Artificial Intelligence 252 (2017) 232–266

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

MM: A bidirectional search algorithm that is guaranteed to meet in the middle ☆

Robert C. Holte [a], Ariel Felner [b], Guni Sharon [c], Nathan R. Sturtevant [d], Jingwei Chen [d]

[a] Department of Computing Science, University of Alberta, Edmonton, Canada
[b] Information Systems Engineering, Ben Gurion University, Beer-Sheva, 85104, Israel
[c] Department of Computer Science, University of Texas at Austin, Austin, TX, USA
[d] Department of Computer Science, University of Denver, Denver, CO, USA

ARTICLE INFO

Article history:
Received 17 June 2016
Received in revised form 29 May 2017
Accepted 30 May 2017
Available online 22 August 2017

Keywords:
Heuristic search
Bidirectional search

ABSTRACT

Bidirectional search algorithms interleave two separate searches, a normal search forward from the start state, and a search backward from the goal. It is well known that adding a heuristic to unidirectional search dramatically reduces the search effort. By contrast, despite decades of research, bidirectional heuristic search has not yet had a major impact. Additionally, no comprehensive theory was ever devised to understand the nature of bidirectional heuristic search. In this paper we aim to close this gap. We first present MM, a novel bidirectional heuristic search algorithm. Unlike previous bidirectional heuristic search algorithms, MM's forward and backward searches are guaranteed to "meet in the middle", i.e. never expand a node beyond the solution midpoint. Based on this unique attribute we present a novel framework for comparing MM, A*, and their brute-force variants. We do this by dividing the entire state space into disjoint regions based on their distance from the start and goal. This allows us to perform a comparison of these algorithms on a per region basis and identify conditions favoring each algorithm. Finally, we present experimental results that support our theoretical analysis.

1. Introduction

In a *least-cost path* problem over a *state space* the task is to find a least-cost path from an initial state (*start*) to a goal state (*goal*). *Breadth-first search* and its weighted version *uniform cost search* (Dijkstra's algorithm [12]) are best-first search algorithms designed to solve *least-cost path* problems. They are guided by the cost function $f(n) = g(n)$ where $g(n)$ is the cost of the cheapest known path from *start* to node n. We use the term *unidirectional brute-force search*, denoted Uni-BS, to refer to these algorithms.

The A* algorithm [23] enhances Uni-BS by using $f(n) = g(n) + h(n)$ to prioritize nodes, where $h(n)$ is a heuristic function estimating the cost from n to *goal*. If $h(n)$ is *admissible* (i.e., is always a lower bound) then A* is guaranteed to find optimal (least-cost) solutions. The main purpose of the heuristic function is to focus the search towards the goal. This is depicted in Fig. 1 (left). C^* is the cost of an optimal solution, i.e. the distance from *start* to *goal*. The circle of radius C^* represents

☆ This paper is an invited revision of a paper which first appeared at the AAAI-2016 conference.

E-mail addresses: robert.holte@ualberta.ca (R.C. Holte), felner@bgu.ac.il (A. Felner), gunisharon@gmail.com (G. Sharon), sturtevant@cs.du.edu (N.R. Sturtevant), chenjingwei1991@gmail.com (J. Chen).

http://dx.doi.org/10.1016/j.artint.2017.05.004
0004-3702/

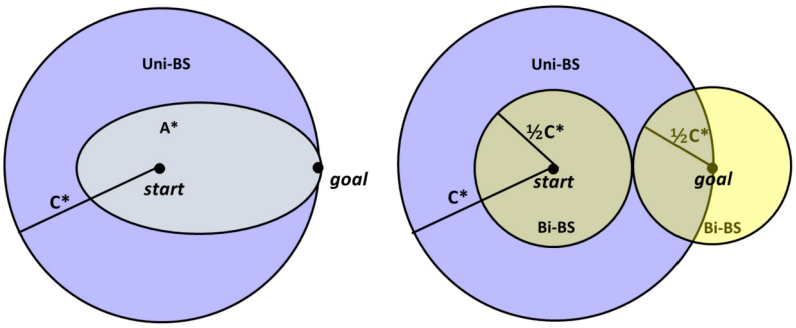

Fig. 1. Uni-BS compared to A* (left) and Bi-BS (right).

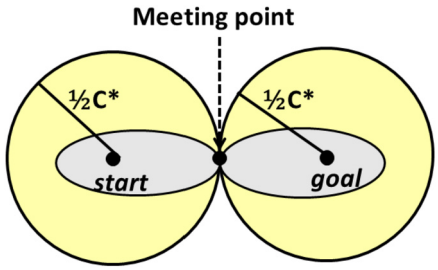

Fig. 2. The circles represent a Bi-BS system that meets in the middle and the ovals represent a Bi-HS that does further pruning.

the states expanded by Uni-BS. The oval inside this circle represents the states expanded by A*. Nodes inside the circle will be expanded by Uni-BS but not by A* if they have a sufficiently large heuristic value (h-value). By using a heuristic A* can reduce the number of nodes expanded by many orders of magnitude compared to Uni-BS. A* and its many variants are therefore commonly used when solving search problems.

A bidirectional search algorithm interleaves two separate searches, a normal search forward from *start*, and a search backward (i.e. using reverse operators) from *goal*. In the backward search $g(n)$ measures the cost of the reverse path from *goal* to n, which is the cost of the forward version of the path from n to *goal*. Every node that has been generated in both directions represents a solution, i.e. a path, possibly suboptimal, from *start* to *goal*. The first solution found is not, in general, optimal, so additional search, or some sort of additional processing, is required after the two searches have met for the first time to ensure the solution returned is optimal (see Section 2 for details). In its simplest form [47], bidirectional brute-force search, denoted Bi-BS, is guided by $g(n)$, just like Uni-BS, i.e. it selects for expansion a node n with minimum $g(n)$ among all the nodes that are open in either search direction. Selecting nodes in this way guarantees that the forward and backward searches "meet in the middle", in the sense that the forward search expands no state whose distance from *start* is greater than $\frac{1}{2}C^*$ and, likewise, the backward search expands no state whose distance to *goal* is greater than $\frac{1}{2}C^*$. This is the definition of "meet in the middle" we will use throughout this paper.[1] The nodes expanded by a Bi-BS system that meets in the middle are depicted by the two smaller circles in Fig. 1 (right). In exponentially growing spaces, a Bi-BS system that meets in the middle can expand exponentially fewer nodes than Uni-BS.

Since A* and Bi-BS speed up Uni-BS in two different ways, it is natural to try to combine them together into *bidirectional heuristic search* (denoted Bi-HS). The expectation in combining them is that their benefits will be compounded. By being bidirectional (as Bi-BS is) Bi-HS will be constrained to only expand nodes around *start* and *goal*. By using a heuristic (as A* does), it will expand a small subset of those nodes. This expectation is depicted graphically in Fig. 2, where the ovals representing nodes expanded by Bi-HS are a small subset of the circles representing nodes expanded by Bi-BS. Pursuit of this idea began 50 years ago [14,15] but to date has produced little success in meeting these expectations.

An intuitive explanation for the failure of Bi-HS to live up to expectations was presented by Barker and Korf [4]. Their analysis resulted in two main conclusions (for caveats, see their Section 3), which we call BK1 and BK2:

- **BK1:** if more than half of the nodes expanded by Uni-HS have $g(n) \leq \frac{1}{2}C^*$, then Uni-HS will expand fewer nodes than Bi-HS.
- **BK2:** If fewer than half of the nodes expanded by Uni-HS using heuristic h have $g(n) \leq \frac{1}{2}C^*$, then adding h to Bi-BS will not decrease the number of nodes it expands.

[1] Meeting in the middle, according to our definition, is not so much about where the searches meet; it is about how far they venture from their starting point.

In other words, Barker and Korf claim that except for pathological cases (see their Section 3) there is no situation in which Bi-HS will be the method of choice; either Uni-HS will be best or Bi-BS will.[2]

A central assumption in Barker and Korf's analysis is that Bi-HS's forward and backward searches meet in the middle, in the sense we have defined. However, no known Bi-HS algorithm is guaranteed to meet in the middle under all circumstances (see Section 3). As a consequence, Barker and Korf's analysis does not immediately apply to existing Bi-HS systems. More importantly, because existing Bi-HS systems are not constrained to search within the two circles in Fig. 2 they can expand nodes that are further than $\frac{1}{2}C^*$ from both *start* and *goal*. For example, in Barker and Korf's Rubik's Cube experiment BS* [40] often expanded nodes at depth 13 in each direction even though C^* was 16 or less.[3] Because of this, it is easy to imagine existing Bi-HS systems expanding more nodes than Bi-BS instead of fewer.

This paper aims to address these issues and makes the following contributions:

1. A new Bi-HS algorithm, MM, along with a formal proof that given an admissible heuristic (not necessarily consistent), MM is guaranteed to meet in the middle and to return an optimal solution.
2. The brute-force version of MM (using $h(s) = 0\ \forall s$), MM_0, is equivalent to Nicholson's Bi-BS algorithm [47] but with an improved termination condition.
3. An enhanced variant of MM called MMe which has an enhanced priority rule for expanding nodes.
4. A version of MM adapted for parallel external-memory search called PEMM. PEMM can solve much larger problems than would be possible in RAM.
5. A new analytical framework that divides the entire state space into disjoint regions based on the distances of nodes from *start* and *goal*. Because MM meets in the middle, we can use this framework to do a careful comparison of MM_0, Uni-BS, MM, and A* on a region-by-region basis. We use this framework to identify conditions under which one method is expected to expand fewer nodes than another. We summarize this analysis by providing general rules, based on certain characteristics of the state space and the relative strength of the heuristic, that predict which algorithm is expected to expand the fewest nodes.
6. The general rules we propose are tested experimentally in three domains, the 10-Pancake puzzle, a grid-based map from the game Dragon Age: Origins, and Rubik's Cube. The first two domains are small enough to allow a fully detailed examination of each algorithm's node expansions in each of the regions we defined. Rubik's Cube is a large state space (approximately 10^{19} states). It does not allow a detailed analysis, but gives confirmation that our general rules continue to make correct predictions at that scale. Our experiments with Rubik's Cube are the first experiments using general-purpose search methods to solve problems with solutions at the greatest depth possible ($C^* = 20$).
7. We show that Bi-HS is fundamentally different than Uni-HS. With a consistent[4] non-zero heuristic ($h(n) \neq 0$ for every non-goal node) Uni-HS cannot possibly expand more nodes than Uni-BS (Result 6, p. 81 [49]). The corresponding statement does not hold for bidirectional search. We present an example in which MM, or any Bi-HS algorithm guided by $f(n)$, expands more nodes than MM_0, and witness this occurring in our experiments.

Although we introduce a new algorithm (MM), we do not claim that MM_0 or MM are the best bidirectional search algorithms in terms of minimizing run time or the number of nodes expanded under all circumstances. MM's significance is that it is the only Bi-HS algorithm that is guaranteed to meet in the middle. This has numerous benefits (see Section 3.3), one of which is that our analysis, and Barker and Korf's, applies to MM. These theories give strong justification for bidirectional search algorithms that meet in the middle. As the first of its breed, MM represents a new direction for developing highly competitive bidirectional heuristic search algorithms.

A preliminary version of this paper appeared in AAAI-2016 [29]. The MMe variant of MM was presented at SoCS-2016 [59] and the PEMM variant was presented at IJCAI-2016 [61]. In addition to drawing together the material in those papers, the current paper extends them in the following ways. First, it provides a deeper study of the algorithm and a deeper analysis of the new framework. Second, it includes all the theorems about MM with full proofs (Appendix A). Third, it contains a comprehensive discussion of previous work on bidirectional search. Finally, a more thorough experimental section (including more domains) is provided that supports the theoretical claims.

2. Terminology and previous work

A *problem instance* is a pair (*start*, *goal*) of states. If x and y are states, with y a successor of x, then $cost(x, y)$ is the cost of the edge from x to y. We assume all edge costs are non-negative (a cost of 0 is permitted). The aim of search is to find a *least-cost path* from *start* to *goal*. For any two states, u and v, $d(u, v)$ is the distance (cost of a least-cost path) from u to v. $C^* = d(start, goal)$ is the cost of an optimal solution. Unless otherwise stated, we assume the heuristics used are admissible but not necessarily consistent.

[2] Barker and Korf's theory was further discussed in our conference paper [29] where we compared it to our new theoretical findings. The interested reader is referred to that paper.

[3] Personal communication, Joseph Barker to R. Holte, July 3, 2015.

[4] A heuristic h is *consistent* if for any two states, n and m, $h(n) \leq d(n, m) + h(m)$ where $d(n, m)$ is the cost of a least-cost path from n to m.

We use the usual notation—$f, g, Open$, etc.—and use $gmin$ and $fmin$ for the minimum g- and f-value on $Open$. For bidirectional search algorithms, we have separate copies of these variables for each search direction, with a subscript (F or B) indicating the direction[5]:

- **Forward search:** f_F, g_F, h_F, $Open_F$, $Closed_F$, $gmin_F$, etc.
- **Backward search:** f_B, g_B, h_B, $Open_B$, $Closed_B$, $gmin_B$, etc.

There are three main algorithmic design decisions on which bidirectional search algorithms differ:

- stopping condition
- selecting which node to expand
- the nature of the heuristic used (for Bi-HS)

The alternatives used for these design decisions in existing bidirectional search systems are discussed individually in the following subsections.

2.1. Stopping condition

For Bi-BS there are two different stopping conditions that guarantee optimal solutions are returned, Nicholson's [47] and Dreyfus's [16].[6] In our notation, Nicholson's condition terminates the search when there is a node n closed in both directions such that $g_F(n) + g_B(n) \leq gmin_F + gmin_B$. In Section 4.2 we will give two improvements to this stopping condition. We use the name "$gmin$ stopping condition" to refer to any stopping condition of the form $U \leq gmin_F + gmin_B$, where U is the cost of some solution (path from *start* to *goal*) that the search has so far found. Nicholson's stopping condition is a $gmin$ stopping condition in which U is the minimum cost solution path passing through a node closed in both directions. Dreyfus [16] showed that the solution path created by the first node that becomes closed in both directions is not necessarily optimal.[7] Dreyfus further observed that search can indeed stop as soon as the first node n is closed in both directions, but must be followed by some final processing. Once a node n is closed in both directions the optimal path is either that path through n or a path through some node that is currently closed in one direction and open in the other. Those nodes need to be inspected to see if they form a cheaper path than the one through n. Helgason et al. [26] refer to this final checking for optimal paths after search has stopped as the "mop-up" phase. Variations of Dreyfus's stopping condition have been used by several authors [21,30,43].

For Bi-HS, Pohl (p. 92 [53]) showed that Dreyfus's stopping condition is not correct when generalized to Bi-HS and proposed a stopping condition more like Nicholson's, but based on f not g. The search algorithm keeps track of the cost of the cheapest path from *start* to *goal* it has found so far—we use the variable U for this value—and stops when $U \leq \max(fmin_F, fmin_B)$. We call this the $fmin$ stopping condition. Most Bi-HS algorithms to date have used Pohl's $fmin$ stopping condition, the only difference being that some update U only when a node becomes closed in both directions (e.g. [3,8,46,53]) while others update U when a node becomes open in both directions (e.g. [22,38,40,52,64]). Barker and Korf [3] were the first to recognize that Bi-HS could use a $gmin$ stopping condition in addition to Pohl's $fmin$ stopping condition. An entirely different approach to terminating a Bi-HS is used in "two-phase" systems [33,55]. These stop their bidirectional search as soon as a node is open in both directions, choose a search direction, and proceed thenceforth with a unidirectional heuristic search until the usual Uni-HS stopping condition is satisfied. These can be regarded as generalizing the "mop-up" phase associated with Dreyfus's Bi-BS stopping condition.

2.2. Selecting the next node to expand

For Bi-BS there are two main methods for deciding which node to expand next. The most common method is to strictly alternate between the search directions: expand a node with a minimum g_F-value in the forward direction, then one with a minimum g_B-value in the backward direction, etc. [21,26,30,43]. This method results in the number of node expansions in each search direction being within one of each other, but it also means they might expand a node in one search direction even though there is an open node with a strictly smaller g-value in the other direction. By contrast, Nicholson's algorithm [47] selects a node with the smallest g-value in either search direction and has no explicit policy (such as strictly alternating search direction) for changing search direction from time to time. Indeed, if all the edges leading to *goal* from its predecessors cost more than $\frac{1}{2}C^*$, Nicholson's selection policy will result in *goal* being the only node expanded in the backward direction, all the rest of the search will done in the forward direction. Nicholson's algorithm also differs from all other Bi-BS algorithms in that it expands all nodes with the minimum g-value in the chosen direction at once; other methods expand one node at a time.

[5] We define "forward" to be the direction in which Uni-HS searches.

[6] Pohl (p. 13 [53]) gives a full description of this stopping condition and credits it to Dreyfus, citing the August 1967 version of Dreyfus's unpublished technical report, but it is not present in the October 1967 version [16], the earliest version we have been able to obtain.

[7] This had incorrectly be used as a stopping condition by Berge [5].

For Bi-HS, Pohl (p. 96 [53]) proposed that the next node to be expanded be chosen in two steps—(1) choose a search direction, and then (2) expand a node with the minimum f-value in the chosen direction—and proved that a Bi-HS algorithm using his stopping condition would return optimal solutions no matter how the search direction is selected in step (1). He then defined the "cardinality criterion" for choosing a search direction: choose the search direction whose Open list is smaller. Pohl gave extensive arguments in favor of this criterion and almost all Bi-HS algorithms have used it. Auer and Kaindl's BiMax-BS$^*_F$ [2] applies the cardinality criterion to choose a direction, but then expands all nodes in that direction with the minimum f-value, even newly generated ones. Kowalski (p. 186 [38]) proposed a variation: choose the search direction having the smaller number of nodes in its open list with the minimum f-value. Similarly, Barker and Korf [3] expand an entire f-level in one direction and then choose the direction, for the next round of expansions, that did the fewest node expansions when it was last used.

Other selection policies are oblivious to the size and contents of the open lists and have simple switching policies such as strict alternation between the two directions [1,51,58] or switching to maintain a fixed ratio between the number of nodes expanded in the two directions [64]. An extreme policy of this form is perimeter search [13,32,41,44], which begins by doing a fixed amount of search in the backward direction and then does all the remaining search in the forward direction.

The Bi-HS analog of Nicholson's selection policy is to select a node with the smallest f-value in either search direction. Despite its simplicity, it has been used in only one Bi-HS algorithm [52].

2.3. Heuristics for bidirectional heuristic search

The heuristics (h_F and h_B) used in Bi-HS can either be static or dynamic. A static heuristic (also called a front-to-end heuristic [32]) is the kind of heuristic used by Uni-HS: it directly estimates the distance from node n to the target of the search (*goal* is the target of the forward search, *start* is the target of the backward search). Most Bi-HS systems use static heuristics.

A dynamic heuristic also estimates the distance from a node n to the search target, but it takes into account information generated by search in the opposite direction, so its value for a state may change as search proceeds. The first dynamic heuristics were the "front-to-front" heuristics introduced by de Champeaux and Sint [10,11] and later used by other "front-to-front" Bi-HS systems [1,8,9,18,54] and some perimeter search algorithms [13,44]. These estimate the distance from n to the search target indirectly, using a function $h(n,m)$ that estimates the distance between any two nodes. Given $h(n,m)$, the front-to-front heuristic for forward search is $h_F(n) = \min_{m \in Open_B}\{h(n,m) + g_B(m)\}$ (h_B is defined analogously).

Front-to-front heuristics are computationally expensive to use, leading Kaindl and Kainz [32] to develop two inexpensive alternatives, which they called Add and Max. These were developed in the context of perimeter search, where they were static adjustments only applicable to h_F. Later work showed that they could be applied in both search directions and no matter how the search direction is selected [2,31,51,64].

Single-frontier Bidirectional Search (SFBDS) [19,42], called BDS2 by Eckerle and Ottmann [18], also uses a heuristic $h(n,m)$ that estimates the distance between any two nodes, but in a different way than front-to-front search. In SFBDS each node in the search tree is a pair of states (n,m) whose successors are either of the form (n',m) or of the form of (n,m'), where n' is a successor of n in the forward direction and m' is a successor of m in the backward direction. The root node of SFBDS's search tree is $(start, goal)$ and any node of the form (x,x) is a goal node. The g-value of node (n,m) is $g_F(n) + g_B(m)$ and its h-value is $h(n,m)$. Bidirectional search in the original state space is simulated by applying any Uni-HS system to this expanded state space. The core idea underlying SFBDS was first described in 1966 in an unpublished report [14]. Complete pseudocode for a SFBDS-like system, BHFFA2, was published in 1975 [10] (with an incorrect stopping condition) but never implemented.

3. Meeting in the middle

Recall the definition of "meet in the middle" that we use throughout this paper.

Definition 1 *(Meeting in the middle).* A bidirectional search algorithm *meets in the middle* if its forward search never expands a node n with $g_F(n) > \frac{1}{2}C^*$ and its backward search never expands a node n with $g_B(n) > \frac{1}{2}C^*$.

3.1. Previous Bi-HS algorithms fail to meet in the middle

We now show that previous Bi-HS algorithms fail to meet in the middle on the graph in Fig. 3. All edges cost 1 and the optimal path is $start, A, B, C, D, goal$ ($C^* = 5$). Inside each node are its h_F-value (with a right-pointing arrow overhead) and its h_B-value (with a left-pointing arrow overhead). The heuristic values in each direction are consistent.

Any search algorithm that meets in the middle—Nicholson's algorithm [47], for example, or the MM algorithm we will define in Section 4—will expand $start$, A, B and perhaps some (or all) of the X_i in the forward direction and C, D, and $goal$ in the backward direction. In particular, B will certainly not be expanded in the backward direction. We will now show that existing Bi-HS systems will expand B in the backward direction, and therefore do not meet in the middle according to our definition.

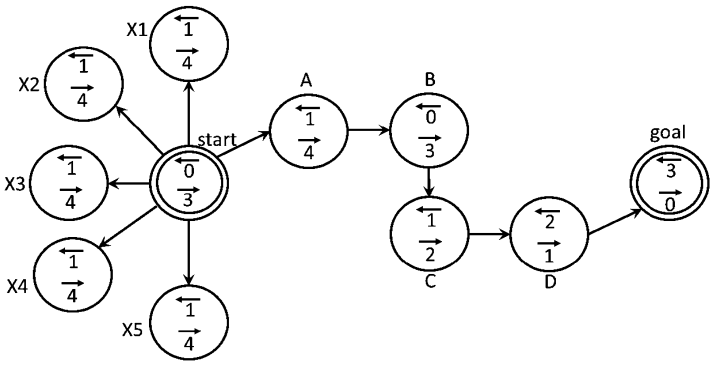

Fig. 3. Graph on which previous Bi-HS systems fail to meet in the middle.

3.1.1. Cardinality criterion variants

Systems that use the cardinality criterion will only expand *start* in the forward direction. After that $Open_F$ is always strictly larger than $Open_B$ so all further search will be in the backward direction. Kowalski's variation on the cardinality criterion does not change this behavior since all the nodes in $Open_F$ have the same f_F-value. With Barker and Korf's variation of the cardinality criterion *start* will be expanded in the forward direction, then *goal*, D, C, and B will be expanded in the backward direction because they all have the same f_B-value ($f_B = 3$).

3.1.2. Alternation or smallest f

Systems that strictly alternate search direction will expand B in their backward search because their forward search will be pre-occupied expanding the X_i nodes. Systems that expand a node with the smallest f-value will expand B in the backward direction before expanding A or any of the X_i in the forward direction because $f_B(B) = 3 < f_F(A) = f_F(X_i) = 5$.

3.1.3. Front-to-front systems

Front-to-front heuristics were introduced specifically to remedy the problem that Bi-HS systems using static heuristics were not meeting in the middle [10,11].[8] Some papers with front-to-front heuristics claim their searches meet in the middle [8,10,11] but none has a theorem to this effect. The example in Fig. 3 can be adapted to show that Bi-HS with a front-to-front heuristic will expand B in the backward direction if its policy for selecting nodes to expand is to alternate search directions and then to select a node with the minimum f-value in the chosen direction. This is the exact policy used by Arefin and Saha [1] and is a permissible policy for other front-to-front Bi-HS systems [8–11,18] since they are indifferent to how the search direction is chosen. The front-to-front heuristic for this example has $h(X_i, C) = h(X_i, D) = 1$ and $h(A, C) = h(A, D) = 2$ (the other values do not matter, they can be set in several ways to make h admissible and bi-monotone [17]). With this heuristic $f_F(A)$ is strictly larger than $f_F(X_i)$ when $Open_B = \{D\}$ and when $Open_B = \{C\}$, so the search will be proceed exactly as described above for alternating search and B will expanded in the backward direction.

3.2. Transforming heuristic search into brute-force search

The only previously existing bidirectional search algorithm guaranteed to meet in the middle is therefore Nicholson's [47]. It is not a heuristic search algorithm, but, when h_F and h_B are consistent, it can be made to exactly simulate heuristic search by transforming the edge costs in the state space to take into account the heuristic values, as follows [30]. If u and v are nodes, with v a successor of u (in the forward direction), redefine the edge cost $cost(u, v)$ to be:[9]

$$cost'(u, v) = cost(u, v) + \frac{1}{2}(h_B(u) + h_F(v) - (h_F(u) + h_B(v))). \tag{1}$$

When Nicholson's algorithm is run using the transformed edge costs, it is a Bi-HS algorithm that is guaranteed to meet in the middle. Unfortunately, the "middle" at which it meets is with respect to the transformed edge costs not the original ones. For example, if this transformation is applied to Fig. 3, it leaves all the edges emanating from *start* with a cost of 1 but changes the costs of edges (B, C), (C, D), and $(D, goal)$ to zero. When Nicholson's algorithm is run on this transformed graph it will expand D, C, and B in the backward direction before expanding A or any X_i in the forward direction because the latter have $g_F = 1$ while the former have $g_B = 0$. As it must be, this is exactly the same as the behavior on Fig. 3, described above, of a Bi-HS system that selects nodes for expansion based on the minimum f-value.

[8] It is not clear if de Champeaux and Sint [10,11] were using "meet in the middle" in the sense we have defined or in some other sense.

[9] The justification of this formula is non-trivial. It is fully explained in the original paper [30].

3.3. Why meet in the middle?

Why is it important for a bidirectional search algorithm to "meet in middle" as we have defined it? There are several reasons:

1. The original motivation for Bi-HS was to be able solve a problem whose optimal solution cost was C^* doing only twice the work it would take A^* to solve a problem whose solution cost was $\frac{1}{2}C^*$ (p. 108 [53]). Meeting in the middle is directly aimed at achieving this goal.
2. Meeting in the middle provides an upper bound on how many nodes a Bi-HS system will expand in the worst case and an upper bound on how much memory it will need. In state spaces where the number of states at distance d from *start* or *goal* grows exponentially with d, a system that ventures beyond $d = \frac{1}{2}C^*$, whether it be bidirectional or unidirectional, is at risk of expanding exponentially more nodes than a system that meets in the middle.
3. Meeting in the middle provides a characterization of nodes that are guaranteed **not** to be expanded, analogous to the fact that A* is guaranteed not to expand nodes with $f(n) > C^*$ (Lemma 4 [24]). This characterization enables us to partition the state space into disjoint regions and then provide an analytical comparison of bidirectional and unidirectional searches on a region-by-region basis (see Section 6). It also enables other analyses to be applied. For example, Barker and Korf's analysis [4] only applies to systems that meet in the middle.
4. Meeting in the middle guarantees that a state expanded in one direction will *not* be expanded in the other direction unless it is exactly distance $\frac{1}{2}C^*$ from both *start* and *goal*. In state spaces where no such states exist (e.g. unit-cost state spaces when C^* is odd) a system that meets in the middle does not need consistent heuristics or special mechanisms, such as Kwa's "nipping" and "pruning" [40], to prevent states from being expanded in both directions.

4. MM: a novel Bi-HS family of algorithms

In this section we describe MM, our new Bi-HS algorithm. We first describe the original version [29], which we refer to as "basic MM", and then describe an improved version, called MM∈ [59].

4.1. Basic MM

Basic MM[10] runs an A*-like search in both directions, except that MM orders nodes on the Open list in a novel way. The priority of node n on $Open_F$, $pr_F(n)$, is defined to be:

$$pr_F(n) = \max(f_F(n), 2g_F(n)). \tag{2}$$

$pr_B(n)$ is defined analogously. We use $prmin_F$ and $prmin_B$ for the minimum priority on $Open_F$ and $Open_B$, respectively, and $C = min(prmin_F, prmin_B)$. On each iteration MM expands a node with priority C.

When a state s is generated in one direction MM checks whether s is in the Open list of the opposite direction. If it is, a solution (path from *start* to *goal*) has been found. MM maintains the cost of the cheapest solution found so far in the variable U. U is initially infinite and is updated whenever a better solution is found. MM stops when

$$U \leq \max(C, fmin_F, fmin_B, gmin_F + gmin_B + \epsilon) \tag{3}$$

where ϵ is the cost of the cheapest edge in the state space.

Each of the terms inside the max is a lower bound on the cost of any solution that might be found by continuing to search. Therefore, if U is smaller than or equal to any of them, its optimality is guaranteed and MM can safely stop.

4.1.1. Pseudocode for MM

Algorithm 1 gives the pseudocode for MM. When $prmin_F = prmin_B$ any rule could be used to break the tie (e.g. Pohl's cardinality criterion [53]), it is not necessary to break such ties in favor of the forward direction as is done in line 9. In addition, tie breaking within a given direction should be performed in Line 11. Lines 5–23 are the usual *best-first search* expansion cycle. Duplicate detection is done in line 14. U is updated in line 21 and checked in line 7. Note that to determine if a better solution path has been found, MM only checks (line 20) if a newly generated node is in the Open list of the opposite search direction. That is all that is required by our proofs; it is not necessary to also check if a newly generated node is in the Closed list of the opposite search direction. We introduce the term *solution detection* for this check.

As presented, only the cost of the optimal path is returned (line 8). It is straightforward to add code to return the solution path.

[10] MM is used to denote both the entire family of algorithms using some variant of the priority function, stopping conditions, and tie-breaking described here as well as the specific basic variant. This is similar to A* which is used to denote both a general family of algorithms as well as the basic implementation.

Algorithm 1: Pseudocode for MM

```
1  g_F(start) := g_B(goal) := 0;
2  Open_F := {start};
3  Open_B := {goal};
4  U := ∞
5  while (Open_F ≠ ∅) and (Open_B ≠ ∅) do
6      C := min(prmin_F, prmin_B)
7      if U ≤ max(C, fmin_F, fmin_B, gmin_F + gmin_B + ε) then
8          return U
9      if C = prmin_F then
10         // Expand in the forward direction
11         choose n ∈ Open_F for which pr_F(n) = prmin_F
12         move n from Open_F to Closed_F
13         for each child c of n do
14             if c ∈ Open_F ∪ Closed_F and g_F(c) ≤ g_F(n) + cost(n, c) then
15                 continue
16             if c ∈ Open_F ∪ Closed_F then
17                 remove c from Open_F ∪ Closed_F
18             g_F(c) := g_F(n) + cost(n, c)
19             add c to Open_F
20             if c ∈ Open_B then
21                 U := min(U, g_F(c) + g_B(c))
22     else
23         // Expand in the backward direction, analogously
24 return ∞
```

4.1.2. Properties of MM

When MM's heuristics are admissible, it has the following properties:

(P1) MM's forward and backward searches meet in the middle, i.e. neither search expands a node whose distance from the search's origin ($g_F(n)$ for forward search, $g_B(n)$ for backward search) is larger than $\frac{1}{2}C^*$.
(P2) MM never expands a node whose f-value exceeds C^*.
(P3) MM returns C^*.

These are formally stated and proven in Appendix A. Here we provide sketches of their proofs based on the following lemmas (L1–L3). For simplicity, in these sketches we assume MM stops if and only if $U \leq C$.[11]

L1: If $d(start, s) > \frac{1}{2}C^*$, then $pr_F(s) > C^*$ and if $d(s, goal) > \frac{1}{2}C^*$, then $pr_B(s) > C^*$.

Proof for the forward direction. $pr_F(s) \geq 2g_F(s) \geq 2d(start, s)$. If $d(start, s) > \frac{1}{2}C^*$ then $pr_F(s) > C^*$. □

For the next lemma we need the following definition.

Definition 2. For any optimal path $P = s_0, s_1, \ldots s_n$ from *start* (s_0) to *goal* (s_n), let i be the largest index such that $s_k \in Closed_F\ \forall k \in [0, i-1]$, and let j be the smallest index such that $s_k \in Closed_B\ \forall k \in [j+1, n]$. We say that P "has not been found" if $i < j$ and that P "has been found" otherwise ($i \geq j$).

For example, if i and j, as defined in Definition 2, are as shown in Fig. 4 then this path has not been found because $i < j$ (s_i is to the left of s_j).

In the proof of the following lemma, i and j are as defined in Definition 2.

L2: If P is an optimal path from *start* to *goal* that has not been found, there will exist a node $n \in P$ such that either $n \in Open_F$ with $pr_F(n) \leq C^*$ or $n \in Open_B$ with $pr_B(n) \leq C^*$.

[11] Additional stopping conditions may improve MM's running time but they do not affect these properties (see Section A.4 in Appendix A).

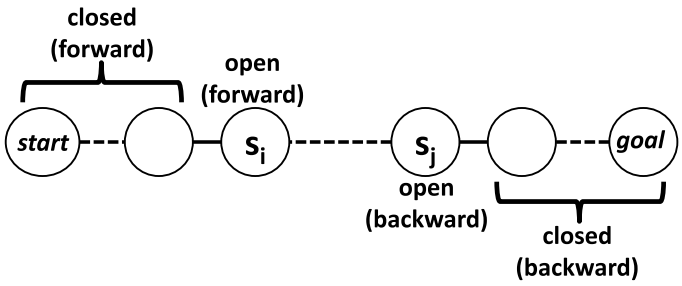

Fig. 4. The gap on an optimal path that has not yet been found.

Proof Sketch. Throughout MM's execution there will be a node in P, s_i,[12] in $Open_F$ with $g_F(s_i) = d(start, s_i)$ and a node in P, s_j, in $Open_B$ with $g_B(s_j) = d(s_j, goal)$. Since P has not yet been found there must exist a gap between s_i and s_j, i.e. one or more edges from P that connect s_i to s_j that MM has not traversed. This situation is depicted in Fig. 4, where the dashed line between s_i and s_j is the gap consisting of one or more edges. In this situation, either $g_F(s_i) \leq \frac{1}{2}cost(P)$ or $g_B(s_j) \leq \frac{1}{2}cost(P)$ (or both), where $cost(P)$ is the sum of the costs of P's edges. Therefore, $pr_F(s_i) \leq cost(P) = C^*$ or $pr_B(s_j) \leq cost(P) = C^*$ (or both). □

L3: $U > C^*$ until the first optimal path from *start* to *goal* is found, at which point $U = C^*$. This is a direct consequence of the process by which U is updated.

We now sketch the proofs that MM has properties P1–P3. L2 and L3 together ensure that MM will not terminate before an optimal path has been found (L2 implies that $C \leq C^*$ until all optimal paths have been found and L3 says $U > C^*$ until the first optimal path is found). P3 follows because $U = C^*$ once an optimal path has been found (L3).

L2 and L1 together ensure that MM will find an optimal path before MM expands any node in the forward direction with $pr_F(n) > C^*$ or any node in the backward direction with $pr_B(n) > C^*$. Together with L3 this implies that MM will terminate before it expands any node in the forward direction with $f_F(n) > C^*$ or $d(start, s) > C^*/2$, or any node in backward direction with $f_B(n) > C^*$ or $d(s, goal) > C^*/2$, thus proving P1 and P2.

4.2. MM_0

MM_0 is the brute-force version of MM, i.e. MM when $h(n) = 0\ \forall n$. Thus for MM_0: $pr_F(n) = 2g_F(n)$ and $pr_B(n) = 2g_B(n)$. MM_0 therefore selects for expansion a node on either open list with the smallest g-value, just as Nicholson's algorithm [47] does, and stops when $U \leq gmin_F + gmin_B + \epsilon$. MM_0's stopping condition is superior to Nicholson's in two ways: (1) Nicholson's does not have the $+\epsilon$ term, and (2) Nicholson's algorithm only updates U when a node is closed in both directions, MM_0 updates U when a node becomes open in both directions. A third difference is that Nicholson's algorithm only checks the stopping condition after expanding all the nodes in the chosen search direction with the minimum g-value whereas MM_0 checks the stopping condition after every node expansion. We return to this issue in Section 5, where we discuss immediate and delayed solution detection.

4.3. MMe

We now present an enhanced version of MM, MMe [59], which is identical to basic MM except for a small change in how an open node's priority is defined. In all subsequent sections of the paper we will use the pr_F and pr_B to refer to MMe's definition of priority, but in this section, to clearly distinguish between basic MM's priority function and MMe's, we will use pr_F and pr_B for basic MM's priority function and use the special notation pr^ϵ_F and pr^ϵ_B for MMe's priority function.

In MMe the priority of $n \in Open_F$ is

$$pr^\epsilon_F(n) = \max(f_F(n), 2g_F(n) + \epsilon) \tag{4}$$

where ϵ is the cost of the cheapest edge in the state space. $pr^\epsilon_B(n)$ is defined analogously.

We now prove that MMe has properties P1–P3 by showing that lemmas L1–L3 hold for MMe. L1 is still true since $pr^\epsilon_F(n) \geq pr_F(n)$ and $pr^\epsilon_B(n) \geq pr_B(n)$. L3 is still true because it is not affected by the definition of a node's priority. To see that L2 is still true, note that $d(s_i, s_j)$ is the cost of the gap illustrated in Fig. 4, i.e. $C^* = g_F(s_i) + d(s_i, s_j) + g_B(s_j)$. Hence, at least one of $g_F(s_i)$ and $g_B(s_j)$ must be less than or equal to $\frac{1}{2}(C^* - d(s_i, s_j))$.

[12] Lemma 17 proves that *goal* will never be closed in the forward direction and *start* will never be closed in the backward direction, so i and j are both always legal indexes of nodes in P.

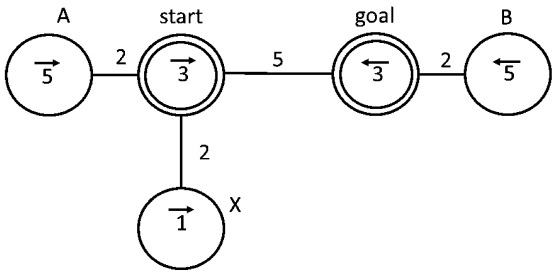

Fig. 5. Basic MM expands more nodes than MMe.

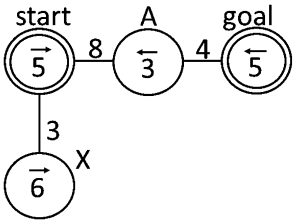

Fig. 6. MMe expands more nodes than basic MM.

The exact value of $d(s_i, s_j)$ is not known, but it always holds that $\epsilon \leq d(s_i, s_j)$.[13] Therefore, either $g_F(s_i) \leq \frac{1}{2}(C^* - \epsilon)$ or $g_B(s_j) \leq \frac{1}{2}(C^* - \epsilon)$. If $g_F(s_i) \leq \frac{1}{2}(C^* - \epsilon)$ then $2g_F(s_i) + \epsilon \leq C^*$. Similar reasoning applies for $g_B(s_j)$. L2 follows because at least one of these must hold.

4.3.1. MM *vs.* MMe

In this section we highlight the differences in behavior of basic MM and MMe with two examples. The first (Fig. 5) represents situations in which basic MM expands nodes that MMe does not expand. This is what is expected, given that $pr_F(n) \leq pr_F^{\epsilon}(n)$. The second example (Fig. 6) shows that the opposite behavior is also possible, there can be nodes that are expanded by MMe but not by basic MM.

- Basic MM expands nodes that MMe does not expand.
  Consider the graph depicted in Fig. 5 with a specific focus on node X. Basic MM proceeds as follows. After *start* is expanded $Open_F$ includes three nodes with the following priorities: $A(7)$, $X(4)$ and $goal(10)$. At this point U is set to 5. Now $goal$ ($pr_B = 3$) is expanded in the backward direction. Finally, X is the only node with $pr_F(X) = 4 \leq U = 5$ so it is expanded. By contrast, for MMe, $pr_F(X) = 6 > U = 5$ and it will not be expanded.
- MMe expands nodes that Basic MM does not expand.
  It is possible for MMe to expand more nodes than basic MM. Fig. 6 gives an example when both algorithms use all of MM's stopping conditions (line 7 in Algorithm 1). Both algorithms begin by expanding *start* (forward) and *goal* (backward). Node A will not be expanded in the forward direction by either algorithm because $g_F(A) = 8 > 6 = \frac{1}{2}C^*$ and both algorithms will halt as soon as A is expanded in the backward direction (when that happens $U = 12 \leq gmin_F + gmin_B + \epsilon = 18$. For both algorithms $pr_F(X) = f_F(X) = 9$. For MM, $pr_B(A) = 8$ so MM will expand A before X and then halt without expanding X. For MMe, $pr_B(A) = 12$ so MM will expand X before A.

Which of these two situations will occur more commonly? We expect the first situation (Fig. 5) will occur much more often than the second (Fig. 6), and therefore expect that MMe will usually expand fewer nodes than basic MM. Our reason is that the second situation requires the priority of at least one node on *every* optimal path to have its priority increased to be greater than the priority of nodes like X, a kind of collective conspiracy on the part of the optimal paths. By contrast, the first situation occurs when an individual node's priority increases beyond C^*.

5. Parallel external-memory MM

In order to scale the size of the problems solvable by MM, we must either purchase the largest possible machine for solving problems or make better use of the resources on available hardware. The most common approach is to use external

[13] Any lower bound on $d(s_i, s_j)$ can be used where we are using ϵ. For example, if $\epsilon_F(n)$ ($\epsilon_B(n)$) is the cost of the cheapest forward (reverse) operator applicable to n then $\epsilon_F(s_i) \leq d(s_i, s_j)$ ($\epsilon_B(s_j) \leq d(s_i, s_j)$) and can be used here instead of ϵ. Similarly, if $\delta(x, y)$ is a lower bound on the number of edges between x and y, then $\min_{y \in Open_B} \epsilon\delta(s_i, y)$ can be used here instead of ϵ.

memory (disk) as storage [6,36,48,39,63] in place of RAM. While disks are much larger than RAM, the cost of random access to disk is high (high latency). The throughput of disk, however, is also high, so once the latency is overcome, disks have relatively high throughput. Thus, if we wish to modify MM, or any other bidirectional search algorithm, to use external memory, we must modify the algorithm to amortize high latency operations by grouping operations that access the same data on disk.

Looking at Algorithm 1, there are three places where MM performs random access to the *Open* and *Closed* lists. In lines 14 and 16 MM performs duplicate detection, checking if a newly generated state has already been generated or expanded. In line 20 MM performs solution detection, checking if a solution has been found.

The standard approach to avoiding the latency associated with duplicate detection in external memory is delayed duplicate detection (DDD) [35]. DDD avoids duplicate detection on individual states, instead performing it on many states at a time later in the search. In DDD successors are written to an open list on disk without duplicate detection. At a later time, such as when all successors have been written to a file, duplicate detection can be performed in batches, amortizing the disk latency.

Approaches for avoiding random access to disk for solution detection have not been previously been studied in external memory search beyond our own work on MM [61], which is expanded on slightly in this presentation. MM by default uses immediate solution detection (ISD), checking for solutions as soon as a state is generated. Delayed solution detection (DSD) refers to any approach that does not perform solution check immediately, but does perform the check before any state with larger f-cost is expanded.

We describe here DSD and our other modifications to MM to create PEMM, the parallel, external-memory version of MM.

5.1. Algorithmic changes to MM

PEMM makes the following changes to MM to ensure the efficiency of the search. While these are all part of our implementation, the core change required for PEMM is the use of DDD, DSD, and the change of termination conditions. It is likely that other changes could be made to further increase the search efficiency without fundamentally changing the nature of PEMM:

1. PEMM separates the states in *Open* and *Closed*, which are stored on disk, from the information about those states, which can be summarized in smaller *Open* and *Closed* data structures in RAM.
2. PEMM partitions *Open* and *Closed* on disk into buckets of states with similar properties for the efficiency of duplicate and solution detection.
3. PEMM expands states with the same priority from low to high g-cost.
4. PEMM uses delayed duplicate detection for finding duplicates.
5. PEMM uses delayed solution detection for finding solutions.
6. PEMM removes ϵ from the *gmin* stopping condition and uses Basic MM's definition of $pr(n)$, because the ϵ-based variants do not work with delayed solution detection.
7. PEMM performs many operations in parallel.
8. PEMM assumes a consistent heuristic, an undirected search space, and unit-cost edges.

These changes are now described in detail. One change not discussed or studied here is the performance of PEMM with inconsistent heuristics. The re-opening of nodes may or may not cause problems with the efficiency of external-memory search; we do not study this issue here, but leave it as a point for future research.

5.1.1. States are not stored in Open and Closed in RAM

When using external memory, the assumption is that all states in the search will not fit in RAM at once. Thus, we maintain separate *Open* and *Closed* lists in memory and on disk. The lists in memory only contain the properties of the states; the actual states are kept on disk. While efficient data structures can be used to access and query this information in RAM, in practice a simple (resizable) array is sufficient, since the cost of iterating through *Open* in RAM is dwarfed by the cost of loading and processing states on *Open* from disk.

Individual states are stored unsorted on *Open* and *Closed* on disk, so writes can be efficiently performed at any time by appending to a file on disk. Writes are buffered both in memory and by the filesystem, which improves efficiency and eliminates latency concerns when writing.

5.1.2. States in Open and Closed are broken into buckets on disk

Since we cannot load all of *Open* into RAM at once, it must be subdivided into smaller buckets that will fit into RAM so that we can load these files when doing duplicate detection.

States are divided into buckets by (1) the priority of a state, (2) the g-cost of the state, (3) the search direction, and (4) the lower i bits of the state hash function. Every state in a given bucket will have the same values for each of these attributes. This means that all duplicates in the same direction will fall in the same bucket, and nodes found on opposite frontiers of the search will be found in similar buckets.

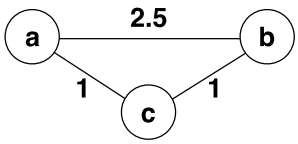

Fig. 7. PEMM cannot use ϵ in its *gmin* stopping condition or in its definitions of pr_F and pr_B.

Our previous implementations also divided states into buckets by h-cost. We have removed this division because we sometimes found buckets with high g-cost (and thus high priority), but low h-cost. These buckets might only have a few dozen states, but required DSD to be performed against the opposite frontier, which might contain billions of states. Removing the h-cost division puts these states into larger buckets to make DSD more efficient, although we sometimes have to expand more nodes to find the solution as a result.

5.1.3. Tie-breaking from low to high g-cost

As in MM, expansions are ordered by priority (low to high). But, in PEMM they are further ordered by search direction (forward then backward), g-cost (low to high), and then by the h-cost and hash function (low to high). The ordering by search direction, hash, and heuristic values does not influence the correctness or efficiency of search, but guarantees consistency between runs and problem instances. Ordering buckets by low to high g-cost may seem counter-intuitive, since it is the opposite of a typical A* ordering. This is important for PEMM, however, as it ensures that once we expand a bucket we will not generate any new states back into that bucket. Without this ordering we would be forced to process buckets multiple times as new states were re-added to the bucket.

For example, assume we process the bucket with g-cost 4 first, followed by the bucket with g-cost 3. Some of the successors of the g-cost 3 bucket will possibly have the same priority as their parents, and thus will get written back into the g-cost 4 bucket which has already been handled. Thus, the g-cost 4 bucket will then have to be processed a second time. This isn't a problem from the perspective of node expansions, but it can be quite inefficient for duplicate and solution detection where we want to limit the number of times we must access disk for these operations.

The consequence of this tie-breaking is that we may have the solution on the open list (with high g-cost) for a long time before it is found. In future work we will consider ways to improve DSD to find solutions earlier.

5.1.4. Delayed duplicate detection (DDD)

Hash-based delayed duplicate detection [35] is used to detect and remove duplicates when a bucket is loaded into RAM. Hash-based DDD works by loading all states into a hash table. Since duplicates will be mapped to the same entries, they are effectively removed in this process.

This is sufficient for removing duplicates within a bucket, but it will not remove duplicates between buckets, such as when a state is re-generated with larger g-cost. This happens, for example, when a state re-generates its parent or one of its siblings. In the general case we must look for duplicates in buckets on *Closed* that could possibly contain a given state with lower g-cost. With unit edge costs this is more efficient, since a state from *Open* with g-cost g_s can only be found in buckets with g-cost $g_s - 1$ or $g_s - 2$. When looking for duplicates on *Closed*, we read the relevant *Closed* files incrementally and check to see if any of their states are found in the hash table (bucket) in RAM. All such states are removed before expansions begin.

Our implementation does not include enhancements for handling arbitrary edge costs, something that is addressed in unidirectional search by PEDAL [25], and does not use frontier search [37], which uses additional domain-specific information to avoid generating duplicates.

5.1.5. Delayed solution detection

The primary idea of delayed solution detection is to check to see if a solution has been found when a state is being expanded as opposed to when the state is generated. We use a hash-based solution detection approach similar to hash-based DDD. When a bucket is loaded into RAM and DDD has been completed, all other buckets in the opposite frontier that could contain the same states as the current bucket are also loaded. These are not stored in RAM; instead we just check whether the states in them are found in the hash table that stores the current bucket. If a duplicate is found, U is updated according to the solution path through that state. For efficiency, only buckets in the opposite frontier that could lead to a better solution than U need to be checked.

5.1.6. Remove ϵ variants from MM

MM uses a termination condition that allows the search to stop early based on the minimum edge cost (ϵ) and minimum g-cost in *Open* in each direction. Unfortunately, this rule does not work in general with DSD. We illustrate this in Fig. 7, where states a and b are the start and goal. All edges are marked with their costs; no heuristic is used.

In this example the search begins with a on $Open_F$ and b on $Open_B$. After each of these states are expanded, c will be on $Open_F$ and $Open_B$. Although c is on both *Open* lists, this will not be detected until c is expanded. Before it is expanded the minimum g-cost in the forward direction is 1 (c), and the minimum g-cost in the backward direction is 1 (also c).

Using the ϵ termination condition we would conclude that we could terminate with an optimal solution cost 3. Thus, the search would terminate with the solution cost 2.5 between a and b, which is incorrect. The ϵ termination condition is justified with ISD because there are no paths through $Open$ that have not been discovered already. Thus, new paths can only be found by adding new edges to existing paths. With DSD there can be undiscovered solutions on $Open$ with cost $gmin_F + gmin_B$ such as through c in this example, hence the ϵ rule cannot be used.

As a result, the termination condition used for PEMM is:

$$U \leq \max(min(prmin_F, prmin_B), fmin_F, fmin_B, gmin_F + gmin_B).$$

MMe introduces a new priority rule $pr_F(n) = \max(f_F(n), 2g_F(n) + \epsilon)$. We show here that the additional ϵ term cannot be used with PEMM. Consider again the graph in Fig. 7, where states a and b are the start and goal. All edges are marked with their costs; no heuristic is used.

In this example the search begins with a and b on $Open$. After each of these states are expanded, c will be on $Open_F$ and $Open_B$, but the solution will not yet be detected. The priority of c will be 3 in each direction. However, the search will have also found the path of cost 2.5 between the start and the goal. So, with DSD and the MMe priority rule, PEMM will terminate with the suboptimal solution cost 2.5.

As a result, the priority of a state in PEMM in the forward direction must be:

$$pr_F(n) = \max(f_F(n), 2g_F(n))$$

An analogous rule is used in the backward direction.

5.1.7. Parallel search

Because of the latency of disk, it makes sense to run as many PEMM operations in parallel as possible. PEMM performs three sets of operations in parallel. (1) The expansions for a given bucket can all be performed in parallel. This is where the greatest parallel efficiency is achieved. (2) Solution detection can be performed in parallel to expansion. If solution detection is not complete when expansion is complete, we wait until solution detection completes before moving to the next bucket. (3) The next bucket can be pre-loaded while the current bucket is being expanded.

5.2. Pseudocode for PEMM

Algorithm 2: Pseudocode for PEMM

```
1  g_F(start) := g_B(goal) := 0;
2  Open_F := {start};
3  Open_B := {goal};
4  U := ∞
5  while (Open_F ≠ ∅) and (Open_B ≠ ∅) do
6      C := min(prmin_F, prmin_B)
7      if U ≤ max(C, fmin_F, fmin_B, gmin_F + gmin_B) then
8          return U
9      if C = prmin_F then
10         // Expand in the forward direction
11         choose bucket b ∈ Open_F with pr_F(b) = prmin_F; break ties by low g_F
12         remove b from Open_F
13         load b into RAM // DDD
14         check for duplicates between b and Closed_F // DDD
15         write states in b to Closed_F
16         check states in b against Open_B for solutions // DSD (in parallel)
17         for each state n in b do
18             // Do in parallel for each n
19             for each child c of n do
20                 g_F(c) := g_F(n) + cost(n, c)
21                 add c to Open_F
22     else
23         // Expand in the backward direction, analogously
24 return ∞
```

Algorithm 2 contains the pseudocode for PEMM. We begin by initializing the data structures (lines 1 through 4). The pseudocode refers to $Open$ as the data structure in RAM containing information about each state and the data on disk (line 11) as buckets. Adding a state to $Open$ implicit adds it both to the data structures in memory and disk.

The important differences between PEMM and MM are that PEMM must explicitly load buckets of states into RAM (line 13). It then performs all duplicate detection (lines 13 and 14) and solution detection (line 16) before expanding the states in a bucket (line 17). Finally, it uses a simpler termination condition (line 8) than MM, omitting the ϵ condition.

We haven't included the re-opening of states that are found with lower g-costs needed for inconsistent heuristics in the pseudocode because we have not tested these conditions to see if they are efficient in practice.

5.3. Correctness of DSD

We analyze the correctness of DSD here assuming the correctness of MM. While Nicholson [47] uses a variant of DSD, MM has only been proven to be correct with ISD. We show that delaying the solution detection cannot lead to termination with a suboptimal solution.

It is clear that for each direction of a bidirectional search, a state will pass monotonically through three phases: *ungenerated* → *open* → *closed*. We assume that solution detection will be performed at some point during the *open* phase or exactly at the point when a state is written to *closed*, but do not distinguish when. In particular, we just need solution detection to be performed before states with higher f are expanded.

Lemma 1. PEMM *with DSD will terminate with an optimal solution.*

Proof. Consider that MM finds an optimal solution through some state s^* when it is generated and solution detection is performed. PEMM will not find the solution through s^*, but instead will put s^* on $Open$. At this point s^* has optimal g-cost in both directions, s^* is on an optimal path, and s^* is found on $Open$ in both directions. We let s^* be any state that has these three properties.

We show that until s^* is removed from $Open$ and a state with higher priority is expanded, PEMM cannot terminate with a suboptimal solution. Since we perform solution detection on s^* before it is placed on *Closed*, PEMM will terminate with the optimal solution when expanding and performing solution detection s^*.

Recall the PEMM termination conditions:

$$U \leq \max(min(prmin_F, prmin_B), fmin_F, fmin_B, gmin_F + gmin_B).$$

Let C^* be the cost of the optimal solution through s^*. As long as s^* is on $Open$ in both directions, the search cannot terminate with a solution cost $> C^*$. We examine the termination conditions one at a time and show that they will not be met.

Since the heuristic is admissible and s^* is on $Open$, $fmin_F, fmin_B \leq C^*$. This handles the second and third termination conditions. Given that s^* has optimal g-cost in each direction, either $g_F(s^*) \leq \frac{1}{2}C^*$ or $g_B(s^*) \leq \frac{1}{2}C^*$, or both. Without loss of generality, assume $g_F(s^*) \leq \frac{1}{2}C^*$. In this case s^*'s bucket in the forward direction must have $prmin_F \leq C^*$. (Because $2g_F(s^*) \leq C^*$ and $f_F(s^*) \leq C^*$.) This handles the first termination condition. Finally, since s^* is on $Open$ in both directions, $gmin_F + gmin_B \leq C^*$.

Thus, until s^* is removed from $Open$, we cannot terminate with a suboptimal solution. Since we will perform solution detection when removing s^* from $Open$, we are guaranteed to find the optimal solution even when performing DSD. □

6. Region based analysis

In this section we analytically compare MM, MM_0, and A*. "MM" here refers to the family of MM algorithms, including MMe.[14] Whenever we illustrate a point with a specific example, we use MMe as the particular member of the MM family in the example. Even the examples involving MM_0 use the MMe version of MM_0, in which a node n's priority is $2g(n) + \epsilon$. We begin by providing a disjoint partitioning of the state space into regions.

6.1. Dividing the state space into disjoint regions

We say state s is "near to *start*" if $d(start, s) \leq \frac{1}{2}C^*$, "far from *start*" if $\frac{1}{2}C^* < d(start, s) \leq C^*$, and "remote" if $d(start, s) > C^*$. "Near", "far" and "remote" for *goal* are defined analogously. Taken together, these categories divide the state space into the 9 disjoint regions shown in Fig. 8. We denote these regions by two letter acronyms. The first letter (N=near, F=far, R=remote) indicates the distance from *start*, the second letter indicates the distance from *goal*. For example, FN is the set of states that are far from *start* and near to *goal*. NN includes only those states at the exact midpoint of optimal solutions.

For the purpose of this paper we will only be interested in whether a node is near or not near to *goal* since MM's backward search is guaranteed to expand no nodes that are far or remote from *goal*. Therefore, we say that a node is "distant" from *goal* if it is either far or remote from *goal*. That is D = F ∪ R. We thus use the following unified regions: RD = RR ∪ RF, FD = FR ∪ FF, and ND = NR ∪ NF. This leaves only 6 regions depicted in Fig. 9. None of the search algorithms in this paper expands a state in RD, so only 5 regions will enter our discussions.

[14] In fact, our analysis applies to any Bi-HS that meets in the middle.

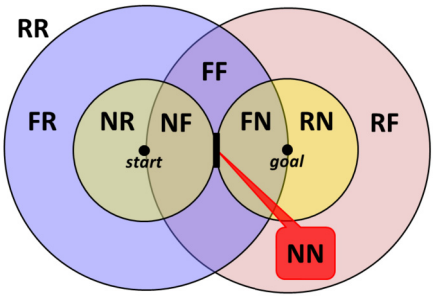

Fig. 8. Diagrammatic depiction of the 9 different regions.

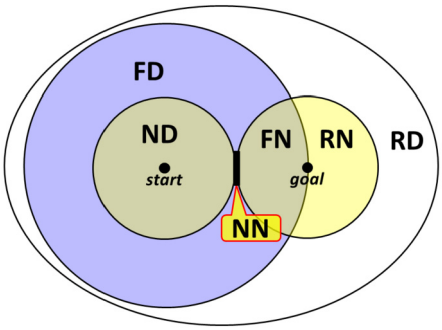

Fig. 9. The 6 regions after unification.

6.2. Preliminaries

In Subsections 6.3–6.6 we compare $\mathtt{MM}_0$, MM, Uni-BS, and A* based mainly on the nodes they expand in each region. In our analysis a region's name denotes both the set of states and the number of states in the region. We will use the names in equations and inequalities. An inequality involving two algorithms, e.g. A* $<$ MM, indicates that one algorithm (A* in this example) expands fewer nodes than the other. Since the region names denote both the number and set of states we will use one symbol, "+", to indicate both adding the number of states in two regions and the set of states defined by their union. For example, the expression NN + FN + RN denotes both the union of those three regions and the number of states in their union.

The novelty of our analysis stems from the fact that we isolate the behavior of the different algorithms to each of the regions and then sum up these behaviors. As we will see, in all cases except Uni-HS vs. Uni-BS no algorithm-type is superior to any other in all regions. We will summarize these analyses with three general rules (GR1, GR2, and GR3). These are general expectations, not iron-clad guarantees. There are many factors in play in a given situation, some favoring one algorithm-type, some favoring another. It is the net sum of these factors that ultimately determines which algorithm-type outperforms another. Our general rules state what we expect will usually be the dominant forces.

Although MM and A* do not require their heuristics to be consistent, the analysis in this section, with the exception of Section 6.5.1, is directly applicable only when the heuristics are consistent, for two reasons. First, our analysis reasons about the number of distinct nodes that are expanded in each region, it does not take into account the number of times the same node is expanded. With a consistent heuristic, a node will be expanded at most once. With an inconsistent heuristic the same node can be expanded many times [20,45]. For example, the number of nodes in FD could be much larger than the number of nodes in RN, but A* could do fewer node expansions in FD than MM does in RN if the heuristic is inconsistent. The second reason is that our analysis assumes that A* will expand every node with $f(n) < C^*$. This is true when A*'s heuristic is consistent but it is not necessarily true when A*'s heuristic is inconsistent.

6.3. $\mathtt{MM}_0$ compared to Uni-BS

We begin by analyzing the brute-force algorithms $\mathtt{MM}_0$ and Uni-BS since this lays the foundation for the subsequent comparisons.

Uni-BS only expands nodes that are near to or far from *start*. We write this as the equation:

$$\text{Uni-BS} = \text{ND} + \text{NN} + \text{F}'\text{N} + \text{F}'\text{D}. \tag{5}$$

F′ here indicates that Uni-BS might not expand all the nodes that are far from *start*. For example, Uni-BS will usually not expand all nodes that are exactly distance C^* from *start*. By contrast, Uni-BS must expand all nodes near to *start*.

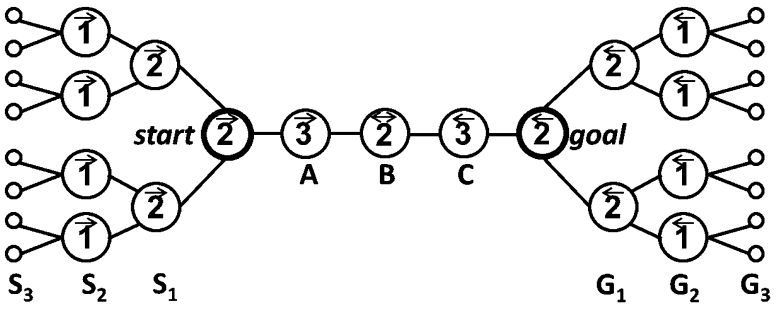

Fig. 10. MM$_0$ need not expand all nodes with $g_F(n) < (C^* - \epsilon)/2$ or $g_B(n) < (C^* - \epsilon)/2$.

MM$_0$ only expands nodes that are near to *start* or to *goal* as shown in the following equation:

$$\text{MM}_0 = \text{N}'\text{D} + \text{N}'\text{N}' + \text{FN}' + \text{RN}'. \tag{6}$$

N′ here indicates that MM$_0$ might not expand all the nodes that are near to *start* or *goal*. For example, if $\epsilon > 0$, MM$_0$ will not expand any node in NN. Moreover, MM$_0$ can terminate before some nodes with $g_F(n) < (C^* - \epsilon)/2$ or $g_B(n) < (C^* - \epsilon)/2$ have been expanded. This is illustrated in Fig. 10. All edge costs in the figure are 1. The numbers in the nodes are discussed in Sections 6.4 and 6.6; they may be ignored for now. S_i (G_i) is the layer of nodes at depth i in the tree rooted at *start* (*goal*) growing away from the optimal solution. After MM$_0$ expands *start* and *goal*, A and S_1 will be in $Open_F$, and C and G_1 will be in $Open_B$, all with $g = 1$ ($pr = 2g + \epsilon = 3$). Assuming ties are broken in favor of the forward direction, MM$_0$ will next expand A and S_1, generating B and S_2 with $g_F = 2$ ($pr_F = 5$). It will then switch directions and expand C and G_1 in some order. As soon as C is expanded a solution costing $U = 4$ is found. Since $gmin_F + gmin_B + \epsilon = 2 + 1 + 1 \geq U$, MM$_0$ can stop. This may happen before some nodes in G_1 are expanded even though they are distance 1 from *goal* and $(C^* - \epsilon)/2 = 1.5$.

The difference between F′N in Equation (5) and FN′ in Equation (6) is the following. FN is the intersection of two sets, the set of states far from *start*, denoted F*, and the set of states near to *goal*, denoted *N. In F′N, F′ is a subset of F*, so F′N is the intersection of *N and a subset of F*. By contrast, FN′, is the intersection of F* and a subset of *N.

Uni-BS expands more nodes than MM$_0$ iff (Eq. (5) > Eq. (6)) as written in the following inequality:

$$\text{ND} + \text{NN} + \text{F}'\text{N} + \text{F}'\text{D} > \text{N}'\text{D} + \text{N}'\text{N}' + \text{FN}' + \text{RN}'. \tag{7}$$

To identify the *core differences* between the algorithms, i.e. regions explored by one algorithm but not the other, we ignore the difference between N and N′ and between F and F′, which simplifies Eq. (7) to:

$$\text{FD} > \text{RN}. \tag{8}$$

We have identified two conditions that guarantee FD > RN:

- When $C^* = D$, the diameter of the space, there are no remote states, by definition, so RN is empty.
- When the number of states far from *start* is larger than the number of states near to *goal*, i.e. if FD + FN > FN + NN + RN, or equivalently,[15] FD > NN + RN. We say a problem (*start*, *goal*) is *bi-friendly* if it has this property.

A special case of bi-friendly problems occurs when the following symmetry occurs: the number of states at any distance d from *start* is the same as the number of states at distance d from *goal*, for all $d \leq C^*$. This occurs often in standard heuristic search testbeds, e.g. the Pancake Puzzle, Rubik's Cube, and the Sliding Tile Puzzle when the blank is in the same location type (e.g. a corner) in both *start* and *goal*. In such cases, a problem is bi-friendly if the number of states near to *start* is less than the number of states far from *start*, i.e. more than half the states at depths $d \leq C^*$ occur after the solution midpoint. This is similar to the condition in BK1 with $h(s) = 0\ \forall s$. In many testbeds this occurs because the number of states distance d from any state continues to grow as d increases until d is well past $\frac{1}{2}D$. For example, Rubik's Cube has $D = 20$ and the number of states at distance d only begins to decrease when $d = 19$ (Table 5.1 in [57]).

Non-core differences (ND, NN, FN) can sometimes cause large performance differences. The example in Fig. 11 exploits the fact that Uni-BS always expands all nodes in NN but MM$_0$ expands none when $\epsilon > 0$. All edges cost 1. *start* and *goal* each have one neighbor (s and g respectively) that are roots of depth d binary trees that share leaves (the middle layer, which is NN). $C^* = 2d + 2$ and all paths from *start* to *goal* are optimal. FD and RN are empty. The values on the figure's left may be ignored for now; they are used in Section 6.4. MM$_0$ expands all the nodes except those in the middle layer, for a total of $2 \cdot 2^d$ nodes expanded. Uni-BS will expand all the nodes except *goal*, for a total of $3 \cdot 2^d - 1$ nodes, 1.5 times as many as MM$_0$. This ratio can be made arbitrarily large by increasing the branching factor of the trees.

[15] Technically, it is not correct to cancel the occurrences of FN on either side of the inequality because FN is being searched in the forward direction by Uni-HS but in the backward direction by MM$_0$. This can result in very different sets of nodes being expanded in FN. But just as we have ignored the difference between N and N′ and between F and F′ in order to identify core differences, here we are ignoring this difference.

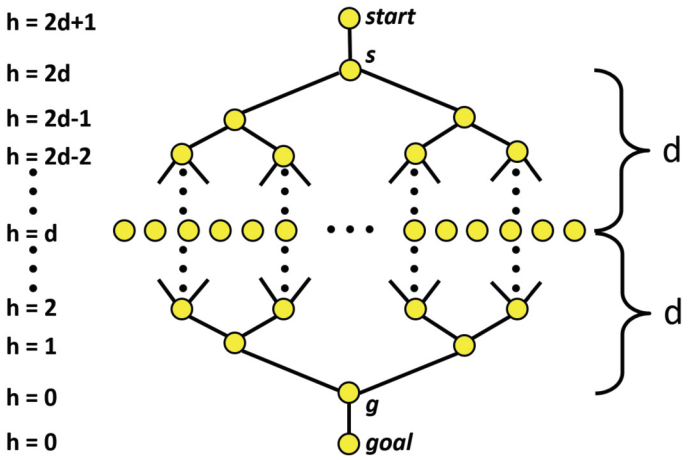

Fig. 11. State space in which NN is large.

The general rule based on the analysis in this section is:

GR1: FD and RN usually determine whether MM$_0$ will expand fewer nodes than Uni-BS or more.

6.4. MM$_0$ compared to A*

A heuristic, h, splits each region into two parts, the states in the region that are pruned by h, and the states that are not pruned. For example, FNU is the unpruned part of FN. The set of states expanded by A* is therefore (modified Eq. (5)):

$$A^* = \text{NDU} + \text{NNU} + \text{FNU} + \text{FDU}. \tag{9}$$

We first compare the first three terms to the corresponding terms in Eq. (5) for MM$_0$ and then compare FDU to RN′.

Region ND: We expect A* to expand many nodes in ND. These nodes have $g_F(n) \leq \frac{1}{2}C^*$ so A* would prune them only if $h_F(n) > \frac{1}{2}C^*$. One might expect MM$_0$'s N′D to be larger than A*'s NDU because A* prunes ND with a heuristic. This underestimates the power of the $gmin_F + gmin_B + \epsilon$ termination condition, which can cause N′D to be much smaller than NDU. In Fig. 10, a number inside node n with a right-pointing arrow over it is $h_F(n)$. Not shown are $h_F(C) = 1$ and $h_F(s) = 1$ $\forall s \in S_3$. Region ND contains *start*, A, S_1 and S_2. The heuristic does no pruning in this region so these are all expanded by A*. MM$_0$ will not expand any node n with $g_F(n) = \frac{1}{2}C^*$ (e.g. S_2) so N′D is half the size of NDU. As a second example, on Rubik's Cube instances with $C^* = 20$, MM$_0$ only expands nodes with $g_F(n) \leq 9$ because of this termination condition. The heuristic used by Korf [34] to solve Rubik's Cube has a maximum value of 11, so A* with this heuristic will not prune any nodes in N′D. In general, we do not expect A* to have a large advantage over MM$_0$ in ND unless its heuristic is very accurate.[16]

Region NN: As discussed above, MM$_0$ usually expands no nodes in NN. Nodes in NN have $g_F(n) = g_B(n) = \frac{1}{2}C^*$, so A*'s $f(n)$ cannot exceed C^* on them. Therefore, even with an extremely accurate heuristic, A* may do little pruning in NN. For example, the heuristic values shown on the left side of Fig. 11 are consistent and "almost perfect" [28] yet they produce no pruning at all. A* behaves exactly the same on this example as Uni-BS and expands 1.5 times as many nodes as MM$_0$.

Region FN: We expect A* to expand far fewer nodes than MM$_0$ in FN. These nodes have $g_F(n) > \frac{1}{2}C^*$ and are relatively close to *goal*. It is common for heuristics to be very accurate near *goal* so we expect the heuristic values for these nodes to be sufficiently large that many nodes in FN are pruned.

FDU vs RN′: RN′ certainly can be much smaller than FDU. In Fig. 10, RN ($G_1 + G_2$) is about the same size as FD (S_3), which is the same as FDU in this example. However, because MM$_0$ will not expand any nodes with $g_B(n) > \frac{1}{2}(C^* - \epsilon)$ (= 1.5 in this example), RN′ is half the size of RN (RN′ contains G_1 but not G_2), so MM$_0$ expands many fewer nodes in RN than A* does in FD. On the other hand, with a sufficiently accurate heuristic, FDU will certainly be the same size as or smaller than RN′. In the extreme case, when RN′ is empty, this requires a heuristic that prunes every node in FD. This is not impossible, since no optimal path passes through FD, but it does require an extremely accurate heuristic. Moreover, FD even without any pruning can be much smaller than RN′. Deleting S_3 from Fig. 10 makes FD empty, while RN′ can be made arbitrarily large.

The general rule based on the analysis in this section is:

GR2: When FD > RN, A* will expand more nodes than MM$_0$ unless A*'s heuristic is very accurate.

[16] We recognize the imprecision in terms like "very accurate", "inaccurate", etc. We use these qualitative gradations to highlight that as the heuristic's accuracy increases or decreases, the advantage shifts from one algorithm to another. This is discussed in Section 6.7.

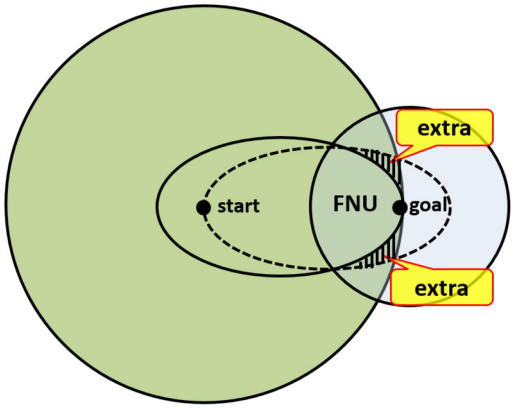

Fig. 12. Depiction of the situation when FNB = FNU + "extra".

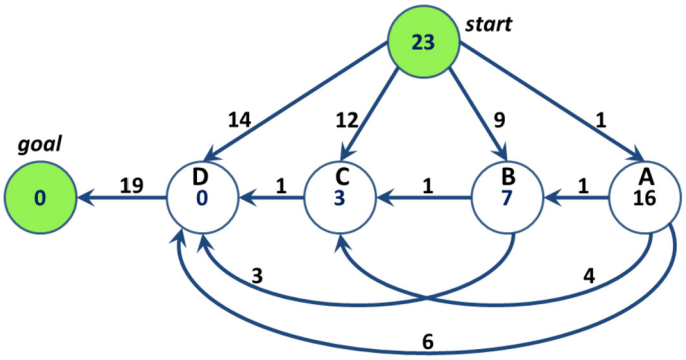

Fig. 13. State space in which A* re-opens nodes an exponential number of times but MM expands each node once.

6.5. *MM compared to A**

Modifying Eq. (6), the equation for MM is:

$$\text{MM} = \text{N}'\text{DU} + \text{N}'\text{N}'\text{U} + \text{FN}'\text{B} + \text{RN}'\text{B}. \tag{10}$$

B has the same meaning as U, but is based on h_B, the heuristic of MM's backwards search. For example, FNB is the part of FN that is not pruned by h_B. In general, FNB will be different than FNU, the part of FN that is not pruned by h_F, the heuristic used by A*.

Regions ND and NN: By definition, $\text{N}'\text{DU} \leq \text{NDU}$ and $\text{N}'\text{N}'\text{U} \leq \text{NN}$, so MM has an advantage over A* in ND and NN.

Region FN: Because A* and MM are searching in different directions in FN, FNU is almost certainly smaller than FN′B. In A*'s forward search nodes in FN have $g_F(n) > \frac{1}{2}C^*$ and h_F is estimating a small distance (at most $\frac{1}{2}C^*$). By contrast, in MM's backwards search, nodes in FN have $g_B(n) \leq \frac{1}{2}C^*$ and h_B would need to accurately estimate a distance larger than $\frac{1}{2}C^*$ to prune them. So, A* has an advantage over MM's backward search in FN. This is illustrated by the two ovals in Fig. 12. The left oval (solid) represents A* in the forward direction. The right oval (dotted) represents MM in the backward direction. All the nodes in FNU are also inside the dotted oval, i.e. are expanded by the backward search. In addition, FNB includes the striped areas marked "extra" in the figure. These are nodes expanded by MM's backward search but not by A*'s forward search.

FDU vs RNB: Not much pruning will usually occur during MM's backward search in RN because RN's g_B-values are small and the distances being estimated by h_B are large. The comparison of FDU and RNB therefore has the same general conclusion as the comparison in the previous subsection of FDU and RN′ namely, that FDU will certainly be the same size as or smaller than RNB with a sufficiently accurate heuristic.

The general rule based on this section's analysis is the same as GR2 with MM_0 replaced by MM.

6.5.1. *MM compared to A* with inconsistent heuristics*

Martelli [45] showed that A* could re-expand nodes an exponential number of times if its heuristic is inconsistent. This will also happen with MM when $pr(n) = f(n)$ for all nodes, since in that case MM's forward search will be identical to A*'s. However, if the heuristic is sufficiently weak that $pr(n) = 2g(n)[+\epsilon]$ for many nodes, MM can do exponentially fewer node re-expansions than A*. This is illustrated in Fig. 13, which is a slightly modified version of Martelli's 6-node graph. The

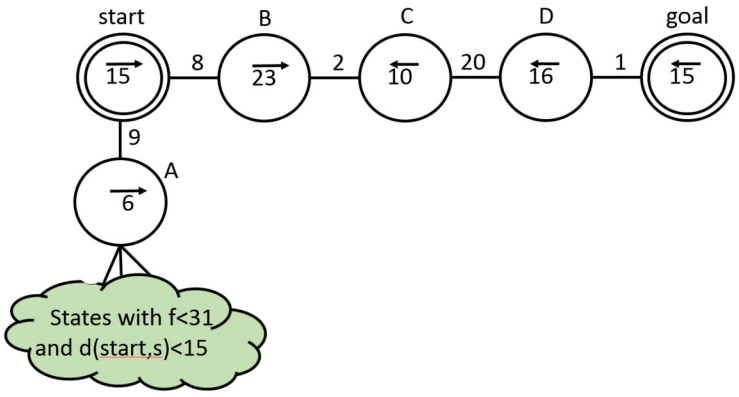

Fig. 14. Example where MMe expands an arbitrarily large number of nodes more than MM_0 even though there are no ties to break.

optimal path is $start - A - B - C - D - goal$, with $C^* = 23$. The numbers inside the nodes are their h_F values. They are admissible but not consistent. $h_B(goal)$ (not shown) is 23, which is larger than any $prmin_F$-value that occurs during search after *start* is expanded, so on this graph MM only searches in the forward direction, the same as A*. Just as in Martelli's original 6-node graph, A* re-expands nodes an exponential number of times on this graph. Nodes D, C and B are expanded a total of 8, 4 and 2 times, respectively, by A*. By contrast, MM expands each node only once.

We believe, but have not fully proven, that the opposite situation cannot occur, i.e. that if MM's forward search re-expands a node then A* must also re-expand it. It is easy to show that if node X is expanded by MM's forward search when $g_F(X)$ is suboptimal it is because of the heuristic function, i.e. $pr_F(X) = f_F(X)$ and, at the time X is expanded with a suboptimal $g_F(X)$, there is a node N on the optimal path to X such that $pr_F(N) = f_F(N) > f_F(X)$. This shows that if A* reaches X by a suboptimal path of the same cost, A* will expand X with this suboptimal g_F-value and later have to re-expand it. What we have not yet proven is that if MM's forward search reaches X by a suboptimal path then A* will also reach X by a suboptimal path of the same cost. Of course, node re-expansions by MM's backward search are not connected in any direct way to A*'s forward search.

6.6. MM_0 compared to MM: an anomaly

If h_1 and h_2 are admissible heuristics and $h_1(s) > h_2(s)$ for all non-goal nodes, then every node expanded by A* using h_1 will also be expanded by A* using h_2 (RESULT 6, p. 81 [49]). In particular, A* with a consistent non-zero heuristic cannot expand more nodes than Uni-BS.

This is not necessarily true for MM or most Bi-HS algorithms. In Fig. 10 the value in a node is its h-value in the direction indicated by the arrow. All nodes in layer S_3 (G_3) have $h_F(s) = 1$ ($h_B(s) = 1$). The heuristic values in each direction are consistent. MMe expands all the nodes in S_1 and G_1 because they have $pr(s) = 2g(s) + \epsilon = 3$ while $pr_F(A) = pr_B(C) = 4$. By contrast, we saw (Section 6.3) that MM_0 could stop before expanding all the nodes in S_1 and G_1. Thus we see that MM_0 can expand strictly fewer nodes than MM with a consistent, non-zero heuristic. Bi-HS algorithms that strictly alternate search direction or use the cardinality criterion to choose the direction and then expand the node in the chosen direction with the smallest f-value will expand even more nodes—they will expand all the nodes in S_2 and G_2 ($f(n) = 3$) before expanding A and C ($f_F(A) = f_B(C) = 4$).

This example mimics behavior we report with the GAP-2 and GAP-3 heuristics in the Pancake puzzle experiments below. We believe it occurs commonly with heuristics that are very accurate near the goal but inaccurate elsewhere.

The largest excess of MMe over MM_0 occurs when $f_F(n) = f_B(n) = C^*$ for all nodes n on all optimal paths, and $f_F(n)$ and $f_B(n)$ are both strictly less than C^* for all other nodes. The situation for unit edge costs is different than that for non-unit edge costs, as we now discuss.

- In state spaces where all edge costs are 1 there are two cases. If C^* is odd, the difference between the two algorithms is entirely due to how they break ties among nodes with $g_F(n) = (C^* - 1)/2$ and $g_B(n) = (C^* - 1)/2$. If C^* is even, there can be nodes with $g_F(n) = C^*/2 - 1$ or $g_B(n) = C^*/2 - 1$ that MMe must expand but that MM_0 will only expand in its worst case (G_1, S_1, A and C in Fig. 10).
- For state spaces with non-unit costs there is even greater scope for MMe to expand more nodes than MM_0, even when there are no ties to break. This is illustrated in Fig. 14. MM_0 will expand *start* (forward), *goal* (backward), D (backward) and B (forward) and then stop because it has found the path through C costing $U = 31$ and $gmin_F + gmin_B + \epsilon = 9 + 21 + 1 = 31$. By contrast, once MMe has expanded *start* (forward), *goal* (backward), and D (backward), node A will have the lowest priority ($pr_F(A) = 2g_F(A) + \epsilon = 19$, compared to $pr_F(B) = 31$ and $pr_B(C) = 43$), so MMe will expand A and the whole cloud of nodes below it before expanding B (forward) and stopping.

The phenomenon we have just been discussing causes MM to expand more nodes than MM_0, but MM's heuristic can result in nodes being pruned that MM_0 would expand. If MM's heuristic is sufficiently accurate, the number of nodes its heuristic prunes will exceed the number of excess nodes MM expands because of the anomalous behavior described above, and MM will expand fewer nodes than MM_0. This is the general conclusion we draw from this section's analysis.

GR3: Bi-HS with an inaccurate heuristic will expand more nodes than Bi-BS.

6.7. Summary

Many factors come into play in determining which algorithm—A*, MM_0, or MM—will expand the fewest nodes. Our general rules express what we think will be the most common outcomes. GR1 says that MM_0 will expand more nodes than Uni-BS if region FD is smaller than region RN. Adding a heuristic to both brute-force searches in this situation is not expected to reverse this conclusion, so we expect A* to expand fewer nodes than MM_0 and MM (and of course, Uni-BS) when FD is smaller than RN.

When FD is larger than RN, the algorithm that expands the fewest nodes will depend on the accuracy of the heuristics. With sufficiently inaccurate heuristics, MM_0 is expected to expand fewer nodes than A* (GR2) and MM (GR3). As the heuristics' accuracy increases, the advantage of MM_0 over MM will diminish and eventually MM will expand fewer nodes than MM_0. Because MMe is generally stronger than basic MM, MMe is expected to expand fewer nodes than MM_0 with a less accurate heuristic than basic MM. Of course, A* will also be benefiting from the improved heuristic, but, contrary to Barker and Korf [4], we expect MM to expand fewer nodes than both MM_0 and A* when the heuristics are moderately accurate. As the accuracy continues to increase, A* will eventually expand fewer nodes than either of the bidirectional searches (because FDU will become very small (GR2)).

7. Experiments

The purpose of the experiments in this section is to verify the correctness of our general rules (GR1–GR3) and the conjectures in Section 6.7 about which algorithm—A*, MM_0, MM,[17] or MMe—will expand the fewest nodes. Since some of the rules refer to the sizes of certain regions, they could only be tested in domains small enough to be fully enumerated. Likewise, since some of the rules and conjectures refer to a heuristic's relative accuracy, we used at least two heuristics of different accuracy in each domain. All heuristics used in these experiments were consistent, not just admissible. The three domains used in our study are the 10-Pancake Puzzle, grid pathfinding, and Rubik's Cube. In these domains all problems are bi-friendly. Because GR1–GR3 make predictions about the number of nodes expanded, that is the quantity we focus on in our experiments.

7.1. 10-Pancake puzzle

In the 10-pancake puzzle a state is a vector with 10 numbers and the task is to sort them. Operators only allow the reversal of prefixes of the vector. We ran MM_0, MM, MMe, Uni-BS, and A* on 30 random instances for each possible value of C^* ($1 \leq C^* \leq 11$). We used the GAP heuristic [27][18] and derived less accurate heuristics from it, referred to as GAP-X, by not counting the gaps involving any of the X smallest pancakes. For example, GAP-2 does not count the gaps involving pancakes 0 or 1.

7.1.1. $C^* = 10$

Table 1 shows the number of nodes expanded in each region for each algorithm using each heuristic for $C^* = 10$. The first row, "$|Region|$" shows the number of states in each region. Column "Total" is the total of the five rightmost columns. The total for $|Region|$ is not the size of the entire state space because it does not include region RD (it is not in the table because none of the algorithms expand nodes in RD).

We see that RN is small (929) and FD is very large (3.5M). As a consequence, MM_0 expands many fewer nodes than Uni-BS (GR1), as shown in the first two rows. ND is identical in size to FN+RN (= DN) because of the symmetry in this space. The asymmetry of MM_0's expansions in ND and FN+RN is because, for $C^* = 10$, MM_0 must expand all the nodes with $g(s) = 4$ in one direction but not the other. MM's expansions in these regions are much more balanced.

For the more accurate heuristics (GAP and GAP-1), FDU is very small, and A*'s total is largely determined by ND. By contrast for the less accurate heuristics (GAP-2, GAP-3, and GAP-4) A*'s total is largely determined by FDU. Even with these less accurate heuristics FDU is less than 10% of FD but FDU is large enough to dominate the other regions.

The **bold** numbers show the best algorithm for a given heuristic. Depending on the heuristic, the algorithm expanding the fewest nodes is A* (GAP), MM/MMe (GAP-1 and GAP-2), or MM_0 (GAP-3 and GAP-4). MM and MMe were identical except for GAP and GAP-4 where MMe had a slight advantage. We explain this behavior below (Section 7.1.3).

[17] In this section, "MM" refers to a specific implementation of basic MM, not to the general MM family.

[18] A *gap* occurs when two adjacent tokens in the current state are not consecutive numbers. The GAP heuristic counts the number of gaps in the entire state.

Table 1
10-Pancake results: average nodes expansions by region for instances with $C^* = 10$.

	Total	$f < C^*$	ND	NN	FD	FN	RN
$\|Region\|$	3,556,497	NA	27,432	13	3,501,619	26,504	929
			Brute-force searches				
Uni-BS	2,078,788	NA	27,432	13	2,040,232	11,112	0
MM_0	6,070	NA	4,620	0	0	1,390	60
			GAP-4				
A*	270,337	255,745	25,906	13	240,118	4,300	0
MM	8,944	8,944	4,324	0	0	4,447	173
MMe	8,812	8,812	4,192	0	0	4,447	173
BS*	119,032	119,032	16,071	13	83,779	18,741	420
MM-2g	144,138	144,138	19,403	13	104,315	19,942	457
			GAP-3				
A*	68,344	66,693	17,981	13	48,194	2,156	0
MM	8,415	8,415	4,223	0	0	4,050	141
MMe	8,415	8,415	4,223	0	0	4,050	141
BS*	77,095	77,095	10,563	13	51,903	14,469	147
MM-2g	95,900	95,900	16,589	13	63,145	15,924	227
			GAP-2				
A*	12,124	11,741	6,153	12	5,101	857	0
MM	**5,037**	5,037	2,517	0	0	2,498	22
MMe	**5,037**	5,037	2,517	0	0	2,498	22
BS*	16,949	16,949	4,696	12	7,990	4,244	8
MM-2g	21,641	21,641	6,208	12	9,060	6,342	19
			GAP-1				
A*	909	864	627	11	128	143	0
MM	**771**	771	399	0	0	372	0
MMe	**771**	770	399	0	0	372	0
BS*	1,144	1,144	488	12	159	485	0
MM-2g	1,423	1,423	647	12	185	579	0
			GAP				
A*	**38**	17	22	2	3	11	0
MM	91	28	48	4	0	39	0
MMe	60	24	34	0	0	26	0
BS*	47	30	20	3	3	21	0
MM-2g	46	30	21	3	3	19	0

We also experimented with two Bi-HS algorithms that are not guaranteed to meet in the middle. The first was BS* [40] which uses Pohl's cardinality criterion for selecting the next node to expand. In addition, to examine the effect of the $2g$ term in MM's definition of a node's priority, we ran an altered version of MM, called MM-2g, which is identical to MM except it omits the $2g$ term in the definition of $pr(n)$, so node n's priority is the usual $f(n)$. We also added code to prevent MM-2g from expanding the same node in both directions. MM-2g is reminiscent of the algorithm by Piljs & Post [52] in that both algorithms choose the node with the minimal f-value in either direction. BS* usually outperforms MM-2g, showing that the cardinality criterion is a better approach. Unlike MM, these algorithms expand many nodes in FD and many more nodes than MM in ND, NN, and FN. Therefore, these algorithms are much worse than MM, highlighting the importance of meeting in the middle. The GAP heuristic is an exception where BS* outperforms MM. The reason is that, similar to A*, when the heuristic is very accurate, using only f-values is beneficial.

GR1, GR2, and GR3 are all confirmed by this experiment.

- **GR1:** For every instance for every value of C^*, FD $>$ RN and MM_0 expanded fewer nodes than Uni-BS.
- **GR2:** A* expands more and more nodes in FD as the heuristic becomes less accurate, while MM and MM_0 always expand a small fraction of the nodes in RN. With GAP-4 through GAP-1 A* expanded more nodes in FD than MM and MM_0 expanded in RN. As expected by GR2, A* is inferior with these heuristics. With GAP, A* expanded fewer nodes in FD than MM_0 expanded in RN. Thus, A* was the best with GAP. We note that A* expanded slightly more nodes in FD than MM in RN (3 compared to 0). Still, A* expanded fewer nodes than both MM_0 and MM with GAP because of ND and FN (non-core regions).
- **GR3:** With the best heuristic, GAP, MM expands many fewer nodes than MM_0. As the heuristic becomes less accurate, the difference between MM and MM_0 steadily diminishes and eventually (GAP-2) turns into a steadily growing advantage for MM_0.

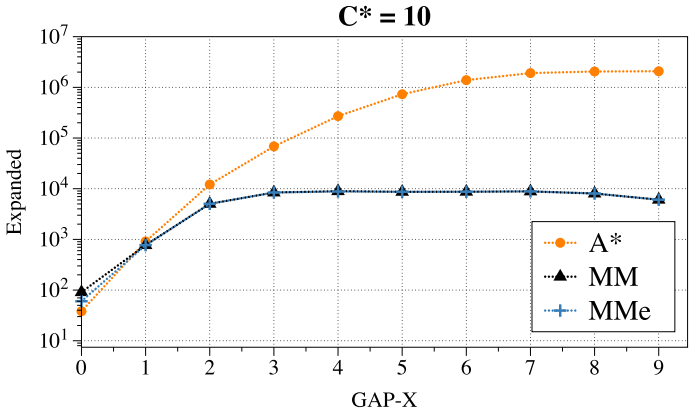

Fig. 15. Nodes expanded vs. heuristic accuracy for $C^* = 10$ ($X = 0$ is the most accurate).

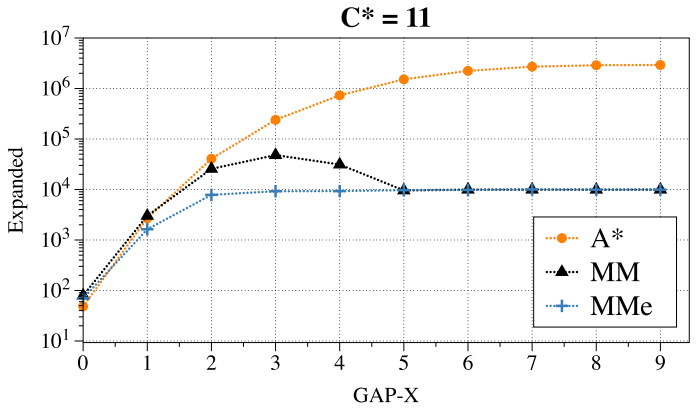

Fig. 16. Nodes expanded vs. heuristic accuracy for $C^* = 11$ ($X = 0$ is the most accurate).

In summary, as expected, with the very accurate heuristic (GAP), A* was the best. With moderately accurate heuristics (GAP-1 and GAP-2) MM was the best. Finally, with the weaker heuristics (GAP-3 and GAP-4) MM_0 was the best. We have performed similar experiments with $C^* = 11$. Similar trends were observed with some exceptions, which we explain below in Section 7.1.3.

7.1.2. Does MM outperform A because of tie-breaking?*

If MM's savings over A* all occurs on nodes with $f(n) = C^*$ then it could be argued that MM just has a better tie-breaking rule than A* and the gain does not really show that bidirectional search is inherently superior to unidirectional search (on our instances). On the other hand, we know that A*, with a consistent heuristic, must expand all nodes with $f(n) < C^*$, but MM does not. So, if we see MM expanding fewer nodes with $f(n) < C^*$ than A*, then we have shown that MM is fundamentally superior to unidirectional search on our test instances.

The $f < C^*$ column in Table 1 presents the number of nodes with $f < C^*$ expanded by the different algorithms. Clearly, MM expands fewer such nodes than A*, showing that its advantage is not due to better tie breaking.

7.1.3. Convergence to MM_0

As the heuristics get weaker the relative advantage of MM_0 over MM increases due to the phenomenon underlying GR3. However, at some stage, this trend must stop as MM with the least accurate heuristic possible ($h(s) = 0\ \forall s$) is equivalent to MM_0. In order to validate this we varied the heuristics from GAP through GAP-9 (always returns 0) for A*, MM, MMe. Figs. 15 and 16 present the number of nodes expanded (y-axis), averaged over 30 random instances, as a function of the heuristic (x-axis; $x = 0$ is GAP). The rightmost point for MM and MMe in each plot ($x = 9$ is GAP-9) is MM_0. For $C^* = 10$ (Fig. 15), the curves for MM and MMe are indistinguishable and are slightly inferior to MM_0 for heuristics GAP-3 through GAP-8. For $C^* = 11$ (Fig. 16), MMe never expands more nodes than MM_0, but the advantage of MM_0 over MM increases as we move from GAP through to GAP-3, decreases at GAP-4, and at GAP-5 the two algorithms expand the same number of nodes.

We note that, for $C^* = 10$ MM and MMe were close or identical in the number of nodes expanded. By contrast, for $C^* = 11$ MMe outperformed MM for the range of GAP-1 until GAP-4. The difference between $C^* = 11$ and $C^* = 10$ is explained as

Table 2
Results on the Dragon Age: Origins *brc203* map: average node expansions by region with different heuristic weights.

	Total	$f < C^*$	ND	NN	FD	FN	RN
$\|Region\|$	20,713.0	NA	6,427.5	0.8	4,261.5	3,537.0	1,149.9
			Brute-force searches				
Uni-BS	14,213.0	14,197.8	6,427.5	0.8	4,254.8	3,529.8	0.0
MM$_0$	**11,025.2**	11,025.2	6,381.1	0.0	0.0	3,505.9	1,138.3
			Octile. $W = 0.1$				
A*	13,726.3	13,725.3	6,427.5	0.8	3,863.7	3,434.2	0.0
MMe	11,025.3	11,025.3	6,381.1	0.0	0.0	3,505.9	1138.3
			Octile. $W = 0.4$				
A*	12,118.5	12,117.5	6,396.4	0.8	2,631.1	3,090.2	0.0
MMe	**10,999.1**	10,999.1	6,355.8	0.0	0.0	3,512.0	1,131.3
			Octile. $W = 0.6$				
A*	10,526.7	10,525.7	5,921.4	0.8	1,835.2	2,769.3	0.0
MMe	**10,353.3**	10,353.3	5,892.3	0.0	0.0	3,512.3	948.8
			Octile. $W = 0.8$				
A*	**8,543.6**	8,542.6	5,140.3	0.8	1,071.6	2,330.8	0.0
MMe	9,086.3	9,086.3	5,124.0	0.0	0.0	3,330.2	632.1
			Octile. $W = 1$				
A*	**5,781.1**	5,678.6	3,943.1	0.6	644.3	1,193.1	0.0
MMe	6,752.0	6,713.7	3,941.8	0.0	0.0	2,504.5	305.7

follows. As in Section 4.3, to clearly distinguish between basic MM's priority function and MMe's, in this explanation we will use pr_F and pr_B for basic MM's priority function and use the special notation pr_F^ϵ and pr_B^ϵ for MMe's priority function. When $C^* = 11$ all nodes u at depth 5 have $pr_F(u) = 10 < C^*$ but they have $pr_F^\epsilon(u) = 11 = C^*$. So, MM is likely to expand many more such nodes than MMe. Thus, for $C^* = 11$ MMe has a large advantage over MM due to such nodes. For $C^* = 10$ nodes u at depth 4 will have $pr_F(u) = 8 < C^*$ and $pr_F^\epsilon(u) = 9 < C^*$. Thus, MMe will have no advantage over MM for such nodes since for both systems these nodes have a priority smaller than C^*. Nodes u at depth 5 will have $pr_F(u) = 10 = C^*$ and $pr_F^\epsilon(u) = 11 > C^*$. MMe will never expand such nodes. However, only very few (or none) such nodes are expanded by MM, so the advantage of MMe over MM is very small (e.g. for GAP and GAP-4) or does not exist (e.g. for GAP-1, GAP-2 and GAP-3).

7.2. Grid maps

In this section we investigate grid maps because they have significantly different properties than the other domains that we study. First, they have non-unit edge costs. Second, the maps are irregular. Third, the size of the various regions in the maps can differ drastically between instances. To avoid averaging away too many differences we select a single map (*brc203*) from the game Dragon Age: Origins in the moving AI benchmark repository [60]. Agents can move in any of 8 directions on these maps, and the default heuristic is the octile heuristic, which is the perfect free-space heuristic for 8-way movement. This heuristic is relatively strong. Therefore, similar to the GAP heuristic for the Pancake puzzle, we weakened it artificially in order have a spectrum of heuristic strengths. For this we multiplied the octile heuristic by the following weights: $W = \{0.1, 0.4, 0.6, 0.8, 1\}$.

In Table 2 we report the primary results of this experiment on a variety of weights. Results are averages over 1320 problem instances that vary in length from 0 to 527. We focus on the A* and MMe algorithms, and report the average node expansions and region sizes across all problems on the map. We analyze GR1–GR3 in grid worlds and make the following observations:

- **GR1:** On average, in this map FD > RN and, as predicted, MM$_0$ expanded fewer nodes than Uni-BS.
- **GR2:** Because FD > RN we expect MMe to outperform A* unless the heuristic is accurate. As predicted, A* outperformed MMe with stronger heuristics, as it is able to perform significant pruning in FN. For weaker heuristics MMe was better than A*.
- **GR3:** With a heuristic weight of 0.1, MMe expands 0.1 more nodes than MM$_0$ on average. This difference is minimal, suggesting that there are not a significant number of nodes that meet the conditions for GR3 in this state space.

Thus, two of the three general rules are strongly confirmed, and the third is weakly confirmed. As we saw with the Pancake puzzle, tie breaking with f-cost < C* does not play a significant role in node expansions, especially with weaker heuristics.

Table 3
Summary node expansion results on the 10 Korf instances [34] and the superflip (S) position. M = millions; B = billions; T = trillions.

		PDB						
		0	1997		888		8210	
#	Depth	PEMM$_0$	PEMM	IDA*	PEMM	IDA*	PEMM	IDA*
1	16	1.08B	428M	**244M**	95M	**19M**	18M	**4M**
2	17	1.69B	**1.00B**	1.51B	249M	**116M**	165M	**30M**
3	17	2.86B	**1.54B**	8.13B	**608M**	675M	202M	**127M**
4	17	2.33B	**949M**	6.56B	570M	**467M**	**18M**	86M
5	18	4.12B	**3.91B**	29.69B	3.26B	**2.40B**	1.20B	**435M**
6	18	7.59B	**6.74B**	15.37B	4.23B	**1.04B**	1.32B	**208M**
7	18	16.32B	**13.99B**	41.57B	6.80B	**3.13B**	1.63B	**659M**
8	18	7.56B	**6.72B**	45.88B	4.56B	**3.75B**	1.40B	**659M**
9	18	6.25B	**5.67B**	58.35B	**4.01B**	5.00B	1.52B	**1.17B**
10	18	5.29B	**4.84B**	70.31B	**3.48B**	4.78B	1.15B	**1.01B**
S	20	38.52B	**38.08B**	3.03T	**38.08B**	116.39B	35.61B	**24.59B**

7.3. Rubik's cube

Rubik's cube is a well-studied domain with 4.3×10^{19} states; the maximum distance between any two states is twenty moves [57]. Optimal solutions to random Rubik's cube instances were first found in 1997 [34] using pattern databases (PDBs) [7]. The cube is made up of 8 corner cubes which have 3 possible orientations each and 12 edge cubes which have 2 possible orientations each.

We study PEMM on Rubik's cube with three different heuristics and with no heuristic (PEMM$_0$). The first heuristic is h_{1997}, the heuristic used for the first optimal solutions. This heuristic uses an 8-corner PDB and two 6-edge PDBs. We use 4-bits per state for the PDBs. The 8-corner PDB has $8!3^7 = 88{,}179{,}840$ entries[19] and requires 42 MB of RAM. The 6-edge PDBs have $\frac{12!}{6!}2^6 = 42{,}577{,}920$ entries and require 20 MB of RAM. Problem symmetry is used for reverse heuristic lookups, so the same PDB can be used in both directions.

The second heuristic is h_{888}. This heuristic uses the 8-corner PDB and two 8-edge PDBs. The 8-edge PDBs have $\frac{12!}{4!}2^8 = 5{,}109{,}350{,}400$ entries and require 2.4 GB of space each. The final heuristic is h_{8210}. This uses the 8-corner PDB, a 2-edge PDB and a 10-edge PDB. The 10-edge PDB has $\frac{12!}{2!}2^{10} = 245{,}248{,}819{,}200$ entries and requires 114.2 GB of RAM.

All experiments are run on a server with dual 2.4 GHz Intel Xeon E5 processors with 128 GB of RAM. Each processor has 8 cores, so a total of 16 compute cores are available, and 32 cores are available with hyperthreading. The machine has two 8TB SAS hard disk drives (HDD) combined in a RAID and two 1.6 TB SATA solid state drives (SSD). The experiments that we perform here use the HDDs for external memory and the SSDs for storing PDBs, but alternate configurations could provide higher performance.

In addition to running PEMM, we compare against IDA* using the standard operator pruning rules for Rubik's Cube [34]. To compare the implementations more fairly, we parallelized IDA* using the approach of AIDA* [56]. This parallel implementation is very efficient, achieving almost perfect speedups.

In our experimental results we solve the 10 instances from Korf [34] as well as the superflip (S) position [57], one of the positions that is maximally distant (20 moves) from the goal. This is the first time, to our knowledge, that a 20-move Rubik's cube position has been solved with general-purpose search algorithm.

For IDA* we report the total running time and total number of nodes expanded. Note that experiments with IDA* typically report node generations. We report expansions here to be comparable to the expansions performed by PEMM. In our previous results [29], we reported the number of nodes that would be expanded by a unidirectional search without parallelization. Here, we report the total number of nodes actually expanded by our parallel implementation. This might be more or less than a unidirectional implementation, depending on which thread finds the solution and how many solutions there are.

The efficiency of PEMM does depend, to some extent, on the number of buckets used during search. In all experiments here PEMM uses 9 bits of the hash in addition to the other bucket parameters, although using fewer bits is faster on easier problems. Using 9 bits is necessary for PEMM$_0$ when solving the superflip position; in preliminary results increasing from 7 to 9 bits reduced the running time from 195,380 seconds to 102,893 seconds. A custom hash table and other enhancements further reduced this time to 59,743 seconds.

Summary results comparing IDA* and PEMM are found in Tables 3 and 4; more detailed results on the performance of PEMM follow.

We first look at node expansions in Tables 3. First, note that Rubik's cube is bi-friendly, so FD > RN. We would then, according to GR2, expect that A* does more work than PEMM$_0$, except with a very accurate heuristic. Indeed, with the less-accurate 1997 heuristic, PEMM dominates IDA*, except on the first (easiest) problem. PEMM$_0$ also does fewer node

[19] Parity between the corners means that the last corner orientation is fixed given the rotation of the remaining corners.

Table 4
Summary timing results. All times in seconds.

		PDB						
		0	1997		888		8210	
#	Depth	PEMM$_0$	PEMM	IDA*	PEMM	IDA*	PEMM	IDA*
1	16	898	1,025	**135**	418	**15**	1,046	**6**
2	17	1,652	2,438	**823**	864	**74**	1,234	**23**
3	17	4,463	**3,637**	4,421	1,406	**416**	2,038	**79**
4	17	3,243	**2,276**	3,570	1,340	**288**	708	**53**
5	18	5,994	**9,541**	16,281	8,010	**1,472**	3,649	**259**
6	18	10,186	13,895	**8,413**	9,506	**637**	4,221	**123**
7	18	14,235	**22,691**	22,843	13,514	**1,918**	4,662	**382**
8	18	7,579	**13,758**	25,156	10,116	**2,294**	4,215	**385**
9	18	6,587	**12,065**	32,066	9,227	**3,058**	4,791	**681**
10	18	5,850	**10,880**	38,642	8,161	**2,924**	3,513	**586**
S	20	59,743	**110,842**	1,794,329	90,148	**71,848**	83,591	**14,129**

Table 5
Disk usage required to solve each problem.

#	0	1997	888	8210
1	143G	57G	13G	2.4G
2	224G	132G	33G	22G
3	378G	204G	81G	27G
4	309G	126G	76G	2.5G
5	549G	520G	433G	159G
6	1002G	893G	560G	175G
7	2.2T	1.9T	900G	215G
8	1003G	892G	604G	186G
9	830G	752G	532G	201G
10	699G	643G	461G	153G
S	5.0T	5.0T	5.0T	4.7T

expansions than IDA*, except on the first two problems. PEMM$_0$ does worse than IDA* with the more-accurate 888 heuristic on all problems except the superflip problem; PEMM is only better on the hardest problems.

When we use the 114 GB 8210 PDB, which is extremely accurate, IDA* has better performance than PEMM across all problems except one. But, digging into the results, there is significant room for PEMM to improve. On the superflip position less than 1% of the states expanded by IDA* were expanded while searching at depth 20. PEMM$_0$ also expands about 1% of the total states at depth 20, finding the solution in the first bucket. PEMM with the 888 heuristic expanded 3% of the total states at depth 20, but with the 8210 heuristic 65.8% of its node expansions are at depth 20. Here there is a conflict between the g-cost ordering needed for efficient search and the g-cost ordering needed for efficient tie-breaking. More research is needed to resolve this conflict.

In general, these results confirm GR2 and show the phase transition that occurs from PEMM$_0$ to PEMM to IDA* as the problems get easier or the heuristic gets more accurate. PEMM still has room for optimization, as we will see clearly when looking at timing results.

In Table 4 we look at timing for each algorithm. Here we see a slightly different picture than node expansions. PEMM$_0$ is faster than all other approaches by a wide margin on the superflip position, except when IDA* has a 114 GB pattern database. With the 1997 heuristic PEMM is generally faster than IDA*, but with the 888 heuristic IDA* is faster than PEMM.

One significant reason for the difference is that we expand states much slower when using a heuristic. The primary cost during search is the ranking function that converts states to and from integers for storage on disk. When using the heuristics we must perform additional ranking operations, which slows down the search. While our IDA* implementation is very efficient, there is still significant research that can be done on more efficient search for PEMM.

Looking into detail at the performance of PEMM and PEMM$_0$ we find that PEMM spends from 95–99% of its time performing node expansions, with a negligible amount of time doing I/O. Before implementing a better hash table and removing the h-cost from the bucket computation PEMM was spending up to 40% of its time doing I/O, so these optimizations were effective in reducing I/O, which does not parallelize well. PEMM$_0$ spends 40–50% of its time doing I/O, except on the easiest two problems. The majority of this time (30–40% overall) is spend on DSD. This suggest that PEMM$_0$ needs different optimizations than PEMM moving forward, even though PEMM and PEMM$_0$ are using exactly the same code.

Finally, we look at the disk usage required to solve each problem. These results are in Table 5. The superflip position consistently takes 5 TB of disk to solve, while the memory required for the other problems is significantly less, depending on the strength of the heuristic, which reduces storage, and the difficulty of the problem, which increases storage.

To summarize the results here, we see that GR2 is descriptive of performance on Rubik's cube. While our implementation of $\texttt{PEMM}_0$ set a new milestone by solving a depth-20 problem without heuristic guidance, there is still room for improving the performance of PEMM and $\texttt{PEMM}_0$ by reducing the overhead from DSD in $\texttt{PEMM}_0$ and the cost of heuristic lookups in PEMM.

8. Conclusions and future work

In this paper we introduced MM, the first Bi-HS algorithm guaranteed to meet in the middle. We also introduced a framework that divides the state space into disjoint regions and allows a careful analysis of the behavior of the different algorithms in each of the regions. We studied the various types of algorithms and provided some general rules that were confirmed by our experiments.

This paper initiated this direction. Future work will continue as follows: **(1)** A deeper analysis on current and new MM variants may further deepen our knowledge in this issue. **(2)** A thorough experimental comparison should be done on more domains and with more bidirectional search algorithms. **(3)** Heuristics that are specifically designed for MM, i.e., that only return values larger than $C^*/2$ are needed [62].

Acknowledgements

Thanks to Joseph Barker for answering questions and providing extra data related to [4] and to Sandra Zilles and André Grahl Pereira for suggesting improvements in the theoretical analysis of MM. Financial support for this research was in part provided by the Natural Sciences and Engineering Research Council of Canada (NSERC) and by Israel Science Foundation (ISF) grants #417/3 and #212/17. Computational facilities for some of our experiments were provided by Compute Canada. This material is based upon work supported by the National Science Foundation under Grant No. 1551406.

Appendix A. Proofs of MMe's properties

In this appendix we prove that given an admissible heuristic (not necessarily consistent) MMe has various properties including:

(P1) MMe's forward and backward searches meet in the middle; neither search expands a node whose distance from the search's origin ($g_F(n)$ for forward search, $g_B(n)$ for backward search) is larger than $\frac{1}{2}(C^* - \epsilon)$ (Corollary 15).
(P2) MMe never expands a node whose f-value exceeds C^* (Corollary 15).
(P3) MMe returns C^* (Lemma 8 if there is no path from *start* to *goal*, Theorem 18 if there is).
(P4) If there exists a path from *start* to *goal* MMe never expands a state in both search directions (Theorem 16).

A.1. Terminology – nodes vs. states

States and nodes are different kinds of entities. A state is an immutable element of a state space, with a fixed distance to *start* and *goal*. A node, by contrast, is an entity created and updated by a search algorithm representing a path (or set of paths) in the state space. At a minimum, node n stores the path's cost ($g(n)$) and the last state on the path, which we call the state associated with n.

Rarely, if ever, is there ambiguity about which term should be used in a given context. Nodes are expanded, not states, because the process of expansion requires a g-value and states do not have g-values, only nodes do. Similarly, the regions defined in Section 6.1 (FD, NN, etc.) are regions of a state space—sets of states—because they are defined in terms of distances to *start* and *goal* and only states have such distances (nodes have g-values).

However, there are a few situations where the correct wording would be awkward. For example, it is technically incorrect to write "how many nodes are expanded in region FD?" The correct way to say this is "how many nodes are expanded whose associated state is in FD?". We prefer the simpler expression even though it is not technically correct. As a second example, property P4 has the technically incorrect wording "never expands a state in both search directions". What is meant is that if state s is associated with a node expanded in one direction s will not be associated with any of the nodes expanded in the other direction. Likewise, if we say a state s is open (or closed), we mean there is an open (or closed) node whose associated state is s.

A.2. Formal definitions, theorems, and proofs

In this appendix we will use the pseudocode in Algorithm 1 except for the stopping condition (line 7). For the moment, we will use a weaker stopping condition: MMe will terminate search as soon as $U \leq C$. This simplifies the proofs of MMe's key properties. In Section A.4 we will replace this stopping condition with the stronger stopping condition used in Algorithm 1 and show that MMe maintains all its key properties when the stronger stopping condition is used.

In this appendix we use the pr_F and pr_B (and related terms in the pseudocode such as $prmin_F$ and $prmin_B$) to refer to MMe's definition of priority[20]

$$pr_F(n) = \max(f_F(n), 2g_F(n) + \epsilon) \tag{A.1}$$

and $pr_B(n)$ is defined analogously.

The core reasoning behind the proofs of P1–P3 is as follows. As long as there remains an optimal path that has not been "found" (Definition 2, page 239) C will be less than or equal to C^* (Theorem 7), and as soon the first optimal path is found U will be set to C^* (Lemma 13). Property P3 follows directly from these two facts (Theorem 18), since they show that MMe will not halt until after U has been set to C^*. These two facts also imply that MMe will terminate without expanding any node whose priority is greater than C^* (Lemma 14), which immediately proves properties P1 and P2 (Corollary 15), since any node n with $g_X(n) > \frac{1}{2}(C^* - \epsilon)$ or $f_X(n) > C^*$ will have $pr_X(n) > C^*$ (X here is a search direction, either F or B).

We assume that all edge costs ($cost(u, v)$) are non-negative (zero-cost edges are allowed), that $start \neq goal$, and that the heuristic used by MMe in each search direction is admissible.

The following definition and Lemmas 2–4 are closely based on Hart, Nilsson, and Raphael's Lemma 1 and its proof [23].

Definition 3. Node n is "permanently closed" in the forward search direction if $n \in Closed_F$ and $g_F(n) = d(start, n)$. Likewise, n is permanently closed in the backward search direction if $n \in Closed_B$ and $g_B(n) = d(n, goal)$.

The name "permanently closed" is based on the following lemma.

Lemma 2. *If node n is permanently closed in a particular search direction at the start of some iteration, it will be permanently closed in that direction at the start of all subsequent iterations.*

Proof. This proof is for the forward search, the proof for the backward search is analogous. There is no code in Algorithm 1 to directly change $g_F(n)$ while $n \in Closed_F$, so n can only stop being permanently closed by being removed from $Closed_F$. This is possible (line 17) but only if a strictly cheaper path to n is found (line 14). This is not possible since $g_F(n) = d(start, n)$ for a node permanently closed in the forward direction. Therefore, once n is permanently closed in the forward direction it will remain so. □

Lemma 3. *Let $P = s_0, s_1, \ldots s_n$ be an optimal path from start (s_0) to any state s_n. If s_n is not permanently closed in the forward direction and either $n = 0$ or $n > 0$ and s_{n-1} is permanently closed in the forward direction, then $s_n \in Open_F$ and $g_F(s_n) = d(start, s_n)$. Analogously, let $P = s_0, s_1, \ldots s_n$ be an optimal path from any state s_0 to goal $= s_n$. If s_0 is not permanently closed in the backward direction and either $n = 0$ or $n > 0$ and s_1 is permanently closed in the backward direction, then $s_0 \in Open_B$ and $g_B(s_0) = d(s_0, goal)$.*

Proof. This proof is for the forward search, the proof for the backward search is analogous. If $n = 0$, $s_0 = start$ has not been closed in the forward direction and the lemma is true because lines 1–2 put $start \in Open_F$ with $g_F(start) = d(start, start) = 0$. Suppose $n > 0$. When s_{n-1} was expanded to become permanently closed in the forward direction s_n was generated via an optimal path (in lines 14 and 18, $g_F(n) + cost(n, c) = d(start, s_{n-1}) + cost(s_{n-1}, s_n) = d(start, s_n)$). s_n cannot have been permanently closed in the forward direction at that time because if it was, it still would be (Lemma 2). If $s_n \in Closed_F \cup Open_F$ at that time with a suboptimal g-value, then it would have been removed from $Closed_F \cup Open_F$ (line 17) and added to $Open_F$ (line 19) with $g_F = d(start, s_n)$. If $s_n \notin Closed_F \cup Open_F$ at that time, it would likewise have been added to $Open_F$ with $g_F = d(start, s_n)$ (line 19). Finally, if $s_n \in Open_F$ at that time with $g_F(s_n) = d(start, s_n)$ it would have remained so. Therefore, no matter what s_n's status was at the time s_{n-1} was expanded to become permanently closed in the forward direction, at the end of that iteration $s_n \in Open_F$ and $g_F(s_n) = d(start, s_n)$. In subsequent iterations $g_F(s_n)$ cannot have changed, since that only happens if a strictly cheaper path to s_n is found (lines 14 and 15), which is impossible. It also cannot have been closed, since if that had happened it would now be permanently closed. □

Lemma 4. *Let $P = s_0, s_1, \ldots s_n$ be an optimal path from start (s_0) to any state s_n. If s_n is not permanently closed in the forward direction then there exists an i ($0 \leq i \leq n$) such that $s_i \in Open_F$ and $g_F(s_i) = d(start, s_i)$. Let i_{min} be the smallest such i and define n_F (for path P) to be $s_{i_{min}}$. Analogously, let $P = s_0, s_1, \ldots s_n$ be an optimal path from any state s_0 to goal $= s_n$. If s_0 is not permanently closed in the backward direction then there exists a j ($0 \leq j \leq n$) such that $s_j \in Open_B$ and $g_B(s_j) = d(s_j, goal)$. Let j_{max} be the largest such j and define n_B (for path P) to be $s_{j_{max}}$.*

Proof. This proof is for the forward search, the proof for the backward search is analogous. If $start \notin Closed_F$ then $i = 0$ has the required properties ($start \in Open_F$ and $g_F(start) = d(start, start) = 0$, because of lines 1–2). Suppose $start \in Closed_F$. Let k ($0 \leq k < n$) be the largest index such that s_k is permanently closed. Such a k must exist because $start$ ($k = 0$) is permanently closed. By Lemma 3 $s_{k+1} \in Open_F$ and $g(s_{k+1}) = d(start, s_{k+1})$. Therefore $i = k + 1$ has the required properties. □

[20] MM is a special case of MMe for $\epsilon = 0$, so anything said about MMe also holds for MM.

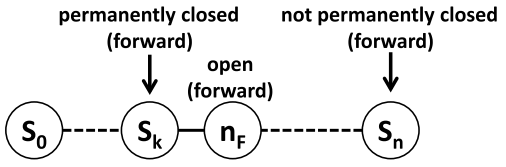

Fig. 17. Illustration of n_F for path $s_0(start), s_1, \ldots s_n$ as defined in Lemma 4.

The definition of n_F for path $s_0(start), s_1, \ldots s_n$ is illustrated in Fig. 17.

The following lemma shows that, for an optimal path P from *start* to *goal*, n_F and n_B for P are exactly the states s_i and s_j for P defined in Definition 2 (page 239).

Lemma 5. *Let $P = s_0, s_1, \ldots s_n$ be an optimal path from start(s_0) to goal(s_n) that has not been found. Then n_F and n_B, as defined in Lemma 4, both exist for P and $n_F = s_i$ and $n_B = s_j$, where s_i and s_j are as defined in Definition 2.*

Proof. Let i and j be as in Definition 2. For the forward search, $s_0, s_1, \ldots s_i$ is an optimal path from *start* to s_i and $s_i \notin Closed_F$ and therefore is not permanently closed in the forward direction. Therefore, $s_0, s_1, \ldots s_i$ satisfies the conditions of Lemma 4 for the forward direction and $n_F = s_{i'}$ exists for path $s_0, s_1, \ldots s_i$. Because $s_0, s_1, \ldots s_{i-1}$ are all in $Closed_F$, it must be that $i' = i$. Since i' is the smallest index between 0 and i such that $s_{i'} \in Open_F$ and $g_F(s_{i'}) = d(start, s_{i'})$, it is also the smallest index between 0 and n with these properties, so $s_{i'}$ is also n_F for path P. For the backward search, the reasoning is analogous. $s_j, s_{j+1}, \ldots s_n$ is an optimal path from s_j to *goal* and $s_j \notin Closed_B$ and therefore is not permanently closed in the backward direction. Therefore, $s_j, s_{j+1}, \ldots s_n$ satisfies the conditions of Lemma 4 for the backward direction and $n_B = s_{j'}$ exists for path $s_j, s_{j+1}, \ldots s_n$. Because $s_{j+1}, s_{j+2}, \ldots s_n$ are all in $Closed_B$, it must be that $j' = j$. Since j' is the largest index between j and n such that $s_{j'} \in Open_B$ and $g_B(s_{j'}) = d(s_{j'}, goal)$, it is also the largest index between 0 and n with these properties, so $s_{j'}$ is also n_B for path P. □

Lemma 6. *If $P = s_0, s_1, \ldots s_n$ is an optimal path from start(s_0) to goal(s_n) that has not been found, let $n_F = s_i$ and $n_B = s_j$ be as defined in Lemma 4. Then $g_F(n_F) + g_B(n_B) \leq C^* - \epsilon$.*

Proof. Lemma 5 guarantees that n_F and n_B exist for P. Because $i < j$, $d(start, s_i) + d(s_i, s_j) + d(s_j, goal) = C^*$, the cost of the whole path P. Because edge costs are non-negative $d(s_i, s_j) \geq \epsilon$, and therefore $d(start, s_i) + d(s_j, goal) \leq C^* - \epsilon$. The lemma follows because $g_F(n_F) = d(start, s_i)$ and $g_B(n_B) = d(s_j, goal)$. □

Theorem 7. *If, at the beginning of an* MMe *iteration, there exists an optimal path P from start to goal that has not been found, then $C \leq C^*$.*

Proof. Let n_F and n_B on path P be as defined in Lemma 4. By Lemma 6, $g_F(n_F) + g_B(n_B) \leq C^* - \epsilon$, and therefore at least one of $g_F(n_F)$ and $g_B(n_B)$ must be less than or equal to $\frac{1}{2}(C^* - \epsilon)$. Suppose, without loss of generality, that $g_F(n_F) \leq \frac{1}{2}(C^* - \epsilon)$. Then $pr_F(n_F) \leq C^*$ because $f_F(n_F) \leq C^*$ (because the heuristic h_F is admissible and $g_F(n_F)$ is optimal) and $2g_F(n_F) + \epsilon \leq C^*$. Since C is the minimum priority of all the nodes in both Open lists and $n_F \in Open_F$, C cannot be larger than $pr_F(n_F)$ and therefore $C \leq C^*$. □

Much of the following proof is closely based on Pearl's proof that A* always terminates on finite graphs (Section 3.1.2 in [50]).

Lemma 8. *For any finite state space S with non-negative edge costs* MMe *halts for any start and goal states in S. If there is no path from start to goal,* MMe *returns ∞.*

Proof. If the condition in line 7 is satisfied on some iteration, MMe will halt immediately. Suppose the condition in line 7 is never satisfied. Lines 14 and 15 ensure that MMe never expands a node via the same path twice and, because there are no negative-cost cycles[21] (non-negative edge costs guarantee this), they also ensure that MMe never expands a node via a path containing a cycle. In a finite space there are a finite number of acyclic paths to each state. Therefore each state can only be expanded a finite number of times in each search direction before it becomes permanently closed in that direction, and once it becomes permanently closed in a direction it remains so (Lemma 2). Since each iteration expands a node in one of the search directions, after a finite number of iterations MMe will have permanently closed all the nodes reachable in one of

[21] The observation that the proof only requires that there be no negative-cost cycles, as opposed to requiring all edge costs to be non-negative, is due to Gaojian Fan (University of Alberta).

the search directions, the Open list for that search direction will be empty, the condition in line 5 for continuing to iterate will not be satisfied, and MMe will halt (line 24).

If there is no path from *start* to *goal* the condition in line 20 will never be satisfied, so U will always have its initial value of ∞. If C becomes infinite—for example because all $n \in Open_F$ have $h_F(n) = \infty$ indicating that *goal* cannot be reached from them and all $n \in Open_B$ have $h_B(n) = \infty$ indicating that they cannot be reached from *start*—then the condition in line 7 will be satisfied and MMe will return $U = \infty$. If C never becomes infinite, we have shown in the previous paragraph that, after a finite number of iterations, the condition in line 5 for continuing to iterate will not be satisfied, and MMe will return ∞ (line 24). □

Implementations of search algorithms often maintain "parent pointers" for each open and closed node. The parent pointer for an open node, n, points to the node p that is responsible for n being on Open with its current $g(n)$ value. If n is closed it keeps the parent pointer it had on Open at the time it was expanded. Our pseudocode for MMe does not contain parent pointers, but the proof of Lemma 12 makes use of properties of the "generating path" of an open node n, which is the sequence of nodes defined by the parent pointers from n back to *start* (for forward search, back to *goal* for backward search). Definitions 4 and 5 are the formal definitions of "parent" and "generating path" and Lemmas 9 to 11 prove basic properties about them.

Definition 4. If $s \neq start$ and $s \in Open_F \cup Closed_F$ at the start of iteration t with $g_F(s) = g$ then $parent_F(\langle s, t, g\rangle)$ is defined to be the triple $\langle s', t', g'\rangle$ such that on iteration t', s was added to $Open_F$ with $g_F(s) = g$ as a consequence of s' being expanded in the forward direction with $g' = g_F(s') = g - cost(s', s)$. $parent_F(\langle start, t, g\rangle)$ is undefined. Similarly, if $s \neq goal$ and $s \in Open_B \cup Closed_B$ at the start of iteration t with $g_B(s) = g$ then $parent_B(\langle s, t, g\rangle)$ is defined to be the triple $\langle s', t', g'\rangle$ such that on iteration t', s was added to $Open_B$ with $g_B(s) = g$ as a consequence of s' being expanded in the backward direction with $g' = g_B(s') = g - cost(s, s')$. $parent_B(\langle goal, t, g\rangle)$ is undefined.

Lemma 9. *Suppose $s \neq start$ and $s \in Open_F \cup Closed_F$ at the start of iteration t with $g_F(s) = g$. Then:*

(a) $parent_F(\langle s, t, g\rangle)$ *exists,*
(b) $parent_F(\langle s, t, g\rangle)$ *is unique, and*
(c) *If* $parent_F(\langle s, t, g\rangle) = \langle s', t', g'\rangle$ *then* $t' < t$.

Likewise, suppose $s \neq goal$ and $s \in Open_B \cup Closed_B$ at the start of iteration t with $g_B(s) = g$. Then:

(a) $parent_B(\langle s, t, g\rangle)$ *exists,*
(b) $parent_B(\langle s, t, g\rangle)$ *is unique, and*
(c) *If* $parent_B(\langle s, t, g\rangle) = \langle s', t', g'\rangle$ *then* $t' < t$.

Proof. This proof is for the forward direction, the proof for the backward direction is analogous.

(a) If $s \neq start$, the only way it can be added to $Open_F$ is by having been generated by some other node being expanded, and the only way it can be added to $Closed_F$ is to have first been added to $Open_F$.

(b) If a state is added to $Open_F$ multiple times, it must be with a different g-value each time. Therefore s and g together uniquely identify the state (s') that caused s to be added to $Open_F$ with $g_F(s) = g$.

(c) A state cannot be on $Open_F$ or $Closed_F$ with $g_F(s) = g$ until after it has been added to $Open_F$ with $g_F(s) = g$. □

Lemma 10. *Suppose $s \neq start$ and $s \in Open_F \cup Closed_F$ at the start of iteration t with $g_F(s) = d(start, s)$. If $parent_F(\langle s, t, g\rangle) = \langle s', t', g'\rangle$, then s' is permanently closed in the forward direction. Likewise, Suppose $s \neq goal$ and $s \in Open_B \cup Closed_B$ at the start of iteration t with $g_B(s) = d(s, goal)$. If $parent_B(\langle s, t, g\rangle) = \langle s', t', g'\rangle$, then s' is permanently closed in the backward direction.*

Proof. This proof is for the forward direction, the proof for the backward direction is analogous. $d(start, s) = g_F(s) = g' + cost(s', s) \geq d(start, s') + cost(s', s) \geq d(start, s)$. Therefore all these terms are equal. In particular $g' + cost(s', s) = d(start, s') + cost(s', s)$, i.e. $g' = d(start, s')$. Hence, s' became permanently closed on iteration t' and will remain so for all future iterations (Lemma 2). □

Definition 5. If $s \neq start$ and $s \in Open_F \cup Closed_F$ at the start of iteration t with $g_F(s) = g$ then the forward generating path for $\langle s, t, g\rangle$, $GenPath_F(\langle s, t, g\rangle)$, is defined recursively:
$GenPath_F(\langle start, t, g\rangle) = \emptyset$
if $s \neq start$, $GenPath_F(\langle s, t, g\rangle) = GenPath_F(parent_F(\langle s, t, g\rangle)) :: parent_F(\langle s, t, g\rangle)$,
where $X :: Y$ adds element Y to the end of a sequence X. Likewise, if $s \neq goal$ and $s \in Open_B \cup Closed_B$ at the start of iteration t with $g_B(s) = g$ then the backward generating path for $\langle s, t, g\rangle$, $GenPath_B(\langle s, t, g\rangle)$, is defined analogously.

The forward (backward) generating path for $\langle s, t, g\rangle$ is well-defined because the recursion must terminate (t strictly decreases as each recursive call is made (Lemma 9(c)), and t cannot be negative) and it cannot terminate at any state other

than *start* (for the forward direction, *goal* for the backward direction) because $parent_F(\langle s,t,g\rangle)$ (for the forward direction, $parent_B(\langle s,t,g\rangle)$ for the backward direction) exists for all the $\langle s,t,g\rangle$ generated in this sequence of recursive calls unless $s = start$ (Lemma 9(a)).

Lemma 11. *Suppose $s \neq start$ and $s \in Open_F \cup Closed_F$ at the start of iteration t with $g_F(s) = d(start, s)$. Then all the states in $GenPath_F(\langle s,t,g\rangle)$ are permanently closed in the forward direction. Likewise, suppose $s \neq goal$ and $s \in Open_B \cup Closed_B$ at the start of iteration t with $g_B(s) = d(start, s)$. Then all the states in $GenPath_B(\langle s,t,g\rangle)$ are permanently closed in the backward direction.*

Proof. This proof is for the forward direction, the proof for the backward direction is analogous. By Lemma 10, if $parent_F(\langle s,t,g\rangle) = \langle s',t',g'\rangle$ then s' is permanently closed in the forward direction. The same lemma can therefore be applied to $\langle s',t',g'\rangle$ to show that s'' is permanently closed, where $\langle s'',t'',g''\rangle = parent_F(s',t',g')$. This process can be repeated backwards through the entire chain, showing that all states in $GenPath_F(\langle s,t,g\rangle)$ are permanently closed in the forward direction. □

Lemma 12. *If there exists a path from start to goal, MM∈ will not terminate until at least one optimal path from start to goal has been found.*

Proof. Lemma 5 guarantees that $Open_F$ and $Open_B$ are both non-empty as long as there is any optimal path from *start* to *goal* that has not been found, so the termination condition in Line 5 cannot be satisfied until all optimal paths from *start* to *goal* have been found. The only other termination condition is $U \leq C$ (the version of line 7 we are using in these proofs). Assume (for the purpose of contradiction) that this termination condition is satisfied before any optimal path from *start* to *goal* has been found. Theorem 7 shows that $C \leq C^*$ until all optimal paths from *start* to *goal* have been found, so for $U \leq C$ to hold if no optimal paths from *start* to *goal* have been found, U must be equal to C^*. We will now show that $U = C^*$ implies an optimal path from *start* to *goal* has been found, contradicting our assumption, thereby proving the lemma. U is set in line 21. On the iteration in which U was set to C^*, there must have been a child node generated, c, that satisfied the conditions of line 20, i.e. $c \in Open_B$ and $g_F(c) + g_B(c) = C^*$. The latter implies $g_F(c) = d(start, c)$ and $g_B(c) = d(c, goal)$, i.e. c is on an optimal path from *start* to *goal* with optimal g-values in both directions. This means Lemma 11 applies to c in both directions, i.e. that all the nodes on the forward and backward generating paths for c are permanently closed. The concatenation of these two paths, with c in between, is an optimal path from *start* to *goal* that was found on the iteration when U was set to C^*. □

Lemma 13. *If there exists a path from start to goal, let $P = s_0, s_1, \ldots s_n$ be the first optimal path from start(s_0) to goal(s_n) that is found during MM∈'s execution, and let $n_F = s_i$ and $n_B = s_j$ be as defined in Lemma 4 at the beginning of the iteration on which P is found. Then during that iteration U will be set to C^* in line 21.*

Proof. Lemma 12 guarantees that P exists, and Lemma 5 guarantees that n_F and n_B exist for P at the beginning of the iteration on which it becomes found. One of them must be expanded on this iteration because P's status will not change from "not found" to "found" if n_F remains on $Open_F$ and n_B remains on $Open_B$. We will complete the proof assuming that n_F is expanded. The proof in the case that n_B is expanded is analogous. We will prove the following before proving the lemma:

(a) When n_F is expanded, n_B will be generated as one of its children;
(b) When the test in Line 14 is applied to n_B ($n_B \in Open_F \cup Closed_F$ and $g_F(n_B) \leq g_F(n_F) + cost(n_F, nB)$) it will fail.

Proof of (a): Suppose n_B is not generated as a child of n_F when it is expanded in the forward direction. Then there must exist one or more nodes between them, i.e. $P = start \ldots n_F\ t_1 \ldots t_k\ n_B \ldots goal (k \geq 1)$. In order for P to be "found" at the end of this iteration, it must be the case that $t_i \in Closed_F\ \forall 1 \leq i \leq k$. Since the path $start \ldots n_F\ t_1$ is an optimal path from *start* to t_1, $t_1 \in Closed_F$ after being generated by n_F means that it had previously been generated via a different optimal path, which implies an optimal path had previously been found from *start* to all the t_i and, indeed, to n_B. Combining this previously found optimal path from *start* to n_B with the optimal path found by the backwards search from n_B to *goal* creates an optimal path from *start* to *goal* that had been found prior to P. This contradicts the premise that P is the first optimal path found from *start* to *goal*.

Proof of (b): The path $start \ldots n_F\ n_B$ is optimal, i.e. $g_F(n_F) + cost(n_F, n_B) = d(start, n_B)$. The test in line 14 can therefore only succeed if an optimal path from *start* to n_B had previously been found, which contradicts the premise that P is the first optimal path found from *start* to *goal*.

Proof of the lemma: Because of (b), the test in line 20 succeeds because n_B is a child of n_F (by (a)) and $n_B \in Open_B$ by definition. Because of (b), $g_F(n_B) + g_B(n_B) = d(start, n_B) + d(n_B, goal) = C^*$, so U will be set to C^* in line 21. □

Lemma 14. *If there exists a path from start to goal and MM∈ begins an iteration with $C > C^*$ it will terminate immediately (i.e. without expanding a node on this iteration) and return $U = C^*$.*

Proof. By Theorem 7, $C > C^*$ implies that all optimal solutions have been found, which implies (Lemma 13) $U = C^*$ so the termination criterion $U \leq C$ in line 7 is satisfied (and it is tested before a node is expanded). □

The following establishes MMe's properties P1 and P2.

Corollary 15. *MMe's forward search never expands a node n with $f_F(n) > C^*$ or $g_F(n) > \frac{1}{2}(C^* - \epsilon)$, and MMe's backward search never expands a node n with $f_B(n) > C^*$ or $g_B(n) > \frac{1}{2}(C^* - \epsilon)$.*

Proof. If there does not exist a path from *start* to *goal*, $C^* = \infty$ and nothing can be strictly larger than C^*. If there exists a path from *start* to *goal*, the proof for the forward search is as follows. The proof for the backward search is analogous. By Lemma 14, MMe's forward search never expands a node when $C > C^*$, so if n was expanded in the forward search $pr_F(n) \leq C^*$. Since $pr_F(n) = \max(f_F(n), 2g_F(n) + \epsilon)$ this means both $f_F(n)$ and $2g_F(n) + \epsilon$ are less than or equal to C^*. □

The following establishes property P4.

Theorem 16. *If MMe's heuristics are admissible and there exists a path from start to goal then MMe never expands the same state in both search directions.*

Proof. Suppose (for the purpose of contradiction) that there is a state that is expanded in the forward direction and in the backward direction. Let n be the first state to be expanded in both directions. By Corollary 15, $g_F(n) \leq \frac{1}{2}(C^* - \epsilon)$ and $g_B(n) \leq \frac{1}{2}(C^* - \epsilon)$, and the path from *start* to *goal* via n would therefore cost $C^* - \epsilon$ or less. Because C^* is finite and optimal, this implies $\epsilon = 0$ and that $g_F(n) = g_B(n) = \frac{1}{2}C^*$, i.e. n is a state on an optimal solution path and $pr_F(n) = pr_B(n) = C^*$. On the iteration when n is about to be expanded for the second time (in direction $X \in \{F, B\}$), $C = pr_X(n) = C^*$. We will prove in the next paragraph that U will be equal to C^* before n is expanded for the second time. Thus, on the iteration when n is about to be expanded for the second time the test in line 7 ($U \leq C$) will succeed and MMe will terminate immediately, without expanding n for the second time. Therefore no state is expanded in both directions.

To be expanded in both directions, n must first have been open, with its optimal g-value, in both directions. Suppose n first becomes open with its optimal g-value in the backward direction (an analogous argument holds if this happens first in the forward direction). There are two cases to consider:

Case 1. n is generated with its optimal g_F-value in the forward direction before it is expanded, with its optimal g_B-value, in the backward direction. In this case, the test in line 20 will succeed and U will be set to $g_F(n) + g_B(n) = C^*$.

Case 2. n is not generated with its optimal g_F-value in the forward direction until after it is expanded in the backward direction with its optimal g_B-value. Let P be the path from *start* to n that eventually results in n being added to $Open_F$ with its optimal g_F-value, and let p be the node on P that immediately precedes n, i.e. p is the node that is expanded in the forward direction, with its optimal g_F-value, to put n on $Open_F$ with its optimal g_F-value.[22] There are two subcases to consider.

Case 2.1. *$p \in Open_F$ with its optimal g_F-value at the time n is expanded in the backward direction with its optimal g_B-value.* In this case the test in line 20 will succeed and U will be set to $g_F(p) + g_B(p) = C^*$.

Case 2.2. p is not in $Open_F$ with its optimal g_F-value at the time n is expanded in the backward direction with its optimal g_B-value. Expanding n in the backward direction with its optimal g_B-value will add p to $Open_B$ with its optimal g_B-value and, because n is the first node expanded in both directions, p cannot be expanded in the backward direction until after n has been expanded in the forward direction. Therefore, when p is eventually generated in the forward direction with its optimal g_F-value, the test in line 20 will succeed and U will be set to $g_F(p) + g_B(p) = C^*$.

Summing up, regardless of the sequence of events, if there is any possibility of n being expanded in both directions, U is guaranteed to be set to C^* before n is expanded for the second time. □

Lemma 17. *If MMe's heuristics are admissible and there exists a path from start to goal then $Open_F$ and $Open_B$ are never empty.*

Proof. This is the proof for the forward direction. The proof for the backward direction is analogous. By Lemma 4, for $Open_F$ to be empty all states reachable from *start* must be permanently closed in the forward direction. This is impossible because *goal* is reachable from *start* but, as we will now show, it will never be permanently closed in the forward direction.

Suppose, for the sake of contradiction, that *goal* becomes permanently closed in the forward direction on some iteration, t. This implies that *goal* was added to $Open_F$ with $g_F(goal) = C^*$ on some earlier iteration and, by Theorem 16, that

[22] p is guaranteed to exist because n first becomes open with its optimal g-value in the backward direction and therefore $n \neq start$.

goal is never closed in the backward direction, i.e. that all search is in the forward direction. In particular, $goal \in Open_B$ on the iteration when *goal* was added to $Open_F$ with $g_F(goal) = C^*$ and therefore the test in line 20 would have succeeded and U would have been to set to C^*. At the beginning of iteration t we therefore would have $U = C^*$ and $C = pr_F(goal) = 2C^* + \epsilon$, so the test in line 7 ($U \leq C$) would have succeeded and MMe would have terminated immediately, without permanently closing *goal* in the forward direction. □

The following establishes MMe's property P3.

Theorem 18. *If there exists a path from start to goal* MMe *returns* $U = C^*$.

Proof. Lemma 17 has shown that, if there is a path from *start* to *goal*, MMe will never terminate by $Open_F$ or $Open_B$ becoming empty, and MMe cannot terminate if $C < C^*$, because U cannot be smaller than C^*. Therefore, MMe is certain to reach an iteration where $C \geq C^*$. If MMe reaches an iteration where $C > C^*$, Lemma 14 guarantees MMe will return $U = C^*$. The only reason it might not reach an iteration with $C > C^*$ is that it might terminate on an iteration with $C = C^*$. If termination occurs on such an iteration then we have $U \leq C = C^*$ and therefore $U = C^*$ is returned. □

A.3. MMe *with consistent heuristics*

In this section we consider additional properties of MMe if its heuristics are consistent.
The following trivial lemma will be used in the proofs of Lemmas 20 and 24.

Lemma 19. *If* $a_1 > a_2$ *and* $b_1 > b_2$ *then* $\max(a_1, b_1) > \max(a_2, b_2)$.

Proof. Suppose $\max(a_1, b_1) = a_1$. Then $a_1 \geq b_1 > b_2$. In addition, $a_1 > a_2$ is a premise of the lemma. Together these imply $a_1 > \max(a_2, b_2)$. Combining this with the symmetric argument when $\max(a_1, b_1) = b_1$ we have proven the lemma. □

Lemma 20. *If* MMe*'s heuristics are consistent and node c was added to* $Open_F$ *as the result of expanding node n, then* $pr_F(c) \geq pr_F(n)$. *Likewise if* MMe*'s heuristics are consistent and node c was added to* $Open_B$ *as the result of expanding node n, then* $pr_B(c) \geq pr_B(n)$.

Proof. This proof is for the forward search, the proof for the backward search is analogous. $f_F(c) \geq f_F(n)$ because the heuristic is consistent, and $g_F(c) = g_F(n) + cost(n, c) \geq g_F(n)$ because edge costs are non-negative. Therefore, by Lemma 19, $pr_F(c) = \max(f_F(c), 2g_F(c) + \epsilon) \geq pr_F(n) = \max(f_F(n), 2g_F(n) + \epsilon)$. □

The proof of Lemma 3 requires the ability to re-open closed nodes, and virtually all the results of the previous section depend on that lemma. With consistent heuristics we wish to remove the re-opening of closed nodes from the algorithm, so we now must re-prove the equivalent of Lemma 3 without re-opening closed nodes.

Lemma 21. *If* MMe*'s heuristics are consistent then C never decreases from one iteration* ($n - 1 \geq 1$) *of* MMe *to the next (n).*

Proof. Let $n \geq 2$ be any iteration that MMe executed beyond line 8 in solving a given problem, and let s_i denote the node chosen for expansion on iteration $i \leq n$ and X the search direction (forward or backward) used for expanding s_i, i.e. on iteration i s_i was moved from $Open_X$ to $Closed_X$ and its children added to $Open_X$ with no changes being made to the open and closed lists in the other direction. Finally, let $C_i = pr_X(s_i)$ be MMe's C value as set in line 6 on iteration i.

If $s_n \neq start$ and $s_n \neq goal$ then s_n was added to $Open_X$ with priority $pr_X(s_n)$ on some previous iteration. Let $p < n$ (p for "parent") be the iteration that most recently added s_n to $Open_X$ with priority $pr_X(s_n)$. If $p = n - 1$ then $C_n = pr_X(s_n) \geq pr_X(s_{n-1}) = C_{n-1}$ follows directly from Lemma 20. If $p < n - 1$, then s_n has been on $Open_X$ with its current pr_X-value ever since iteration $p + 1$, so it has been available for expansion, but not selected, on all iterations from $p + 1$ up to and including $n - 1$. In particular, it was on $Open_X$ with its current pr_X-value in the most recent iteration $n - 1$, where MMe chose to expand a different node s_{n-1} in a possibly different direction Y, instead of expanding s_n in direction X. Since MMe chooses a node with the smallest priority on either open list $C_n = pr_X(s_n) \geq C_{n-1} = pr_Y(s_{n-1})$.

Now consider *start* and *goal*. Before the first iteration begins $Open_F$ is initialized to contain *start* and $Open_B$ is initialized to contain *goal*. Because $g_F(start) = g_B(goal) = 0$, once these are expanded they will never be added to the open list in that direction again, since 0 is the shortest possible path to them. One of these was expanded on MMe's first iteration ($n = 1$). Suppose it was *start* (analogous reasoning applies if *goal* was expanded on the first iteration). If *goal* was never expanded then our proof is complete since it plays no role in determining a C value for any of MMe's iterations. If *goal* was first expanded in the backwards direction on the very next iteration ($n = 2$) then, because MMe chooses the node with the smallest priority on either open list we must have $C_2 = pr_B(goal) \geq pr_F(start) = C_1$. If *goal* was first expanded in the backwards direction on a subsequent iteration, $n > 2$, then similar reasoning to $p < n - 1$ case (above) applies, as follows. *goal* has been on $Open_B$ with its current pr_B value ever since the first iteration ($n = 1$), so it has been available for

expansion, but not selected, on all iterations up to and including $n-1$. In particular, it was on $Open_B$ with its initial pr_B value in the most recent iteration $n-1$, where MMe chose to expand a different node s_{n-1} in a possibly different direction Y, instead of expanding *goal* in the backwards direction. Since MMe chooses a node with the smallest priority on either open list $C_n = pr_B(goal) \geq C_{n-1} = pr_Y(s_{n-1})$. □

Lemma 22. *Suppose MMe's heuristics are consistent. Let $P = s_0, s_1, \ldots s_n$ be an optimal path from start (s_0) to any state s_n. If s_n is not permanently closed in the forward direction and either $n = 0$ or $n > 0$ and s_{n-1} is permanently closed in the forward direction, then on no iteration is $s_n \in Closed_F$ with $g_F(s_n) > d(start, s_n)$. Analogously, let $P = s_0, s_1, \ldots s_n$ be an optimal path from any state s_0 to goal $= s_n$. If s_0 is not permanently closed in the backward direction and either $n = 0$ or $n > 0$ and s_1 is permanently closed in the backward direction, then on no iteration is $s_0 \in Closed_B$ with $g_B(s_0) > d(s_0, goal)$.*

Proof. This proof is for the forward search, the proof for the backward search is analogous. If $n = 0$, $s_0 = start$ has not been closed in the forward direction and the lemma is true because $Closed_F$ is initially empty and remains so until *start* is closed, permanently, in the forward direction.

Suppose $n > 0$ and let t_1 be the iteration on which s_{n-1} was expanded to become permanently closed in the forward direction. The value of $C = C_{t_1}$ during that iteration was $pr_F^{t_1}(s_{n-1})$. If s_n was added to $Open_F$ as a result of expanding s_{n-1} on iteration t_1 then its priority $pr_F^{t_1}(s_n) \geq pr_F^{t_1}(s_{n-1})$ (by Lemma 20). If s_n was not been added to $Open_F$ as a result of expanding s_{n-1} on iteration t_1, it must have already been on $Open_F$ with its optimal $g_F(s_n)$ value, and therefore have the same priority that it would have had if it had been added as a result of expanding s_{n-1} on iteration t_1. In either case $pr_F^{t_1}(s_n) \geq pr_F^{t_1}(s_{n-1})$. If s_n had been on $Closed_F$ with a suboptimal g_F-value prior to iteration t_1 it must have been expanded on an earlier iteration $t_0 < t_1$. The value of C on that iteration was $C_{t_0} = pr_F^{t_0}(s_n)$. Because s_n had only been reached via a suboptimal path, $pr_F^{t_0}(s_n)$ (i.e. C_{t_0}) would be strictly greater than $pr_F^{t_1}(s_n)$. Hence $C_{t_0} > C_{t_1}$ even though $t_0 < t_1$, contradicting Lemma 21. So s_n cannot have been on $Closed_F$ with a suboptimal g_F-value prior to iteration t_1. On iteration t_1 it was added to $Open_F$ (if it was not already there) with an optimal g_F-value, so it will never subsequently be added to $Open_F$ with a suboptimal g_F-value, hence it will never subsequently be on $Closed_F$ with a suboptimal g_F-value. □

The following is the equivalent of Lemma 3 when MMe has consistent heuristics but does not have the ability to re-open closed nodes.[23]

Corollary 23. *Suppose MMe's heuristics are consistent and MMe does not re-open closed nodes. Let $P = s_0, s_1, \ldots s_n$ be an optimal path from start (s_0) to any state s_n. If s_n is not permanently closed in the forward direction and either $n = 0$ or $n > 0$ and s_{n-1} is permanently closed in the forward direction, then $s_n \in Open_F$ and $g_F(s_n) = d(start, s_n)$. Analogously, let $P = s_0, s_1, \ldots s_n$ be an optimal path from any state s_0 to goal $= s_n$. If s_0 is not permanently closed in the backward direction and either $n = 0$ or $n > 0$ and s_1 is permanently closed in the backward direction, then $s_0 \in Open_B$ and $g_B(s_0) = d(s_0, goal)$.*

Proof. The proof of Lemma 3 applies directly since, by Lemma 22, there is no need to test if a newly generated node is on Closed with a suboptimal value. □

Since the proofs of the other lemmas and theorems in the previous section are all based on the conclusion of Lemma 3, because of Corollary 23 they continue to hold when MMe has consistent heuristics but does not have the ability to re-open closed nodes.

Lemma 24. *If MMe's heuristics are consistent and MMe does not re-open closed nodes, then when MMe expands a node its g-value is optimal.*

Proof. This is the proof for nodes expanded in MMe's forward search. The proof for its backward search is analogous. Suppose node n has just been added to $Open_F$ (line 19) with a suboptimal cost c, i.e. $n \in Open_F$ with $g_F(n) = c > d(start, n)$. Let P be an optimal path from *start* to n. Since $n \notin Closed_F$, P satisfies the conditions of Lemma 4 and there exists a node $m = n_F \in Open_F$ on P with $g_F(m) = d(start, m)$. To prove the lemma, all that we need to show is that m will be expanded before n, i.e. that $pr_F(m) < pr_F(n)$. By definition, $pr_F(m) = \max(d(start, m) + h_F(m), 2d(start, m) + \epsilon)$ and $pr_F(n) = \max(c + h_F(n), 2c + \epsilon)$. By Lemma 19, to show that $pr_F(m) < pr_F(n)$ it suffices to show that $d(start, m) + h_F(m) < c + h_F(n)$ and that $d(start, m) < c$. The latter follows because edge costs are non-negative, so $d(start, m) \leq d(start, n) < c$. The former follows because the heuristic h_F is consistent, i.e. $h_F(m) \leq d(m, n) + h_F(n)$. This implies $d(start, m) + h_F(m) \leq d(start, m) + d(m, n) + h_F(n) = d(start, n) + h_F(n) < c + h_F(n)$. □

[23] The changes to Algorithm 1 are: (a) "∪ $Closed_F$" is removed from lines 16 and 17, and (b) the test in line 14 is changed to "($c \in Closed_F$) or ($c \in Open_F$ and $g_F(c) \leq g_F(n) + cost(n, c)$)".

Theorem 25. *Suppose* MMe*'s heuristics are consistent and* MMe *does not re-open closed nodes. If there exists a path from start to goal then* MMe *never expands a state twice.*

Proof. MMe will not expand a state twice in the same search direction because Lemma 24 guarantees that the first time a state becomes closed it becomes permanently closed. The only remaining possibility for a state to be expanded twice is that it is expanded once in the forward direction and once in the backward direction. Theorem 16 shows that this cannot happen. □

A.4. Using a stronger stopping condition

We will now show that MMe maintains its four key properties (P1–P4) if it stops as soon as any of the following conditions is true:

1. $U \leq C$ (the stopping condition used above)
2. $U \leq fmin_F$
3. $U \leq fmin_B$
4. $U \leq gmin_F + gmin_B + \epsilon$

i.e. $U \leq \max(C, fmin_F, fmin_B, gmin_F + gmin_B + \epsilon)$.

P3 continues to hold because $fmin_F$, $fmin_B$, and $gmin_F + gmin_B + \epsilon$ are all lower bounds on the cost of any solution that might be found by continuing to search. The other properties continue to hold when MMe uses the stronger stopping condition because with a stronger stopping condition MMe will execute a subset of the iterations it executed with the stopping condition used to prove MMe's properties. Since those properties were true of every iteration done with MMe's original stopping condition, they are true of every iteration done with the stronger stopping condition.

References

[1] Kazi Shamsul Arefin, Aloke Kumar Saha, A new approach of iterative deepening bi-directional heuristic front-to-front algorithm (IDBHFFA), Int. J. Electr. Comput. Sci. (IJECS-IJENS) 10 (2) (2010).
[2] Andreas Auer, Hermann Kaindl, A case study of revisiting best-first vs. depth-first search, in: Proc. 16th European Conference on Artificial Intelligence, ECAI, 2004, pp. 141–145.
[3] Joseph K. Barker, Richard E. Korf, Solving peg solitaire with bidirectional BFIDA*, in: Proc. 26th AAAI Conference on Artificial Intelligence, 2012, pp. 420–426.
[4] Joseph Kelly Barker, Richard E. Korf, Limitations of front-to-end bidirectional heuristic search, in: Proc. 29th AAAI Conference on Artificial Intelligence, 2015, pp. 1086–1092.
[5] Claude Berge, Alain Ghouila-Houri, Programming, Games and Transportation Networks, Methuen, 1965, originally published in French in 1962.
[6] Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, Jeffrey Scott Vitter, External-memory graph algorithms, in: SODA, vol. 95, 1995, pp. 139–149.
[7] Joseph Culberson, Jonathan Schaeffer, Pattern databases, Comput. Intell. 14 (3) (1998) 318–334.
[8] Henry W. Davis, Randy B. Pollack, Thomas Sudkamp, Towards a better understanding of bidirectional search, in: Proc. National Conference on Artificial Intelligence, AAAI, 1984, pp. 68–72.
[9] Dennis de Champeaux, Bidirectional heuristic search again, J. ACM 30 (1) (1983) 22–32.
[10] Dennis de Champeaux, Lenie Sint, An improved bi-directional heuristic search algorithm, in: Fourth International Joint Conference on Artificial Intelligence, 1975, pp. 309–314.
[11] Dennis de Champeaux, Lenie Sint, An improved bidirectional heuristic search algorithm, J. ACM 24 (2) (1977) 177–191.
[12] E.W. Dijkstra, A note on two problems in connexion with graphs, Numer. Math. 1 (1959) 269–271.
[13] John F. Dillenburg, Peter C. Nelson, Perimeter search, Artif. Intell. 65 (1) (1994) 165–178.
[14] James E. Doran, Double Tree Searching and the Graph Traverser, Technical Report Research Memo EPU-R-22, Dept. of Machine Intelligence and Perception, University of Edinburgh, 1966.
[15] James E. Doran, D. Michie, Experiments with the Graph Traverser Program, in: Proc. of the Royal Society, Ser. A, vol. 294, 1966, pp. 235–259.
[16] Stuart E. Dreyfus, An Appraisal of Some Shortest Path Algorithms, Technical Report RM-5433-PR, RAND Corporation, Santa Monica, California, October 1967.
[17] Jürgen Eckerle, An optimal bidirectional search algorithm, in: Proc. KI-94: Advances in Artificial Intelligence, 18th Annual German Conference on Artificial Intelligence, 1994, p. 394.
[18] Jürgen Eckerle, Thomas Ottmann, An efficient data structure for bidirectional heuristic search, in: ECAI, 1994, pp. 600–604.
[19] Ariel Felner, Carsten Moldenhauer, Nathan R. Sturtevant, Jonathan Schaeffer, Single-frontier bidirectional search, in: Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, 2010.
[20] Ariel Felner, Uzi Zahavi, Robert Holte, Jonathan Schaeffer, Nathan R. Sturtevant, Zhifu Zhang, Inconsistent heuristics in theory and practice, Artif. Intell. 175 (9–10) (2011) 1570–1603.
[21] Andrew V. Goldberg, Chris Harrelson, Computing the shortest path: A* search meets graph theory, in: Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA'05, 2005, pp. 156–165.
[22] Patrick A.V. Hall, Branch-and-bound and beyond, in: Proceedings of the 2nd International Joint Conference on Artificial Intelligence, IJCAI'71, 1971, pp. 641–650.
[23] Peter E. Hart, Nils J. Nilsson, Bertram Raphael, A formal basis for the heuristic determination of minimum cost paths, IEEE Trans. Syst. Sci. Cybern. 4 (2) (1968) 100–107.
[24] Peter E. Hart, Nils J. Nilsson, Bertram Raphael, Correction to "A formal basis for the heuristic determination of minimum cost paths", SIGART Newsl. 37 (1972) 28–29.
[25] Matthew Hatem, Ethan Burns, Wheeler Ruml, Heuristic search for large problems with real costs, in: Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, 2011, pp. 30–35.

[26] Richard V. Helgason, Jeffery L. Kennington, B. Douglas Stewart, The one-to-one shortest-path problem: an empirical analysis with the two-tree Dijkstra algorithm, Comput. Optim. Appl. 2 (1) (1993) 47–75.
[27] Malte Helmert, Landmark heuristics for the pancake problem, in: Proc. 3rd Annual Symposium on Combinatorial Search, SoCS, 2010.
[28] Malte Helmert, Gabriele Röger, How good is almost perfect?, in: Proc. 23rd AAAI Conference on Artificial Intelligence, 2008, pp. 944–949.
[29] Robert C. Holte, Ariel Felner, Guni Sharon, Nathan R. Sturtevant, Bidirectional search that is guaranteed to meet in the middle, in: Proceedings of the AAAI Conference on Artificial Intelligence, 2016.
[30] Takahiro Ikeda, Min-Yao Hsu, Hiroshi Imai, Shigeki Nishimura, Hiroshi Shimoura, Takeo Hashimoto, Kenji Tenmoku, Kunihiko Mitoh, A fast algorithm for finding better routes by AI search techniques, in: Proc. Vehicle Navigation and Information Systems Conference, 1994, pp. 291–296.
[31] Marcelo Johann, Andrew Caldwell, Andrew Kahng, Ricardo Reis, A new bidirectional heuristic shortest path search algorithm, in: Symposium on Computational Intelligence in the International ICSC Congress on Intelligent Systems and Applications, 2000.
[32] Hermann Kaindl, Gerhard Kainz, Bidirectional heuristic search reconsidered, J. Artif. Intell. Res. (JAIR) 7 (1997) 283–317.
[33] Hermann Kaindl, Gerhard Kainz, Roland Steiner, Andreas Auer, Klaus Radda, Switching from bidirectional to unidirectional search, in: Proc. 16th International Joint Conference on Artificial Intelligence, IJCAI, 1999, pp. 1178–1183.
[34] Richard E. Korf, Finding optimal solutions to Rubik's Cube using pattern databases, in: Proc. 14th National Conference on Artificial Intelligence, AAAI, 1997, pp. 700–705.
[35] Richard E. Korf, Best-first frontier search with delayed duplicate detection, in: Proc. 19th National Conference on Artificial Intelligence, AAAI, 2004, pp. 650–657.
[36] Richard E. Korf, Peter Schultze, Large-scale parallel breadth-first search, in: Proc. 20th National Conference on Artificial Intelligence, AAAI, 2005, pp. 1380–1385.
[37] Richard E. Korf, Weixiong Zhang, Ignacio Thayer, Heath Hohwald, Frontier search, J. ACM 52 (5) (2005) 715–748.
[38] Robert A. Kowalski, AND/OR graphs, theorem-proving graphs, and bidirectional search, in: B. Meltzer, D. Michie (Eds.), Mach. Intell., vol. 7, Edinburgh University Press, 1972, pp. 167–194.
[39] Daniel Kunkle, Gene Cooperman, Solving Rubik's Cube: disk is the new RAM, Commun. ACM 51 (4) (2008) 31–33.
[40] James B.H. Kwa, BS*: an admissible bidirectional staged heuristic search algorithm, Artif. Intell. 38 (1) (1989) 95–109.
[41] Carlos Linares López, Andreas Junghanns, Perimeter search performance, in: Proc. 3rd International Conference on Computers and Games, CG, 2002, pp. 345–359.
[42] Marco Lippi, Marco Ernandes, Ariel Felner, Efficient single frontier bidirectional search, in: Proceedings of the Fifth Annual Symposium on Combinatorial Search, SoCS, 2012.
[43] Michael Luby, Prabhakar Ragde, A bidirectional shortest-path algorithm with good average-case behavior, Algorithmica 4 (1) (1989) 551–567.
[44] Giovanni Manzini, BIDA*: an improved perimeter search algorithm, Artif. Intell. 75 (2) (1995) 347–360.
[45] Alberto Martelli, On the complexity of admissible search algorithms, Artif. Intell. 8 (1) (1977) 1–13.
[46] Th. Mohr, C. Pasche, A parallel shortest path algorithm, Computing 40 (4) (1988) 281–292.
[47] T.A.J. Nicholson, Finding the shortest route between two points in a network, Comput. J. 9 (3) (1966) 275–280.
[48] Robert Niewiadomski, José Nelson Amaral, Robert C. Holte, A parallel external-memory frontier breadth-first traversal algorithm for clusters of workstations, in: International Conference on Parallel Processing (ICPP), 2006, pp. 531–538.
[49] Nils J. Nilsson, Principles of Artificial Intelligence, Tioga Press, 1980.
[50] Judea Pearl, Heuristics – Intelligent Search Strategies for Computer Problem Solving, Addison-Wesley, 1984.
[51] Wim Pijls, Henk Post, A new bidirectional algorithm for shortest paths, Eur. J. Oper. Res. 198 (2009) 363–369.
[52] Wim Pijls, Henk Post, Note on "A new bidirectional algorithm for shortest paths", Eur. J. Oper. Res. 207 (2) (2010) 1140–1141.
[53] Ira Pohl, Bi-Directional and Heuristic Search in Path Problems, Technical Report 104, Stanford Linear Accelerator Center, 1969.
[54] George Politowski, Ira Pohl, D-node retargeting in bidirectional heuristic search, in: Proc. National Conference on Artificial Intelligence, AAAI, 1984, pp. 274–277.
[55] Francisco Javier Pulido, Lawrence Mandow, José-Luis Pérez de la Cruz, A two-phase bidirectional heuristic search algorithm, in: Proc. 6th Starting AI Researchers Symposium, STAIRS, 2012, pp. 240–251.
[56] Alexander Reinefeld, Volker Schnecke, AIDA*-asynchronous parallel IDA*, in: Proceedings of the Biennial Conference-Canadian Society for Computational Studies of Intelligence, Canadian Information Processing Society, 1994, pp. 295–302.
[57] Tomas Rokicki, Herbert Kociemba, Morley Davidson, John Dethridge, The diameter of the Rubik's Cube group is twenty, SIAM J. Discrete Math. 27 (2) (2013) 1082–1105.
[58] Samir K. Sadhukhan, A new approach to bidirectional heuristic search using error functions, in: Proc. 1st International Conference on Intelligent Infrastructure at the 47th Annual National Convention COMPUTER SOCIETY of INDIA, CSI-2012, 2012.
[59] Guni Sharon, Robert Holte, Nathan Sturtevant, Ariel Felner, An improved priority function for MM, in: SoCS, 2016.
[60] N. Sturtevant, Benchmarks for grid-based pathfinding, IEEE Trans. Comput. Intell. AI Games 4 (2) (2012) 144–148.
[61] Nathan Sturtevant, Jingwei Chen, External memory bidirectional search, in: International Joint Conference on Artificial Intelligence, IJCAI, 2016.
[62] Nathan R. Sturtevant, Ariel Felner, Malte Helmert, Value compression of pattern databases, in: AAAI Conference on Artificial Intelligence, 2017.
[63] Nathan R. Sturtevant, Matthew J. Rutherford, Minimizing writes in parallel external memory search, in: Proc. 23rd International Joint Conference on Artificial Intelligence, IJCAI, 2013.
[64] Christopher Makoto Wilt, Wheeler Ruml, Robust bidirectional search via heuristic improvement, in: Proc. 27th AAAI Conference on Artificial Intelligence, 2013, pp. 954–961.