Multiagent Pathfinding
AAAI Technical Report WS-12-10

A* Variants for Optimal Multi-Agent Pathfinding

Meir Goldenberg
ISE Department
Ben-Gurion University, Israel
mgoldenbe@gmail.com

Ariel Felner, Roni Stern, Guni Sharon
ISE Department
Ben-Gurion University, Israel
felner@bgu.ac.il,
{roni.stern, gunisharon}@gmail.com

Jonathan Schaeffer
CS Department
University of Alberta, Canada
jonathan@cs.ulaberta.ca

Abstract

Several variants of A* have been recently proposed for finding optimal solutions for the multi-agent pathfinding (MAPF) problem. However, these variants have not been deeply compared either quantitatively or qualitatively. In this paper we aim to fill this gap. In addition to obtaining a deeper understanding of the existing algorithms, we describe in detail the application of the new enhanced partial-expansion technique to MAPF and show how pattern databases can be applied on top of this technique.

Introduction

Multi-agent pathfinding (MAPF) is a challenging problem with many practical applications in robotics, video games, vehicle routing, etc. (Silver 2005; Dresner and Stone 2008). Instances of the problem consist of a graph $G = (V, E)$ and k agents. Each agent has a start position and a goal position. The task is to move all the agents to their goals without collisions (the precise definition will be given below) while minimizing a cumulative cost function. In its general form, MAPF is NP-complete, because it is a generalization of the sliding tile puzzle which is NP-complete (Ratner and Warnuth 1986). Because of the problem's difficulty, most research focused on decentralized approaches that may return non-optimal solutions and, in some cases, are not complete.

Most recently, solving MAPF optimally has gained more attention, resulting in several new algorithms (Sharon et al. 2011; 2012) and variants of A* (Standley 2010; Felner et al. 2012). In this paper, we focus on the new variants of A*.

A recently developed technique called *enhanced partial expansion A** (EPEA*) (Felner et al. 2012), uses domain-specific knowledge to avoid the generation of nodes whose f-value is greater than the cost of the optimal solution. EPEA* was applied to a large variety of domains. The MAPF problem is the most challenging but it only received very limited attention. The main contribution of our paper is the detailed presentation of a way to apply EPEA* to MAPF. Furthermore, our current method of applying EPEA* is generalized to allow using pattern databases (PDBs) on top of EPEA*. We also present a number of enhancements to the basic method of EPEA* for MAPF.

Effectively applying PDBs to MAPF is a challenging task. To the best of our knowledge, no application of PDBs to MAPF has ever been reported and very simple heuristic functions were used by previous A* solvers for MAPF. In this paper, we report an application of PDBs to MAPF for the first time. In fact, we describe how to apply PDBs on top of EPEA*, with the result of further significant improvement in time performance. This is the second contribution of this paper.

Our third contribution is a new variant of A* for optimally solving MAPF, which is a hybrid of the *operator decomposition* (OD) technique of (Standley 2010) and the *partial expansion* technique of (Yoshizumi, Miura, and Ishida 2000) (the latter is referred to as *basic partial expansion* (BPE) hereafter).

Lastly, we present important insights about the unreported properties of the techniques introduced by (Standley 2010) and suggest ways to take advantage of some of these properties.

We start by formally describing the MAPF problem in the next section. We then present all necessary background. Following that are the sections with our contributions. We follow up with experimental results, which **(1)** show that EPEA* (in particular, combined with PDBs) is the current state-of-the-art among A*-based approaches to optimal MAPF and **(2)** provide deeper understanding of the differences in performance between EPEA* and ODA*. Finally, we conclude.

Background

We now give background on the problem and the related techniques.

Multi-agent pathfinding: formal definition

We focus on the following commonly used variant of MAPF (Standley 2010; Sharon et al. 2011; 2012). The *input* to MAPF is: **(1)** A graph $G(V, E)$ and **(2)** k agents labeled $a_1, a_2 \dots a_k$. Every agent a_i is coupled with a start and a goal vertices: s_i and g_i. At the initial time point $t = 0$ every agent a_i is located in location s_i. Between successive time points, each agent can perform a `move` action to a neighboring location or can `wait` (stay idle) at its current location. The main constraint is that each vertex can be occupied by at most one agent at a given time. In addition, if a and b are neighboring vertices, two different agents cannot simultaneously traverse the connecting edge in opposite directions (from a to b and from b to a). However, agents are allowed to *follow* each other, i.e., agent a_i could move from x to y at the same time as agent a_j moves from y to z.

The task is to find a sequence of $\{move, wait\}$ actions for each agent such that each agent will be located in its goal position while aiming to minimize a global cost function. In our variant of the problem, the cost function is the summation (over all agents) of the number of time steps required to reach the goal location. Therefore, both `move` and `wait` actions cost 1.0, except for the case when the `wait` action is applied at an agent's goal location and costs zero. If an agent waits m times at its goal location and then moves, the cost of that move is $m + 1$.

The standard A* approach

The state space for an A*-based search consists of all possible permutations of the k agents on the $|V|$ vertices. Let b_{base} be the branching factor for a single agent. The global branching factor is $b = O((b_{base})^k)$. All $(b_{base})^k$ combinations of actions should be considered and only those with no conflicts represent the *legal* moves.

The commonly-used admissible heuristic is the *sum of individual costs* (SIC) (Sharon et al. 2011) defined as the sum of the optimal solution costs of single-agent pathfinding problems for the individual agents.

Pattern databases

Pattern Databases (PDBs) (Culberson and Schaeffer 1998; Felner, Korf, and Hanan 2004) is a powerful method for automatically building admissible memory-based heuristics based on domain abstractions. The main idea of PDBs is to first abstract the state space by only considering a subset of the variables or constraints. Then, a full breadth-first search is performed in the abstract state space (aka pattern space) from the abstract goal. Distances to all abstract states (patterns) are calculated and stored in a lookup table (PDB). These values are then used throughout the search as admissible heuristics for states in the original state space.

In our experiments, we used instance-dependent on-demand pattern databases (Felner and Adler 2005). On-demand PDBs are built lazily during the search and are particularly effective in domains where the abstract space is too big to be stored completely in memory. At first, a directed search in the pattern state space is performed from the goal pattern to the start pattern (unlike regular PDBs where a complete breadth-first search is performed). All the patterns seen in this search are saved in the PDBs. Then, the main search in the real state space begins. As more nodes are generated, the search in the pattern space is continued lazily and more PDB values are found and stored.

Standley's A* variants

The current line of development of specific new algorithms for optimal MAPF started with the seminal work of (Standley 2010). Standley suggested two main improvements to the classic A* search for MAPF. We describe them briefly.

Independence detection (ID) The size of the state space of MAPF is exponential in the number of agents. Standley introduced the *independence detection* (ID) framework to reduce the number of agents that participate in the actual A* searches as follows. Two groups of agents are designated as *independent* if there is an optimal solution for each group such that the two solutions do not conflict. The basic idea of ID is to divide the agents into *independent* groups. Initially each agent is placed in its own group. Shortest paths are found for each group separately. The resulting paths of all groups are simultaneously performed until a conflict occurs between two (or more) groups. Several heuristic methods are applied to try and resolve the conflict. If all methods fail, the agents in the conflicting groups are unified into a new single group. Whenever a new group of $k \geq 1$ agents is formed, this new k-agent problem is solved optimally by an A^*-based search. This process is repeated until no conflicts between groups occur. Standley 2010 observed that the A^*-search of the largest group dominates the running time of solving the entire problem. Standley reported exponential improvement in time performance using ID.

Operator decomposition (OD) Not only the number of possible states for an instance of MAPF is exponential in the number of agents (k), but, as explained above, even the branching factor of a given state may be exponential in k. Suppose a state with 20 agents on a 4-connected grid. Each agent may have up to 5 possible moves (4 cardinal directions and `wait`). Fully expanding all the $5^{20} = 9.53 \times 10^{14}$ neighbors of such a state is computationally infeasible. In addition, each of the agents can either move towards the goal, stay idle (wait action), or move away from its goal. The agent's individual f-value grows by zero, one or two, respectively. In our example with 20 agents, children with up to 41 different f-values will be generated. Most of these children may never need to be expanded if their f-value is larger than the cost of the optimal solution. We designate such nodes, i.e. the nodes with f-value greater than the cost of the optimal solution, as the *surplus* nodes.

To deal with these problems, *operator decomposition* (OD) was introduced by Standley 2010. Agents are assigned an arbitrary (but fixed) order. When a regular A* node is expanded, OD considers only the moves of the first agent, which results in generating the so called *intermediate nodes*. At these nodes, only the moves of the second agent are considered and more intermediate nodes are generated. When an operator is applied to the last agent, a regular node is generated. Once the solution is found, intermediate nodes in the open list are not developed further into regular nodes, so that the number of regular surplus nodes is significantly reduced. We refer to this variant of A* as ODA*.

Enhanced partial expansion (EPE)

Recall that *surplus* nodes are the nodes with f-value larger than the cost of the optimal solution. Due to the large branching factor, the number of surplus nodes in MAPF can be very large.

*Partial Expansion A** (PEA*) (Yoshizumi, Miura, and Ishida 2000) addresses the problem of surplus nodes. When

PEA* expands a node n, b children are generated but only those with $f = f(n)$ are inserted into the open list. The rest of the generated children are discarded. n is re-inserted into the open list, but with the f-cost of its best child that was discarded. Such a node may be then re-expanded but with the new f-value. Note that each child of a given node n may be generated many times – once for every re-expansion of n.

In domains with a large branching factor, PEA* will gain a large reduction in the size of the open list, which, depending on the implementation of the open list, may have positive time performance implications as well. Hereafter, we refer to PEA* as the basic partial expansion A* (BPEA*).

The recently introduced Enhanced Partial Expansion A* (EPEA*) (Felner et al. 2012) takes BPEA* further. BPEA* generates *all* children of n but only those with $f = f(n)$ are inserted into the open list. In contrast, EPEA* uses a mechanism which generates only the children with $f = f(n)$, without generating and discarding the other children. Thus, each node is generated only once throughout the search process and no child is regenerated when its parent is re-expanded.

EPEA* uses *a priori* domain knowledge to avoid generating surplus nodes as follows. First, distinction is made between the regular f-value ($g+h$) of a node n, called its *static value* and denoted by $f(n)$ (small f), and the value currently stored for n in the open list, called the *stored value* of n and denoted by $F(n)$ (capital F). Initially $F(n) = f(n)$. When expanding a node n, EPEA* generates only the children n_c with $f(n_c) = F(n)$. The stored value of n, $F(n)$, is updated to the f-cost of the next best child and n is re-inserted into the open list.

This is achieved with the following idea. In many domains, one can classify the operators applicable to a node n based on the change to the f-value, $\Delta f = f(n_c) - f(n)$, of the children n_c of n that they generate. The idea is to use this classification and apply only the operators of the relevant class. For its operation, EPEA* needs to be supplied with a domain-specific *operator selection function* (OSF) which receives a state p and a value v. The OSF has two outputs: **(1)** a list of operators that, when applied to state p, will have $\Delta f = v$. **(2)** v_{next} — the value of the next Δf in the set of applicable operators.

Assume that a node n is expanded with a stored value $F(n)$ and static value $f(n)$. We only want to generate a child n_c if $f(n_c) = F(n)$. Since the static value of n is $f(n)$, we only need the operators which will increase $f(n)$ by $\Delta f = F(n) - f(n)$. Therefore, $OSF(n, \Delta f)$ is used to identify the list of relevant operators. Node n is re-inserted into the open list with the next possible value for this node, $f(n) + v_{next}(n, \Delta f)$. If the v_{next} entry is nil, meaning that all children of n have been generated, then n is moved to the closed list.

Application of enhanced partial expansion to MAPF

In this section we describe an *operator selection function* (OSF) for MAPF that is generalized to allow for the usage of PDBs. We define a *composite* agent (CA) as a group of agents. For example, suppose that there are 5 agents. We might build pairwise PDBs for agents 1 and 2 and for agents 3 and 4. In this case, we will build an OSF for three CAs: CA_1 consisting of agents 1 and 2, CA_2 consisting of agents 3 and 4 and CA_3 consisting of the single agent 5. As a result of a move, an individual agent's f-value can either remain the same ($\Delta f = 0$) or grow by one ($\Delta f = 1$) or grow by 2 ($\Delta f = 2$). The f-value of a composite agent with l agents that moved can grow by any amount from $\Delta f = 0$ to $\Delta f = 2l$.

A special data structure, called the *composite agent operators structure* (CAOS) contains, for each possible state of each composite agent all legal (i.e. without collisions of agents within the CA) operators ordered by Δf. Since this data structure can be very large, we compute it on demand (using the same lazy technique of instance-dependent PDBs which are computed on demand).[1] At the beginning of the search the CAOS is empty. When EPEA* expands a node for the first time, it searches for each composite agent's state in that structure. Whenever a CA's state is not found, CAOS is updated (by activating the on-demand function for this state) with the relevant list of operators ordered by Δf.

Figure 1 (left) shows an example with 3 composite agents with 5, 3 and 4 operators, respectively. Suppose that the node being expanded has the static value of $f = 2$ and the stored value of $F = 10$. The OSF will need to find all combinations of operators for composite agents with the sum of Δf's equal to $10 - 2 = 8$. In Figure 1 (left), the first such choice is shown in solid.

Effectively, we have to solve the following combinatorial enumeration problem: given k bins with balls each tagged with a number, enumerate all ways of choosing one ball from each bin, such that the total sum of the numbers on the balls is $F - f$. It is easy to see that this problem is exponential in the number of bins (which corresponds to the number of composite agents). Since this is done for every expansion, it is critical that this problem be solved efficiently.

Our solution is a simple recursive procedure with three enhancements, two generic and one domain-specific. We explain the recursive procedure with the above example with three bins. The recursive procedure tries each of the choices for the first bin and performs a recursive call with the update sum for the remaining bins. For example, when third choice (which is the first operator with $\Delta f = 3$) is tried for the first (i.e. left-most) bin, the remaining two bins have to contribute $8 - 3 = 5$ to the sum. Therefore, we can use a recursive call to our procedure for the remaining two bins and the required sum of 5.

The two enhancements are as follows. First, for each

[1]Note that several copies must be stored for states of a CA where one of the individual agents is at its goal location. For example, consider an agent that has arrived at its goal location with the individual g-value of 5 (i.e. $f = g = 5$) and made two `wait` moves. Since wait moves at the goal are free, this agent still has $f = g = 5$. However, if we now choose to move this agent, then, retroactively, all of the previous moves are not free and the agent's g-value grows by 3. This causes a different list of operators ordered by Δf for the CA than if that agent had just arrived to its goal location.

Bins for three CAs		
CA_1	CA_2	CA_3
0	0	0
0	**1**	1
3	3	**4**
3		4
5		

ND-bins for three CAs		
CA_1	CA_2	CA_3
0	0	0
3	1	1
5	3	4

Figure 1: Computing OSF for MAPF

CA, there can be several operators with the same Δf. We can considerably speed up the procedure by getting rid of these duplicates. For our example, this is shown in Figure 1 (right). These bins with no duplicates (ND-bins) are also stored in CAOS. For each combination with suitable sum in ND-bins, the concrete operators with the corresponding Δf for each CA have to be applied. In our example, suppose that the combination where the CAs contribute the Δf's of 3,1 and 4, respectively, is found. We have two consider 2 operators for CA_1 and 2 operators for CA_2, for a total of 4 combinations of operators.

The second enhancement is that significant number of options can be pruned by storing the sums of the smallest and the largest numbers in ND-bins. For example, once the number 0 is chosen in the first ND-bin, the total sum cannot be smaller than $0 + (0 + 0) = 0$ and cannot be larger than $0 + (3 + 4) = 7$. Thus, we know that the sum of 8 cannot be achieved without considering options for the remaining two bins.

The third enhancement takes place when combinations of operators prunes combinations of operators for composite agents by checking for collisions between agents that belong to different composite agents. In our example, if applying the first operator with $\Delta f = 3$ for CA_1 and the (only) operator with $\Delta f = 1$ for CA_2 results in an illegal move, then we do not have to consider the operators with $\Delta f = 3$ for CA_3.

In order to compute the next stored value for the node being expanded, we maintain two quantities when searching for suitable combinations in ND-bins: **(1)** the sum of the current choices for the ND-bins and **(2)** the smallest possible choices for the remaining beans is maintained at all times. The next stored value is the smallest sum of these quantities that has been encountered.

PDBs for MAPF

The task of effectively applying PDBs to MAPF presents the following challenge. PDBs are effective only when the entries of PDB contain abstract states with distance to the goal higher than the base heuristic (such as the Manhattan distance for pathfinding) for this abstract state. In case of MAPF, the abstract states are projections of regular states onto different subsets of agents, while the base heuristic is the *sum of individual costs* heuristic (SIC) (defined above). In our terminology, abstract states are composite agents with their locations. Consider, as an example, a composite agent consisting of two agents at particular locations. The SIC heuristic of this state will underestimate the true distance to

s_1		s_2, g_2	
X	X	g_1	X

Figure 2: An example of inconsistency of PDBs applied to MAPF

the goal only if there is a conflict between the agents. Therefore, PDBs for MAPF can be effective only if they are built for agents that participate in many conflicts. However, this information is not known *a priori*. For example, given an instance with 20 agents, a special method is needed to find an effective way to pair up the agents for pairwise databases.

We overcome this problem by using the ID framework. Namely, whenever ID joins two agents into a group, we use this information to build pairwise PDBs at later stages of ID. Suppose, for example an instance with 10 agents. Let us consider an execution of ID, while ignoring all operations except the joining of two groups into a single group. Suppose that ID joined agents $\{1, 5\}$, then joined agents $\{2, 8\}$ and then joined the two groups together, forming the group consisting of agents $\{1, 2, 5, 8\}$. When looking for an optimal path for this group, we will use two 2-agent PDBs: one with states projected onto agents $\{1, 5\}$ and the other using projections onto agents $\{2, 8\}$. In our experiments, we used instance dependent pattern databases (Felner and Adler 2005) described above.

It is important to note that, in our formulation of MAPF, the PDB-heuristic can be inconsistent (Felner et al. 2011). For example, suppose a pattern that consists of two agents, whose start and goal locations are shown in Figure 2. The PDB entry for the start location of the agents contains the value of 6 (Agent a_1 moves three steps, while agent a_2 the must wait, move away and move back). Suppose that a_1 moves to the right, while a_2 waits (this wait is free, since a_2 is located at it's goal). The resulting node has $g = 1$ while it's PDB entry contains 4, which means that the f-value has decreased to 5, signifying inconsistency. Standard techniques, such as BPMX (Zahavi et al. 2007) can be applied to take advantage of this property of PDBs for MAPF, which we did in our experiments.

Basic partial expansion (BPE) A* with operator decomposition

We note that BPEA* is generic and can be applied on top of any procedure for neighbor generation. In particular, BPEA* can be applied on top of OD as follows. Each intermediate note is a successor of some standard node. The most immediate such standard node is called the *standard predecessor* of the intermediate node. When a node n is expanded there are two cases: **(1)** if n is standard, then the regular BPE condition applies, otherwise **(2)** a child n_c is kept only if it's f-value is equal to the stored value of the standard predecessor of the node being expanded. We call this algorithm BPEODA*.

We note that BPEODA* suffers much less than BPEA* from having to generate all children of a node being expanded before discarding the surplus nodes. This is because operator decomposition reduces the branching factor to that of a single agent, so that the overhead of generating

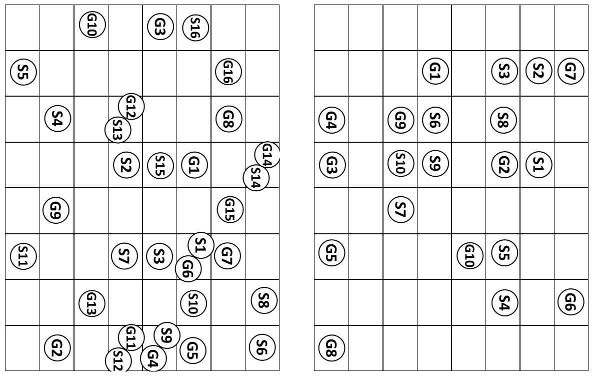

Figure 3: Limitations of ID

all neighbors is rather small. However, this algorithm suffers from the need to the generate intermediate nodes, which it inherits from ODA*. BPEODA* inherits the advantage of ODA* as well – early duplicate detection. However, our experiments showed that this duplicate detection becomes less effective when BPE is enabled.

Insights About Standley's Techniques

Algorithm-dependence of ID

Note that the ID framework treats the underlying search algorithm as a black box that returns an optimal solution given a set of constraints (such as the paths chosen by other agents). As such, the groups formed by ID depend very much on the move ordering and tie-breaking rules of the underlying search algorithm. In particular, the size of the largest group may change depending on that move ordering.

Furthermore, this effect may appear in a stronger manner when ID is used with different underlying algorithms. We report that ID's performance can dramatically differ from one algorithm to another. For the instance in Figure 3 (left), ID on top of ODA* created a group of 12 agents (numbered 1-4, 7-11 and 13-15 in the figure) and failed to solve the problem in two minutes. When ID was used on top of EPEA*, the same problem was solved in under half-a-second with only 9 agents (the as above without agents 2,9 and 10) in the largest group.

We believe that an important direction for future work would be to investigate such instances in order to develop move ordering heuristics that would result in higher performance of ID. In particular, we are working on a framework that runs several instances of ID in parallel, each instance using a different search algorithm. The instance with the smallest, among all instances, largest group of inter-dependent agents would be allowed to proceed at any given time.

ID: Large group vs. hard instance

In our experiments, there were many instances, where SIC was a perfect estimate of the solution cost and yet ID formed large groups of inter-dependent agents. For the instance in Figure 3 (right), ID on top of ODA* created a group of six agents (numbered 1,2,5,6,8 and 10 in the figure) although the SIC estimate at the start stage is perfect (that is, there is an optimal solution where each agent follows his individual optimal path; however, the underlying search algorithm happened supply other, conflicting, solutions to ID).

This suggests two insights. First, the observation of Standley that solving the largest group dominates the running time of solving the entire problem is not strictly true. Sometimes, one of the smaller groups contains the conflicts of interest between the agents responsible for large execution times. When analyzing the instances, one should be careful not to be misled by the large group sizes. Second, there may be a lot of space for improving ID, with potential for exponential increase in performance.

OD: exponential number of high-value nodes

Standley uses the following theoretical reasoning to show the benefits of OD: "When coupled with a perfect heuristic and a perfect tie breaking strategy, A* search generates bd nodes where b is the branching factor, and d is the depth of the search. Since the standard algorithm has a branching factor of approximately 9^n (Standley experimented with 8-connected grids) and a depth of t (the number of timesteps in the optimal solution), A* search on the standard state space generates approximately $(9^n)t$ nodes when coupled with a perfect heuristic. A* with OD, however, will generate no more than $9nt$ nodes in the same case because its branching factor is reduced to 9, and its depth only increases to nt. This is an exponential savings with a perfect heuristic."

However, the following theoretical example suggests a possibility for poor node performance of OD. As explained above, OD helps reduce the number of surplus nodes by reducing the branching factor of each node. We show that even with OD, it is possible to have an exponential number of surplus nodes in a MAPF problem instance.

Let n be a full state in the open list with $f(n) = 10$. Assume that the cost of the optimal solution is also 10. This means that all the descendants of n with f-cost larger than 10 are surplus nodes. Using OD, the children of n are the intermediate states where the first agent a_1 has moved. Any child of n that is generated by a_1 making a move that decreases the heuristic value will have the same f-value of n and will also be expanded. In a 4-connected grids with the Manhattan Distance heuristic there can be two such children of n. When each of these children are expanded, they too can generate two nodes with the same f-value as n. Thus, if there are k agents, the node n can have potentially 2^k intermediate nodes that have f-value equal to $f(n)$. Now assume that the last agent do not have any move that decreases the h-value, e.g., because that agent is blocked. This means that all the children of node n are in fact surplus nodes, since non of them lead to a full state with f-value lower than the optimal solution (10). Thus, in such a case a total of 2^{k-1} nodes are surplus nodes.

Since this worst case can potentially occur to every expanded node, then if A* expands X nodes, A*+OD may generate $X \cdot 2^{k-1}$ nodes.

		Unique Nodes Generated, $\times 10^3$						Run-Time, ms					
k	Ins	A*	ODA*	BPEA*	BPEODA*	EPEA*	EPEA*+PDBs	A*	ODA*	BPEA*	BPEODA*	EPEA*	EPEA*+PDBs
Instances solved by both A* and BPEA* within two minutes and 2GB memory													
2-6	793	46.35	1.27	0.08	0.38	0.08	**0.06**	647	5	606	5	**2**	33
7-8	34	1,261.04	3.26	0.11	0.88	0.10	**0.05**	22,440	14	14,886	12	**2**	100
9-10	0	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
Instances solved by neither A* nor BPEA* within two minutes and 2GB memory													
2-6	1	n/a	335.34	n/a	105.14	9.94	**9.31**	n/a	2,153	n/a	1,803	**278**	354
7-8	25	n/a	219.11	n/a	67.04	7.82	**4.41**	n/a	1,637	n/a	1,312	335	**232**
9-10	13	n/a	705.76	n/a	211.54	17.57	**10.01**	n/a	16,660	n/a	8,846	3,062	**1,089**

Table 1: First comparison of different algorithms for MAPF

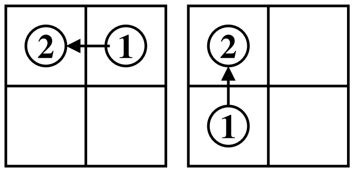

Figure 4: Duplicate detection for intermediate nodes

OD as a method for early duplicate pruning

One of the great benefits of A* is its usage of memory which prevents any path from being explored more than once. In most applications, in order to make sure that a given node is not a re-exploration of a path that has been previously explored, it is enough to check that the open/closed list does not contain the same state with the same or a lower g-value. However, it turns out that this simple condition is not sufficient to prune duplicate **intermediate** states. Therefore, as a default, (Standley 2010) suggests that intermediate nodes not be put on the closed list at all, resulting in 93% savings of memory. As a second option, (Standley 2010) explains how to perform correct duplicate detection of intermediate nodes and concludes that the result is worth the effort. However, he does not explain why this duplicate detection is so effective. We would like to offer an explanation. We contribute our explanation.

Suppose an instance with 20 agents and suppose two intermediate states s_1 and s_2 with the following properties: **(1)** Agents in s_1 occupy the same locations as in s_2, **(2)** in both s_1 and s_2 only the first agent has moved and **(3)** The sets of legal moves possible at s_1 and s_2 are identical. This condition is not trivial. Figure 4 shows two different standard states together with a move of the first agent. Intermediate states with the same locations of agents result. However, in Figure 4 (left), the second agent cannot move to the right, while in Figure 4 (right) it can move to the right, but cannot move down.

Suppose that the duplicate pruning for intermediate nodes is not used. Note that the set of the standard nodes that can result from s_1 by moving the remaining 19 agents is the same as the set of the standard nodes that can result from s_2. All of these nodes will be generated to be pruned. On the other hand, if duplicate pruning at intermediate nodes is used, then all of this work will be saved.

In our opinion, early duplicate detection is one of the main benefits of OD.

Experimental results

We start by an overall comparison of the different variants of A* for solving MAPF optimally. After that, we focus on detailed comparison of EPEA* and ODA*.

Table 1 shows the comparison of five algorithms on a four-connected 8x8 grid with no obstacles with various numbers of agents. Since the ID framework can produce very different result due to reasons that are not related to the performance of the algorithms *per se* as discussed above, we compared the algorithms using the following approach. For each given instance, we first ran ODA* under the ID framework and saved the largest group of inter-dependent agents. Then, the original instance was substituted by another instance, where the agents not in the largest group are discarded. The algorithms were then compared on these instances without the use of ID.

There were a total of 1,000 instances. All algorithms were given up to two minutes and two gigabytes of memory per instance. The results were bucketed according to the number of agents and results for instances falling into the same bucket were averaged. Since the basic A* and BPEA* perform much worse than the other algorithms, we split the table into two halves. The upper part of the table shows results for instances that were solved within allowed resources by both A* and BPEA*. The lower part of the table shows results for instances that were not solved by either A* or BPEA*. We see that none of the instance with 9 or 10 agents were solved by A* and BPEA*, which supports Standley's claim about the importance of the number of agents in the largest group of ID. On the other hand, let us note the time performance of ODA* for the only instance of 2-6 agents that was not solved by either A* or BPEA* – 2,153ms. However, the average time performance of ODA* for the instances of 7-8 agent is 1,637ms. A similar phenomenon can be noted for BPEODA* and EPEA*+PDBs. This shows that the number of agents in the largest group is not a reliable indicator of an instance's hardness.

The following trends can be observed. First, as reported by (Standley 2010), ODA* is faster than A*. Second, BPEA* is faster than A*, but it suffers from the overhead of generating the surplus nodes in order to prune them away. Third, BPEODA* significantly outperforms both A* and ODA* due to maintaining a much smaller open list resulting in cheaper open list operations. EPEA* is faster than all these variants. In addition, for hard instances, PDBs give a significant (up to three times on average) improvement on top of EPEA*. Since the PDBs are instance-dependent, their building times cannot be amortized over all instances.

	Hard for 10 agents (ODA* took 1000 ms or longer)											
	Time[a]	Expanded, $\times 10^3$		Expansions	nFs[b]	Generated, $\times 10^3$						
		Total	Std./Frst.[c]	per node[d]		With duplicates		Unique				
						Tot.	Std.	Tot.	Std.	$f_{static} = c_{opt}$		$f_{static} > c_{opt}$
										Tot.	Exp.	
ODA*	8,435	233.50	20.76	0.31	0.00	983.19	92.12	741.81	71.25	185.67	0.96	323.64
EPEA*	2,376	32.14	20.81	1.46	0.31	561.76	561.76	20.81	20.81	0.10	0.10	0.00

[a]Time in milliseconds.
[b]Number of different stored f-values per node on average.
[c]Standard for OD and first expansion for PE
[d]Number of times a node was expanded on average.

Table 2: Comparison of ODA* and EPEA*

However, we see that building PDBs well pays off for the hard instances. For easier instances, EPEA*+PDBs was still the best algorithm in terms of nodes, but not in terms of time. This is because, besides the nodes generated during the main search, the version with PDBs generates nodes in order to build the PDBs. Hence for easy instances, EPEA* is the current state-of-the-art, while for harder instances, EPEA*+PDBs is.

We now compare of EPEA* and ODA* more deeply in order to understand why EPEA* was better. Table 2 presents detailed statistics for experiments with the two algorithms. The meaning of the columns is explained in the table's footnotes. We need to explore the following trade-off. On the one hand, ODA* provides early duplicate pruning as explained above. On the other hand, ODA* generates surplus and intermediate nodes.

Let us first focus on the last column. We see that the number of surplus nodes is small relative to the total number of generated nodes. Therefore, surplus nodes is not the reason for EPEA*'s advantage over ODA* (but it is the reason for EPEA*'s advantage over the basic A*). Now, let us shift our attention to the total number of unique generated nodes. Here, we see a factor of eight difference between the two algorithms. However, when we look at the total numbers of generated nodes including duplicates, the difference is much smaller. We can conclude that the difference between the numbers of unique generated nodes is due to the large number of intermediate nodes generated by IDA* and that the early duplicated detection does not quite cover that gap.

Conclusions

We presented a study of several variants of A* for optimally solving the multi-agent pathfinding (MAPF) problem. An application of the novel EPEA* technique to MAPF that supports PDBs was fully described and several enhancements proposed. We also presented several important and hitherto uncovered insights about the techniques reported in the existing literature. These insights open several interesting directions for future work.

Acknowledgements

This research was supported by the Israeli Science Foundation (ISF) grant 305/09 to Ariel Felner and the Natural Sciences and Engineering Research Council of Canada grant to Jonathan Schaeffer.

References

Culberson, J. C., and Schaeffer, J. 1998. Pattern databases. *Computational Intelligence* 14(3):318–334.

Dresner, K., and Stone, P. 2008. A multiagent approach to autonomous intersection management. *JAIR* 31:591–656.

Felner, A., and Adler, A. 2005. Solving the 24-puzzle with instance dependent pattern databases. In *SARA-05*, 248–260.

Felner, A.; Zahavi, U.; Holte, R.; Schaeffer, J.; Sturtevant, N.; and Zhang, Z. 2011. Inconsistent heuristics in theory and practice. *Artif. Intell.* 175(9-10):1570–1603.

Felner, A.; Goldenberg, M.; Sharon, G.; Stutervant, N.; Stern, R.; Beja, T.; Schaeffer, J.; and Holte, R. 2012. Partial-expansion a* with selective node generation. In *to appear in AAAI.*

Felner, A.; Korf, R. E.; and Hanan, S. 2004. Additive pattern database heuristics. *Journal of Artificial Intelligence Research* 22:279–318.

Ratner, D., and Warrnuth, M. 1986. Finding a shortest solution for the N $\times$ N extension of the 15-puzzle is intractable. In *AAAI-86*, 168–172.

Sharon, G.; Stern, R.; Goldenberg, M.; and Felner, A. 2011. The increasing cost tree search for optimal multi-agent pathfinding. In *IJCAI*, 662–667.

Sharon, G.; Stern, R.; Felner, A.; and Sturtevant, N. R. 2012. Conflict-based search for optimal multi-agent path finding. In *to appear in AAAI.*

Silver, D. 2005. Cooperative pathfinding. In *AIIDE*, 117–122.

Standley, T. 2010. Finding optimal solutions to cooperative pathfinding problems. In *AAAI*, 173–178.

Yoshizumi, T.; Miura, T.; and Ishida, T. 2000. A* with partial expansion for large branching factor problems. In *AAAI/IAAI*, 923–929.

Zahavi, U.; Felner, A.; Schaeffer, J.; and Sturtevant, N. R. 2007. Inconsistent heuristics. In *National Conference on Artificial Intelligence (AAAI-07)*, 1211–1216.