Exploring Infeasibility for Abstraction-Based Heuristics

Fan Yang
Computing Science Department
University of Alberta
Edmonton, Alberta T6G 2E8 Canada
fyang@cs.ualberta.ca

Abstract

Infeasible heuristics are heuristic values that cannot be the optimal solution cost. Detecting infeasibility is a useful technique (Yang *et al.* 2008) to improve the quality of heuristics because it allows the heuristic value to be increased without risking it becoming inadmissible. However, extra memory is required when applying this technique. Is checking for infeasibility the best way to use this extra memory? Can this technique be extended to problems with non-uniform edge costs? Can infeasibility only be detected for additive heuristics? These questions guide us to explore infeasibility further. Comparative experimental results show the potential benefits of this technique.

Introduction

Heuristic search is a general problem-solving mechanism in artificial intelligence. Guided by a heuristic evaluation function, heuristic search finds the shortest path between two nodes in a problem-space graph. A common way to compute heuristic values includes two steps. The first step is to define the abstract state space and the second is to determine the distance from the abstract state to the abstract goal.

Given multiple abstractions of a state space, a standard method for defining a heuristic function is the maximum of the abstract distances given by the abstractions individually. This heuristic is referred to as h_{max}.

A set of abstractions is called "additive" if the sum of the costs returned by a set of abstractions is admissible(Korf & Felner 2002; Felner, Korf, & Hanan 2004). The heuristic calculated using additive abstractions is referred to as h_{add}. For some cases (Korf & Felner 2002; Felner, Korf, & Hanan 2004; Yang *et al.* 2008) additive abstraction-based heuristics can be very powerful, but h_{add} is not always as accurate as we expected.

A new technique(Yang, Culberson, & Holte 2007; Yang *et al.* 2008) is sometimes able to identify that some additive abstraction-based heuristic value is provably too small (infeasible). Detecting the infeasible heuristic value is very useful because it allows the heuristic value to be increased without risking it becoming inadmissible. The additive abstraction-based heuristic improved by checking for infeasibility is referred to as $h_{add-check}$. As additional memory is required, some questions arise when applying $h_{add-check}$. Given extra memory, is it the best way to store extra information only to identify infeasibility? As a great variety of real-world problems can be modeled as problems of searching with non-uniform edge costs, can this technique also be extended for problems with non-uniform edge costs? Is this technique only effective for additive abstractions? These questions guide us to explore infeasibility further. The following summarizes our research contributions in this paper.

- Comparative results showed that given additional memory, it is a competitive choice to store extra information to detect infeasibility.
- We showed that checking for infeasibility can also be useful for problems with non-uniform edge costs.
- A first attempt was made to identify infeasibility for standard abstractions.

The remainder of the paper is organized as follows. First we present the technical background for infeasibility. Second we give an algorithm to compute necessary information to improve the quality of heuristics. Then in order to show the potential benefits of $h_{add-check}$, taking the sliding tile puzzles and the pancake puzzle as selected case studies, we compare the performances of heuristic search using h_{max} and $h_{add-check}$ under different edge cost definitions. Next we explore the method to identify infeasibility for standard abstractions (*i.e.* h_{max}). The last section summarizes our work and directions of our future work.

Background

This section provides the technical background useful for understanding the approach to identify infeasibility.

Definitions and Notations

Definition 1: A *state space* is a weighted directed graph $\mathcal{S} = \langle T, \Pi, C \rangle$ where T is a finite set of states, $\Pi \subseteq T \times T$ is a set of directed edges (ordered pairs of states) representing state transitions, and $C : \Pi \longrightarrow \mathcal{N} = \{0, 1, 2, 3, \dots\}$ is the edge cost function.

Definition 2: An *abstract state space* is a directed graph with two weights per edge, defined by a four-tuple $\mathcal{A}_i = \langle T_i, \Pi_i, C_i, R_i \rangle$. T_i is the set of abstract states and Π_i is the set of abstract edges, as in the definition of a state space. In an abstract space there are two costs associated with each $\pi_i \in \Pi_i$, the *primary cost* $C_i : \Pi_i \longrightarrow \mathcal{N}$ and the *residual cost* $R_i : \Pi_i \longrightarrow \mathcal{N}$. We split each abstract edge cost into two parts. This idea is inspired by the fact that the most common way to define additive abstractions (Korf & Felner 2002; Felner, Korf, & Hanan 2004) is the sum of the distances in a set of abstract spaces in which only some edge costs are counted and others are ignored. Previous papers (Yang, Culberson, & Holte 2007; Yang *et al.* 2008) have provided examples to explain this idea in detail.

Definition 3: An *abstraction mapping* $\psi_i : \mathcal{S} \longrightarrow \mathcal{A}_i$ between state space $\mathcal{S}$ and abstract state space $\mathcal{A}_i$ is defined by a mapping between the states of $\mathcal{S}$ and the states of $\mathcal{A}_i$, $\psi_i : T \rightarrow T_i$, that satisfies the two conditions. The first condition is that the connectivity in the original space be preserved, *i.e.*, $\forall (u,v) \in \Pi, (\psi_i(u), \psi_i(v)) \in \Pi_i$. The second condition is that abstract edges must not cost more than the edges they correspond to in the original state space, *i.e.*, $\forall \pi \in \Pi, C_i(\pi_i) + R_i(\pi_i) \leq C(\pi)$. These two conditions guarantee that the heuristic generated by each individual abstraction is admissible and consistent.

We use the shorthand notation $t_i = \psi_i(t)$ for the abstract state in T_i corresponding to $t \in T$.

Definition 4: A *path* $\vec{q}$ from state t_i to state g_i in the abstract state space $\mathcal{A}_i$ is defined by $\vec{q} = \langle \pi_i^1, \ldots, \pi_i^n \rangle, \pi_i^j \in \Pi_i$ where $\pi_i^j = (t_i^{j-1}, t_i^j), j \in \{1, \ldots, n\}$ and $t_i^0 = t_i, t_i^n = g_i$. We use $Paths(\mathcal{A}_i, t_i, g_i)$ to denote the set of all paths from t_i to g_i in $\mathcal{A}_i$.

Definition 5: An *Abstraction System* is a triple $< \mathcal{S}, \mathbf{A}, \mathbf{\Psi} >$ where $\mathcal{S}$ is a state space, $\mathbf{A} = \{\mathcal{A}_1, \ldots, \mathcal{A}_k\}$ is an indexed set of abstract state spaces and $\mathbf{\Psi} = \{\psi_1, \ldots, \psi_k\}$ is an indexed set of abstraction mappings $\psi_i : \mathcal{S} \longrightarrow \mathcal{A}_i$.

Definition 6: The heuristic obtained from abstract space $\mathcal{A}_i$ for the cost from state t to g is defined by

$$h_i(t,g) = \min_{\vec{q} \in Paths(\mathcal{A}_i, t_i, g_i)} \{C_i(\vec{q}) + R_i(\vec{q})\}.$$

Definition 7: The *common h_{max} heuristic* from state t to state g defined by an abstraction system $< \mathcal{S}, \mathbf{A}, \mathbf{\Psi} >$ is

$$h_{max}(t,g) = \max_{i=1}^{k} h_i(t,g)$$

Definition 8: An abstraction system $< \mathcal{S}, \mathbf{A}, \mathbf{\Psi} >$ is *additive* if $\forall \pi \in \Pi, \sum_{i=1}^{k} C_i(\pi_i) \leq C(\pi)$.

Definition 9: Given an additive abstraction system the *additive heuristic h_{add}* is defined to be

$$h_{add}(t,g) = \sum_{i=1}^{k} C_i^*(t_i, g_i) \text{ , where}$$

$$C_i^*(t_i, g_i) = \min_{\vec{q} \in Paths(\mathcal{A}_i, t_i, g_i)} C_i(\vec{q})$$

is the minimum primary cost of a path in the abstract space from t_i to g_i.

Definition 10: The *conditional optimal residual cost* is the minimum residual cost among the paths in $\vec{P}_i(t_i, g_i)$:

$$R_i^*(t_i, g_i) = \min_{\vec{q} \in \vec{P}_i(t_i, g_i)} R_i(\vec{q})$$

where $\vec{P}_i(t_i, g_i)$ is the set of abstract paths from t_i to g_i whose primary cost is minimal, *i.e.*, $\vec{P}_i(t_i, g_i) = \{\vec{q} \mid \vec{q} \in Paths(\mathcal{A}_i, t_i, g_i) \text{ and } C_i(\vec{q}) = C_i^*(t_i, g_i)\}$.

The Approach to identify infeasibility

Given an additive abstraction system, the key to identify infeasibility is to check whether there exists some $j (1 \leq j \leq k)$ such that $h_{add}(t,g) < C_j^*(t_j, g_j) + R_j^*(t_j, g_j)$. If there exists such j, then $h_{add}(t,g)$ is an infeasible heuristic value. Once identified the infeasible values can be increased to give a better estimate of the solution cost. Formally, the heuristic $h_{add-check}$ is defined by $h_{add-check}(t,g) =$

$$\begin{cases} h_{add}(t,g) + \varepsilon, & \text{If } h_{add}(t,g) \text{ is identified to be infeasible.} \\ h_{add}(t,g), & \text{Otherwise.} \end{cases}$$

Generally, ε is assigned to be one for the state space with unit edge costs, or more according to some special structural property. For example, it is well-known that the additive heuristic value of the sliding tile puzzle has the parity property, therefore 2 can be added to the infeasible $h_{add}(t,g)$ of the sliding tile puzzle.

The algorithm to compute C^* and R^*

To obtain $h_{add-check}$, in addition to the primary cost (C^* for short), we need to store values of the conditional optimal residual cost (R^* for short) to identify infeasibility. This section describes an algorithm to compute both C^* and R^* as follows.

Algorithm: To compute C^* and R^*

```
//X – a Min-Heap working as an Open List
// C_ij – the primary cost for arc(i, j)
// R_ij – the residual cost for arc(i, j)
// MUL – a fixed number that is larger than any R_i.
// V_i – to store the value of C_i× MUL +R_i for node i.
Initialize the values of V_i to be infinit for any node i.
X ⟵ φ
X ⟵ X ∪ {s} /∗ s is the goal state.∗/
While (X≠ φ)
Do begin
  k ⟵ element of X such that V_k ≤ V_x, for any x ∈ X.
  X ⟵ X − {k}.
  If k is in the the Closed List
      continue;
  end
  Put k to the Closed List;
  For each arc(k,j)
  Do begin
    cost_kj=C_kj× MUL +R_kj
    if ((V_k + cost_kj) < V_j)
      V_j = V_k + cost_kj
      X ⟵ X ∪ {j}
    end
  end
end
```

This algorithm is adapted from Dijkstra's algorithm (Cormen *et al.* 2001). Because it always chooses the vertex with the least value of V, this algorithm terminates with $V_i = C_i^* \times$ MUL $+ R_i^*$. Here MUL is defined by the minimum multiple of 10 such that for any possible value of the residual cost R, R / MUL=0, and R mod MUL=R. For example, assuming that the maximum optimal solution cost in the state space is no more than 50 such that $R \leq 50$, we can define MUL=100. If $C_i = 17$ and $R_i = 9$, then $V_i = 17 \times 100 + 9 = 1709$ and when the algorithm terminates, if $V_i = 1709$, C_i^* and R_i^* can be calculated to be 17 and 9, respectively.

All experiments in this paper will store the values of C^* and R^* defined by each abstraction into a lookup table in the form of a pattern database(Culberson & Schaeffer 1994), and we perform *IDA** (Korf 1985) as the heuristic search algorithm.

Comparison between h_{max} and $h_{add-check}$

As shown in the previous work (Yang *et al.* 2008), it is necessary to store R^* to identify the infeasible heuristic values. Therefore, more memory is needed for this technique. Given additional memory, to show that checking for infeasibility is a competitive choice we compare $h_{add-check}$ to h_{max} with the same memory requirement in two domains: the 15-puzzle and the 14-puzzle.

In the 15-puzzle there are 16 locations in the form of a 4×4 grid and 15 tiles, numbered 1–15, with the 16^{th} location being empty (or blank). A tile that is adjacent to the empty location can be moved into the empty location vertically or horizontally. The 14-puzzle is defined to be the same as the 15-puzzle, except that a tile numbered 5 is always fixed in its goal location, the shaded square shown in Figure 3 and Figure 4.

Experimental Results

The results are shown in Table 1 and Table 2. The **Abs** column shows the set of abstractions used to generate heuristics whose partitionings were described in Figure 1–4. The **Heuristic** column indicates different methods to combine the costs returned by the abstractions. The **Nodes** column shows the average number of nodes generated in solving randomly generated start states. The **Time** column gives the average number of CPU seconds needed to solve these start states on an AMD Athlon(tm) 64 Processor 3700+ with 2.4GHz clock rate and 1GB memory. According to the parity property of the puzzle, once h_{add} is identified to be infeasible, $h_{add-check} = h_{add} + 2$.

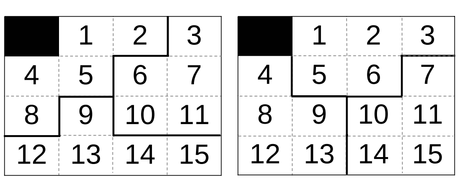

Figure 1: Left: 5-5-5* partitioning. Right: 5-5-5$_a^*$ partitioning.

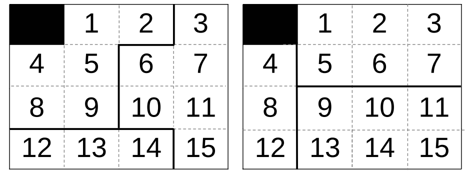

Figure 2: Left: 6-6-3* partitioning. Right: 6-6-3$_a^*$ partitioning.

Abs	Heuristic	Nodes	Time
5-5-5*	h_{add_1}	2,237,899	0.552
5-5-5$_a^*$	h_{add_2}	5,929,024	1.471
5-5-5* & 5-5-5$_a^*$	h_{max}	946,754	0.349
5-5-5*	$h_{add-check}$	912,661	0.340
6-6-3*	h_{add_1}	1,261,566	0.336
6-6-3$_a^*$	h_{add_2}	3,041,540	0.817
6-6-3* & 6-6-3$_a^*$	h_{max}	415.075	0.162
6-6-3*	$h_{add-check}$	479,781	0.173

Table 1: Experimental results on 1000 standard test problems for the 15-puzzle. (The average solution length was **52.522** moves). h_{max}=max$\{h_{add_1}, h_{add_2}\}$.

Every four rows consist of a group. In each group, the first two rows show results using different h_{add} respectively; the third row presents the result using h_{max}, the maximum of the above two h_{add}; the fourth row shows the result of using $h_{add-check}$. h_{max} and $h_{add-check}$ have the same memory requirement, because $h_{add-check}$ is just based on one set of additive abstractions and it requires double size of the space to store extra information to detect infeasibility. While h_{max} need two sets of additive abstractions that need double size of the space for a single set of additive abstractions. The blank is always regarded as a distinguished tile for each abstraction since there is sufficient memory to store these pattern databases. It leads to the results that the performance of original h_{add} reported in Table 1 is somewhat better than that reported in the previous work (Felner, Korf, & Hanan 2004; Yang *et al.* 2008).

In Table 1 and Table 2, the average running time of IDA* using $h_{add-check}$ is over 2 times faster than the running time required on average without checking for infeasibility (h_{add_1}) on the same machine.

The first group of each table shows that when the abstraction is based on smaller number of distinguished tiles, $h_{add-check}$ outperforms h_{max} in terms of nodes generated and the running time. The second group of results implies when the abstraction involves more distinguished tiles, the effectiveness of $h_{add-check}$ drops a little compared to that of h_{max}. It is because more distinguished tiles involved provide more accute h_{add} such that h_{max} benefits directly but little room is left for improving h_{add} by detecting infeasibility. This character is very important because as problems scale up, memory limitations will preclude using abstraction with more distinguished tiles and the only option will be to use abstractions with fewer distinguished tiles each.

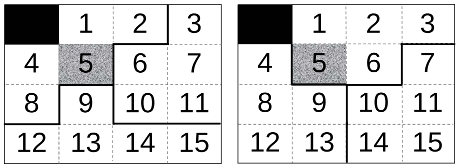

Figure 3: Left: 4-5-5* partitioning. Right: 4-5-5^*_a partitioning. Tile 5 is a fixed tile

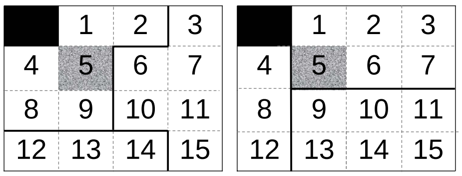

Figure 4: Left: 5-6-3* partitioning. Right: 5-6-3^*_a partitioning.

The results shown in both Table 1 and Table 2 indicate that $h_{add-check}$ will be the method of choice in this situation.

$h_{add-check}$ For Non-uniform Edge Costs

The usefulness of infeasibility has been demonstrated experimentally for the domain with unit edge cost(Yang *et al.* 2008). In this section, we present the experimental results to show that this technique can also be effective for problems with non-uniform edge costs. As a demonstration, we consider the pancake puzzle with non-uniformed costs.

In the domain of the N-pancake puzzle, for each state defined by a permutation of N tiles (1...N), there are $N-1$ applicable operators with the N^{th} operator reversing the order of the first $(N+1)$ tiles of the permutation.

A Simple Example

The *location-based* cost defintion (Yang *et al.* 2008) is formally introduced as an effective method to generate additive heuristics for the pancake puzzle. Now we present a simple example to explain how to apply this method for the same domain with non-uniform costs. In this example the N^{th} operator OP_N cost N. The problem is represented by STRIPS planning model where a state is represented by a set of logical atoms that are true in the state; the atom (at l n) indicates that the tile numbered n is at the l^{th} location; P, A, and D are precondition list, add list and delete list respectively. For example, in Figure 5 the state | 3 | 4 | 2 | 1 | is represented by {(at 1 3), (at 2 4), (at 3 2), (at 4 1)}.

As shown in Figure 6, an abstraction is defined by specifying a subset of the atoms and restricting the abstract state descriptions and operator definitions to include only atoms in the subset. The two subsets in this example are {(at – 1), (at – 3)} and {(at – 2), (at – 4)} repectively, where {(at – n)} represents a set of atoms indicating the location of tile numbered n. The location-based costs are defined by choosing a set of atoms B in the add list for each operator and assigning the full original cost to the primary cost of an

Abs	Heuristic	Nodes	Time
4-5-5*	h_{add_1}	2,945,864	0.594
4-5-5^*_a	h_{add_2}	15,432,669	3.060
4-5-5*,4-5-5^*_a	h_{max}	1,149,332	0.320
4-5-5*	$h_{add-check}$	996,210	0.230
5-6-3*	h_{add_1}	2,185,207	0.457
5-6-3^*_a	h_{add_2}	14,754,628	3.124
5-6-3*,5-6-3^*_a	h_{max}	553,711	0.161
5-6-3*	$h_{add-check}$	739,990	0.183

Table 2: Experimental results on 100 random problems for the 14-puzzle. (The average solution length was **53.280** moves.) h_{max}=max$\{h_{add_1}, h_{add_2}\}$.

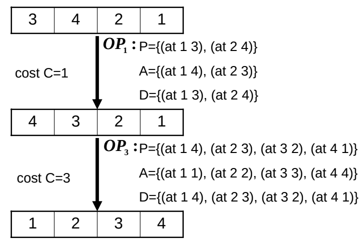

Figure 5: Two state transitions in the original state space of the 4-pancake puzzle.

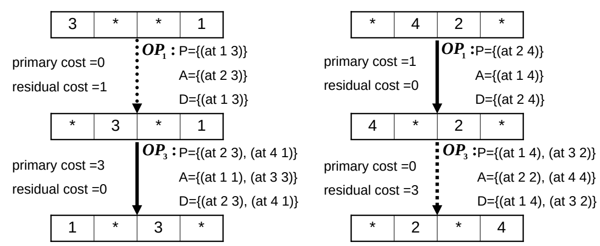

Figure 6: Two state transitions in additive abstractions using location-based costs. Left: state transitions in the first abstract state space. Right: state transitions in the second abstract state space.

operator, if B appears in the add list of the operator definition; otherwise, the primary cost is zero. The residual costs are defined to be complementary to the primary costs (i.e. $R_i(\pi_i) = C(\pi) - C_i(\pi_i)$). For the pancake puzzle, B=(at 1 –) that represents a set of atoms describing which tile is in the first location, because in this domain the first location is so special that every operator changes the tile in this location. Since atoms are partitioned such that any atom (at 1 –) appears in at most one abstraction, this method will define additive costs.

Experimental Results

Two questions guide our study: does the *location-based* method still work on the non-uniform pancake puzzle? Is infeasibility check still effective to improve heuristics? We consider the 12-pancake puzzle with non-uniform edge costs and our experiments compares $h_{add-check}$ to h_{max} under different edge cost definitions. Edge costs are defined in terms of eleven operators (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11). Both h_{add} and h_{max} were obtained using the same 4-4-4[1] abstractions. The results of these experiments are shown in Table 3. The **Edge Cost Def. No.** column indicates the edge cost definition with d_i implying that if the edge applies the i^{th} operator the edge cost is i; otherwise edges cost 1. The **Average Solution** column shows the average solution length of 1000 start states. The **Heuristic** column indicates different methods to combine the costs returned by abstractions. The **Nodes** column shows the average number of nodes generated in solving randomly generated start states. The **Time** column gives the average number of CPU seconds needed to solve these start states on an AMD Athlon(tm) 64 Processor 3700+ with 2.4GHz clock rate and 1GB memory.

h_{add} outperforms h_{max} for all edge cost definitions listed in Table 3, which shows that the *location-based* method can work on the domain with non-uniform edge costs. Compared to h_{add}, $h_{add-check}$ results in further reductions in nodes generated and CPU time. Although $h_{add-check}$ doubles the amount of memory required by h_{max}, $h_{add-check}$ reduces the number of nodes generated and the CPU time by over a factor of 40. Note that when the edge cost is defined by "d_{11}" (i.e. only the 11^{th} operator cost 11; other edges cost 1.), the performance achieved by h_{max} is the worst compared to that of the other edge cost definition. While in this situation, $h_{add-check}$ results in reductions in nodes generated and CPU time by over a factor of 400.

An Attempt to Identify Infeasibility of h_{max}

Previous work(Yang *et al.* 2008) on infeasibility is restricted to h_{add}. Now we made a first attempt to detect infeasibility for h_{max} that is the maximum of k standard abstractions. Define $C_i^{max}(h) = max\{C_i : C_i + R_i = h\}$ and $R_i^{min}(h) = min\{R_i : C_i + R_i = h\}$. It follows that $C_i^{max}(h) + R_i^{min}(h) = h$. Now we define two types of infeasibility, called "*Type I*" and "*Type II*" infeasibility.

As each abstraction preserves any path in the original space, there must be a pair of values $(C_i, R_i)(1 \leq i \leq k)$ representing the solution path. Thus, in an abstract space if the pair of $(C_i^{max}(h), R_i^{min}(h))$ does not exist, h is infeasible and it is referred to as *Type I infeasibility*.

The following lemma defines *Type II Infeasibility* assuming that $(C_i^{max}(h), R_i^{min}(h))$ exists for each abstract space A_i $(i = \{1...k\})$.

Lemma 1: Given k additive abstractions, $\forall i, j \in \{1...k\}$, $R_j^{min}(h) \leq \sum_{i \neq j} C_i^{max}(h)$. If $\sum C_i^{max}(h) < h$, then h is infeasible.

[1] 4-4-4 denotes a set of three abstractions in which the subset of atoms for each abstractions are $\{(\text{at} - k_1) : 1 \leq k_1 \leq 4\}$, $\{(\text{at} - k_2) : 5 \leq k_2 \leq 8\}$, and $\{(\text{at} - k_3) : 9 \leq k_3 \leq 12\}$, respectively

Edge Cost Def. No.	Average Solution	Heuristic	Nodes	Time
d_2	11.039	h_{max}	1,607,139	0.372
		h_{add}	43,244	0.010
		$h_{add-check}$	28,671	0.006
d_3	11.019	h_{max}	1,392,756	0.324
		h_{add}	43,294	0.010
		$h_{add-check}$	27,746	0.006
d_4	11.027	h_{max}	1,287,521	0.301
		h_{add}	44,676	0.010
		$h_{add-check}$	27,139	0.006
d_5	11.025	h_{max}	1,234,990	0.285
		h_{add}	42,193	0.009
		$h_{add-check}$	24,957	0.005
d_6	11.032	h_{max}	1,261,257	0.293
		h_{add}	44,665	0.010
		$h_{add-check}$	26,870	0.006
d_7	11.036	h_{max}	1,371,601	0.317
		h_{add}	44,582	0.010
		$h_{add-check}$	27,300	0.006
d_8	11.085	h_{max}	1,325,913	0.308
		h_{add}	48,583	0.011
		$h_{add-check}$	26,947	0.006
d_9	11.175	h_{max}	1,422,619	0.331
		h_{add}	67,715	0.015
		$h_{add-check}$	32,484	0.007
d_{10}	11.425	h_{max}	1,773,078	0.414
		h_{add}	176,284	0.040
		$h_{add-check}$	49,785	0.011
d_{11}	19.873	h_{max}	6,674,119	1.588
		h_{add}	24,903	0.006
		$h_{add-check}$	15,459	0.003

Table 3: Experimental results on the 12-pancake puzzle. If h_{add} is infeasible, $h_{add-check}$=h_{add}+1.

Proof: Suppose for a contradiction that h is not infeasible, *i.e.*, h is a solution cost. $\forall i, j \in \{1...k\}$, $R_j^{min}(h) \leq \sum_{i \neq j} C_i^{max}(h) \implies h = C_j^{max}(h) + R_j^{min}(h) \leq \sum C_i^{max}(h)$. It contradicts with the condition that $\sum C_i^{max}(h) < h$. Therefore, h is infeasible. ■

Table 4 presents an example to detect *Type I* and *Type II* infeasibility for h_{max} that is the maximum of three standard abstractions (Abs_1, Abs_2 and Abs_3). In this example, at least in one abstract space there exist no pair of (C_i, R_i) such that $C_i + R_i = 5$, 7 or 9. So the heuristic values 5, 7 and 9 are infeasible (*Type I* infeasibility). In the last row of Table 4, $\sum C_i^{max}(11) = 3 + 4 + 2 < 11$. By Lemma 1 this intance cannot be solved by the cost of 11, *i.e.*, 11 is detected to be infeasible (*Type II* infeasibility).

Experimental Results

The condition that $\forall i, j \in \{1...k\}$, $R_j^{min}(h) \leq \sum_{i \neq j} C_i^{max}(h)$ is satisfied in the sliding tile puzzle. The

h value	$C_i^{max}(h), R_i^{min}(h)$	Abs_1	Abs_2	Abs_3
5	$(C_i^{max}(5), R_i^{min}(5))$		(2,3)	
7	$(C_i^{max}(7), R_i^{min}(7))$	(3,4)		(2,5)
9	$(C_i^{max}(9), R_i^{min}(9))$			(2,7)
11	$(C_i^{max}(11), R_i^{min}(11))$	(3,8)	(4,7)	(2,9)

Table 4: An example to show infeasible heuristic values for h_{max}. Empty entry indicates that there exists no pair of (C_i, R_i) in the abstract space such that $h = C_i + R_i$.

key idea is that in the abstract space it always takes more steps to put all distinguished tiles to their goal locations than that of locating them as "don't care" tiles which are indistinguishable from each other.

Abs	Distinguished Tiles	Infeasible value		
		Type I	Type II	Total
Abs_1 Abs_2	1,3,5,7 2,4,6,8	3,812	1,760	5,572
Abs_3 Abs_4	1,2,3,4 5,6,7,8	2,284	1,764	4,048
Abs_5 Abs_6	1,2,3,4,5 6,7,8	1,695	813	2,508
Abs_7 Abs_8 Abs_9	1,3,5 2,4,7 6,8	3,185	202	3,387
Abs_{10} Abs_{11} Abs_{12}	1,2,3 4,5,6 7,8	2,535	191	2,726

Table 5: The number of infeasible values detected in standard abstractions for all solvable instances ($9!/2 = 181,440$) of the 8-puzzle.

A large number of infeasible values detected for h_{max} of the eight puzzle is shown in Table 5. The **Abs** column shows the set of abstractions used to generate heuristics. The **Distinguished Tiles** column indicates different tile partitionings for the abstractions. The **Infeasible value** column shows the number of infeasible values of two types detected over all solvable instances of the eight puzzle.

The results show that there is a large portion of infeasible h_{max} generated from some abstractions. Due to the well-known parity of the puzzle, detecting infeasibility and adding 2 to the infeasible h_{max} will speed up the search. However there is a space penalty for this improvement, because $R^{min}(h)$ values must be stored in memory and it is not clear if storing $R^{min}(h)$ is the best way to use this extra memory. This experiment just shows that infeasibility checking is one way to use extra memory to speed up search for some problems.

Conclusions and Future Work

Our research and future work on detecting infeasibility are summarized as follows.

Given additional memory, the new technique to identify infeasibility can be a comptetive choice to enhance the search performance. For future work, it would be interesting to compare it with other effective memory-based techniques.

We use STRIPS planning model for the problem representation to imply the extension of location-based cost definition to the area of Planning. Empirical results show that the technique of identifying infeasibility can also be effective for problems with non-uniform edge costs. But sometimes this effectiveness is closely based on the effectiveness of the additive abstractions. We will investigate how this limitation can be overcome by detecting and improving infeasibility more efficiently.

Our theory and experiments shed some light on the question of how to detect infeasibility of h_{max}. Numerous possibilities for improving the approach remain to explore. For example, it would be of interest to investigate how to best integrate structure properties into the presented scheme to identify infeasibility more efficiently, and to analyze what impact this would have on the quality of heuristics and the performance of heuristic search.

Generally, we safely add one to an infeasible heuristic value for problems with unit edge cost. But this improvement seems weak when most of the edges cost more than one. It is necessary to explore the method to increase more without lossing the admissibility of the infeasible heuristic values. One way is to introduce the second minimum primary cost for the improvement. But there is a space penalty because we need to store more primary costs in memory and it is not clear if it is the best way to use this extra memory.

Acknowledgments

Special thanks to Dr. Joseph Culberson and Dr. Robert Holte, for their motivation, encouragement and useful discussions on this research.

References

Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; and Stein, C. 2001. *Introduction to Algorithms*. Cambridge, Massachusetts: The MIT Press.

Culberson, J. C., and Schaeffer, J. 1994. Efficiently searching the 15-puzzle. Technical Report TR94-08, Department of Computing Science, University of Alberta.

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

Korf, R. E., and 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.

Yang, F.; Culberson, J.; Holte, R.; Zahavi, U.; and Felner, A. 2008. A general theory of additive state space abstraction. *Journal of Artificial Intelligence Research (to appear).*

Yang, F.; Culberson, J.; and Holte, R. C. 2007. Using infeasibility to improve abstraction-based heuristics. *Proc. SARA-2007, LNAI* 4612:413–414.