Front-to-End Bidirectional Heuristic Search with Near-Optimal Node Expansions

Jingwei Chen
University of Denver
Denver, CO, USA
jingchen@cs.du.edu

Robert C. Holte
University of Alberta
Edmonton, AB, Canada
rholte@ualberta.ca

Sandra Zilles
University of Regina
Regina, SK, Canada
zilles@uregina.ca

Nathan R. Sturtevant
University of Denver
Denver, CO, USA
sturtevant@cs.du.edu

Abstract

It is well-known that any admissible unidirectional heuristic search algorithm must expand all states whose f-value is smaller than the optimal solution cost when using a consistent heuristic. Such states are called “surely expanded” (s.e.). A recent study characterized s.e. pairs of states for *bidirectional* search with consistent heuristics: if a pair of states is s.e. then at least one of the two states must be expanded. This paper derives a lower bound, VC, on the minimum number of expansions required to cover all s.e. pairs, and present a new admissible front-to-end bidirectional heuristic search algorithm, Near-Optimal Bidirectional Search (`NBS`), that is guaranteed to do no more than 2VC expansions. We further prove that no admissible front-to-end algorithm has a worst case better than 2VC. Experimental results show that `NBS` competes with or outperforms existing bidirectional search algorithms, and often outperforms A* as well.

1 Introduction

One method of formally assessing the efficiency of a heuristic search algorithm is to establish upper and lower bounds on the number of nodes it expands on any given problem instance I. Such bounds can then be compared to a theoretical minimum that a competing heuristic search algorithm would have to expand on I. In this context, one is interested in finding sufficient conditions for node expansion, i.e., conditions describing nodes that must provably be expanded by any competing algorithm. In a unidirectional search an algorithm must expand every node whose f-value is less than the optimal solution cost; this condition establishes the optimality of A* [Dechter and Pearl, 1985].

Sufficient conditions for node expansion have recently been developed for front-to-end bidirectional heuristic search [Eckerle *et al.*, 2017], but no existing front-to-end bidirectional search algorithm is provably optimal. In this paper, we use these sufficient conditions to derive a simple graph-theoretic characterization of nodes that must provably be expanded on a problem instance I by any admissible front-to-end bidirectional search algorithm given a consistent heuristic. In particular, the set of nodes expanded must correspond to a vertex cover of a specific graph derived from I. We then adapt a known vertex cover algorithm [Papadimitriou and Steiglitz, 1982] into a new admissible front-to-end bidirectional search algorithm, `NBS` (Near-Optimal Bidirectional Search), and prove that `NBS` never expands more than twice the number of nodes contained in a minimum vertex cover. Hence, the number of nodes expanded by `NBS` is provably within a factor of two of optimal.

We further establish that no admissible bidirectional front-to-end algorithm can be better than `NBS` in the worst case. In that sense, we formally verify that `NBS` is near-optimal in the general case and optimal in the worst case. In an experimental study on a set of standard benchmark problems, `NBS` either competes with or outperforms existing bidirectional search algorithms, and it often outperforms the unidirectional algorithm A*, especially when the heuristic is weak or the problem instance is hard.

2 Related Work

Bidirectional search has a long history, beginning with bidirectional brute force search [Nicholson, 1966], and proceeding to heuristic search algorithms such as BHPA [Pohl, 1971]. Other notable algorithms include BS* [Kwa, 1989], which avoids re-expanding states in both directions, and MM [Holte *et al.*, 2016], which ensures that the search frontiers meet in the middle. Along with these algorithms there have been explanations for the poor performance of bidirectional heuristic search, including that the frontiers miss [Nilsson, 1982] or that the frontiers meet early, and a long time is spent proving the optimal solution [Kaindl and Kainz, 1997]. Recent work has refined this, showing that with strong heuristics the frontiers meet later [Barker and Korf, 2015].

3 Terminology and Notation

We use the same notation and terminology as [Eckerle *et al.*, 2017]. A state space G is a finite directed graph whose vertices are states and whose edges are pairs of states.[1] Each edge (u, v) has a cost $c(u, v) \geq 0$. A forward path in G is a finite sequence $U = (U_0, \ldots, U_n)$ of states in G where (U_i, U_{i+1}) is an edge in G for $0 \leq i < n$. We say that forward path U contains edge (u, v) if $U_i = u$ and $U_{i+1} = v$

---

[1] If G has multiple edges from state u to state v, we ignore all but the cheapest of them.

for some i. Likewise, a backward path is a finite sequence $V = (V_0, \ldots, V_m)$ of states where (V_i, V_{i+1}) is a "reverse" edge, i.e. (V_{i+1}, V_i) is an edge in G for $0 \leq i < m$. Backward path V contains reverse edge (u, v) if $V_i = u$ and $V_{i+1} = v$ for some i. The reverse of path $V = (V_0, \ldots, V_m)$ is $V^{-1} = (V_m, \ldots, V_0)$. The cost of a reverse edge equals the cost of the corresponding original edge. A path pair (U, V) has a forward path (U) as its first component and a backward path (V) as its second component.

If U is a path (forward or backward), $|U|$ is the number of edges in U, $c(U)$ is the cost of U (the sum of the costs of all the edges in U), and U_i is the i^{th} state in U $(0 \leq i \leq |U|)$. $U_{|U|}$ is the last state in path U, which we also denote $end(U)$. $\lambda_F = (start)$ and $\lambda_B = (goal)$ are the empty forward and backward paths from $start$ and $goal$, respectively. Note that $end(\lambda_F) = start$ while $end(\lambda_B) = goal$. Both λ_F and λ_B have a cost of 0. Forward (backward, resp.) path U is optimal if there is no cheaper forward (backward, resp.) path from U_0 to $end(U)$. $d(u, v)$ is the distance from state u to state v, i.e., the cost of the cheapest forward path from u to v. If there is no forward path from u to v then $d(u, v) = \infty$. Given two states in G, $start$ and $goal$, a solution path is a forward path from $start$ to $goal$. $C^* = d(start, goal)$ is the cost of the cheapest solution path.

A heuristic maps an individual state in G to a non-negative real number or to ∞. Heuristic h_F is *forward admissible* iff $h_F(u) \leq d(u, goal)$ for all u in G and is *forward consistent* iff $h_F(u) \leq d(u, u') + h_F(u')$ for all u and u' in G. Heuristic h_B is *backward admissible* iff $h_B(v) \leq d(start, v)$ for all v in G and is *backward consistent* iff $h_B(v) \leq d(v', v) + h_B(v')$ for all v and v' in G. For any forward path U with $U_0 = start$ define $f_F(U) = c(U) + h_F(end(U))$, and for any backward path V with $V_0 = goal$ define $f_B(V) = c(V) + h_B(end(V))$.

A problem instance is defined by specifying two front-to-end heuristics, h_F and h_B, and a state space G represented implicitly by a 5-tuple $(start, goal, c, expand_F, expand_B)$ consisting of a start state ($start$), a goal state ($goal$), an edge cost function (c), a successor function ($expand_F$), and a predecessor function ($expand_B$). The input to $expand_F$ is a forward path U. Its output is a sequence $(U^1, \ldots, U^n)$, where each U^k is a forward path consisting of U followed by one additional state ($end(U^k)$) such that $(end(U), end(U^k))$ is an edge in G. There is one U^k for every state s such that $(end(U), s)$ is an edge in G. Likewise, the input to $expand_B$ is a backward path V and its output is a sequence $(V^1, \ldots, V^m)$, where each V^k is a backward path consisting of V followed by one additional state ($end(V^k)$) such that $(end(V^k), end(V))$ is an edge in G. There is one V^k for every state s such that $(s, end(V))$ is an edge in G.

Although the expand functions operate on paths, it is sometimes convenient to talk about states being expanded. We say state u has been expanded if one of the expand functions has been applied to a path U for which $end(U) = u$. Finally, we say that a state pair (u, v) has been expanded if either u has been expanded in the forward direction or v has been expanded in the backward direction (we do not require both).

A problem instance is solvable if there is a forward path in G from $start$ to $goal$. I_{AD} is the set of solvable problem instances in which h_F is forward admissible and h_B is backward admissible. I_{CON} is the subset of I_{AD} in which h_F is forward consistent and h_B is backward consistent. A search algorithm is *admissible* iff it is guaranteed to return an optimal solution for any problem instance in I_{AD}.

We only consider DXBB [Eckerle *et al.*, 2017] algorithms. These are deterministic algorithms that proceed by expanding states that have previously been generated and have only black-box access to the expand, heuristic, and cost functions.

4 Sufficient Conditions for Node Expansion

This paper builds on recent theoretical work [Eckerle *et al.*, 2017] defining sufficient conditions for state expansion for bidirectional DXBB search algorithms. While A* with a consistent heuristic necessarily expands all states with $f_F(s) < C^*$, in bidirectional search there is no single state that is necessarily expanded, as the search can proceed forward or backwards, avoiding the need to expand any single state. However, given a path pair (U, V), that meets the following conditions, one of the paths' end-states must necessarily be expanded.

Theorem 1. *[Eckerle* et al.*, 2017] Let $I = (G, h_F, h_B) \in I_{CON}$ have an optimal solution cost of C^*. If U is an optimal forward path and V is an optimal backward path such that $U_0 = start$, $V_0 = goal$, and:*

$$\max\{f_F(U), f_B(V), c(U) + c(V)\} < C^*,$$

then, in solving problem instance I, any admissible DXBB bidirectional front-to-end search algorithm must expand the state pair $(end(U), end(V))$.

The $c(U) + c(V)$ condition was not used by BS*, is degenerate in A*, but is used by MM. When the heuristic is weak, this is an important condition for early termination.

We now use these conditions for state-pair expansion to define a bipartite graph.

Definition 1. *For path pair (U, V) define*

$$lb(U, V) = \max\{f_F(U), f_B(V), c(U) + c(V)\}.$$

When h_F is forward admissible and h_B is backward admissible, $lb(U, V)$ is a lower bound on the cost of a solution path of the form UZV^{-1}, where Z is a forward path from $end(U)$ to $end(V)$.

Definition 2. *The Must-Expand Graph $G_{MX}(I)$ of problem instance $I = (G, h_F, h_B) \in I_{CON}$ is an undirected, unweighted bipartite graph defined as follows. For each state $u \in G$, there are two vertices in $G_{MX}(I)$, the left vertex u_F and right vertex u_B. For each pair of states $u, v \in G$, there is an edge in $G_{MX}(I)$ between u_F and v_B if and only if there exist an optimal forward path U with $U_0 = start$ and $end(U) = u$ and an optimal backward path V with $V_0 = goal$ and $end(V) = v$ such that $lb(U, V) < C^*$. Thus, there is an edge in $G_{MX}(I)$ between u_F and v_B if and only if Theorem 1 requires the state pair (u, v) to be expanded.*

We illustrate this in Figures 1 and 2. Figure 1 shows a problem instance $I = (G, h_F, h_B) \in I_{CON}$. In this example a is the start state, f is the goal, and $C^* = 3$. Figure 2 shows $G_{MX}(I)$, where d refers to the cost of the shortest path to each state and f refers to the f-cost of that path. By construction, the edges in $G_{MX}(I)$ exactly correspond to the state pairs that must be expanded according to Theorem 1, and therefore any vertex cover for $G_{MX}(I)$ will, by definition, represent a set of expansions that covers all the required state pairs. For example, one possible vertex cover includes exactly the vertices in the left side with at least one edge–$\{a_F, c_F, d_F, e_F\}$. This represents expanding all the required state pairs in the forward direction. This requires four expansions and is not optimal because the required state pairs can be covered with just three expansions: a and c in the forward direction and f in the backward direction. This corresponds to a minimum vertex cover of $G_{MX}(I) : \{a_F, c_F, f_B\}$.

Theorem 2. *Let $I \in I_{CON}$. Let A be an admissible DXBB bidirectional front-to-end search algorithm, and S_F (resp. S_B) be the set of states expanded by A on input I in the forward (resp. backward) direction. Together, S_F and S_B correspond to a vertex cover for $G_{MX}(I)$. In particular, $|S_F|+|S_B|$ is lower-bounded by the size of the smallest vertex cover for $G_{MX}(I)$.*

Proof. Let (u_F, v_B) be an edge in $G_{MX}(I)$. Then Theorem 1 requires the state pair (u, v) to be expanded by A on input I, i.e., A must expand u in the forward direction or v in the backward direction. Thus the set of states expanded by A on input I corresponds to a vertex cover of $G_{MX}(I)$. □

We will show below (Theorem 5) that this lower bound cannot be attained by any admissible DXBB bidirectional front-to-end search algorithm. However, we devise an *admissible* algorithm, `NBS`, which efficiently finds a near-optimal vertex cover and thus is near-optimal in terms of necessary node expansions.

The claim of near-optimality is only with respect to the state pairs that must be expanded according to Theorem 1, it does not take into account state pairs of the form $(end(U), end(V))$ when $lb(U, V) = C^*$. In principle, `NBS` could expand many such state pairs while some other algorithm does not. We investigate this further in our experiments.

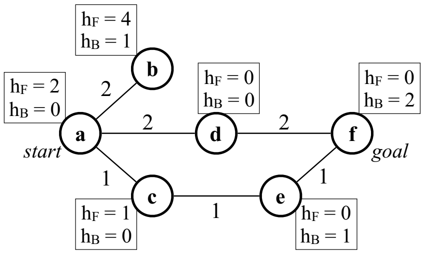


Figure 1: A sample problem instance.

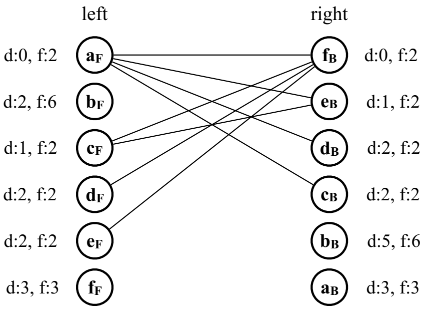


Figure 2: The Must-Expand Graph for Figure 1, where C^*=3.

5 `NBS`: A Near-Optimal Front-to-End Bidirectional Search Algorithm

While a vertex cover can be computed efficiently on a bipartite graph, in practice, building $G_{MX}(I)$ is more expensive than solving I. Instead, we adapt a greedy algorithm for vertex cover [Papadimitriou and Steiglitz, 1982] to achieve near-optimal performance. The greedy algorithm selects a pair of states that are not part of the vertex cover subset selected so far and are connected by an edge in the graph. It then adds both states to the vertex cover. We introduce a new algorithm, `NBS`, which uses the same approach to achieve near-optimal node expansions while finding the shortest path.

The pseudocode for `NBS` is shown in Algorithms 1 and 2. `NBS` considers all pairs for which lb is smallest (line 9). Among these pairs, it first chooses the pairs (U, V) with smallest cost $c(U)$ (line 12) and then, among those, the ones with smallest cost $c(V)$ (line 15). `NBS` picks an arbitrary pair (U, V) from the remaining candidates and expands both U and V (lines 16/17). Breaking ties in this way is necessary to guarantee that `NBS` never expands a suboptimal path when its heuristics are consistent; other tie-breaking rules can be used.An efficient data structure for implementing this path pair selection is described in Section 7.

The pseudocode for backwards expansion is not shown, as it is analogous to forward expansion. We have proofs that, for all problem instances in I_{AD}, `NBS` returns C^*. These are not included here because of space limitations, and because they are very similar to the corresponding proof for MM.

6 Bounded Suboptimality in State Expansions

Theorem 1 identifies the set of state pairs that must be expanded by any admissible DXBB front-to-end bidirectional algorithm. We refer to these as surely expanded (s.e.) path pairs. The theorem does not stipulate which state in each s.e. pair must be expanded; an algorithm is free to make that choice in any manner. Different choices can lead to vastly

Algorithm 1 `NBS`

```
1: $C \leftarrow \infty$
2: $Open_F \leftarrow \{\lambda_F\}; Open_B \leftarrow \{\lambda_B\}$
3: $Closed_F \leftarrow \emptyset$ ; $Closed_B \leftarrow \emptyset$
4: while $Open_F \neq \emptyset$ and $Open_B \neq \emptyset$ do
5:     $Pairs \leftarrow Open_F \times Open_B$
6:     $lbmin \leftarrow \min\{lb(X,Y) \mid (X,Y) \in Pairs\}$
7:     if $lbmin \geq C$ then return $C$
8:     end if
9:     $minset \leftarrow \{(X,Y) \in Pairs \mid lb(X,Y) = lbmin\}$
10:    $Uset \leftarrow \{X \mid \exists Y (X,Y) \in minset\}$
11:    $Umin \leftarrow \min\{c(X) \mid X \in Uset\}$
12:    Choose any $U \in Uset$ such that $c(U) = Umin$
13:    $Vset \leftarrow \{Y \mid (U,Y) \in minset\}$
14:    $Vmin \leftarrow \min\{c(Y) \mid Y \in Vset\}$
15:    Choose any $V \in Vset$ such that $c(V) = Vmin$
16:    Forward-Expand($U$)
17:    Backward-Expand($V$)
18: end while
19: return $C$
```

Algorithm 2 `NBS`: Forward-Expand(U)

```
Move $U$ from $Open_F$ to $Closed_F$
for each $W \in expand_F(U)$ do
    if $\exists Y \in Open_B$ with $end(Y) = end(W)$ then
        $C = \min(C, c(W) + c(Y))$
    end if
    if $\exists X \in Open_F \cup Closed_F$ with $end(X) = end(W)$ then
        if $c(X) \leq c(W)$ then
            Continue for loop // discard $W$
        else
            remove $X$ from $Open_F / Closed_F$
        end if
        Add $W$ to $Open_F$
    end if
end for
```

different numbers of expansions. Given that VC is the size of a minimum vertex cover for G_{MX}, we have shown above that at least VC expansions are required. In this section we prove that on the subset of consistent problem instances `NBS` never expands more than 2VC states to cover all the s.e. pairs, and that for every DXBB front-to-end bidirectional algorithm A there exists a problem instance in I_{CON} on which A expands at least 2VC states to cover all the s.e. pairs. That means that the suboptimality of `NBS` is bounded by a factor of two, and that no competing algorithm can do better in the worst case.

Theorem 3. *Let $I \in I_{CON}$, let $G_{MX}(I)$ be the Must-Expand Graph, and let VC(I) be the size of the smallest vertex cover of G_{MX}. Then* `NBS` *does no more than $2VC(I)$ state expansions on $G_{MX}(I)$ to cover its s.e. pairs.*

Proof. If (u, v) is a s.e. pair then $lb(U, V) < C^*$ for every optimal forward path U from *start* to u and every optimal backward path from *goal* to v. `NBS` will select exactly one such (U, V) pair for expansion and expand both $end(U) = u$ and $end(V) = v$. A minimum vertex cover for I might require only one of them to be expanded, so for each expansion required by a minimum vertex cover, `NBS` might do two. ☐

Theorems 2 and 3 yield the following result.

Corollary 4. *Let $I \in I_{CON}$ and let A be any admissible front-to-end DXBB bidirectional algorithm. Then* `NBS` *makes no more than twice the number of state expansions on input I than A does in covering I's s.e. pairs.*

For any algorithm A, let us use the term *worst-case expansion ratio of A* to refer to the ratio $\max_{I \in I_{CON}} \frac{\#A(I)}{\#(I)}$, where $\#A(I)$ is the number of states A expands in covering the s.e. pairs in instance I, and $\#(I)$ is the smallest number of states any DXBB front-to-end bidirectional search algorithm expands in covering the s.e. pairs in I. By definition, $\#(I) \geq VC(I)$, so we can rephrase Corollary 4 as follows:

$$\texttt{NBS}\text{'s worst-case expansion ratio is at most 2.} \qquad (1)$$

We now demonstrate that `NBS` is optimal in the sense that no admissible DXBB front-to-end bidirectional search algorithm has a worst-case expansion ratio smaller than 2.

Theorem 5. *Let A be any admissible DXBB front-to-end bidirectional search algorithm. Then there exists a problem instance I and a DXBB front-to-end bidirectional search algorithm B such that A expands at least twice as many states in solving I as B expands in solving I.*

Proof. Consider the two problem instances I_1 and I_2 in Figure 3. In these instances $h_F(n) = h_B(n) = 0$ for all n, s is the start and g is the goal. Assume A is given either one of these instances as input. Since A is DXBB and cannot initially distinguish I_1 from I_2, it must initially behave the same on both instances. Hence, on both instances, A will initially either expand s in the forward direction, expand g in the backward direction, or expand both s and g.

If A first expands s in the forward direction, consider $I = I_1$. On instance I, the algorithm A has to expand a second state (either g in the backward direction or t in the forward direction) in order to be able to terminate with the optimal solution path (s, g). By comparison, an algorithm B that first expands g in the backward direction will terminate with the optimal solution after just a single state expansion. Here we assume that B terminates when there are no pairs satisfying the sufficient condition for node expansion.

If A first expands g in the backward direction, one can argue completely symmetrically, with $I = I_2$ and B being an algorithm that first expands s in the forward direction.

If A begins by expanding both s and g, as `NBS` does, then on both these instances it will have expanded two states when only one expansion was required. ☐

7 Efficient Selection of Paths for Expansion

Algorithm 1 assumes that `NBS` can efficiently compute *lbmin* and select the best path pair (U, V) for expansion. In this section we provide a new *Open* list that can do this efficiently. The data structure works by maintaining a lower bound on *lbmin*, C_{lb}. `NBS` initalizes C_{lb} to 0 prior to its first iteration and each time a new path pair is needed, C_{lb} is raised until it reaches *lbmin* and a new path pair is found.

This data structure is illustrated in Figure 4. The data structure is composed of two priority queues for each direction. The first priority queue is $waiting_F$. It contains paths with $f_F \geq C_{lb}$ sorted from low to high f_F. The second priority queue is $ready_F$. It contains paths with $f_F \leq C_{lb}$ sorted from low to high c. Analogous queues are maintained in the backward direction. When paths are added to *Open*, they are first added to $waiting_F$ and $waiting_B$. After processing, paths are removed from the front of $ready_F$ and $ready_B$. The current value of C_{lb} is the minimum of the f-costs at the front of $waiting_F$ and $waiting_B$ and the sum of the c-costs at the front of $ready_F$ and $ready_B$.

Pseudocode for the data structure is in Algorithm 3. Where forward or backwards queues are designated with a D, operations must be performed twice, once in each direction. We use the notation $ready_F.c$ to indicate the smallest c-cost on *ready* in the forward direction. At the end of the procedure

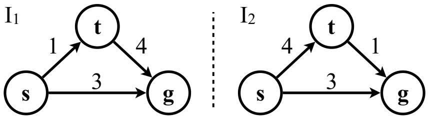


Figure 3: Two problem instances with $C^* = 3$ differing only in the costs of the edges (s, t) and (t, g).

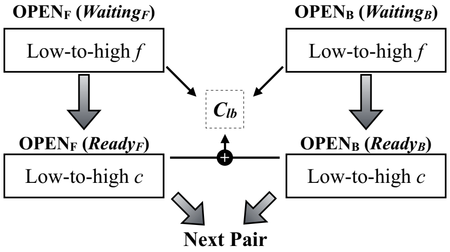


Figure 4: The open list data structure.

Algorithm 3 `NBS` pseudocode for selecting the best pair from $Open$ list. C_{lb} is set to 0 when the search begins.

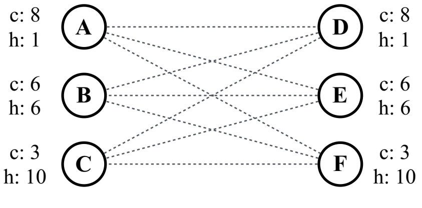
```
1: procedure PrepareBest
2:     while min $f$ in $waiting_D < C_{lb}$ do
3:         move best node from $waiting_D$ to $ready_D$
4:     end while
5:     while true do
6:         if $ready_D \cup waiting_D$ empty then return false
7:         end if
8:         if $ready_F.c + ready_B.c \le C_{lb}$ then return true
9:         end if
10:        if $waiting_D.f \le C_{lb}$ then
11:            move best node from $waiting_D$ to $ready_D$
12:        else
13:            $C_{lb}$ = min($waiting_F.f$, $waiting_B.f$, $ready_F.c$+$ready_B.c$)
14:        end if
15:    end while
16: end procedure
```

the paths on $ready_F$ and $ready_B$ with the smallest individual c-costs together form the pair to be expanded.

The procedure works as follows. First, paths with f-cost lower than C_{lb} must immediately be moved to *ready* (line 2). If $ready_D$ and $waiting_D$ are jointly empty in either direction, the procedure is halted and the search will terminate (line 6). If the best paths in *ready* have $c(U) + c(V) \leq C_{lb}$, the procedure completes; these paths will be expanded next (line 8).

If the $ready_D$ queue is empty in either direction, any paths with $f = C_{lb}$ can be moved to *ready* (line 10). While we could, in theory, move all such paths to *ready* in one step, doing so incrementally allows us to break ties on equal f towards higher c first, which slightly improves performance. If there are no paths with $f \leq C_{lb}$ in *waiting* and in *ready* with $c(U) + c(V) \leq C_{lb}$, then the C_{lb} estimate is too low, and C_{lb} must be increased (line 13).

We illustrate this on an artificial example from Figure 5.[2] To begin, $C_{lb} \leftarrow 0$ and we assume that (A, B, C) are on $waiting_F$ and (D, E, F) are on $waiting_B$. First, C_{lb} is set to 9 (line 13). Then, A and D can be added to *ready* because they have lowest f-cost, and $C_{lb} = 9$. However, $c(A) + c(D) = 16 > C_{lb} = 9$, so we cannot expand A and D. Instead, we increase C_{lb} to 12 and then add B and E to ready. Now, because $c(B) + c(E) = 12 \leq C_{lb}$ we can expand B and E.



Lower Bound	
AD	16
AE	14
AF	13
BD	14
BE	12
BF	13
CD	13
CE	13
CF	13

Figure 5: Sample state space and priorities of state pairs

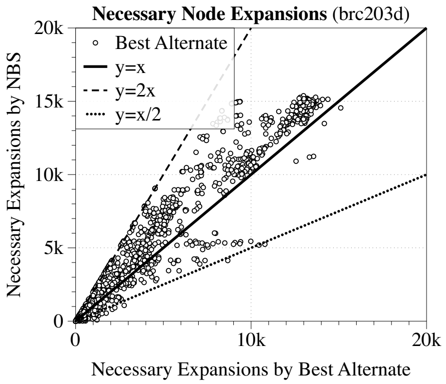


Figure 6: A comparison between necessary expansions by `NBS` (y-axis) and the minimum of the expansions with $f < C^*$ by MMe, BS* and A* (x-axis) on each problem instance.

We can prove that the amortized runtime over a sequence of basic operations of our data structure (including insertion and removal operations) is $O(\log(n))$, where n is the size of *waiting* $\cup$ *ready*.

8 Experimental Results

There are two primary purposes of the experimental results. First, we want to validate that our implementation matches the theoretical claims about `NBS`. Second, we want to compare the overall performance of `NBS` to existing algorithms. This comparison includes total node expansions, necessary expansions, and time. We compare A*, BS*[3], MMe (a variant of MM) [Sharon *et al.*, 2016], `NBS`, and MM_0 (bidirectional brute-force search). We also looked at IDA*, but it was not competitive in most domains due to cycles.

In Table 1 we present results on problems from four different domains, including grid-based pathfinding problems [Sturtevant, 2012] ('brc' maps from Dragon Age: Origins (DAO)), random 4-peg Tower of Hanoi (TOH) problems, random pancake puzzles, and the standard 15 puzzle instances [Korf, 1985]. The canonical goal state is used for all puzzle problems.[4] On each of these domains we use standard heuristics of different strength. The octile, GAP [Helmert, 2010], and Manhattan Distance heuristics can be easily computed at runtime for any two states. The additive pattern databases used for Towers of Hanoi are computed for each instance. We selected the size of problems to ensure that as many algorithms as possible could solve them in RAM.

---

[2] This example also illustrates why we cannot just sort by minimum f_F or c when performing expansions.

[3] BS* is not an admissible algorithm (it will not optimally solve problems with an inconsistent heuristic) so the theory in this paper does not fully apply to BS*.

[4] On the 15-puzzle and TOH it is more efficient to search backwards because of the lower branching factor, but we search forward to the standard goal states.

Table 1: Average state expansions for unidirectional (A*) and bidirectional search across domains.

Domain	Instances	Heuristic	Strength	A*	BS*	MMe	`NBS`	MM$_0$
Grids	DAO	Octile	$+$	**9,646**	11,501	13,013	12,085	17,634
Grids	Mazes	Octile	$-$	64,002	42,164	51,074	**34,474**	51,075
4 Peg TOH	50 random	12+2 PDB	$++$	1,437,644	**1,106,189**	1,741,480	1,420,554	12,644,722
4 Peg TOH	50 random	10+4 PDB	$-$	19,340,099	8,679,443	11,499,867	**6,283,143**	12,644,722
16 Pancake	50 random	GAP	$+++$	**125**	339	283	335	*unsolvable*
16 Pancake	50 random	GAP-2	$-$	1,254,082	947,545	**578,283**	625,900	*unsolvable*
16 Pancake	50 random	GAP-3	$--$	*unsolvable*	29,040,138	7,100,998	**6,682,497**	*unsolvable*
15 puzzle	[Korf,1985]	MD	$+$	15,549,689	**12,001,024**	13,162,312	12,851,889	*unsolvable*

Table 2: Average running time and expansions per second for unidirectional (A*) and bidirectional search across domains.

Average Running Time (in seconds)					
Domain	h	A*	BS*	MMe	`NBS`
DAO	Octile	0.005	0.006	0.015	0.007
Mazes	Octile	0.035	0.022	0.060	0.019
TOH4	12+2	3.23	2.44	4.17	3.54
TOH4	10+4	52.08	23.06	30.64	16.60
Pancake	GAP	0.00	0.00	0.00	0.00
Pancake	GAP-2	14.16	4.91	5.25	5.23
Pancake	GAP-3	N/A	212.33	72.13	77.17
15 puzzle	MD	47.68	29.59	41.38	37.67
Expansion Rate ($\times 10^3$ nodes per seconds)					
DAO	Octile	1,896	1,912	851	1,662
Mazes	Octile	2,225	2,366	848	2,290
TOH4	12+2	444	453	418	401
TOH4	10+4	371	376	375	379
Pancake	GAP	156	564	564	153
Pancake	GAP-2	89	193	109	120
Pancake	GAP-3	N/A	137	98	87
15 puzzle	MD	326	406	318	338

Table 3: Percent of expansions with (f-cost $= C^*$) for each algorithm/domains.

Domain	Heuristic	A*	BS*	MMe	`NBS`
DAO	Octile	1.3%	0.6%	0.7%	1.2%
Mazes	Octile	0.0%	0.0%	0.0%	0.0%
TOH4	12+2 PDB	0.0%	0.0%	0.0%	0.0%
TOH4	10+4 PDB	0.0%	0.0%	0.0%	0.0%
Pancake	GAP	60.7%	81.6%	76.2%	81.0%
Pancake	GAP-2	0.2%	1.6%	6.0%	0.0%
Pancake	GAP-3	N/A	-0.6%	5.7%	0.0%
15 puzzle	MD	5.5%	0.5%	0.6%	0.3%

In grid maps we varied the difficulty of the problem by changing the map type/topology. In TOH and the pancake puzzle we varied the strength of the heuristic. The GAP-k heuristic is the same as the GAP heuristic, except that gaps involving the first k pancakes are not added to the heuristic. The approximate heuristic strength on a problem is indicated by a $+$ or $-$. The general trend is that with very strong heuristics, A* has the best performance. As heuristics get weaker, or the problems get harder, the bidirectional approaches improve relative to A*. `NBS` is never far from the best algorithm, and on some problems, such as TOH, it has significantly better performance than all previous approaches. Runtime and node expansions/second are found in Table 2. `NBS` is 30% slower than A* on the DAO problems, but competitive on other problems. `NBS` is slower than BS*, but this is often compensated for by performing fewer node expansions.

In Table 3 we look at the percentage of total nodes on closed compared to total expansions with $f = C^*$ by each algorithm in each domain. For the majority of domain and heuristic combinations there are very few expansions with $f = C^*$. The exception is the pancake puzzle with the GAP heuristic. On random instances this heuristic is often perfect, so all states expanded have $f = C^*$. This is why `NBS` does more than twice the number of expansions as A* on these problems—these expansions are not accounted for in our theoretical analysis. BS* puts nodes on closed that it does not expand, which is why it has a negative percentage.

Figure 6 shows a scatter plot of necessary node expansions on 1171 instances from the *brc203d* grid map where `NBS` has slightly worse performance than A* and BS*, but better performance than MM. Each point represents one problem instance, and plots `NBS` necessary expansions against the *minimum* of the expansions with $f < C^*$ by A*, BS* and MMe. The $y = 2x$ line represents the theoretical maximum expansion ratio (Theorem 4), which is never crossed. `NBS` often does much better than this theoretical worst case, and on approximately 30 instances is more than 2x better than all alternate algorithms, as they do not have similar 2x expansion bounds.

9 Conclusion

This paper presents the first front-to-end heuristic search algorithm that is near-optimal in necessary node expansions. It thus addresses questions dating back to Pohl's work on the applicability of bidirectional heuristic search [Pohl, 1971]. When the problems are hard or the heuristic is not strong, `NBS` provides a compelling alternative to A*.

Acknowledgements

Financial support for this research was in part provided by Canada's Natural Sciences and Engineering Research Council (NSERC). This material is based upon work supported by the National Science Foundation under Grant No. 1551406.

References

[Barker and Korf, 2015] Joseph Kelly Barker and Richard E. Korf. Limitations of front-to-end bidirectional heuristic search. In *Proc. 29th AAAI Conference on Artificial Intelligence*, pages 1086–1092, 2015.

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

[Eckerle *et al.*, 2017] Jürgen Eckerle, Jingwei Chen, Nathan Sturtevant, Sandra Zilles, and Robert Holte. Sufficient conditions for node expansion in bidirectional heuristic search. In *International Conference on Automated Planning and Scheduling (ICAPS)*, 2017.

[Helmert, 2010] Malte Helmert. Landmark heuristics for the pancake problem. In *Symposium on Combinatorial Search*, pages 109–110, 2010.

[Holte *et al.*, 2016] Robert C. Holte, Ariel Felner, Guni Sharon, and Nathan R. Sturtevant. Bidirectional search that is guaranteed to meet in the middle. In *AAAI Conference on Artificial Intelligence*, pages 3411–3417, 2016.

[Kaindl and Kainz, 1997] Hermann Kaindl and Gerhard Kainz. Bidirectional heuristic search reconsidered. *J. Artificial Intelligence Resesearch (JAIR)*, 7:283–317, 1997.

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

[Kwa, 1989] James B. H. Kwa. BS*: An admissible bidirectional staged heuristic search algorithm. *Artificial Intelligence*, 38(1):95–109, 1989.

[Nicholson, 1966] T. A. J. Nicholson. Finding the shortest route between two points in a network. *The Computer Journal*, 9(3):275–280, 1966.

[Nilsson, 1982] Nils J. Nilsson. *Principles of Artificial Intelligence*. Springer, 1982.

[Papadimitriou and Steiglitz, 1982] Christos H Papadimitriou and Kenneth Steiglitz. *Combinatorial optimization: algorithms and complexity*. Courier Corporation, 1982.

[Pohl, 1971] Ira Pohl. Bi-directional search. *Machine Intelligence*, 6:127–140, 1971.

[Sharon *et al.*, 2016] Guni Sharon, Robert C. Holte, Ariel Felner, and Nathan R. Sturtevant. Extended abstract: An improved priority function for bidirectional heuristic search. *Symposium on Combinatorial Search (SoCS)*, pages 139–140, 2016.

[Sturtevant, 2012] Nathan R. Sturtevant. Benchmarks for grid-based pathfinding. *Transactions on Computational Intelligence and AI in Games*, 4(2):144–148, 2012.