The Consistent Case in Bidirectional Search and a Bucket-to-Bucket Algorithm as a Middle Ground between Front-to-End and Front-to-Front

Vidal Alcázar
Riken AIP, Tokyo, Japan
vidal.alcazar@riken.jp

Abstract

Recently, the proposal of individual bounds that use consistent heuristics in front-to-end bidirectional search has improved the state of the art. However, modern theory in bidirectional search does not include algorithms that explicitly exploit consistency. Here we extend past theoretical work, namely must-expand pairs and derived concepts, to the case in which consistency is used, and clarify their relationship with the aforementioned individual bounds.

Departing from the new theory, we show that consistent front-to-end heuristics can also be seen as an admissible estimation of the lowest cost of any path between any two nodes. Therefore, by grouping nodes by g and their heuristic values in buckets, such an estimate can be computed for sets of nodes and not individual pairs without loss of information. This bucket-to-bucket computation, although as expensive as front-to-front in the worst case, is the state of the art in the Pancake Problem, and allows implementing *near-optimal* algorithms that exploit consistency. Also, experiments offer an insightful measurement of how far front-to-end algorithms are from their theoretical limit.

Introduction

After several decades without major developments, bidirectional search has received a renewed interest. On the one hand, algorithms that exploit the interactions of g on both sides have obtained good empirical results (Holte et al. 2017; Barley et al. 2018; Shperberg et al. 2019); on the other, a thorough theoretical analysis has identified both the minimum number of necessarily-expanded nodes and the pairs of nodes that all front-to-end bidirectional algorithms have to expand (Eckerle et al. 2017; Shaham et al. 2017, 2018). This has allowed to develop groundbreaking concepts, like *near-optimal* algorithms (Chen et al. 2017), algorithms that never expand more than twice the necessarily-expanded nodes of any other front-to-end algorithm.

This theoretical analysis relied on instances with consistent heuristics, but the algorithms themselves did not exploit this fact, a setting known as the I_{AD}/I_{CON} case. More recently, however, the definition of individual bounds, some of which exploit heuristic inaccuracies that rely on the consistency of heuristics, has allowed to push the state of the art even beyond the limits of previous *near-optimal* algorithms (Alcázar, Riddle, and Barley 2020). Algorithms that exploit consistency explicitly are known to belong to the I_{CON}/I_{CON} case. Therefore, currently there is a gap in the theory about bidirectional search in the sense that key definitions are not yet covered for the I_{CON}/I_{CON} case. In order to address this, we define must-expand pairs for the I_{CON}/I_{CON} case and analyze the impact of this new definition for other relevant concepts, like the must-expand graph.

Additionally, consistency can also be seen as a triangle inequality that provides a lower bound on the cost between two arbitrary nodes. Consequently, this cost can then be used as a front-to-front heuristic, despite using the same heuristics as previous front-to-end algorithms. Although computing this cost for all pairs of nodes is not practical, such estimation depends on node values often common to large sets of nodes. Hence, much like some implementations group nodes in buckets by their node values to improve performance (Burns et al. 2012), one may compute this estimation between sets of nodes using these buckets as opposed to using individual nodes. We show how this cost estimation and bucket-to-bucket computation are closely related to concepts like individual bounds and the dynamic must-expand graph (Shperberg et al. 2019). Additionally, we show how these concepts allow implementing *near-optimal* algorithms with respect to algorithms that exploit consistency, solving the question whether it is possible to implement *near-optimal* algorithms in the I_{CON}/I_{CON} case.

In summary, in this paper we define must-expand pairs when consistency is used and extend this definition to other relevant concepts, link must-expand pairs to individual bounds, show the similarities of the bucket-to-bucket computation with the dynamic must-expand graph, and perform experiments that both assess the viability of such an approach and give an insight on the minimum number of nodes that current front-to-end algorithms must expand.

Background

A search instance is a tuple $I = (G = (S, E), start, goal, h_f, h_b)$. G is an implicit directed graph; S is the set of vertices (states in explicit-state search); E is the set of edges, each with a non-negative arbitrary cost; $start \in S$ is the initial state; $goal \in S$ is the goal; h_f, h_b are the forward and backward heuristics respectively. ϵ is the value of the edge with minimum cost, which can be assumed to be 0 without loss of generality if it is unknown.

The cost of the least-cost path between two states in S is $c : S \times S \rightarrow \mathbb{R}_{\geq 0}$. Search algorithms keep track of states using nodes; a node makes reference to a single state, but a state can be referenced by different nodes. Nodes have node values (*e.g.* g) labeled with the direction of the search, like h_f and h_b. When the direction is the same in an equation but can be either one, the label is x, like f_x. Other terms related to direction can be labeled too, *e.g.* $Open_x$ is the open list of direction x. The opposite direction of x is $\bar{x}$. Given node n referencing state $s \in S$, $g_f(n)$ is the cost of the forward path from *start* to n, $g_b(n)$ is the cost of the backward path from *goal* to n, $h_f(n) = h_f(s, goal)$, $h_b(n) = h_b(s, start)$, $f_x(n) = g_x(n) + h_x(n)$, $d_x(n) = g_x(n) - h_{\bar{x}}(n)$ and $b_x(n) = f_x(n) + d_x(n)$. For $g_f(n)$ and $g_b(n)$ to be optimal, $g_f(n) = c(start, s)$ and $g_b(n) = c(s, goal)$ respectively. h_f and h_b are admissible *iff*, for any state $s \in S$, $h_f(s) \leq c(s, goal)$ and $h_b(s) \leq c(start, s)$ respectively. h_x is consistent *iff* it is admissible and $h_x(n) \leq c(n, n') + h_x(n')$ for any pair of nodes n, n'.

$C^* = c(start, goal)$ is the cost of an optimal solution. U is the cost of the best solution found so far, and thus monotonically decreasing. For an algorithm to be optimal, $C^* = U$ at the end if there is at least one solution path. C is a lower bound on the cost of any new solution, and thus monotonically increasing. When $C \geq U$, necessarily $C^* = U$ and thus an optimal solution has been found.

A pair of nodes (n, n') is expanded if at least either n or n' is expanded. $lb(n, n')$ is a lower bound of any solution path that contains the states that n and n' make reference to. A pair that will be expanded by any bidirectional algorithm is a *must-expand pair*, but an optimal bidirectional algorithm does not have to identify *must-expand pairs* explicitly to guarantee optimality. The set of all must-expand pairs define a bipartite graph know as the *must-expand graph* (G_{MX}). The cardinality of its minimum vertex cover is the minimum number of necessarily-expanded nodes (nodes expanded before $C = C^*$) that any algorithm must expand. An algorithm that guarantees never expanding more than twice the minimum number of necessarily-expanded nodes is said to be *near-optimal*. Related to G_{MX}, the dynamic must-expand graph DG_{MX} is a bipartite graph defined by all pairs (n, n') such that $n \in Open_f$, $n' \in Open_b$ and $lb(n, n') = C$. At any time, in order to increase C, all pairs $(n, n') \in DG_{MX}$ must be expanded.

Formally, the I_{AD}/I_{CON} case was defined by Eckerle *et al.* (2017) as *"What can be assumed by the algorithm on the problem instances / The instances the algorithm is executed on"*. The I_{CON}/I_{CON} case is the one in which the algorithm is aware of the consistency of h_f and h_b, and therefore cannot be classified as a *DXBB* algorithm.

Theory of the I_{CON}/I_{CON} Case

In this section we identify the conditions that define a must-expand pair in the I_{CON}/I_{CON} case. Additionally, we analyze how these conditions affect other relevant concepts and how they are linked to the individual bounds. The specific case of undirected graphs is also covered.

Must-Expand Pairs with Consistency

The original theoretical work about must-expand pairs by Eckerle *et al.* (2017) assumed that heuristics were consistent but that the algorithm had no knowledge about it. More recent work based on the fact that C is monotonically non-decreasing with consistent heuristics (Shperberg et al. 2019; Alcázar, Riddle, and Barley 2020) purposely exploits consistency, *e.g.* front-to-end bounds that require consistency, like the *KK bounds* (Alcázar, Riddle, and Barley 2020).

Shaham *et al.* (2018) formalized the distinction between both cases: the I_{AD}/I_{CON} case is the case in which algorithms exploit only admissibility despite having consistent heuristics; and the I_{CON}/I_{CON} case, in which algorithms exploit this fact. In their analysis of I_{CON}/I_{CON}, they realized that consistency imposes a lower bound on the cost of the path between any two nodes $n, n' \in S$. This lower bound comes from the formalization of consistency as a triangle inequality between any three states $a, b, c \in S$ defined in their Lemma 8. Thus, they define a new front-to-front heuristic: $h_C(n, n') = max(h_x(n) - h_x(n'), h_{\bar{x}}(n') - h_{\bar{x}}(n))$[1]. Based on the definition of must-expand pair for front-to-front heuristics, they gave a new definition of must-expand pair for front-to-end I_{CON}/I_{CON} algorithms.

Definition 1. *(Definition of must-expand pair for front-to-end algorithms using consistency by Shaham et al. (2018)). Two nodes n, n' are a must expand pair if all of the following conditions are met:*

1. $f_x(n) < C^*$
2. $f_{\bar{x}}(n') < C^*$
3. $g_x(n) + g_{\bar{x}}(n') + h_C(n, n') < C^*$

This definition is redundant, because the third condition, $g_x(n) + g_{\bar{x}}(n') + h_C(n, n')$, is an upper bound on both $f_x(n)$ and $f_{\bar{x}}(n')$. Then, if the third condition is met, the first and second conditions are necessarily met.

Lemma 1. *(The value of $g_x(n) + g_{\bar{x}}(n') + h_C(n, n')$ dominates the f values). Let h_f, h_b be consistent heuristics and n, n' be a forward and a backward node respectively. Then, $g_x(n) + g_{\bar{x}}(n') + h_C(n, n') \geq max(f_x(n), f_{\bar{x}}(n'))$*

Proof. We prove the forward case. $h_C(n, n') = max(h_x(n) - h_x(n'), h_{\bar{x}}(n') - h_{\bar{x}}(n))$, so $h_C(n, n') \geq h_x(n) - h_x(n')$; therefore, $g_x(n) + g_{\bar{x}}(n') + h_C(n, n') \geq g_x(n) + g_{\bar{x}}(n') + h_x(n) - h_x(n')$. $f_x(n) = g_x(n) + h_x(n)$, so for $f_x(n)$ to be higher, $g_x(n) + h_x(n) > g_x(n) + g_{\bar{x}}(n') + h_x(n) - h_x(n')$ and therefore $0 > g_{\bar{x}}(n') - h_x(n')$ and $h_x(n') > g_{\bar{x}}(n')$. h_x is consistent and thus admissible, so $h_x(n') > g_{\bar{x}}(n')$ is impossible and thus $f_x(n)$ will never be higher than $g_x(n) + g_{\bar{x}}(n') + h_x(n) - h_x(n')$ and nor higher than $g_x(n) + g_{\bar{x}}(n') + h_C(n, n')$ either. ☐

[1]This definition is not exactly Shaham *et al.*'s, as it is a more general definition for both directed and undirected graphs. The case in which the graph is undirected will be analyzed later.

Indeed, Shaham *et al.* (2018) recognize Definition 1 as equivalent to the *KKAdd* method by Kaindl and Kainz (1997). In fact, it is also equivalent to the *KKMax* method, as they are part of the same bound definition (Alcázar, Riddle, and Barley 2020). These bounds/methods strengthen f using heuristic inaccuracies, so it is clear that f will be dominated by the use of h_C anyway.

Another aspect not covered by Shaham *et al.* (2018) is the use of ϵ. If n and n' are not nodes referencing the same state, then they must be at least one step away and so ϵ is obviously a lower bound on the cost of any path between n and n'. Here we redefine h_C to account for this fact.

Definition 2. *(Definition of h_C using ϵ).*[2]

$$h_C(n,n') = \mathit{max}(h_x(n) - h_x(n'), h_{\bar{x}}(n') - h_{\bar{x}}(n), \epsilon) \quad (1)$$

Given two nodes n, n', then $lb(n,n') = g_x(n) + g_{\bar{x}}(n') + h_C(n,n')$ in the I_{CON}/I_{CON} case. Therefore, as claimed by Shaham *et al.* (2018), (n,n') will be a must-expand pair in the I_{CON}/I_{CON} case if $lb(n,n') < C^*$. However, a gap in the theory is that Shaham *et al.* (2018) relied on a proof for the I_{AD}/I_{CON} case to show that such pairs of nodes are indeed must-expand pairs. This is not extensible to the I_{CON}/I_{CON} case in an straightforward way, though. Therefore, here we provide a proper proof for the I_{CON}/I_{CON} case, first for front-to-front heuristics.

Theorem 1. *(Must-expand pair in the front-to-front I_{CON}/I_{CON} case). Let h_f, h_b be consistent front-to-front heuristics, n, n' be a forward and a backward node respectively, and $g_f(n)$ and $g_b(n')$ be optimal. Then, any admissible deterministic algorithm must expand the pair (n,n') if $lb(n,n') < C^*$.*

Proof. We prove the contrapositive with a proof analogous to the one for Theorem 1 of Eckerle *et al.* (2017). Assume that n, n' point to states $s, s' \in S$ that belong to the same optimal solution path P in instance I, $lb(n,n') < C^*$, and that a deterministic algorithm B returns a path of cost C^* different to P without expanding (n,n'). Then a new instance $I' \in I_{CON}$ can be constructed having an optimal solution strictly cheaper than C^* on which B returns P, thereby showing that B is not admissible.

I' is identical to I but has an additional edge e from s to s' such that $c(e) = \frac{C^* - lb(n,n')}{2} + h(n,n')$. $c(e)$ is strictly positive because $C^* > lb(n,n')$ and $h(n,n') \geq 0$. The new edge e creates a solution path P' of cost $\frac{C^* + lb(n,n')}{2}$, which is strictly less than C^* because $lb(n,n') < C^*$. Thus, e is an essential part of any optimal solution to I'.

We show that $I' \in I_{CON}$, *i.e.* that h_f is consistent on I' (the proof for h_b is analogous). Because h_f is consistent along all paths that do not go through e, it suffices to prove that h_f is consistent too when going through e. Let x be an arbitrary backward node, then h_f is not consistent *iff* $h(n,x) > c(e) + h(n',x)$. h_f is consistent on I, so $c(n,n') + h(n',x) \geq h(n,x)$ and $c(n,n') \geq h(n,n')$, and thus $h(n,n') + h(n',x) \geq h(n,x)$. Therefore, h_f is not consistent *iff* $h(n,n') + h(n',x) > c(e) + h(n',x)$, which simplified is $h(n,n') > c(e)$, further simplified to $h(n,n') > \frac{C^* - lb(n,n')}{2} + h(n,n')$ and $0 > \frac{C^* - lb(n,n')}{2}$. $C^* > lb(n,n')$, so $0 > \frac{C^* - lb(n,n')}{2}$ is not true and therefore h_f is consistent on I'.

Because B is deterministic it will behave exactly the same on I' as it did on I. In particular it will not expand n nor n', so it will not discover the edge e and will incorrectly return P as the optimal solution for I'. Hence, B is not admissible. □

Theorem 1 requires front-to-front heuristics. However, as shown in Definition 2, h_C is a front-to-front heuristic built using front-to-end heuristics. Therefore, Definition 2 holds for the front-to-end I_{CON}/I_{CON} case too.

Theorem 2. *(Must-expand pair in the front-to-end I_{CON}/I_{CON} case). Let h_f, h_b be consistent front-to-end heuristics, n, n' be a forward and a backward node respectively, and $g_f(n)$ and $g_b(n')$ be optimal. Then, any admissible deterministic algorithm must expand the pair (n,n') if $lb(n,n') < C^*$.*

Proof. h_f, h_b can be used to create a front-to-front heuristic h_C. Hence, this theorem follows from Theorem 1. □

Comparison between h_C and Individual Bounds

Recently, new individual bounds that yield a lower bound between two set of nodes in opposite directions were defined for front-to-end algorithms (Alcázar, Riddle, and Barley 2020), pushing the state of the art in bidirectional search. Barring the *b bound* for reasons explained below, these are the bounds:

- *g bound*: $gMin_f + gMin_b + \epsilon$
- *forward KK bound*: $fMin_f + dMin_b$
- *backward KK bound*: $fMin_b + dMin_f$

These sets of nodes can also be composed by a single node. Hence, the previous bounds define the following lower bounds for a pair of nodes n, n' (we use $_x$ and $_{\bar{x}}$ instead of $_f$ and $_b$ so the directions of n and n' are interchangeable):

- *g bound*: $g_x(n) + g_{\bar{x}}(n') + \epsilon$
- *forward KK bound*: $f_x(n) + d_{\bar{x}}(n')$
- *backward KK bound*: $f_{\bar{x}}(n') + d_x(n)$

Two observations that link these bounds to h_C can be done: first, they exploit consistency; second, they were defined individually because previous attempts of exploiting bounds maximized over several equations. h_C exploits consistency and maximizes over several equations, so we split Definition 2 in several cases. Also, we add $g_x(n) + g_{\bar{x}}(n')$ to $h_C(n,n')$ in order to obtain the lower bound $lb(n,n')$:

- $g_x(n) + g_{\bar{x}}(n') + \epsilon$
- $g_x(n) + g_{\bar{x}}(n') + h_x(n) - h_x(n')$
- $g_x(n) + g_{\bar{x}}(n') + h_{\bar{x}}(n') - h_{\bar{x}}(n)$

[2]From here on we assume that ϵ is always used, omitting its explicit mention in the notation for succinctness.

As we can see, the first case of $g_x(n) + g_{\bar{x}}(n') + h_C$ is equivalent to the *g bound*. Moreover, that is the case too for cases 2 and 3 of $g_x(n) + g_{\bar{x}}(n') + h_C$ and the forward and backward KK bounds respectively:

- $f_x(n) + d_{\bar{x}}(n') = (g_x(n) + h_x(n)) + (g_{\bar{x}}(n') - h_x(n')) = g_x(n) + g_{\bar{x}}(n') + h_x(n) - h_x(n')$
- $f_{\bar{x}}(n') + d_x(n) = (g_{\bar{x}}(n') + h_{\bar{x}}(n')) + (g_x(n) - h_{\bar{x}}(n)) = g_x(n) + g_{\bar{x}}(n') + h_{\bar{x}}(n') - h_{\bar{x}}(n)$

Following from this equivalence, we can also define must-expand pairs for front-to-end consistent algorithms using the individual bounds from Alcázar *et al.* (2020):

Lemma 2. *(Must-expand pair using individual bounds). Two nodes n, n' are a must-expand pair if all of the following conditions are met:*

1. $g_x(n) + g_{\bar{x}}(n') + \epsilon < C^*$
2. $f_x(n) + d_{\bar{x}}(n') < C^*$
3. $f_{\bar{x}}(n') + d_x(n) < C^*$

This further confirms the relationship between h_C and Kaindl and Kainz's work. The *b bound*, however, cannot be linked to any condition in Lemma 2. That is because, although in the front-to-end case the *b bound* is not dominated by the *KK bounds*, for pairs of nodes its value $\frac{b_x(n)+b_{\bar{x}}(n')}{2}$ can be formulated as $\frac{f_x(n)+d_{\bar{x}}(n')+f_{\bar{x}}(n')+d_x(n)}{2}$. As already pointed out by Alcázar *et al.* (2020), this averages over both *KK bounds* and thus will never be higher than either the forward or the backward *KK bound*.

Must-expand Graph in the I_{CON}/I_{CON} case

The new definition of must-expand pair in the I_{CON}/I_{CON} case shares all the characteristics of the original definition. This way, the whole set of must-expand pairs of an instance defines a must-expand graph G_{MX} as defined by Chen *et al.* (2017) for the I_{CON}/I_{CON} case. Consequently, a minimum vertex cover of such a must-expand graph is the minimum number of necessarily-expanded nodes that a front-to-end I_{CON}/I_{CON} algorithm can expand. Additionally, any algorithm that correctly identifies must-expand pairs at layer C and expands both nodes will be *near-optimal* with respect to any other I_{CON}/I_{CON} algorithm following Theorems 2 and 3 of Chen *et al.* (2017).

Nevertheless, there is an important difference. When using consistency, G_{MX} is neither *contiguous* nor *restrained* (Shaham et al. 2017), meaning that there are no thresholds $t_F, t_B \in \mathbb{R}_{\geq 0}$ such that $t_F + t_B + \epsilon = C^*$ for which a vertex u in direction x is in the minimum vertex cover *iff* $g_x(u) < t_x$. We illustrate this in Figure 1, in which $\{n_f^1, \ldots, n_f^k\}$ and $\{n_b^1, \ldots, n_b^k\}$ are arbitrarily large sets of nodes with the same node values. Apart from the must-expand pairs containing *start* (S) and *goal* (G) we have the following must-expand pairs (pairs that comply with Lemma 2):

- m_f with all $\{n_b^1, \ldots, n_b^k\}$
- m_b with all $\{n_f^1, \ldots, n_f^k\}$
- P_f with P_b

The minimum vertex cover (MVC) of G_{MX} is then $\{S, G, m_f, m_b, P_f \vee P_b\}$. Because m_f belongs to the MVC but $\{n_f^1, \ldots, n_f^k\}$ do not, G_{MX} is not *contiguous*, as not all nodes with a g value lower than $g(m_f)$ are in the MVC. Because m_f and m_b are both in the MVC and $g(m_f) + g(m_b) + \epsilon > C^*$, G_{MX} is not *restrained*, as no pair of thresholds (t_F, t_B) can satisfy the property. Because of this, no known algorithm in the vein of Shaham *et al.* (2017) is able to compute the MVC in linear time.

Consistency in Undirected Graphs

A special case considered by Shaham *et al.* (2018) occurs when G is undirected. In that case, the constraint imposed by consistency is stronger: h_x is consistent *iff* it is admissible and $|h_x(n) - h_x(n')| \leq c(n, n')$ for any pair of nodes n, n'. Therefore, h_C can be redefined as $h_C(n, n') = max(|h_x(n) - h_x(n')|, |h_{\bar{x}}(n') - h_{\bar{x}}(n)|, \epsilon)$. However, front-to-end bounds for the specific undirected case can also be defined. Let us define the following node values:

Definition 3. *(rf value of a node). The rf (reverse f) value of a node n is defined as rf$_x(n) = g_x(n) - h_x(n)$.*

Definition 4. *(rd value of a node). The rd (reverse d) value of a node n is defined as rd$_x(n) = g_x(n) + h_{\bar{x}}(n)$.*

These nodes values are called reverse f and d values because they are based on the notion of reversing paths in undirected graphs and because they are the same as f and d respectively but with an inverted symbol.

Let us define *rfMin*$_x$ and *rdMin*$_x$ as the minimum *rf* and *rd* values among expandable nodes in direction x. Then, we can define new bounds for undirected graphs: the *rc bounds*.

Lemma 3. *(rc bounds). The forward rc (reverse consistent) bound is defined as rfMin$_f$ + rdMin$_b$. The backward rc bound is defined as rfMin$_b$ + rdMin$_f$.*

Based on these new node values, Lemma 2 can be adapted to the undirected case by adding new conditions:

Lemma 4. *(Must-expand pair using individual bounds in indirected graphs). Two nodes n, n' are a must-expand pair if all of the following conditions are met:*

1. $g_x(n) + g_{\bar{x}}(n') + \epsilon < C^*$
2. $f_x(n) + d_{\bar{x}}(n') < C^*$
3. $f_{\bar{x}}(n') + d_x(n) < C^*$
4. *rf*$_x(n)$ + *rd*$_{\bar{x}}(n') < C^*$
5. *rf*$_{\bar{x}}(n')$ + *rd*$_x(n) < C^*$

We omit the proofs for lack of space, although admissibility can be easily proved by linking *rf* and *rd* to h_C when G is undirected, for instance.

A Bucket-to-Bucket Algorithm

Front-to-front algorithms estimate the distance between pairs of nodes to determine the best node to expand. The main drawback of this approach is that a Cartesian product of both open lists is required, which is often too computationally expensive. Computing the lower bound with h_C requires the g value and the heuristic estimates of the nodes. However, when nodes share certain characteristics, they can

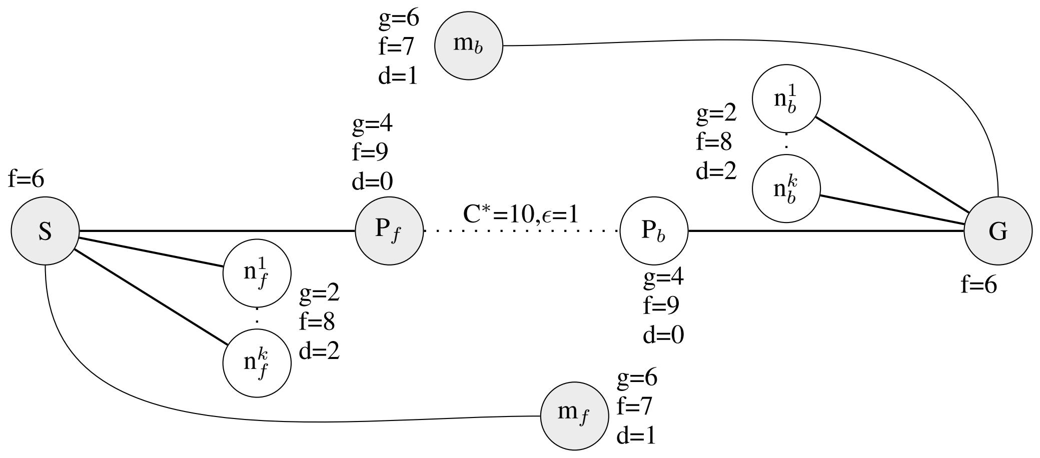

Figure 1: Example of non-contiguous and unrestrained *G*$_{MX}$. Nodes with a gray background belong to a *MVC* of *G*$_{MX}$.

be grouped together for different reasons, like using buckets to speed up the expansion of nodes (Burns et al. 2012). Using g-h_x-$h_{\bar{x}}$ buckets groups all nodes in Open$_x$ with the same g, h_x and $h_{\bar{x}}$ together. Therefore, if the lower bound is computed between two buckets, it can be used for all nodes in those buckets, hence turning it into an estimate between sets of nodes. We call this a bucket-to-bucket computation.

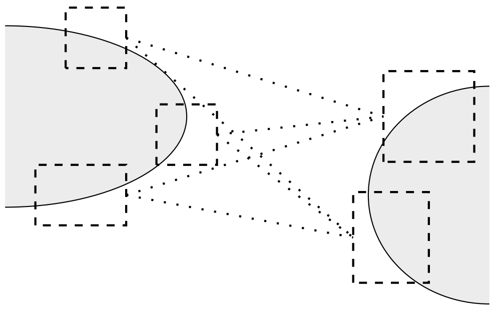

Figure 2: Illustration of a bucket-to-bucket computation.

Figure 2 shows an example of a bucket-to-bucket computation. Dashed rectangles represent buckets that contain nodes from the open list with the same values, while the dotted lines represent the cost estimation between pairs of buckets. In the worst case, buckets will contain a single node, turning this computation into a regular front-to-front heuristic. However, an arbitrarily large number of nodes can be contained within a single bucket, reducing significantly the time required for the computation.

The result of the bucket-to-bucket computation is a set of edges that represent pairs of buckets (u, v). Each edge (u, v) has a corresponding lower bound $lb(u, v)$. Hence, a front-to-front algorithm can be implemented by expanding nodes from buckets in at least one edge such that $lb(u, v) = C$.

It is also worth mentioning that the set of edges and buckets with minimum lower bound can be seen as an I_{CON}/I_{CON} version of the dynamic must-expand graph (*DG*$_{MX}$) used by DVCBS (Shperberg et al. 2019). Thus, we can draw a parallel between an efficient bucket-to-bucket computation of estimates between sets of nodes thanks to h_C and previous work on must-expand pairs.

BTB is our bucket-to-bucket implementation. Until a solution is proven optimal or either open list is empty, a bucket-to-bucket computation is performed. This computation updates C if necessary and returns a set of edges such that u, v are buckets and $lb(u, v) = C$. Buckets can be chosen for expansion in different ways. We have implemented the following versions of BTB:

- **BTB (NBS)**: NBS stands for Near-optimal Bidirectional Search. It picks the bucket with the fewest number of nodes and, among the edges connected to that bucket, it picks the edge that connects it to the smallest bucket in the opposite direction. Then, pairs of nodes from both buckets are expanded until the smallest bucket is empty. Because BTB (NBS) expands both nodes of must-expand pairs such that their lower bound is minimum among all non-expanded pairs, *i.e.* $lb(u, v) = C$, it is *near-optimal*. This follows from Theorems 2 and 3 of Chen *et al.* (2017).
- **BTB (small)**: It expands the smallest bucket. This tries to minimize the number of necessarily-expanded nodes by expanding small buckets.
- **BTB (conn)**: *conn* stands for "connected". It expands the bucket such that the sum of the nodes of the buckets in the other direction connected to it is maximal. This way, the number of nodes whose expansion is prevented in the opposite direction is maximized, hopefully leading to a smaller number of necessarily-expanded nodes.
- **BTB (DVC)**: DVC stands for Dynamic Vertex Cover. Inspired by the dynamic must-expand graph *DG*$_{MX}$ used by DVCBS (Shperberg et al. 2019), but procedurally very different. BTB (DVC) obtains a *DG*$_{MX}$ from the bucket-to-bucket computation, and expands all buckets in a vertex cover. Because all pairs such that $lb(u, v) = C$ must be eventually expanded for C to increase, expanding a whole

vertex cover will prevent bucket-to-bucket computations (potentially saving a high number of costly computations) at the expense of choosing buckets when less information is available. Computing the minimum weighted vertex cover of a bipartite graph can be done in polynomial time; however, DG_{MX} does not necessarily contain all remaining must-expand pairs such that $lb(u, v) = C$, so computing the exact minimum vertex cover may not pay off. Instead, we greedily expand the smallest buckets with edges still in DG_{MX}, removing edges until no edges remain in DG_{MX} whenever a bucket linked to an edge is expanded. When no edges remain in DG_{MX}, a vertex cover has been found and the bucket-to-bucket computation is repeated, updating C if necessary.

Algorithm 1 describes this process. An outer loop repeats the bucket-to-bucket computation (lines 3-4), updating C as long as no solution has been proven to be optimal ($U > C$). The bucket-to-bucket computation returns C, a set of buckets in both directions and a set of edges connecting buckets (line 4). Among buckets still connected to an edge, the smallest one is expanded in its direction (line 6). If no solution of cost C is found (line 8), all edges connected to the bucket are removed (line 11). If no edges remain, a new bucket-to-bucket computation is performed.

Algorithm 1 BTB (DVC)

```
 1: $U \Leftarrow \infty$; $C \Leftarrow 0$
 2: $Open_f \Leftarrow \{start\}$; $Open_b \Leftarrow \{goal\}$
 3: while $Open_f \neq \emptyset \wedge Open_b \neq \emptyset \wedge U > C$ do
 4:     $C$, buckets, edges $\Leftarrow$ ComputeBTB($Open_f$, $Open_b$)
 5:     while edges $\neq \emptyset$ do
 6:         bucket $\Leftarrow$ SelectSmallestConnectedBucket(buckets)
 7:         Expand(bucket)
 8:         if $C \geq U$ then
 9:             return $U$
10:         end if
11:         RemoveEdgesConnectedToBucket(bucket, edges)
12:     end while
13: end while
14: return $U$
```

Experiments

In this section we compare experimentally state-of-the-art front-to-end algorithms that exploit heuristic inaccuracies with the new bucket-to-bucket family of algorithms. The benchmarks and seeds are exactly the same as in Alcázar *et al.* (2020); therefore, the results of any algorithm in Alcázar *et al.* (2020) and not in here are still relevant. The included algorithms apart from BTB are:

- **BAE*.** A heap-based reimplementation of BAE* (Sadhukhan 2013; Alcázar, Riddle, and Barley 2020). It uses $b_x(n)$ as its priority function in both directions. The termination criterion checks whether the value of the *b bound* is equal or higher than U. BAE* (a) expands nodes alternating in directions whereas BAE* (p) follows Pohl's cardinality criterion, that is, it expands nodes in the direction of the smallest open list. Additionally, in this paper h_x (but not $h_{\bar{x}}$) is strengthened using ϵ *i.e.* if $h_x(n) < \epsilon$ then $h_x(n) = \epsilon$ instead. This helps to obtain higher bounds when both $h_x < \epsilon$ and $h_{\bar{x}} < \epsilon$ without compromising neither admissibility nor consistency.
- **DBBS.** DBBS as in Alcázar *et al.* (2020). It uses all known front-to-end bounds, doing a fixpoint computation to determine which nodes have a lower bound higher than C. There are three differences with the previous work: first, all benchmarks are undirected and hence we use the *rc bounds* as well. For this, we use DBBS g-h_x-$h_{\bar{x}}$ buckets instead of g-f-d ones, as this allows us to compute all node values used by the bounds.

  Second, it was already pointed out by Alcázar *et al.* (2020) how expanding by b yielded better results on average. Because of that, we show results both for DBBS expanding by g (DBBS$_g$) and expanding by b (DBBS$_b$).

  Third, in the original implementation DBBS (p) counted the number of expandable nodes with minimum g to decide in which direction to expand. Since we also expand by b in the experiments, now both DBBS$_g$ (p) and DBBS$_b$ (p) just count all expandable nodes, and expand in the direction of the smallest set of expandable nodes.

Experiments show average total expanded nodes (expanded when $C \leq C^*$), average necessarily-expanded nodes (expanded when $C < C^*$), and nodes expanded per second (n/s) for the hardest instance of the set. We pick as the hardest instance the instance for which BAE* (p) needs more time. We do so because BAE*'s heap-based implementation is less sensitive to the distribution of values of g, h_x and $h_{\bar{x}}$. Also, not alternating directions, that is, using Pohl's cardinality criterion, can lead to more cache hits, making BAE* (p) faster than BAE* (a). Results in bold represent the best result for a benchmark. BTB (NBS) appears in a highlighted row, as it is *near-optimal* and separates in the table the front-to-end algorithms from BTB.

Table 1 shows results for 100 random instances in the **14-Pancake Puzzle** with the GAP heuristic (Helmert 2010). GAP-k means that pancakes $(0, k-1)$ of the target state are ignored to get an asymmetric weaker heuristic. We omit nodes generated per second for GAP-0 and GAP-1 because the number of expanded nodes is too low to have a clear understanding of the efficiency of the algorithms.

In this domain, BTB (conn) is clearly the most informed algorithm, as it expands the fewest necessarily-expanded nodes for all heuristics but GAP-0. The rest of the BTB configurations are not as competitive, being better than front-to-end algorithms only for GAP-5 and GAP-6. For DBBS, the degradation of the heuristic highlights how the importance of the bounds is relative to the accuracy of the heuristic: while DBBS$_b$ is clearly the best front-to-end algorithm up to GAP-4, DBBS$_g$ catches up for GAP-5 and becomes the best front-to-end algorithm for GAP-6, as the *g bound* is the main bound raising C when heuristics are weak. Understandably, BAE* displays a similar behavior to DBBS$_b$ albeit expanding more nodes on average, as both algorithms use the *b bound* and expand by b but DBBS$_b$ uses other bounds as well. As opposed to DBBS, BTB is not affected

Algorithm	GAP-0		GAP-1		GAP-2			GAP-3			GAP-4			GAP-5			GAP-6		
	$\leq C^*$	$< C^*$	$\leq C^*$	$< C^*$	$\leq C^*$	$< C^*$	n/s	$\leq C^*$	$< C^*$	n/s	$\leq C^*$	$< C^*$	n/s	$\leq C^*$	$< C^*$	n/s	$\leq C^*$	$< C^*$	n/s
BAE* (a)	90	65	654	552	15110	11252	156k	132k	113k	86k	469k	465k	77k	1116k	1087k	71k	1786k	1750k	70k
BAE* (p)	**88**	64	729	620	17581	13420	168k	154k	135k	85k	480k	476k	76k	1117k	1087k	72k	1779k	1743k	70k
DBBS$_g$ (a)	521	58	13924	1780	136458	39776	47k	404k	240k	53k	544k	486k	56k	664k	617k	56k	786k	711k	56k
DBBS$_g$ (p)	312	45	4653	921	43227	22642	123k	271k	173k	58k	483k	426k	57k	655k	588k	57k	795k	703k	57k
DBBS$_b$ (a)	113	57	688	458	10417	8801	136k	77k	70k	67k	277k	272k	58k	653k	640k	56k	1040k	1013k	52k
DBBS$_b$ (p)	151	42	609	325	**9143**	7075	150k	**72k**	64k	68k	258k	248k	60k	592k	574k	57k	939k	901k	56k
BTB (NBS)	112	64	617	530	11432	10608	130k	83k	81k	64k	271k	269k	57k	539k	520k	54k	754k	748k	51k
BTB (small)	93	52	**538**	439	10026	9311	136k	**72k**	70k	68k	**231k**	229k	58k	**460k**	441k	55k	**670k**	654k	57k
BTB (conn)	191	50	869	**317**	12089	**6725**	145k	83k	**59k**	71k	271k	**208k**	57k	507k	**399k**	56k	700k	**576k**	56k
BTB (DVC)	285	**44**	2096	498	21097	11433	151k	113k	88k	71k	330k	257k	57k	634k	462k	56k	842k	667k	57k

Table 1: Pancake Problem

by the degradation of the heuristic because it accurately selects nodes with minimum lower bound.

In terms of time, in this domain surprisingly BTB is not slower than DBBS, and even BAE* is not substantially faster than either DBBS or BTB. Also, BTB (DVC) is not faster than the rest of the BTB configurations. This is because, in the Pancake Puzzle, solutions are short and the range of values of g, h_x and $h_{\bar{x}}$ is small compared to the number of nodes, which means that the number of buckets is relatively small compared to the total number of nodes in the open lists. All algorithms become slower as they have to expand more nodes due to loss of accuracy of the heuristic, which also explains why, when an algorithm expands significantly more nodes than another (*e.g.* DBBS$_g$ (a) compared to DBBS$_b$ (a) in GAP-2), it is also slower.

In summary, BTB (conn) is the state of the art in this domain except for its bad last-layer (when $C = C^*$) behavior, which makes other algorithms more competitive in terms of total expanded nodes. Nevertheless, algorithms with monotonically non-decreasing C can fix this by implementing a specific last-layer tie-breaker routine (Alcázar, Barley, and Riddle 2019; Alcázar, Riddle, and Barley 2020).

Algorithm	ToH-12 (10+2)			ToH-12 (8+4)			ToH-12 (6+6)		
	$\leq C^*$	$< C^*$	n/s	$\leq C^*$	$< C^*$	n/s	$\leq C^*$	$< C^*$	n/s
BAE* (a)	47k	46k	419k	187k	186k	348k	383k	382k	362k
BAE* (p)	46k	45k	436k	185k	182k	364k	375k	374k	389k
DBBS$_g$ (a)	70k	66k	187k	307k	286k	177k	622k	583k	191k
DBBS$_g$ (p)	57k	53k	199k	232k	221k	192k	493k	477k	217k
DBBS$_b$ (a)	48k	46k	193k	189k	186k	183k	383k	379k	194k
DBBS$_b$ (p)	45k	43k	190k	181k	178k	200k	**371k**	366k	218k
BTB (NBS)	47k	47k	14k	189k	189k	33k	388k	388k	67k
BTB (small)	50k	50k	17k	195k	195k	34k	396k	396k	69k
BTB (conn)	**42k**	**41k**	19k	**180k**	**174k**	44k	375k	**364k**	79k
BTB (DVC)	51k	50k	173k	203k	196k	180k	408k	400k	206k

Table 2: Towers of Hanoi

Table 2 shows results for 50 instances of the **12-disk 4-peg Towers of Hanoi** with (10+2), (8+4) and (6+6) additive PDBs (Felner, Korf, and Hanan 2004). Here BTB (conn) is also the most informed algorithm, but BAE* (p) and DBBS$_b$ (p) are both very close. Again, the other BTB are not as informed, expanding more nodes than front-to-end algorithms despite belonging to a more informed family of algorithms.

In terms of time, the situation is different from the Pancake Puzzle. In this case, BTB (NBS/small/conn) are up to one order of magnitude slower than BAE* and DBBS. On the other hand, BTB (DVC) is as fast as DBBS, showing that computing a vertex cover can indeed save a substantial number of bucket-to-bucket computations.

As for how the analyzed algorithms behave with different heuristics, surprisingly Towers of Hanoi shows the opposite of the Pancake Puzzle. Here, BAE* is slower with the (8+4) PDB, but then becomes slightly faster with the (6+6) PDB. The rest of the algorithms never become substantially slower and indeed the slowest algorithms, BTB (NBS/small/conn), become increasingly faster with weaker heuristics. This suggests that the average number of nodes per bucket goes up when weaker heuristics are employed, but a deeper analysis is needed to find out if that is the case and why it contradicts the findings in the Pancake Puzzle.

Algorithm	15 Puzzle		
	$\leq C^*$	$< C^*$	n/s
BAE* (a)	2707k	2700k	381k
BAE* (p)	2837k	2829k	399k
DBBS$_g$ (a)	22460k	10146k	192k
DBBS$_g$ (p)	3941k	3082k	200k
DBBS$_b$ (a)	2262k	1701k	184k
DBBS$_b$ (p)	**2011k**	1314k	187k
BTB (NBS)	2015k	1901k	31k
BTB (small)	2350k	2214k	31k
BTB (conn)	2231k	**1310k**	35k
BTB (DVC)	2694k	2243k	190k

Table 3: Sliding Tile Puzzle

Table 3 shows results for 100 instances of the **15 Sliding Tile Puzzle** (Korf 1985) with Manhattan Distance. Results are very similar to those of Towers of Hanoi, confirming aforementioned trends. BTB (conn) is the best algorithm in terms of necessarily-expanded nodes, but DBBS$_b$ (p) is almost as good. BTB algorithms are again up to an order of magnitude slower than BAE* and DBBS, but BTB (DVC) is again as fast as DBBS.

Algorithm	Mazes			DAO		
	$\leq C^*$	$< C^*$	n/s	$\leq C^*$	$< C^*$	n/s
A*	99396	99369	1880k	**5406**	**5321**	1537k
BAE* (a)	80835	80809	1229k	6718	6668	1044k
BAE* (p)	**74069**	**74033**	1250k	5995	5861	1192k
DBBS$_g$ (a)	77929	77841	26k	6157	5824	4k
DBBS$_g$ (p)	75771	75728	12k	5601	**5321**	3k
DBBS$_b$ (a)	77946	77889	20k	6105	5828	3k
DBBS$_b$ (p)	77098	77065	10k	5689	5395	2k
BTB (NBS)	77243	77209	3k	5984	5772	217
BTB (small)	86517	86497	830	6005	5879	173
BTB (conn)	-	-	149	-	-	9
BTB (DVC)	86612	86561	24k	5997	5656	6k

Table 4: Grids

Finally, Table 4 shows results for Sturtevant's **grid-based pathfinding** benchmarks (Sturtevant 2012) using the octile heuristic. We used mazes and maps from the video game Dragon Age: Origins (DAO). Diagonal moves are allowed with a cost of 1.5. Because A* is competitive in these domains, we also include its results for reference.

These domains are characterized by having a very large number of buckets with very few nodes, often a single node. Hence, any algorithm that relies on information computed using those buckets will be much slower than heap-based implementations. BTB becomes extremely slow, sometimes expanding only hundreds of nodes per second. In particular, BTB (conn) is so slow that it could not solve the set of instances after several days of computation. BTB (DVC) is again the exception, with a performance similar to DBBS.

Overall, results show that, while BTB is viable in some domains, front-to-end algorithms are on average as good while also being faster and more flexible. Nevertheless, the efficiency of BTB (DVC) shows that, by computing a vertex cover of *DG*$_{MX}$, the main shortcoming of BTB can be alleviated. Additionally, BTB (NBS) shows that front-to-end algorithms are not far from their theoretical limit, as their results are similar or better than BTB (NBS) itself and hence they cannot expand fewer than half of the necessarily-expanded nodes that they currently expand. Nevertheless, these experiments do not offer a final picture of the state of the art in bucket-to-bucket algorithms. For instance, the vertex cover algorithm is a greedy method that yields good results, but it can be improved. In particular, other ways of selecting nodes in BTB may be able to obtain results comparable to BTB (conn) and performance similar to BTB (DVC) in the future.

The Impact of the *rc bounds* The *rc bounds* have been first proposed in this work, so an analysis of their impact is in order. DBBS$_g$ (a) is the same as DBBS (a) by Alcázar *et al.* (2020) except for the *rc bounds*, so we can analyze their impact by comparing them. As expected, the number of necessarily-expanded nodes is similar, as the occasions in which the heuristic value increases along a relevant path are rare in most domains. The biggest difference occurs in Mazes, where the *rc bounds* bring down the number of necessarily-expanded nodes from 78458 to 77841. Mazes are built to misguide the heuristic, and thus the behavior of the heuristic can be erratic, explaining this gain in performance. Still, the *rc bounds* prevent the expansion of only around 1% of the necessarily-expanded nodes, confirming that its utility is limited to a very specific type of graphs.

A Note on the Inefficiency of DBBS After the experiments, the reader may be surprised to realize that the efficiency of BTB (DVC) is similar to that of DBBS despite its expensive bucket-to-bucket computation. However, this is due not only by the speed-up of the vertex cover computation of BTB (DVC), but also to the naive caching scheme of the implementation of DBBS. DBBS performs a fixpoint computation of minimum node values whenever a bucket is removed; however, this may not be always necessary.

Picture the case in which a bucket such that $g_x(n) = gMin_x$ has been emptied and removed. A fixpoint computation of minimum node values is needed only if there are no other expandable buckets such that $g_x(n') = gMin_x$, because otherwise the value of *g bound* will not increase. This is extensible to any other node value. Therefore, one may cache all expandable buckets such that at least one of their node values is minimum and perform the fixpoint computation of minimum values only after all the buckets that determine a minimum node value have been removed. Such a caching scheme is conceptually close to *DG*$_{MX}$ — without requiring a pairwise computation of bounds between buckets —, and thus a similar vertex cover computation may be possible. The formalization and implementation of this caching scheme is left as future work.

Conclusions and Future Work

Here we redefined must-expand pairs for the class of algorithms that exploit consistency, proposed the use of bucket-to-bucket algorithms to obtain near-optimality with respect to the new front-to-end bounds, and linked the bucket-to-bucket computation to the concept of *DG*$_{MX}$. We performed experiments with a broad range of approaches, aiming to test the performance and informedness of the different families of algorithms. Results show that BTB is the state of the art in the Pancake Problem, and that a vertex cover computation of *DG*$_{MX}$ obtained from the bucket-to-bucket computation can achieve a performance close to state-of-the-art bidirectional search algorithms. Also, the near-optimal bucket-to-bucket version shows that front-to-end algorithms are close to their theoretical limit in the tested benchmarks.

In summary, bidirectional search can be seen as a continuum of techniques from front-to-front to bucket-to-bucket to front-to end, and, within front-to-end, from using all individual bounds to a subset thereof. This continuum offers a trade off between informedness and performance, presenting the user with a broad set of options.

As future work we want to formalize the caching schemes that front-to-end algorithms use in their computation of bounds, trying to link the buckets that determine the minimum relative values with concepts like *DG*$_{MX}$. Additionally, we want to improve the performance of front-to-end and bucket-to-bucket algorithms and try alternative configurations of BTB, in particular alternative vertex cover algorithms for the bucket-to-bucket computation.

References

Alcázar, V.; Barley, M.; and Riddle, P. J. 2019. A Theoretical Comparison of the Bounds of MM, NBS, and GBFHS. In *Proceedings of the Twelfth International Symposium on Combinatorial Search, SoCS*, 160–161.

Alcázar, V.; Riddle, P. J.; and Barley, M. 2020. A Unifying View on Individual Bounds and Heuristic Inaccuracies in Bidirectional Search. In *Proceedings of the Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI*, 2327–2334.

Barley, M. W.; Riddle, P. J.; Linares López, C.; Dobson, S.; and Pohl, I. 2018. GBFHS: A Generalized Breadth-First Heuristic Search Algorithm. In *Proceedings of the Eleventh International Symposium on Combinatorial Search, SoCS*, 28–36.

Burns, E. A.; Hatem, M.; Leighton, M. J.; and Ruml, W. 2012. Implementing Fast Heuristic Search Code. In *Proceedings of the Fifth Annual Symposium on Combinatorial Search, SoCS*, 25–32.

Chen, J.; Holte, R. C.; Zilles, S.; and Sturtevant, N. R. 2017. Front-to-End Bidirectional Heuristic Search with Near-Optimal Node Expansions. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI*, 489–495.

Eckerle, J.; Chen, J.; Sturtevant, N. R.; Zilles, S.; and Holte, R. C. 2017. Sufficient Conditions for Node Expansion in Bidirectional Heuristic Search. In *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling, ICAPS*, 79–87.

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

Helmert, M. 2010. Landmark Heuristics for the Pancake Problem. In *Proceedings of the Third Annual Symposium on Combinatorial Search, SoCS*, 109–110.

Holte, R. C.; Felner, A.; Sharon, G.; Sturtevant, N. R.; and Chen, J. 2017. MM: A bidirectional search algorithm that is guaranteed to meet in the middle. *Artif. Intell.* 252: 232–266.

Kaindl, H.; and Kainz, G. 1997. Bidirectional Heuristic Search Reconsidered. *J. Artif. Intell. Res.* 7: 283–317.

Korf, R. E. 1985. Depth-First Iterative-Deepening: An Optimal Admissible Tree Search. *Artif. Intell.* 27(1): 97–109.

Lelis, L. H. S.; Stern, R.; Arfaee, S. J.; Zilles, S.; Felner, A.; and Holte, R. C. 2016. Predicting optimal solution costs with bidirectional stratified sampling in regular search spaces. *Artif. Intell.* 230: 51–73.

Sadhukhan, S. K. 2013. Bidirectional Heuristic Search using Error Estimate. *CSI Journal of Computing* 2(1-2): S1:57–S1:64.

Shaham, E.; Felner, A.; Chen, J.; and Sturtevant, N. R. 2017. The Minimal Set of States that Must Be Expanded in a Front-to-End Bidirectional Search. In *Proceedings of the Tenth International Symposium on Combinatorial Search, SoCS*, 82–90.

Shaham, E.; Felner, A.; Sturtevant, N. R.; and Rosenschein, J. S. 2018. Minimizing Node Expansions in Bidirectional Search with Consistent Heuristics. In *Proceedings of the Eleventh International Symposium on Combinatorial Search, SoCS*, 81–98.

Shperberg, S.; Felner, A.; Sturtevant, N. R.; Hayoun, A.; and Shimony, E. S. 2019. Enriching Non-parametric Bidirectional Search Algorithms. In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence, AAAI*, 2379–2386.

Sturtevant, N. R. 2012. Benchmarks for Grid-Based Pathfinding. *IEEE Trans. Comput. Intellig. and AI in Games* 4(2): 144–148.

Zahavi, U.; Felner, A.; Burch, N.; and Holte, R. C. 2010. Predicting the Performance of IDA* using Conditional Distributions. *J. Artif. Intell. Res.* 37: 41–83.