Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence

Partial-Expansion A* with Selective Node Generation

Ariel Felner
Meir Goldenberg, Guni Sharon
Roni Stern, Tal Beja
ISE Department, Ben-Gurion University (Israel)
felner@bgu.ac.il
{mgoldenbe,gunisharon,roni.stern}@gmail.com

Nathan Sturtevant
CS Department
University of Denver (USA)
Sturtevant@cs.du.edu

Jonathan Schaeffer
Robert C. Holte
CS Department
University of Alberta (Canada)
{Jonathan,holte}@cs.ualberta.ca

Abstract

A* is often described as being 'optimal', in that it expands the minimum number of unique nodes. But, A* may generate many extra nodes which are never expanded. This is a performance loss, especially when the branching factor is large. *Partial Expansion A** (PEA*) (Yoshizumi, Miura, and Ishida 2000) addresses this problem when expanding a node, n, by generating *all* the children of n but only storing children with the same f-cost as n. n is re-inserted into the OPEN list, but with the f-cost of the next best child. This paper introduces an enhanced version of PEA* (EPEA*). Given *a priori* domain knowledge, EPEA* generates only the children with the same f-cost as the parent. EPEA* is generalized to its iterative-deepening variant, EPE-IDA*. For some domains, these algorithms yield substantial performance improvements. State-of-the-art results were obtained for the pancake puzzle and for some multi-agent pathfinding instances. Drawbacks of EPEA* are also discussed.

Introduction

The A* algorithm and its derivatives such as IDA* (Korf 1985) and RBFS (Korf 1993) are general heuristic search solvers guided by the cost function of $f(n) = g(n) + h(n)$. A* is often described as being 'optimal', in that it *expands* the minimum number of unique nodes. If $h(n)$ is admissible (never overestimates the real cost to the goal) then the set of nodes expanded by A* is both necessary and sufficient to find the optimal path to the goal (Dechter and Pearl 1985).[1]

But A* also *generates* many nodes that it doesn't expand. Let X be the number of nodes that A* expands and let b be the average branching factor. Every time a node is expanded, b children are inserted into the OPEN list. Therefore, a total of $b \times X$ nodes are generated. However, once a solution of cost C is found, the algorithm only needs to verify that no solution with cost $< C$ exists. Therefore, when the minimal-cost node in OPEN has $f = C$, the algorithm halts and all other nodes in OPEN (mostly those with $f > C$) are discarded.[2] Nodes with $f > C$ are designated as being *surplus*. The number of surplus nodes in OPEN can grow exponentially in the size of the domain, resulting in significant costs. An extreme example is multi-agent pathfinding, where b itself is exponential in the number of agents.

*Partial Expansion A** (PEA*) (Yoshizumi, Miura, and Ishida 2000) addresses this problem. When PEA* expands a node n, b children are generated but only those with $f = f(n)$ are inserted into OPEN. The rest of the generated children are discarded. n is re-inserted into OPEN, but with the f-cost of its best child that was not placed in OPEN.

This paper takes the idea of PEA* farther. Whereas PEA* generates *all* children, our new approach, called *Enhanced PEA** (EPEA*), generates only those children with the desired f-cost. Given *a priori* domain knowledge, it is often possible to predict the f-cost of the children of a node without actually generating them. With this information, *only* the children that have the desired f-cost need be generated. We discuss methods for creating an *operator selection function* (OSF) to identify operators that will produce the desired f-value for a given state. Only these operators should be applied. We show such OSFs for a number of popular heuristics, including pattern databases. A time and memory analysis gives insights into when the use of an OSF will be effective. The OSF is shown to be also effective in IDA*, yielding *Enhanced Partial Expansion IDA** (EPE-IDA*).

Experiments on puzzle and pathfinding domains show significant speedup, in some cases producing state-of-the art results. The limitations of PEA* and EPEA* are discussed, including an example where EPEA* is slower than A*.

A* with partial expansion

The key idea of PEA* is to never add surplus nodes (with $f > C$) to OPEN (Yoshizumi, Miura, and Ishida 2000). When expanding node n, PEA* first generates a list of *all* the children of n, $CH(n)$. Only nodes c from $CH(n)$ with $f(c) = f(n)$ are added to OPEN. The remaining children are discarded but n is added back to OPEN with the smallest f-value greater than $f(n)$ among the remaining children.[3]

[1] A* has similar guarantees on the set of nodes expanded with an inconsistent heuristic, but may perform many unnecessary re-expansions (Felner et al. 2011). Throughout this paper we assume a consistent heuristic.

[2] Usually, if OPEN breaks ties in favor of small h-values, the goal node with $f = C$ will be expanded as soon as it is generated.

[3] *Collapsing* frontier nodes into one of their ancestors a and only keeping a in OPEN has been used by different search algorithms

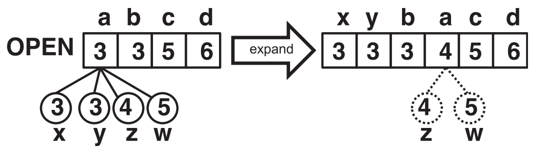
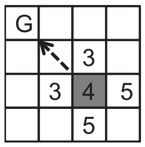


Figure 1: Example of PEA*



Δf	Δh	Operators	v_{next}
0	-1	North, West	1
1	0	Wait	2
2	1	South, East	nil

Figure 2: MD-based OSF table for the shaded cell

We borrow the terminology first used in RBFS. Denote the regular f-value $(g + h)$ of node n as its *static value*, which we denote by $f(n)$ (small f). The value stored in OPEN for n is called the *stored value* of n, which we denote by $F(n)$ (capital F). Initially $F(n) = f(n)$. After it is expanded for the first time, $F(n)$ might be set to $v > f(n)$ when its minimal remaining child has a static f-value of v.

The PEA* idea is shown in Figure 1 where node a is being expanded. All of its children (x, y, z, w) are generated. Then, its children with f-value of 3 (x and y) are inserted into OPEN.[4] Node a is re-inserted to OPEN, but this time with a stored value $F(a) = 4$, the lowest cost of the remaining children (node z). There are now two cases.

Case 1: Re-expansion of *a*: When a is later chosen for expansion with $F = 4$, all its children are generated. Those with $f < 4$ are discarded. Those with $f = 4$ are placed in OPEN (node z). If a has other children with $f > 4$, then a is added back to OPEN with the lowest cost among them (5 in our example).

Case 2: No re-expansion: Assume that one of the descendants of node x with $f = 3$ is the goal. In this case, children of a with costs larger than 3 will never be added to OPEN.

We refer to this algorithm as *Basic PEA** (BPEA*). The memory benefit of Case 2 is straightforward. Assume the goal node has $f = C$. *All* nodes with $f > C$ that were generated (as children of expanded nodes with $f \leq C$) will not be added to OPEN. While Case 2 will only occur for $f = C$, domains with a large branching factor will have a large reduction in the size of OPEN. Indeed, reducing the memory usage of A* was the motivation for creating BPEA* so as to be able to solve larger instances of the multiple sequence alignment problem. (Yoshizumi, Miura, and Ishida 2000) did not analyze the time needs of PEA* (BPEA*) nor did they provide timing results. However, it is easy to see that BPEA* has time tradeoffs, as discussed below.

in several contexts. For example, in SMA* (Russell 1992) once OPEN is too large, areas with the highest f-values in OPEN are collapsed. In IDA*, the end of an iteration can be seen as collapsing the entire tree into the start node with the f-value of the minimal frontier node. In RBFS (Korf 1993) only one branch of the tree (plus the children of nodes of that branch) are kept in memory; any frontier below these nodes is collapsed.

[4]Applicable here is the general method of *immediate expansion* (Stern et al. 2010; Sun et al. 2009): when a generated node has exactly the same f-value as its parent, it bypasses OPEN and is expanded immediately. If this idea is applied, nodes x and y will be immediately expanded and will never be in OPEN with f-values of 3. This method is orthogonal to the ideas of this paper but has been used in all our experiments.

Enhanced PEA*

Assume that we are now expanding node n where $F(n) = K$. BPEA* generates *all* children of n but only those with $f = K$ are inserted into OPEN. Other children (values $f < K$ or $f > K$) are discarded. In contrast, Enhanced PEA* (EPEA*) uses a mechanism which only generates the children with $f = K$, without generating and discarding the children with values $f < K$ or $f > K$. Thus, each node is generated only once throughout the search process and no child is regenerated when its parent is re-expanded.

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 (denoted Δf) of the children of n that they generate. The idea is to use this classification and only apply the operators of the relevant class.

For example, consider a 4-connected grid with 4 cardinal moves and a *wait* move (where the agent stands still). Assume that all moves cost 1 and the heuristic function is *Manhattan Distance* (MD). Without loss of generality, assume that the goal node is located *northwest* of the current location. It is easy to see that MD decreases by 1 when going *north* or *west*, remains the same for the *wait* action, and increases by 1 when going *south* or *east*. This is shown in Figure 2 (left). Numbers inside cells are their h-values and the current node is the shaded cell with $h = 4$.

EPEA* creates a domain-dependent *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.

The OSF for our example is shown in Figure 2 (right) in the form of a table. This table orders the operators according to their Δf when applied to the shaded cell. The first column gives Δf and the third column gives the list of the operators that achieve Δf. The second column, provided for completeness, shows the change in the h-value (Δh). Finally, the last column gives v_{next}, i.e., the next Δf value.

Assume node n is expanded with a stored value $F(n)$ and static value $f(n)$. We only want to generate a child c if $f(c) = F(n)$. 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 OPEN with the next possible value for this node, $f(n) + v_{next}(n, \Delta f)$. If the v_{next} entry is *nil*, meaning no larger value is possible, then node n is moved to the CLOSED list.

Consider Figure 2 with the shaded cell (with $h = 4$) as the start node n. When it is expanded for the first time,

#	Operation	A*		BPEA*		EPEA*	IDA*		EPE-IDA*
1	Remove from OPEN (expand)	X	$<$	αX	$=$	αX	X	=	X
2	Check one operator in OSF_O	0	$=$	0	$<$	$b\alpha X$	0	$<$	bX
3	Check one operator in OSF_D	0	$=$	0	$<$	$\beta\alpha X$	0	$<$	X
4	Generate one child	bX	$<$	$b\alpha$ X	$>$	$\beta\alpha X$	bX	$>$	X
5	Insert one child into OPEN	bX	$>$	$\beta\alpha X$	$=$	$\beta\alpha X$		N/A	
6	Re-insert n into OPEN	0	$<$	$(\alpha-1)X$	$=$	$(\alpha-1)X$		N/A	

Table 1: Time overhead of the different operations

$F(n) = f(n) = 4$ and thus $\Delta f = 0$. Therefore $OSF(n, 0)$ is examined and only the operators in the $\Delta f = 0$ row (*North* and West) are considered. $OSF_{next}(n, 0)$ says that the next Δf value is 1. n is re-inserted into OPEN with $F(n) = 4 + 1 = 5$. Later on, when n gets expanded for the second time, $\Delta f = 5 - 4 = 1$ and only the *wait* operator is applicable. n is re-inserted into OPEN with $F(n) = 6$. Finally, if n gets visited for a third time, only the operators for $\Delta f = 2$ are applied (*South* and *East*). There are no rows for $\Delta f > 2$ (the $next$ value is nil) and n is now closed. Of course, if the goal node is found before n is moved to CLOSED, EPEA* never generates any of the nodes with an f-value larger than the f-value of the goal.

Enhanced partial expansion IDA* (EPE-IDA*)

Partial expansion and OSF work naturally in IDA*. In fact, regular IDA* can be viewed as using basic partial expansion. Assume that the IDA* threshold is T. Once a node n is expanded, all the children are generated. Children with $f \leq T$ are expanded, while children with $f > T$ are generated and discarded. This satisfies the basic partial expansion criteria. However, augmenting IDA* with an OSF (enhanced partial expansion) may significantly reduce the number of node generations. This is done with EPE-IDA*.

EPE-IDA* is simpler than EPEA*. EPE-IDA* associates each node only with its static value. There is only one "stored" value for the entire tree — the value for the next iteration. Given an expanded node n, let $d = T - f(n)$. The OSF will identify the applicable operators with $\Delta f \leq d$. These operators will be applied. Operators with $\Delta f > d$ need not be applied as the children they produce will have f-value $> T$. Thus, for every iteration, EPE-IDA* generates exactly the nodes that will be expanded. The next IDA* threshold is set to the minimal $OSF_{next}()$ value among the expanded nodes that exceeds the current threshold.

Classes of OSFs

The use of enhanced partial expansion for a particular domain requires that, for each node being expanded, the operators applicable to this node can be classified based on their Δf value without having to apply the operators. In general, this will be possible if the g-cost of the operators is known *a priori* and their Δh value can be predicted.

OSFs can be classified into two main types:

Type 1: Ordered by Operators (OSF_O)**:** A general and simple way would be to iterate through all the applicable operators in a state and decide whether it will produce the desired Δf. The cost is linear in the number of applicable operators ($O(b)$). This approach is faster than the traditional approach of applying the different operators, generating the children states, and invoking the heuristic function for each.

Type 2: Direct Function (OSF_D)**:** If applicable, the OSF can be a function which when applied to a state returns exactly the set of operators with the desired Δf value. The cost would be $o(\beta)$ where $\beta \leq b$ is the number of operators with the desired Δf.

Both OSF types may be implemented as a lookup table. Type 2 tables may be faster but the data structures involved are more complicated and need larger amount of memory.

Time Overhead Analysis

Table 1 shows how often the algorithms perform each operation. X is the number of nodes expanded by A* (or IDA*), and b is the branching factor. The right part of the table deals with IDA* and EPE-IDA*. Here, the numbers refer to a given IDA* iteration with threshold T. Summation over all iterations will give the exact numbers for the entire run of IDA*. For IDA*, X nodes are expanded and bX nodes are generated. As explained above EPE-IDA* generates exactly X nodes. As long as the constant time cost per node of EPE-IDA (including operation #2 or #3) is not larger by more than a factor b over the constant node cost of IDA*, EPE-IDA* will run faster.

The A* versions are more complex. Let α be the average number of times BPEA* (and EPEA*) expands the same node to generate some of its children, and β be the average number of children that BPEA* (and EPEA*) adds to OPEN when it expands a node. The inequalities shown are not necessarily strict; it could happen that $\alpha = 1$ and $\beta = b$. Note that if BPEA* and EPEA* eventually add all of the children to OPEN of every node expanded by A* then $\beta\alpha =$ b.

The only operation that is executed fewer times by BPEA* than by A* is inserting a child into OPEN ($\#5$). By contrast, it expands ($\#1$) and generates ($\#4$) nodes a factor of α times more than A* and re-inserts nodes $(\alpha - 1)X$ more nodes ($\#6$). BPEA* will run faster than A* only if 1) $\beta\alpha$ is sufficiently smaller than b, and 2) the insertion operation ($\#5$) is sufficiently costly compared to the operations that are executed more times by BPEA*.[5]

There is one operation that BPEA* performs more frequently than EPEA* ($\#4$) and one that EPEA* performs

[5]Operation $\#6$ might be implemented as a *change-priority* action on OPEN. This includes $(\alpha - 1)X$ of the operations listed for operator $\#1$. Such an implementation has speedup potential.

	IDA*	EPE-IDA*	Ratio
Gen Nodes	363,028,079	184,336,705	1.97
Time (ms)	17,537	14,020	1.25

Table 2: Experimental results: 15-puzzle

more frequently than BPEA* (#2 or #3). The relative cost and relative frequencies of these operations will determine how much faster (or slower) EPEA* will be than BPEA* (and similarly, for EPE-IDA* versus IDA*).

Thus, depending on the branching factor and how efficiently an OSF can be implemented for a given domain, any of these algorithms could be the fastest, the slowest, or in between the other two. Note however, that in many cases, especially in exponential domains with a large b, the number of *surplus nodes* with $f > C$ is up to b times more than X. Most of their parents (which are included in X) will have small values of α and β. On average, in such cases $\beta\alpha << b$, and EPEA* will run faster than A*. This is the case for many of the domains we experimented with.

Experiments: 15-Puzzle (MD heuristic)

Optimal solutions to random instances of the 15-puzzle were first found by (Korf 1985) using IDA* and Manhattan Distance. Korf graciously made his code available to the public. This piece of code is known to be highly optimized and runs extremely fast. While reading the code, we realized that Korf actually implemented an OSF function in the form of a lookup table ordered by operators (OSF_O). For each neighboring tile of the blank and its location, Korf's OSF table returns the Δf value for each operator. However, Korf did not implement EPE-IDA* to its full extent. He first generates a node and then consults the OSF to calculate the heuristic.

EPE-IDA* required only a small change to the code. First, consult the OSF. Then, only apply those operators that will produce children with $f \leq T$ Unlike Korf's original code, nodes with $f > T$ are not generated.

Table 2 presents the results of running Korf's IDA* code and EPE-IDA* on his 100 random instances. A factor of 2 reduction in nodes is seen. This translates to a 1.25-fold improvement in run-time. Improving the run-time of this highly efficient code is a nice achievement. All our experiments (except the last section) were conducted on a 3.0GHz Intel dual-core processor under Windows 7.

Experiments: Rubik's Cube (Delta PDBs)

A pattern database (PDB) is based on an abstraction ϕ of the state space (Culberson and Schaeffer 1996). The PDB contains the distance from an abstract state to the abstract goal. When a heuristic value is required for state p one simply looks up PDB[$\phi(p)$]. An OSF for any PDB can be built in a preprocessing step by applying each operator σ to each abstract state a and recording the difference between PDB[$\sigma(a)$] and PDB[a]. This creates a data structure that is indexed by abstract states and operators. We call this data structure the *Delta PDB* or Δ-PDB.[6] The entry Δ-PDB[a, σ] indicates how much the heuristic value $h(p)$ will change when σ is applied to any state p such that $\phi(p)$=a. In state spaces where all operators have a cost of one, only two bits are needed for each entry since the difference between distances to the abstract goal of an abstract state and one of its children will be either -1, 0, or 1.[7] Given an original state (p) and desired change in f-value (v), scan through all the operators (σ), look up Δ-PDB[$\phi(p), \sigma$], and return those operators for which Δ-PDB[$\phi(p), \sigma$]$-$cost(σ) is equal to v.

#	IDA*	EPE-IDA	ratio	IDA*	EIDA*	ratio
Rubik's Cube (Corner PDB)						
	Generated Nodes - Thousands			Time (mm:ss)		
13	434,671	32,610	13.32	0:53	0:15	3.53
14	3,170,960	237,343	13.37	5:31	1:32	3.68
15	100,813,966	7,579,073	13.30	175:25	47:16	3.71
Pancake Puzzle (GAP Heuristic)						
	Generated Nodes			Time (ms)		
20	18,592	1,042	17.84	1.5	0.1	11.23
30	241,947	8,655	27.95	24.9	1.2	20.00
40	1,928,771	50,777	37.98	247	8.5	30.75
50	13,671,072	284,838	47.99	2,058	57	36.15
60	92,816,534	1,600,315	57.99	16,268	359	45.32
70	754,845,658	11,101,091	67.99	155,037	2,821	54.90

Table 3: Rubik's Cube and pancake puzzle results

This technique was applied to the Rubik's Cube PDB based on the corner cubies (Korf 1997). This abstraction has $88,179,840$ abstract states. In the Δ-PDB, each of these states further included an array of size 18, one index per operator (each requiring 2 bits) for a total of 396,809,280 bytes of memory (396 MB, compared to 42 MB for the original PDB). Note that the Δ-PDB can be potentially compressed to reduce its memory needs.

Results using the corner Δ-PDB are given in Table 3 (top). Each line is the average over 100 instances of depth 13-15. The reduction (*ratio* column) in the number of nodes generated is a factor of 13.3 (the known effective branching factor) and the time improvement is only 3.7-fold. The reason for the discrepancy is that the constant time per node of EPE-IDA* is larger than that of IDA* since it includes the time to retrieve values from the Δ-PDB which was ordered by operators (OSF_O) (#2 in Table 1).

Experiments: Pancake Puzzle (GAP heuristic)

In the pancake puzzle (Dweighter 1975), a state is a permutation of the values $1...N$. Each state has $N - 1$ children, with the k^{th} successor formed by reversing the order of the first $k + 1$ elements of the permutation ($1 \leq k < N$). For example, if $N = 4$ the children of state $\{1, 2, 3, 4\}$ are $\{2, 1, 3, 4\}$, $\{3, 2, 1, 4\}$ and $\{4, 3, 2, 1\}$. A number of PDB heuristics have been used for this puzzle (Zahavi et al. 2008;

[6]The idea of Δ-PDBs was independently discovered by (Schreiber and Korf 2012) under the name Locality-Preserving PDB. Their work focuses on the cost of cache-misses with memory-based heuristics

[7]Note that (Breyer and Korf 2010) compressed ordinary PDBs to use two bits by only storing the h-value modulo 3.

Felner et al. 2011; Yang et al. 2008), but the recently introduced GAP heuristic significantly outperforms them all (Helmert 2010).

Two integers a and b are *consecutive* if $|a - b| = 1$. For a given state, a *match* at location j occurs when the pancakes at location j and $j+1$ are consecutive. Similarly, a *mismatch* occurs when these pancakes are not consecutive. The goal can be defined such that pancake 1 is at location 1 and there are no mismatches. Note that in the case of a mismatch there must be a unique reverse operation that will separate these two pancakes and will not influence the relative positions of any other adjacent pancakes. The GAP heuristic iterates through the state and counts the number of mismatches. This heuristic is admissible.

A reverse operator of j pancakes only affects the match/mismatch situation of three pancakes. Assume the mismatch is between locations j and $j + 1$. The affected locations are 1 (named P), j (named X), and $j + 1$ (named Y). Originally, X and Y were adjacent but now P and Y are adjacent. There are three cases for Δh in the GAP heuristic:
(Case 1) $\Delta h = -1$. A mismatch is removed: X and Y were not consecutive but P and Y are.
(Case 2) $\Delta h = 0$. No change: X and Y are (are not) consecutive and P and Y are (are not).
(Case 3) $\Delta h = 1$. A mismatch is introduced: X and Y were consecutive, but P and Y are not.

Each of these cases can be recognized by looking only at the three affected pancakes. Thus, a direct OSF was built.[8]

The experimental results for 100 random instances for 10 to 70 pancakes are given in Table 3 (bottom). For 70 pancakes, EPE-IDA* generated 68 times fewer nodes than IDA*. Most of this is reflected in the running time (54-fold) as here a direct OSF was used and the overhead was very small (#3). To the best of our knowledge, these are the state-of-the-art results for this puzzle.

Experiments: Multi-agent Pathfinding (SIC)

Experiments were performed in the multi-agent pathfinding domain (MAPF) (Standley 2010) where we seek for optimal solutions. In MAPF, we are given a graph $G = (V, E)$ and k agents. Each agent has a start position and goal position. The task is to move all the agents to their goals without colliding into other agents (i.e., two agents cannot be in the same location at the same time) while minimizing a cumulative cost function. The straightforward optimal solver uses A* where the state space includes all possible permutations of the k agents on the $|V|$ vertices. A commonly-used heuristic function is the *sum of individual costs* (SIC), the summation of the individual shortest paths ignoring the presence of other agents.

If b is the number of moves of each individual agent, then up to b^k legal operators can be available and the number of surplus nodes (with $f > C$) grows exponentially in the number of agents. (Standley 2010) introduced *operator decomposition* (OD) which reduces the number of surplus nodes generated for MAPF. OD introduces *intermediate nodes* between the regular states of the A* search as follows. Agents are assigned an arbitrary (but fixed) order. When a regular A* state 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 OPEN are not developed further into regular nodes, so that the number of surplus nodes is significantly reduced. This variant of A* is referred to as ODA*. ODA* can still generate surplus nodes, both intermediate and regular. By contrast, EPEA* never generates any surplus node.

(Sharon et al. 2011a; 2011b) introduced the *increasing cost tree search* (ICTS) algorithm for MAPF. It is a two-level algorithm where the high-level searches combinations of individual agent costs, while the low-level checks whether there is a valid solution for the given cost. ICTS was shown to outperform A* and ODA* in a large class of instances.

EPEA* was implemented for MAPF with an OSF that works in two stages. For each agent a_i we pre-compute and store a special OSF table (OSF_i) while ignoring other agents. For each vertex of the graph, OSF_i stores the Δf value for each possible move that a_i can take. OSF_i is ordered by Δf. Now, the first stage computes all possible combinations of Δf for the individual agents that sum up to the desired cumulative Δf. For example, assume two agents a_1 and a_2 and that the desired cumulative $\Delta f = 3$. Suppose that OSF_1 says that a_1 has operators with individual Δf of 0, 1 and 2, and that OSF_2 says that a_2 has operators with Δf of 0, 2 and 3. Then, the following combinations of individual Δf's are produced: (0,3) and (1,2). The second stage expands these combinations of individual Δf's into combinations of concrete operators, while pruning away the combinations that result in collisions between the agents.

Following (Standley 2010; Sharon et al. 2011a), 3x3 and 8x8 grids with no obstacles were tested. Results are shown in Table 4. Values are averaged over all random instances (out of 100) that were solved under 5 minutes by all algorithms (except those labeled by "NA"). The number of such instances solved is shown in the *Ins* column.[9] Five algorithms were compared: A*, ODA*, BPEA*[10], EPEA* and the highest performing variant of ICTS (Enhanced ICTS, $ICTS_E$) (Sharon et al. 2011b). Following (Sharon et al. 2011a) we report only time for ICTS, because ICTS has components that are not based on ordinary search.

[8]In fact, it was purely direct only for $\Delta h = -1$ (i.e., for $\Delta f = 0$). In this case, all that is needed, is to check the two operators where Y is either $P + 1$ or $P - 1$ and confirm that X is not $P - 1$ or $P + 1$, respectively. For the other cases, the OSF was not purely direct and there were more technical details which we omit here. Nevertheless, the number of times $\Delta f = 0$ was called significantly dominated all other cases of $\Delta f = 1$ and $\Delta f = 2$

[9](Standley 2010) introduced *independent detection* (ID) which identifies independent groups of agents; each is treated as a separate sub-problem. We followed (Sharon et al. 2011a) and used ID. Results reported are for k agents which are all in the same group.

[10]BPEA* can be implemented on top of OD (BPEODA*). In our experiments it was outperformed by EPEA* by a factor of 3.

		Nodes Generated				Run-Time (ms)				
k	Ins	A*	ODA*	BPEA*	EPEA*	A*	ODA*	BPEA*	$ICTS_E$	EPEA*
3x3 grid										
5	100	780	255	3,433	16	100	7	95	7	**4**
6	100	2,767	1,134	14,126	48	660	40	494	90	**25**
7	100	6,634	5,425	42,205	144	3,593	777	2,190	2,761	**176**
8	100	9,003	16,029	39,344	235	5,187	3,475	4,371	71,612	**583**
8x8 grid										
5	100	19,061	556	36,948	27	9,311	7	743	**6**	**6**
6	100	NA	1,468	353,479	54	NA	49	6,881	**13**	17
7	95	NA	2,401	1,676,093	71	NA	68	33,347	37	**26**
8	99	NA	8,035	NA	182	NA	593	NA	125	**99**
9	94	NA	30,707	NA	616	NA	6,612	NA	897	**438**
10	96	NA	NA	NA	2,448	NA	NA	NA	3,865	**3,125**
11	81	NA	NA	NA	2,739	NA	NA	NA	5,038	**4,324**
12	89	NA	NA	NA	11,343	NA	NA	NA	25,369	**24,404**

Table 4: Multi-agent pathfinding results

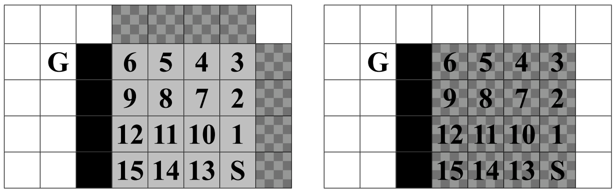


Figure 3: OPEN (checkerboard) and CLOSED (gray) states after 16 node expansions for A* (left) and PEA* (right)

The following trends can be observed. First, as reported by (Standley 2010) ODA* is faster than A*. Second, BPEA* is faster than A*, but still suffers from generating the surplus nodes (without putting them into OPEN). Third, EPEA* generates significantly fewer nodes than BPEA* and ODA*, since it does not generate any surplus node. Timing results show that EPEA* outperforms ODA* by a factor of up to a full order of magnitude. Furthermore, EPEA* was faster than the enhanced version of ICTS. This trend held even for the 8x8 grid, where ICTS was relatively strong in the experiments of (Sharon et al. 2011a) .

Case Study: Single-agent Pathfinding (MD)

BPEA* and EPEA* have limitations and might perform worse than A* in some cases. Single-agent path-finding on a 4-connected 2D grid with obstacles is used to illustrate this. The OSF is a direct function, and stored in a pre-computed table of incremental costs (similar to Figure 2 (right)). The heuristic function is Manhattan Distance.

In this domain EPEA* will not necessarily outperform A*. Consider Figure 3 (left) where the shortest path is needed from cell S to cell G. Black cells are obstacles. Note that the shortest path is of length 10 but the 16 gray cells all have f-value of 8. Thus, A* starts expanding states with $f = 8$. After 16 expansions all these states are in CLOSED, and the 8 states around them (marked with a checkerboard) are in OPEN, all with $f = 10$.

Now consider EPEA* (and BPEA*). When EPEA* expands the gray nodes it does not close them because each has a child with $f = 10$. For example, when expanding state 10 the OSF says that there are two operators (north and west) with $\Delta f = 0$ and one operator (east or south) with $\Delta f = 2$. The cell east of 10 is cell 1 which was already generated with a lower g-value and with $f = 8$ but EPEA* cannot perform duplicate detection without actually generating the successor. Thus, cell 10 is re-inserted into OPEN with $f = 10$, received from cell 1. Consequently, after expanding all the gray cells for the first time, EPEA* has 16 nodes in OPEN as shown in Figure 3 (right) and none in CLOSED. In terms of memory EPEA* has an advantage as it only stores 16 states while A* stores 16+8=24 states. However, if OPEN is implemented as a heap, there will be log overhead for all heap operations, and so a larger OPEN can greatly increase the running time. On a grid map where there were previously $O(r)$ states in OPEN at depth r, there is now the potential for $O(r^2)$ states to be in OPEN at depth r. That is, where A* will only store the perimeter of the examined states in OPEN, EPEA* potentially can store all states. In general, when there are many small cycles and a small branching factor, EPEA* will likely have worse performance than A*.

For grids, however, OPEN can be implemented using f-cost buckets, which have amortized constant-time overhead. We thus tested all algorithms with OPEN implemented both as a heap (logarithmic) and with buckets (constant). The problem set tested is all 3,803 problem instances from the pathfinding repository (Sturtevant 2012) of optimal octile length 508-512. The experiments were run on a 2.66GHz Intel Core i7 processor running Mac OS X, and are summarized in Table 5. The table shows the average time to solve a problem in the set as well as the average number of states in OPEN when the goal is found for the different algorithms with the different OPEN implementations. A* is the fastest and has the fewest number of states in OPEN. The constant-time OPEN saves about 5ms per problem. EPEA* has the next-best performance. It has over three times the number of nodes on OPEN as A*. The constant-time OPEN is 13ms faster than the log-time OPEN.

	Logarithmic		Constant	
Algorithm	Time (ms)	OPEN	Time (ms)	OPEN
A*	29.1	648	24.7	645
EPEA*	44.2	2371	31.5	2115
BPEA*	44.4	2349	34.0	2117
BPEA*$_{dd}$	47.7	1908	36.2	1724
BPEA*$_{dd-o}$	53.0	741	-	-

Table 5: Comparison between different PEA* variants

Three variants of BPEA* were implemented. The first one completely ignores states with f-cost larger than the current state. This has similar performance to EPEA* with the log-time OPEN, but worse performance compared to constant-time OPEN. The second implementation, BPEA*$_{dd}$, performs duplicate detection of all successors of a state. In some cases this allows it to remove additional states from OPEN, at the cost of looking up more states in OPEN/CLOSED. There was a small reduction in the size of OPEN, but it is still larger than that of A*. This is dependent on the order that nodes are expanded. The third variant, BPEA*$_{dd-o}$, breaks ties in f-cost towards nodes with low g-cost. This can significantly increase the number of nodes expanded but it also reduces the size of OPEN and this is reflected in the running time.

In summary, EPEA* is faster than BPEA* on 4-connected grids. However with a small branching factor the potential speedup is limited as the number of surplus nodes is modest. Similarly, larger OPEN increased the time to perform OPEN-based operations (#1, #5 and #6). Thus, EPEA* and BPEA* were not able to outperform A* in such domains.

Conclusions

This paper presented the enhanced versions of PEA* and IDA*, including showing a number of methods for producing OSFs. The success of enhanced partial expansion depends on properties of the domain. In general, in exponential domains b is large and the potential savings are large, as many states that would otherwise be generated are now bypassed. Experimental results on such domains support this trend and state-of-the-art results were obtained for the pancake puzzle and MAPF. In contrast, in polynomial domains with small b and many cycles, there is the danger of inflating the OPEN list, thus reducing the potential for performance gains despite fewer distinct nodes being generated.

Future work will further study different ways to build OSFs. Similarly, deeper analysis on the characteristics of these algorithms will further explain the circumstances for potential performance improvements.

Acknowledgements

This research was supported by the Israeli Sience Foundation (ISF) under grant #305/09 to Ariel Felner and by NSREC.

References

Breyer, T. M., and Korf, R. E. 2010. 1.6-bit pattern databases. In *AAAI*, 39—44.

Culberson, J. C., and Schaeffer, J. 1996. Searching with pattern databases. *Advances in Artificial Intelligence* 402–416. LNAI 1081.

Dechter, R., and Pearl, J. 1985. Generalized best-first search strategies and the optimality of A*. *Journal of the ACM* 32(3):505–536.

Dweighter, H. 1975. Problem e2569. *American Mathematical Monthly* 82:1010.

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

Helmert, M. 2010. Landmark heuristics for the pancake problem. In *SOCS*, 109–110.

Korf, R. E. 1985. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence* 27(1):97–109.

Korf, R. E. 1993. Linear-space best-first search. *Artificial Intelligence* 62(1):41–78.

Korf, R. E. 1997. Finding optimal solutions to Rubik's Cube using pattern databases. In *AAAI*, 700–705.

Russell, S. J. 1992. Efficient memory-bounded search methods. *ECAI* 1–5.

Schreiber, E. L., and Korf, R. E. 2012. Locality-preserving pattern databases. Technical Report 120010, Department of Computer Science, University of California, Los Angeles.

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

Sharon, G.; Stern, R.; Goldenberg, M.; and Felner, A. 2011b. Pruning techniques for the increasing cost tree search for optimal multi-agent pathfinding. In *SOCS*, 150–157.

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

Stern, R.; Kulberis, T.; Felner, A.; and Holte, R. 2010. Using lookaheads with optimal best-first search. In *AAAI*, 185–190.

Sturtevant, N. 2012. Benchmarks for grid-based pathfinding. *Transactions on Computational Intelligence and AI in Games*.

Sun, X.; Yeoh, W.; Chen, P.; and Koenig, S. 2009. Simple optimization techniques for A*-based search. In *AAMAS*, 931–936.

Yang, F.; Culberson, J.; Holte, R. C.; Zahavi, U.; and Felner, A. 2008. A general theory of additive state space abstractions. *JAIR* 32:631–662.

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

Zahavi, U.; Felner, A.; Holte, R. C.; and Schaeffer, J. 2008. Duality in permutation state spaces and the dual search algorithm. *Artificial Intelligence* 172(4–5):514–540.