A General Additive Search Abstraction

Fan Yang FYANG@CS.UALBERTA.CA
Joseph Culberson JOE@CS.UALBERTA.CA
Robert Holte HOLTE@CS.UALBERTA.CA
Computing Science Department, University of Alberta
Edmonton, Alberta T6G 2E8 Canada

Abstract

The effectiveness of heuristics search is influnced by the accuracy of the heuristic values. State space abstractions have been proved to be effective for generating admissible heuristics. In this paper, A general definition for abstractions is given. As a demonstration we apply additive abstractions to the domain of the 17 pancake puzzle. Experimental results show that using heurisitcs defined by additive abstractions combined with state-of-the-art techniques, the number of nodes generated for the 17 pancake puzzle is reduced by over 2000-fold, compared to the best result to date. Comparative results are also given to show a new approach to enhance additive heuristic values by checking for infeasibility.

1. Introduction

In this paper, we present a formal definition for space abstractions based on graph homomorphisms. This definition provides a generic methodology for generating admissible and consistent heuristics on a wide variety of problems in combinarial search and optimization. It is sufficiently general to cover many previous definitions of abstractions that generate admissible heuristics, including patterns (Culberson & Schaeffer, 1994, 1998; Felner, Korf, & Hanan, 2004), domain abstractions (Hernádvölgyi & Holte, 2000) and variable projections (Edelkamp, 2001).

Our definition also covers some properties that are not covered by previous definitions. One property is that to be additive, previous definitions need to partition state variables into disjoint groups, while by our definition, additive abstractions can be independent of state variables. Hence our definition is more general than many previous definitions. In addition, we also define infeasibility in additive abstractions and introduce a new approach to enhance additive heuristic values by checking for infeasibility.

As a demonstration we apply additive abstractions to the domain of the 17 pancake puzzle. Our experimental results show that using heurisitcs defined by additive abstractions combined with state-of-the-art techniques, the number of nodes generated for the 17 pancake puzzle is reduced by over 2000-fold, compared to the best result to date.

The remainder of the paper is organized as follows. An overview of previous methods to define and use abstractions is given in Section 1. Section 2 presents formal general definitions for abstractions. Section 3 shows how to apply additive abstractions to the domain of the pancake puzzle. Section 4 explores a new method to enhance additive heuristic values by checking for infeasibility.

1.1 Heuristic Search

Heuristic search algorithms such as IDA* (Korf, 1985) visit states guided by the cost function $f(n) = g(n) + h(n)$, where $g(n)$ is the actual distance from the initial state to the current state n. To guarantee finding optimal paths to the goal, $h(n)$ should be an admissible heurisitc function estimating the cost from n to a goal state. Therefore the effectiveness of heuristic search is influenced by the accuracy of the heuristic values.

1.2 Heuristics defined by abstractions

State space abstractions are effective to generate admissible heuristics. Figure 1 presents the method to define an abstract space for the 3×3 sliding tile puzzle. Considering only the blank tile's position, we create an abstract space consisting of nine abstract states. We assume that the abstract goal is the abstract state with the blank tile on the top-left position. For each abstract state, we count the minimum total moves starting from this abstract state to the abstract goal. For example, in Figure 1, it takes at least 2 moves for abstract state e to reach the abstract goal.

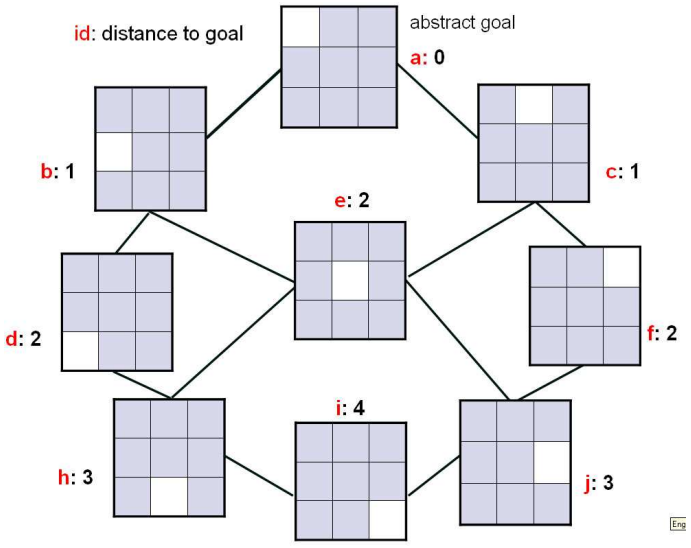

Figure 1: The abstract space of the 3×3 puzzle

We may look up the heuristic values from different abstractions and take the maximum value over all such lookups. Take a state of the 3×3 sliding tile puzzle for instance. We compute the heuristic value for state A as shown in the top-left part of Figure 2. First we define two abstractions. Then we count the minimum total moves from one abstract state to the corresponding abstract goal. The first abstraction is defined by only considering the positions of blank tile, tiles 1, 3, 5 and 7. The second abstraction is defined by considering the postitions of blank tile as well as tiles 2,4,6 and 8. As shown in Figure 2, we use arrows to depict the moves of the blank. Abstract states A1 and A2 are corresponding abstract states for state A in two abstract spaces. Likewise, abstract goals g1 and g2 are two abstract goals in these two abstract spaces. We compute that the minimum number of total moves

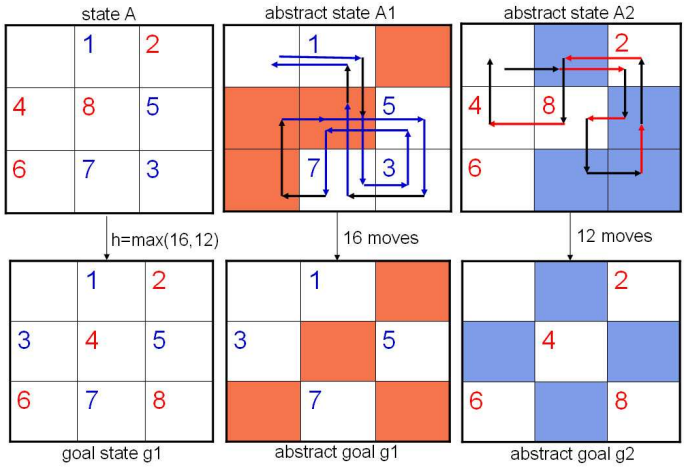

Figure 2: Get the maximum heuristic value for state A. It takes at least 16 moves from abstract state A1 to abstract goal g1 and it takes at least 12 moves from abstract state A2 to abstract goal g2. So the maximum heuristic value h=max(16,12)=16.

from A1 to g1 is 16 and that from A2 to g2 is 12. Therefore, the heuristic value of state A is 16, the maximum of these two heuristic values.

1.3 Additive Abstractions

Instead of taking the maximum of different heuristic values, Edelkamp (2001) instroduced additivity. He presented the definition of disjoint pattern databases and claimed that disjoint pattern databases add estimates according to different abstractions such that the accumulated estimates still provide a lower bound heuristic. Edelkamp also studied the effect of pattern databases in the context of deterministic planning. Korf and Felner (2002) went into more detail about the additive heuristics and explored its applications in the domains of combinatorial puzzles. They partition the problems into disjoint sub-problems, so that the costs of solving each sub-problem can be added together without overestimating the cost of solving the original problem. Korf and Felner (2002), Felner et al. (2004) successfully applied this technique to different domains, such as the Sliding Tile Puzzle, Towers of Hanoi Problem (4-peg Towers), Vertex Cover and so on.

To understand the concepts introduced so far, let us still consider state A of 3×3 sliding puzzle as shown in Figure 3. Suppose that to create two additive abstractions for the 3×3 sliding tile puzzle, we partition the tiles into two disjoint groups. Group I includes distinguished tiles 1,3,5,7 and group II includes distinguished tiles 2,4,6,8. Then only considering the blank tile and the distinguished tiles in each group, we solve each subproblem. If a move involves distinguished tiles, we call it a *distinguished move*. Otherwise, we call the move a *don't-care move*. Instead of counting all moves from an abstract space to the abstract goal, we only count the distinguished moves to compute additive heuristics. To solve the subproblem shown in the left part of Figure 3, we need at least 5 moves of distinguished

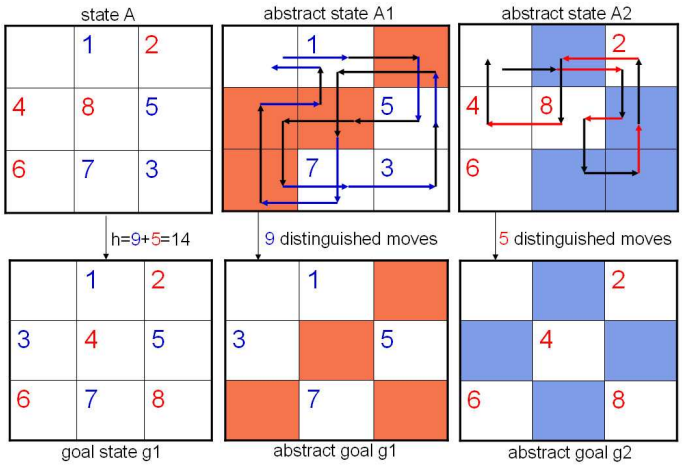

Figure 3: Get the additive heuristic value for state A. It takes at least 9 distinguished moves from abstract state A1 to abstract goal g1 and it takes at least 5 moves from abstract state A2 to abstract goal g2. So the additive heuristic value h=9+5=14.

tiles(tiles in group I). Likewise, we need at least 9 moves of distinguished tiles(tiles in group II) to solve subproblem in the right part of Figure 3.

Since we only count the minimum number of moves of the distinguished tiles in each disjoint goup, we may add the values to get an additive admissible heuristic.

One limitation of additive heuristics defined in (Korf & Felner, 2002; Felner et al., 2004) is that they need to divide state variables into disjoint groups and the heuristic value is computed by counting the minimum number of moves of the tiles in each group that are required to get those tiles to their goal positions. Therefore, this technique cannot be applied to other domains in which the operators can move more than one tile at each time.

Take the pancake puzzle for example. In the N-pancake puzzle, a state is a permutation of N tokens$(0, 1, ..., N-1)$. A state has $N-1$ successors, with the kth successor formed by reversing the order of the first $k+1$ positions of the permutation $(1 \leq k \leq N)$. For example, in the four pancake puzzle shown in Figure 4, a state s has three successors, which are formed by reversing the order of the first two tiles, first three tiles and all four tiles, respectively. As each operator moves at least two tiles, we cannot achieve additive abstractions by previous definitions.

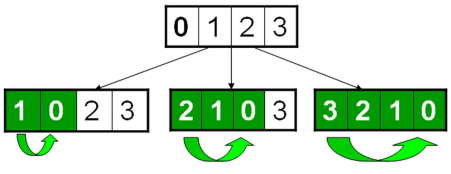

Figure 4: In the four pancake puzzle, each state has three successors

2. General Definitions

In this section, we give definitions and lemmas, and we discuss their meanings. The proofs are given in Appendix A

2.1 State Space

A *state space* is a weighted directed graph $\mathcal{S} = \langle T, \Pi, C \rangle$ where T is a (usually finite) set of states, $\Pi \subseteq T \times T$ is a set of directed edges (ordered pairs of states) and $C : \Pi \longrightarrow \mathcal{N} = \{0, 1, 2, 3, \dots\}$ is the edge cost function.[1] In typical practice, $\mathcal{S}$ is implicitly defined by some combinatorial search problem. For example, T may be defined by the set of all possible assignments to a set of state variables, and Π and C will depend on the operations on the variable sets.

We define a *path* from t to g, where $t, g \in T$, by $\vec{p} = \vec{p}(t, g) = \langle \pi^1, \dots, \pi^n \rangle, \pi^j \in \Pi$ where $\pi^j = (t^j, t^{j+1}), 1 \leq j \leq (n-1)$ and $t^1 = t, t^{n+1} = g$. Note the use of superscripts to distinguished states and edges within a state space. We say the *length* of $\vec{p}$ is the number of edges n and define the *cost* $C(\vec{p}) = \sum_{j=1}^{n} C(\pi^j)$.

An *instance* of a *path problem* is a triple $\langle \mathcal{S}, t, g \rangle$, where $t, g \in T$ with the objective of finding the *minimum cost* $\mathrm{OPT}(t, g) = \min\{C(\vec{p}) \mid \vec{p} = \vec{p}(t, g)\}$, or in some cases finding $\vec{p} = \vec{p}(t, g)$ such that $C(\vec{p}) = \mathrm{OPT}(t, g)$.

Note that this paper is restricted to path problems, wherein the start and goal states are explicitly stated, and we only need to optimize the path between them. We expect that the results may be extended to more general combinatorial search problems where the goal g is not explicitly given, but instead must be recognized by satisfying some implicit conditions.

2.2 Abstractions

An *Abstraction System* is a triple $\langle \mathcal{S}, \mathbf{A}, \mathbf{\Psi} \rangle$ where $\mathcal{S} = \langle T, \Pi, C \rangle$ is a state space, $\mathbf{A} = \{\mathcal{A}_i, 1 \leq i \leq k\}$ is a set of abstract spaces defined below, and $\mathbf{\Psi} = \{\psi_1, \dots, \psi_k\}$ is a set of graph homomorphisms[2] extended to include edge cost $\psi_i : \mathcal{S} \longrightarrow \mathcal{A}_i$.

An *abstract space* is a weighted directed graph defined by a four-tuple $\mathcal{A}_i = \langle T_i, \Pi_i, C_i, R_i \rangle$. T_i, the set of abstract states, is usually equal to $\psi_i(T)$, but it can be a superset. Likewise, Π_i, the set of abstract edges, is usually equal to $\psi_i(\Pi) = \{(\psi_i(t^1), \psi_i(t^2)) \mid (t^1, t^2) \in \Pi\}$ but it can be a superset. We will frequently use a shorthand notation $t_i^j = \psi_i(t^j)$ for the abstract state in T_i corresponding to $t^j \in T$, and $\pi_i^j = \psi_i(\pi^j)$ for the abstract edge in Π_i corresponding to $\pi^j \in \Pi$.

An abstract space has two costs associated with each $\pi_i \in \Pi_i$, the *primary cost* $C_i : \Pi_i \longrightarrow \mathcal{N}$ and a *residual cost* $R_i : \Pi_i \longrightarrow \mathcal{N}$. To guarantee admissibility, we require the following condition to hold for all $1 \leq i \leq k$

$$\forall \pi \in \Pi, C(\pi) \geq C_i(\pi_i) + R_i(\pi_i) \tag{1}$$

1. Real-valued edge costs are permissible provided conditions are imposed to ensure that the set of paths connecting any given pair of states has a well-defined minimum cost.
2. A graph homomorphism f from a graph $G = (V, E)$ to a graph $G' = (V', E')$, written $f : G \to G'$ is a mapping $f : V \to V'$ from the vertex set of G to the vertex set of G' such that $(f(u), f(v)) \in E'$ whenever $(u, v) \in E$

As shown in Figure 2 and Figure 3, there are two types of edges in the abstract space, representing distinguished moves and don't-care moves. Here we generalized the edge costs according to these two types of moves. Note that previous abstraction definitions often ignored the cost of the don't-care moves. The reason for considering the cost of don't-care moves will become apparent when we talk about eliminating infeasible costs in Section 4.

Likewise, each abstract path $\vec{p}_i = \langle \pi_i^1, \ldots, \pi_i^n \rangle$ in $\mathcal{A}_i$ has a primary and residual cost: $C_i(\vec{p}_i) = \sum_{j=1}^{n} C_i(\pi_i^j)$, and $R_i(\vec{p}_i) = \sum_{j=1}^{n} R_i(\pi_i^j)$.

Because $\Pi_i \supseteq \psi_i(\Pi)$ the following is ensured.

Lemma 2.1 *For any path $\vec{p} = \vec{p}(t, g)$ in $\mathcal{S}$, there is a corresponding abstract path $\psi_i(\vec{p})$ from t_i to g_i in $\mathcal{A}_i$. Also, $C(\vec{p}) \geq C_i(\psi_i(\vec{p})) + R_i(\psi_i(\vec{p}))$.*

For example, let us consider state A shown in Figure 2. Any solution path for state A in the original space is also a solution path for pattern A1 and pattern A2 in the abstract spaces. By holding the condition (1) for all edges, the cost of the solution path in the original space is larger than or equal to the sum of the primary cost and the residual cost of the abstract solution path in the abstract space.

Define the *optimal abstract cost* from abstract state t_i to abstract state g_i by $\mathrm{OPT}_i(t_i, g_i) = \min\{C_i(\vec{p}_i) + R_i(\vec{p}_i) \mid \vec{p}_i \text{ is an abstract path in } \mathcal{A}_i \text{ from } t_i \text{ to } g_i\}$. Note that in this definition, $\vec{p}_i$ is not required to be the image, $\psi_i(\vec{p})$, of a path, $\vec{p} = \vec{p}(t, g)$, in $\mathcal{S}$. Given a goal state g, the heuristic of state t derived from abstract space $\mathcal{A}_i$ is $h_i(t) = \mathrm{OPT}_i(t_i, g_i)$. The following two lemmas state that each individual abstraction generates admissible and consistent heuristics, respectively.

Lemma 2.2 $\forall t, g \in T, \mathrm{OPT}(t, g) \geq \mathrm{OPT}_i(t_i, g_i)$.

Lemma 2.3 $\forall t^1, t^2 \in T, \mathrm{OPT}_i(t_i^1, g_i) \leq \mathrm{OPT}(t^1, t^2) + \mathrm{OPT}_i(t_i^2, g_i)$.

Proof:

By the definition of $\mathrm{OPT}_i(t_i, g_i)$, $\mathrm{OPT}_i(t_i^1, g_i) \leq \mathrm{OPT}_i(t_i^1, t_i^2) + \mathrm{OPT}_i(t_i^2, g_i)$.

By lemma 2.2, $\mathrm{OPT}(t^1, t^2) \geq \mathrm{OPT}_i(t_i^1, t_i^2)$.

Hence, $\mathrm{OPT}_i(t_i^1, g_i) \leq \mathrm{OPT}i(t^1, t^2) + \mathrm{OPT}_i(t_i^2, g_i)$. ■

2.2.1 Relations with Prevous definitions of abstractions

The use of abstractions to guide search is a well-known concept in Artificial Intelligence. In fact, the earliest attempts to solve problems faster by relaxing the original problem definition were used to solve planning problems. Prieditis (1993) extended and unified these early definitions of abstractions that generate admissible heuristics. Our definition is similiar to the work of Prieditis (1993). And it is sufficiently general to cover many previous defintions of abstractions that generate admissible heuristics, including patterns (Culberson & Schaeffer, 1994, 1998; Felner et al., 2004), domain abstractions (Hernádvölgyi & Holte, 2000) and variable projections (Edelkamp, 2001). In the following, we present previous work on abstractions. In addition, we take simple examples to explain how our definitions cover domain abstraction and variable pojection.

Culberson and Schaeffer (1994, 1998) introduced a *pattern* as the partial specification of a state and claimed that the cost of the pattern induced by the state is a lower bound

on the cost of the state. A pattern database is a heuristic function in the form of a lookup table which stores the cost of optimal solutions for instances of subproblems. Using the corner and fringe databases of the 15 sliding tile puzzle, they demonstrated a 1038-fold improvement in the number of nodes generated and reduced the running time by a factor of twelve, compared to the manhattan distance heuristic. Pattern databases were also the key breakthrough that enabled Korf (1997) to solve Rubik's Cube optimally.

PDBs are large and time-consuming to build, therefore they are best suited to situations where many problem instances are to be solved. There are circumstances in which the cost of building an entire PDB cannot reasonably be amortized over a large number of problem instances. In this case, it is better to minimize the time and space overhead of using a PDB by computing only those entries of the PDB that are actually needed to solve a given problem instance. The idea of on-demand calculation of PDB entries by hierarchical heuristic search was introduce by Holte, Perez, Zimmer, and MacDonald (1996).

Hernádvölgyi and Holte (2000) first defined domain abstraction which extends the notion of *pattern* in the pattern database work (Culberson & Schaeffer, 1994, 1998). The key property of domain abstraction is that they are homomorphisms and therefore the distance between two states in the original space is always greater than or equal to the distance between the corresponding abstract states. Take the four pancake puzzle for example. The abstraction is defined as shown in Figure 5. While the original state space of the four pancake puzzle has twenty-four states, the abstract state space has only six abstract states: (0,0,2,2), (0,2,0,2), (0,2,2,0), (2,0,0,2), (2,0,2,0), (2,2,0,0), Likewise, our definitions

Domain=	0	1	2	3
Abstract=	0	0	2	2

Figure 5: Domain Abstraction for the four pancake puzzle

of abstractions are also defined as graph homomorphisms. So our definitions cover the domain abstraction.

The abstraction defined by Edelkamp (2001) is achieved by projecting the state representation (Variable Projection). Edelkamp applied previous progress of heuristic search in finding optimal solutions to planning problems by devising an automatic abstraction sheme to create admissible domain-independent memory-based heuristics. Variable Projection is done by ignoring the value of some variable in the state representation and removing all references to the variable in the operators.

Edelkamp applied this method to planning problems, since planning states are interpreted as conjuncts of atoms whose values are either true or false. For the sake of simplicity, let's take the 4 Arrow Puzzle(Korf, 1980) for example. Assume that in the state space of the 4 Arrow Puzzle, each state represents the directions of 4 arrows and each arrow has two directions, up and down. In the state description, 0 represents up and 1 represents down. For example, the goal state $(\uparrow, \uparrow, \uparrow, \uparrow)$ is represented by $(0, 0, 0, 0)$ and the start state $(\downarrow, \downarrow, \uparrow, \downarrow)$ is represented by $(1, 1, 0, 1)$. We define four operators as follows.

operator A: $(v_1, v_2, v_3, v_4) \rightarrow (v_1, \overline{v_2}, v_3, v_4)$
operator B: $(v_1, v_2, v_3, v_4) \rightarrow (\overline{v_1}, \overline{v_2}, v_3, v_4)$
operator C: $(v_1, v_2, v_3, v_4) \rightarrow (v_1, \overline{v_2}, \overline{v_3}, v_4)$
operator D: $(v_1, v_2, v_3, v_4) \rightarrow (v_1, v_2, \overline{v_3}, \overline{v_4})$

In the original state space (shown in Figure 6), Operator A flips Arrow 2, Operator B flips Arrow 1 and Arrow 2, Operator C flips Arrow 2 and Arrow 3, and Operator D flips Arrow 3 and Arrow 4.

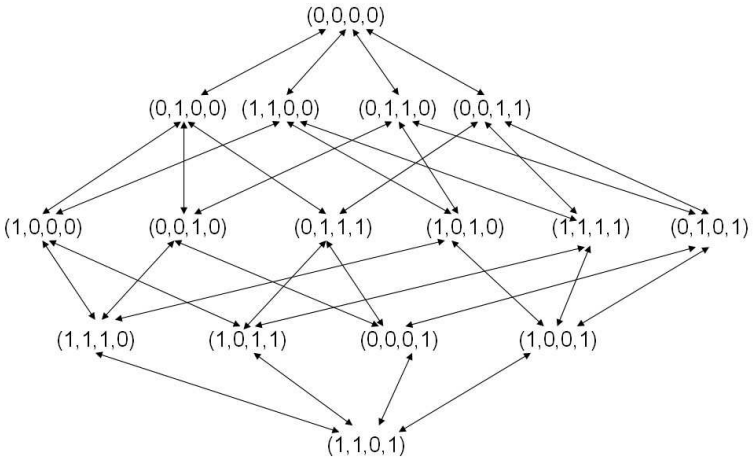

Figure 6: The original space of the 4 Arrow Puzzle

In our example, variable projection is done by ignoring Arrow 2 and removing all references to Arrow 2. Therefore, our abstract goal is (0,0,0) and the start state is (1,0,1). Abstract operators are defined as follows.

abstract operator A': $(v_1, v_3, v_4) \rightarrow (v_1, v_3, v_4)$

abstract operator B': $(v_1, v_3, v_4) \rightarrow (\overline{v_1}, v_3, v_4)$

abstract operator C': $(v_1, v_3, v_4) \rightarrow (v_1, \overline{v_3}, v_4)$

abstract operator D': $(v_1, v_3, v_4) \rightarrow (v_1, \overline{v_3}, \overline{v_4})$

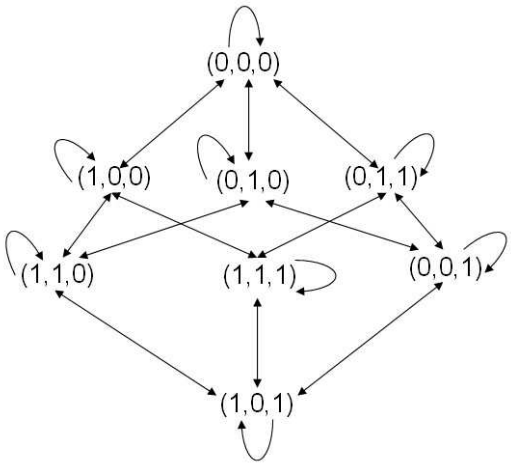

Figure 7: The abstract space achieved by variable projection

This means that in the abstract space (shown in Figure 7), Operator A' does not flip any arrow. Operator B' flips Arrow 1, Operator C' flips Arrow 3, and Operator D' flips Arrow 3 and Arrow 4.

Note that as shown in Figure 6 and Figure 7, the abstraction is a graph homomorphism from the original state space to the abstract space. Therefore, it is covered by our definitions for abstractions.

2.3 Additive Abstractions

We say that $\mathbf{\Psi}$ is *additive* if $\forall \pi \in \Pi, C(\pi) \geq \sum_{i=1}^{k} C_i(\pi_i)$.

Define $C_i^*(t_i, g_i) = \min\{C_i(\vec{p_i}) \mid \vec{p_i}$ is an abstract path from t_i to $g_i\}$, the minimum primary cost of an abstract path from t_i to g_i. Given a goal state g, the heuristic of state t defined by k additive abstractions is $h(t) = \sum_{i=1}^{k} C_i^*(t_i, g_i)$.

To understand the definition of $C_i^*(t_i, g_i)$, let us consider state A shown in Figure 3. To solve the subproblem shown in the left part of Figure 3, we need at least 9 distinguished moves. Likewise, we need at least 5 distinguished moves to solve subproblem in the right part of Figure 3. Therefore $C_1^* = 9$, $C_2^* = 5$, and $h(A) = 9 + 5 = 14$.

Lemma 2.4 *If* $\mathbf{\Psi}$ *is additive, then* $\mathrm{OPT}(t, g) \geq \sum_{i=1}^{k} C_i^*(t_i, g_i)$.

Proof: Assume that $\mathrm{OPT}(t, g) = C(\vec{p})$, where $\vec{p} = \vec{p}(t, g) = \langle \pi^1, \ldots, \pi^n \rangle$. Therefore, $\mathrm{OPT}(t, g) = \sum_{j=1}^{n} C(\pi^j)$. Since $\mathbf{\Psi}$ is additive, it follows that
$\sum_{j=1}^{n} C(\pi^j) \geq \sum_{j=1}^{n} \sum_{i=1}^{k} C_i(\pi_i^j) = \sum_{i=1}^{k} \sum_{j=1}^{n} C_i(\pi_i^j) \geq \sum_{i=1}^{k} C_i^*(t_i, g_i)$ ■

Lemma 2.4 proves that the additive heuristic is admissible. And the following lemma indicates that the additive heuristic is consistent.

Lemma 2.5 $\forall t^1, t^2 \in T$, $\sum_{i=1}^{k} C_i^*(t_i^1, g_i) \leq \mathrm{OPT}(t^1, t^2) + \sum_{i=1}^{k} C_i^*(t_i^1, g_i)$.

Proof: By the definition of $C_i^*(t_i, g_i)$, $\forall t^1, t^2 \in T, C_i^*(t_i^1, g_i) \leq C_i^*(t_i^1, t_i^2) + C_i^*(t_i^2, g_i)$.
It follows that $\sum_{i=1}^{k} C_i^*(t_i^1, g_i) \leq \sum_{i=1}^{k} C_i^*(t_i^1, t_i^2) + \sum_{i=1}^{k} C_i^*(t_i^2, g_i)$.
Since $\mathbf{\Psi}$ is additive, by Lemma 2.4, $\mathrm{OPT}(t^1, t^2) \geq \sum_{i=1}^{k} C_i^*(t_i^1, t_i^2)$.
Thus, $\forall t^1, t^2 \in T$, $\sum_{i=1}^{k} C_i^*(t_i^1, g_i) \leq \mathrm{OPT}(t^1, t^2) + \sum_{i=1}^{k} C_i^*(t_i^2, g_i)$. ■

2.3.1 Relations with previous definitions of additivity

Most previous methods for additivity (Edelkamp, 2001; Korf & Felner, 2002; Felner et al., 2004) are to partition the state variables into disjoint groups to create additive abstractions. We can generalize these methods by our definition as follows.

Suppose some variable subset X' can be identified such that each edge changes at most one variable in X'. Then given a set of abstractions ψ_i, we partition X' into P_i. In the i^{th} abstract state space, if an edge corresponds to a change of a variable in P_i, the primary cost of this edge is C, where C is the minimum cost of its preimages in the original state space; otherwise, the primary cost of this edge is zero. The way to see this method application on the 15 sliding tile puzzle is to give all variables not in the partition P_i the domain $\{*\}$. This means that the state space only considers the values of the identified variables. Once we partition the state variables into disjoint groups, for each abstract edge π_i in Π_i, we define

the primary cost and the residual cost as follows.

$$C_i(\pi_i) = \begin{cases} min\{C(\pi) : \psi_i(\pi) = \pi_i\}, & \text{If } \pi_i \text{ correponds to a change} \\ & \text{of a variable in } P_i \\ 0, & \text{Otherwise} \end{cases}$$

Since state variables are partitioned into disjoint groups, $\forall \pi \in \Pi, C(\pi) \geq \sum_{i=1}^{k} C_i(\pi_i)$. Hence additive abstractions are defined.

As we have mentioned in Section 1.3, one limitation of additive heuristics defined in (Korf & Felner, 2002; Felner et al., 2004) is that this technique cannot be applied to other domains in which the operators can move more than one state variable at each time. While by our definition, we can define additive abstraction more freely for more domains, such as the domain of the pancake puzzle. In the next section, we will investigate the application to the pancake puzzle in detail.

3. Applications

According to our definition, the key point to apply additive abstractions lies in what form in the definition of the primary cost C_i. Based on this rule, we can define additive abstractions by several methods for different domains. As a demonstration, here we show how to apply additive abstractions to the domains of the TopSpin puzzle and the pancake puzzle. For the sake of simplicity, we assume that the cost of each edge in the original state space is one. In the following sections, we assume that a state t in T is represented by a vector of state variables $X_0, ..., X_{n-1}$. For example, in the pancake puzzle, there is a variable for each physical tile with a value that indicates the position occupied by that tile. Given ψ_i and a state $t = (t(0), ..., t(n-1))$ where $t(j)$ is the value assigned to the variable X_j, $t(j) \in D_j (0 \leq j \leq n-1)$ and D_j is the domain for varialbe X_j, an abstract state t_i=$(t_i(0), ..., t_i(n-1))$, where $t_i(j) \in \psi_i(D_j)$ and $0 \leq j \leq n-1$.

3.1 Cost Splitting

The generalized cost splitting method can be applied to any puzzle of this type, even if the operations are not of uniform size, for example the pancake puzzle. For an operation involving k state variables, each state variable changed is charged $1/k$ of the operation cost. Given a partition of state variables, in a given abstraction each distinguished state variable is charged $1/k$ of the operation cost when involved in an operation on k state variables. To avoid floating point errors, if the set of operation sizes is $\{k_1, ..., k_m\}$, then we compute L $=lcm\{k_1, ..., k_m\}$. Each operation then costs L and the cost of changing one state variable by an operation changing the values of k_i state variables is L/k_i. ... ?

3.2 Fixed Reference

The key idea of this method is as follows. For each abstract edge π_i in Π_i, we define fixed reference to some value in the domain. Then in the abstract space, we only count the cost of the edge that assigns the fixed value to an identified state variable, although each edge may correspond the change of many state variables. So we define the primary cost as follows.

$$C_i(\pi_i) = \begin{cases} 1, & \text{if the fixed value is assigned} \\ & \text{to an identified state variable.} \\ 0, & \text{Otherwise.} \end{cases}$$

And we require that the set of identified state variables are disjoint in all abstract state space. This condition guarantees that $\forall \pi \in \Pi, C(\pi) \geq \sum_{i=1}^{k} C_i(\pi_i)$.

3.2.1 Experimental Results on the Pancake puzzle

We have tried on the pancake puzzle to create additive PDBs by all ideas mentioned in preceding sections. Our theory proves that they are admissible additive heuristics, but unfortunately for the pancake puzzle domain most of them turn out to be poor heuristics, which are weaker than taking the maximum (using the same abstractions). We believe all these ideas are worth trying for other domains. But to our knowledge, the method of *Fixed Reference* is the most effective so far for the pancake puzzle.

In this section, using heuristics based on additive abstractions defined by *Fixed Reference*, we present the experimental results on the 17 pancake. We then create additive PDBs by the following steps.

1. Partition the state variables into k disjoint groups $P_1, ..., P_k$. In each abstract space, give all variables not in the partition P_i the domain $\{*\}$. This means that the state space only considers the values of the identified variables.
2. For each edge π_i in Π_i, we define the primary cost and the residual cost

$$C_i(\pi_i) = C_i(t_i^1, t_i^2) = \begin{cases} 1, & \text{If in } t_i^2, \text{ the value 0 is assigned} \\ & \text{to a distinguished state variable.} \\ 0, & \text{Otherwise} \end{cases}$$

$R_i(\pi_i) = 1 - C_i(\pi_i)$.

In the pancake puzzle, the value 0 represents the first position. We use the value 0 to define C and R because the first position has two special properties. First, any state variable can be assigned to be 0. Second, the value 0 is assigned to different state variables after applying any operator.

By the above definitions of C and R, for all $1 \leq i \leq k$, $\forall \pi \in \Pi, C(\pi) \geq C_i(\pi_i) + R_i(\pi_i)$ and $C(\pi) \geq \sum_{i=1}^{k} C_i(\pi_i)$. Therefore, our theory proves that they are admissible additive heuristics.

Although our theory is not specific to PDBs, in our experiments we will use PDBs as an efficient way to implement the calculation of abstract distances. We compare our implementation of additive pattern databases for the 17 pancake puzzle. Note that the best results were reported in (Felner, Zahavi, Holte, & Schaeffer, 2006), where the average number of nodes generated over 30 instances of the 17 pancake puzzle is 223,305,375. Combined with the techniques of dual lookups[3] and dual search[4], our results are over 2000-fold improvement over the best results reported in (Felner et al., 2006).

3. Dual lookups means that for any given state s, the heuristic of s, $H(s) = max(H(s), H(s^d))$, where s^d is the dual state of s.
4. Dual search means that the search algorithm might decide to jump to the regular side or to the dual side, depending on the larger heuristic values of s and s^d. In (Felner et al., 2006), they call this the *jump if larger*(JIL) policy.

We apply a compact mapping for each pattern database. For example, a pattern of 4 tiles of the 17 pancake puzzle needs an array of size $17 \times 16 \times 15 \times 14$. As described in (Felner et al., 2004), this compact mapping doesn't waste space, but computing the indices is more complex and therefore consumes more time.

Our search algorithms are IDA^* or $DIDA^*$ written in C. To propagate inconsistent heuristic values during search, our search algorithm also applies the bidirectional path max(BPMX) method, which is introduced in the paper (Felner, Zahavi, Schaeffer, & Holte, 2005).

Table 1 shows comparative results over 1000 random instances of the 17-pancake problems, whose average solution length is 15.77. The **Algorithms** column indicates the heuristic search algorithm. The **PDB** column shows the pattern databases used to generate heuristics. The **H** column is the average heuristic value of 1000 initial instances. The **Nodes** column shows the average number of nodes generated to find an optimal solution. The **Sec** column gives the average amount of CPU time that was needed to solve a problem on a machine with AMD Athlon(tm) 64 Processor 3700+ with 2.4G clock rate and 1G memory. The **Memory** column indicates the size of each set of Pattern Databases.

Algorithms	PDB	H	Nodes	Sec	Memory
$DIDA^*$	Single7	10.858	124,198,462	37.713	98,017,920
IDA^*	ADD4-4-4-5	12.533	14,610,039	4.302	913,920
	ADD5-6-6	13.393	1,064,108	0.342	18,564,000
	ADD3-7-7	13.423	1,061,383	0.383	196,039,920

Table 1: 17 pancake puzzle results

The first data row is the result based on a PDB on the seven rightmost tiles. This is the best result using previous state-of-the-art techniques.

The last three rows are results by standard IDA^* using heuristics from three sets of additive PDBs whose partitionings are shown in Figure 8, respectively.

4-4-4-5 partitioning	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
5-6-6 partitioning	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
3-7-7 partitioning	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Figure 8: Different partitionings for the 17 pancake puzzle

It is clear that using our additive PDBs results in very significant reduction in nodes generated over using a single large pattern database. Note that additive PDB with 4-4-4-5 partitioning uses only one percent memory but reduces the number of nodes generated a factor of 10 over a single pattern database with state-of-the-art techniques. The most important thing is that since there are only $913,920$ entries in the databases, the time to compute ADD4-4-4-5 PDBs is much smaller compared to other PDBs. (Holte, Grajkowski,

& Tanner, 2005) mentioned that the disadvantages of using pattern databases is the cost of building and storing entire pattern databases. For ADD 4-4-4-5, these disadvantages disappear, to some extent, compared to other PDBs.

Algorithms	PDB	H	Nodes	Sec	Memory
IDA^*	ADD4-4-4-5	12.784	521,713	0.279	913,920
with dual lookups	ADD5-6-6	13.594	60,002	0.037	18,564,000
	ADD3-7-7	13.628	52,237	0.036	196,039,920
$DIDA^*$	ADD4-4-4-5	12.784	368,925	0.195	913,920
(dual lookups)	ADD5-6-6	13.594	44,618	0.028	18,564,000
(dual search)	ADD3-7-7	13.628	37,155	0.026	196,039,920

Table 2: Use DIDA* with additive PDBs to solve 1000 random instances of 17 pancake puzzle

In the following experiments as shown in Table 2 , we apply $DIDA^*$. During the search of $DIDA^*$, we either combine additive PDB with dual lookups, or combine additive PDB with dual lookups and dual search.

The first three rows are results using heuristics with dual lookups from three sets of additive PDBs whose partitionings are shown in Figure 8, respectively.

The last three rows are results using dual lookups and dual search using heurisitcs from additive PDBs of 4-4-4-5 partitioning, 5-6-6 partitioning and 3-7-7 partitioning, respectively.

Combining our additive PDBs with state-of-the-art techniques results in a more significant reductions in nodes generated. As shown in Table 2, IDA* with dual lookups from PDBs of ADD 5-6-6 uses only 1/5 size of memory, but it reduces the number of nodes generated by over 2000-fold, compared to the result of single large PDB shown in the first row of Table 1. Likewise, $DIDA^*$ with ADD 5-6-6 gives an over 1000-fold improvement over $DIDA^*$ with single large PDB(shown in the first row of Table 1), while the PDB size of ADD 5-6-6 is only one-fifth of that of the single PDB.

The Largest part of our simplest additive is 5, while the previous method used MAX of one 7 set according to preliminary tests (Felner et al., 2006). Furthermore, our 5 set is a subset of their 7, so this is the best of the five's and the other parts (which are actually 4's) are unlikely to improve on it much. But our 5 will clearly carry much less info than their 7. Thus, MAX over our partition will under perform the single 7, and so will certainly far under perform our additive.

Note that the memory reqirements increase as the partitions use larger patterns, but the improvement of the running time is not increased accordingly. For example, ADD 3-7-7 uses ten times size of memory to store the PDBs, but the running time is almost the same, compared to the results of ADD5-6-6. This is because there is little room left for an improvement over ADD5-6-6. Meanwhile, since we are using compact mapping funtion during the search, it takes more time to compute for larger PDB like ADD3-7-7.

4. Infeasible Additive Values

In this section, we describe a new approach to improve the quality of the heuristic values defined by additive abstractions. Basically, we improve the quality of additive heuristic values by identifying and increasing infeasible values.

What is an infeasible value? For a state t, the heuristic value h is infeasible if it can be proved that the cost of the solution for t cannot be h. Take the well-kown Sliding Tile Puzzle for an example, assuming that the manhattan heuristic value for state t is $MD(t)$. If $MD(t)$ is even, then any odd number is an infeasible value.

In the following sections, we use an example to illustrate what is an infeasible heuristic value defined by additive abstractions. Then we give the formal definition of the infeasible additive values. Finally we show the comparative results with/without checking for the infeasible values on the 15 sliding tile puzzle.

4.1 Simple Example

To detect infeasible additive values, we use C^* and R^*. As defined in Section 2.3, C^* is the minimum primary cost of an abstract path. Here we refer to R^* as the minimum residual cost among the abstract paths, each of which has primary cost of C^*. The formal definition of R^*will be given in Section 4.2.

Let us consider the example state A introduced earlier in Figure 3. We can compute values of C^* and R^* for two subproblems.

To solve the subproblem shown in the left part of Figure 3, we need at least 9 moves of the distinguished tiles(tiles in groug I) and at least 9 moves of non-distinguished tiles (tiles in group II). Likewise, to solve the subproblem shown in the right part of Figure 3, we need at least 5 moves of the distinguished tiles(tiles in groug II) and at least 7 moves of non-distinguished tiles (tiles in group I). Therefore, by definitions of C^* and R^*, $(C_1^*, R_1^*) = (9, 9), (C_2^*, R_2^*) = (5, 7)$.

Hence the additive heuristic value of state A is the sum of C_1^* and C_2^*, assuming that there is a solution path involving 9 moves of tiles in group I and 5 moves of tiles in group II. However, from the values of C_1^* and R_1^*, we note that if a path involves only 9 moves of tiles in group I, we need at least 9 moves of tiles in group II. Then there is not a solution path involving 9 moves of tiles in group I and only 5 moves of tiles in group II. Therefore, the additive heuristic value $\Sigma C^* = C_1^* + C_2^* = 9 + 5 = 14$ is an infeasible additive heuristic value.

4.2 Detecting Infeasible Additive Values

Let $\vec{P}_i(t_i, g_i) = \{$ abstract paths $\vec{p}_i$ from t_i to $g_i \mid C_i(\vec{p}_i) = C_i^*(t_i, g_i)\}$ be the set of abstract paths from t_i to g_i whose primary cost is minimal. Define the *conditional optimal residual cost* to be the minimum residual cost among the paths in $\vec{P}_i(t_i, g_i)$: $R_i^*(t_i, g_i) = \min\{R_i(\vec{p}_i) \mid \vec{p}_i \in \vec{P}_i(t_i, g_i)\}$.

Note that the value of $(C_i^*(t_i, g_i) + R_i^*(t_i, g_i))$ may or may not be equal to the optimal abstract cost $\mathrm{OPT}_i(t_i, g_i)$. In Figure 2 and Figure 3 for example, $\mathrm{OPT}_1(A1, g1) = 16$ and $C_1^*(A1, g1) + R_1^*(A1, g1) = 18$, while $C_2^*(A2, g2) + R_2^*(A2, g2) = \mathrm{OPT}_2(A2, g2) = 12$.

Lemma 4.1 *Given an additive* $\mathbf{\Psi}$*, and a path* $\vec{p} = \vec{p}(t, g)$ *with* $C(\vec{p}) = \sum_{i=1}^{k} C_i^*(t_i, g_i)$*, then, for all* $1 \leq h \leq k$*,* $C_h(\psi_h(\vec{p})) = C_h^*(t_h, g_h)$*.*

Proof: Suppose for a contradiction that there exists some i_1, such that $C_{i_1}(\psi_{i_1}(\vec{p})) > C_{i_1}^*(t_{i_1}, g_{i_1})$. Then because $C(\vec{p}) = \sum_{i=1}^{k} C_i^*(t_i, g_i)$, there must exist some i_2, such that $C_{i_2}(\psi_{i_2}(\vec{p})) < C_{i_2}^*(t_{i_2}, g_{i_2})$, which contradicts the definition of C_i^*. Therefore, such an i_1 does not exist and for all $1 \leq h \leq k$, $C_h(\psi_h(\vec{p})) = C_h^*(t_h, g_h)$. ■

Lemma 4.2 *Given an additive* $\mathbf{\Psi}$*, and a path* $\vec{p} = \vec{p}(t, g)$ *with* $C(\vec{p}) = \sum_{i=1}^{k} C_i^*(t_i, g_i)$*, then, for all* $1 \leq h \leq k$*,* $R_h(\psi_h(\vec{p})) \geq R_h^*(t_h, g_h)$*.*

Proof: By the definition of $\vec{P}_i(t_i, g_i)$, for all $1 \leq h \leq k$, $\psi_h(\vec{p}) \in \vec{P}_i(t_i, g_i)$. Because $R_h^*(t_h, g_h)$ is the smallest residual cost of paths in $\vec{P}_i(t_i, g_i)$, it follows that $R_h(\psi_h(\vec{p})) \geq R_h^*(t_h, g_h)$. ■

Lemma 4.3 *Given an additive* $\mathbf{\Psi}$*, and a path* $\vec{p} = \vec{p}(t, g)$ *with* $C(\vec{p}) = \sum_{i=1}^{k} C_i^*(t_i, g_i)$*, then, for all* $1 \leq h \leq k$*,* $\sum_{i=1}^{k} C_i^*(t_i, g_i) \geq C_h^*(t_h, g_h) + R_h^*(t_h, g_h)$*.*

Proof: By Lemma 2.1, for all $1 \leq h \leq k$, $C(\vec{p}) \geq C_h(\psi_h(\vec{p})) + R_h(\psi_h(\vec{p}))$. By Lemma 4.1, $C_h(\psi_h(\vec{p})) = C_h^*(t_h, g_h)$, and by Lemma 4.2, $R_h(\psi_h(\vec{p})) \geq R_h^*(t_h, g_h)$. Therefore $C(\vec{p}) \geq C_h^*(t_h, g_h) + R_h^*(t_h, g_h)$, and the lemma follows from the premise that $C(\vec{p}) = \sum_{i=1}^{k} C_i^*(t_i, g_i)$. ■

We say that a value x is *infeasible* for state t and goal g if there is no path $\vec{p} = \vec{p}(t, g)$ with $C(\vec{p}) = x$.

Lemma 4.4 *Given an additive* $\mathbf{\Psi}$*, if for some* $h, 1 \leq h \leq k$*, we have* $\sum_{i=1}^{k} C_i^*(t_i, g_i) < C_h^*(t_h, g_h) + R_h^*(t_h, g_h)$ *then* $\sum_{i=1}^{k} C_i^*(t_i, g_i)$ *is infeasible.*

Proof: This is the contrapositive of Lemma 4.3. ■

Lemma 4.5 *If* $\sum_{i=1}^{k} C_i^*(t_i, g_i)$ *is feasible. Then* $\sum_{i=1}^{k} C_i^*(t_i, g_i) \geq max_{i=1}^{k}(\mathrm{OPT}_i(t_i, g_i))$

Proof: Since $\sum_{i=1}^{k} C_i^*(t_i, g_i)$ is feasible, by the contrapositive of Lemma 4.4, $\forall h, 1 \leq h \leq k, \sum_{i=1}^{k} C_i^*(t_i, g_i) \geq C_h^*(t_h, g_h) + R_h^*(t_h, g_h)$.

By the definition of $\mathrm{OPT}_i(t_i, g_i)$ $\forall h, 1 \leq h \leq k, C_h^*(t_h, g_h) + R_h^*(t_h, g_h) \geq \mathrm{OPT}_h(t_h, g_h)$.

Therefore, $\forall h, 1 \leq h \leq k$,

$\sum_{i=1}^{k} C_i^*(t_i, g_i) \geq \mathrm{OPT}_h(t_h, g_h) \Rightarrow \sum_{i=1}^{k} C_i^*(t_i, g_i) \geq max_{i=1}^{k}(\mathrm{OPT}_i(t_i, g_i))$

■

Lemma 4.5 indicates that if the heuristic value h defined by additive abstractions is feasible, h is as good as the maximum over standard heuristic values defined by the same abstractions. However, the reverse implication does not hold in general, and determining which abstractions will produce additives that are better than max over standards is still a big research issue.

4.3 Experimental Results

Different additive abstractions generate different number of infeasible heuristic values. For the pancake puzzle, almost all heuristics are feasible based on the additive abstractions defined by the method of *Fixed Reference* mentioned in Section 3.2. While for the 15 sliding tile puzzle, approximately one-third heuristics are infeasible based on the additive abstractions defined by Korf and Felner (2002), Felner et al. (2004). So we choose the fifteen sliding tile puzzle to show that the quality of the heuristics can be enhanced by checking for infeasibility. We use Dijkstra's algorithm to build the pattern databases and our heuristic search algorithm is IDA^*.

Table 3 shows comparative results over 1000 random instances of the 15-sliding tile problems, whose average solution length is 52.522. The **H** column is the average heuristic value of 1000 initial instances. The **Nodes** shows the avarage number of nodes generated to find an optimal solution. The **Sec** column gives the average amount of CPU time that was needed to solve a problem on a machine with AMD Athlon(tm) 64 Processor 3700+ with 2.4G clock rate and 1G memory. The **Memory** column indicates the size of each set of Pattern Databases. We applied sparse mapping for databases.

According to the parity of the solution length for the sliding tile puzzle, in our experiments, we add 2 to each infeasible additive heuristic value.

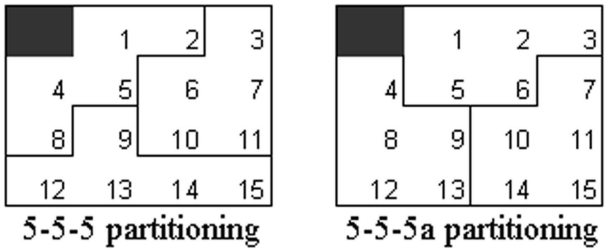

Figure 9: Different disjoint databases for the Fifteen Puzzle

	H	Nodes	Sec
Static5-5-5	41.56	3,186,654	0.642
Check:5-5-5	42.10	1,453,358	0.312
Static:5-5-5a	40.34	8,165,432	1.640
Check:5-5-5a	41.11	3,091,364	0.670

Table 3: The fifteen sliding puzzle results

The "Static5-5-5" data row gives results for the normal additive PDBs with 5-5-5 partitioning shown in the left part of Figure 9. The second row is for additive PDBs with 5-5-5 partitioning enhanced by checking for infeasibility.

The last two rows report results using 5-5-5a partitioning(shown in right part of Figure 9) of the tiles without/with checking for infeasible additive heuritics.

Note that the average running time of IDA* using the heuristics enhanced by checking for infeasible additive values is over 2 times faster than the running time required on average

without checking for infeasibility on the same machine. However, there is a space penalty for this improvement, as we need more memory for storing more information to check for infeasibility. Therefore, we conclude that if sufficient memory is available and additive heuristic values are frequently infeasible, checking for infeasibility may be an effective way to improve the quality of the additive heuristic value.

5. Conclusions and future work

We have presented more general definitions for the additive abstractions. The key difference between our definition and previous ones is that by our definition, the additive abstractions can be defined more freely and more independently. We have applied the additive abstractions to the domain of the pancake puzzle, where the previous definitions of additive abstractions could not be applied.

The heuristics may be improved by checking for infeasibility. To our knowledge, this is the first time for considering non-distinguished moves (R moves).

Future work can continue in the following directions. First, how to identify effective additive abstractions for other domains should be analyzed. Second, since the pancake puzzle bears some resemblance to some real-world problems, such as genome rearrangement problem descirbed in (Erdem & Tillier, 2005), we want our research to expand to these real-world problems.

Appendix A. lemma and proof

Lemma A.1 *For any path $\vec{p} = \vec{p}(t, g)$ in $\mathcal{S}$, there is a corresponding abstract path $\psi_i(\vec{p})$ from t_i to g_i in $\mathcal{A}_i$. Also, $C(\vec{p}) \geq C_i(\psi_i(\vec{p})) + R_i(\psi_i(\vec{p}))$.*

Proof: By definition, $\vec{p}(t, g)$ is a sequence of edges $\langle \pi^1, \ldots, \pi^n \rangle, \pi^j \in \Pi$ where $\pi^j = (t^j, t^{j+1}), 1 \leq j \leq n$ and $t^1 = t, t^{n+1} = g$. Because $\Pi_i \supseteq \psi_i(\Pi)$, each of the corresponding abstract edges exist ($\pi_i^j \in \Pi_i$). Because $\pi_i^1 = (t_i, t_i^2)$ and $\pi_i^n = (t_i^n, g_i)$, the sequence, $\psi_i(\vec{p}) = \langle \pi_i^1, \ldots, \pi_i^n \rangle$ is a path from t_i to g_i.

By definition, $C(\vec{p}) = \sum_{j=1}^{n} C(\pi^j)$. For each π^j, condition 1 ensures that $C(\pi^j) \geq C_i(\pi_i^j) + R_i(\pi_i^j)$, and therefore $C(\vec{p}) \geq \sum_{j=1}^{n}(C_i(\pi_i^j) + R_i(\pi_i^j)) = \sum_{j=1}^{n} C_i(\pi_i^j) + \sum_{j=1}^{n} R_i(\pi_i^j) = C_i(\psi_i(\vec{p})) + R_i(\psi_i(\vec{p}))$. ■

Lemma A.2 $\forall t, g \in T, \min\{C_i(\psi_i(\vec{p})) + R_i(\psi_i(\vec{p})) \mid \vec{p} = \vec{p}(t, g)\} \geq \mathrm{OPT}_i(t_i, g_i)$.

Proof: $\{C_i(\psi_i(\vec{p})) + R_i(\psi_i(\vec{p})) \mid \vec{p} = \vec{p}(t, g)\} \subseteq \{C_i(\vec{p_i}) + R_i(\vec{p_i}) \mid \vec{p_i}$ is an abstract path in $\mathcal{A}_i$ from t_i to $g_i\}$. Therefore, $min\{C_i(\psi_i(\vec{p})) + R_i(\psi_i(\vec{p})) \mid \vec{p} = \vec{p}(t, g)\} \geq min\{C_i(\vec{p_i}) + R_i(\vec{p_i}) \mid \vec{p_i}$ is an abstract path in $\mathcal{A}_i$ from t_i to $g_i\} = \mathrm{OPT}_i(t_i, g_i)$. ■

Lemma A.3 $\forall t, g \in T, \mathrm{OPT}(t, g) \geq \mathrm{OPT}_i(t_i, g_i)$.

Proof: By Lemma A.1, $C(\vec{p}) \geq C_i(\psi_i(\vec{p})) + R_i(\psi_i(\vec{p}))$, and therefore $\min\{C(\vec{p}) \mid \vec{p} = \vec{p}(t, g)\} \geq \min\{C_i(\psi_i(\vec{p})) + R_i(\psi_i(\vec{p})) \mid \vec{p} = \vec{p}(t, g)\}$. The left hand side of this inequality is $\mathrm{OPT}(t, g)$, by definition, and the right hand side was shown in Lemma **??** to be greater than or equal to $\mathrm{OPT}_i(t_i, g_i)$. Therefore, $\mathrm{OPT}(t, g) \geq \mathrm{OPT}_i(t_i, g_i)$. ■

References

Culberson, J. C., & Schaeffer, J. (1994). Efficiently searching the 15-puzzle. Tech. rep., Department of Computing Science, University of Alberta.

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

Edelkamp, S. (2001). Planning using pattern databases. In *Proceedings of the 6th European Conference on Planning*, pp. 13–34.

Erdem, E., & Tillier, E. (2005). Genome rearrangement and planning. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-05)*, pp. 1139–1144.

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

Felner, A., Zahavi, U., Holte, R., & Schaeffer, J. (2006). Dual search in permutation state spaces. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence (AAAI-06)*, pp. 1076–1081.

Felner, A., Zahavi, U., Schaeffer, J., & Holte, R. (2005). Dual lookups in pattern databases. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI-05)*, pp. 103–108.

Hernádvölgyi, I., & Holte, R. C. (2000). Experiments with automatically created memory-based heuristics. *Proc. SARA-2000, Lecture Notes in Artificial Intelligence*, *1864*, 281–290.

Holte, R. C., Perez, M. B., Zimmer, R. M., & MacDonald, A. J. (1996). Hierarchical A*: Searching abstraction hierarchies efficiently. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pp. 530–535.

Holte, R. C., Grajkowski, J., & Tanner, B. (2005). Hierarchical heuristic search revisited. *Proc. SARA-2005, Lecture Notes in Artificial Intelligence*, *3607*, 121–133.

Korf, E., & Felner, A. (2002). Disjoint pattern database heuristics. *Artificial Intelligence*, *134*, 9–22.

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

Korf, R. E. (1980). Toward a model of representation changes. *Artificial Intelligence*, *14*(1), 41–78.

Korf, R. E. (1997). Finding optimal solution to Rubik's Cube using pattern databases. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, pp. 700–705, Providence, Rhode Island.

Prieditis, A. E. (1993). Machine discovery of effective admissible heuristics. *Machine Learning*, *12*, 117–141.