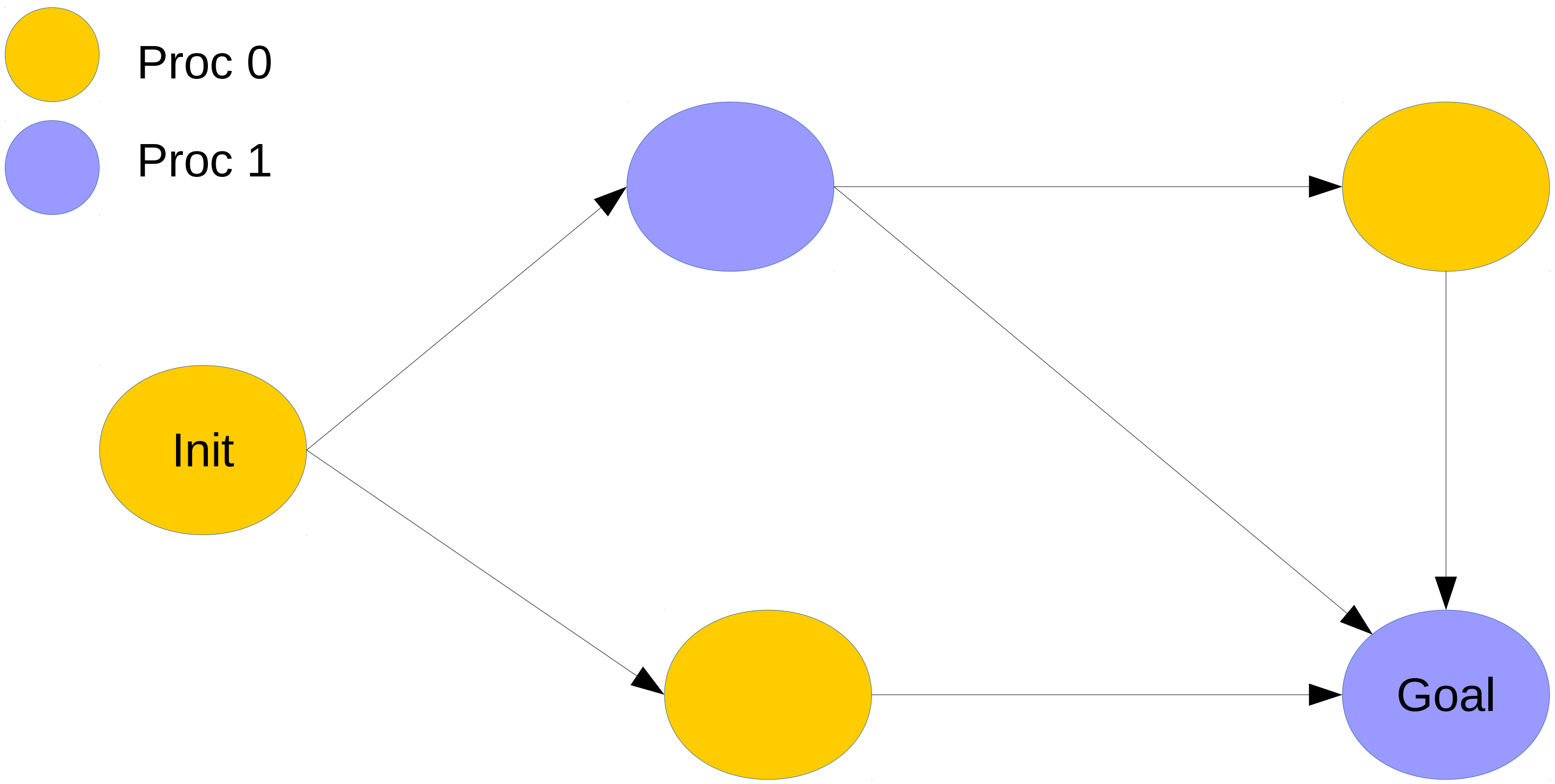
$$eff_s := \frac{1}{1+p(LB-1)} \quad \text{where} \quad p := \text{number of processes}$$

Model Efficiency

- Model Efficiency
  - Assume communication and search overheads are the dominant overhead

$$eff_{esti} := eff_c \cdot eff_s$$
$$= \frac{1}{(1 + cCO)(1 + p(LB - 1))}$$

Model of Parallel Search



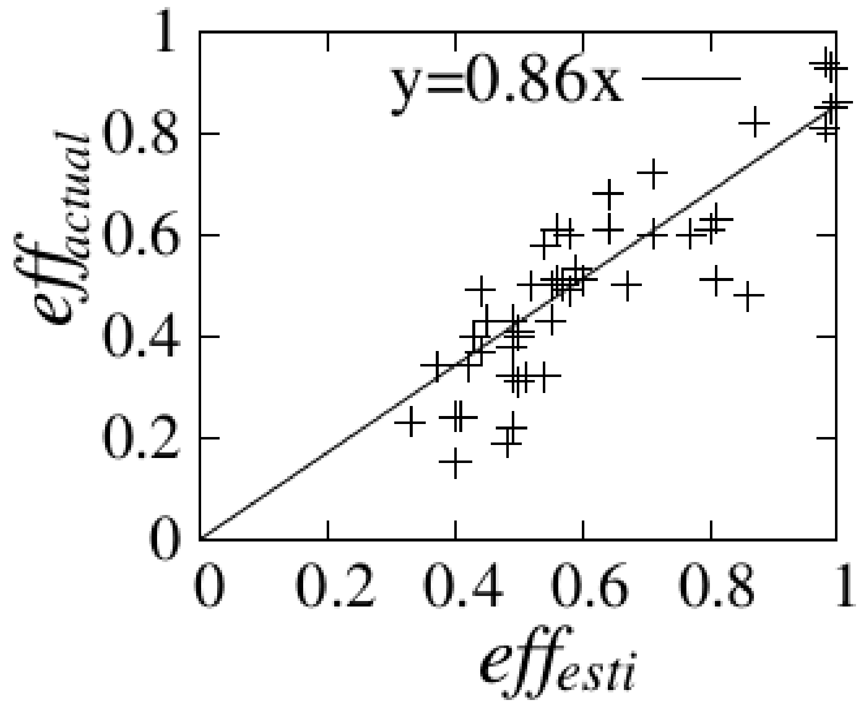
From the partitioning of the workload graph, we can calculate the model efficiency:

$$eff_{esti} := \frac{1}{(1+cCO)(1+p(LB-1))}$$

$$= \frac{1}{(1+1\cdot 4/6)(1+2(3/2.5-1))} = 0.42$$

(where c = 1)

Model vs. Actual Efficiency

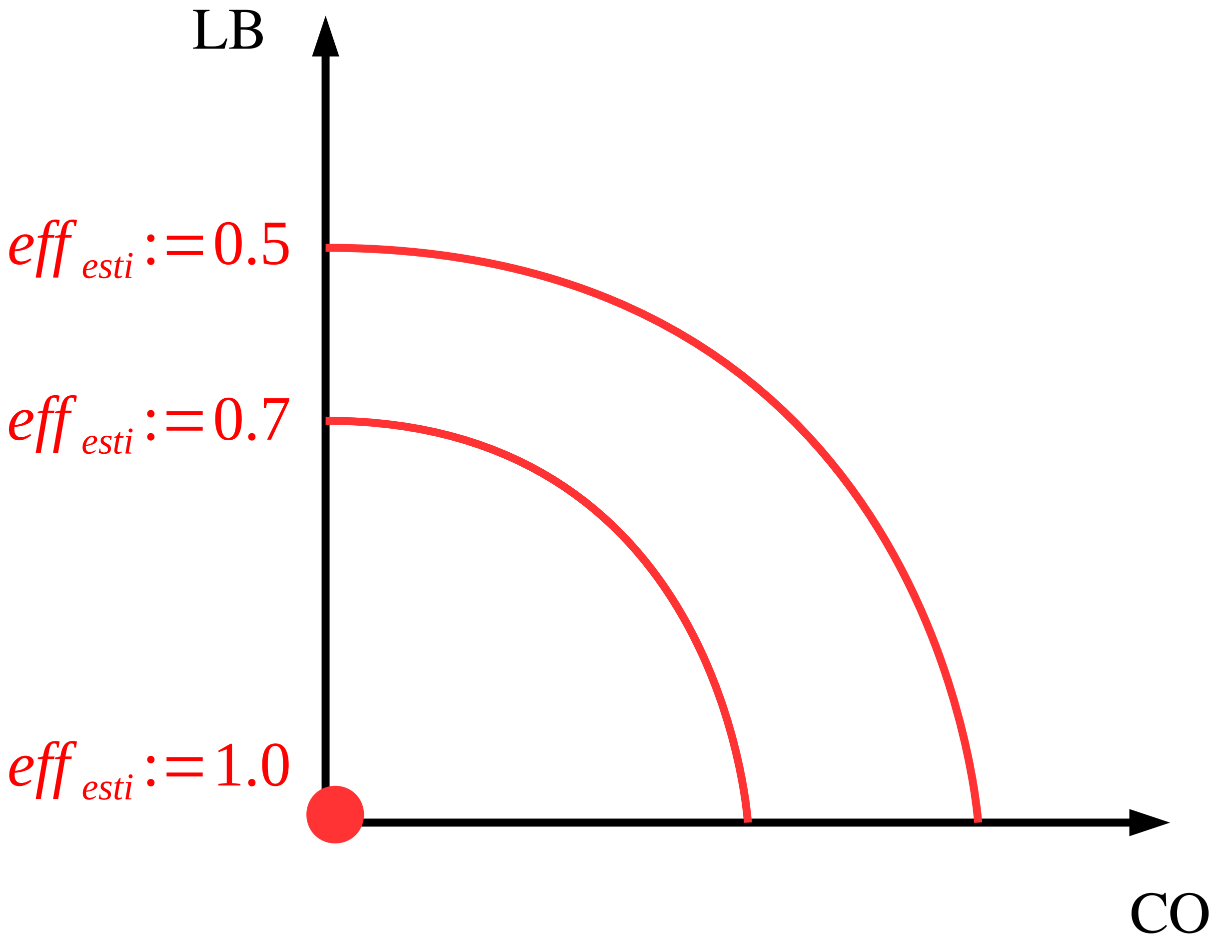


$$c\left(=\frac{t_{com}}{t_{proc}}\right)=1$$

- Calculated model efficiency by 5 HDA* variants
- 48 core machine
- 14 instances from IPC benchmarks
- M&S heuristic (Helmert et al. 2014)

Model in Practice

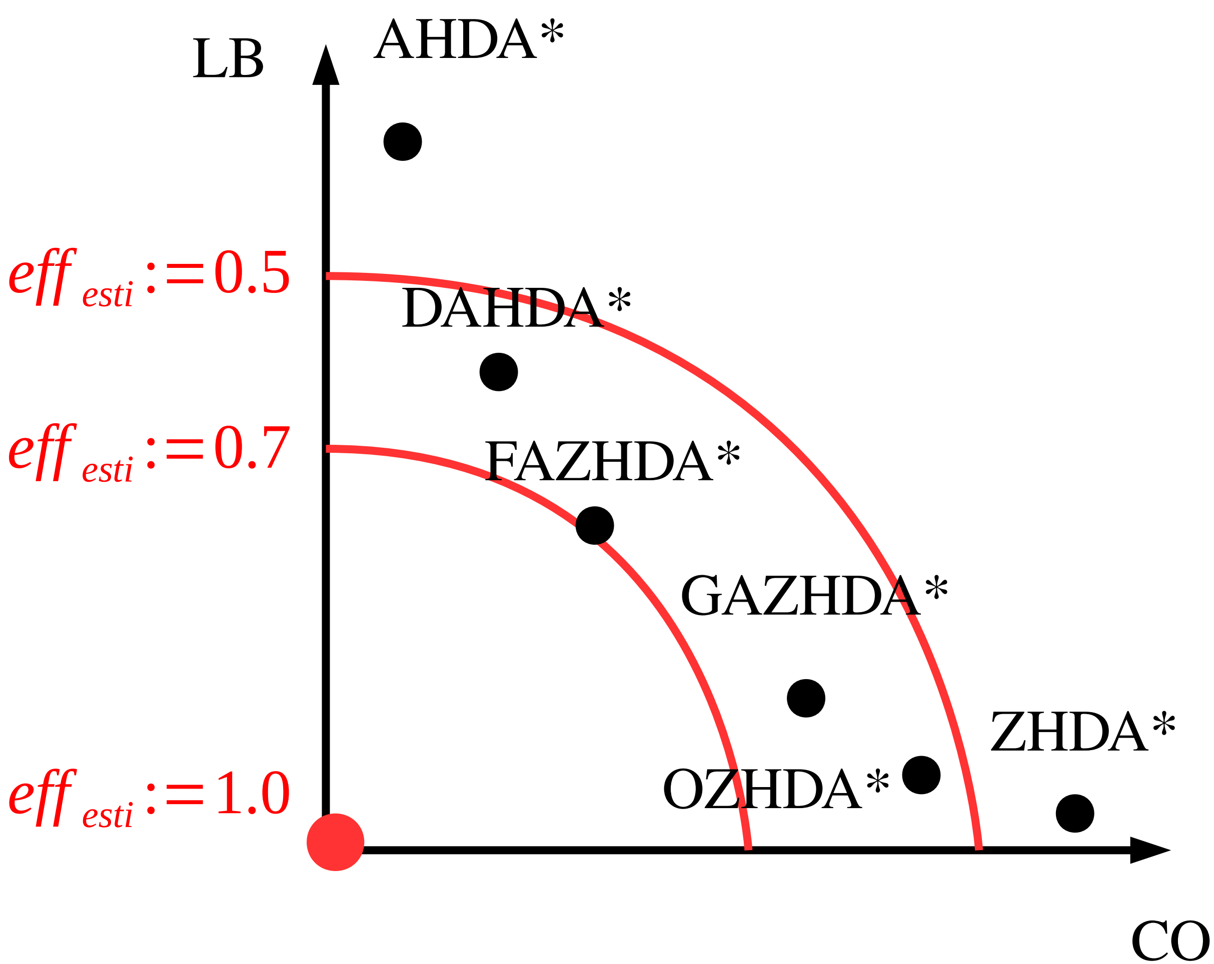
$$eff_{esti} := \frac{1}{(1+cCO)(1+p(LB-1))}$$



$$c\left(=\frac{t_{com}}{t_{proc}}\right)=1.0$$

Model in Practice

$$eff_{esti} := \frac{1}{(1+cCO)(1+p(LB-1))}$$



$$c\left(=\frac{t_{com}}{t_{proc}}\right)=1.0$$

Use of the Model

- We cannot calculate LB and CO beforehand of the search

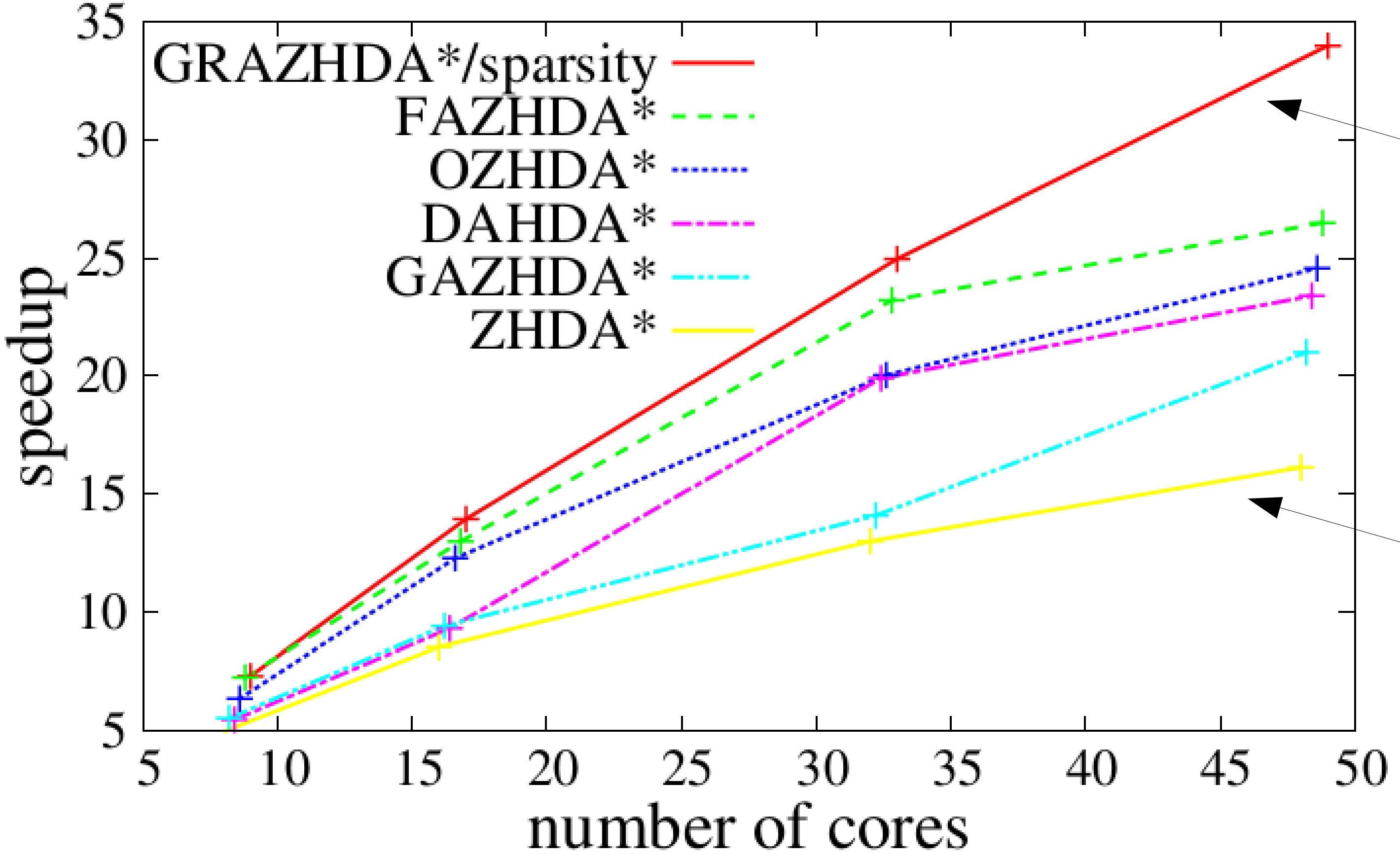
  → The model cannot be used to predict the performance
- So what it the takeaway from the model?

Work Distribution By DTG-Partitioning (GRAZHDA*/sparsity)

- Domain Transition Graph (DTG) is an abstraction of the state-space
- By partitioning each DTG we can approximate partitioning the whole state-space graph.

  (see the paper for detail)

Experimental Results



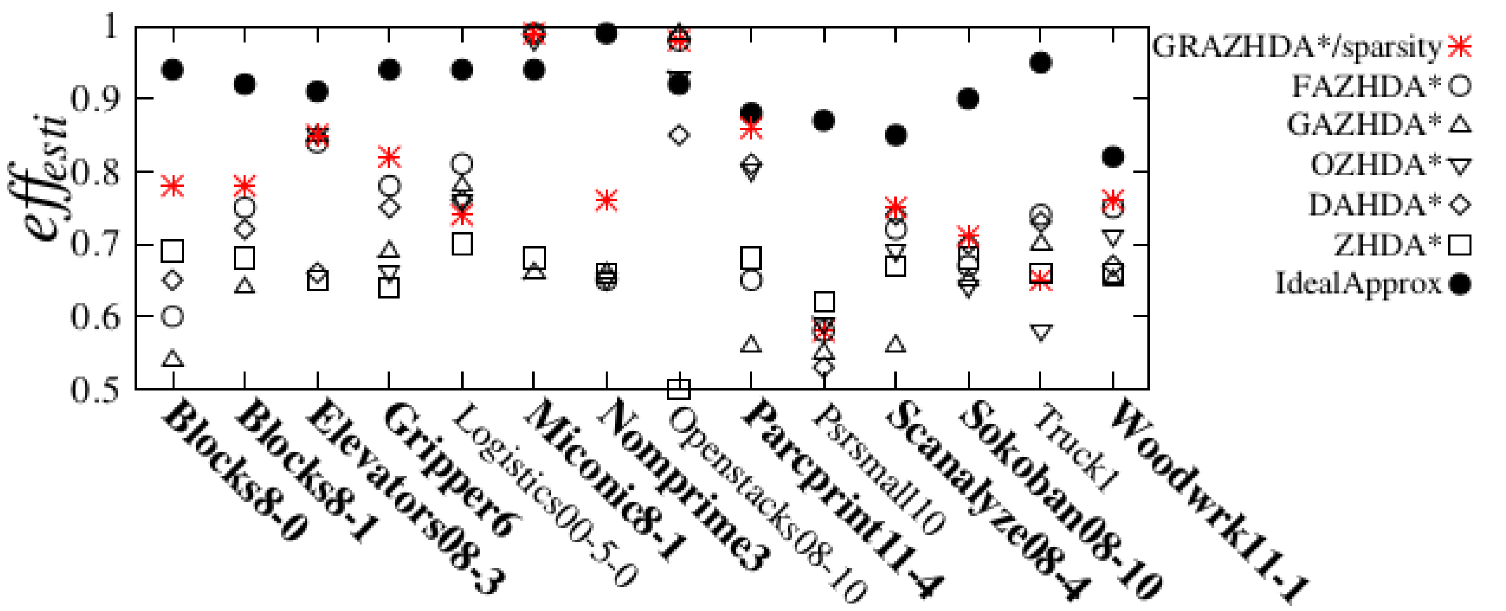
Jinnai&Fukunaga 2017

Kishimoto et al. 2013

Comparison of Model Efficiency

- GRAZHDA*/sparsity has the best model efficiency

$$eff_{esti} := \frac{1}{(1+cCO)(1+p(LB-1))}$$



Summary

- Developed a model to estimate the walltime efficiency of HDA*
- Code available at my github:

  https://github.com/jinnaiyuu/Parallel-Best-First-Searches

  https://github.com/jinnaiyuu/fast-downward (spaghetti right now)
- Journal version available at arXiv

  Jinnai Y, Fukunaga A. 2017. On Hash-Based Work Distribution Methods for Parallel Best-First Search

Open Questions

- Parallelizing other searches (e.g. width-based search)

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

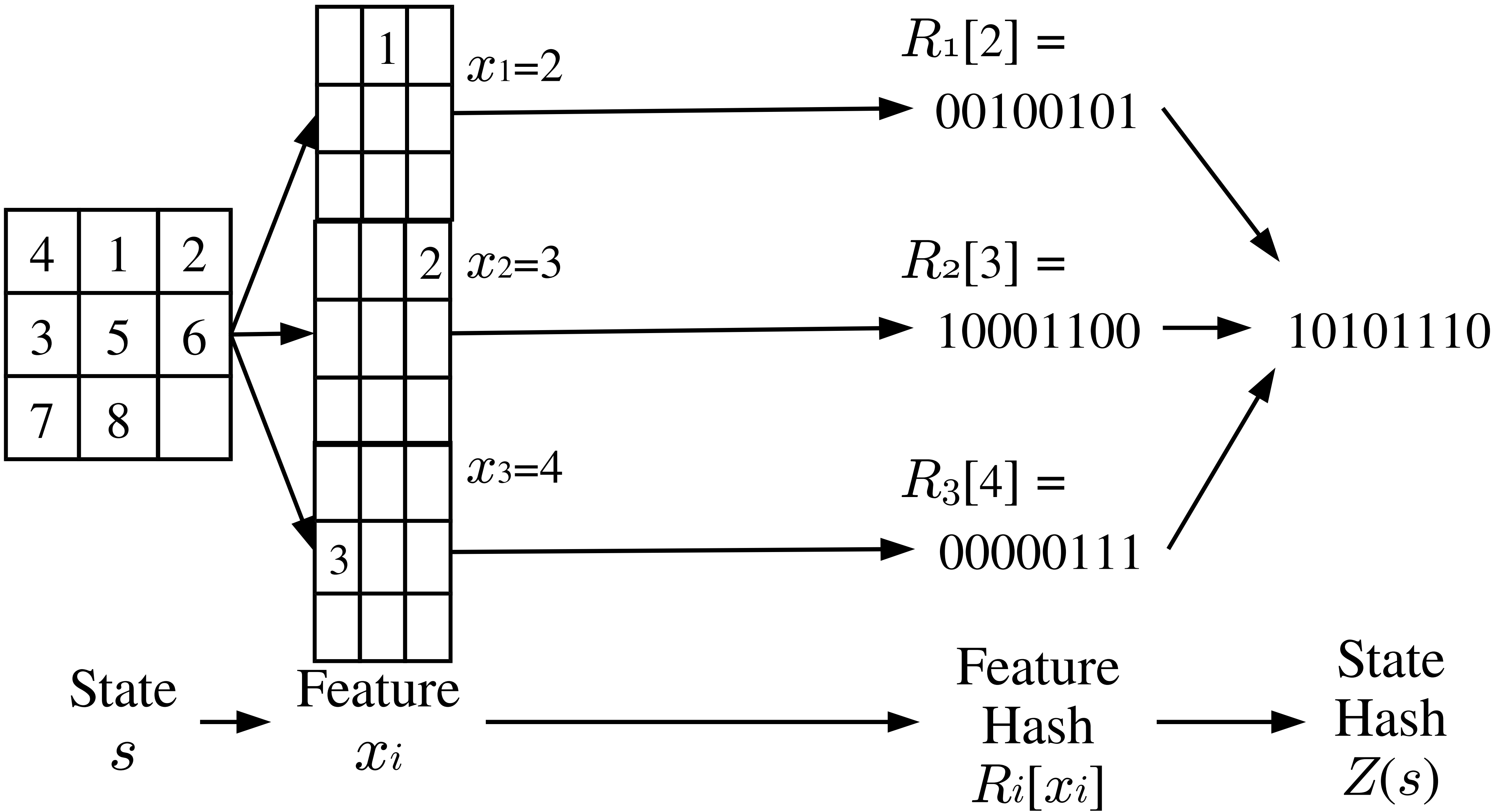
$$Z(s) = R_1[x_1] \text{ xor } R_2[x_2] \text{ xor } \ldots \text{ xor } R_n[x_n]$$

Zobrist Hashing (ZHDA*)

Zobrist (1970); Kishimoto et al. (2009)

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

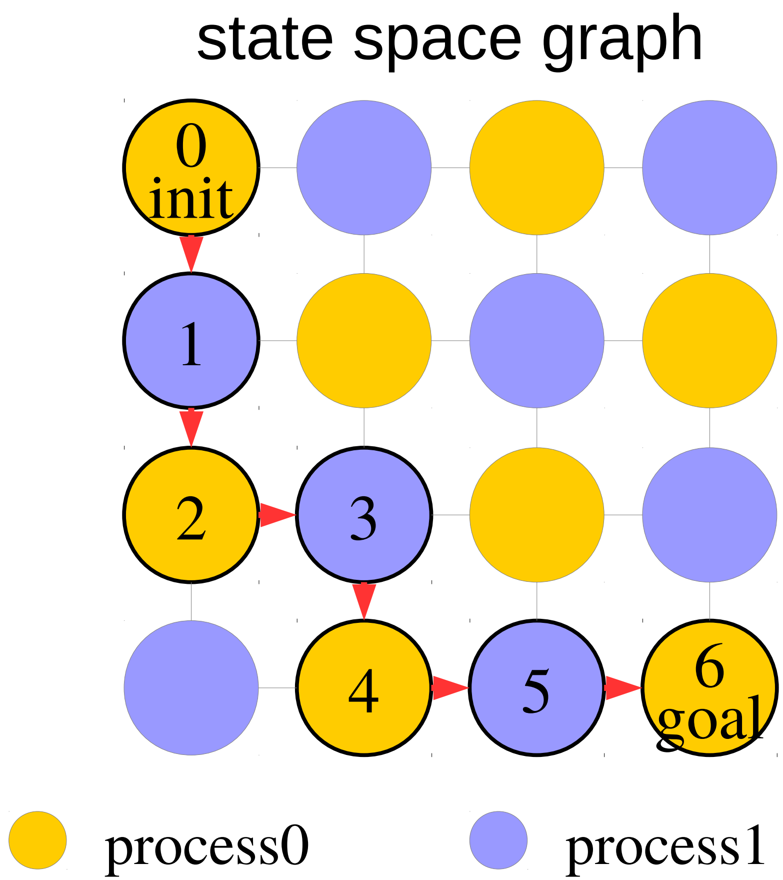
(x_i represents the position of tile i)

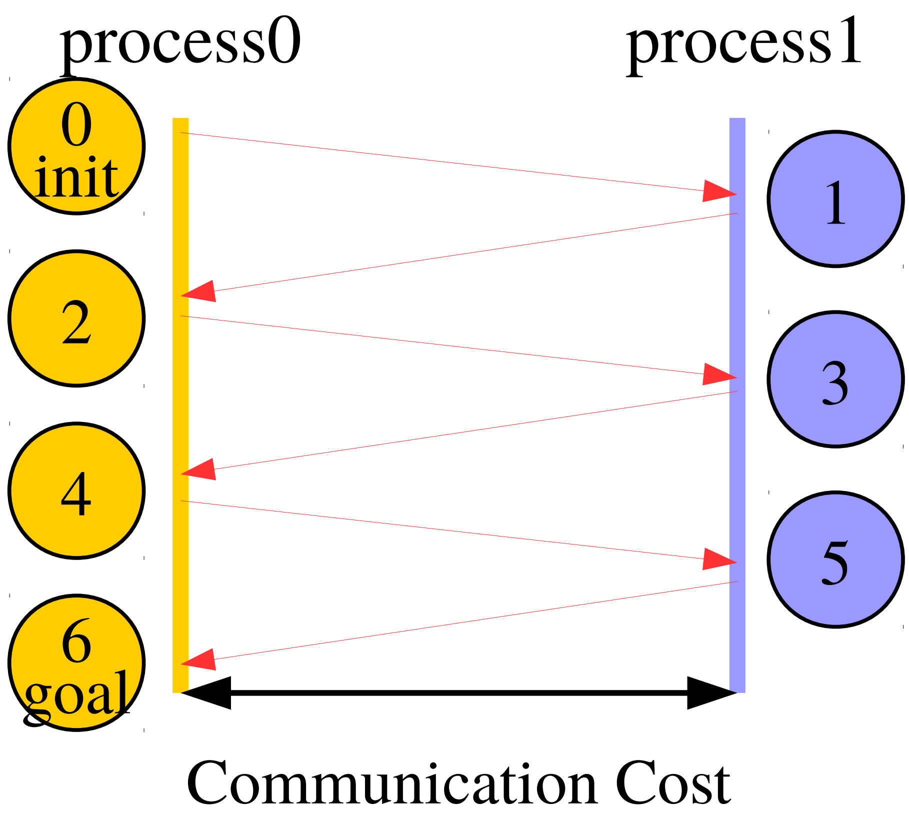


Zobrist Hashing (ZHDA*)

Zobrist (1970); Kishimoto et al. (2009)

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





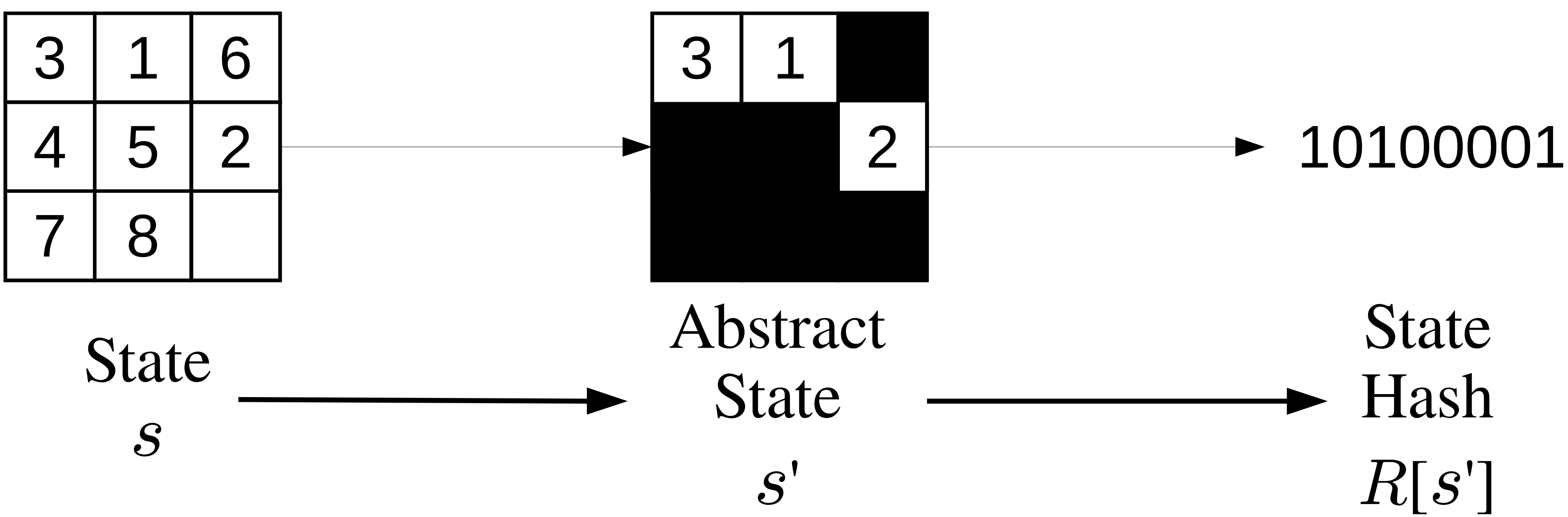
State abstraction (AHDA*)

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:

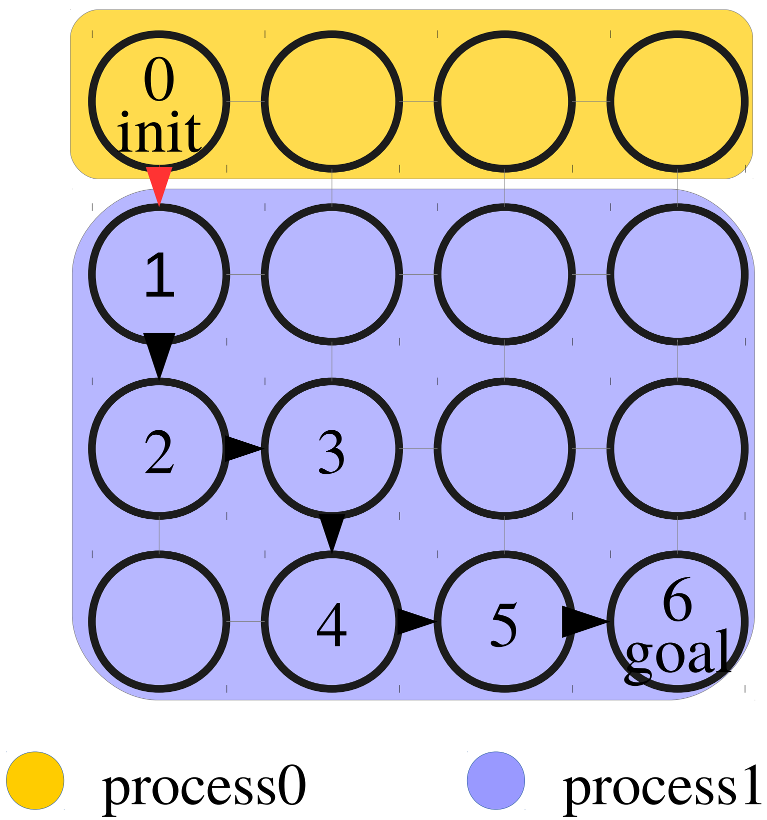


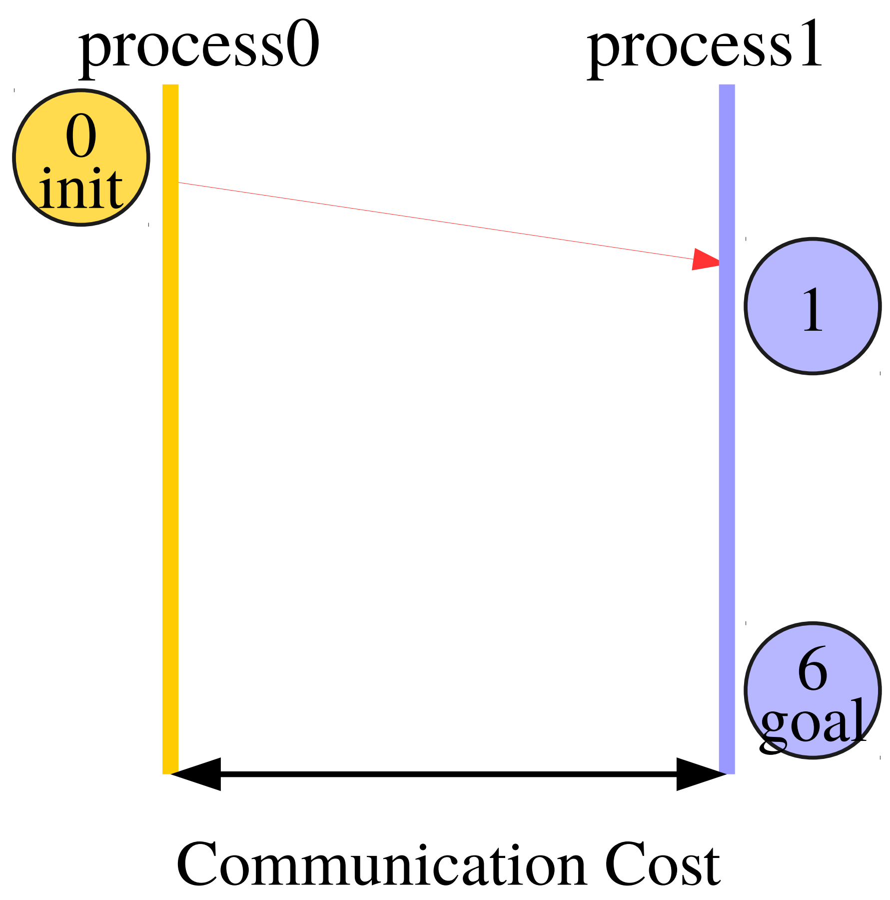
State abstraction (AHDA*)

Burns et al. (2010)

- Strenght: low communication overhead
- Limitation: worse load balance

state space graph





Abstract Zobrist Hashing (AZHDA*)

Jinnai&Fukunaga (2016)

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

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

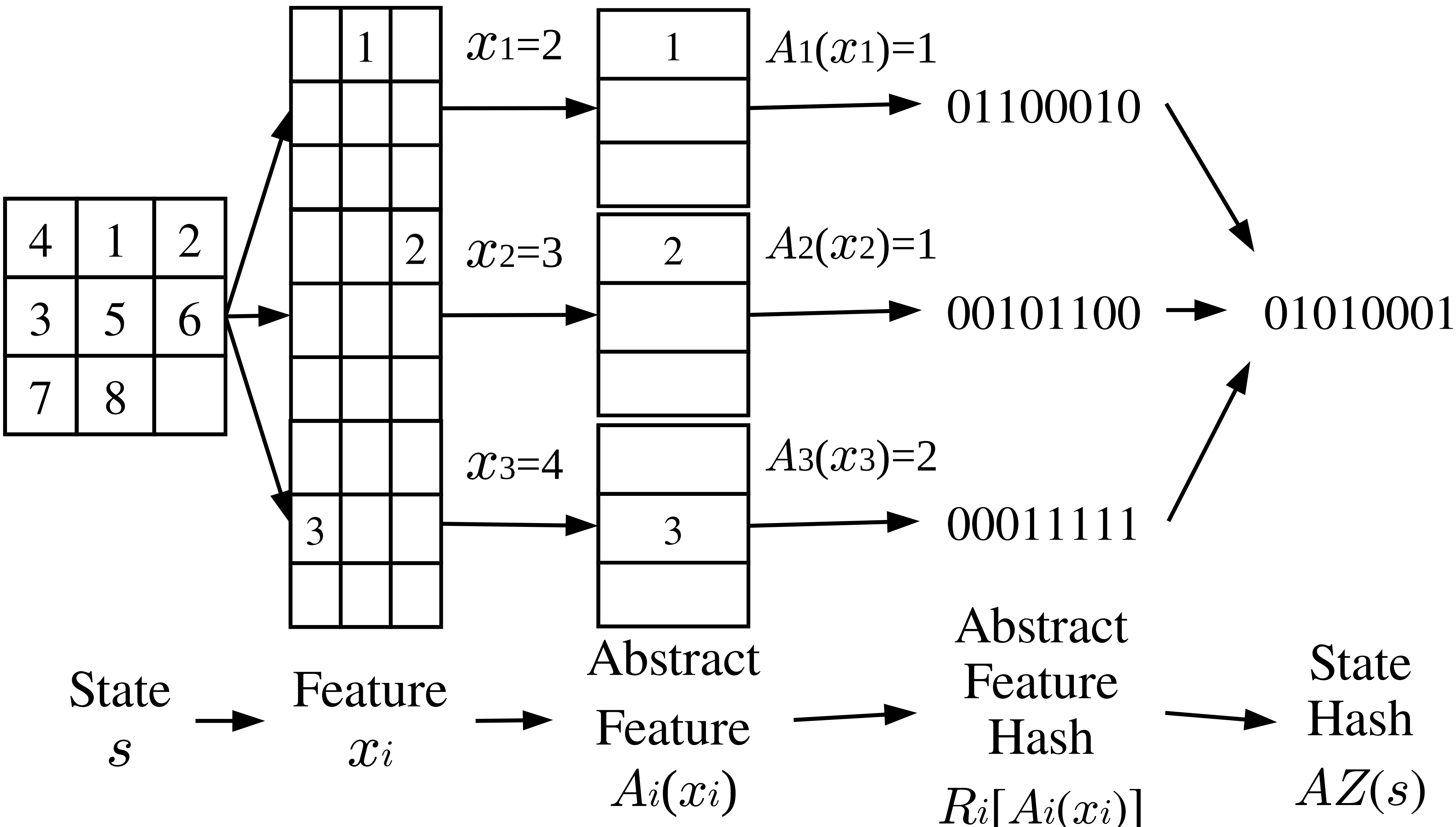
or

$$AZ(s) = Z(s'), \text{ where } s' = (A_1(x_1), A_2(x_2), \ldots, A_n(x_n))$$

Abstract Zobrist Hashing (AZHDA*)

Jinnai&Fukunaga (2016)

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

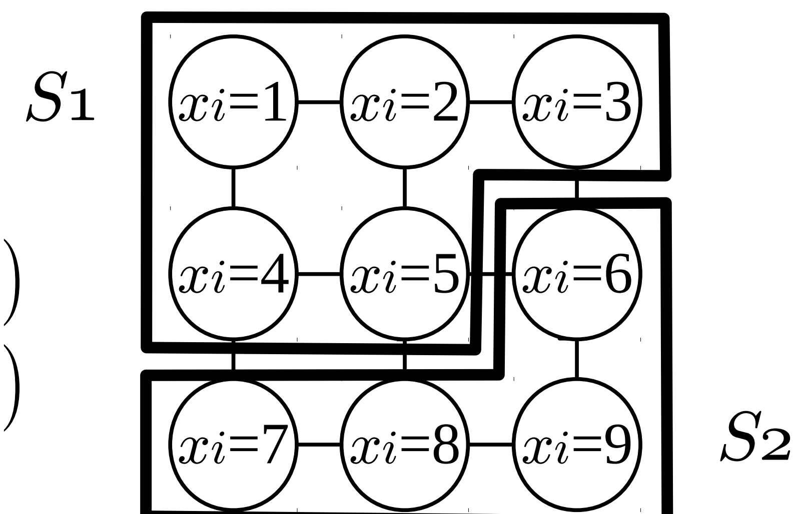


Greedy abstract feature generation
(Jinnai&Fukunaga 2016)

Approach: maps each SAS+ variable xi to abstract feature S_1 and S_2 based on xi'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

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



DTG of a variable xi represents the transition of the value

GreedyAFG applied to DTG of 8-puzzle