The Thirty-Second International Florida
Artificial Intelligence Research Society Conference (FLAIRS-32)

Front-to-Front Bidirectional Best-First Search Reconsidered

Leopold E. Mayer, Kurt D. Krebsbach
Department of Mathematics and Computer Science
Lawrence University, Appleton, Wisconsin 54911
{leopold.e.mayer, kurt.krebsbach}@lawrence.edu

Abstract

We present several new algorithms for bidirectional best-first search that employ a *front-to-front* strategy of estimating distances from newly-generated frontier nodes in one search direction to existing frontier nodes in the other search direction, rather than estimating distances to terminal nodes in both searches. Unlike previous front-to-front strategies that use a shared priority queue to manage both frontiers, we use a separate data structure for each search, and choose that data structure to minimize the amount of computational effort required by the best-first search algorithm it supports. We demonstrate several results. First, we show that *Bidirectional Front-to-Front Greedy* (BFFG) is able to quickly find sub-optimal solutions to very large state-space problems and with a small fraction of nodes expanded (and stored) compared to other unidirectional and bidirectional greedy techniques. Secondly, we show that *Bidirectional Front-to-Front A** (BFFA*) similarly outperforms both *Unidirectional A** and *Bidirectional Front-to-End A** (BFEA*) in terms of node expansions when searching for optimal solutions. Finally, we describe three improvements to BFFA*, each of which reduces the overall runtime by limiting the number of opposing frontier nodes that need be considered while preserving the optimality criterion.

Best-First Search

Best-first search is so-called because we choose to select for expansion the "best-looking" node from the candidates that have been generated but not yet expanded. We refer to this list of candidate (leaf) nodes as the *frontier* of the search. Table 1 provides a guide to both the acronyms we use for the algorithms we discuss, and the (slightly extended) conventional notation used when discussing best-first search algorithms.

Since its introduction, the A* search algorithm (Hart, Nilsson, and Raphael 1968) has become the standard by which best-first search algorithms are judged. *Unidirectional A** expands nodes based on the sum (denoted f) of the cost accumulated along a path (g), and a heuristic estimate of the distance from that node to the nearest goal state (h). This node-expansion strategy guarantees that A* is complete and optimal, provided that h never overestimates the actual distance to a goal state (Dechter and Pearl 1985).

Notation	Meaning
BFFG	*Bidirectional Front-to-Front Greedy*
BFEG	*Bidirectional Front-to-End Greedy*
BFFA*	*Bidirectional Front-to-Front A**
BFEA*	*Bidirectional Front-to-End A**
GR	*Unidirectional Greedy*
A*	*Unidirectional A**
S	Start node (root)
T	Terminal node (goal)
$g(n)$	Actual distance of $S \to n$
$h(n)$	Estimated distance of $n \to T$
$f(n)$	Estimated distance of $S \to n \to T$
$h(n,m)$	Estimated distance of $n \to m$
$f(n,m)$	Estimated distance of $S \to n \to m \to T$
$d(n,m)$	Actual distance of $n \to m$
$c(n_1,n_2)$	Step cost from n_1 to n_2

Table 1: Bidirectional Search Notational Conventions. Note also that whenever we use n, n_i, m, m_i, the all the n nodes are on the opposite search as the m nodes. There are technically forward and reverse functions for g, h and f, but for concision we leave those out, and which function we use will be dependent on the context of the nodes.

Unfortunately, A* requires an amount of memory exponential in the length of the solution path, causing the algorithm to run out of memory before solving difficult problems. For problems not requiring an optimal solution, a simplification of A* in which g, the accumulated cost, is ignored ($f = h$) is known as *Unidirectional Greedy* (GR), since the search is solely guided by estimated nearness to the goal without regard for the accumulated path cost. For this reason, GR tends to follow fewer (but longer) paths, often finding a solution very quickly without exploring much of the state space. This efficiency comes at the cost of low-quality solutions.

Bidirectional Best-First Search

To reduce the exponential memory required for best-first search, researchers have proposed to conduct one such search *forward* from start to goal, while simultaneously conducting another search *backward* from goal to start (Pohl 1971). While such a strategy is only possible if certain conditions hold—e.g., that *predecessor states* of a node can be computed to support the backward search—it holds the promise of halving the exponent of the space and time complexity by stopping when a common node is found between the two searches.

We have several choices when designing a bidirectional best-first search algorithm. For example, the algorithms we design and implement in this paper (BFFG and BFFA*) as well as the algorithms we implement for empirical comparison (A*, GR, BFEG, and BFEA*) use two separate queues to manage two frontiers, one for each search direction. Furthermore, these algorithms strictly alternate between expanding a single node in one direction followed a single node in the other.

Alternative approaches to managing frontiers include using a single queue to represent the union of the frontiers, from which the best-looking node is chosen for expansion regardless of direction. Each node is marked as forward or backward for the purpose of generating appropriate successor nodes, but nodes from each search could still be compared according to f-value (see (Kaindl and Kainz 1997) for a thorough treatment). A second single-queue approach includes nodes that represent a pair of states: one from each search direction (Felner et al. 2010).

Others have suggested more elaborate switching schemes for two-queue approaches. For example, (Holte et al. 2016) switch between queues to ensure that the two searches "meet in the middle." Still others expand many nodes in one direction, then run the rest of the search from the other direction, possibly with different search algorithms (Kaindl et al. 1999).

Finally, there is a choice in bidirectional search regarding precisely what the heuristic estimate $h(n)$ measures. We now turn our attention to these strategies, which can be classified as *front-to-end* and *front-to-front* bidirectional search strategies.

Bidirectional Front-to-End Search

In Bidirectional Front-to-End search, two heuristic functions are necessary. The first, h_f, is the estimated distance from a node to the goal state, and is used by the forward search. The second, h_b is the estimated distance from a node to the start state, using reverse actions, and it is used by the backward search. In the case of BFEG the f-value of a node is equal to its h-value, and in the case of BFEA* the f-value of a node is equal to the sum of its h-value and g-value. These are exactly the same as *Unidirectional Greedy* and *Unidirectional A**, and that is because Front-to-End bidirectional search is precisely running two unidirectional searches in parallel. The only communication between the two searches is that whenever a node is expanded, the algorithm checks if there is a node in the other search's open or closed list representing the same state, and if there is it reconstructs a solution path from the two parts. Since BFEG is sub-optimal anyway, there is no point in continuing search after the first solution is found. But for BFEA*, the first common node found might not constitute an optimal solution, and search may need to continue.

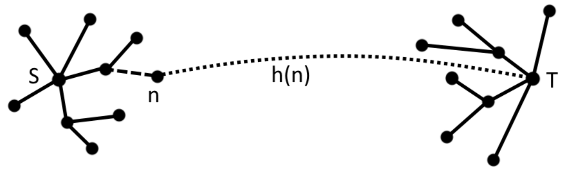


Figure 1: *Front-to-End:* Bidirectional Heuristic Search using front-to-end estimates. When a node n is generated in the forward search, $h(n)$ estimates the distance to the goal. Since n is in the forward search, it estimates to T. Nodes generated in the backward search instead estimate to S.

Bidirectional Front-to-Front Search

In Bidirectional Front-to-Front (Sint and de Champeaux 1977), when a node is added to a frontier, we estimate the distance to the frontier of the opposite search. Since the frontier is always in flux, we need to generalize the notion of heuristic function. For any two nodes n_1, n_2, let $h(n_1, n_2)$ be the estimated cost to move from the state represented by n_1 to the state represented by n_2. All the familiar definitions are similarly generalized: h is admissible if, for all n_1, n_2 we have $h(n_1, n_2) \leq d(n_1, n_2)$. We call h consistent if for all n_1, n_2, n_3 with n_1 and n_2 adjacent, we have $h(n_1, n_3) \leq c(n_1, n_2) + h(n_2, n_3)$. The h-value of a node will ultimately be determined by the estimate to a single node on the other frontier, and strategies for determining which node to estimate to can vary. For BFFG the f-value is equal to the h-value. In theory, it makes sense to estimate to the node that minimizes h, but in practice it is effective to use the top of the opposite frontier (explained in more depth shortly). Unless otherwise stated, we assume BFFG always only estimates to the top of the other frontier.

For BFFA* the f-value of a node is the sum of three terms: the h-value, the g-value, and the g-value of the node on the other frontier being estimated to. For admissible heuristics, $f(n, m)$ will always be less than or equal to the actual shortest path from S to T through n and m. To preserve optimality, a node must be chosen which minimizes this f-value. This guarantees that for each node n, we have $f(n)$ less than or equal to the actual shortest path through n. This is because the actual shortest path will necessarily go through some node m on the other frontier, and by the way we chose the f-value we have $f(n) \leq f(n, m)$. In Front-to-Front, the two searches are in constant communication; a node from one search direction makes use of the progress made in the other direction by estimating to a node on its frontier. Since part of the f-value comes from the g-value of the node being estimated to, and g-values are never less than the corresponding h-values, the overall f-values of nodes in BFFA* will be greater than or equal to the f-values of the same nodes in BFEA*, which means more nodes will be pruned. An important note is that this version of the algorithm is slightly different from the originally proposed Front-to-Front A* from (Sint and de Champeaux 1977). In that version, when the time came to expand another node, $f(n, m)$ was computed for every node n on the forward frontier as well as every node m on the reverse frontier, and when $f(n', m')$ was found to be the minimum, one of n' or m' was to be expanded. In our version, when any node n is generated, its f-value, $f(n)$, is computed, stored, and never recomputed. This implies that $f(n)$ is not always "up-to-date": when n is expanded, the node it originally estimated to may no longer exist on the opposite frontier. There is a tradeoff here: it takes more computations to keep everything "up-to-date", but the f-values would generally be higher, resulting in more nodes being pruned.

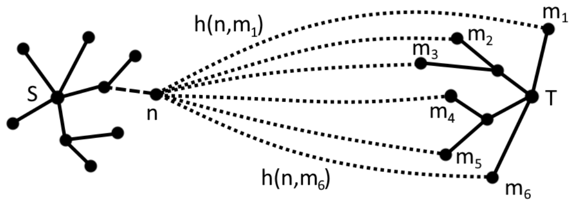


Figure 2: *Front-to-Front:* Bidirectional Heuristic Search using front-to-front estimates. When a node is generated, it estimates to individual nodes on the opposite frontier.

Empirical Results

We now present empirical results of running the search algorithms discussed under a variety of assumptions.

Greedy Bidirectional Search

In our first experiment, we compare GR to both BFEG and BFFG (Figure 3). The graph shows the number of nodes expanded and solution cost from 10 randomly-generated 35-puzzles. While there is not much difference in solution quality, front-to-front clearly generates significantly fewer nodes on average, and front-to-end performs similarly to unidirectional. More precisely, the average number of nodes expanded for BFFG was only 10.1% of the average for GR, while BFEG expanded 8.1% more nodes than GR, on average.

In the second greedy experiment, we investigate if it is justifiable to always only estimate to the top node on the other frontier. In Figure 4, we compare the number of nodes expanded and the total cost of the solution found to a quantity we call "sample size". The sample size refers to how many nodes are estimated to whenever a new node is generated. Specifically, BFFG with a sample size of k means that whenever a new node is generated, it estimates to the top k nodes on the other frontier, and the smallest of those estimates is used for its h-value. Our implementation used a heap for the frontier, so "the top k nodes" does not necessarily mean the lowest k nodes when ordered by f-value, but rather the first k nodes reached by iterating through the heap. Figure 4 shows a minor trend toward better performance as we increase the sample size, but there appears to be a lot of unpredictability.

We also note that BFEG performs well in very large state spaces. We used randomly-generated 35-puzzles for Figure 3 because GR and BFEG ran out of memory too regularly on 48-puzzles, but BFFG was able to solve 93 out of 100 randomly-generated 80-puzzles, which have a state space size of approximately 3.58×10^{118}.

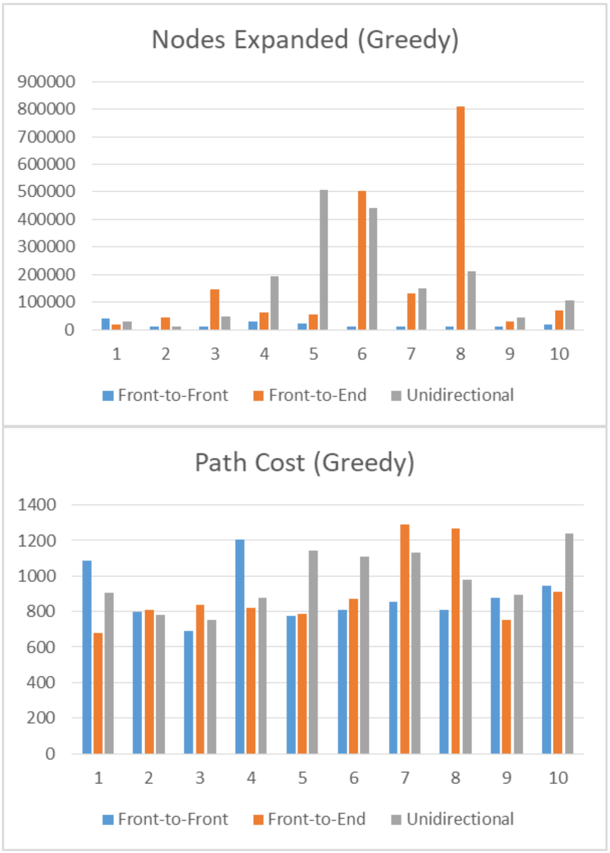


Figure 3: *Greedy: Unidirectional vs. Front-to-End vs. Front-to-Front* Number of nodes expanded and solution cost from 10 randomly-generated 35-puzzles, solved by the three algorithms using the Manhattan heuristic.

A* Bidirectional Search

We now present empirical results of running the three forms of A* search under varying assumptions. In all cases, we preserve optimality, in contrast to the greedy experiments reported on above.

In our first experiment, we compare A* to both BFEA* and BFFA* on 10 randomly-generated 11-puzzles (3×4 sliding-tile puzzle) (Figure 5). From the graph, BFFA* has the clear advantage. In almost every case, it expanded only $5 - 10\%$ of the nodes A* did. BFEA* performed better than its greedy counterpart, but still was much worse than BFFA*. The biggest challenge for BFFA* that is not factored into this graph is the runtime. BFFA* needs to scan the entire opposite frontier each time a node is expanded. A more accurate way to compare these algorithms, and what we propose as the standard for future analysis of Front-to-Front algorithms, is to compare instead how many h-calculations the algorithm performs. For A* and BFEA* this measure is equal to the nodes expanded plus the size of the frontier when the search ends. This is simply because

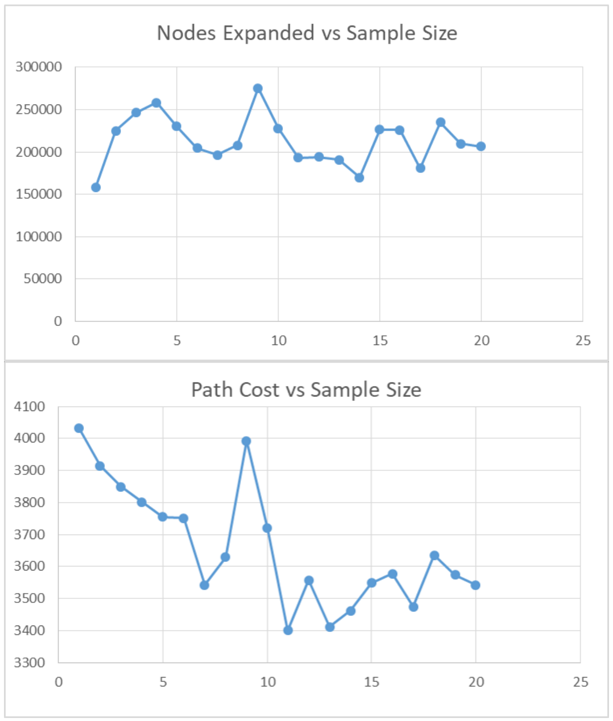


Figure 4: B*FFG with varied sample size on opposing frontier:* Average number of nodes expanded and average path cost over 100 randomly-generated 80-puzzles, as a function of the "sample size" on the opposite frontier.

a node's f-value is computed when it is added to the frontier, and for those two algorithms, determining a node's f-value requires a single h-calculation. A consequence of is that we can now quantify how many h-calculations per node BFFA* can afford while still being competitive with A*. This will depend on the size of the problem, but our empirical data show that BFFA* needs to average 10-20 h-calculations per node in order to be competitive with A* for 11-puzzles. Currently, BFFA* does as many h-calculations as there are nodes on the opposite frontier, but in the next section we introduce and discuss strategies for driving the number of h-calculations down.

A* Front-to-Front Improvements

The primary disadvantage of *Bidirectional Front-to-Front A** is that it takes so long to scan the opposite frontier. Under the naive implementation of the algorithm, the entire frontier needs to be evaluated each time a node is generated. This downside has historically prevented BFFA* from serious consideration. The goal of this section is to explore improvements to limit the amount of scanning required.

Improvement 1: Ordered Scanning

Suppose n is a node being generated and m is a node on the opposite frontier. Then we necessarily have $f(n, m) \geq f(m)$. To see this, recall that for an admissible heuristic, f-values are never overestimates. The value $f(m)$ never over-

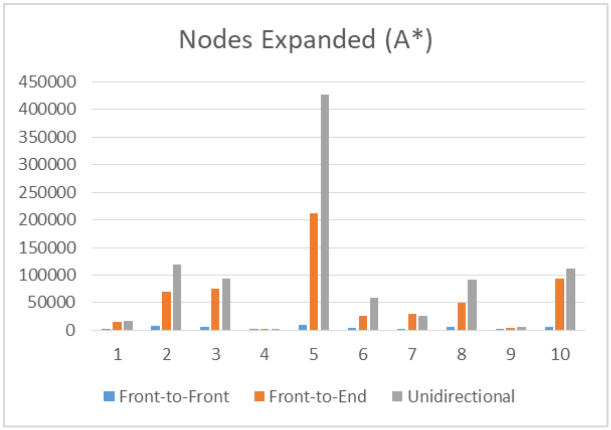
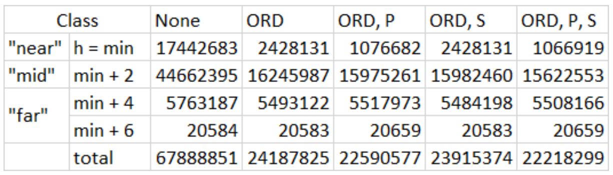


Figure 5: *A*: Unidirectional vs. Front-to-End vs. Front-to-Front:* Number of nodes expanded for 10 randomly-generated 11-puzzles, solved by the three algorithms using the Manhattan heuristic. Solution cost is not compared because all three algorithms are optimal.

estimates the shortest solution path through m, and $f(n, m)$ never overestimates the shortest solution through n and m. Since m is already on the frontier, and n is just being added, there was some ancestor n' of n that was on the frontier when m was generated. That means $f(m, n') \geq f(m)$, but if the heuristic is consistent we also have $f(m, n) \geq f(m, n')$. Now we can describe the improvement. At all times, keep both frontiers ordered by f-value. When a node n is expanded, first estimate to m_0, the node in the opposite frontier with minimum f-value. Continue scanning the opposite frontier in ascending order, keeping track of the lowest f-value so far, until the f-value of the next node is greater than or equal to the current best f-value. For each node m_i yet to be scanned, we have $f(m_i)$ no less than the current best, so the remaining values $f(n, m_i)$ will also be no less than the current best. Thus, the current best is provably the best in the entire other frontier, and there is no need to continue scanning. As Figure 6 shows, Ordered Scanning was able to reduce the number of h-calculations by 64%.

Analysis of number of h-calculations

Suppose a node n is added to the frontier. Although there is no way to know $f(n)$ before scanning, BFFA* with Ordered Scanning necessarily scans every node m with $f(m) < f(n)$. This means that the number of h-calculations for a node is dependent on its f-value, and also on the state of the frontier. Figure 6 depicts the number of h-calculations across a single instance of BFFA* solving a 15-puzzle. The first row is the number of h-calculations done from nodes whose f-value ended up being equal to the minimum f-value on the opposite frontier. We will call these nodes "near". The second row accounts for all the h-calculations done from nodes whose f-value ended up being two more than the minimum f-value on the opposite frontier. We call these nodes “mid”. This continues for the rest of the rows[1], whose nodes we will call “far”. The results also go along with intuition. We expect most nodes to be “near”, and according to the comment above, “near” nodes require relatively few h-calculations. It is more unlikely to find “far” nodes, but when we do, more calculations are required. Ultimately, the most work is done on the “mid” nodes, since these are still common, but also require scanning a significant portion of the opposite frontier.



Class		None	ORD	ORD, P	ORD, S	ORD, P, S
"near"	h = min	17442683	2428131	1076682	2428131	1066919
"mid"	min + 2	44662395	16245987	15975261	15982460	15622553
"far"	min + 4	5763187	5493122	5517973	5484198	5508166
	min + 6	20584	20583	20659	20583	20659
	total	67888851	24187825	22590577	23915374	22218299

Figure 6: Bidirectional Front-to-Front A* *h-calculation count:* The number of h-calculations made while solving a single randomly-generated 15-puzzle, classified by the difference between the node’s f-value and the minimum f-value on the other frontier. We tested BFFA* combinations of the three improvements: Ordered Scanning (ORD), Parent Check (P) and Sibling Check (S).

Improvement 2: Parent Check

Currently, BFFA* only remembers nodes’ f-values, and there’s no keeping track of which node was ultimately estimated to. Parent Check does keep track, and it uses that information to save on the number of h-calculations. When generating a node, find the node that its parent pointed to, then estimate to the descendants of that node which are currently on the opposite frontier. Scanning the rest of the frontier will only be necessary if there’s a possibility of finding a lower f-value than the one created by the initial guess. For example, in Figure 7, node n_1 was just expanded, so we need to compute an f-value for the newly generated n_2. Parent Check tells us that n_1 estimated to m_1 (i.e. $f(n_1) = f(n_1, m_1)$). Since m_2 and m_3 are the frontier nodes descended from m_1, we first calculate $f(n_2, m_2)$ and $f(n_2, m_3)$. This is a good idea because m_1 seemed closest to n_1, so there is a good chance that the descendants of m_1 will be close to the children of n_1. This improvement mainly targets nodes that fall into the “near” section of the h-count distribution. Even if a node’s f-value is equal to the minimum in the opposite frontier, there could be lots of nodes in the frontier with equal f-values, and there is still the potential of having to scan most of those before finding one that yields the correct f-value. The parent check makes an initial guess, and tries find an f-value right away that can’t be improved. If the node being generated is not in the “near” section, scanning will still be necessary, since by definition, the node’s f-value is greater than the minimum of the frontier. The initial guess can only prevent having to scan nodes with f-value equal to the node being evaluated, and so the farther from “near” a node is, the less effective Parent Check will be. This line of reasoning is confirmed by Figure 6. The column labeled “ORD, P” shows the distribution of h-calculations for BFFA* with Parent Check. As expected, far fewer calculations were made in the “near” section, slightly fewer in “mid”, and slightly more in “far”.

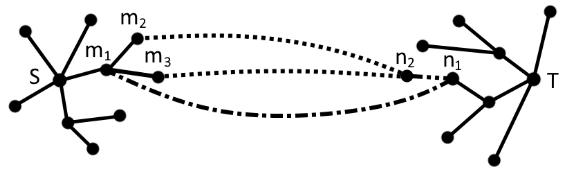


Figure 7: *Parent Check Graph:* Node n_2 is being added to the frontier. Its parent, n_1, ended up estimating to node m_1, which has descendants m_2, m_3 on the forward frontier. Using Parent Check, n_2 would first estimate to m_2 and m_3, and only scan the rest of the frontier as necessary.

Improvement 3: Sibling Check

In this section we describe an improvement that only works on environments where every action has an inverse. Once we’ve calculated the f-value for a node, we can use that to bound the possible f-values of its siblings. Given an optimal path through its sibling, a path to the node can easily be created with only slightly higher cost. Since that path cannot be lower than the node’s f-value, we can bound the f-value of the sibling. A higher lower-bound is useful because it could result in fewer nodes being scanned. In Figure 8, let c be the cost of moving from n_1 to n_2. If there exists a path of length $f(n_3)$, then the path can be modified in the way Figure 8 illustrates to end up with a path through n_2 of length $f(n_3)+2c$. Since an admissible heuristic implies f-values are never overestimates, it follows that $f(n_2) \leq f(n_3)+2c$, or $f(n_3) \geq f(n_2)-2c$. This is useful, because now when we want to compute $f(n_3)$, we might not have to scan every opposing frontier node with f-value less than $f(n_3)$. If we scan a node m on the opposing frontier such that $f(n_3, m) = f(n_2) - 2c$, we can stop scanning, since it will be impossible to get a lower f-value. Essentially, Sibling Check makes note whenever a “far” node is found, and then lowers the expectations for the siblings of that “far” node. Because of this, Sibling Check is designed to reduce the number of h-calculations spent on “far” and “mid” nodes. For “near” nodes, the lower bound is less than or equal to the minimum of the opposite frontier, which was already a lower bound on $f(n_3)$, so there is nothing to be gained. Figure 6 shows that Sibling Check decreased the number of h-calculations for “mid” and “far” nodes, and didn’t change the count for “near” nodes, which is exactly what was expected.

Future Work

We are currently exploring several directions. Parent Check targeted “near” nodes, and Sibling Check targeted siblings of “far” nodes, but a clear majority of the h-calculations

[1]Oddly enough, our tests suggest that N-Puzzle either only had even f-values or only had odd f-values. There also never were two nodes in the same frontier whose difference in f-value exceeded 6.

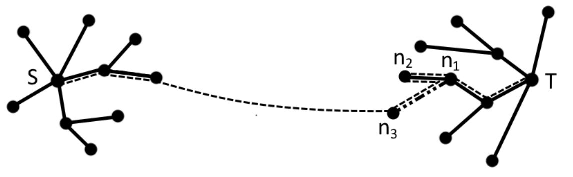


Figure 8: *Sibling Check Graph:* Node n_2 has already been added to the frontier, and node n_3 will be added next. Any solution passing through n_3 can be modified to pass through n_2. This is done by adding a step from n_1 to n_2 and another from n_2 back to n_1. We can use this construction to bound $f(n_3)$.

came from the "mid" section. Developing improvements that target that class more directly could have a significant impact on performance.

The improvement that led to the greatest reduction in runtime for BFFA* was Ordered Scanning. It worked by being able to iterate through the frontiers in order, and to do that it used a better data structure than a simple heap for the frontiers. Perhaps other data structures would be even better-suited for the frontiers, allowing the algorithm to find the minimum estimate faster.

Because questions still remain about the number of nodes expanded by BFFA* we would like to identify certain conditions that allow us to prove that BFFA* expands fewer nodes than A*, as the empirical results suggest.

In our version of BFFG only the top of the frontier was considered, and that allowed the algorithm to avoid the runtime challenges faced by BFFA*. Only estimating to the top of the other frontier (or the top k nodes on the other frontier) for BFFA* would greatly reduce the time needed to expand each node. Optimality would be sacrificed, but near-optimal solutions might often be found. Characterizing the relationship between k and bounds on near-optimality would be a useful result.

Conclusion

We have introduced two new algorithms, BFFG (suboptimal) and BFFA* (optimal), that employ a *front-to-front* bidirectional search strategy that uses alternative data structures to manage the opposing search frontiers. We have shown that both algorithms expand many fewer nodes than competing approaches. Finally, we develop and evaluate three improvements to BFFA* to limit the number of h-calculations required for each node generation while preserving optimality.

References

Dechter, R., and Pearl, J. 1985. Generalized best-first search strategies and the optimality of A*. *Journal of the ACM* 32(3):505–536.

Felner, A.; Moldenhauer, C.; Sturtevant, N. R.; and Schaeffer, J. 2010. Single-frontier bidirectional search. In *AAAI*.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* SSC-4(2):100–107.

Holte, R. C.; Felner, A.; Sharon, G.; and Sturtevant, N. R. 2016. Bidirectional search that is guaranteed to meet in the middle. In *AAAI*, volume 16, 3411–3417.

Kaindl, H., and Kainz, G. 1997. Bidirectional heuristic search reconsidered. *Journal of Artificial Intelligence Research* 7:283–317.

Kaindl, H.; Kainz, G.; Steiner, R.; Auer, A.; and Radda, K. 1999. Switching from bidirectional to unidirectional search. In *IJCAI*, 1178–1183.

Pohl, I. 1971. Bi-directional search. In Meltzer, B., and Michie, D., eds., *Machine Intelligence 6*. 127–140.

Sint, L., and de Champeaux, D. 1977. An improved bidirectional heuristic search algorithm. *Journal of the ACM (JACM)* 24(2):177–191.