Journal of Artificial Intelligence Research 50 (2014) 141-187 Submitted 9/13; published 5/14

Enhanced Partial Expansion A*

Meir Goldenberg MGOLDENBE@GMAIL.COM
Ariel Felner FELNER@BGU.AC.IL
Roni Stern RONI.STERN@GMAIL.COM
Guni Sharon GUNISHARON@GMAIL.COM
Ben-Gurion University of the Negev
Beer-Sheva, Israel

Nathan Sturtevant STURTEVANT@CS.DU.EDU
The University of Denver,
Denver, USA

Robert C. Holte HOLTE@CS.UALBERTA.CA
Jonathan Schaeffer JONATHAN@CS.UALBERTA.CA
The University of Alberta
Edmonton, Canada

Abstract

When solving instances of problem domains that feature a large branching factor, A* may generate a large number of nodes whose cost is greater than the cost of the optimal solution. We designate such nodes as *surplus*. Generating surplus nodes and adding them to the OPEN list may dominate both time and memory of the search. A recently introduced variant of A* called *Partial Expansion A** (PEA*) deals with the memory aspect of this problem. When expanding a node n, PEA* generates *all* of its children and puts into OPEN only the children with $f = f(n)$. n is re-inserted in the OPEN list with the f-cost of the best discarded child. This guarantees that surplus nodes are not inserted into OPEN.

In this paper, we present a novel variant of A* called *Enhanced Partial Expansion A** (EPEA*) that advances the idea of PEA* to address the time aspect. Given *a priori* domain- and heuristic-specific knowledge, EPEA* generates *only* the nodes with $f = f(n)$. Although EPEA* is not always applicable or practical, we study several variants of EPEA*, which make it applicable to a large number of domains and heuristics. In particular, the ideas of EPEA* are applicable to IDA* and to the domains where pattern databases are traditionally used. Experimental studies show significant improvements in run-time and memory performance for several standard benchmark applications. We provide several theoretical studies to facilitate an understanding of the new algorithm.

1. Introduction

A* and its derivatives such as IDA* (Korf, 1985) and RBFS (Korf, 1993) are general state-based search solvers guided by the cost function $f(n) = g(n) + h(n)$. Given an *admissible* (i.e. non-overestimating) heuristic function h, A* is guaranteed to find an optimal solution. Until the paper by Yoshizumi, Miura, and Ishida (2000), performance studies of A* mostly concentrated on the number of nodes that A* expanded.[1] Furthermore, a general result about optimality of A* with

1. Papers on memory-bounded A* such as the work of Russell (1992) and the more recent work of Zhou and Hansen (2002) form one notable exception to this statement. However, saving time of heap operations by reducing the size of the OPEN list was not the central focus of these papers.

respect to the expanded nodes has been established (Dechter & Pearl, 1985). However, the number of expanded nodes clearly does not represent the performance of A* accurately enough. Rather, the run-time performance of A* is determined by the net total of many factors, such as the number of generated nodes and the time of updating the data structures (the OPEN and CLOSED lists). Motivated by the need to view the performance of A* in this general context, we focus on both the number of nodes that A* expands and the number of nodes that A* generates.

Let C^* be the cost of the optimal solution for a particular problem instance. A* has to expand all nodes n with $f(n) < C^*$ to guarantee optimality of the solution. These nodes have to be generated as well. However, A* generates many nodes with $f(n) = C^*$ and nodes with $f(n) > C^*$ as well. We call the latter group of nodes *surplus* (with $f(n) > C^*$). Surplus nodes are never expanded by A* and thus do not contribute to finding an optimal solution. Non-surplus nodes are called *useful*.

Many important problem domains feature a large branching factor. When solving these problems, A* may generate a large number of surplus nodes. In fact, the execution time spent on generating these nodes may be much larger than the time spent on generating and expanding the useful nodes. When solving instances of such domains, the number of generated nodes is a central performance issue.

A first step to address this problem in a general manner was taken by Yoshizumi et al. (2000). They introduced a variant of A* called *Partial Expansion A** (PEA*), which deals with the memory aspect of the surplus nodes problem. Throughout the paper, n denotes the current node being expanded (unless a different meaning is explicitly specified), while n_c denotes a child of n. When PEA* expands n, it generates *all* of n's children and adds to OPEN only the children with $f(n_c) = f(n)$. This guarantees that surplus nodes are not inserted into OPEN. The rest of the children are discarded. n is re-inserted into OPEN with the f-cost of the best discarded child with $f(n_c) > f(n)$ (the one with minimal f-value). A node n can be expanded in this way several times, each time with a different $f(n)$. Note that PEA* addresses only the problem of high memory consumption caused by storing surplus nodes in OPEN. At the same time, PEA* incurs a high price in terms of run-time performance, since it may have to generate children of a given node more than once.

We preface the presentation of our contributions with the following observation. We know that A* is optimal with respect to the number of *expanded* nodes, but does not give any optimality guarantees with respect to the number of *generated* nodes. The reason for the optimality of A* is the way it consults the heuristic to decide which node to expand next (i.e. the node with a minimal $g(n) + h(n)$ among the nodes in OPEN). However, A* does not take $h(n)$ into consideration at all when it generates the children of n. If A* could know, based on h, that n_c is surplus without actually generating n_c, then we could avoid generating the surplus nodes altogether.

This observation leads directly to the first and most important contribution of this paper: a variant of A* called *Enhanced Partial Expansion A** (EPEA*) (Section 3). EPEA* generates *only* those children with $f(n_c) = f(n)$. In contrast to PEA*, the other children are not even generated. This is enabled by using *a priori* domain- and heuristic-specific knowledge to compute the list of operators that lead to the children with the needed f-cost without actually generating any children. This knowledge and the algorithm for using it (also domain- and heuristic-specific) form an *Operator Selection Function* (OSF). For a given node expansion, an OSF returns the set of children with $f(n_c) = f(n)$ and the smallest f-value among the children with $f(n_c) > f(n)$.

The following example gives a preview of EPEA*. Consider the four-connected grid-based single-agent pathfinding domain (SAPF) with the Manhattan distance heuristic. The OSF represents the following piece of domain- and heuristic-specific knowledge: "if the goal is to the North-West of

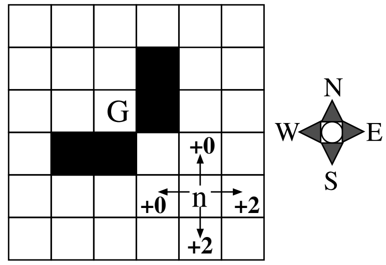


Figure 1: EPEA* uses domain and heuristic-specific knowledge to obtain the set of children n_c of the node n being expanded with $f(n_c) = f(n)$.

the current location of the agent, then moving North or West will result in the same f-value, while moving South or East will increase the f-value by two". An example of this knowledge is shown in Figure 1. When EPEA* expands node n for the first time, it generates only the children nodes with $f(n_c) = f(n)$. The OSF tells EPEA* that these nodes result from applying the operators North and West. Also, the OSF determines that the cost of n can be increased by 2, corresponding to the cost of the best child that is not being generated. EPEA* keeps n with the new cost in OPEN. When n is re-expanded, the children of n corresponding to the remaining operators (i.e. South and East) are generated.

OSFs are domain and heuristic-dependent. The minimum prerequisite for applying EPEA* to solve problem instances of a particular domain with a particular heuristic is the existence of the *full-checking OSF* that we will introduce in Section 5 for those specific domain and heuristic.

The paper is organized as follows. PEA* is described in Section 2. We note that PEA* can be described in terms of *Collapsing the Frontier Nodes* (CFN) – a general technique that has been used by a number of well known algorithms. Section 3 presents EPEA*. Section 4 extends our ideas to IDA* resulting in a variant of IDA* called *Enhanced Partial Expansion IDA** (EPEIDA*). Sections 5-8 describe and study experimentally the different kinds of OSFs. Sections 9 and 10 presents analytical insights into aspects of EPEA*. Section 11 touches upon the topic of using PEA* and EPEA* with an inconsistent heuristic and exposes the trade-off that exists in that situation.

The paper introduces a significant amount of terminology. For the reader's convenience, we provided a glossary of terms in Appendix A.

A preliminary version of this paper was presented at AAAI-2012 (Felner et al., 2012). The current paper extends the conference version by a deeper treatment of the EPEA* algorithm including an extended set of experimental and theoretical studies.

2. Background Knowledge

EPEA* advances the idea of *Partial Expansion A** (PEA*) (Yoshizumi et al., 2000). We describe PEA* in the general context of the technique called *Collapsing Frontier Nodes* (CFN).

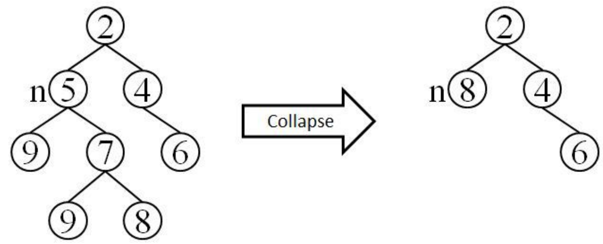


Figure 2: The subtree rooted at the node n is collapsed into n, which receives the new stored value $F(n) = 8$.

2.1 Collapsing Frontier Nodes

Designate the set of leaf nodes of the search tree as the *frontier* of the search. Best-first search algorithms choose a frontier node with the lowest cost for expansion. When n is expanded, n is removed from the frontier and its children are added to the frontier. The frontier can be maintained either in memory (OPEN in A*) or only logically (IDA*, RBFS). For example, the frontier of IDA* for a given iteration are all the nodes that have been generated but not expanded in that iteration (these nodes would have been stored in OPEN if A* was used). In either case, the following *admissibility invariant* is maintained: for every node n in the frontier, the cost of n does not exceed the cost of any possible solution that passes through n.

Let n be a non-leaf node of the search tree and R be a subset of the frontier, such that, for all $r \in R$, the path in the search tree from the root to r passes through n. That is, n is a common ancestor of the nodes in R. The *Collapsing Frontier Nodes* (CFN) technique relies on the observation that it is possible to obtain a smaller frontier by replacing R with n. Furthermore, the cost of n can be increased, without violating the admissibility invariant, to the minimum cost among the nodes in R. This is illustrated in Figure 2 where the part of the frontier corresponding to the entire left subtree (i.e. R consists of three nodes with costs 9, 9 and 8) is *collapsed* into the node n. The cost of n is now modified to be the minimum cost among the collapsed leaves (8 in this case).

Such *collapse* actions have been used by many algorithms. In SMA* (Russell, 1992), for example, once OPEN is too large, areas with the highest f-values in OPEN are collapsed. As another example, consider IDA*. Once an IDA* iteration ends, it can be seen as collapsing the entire frontier into the start node, which gets the f-value of the least cost frontier node – the threshold for the next iteration. In ITS (Ghosh, Mahanti, & Nau, 1994), IDA* is allowed to use a given amount of memory. When that memory limit is exceeded, collapsing is used to remove the nodes with the largest cost. Another important example is RBFS. Here only one branch of the tree plus the children of nodes in that branch are kept in memory at all times. Any frontier below these nodes is collapsed.

We use the terminology coined by Korf (1993). The regular $f = g + h$ value of a node is designated as its *static value* and denoted by $f(n)$. The f-value of the leaves that was propagated up to their common ancestor n by a collapse action is called the *stored value* of n and denoted by $F(n)$. The *static value* of n in Figure 2 is 5, while its *stored value* after the collapse action is 8. When we reach a node n in OPEN with a stored value $F(n) = x$ such that $x > f(n)$, we know that n has been expanded before and that its least-cost frontier node had the static f-value of x. It is

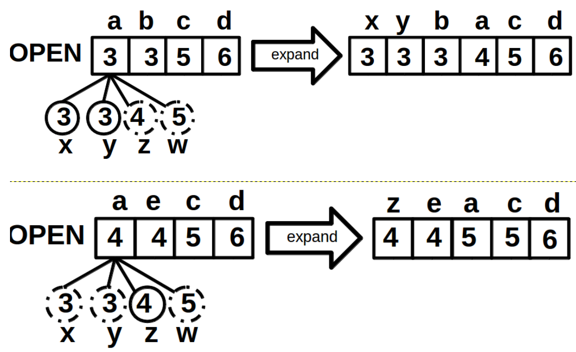


Figure 3: Example of PEA* and EPEA*. Two expansions of the node a are shown. Solid circles denote the generated children, while the non-solid circles denote the discarded/collapsed children.

important to note that each time a set of nodes is collapsed into n, the stored value of n may grow. The stored value of a node never decreases.

2.2 Partial Expansion A*

*Partial Expansion A** (PEA*) (Yoshizumi et al., 2000) is a variant of A* that reduces the memory overhead of A* by not putting the surplus nodes into OPEN. PEA* preserves admissibility by using collapse actions. EPEA*, the new variant of A* introduced in this paper, works similarly to PEA*. Therefore, a clear grasp of PEA* will be essential for understanding EPEA*. In the following, we will (1) define PEA* and (2) discuss the performance overheads of PEA* that EPEA* addresses. For simplicity, we assume a consistent heuristic.[2] We will touch upon the case of an inconsistent heuristic in Section 11.

PEA* maintains both the static and the stored value for each node. The stored value is obtained by collapse actions as we describe below. PEA* expands nodes in the order of the least stored value. To avoid putting surplus nodes into OPEN, PEA* distinguishes two kinds of n_c:

1. *Provably useful*, i.e. the children with $f(n_c) \leq F(n)$. Since n satisfies the admissibility invariant, we have $F(n) \leq C^*$. We infer that $f(n_c) \leq C^*$, which means that n_c is useful. Thus, checking that $f(n_c) \leq F(n)$ proves that n_c is useful. In other words the children with $f(n_c) \leq F(n)$ are provably useful in the sense that A* cannot be optimal without putting these nodes into the OPEN list. That a node is provably useful does not necessary imply that A* will expand this node.

2. *Possibly surplus*, i.e. the children with $f(n_c) > F(n)$. Such a node n_c might be useful, but this cannot be proven at this point in the search. Note that a possibly surplus n_c can become provably useful when n is re-expanded with a higher stored value.

2. A heuristic h is called *consistent* if, for every node n and its child n_c the inequality $h(n) \leq h(n_c) + cost(n, n_c)$ holds. Equivalently, a heuristic h is consistent if it results in monotonically increasing f-values, i.e. if for every node n and its child n_c, the inequality $f(n_c) \geq f(n)$ holds.

When PEA* expands n, it generates all children of n, but inserts into OPEN only the provably useful children with $f(n_c) = F(n)$. Since $F(n)$ can only grow, the provably useful children with $f(n_c) < F(n)$ are known to have been put into OPEN by previous expansions of n. The possibly surplus children of n are collapsed into n, which is re-inserted into OPEN with the stored value obtained by this collapse action. If n does not have possibly surplus children, it is not re-inserted into OPEN, but is moved to CLOSED instead.

We refer to the provably useful children with $f(n_c) = F(n)$ as *currently needed*. We refer to the other children (i.e. the possibly surplus children and the provably useful children with $f(n_c) < F(n)$) as *currently unneeded*. In other words, currently needed children of n are the ones that PEA* would insert into OPEN during the current expansion of n. We designate operators that result in currently needed children as *currently needed*. The other operators are designated as *currently unneeded*.

The idea of PEA* is demonstrated in Figure 3. In the top part, when the node a is expanded, all of its children (x, y, z, w) are generated. However, only the children with the f-value of 3 (x and y) are currently needed. They are inserted into OPEN. All the other children are possibly surplus. They are collapsed back into a, who gets the new stored value $F(a) = 4$ (since $f(z) = 4$ is the least static value among the possibly surplus children of a). Following this, a is re-inserted into OPEN with $F(a) = 4$. There are now two cases.

- **No re-expansion**: Assume that a node g with $f(g) = 3$ is the goal. In this case, a will not be expanded again and the children of a with costs larger than 3 (z and w) will never be put into OPEN.

- **Re-expansion of *a***: When a is chosen for re-expansion with $F(a) = 4$ (Figure 3 (bottom)), all its children are generated. Only the node z with $f(z) = 4$ is currently needed. It is placed into OPEN. The possibly surplus node w is collapsed into a who gets the new stored value of $F(a) = 5$. The other currently unneeded nodes (x and y) are discarded. Since a has possibly surplus children, it is put back into OPEN.

The pseudo-code of PEA* is shown in Procedure 1. To clearly show the differences between the different variants of A*, we show A*, PEA* and EPEA* in the same pseudo-code. To make the pseudo-code easier to read, we assign a dummy stored value to the nodes in A*, although A* does not really ever use it. When PEA* generates a new node, it sets the new node's stored value to be equal to its static value (lines 2, 18).[3] Nodes are expanded in the order of the best stored value (line 5). When expanding n, PEA* generates *all* of its children (line 7) and inserts into OPEN only the children with $f(n_c) = F(n)$ (lines 11-18). These nodes are currently needed. The children with $f(n_c) > F(n)$ are possibly surplus. They are collapsed into n, whose stored value becomes the lowest static value among these children (line 14). The children with $f(n_c) < F(n)$ are discarded (line 15). If n has no possibly surplus children (i.e. all children of n have been inserted into OPEN), then n is put into CLOSED (line 20). Otherwise, n is re-inserted into OPEN with the new stored value (line 22).

3. Although the notion of stored value is defined only in the context of collapsing, we can view an yet unexpanded node as being a frontier node that has been collapsed into itself.

Procedure 1 A*, PEA* and EPEA*.

```
1:  Generate the start node $n_s$
2:  Compute $h(n_s)$ and set $F(n_s) \leftarrow f(n_s) \leftarrow h(n_s)$
3:  Put $n_s$ into OPEN
4:  while OPEN is not empty do
5:      Get $n$ with lowest $F(n)$ from OPEN
6:      if $n$ is goal then exit                                  // optimal solution is found!
7:      For A* and PEA*: set $N \leftarrow$ set of all children of $n$ and initialize $F_{next}(n) \leftarrow \infty$
8:      For EPEA*: set $(N, F_{next}(n)) \leftarrow OSF(n)$.
9:      for all $n_c \in N$ do
10:         Compute $h(n_c)$, set $g(n_c) \leftarrow g(n) + cost(n, n_c)$ and $f(n_c) \leftarrow g(n_c) + h(n_c)$
11:         For PEA*:
12:             if $f(n_c) \neq F(n)$ then
13:                 if $f(n_c) > F(n)$ then
14:                     Set $F_{next}(n) \leftarrow min(F_{next}(n), f(n_c))$
15:                 Discard $n_c$
16:                 continue                                      // To the next $n_c$
17:         Check for duplicates
18:         Set $F(n_c) \leftarrow f(n_c)$ and put $n_c$ into OPEN
19:     if $F_{next}(n) = \infty$ then
20:         Put $n$ into CLOSED                                   // For A*, this is always done
21:     else
22:         Set $F(n) \leftarrow F_{next}(n)$ and re-insert $n$ into OPEN        // Collapse
23: exit                                                          // There is no solution.
```

2.3 The Parameter C and Comparison of EPEA* with PEA*

PEA* saves memory by not putting surplus nodes into OPEN, but incurs a large time performance overhead, since, whenever it re-expands a node n, it generates *all* of the children nodes of n and computes their f-values, thus repeating the work that it did when it expanded n for the first time. In domains where children of n can assume a large number of different static values, this run-time overhead is large. To make PEA* practicable for these domains, the authors of PEA* introduced a C-parameter, which determines the trade-off between the amount of memory saved and the run-time overhead paid. When a node n is expanded, the children with $F(n) \leq f(n_c) \leq F(n) + C$ are added into OPEN (the change is made in line 12 of the above pseudo-code). When $C = 0$ maximal memory savings are obtained. This is the variant shown in the pseudo-code. When $C = \infty$, PEA* becomes equivalent to A*. The best choice of C is domain-, heuristic- and instance-dependent and no policy for selecting C has been reported.

EPEA* does not have this memory-time trade-off. It saves the same amount of memory as PEA* with $C = 0$, while saving run-time as well. Therefore, in our experiments, we always compare EPEA* to PEA* with $C = 0$.

We will show in Section 9 that, although PEA* is motivated by trying to make OPEN smaller, it sometimes has exactly the opposite effect.

3. Enhanced Partial Expansion A*

This section presents our new variant of A*, the *Enhanced Partial Expansion A** (EPEA*). We start with an important observation.

3.1 The Three Phases on the Way to OPEN

One of key observations used in EPEA* is that every node passes three phases from its inception to its entry into OPEN:

1. **Discovering and checking an operator**. The search algorithm **(a)** discovers an applicable operator and **(b)** checks whether this operator should be applied. In many commonly used domains, the list of applicable operators (step (a) above) is determined easily by looking at the current state. In planning domains, discovering an operator consists of checking the operator's preconditions. In either case, both A* and PEA* generate children nodes corresponding to *all* applicable operators. They do not perform step (b) above. *Checking an operator* refers to deciding whether to apply an available operator. Until now, this step has gone unnoticed. Distinguishing this step is a key idea of EPEA*.

2. **Applying an operator**. We will also refer to this phase as *generating a node*, since the search algorithm generates a child node by applying an operator to the node being expanded. In A*, generating a node is preceded by making a copy of the parent, while in IDA* this is not needed.

3. **Inserting a node into OPEN**. The new node becomes part of the frontier of nodes maintained by the search algorithm.

3.2 The Operator Selection Function (OSF)

The difference between EPEA* and PEA* can be summarized as follows. PEA* generates *all* children n_c of n with two objectives:

1. To put into OPEN the children with $f(n_c) = F(n)$ and

2. To update the stored value $F(n)$.

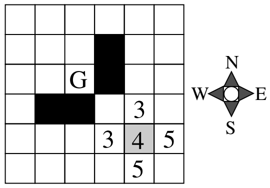
EPEA* achieves these objectives without generating all children. Instead, it employs a domain- and heuristic-specific *Operator Selection Function* (OSF) to both generate only the children n_c of n with $f(n_c) = F(n)$ and compute the next $F(n)$. In this section, we describe OSF. In the next section, we will present the EPEA* algorithm.

An OSF consists of two components:

1. The *Knowledge Component* – domain- and heuristic-specific knowledge.

2. The *Algorithmic Component* – an algorithm that uses the knowledge component to attain the stated above objectives of the OSF.

For an example of OSF, consider the single-agent pathfinding domain (SAPF) in Figure 4 (left). The agent is placed into a 4-connected grid (the shaded cell in the figure). The agent's arsenal of actions consists of 4 cardinal moves and a Wait move (where the agent stands still). Assume that



$\Delta f(n_c)$	$\Delta h(n_c)$	Operators	$\Delta F_{next}(n)$
0	-1	North, West	1
1	0	Wait	2
2	1	South, East	∞

Figure 4: The knowledge component of an OSF for the single-agent pathfinding with the Manhattan distance heuristic. The part of the knowledge component for the case of the goal located to the north-west of the current location is shown.

all moves cost 1 and the heuristic function is *Manhattan distance* (MD). In the figure, the numbers inside the cells are the corresponding h-values.

Consider the case when the goal is located to the north-west of the current location. The knowledge component of the OSF for this example is shown in Figure 4 (right) in the form of a table. This table uses the following convenient notation, which we will continue to use throughout the paper. Consider an expansion of n and the operator that produces the child n_c. We denote the change in heuristic value resulting from applying this operator by $\Delta h(n_c) = h(n_c) - h(n)$ and the change in the f-value by $\Delta f(n_c) = f(n_c) - f(n)$. The current difference between the stored and the static value of n is denoted by $\Delta F(n) = F(n) - f(n)$. The next stored value of n is denoted by $F_{next}(n)$, while the $\Delta F(n)$ that corresponds to that stored value is denoted by $\Delta F_{next}(n)$. The table groups the operators according to $\Delta f(n_c)$ and orders the groups according to the increasing order of this quantity. Note that, since the quantities $\Delta f(n_c)$ and $\Delta h(n_c)$ differ by $cost(n, n_c)$, which is a constant in this domain, ordering the operators by either $\Delta f(n_c)$ or $\Delta h(n_c)$ produces the same result.

When EPEA* expands a node n with the stored value $F(n)$ and needs to generate the children n_c with $f(n_c) = F(n)$, it would invoke the algorithmic component of this OSF to:

1. Find the row that correspond to $\Delta f(n_c) = F(n) - f(n)$,
2. Generate the currently needed children by applying the operators from that row, and
3. Return the next stored value of n: $F_{next}(n) = f(n) + \Delta F_{next}(n)$.

OSFs are domain and heuristic-dependent. Using EPEA* to solve problem instances of a particular domain with a particular heuristic requires the creation of an OSF for those specific domain and heuristic. We provide a classification of OSFs in Section 5. In particular, the minimum prerequisite for applying EPEA* is the existence of the *full-checking OSF* described in that section for a given domain and heuristic. The classification of OSFs will also serve as general guideline for OSF construction.

An OSF bears some resemblance to the concept of *preferred operators*, often used by domain independent planners such as FF (Hoffmann & Nebel, 2001) and Fast Downward (Helmert, 2006; Richter & Helmert, 2009).[4] Preferred operators are a subset of actions that are assumed to be more likely to lead to a goal. When expanding a node, FF generates nodes by using only the

4. In FF, preferred operators are called *helpful actions*.

preferred operators while ignoring all other actions. This often reduces search time at the cost of the loss of completeness. Fast Downward uses preferred operators in a more conservative manner that preserves completeness. Nodes generated by the preferred operators are prioritized so as to be expanded more often. This is done by maintaining an additional OPEN list containing only nodes generated by the preferred operators. Fast Downward alternates between expanding a node from the additional OPEN list and the regular OPEN list containing all the generated nodes.

Preferred operators impose a binary partition over the set of operators: an operator is either preferred or not. By contrast, an OSF provides a finer grained operator partitioning, grouping operators according to their Δf values. Moreover, neither FF nor Fast Downward is able to completely avoid generating surplus nodes by using preferred operators, while the EPEA* algorithm described below does exactly that by using an OSF.

Note that Fast Downward uses preferred operators in conjunction with another technique called *deferred heuristic evaluation* (Helmert, 2006; Richter & Helmert, 2009). Deferred heuristic evaluation saves heuristic computation time by inserting generated nodes into OPEN with the f-value of their parent. The heuristic of a node is computed only when the node reaches the top of the OPEN list. Thus, deferred heuristic evaluation trades memory and the time spent on the OPEN list operations for saving on the heuristic computation time. One can view deferred heuristic evaluation as the opposite of the collapse action (Section 2.1): a collapse saves memory and the time needed by the OPEN list operations.

We are now ready to describe the EPEA* algorithm.

3.3 Definition of EPEA*

The flow of EPEA* (Procedure 1) is identical to that of PEA*: (1) it puts nodes into OPEN and expands them in the same order as PEA* and (2) it collapses the possibly surplus children nodes. Therefore, the pseudo-code of EPEA* is similar to that of PEA*. The few differences stem from the fact that EPEA* uses a domain- and heuristic-specific *Operator Selection Function* (OSF):

1. Instead of generating *all* children nodes and discarding the currently unneeded ones (PEA*, lines 7 and 15), EPEA* uses the OSF to generate only the currently needed children nodes to start with (line 8).

2. Instead of looking at *all* children of n to compute its next stored value (PEA*, line 14), EPEA* receives that value from the OSF (line 8).

We have already seen the operation of PEA* for the example in Figure 3. Consider the operation of EPEA* for the same example. In Figure 3 (top), EPEA* uses an OSF to generate only the currently needed children x and y. It does not generate the nodes z and w, since they are possibly surplus. This OSF also determines the next stored value of a ($F(a) = 4$) – the lowest cost among the nodes z and w. In Figure 3 (bottom), the children x, y and z are provably useful. However, only z is currently needed and is generated by the OSF, since the costs of x and y (3) are lower than $F(a)$ (4). It is important to clearly see the distinction between PEA* and EPEA*: the nodes generated and discarded by PEA* are the same nodes that EPEA* does not generate.

For an example with a concrete OSF, consider the operation of EPEA* for the SAPF example in Figure 4 (left) with the OSF shown in Figure 4 (right) and described in Section 3.2. Suppose that n (with the agent in the shaded cell) is expanded for the first time with $f(n) = F(n) = 10$. EPEA* uses the OSF to get the children with $f(n_c) = F(n) = 10$. That is, we are interested in the children with $\Delta f(n_c) = 0$. The OSF uses the information in Figure 4 (right) to produce the children that correspond to the operators North and West. The children with $f(n_c) > F(n)$ are collapsed into n, which gets the stored value determined by Δf-value of the operator in the next row. In this case, the next row contains the operator Wait with $\Delta f = 1$, so the next stored value of n will be $F_{next}(n) = f(n) + 1 = 11$. For convenience, the next value of $\Delta F(n)$, $\Delta F_{next}(n) = F_{next}(n) - f(n)$, is shown in the rightmost column of the table.

If n is re-expanded with $f(n) = 10$ and $F(n) = 11$, the OSF returns the nodes n_c with $f(n_c) = F(n) = 11$ or, equivalently, $\Delta f(n_c) = 1$. This corresponds to applying the Wait operator. The remaining children with $f(n_c) > F(n)$ will be collapsed into n, which will get the stored value of $f(n) + \Delta F_{next}(n) = 10 + 2 = 12$.

Note that the table in Figure 4 (right) is only for expanding n located to the South-East of the goal. The complete knowledge component of this OSF for this domain contains seven more tables for the other possible locations of n relative to the goal.

3.4 A High-Level Comparison of A*, PEA* and EPEA*

In this section, we compare A*, PEA* and EPEA* with respect to their memory and run-time performance.

3.4.1 Memory Performance

A* puts into OPEN every node that it generates and that has not been generated before with the same or lower cost. PEA* improves on the memory performance of A* by not putting into OPEN the currently unneeded nodes. The actual memory saving is determined by the parameter C (see Section 2.2), with the greatest saving achieved by setting $C = 0$. EPEA* puts into OPEN the same nodes as PEA* with $C = 0$. Therefore, EPEA* affords the same memory savings as the most memory-effective variant of PEA*.

3.4.2 Run-Time Performance

First, compare the *node* performance of the three algorithms:

- A* is known to be optimal with respect to the number of expanded nodes (Dechter & Pearl, 1985). It does not give any guarantees with respect to the number of generated nodes.

- Consider the first expansion of each node expanded by PEA*. These are the same expansions that A* performs. Also, PEA* performs these expansions in the same order as A*. Therefore, PEA* is optimal with respect to the number of *unique* node expansions.[5] However, PEA* may have to re-expand the same node many times. During *each* re-expansion, PEA* generates *all* of the node's children. Therefore, PEA* may perform many more node generations than A*.

- EPEA* expands the same nodes as PEA* in the same order. Therefore, just like PEA*, EPEA* is optimal with respect to the number of *unique* node expansions. Unlike PEA*, EPEA* does not generate the currently unneeded nodes. Therefore, EPEA* may skip generating some states that A* would generate. In fact, optimality results with respect to the

5. Although the number of unique node expansions is not of practical interest, it is interesting as a means of understanding the algorithm and its performance. Therefore, we measure the number of unique node expansions in some of our experiments.

Variant	Which nodes are generated	Which nodes are put into OPEN
A*	All	All
PEA*	All	Currently needed
EPEA*, full-checking OSF	Currently needed, obtained by checking all operators	Currently needed
EPEA*, direct-computation OSF	Currently needed, obtained by direct computation	Currently needed

Table 1: High-level comparison of A*, PEA* and EPEA*.

number of generated nodes were recently proven about variants of EPEA* (Goldenberg, Felner, Stutervant, Holte, & Schaeffer, 2013).

Now consider the run-time performance of the three algorithms:

- Depending on the domain and the heuristic, A* may spend much time generating surplus nodes. Also, A* may have to pay the penalty related to its usage of large amounts of memory. In particular, the OPEN list operations become expensive when the size of the search frontier grows.

- In most cases (see Section 9), PEA* uses a smaller amount of memory than A*, but pays the overhead of possible re-expansions of a single node and generating all children of a node for each re-expansion.

- Similar to PEA*, EPEA* has to pay the price of many possible re-expansions of a single node. However, EPEA* avoids the two run-time overheads that PEA* suffers from: generating the same useful node many times and generating possibly surplus nodes. The significance of these savings depends on the size of the states of the domain and the amount of computation that needs to be performed to apply an operator.

  Depending on the kind of OSF being used (the different kinds of OSFs will be considered in Section 5), EPEA* may be able to avoid not only generating the currently unneeded nodes, but also checking the operators that result in these nodes (i.e. the currently unneeded operators). The differences between A*, PEA* and EPEA* with respect to the phases described in Section 3.1 are summarized in Table 1. We study the time performance of EPEA* more deeply in Section 10.

4. Enhanced Partial Expansion IDA* (EPEIDA*)

We begin by noting that IDA* can be viewed as having partial expansion built into the algorithm. Namely, consider the current iteration's threshold as the stored value of all nodes during this iteration. When IDA* completes an iteration, it collapses all of the frontier nodes of the current iteration into the root node, which gets an updated stored value – the next iteration's threshold. Suppose that the current iteration's threshold is T. Once n is expanded, all its children are generated. At the next level of the depth-first search, the children n_c of n with $f(n_c) \leq T$ are expanded, while the children with $f(n_c) > T$ are discarded. This is partial expansion with the following one difference.

In PEA*, the currently needed children are the children with $f(n_c) = F(n)$. The children with $f(n_c) < F(n)$ do not need to be put into OPEN, since such children were put into OPEN during the previous expansions of n. IDA* does not store information from the previous iterations and therefore it needs to search the children with $f(n_c) < T$ as well. Therefore, in the context of IDA*, we need to re-define the notion of currently needed children of n to include all children n_c with $f(n_c) \leq T$.

We now describe EPEIDA*. To clearly see the distinctions between IDA* and EPEIDA*, both variants are shown in the same pseudo-code (Procedure 2). The first difference is that EPEIDA* uses an OSF to obtain both the list of currently needed children (those with $f(n_c) \leq T$) and the lowest cost among the currently unneeded children (line 10). The latter cost is used to update the threshold T_{next} for the next iteration (line 11). This threshold is initialized to infinity at the beginning of the current iteration (not shown in the pseudo-code). In contrast, IDA* generates all children of n and updates the value of the threshold for the next iteration by iterating through the list of these children (IDA*, lines 5, 8). The second difference is that EPEIDA* does not need to check the threshold condition (IDA*, line 4). This is because the threshold condition is equivalent to the condition in our definition of a currently needed node for IDA*. Since OSF generates only the currently needed children, no additional check is needed.

The number of nodes that EPEIDA* generates is approximately b times smaller than the number of nodes generated by IDA*, where b is the average branching factor of the domain. To verify this, let X be the number of nodes expanded by the last iteration of IDA*. The number of nodes generated by this iteration can be approximated by bX. On the other hand, EPEIDA* generates up to $X + (b-1)d$ nodes (more details are given in Section 10.1), where d is the depth of the search. Since $(b-1)d$ is usually very small compared to X, the ratio between bX and $X + (b-1)d$ is approximately equal to b. We will repeatedly point to this fact in the discussion of our experimental results in Sections 6-7.

The experimental results will compare the performance of EPEIDA* and the performance of IDA* for several domains and heuristics. When studying those results, it is important to have in mind that IDA* can be viewed as having partial expansion built into the algorithm. Therefore, when we compare EPEIDA* with IDA*, we are really comparing EPEIDA* with the iterative deepening version of PEA*.

5. Classification of OSFs

Recall the high-level comparison of A*, PEA* and EPEA* in Table 1. In that table, a principle distinctions between EPEA* and PEA* is shown: EPEA* applies only the currently needed operators to generate only the currently needed children, while PEA* generates all children of the node n being expanded. Furthermore, Table 1 shows that two possibilities exist for obtaining the list of currently needed operators. These possibilities correspond to two classes of OSFs:

1. A *full-checking OSF* obtains the list of currently needed operators by checking all operators, including the currently unneeded ones.

2. A *direct-computation OSF* obtains the list of currently needed operators by means of direct computation.

These two kinds of OSF are the focus of this section. In addition to these two *pure* classes, we will encounter a *hybrid OSF*. This will happen in domains and heuristics for which we are

Procedure 2 IDA* and EPEIDA*.
Variables:
i – current depth in the search tree
T – threshold of the current iteration
T_{next} – threshold for the next iteration, set to infinity before calling DFS().

DFS(n, i, T)
  Compute $h(n)$ and set $f(n) \leftarrow g(n) + h(n)$
  For **IDA***:
    if $f(n) > T$ **then**
      Set $T_{next} = \min(T_{next}, f(n))$
      return // Threshold cut-off
  if n is goal **then halt** // The optimal solution is found!
  For **IDA***: Set $N \leftarrow$ set of all children of n
  For **EPEIDA***:
    Set $N, fBestNotNeeded \leftarrow OSF(n)$
    Set $T_{next} = \min(T_{next}, fBestNotNeeded)$
  for all $n_c \in N$ **do**
    Set $g(n_c) \leftarrow g(n) + cost(n, n_c)$
    Call $DFS(n_c, i + 1, T)$

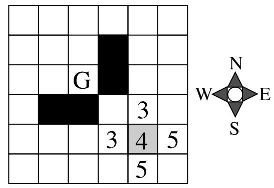
able to construct a direct-computation OSF to generate only the currently needed nodes for certain values of $\Delta f(n_c)$, but not for the other values of $\Delta f(n_c)$. For these domains, we construct a hybrid OSF, which can behave either as a direct-computation or as a full-checking OSF for different (re-)expansions of n depending on the required $\Delta f(n_c)$.

We will show below (Sections 6-8) how an OSF can be constructed for some well known domains and heuristics. For now, we explain the two pure classes of OSFs using the SAPF domain as a running example.

5.1 Full-Checking OSFs

The knowledge component of the full-checking OSF for SAPF consists of eight tables. Each table corresponds to one of the eight possible positions of n relative to the goal. The algorithmic component of this OSF determines which of these tables must be used for a given n based on the position of n relative to the goal. For example, the table for the case of n located South-East of the goal (e.g. the shaded cell in Figure 5 (left)) is shown in Figure 5 (right). From now on, we use the table in Figure 5 (right) as the running example.

For each operator applicable to n, the table records the resulting changes in the h- and f-values. The OSF's algorithmic component needs to check each of the operators to decide which children nodes are currently needed. It then generates only these children. Also, it computes the next stored value of n, $F_{next}(n)$, as follows. $F_{next}(n)$ is initialized to infinity. Whenever the OSF checks an operator that results in n_c and determines that n_c is possibly surplus (i.e. $f(n_c) > F(n)$), it updates $F_{next}(n)$ to be $F_{next}(n) = \min(F_{next}(n), f(n_c))$.



Operator	$\Delta h(n_c)$	$\Delta f(n_c)$
Wait	0	1
East	1	2
South	1	2
West	-1	0
North	-1	0

Figure 5: The knowledge component of a full-checking OSF for the single-agent pathfinding with the Manhattan distance heuristic. The part of the knowledge component for the case of the goal located to the north-west of the current location is shown.

5.2 Direct-Computation OSFs

A direct-computation OSF computes the list of the currently needed operators directly, without the need to check all the applicable operators. In the case of SAPF, the knowledge component of a direct-computation OSF can be obtained from the described above full-checking OSF by ordering the rows of each of the eight tables in the order of increasing $\Delta f(n_c)$ and merging the rows corresponding to the same $\Delta f(n_c)$. For the table in Figure 5 (right), the corresponding knowledge component of the direct-computation OSF is shown in Figure 4 (right). Given the required $\Delta f(n_c)$, the row with the currently needed operators can be found by the algorithmic component quickly without checking all operators.[6] EPEA* will then apply these operators to generate the corresponding currently needed children nodes.

In general, a direct-computation OSF can be constructed when the states of the domain can be classified into several classes, such that, for each class C, one of the following holds:

- A table with operators applicable to states in C ordered by $\Delta f(n_c)$ can be computed and stored before the main search begins. This is the option that we used in the above OSF for SAPF. In that case, each class corresponded to one of the eight possible locations of n with respect to the goal.

- The set of operators with a given $\Delta f(n_c)$ can be computed on-the-fly during the search. This is the option that we will use in Section 6.2 for the pancake puzzle with the GAP heuristic (Helmert, 2010).

Even if one or both of the above conditions are satisfied, it may be impossible or impractical to construct a direct-computation OSF for a given domain and heuristic for one or more of the following reasons:

1. To construct the tables of operators ordered by $\Delta f(n_c)$, a large number of state classes must be defined, resulting in large time and memory requirements to pre-compute and store these tables.

2. To determine which class a given state belongs to, all operators applicable to that state need to be checked.

6. In a domain with a large number of $\Delta f(n_c)$-values, binary search or a hashing mechanism can be used.

Both these reasons will be clarified in Section 6.1, where we will explain why we cannot construct a direct-computation OSF for the 15-Puzzle with the Manhattan distance heuristic.

Note that even in the case when a direct-computation OSF is not available for a given domain and heuristic, we can still construct a direct-computation OSF for each given search node. Namely, when n is expanded for the first time, we can compute a table of operators applicable to n ordered by $\Delta f(n_c)$ and store this table together with n in OPEN. During the subsequent re-expansions of n, this table can be used as the knowledge component of a direct-computation OSF. The decision of whether to construct a direct-computation OSF for individual search nodes depends on the following time-memory trade-off:

- Using this technique eliminates the need to check all operators at each re-expansion of n. The amount of saved execution time depends on how many times n is re-expanded, which in turn depends on how many different f-values are taken by the children of n.
- Storing a table of applicable operators with every node in OPEN may be prohibitively expensive (memory-wise) for the domains with high average branching factor.

6. Experimental Study of the Different Kinds of OSFs

In this and the two following sections, we construct an OSF and show experimental results for several domains and heuristics. We start with relatively simple OSFs in this section and move towards the more complicated PDBs-based OSFs in Section 7. We then move to the yet more complicated additive PDBs-based OSFs in Section 8. Each description of an OSF is immediately followed by an experimental study of that OSF.

We start with EPEIDA* for two domains and heuristics:

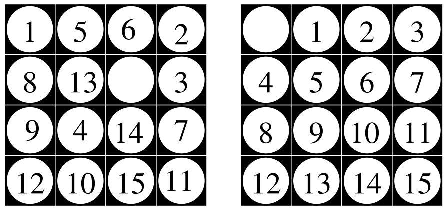
- **The 15-puzzle with the Manhattan distance heuristic.** For this domain, we construct a full-checking OSF and explain why constructing a direct-computation OSF is impractical.
- **The pancake puzzle with the GAP heuristic.** For this domain, we construct a hybrid OSF. We will see that, in practice, this OSF behaves as a direct-computation OSF for most expansions.

One can easily come up with interesting domains and heuristics, for which a *pure* direct-computation OSF can be constructed. However, we decided to focus on well known benchmark domains. We will be able to show direct-computation OSFs for such domains in the context of OSFs based on pattern databases (Culberson & Schaeffer, 1998; Felner, Korf, & Hanan, 2004) in Sections 7 and 8.[7]

6.1 A Full-Checking OSF for the 15-puzzle

The 15-puzzle consists of a 4×4 square frame containing 15 numbered square *tiles*, and an empty position called the *blank*. The legal operators are to slide any tile that is horizontally or vertically adjacent to the blank into the blank position. The problem is to rearrange the tiles from some random initial configuration, e.g. Figure 6 (left), into the configuration in Figure 6 (middle). *Manhattan distance* (MD) is the classic heuristic function for this puzzle. It is computed by counting the

7. We saw above a direct-computation OSF for SAPF. However, searches in the state space of SAPF feature a small number of surplus nodes, which makes it an uninteresting application of EPEA*.



Operator	$\Delta h(n_c)$	$\Delta f(n_c)$
13 moves East	1	2
6 moves South	-1	0
3 moves West	1	2
14 moves North	1	2

Figure 6: Left: a possible state of the 15-Puzzle. Middle: the goal state of the 15-Puzzle. Right: part of the knowledge component of an OSF for the 15-Puzzle with the Manhattan distance heuristic for the case when the set of applicable operators is as in the state on the left.

number of grid units that each tile is displaced from its goal position, and summing these values over all tiles.

6.1.1 The OSF

In the 15-puzzle, the quantity $\Delta f(n_c)$ of each operator applicable to n is completely determined by:

1. The location of the blank,
2. The identity of the tile being moved by the operator, and
3. The position of that tile relative to the blank.

Therefore, we can construct the knowledge component of a full-checking OSF for the 15-puzzle as a three-dimensional array with dimensions $16 \times 15 \times 4$. These dimensions stand for the 16 possible locations of the blank, 15 possible identities of the tile being moved and 4 possible positions of the tile being moved relative to the blank. Each element in this array is the $\Delta f(n_c)$ of the corresponding operator.

Example. For the operators applicable to the node n in Figure 6 (left), the knowledge component of the OSF would store the $\Delta f(n_c)$-values shown in Figure 6 (right). For example, for the operator that moves 13 into the blank position, the $\Delta f(n_c)$-value (2) is stored in the element [6][13][0], where 6 denotes the position of the blank, 13 is the identity of the tile being moved and 0 denotes the operator that moves the tile to the West of the blank into the blank position. For the same node n, the algorithmic component of this OSF would **(1)** check the operators in Figure 6 (right) by looking up the corresponding entries in the three-dimensional array (i.e. the knowledge component), **(2)** generate the currently needed nodes by applying the operators with $\Delta f(n_c) = F(n) - f(n)$, and **(3)** compute the next stored value of n using the minimal $\Delta f(n_c)$ among the possibly surplus children: $F_{next}(n) = f(n) + \Delta f(n_c)$.

To construct a direct-computation OSF for this domain, we need to classify states such that two states s_1 and s_2 are in the same class if and only if **(1)** the location of blank is same in s_1 and s_2 and **(2)** the identities of tiles surrounding the blank and their positions relative to the blank are same in s_1 and s_2. Therefore, we need to define $16 \times (15 \times 14 \times 13 \times 12)$ classes. Pre-computing and storing a table of operators for each of these classes is a computation and memory overhead. Furthermore, to decide which class a given state belongs to, the OSF will need to look at the identities of the tiles

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

Table 2: Comparison of generated nodes and time performance of IDA* and EPEIDA* for the 15-Puzzle.

surrounding the blanks, which is equivalent to checking all operators. We conclude that constructing a direct-computation OSF for the 15-puzzle domain with the MD heuristic is impractical. The provided below experimental results were obtained with the full-checking OSF.

6.1.2 EXPERIMENTAL RESULTS

Optimal solutions to random instances of the 15-puzzle were first found by Korf (1985) using IDA* and the MD heuristic. Korf has graciously made this code available to the public. In this code (known to be highly optimized), a look-up table is pre-computed to give the heuristic value based on the current location of the tile, the operator chosen and the tile currently occupying the proposed new location of the blank. Note that this information is exactly the knowledge component of the full-checking OSF described above. However, Korf's code did not exploit this information to avoid generating the currently unneeded nodes. Instead, it generates all the children nodes and uses the look-up tables only for the heuristic calculation.

Table 2 presents the results of running Korf's IDA* code and EPEIDA* on his 100 random instances. We observe a factor of 1.97 reduction in the number of generated nodes. This number is close to the asymptotic branching factor of this domain when only the reverse moves are eliminated, which was reported to be 2.13 (Edelkamp & Korf, 1998). This reduction in the number of generated nodes translates to a 1.25-fold improvement in run-time, which is significant given the well known efficiency of Korf's code. The timing results were obtained on Dell Optiplex 760.

6.2 A Hybrid OSF for the Pancake Puzzle

The *pancake puzzle* (Dweighter, 1975) is analogous to a waiter navigating a busy restaurant with a stack of N pancakes. The waiter wants to sort the pancakes ordered by size. Having only one free hand, the only available operation is to lift a top portion of the stack and reverse it. 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 $(4,3,2,1)$. However, we may safely not consider two successors – the parent position and the complete reverse of the pancakes (the latter can be ignored for most search nodes when the GAP heuristic described below is used). Therefore, the branching factor of this domain is approximately equal to the number of pancakes minus two. Since all states are reachable from any start state, the size of the state space is $N!$. A number of heuristics based on pattern databases have been used for this puzzle (Zahavi, Felner, Holte, & Schaeffer, 2008; Felner, Zahavi, Holte, Schaeffer, Sturtevant, & Zhang, 2011; Yang, Culberson, Holte, Zahavi, & Felner, 2008), but the GAP heuristic discussed by Helmert significantly outperforms them all (Helmert, 2010).

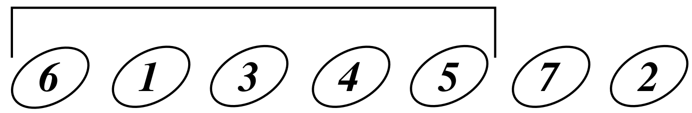


Figure 7: A state of the Pancake Puzzle with seven pancakes. An operator can affect the gap at one location only.

We now describe the GAP heuristic. Two integers a and b are *consecutive* if $|a - b| = 1$. For a given state, a *gap* at location j occurs when the pancakes at location j and $j+1$ are not consecutive. The goal can be defined such that pancake 1 is at location 1 and there are no gaps. The GAP heuristic iterates through the state and counts the number of gaps. Since an operator can reduce the number of gaps by at most one, this heuristic is admissible.

To gather the insight necessary to construct an OSF, consider the example in Figure 7. There are seven pancakes, with pancake 6 occupying location 1. Consider the operator that reverses the first five pancakes. We note that this operator does not affect the number of gaps between the pancakes being reversed (i.e. the gaps at locations 1-5 between the pancakes 6, 1, 3, 4, 5). Indeed, only the gaps adjacent to the extreme pancakes being reversed (i.e. pancakes 6 and 5) are affected. In our case, the consecutive pancakes 6 and 7 become adjacent, while the gap between pancakes 5 and 7 ceases to exist. Thus, just by looking at three pancakes (in this case, 5, 6, and 7), we know that the number of gaps (and hence the GAP heuristic) has been decreased by one and $\Delta f(n_c) = 0$.

More generally, an operator that reverses j pancakes affects only the gaps formed by the three pancakes at locations 1 (pancake P), j (pancake X), and $j+1$ (pancake Y). Three cases are possible:

1. X and Y were not consecutive but P and Y are. In this case, one gap is removed and the heuristic value decreases by one. This corresponds to $\Delta f(n_c) = 0$.

2. Both the pancakes X, Y and the pancakes P, Y form or do not form a gap. In this case the heuristic value does not change. This corresponds to $\Delta f(n_c) = 1$.

3. The pancakes X, Y do not form a gap, while the pancakes P, Y do. In this case, a new gap is introduced and the heuristic grows by one. This corresponds to $\Delta f(n_c) = 2$.

Suppose that we need to generate children of n with $\Delta f(n_c) = 0$. We note that this can be done without checking the currently unneeded operators (i.e. operators with $\Delta f(n_c) > 0$). To do this, we classify the states based on the identity of the pancake in location 1. The state shown in Figure 7 belongs to the class with the pancake 6 in location 1. We enable the OSF to compute operators with $\Delta f(n_c) = 0$ for each class on-the-fly by maintaining the following additional information for *each* node of the main search. For each pancake, we keep the current location of the pancakes that are consecutive to it (note that there are at most two such pancakes; in the example, the pancakes consecutive to the pancake 6 are 5 and 7 in locations 5 and 6, respectively). Now, suppose that n has pancake P in location 1. Then, all we have to do is to use the information stored with P to locate pancakes $P-1$ and $P+1$ and check the pancakes directly to the left of these pancakes. For example, if $P+1$'s left neighbor is any pancake other than $P+2$, then reversing the pancakes to

	Generated Nodes				Time (ms)		
#	IDA*	EPEIDA*		ratio	IDA*	EPEIDA*	ratio
		Total	Full-checking OSF				
20	18,592	1,042	12	17.84	1.5	0.1	11.23
30	241,947	8,655	25	27.95	24.9	1.2	20.00
40	1,928,771	50,777	49	37.98	247	8.5	30.75
50	13,671,072	284,838	131	47.99	2,058	57	36.15
60	92,816,534	1,600,315	121	57.99	16,268	359	45.32
70	754,845,658	11,101,091	1,118	67.99	155,037	2,821	54.90

Table 3: Comparison of generated nodes and time performance of IDA* and EPEIDA* for the Pancake Puzzle.

the left of $P + 1$ will decrease the GAP heuristic and result in $\Delta f(n_c) = 0$. Thus, we can construct a direct-computation OSF for the case of $\Delta f(n_c) = 0$.

We cannot construct a direct-computation OSF for all cases of $\Delta f(n_c)$, since we have to check all operators to find the operators resulting in $\Delta f(n_c) = 1$ or $\Delta f(n_c) = 2$. For these cases, a full-checking OSF is constructed. This OSF has no knowledge component. The algorithmic component simply checks all the operators applicable to n and, for each operator, computes the resulting $\Delta f(n_c)$ by looking at the three affected pancakes as described above. If the operator is currently needed, then the corresponding child node is generated.

The experimental results for 100 random instances for 20 to 70 pancakes are given in Table 3. In this table, we compare the performance of EPEIDA* and IDA*, both using the GAP heuristic. For 70 pancakes, EPEIDA* generated (the column titled "Total") 68 times fewer nodes than IDA*. Most of this is reflected in the running time (54-fold). To the best of our knowledge, these are the state-of-the-art results for this puzzle. Note that for all versions tested of this domain, the reduction in the number of generated nodes is almost the same as the branching factor of the domain.

To better understand the behavior of the hybrid OSF for this domain, compare the numbers in the two columns for the EPEIDA* generated nodes. The column on the left shows the total number of generated nodes, while the column on the right shows how many nodes were generated by using a full-checking OSF. The comparison between these two columns reveals that, for the vast majority of expansions, only the operators with $\Delta f(n_c) = 0$ were currently needed, whereby EPEIDA* applied the direct-computation OSF.

7. OSFs Based on Pattern Databases

Pattern Databases (PDBs) (Culberson & Schaeffer, 1998; Felner et al., 2004) are a powerful method for automatically building admissible memory-based heuristics based on domain abstractions. After a short background section, we explain how PDB-based full-checking and direct-computation OSFs can be constructed and provide experimental results for the Rubik's cube domain with the Corners-PDB heuristic.

7.1 Background

View a state of a domain as an assignment of values to a number of *domain variables* (hereafter *variables*). The main idea of PDBs is to abstract the state space by only considering a subset of the variables. For a concrete choice of variables, this abstraction is formalized as an *abstraction mapping*, which we denote by ϕ and immediately define. For each state s of the original state, the *abstract state* (or *pattern*) $\phi(s)$ is the *projection* of s onto the variables that participate in ϕ (or, for short, the projection of s onto ϕ).

A PDB for a given ϕ is constructed by performing a full breadth-first search in the abstract state space from the abstract goal, i.e. $\phi(g)$, where g is the goal state. Distances to all abstract states 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. Formally, for each state s, the PDB contains the distance in the abstract space from $\phi(s)$ to $\phi(g)$. When a heuristic value is required for n, one simply looks up PDB[$\phi(n)$].

An interesting variation of PDBs are the instance-dependent pattern databases (IDPDBs) (Felner & Adler, 2005; Zhou & Hansen, 2004; Silver, 2005). IDPDBs can be 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 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. Hierarchical search algorithms, such as Hierarchical A* (Holte, Perez, Zimmer, & MacDonald, 1996) and Switchback (Larsen, Burns, Ruml, & Holte, 2010), use a hierarchy of abstract spaces to create a hierarchy of PDBs, also created lazily in a similar manner. We will use IDPDBs in our experiments in Section 8.4.

We now present a method for constructing a PDB-based OSF.

7.2 PDB-Based OSF

Let ϕ be an abstraction mapping for a given domain. Note that each operator in the original space has a *projection* onto ϕ – an *abstract operator* that modifies the variables of $\phi(s)$ in the same way that the original operator modifies these variables in s.

For a given ϕ, we can construct either a full-checking or a direct-computation OSF. Intuitively, the knowledge components of these OSFs are similar to the tables that we constructed for SAPF in Sections 5.1 and 5.2. Recall that, for SAPF, the knowledge component of both OSFs consisted of eight tables. Each table corresponded to one of the eight possible positions of n relative to the goal. Each table recorded the operators applicable to n and the resulting $\Delta f(n_c)$. Depending on the kind of OSF, the tables were sorted by either the operators or the $\Delta f(n_c)$-values. We adopt this method to construct an OSF based on ϕ as follows. The knowledge component of the OSF contains:

- A table for each abstract state a. This table records the abstract operators applicable to a as well as the resulting change in the f-value. We will denote an abstract operator by σ, the result of applying σ to a by $a_c(=\sigma(a))$ and the resulting change in the f-value by $\Delta f(a_c)$.

- Depending on whether we are building a full-checking or a direct-computation OSF, each table is sorted either by abstract operators or by $\Delta f(a_c)$.

We call the data structure employed by this knowledge component a Δ-*PDB* and distinguish a Δ-*PDB sorted by operators* (used as the knowledge component of the full-checking OSF) and a Δ-*PDB sorted by* $\Delta f(a_c)$ (used as the knowledge component of the direct-computation OSF).

It is important to note that:

1. A Δ-PDB does not include the regular PDB that it is based on. Rather, Δ-PDBs are additional data structures that enable a PDB-based OSF. In some domains, there is a trade-off between **(1)** building a Δ-PDB, which affords using EPEA* and **(2)** building an additional PDB, which affords a more accurate heuristic. We do not explore this trade-off.

2. The entry of a Δ-PDB is different than the entry of a regular PDB. An entry of a regular PDB contains one value – the distance from a given abstract state to the abstract goal. An entry of a Δ-PDB contains a table of abstract operators and $\Delta f(a_c)$-values for a given abstract state.

The algorithmic component of the full-checking PDB-based OSF operates as follows. For each operator o applicable to n, the OSF **(1)** locates the table for $\phi(n)$ in the Δ-PDB ordered by operators and **(2)** looks up in this table the $\Delta f(a_c)$-value of the projection of o onto ϕ. If this value is the required $\Delta f(n_c)$, n_c is included in the set of currently needed children.

The algorithmic component of the direct-computation OSF operates as follows. For a given n and required $\Delta f(n_c)$, the OSF **(1)** locates the table for $\phi(n)$ in the Δ-PDB ordered by $\Delta f(a_c)$-values and **(2)** looks up in this table the abstract operators with the $\Delta f(a_c)$-value equal to the required $\Delta f(n_c)$. For each such abstract operator, the OSF determines the set of operators in the original space that correspond to these abstract operators. Thus, a direct-computation OSF can be constructed only if every abstract operator can be efficiently mapped to the corresponding operator(s) in the original space.

It should be clear that, unless the branching factor of the abstract space is large, the improvement in the run-time performance afforded by using the direct-computation OSF rather than the full-checking OSF cannot be large. This is because the algorithmic components of both kinds of OSF need to perform only one lookup in the Δ-PDBs. The only difference is that the full-checking OSF needs to scan through the array of $\Delta f(a_c)$-values corresponding to all abstract operators, which is usually not a large overhead.

In the next section, we consider an OSF for Rubik's Cube with the Corner PDBs heuristic as a simple example of a PDB-based OSF. A more involved example – an OSF based on additive PDBs (Felner et al., 2004; Yang et al., 2008) for Multi-Agent Pathfinding domain (MAPF) (Standley, 2010; Sharon, Stern, Goldenberg, & Felner, 2011, 2012) will be given in Section 8.

7.3 An OSF for Rubik's Cube

Rubik's Cube was invented in 1974 by Erno Rubik of Hungary. The standard version consists of a $3 \times 3 \times 3$ cube with different colored stickers on each of the exposed squares of the sub-cubes, or cubies. There are 20 movable cubies and 6 stable cubies in the center of each face. The movable cubies can be divided into eight corner cubies, with three faces each, and twelve edge cubies, with two faces each. Corner cubies can only move among corner positions, and edge cubies can only move among edge positions.

Each one of the 6 faces of the cube can be rotated 90, 180, or 270 degrees relative to the rest of the cube. This results in 18 possible moves for each state. Since twisting the same face twice in a row is redundant, the branching factor after the first move can be reduced to 15. In addition, movements of opposite faces are independent. For example, twisting the left face and then the right face leads to the same state as performing the same moves in the opposite order. Pruning redundant moves results in a search tree with an asymptotic branching factor of about 13.34847 (Korf, 1997).

#	IDA*	EPEIDA*	ratio	IDA*	EPEIDA*	ratio
Corner PDB						
	Generated Nodes - Thousands			Time (mm:ss)		
12	45,800	3,441	13.31	0:05	0.01	4
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

Table 4: Comparison of generated nodes and time performance of IDA* and EPEIDA* for the Rubik's Cube.

In the goal state, all the squares on each side of the cube are the same color. The puzzle is scrambled by making a number of random moves, and the task is to restore the cube to its original unscrambled state. There are about 4×10^{19} different reachable states.

A classic abstraction mapping for Rubik's cube is the PDB based on the corner cubies (Korf, 1997). This abstraction has $88,179,840$ abstract states. The branching factor of the abstract space is same as the branching factor of the original space. In fact, each operator in the abstract space is trivially one-to-one-mapped to an operator in the original space.

We experimented with both the full-checking and the direct-computation OSFs. As expected, the branching factor of 18 was not large enough to achieve a run-time advantage by using a direct-computation OSF. In fact, the full-checking OSF was marginally faster in our experiments.

The results for the full-checking OSF are given in Table 4. Each line is the average over 100 instances of depth 12-15. The reduction (*ratio* column) in the number of nodes generated is a factor of 13.3 (again very close to the known effective branching factor), while the time improvement is only 3.7-fold. The reason for the discrepancy is that the constant time per node of EPEIDA* is larger than that of IDA* since it includes the time to retrieve values from the Δ-PDB.

8. Additive PDBs-Based OSF

This section is motivated by the Multi-Agent Pathfinding domain (MAPF), which has recently attracted significant attention of researchers (Standley, 2010; Sharon et al., 2011, 2012). During the recent workshop dedicated to this problem, a way to use *additive PDBs* (Felner et al., 2004; Yang et al., 2008) to solve instances of this problem was presented (Goldenberg, Felner, Stern, & Schaeffer, 2012). We develop an additive PDBs-based OSF, so that generation of surplus nodes can be avoided when solving instances of MAPF.

To keep this section as simple as possible, we leave the description of technically complicated background and details of implementation of OSF to Appendices C and D. The current section is organized as follows:

1. A brief description of MAPF (Section 8.1). Standley (2010) introduced two MAPF-specific algorithmic enhancements to A* that make problem instances of MAPF solvable within reasonable time resources. We mention these enhancements and describe them in Appendix C for the purpose of being self-contained.

2. The definition of additive PDBs and the explanation of how such PDBs can be built for MAPF (Section 8.2).

3. The description of the basic additive PDBs-based OSF for MAPF (Section 8.3). We have a number of performance enhancements for this OSF, which are described in Appendix D.

4. The report on our experimental results (Section 8.4).

8.1 The Multi-Agent Pathfinding Domain (MAPF)

Consider the following commonly used variant of MAPF (Standley, 2010; Sharon et al., 2011, 2012). The input consists of: **(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 at the location s_i. Between successive time points, each agent can perform a move action to a neighboring location or can wait (i.e. stay idle) at its current location. An operator consists of an action (which may be Wait) for every agent. Every legal operator has to respect two constraints:

1. Each vertex can be occupied by at most one agent at a given time and

2. If x and y are neighboring vertices, two different agents cannot simultaneously traverse the connecting edge in opposite directions (from x to y and from y to x). 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 legal operators that bring each agent to its goal position while minimizing 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, except for the case when the Wait action is applied at an agent's goal location, in which case it costs 0. An exception is the case when an agent waits m times at its goal location and then moves: the cost of that move is $m + 1$.

A*-based optimal solvers of MAPF use the *Sum of the Individual Costs* (SIC) heuristic, which is the optimal cost of solving the problem when the legal operator constraints are ignored. The first use of PDBs for MAPF was recently reported (Goldenberg et al., 2012). We will explain below why applying abstraction-based heuristics is challenging for MAPF. Two techniques have become standard in A*-based optimal MAPF solvers (Standley, 2010):[8]

- *Independence Detection* (ID) tries to reduce the part of a MAPF problem instance's complexity that is due to agents' interactions (i.e. the legal operator constraints).

- *Operator Decomposition* (OD) reduces the number of surplus nodes generated by A*. We call the resulting variant of A* the *Operator Decomposition A** (ODA*).

Both ID and ODA* are described in more detail in Appendix C. In that appendix, we also introduce the *Partial Expansion ODA** (PEODA*), our hybrid between ODA* and PEA*.

8. The recent papers by Röger and Helmert (2012) and Bouzy (2013) stand out as far as solving MAPF non-optimally is concerned.

8.2 Additive PDBs for MAPF

In this section, we define additive PDBs and show why constructing such PDBs for MAPF is challenging.

8.2.1 THE DEFINITION OF ADDITIVE PDBS

Consider a set of abstractions $\Phi = \{\phi_1, \phi_2, \ldots, \phi_k\}$ for some given domain. For each abstraction ϕ_i, a PDB, denoted PDB_i can be built. For an arbitrary state s, $PDB_i[\phi_i(s)]$ stores the distance from s to the goal in the abstract space of ϕ_i. The set Φ is called *additive* if, for any state s of the original state space, the sum $\sum_i PDB_i[\phi_i(s)]$ is not greater than the distance from s to the goal in the original space. PDBs based on additive abstractions are called *additive PDBs* (Felner et al., 2004; Yang et al., 2008).

Suppose that we have built and stored a Δ-PDB ordered by $\Delta f(a_c)$ for each abstraction in the additive set of abstractions Φ. We denote these Δ-PDBs by Δ-PDB$_1$, Δ-PDB$_2, \ldots,$ Δ-PDB$_k$. We make two assumptions motivated by MAPF:

1. Each variable participates in at least one of the abstractions in Φ.
2. Given a node n and an operator that results in n_c, $\Delta f(n_c)$ is given by:

$$\Delta f(n_c) = \sum_{i=1}^{k} \Delta f(\phi_i(n_c)), \tag{1}$$

   Intuitively, this means that each operator may affect all of the variables and so all of the Δ-PDBs have to be looked up to determine $\Delta f(n_c)$ of a particular operator.

8.2.2 ADDITIVE PDBS FOR MAPF

For a given node n, we say that agents $a_1, a_2, \ldots, a_m$ are *in conflict* if the SIC heuristic for these agents (i.e. when all other agents are ignored) is not perfect. Intuitively this definition means that, whatever optimal plan is chosen for each individual agent, these plans cannot be executed simultaneously without agents colliding into each other. PDBs for MAPF can provide useful heuristic information only if they are built for agents that will be in conflict for many nodes during the main search. However, this information is not known *a priori*. We solve this problem for the case of abstractions that consist of two variables (i.e. locations of two agents). We call PDBs based on such abstractions *pairwise PDBs*.

Recall that ID is used on top of EPEA* (the reader unfamiliar with ID should refer to Appendix C at this point). The idea of our solution is to consider the merge actions of ID to be an indication that the agents being merged are in conflict for many nodes of the main search. Namely, whenever ID merges two agents into a group, we use this information to build pairwise PDBs at later stages of ID. We build *additive PDBs* that consist of pairwise PDBs built this way.

Suppose, for example, an instance with 10 agents. Consider an execution of ID, while ignoring all operations except the operation of merging two groups into a single group. Suppose that ID merged agents $\{1, 5\}$, then merged agents $\{2, 8\}$ and then merged 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

$\Delta f(\sigma[\phi_1(n)])$	σ	$\Delta f(\sigma[\phi_2(n)])$	σ	$\Delta f(\sigma[\phi_3(n)])$	σ
0	$\Sigma_{11} = \{\sigma_{11}, \sigma_{12}\}$	0	$\Sigma_{21} = \{\sigma_{21}\}$	0	$\Sigma_{31} = \{\sigma_{31}\}$
3	$\Sigma_{12} = \{\boldsymbol{\sigma_{13}}, \sigma_{14}\}$	**1**	$\Sigma_{22} = \{\boldsymbol{\sigma_{22}}\}$	1	$\Sigma_{32} = \{\sigma_{32}\}$
5	$\Sigma_{13} = \{\sigma_{15}\}$	3	$\Sigma_{23} = \{\sigma_{23}\}$	**4**	$\Sigma_{33} = \{\boldsymbol{\sigma_{33}}, \sigma_{34}\}$

Figure 8: Entries of Δ-PDBs for a given node n.

onto agents $\{2, 8\}$. In our experiments, we used instance dependent pattern databases described above.

In the following section, we describe a direct-computation OSF constructed for additive PDBs built as described above. It is important to note that, in MAPF, an operator affects *all* of the agents and Equation 1 holds.

Note that SIC is the trivial case of PDBs, since the heuristic based on single-agent additive PDBs is exactly the SIC heuristic. Therefore, the OSF based on additive abstractions that we will develop below will be applicable to SIC as well.

8.3 Constructing a Direct-Computation OSF for Additive PDBs

Recall from Section 5.2 that one way to build a direct-computation OSF is by defining a classification of the states. In particular, when we built a direct-computation OSF for a single PDB in Section 7.2, we defined these classes by putting each abstract state into a class of its own. In contrast, a set of abstractions does not define a classification of the states. Therefore, we do not use classes (this is equivalent to putting all states into one class). Instead, the algorithmic component of the OSF will compute the set of operators with the required value of $\Delta f(n_c)$ on-the-fly as we describe below.

For example, suppose three additive abstractions: ϕ_1, ϕ_2, ϕ_3. Figure 8 shows the entries that the corresponding Δ-PDBs ordered by $\Delta f(a_c)$ might contain for a particular node n being expanded. Let σ_{ij} denote the j^{th} abstract operator applicable to $\phi_i(n)$. Figure 8 groups the abstract operators that result in the same $\Delta f(a_c)$. These groups are denoted by Σ, i.e. Σ_{ij} is the j^{th} group of abstract operators applicable to $\phi_i(n)$. In our example, the group Σ_{11} contains two abstract operators: σ_{11} and σ_{12}. There are 5, 3 and 4 abstract operators available for projections of n onto each of the three abstractions, respectively. This means that there could be up to $5 \times 3 \times 4 = 60$ operators for n in the original space. However, it may be that *not* every combination is a legal operator in the original space. For example, it could be that the changes to the variables performed by σ_{14} and the changes to the variables performed by σ_{21} cannot be applied at the same time according to the rules of the domain (e.g. because two agents cannot occupy the same location in MAPF).

Suppose that we need to compute the set of operators applicable to n with $\Delta f(n_c) = 8$. According to Equation 1, we need to find all ways to choose one abstract operator for each of the three projections of n, such that the sum of corresponding $\Delta f(a_c)$-values would equal eight. One such choice is σ_{13}, σ_{22}, σ_{33} (these abstract operators and their $\Delta f(a_c)$-values are shown in Figure 8 in bold). Furthermore, we could choose any one operator from each of the groups Σ_{12}, Σ_{22}, Σ_{33}.

In general, for k abstractions, we find all operators with the required $\Delta f(n_c)$ by finding all ways to choose one abstract operator for each abstraction, such that the sum of corresponding $\Delta f(a_c)$-values is equal to the required $\Delta f(n_c)$. We note that this is a combinatorial problem that is exponential in the number of abstractions. Since this problem has to be solved for every expansion, it

Procedure 3 Algorithmic component of an additive PDBs-based OSF for MAPF.

Variables:

k – number of abstractions
i – current abstraction
C – sum of $\Delta f(a_c)$-values selected from abstractions $1, 2, \ldots, i-1$
sum – sum of $\Delta f(a_c)$-values selected from abstractions $1, 2, \ldots, i$
$nOps_i$ – number of abstract operators applicable to $\phi_i(n)$
op_i – the current choice of abstract operator for $\phi_i(n)$
op – the current choice of abstract operators for abstractions $1, 2, \ldots, i$
OP – the set of all op's that result in the required $\Delta f(n_c)$. Some of these op's may be illegal
N – the set of currently needed nodes returned by the OSF

1: **FindAbsOpSets**(i, C, op)
2:     **for** j **from** 1 **to** $nOps_i$ **do**
3:         Set $op_i \leftarrow \sigma_{ij}$
4:         Set $sum \leftarrow C + \Delta f(\sigma_{ij}[\phi_i(n)])$
5:         **if** $sum > \Delta f(n_c)$ **then**
6:             Set $F_{next}(n) \leftarrow \min(F_{next}(n), sum)$
7:             **return**
8:         **if** $i == k$ **then**
9:             **if** $sum == \Delta f(n_c)$ **then** append op to OP
10:             **continue**
11:         **Call FindAbsOpSets(**$i+1$**,** sum**,** op**)**
12: **OSF**($\Delta f(n_c)$)
13:     Set $N \leftarrow \emptyset$, $OP \leftarrow \emptyset$ and $F_{next}(n) \leftarrow \infty$
14:     **Call FindAbsOpSets(**1**,** 0**,** $\emptyset$**)**
15:     **for each** op **in** OP **do**
16:         **if** op is legal **then**
17:             Append n_c corresponding to op to N
18:     **return** $N, F_{next}(n)$

---

is critical that this problem be solved efficiently. We present a simple basic algorithm for solving this problem in Procedure 3. Enhancements to this simple algorithm are presented in Appendix D. In addition to finding operators with the required $\Delta f(n_c)$, this algorithm computes the next stored value of n, $F_{next}(n)$. Thus, Procedure 3 is a full-fledged direct-computation OSF.

In Procedure 3, the main part of the OSF starts on line 12. First, the set of currently needed operators, N, is initialized to be empty and the next stored value, $F_{next}(n)$ is initialized to infinity. We also initialize to empty the set OP, whose meaning is explained below. Then a call to the FindAbsOpSets (which stands for "find abstract operator sets") procedure is made. This procedure takes three parameters:

1. i – the current abstraction,
2. C – the sum of $\Delta f(a_c)$'s corresponding to the abstract operators chosen for abstractions 1 though $i-1$, and

3. op – the current choice of abstract operators for abstractions $1, 2, \dots, i$.

and produces two results:

1. OP, the set of choices of abstract operators, one per abstraction, that result in the required $\Delta f(n_c)$ and
2. $F_{next}(n)$.

The OSF computes N by filtering out the illegal operators from OP and returns (lines 15-end).

We now walk through FindAbsOpSets, which implements a simple recursive algorithm. When a given level of recursion begins, an abstract operator is chosen for each of the abstractions $1, 2, \dots, i-1$. These operators are stored in op. The sum of $\Delta f(a_c)$'s corresponding to these abstract operators is stored in C. There are two cases:

1. The current abstraction is k (i.e. $i = k$). This is the base of the recursion. In this case, we look for the abstract operators for the current abstraction with $\Delta f(a_c)$ that complements C to the required $\Delta f(n_c)$ ($C + \Delta f(a_c)$ is denoted by sum in Procedure 3). For each chosen operator, the tuple op appended by this operator is stored in OP (lines 8-10).
2. The current abstraction is not k (i.e. $i < k$). In this case, we look for the abstract operators for the current abstraction such that $sum = C + \Delta f(a_c)$ would not exceed the required $\Delta f(n_c)$ and invoke the next level of recursion (line 11).

In addition, whenever an abstract operator that results in $sum = C + \Delta f(a_c)$ that exceeds the required $\Delta f(n_c)$ is encountered, we can update $F_{next}(n)$ and not look at the following abstractions (lines 5-7).

In Appendix D, we present four enhancements to the basic algorithm just described.

8.4 Experimental Results

Tables 5 and 6 compare node and time performance, respectively, of six A*-variants for optimally solving MAPF. For EPEA*, two variants are shown: one using the SIC heuristic and the other using an additive PDBs-based OSF as described above. The Δ-PDBs were constructed on-the-fly. That is, when a node n was expanded for the first time, the entries of the Δ-PDBs corresponding to the abstractions of n were computed if they had not existed already.

There were a total of 1,000 instances. In all instances, agents were placed onto a four-connected 8x8 grid with no obstacles. We varied the number of agents. All algorithmic variants were run under the ID framework as described in the footnote.[9] All variants were given up to two minutes and two gigabytes of memory per instance. The results were bucketed according to the number of agents (k,

9. Since the ID framework can produce different results due to reasons that are not related to the performance properties of the different A* variants for MAPF (see Appendix C), we compared these variants using the following approach (Sharon et al., 2011). 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 were discarded. The variants were then compared on these instances without the use of ID.

k	Ins	Unique Nodes Generated, $\times 10^3$ A*	ODA*	PEA*	PEODA*	EPEA*	EPEA* with abstractions
Instances solved by both A* and PEA* within two minutes and 2GB memory							
2-6	794	46.35	1.27	0.08	0.38	0.08	**0.06**
7-8	34	1,261.04	3.26	0.11	0.88	0.10	**0.05**
9-10	0	n/a	n/a	n/a	n/a	n/a	n/a
Instances solved by neither A* nor PEA* within two minutes and 2GB memory							
2-6	1	n/a	335.34	n/a	105.14	9.94	**9.31**
7-8	25	n/a	219.11	n/a	67.04	7.82	**4.41**
9-10	13	n/a	705.76	n/a	211.54	17.57	**10.01**

Table 5: Comparison of nodes performance of six algorithms for the Multi-Agent Pathfinding.

k	Ins	Run-Time, ms A*	ODA*	PEA*	PEODA*	EPEA*	EPEA* with abstractions
Instances solved by both A* and PEA* within two minutes and 2GB memory							
2-6	794	647	5	606	5	**2**	33
7-8	34	22,440	14	14,886	12	**2**	100
9-10	0	n/a	n/a	n/a	n/a	n/a	n/a
Instances solved by neither A* nor PEA* within two minutes and 2GB memory							
2-6	1	n/a	2,153	n/a	1,803	**278**	354
7-8	25	n/a	1,637	n/a	1,312	335	**232**
9-10	13	n/a	16,660	n/a	8,846	3,062	**1,089**

Table 6: Comparison of time performance of six algorithms for the Multi-Agent Pathfinding.

shown in the first column of Tables 5 and 6) and results for instances falling into the same bucket were averaged.

Since the basic A* and PEA* perform much worse than the other variants, we split the tables into two halves. The upper part of the table shows results for instances that were solved within the allowed resources by both A* and PEA*. We see that EPEA* generates four orders of magnitude less nodes and is three orders of magnitude faster than A*. Also, it generates an order of magnitude less nodes and is up to seven times faster than ODA*. EPEA* with an additive PDBs-based OSF generates the fewest number of nodes among all variants, but is not the fastest in terms of time. This is because, for these simple instances, the overhead of building the (Δ-)PDBs does not pay off.

The lower part of the table shows results for instances that were not solved by either A* or PEA*, but solved by the other variants. The following trends can be observed. Applying partial expansion on top of ODA* results in three-fold reduction in the number of generated nodes and up to two times speed-up over ODA*. However, EPEA* generates yet another order of magnitude less nodes and is up to four times faster. EPEA* with additive PDBs-based OSF is the clear winner for the hard instances, running up to three times faster than EPEA* based on the SIC heuristic.

The next two sections are a theoretical study of PEA* and EPEA*.

9. The Size of OPEN: A Limitation of PEA* and EPEA*

In this section, we will show that, although the purpose of PEA* is to make OPEN smaller, it sometimes has exactly the opposite effect.

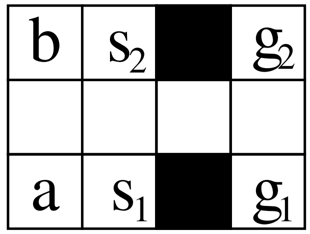


Figure 9: For this instance of MAPF, PEA*/EPEA* will not close even the start node.

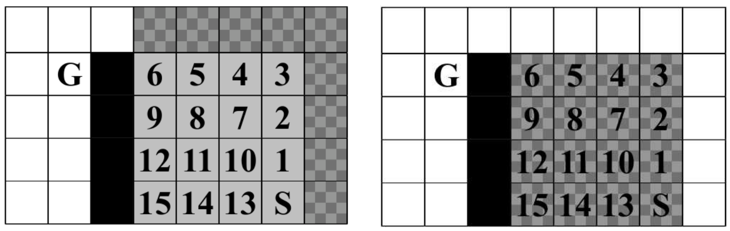


Figure 10: Open (checkerboard) and closed (gray) nodes after 16 node expansions for A* (left) and PEA*/EPEA* (right).

When PEA*/EPEA* expands a node n, it puts into OPEN only the currently needed children of n. If n has possibly surplus children, then n will be put back into OPEN with an updated stored value. This is in contrast to A*, which always puts n into CLOSED after the expansion. This may result in PEA*/EPEA* maintaining a much larger OPEN than A*. Note that OPEN is frequently implemented as a heap and heap operations become slower as the number of elements stored in the heap grows. Therefore, the time performance may suffer considerably when the size of OPEN increases.

Consider the MAPF example in Figure 9. In this instance, two agents start at locations (s_1, s_2) and need to get to locations (g_1, g_2). The SIC heuristic at the start state is 8, while the optimal solution length is 9 (namely, one agent has to wait for the other agent to pass). However, the highest-cost (i.e. with the agents at (a, b)) child of the start node has the f-value of 12. Therefore, while A* closes all the expanded nodes, PEA*/EPEA* will never close any of the nodes, not even the start node.

Furthermore, it is possible for a node n not to be closed by PEA*/EPEA* even though the neighbors of n are surely not surplus. Figure 10 (left) shows this with an instance of the single-agent pathfinding problem on a four-connected grid, where the shortest path is needed from cell S to cell G. Black cells are obstacles. We assume the Manhattan distance heuristic. 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 closed, and the 8 states around them (marked with a checkerboard) are in the OPEN list, all with $f = 10$.

Consider the operation of PEA*/EPEA* on the same example. When PEA*/EPEA* expands the gray nodes it does not close them because each has a child with $f = 10$. For example, when expanding state 10, operators North and West have $\Delta f(n_c) = 0$, while the operators East and South have $\Delta f(n_c) = 2$. The cell east of 10 is the cell 1 which was already generated with a lower g-value and with $f = 8$. However, PEA*/EPEA* do not perform duplicate detection for the currently unneeded nodes. Thus, cell 10 is re-inserted into the OPEN list with $F = 10$, received from cell 1. Consequently, after expanding all the gray cells for the first time, PEA*/EPEA* has 16 OPEN nodes as shown in Figure 10 (right) and no CLOSED nodes. In the case of PEA*, this problem can be fixed at the expense of the run-time overhead of performing a hash look up for the currently unneeded children before discarding them (note that this fix does not address the large OPEN problem shown in the MAPF example above). In the case of EPEA*, such duplicate detection is impossible, since EPEA* does not actually generate the currently unneeded children.

To summarize, A* stores only the perimeter of the generated states in the OPEN list, while PEA*/EPEA*, in the worst case, stores in the OPEN list all of the states that have ever been generated during the search. For polynomial domains (defined in the footnote[10]), this considerably affects the time performance of PEA*/EPEA*. For exponential domains, this factor is of lesser importance, since, for these domains, A* stores most of the nodes in OPEN as well. That is why we did not observe that EPEA* was slower than A* in any of our experiments besides the experiments with SAPF (not shown), where EPEA* was slightly slower than A*.

10. Performance analysis

The purpose of this section is to analytically estimate how the time performance of EPEA*/EPEIDA* and the regular A*/IDA* compare. To do this, we identify the basic operations within each of the algorithms and give notation to the time costs of these operations. We then use this notation to give precise conditions for obtaining a time speed-up by using the enhanced partial expansion variant of A*/IDA*.

The operations of the algorithms under consideration are listed in Table 7. For each operation, we denote its applicability to a given algorithm by putting a "+" in the respective column. If an operation is applicable only when a full-checking OSF or a direct-computation OSF is used, we denote this fact by "+(FC)" and "+(DC)", respectively. The operations appear in the same order in which they appear in Procedures 1 and 2.[11] To introduce as little notation as possible, we group some of these operations together and assign notation to the total time cost of the operations in each group as shown in Table 8. Note that these time costs are the averages over all node expansions or generations, whichever is applicable to the particular group of operations. We skip the word "average" in the text for brevity. We will explain these operations and groups on-the-fly when they are used in the analysis. Analysis of EPEIDA* is simpler than the analysis of EPEA*. Therefore we start with the analysis of EPEIDA*.

10.1 Analysis of EPEIDA*

For simplicity, we restrict our analysis to the last iteration of IDA*. Let X be the number of nodes expanded by this iteration. Let b be the average branching factor of the domain. We approximate the

10. *Polynomial domains* are domains where the number of distinct states at depths up to d of the breadth-first search tree starting from any given state is $\Theta(d^m)$, where m is a domain-specific constant. SAPF is a classical example of a polynomial domain. *Exponential domains* are domains where the number of distinct states at depths up to d of the breadth-first search tree starting from any given state is $\Theta(b^d)$, where b is the average branching factor of the domain. 15-puzzle, Rubik's cube and pancake puzzle are examples of exponential domains.

11. Hence the heuristic computation appears twice: operation 3 is for A*, while operation 9 is for IDA*.

#	Operation	Applicability				
		A*	PEA*	EPEA*	IDA*	EPEIDA*
1	Remove from OPEN	+	+	+	-	-
2	Recursive call of IDA*	-	-	-	+	+
3	Compute the node's heuristic (IDA*)	-	-	-	+	+
4	Check the threshold condition of IDA*	-	-	-	+	+
5	Compute a child node's heuristic when the node has not been generated	-	-	+(FC)	-	+(FC)
6	Check one **operator** using a full-checking OSF	-	-	+(FC)	-	+(FC)
7	Check one **operator** using a direct-computation OSF	-	-	+(DC)	-	+(DC)
8	Apply an operator (in A*, includes copying the parent)	+	+	+	+	+
9	Compute a child node's heuristic when the node has been generated	+	+	-	-	-
10	Insert one child into OPEN	+	+	+	-	-

Table 7: Summary of operations performed by the five algorithms under consideration. A plus sign stands for an operation that is performed by the given algorithm, while a minus sign stands for an operation that is not applicable in the context of the given algorithm.

Notation	Operations	Comments
t_e	1	Has different cost in A* and EPEA*, so, in EPEA*, we use t'_e
t_r	8, 2, 3, 4	
t_{of}	6	In the case of EPEA*/EPEIDA*, also calls operation 5.
t_{od}	7	
t_m	8, 9, 10	For A*
t'_m	8, 10	For EPEA*

Table 8: Notation for the time costs of the groups of operations. The operations are denoted by the numbers introduced by Table 7.

number of nodes generated by IDA* by bX. Since EPEIDA* generates only the currently needed children (Section 4), it expands all of the nodes that it generates. The only exception from this rule is when the solution has been found. In that case, some nodes generated at the upper levels of the depth-first tree may remain unexpanded. Therefore, EPEIDA* generates up to $X + (b-1)d$ nodes. Since $(b-1)d$ is usually very small compared to X, we will ignore this quantity in the following analysis. This will allow us to express the ratio between the run-time costs of IDA* and EPEIDA* in terms of the parameters of the domain and the implementation without the usage of X.

For each operator available for a given node n, IDA* generates a child n_c by applying the operator (operation 8), makes the recursive call of IDA* (operation 2), computes $h(n_c)$ (operation 3) and checks the threshold condition (operation 4). We denote the total time cost of these operations by t_r. Since these operations are performed by IDA* for each generated node, the run-time cost of IDA* is:

$$bXt_r. \tag{2}$$

10.1.1 Analysis of EPEIDA* with a Full-Checking OSF

In EPEIDA* with a full-checking OSF, when a node n is expanded, the cost t_r is spent only on currently needed operators applicable to n. The other operators are checked (operation 6), but the children are not generated. After denoting the time cost of checking an operator by a full-checking OSF by t_{of}, the run-time cost of EPEIDA* is given by:

$$bXt_{of} + Xt_r. \tag{3}$$

The ratio between the run-time costs of IDA* and EPEIDA* is expressed by:

$$\frac{bt_r}{bt_{of} + t_r}. \tag{4}$$

EPEIDA* is faster than IDA* when this ratio is greater than one, i.e. when

$$t_{of} < t_r \frac{b-1}{b}. \tag{5}$$

10.1.2 Analysis of EPEIDA* with a Direct-Computation OSF

EPEIDA* with a direct-computation OSF checks only the operators leading to the X expanded nodes. After denoting the time cost of checking an operator by a direct-computation OSF (operation 7) by t_{od}, the run-time cost of EPEIDA* is:

$$X(t_{od} + t_r). \tag{6}$$

The ratio between the run-time costs of IDA* and EPEIDA* is expressed by:

$$b\frac{t_r}{t_{od} + t_r}. \tag{7}$$

EPEIDA* is faster than IDA* when this ratio is greater than one, i.e. when

$$t_{od} < t_r(b-1). \tag{8}$$

10.1.3 Analysis of EPEIDA* with a Hybrid OSF

Recall that a hybrid OSF can behave either as a direct-computation or as a full-checking OSF for different expansions depending on the required $\Delta f(n_c)$. For example, the OSF for the pancake puzzle introduced in Section 6.2 is a direct-computation-OSF when nodes with $\Delta f(n_c) = 0$ are needed and is a full-checking-OSF when nodes with $\Delta f(n_c) = 1$ or $\Delta f(n_c) = 2$ are needed.

Let $G' \leq X$ be the number nodes for whose *generation* a direct-computation OSF is used. Also, let $E' \leq X$ be the number of nodes for whose *expansion* a direct-computation OSF is used. Note that we cannot easily express G' in terms of E' (i.e. it would be wrong to state that $G' = bE'$), since the direct-computation OSF generated only the currently needed children of n. The run-time cost of EPEIDA* is given by

$$G't_{od} + b(X - E')t_{of} + Xt_r. \tag{9}$$

The ratio between the run-time costs of IDA* and EPEIDA* is expressed by:

$$\frac{bXt_r}{G't_{od} + b(X - E')t_{of} + Xt_r}. \tag{10}$$

G' and E' are not known and we need to use a model that allows for a more detailed analysis to estimate them. Fortunately, a suitable model has been reported (Zahavi, Felner, Burch, & Holte, 2010). Since using this model to estimate G' and E' is technically involved and does not provide an immediate insight into the efficiency of EPEA*, we defer this analysis to Appendix B.

10.2 Analysis of EPEA*

To analyze the time performance of EPEA*, we will need to introduce two auxiliary parameters that characterize the performance of EPEA*. Let α be the average number of times that any given node is expanded (each time with a different stored value). Let β be the average number of currently needed children during any given node expansion. In particular, when $\alpha = 1$ and $\beta = b$ (b is the average branching factor), EPEA* is equivalent to A*. Note that these parameters are domain, heuristic and problem instance-dependent.

Suppose that A* performs X expansions and bX generations. Since EPEA* expands each node an average of α times, it will perform a total of αX expansions. Since EPEA* performs an average of β generations per expansion, it will perform a total of $\beta\alpha X$ generations.

Let t_e denote the time cost of getting the least cost node from OPEN (operation 1). For each operator applicable to n, A* generates a child n_c (operation 8), computes $h(n_c)$ (operation 9) and inserts n_c into OPEN (operation 10). We denote the total cost of these three operations by t_m. The run-time cost of the regular of A* is given by:

$$Xt_e + bXt_m. \tag{11}$$

10.2.1 ANALYSIS OF EPEA* WITH A FULL-CHECKING OSF

Recall from Section 5.2 that one can trade memory for speed by imitating a direct-computation OSF even when such an OSF is not available. The following analysis is for the "pure" version of EPEA* with a full-checking OSF where this trade-off is not used.

EPEA* performs αX expansions, each time checking b operators, followed by generating β currently needed children nodes. Denote the time cost of expanding a node by EPEA* by t'_e. Note that t_e and t'_e may be different for the same problem instance, since the size of OPEN differs between the expansions of A* and EPEA*. Similarly, we denote the total time cost of generating a child node n_c, computing $h(n_c)$ and putting n_c into OPEN by EPEA* by t'_m, which might differ from t_m. The run-time cost of EPEA* is given by:

$$\alpha Xt'_e + b\alpha Xt_{of} + \beta\alpha Xt'_m. \tag{12}$$

The ratio between the run-time costs of the regular A* and EPEA* is expressed by:

$$\frac{t_e + bt_m}{\alpha\big(t'_e + bt_{of} + \beta t'_m\big)}. \tag{13}$$

EPEA* is faster than the regular A* when this ratio is greater than one, i.e. when

$$t_{of} < \frac{t_e + bt_m - \alpha(t'_e + \beta t'_m)}{\alpha b}. \tag{14}$$

10.2.2 ANALYSIS OF EPEA* WITH A DIRECT-COMPUTATION OSF

With a direct-computation OSF, EPEA* performs αX expansions, each time checking β operators and generating β currently needed children nodes. The run-time cost of EPEA* is given by:

$$\alpha X t'_e + \beta\alpha X(t_{od} + t'_m). \tag{15}$$

The ratio between the run-time costs of the regular A* and EPEA* is expressed by:

$$\frac{t_e + bt_m}{\alpha\big(t'_e + \beta(t_{od} + t'_m)\big)}. \tag{16}$$

EPEA* is faster than the regular A* when this ratio is greater than one, i.e. when

$$t_{od} < \frac{bt_m - \alpha t'_e + t_e}{\alpha\beta} - t'_m. \tag{17}$$

10.3 Conclusion from the Performance Analysis

From the above analysis we see that whether EPEA*/EPEIDA* is faster or slower than A*/IDA* depends on:

1. The average branching factor b of the domain,
2. The efficiency of implementation of A*/IDA* (expressed by t_e, t_r and t_m),
3. The efficiency of the OSF being used (expressed by t_{of} or t_{od}), and
4. For EPEA*, the domain and heuristic-dependent parameters α and β.

To obtain experimental evidence of the analysis of this section, one would have to find a way to reliably estimate the quantities in Table 8. However, such estimation meets with two challenges:

1. The operations under discussion are of very fine granularity. The time performance of these operations cannot be measured directly, but has to be measured by a sampling procedure.
2. For EPEA*, estimating the parameters α and β presents and additional challenge.
3. The time performance of some of the operations under discussion is not constant throughout the search. For example, the cost of inserting a node into OPEN depends on the current size of OPEN. Also, there are domains where nodes have variable branching factors.

The latter observation opens a possibility for an algorithm that, depending on the current state of the search, switches between A*/IDA* and EPEA*/EPEIDA*.

11. EPEA* With Inconsistent Heuristics

A heuristic h is called *consistent* if f-value does not decrease along any path, i.e. $f(n_c) \geq f(n)$. Otherwise, h is called *inconsistent*. So far, we assumed a consistent heuristic. In this section, we will **(1)** show the changes that need to be made to EPEA* (Procedure 1 of Section 2.2) when an inconsistent heuristic h is used and **(2)** point out that one has to make a choice between using EPEA*

Procedure 4 A*, PEA* and EPEA* for an inconsistent heuristic.

```
Generate the start node n_s
Compute h(n_s) and set F(n_s) ← f(n_s) ← h(n_s)
Put n_s into OPEN
while OPEN is not empty do
    Get n with lowest F(n) from OPEN
    if n is goal then exit                                  // optimal solution is found!
    For A* and PEA*: set N ← set of all children of n and initialize F_next(n) ← ∞
    For EPEA*:
        if f(n) = F(n) then
            Set (N, F_next(n)) ← OSF_e(n).
        else
            Set (N, F_next(n)) ← OSF(n).
    for all n_c ∈ N do
        Compute h(n_c), set g(n_c) ← g(n) + cost(n, n_c) and f(n_c) ← g(n_c) + h(n_c)
        For PEA*:
            if f(n_c) ≠ F(n) then
                if f(n_c) > F(n) then
                    Set F_next(n) ← min(F_next(n), f(n_c))
                if f(n_c) > F(n) or F(n) ≠ f(n) then
                    Discard n_c
                    continue                                // To the next n_c
        Check for duplicates
        Set F(n_c) ← f(n_c) and put n_c into OPEN
    if F_next(n) = ∞ then
        Put n into CLOSED                                   // For A*, this is always done
    else
        Set F(n) ← F_next(n) and re-insert n into OPEN      // Collapse
exit                                                        // There is no solution.
```

and using heuristic value propagation techniques (such as *bidirectional pathmax* (BPMX), Zahavi, Felner, Schaeffer, & Sturtevant, 2007) for inconsistent heuristics. A deep experimental study of EPEA* with inconsistent heuristics including the trade-off due to the aforementioned choice is beyond the scope of this paper.

11.1 Changes to PEA* and EPEA*

We will start by discussing PEA*, since PEA* makes the choice of what nodes to insert into OPEN explicitly (line 12 of Procedure 1).

When the heuristic is inconsistent, it is possible that there are children n_c of n with $f(n_c) < f(n)$. These children nodes are not surplus and PEA* will need to insert them into OPEN during the first expansion of n. Thus, during the first expansion of n the definition of a *currently needed* node is changed to include all children with $f(n_c) \leq F(n)$.

This is demonstrated in Figure 11. When the node a is expanded, all of its children (x, y, z, w) are generated. However, only the children with the f-values *not exceeding* 3 (x and y) are currently

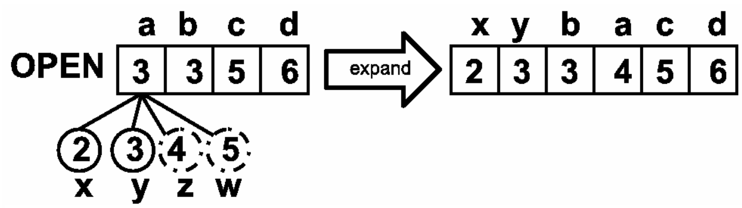


Figure 11: Example of PEA* with an inconsistent heuristic. The first expansions of the node a is shown. Even the children nodes whose f-value is less than $f(a)$ are generated.

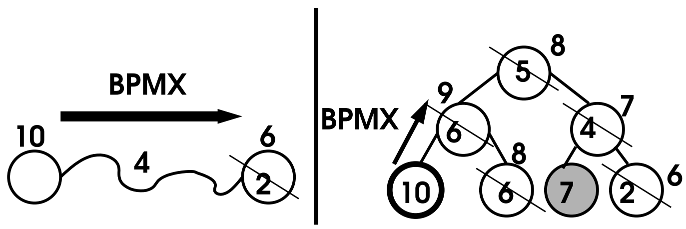


Figure 12: Two examples of value propagation by BPMX.

needed. They are inserted into OPEN. All other children are possibly surplus. They are collapsed back into a, who gets the new stored value $F(a) = 4$. Following this, a is re-inserted into OPEN. From here on, a is treated as for the consistent heuristic case.

The pseudo-code of A*, PEA* and EPEA* for the case of an inconsistent heuristic is shown in Procedure 4. Compared to Procedure 1 in Section 2.2, PEA* handles the inconsistent heuristic by performing the check in line 19 before discarding n_c.

In EPEA*, the case of inconsistent heuristic is also handled by checking whether n is being expanded for the first time (line 9). For the first expansion of n, the OSF that returns the set of children with $f(n_c) \leq F(n)$ is used. We call this an *extended OSF* and denote it by OSF_e (line 10). Note that a similar OSF is used by EPEIDA* (Section 4) as well. For other expansions of n, no change in the OSF is needed (line 12).

11.2 The Trade-Off

When an inconsistent heuristic is used, heuristic value propagation techniques can be applied to take advantage of regions of the state space with high heuristic values. One of the most effective such techniques is the *bidirectional pathmax* (BPMX) (Zahavi et al., 2007). An example of BPMX's operation is shown in Figure 12 (right). Assuming unit edge costs, the h-value of the left grandchild (10) is propagated up and then down the search tree, increasing heuristic estimates of all states in its neighborhood except for the gray node. In general, for two arbitrary states $a, b \in V$, the heuristic estimate $h(a, g)$ can be updated to be $\max\{h(a, g), h(b, g) - d(a, b)\}$ (shown in Figure 12 (left)) and BPMX uses this rule in both directions of the search tree.

When EPEA* is used, the possibly surplus children of n are not generated and, therefore, the propagation of values from these children to n is not available. This introduces an interesting trade-off:

- EPEA* saves both memory and time by not generating surplus nodes, but
- The opportunity to increase the quality of heuristic estimates by using the non-monotonicity of f is being missed.

A separate study might be dedicated to this trade-off.[12]

12. Conclusions and Future Work

We have presented the *Enhanced Partial Expansion A** (EPEA*), a novel variant of A* which avoids *generating* the surplus nodes. This is enabled by using *a priori* domain- and heuristic-specific knowledge in the form of an *Operator Selection Function* (OSF) to compute the list of operators that lead to the children with the needed f-cost without actually generating any children of the node being expanded. We studied several kinds of OSF, including the OSFs based on (additive) pattern databases. We extended the principles of EPEA* to IDA* resulting in the *Enhanced Partial Expansion IDA** (EPEIDA*). Experimental results with the 15-puzzle, the pancake puzzle, the Rubik's cube and the multi-agent pathfinding show that EPEA*/EPEIDA* achieves state-of-the-art run-time performance. Furthermore, EPEA* fully maintains the memory savings offered by PEA*.

EPEA* is most effective for domains and heuristics that meet the following criteria:

1. The domain possesses a large branching factor. The large branching factor is an indicator that A* may generate a large number of surplus nodes. EPEA* will save this overhead.
2. The set of possible f-values of the children of a given node is small. This means that EPEA* will not re-expand the same node many times.
3. The operators can be classified according to their $\Delta f(n_c)$ value, so that operators of the needed class can be applied without the need to check all operators of the node being expanded. In other words, a direct-computation OSF is available.

If only some of these conditions are met by the specific domain and heuristic of interest, then a more thorough assessment, such as experimenting with a prototype implementation, is needed. In addition, we explained that there is a possibility of poor performance of EPEA* in polynomial domains.

Future work will seek applications of EPEA* to other domains and heuristics. In particular, it would be very interesting to see whether EPEA* can be implemented for the best heuristics used in domain-independent planning. It seems that EPEA* can be easily implemented for the

12. Related to this trade-off is the following observation (Felner et al., 2011, p. 22). They studied the following trade-off in the context of applying BPMX to IDA*. Once propagation from children to a parent results in a cut-off, IDA* can either backtrack immediately (this option is called *lazy propagation*) or look at the other children in hopes to obtain an even higher value for the parent and obtain more cut-offs in the future. They concluded that backtracking immediately was preferable for all domains that they studied. However, we cannot conclude from this that one should always use EPEA* and forgo propagation of values from children to parents, since IDA* with lazy BPMX propagation does perform an effective propagation before backtracking.

STRIPS (Fikes & Nilsson, 1971) and the variable abstraction heuristics (Edelkamp, 2001). However, it remains to see whether EPEA* can be applied for merge-and-shrink abstractions (Helmert, Haslum, & Hoffmann, 2007) and landmarks (Karpas & Domshlak, 2009).

A promising direction for future work is to implement an OSF by using symbolic representation, such as BDDs (Binary Decision Diagrams). Jensen, Bryant, and Veloso (2002) proposed a method to group state transitions that have the same effect on the f-value. Such grouping of state transitions was referred to as an *improvement partitioning* and was used to avoid generating node with f-values larger than a given upper bound (Jensen, Hansen, Richards, & Zhou, 2006). Thus, we believe that it may be possible to create an OSF with a similar method.

Another interesting direction is to see how EPEA* can be used for non-optimal searches. In these searches, the notion of a surplus node needs to be defined based on the required quality of the solution.

Furthermore, we only touched upon the topic of EPEA* with inconsistent heuristics and pointed out that a trade-off exists between using EPEA* and leveraging the full power of value propagation techniques. An experimental study of this trade-off remains a subject for future work.

13. 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.

Acknowledgements are due to Kobi Shmerkovich, Tal Beja, Idan Morad and Shaked Zimmermann for performing some of the experiments for this paper.

Appendix A. Glossary of EPEA* Terms

Table 9 lists the terms introduced in this paper in the alphabetical order. It provides a brief definition of each term and a reference to the place where this term is defined in the paper.

Term	**Brief definition**	**Section(s)**
Algorithmic component of OSF	An algorithm that OSF employs to generate only the currently needed nodes and compute the next stored value of the node being expanded.	3.2
CFN	See "collapsing frontier nodes".	2.1
Checking an operator	Deciding whether to apply an available operator to generate a child node.	3.1
Collapse action	The operation of substituting some nodes in the search frontier by their common ancestor n, while increasing the cost of n. See "stored value".	2.1
Collapsing frontier nodes (CFN)	The technique used by a number of algorithms. See "collapse action".	2.1

Currently needed child node	A node that A*/IDA* has to generate during the current expansion to guarantee optimality. The definition differs for A* and IDA*.	2.2 and 4
Currently needed operator	An operator the results in a currently needed child node.	2.2 and 4
Currently unneeded child node	A child node that is not currently needed.	2.2 and 4
Currently unneeded operator	An operator that is not currently needed.	2.2 and 4
Direct-computation OSF	An OSF that does not need to check all operators to generate only the currently needed nodes.	5
Full-checking OSF	An OSF that needs to check all operators to generate only the currently needed nodes.	5
Hybrid OSF	An OSF that behaves either as a direct-computation or as a full-checking OSF depending on the needed change in the f-value ($\Delta f(n_c)$).	5
Knowledge component of OSF	A domain- and heuristic-specific data structure that the OSF stores. The algorithmic component uses the knowledge component to generate only the currently needed nodes and compute the next stored value of the node being expanded.	3.2
Operator selection function (OSF)	A function that uses domain- and heuristic-specific knowledge to generate only the currently needed nodes and compute the next stored value of the node being expanded.	3.2
OSF	See “operator selection function”.	3.2
Possibly surplus child node	A child node that is not provably useful.	2.2
Provably useful child node	A child node that can be proved to be useful. A child node n_c is provably useful if $f(n_c) \leq F(n)$.	2.2
Pure OSF	Non-hybrid OSF.	5
SAPF	The single-agent pathfinding domain.	3.2
Static value	The regular $g + h$ cost of a node, denoted $f(n)$.	2.1
Stored value	The cost of a node obtained by the collapse action and denoted $F(n)$.	2.1

Surplus node	A node whose static value is greater than the cost of the optimal solution.	1
Useful node	A node that is node surplus.	1

Table 9: The terminology used in the paper.

Appendix B. Analysis of the Hybrid OSF

We start by describing a model that aims to predict the number of nodes expanded by IDA* (Zahavi et al., 2010) (ZFBH).[13] Later in this section, we will show how the same model can be used to estimate the quantities G' and E' introduced in Section 10.1.3. The model uses the assumption of unit cost operators. Our analysis will make this assumption as well.

We start with a result that puts our study of the hybrid OSF into a more general framework than just the pancake puzzle example.

Lemma 1. *Let H be a natural number such that the hybrid OSF is direct when nodes with $\Delta f(n_c) \leq H$ are needed. Then, whenever a node n with $f(n) \geq T - H$ is expanded, where T is the threshold of the current iteration, this hybrid OSF is direct.*

Proof. Suppose that n with $f(n) \geq T - H$ is being expanded. Since in IDA* only the nodes with $f(n_c) \leq T$ are needed, we need to generate only the nodes with $\Delta f(n_c) = f(n_c) - f(n) \leq f(n_c) - T + H \leq H$. But for these nodes our OSF is direct. □

Remark 1. *In the pancake puzzle, H is zero.*

Remark 2. *We chose to focus on the simple model with two ranges of values of $\Delta f(n_c)$. The results of this section can be generalized to a model with a number of ranges designated by $H_1, H_2, \ldots, H_m$.*

B.1 The Model of ZFBH.

ZFBH define $N_i(s, d, v)$ to be the number of nodes that IDA* will generate at level i with a heuristic value equal to v when s is the start state and d is the IDA*'s iteration threshold. They give a recursive formula to approximate this quantity as follows:

$$\tilde{N}_i(s, d, v) = \sum_{v_p=0}^{d-(i-1)} \tilde{N}_{i-1}(s, d, v_p) \cdot b_{v_p} \cdot p(v|v_p). \quad (18)$$

In this equation, $p(v|v_p)$ is the probability of a child node having the heuristic value of v given that its parent heuristic value is v_p, b_{v_p} is the average branching factor of nodes whose heuristic value is v_p. Both these quantities can be obtained by sampling the state space (we refer the reader to ZFBH for details of this sampling process). We now explain the reasoning behind this equation. Since the parent node is located at level $i - 1$ of the depth-search tree, it could have been expanded only if

13. We use only the basic one-step model of ZFBH.

its heuristic was less than or equal to $d-(i-1)$ (otherwise, the threshold condition would have failed). For each heuristic value v_p, there are $\tilde{N}_{i-1}(s,d,v_p)$ parent nodes, each of whom have b_{v_p} children. However, only the children with the heuristic value of v are of interest. Since, the parent's heuristic value is fixed at v_p, we multiply by the conditional probability.

The total number of nodes expanded by this iteration of IDA* is then approximated by:

$$X \approx \sum_{i=0}^{d} \sum_{v=0}^{d-i} \tilde{N}_i(s,d,v). \tag{19}$$

The first summation runs through all depths of the current iteration's search tree. At a given depth i, only the nodes with heuristic value is less than or equal to $d-i$ satisfy the threshold condition. The second summation runs through all such heuristic values. Once both the depth i and the heuristic value v are fixed, $\tilde{N}_i(s,d,v)$, given by Equation 18, approximates the number of nodes expanded by IDA*.

B.2 Estimating E'.

By Lemma 1, our hybrid OSF is direct if $f(n) \geq d-H$. Subtracting the level number i from both d and $f(n)$, we get the equivalent condition:

$$v \geq (d-i)-H. \tag{20}$$

Therefore, we can modify Equation 19 to estimate E':

$$E' \approx \sum_{i=0}^{d} \sum_{v=(d-i)-H}^{d-i} \tilde{N}_i(s,d,v). \tag{21}$$

B.3 Estimating G'.

Let $G'_i(s,d,v)$ denote the number of nodes that EPEIDA* will generate by using direct-computation OSF at level i with a heuristic value equal to v when s is the start state and d is the IDA*'s iteration threshold. We can approximate $G'_i(s,d,v)$ by restricting the summation in Equation 18 to the parent nodes that are expanded with a direct-computation OSF. From Equation 20, this condition is: $v_p \geq [d-(i-1)]-H$. We have:

$$\tilde{G'}_i(s,d,v) = \sum_{v_p=[d-(i-1)]-H}^{d-(i-1)} \tilde{N}_{i-1}(s,d,v_p) \cdot b_{v_p} \cdot p(v|v_p). \tag{22}$$

We estimate G' as:

$$G' \approx \sum_{i=0}^{d} \sum_{v=0}^{d-i} \tilde{G'}_i(s,d,v). \tag{23}$$

Once G' and E' are estimated, Equation 10 gives the time speed-up of EPEIDA* over IDA*. □

It is of interest to note that we can also estimate the number of nodes whose generation EPEIDA* will save compared to IDA* at each particular level i:

$$\sum_{v=d-i+1}^{+\infty} \tilde{N}_i(s,d,v). \tag{24}$$

In this summation, v starts at $d - i + 1$ because only nodes with heuristic values in the range $[0 \ldots d - i]$ will be expanded at level i.

Appendix C. Algorithmic Enhancements for Solving MAPF

In this section we describe three MAPF-specific algorithmic enhancements to A*. Two of them, the *Independence Detection* (ID) and the *Operator Decomposition A** (ODA*) are due to Standley (2010). The *Partial Expansion ODA** (PEODA*) is our new hybrid between ODA* and PEA*.

C.1 The Independence Detection (ID)

Two groups of agents are called *independent* if there is an optimal solution for each group such that the two solutions do not conflict. The basic idea of *Independence Detection* (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. Then, all agents in the conflicting groups are unified into a new group, i.e. the two groups are *merged*. 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 observed that since the problem is exponential in k, the A*-search of the largest group dominates the running time of solving the entire problem, as all other searches involve smaller groups (for more details on ID, see Standley, 2010).

Note that ID can be combined with any optimal MAPF solver. In the paper of Standley (2010), ID is combined with ODA*. Our approach combines ID and EPEA* with an OSF based on additive abstractions. In fact, we use the feedback received from ID to determine which variables are included in each of the additive abstractions as will be explained below.

C.2 The Operator Decomposition (OD)

Standley (2010) introduced *Operator Decomposition* (OD) which reduces the number of surplus nodes generated for MAPF as follows. 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 nodes.

C.3 Partial Expansion ODA*

We introduce *Partial Expansion ODA** (PEODA*), a hybrid between ODA* and PEA*. This variant operates similarly to ODA* with one exception. When PEODA* generates intermediate children n_c of n, it puts n_c into OPEN only if $f(n_c) = F(n_a)$, where n_a is the standard node ancestor of n_c. In particular, if n is a standard node, then n_a is n.

Appendix D. Enhancements of the OSF Based on Additive PDBs

In this section, we present four enhancements to the OSF described in Section 8.3. Each enhancement is presented in a separate subsection.

D.1 Avoiding Linear Search for $i = k$

The most obvious enhancement of FindAbsOpSets is that, at the last level of recursion ($i = k$), we can use the fact that Δ-PDB$_k$ is ordered by $\Delta f(a_c)$ to quickly locate (e.g. by binary search) the $\Delta f(a_c)$-values that complete sum to the required $\Delta f(n_c)$.

D.2 Obtaining More Cut-offs in Line 5

We note that it is possible to obtain a stronger condition for the cut-off in line 5 of Procedure 3. To do this, at the first expansion of n, we compute and store with n in OPEN the array of k (the number of abstractions) elements:

$$smallSums[i] = \sum_{l=i+1}^{k} \Delta f(\sigma_{l1}[\phi_l(n)])$$

Intuitively, $smallSums[i]$ is the sum of the smallest entries in Δ-PDBs for n in the abstractions $i+1$ through to k. We note that a cut-off can be performed whenever $sum + smallSums[i] > \Delta f(n_c)$ holds. Intuitively, this means that sum is provably too large, since even choosing the smallest entries for the remaining abstractions would result in exceeding the required $\Delta f(n_c)$. Whenever such a cut-off occurs, $F_{next}(n)$ is set to the minimum between the current value of $F_{next}(n)$ and $sum + smallSums[i]$.

D.3 Obtaining and Additional Cut-off

Just as we can obtain a cut-off when sum is provably too large, we can obtain a cut-off when sum is provably too small. To do this, at the first expansion of n, we compute and store with n in OPEN the array of k (the number of abstractions) elements:

$$largeSums[i] = \sum_{l=i+1}^{k} \Delta f(\sigma_{l,nOps_l}[\phi_l(n)])$$

A cut-off can be performed whenever $sum + largeSums[i] < \Delta f(n_c)$ holds.

D.4 Per-Group Search

We already observed above that, once a choice of $\Delta f(a_c)$-value for each of the abstractions that results in the required $\Delta f(n_c)$ is found, we can take any one abstract operator from the groups resulting in those $\Delta f(a_c)$-values. Therefore, we can change the loop in line 2 to go through the possible groups instead of the possible abstract operators. For our example, two choices of groups of operators result in $\Delta f(n_c) = 8$:

$$\{(\Sigma_{12}, \Sigma_{22}, \Sigma_{33}), (\Sigma_{13}, \Sigma_{23}, \Sigma_{31})\}.$$

Once FindAbsOpSets returns all choices of groups, one per abstractions that result in the required $\Delta f(n_c)$ (line 14), a post-processing step would compute the corresponding choices of abstract operators. For example, this post-processing step would take the choice $(\Sigma_{12}, \Sigma_{22}, \Sigma_{33})$ and compute four choices of abstract operators:

$$\{(\sigma_{13}, \sigma_{22}, \sigma_{33}), (\sigma_{14}, \sigma_{22}, \sigma_{33}), (\sigma_{13}, \sigma_{22}, \sigma_{34}), (\sigma_{14}, \sigma_{22}, \sigma_{34})\}.$$

For a further enhancement, in domains such as MAPF, illegal operator pruning can be integrated with this post-processing step. In our example, if σ_{22} cannot be performed together with σ_{13}, then there is no need to consider the choice of operator of the third abstraction. In MAPF, this results in a further significant time speed-up.

References

Bouzy, B. (2013). Monte-carlo fork search for cooperative path-finding. In *IJCAI Workshop on Computer Games*.

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

Dechter, R., & Pearl, J. (1985). Generalized best-first search strategies and the optimality of A*. *Journal of the Association for Computing Machinery*, *32(3)*, 505–536.

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

Edelkamp, S. (2001). Planning with pattern databases. In *European Conference on Planning (ECP-01)*, pp. 13–24.

Edelkamp, S., & Korf, R. E. (1998). The branching factor of regular search spaces. In *AAAI*, pp. 299–304.

Felner, A., & Adler, A. (2005). Solving the 24-puzzle with instance dependent pattern databases. In *SARA*, pp. 248–260.

Felner, A., Goldenberg, M., Sharon, G., Stutervant, N., Stern, R., Beja, T., Schaeffer, J., & Holte, R. (2012). Partial-expansion A* with selective node generation. In *AAAI*.

Felner, A., Korf, R. E., & Hanan, S. (2004). Additive pattern database heuristics. *Journal of Artificial Intelligence Research (JAIR)*, *22*, 279–318.

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

Fikes, R. E., & Nilsson, N. J. (1971). Strips: A new approach to the application of theorem proving to problem solving. Tech. rep. 43R, AI Center, SRI International.

Ghosh, S., Mahanti, A., & Nau, D. S. (1994). ITS: An efficient limited-memory heuristic tree search algorithm. In *AAAI*, pp. 1353–1358.

Goldenberg, M., Felner, A., Stern, R., & Schaeffer, J. (2012). A* variants for optimal multi-agent pathfinding. In *Workshop on Multiagent Pathfinding*.

Goldenberg, M., Felner, A., Stutervant, N., Holte, R., & Schaeffer, J. (2013). Optimal-generation variants of EPEA*. In *International Symposium on Combinatorial Search (SoCS)*.

Helmert, M. (2006). The fast downward planning system.. *Journal of Artificial Intelligence Research (JAIR)*, *26*, 191–246.

Helmert, M. (2010). Landmark heuristics for the pancake problem. In *International Symposium on Combinatorial Search (SoCS)*, pp. 745–750.

Helmert, M., Haslum, P., & Hoffmann, J. (2007). Flexible abstraction heuristics for optimal sequential planning. In *ICAPS*, pp. 176–183.

Hoffmann, J., & Nebel, B. (2001). The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research (JAIR)*, *14*, 253–302.

Holte, R. C., Perez, M. B., Zimmer, R. M., & MacDonald, A. J. (1996). Hierarchical A*: Searching abstraction hierarchies efficiently. In *AAAI*, pp. 530–535.

Jensen, R. M., Hansen, E. A., Richards, S., & Zhou, R. (2006). Memory-efficient symbolic heuristic search. In *ICAPS*, pp. 304–313.

Jensen, R. M., Bryant, R. E., & Veloso, M. M. (2002). SetA*: An Efficient BDD-Based Heuristic Search Algorithm. In *AAAI/IAAI*, pp. 668–673.

Karpas, E., & Domshlak, C. (2009). Cost-optimal planning with landmarks. In *IJCAI*, pp. 1728–1733.

Korf, R. E. (1985). Depth-first iterative-deepening: An optimal admissible treesearch. *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*, pp. 700–705.

Larsen, B. J., Burns, E., Ruml, W., & Holte, R. (2010). Searching without a heuristic: Efficient use of abstraction. In *AAAI*.

Richter, S., & Helmert, M. (2009). Preferred operators and deferred evaluation in satisficing planning.. In *ICAPS*, pp. 273–280.

Röger, G., & Helmert, M. (2012). Non-optimal multi-agent pathfinding is solved (since 1984). In *International Symposium on Combinatorial Search (SoCS)*, pp. 1–5.

Russell, S. J. (1992). Efficient memory-bounded search methods. In *ECAI-92*.

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

Sharon, G., Stern, R., Goldenberg, M., & Felner, A. (2012). Meta-agent conflict-based search for optimal multi-agent path finding. In *International Symposium on Combinatorial Search (SoCS)*.

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

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

Yang, F., Culberson, J., Holte, R. C., Zahavi, U., & Felner, A. (2008). A general theory of additive state space abstractions. *Journal of Artificial Intelligence Research (JAIR)*, *32*, 631–662.

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

Zahavi, U., Felner, A., Burch, N., & Holte, R. C. (2010). Predicting the performance of IDA* (with BPMX) with conditional distributions. *Journal of Artificial Intelligence Research (JAIR)*, *37*, 41–83.

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

Zahavi, U., Felner, A., Schaeffer, J., & Sturtevant, N. R. (2007). Inconsistent heuristics. In *AAAI*, pp. 1211–1216.

Zhou, R., & Hansen, E. (2004). Space-efficient memory-based heuristics. In *AAAI*, pp. 677–682.

Zhou, R., & Hansen, E. A. (2002). Memory-bounded A* graph search. In *Florida Artificial Intelligence Research Society (FLAIRS-02)*, pp. 203–209.